Introduction to Software Testing

A Practical Guide to Testing, Design, Automation, and Execution

—

Panagiotis Leloudas

Apress®

Introduction to Software Testing

A Practical Guide to Testing, Design, Automation, and Execution

Panagiotis Leloudas

Apress®

Introduction to Software Testing: A Practical Guide to Testing, Design, Automation, and Execution

Panagiotis Leloudas
Ilioupoli, Greece

ISBN-13 (pbk): 978-1-4842-9513-7 ISBN-13 (electronic): 978-1-4842-9514-4
https://doi.org/10.1007/978-1-4842-9514-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Divya Modi
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image by Freepik.com

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To my family

Table of Contents

About the Author

Panagiotis Leloudas is a software quality assurance engineer with more than 10 years of working experience in the industry. He holds several ISTQB certifications and is an expert in testing principles, methodologies, and techniques.

He decided to write this quick guide to software testing because he needed a go-to document for all the decisions he had to make every day on the job and there wasn't one. He tried to write down everything that he would have liked to know when he started his career.

About the Technical Reviewer

Sourabh Mishra is an entrepreneur, developer, speaker, author, corporate trainer, and animator. He is passionate about Microsoft technologies and a true .NET warrior. Sourabh has loved computers from childhood and started his career when he was just 15 years old. His programming experience includes C/C++, ASP.NET, C#, VB .NET, WCF, SQL Server, Entity Framework, MVC, Web API, Azure, jQuery, Highcharts, and Angular. Sourabh has been awarded Most Valuable Professional (MVP) status. He has the zeal to learn new technologies, sharing his knowledge on several online community forums.

He is a founder of IECE Digital and Sourabh Mishra Notes, an online knowledge-sharing platform where people can learn new technologies.

Acknowledgments

Writing a book is a significant undertaking, and I couldn't have done it without the support and assistance of many people along the way. I want to express my gratitude to everyone who contributed to the creation of this book and helped me throughout the process.

First and foremost, I want to thank my family for their unwavering support, encouragement, and patience during the many months of writing and editing. Your love and support mean the world to me, and I couldn't have done this without you.

I also want to thank my editors, Divya Modi and James Markham, for their insightful feedback, guidance, and support throughout the writing process. Your expertise and attention to detail were invaluable in shaping this book into its final form.

I am grateful to Apress Media LLC for believing in this project and providing the resources necessary to bring it to fruition.

I am indebted to the many experts, researchers, and consultants who generously shared their knowledge, insights, and expertise to help me with my research. Your contributions have added depth and richness to this book.

Finally, I want to thank my readers for their interest in this book and for taking the time to read it. Your support and feedback mean the world to me, and I hope this book can make a positive difference in your life.

Thank you all for your support, encouragement, and contributions to this book. I am forever grateful.

Introduction

A collection of notes, thoughts, and experiences written down to be shared with the world: this is how I see the creation of this book. It is not meant to be the ultimate truth about software testing; there is no such a thing after all. I have taken several courses about software testing from different organizations, in numerous countries, and I have spent weeks studying testing material, but most of my knowledge comes from trial and error in the industry. It is true that in the end you learn only what you practice.

This book takes you on a journey around the software testing world, covering the basic principles and techniques and showing examples of how to apply them. Software testing is a safeguard of the quality of a software product, and a tester is responsible for reporting on the quality status. Keep in mind that the quality of the product is not the sole responsibility of a tester; rather, it's a collective effort from every individual in the organization. A chain is only as strong as its weakest link.

Different types of testing are applicable to every phase of the software development life cycle; of course, the priorities and the risks are not the same in all the products. Testing is all about identifying those risks and setting up a mitigation plan by executing the necessary tests and analyzing the results.

In this book, I will demonstrate some of the most important software testing types and techniques that exist and how to apply them. You will find out what testing activities take place during every phase of software development, how to plan the testing activities, how to design the test cases, how to execute them, and how to report defects and the status of your activities. It is meant to be easy to understand for everyone, even with only the slightest technical background or involvement in a product. I hope you will enjoy the process and pick up a thing or two!

CHAPTER 1

The Importance of Software Testing

Software testing is a crucial part of the software development process. It is the process of evaluating a software system or application to find defects, errors, or bugs, and verifying that it meets its intended requirements and functions correctly. Software testing is essential because it ensures that the software performs as expected, meets user needs, and is reliable and efficient.

Testing is not a one-time event, but rather a continuous process that begins in the early stages of development and continues through the software's life cycle. The process involves planning, designing, executing, and evaluating tests to identify and fix issues and to improve the quality of the software.

There are various types of software testing, each with a unique focus and objective. Some of the common types include unit testing, integration testing, system testing, acceptance testing, regression testing, performance testing, and security testing. Each type of testing has a specific purpose and is conducted at different stages of the software development life cycle.

One common misconception about software testing is that it is just about finding defects or errors in software. While detecting defects is an essential part of software testing, it is not the only goal. Testing is also about verifying that the software meets user requirements, is easy to use, and performs as expected. Testing also involves ensuring that the software is scalable, secure, and efficient.

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4_1

Another misconception is that testing can be eliminated by writing perfect code or using the right tools. However, testing is an integral part of software development, and there is no way to guarantee that software is completely free of errors. No matter how skilled the development team is or how advanced the tools and technologies used, there is always a possibility of errors or unexpected outcomes. Therefore, testing is essential to identify and fix issues early in the development process before they become major problems.

Black-box testing is a testing technique that examines the functionality of an application without knowing its internal code structure. The tester focuses on the inputs and outputs of the system and tests the application based on the specifications or requirements. The goal of black-box testing is to identify defects in the functionality, usability, and performance of the application.

White-box testing, on the other hand, is a testing technique that examines the internal structure of the application. The tester focuses on the application's code structure, internal logic, and algorithms to test the application. The goal of white-box testing is to identify defects in the code structure and ensure that the application functions correctly.

Gray-box testing is a combination of black-box and white-box testing. The tester has some knowledge of the internal code structure, but not full access. The goal of gray-box testing is to identify defects in the code structure and ensure that the application functions correctly.

Exploratory testing is a testing technique that involves testing the software without predefined test cases or scripts. The tester explores the software and tests it based on their intuition and experience. The goal of exploratory testing is to identify defects that may not be found using traditional testing techniques.

In addition to testing the software's functionality, it is also essential to test nonfunctional aspects, such as performance, security, and usability. These types of testing ensure that the software meets its nonfunctional requirements and provides a positive user experience.

Software testing is an ongoing process that continues throughout the software's life cycle. It is essential to conduct regular testing, even after the software has been released to production, to ensure that it continues to function correctly and meets the users' needs. This is known as **maintenance testing**.

This book will provide a simple overview of the testing activities, tools, and techniques. By the end of it, you will have all the basic knowledge you need in a day as a tester.

I have tried to assemble all the knowledge I have accumulated so far into a small practical guide on how software testing is. The purpose of this book is not to enforce processes and tools in your organization, but to give you some ideas and best practices you might find useful in your daily activities.

It is the responsibility of the tester to report on the quality of the product, but the quality of a company relies on every single individual. Software quality is the responsibility of everyone involved in the software development process, including developers, testers, project managers, business analysts, and other stakeholders. Here are some ways that each individual can contribute to software quality in a company.

Developers play a critical role in ensuring software quality by writing high-quality, well-designed, and maintainable code. They can also contribute to software quality by adhering to coding standards, performing code reviews, and writing automated tests.

Testers play a crucial role in ensuring software quality by designing and executing test plans and test cases to verify that software products meet the requirements and expectations of end users. They can also provide feedback to development teams to help improve the quality of the code.

Product owners are responsible for defining and prioritizing product requirements. They can contribute to software quality by ensuring that requirements are clear, concise, and testable, and that they meet the needs of end users.

Project managers are responsible for managing project timelines, resources, and budgets. They can contribute to software quality by ensuring that projects are adequately resourced, that timelines are realistic, and that projects are well-planned and executed.

Business analysts can contribute to software quality by ensuring that the business requirements are clear, complete, and testable, and that testing teams have the information they need to design effective test cases. They can also validate the requirements with stakeholders and subject-matter experts to identify any gaps or inconsistencies in the requirements early in the development process.

User experience designers are responsible for ensuring that software products are user-friendly and meet the needs of end users. They can contribute to software quality by conducting user research, designing intuitive user interfaces, and ensuring that products meet accessibility and usability standards.

Operations teams are responsible for deploying and maintaining software products. They can contribute to software quality by ensuring that software is deployed in a reliable way.

Technical writers can contribute to software quality by ensuring that user documentation is clear, concise, and easy to understand. This can help to reduce the risk of user errors and improve the overall user experience.

Summary

To summarize, software quality is a collective effort, and the responsibilities are distributed among the various members of a company. There are plenty of software testing types and techniques for the core testing team of a project, and the next chapter will dive into the details of each and every one.

CHAPTER 2

Software Testing Types and Techniques

Software testing is a complex process that involves various types and techniques to ensure that the software is working correctly and meets its intended requirements. In this chapter, we will discuss the different types of software testing and the techniques used in each type, starting with functional testing and the four levels it involves. We will continue with nonfunctional testing, which can include performance testing, compliance testing, and all the different types of -ility testing (usability, compatibility, availability, etc.). After that, we will see the benefits of static testing, where we can detect potential defects just by carefully examining the code without even executing it.

Functional Testing

This type of testing is performed to validate that the software is working as expected and meets the user's requirements. The primary focus is on the software's functionality and includes various levels of testing such as unit testing, integration testing, system testing, and acceptance testing. These levels are widely known as the *testing pyramid*, a concept introduced by Mike Cohn.

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4_2

The lower we go on the testing pyramid (Figure 2-1), the faster and more isolated the tests are. On the contrary, as we go higher on the testing pyramid, the tests become slower, and more components are integrated.

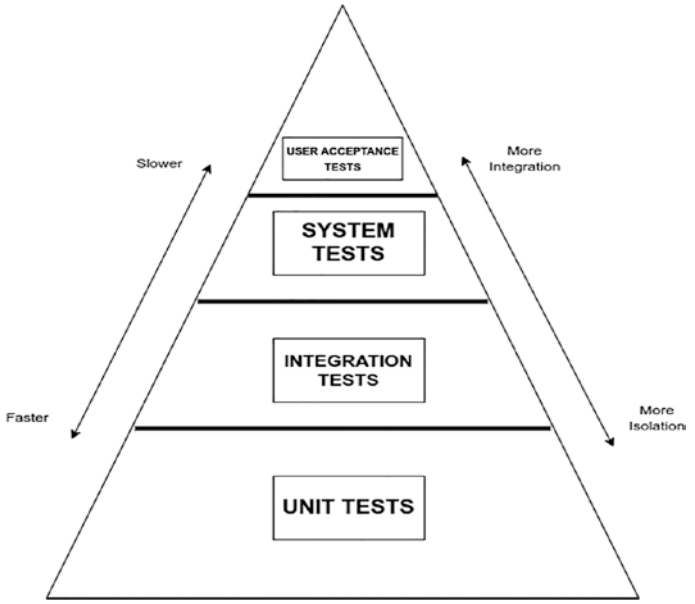

Figure 2-1. *The testing pyramid*

Unit testing involves testing individual units or components of the software. The objective is to validate that each unit functions correctly and meets its intended requirements.

Integration testing involves testing the integration of individual units or components to ensure that they work together correctly.

System testing involves testing the entire system to ensure that it meets the user's requirements and functions correctly.

User acceptance testing involves testing the software from the user's perspective to ensure that it meets the user's expectations and requirements.

Unit Testing

Unit testing is a software testing technique that focuses on testing individual units or components of a software system. A unit is the smallest testable part of any software application, usually a function, method, or class. The purposes of unit testing are to verify that each unit of the software performs as expected and to catch errors as early as possible in the development process.

During unit testing, the individual units of the software are tested in isolation from the rest of the system. This is achieved by using stubs or mock objects to simulate the behavior of other parts of the system that the unit depends on. Unit tests are typically automated, which enables them to be run frequently and easily, ensuring that changes to the code do not break existing functionality.

Unit tests are typically written by developers as part of the development process. Here are some key characteristics of unit tests:

- *Isolation*: Unit tests are designed to test individual units of code in isolation from the rest of the application. This is typically achieved by using mock objects or stubs to simulate the behavior of other parts of the application.
- *Automation*: Unit tests are typically automated using testing frameworks such as JUnit or NUnit. This allows developers to run tests frequently and quickly and to identify and fix defects early in the development process.
- *Coverage*: Unit tests should provide coverage for all possible execution paths through the unit of code being tested. This helps to ensure that all possible scenarios are tested and that defects are identified early in the development process.

- *Independence*: Unit tests should be independent of each other and should not rely on the results of other tests. This helps to ensure that defects are identified and fixed in a timely manner and that tests do not interfere with each other.
- *Maintainability*: Unit tests should be easy to maintain and update as the code evolves over time. This can help to ensure that tests continue to provide accurate results and that defects are identified and fixed early in the development process.

The following are the main benefits of unit testing:

- *Early detection of bugs*: By testing individual units in isolation, unit testing can detect errors early in the development process, which makes them easier and cheaper to fix.
- *Faster debugging*: Unit testing allows developers to isolate and identify the cause of errors quickly and efficiently.
- *Better code quality*: Unit testing forces developers to write modular, well-structured code that is easier to understand and maintain.
- *Improved design*: Unit testing encourages developers to design their code with testability in mind, which can lead to better software design.

Overall, unit testing is an essential practice in software development that helps ensure the quality of software applications.

Integration Testing

Integration testing is a type of testing that focuses on testing the interaction between different software components, subsystems, or modules. The purpose of integration testing is to verify that the integrated software system works as intended and that the individual components are able to work together without errors or issues.

The main goal of integration testing is to identify any defects or issues that may arise when multiple software components are combined. The objective is to detect problems early in the software development cycle, before the product is released to the customers. This helps in reducing the cost and effort required to fix issues in later stages of the software development process.

There are different approaches to integration testing such as top-down, bottom-up, and hybrid. In the top-down approach, testing starts from the highest level and works its way down to the lowest level of the software hierarchy. In the bottom-up approach, testing starts from the lowest level and works its way up to the highest level of the software hierarchy. The hybrid approach combines the top-down and bottom-up approaches.

In integration testing, individual units of code are combined and tested as a group. Integration tests can be performed at various levels, such as the following:

- *Component-level integration*: This involves testing the integration of individual components or modules of an application. Component-level integration tests ensure that individual components work together correctly and can communicate with each other.
- *System-level integration*: This involves testing the integration of multiple components or subsystems of an application. System-level integration tests ensure that the application functions correctly as a whole and that different subsystems work together seamlessly.

- *End-to-end integration*: This involves testing the integration of the entire application, including all subsystems and external systems that the application interacts with. End-to-end integration tests ensure that the application meets the needs of end users and stakeholders and that all the components work together correctly.

During integration testing, different types of testing techniques are used such as functional testing, performance testing, security testing, and usability testing. The testers ensure that the different software components are integrated correctly and that they work together as intended.

The following are some common types of integration testing:

- *Big Bang integration testing*: In this type of testing, all the software components are integrated together and tested as a whole.
- *Incremental integration testing*: In this type of testing, the software components are integrated incrementally, one at a time, and tested as they are added to the system.
- *Top-down integration testing*: In this type of testing, testing starts with the highest level modules and works its way down to the lowest level modules.
- *Bottom-up integration testing*: In this type of testing, testing starts with the lowest-level modules and works its way up to the highest-level modules.
- *Sandpit integration testing*: In this type of testing, a new module is integrated with the rest of the system in a separate environment called a *sandbox* or *sandpit*. This allows the new module to be tested in a controlled environment without affecting the rest of the system.

System Testing

System testing is a type of software testing that is performed on a complete, integrated system to evaluate the system's compliance with its specified requirements. It is generally a black-box testing approach, which means that testers are not concerned with the internal workings of the system. Instead, they focus on testing the system as a whole to ensure that it meets the functional and nonfunctional requirements of the stakeholders.

System testing is typically conducted after integration testing, which tests the integration of different components or subsystems of the system. The objective of system testing is to verify that the integrated system satisfies the requirements and behaves as expected in the target environment.

User Acceptance Testing

User acceptance testing (UAT) is a process to validate whether a system meets the specified requirements and works as expected in the real-world scenario. UAT is performed by end users, business stakeholders, or domain experts to ensure the system's functionality and usability.

The following are some key characteristics of user acceptance testing:

- *User involvement*: UAT involves end users or stakeholders testing the software product, providing feedback, and ensuring that it meets their needs and requirements.
- *Real-world testing*: UAT involves testing the software product in a real-world environment, using real data and scenarios, to ensure that it meets the user's business processes and workflows.
- *Validation of business requirements*: UAT ensures that the software product meets the business requirements and goals of the user or stakeholder.

- *Acceptance criteria*: UAT defines acceptance criteria for the software product, which are used to determine whether the product is ready for deployment.
- *Sign-off*: UAT requires the sign-off of the user or stakeholder to indicate that they are satisfied with the software product and that it is ready for deployment.

The primary objectives of UAT are as follows:

- To validate the system's functionality from an end user's perspective
- To check whether the system meets the business requirements
- To ensure that the system is user-friendly and easy to use
- To find defects that were not detected in previous testing stages
- To reduce the risk of system failure in production

There are three types of UAT:

- *Alpha testing*: It is performed in-house by the development team or a group of testers before releasing the product to the end users.
- *Beta testing*: It is performed by a group of end users in a real-world scenario before releasing the product to the market.
- *Acceptance testing*: It is performed by the end users or business stakeholders to validate whether the system meets the specified requirements and works as expected in the real-world scenario.

Nonfunctional Testing

This type of testing focuses on the software's nonfunctional aspects, such as performance, security, and usability. The objective is to ensure that the software meets its nonfunctional requirements and provides a positive user experience. Here are some examples of nonfunctional testing:

Performance testing involves testing the software's performance under various conditions, such as load, stress, and scalability.

Security testing involves testing the software's security features to ensure that it is secure from various types of attacks and vulnerabilities.

Usability testing involves testing the software's user interface and user experience to ensure that it is user-friendly and easy to use.

Compatibility testing is testing whether the system can work with different software, hardware, and operating systems.

Scalability testing is testing whether the system can handle increasing amounts of data and users.

Reliability testing is evaluating how well the system performs under different conditions and over time.

Availability testing is testing whether the system is available to users and can be accessed at all times.

Installability testing is testing how easy it is to install and configure the software and whether it meets installation requirements and specifications.

Maintainability testing is testing the software's ability to be updated, maintained, and supported over time, and whether it meets maintenance requirements and specifications.

Compliance testing is testing whether the software meets specific regulatory or industry standards, such as HIPAA, GDPR, or PCI DSS.

Performance Testing

Performance testing measures how well a system or application performs under a given workload. It helps identify bottlenecks and problems in the application's performance, and it can also help determine whether the application can handle expected volumes of data or users. In this section, we'll discuss the different types of performance testing, how to plan for and execute performance tests, and tools and techniques used in performance testing.

There are different types of performance testing that can be used to evaluate the performance of an application.

- *Load testing*: This type of testing evaluates how an application performs under normal and peak load conditions. It involves testing the application with a large number of users, requests, and data to determine its performance under different workloads.
- *Stress testing*: This type of testing is performed to determine the application's stability under extreme loads. Stress testing pushes the application beyond its normal operating conditions to evaluate how it responds to these conditions.
- *Spike testing*: This type of testing is used to evaluate the application's performance when there is a sudden and significant increase in traffic. It is designed to determine the application's ability to handle sudden and unexpected spikes in traffic.
- *Endurance testing*: This type of testing is used to evaluate the application's performance over an extended period of time. It is designed to test the application's ability to maintain its performance over a long period.

- *Volume testing*: This type of testing is used to evaluate how well the application can handle large volumes of data. It is used to determine the application's performance under varying data volumes.
- *Scalability testing*: Scalability testing involves testing the system's ability to handle an increasing amount of users or data without a decrease in performance.
- *Soak testing*: Soak testing involves testing the system under a normal load for an extended period to determine if there are any issues that arise over time.
- *Configuration testing*: Configuration testing involves testing the system under different configurations to determine how performance is affected by different settings.
- *Isolation testing*: Isolation testing involves testing individual components of the system to determine their performance under specific conditions.
- *Comparative testing*: Comparative testing involves comparing the performance of different systems or configurations to determine which one performs better.

There are different tools and techniques that can be used in performance testing.

- *Load testing tools*: These tools are used to simulate a large number of users, requests, and data to evaluate the application's performance under different workloads.
- *Performance monitoring tools*: These tools are used to monitor the application's performance during the testing to identify bottlenecks and issues.

- *Profiling tools*: These tools are used to identify performance bottlenecks in the application's code.
- *Cloud-based performance testing*: This approach involves using cloud-based infrastructure and services to simulate real-world usage scenarios.
- *Performance monitoring tools*: Performance monitoring tools are used to monitor the application's performance and identify any issues that may arise during the testing phase. These tools can be used to monitor various metrics such as CPU usage, memory usage, and response time.
- *Profiling tools*: Profiling tools are used to analyze the code and identify areas that can be optimized to improve performance. These tools can be used to identify memory leaks, deadlocks, and other performance issues.
- *Log analysis tools*: Log analysis tools are used to analyze the log files generated during the performance testing. These tools can be used to identify any errors or issues that may have occurred during the testing phase.
- *Statistical analysis tools*: Statistical analysis tools can be used to analyze the performance test results and identify any trends or patterns. These tools can be used to determine the application's performance under varying loads and identify any bottlenecks.
- *Root-cause analysis tools*: Root-cause analysis tools are used to identify the root cause of any performance issues that may arise during the testing phase. These tools can be used to drill down into the application code to identify the underlying cause of the issue.

- *Benchmarking tools*: Benchmarking tools are used to compare the application's performance against industry standards and best practices. These tools can be used to identify any performance issues that may arise and help to improve the application's performance.

Security Testing

Security testing is a type of software testing that focuses on identifying vulnerabilities and potential security risks within an application or system. Security testing is an essential process for ensuring that an application or system is secure and safe from potential attacks or malicious activity.

There are various types of security testing techniques and tools available that can be used to identify and address potential security issues. The following are some of the common techniques:

- *Penetration testing*: Penetration testing, or pen-testing, is a security testing technique that involves simulating an attack on a software system to identify and exploit vulnerabilities in the system. Penetration testing tools include Metasploit, Nessus, and Wireshark.
- *Vulnerability scanning*: Vulnerability scanning tools are used to identify vulnerabilities in a software system by scanning for weaknesses in the system. Some examples of vulnerability scanning tools are Nmap, OpenVAS, and Retina.
- *Fuzz testing*: Fuzz testing is a technique that involves feeding large amounts of random data to a software system to identify vulnerabilities in the system. Fuzz testing tools include Peach Fuzzer, JBroFuzz, and SPIKE.

- *Code review*: Code review is the process of analyzing source code to identify and fix potential security vulnerabilities. Code review tools include SonarQube, Codacy, and CodeClimate.
- *Authentication testing*: Authentication testing involves testing the login process of a software system to ensure that it is secure and resistant to attacks. Authentication testing tools include Burp Suite, ZAP, and Acunetix.
- *Authorization testing*: Authorization testing involves testing the permissions and access control of a software system to ensure that users have access only to the resources they are authorized to use. Authorization testing tools include OWASP WebScarab, IronWASP, and IBM AppScan.
- *Security configuration testing*: This type of testing checks the configuration of the system and identifies vulnerabilities in the configuration of the operating system, network devices, firewalls, and other security devices.
- *Encryption testing*: This type of testing involves testing the effectiveness of encryption mechanisms in the application. It evaluates the strength of the encryption algorithms and checks if the data is being stored and transmitted securely.
- *Input validation testing*: This type of testing checks if the application is protected from malicious inputs. It tests if the application validates user input and does not accept malicious input from users.

- *Security code review*: This type of testing involves reviewing the application code for security vulnerabilities. It can be done manually or using automated tools.
- *Security compliance testing*: This type of testing checks if the application meets the security standards and guidelines. It includes testing against regulatory compliance requirements such as HIPAA, PCI DSS, and GDPR.
- *Disaster recovery and business continuity testing*: This type of testing is done to ensure that the application can continue to operate during and after a disaster. It includes testing the backup and recovery processes and the effectiveness of the disaster recovery plan.

There are numerous security vulnerabilities that can exist in a system, and detecting them requires knowledge of the common ones and the techniques used to find them. Here are a few common security vulnerabilities and how they can be detected:

- *SQL injection*: SQL injection is a type of attack in which an attacker tries to inject malicious SQL code into a database. This can be detected by testing inputs for SQL keywords or special characters.
- *Cross-site scripting (XSS)*: This is a type of vulnerability that allows an attacker to inject malicious scripts into a web page viewed by other users. XSS can be detected by testing inputs for special characters, such as < and >, or by using an automated scanner.

- *Cross-site request forgery (CSRF)*: This vulnerability allows an attacker to force a user to perform an action on a website without the user's knowledge or consent. CSRF can be detected by checking the HTTP Referer header or by using an automated scanner.
- *Broken authentication and session management*: This vulnerability allows an attacker to gain access to a user's account or session. It can be detected by testing for weak passwords, session hijacking, or using an automated scanner.
- *Buffer overflow*: This vulnerability occurs when a program tries to write more data to a buffer than it can hold, which can cause memory corruption and other security issues. Buffer overflow can be detected by performing boundary value testing or using an automated scanner.
- *Information disclosure*: This vulnerability occurs when sensitive information is exposed to unauthorized users. It can be detected by performing security audits or using an automated scanner.

These are just a few examples of common security vulnerabilities, and there are many more that can exist in a system. It's important to use a variety of tools and techniques to detect and prevent security vulnerabilities.

Usability Testing

Usability testing is a type of testing performed to evaluate how user-friendly a product is. The objective of usability testing is to determine how easy it is for users to learn and use the product, how efficient they are at completing tasks, and how satisfied they are with the product.

Usability testing typically involves selecting a group of users who represent the target audience for the product. These users are then given a set of tasks to perform using the product, while the tester observes their behavior and records their comments. The tasks are designed to be representative of the kinds of tasks that the product is intended to support.

The usability tester may use a variety of techniques to gather feedback from users, including questionnaires, interviews, and direct observation of user behavior. The feedback gathered during the testing process is then used to identify problems with the product's usability and to suggest improvements that can be made to the design.

Usability testing can be conducted at various stages of the product development life cycle, from early prototypes to finished products. It can be performed in a lab setting, in the user's own environment, or online.

The results of usability testing are used to inform design decisions and to improve the overall user experience of the product. By identifying usability problems early in the development process, usability testing can help to save time and money by avoiding costly redesigns and rework.

Compatibility Testing

Compatibility testing is a type of software testing that is performed to ensure that an application is compatible with various hardware, operating systems, browsers, databases, and other third-party software components.

The purpose of compatibility testing is to identify compatibility issues between the software application and the systems on which it will be deployed. The goal is to ensure that the application functions properly across a variety of environments and configurations.

To perform compatibility testing, the testing team needs to identify the hardware, software, and other components that the application will be required to work with. Based on this information, the testing team creates a test plan that outlines the specific tests that need to be conducted to ensure compatibility.

Compatibility testing can be conducted in a number of different ways, including manual testing, automated testing, or a combination of both. The testing team may use a variety of tools and techniques to simulate different hardware and software configurations and to automate the testing process where possible.

The following are some common areas of focus in compatibility testing:

- Operating system compatibility
- Browser compatibility
- Database compatibility
- Mobile device compatibility
- Third-party software compatibility

Scalability Testing

Scalability testing is a type of nonfunctional testing that evaluates the ability of a system or application to scale up or scale down in response to changing workload and data volume. The purpose of scalability testing is to identify the maximum capacity of a system or application and to ensure that it can handle increasing levels of load without compromising its performance or stability.

Scalability testing is particularly important for systems that are expected to handle a large number of users, transactions, or data volume. It is typically performed in a controlled environment, using a combination of manual and automated testing techniques to simulate different levels of load and measure the system's response time, throughput, and resource utilization.

Reliability Testing

Reliability testing is a type of software testing that aims to determine the reliability of a software application by measuring its performance in various conditions over a certain period of time. The goal of this testing is to identify any defects or issues that could affect the overall reliability of the system.

Reliability testing typically involves subjecting the software to a series of tests that simulate real-world scenarios and stress conditions. These tests may include load testing, performance testing, and endurance testing. The software is also tested for its ability to recover from errors or failures and its ability to maintain its performance levels under varying conditions.

The following are some common techniques used in reliability testing:

- *Stress testing*: This involves testing the software application under extreme conditions, such as high loads, to see how it behaves and whether it is able to handle the stress.
- *Endurance testing*: This involves testing the software application over a period of time to see how it performs over a long period of use.
- *Performance testing*: This involves testing the software application under normal and peak loads to measure its performance and identify any bottlenecks or issues.
- *Fault tolerance testing*: This involves testing the software application for its ability to continue functioning even in the event of a system failure or hardware malfunction.
- *Recovery testing*: This involves testing the software application for its ability to recover from errors or failures and return to normal functioning.

Availability Testing

Availability testing is a type of nonfunctional testing that is performed to measure the ability of a system or application to remain available and accessible for use by its users over a specified period of time. The primary objective of availability testing is to identify and mitigate any factors that could lead to service disruptions or downtime, such as hardware or software failures, network connectivity issues, or system configuration errors.

During availability testing, testers simulate various failure scenarios and measure the system's ability to recover from them in a timely manner.

Availability metrics are used to measure the system's performance. These metrics may include the percentage of uptime, the mean time between failures (MTBF), the mean time to repair (MTTR), and the recovery time objective (RTO).

Installability Testing

Installability testing is a type of software testing that evaluates how easy it is to install, set up, and configure the software. This type of testing is essential to ensure that the installation process is straightforward, reliable, and consistent, and that the software meets installation requirements and specifications.

Installability testing involves testing the software installation process on different hardware, operating systems, and configurations, and verifying that it meets the following criteria:

- *Installation instructions*: The installation instructions should be clear, concise, and easy to follow, with step-by-step guidance on how to install and configure the software.

- *Installation process*: The installation process should be straightforward, with no unexpected or confusing steps, and should not require any specialized knowledge or expertise.
- *Installation options*: The software should provide users with installation options, such as custom installation, silent installation, or network installation, to suit their specific needs and preferences.
- *Compatibility*: The software should be compatible with different hardware, operating systems, and configurations, and should not cause conflicts or errors during installation.
- *Resource requirements*: The software should meet the minimum resource requirements for installation, such as disk space, memory, and processor speed, and should not require additional resources that are not readily available.
- *Error handling*: The software should have appropriate error handling mechanisms in place to handle installation errors and failures and should provide clear and helpful error messages to users.
- *Uninstallation*: The software should provide a straightforward and reliable uninstallation process, with no residual files or registry entries left behind, and should not cause any system instability or conflicts.

Overall, installability testing is essential to ensure that the software is easy to install, configure, and use, and that it meets user expectations and requirements. By testing the software installation process thoroughly, testers can identify and address any installation issues and can ensure that the software is of high quality and meets user needs.

Maintainability Testing

Maintainability testing is a type of software testing that evaluates the software's ability to be updated, maintained, and supported over time. This type of testing is essential to ensure that the software can be modified or enhanced easily and that it remains stable and reliable after updates and changes.

Maintainability testing involves testing the software's maintainability by evaluating the following criteria:

- *Code quality*: The code quality should be of high quality and maintainable. This includes variables and functions being named appropriately, modular code that is easy to understand and reusable, code comments, and so on.
- *Documentation*: The software should have clear, concise, and up-to-date documentation that explains how the software works, how it should be maintained, and how to modify it.
- *Code complexity*: The software should have a low code complexity, which means that the code should be simple, easy to understand, and easy to modify. This makes it easier for developers to add new features or fix bugs.
- *Testability*: The software should be easily testable, and the test cases should be well documented and easy to execute. This helps in identifying bugs and fixing them faster.
- *Version control*: The software should be under version control to keep track of changes made to the software, who made the changes, and when they were made. This helps in troubleshooting and identifying issues.

- *Error handling*: The software should have appropriate error handling mechanisms in place to handle errors and failures. This helps in identifying and fixing issues faster.

Overall, maintainability testing is essential to ensure that the software can be updated, maintained, and supported over time, and that it remains stable and reliable. By testing the software's maintainability thoroughly, testers can identify any issues and make recommendations to improve the software's maintainability. This helps ensure that the software remains of high quality and meets user expectations over time.

Compliance Testing

Compliance testing is a type of software testing that ensures that software applications meet certain regulatory, legal, and industry-specific standards. This type of testing is essential to ensure that the software meets legal and regulatory requirements and is safe and secure for users.

Compliance testing involves testing the software for compliance with the following criteria:

- *Legal and regulatory requirements*: The software should comply with applicable laws and regulations, such as data privacy laws, security standards, accessibility standards, and other industry-specific regulations.
- *Security*: The software should be secure and protect sensitive data and information from unauthorized access, theft, or misuse. Compliance testing should evaluate the software's security features, such as encryption, access control, and secure communication protocols.

- *Data privacy*: The software should be designed to protect user data and comply with data privacy laws and regulations. This includes the collection, storage, and use of personal data, such as names, addresses, and financial information.
- *Accessibility*: The software should be accessible to all users, including those with disabilities. Compliance testing should evaluate the software's accessibility features, such as keyboard navigation, screen reader compatibility, and alternative text for images.
- *Industry-specific standards*: The software should meet industry-specific standards, such as healthcare standards, financial services standards, or government standards.
- *Documentation*: The software should have clear and up-to-date documentation that explains how the software meets compliance requirements, how it should be used, and how to handle any compliance issues that may arise.

Overall, compliance testing is essential to ensure that the software meets legal and regulatory requirements and is safe and secure for users. By testing the software's compliance thoroughly, testers can identify any compliance issues and make recommendations to ensure that the software meets the required standards. This helps ensure that the software remains of high quality and meets user expectations.

Static Testing

Static testing is a software testing technique that involves the examination of code or software documentation without actually executing the program. The goal of static testing is to identify defects in the code, requirements, or design before the software is actually executed. This type of testing is also known as *nonexecution testing* or *verification testing*.

Code reviews involve a team of developers and/or testers reviewing each other's code to identify any defects or potential issues.

Requirements reviews are about reviewing the requirements documentation to ensure that they are complete, clear, and accurate.

Design reviews where the team is reviewing the design documentation to ensure that it meets the requirements and is technically feasible.

Walk-throughs usually are a team of developers looking through the code or documentation to ensure that it meets the required specifications and standards.

Inspections are a formal process of reviewing the code or documentation to identify and document defects.

Code Reviews

Code review is a form of static testing that involves reviewing the source code of a software application to identify defects, bugs, or other issues that may impact performance or functionality. Code reviews are typically performed by one or more individuals who have expertise in the programming language, design patterns, and coding standards used in the software application.

The goals of a code review are to identify any defects or issues that may impact the quality or functionality of the software and to provide feedback to the developer so that they can make improvements. Code reviews can be performed manually, using a document or spreadsheet to track issues, or via specialized software tools that automate the review process.

The following are some key benefits of code reviews:

- *Improved code quality*: Code reviews can help to identify and address defects or issues early in the development process, leading to improved code quality.
- *Knowledge sharing*: Code reviews can help to share knowledge and best practices among development teams, leading to improved collaboration and productivity.
- *Reduced development time and costs*: Code reviews can help to identify defects or issues early in the development process, reducing the time and costs associated with testing and debugging.
- *Improved maintainability*: Code reviews can help to ensure that the code is maintainable and easy to update, reducing the risk of introducing new defects or issues in future development.

Requirement Reviews

Requirement reviews are a form of static testing that involves reviewing the software requirements to identify defects, errors, or inconsistencies. This type of review is typically performed by stakeholders, including business analysts, project managers, developers, testers, and end users.

The goal of a requirement review is to ensure that the software requirements accurately reflect the needs and expectations of the stakeholders and that they are complete, unambiguous, and consistent. During the review process, stakeholders may ask questions, clarify requirements, or suggest changes or additions to the requirements.

The following are some key benefits of requirement reviews:

- *Improved software quality*: Requirement reviews can help to ensure that the software requirements accurately reflect the needs and expectations of the stakeholders, leading to improved software quality.
- *Early defect detection*: Requirement reviews can help to identify defects or issues early in the development process, before the software is executed.
- *Reduced development time and costs*: Requirement reviews can help to ensure that the software requirements are complete and accurate, reducing the time and costs associated with rework or changes to the requirements later in the development process.
- *Improved stakeholder communication*: Requirement reviews can help to promote communication and collaboration among stakeholders, leading to a shared understanding of the software requirements.

Design Reviews

Design reviews are a form of static testing that involves reviewing the design documents of a software application to identify defects, errors, or inconsistencies. The design documents may include architectural diagrams, flowcharts, data models, and other design artifacts that describe how the software will be implemented.

The goal of a design review is to ensure that the software design meets the requirements of the stakeholders, and that it is efficient, maintainable, and scalable. During the review process, stakeholders may ask questions, suggest improvements, or identify potential issues or risks in the design.

The following are some key benefits of design reviews:

- *Improved software quality*: Design reviews can help to identify defects or issues early in the development process, leading to improved software quality.
- *Early defect detection*: Design reviews can help to identify defects or issues early in the development process, before the software is implemented.
- *Reduced development time and costs*: Design reviews can help to ensure that the software design is efficient and scalable, reducing the time and costs associated with rework or changes to the design later in the development process.
- *Improved maintainability*: Design reviews can help to ensure that the software design is maintainable and easy to update, reducing the risk of introducing new defects or issues in future development.

Walk-Throughs

Walk-throughs are a form of static testing that involves a group of stakeholders reviewing a software artifact, such as a requirements document, design document, or code, in a step-by-step manner. During the walk-through, the participants may ask questions, suggest improvements, or identify potential issues or risks in the artifact.

The goal of a walk-through is to ensure that the software artifact meets the requirements of the stakeholders, and that it is complete, accurate, and well-documented. The walk-through process can help to identify defects or issues early in the development process, before the software is implemented or executed.

The following are some key benefits of walk-throughs:

- *Improved software quality*: Walk-throughs can help to identify defects or issues early in the development process, leading to improved software quality.
- *Early defect detection*: Walk-throughs can help to identify defects or issues early in the development process, before the software is implemented or executed.
- *Reduced development time and costs*: Walk-throughs can help to ensure that the software artifact is complete and accurate, reducing the time and costs associated with rework or changes to the artifact later in the development process.
- *Improved stakeholder communication*: Walk-throughs can help to promote communication and collaboration among stakeholders, leading to a shared understanding of the software artifact.

Inspections

Inspections are a form of static testing that involves a structured and formal process of reviewing a software artifact, such as a requirements document, design document, or code, with the goal of identifying defects or issues. Unlike walk-throughs, inspections are typically conducted by a small group of trained individuals who follow a set of defined procedures and checklists.

The inspection process typically involves several stages, including planning, preparation, inspection, and follow-up. During the inspection stage, the participants review the software artifact line by line, looking for defects such as syntax errors, logical inconsistencies, and violations of coding standards.

The following are some key benefits of inspections:

- *Improved software quality*: Inspections can help to identify defects or issues early in the development process, leading to improved software quality.
- *Early defect detection*: Inspections can help to identify defects or issues early in the development process, before the software is implemented or executed.
- *Reduced development time and costs*: Inspections can help to ensure that the software artifact is complete and accurate, reducing the time and costs associated with rework or changes to the artifact later in the development process.
- *Improved stakeholder communication*: Inspections can help to promote communication and collaboration among stakeholders, leading to a shared understanding of the software artifact.

Summary

The different software testing types have different purposes. Functional testing is responsible for verifying the correct functionality of our product and finding edge case scenarios that might have been missed. Nonfunctional testing is about all the things that might affect the product while its functionality stays intact. Static testing takes care of everything around the software, without executing the software itself. That can be coding standards, documentation, or designs. All these software testing types have their own place in the software development life cycle, and in the next chapter we are going to understand where we can apply them.

CHAPTER 3

Software Development Life Cycle

The software development life cycle (SDLC) is a process that software development teams follow to design, develop, test, and deploy high-quality software applications. The SDLC provides a framework that ensures the software meets the user's requirements, is delivered on time and within budget, and is of high quality. In this chapter, we will discuss the different phases of the SDLC (Figure 3-1) and their importance in software development.

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4_3

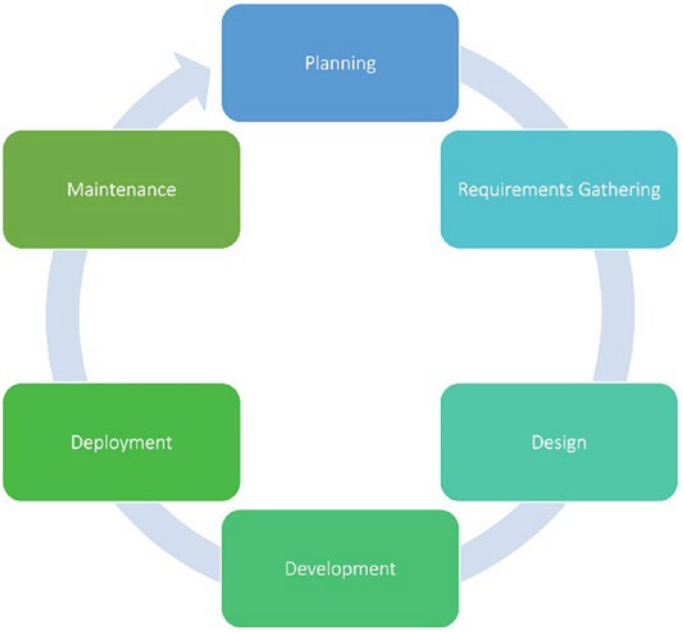

Figure 3-1. *The software development life cycle*

Let's walk through each of these phases one by one.

Planning Phase

The planning phase of the SDLC is the initial phase where the project's objectives and requirements are established. This phase is crucial to ensure the success of the project, and it involves several key activities, such as gathering and analyzing requirements, defining the scope of the project, identifying the stakeholders, and creating a project plan.

During the planning phase, the project team works with stakeholders to determine the project's objectives and to identify the business and technical requirements that the project must meet. The team must analyze the requirements carefully to ensure that they are accurate, complete, and consistent with the business goals. This analysis helps the team to determine the resources required to complete the project, including the team size, budget, and schedule.

Once the requirements have been analyzed, the team defines the project scope, which outlines the deliverables, the constraints, and the assumptions of the project. The scope helps to provide a clear understanding of what the project will deliver and what it will not. The scope also helps to identify the project's risks, assumptions, and dependencies, which are important factors that must be considered during the project planning process.

The next step in the planning phase is to identify the project stakeholders, which include both the internal and external stakeholders. The team must identify the stakeholders' needs, expectations, and requirements, and develop a communication plan to ensure that the stakeholders are informed and engaged throughout the project.

Finally, the project team creates a project plan that includes a detailed timeline, milestones, and deliverables. The project plan outlines the tasks, resources, and timelines required to complete the project, and it helps to ensure that the project is completed on time, within budget, and with the expected quality. The project plan also helps to track progress, identify issues, and manage changes throughout the project's life cycle.

Testing is not a typical activity in the planning phase of the SDLC. This phase is primarily focused on defining the project's scope, goals, objectives, and requirements, as well as identifying the resources and technology required to complete the project. Testing activities are not usually performed in this phase since there is no code or application to test.

However, there are a few test-related activities that can be performed during the planning phase. For instance, the test manager or team can start developing a test strategy that outlines the testing approach, test plan, testing scope, and test schedule for the project. The test manager can also establish testing goals and objectives that align with the project's goals and objectives.

The test team can also start identifying the resources and tools required for testing, such as test management tools, test automation tools, and test data management tools. This can help ensure that the necessary testing resources are available when testing activities begin in the later phases of the SDLC.

Here are some ways that QA can contribute to the planning phase:

- *Reviewing project requirements*: QA can review project requirements to ensure that they are clear, complete, and testable. This can help to identify any gaps or inconsistencies in the requirements early in the development process and ensure that the requirements are aligned with business needs.
- *Defining quality standards*: QA can work with stakeholders to define quality standards for software products. This can include criteria for usability, performance, security, and reliability, among others. Defining quality standards up front can help to ensure that software products meet the needs of end users and stakeholders and reduce the risk of defects and production issues.
- *Developing a test plan*: QA can develop a test plan that outlines the testing strategy for the project. The test plan can include details such as the types of testing that will be performed, the tools and techniques that will be used, and the roles and responsibilities of the testing team.
- *Identifying testing resources*: QA can work with project managers to identify the testing resources that will be needed for the project. This can include tools, equipment, and personnel. By identifying testing resources early in the planning phase, QA can help to ensure that adequate resources are available for testing and quality assurance activities.

- *Collaborating with development teams*: QA can work closely with development teams to ensure that quality is built into the software products from the outset. This can include participating in code reviews, providing feedback on testability and maintainability, and identifying potential defects early in the development process.

The planning phase of the SDLC typically ends with the creation of a detailed project plan or a *software project management plan* (SPMP).

The project plan outlines the scope, objectives, timelines, resource requirements, and budget for the project. It also includes a risk management plan and a quality assurance plan that identifies the testing methodologies, tools, and techniques to be used during the testing phase.

Once the project plan is finalized, the next step is to get the necessary approvals and funding for the project, after which the actual development work begins.

It is important to note that the planning phase is not a one-time event, but an ongoing process that requires continuous review and updates throughout the SDLC to ensure that the project stays on track and meets the objectives and goals of the stakeholders.

When everything is in place, we are ready to proceed to the next phase.

Requirements Gathering Phase

The requirements gathering phase is the second phase of the SDLC. It is also known as the analysis phase. During this phase, the development team works closely with the stakeholders to gather and document the requirements for the software project. The main goal of this phase is to understand the business requirements and objectives, as well as the user requirements and expectations.

The following are the key activities that take place during the requirements gathering phase:

- *Gathering and analyzing requirements*: The development team works with the stakeholders to understand their needs and expectations. This involves conducting interviews, focus groups, and surveys to gather information.
- *Documenting requirements*: The requirements are documented in a detailed manner. This includes use cases, user stories, functional and nonfunctional requirements, and acceptance criteria.
- *Prioritizing requirements*: Once the requirements are documented, they are prioritized based on their importance and impact on the project.
- *Reviewing requirements*: The development team reviews the requirements with the stakeholders to ensure that they are accurate, complete, and feasible.
- *Getting sign-off on requirements*: Once the requirements have been reviewed and finalized, the stakeholders sign off on them. This signifies their agreement that the requirements accurately represent their needs and expectations.

The outcome of the requirements gathering phase is a detailed requirements specification document. This document serves as a reference point for the rest of the SDLC phases. Any changes to the requirements during the development process must be managed through a formal change control process to ensure that the impact of any changes is properly assessed and that the project stays on track.

In the requirements gathering phase of the SDLC, testing involves verifying that the gathered requirements are correct, complete, and testable. The testing team reviews the requirements to ensure that they are clear, unambiguous, and consistent. They work with the business analysts and stakeholders to clarify any areas of confusion or ambiguity.

During this phase, testing also involves identifying any missing requirements, as well as requirements that may conflict with other requirements or the project's overall goals. This is to ensure that the requirements are feasible and can be implemented in the project's scope and timeline.

Additionally, testing in the requirements gathering phase includes defining acceptance criteria and identifying key performance indicators (KPIs) that will be used to measure the success of the project. This helps to ensure that the project will meet the needs of the stakeholders and users and that the testing team can adequately measure the quality of the product once it is completed.

Let's say that a software development team is tasked with building an e-commerce platform. During the requirements gathering phase, the team identifies a requirement that states, "The platform must be able to process 1,000 transactions per minute."

To ensure that this requirement can be met, the QA team can ask the following questions:

- What is the expected load on the platform during peak hours?
- What is the maximum number of concurrent users that the platform will need to support?
- What types of transactions will be processed (e.g., purchases, returns, refunds)?

- Will the platform need to integrate with any third-party payment gateways?
- What security measures will be in place to protect user data during transactions?

By asking these questions, the QA team can ensure that the requirement is well-defined and that the development team has all the information they need to build a platform that can meet the performance and security needs of the business.

After the requirements are defined, reviewed, and accepted by all the stakeholders, then the design phase is ready to start.

Design Phase

During the design phase of the SDLC, the requirements gathered in the previous phase are used to create a detailed design for the software system. The design phase typically involves the following steps:

1. *Architectural design*: In this step, the overall system architecture is defined, including the hardware, software, and network components.
2. *High-level design*: Based on the architectural design, the high-level design is created, which specifies the overall structure of the software system, including the modules or components and their relationships.
3. *Detailed design*: This step involves designing each module or component in detail, including its input and output, algorithms, data structures, and interfaces with other modules.

4. *Prototyping*: Prototyping is often used to validate the design and ensure that it meets the user requirements. This may involve creating a working model of the software system or individual components.
5. *Design review*: Once the detailed design is complete, it is reviewed by stakeholders to ensure that it meets the requirements and is technically feasible.
6. *Design documentation*: The design is documented to ensure that it can be easily understood and maintained by other developers.

The output of the design phase is a detailed design specification, which is used as the basis for the next phase of the SDLC.

In the design phase, the testing team may be involved in the review and evaluation of the design documents to ensure that the design meets the requirements specified in the requirements documents. This can help to identify design issues and inconsistencies early in the development cycle, which can reduce the cost and time associated with fixing defects in later phases.

In addition, the testing team may be involved in the creation of test plans and test cases based on the design documents. This can help to ensure that the testing effort is aligned with the design and that all design components are tested thoroughly.

The following are some common testing activities in the design phase:

- Reviewing the design documents for accuracy, completeness, and consistency with the requirements
- Creating test plans and test cases based on the design documents
- Identifying potential defects or issues in the design and reporting them to the development team

- Conducting design walk-throughs or reviews to ensure that the design meets the specified requirements
- Validating the design against industry standards and best practices

Overall, testing in the design phase is important to ensure that the design is sound and meets the specified requirements. By identifying and addressing design issues early in the development cycle, the testing team can help to reduce the cost and time associated with fixing defects in later phases.

Let's say that a software development team is designing a new feature for a social media platform. During the design phase, the team creates a mockup of the new feature that includes a form for users to enter personal information.

To ensure that the design is user-friendly and meets the platform's quality standards, the QA team can ask the following questions:

- Is the form easy to navigate and understand?
- Are all necessary fields clearly labeled and easy to find?
- Are there any unnecessary fields that could be removed to streamline the user experience?
- Does the design comply with accessibility standards, such as providing alternatives for visually impaired users?
- Is the design consistent with the platform's overall look and feel, including colors, fonts, and logos?

By asking these questions, the QA team can ensure that the design is intuitive, accessible, and consistent with the platform's standards.

When the designs are finalized and everyone involved has agreed, the actual development can start.

Development Phase

During the development phase of the SDLC, the software is designed and developed based on the requirements gathered and design specifications created during the previous phases.

The development phase involves creating the software code, building and testing the software, and fixing any issues that arise during testing. The development team typically consists of software developers, testers, and technical writers who create user manuals, guides, and other technical documentation for the software.

The software code is usually written in a high-level programming language and is translated into machine code by a compiler or interpreter. The code is then tested using a variety of testing techniques, including unit testing, integration testing, and system testing, to ensure that it meets the functional and nonfunctional requirements specified in the earlier phases.

During the development phase, it is important to follow coding standards and best practices to ensure that the software code is maintainable, reusable, and easily extendable in the future. It is also important to ensure that the software code is well-documented and that all functions and procedures are clearly defined and documented to aid in future maintenance and support.

During the development phase, the focus of testing shifts from validating the system design and requirements to identifying and resolving defects in the software code. The objective of this phase is to ensure that the software is functioning as expected and that all the requirements are met.

Testing in the development phase involves the following activities:

- *Unit testing*: Unit testing is a type of testing that verifies the smallest piece of testable code, such as functions, methods, or procedures. It is performed by the developers themselves and is focused on validating that the code meets the technical specifications and works as intended.

- *Integration testing*: Integration testing is the process of testing how different software components interact with each other. It is performed after unit testing and ensures that the different components are properly integrated and work together as expected.
- *System testing*: System testing is the process of testing the entire system as a whole. It involves testing the software in a test environment that is similar to the production environment, with real-world data and simulated users. This type of testing verifies that the system meets all the functional and nonfunctional requirements.
- *Acceptance testing*: Acceptance testing is the final phase of testing before the software is released to the end users. It is a type of testing where the software is evaluated against the user requirements, and it is performed by the business stakeholders or end users.
- *Regression testing*: Regression testing is the process of testing the system after changes have been made to the software code. It ensures that the existing functionality has not been impacted by the changes made and that the new functionality works as expected.
- *Performance testing*: Performance testing is performed to ensure that the system is functioning within the acceptable limits in terms of speed, scalability, and stability. It involves testing the system under various workloads and conditions to identify and fix performance-related issues.

- *Security testing*: Security testing is performed to ensure that the system is secure and protected against external threats. It involves identifying vulnerabilities and testing the system for unauthorized access, data breaches, and other security-related issues.

Testing in the development phase is an iterative process and involves continuous testing and validation of the software code. It is critical to ensure that the software is fully functional, meets the business requirements, and is ready for release to the end users.

Let's say that a software development team is building a mobile app that includes a feature for users to upload photos. During the development phase, the team has written the code to handle photo uploads.

To ensure that the feature works as expected and meets the platform's quality standards, the QA team can perform the following tests:

- The QA team can test the upload feature to ensure that it works correctly and uploads the photos to the correct location. They can also test that the app can handle different file formats and file sizes.
- The QA team can test the upload feature to ensure that it works quickly and efficiently, even when many users are uploading photos simultaneously.
- The QA team can test the upload feature to ensure that it is secure and that user data is protected during the upload process.
- The QA team can test the upload feature to ensure that it is intuitive and easy to use for users. They can also test that any error messages or notifications are clear and helpful.

By performing these tests, the QA team can identify any issues or bugs in the feature and work with the development team to fix them.

The development phase stops when all the items on the definition of done list are checked. This means that all the found defects are either solved or accepted as a risk, the documentation is in place, and the product owner has accepted the implementation. Then we can deploy the changes.

Deployment Phase

The deployment phase in the SDLC is the phase in which the developed software is delivered to the customer for their use. It involves installing the software in the customer's environment, configuring it, and making sure that it works as expected.

The deployment phase involves the following activities:

- *Release planning*: In this phase, the deployment team identifies the necessary resources, timelines, and activities required to deploy the software to the customer's environment.
- *Configuration management*: The deployment team ensures that the software is configured correctly and all necessary dependencies are in place.
- *Deployment testing*: The deployment team performs testing to ensure that the software works as expected in the customer's environment.
- *Data migration*: The deployment team ensures that all data from the previous system is transferred to the new system.
- *User training*: The deployment team provides training to the end users to use the software effectively.

- *User acceptance testing*: The deployment team works with the customer to perform user acceptance testing to ensure that the software meets their requirements.
- *Go-live*: The deployment team deploys the software to the customer's environment and ensures that it is working correctly. The team also provides post-deployment support to the customer.

The deployment phase is critical because it is the final stage before the software is used in a production environment. A successful deployment ensures that the software meets the customer's requirements and is ready for use.

Testing in the deployment phase is important to ensure that the software is correctly installed and configured in the target environment. This phase involves tasks such as the following:

- *Installation testing*: Verifying that the software is installed correctly on the target environment and that it meets the system requirements and compatibility criteria
- *Configuration testing*: Testing the various configuration options available in the software and ensuring that the configuration is set up correctly for the target environment
- *Compatibility testing*: Verifying that the software is compatible with the hardware and software components in the target environment and that there are no conflicts or issues
- *Security testing*: Ensuring that the software is secure in the target environment and that it adheres to the security policies and requirements

- *Performance testing*: Verifying that the software is performing well in the target environment and that it meets the performance requirements
- *Acceptance testing*: Testing the software with end users and stakeholders to ensure that it meets their expectations and requirements

The testing in the deployment phase is often done in collaboration with the system administrators, network administrators, and other IT professionals who are responsible for deploying and maintaining the software in the target environment.

Let's say that a software development team is deploying a new version of their web application to production. During the deployment phase, the team needs to ensure that the application is deployed correctly and works as expected in the production environment.

To ensure that the deployment is successful and meets the platform's quality standards, the QA team can perform the following tests:

- The QA team can perform a quick check to ensure that the application is accessible and that the main functionality is working as expected. This can include verifying that pages load properly, forms can be submitted, and data is displayed correctly.
- The QA team can test the application to ensure that existing features are still working correctly after the deployment. This can include testing that all links still work, pages load quickly, and data is displayed correctly.

- The QA team can test the application to ensure that it can handle the expected user load. This can include testing that the application can handle a certain number of concurrent users and that performance is not degraded when many users are using the application at the same time.
- The QA team can test the application to ensure that it is secure and that user data is protected. This can include testing that user data is encrypted during transmission and storage, that access controls are implemented correctly, and that there are no vulnerabilities that could be exploited by attackers.

By performing these tests, the QA team can ensure that the deployment is successful, that the application works as expected in the production environment, and that it meets the platform's quality standards. Then we can proceed to the maintenance phase.

Maintenance Phase

The maintenance phase in the SDLC is the phase in which the software is released to the market, and it is made available to the end users. During this phase, the software development team is involved in fixing any bugs, errors, or defects that are found in the software.

The maintenance phase is an important phase because it is the time when the software is in use, and it is the time when any issues that arise must be addressed. The maintenance phase is also the time when updates, patches, and new features are added to the software.

There are generally three types of maintenance.

- *Corrective maintenance*: This type of maintenance involves fixing defects, errors, and bugs in the software.
- *Adaptive maintenance*: This type of maintenance involves modifying the software to adapt it to changes in the environment, such as changes in the operating system, hardware, or other software.
- *Perfective maintenance*: This type of maintenance involves improving the software to make it more efficient, reliable, or user-friendly.

During the maintenance phase, the software development team may also be involved in providing technical support to end users and in providing training to help users to get the most out of the software.

Testing in the maintenance phase of the SDLC involves verifying and validating changes made to the software after it has been deployed to production. The primary focus of maintenance testing is to ensure that the software continues to meet the desired levels of quality and functionality, while also addressing any issues that may arise during its use in the field.

Maintenance testing can take on several different forms, depending on the nature and scope of the changes being made. The following are some of the most common types of maintenance testing:

- *Regression testing*: This involves running a full suite of tests to ensure that the changes made to the software have not introduced any new defects or issues and that the existing functionality remains intact.
- *Patch testing*: This involves testing individual patches or updates to the software to ensure that they do not cause any adverse effects or issues.

- *Integration testing*: This involves testing the integration of any new or updated components of the software to ensure that they work as expected and do not negatively impact existing functionality.
- *User acceptance testing*: This involves testing changes made to the software from the perspective of end users to ensure that they meet the desired levels of quality and usability.

In addition to these types of testing, maintenance testing also involves monitoring the performance of the software in the field, identifying and addressing any defects or issues that arise, and ensuring that the software continues to meet the evolving needs and expectations of its users. This ongoing testing and maintenance is critical to ensuring that the software remains functional, reliable, and effective over the long term.

Let's say that a software development team has released a mobile app and is now in the maintenance phase, where they are addressing bugs and making updates to the app. During this phase, the team needs to ensure that any changes they make to the app do not introduce new issues or negatively impact the user experience.

To ensure that the maintenance work is successful and meets the platform's quality standards, the QA team can perform the following tests:

- The QA team can test the app to ensure that changes made to address bugs or add new features have not introduced new issues. This can include testing that existing functionality still works correctly and that no new bugs have been introduced.
- The QA team can test the app to ensure that any changes made to the user interface or user experience do not negatively impact the user experience. This can include testing that the app is still intuitive and easy to use for users.

- The QA team can test the app to ensure that it still works correctly across different devices and operating systems. This can include testing that the app works correctly on the latest versions of popular devices and operating systems.
- The QA team can test the app to ensure that it is still secure and that user data is protected. This can include testing that any new features or changes do not introduce new security vulnerabilities.

By performing these tests, the QA team can ensure that any maintenance work is successful, that the app continues to meet the platform's quality standards, and that users continue to have a positive experience.

The Role of Testing in the SDLC

You have probably noticed that there is no testing phase in the SDLC, or at least I did not include it. In the past, the testing phase was placed right after the development phase, but years of collective experience in the industry proved that testing is not an autonomous phase and it needs to be part of every phase in the SDLC.

Testing is an essential part of the SDLC, and it plays a critical role in ensuring the quality of the software application. The primary purpose of testing is to identify defects or errors in the software application and ensure that the software meets the user's requirements. Testing also helps to ensure that the software application is reliable, scalable, and maintainable.

The role of testing in the SDLC can be summarized as follows:

- *Verify that the software application meets the user's requirements*: Testing helps to ensure that the software application meets the user's requirements and performs the intended functions.
- *Identify defects or errors in the software application*: Testing helps to identify defects or errors in the software application that may impact the software's performance or functionality.
- *Ensure the software application is of high quality*: Testing helps to ensure that the software application is of high quality, reliable, and maintainable.
- *Reduce the risk of software failure*: Testing helps to reduce the risk of software failure and ensures that the software application is stable and performs as intended.
- *Improve user satisfaction*: Testing helps to improve user satisfaction by ensuring that the software application meets the user's needs and performs as expected.

Summary

The software development life cycle comprises several stages, each with its set of activities, deliverables, and milestones and the responsibilities of a tester vary and overlap in every phase. It is important for the tester to be involved as early as possible and to have a deep understanding of the system being tested. Since there are plenty of activities for the tester during the SDLC, it is crucial to plan ahead, estimate, and prioritize all the activities.

CHAPTER 4

Test Planning

Test planning is an essential activity in software testing that involves creating a comprehensive plan for testing the software application. The test plan outlines the testing objectives, scope, approach, resources, and schedule for the testing phase of the software development life cycle (SDLC). Test planning ensures that testing is done effectively and efficiently and helps to minimize the risk of defects in the software application.

In this chapter, we will go through the most important aspects of test planning. First we will examine how to define the testing objectives and then how to determine the scope of the system under test. It is important to select the right testing approach for the product, because different products have different priorities and risks. Then, based on the testing approach, we are going to identify the resources and develop a test schedule. We'll continue with the definition of the test cases, and we will find the applicable test data for the defined test cases.

We will see what a defect management process should contain and when we can stop testing. After everything is in place, we will create a test plan document, and we will see which tools and techniques are useful.

Defining Testing Objectives

Defining testing objectives is a critical step in the software testing process as it sets the direction and purpose of the testing effort. Testing objectives are the specific goals and outcomes that the testing team wants to achieve through the testing process.

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4_4

The objectives of testing may vary depending on the project requirements and the SDLC phase, but generally, the main objectives of testing include the following:

- *Ensuring the software meets the business requirements*: Testing should ensure that the software product meets the requirements defined by the stakeholders.
- *Identifying defects*: Testing should aim to identify defects and bugs in the software product. These defects can be functional or nonfunctional.
- *Verifying software functionality*: Testing should verify that the software product functions correctly and provides the expected output.
- *Ensuring software quality*: Testing should ensure that the software product meets the quality standards defined by the organization.
- *Reducing the risk of software failure*: Testing should help in reducing the risk of software failure and ensuring that the software product is reliable.
- *Ensuring software security*: Testing should ensure that the software product is secure and protects user data from unauthorized access.
- *Increasing user satisfaction*: Testing should ensure that the software product meets the needs of the end users and provides an excellent user experience.

By defining clear testing objectives, the testing team can develop a comprehensive testing plan that aligns with the project requirements and helps in achieving the testing goals.

Determining the Scope of Testing

Determining the scope of testing is a critical step in the testing process. It involves identifying the testing areas, features, and functionality that need to be tested, as well as defining the testing boundaries. The scope of testing is determined based on various factors such as the application's complexity, the risk associated with the application, and the testing resources available. The testing scope should be defined and documented in the test plan to ensure that all stakeholders understand what is in and out of scope for testing.

A well-defined testing scope helps in ensuring that all critical features and functionalities of the application are thoroughly tested, and the application is tested to a level that meets the business requirements. The scope of testing may also be impacted by factors such as time, budget, and resource constraints. Therefore, it is important to prioritize the testing efforts based on the criticality of the features and functionalities.

The scope of testing should be reviewed and updated regularly during the testing process to ensure that the testing effort is on track, and all the testing objectives are being met. Regular reviews and updates also help to identify any changes in the scope of the application and its functionalities, which may require additional testing efforts.

Selecting the Testing Approach

Selecting the appropriate testing approach is an essential step in the software testing process. The approach taken will depend on a number of factors, such as the complexity of the software, the project timeline, and the available resources.

There are several testing approaches, each with its own set of advantages and disadvantages. The following are some of the most common testing approaches:

- *Black-box testing*: This approach tests the functionality of the software without any knowledge of the internal workings of the software. Testers use requirements specifications to create test cases.
- *White-box testing*: This approach tests the internal workings of the software, such as code, logic, and algorithms. Testers use programming skills to create test cases.
- *Gray-box testing*: This approach combines the elements of black-box and white-box testing. Testers have limited knowledge of the internal workings of the software, allowing them to test both the functionality and the internal workings of the software.
- *Manual testing*: This approach involves manual execution of test cases. Testers execute the test cases by hand and record the results.
- *Automated testing*: This approach involves the use of software tools to execute test cases automatically. This is a faster and more efficient testing approach than manual testing.
- *Exploratory testing*: This approach involves testing the software without a formal test plan. Testers use their knowledge of the software to identify defects.
- *Regression testing*: This approach involves retesting the software after a change has been made. The purpose of regression testing is to ensure that the change did not introduce any new defects.

It is important to select the appropriate testing approach for each project. The testing approach should be chosen based on the project requirements, available resources, and time constraints.

Identifying Testing Resources

Identifying testing resources is an important step in the testing process as it helps to determine the skills, tools, and infrastructure needed to perform the testing activities.

The testing resources that may be required include the following:

- *Personnel*: This includes the testing team members, test leads, and test managers who will be responsible for performing the testing activities.
- *Testing tools*: The testing team may require various testing tools such as test management tools, defect tracking tools, automation tools, performance testing tools, and security testing tools to help perform the testing activities.
- *Infrastructure*: The testing team may require hardware and software infrastructure such as servers, network equipment, and testing environments.
- *Data*: The testing team may require data that represents the actual production environment to perform the testing activities. This may include data such as customer data, product data, and transaction data.
- *Budget*: The testing team may require a budget to cover the costs of the testing activities. This may include costs for personnel, testing tools, infrastructure, and data.

Identifying and obtaining these resources early in the testing process can help ensure that the testing activities are performed efficiently and effectively.

Developing the Test Schedule

Developing the test schedule is an important aspect of software testing planning. A test schedule is a detailed plan that outlines the testing activities, timelines, and resources required to carry out the testing activities.

The following are the steps to develop a test schedule:

1. *Determine the testing timeline*: The first step is to determine the testing timeline. The testing timeline is the period in which the testing activities will take place. It is important to consider the project timeline, including the delivery date, milestones, and other important events.
2. *Identify the testing activities*: Identify the testing activities required for each stage of the SDLC such as functional testing, performance testing, security testing, and so on.
3. *Determine the order of testing activities*: Determine the order in which the testing activities will be carried out. This may depend on the type of testing, the requirements of the project, and the interdependencies of the various testing activities.
4. *Allocate resources*: Allocate resources to each testing activity. This includes personnel, tools, hardware, and software required for each testing activity.

5. *Develop the test schedule*: Using the information collected, create a detailed schedule of the testing activities. The schedule should include the start and end dates of each testing activity, the resources allocated to each activity, and the dependencies between the various testing activities.

6. *Review and refine*: Once the test schedule is created, it should be reviewed and refined to ensure that it is accurate and realistic. The review should also include input from key stakeholders to ensure that the test schedule aligns with the project objectives and priorities.

7. *Communicate the test schedule*: The final step is to communicate the test schedule to all stakeholders, including the development team, project manager, and other relevant parties. It is important to ensure that all stakeholders understand the test schedule and their roles and responsibilities in carrying out the testing activities as per the schedule.

Defining Test Cases

Defining test cases is an essential step in the software testing process that involves the identification of individual test cases that need to be executed to ensure that the software meets the specified requirements. A test case is a set of conditions or variables under which a tester determines whether a software system is working correctly or not.

The process of defining test cases involves the following steps:

1. *Identify the objective of the test case*: Define the purpose of the test case and what specific functionality or behavior is to be tested.
2. *Determine the test input data*: Decide on the input data that needs to be provided to the software for the test case.
3. *Define the expected output*: Determine what the expected output of the test case should be, based on the input data and the requirements.
4. *Specify the steps to be executed*: Define the steps that need to be followed to execute the test case. This should include the setup required to prepare the system for testing, the specific actions to be taken, and the data to be used.
5. *Document the test case*: Write down the details of the test case, including the objective, input data, expected output, and steps to be executed.
6. *Review and refine the test case*: Review the test case to ensure that it is complete, accurate, and effective in verifying the software functionality. Make any necessary revisions to improve the test case.
7. *Group related test cases*: Group similar or related test cases together, to make it easier to manage and execute them.
8. *Prioritize the test cases*: Assign priorities to the test cases, based on their importance and the likelihood of finding defects.

9. *Define the test coverage*: Determine the coverage of the test cases by identifying the requirements, features, and areas of the software that the tests are designed to cover.

Identifying Test Data

Identifying test data is the process of selecting and preparing data that will be used to test the software. Test data includes all input values, configuration settings, and other necessary data needed to perform testing on the system. The selection of test data is important, as it will be used to validate the functionality, performance, and security of the software.

There are different types of test data, including normal data, boundary values, negative data, and error messages. Normal data is used to test the software under typical operating conditions, while boundary values are used to test the limits of the software. Negative data is used to test the software's ability to handle invalid input, while error messages are used to test the software's response to errors.

To identify test data, testers need to analyze the software requirements and identify the different scenarios and use cases that need to be tested. They should also consider the expected results and how the software should behave under different conditions. Testers may use tools such as data generators and test data management tools to generate or manage test data.

Defect Management Process

Defining the defect management process is an important aspect of software testing. When defects are found, they need to be recorded, tracked, and managed to ensure that they are fixed and the software meets the desired level of quality. The defect management process involves the following steps:

1. *Defect logging*: Defects need to be logged in a defect tracking tool or system. The defect should be logged with a unique identifier, a summary of the issue, a detailed description, the steps to reproduce the defect, and the severity of the defect.
2. *Defect classification*: Once the defect is logged, it needs to be classified based on its severity and priority. This helps in prioritizing the defects based on the level of impact they have on the software.
3. *Defect analysis*: Once the defects are classified, the next step is to analyze the defects to identify the root cause of the issue. This helps in fixing the issue and preventing it from occurring in the future.
4. *Defect assignment*: After the defect analysis, the defect is assigned to a developer or a team for fixing. The developer or the team should be provided with all the necessary details to reproduce the issue and fix the defect.
5. *Defect fixing*: The assigned developer or team will work on fixing the defect. Once the defect is fixed, the developer or the team should update the status of the defect in the defect tracking system.

6. *Defect verification*: After the defect is fixed, it needs to be verified to ensure that the issue has been resolved. The tester or the test team should verify the defect to ensure that it has been fixed and that there are no side effects.
7. *Defect closure*: Once the defect has been verified, it can be closed in the defect tracking system. The status of the defect should be updated as "Closed" in the defect tracking system.

Stop Testing Criteria

Stop testing criteria refers to the conditions that need to be met for the testing process to be stopped. These criteria help determine whether the product or system under test is ready for release or if further testing is required. Stop testing criteria may include factors such as meeting predefined requirements, passing a set number of test cases, reaching a specific level of test coverage, and achieving acceptable performance and quality levels. These criteria can vary depending on the specific project, testing goals, and other factors, and they are typically defined in the test plan. It is important for testers and stakeholders to agree on the stop testing criteria before the testing process begins to avoid confusion and ensure that everyone is working toward the same goal.

Reviewing and Approving the Test Plan

Reviewing and approving the test plan is an important step in the testing process. It helps ensure that the test plan is accurate, complete, and meets the requirements of the project.

The review process should involve stakeholders, including the project team, quality assurance team, and management. The review should include a comprehensive analysis of the test plan, including test objectives, scope, approach, resources, schedule, test cases, and test data.

Once the review is complete, the test plan should be approved by the project sponsor or project manager. Approval indicates that the test plan is acceptable, and the testing process can proceed.

It is important to remember that the test plan is a living document, and it may need to be updated or modified as the project progresses. Any changes to the test plan should be reviewed and approved by the appropriate stakeholders.

Benefits of Test Planning

The importance of test planning in software testing cannot be overstated. It is a critical activity that ensures that testing is done effectively and efficiently. The following are the reasons why test planning is important:

- *Provides a clear road map*: Test planning provides a clear roadmap for the testing phase of the SDLC. It outlines the testing objectives, scope, approach, resources, and schedule for testing. This road map helps to ensure that testing is done in a structured and organized manner, which helps to minimize the risk of defects in the software application.
- *Minimizes risk*: Test planning helps to minimize the risk of defects in the software application. By defining a comprehensive set of test cases and identifying the required resources, test planning ensures that all aspects of the software application are tested thoroughly.

- *Reduces costs*: Test planning helps to reduce the costs of testing. By identifying the required resources and developing a realistic and achievable test schedule, test planning helps to ensure that testing is done efficiently and cost-effectively.
- *Improves test coverage*: Test planning helps to improve test coverage by ensuring that all features and functionalities of the software application are tested. This helps to ensure that the software application meets the required quality standards.
- *Increases test efficiency*: Test planning increases test efficiency by defining a testing approach that is appropriate for the testing objectives, scope, and available resources. This ensures that testing is done in a manner that is both effective and efficient.
- *Provides a basis for review and approval*: Test planning provides a basis for review and approval of the testing phase of the SDLC. It ensures that all stakeholders are aware of the testing objectives, scope, approach, resources, and schedule, and have an opportunity to provide input and feedback.

Test Plan Document

Creating a test plan document is an essential part of the test planning process, and it provides a detailed road map for the testing process. The following are usually the contents of a test plan document:

- *Introduction*: This section should introduce the purpose and scope of the test plan document. It should also provide an overview of the software application being tested.

- *Testing objectives*: This section should clearly state the testing objectives and goals that the testing team aims to achieve. The testing objectives should be aligned with the business requirements and should be measurable and achievable.
- *Testing approach*: This section should describe the overall testing approach that the testing team will follow. It should include the testing methods and techniques that will be used and how the tests will be conducted. This section should also include the roles and responsibilities of the testing team.
- *Testing schedule*: This section should provide a detailed timeline for the testing process. It should include the start and end dates for each testing phase, including planning, design, execution, and reporting.
- *Test environment*: This section should describe the test environment required to execute the test cases. It should include information on the hardware, software, and network configurations required for the testing.
- *Test data*: This section should describe the test data required to execute the test cases. It should include the source of the test data, the type of data required, and any constraints or limitations on the test data.
- *Test cases*: This section should provide a detailed list of test cases that will be executed during the testing process. It should include information on the test case ID, the test case description, the expected result, and the status of each test case.

- *Test automation*: This section should describe any test automation that will be used during the testing process. It should include information on the tools that will be used, the test scripts that will be created, and the types of tests that will be automated.
- *Risks and issues*: This section should provide a list of potential risks and issues that could impact the testing process. It should include a description of each risk or issue, the impact it could have on the testing process, and any mitigation strategies that will be used to address it.
- *Reporting and communication*: This section should describe how the testing results will be reported and communicated to the stakeholders. It should include information on the reporting frequency, the format of the reports, and the stakeholders who will receive the reports.
- *Conclusion*: This section should summarize the key points of the test plan document and provide any additional information or recommendations.

Test Planning Tools and Techniques

There are various tools and techniques available for test planning, which can help testing teams to create a comprehensive test plan. The following are some of these tools and techniques:

- *Risk analysis*: Risk analysis is a technique used to identify and analyze potential risks that may occur during the testing process. Risk analysis helps to prioritize and allocate resources effectively to manage risks. There are several methods for performing risk

analysis, such as Failure Modes and Effects Analysis (FMEA), Strengths, Weaknesses, Opportunities, and Threats (SWOT), and Political, Economic, Social, Technological, Environmental, and Legal (PESTEL).

- *Test estimation*: Test estimation is a technique used to estimate the time and effort required to complete the testing process. There are several methods for test estimation, such as expert judgment, historical data, and bottom-up and top-down estimation.
- *Test case design techniques*: Test case design techniques are used to create test cases that cover all possible scenarios and use cases of the software application being tested. Some of the popular test case design techniques are boundary value analysis, equivalence partitioning, and decision table testing.
- *Test management tools*: Test management tools are used to manage the testing process, including test planning, test case creation, execution, and reporting. Some popular test management tools are Jira, HP ALM, and TestRail.
- *Test automation tools*: Test automation tools are used to automate the testing process and improve the efficiency and accuracy of the testing. Some popular test automation tools are Selenium, Appium, and TestComplete.
- *Traceability matrix*: A traceability matrix is a tool used to track the relationship between requirements and test cases. It helps to ensure that all the requirements are covered by the test cases and provides visibility into the testing process.

- *Gantt chart*: A Gantt chart is a tool used to represent the testing schedule in a graphical format. It helps to visualize the testing timeline and identify any potential schedule conflicts.

Summary

When a new feature is under test, we need to make sure that we plan it accurately. This will give a nice overview of what we test and how long it takes. We started simply, by defining the testing objectives and the scope, and then we move on with the testing approach, the resources we need, and the expected schedule. We saw the items that need to be present in the defect management process and when we can decide to stop testing. At the end we created a test plan document, using some common tools and techniques. In my view the most important step is the creation of the test cases, and thankfully there are several test design techniques we can follow to generate more test cases and to test the right things.

CHAPTER 5

Test Design Techniques

Test design techniques are methods used to create test cases that cover all possible scenarios and use cases of the software application being tested. Several test design techniques are available, each with its own strengths and weaknesses. In this chapter, we will discuss some of the popular test design techniques. We will see some common techniques, such as equivalence partitioning and boundary value analysis, and some more obscure but quite useful in complex situations, such as pairwise testing and modified condition decision coverage.

Black-Box Testing

Black-box testing is a software testing technique in which the internal workings of the system being tested are not known to the tester. The tester focuses on the system's inputs and outputs and tests the system's functionality based on predefined requirements and specifications.

In black-box testing, the tester treats the system as a black box and tests it by providing inputs and observing the outputs. The objective of black-box testing is to detect any errors, bugs, or defects that are present in the system's functionality, without knowing how the system works internally.

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4_5

Black-box testing can be performed at different levels of software testing, such as unit testing, integration testing, system testing, and acceptance testing. The techniques used in black-box testing include equivalence partitioning, boundary value analysis, decision table testing, and use-case testing.

Black-box testing has several advantages, such as the following:

- It enables testers to focus on the functionality of the system without worrying about its internal workings.
- It provides a clear separation between the tester and the developer, as the tester does not need to know how the system is implemented.
- It helps to uncover defects and errors that are not visible from the system's code or architecture.

However, black-box testing also has some limitations, such as the following:

- It may not detect errors that are related to the system's internal workings.
- It may not be as effective in identifying complex defects or bugs.
- It may not be suitable for testing performance or security-related issues.

Overall, black-box testing is an important testing technique that can help to ensure the quality and reliability of software systems by detecting defects and errors in the system's functionality.

In this chapter we will go through the following black-box test design techniques:

- *Equivalence partitioning*: Dividing the input data into partitions that should exhibit similar behavior and then selecting representative test cases from each partition

- *Boundary value analysis*: Testing the boundary conditions of input values by selecting test cases at the minimum and maximum input values, just above and below these values, and at points where the input changes value
- *Decision table testing*: A table-based technique that identifies the inputs, conditions, and actions of a system and then constructs combinations of test cases for each possible combination of inputs
- *State transition testing*: Testing a system that can be in one of several states by identifying the states and transitions between them and then constructing test cases that cover each transition at least once
- *Use-case testing*: Testing a system by focusing on the user's needs and requirements, identifying scenarios that represent how a user might interact with the system, and then testing those scenarios
- *Pairwise testing*: A combinatorial technique that generates test cases that exercise all possible pairs of input values
- *Error guessing*: Creating test cases based on an understanding of likely errors that might occur in a system
- *Exploratory testing*: Testing a system by exploring it in an unscripted manner, trying to find errors and unexpected behavior
- *Random testing*: Selecting inputs randomly from the input space to create test cases
- *Ad hoc testing*: Testing a system in an unplanned and unstructured manner, often without a formal test plan or test case documentation

Equivalence Partitioning

Equivalence partitioning is a technique used in software testing that involves dividing a set of test conditions into groups or partitions that are equivalent or similar to each other. The idea behind this technique is to minimize the number of test cases that need to be executed while still ensuring that all possible scenarios are covered.

In equivalence partitioning, test cases are created based on the equivalence classes. Equivalence classes are a set of input conditions that are expected to behave in the same way. For example, if an application has a field that accepts numbers, the input range can be divided into two equivalence classes: valid numbers and invalid numbers. This way, instead of testing each input number, we can test a representative value from each equivalence class.

The objective of equivalence partitioning is to minimize the number of test cases needed to test a system while still ensuring that all possible scenarios are covered. This technique can be used in both functional and nonfunctional testing.

These steps can be followed while applying equivalence partitioning:

1. Identify the input variables of the system to be tested.
2. Divide the input variables into different equivalence classes based on their expected behavior.
3. Develop test cases for each equivalence class.
4. Execute the test cases and analyze the results.
5. Repeat the process for each input variable.

By using equivalence partitioning, testers can save time and effort in testing and ensure that the most important test cases are covered. It is a widely used technique in both manual and automated testing.

Equivalence partitioning is a powerful testing technique that can be used in many situations to optimize the number of test cases needed to achieve high test coverage. It is particularly useful when testing inputs that can take on many different values or ranges of values. By dividing the input space into equivalence classes, we can test a representative set of values from each class, rather than exhaustively testing every possible input value.

However, there are situations where equivalence partitioning may not be the most effective testing technique. For example, if the input space is relatively small and well-defined, it may be more efficient to test every input value rather than dividing them into equivalence classes. Additionally, if the software system being tested is critical or safety-critical, it may be necessary to test every possible input value to ensure that no errors or bugs are present. In general, the decision to use equivalence partitioning or another testing technique should be based on the specific characteristics of the system being tested, the nature of the inputs and outputs, and the desired level of test coverage.

The following are two examples of equivalence partitioning:

User Login Page

Suppose you are testing a user login page for a website. The page requires users to enter their username and password to log in. The input fields for the username and password have specific requirements. The username must be between 6 to 15 characters, and the password must be between 8 to 20 characters. Equivalence partitioning can be used to identify the valid and invalid input values for each field.

For the username field:

- *Valid inputs*: username, 123456, qwertyuioplkj
- *Invalid inputs*: user, 1, qwertyuioplkjzxcvbnm12345

Then they are split into valid and invalid partitions (Table 5-1).

Table 5-1. *Partitions of Username Field*

Invalid	Valid	Invalid
0-5	6-15	>15
Partition 1	Partition 2	Partition 3

For the password field:

- *Valid inputs*: password1, Password123, abcdefghi123456789
- *Invalid inputs*: pass, 1234567, qwertyuioplkjzxcvbnms

Then they are split into valid and invalid partitions (Table 5-2).

Table 5-2. *Partitions of Password Field*

Invalid	Valid	Invalid
0-7	8-20	>20
Partition 1	Partition 2	Partition 3

Credit Card Payment

Suppose you are testing a credit card payment module of an e-commerce website. The module accepts credit card details such as card number, expiration date, and security code. Equivalence partitioning can be used to identify the valid and invalid input values for each field.

For the credit card number field:

- *Valid inputs*: 16-digit credit card numbers
- *Invalid inputs*: Less than or more than 16 digits credit card numbers, alphabetic or special characters

Then they are split into valid and invalid partitions (Table 5-3).

Table 5-3. *Partitions of Credit Card Number Field*

Invalid	Valid	Invalid	Invalid
0-15	16	>16	Non-numbers
Partition 1	Partition 2	Partition 3	Partition 4

For the expiration date field:

- *Valid inputs*: Any future date in MM/YY format
- *Invalid inputs*: Any past date, any date with invalid format

Then they are split into valid and invalid partitions (Table 5-4).

Table 5-4. *Partitions of Expiration Date Field*

Invalid	Valid	Invalid
Past date	Future Date in MM/YY	Invalid format
Partition 1	Partition 2	Partition 3

Exercise

Suppose you are testing a registration form for a mobile app. The form requires users to enter their name, email address, and phone number. The name field must be between 3 to 20 characters, the email address must be in a valid email format, and the phone number must be a 10-digit number. Use equivalence partitioning to identify the valid and invalid input values for each field.

Boundary Value Analysis

Boundary value analysis (BVA) is a software testing technique that is used to identify errors in software by focusing on the boundary conditions of the input domain. The input domain is the set of all possible inputs to a software system. BVA involves testing values that lie on the boundaries of this input domain, since these values are more likely to cause errors.

The BVA technique involves testing the following values:

- *Minimum boundary values*: These are the smallest values that can be used as input for a particular parameter. For example, if the parameter is the number of items that can be added to a shopping cart, the minimum boundary value would be zero.
- *Maximum boundary values*: These are the largest values that can be used as input for a particular parameter. For example, if the parameter is the number of items that can be added to a shopping cart, the maximum boundary value would be the maximum limit set by the system.
- *Just above the minimum boundary values*: These are values that are just above the minimum boundary value. For example, if the minimum boundary value for the number of items that can be added to a shopping cart is zero, the just above value would be one.
- *Just below the maximum boundary values*: These are values that are just below the maximum boundary value. For example, if the maximum boundary value for the number of items that can be added to a shopping cart is 10, the just below value would be 9.

By testing these boundary values, testers can identify any issues that may exist in the software. For example, if the system is designed to allow a maximum of 10 items to be added to a shopping cart, testing with just above and just below values may reveal issues with how the system handles values that exceed this limit.

BVA is a simple yet effective testing technique that can help identify many types of errors in software. It is often used in conjunction with other testing techniques, such as equivalence partitioning, to ensure that software is thoroughly tested and that all possible scenarios are covered.

However, BVA can be avoided when testing scenarios where inputs are not restricted by boundaries or when the boundaries themselves are not well-defined or may change frequently. Additionally, BVA may not be appropriate for testing scenarios where inputs have a large number of variables and dimensions, as identifying and testing every possible boundary condition can be time-consuming and resource-intensive.

The following are two examples of boundary value analysis.

Age Validation

Suppose you are testing a form that requires users to enter their age. The system allows users who are 18 years or older to register. The input field for age has specific requirements. The user's age must be between 18 to 60. In boundary value analysis, the focus is on values that are at the boundaries of this range, which are 18 and 60. Test cases can be designed for these boundary values to check if the system handles them correctly.

- *Test case 1*: *Enter age as 17*: This should be rejected with an appropriate error message.
- *Test case 2: Enter age as 18*: This should be accepted.
- *Test case 3: Enter age as 60*: This should be accepted.
- *Test case 4: Enter age as 61*: This should be rejected with an appropriate error message.

File Size Validation

Suppose you are testing a file upload feature on a website. The feature allows users to upload files of up to 10 MB in size. The focus in boundary value analysis is on values that are at the boundaries of this range, which are 0 MB and 10 MB. Test cases can be designed for these boundary values to check if the system handles them correctly.

- *Test case 1: Upload a file with 0 MB size*: This should be rejected with an appropriate error message.
- *Test case 2: Upload a file with 10 MB size*: This should be accepted.
- *Test case 3: Upload a file with 11 MB size*: This should be rejected with an appropriate error message.

Exercise

Suppose you are testing a calculator application that allows users to perform addition, subtraction, multiplication, and division operations. The system allows the input of only two numbers at a time. The input fields for numbers have specific requirements. The numbers should be between -99999 and 99999. Use boundary value analysis to identify the valid and invalid input values for each field.

Decision Table Testing

Decision table testing, also known as cause-effect graphing, is a black-box testing technique that is used to test systems that have multiple inputs and outputs, where the output is dependent on the input and a set of rules. It is often used in complex systems where there are multiple combinations of inputs that can be tested.

The process of decision table testing involves creating a table that shows all possible combinations of inputs and their corresponding outputs. Each combination of inputs is referred to as a *rule*. The table also includes additional columns to identify the conditions that must be met for each rule to be executed, as well as the actions that must be taken as a result of each rule.

The main benefit of decision table testing is that it helps identify all possible combinations of inputs and outputs, ensuring that all test scenarios are covered. It is also a very structured approach to testing, which makes it easy to document and review the test cases. Decision table testing is often used in conjunction with other testing techniques to ensure complete test coverage.

Decision table testing should be avoided when there are only a few conditions to be tested as it can be time-consuming to create the table and may not provide significant benefits. It may also be less effective when there are dependencies between the conditions and the order of testing matters. In such cases, other techniques such as pairwise testing or exploratory testing may be more suitable. Additionally, decision table testing may not be the best option when the rules or requirements are likely to change frequently as it may require constant updates to the table.

The following are two examples of decision table testing.

E-shop Discounts

An e-commerce website sells various products, and it has different discount offers for each product. The discounts depend on the product type, product price, and customer's age.

For electronics, when the product price is less than $500, then the discount is 5 percent for customers below 18 and 10 percent for customers who are 18 and over. When the product price is $500 or more, then the discount is 10 percent for customers who are below 18 and 20 percent for customers who are 18 and over.

For clothing, when the product price is less than $100, then the discount is 5 percent for customers below 18 and 10 percent for customers who are 18 and over. When the product price is $100 or more, then the discount is 15 percent for customers who are below 18 and 20 percent for customers who are 18 and over.

In this case, a decision table can be created to identify the combinations of inputs that result in a specific discount offer, as shown in Table 5-5.

Table 5-5. *Decision Table*

Product Type	Product Price	Customer Age	Discount Offer
Electronics	>= $500	< 18	10% off
Electronics	>= $500	>= 18	20% off
Electronics	< $500	< 18	5% off
Electronics	< $500	>= 18	10% off
Clothing	>= $100	< 18	15% off
Clothing	>= $100	>= 18	20% off
Clothing	< $100	< 18	5% off
Clothing	< $100	>= 18	10% off

In this example, decision table testing can be used to test the system's ability to correctly apply the discount offers based on the product type, product price, and customer age.

Transportation Price

A transportation company provides transportation services for customers, and it charges based on the distance and the number of passengers. The pricing depends on the type of transportation, which can be either a taxi or a bus.

A taxi with one or two passengers charges $5, $10, and $20 for a distance of 0–5, 5–10, and 10+ km, respectively.

A taxi with three or four passengers charges $10, $20, and $40 for a distance of 0–5, 5–10, and 10+ km, respectively.

A bus with 1–30 passengers charges $50, $100, and $200 for a distance of 0–10, 10–50, and 50+ km, respectively.

In this case, a decision table (Table 5-6) can be created to identify the combinations of inputs that result in a specific price.

Table 5-6. *Decision Table*

Type of Transportation	Number of Passengers	Distance	Price
Taxi	1–2	0–5	$5
Taxi	1–2	5–10	$10
Taxi	1–2	10+	$20
Taxi	3–4	0–5	$10
Taxi	3–4	5–10	$20
Taxi	3–4	10+	$40
Bus	1–30	0–10	$50
Bus	1–30	10–50	$100
Bus	1–30	50+	$200

In this example, decision table testing can be used to test the system's ability to correctly apply the pricing based on the type of transportation, the number of passengers, and the distance.

Exercise

Suppose you are testing a system that provides insurance quotes for customers based on the type of vehicle, driver's age, and driving history.

The system applies different rules for different vehicle types, driver ages, and driving histories.

For vehicles of less than 10 years, if the driver is between 25 and 50 and has no prior accidents, the quote is $50. With prior accidents, the quote is $75.

If the driver is 18–25 or more than 50 and has no prior accidents, the quote is $60. With prior accidents, the quote is $85.

For vehicles of 10 years or more, if the driver is between 25 and 50 and has no prior accidents, the quote is $65. With prior accidents, the quote is $100.

If the driver is 18–25 or more than 50 and has no prior accidents, the quote is $75. With prior accidents, the quote is $110.

Use decision table testing to identify the different combinations of inputs that result in a specific insurance quote.

State Transition Testing

State transition testing is a black-box testing technique used to test the functionality of a system or software that involves the transition of the system from one state to another. In this technique, the behavior of the system is analyzed based on different input conditions and the resulting transitions from one state to another.

The state transition testing technique is commonly used in systems that have a defined set of states and in which the behavior of the system depends on the current state. The testing process involves the identification of all possible states and the transitions between these states. Test cases are then designed to cover each transition and the associated behavior.

The process of state transition testing involves the following steps:

1. *Identify the states of the system*: The first step is to identify all the states of the system. A state is a condition or mode in which the system operates.
2. *Identify the events that cause transitions*: The next step is to identify all the events that cause the system to transition from one state to another.
3. *Create a state transition diagram*: A state transition diagram is created to illustrate the various states of the system and the transitions between them.
4. *Design test cases*: Test cases are designed to cover each transition and the associated behavior.
5. *Execute test cases*: The test cases are executed to verify that the system is functioning as expected.

State transition testing can be a highly effective way to test the functionality of a system, especially for systems with complex state-based behavior. However, it is important to ensure that all possible transitions are covered by the test cases to ensure adequate testing coverage.

State transition testing is most suitable when testing software applications that have states or modes. Examples of such applications are traffic control systems, automated teller machines (ATMs), and elevators. In such applications, testing is required to ensure that the system transitions correctly from one state to another and the correct response is generated. State transition testing is also beneficial in testing applications that rely on user input to progress through various states, such as software games.

However, state transition testing may not be suitable for all types of software applications. For instance, applications that do not have clear states or modes may not benefit from this technique. Similarly,

applications that have a limited number of states or have simple transitions may not require state transition testing. Additionally, state transition testing may not be suitable for applications that are highly dynamic and constantly changing states.

The following are two examples of state transition testing.

Traffic Light System

Suppose you are testing a traffic light system where the light turns from red to green to yellow to red, based on a timer.

In this example, the traffic light system has three states: Red, Green, Yellow.

State Transition Table

Table 5-7 shows the traffic light states.

Table 5-7. *Traffic Light System States*

Current State	Event	Next State
Red	Timer expired	Green
Green	Timer expired	Yellow
Yellow	Timer expired	Red

State Transition Diagram

Figure 5-1 shows the state transition diagram.

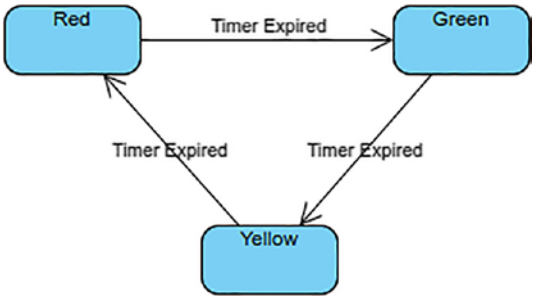

Figure 5-1. *Traffic light state transition diagram*

Shopping Cart

Suppose you are testing an online shopping cart with the option to add or remove items from your cart, check out, complete your order, or cancel everything.

In this example, the shopping cart application has four states: Empty Cart, One Item, Two Items, and Three Items. There is also a special state called Checkout that represents the process of checking out and paying for items in the cart.

The state transition table lists all of the possible events that can occur in each state, along with the resulting next state. For example, if the current state is Empty Cart and the Add Item event occurs, the next state will be One Item. If the current state is One Item and the Proceed to Checkout event occurs, the next state will be Checkout.

By testing all of the possible state transitions and ensuring that the application behaves correctly in each state, we can ensure that the shopping cart application works as expected.

State Transition Table

Table 5-8 shows the shopping cart states.

Table 5-8. *Shopping Cart States*

Current State	Event	Next State
Empty Cart	Add Item	One Item
Empty Cart	Remove Item	Error
One Item	Add Item	Two Items
One Item	Remove Item	Empty Cart
One Item	Proceed to Checkout	Checkout
Two Items	Add Item	Three Items
Two Items	Remove Item	One Item
Two Items	Proceed to Checkout	Checkout
Three Items	Remove Item	Two Items
Three Items	Proceed to Checkout	Checkout
Checkout	Cancel	Empty Cart
Checkout	Confirm Payment	Thank You

State Transition Diagram

Figure 5-2 shows the state transition diagram.

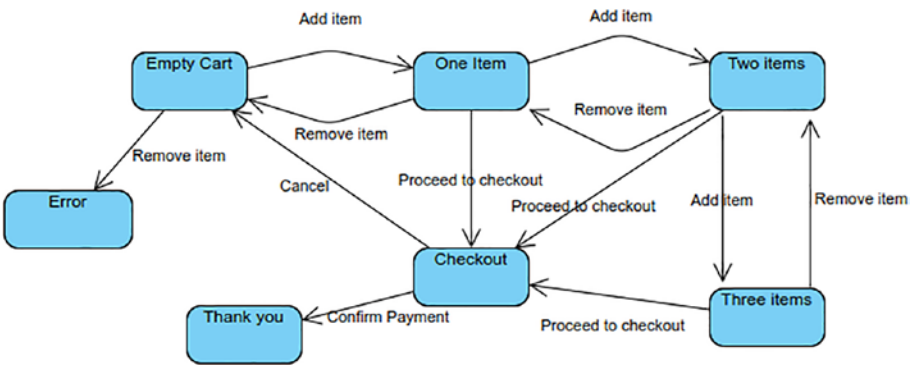

Figure 5-2. *Shopping cart state transition diagram*

Exercise

Suppose you are tasked with testing a vending machine that dispenses beverages. The machine can be in three states: idle, beverage selected, and dispensing. The user can perform the following actions:

1. Insert coins (only accepts denominations of 5, 10, and 25 cents).
2. Select a beverage (cola, lemonade, or water).
3. Cancel the transaction.
4. Dispense the beverage.

Create a state transition diagram for this traffic light system, and test each possible transition to ensure that the system functions as expected. What happens if you try an invalid transition?

Use-Case Testing

Use-case testing is a software testing technique that evaluates the software's ability to meet the end users' requirements. A use case is a specific scenario that describes how a user interacts with the software system. Use-case testing involves creating test cases based on these scenarios to verify that the system behaves as expected.

To perform use-case testing, the tester first identifies the system's different use cases. Then, for each use case, they create test cases that include inputs, expected outputs, and steps to be taken. The test cases are designed to evaluate whether the system meets the specific use case requirements.

The following are the benefits of use-case testing:

- *Improved software quality*: Use-case testing helps to ensure that the software meets the end users' requirements, resulting in higher-quality software.

- *Better test coverage*: Use-case testing ensures that all possible scenarios are tested, leading to better test coverage.
- *Easy to understand*: Use cases are easy to understand, making it easier for stakeholders to provide feedback and review the software.
- *Efficient testing*: Use-case testing is efficient because it focuses on the most important scenarios, reducing the time and effort required for testing.

For example, consider a banking application. A use case for this application could be withdrawing cash from an ATM. Test cases for this use case might include verifying that the ATM is functioning correctly, the user is authenticated, the account balance is updated, and the user receives the requested cash amount. By testing the use case, we can verify that the banking application meets the user's requirements and functions as intended.

Use-case testing is beneficial when the requirements are well-defined and the end users' perspective is considered. This approach is also helpful when the system is expected to undergo changes frequently, as it allows the team to focus on testing the most critical use cases first. Use-case testing can be used to test the system's functionality, as well as to ensure that the system meets the user's needs and expectations.

On the other hand, use-case testing may not be suitable when the system is relatively simple and the requirements are straightforward. It may also not be appropriate when there is not enough information available about the user's perspective and the system's use cases. Additionally, use-case testing can be time-consuming and may require significant effort to design and execute the test cases.

Exercise

Suppose you are tasked with testing a library management system. The use cases for the system include searching for books, checking out books, returning books, and managing user accounts.

Create a list of use cases for this library management system, and test each one, including both valid and invalid scenarios. You also need to ensure that the system can handle all possible combinations of use cases.

Pairwise Testing

Pairwise testing, also known as *all-pairs testing*, is a technique used in software testing to ensure that all possible combinations of input parameters are tested at least once. It is a combinatorial testing method that reduces the number of test cases required to cover all possible combinations of inputs, while still providing a high level of test coverage.

The technique is based on the principle that faults often occur as a result of interactions between input parameters, rather than just individual parameters. By using pairwise testing, it is possible to uncover such interactions and identify defects that might not be detected by other testing methods.

Pairwise testing works by creating test cases that cover all possible pairs of values for each input parameter. For example, if a system has three input parameters, each with four possible values, there would be a total of 64 possible combinations (4×4×4). With pairwise testing, however, it is possible to reduce the number of test cases required to cover all possible pairs of values to just 24, significantly reducing the time and effort required for testing.

Pairwise testing can be performed manually, but there are also many automated tools available that can generate test cases automatically. These tools use algorithms to identify the most efficient set of test cases that will cover all possible pairs of input parameters.

Pairwise testing can be particularly useful in situations where there are a large number of input parameters or where the interactions between input parameters are complex and difficult to predict. By using this technique, testers can ensure that their test cases are efficient and effective, helping to uncover defects early in the development cycle and ultimately improve the quality of the software being developed.

Pairwise testing is particularly effective when there are interactions between different input parameters that may result in unexpected behavior. By systematically varying combinations of input parameters in a controlled manner, it is possible to identify these interactions and ensure that they are adequately tested.

However, pairwise testing may not be appropriate in all situations. For example, if there are dependencies between input parameters that cannot be captured by pairwise testing, additional testing strategies may be necessary. Additionally, if the software being tested has a high degree of complexity or criticality, more rigorous testing approaches may be necessary to ensure that all possible scenarios are adequately covered.

Suppose we are testing a web form that asks for a user's personal information, including their first name, last name, email address, phone number, and country. The form has the following input fields:

First Name (text field, max 50 characters)

Last Name (text field, max 50 characters)

Email Address (text field, max 100 characters)

Phone Number (text field, max 20 characters)

Country (drop-down list of 10 countries)

To apply pairwise testing, we first create a table that lists all the possible values for each input parameter, as shown in Table 5-9.

Table 5-9. *All Users*

First Name	Last Name	Email Address	Phone Number	Country
Alex	Smith	alex@test.com	1234567890	USA
Bob	Jones	bob@test.com	0987654321	Canada
Charlie	Brown	charlie@test.com	5555555555	UK
Dave	Kim	dave@test.com	1112223333	Japan
Emma	Lee	emma@test.com	4445556666	Australia

Next, we group the parameters into pairs and generate all possible combinations, like so:

- First Name, Last Name
- First Name, Email Address
- First Name, Phone Number
- First Name, Country
- Last Name, Email Address
- Last Name, Phone Number
- Last Name, Country
- Email Address, Phone Number
- Email Address, Country
- Phone Number, Country

For each pair, we select test cases that cover all possible combinations of values. For example, for the pair of First Name and Last Name, we can select the following test cases:

- Enter valid values for both fields (e.g., Alex and Smith).
- Enter the maximum allowed number of characters for each field (50 characters).
- Enter an empty value for one field and a valid value for the other (e.g., leave First Name blank and enter Smith for Last Name).
- Enter an invalid value for one field and a valid value for the other (e.g., enter 123 for First Name and Smith for Last Name).
- Enter non-ASCII characters for both fields (e.g., Álex and Smiţh).

We repeat this process for each pair of input parameters, until we have selected test cases that cover all possible combinations of values.

Pairwise testing can significantly reduce the number of test cases required to achieve thorough coverage of the input domain, while still providing good test coverage.

Exercise

Suppose you are testing a reservation system that has four input fields: date, time, location, and number of guests. Using pairwise testing, select a subset of test cases that covers all possible combinations of input values.

Error Guessing

Error guessing is a software testing technique that is based on the tester's experience, intuition, and creativity to identify potential defects in the

software. It is a black-box testing technique that does not follow any formal process or test case design. Instead, the tester applies a range of ad hoc tests based on their intuition and experience with the application and its potential weaknesses.

The process of error guessing involves identifying different scenarios and situations in which the software may behave unexpectedly and then attempting to replicate those scenarios and test the software's response. Testers use their experience and knowledge of the software to anticipate the types of errors that may occur and then design tests to reveal those errors.

Error guessing can be particularly effective when combined with other testing techniques such as equivalence partitioning and boundary value analysis. By using a combination of techniques, testers can identify a wide range of potential errors and increase the likelihood of uncovering defects in the software.

An example of error guessing might be a tester who has extensive experience with a particular software application and knows that it is particularly prone to crashing when certain input values are entered. The tester may design a test case that intentionally enters these input values to see if the software crashes as expected. If the software does not crash, the tester may try other variations on the input values until they uncover the defect. This process of guessing at possible errors and testing them is error guessing.

Error guessing is useful when the tester has limited time and resources to design test cases and execute them. It is also useful when the system being tested has no documentation or formal specifications and the tester needs to rely on their expertise to identify potential errors.

However, it is important to note that error guessing should not be the only technique used for testing. It should be used in conjunction with other techniques such as boundary value analysis, equivalence partitioning, and decision table testing to ensure comprehensive testing of

the system. Additionally, error guessing should not be used as a substitute for proper documentation and requirements gathering, as this can lead to incomplete testing and potential errors being missed.

Registration Form

Suppose you are testing a registration form that has several input fields: name, email address, password, and confirmation password. Based on your experience, you might guess that users will enter invalid characters or leave required fields blank. You can design test cases that try to enter unexpected data such as special characters in the name field, an invalid email address, or a password that is too short.

E-commerce Website

Suppose you are testing an e-commerce website that has a checkout process with several pages. Based on your experience, you might guess that users will have trouble entering shipping information, payment information, or completing the checkout process. You can design test cases that intentionally try to trigger errors, such as trying to submit an order with an invalid credit card number.

Exercise

Suppose you are testing a messaging app that allows users to send and receive messages. Based on your experience, what are some potential errors or defects that users may encounter? Design test cases that intentionally try to trigger those errors.

Exploratory Testing

Exploratory testing is a technique of software testing that emphasizes the freedom and creativity of testers in the software testing process. It is an approach to testing where the tester actively and dynamically learns

about the software while testing it. In other words, the tester designs and executes tests while exploring the software application in an unstructured and informal way.

Exploratory testing is often used in Agile and DevOps environments where the requirements are rapidly changing, and the software needs to be tested quickly and efficiently. It allows testers to adapt quickly to the changes and continuously refine their testing approach based on their understanding of the system.

The process of exploratory testing involves the following steps:

1. *Understanding the objectives*: The tester must understand the objectives and goals of the software testing project.
2. *Creating test charters*: Test charters define the scope of the testing and provide a set of guidelines for the tester to follow.
3. *Executing tests*: During this stage, the tester starts exploring the software application, identifying potential issues and reporting them.
4. *Analyzing test results*: The results of the tests are analyzed, and the tester can identify areas that require further testing.
5. *Documenting the process*: The testing process is documented for future reference, and test cases are updated based on the results.

Exploratory testing is a highly effective technique, as it allows testers to find defects that may not be found through traditional testing methods. It is also an excellent way to identify usability and user experience issues. However, exploratory testing is not a replacement for traditional testing methods. It should be used in conjunction with other testing techniques to ensure that the software is of high quality.

Exploratory testing is useful when the software is complex, when there is limited information available about the software, or when there are no clear specifications or requirements. In such situations, exploratory testing can help to identify defects that may be missed by other types of testing. Exploratory testing is also beneficial when there are new features, changes to existing features, or changes to the environment, and the impact of these changes needs to be tested quickly.

However, exploratory testing may not be the best approach when the software is well-defined and there are clear requirements and specifications available. In such cases, it may be more efficient to use other types of testing, such as boundary value analysis, equivalence partitioning, or decision table testing, which are more structured and systematic. Additionally, exploratory testing may not be suitable for large and complex systems, where a more formal approach to testing is necessary.

Mobile App

Suppose you are testing a mobile app that allows users to take photos and share them with their friends. You might start by exploring the app's user interface and testing the different features, such as taking a photo, adding a filter, and sharing it with friends. As you explore the app, you might try different scenarios, such as taking a photo in low light conditions, using a different filter, or sharing the photo with a friend who has a slow Internet connection.

Online Marketplace

Suppose you are testing an online marketplace that allows users to buy and sell products. You might start by exploring the website's search functionality and testing how well it works for different types of products. As you explore the website, you might try different scenarios, such as searching for a product that has multiple variations, testing the checkout process, or using the website from a mobile device.

Exercise

Suppose you are testing a social media app that allows users to post and share content with their friends. Design and execute an exploratory testing session, where you explore the app and test different scenarios. Try to identify potential defects or issues and document them.

Random Testing

In software testing, there are various techniques and methodologies to design and execute test cases. One such technique is random testing. As the name suggests, it involves selecting random input values from the input domain of the system under test (SUT) and feeding them as input to the system. This approach can uncover faults that may not be detected by other testing techniques. In this chapter, we will discuss the basics of random testing, its advantages, its limitations, and the process involved.

Random testing is a black-box testing technique, which means that the tester does not have knowledge about the internal workings of the software application. The testing technique involves selecting random input values from the input domain of the software application. The input values are selected randomly using a random number generator. The input values can be any combination of characters, numbers, special symbols, or any other data type that the software application accepts as input.

These are some advantages of random testing:

- Random testing is easy to design and execute. Testers do not have to spend a lot of time designing test cases, making it a faster testing technique.
- It is an effective technique for detecting faults in complex systems where it is not possible to test all possible combinations of input values.

- It can detect faults that are not detected by other testing techniques.
- Random testing can also help identify performance issues in the software application as it tests the application with various input values.

These are some limitations of random testing:

- The technique can be less effective for testing applications where there are many dependencies between different parts of the application.
- Since the input values are selected randomly, it is possible that some test cases may not cover important scenarios of the software application.
- It can be difficult to reproduce failures detected by random testing as it may be hard to determine the specific input values that caused the failure.

There are different approaches to random testing, including pure random testing, where test cases are selected completely at random without any constraints or structure, and guided random testing, where inputs are randomly generated based on certain rules or constraints. In pure random testing, inputs are selected using a random number generator, and the output is checked for correctness. Guided random testing, on the other hand, uses domain knowledge or heuristics to guide the selection of input values.

Random testing is an effective technique for detecting corner cases, race conditions, and other subtle errors that might not be found through other testing techniques. However, it is important to note that random testing has some limitations. Since random testing is not based on any specific criteria, it may not be effective if not applied in combination with other test design techniques.

Web Application

Suppose you are testing a web application that allows users to register and log in. You can use random testing to create random strings of characters to use as input for the username and password fields. By doing so, you can test the application's ability to handle different types of input, such as special characters and long strings of text.

Game Testing

Suppose you are testing a new video game that has just been developed. You can use random testing to test the game's ability to handle unexpected input from the player. For example, you can use random movements of the controller or keyboard to simulate different scenarios that may occur during gameplay, such as sudden changes in direction or button mashing.

Exercise

Suppose you are testing a calculator application. How would you use random testing to test the application's functionality?

Ad Hoc Testing

Ad hoc testing is a form of exploratory testing that is performed without a formal test plan or documented test cases. In this technique, testers randomly and informally test the system with the goal of discovering issues that might not be found by scripted testing. Ad hoc testing is often used in conjunction with other testing techniques, such as manual testing and automated testing.

These are some advantages of ad hoc testing:

- *Flexibility*: Ad hoc testing is flexible, allowing testers to improvise and use their knowledge of the system to test it in a way that might not be captured by formal test cases.
- *Early detection of defects*: Ad hoc testing can identify defects in the system early in the development cycle, before formal test cases are developed.
- *Efficient*: Ad hoc testing is an efficient way to test the system because it does not require time to develop formal test cases or test plans.

These are some disadvantages of ad hoc testing:

- *No documentation*: Ad hoc testing is not documented, making it difficult to reproduce defects and track the testing progress.
- *Limited scope*: Ad hoc testing may not cover all the areas of the system because it is not based on a formal test plan.
- *Unstructured*: Ad hoc testing is unstructured, meaning that there is no predefined approach or methodology for testing the system.

Example of Ad Hoc Testing

Suppose a tester is testing an e-commerce website. During ad hoc testing, the tester might decide to test the website's search functionality by entering random search queries to see if the website returns accurate results. The tester might also test the website's navigation by randomly clicking links to see if they take the user to the correct pages.

Exercise

Consider a social media application that allows users to post and view photos. Perform ad hoc testing on the application to identify defects that might not be found by scripted testing. Document the defects found during testing and report them to the development team. Review the ad hoc testing with other testers to ensure that all areas of the system have been covered.

White-Box Testing

White-box testing is a software testing technique that is used to test the internal workings of an application or system. In white-box testing, the tester has knowledge of the internal structure and implementation of the software being tested. Here are some white-box testing techniques:

- *Statement coverage*: This technique involves testing each statement of the code to ensure that every statement has been executed at least once. The goal is to ensure that all statements are covered by tests.
- *Branch coverage*: This technique involves testing each branch of the code to ensure that every possible branch has been executed at least once. The goal is to ensure that all branches of the code have been tested.
- *Path coverage*: This technique involves testing all possible paths through the code to ensure that every possible path has been executed at least once. The goal is to ensure that all possible paths have been tested.
- *Condition coverage*: This technique involves testing each condition of the code to ensure that every possible condition has been executed at least once. The goal is to ensure that all conditions have been tested.

- *Decision coverage*: This technique involves testing every decision point in the code to ensure that each decision has been executed at least once. The goal is to ensure that all decisions have been tested.
- *Multiple condition coverage*: This technique involves testing all possible combinations of conditions in the code to ensure that all possible outcomes have been tested.
- *Modified condition/decision coverage (MC/DC)*: This is a white-box testing technique that ensures that every condition in a decision has been tested independently and that every possible combination of conditions has been tested.
- *Loop testing*: This technique involves testing the code within a loop to ensure that the loop executes correctly and terminates properly.
- *Data flow testing*: This technique involves testing how data flows through the code to ensure that it is being handled correctly.
- *Static testing*: This technique involves analyzing the code without executing it to identify potential defects or issues. This can be done using tools such as code reviews, code analysis, and static analysis.

By using these techniques, white-box testing can help identify defects that might not be caught by other testing techniques. However, it requires a high level of technical knowledge and expertise to be able to design and execute these tests effectively.

Statement Coverage

Statement coverage is a type of white-box testing technique that is used to determine how many statements in the source code of a software system have been executed during the testing process. The goal of statement coverage is to ensure that each statement in the code has been executed at least once during the testing process.

The statement coverage technique involves executing the test cases that have been designed for a software system and tracking the number of statements that are executed during the testing process. A statement is considered to be executed if it is encountered by the program during the execution of a test case. Once all test cases have been executed, the percentage of statements that have been executed is calculated. The goal of statement coverage is to achieve 100 percent statement coverage, which means that every statement in the code has been executed at least once.

Statement coverage is a useful technique for finding defects in software systems. By ensuring that every statement in the code has been executed at least once, statement coverage can help to uncover defects that might not have been found through other testing techniques. Additionally, statement coverage can help to ensure that the software system is functioning as intended and that all of the code is working as expected.

However, statement coverage has some limitations. For example, achieving 100 percent statement coverage does not guarantee that the software system is completely free of defects. It is possible to have a software system with 100 percent statement coverage that still contains defects. Additionally, statement coverage can be a time-consuming process, especially for large software systems.

For example, consider the following code:

```
1. def add_numbers(a, b):
2.     if a > 0 and b > 0:
3.         result = a + b
4.     else:
5.         result = 0
6.     return result
```

To test the statement coverage of this code, we need to write test cases that execute each statement at least once.

Table 5-10 shows the minimum set of test cases to achieve 100 percent statement coverage.

Table 5-10. *Statement Coverage Test Cases (Add Numbers)*

Test Case	A	b	Result
1	3	4	7
2	0	2	0

As another example, consider the following function that calculates the average of a list of numbers:

```
def average(numbers):
    """
    Returns the average of a list of numbers.
    """
    if len(numbers) == 0:
        return None
    else:
        sum = 0
        for number in numbers:
            sum += number
        return sum / len(numbers)
```

To achieve statement coverage for this function, we would need to execute each line of code at least once during testing. This would require at least two test cases: one where the input list is empty, and one where the input list contains at least one number.

Table 5-11. *Statement Coverage Test Cases (Average)*

Test Case	Numbers	Result
1	empty	None
2	1,2,3	2

However, achieving statement coverage alone is not enough to guarantee that the code is completely tested. It is possible for all statements to be executed, but certain logic paths may not be tested. Therefore, other testing techniques such as branch coverage and path coverage should also be used in conjunction with statement coverage to ensure thorough testing.

Exercise

Consider the following Python function that takes a list of integers as input and returns a new list with all the even numbers in the input list:

```
def get_even_numbers(numbers):
    """
    Returns a new list with all the even numbers in the input list.
    """
    result = []
    for number in numbers:
        if number % 2 == 0:
            result.append(number)
    return result
```

Write two test cases that achieve statement coverage for this function. Make sure to include at least one test case where the input list contains no even numbers.

Branch Coverage

Branch coverage is a white-box testing technique that evaluates whether all the branches in the program source code are executed at least once. In other words, it ensures that all the possible paths in the source code are executed at least once. This technique is used to measure the effectiveness of the testing and quality of the code. It helps to ensure that the program's control flow is working as expected.

The primary objective of branch coverage testing is to determine whether each decision point in the code has been tested. The decision point is a code statement that determines which of two paths to take. Branch coverage is measured by the percentage of the total number of decision points executed during the testing process.

The branch coverage testing process involves following the code paths to determine whether each possible branch is executed at least once. This technique requires a thorough understanding of the code structure to identify all the decision points. Once the decision points have been identified, the tester creates a test case that exercises each path.

For example, consider a code block with an `if-else` statement. The `if-else` statement has two branches: one that is executed if the condition is true and the other if the condition is false. To test branch coverage, we must create test cases that execute both the true and false branches at least once.

Here are some benefits of branch coverage testing:

- It helps to identify the potential faults in the code by testing all possible paths.
- It ensures that the program's control flow is working as expected, reducing the risk of runtime errors.
- It provides metrics to measure the testing effectiveness and code quality.
- It provides confidence in the code quality, helping to reduce the cost of fixing defects.

For example, consider the following code:

```
1. def add_numbers(a, b):
2.     if a > 0 and b > 0:
3.         result = a + b
4.     else:
5.         result = 0
6.     return result
```

To test the branch coverage of this code, we need to write test cases that execute each branch at least once.

Table 5-12 shows the minimum set of test cases to achieve 100 percent branch coverage.

Table 5-12. *Branch Coverage (Add Numbers)*

Test Case	A	b	Result
1	3	4	7
2	0	2	0

Suppose we have the following function in Python:

```
def max_of_two_numbers(x, y):
    if x > y:
        return x
    else:
        return y
```

To achieve 100 percent branch coverage for this function, we need to test both possible outcomes of the `if` statement: when x is greater than y, and when x is not greater than y.

Table 5-13. *Branch Coverage (Max of Two Numbers)*

Test Case	a	b	Result
1	3	4	4
2	3	2	3

However, even achieving branch coverage may not be sufficient to test all possible code paths. For example, in the previous code, the case where the input is not an integer is not tested. Therefore, additional testing techniques such as path coverage and condition coverage may also be necessary to ensure thorough testing.

Exercise

Consider the following Python function that takes an integer as input and returns a string indicating whether the input number is positive, negative, or zero:

```
def get_sign(num):
    """
    Returns a string indicating the sign of the input number.
    """
    if num > 0:
        return "positive"
    elif num < 0:
        return "negative"
    else:
        return "zero"
```

Write the test cases that achieve branch coverage for this function.

Path Coverage

Path coverage is a white-box testing technique that involves testing all possible paths of a program. A path is a sequence of statements that begins with the entry point and ends with the exit point of the program. Path coverage aims to test each unique path at least once to ensure that the program is functioning as expected.

To achieve path coverage, a test case must execute every statement in the program at least once. It also requires that all possible branches are executed. A branch is a point in the program where a decision is made, such as an `if` statement or a loop condition. Each branch has at least two possible outcomes, and all possible outcomes must be executed at least once.

Path coverage is a more thorough testing approach than statement coverage or branch coverage because it takes into account the interaction between different paths in the program. It is a useful technique for identifying hard-to-find defects that may not be detected by other testing methods. However, path coverage can be time-consuming and difficult to achieve for complex programs. In addition, it may not be necessary for all programs, depending on the level of risk associated with the software being tested.

For example, consider the following code:

```
1. def add_numbers(a, b):
2.     if a > 0 and b > 0:
3.         result = a + b
4.     else:
5.         result = 0
6.     return result
```

To achieve 100 percent path coverage of this code, we need to test all possible paths through the code. Table 5-14 shows the minimum set of test cases to achieve 100 percent path coverage.

Table 5-14. *Path Coverage (Add Numbers)*

Test Case	A	b	Result
1	3	4	7
2	0	2	0

In this example, we have tested every possible path through the code, ensuring that all possible combinations of control flow paths are executed at least once. However, even achieving path coverage may not be sufficient

to test all possible code paths. Therefore, additional testing techniques such as condition coverage and modified condition/decision coverage may also be necessary to ensure thorough testing.

Consider the following Python function that takes three integers as input and returns the sum of the two largest numbers:

```
def sum_of_largest_two(a, b, c):
    """
    Returns the sum of the two largest numbers among a, b, and c.
    """
    if a >= b and a >= c:
        if b >= c:
            return a + b
        else:
            return a + c
    elif b >= a and b >= c:
        if a >= c:
            return b + a
        else:
            return b + c
    else:
        if a >= b:
            return c + a
        else:
            return c + b
```

To achieve path coverage for this function, we would need to execute every possible path through the control flow of the function. This would require at least three test cases: one where a is the largest number, one where b is the largest number, and one where c is the largest number.

For example, we could write the test cases shown in Table 5-15.

Table 5-15. *Path Coverage (Sum of Largest Two)*

Test Case	A	b	c	Result
1	1	2	3	5
2	2	1	3	5
3	3	2	1	5

Exercise

Consider the following Python function that takes two lists of integers as input and returns a new list containing the elements that are common to both input lists:

```
def find_common_elements(lst1, lst2):
    """
    Returns a new list containing the elements that are common
    """
    result = []
    for item1 in lst1:
        if item1 in lst2:
            result.append(item1)
    return result
```

Write the test cases that achieve path coverage for this function.

Condition Coverage

Condition coverage is a type of white-box testing technique that aims to ensure that all possible combinations of Boolean conditions in a program have been executed at least once. The technique focuses on evaluating the different ways in which a program's conditions can be evaluated as true or false. Condition coverage can be applied to both procedural and

object-oriented programs, and it is typically used to verify the correctness of complex branching structures, such as loops, `if` statements, and `switch` statements.

Condition coverage requires a test case to cover every possible outcome of each Boolean condition in the program. A test case can cover a Boolean condition in one of two ways: (1) it evaluates the condition to true, and (2) it evaluates the condition to false. In this way, condition coverage ensures that all possible combinations of Boolean conditions are evaluated during the testing process.

These are some advantages of condition coverage:

- *Helps identify hidden errors*: Condition coverage ensures that all possible combinations of Boolean conditions are evaluated during testing. This means that even the hidden errors in the program, which might not be evident during normal execution, are discovered.
- *Increases test case effectiveness*: Condition coverage ensures that test cases are optimized to find errors. By focusing on the different ways in which a program's conditions can be evaluated as true or false, testers can identify and eliminate redundant test cases, thereby increasing the effectiveness of the test cases.

These are some disadvantages of condition coverage:

- *Time-consuming*: Condition coverage requires a large number of test cases, which can be time-consuming to design and execute.
- *Code complexity*: Programs with complex branching structures can require a large number of test cases to achieve complete condition coverage. This can make it difficult for testers to achieve complete coverage and make it harder to maintain the test suite.

Here is an example to illustrate condition coverage:

```
if (a > b && c < d) {
  // do something
}
```

To achieve condition coverage, tests would need to be created that cover both the true and false outcomes of both conditions: a > b and c < d. This would require four tests in total:

1. a > b is true, c < d is true.
2. a > b is true, c < d is false.
3. a > b is false, c < d is true.
4. a > b is false, c < d is false.

Another example would be this function that classifies a triangle based on the lengths of its sides:

```
def classify_triangle(a, b, c):
    """
    Classifies a triangle with sides of length a, b, and c
    """
    if a <= 0 or b <= 0 or c <= 0:
        return "Invalid"
    if a == b and b == c:
        return "Equilateral"
    if a == b or b == c or c == a:
        return "Isosceles"
    return "Scalene"
```

To achieve condition coverage, tests would need to be created that cover both the true and false outcomes of all conditions: a <= 0, b <= 0, and c <= 0, a == b, b == c, c == a. This would require two tests in total:

1. a <= 0 is true and b <= 0 is true and c <= 0 is true and a == b is false and b == c is false and c == a is false.
2. a <= 0 is false and b <= 0 is false and c <= 0 is false and a == b is true and b == c is true and c == a is true.

As you probably observed, this set of tests, although it provides 100 percent condition coverage, does not cover all the possible situations that arise in the code.

Exercise

How would you improve the set of tests in the previous example?

Decision Coverage

Decision coverage is a white-box testing technique that is used to measure the effectiveness of testing by checking if all decisions made in the source code have been exercised. A decision is a condition that can lead to either true or false outcome. Decision coverage measures the percentage of decisions that have been executed during testing.

The goal of decision coverage is to ensure that each decision in the code has been executed at least once and that all possible outcomes (true and false) have been tested. This helps to ensure that the code is free of errors and will function as expected under all possible scenarios.

To achieve decision coverage, testers must create test cases that execute every decision point in the code. This means that every possible outcome of each decision point must be tested. For example, if a decision point is a simple `if-else` statement, then there are two possible outcomes that must be tested.

Decision coverage is important because it helps to ensure that all possible outcomes of the code have been tested. This is particularly important in safety-critical systems where a failure in the code could have catastrophic consequences.

Here is an example to illustrate decision coverage:

```
if (a > b && c < d) {
  // do something
}
```

To achieve decision coverage, we need to ensure that both possible outcomes of the `if` statement have been executed. This would require two tests in total.

1. a > b is true; c < d is true.
2. a > b is false; c < d is false.

Another example would be this function that classifies a triangle based on the lengths of its sides:

```
def classify_triangle(a, b, c):
    """
    Classifies a triangle with sides of length a, b, and c
    """
    if a <= 0 or b <= 0 or c <= 0:
        return "Invalid"
    if a == b and b == c:
        return "Equilateral"
    if a == b or b == c or c == a:
        return "Isosceles"
    return "Scalene"
```

To achieve decision coverage, we need to ensure that both possible outcomes of the `if` statements have been executed. This would require six tests in total.

1. If a <= 0 or b <= 0 or c <= 0 is true
2. If a <= 0 or b <= 0 or c <= 0 is false
3. If a == b and b == c is true
4. If a == b and b == c is false

5. If a == b or b == c or c == a is true
6. If a == b or b == c or c == a is false

Exercise

How would you improve the set of tests in the previous example?

Multiple Condition Coverage

Multiple condition coverage (MCC) is a white-box testing technique used to ensure that all possible combinations of Boolean conditions are tested at least once. MCC is a stronger form of condition coverage.

The idea behind multiple condition coverage is to test all possible combinations of Boolean conditions that may appear in a decision. It is important to note that this technique is applicable only to code that uses Boolean conditions to make decisions. To achieve MCC, all possible combinations of each condition in the decision must be evaluated. For example, if a decision has two conditions, A and B, then four test cases are needed to cover all possible combinations (true/true, true/false, false/true, and false/false).

MCC is a powerful technique, as it ensures that all combinations of conditions are tested, which can help uncover complex logic errors. However, it can also result in a large number of test cases, which can be difficult to manage. Therefore, it is important to use MCC judiciously and to combine it with other testing techniques, such as decision coverage, to achieve a more comprehensive test suite.

To implement MCC, the tester must carefully analyze the code to identify all decision statements and then create test cases to cover all possible combinations of conditions within each decision. Test cases are created by varying the values of each condition between true and false and evaluating the resulting decision outcome. Once all combinations have been tested, the decision can be deemed to have achieved MCC.

Here is an example to illustrate MCC:

```
function getResult(x, y, z) {
  if (x > 0 && y > 0 && z > 0) {
    return "Positive";
  } else if (x < 0 && y < 0 && z < 0) {
    return "Negative";
  } else {
    return "Mixed";
  }
}
```

To achieve multiple condition coverage for this function, we need to test all possible combinations of truth and falsehood for each condition. There are three conditions, each with two possible outcomes, so there are a total of $2^3 = 8$ possible combinations to test.

We can use a truth table (Table 5-16) to keep track of which combinations we have tested.

Table 5-16. *MCC (getResult)*

x>0	y>0	z>0	Expected Result
T	T	T	Positive
T	T	F	Mixed
T	F	T	Mixed
T	F	F	Mixed
F	T	T	Mixed
F	T	F	Mixed
F	F	T	Mixed
F	F	F	Negative

By testing all eight combinations, we can be sure that we have achieved multiple condition coverage for this function.

Consider this function that calculates the shipping cost for a package based on its weight, size, and destination:

```
def calculate_shipping(weight, size, destination):
    """
    Calculates the shipping cost for a package based on
    """
    if weight <= 0 or weight > 50:
        return "Invalid weight"
    elif size <= 0 or size > 100:
        return "Invalid size"
    elif destination not in ["USA", "Canada", "Mexico"]:
        return "Invalid destination"
    elif weight <= 10:
        if size <= 20:
            return 10
        elif size <= 50:
            return 20
        else:
            return 30
    else:
        if size <= 20:
            return 20
        elif size <= 50:
            return 30
        else:
            return 40
```

There are multiple Boolean conditions that need to be evaluated for multiple condition coverage:

- `weight <= 0 or weight > 50`
- `size <= 0 or size > 100`
- `destination not in ["USA", "Canada", "Mexico"]`

- `weight <= 10`
- `size <= 20`
- `size <= 50`

Table 5-17 shows a possible set of tests.

Table 5-17. *MCC (calculate_shipping)*

weight	size	destination	Expected Result
5	10	“USA”	10
5	30	“USA”	20
15	70	“Canada”	40
-5	10	“Mexico”	“Invalid weight”
5	-10	“USA”	“Invalid size”
5	10	“Japan”	“Invalid destination”

Exercise

Consider the following function that determines the letter grade for a student based on two exam scores:

```
def get_grade(grade1, grade2):
    """
    Determines the letter grade for a student based on two exam scores.
    """
    if grade1 < 0 or grade1 > 100 or grade2 < 0 or grade2 > 100:
        return "Invalid input"
    average = (grade1 + grade2) / 2
    if average >= 90:
        if grade1 >= 80 and grade2 >= 80:
            return "A"
        else:
            return "A-"
    elif average >= 80:
        if grade1 >= 70 and grade2 >= 70:
            return "B"
        else:
            return "B-"
    elif average >= 70:
        if grade1 >= 60 and grade2 >= 60:
            return "C"
        else:
            return "C-"
    else:
        return "F"
```

Write the test cases required to achieve 100 percent MCC coverage.

Modified Condition/Decision Coverage

Modified condition decision coverage (MC/DC) is a white-box testing technique that ensures all possible combinations of conditions and decisions have been executed in a software system. It is considered to be one of the most stringent testing techniques and is often used in safety-critical systems such as aviation, defense, and medical devices.

The MC/DC technique requires each condition to be evaluated as true and false at least once and that each decision is tested under all possible combinations of conditions. In addition, it requires that each condition must be independently evaluated and have an effect on the decision outcome. This means that all possible variations of conditions and decisions are tested to ensure complete coverage of the code.

The MC/DC technique can be applied at various stages of the software development life cycle, including the requirements phase, design phase, and testing phase. It can also be used in conjunction with other testing techniques to ensure complete code coverage. However, implementing MC/DC can be time-consuming and requires a high level of expertise, as it involves understanding the code logic and identifying all possible decision combinations.

Here is an example to illustrate MC/DC:

```
function checkValues(a, b, c) {
  if (a > 0 && b > 0 || c > 0) {
    return true;
  } else {
    return false;
  }
}
```

First we create a truth table.

We can use the truth table shown in Table 5-18 to keep track of all possible combinations.

Table 5-18. *Truth Table (checkValues)*

Test Case	a>0	b>0	c>0	Expected Result
1	T	T	T	true
2	T	T	F	true
3	T	F	T	true
4	T	F	F	false
5	F	T	T	true
6	F	T	F	false
7	F	F	T	true
8	F	F	F	false

Now we need to find the test cases where changing a condition's input affects the whole decision. We can see that for the condition a > 0 to be the decisive condition, we need to use test cases 2 and 6. For the condition b > 0 to be the decisive condition, we need to use test cases 2 and 4. For the condition c > 0 to be the decisive condition, we need to use test cases 3 and 4 (5 and 6 would also be correct). This leads to the set of test cases shown in Table 5-19.

Table 5-19. *MCDC (checkValues)*

Test Case	a>0	b>0	c>0	Expected Result
1	T	T	F	true
2	T	F	T	true
3	T	F	F	false
4	F	T	F	false

The idea behind MCDC is similar to the algorithm of the Karnaugh map. MCDC's benefit is that you can test all the possible combinations of conditions and decisions with fewer test cases, and it is encouraged to be used in situations where we don't have enough time or resources to execute the complete set of combinations.

For a given N number of inputs, you always get N+1 test cases when you are using the MCDC technique. If we used the MCC, the total number of test cases would be 2^N. In our example, we had three inputs (a > 0, b > 0, c > 0) and the test cases are four (instead of eight for MCC). If we had four inputs (a > 0, b > 0, c > 0, d > 0), the test cases would be five (instead of sixteen for MCC).

Loop Testing

Loop testing is a technique used in software testing to ensure that loops within a program are functioning correctly. This type of testing is used to identify defects or errors that can occur when a program is executed multiple times, as is often the case with loops.

The goal of loop testing is to ensure that the loop is executed the correct number of times, that the loop is exited when it is supposed to be exited, and that the loop is executed with the correct inputs and outputs. This is accomplished by testing the loop under a variety of conditions, including boundary conditions and error conditions.

There are several different types of loop testing techniques that can be used, including the following:

- *Simple loop testing*: In this technique, the loop is executed with the minimum and maximum values of the loop index, as well as with values that are in between the minimum and maximum values. This is done to ensure that the loop is executing the correct number of times.

- *Nested loop testing*: In this technique, the loops that are nested within the main loop are tested individually, as well as together. This is done to ensure that the loops are interacting correctly with one another.
- *Concatenated loop testing*: In this technique, multiple loops are executed together to test their interaction. This is done to ensure that the loops are executing in the correct order and that they are interacting correctly with one another.
- *Iterative loop testing*: In this technique, the loop is executed with a variety of input values to ensure that the loop is functioning correctly under different conditions.

Loop testing is an important part of software testing, as loops are a fundamental part of most software programs. By testing loops thoroughly, software testers can identify and eliminate defects and ensure that the program is functioning correctly under a variety of conditions.

Let's see an example with the following piece of code:

```
function sumArray(arr) {
  let sum = 0;
  for (let i = 0; i < arr.length; i++) {
    sum += arr[i];
  }
  return sum;
}
```

This function takes an array of numbers and returns the sum of all the numbers in the array. The `for` loop iterates over each element in the array and adds it to the sum variable.

To test this function using loop testing, we would want to design our test cases to exercise the loop as thoroughly as possible. The following are some examples of test cases we might consider:

- An array with no elements
- An array with one element
- An array with two elements
- An array with many elements
- An array with all elements negative
- An array with all elements positive
- An array with a mix of positive and negative elements
- An array with large numbers that could cause overflow

We would want to test our function with a variety of inputs to ensure that it handles edge cases and potential issues with the loop, such as off-by-one errors, infinite loops, and so on.

Loop testing can be time-consuming, as it requires running the loop multiple times with different inputs. However, it can be an effective way to identify and correct issues related to loops in code.

Data Flow Testing

Data flow testing is a technique that focuses on the flow of data within the software being tested. It helps to identify whether the data is being processed correctly as it moves through the different components of the software. This testing technique can identify vulnerabilities, such as buffer overflows, and ensure that the software is free of defects that may occur because of the improper flow of data.

An example of data flow testing could involve testing a financial application to ensure that the correct data is being processed at each stage of a transaction. For example, when a customer initiates a transaction, the application must correctly process their account details and the transaction amount. The data must then flow through the different components of the application, such as the authentication module and the payment gateway, without being corrupted or lost.

To perform data flow testing, testers must identify the different data flows within the application and the different components that process the data. They then create test cases that focus on the different flows to ensure that the data is being processed correctly at each stage.

For example, a test case may focus on a specific data flow within the application, such as the flow of user data from the front-end interface to the back-end database. Testers can then create a test scenario that simulates the flow of data, including inputs and expected outputs at each stage. They can then execute the test and compare the actual output with the expected output to identify any discrepancies or errors. This helps to ensure that the data is being processed correctly and that the application is functioning as intended.

Static Testing

Static testing is a software testing technique that involves examining the code, design, and associated documentation of an application without actually executing it. The purpose of static testing is to identify defects early in the development life cycle to reduce the cost of fixing them later on.

Some examples of static testing are the following:

- *Code reviews*: This is a manual inspection of the source code by the developers or other team members to detect any issues or defects before the code is executed. Code reviews can be conducted in different ways, such as formal inspections or informal reviews.
- *Walk-throughs*: This is a collaborative review process where the author of a document or code walks through it with other team members to explain their thought process and clarify any doubts. This technique is helpful in identifying issues early in the development process.
- *Pair programming*: In this technique, two programmers work together on a single workstation to write code, review it, and test it in real time. This technique is particularly useful for complex code where collaboration can improve the quality of the output.
- *Static analysis tools*: These are software tools that analyze the source code without executing it to identify issues such as coding standards violations, security vulnerabilities, or potential performance problems.
- *Document reviews*: This involves reviewing project documentation such as requirements, design specifications, and test plans to identify any inconsistencies or issues that could impact the quality of the final product.
- *Checklist-based reviews*: This is a structured approach to static testing where a checklist of potential issues is used to review the code or documentation. The checklist is based on industry best practices and can be customized to suit the specific project requirements.

- *Standards compliance review*: This is a review to ensure that the project documentation and source code follow the relevant standards and guidelines such as coding standards, software development methodologies, and industry best practices.

Summary

In this chapter, we went through several examples of test design techniques. We saw how to create test cases by using these powerful techniques with real examples. Some techniques are more elaborate than other; for example, decision table testing makes sure that all the possible combinations are performed, and some techniques are using several tricks to eliminate test cases when the time or the budget are not enough for testing everything. For example, pairwise testing will significantly reduce the number of test cases, while covering the most important combinations. There are structured techniques, such as boundary value analysis, and free format techniques, such as exploratory testing. They are all useful, and the best practice is to use them in combination. When all the test cases are defined, it is time to execute them!

CHAPTER 6

Test Execution

Test execution is the process of running the tests that have been designed and planned in the previous stages of the software testing life cycle. This chapter will cover the importance of test execution, the test execution process, and the various techniques and tools used in test execution. We will go through the test execution process, the needs of the environment setup, and the defect reporting and retesting. We will identify the needs for regression, how to report the status of the execution, and when to complete the activities. We will see some techniques and tools that we can assist us, and we will choose some quality metrics for our product.

Getting Started

The first step in preparing for test execution is to ensure that the testing environment is set up correctly. This includes setting up test beds, configuring hardware and software, and installing the necessary tools and utilities. It is also important to ensure that the test environment is as close to the production environment as possible to ensure that the testing results are accurate.

Another important aspect of test execution preparation is the creation of test cases. Test cases should be created based on the requirements and design documents and should cover all possible scenarios that the software could encounter. The test cases should be reviewed and approved by the stakeholders before they are executed.

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4_6

Test data preparation is also an important part of preparing for test execution. Test data should be selected to cover all possible scenarios, and it should be representative of real-world data. The data should be created and selected based on the test cases and should be reviewed and approved by the stakeholders before it is used.

Finally, it is important to ensure that the testing team is trained and prepared to execute the tests. The team should understand the testing process, the test cases, and the expected results. They should also be trained on the tools and utilities that they will be using to execute the tests. This will help ensure that the testing is conducted efficiently and effectively.

Test Execution Process

Test execution ensures that the software under test meets the required quality standards and that it functions as expected. Without proper test execution, defects may go unnoticed, and the software may not function as intended, which can lead to user dissatisfaction, revenue loss, and brand damage.

The test execution process involves the following steps:

1. *Test environment setup*: The test environment must be set up in a way that simulates the production environment, ensuring that the software under test behaves as it would in the real world.

2. *Defect reporting and retesting*: Any defects that are discovered during the test execution process are reported to the development team. After the development team has fixed the reported defects, they must be retested to ensure that they have been resolved.

3. *Regression testing*: Regression testing is the process of re-executing previously passed tests to ensure that the changes made to the software have not introduced new defects.

4. *Test case status reporting*: The status of each test case should be reported to the test manager. This includes information on which test cases have been executed, which have passed, which have failed, and which are still outstanding.

5. *Test case completion*: Once all the test cases have been executed, the test manager should review the results to determine whether the software meets the required quality standards. The test manager should also review the defect reports and determine whether any further testing is required.

The test case execution process should be conducted systematically and rigorously to ensure that all possible defects are identified and fixed before the software is released to the end users.

Test Environment Setup

Test environment setup refers to the process of preparing a testing environment with the required hardware, software, and network configurations to conduct software testing.

The test environment should be similar to the production environment to accurately simulate the end users' experience. A well-designed test environment ensures that the testing process is performed under conditions similar to those of the real-world environment, reducing the risk of errors and providing accurate results.

Here are some steps involved in setting up a test environment:

1. Identify the hardware, software, and network configurations required for testing.
2. Create a plan and a checklist to document the hardware and software requirements.
3. Install the required software and hardware.
4. Configure the network according to the specifications of the production environment.
5. Create user accounts and assign appropriate permissions for testing.
6. Prepare test data and ensure that it accurately reflects the production environment.
7. Install the test management tool and configure it according to the testing requirements.
8. Verify that the test environment is set up correctly by performing smoke testing.
9. Document the test environment setup process and include it in the test plan.

It is essential to test the test environment to ensure that it is stable and working correctly. The test environment should be tested for compatibility, connectivity, and stability. The testing team should also ensure that the test data is accurate and reflects the production environment. The test environment should be documented and version controlled to ensure that it can be quickly restored in case of a failure or issue.

Defect Reporting and Retesting

Defect reporting involves identifying and documenting any issues or defects found during testing and communicating them to the development team to be fixed. Typically the defects are reported on a tool like Jira or TFS, where the testers can create tickets and add the details. The process of defect reporting typically involves the following steps:

1. *Reproduce the defect*: The first step in defect reporting is to try to reproduce the issue or defect. This is important because it helps to verify that the defect is real and not a one-time occurrence.
2. *Document the defect*: Once the defect has been reproduced, it needs to be documented in detail. This should include information such as the steps to reproduce the defect, the expected result, the actual result, and any other relevant information.
3. *Assign a severity and priority*: Defects need to be classified based on their severity and priority. Severity refers to the impact of the defect on the software or system, while priority refers to the urgency of the defect in terms of fixing it. Defects with higher severity and priority should be addressed first.
4. *Assign the defect to a developer*: The next step is to assign the defect to a developer who is responsible for fixing it. This should be done based on the area of the code that is affected by the defect, as well as the expertise of the developer.

5. *Track the defect*: Defect tracking is important to ensure that defects are addressed in a timely manner. This involves using a tracking tool to monitor the progress of the defect from the time it is reported to the time it is fixed.

6. *Verify the fix*: Once the developer has fixed the defect, it needs to be verified to ensure that it has been completely resolved. This involves testing the software or system again to make sure that the defect has been fixed and that there are no new defects introduced as a result of the fix.

Defect reporting is a collaborative effort between the testing team and the development team. It is important to ensure that defects are reported and addressed in a timely manner to ensure that the software or system is of high quality and meets the requirements of the stakeholders.

The contents of a defect, also known as a *bug report*, should include at least the following information:

- *Summary*: A brief description of the issue or problem encountered, ideally in one sentence. It should be clear enough for the reader to understand what the defect is about just by checking the summary.

- *Steps to reproduce*: Detailed steps or a sequence of actions that can be taken to reproduce the problem, including any specific inputs or data used. The steps should be very descriptive, and it is always a good practice to make sure that even someone with zero knowledge of the product will be able to reproduce the defect by following these steps. It is really common in the industry that the junior developers or the

newcomers in the team start with defect fixing to familiarize themselves with the product and the company processes.

- *Expected behavior*: A description of what the expected outcome or behavior should have been. Usually this is already defined in the acceptance criteria or the requirements. It is the behavior the tester was trying to verify.
- *Actual behavior*: A description of what actually happened when the issue occurred. For example, you were expecting to see a yellow button on the screen and the button you saw is green.
- *Environment information*: The environment where the issue occurred, such as the operating system, hardware, software version, and any other relevant information. It should be as detailed as possible, especially in products with many different versions, because it will help you avoid discussions about how "it works on my machine."
- *Severity*: An assessment of how severe the issue is, using a scale that is relevant to the context of the software. Most of the time it is a scale 1–5 or a list like "Small, Medium, High, Critical."
- *Priority*: An assessment of the importance or urgency of the issue, considering its severity and impact on the user or system. The priority can be set by the tester as an indication of how important they feel it is, but the final priority is set by the business, usually a product owner can change the priorities based on the severity of the defect and the customer needs.

- *Attachments*: Any relevant files or screenshots that can help to reproduce or understand the issue. When multiple steps are required to reproduce the defect, a short video attachment might be handy.
- *Assignee*: The name or team responsible for fixing the issue. This can be replaced by a communication channel where all the defects are reported when found and the teams pick them up.

By including this information in a defect, it becomes easier for the development team to understand and reproduce the problem, which in turn helps to resolve the issue faster and with more accuracy.

Regression Testing

Regression testing is a type of software testing that is performed to ensure that changes made to an application or system have not introduced new defects or caused existing features to fail. It is an important part of the testing process, as it helps to maintain the quality of the software and prevent issues from arising in production.

The primary goal of regression testing is to verify that the software still works as intended after changes have been made. This involves re-testing all or a subset of the existing test cases that have been previously executed. The test cases may cover functional, performance, security, or other aspects of the software.

Regression testing can be executed at different stages of the software development life cycle, depending on the type of change made. For example, if a minor bug is fixed, then only a limited number of test cases may need to be re-executed. However, if major changes are made to the software, then a more extensive regression test suite may need to be developed and executed.

The following are some of the benefits of regression testing:

- Ensures the stability and reliability of the software
- Helps to identify defects and issues that may have been introduced as a result of changes made to the software
- Helps to ensure that new features do not impact existing functionality
- Helps to maintain the quality of the software
- Helps to build confidence in the software among users and stakeholders

Of course, there are also some challenges that may be encountered during regression testing:

- Time and resource constraints may limit the amount of regression testing that can be performed.
- Maintaining and updating the test cases can be time-consuming.
- Test cases may need to be executed multiple times, which can be repetitive and monotonous.
- The need to identify and isolate defects can be challenging.

Automation can be a valuable tool for performing regression testing. Automated tests can be designed to quickly and efficiently execute a set of predefined steps, making it possible to test large and complex systems with relative ease. In addition to reducing the amount of time and effort required for regression testing, automation can also help to ensure consistency and accuracy in testing.

Test Case Status Reporting

Test case status reporting provides stakeholders with critical information about the progress and quality of the project. Test case status reporting involves collecting, analyzing, and presenting the results of the executed test cases.

The purpose of test case status reporting is to keep all stakeholders informed about the testing progress and identify any issues or defects that have been found during testing. The report typically includes information about the number of test cases executed, the number of test cases passed, the number of test cases failed, and the number of defects found.

The test case status report should be concise and easy to understand so that stakeholders can quickly identify any issues and take corrective actions if necessary. The report should also include details about the severity and priority of defects found so that stakeholders can make informed decisions about which defects should be addressed first.

It's important to note that test case status reporting should be done regularly throughout the testing process, not just at the end. This allows stakeholders to identify and address issues early on, which can help to reduce the overall testing time and cost.

The reporting is usually automatically generated by the test management tool, and it contains critical information such as the following:

- *Test case status*: This should include the current status of each test case, such as "Passed," "Failed," or "In Progress." This information provides an overview of how many test cases have been executed and how many remain to be tested.

- *Test case execution details*: This should include information on the actual results of each test case, as well as any issues or defects encountered during testing. This information helps stakeholders understand the quality of the software being tested and identify areas for improvement.
- *Test case coverage*: This should include information on the percentage of test cases executed and the areas of the application covered by testing. This information helps stakeholders understand the scope of testing and identify areas that may require additional testing.
- *Test case prioritization*: This should include information on the priority of each test case, which helps stakeholders understand which test cases are most critical and should be addressed first.
- *Test case trends*: This should include information on the trend of test case results over time, such as the number of passed, failed, or in-progress test cases. This information helps stakeholders understand how the quality of the software is changing over time.
- *Defect summary*: This should include a summary of all defects found during testing, including their severity, priority, and status. This information helps stakeholders understand the overall quality of the software and identify areas that require improvement.

Overall, test case status reporting should be clear, concise, and easy to understand so that stakeholders can make informed decisions about the software being tested. It should be tailored to the specific needs of each project and should be updated regularly to reflect the most up-to-date information.

Test Case Completion

Test case completion refers to the process of ensuring that all test cases identified in the test plan have been executed and analyzed. Once all the test cases have been executed and verified, the testing process can be considered complete.

The following steps can be taken to ensure test case completion:

1. *Ensure all test cases are identified*: All test cases that need to be executed should be identified and documented in the test plan.
2. *Execute all identified test cases*: All test cases should be executed according to the test plan. Test cases can be executed manually or with the help of automated testing tools.
3. *Verify test results*: The results of all executed test cases should be analyzed to determine if the software under test behaves as expected. Any deviation from expected results should be logged as a defect.
4. *Mark test cases as passed or failed*: Test cases should be marked as passed or failed based on their test results. A test case is considered to have passed if it meets all the test criteria specified in the test plan. If it fails to meet any of the test criteria, it is considered to have failed.
5. *Review the test case results*: Test case results should be reviewed to ensure that they are accurate and complete. Any discrepancies should be investigated and resolved.

6. *Update the test case status*: The status of each test case should be updated based on its test results. This information can be used to determine the overall status of the testing process.

7. *Report on test case completion*: A report should be generated to indicate the status of the testing process. This report can be used to communicate the progress of the testing process to stakeholders.

By following these steps, testers can ensure that all identified test cases have been executed and analyzed and the testing process is complete. This information can be used to make decisions on the readiness of the software for release or further testing.

Techniques and Tools Used in Test Execution

The following techniques and tools are used in test execution:

- *Manual testing*: Manual testing is the process of executing tests manually, without the use of any automation tools.
- *Automated testing*: Automated testing is the process of executing tests using automation tools like Selenium, UFT, etc.
- *Defect management tools*: Defect management tools are used to manage the defects that are discovered during the test execution process.
- *Test management tools*: These tools help in managing and executing tests. They can automate the testing process, track defects, and generate reports.

- *Automated testing tools*: These tools can be used to automate the testing process, reducing the time and effort required for manual testing.
- *Performance testing tools*: These tools are used to test the performance of the system under different conditions, such as high traffic or heavy load.
- *Security testing tools*: These tools are used to test the security of the system, checking for vulnerabilities and potential risks.
- *Code coverage tools*: These tools help in measuring how much of the code has been tested. They can identify code that has not been tested, helping to ensure that all code paths have been covered.
- *Pair testing*: This technique involves two testers working together to test the system. This can help identify issues and defects more quickly and efficiently.

Quality Metrics

Quality metrics are used to measure and assess the quality of software products and the software development process. There are various quality metrics that can be used either to monitor the quality of the product or to improve the development and testing processes.

Defect Density

Defect density is a quality metric that measures the number of defects found in a software product per unit of code or function point. It is calculated by dividing the total number of defects by the size of the software product, expressed in lines of code, function points, or other relevant measures.

For example, if a software product has 10,000 lines of code and 50 defects are found, the defect density would be 50/10,000 or 0.005 defects per line of code.

Defect density is an important metric as it provides insight into the quality of the code and the potential for further improvements. A high defect density indicates that there may be significant issues in the code that require attention, such as coding errors, design flaws, or other issues. A low defect density suggests that the code is of high quality and that there may be less need for further improvements.

Test Coverage

Test coverage is a quality metric that measures the percentage of code or functionality that has been tested by the software testing process. It is calculated by dividing the number of lines of code or functions that have been tested by the total number of lines of code or functions in the software product.

For example, if a software product has 10,000 lines of code and the testing process has covered 8,000 lines of code, the test coverage would be 80 percent.

Test coverage is an important metric because it helps to assess the effectiveness of the testing process and identify areas of the software product that require additional testing. A high test coverage suggests that most or all of the code or functionality has been tested, reducing the risk of defects going undetected. A low test coverage, on the other hand, indicates that there may be significant areas of the software product that have not been adequately tested, increasing the risk of defects going undetected.

Code Complexity

Code complexity is a quality metric that measures the complexity of software code. It is often assessed using a software tool that calculates a complexity score based on factors such as the number of control structures, conditional statements, and loops in the code.

One commonly used metric for code complexity is cyclomatic complexity, which measures the number of independent paths through the code. The higher the cyclomatic complexity score, the more complex the code is likely to be.

Code complexity is an important metric because complex code can be difficult to maintain, modify, and test. High levels of code complexity can also increase the risk of defects, as it can be more challenging to identify and fix issues in complex code.

By identifying areas of code that are overly complex, developers and testers can focus on reducing complexity to improve code quality and reduce the risk of defects. Code complexity can also be used as a factor in determining code review and testing priorities, with higher complexity code given greater scrutiny.

Overall, code complexity is an important factor in software quality and should be measured and monitored as part of a comprehensive quality assurance strategy.

Code Maintainability

Code maintainability is a quality metric that measures the ease with which code can be modified, updated, and maintained over time. It is often assessed using software tools that analyze code for factors such as readability, clarity, and modularity.

Code maintainability is an important metric because software code is rarely static and often requires updates and modifications over time. Code that is difficult to maintain can result in higher development costs, longer development cycles, and increased risk of defects.

The following are factors that contribute to code maintainability:

- *Readability*: Code should be easy to read and understand, with clear variable and function names, consistent formatting, and well-structured code blocks.
- *Modularity*: Code should be modular, with well-defined interfaces and clear separation of concerns, making it easier to modify and maintain individual modules.
- *Testability*: Code should be designed to be easily tested, with clear test cases and separation of testing concerns from other aspects of the code.
- *Documentation*: Code should be well-documented, with clear comments and other supporting materials that make it easier to understand and modify.

By measuring code maintainability, developers and testers can identify areas of code that may require additional attention, such as refactoring or redesign, to improve maintainability and reduce the risk of defects over time.

Fault Slip Through

Fault slip through (FST) analysis is a quality metric that measures the effectiveness of the software testing process by tracking defects that are not detected during testing and are found later in the development life cycle or after the software has been released.

FST analysis involves tracking defects that were not detected during the testing process and identifying the reasons why they were missed. The goal of FST analysis is to identify areas of the software product or the testing process that may require improvement and to take corrective action to reduce the number of defects that slip through testing.

FST analysis typically involves the following steps:

1. *Collecting data on defects*: This involves collecting data on defects that are found after the testing process, such as defects reported by users or defects detected during maintenance.
2. *Analyzing the data*: This involves analyzing the data to identify patterns and trends in the types of defects that are slipping through testing, as well as the reasons why they are being missed.
3. *Taking corrective action*: Based on the results of the analysis, corrective action can be taken to improve the testing process, such as improving test coverage, increasing the rigor of testing, or improving the quality of the software development process.

FST analysis is an important metric because it helps to identify areas of the software product or the testing process that may require improvement. By taking corrective action based on FST analysis, the effectiveness of the testing process can be improved, resulting in higher-quality software products and reduced costs associated with defects.

Coding Standards

Coding standards violations are quality metrics that measure the adherence of software code to a set of predefined coding standards or best practices. These coding standards can cover a wide range of factors, including code formatting, variable naming conventions, commenting style, and more.

Coding standards violations can be detected using automated tools that scan code for violations of the predefined standards. These tools can be integrated into the software development process, allowing developers to identify and correct coding standards violations in real time.

Coding standards violations are an important metric because adherence to coding standards can help to improve code quality, maintainability, and readability. Consistent adherence to coding standards can also make it easier for multiple developers to work on the same codebase, as everyone is using a consistent approach.

By measuring coding standards violations, developers and testers can identify areas of the codebase that may require additional attention, such as refactoring or reformatting, to improve adherence to coding standards. Over time, this can help to improve the overall quality of the codebase and reduce the risk of defects.

Overall, coding standards violations are an important factor in software quality and should be measured and monitored as part of a comprehensive quality assurance strategy.

Code Duplication

Code duplication is a quality metric that measures the amount of duplicate or redundant code in a software project. Code duplication occurs when similar or identical code is repeated in multiple places within the codebase.

Code duplication can be measured using automated tools that analyze the codebase and identify sections of code that are repeated or very similar. These tools can also calculate the percentage of code that is duplicated, as well as the locations of the duplicated code.

Code duplication is an important metric because it can lead to a number of negative outcomes, such as the following:

- *Increased development time*: Duplication of code can lead to increased development time as developers spend more time writing and testing similar code.
- *Increased maintenance costs*: Duplicated code can make it more difficult to maintain and update the codebase over time, leading to increased maintenance costs.

- *Increased risk of defects*: Duplication of code can lead to inconsistencies in the codebase, which can increase the risk of defects and other quality issues.

By measuring code duplication, developers and testers can identify areas of the codebase that may require refactoring or reorganization to reduce duplication and improve code quality. Over time, this can help to reduce development and maintenance costs, as well as improve the overall quality and reliability of the software product.

Dead Code

Dead code is a quality metric that measures the amount of code in a software project that is no longer used or executed. Dead code can occur due to changes in requirements, due to changes in design, or simply as a result of refactoring or other code changes.

Dead code can be measured using automated tools that analyze the codebase and identify sections of code that are not being executed. These tools can also calculate the percentage of code that is dead, as well as the locations of the dead code.

Dead code is an important metric because it can lead to a number of negative outcomes, such as the following:

- *Increased complexity*: Dead code can make the codebase more complex and difficult to understand, which can increase the time required for development, maintenance, and testing.
- *Increased resource usage*: Dead code can consume resources, such as memory and processing power, which can reduce the performance and scalability of the software product.

- *Increased risk of defects*: Dead code can make the codebase more difficult to maintain and update over time, which can increase the risk of defects and other quality issues.

Lines of Code

The lines of code (LOC) metric is a software quality metric that measures the size or complexity of a software program by counting the number of lines of code in the source code files. It is a simple metric that is easy to measure and understand, but it has limitations and is often criticized as a measure of software quality.

The LOC metric is calculated by counting the number of lines of code in a source code file, including blank lines and comments. The resulting number represents the size or complexity of the software program.

While the LOC metric can provide an indication of the size and complexity of a software program, it has several limitations.

- *It does not measure the quality of the code*: The LOC metric does not take into account the quality or maintainability of the code, which can have a significant impact on software quality.
- *It can be influenced by coding style*: The LOC metric can be influenced by coding style and formatting, which can vary between programmers and teams.
- *It does not account for reuse*: The LOC metric does not account for code reuse, which can lead to an overestimation of the size and complexity of a software program.

Despite these limitations, the LOC metric is still commonly used as a quick and easy way to measure the size or complexity of a software program. However, it should be used in conjunction with other metrics and techniques to provide a more comprehensive assessment of software quality.

Fan-Out

Fan-out is a software metric used to measure the number of dependencies that a module or function has on other modules or functions. Specifically, it measures the number of other modules or functions that are called or invoked by a given module or function. The more dependencies a module or function has, the higher its fan-out metric.

Fan-out is important because modules or functions with a high fan-out metric can be more difficult to understand, test, and maintain. They can also increase the risk of defects and other quality issues.

To calculate the fan-out metric, one can count the number of external modules or functions called or invoked by a given module or function. Alternatively, tools such as static code analysis tools can automatically calculate fan-out as part of a comprehensive assessment of software quality.

By measuring fan-out, developers and testers can identify modules or functions that may require refactoring or optimization to reduce their dependencies on other modules or functions. This can help to improve the maintainability and quality of the software product over time.

Overall, fan-out is an important metric to consider when assessing software quality, as it can provide valuable insights into the complexity and maintainability of the codebase.

Compiler Warnings

Compiler warnings are a type of software metric that measures the number of warnings generated by a compiler during the compilation process of a software program. Compiler warnings are generated when the compiler detects potential issues in the code, such as unused variables, uninitialized variables, or other potential bugs or defects.

The compiler warnings metric is important because it can provide an indication of the quality and maintainability of the code. A high number of compiler warnings can indicate that the codebase has a high potential for defects or other quality issues and may require further attention from developers and testers.

To calculate the compiler warnings metric, one can simply count the number of warnings generated by the compiler during the compilation process. Alternatively, tools such as static code analysis tools can automatically identify and report compiler warnings as part of a comprehensive assessment of software quality.

By measuring compiler warnings, developers and testers can identify areas of the codebase that may require further attention or refactoring to improve software quality. Over time, this can help to reduce the risk of defects and improve the overall reliability and maintainability of the software product.

Summary

In this chapter, we went through the main steps of the test execution process. We saw the needs for the environment setup and how to report and retest the defects. We saw the report we can generate at any moment during the execution but mainly at the completion stage. Several techniques and tools were also used for the execution, and we explored a variety of quality metrics for our product. Regression testing is a big part of test execution, so the best approach for it is to automate it.

CHAPTER 7

Test Automation

Test automation is the use of specialized software tools to control the execution of tests and compare the actual results with the expected results. The use of automation has become increasingly popular as it can help reduce the time and cost associated with testing while also improving the accuracy and reliability of the test results.

In this chapter, I'll cover the benefits of test automation, along with what are generally considered to be the best test automation tools and frameworks.

Benefits of Test Automation

The following are some of the benefits of test automation:

- *Saves time and cost*: Automation allows for the execution of tests at a much faster rate than manual testing, which saves time and reduces the cost of testing.
- *Improves test coverage*: Automation tools can perform repetitive and time-consuming tests, which can improve the overall test coverage.
- *Increases accuracy*: Automation tools can execute tests with greater accuracy than humans, reducing the chances of errors.

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4_7

- *Reusability*: Test scripts can be reused for different versions of the application, which saves time and effort.
- *Enhances efficiency*: Automation testing can run tests continuously, 24/7, which enhances the efficiency of testing.

However, test automation is not without its challenges. The initial cost of automation can be high, and maintenance can be time-consuming. Therefore, before implementing automation, it is important to consider the following:

- *Cost-benefit analysis*: Determine the cost of automation tools and the benefits it will provide.
- *Technical feasibility*: Ensure that the automation tool is compatible with the technology stack and the application under test.
- *Skillset*: Ensure that the team has the required technical skills to use and maintain the automation tool.
- *Scope*: Determine the scope of automation testing and identify the tests that are good candidates for automation.

There are various types of automation testing tools available in the market, and they can be broadly classified into the following categories.

Record and Playback Tools

Record and playback tools are a type of test automation tool that enables testers to create automated test scripts by recording user interactions with an application or system under test. These tools typically capture user actions such as mouse clicks, keyboard inputs, and data entry, and then they generate a script that can be played back to reproduce the same set of actions automatically.

The basic process of using a record and playback tool involves starting the recording, performing a series of user actions on the application or system being tested, and then stopping the recording. The tool then generates a script based on the recorded actions, which can be modified and enhanced as needed.

Record and playback tools are often used for regression testing, which involves retesting software after changes have been made to ensure that existing functionality has not been affected. They can also be used for smoke testing, which involves running a basic set of tests to quickly verify that the application is working correctly after a new build or release.

However, record and playback tools have limitations. They can generate scripts based only on the actions that were recorded, so they may not be able to handle more complex scenarios or variations in user input. In addition, any changes to the application or system under test can cause the recorded scripts to fail, requiring updates to the scripts.

Scripting Tools

Scripting tools are a type of test automation tool that enable testers to write scripts or code to automate testing tasks. These tools provide a programming interface or scripting language that allows testers to create custom test cases and automate complex scenarios that may not be possible with record and playback tools.

Scripting tools typically require more technical expertise and programming knowledge than record and playback tools. Testers need to write the test scripts themselves, which can involve writing code in languages such as Python, Java, or Ruby, depending on the tool.

The main advantage of scripting tools is their flexibility and extensibility. Testers can create custom scripts to automate specific test scenarios and perform more complex testing tasks, such as database

validation or performance testing. This makes scripting tools ideal for more sophisticated test automation projects that require a higher level of customization and control.

However, scripting tools also have some limitations. They require a significant amount of time and effort to create and maintain the test scripts. Additionally, testers need to have programming skills and expertise to write effective and efficient test scripts.

Overall, scripting tools are a powerful and versatile type of test automation tool that can be used for a wide range of testing scenarios. However, they require a higher level of technical expertise and are best suited for larger, more complex projects that require custom scripting capabilities.

Hybrid Tools

Hybrid tools are a type of test automation tool that combine the benefits of both record and playback and scripting tools. They allow testers to create automated test scripts using a combination of record and playback and script-based techniques.

Hybrid tools typically offer a visual interface for recording user interactions, which can then be edited and enhanced using a scripting language or programming interface. This allows testers to easily create basic test scripts using record and playback functionality, while also providing the flexibility to add custom code or scripts as needed.

The main advantage of hybrid tools is their versatility and ease of use. They provide a user-friendly interface for recording and editing test scripts, while also offering more advanced features for customizing and enhancing scripts as needed. This makes them suitable for a wide range of testing scenarios, from simple regression testing to more complex performance testing or load testing.

However, like scripting tools, hybrid tools do require a certain level of technical expertise and programming skills to fully leverage their capabilities. Additionally, they can be more expensive than record and playback tools and may require more maintenance and updates over time.

Overall, hybrid tools offer a powerful and flexible approach to test automation, combining the ease of use of record and playback tools with the customization and control of scripting tools. They are best suited for projects that require a mix of automated testing techniques and a higher level of customization and control.

Frameworks

Frameworks are a collection of guidelines, standards, and tools that provide a structured approach to designing, developing, and executing automated tests. Test automation frameworks aim to standardize the testing process and ensure consistency in the way tests are created and executed.

A test automation framework typically consists of a set of components, such as libraries, APIs, and utilities, that provide the building blocks for creating automated tests. These components may include functions for interacting with the application or system under test, test data management, reporting and logging, and error handling.

There are several types of test automation frameworks, including the following:

- *Linear scripting framework*: This framework is a basic approach to test automation that involves writing simple scripts that perform a series of predefined actions. It is easy to create and maintain but lacks flexibility and scalability.

- *Modular framework*: This framework breaks down the application or system under test into modules and creates separate scripts for each module. This approach provides greater flexibility and reusability but requires more time and effort to design and implement.
- *Data-driven framework*: This framework separates the test data from the test script, allowing for more efficient management of large volumes of test data. It can be useful for testing scenarios that require a large number of input combinations.
- *Keyword-driven framework*: This framework uses a set of keywords or commands to define test steps, making it easier for testers without programming experience to create and execute tests.
- *Hybrid framework*: This framework combines elements of multiple frameworks, providing greater flexibility and scalability. It is often used for larger, more complex testing projects that require a high level of customization and control.

Overall, test automation frameworks provide a structured and efficient approach to automated testing, enabling testers to create and execute tests more quickly and reliably. The choice of framework will depend on the specific requirements of the testing project, including the complexity of the application or system under test, the size of the testing team, and the level of automation needed.

Automated Testing Tools

Automated tests should be run in conjunction with manual testing to ensure the most comprehensive testing possible. Test automation should be done only after a thorough understanding of the application, its business logic, and its user requirements.

Test automation can significantly improve the efficiency and effectiveness of testing. However, it should be done with a full understanding of its potential benefits and challenges and with careful consideration of the most appropriate automation tools and approaches for a given application.

Selecting the right automation tool is a critical step in the test automation process. There are various factors to consider when choosing an automation tool, including the following:

- *Functionality*: The tool must support the required testing activities, such as test case design, execution, and reporting.
- *Platform compatibility*: The tool should be compatible with the application under test, the development environment, and the operating system.
- *Ease of use*: The tool should have an intuitive and user-friendly interface, and the scripting language should be easy to learn and use.
- *Maintenance*: The tool should be easy to maintain and update, and it should provide support for version control and defect tracking.
- *Cost*: The tool should be affordable and provide good value for money, taking into account the licensing, training, and maintenance costs.

- *Integration*: The tool should integrate with other tools used in the software development life cycle, such as bug tracking and test management tools.
- *Support*: The tool should provide good technical support, including documentation, user forums, and access to customer support.

Here are some commonly used automated testing tools:

Selenium: Selenium is an open-source tool used for automating web browser testing. It supports multiple programming languages such as Java, C#, Python, and Ruby, and it can be used to automate both functional and regression testing.

Appium: Appium is an open-source tool for automating mobile application testing. It supports multiple mobile platforms such as iOS and Android and can be used to automate both functional and UI testing.

TestComplete: TestComplete is a commercial testing tool that supports multiple platforms and technologies such as web, desktop, mobile, and API testing. It provides a record and playback functionality to create automated tests and supports scripting using multiple programming languages.

JMeter: JMeter is an open-source tool used for performance testing, load testing, and functional testing. It is mainly used to test web applications and supports multiple protocols such as HTTP, FTP, JDBC, and SOAP.

Cucumber: Cucumber is a behavior-driven development (BDD) testing tool that supports multiple programming languages such as Java, Ruby, and JavaScript. It uses plain text, called Gherkin, to define test scenarios and supports automation of functional and acceptance testing.

SoapUI: SoapUI is an open-source tool used for testing APIs and web services. It supports multiple protocols such as SOAP, REST, JMS, and AMF, and it provides a graphical user interface to create, manage, and execute automated tests.

These are just a few examples of the many automated testing tools available. When selecting an automated testing tool, it's important to consider the requirements of the project, the technology used, and the expertise of the testing team. Selecting the right automation tool can significantly improve the efficiency and effectiveness of the testing process. It is essential to choose a tool that meets the specific needs of the project and the organization.

Automated Test Scripts

Creating automated test scripts is a key aspect of test automation. The process involves converting manual test cases into automated test scripts, which can then be executed by an automation tool.

To create effective automated test scripts, it's important to have a clear understanding of the test case to be automated, including the inputs and expected outputs. Once the test case is understood, the script can be written in the appropriate programming language for the automation tool being used.

The script should be designed to mimic the actions of a human tester, simulating user interactions with the application under test. This involves interacting with user interfaces, entering data, and clicking buttons or links. The script should also be designed to verify the expected outputs and behavior of the application.

It's important to ensure that the automated test script is maintainable and flexible, as changes to the application may require updates to the script. Additionally, the script should be designed to provide clear and informative results that can be used to identify issues and make informed decisions about the quality of the application.

Maintaining automated test scripts and updating them are important aspects of test automation. Once the automated tests have been created, they need to be maintained to ensure that they remain relevant and effective over time. This involves reviewing and updating the test scripts as needed, as well as regularly testing them to ensure they continue to provide accurate results.

Updating automated test scripts is necessary when changes are made to the system under test or to the test automation framework. When changes are made to the system under test, the automated test scripts may need to be modified to accommodate the changes. This may involve updating the test data or the test steps to reflect the new functionality.

Changes to the test automation framework may also require updates to the test scripts. For example, if a new version of the test automation tool is released, the test scripts may need to be updated to work with the new version. Additionally, if new features or functionality are added to the test automation framework, the test scripts may need to be updated to take advantage of them.

Maintaining and updating automated test scripts requires attention to detail and a thorough understanding of the system under test and the test automation framework. It is important to review and update the test scripts on a regular basis to ensure that they remain effective and continue to provide value to the testing process.

Summary

This chapter was an overview of the tools we can use for test automation, the benefits of having automation, and the drawbacks of it. As I often enjoy saying, test automation will only report defects you already anticipate. It should be used together with manual testing, always having the needs and the priorities of the product in mind. Test automation is a continuous process that requires a lot of effort into the setup of the framework and the creation of the tests. This effort is distributed among the sprints when we are working in an Agile environment.

CHAPTER 8

Testing in Agile Environment

Agile development is an iterative and incremental approach to software development that emphasizes flexibility, collaboration, and continuous improvement. Testing is an integral part of the Agile development process, and testers work closely with developers and other stakeholders to ensure that the software meets the needs of the business and end users.

This chapter covers the following topics related to testing in Agile development:

- Agile testing principles
- Agile testing quadrants
- Test-driven development (TDD)
- Behavior-driven development (BDD)
- Acceptance test-driven development (ATDD)
- Continuous integration/continuous delivery (CI/CD)
- Test automation in Agile development
- Agile testing best practices

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4_8

Agile Testing Principles

In Agile development, testing is integrated into the development process, rather than being a separate phase that comes after development. Agile testing involves testing small, incremental changes to the software as they are developed, rather than waiting until the end of a long development cycle to test the entire system. Testing is often done by the development team themselves, rather than by a separate testing team. Testing in Agile development is designed to help ensure that the software being developed meets the needs of the customer and is of high quality, while also keeping pace with the fast-paced development cycle of Agile projects.

Here are some testing principles in Agile development:

- *Collaborate and communicate*: In an Agile environment, collaboration and communication are key. It is important to have a good working relationship with developers, product owners, and other stakeholders to ensure everyone is on the same page.
- *Test early and often*: In Agile development, testing is integrated throughout the development process, not just at the end. This means testing should start as early as possible and should be done frequently.
- *Automate testing*: Automated testing is essential in Agile development. It helps ensure that testing is done quickly and efficiently and can be repeated easily as changes are made.
- *Use continuous integration and delivery*: CI/CD is a key part of Agile development. It allows for faster feedback and testing of changes and helps ensure that the software is always in a releasable state.

- *Emphasize exploratory testing*: Exploratory testing is an important part of testing in Agile development. It allows testers to explore the software in a more open-ended way and to find issues that may not have been caught by automated or scripted tests.
- *Prioritize testing based on risk*: In an Agile environment, it's important to prioritize testing based on risk. This means focusing on testing the features that are most critical to the business or that have the potential to cause the most harm if they fail.
- *Use metrics to track progress*: Metrics can be used to track the progress of testing in an Agile environment. This can help teams identify areas where they need to improve and can help ensure that testing is on track to meet the goals of the project.
- *Embrace change*: In an Agile environment, change is inevitable. Testers need to be adaptable and able to change their approach as the software evolves. It's important to be flexible and open to new ideas and to be willing to adjust the testing approach as needed.

Agile Testing Quadrants

Agile testing quadrants are a way to categorize different types of tests in Agile development. As shown in Figure 8-1, the quadrants are a 2×2 matrix that places tests into four quadrants based on two factors: business-facing versus technology-facing and support versus critique.

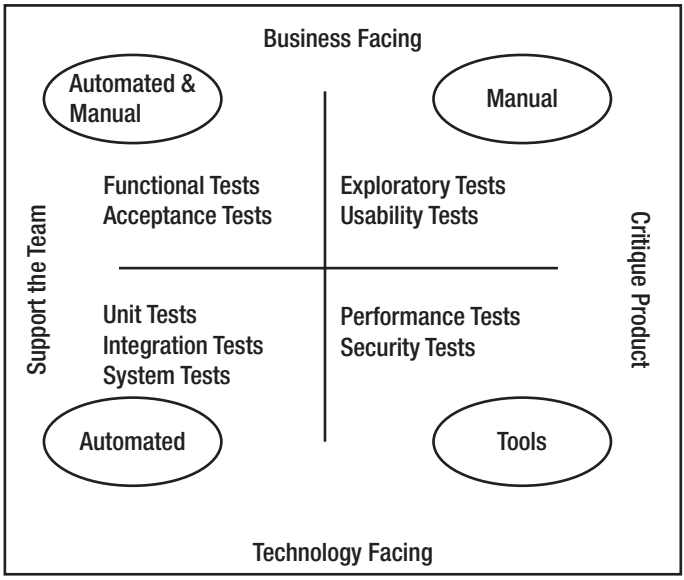

Figure 8-1. *The four quadrants of Agile testing*

The following are the four quadrants (see Figure 8-1):

- *Quadrant Q1*: This quadrant focuses on tests that are technology-driven and are automated. These tests are designed to verify the functionality of the application at a low level. Unit tests, component tests, API tests, and database tests fall under this quadrant.
- *Quadrant Q2*: This quadrant focuses on tests that are business-driven and are automated. These tests are designed to verify the functionality of the application at a high level. Acceptance tests, functional tests, and end-to-end tests fall under this quadrant.
- *Quadrant Q3*: This quadrant focuses on tests that are business-driven and are manual. These tests are designed to verify the application's usability,

accessibility, and user experience. Exploratory testing, usability testing, and accessibility testing fall under this quadrant.

- *Quadrant Q4*: This quadrant focuses on tests that are technology-driven and are manual. These tests are designed to verify the nonfunctional aspects of the application such as performance, security, and reliability. Performance testing, security testing, and reliability testing fall under this quadrant.

Agile teams use the quadrants to plan their testing activities and ensure that they have a balanced approach to testing. By covering all four quadrants, the team can ensure that they are testing the system from different perspectives and reducing the risk of defects slipping through the cracks.

Test-Driven Development

Test-driven development (TDD) is a software development approach where test cases are written before the code is developed. The idea is to create automated test cases for a small piece of functionality and then write just enough code to make those tests pass. Once the code passes the test cases, it is refactored to ensure it is clean and maintainable.

The main objective of TDD is to produce clean code that works as intended and to help ensure that code changes don't introduce new bugs or break existing functionality. TDD also encourages developers to write code in smaller increments, which can help to catch problems earlier in the development process.

TDD is a popular approach in Agile development, as it encourages developers to focus on writing code that meets the requirements and specifications of the user stories. It also helps to identify and address defects early in the development process, leading to faster and more efficient development.

The TDD process involves the following steps:

1. *Write a test case*: Write an automated test case for a specific functionality or feature.
2. *Run the test case*: Run the test case and ensure that it fails, since the functionality is not yet implemented.
3. *Write the code*: Write the code that implements the functionality.
4. *Run the test case*: Run the test case again and ensure that it passes.
5. *Refactor the code*: Refactor the code to ensure it is clean and maintainable.

This process is repeated for each functionality or feature until the entire application is developed. The goal is to create an automated test suite that ensures the application meets the user requirements and specifications, while also being reliable and maintainable.

The following are the benefits of TDD:

- *Better quality code*: Since the code is thoroughly tested at each step, it is more reliable and less prone to errors.
- *Faster development*: TDD can help catch errors early, reducing the amount of time spent debugging and fixing issues.

- *Improved collaboration*: TDD can encourage collaboration between developers and testers, as both parties are involved in writing and executing tests.
- *Greater confidence*: TDD can give developers greater confidence in their code, as they know that it has been thoroughly tested and is less likely to fail in production.

However, TDD also has some challenges, such as the need for a dedicated testing framework and the potential for tests to become overly complex. Additionally, it can be difficult to apply TDD to legacy code or to projects with tight deadlines.

Behavior-Driven Development

Behavior-driven development is an Agile software development technique that promotes collaboration between developers, testers, and business stakeholders. BDD is based on the principles of test-driven development and builds on it by incorporating aspects of domain-driven design (DDD) and object-oriented analysis and design (OOAD).

BDD emphasizes the behavior of a system or feature, rather than its implementation. It encourages the use of natural language in writing tests and specifications, which makes it easier for nontechnical stakeholders to understand the purpose of the feature being developed.

In BDD, the requirements are expressed in the form of scenarios, which describe the behavior of the system from the perspective of a user or stakeholder. These scenarios are written in a specific format called the *given-when-then* (GWT) format, which makes them easy to read and understand.

BDD involves the use of automated tests, such as acceptance tests and integration tests, to ensure that the system or feature behaves as expected. BDD also includes the use of automated tools, such as Cucumber, JBehave,

and SpecFlow, to generate executable tests from the scenarios written in natural language. These tools allow developers and testers to focus on the behavior of the system, rather than the technical details of the implementation.

The BDD process typically involves the following steps:

1. *Define behavior*: The development team and stakeholders collaborate to define the behavior of the system in a structured way using Gherkin.
2. *Automate tests*: Tests are created based on the defined behavior, using tools such as Cucumber, SpecFlow, or Behave to automate the testing process.
3. *Run tests*: Tests are run to verify that the system behaves as expected.
4. *Refactor*: The development team refactors the code to ensure that it is well-designed, readable, and maintainable.

The following are some of the benefits of BDD:

- *Improved collaboration*: BDD encourages collaboration between developers, testers, and stakeholders, ensuring that everyone has a clear understanding of the behavior of the system.
- *Better quality code*: BDD tests the behavior of the system, not just its functionality, resulting in code that is more reliable and less prone to errors.
- *Clear documentation*: The natural language used in BDD provides clear documentation of the behavior of the system, making it easier for everyone involved to understand what the system does.

- *Greater confidence*: BDD gives developers greater confidence in their code, as they know that it has been thoroughly tested and meets the requirements of stakeholders.

Acceptance Test-Driven Development

Acceptance test-driven development is an approach to software development that involves team members from different disciplines, including business stakeholders, developers, and testers. The purpose of ATDD is to ensure that the software being developed meets the business requirements and to promote collaboration between team members.

ATDD involves writing automated acceptance tests before development begins, and these tests serve as a form of requirements documentation. The acceptance tests are based on user stories and describe how the system should behave in different scenarios. The tests are then used to drive the development process, with developers writing code to make the tests pass.

ATDD promotes collaboration between team members by ensuring that everyone is on the same page regarding what the software should do. It also helps to catch defects early in the development process, when they are easier and less expensive to fix. Finally, ATDD can help to ensure that the software meets the business requirements and is of high quality.

The ATDD process typically involves the following steps:

1. *Define acceptance tests*: The development team, stakeholders, and customers collaborate to define acceptance tests that describe the expected behavior of the system from the user's perspective.
2. *Automate acceptance tests*: The acceptance tests are automated using testing tools such as Cucumber, FitNesse, or SpecFlow.

3. *Write code*: The development team writes the code to implement the functionality required by the acceptance tests.
4. *Run acceptance tests*: The acceptance tests are run to verify that the system behaves as expected and meets the requirements defined in the acceptance tests.
5. *Refactor*: The development team refactors the code to ensure that it is well-designed, readable, and maintainable.

The following are the benefits of ATDD:

- *Improved collaboration*: ATDD encourages collaboration between developers, testers, stakeholders, and customers, ensuring that everyone has a clear understanding of the requirements and the behavior of the system.
- *Better quality code*: ATDD tests the system from the perspective of the end user or customer, resulting in code that is more reliable and less prone to errors.
- *Faster feedback*: ATDD provides fast feedback on whether the system meets the requirements and the expectations of the end user or customer, enabling the development team to make changes quickly if needed.
- *Greater confidence*: ATDD gives developers and stakeholders greater confidence in the system, as they know that it has been thoroughly tested and meets the requirements and expectations of the end user or customer.

Continuous Integration and Continuous Delivery

Continuous integration and continuous delivery are software development practices that help teams to deliver software more frequently and with higher quality. CI is a process of regularly integrating code changes into a shared repository, while CD is the process of automatically deploying software changes to production. This is typically achieved through the use of a CI tool such as Bamboo, Jenkins, Travis CI, or CircleCI, which automatically builds and tests code changes whenever they are pushed to a shared code repository.

In CI, developers integrate their code changes to a shared repository multiple times a day, rather than waiting for a big merge at the end of the development cycle. This ensures that issues are detected and fixed earlier, making the process of code integration and deployment more efficient. CI often involves automated build, test, and deployment processes that are triggered by code changes.

CD goes one step further and automates the process of deploying software changes to production. This involves building, testing, and deploying the software changes in a repeatable and reliable manner. CD ensures that software is always in a releasable state, making it easier to release new features and bug fixes quickly and frequently.

Implementing CI/CD requires a combination of tools and practices, such as version control, build automation, automated testing, and deployment automation. It also requires a culture of collaboration and continuous improvement, with a focus on delivering high-quality software to customers.

The following are some of the benefits of CI/CD:

- *Faster feedback*: CI/CD provides fast feedback on code changes, enabling developers to catch and fix issues early in the development process.

- *Improved collaboration*: CI/CD encourages collaboration between developers, testers, and operations teams, ensuring that everyone is working together to deliver high-quality code.
- *Greater efficiency*: CI/CD automates many of the manual tasks involved in building, testing, and deploying code, freeing up developers to focus on other tasks.
- *Better quality code*: CI/CD ensures that code changes are thoroughly tested and integrated with the existing codebase, resulting in code that is more reliable and less prone to errors.
- *Faster time-to-market*: CI/CD enables organizations to deliver code changes quickly and safely, enabling them to respond more quickly to changing market conditions.

However, CI/CD also has some challenges, such as the need for a robust testing infrastructure, the need for good version control practices, and the potential for complex deployments. Additionally, it can be difficult to apply CI/CD to legacy systems or to projects with tight deadlines.

Test Automation in Agile

In Agile development, test automation plays a vital role in delivering high-quality software quickly and efficiently. Here are some ways in which test automation can be used in Agile development:

- *Unit testing*: Test automation can be used to automate unit tests. Unit tests are designed to test the smallest functional components of the software and are written

by developers as part of the coding process. Automated unit tests can be run continuously, providing immediate feedback on any changes that break the code.

- *Functional testing*: Test automation can be used to automate functional tests. Functional tests are designed to test the software from a user's perspective, ensuring that the software meets the specified requirements. These tests can be run continuously, providing immediate feedback on any changes that break the software.
- *Integration testing*: Test automation can be used to automate integration tests. Integration tests are designed to test the interactions between different components of the software. Automated integration tests can be run continuously, providing immediate feedback on any changes that break the integration.
- *Acceptance testing*: Test automation can be used to automate acceptance tests. Acceptance tests are designed to test the software against the specified requirements and ensure that the software meets the user's needs. Automated acceptance tests can be run continuously, providing immediate feedback on any changes that break the software.
- *Regression testing*: Test automation can be used to automate regression tests. Regression tests are designed to test the software after changes have been made to ensure that the changes have not introduced any new bugs. Automated regression tests can be run continuously, providing immediate feedback on any changes that break the software.

In summary, test automation can help teams to deliver high-quality software quickly and efficiently in Agile development by automating various types of tests, such as unit, functional, integration, acceptance, and regression tests.

Agile Testing Best Practices

The following are Agile best practices:

- Involve the testers in the Agile development process from the beginning and throughout the development cycle to ensure that testing is integrated and continuous.
- Use test automation to reduce the time and effort required for regression testing and to facilitate continuous testing.
- Adopt a risk-based approach to prioritize testing efforts on the most critical functionality and high-risk areas.
- Use exploratory testing to identify issues that may not be caught by automated testing or test cases.
- Use testing metrics to monitor the quality of the product and the effectiveness of the testing process.
- Ensure that testing is integrated into the continuous integration and delivery process to ensure that the product is always in a releasable state.
- Foster a culture of collaboration between developers and testers to encourage early defect identification and resolution.

- Encourage continuous feedback and communication among team members, including developers, testers, product owners, and customers.
- Focus on the customer experience and business value to ensure that the product meets the customer's needs and delivers value to the business.
- Continuously improve the testing process and adapt to changing requirements and customer needs.

Summary

The current trend in the industry is to work in an Agile environment, typically Scrum or Kanban, which makes the delivery of the products faster, and then we continuously build on the product. This of course has led to changes in the testing side, because testing has to start sooner and different testing principles are applied to the different testing quadrants. BDD and TDD are increasingly popular, CI/CD is a must have, and testing automation is required in almost every product. Although software testing has been already established for several decades, there are some challenges that we must face on every new or existing project.

CHAPTER 9

Challenges and Solutions in Software Testing

Software testing is an essential part of software development that helps to ensure the quality and reliability of software products. However, the software testing process is often accompanied by various challenges that can affect the effectiveness and efficiency of the testing process. In this chapter, we will discuss some of the common challenges in software testing and provide some solutions to overcome these challenges.

Lack of Clear Requirements

Lack of clear requirements means that the tester does not have a clear understanding of what the software is supposed to do. It could be due to ambiguous or incomplete requirements, a lack of communication between the development team and the testing team, or a lack of understanding of the business domain.

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4_9

Impact of Lack of Clear Requirements on Testing

The lack of clear requirements can have a significant impact on software testing. The following are some of the impacts:

- *Inaccurate testing*: When requirements are unclear, the tester may not have a complete understanding of what the software is supposed to do. This can lead to inaccurate testing, where the tester may test the wrong functionality or miss critical defects.
- *Increased cost*: When the testing team is not clear on what needs to be tested, they may end up testing functionality that is not required, leading to increased testing effort and cost.
- *Delayed testing*: When requirements are unclear, the testing team may need to spend more time clarifying the requirements with the development team, leading to delayed testing.
- *Misaligned expectations*: When the requirements are unclear, the testing team may have different expectations than the development team. This can lead to misunderstandings and disagreements between the two teams.

Mitigating the Impact of Lack of Clear Requirements

To mitigate the impact of lack of clear requirements, the following steps can be taken:

- *Collaboration*: Collaboration between the development and testing teams can help clarify requirements and ensure that everyone has a clear understanding of what needs to be tested.
- *Requirements review*: A requirements review can help identify any ambiguities or incomplete requirements, allowing them to be clarified before testing begins.
- *Traceability*: Traceability between requirements and test cases can help ensure that all requirements are tested and that the testing effort is focused on the required functionality.
- *Domain knowledge*: Ensuring that the testing team has a clear understanding of the business domain can help them better understand the requirements and improve the accuracy of testing.

Time Constraints

A time constraint is a common challenge in software testing, as testing often has to be completed within a limited time frame. This can be due to project deadlines, budget constraints, or other factors. In this section, we will discuss the impact of time constraints on software testing and strategies for managing time constraints in testing.

Impact of Time Constraints on Software Testing

The following are some of the impacts of time constraints on software testing:

- *Inadequate testing*: Time constraints can lead to inadequate testing, as testers may not have enough time to test all the required functionality thoroughly. This can lead to missed defects and reduced software quality.

- *Increased risk*: Inadequate testing increases the risk of defects being discovered in the production environment, which can be costly to fix and can damage the reputation of the development team.
- *Increased pressure*: Time constraints can create pressure on the testing team, leading to increased stress and fatigue, which can impact the quality of testing.
- *Delayed release*: Time constraints can lead to delays in the release of the software, as testing may need to be extended or additional resources may need to be brought in to complete testing.

Strategies for Managing Time Constraints in Testing

The following are some strategies that can be used to manage time constraints in testing:

- *Prioritization*: Prioritization of testing based on criticality and risk can help ensure that the most important functionality is tested first.
- *Test automation*: Test automation can help reduce testing time by automating repetitive tasks and allowing testers to focus on more complex testing scenarios.
- *Agile testing*: Agile testing methodologies, such as Scrum and Kanban, can help manage time constraints by breaking testing into smaller, manageable iterations and providing frequent feedback.

- *Collaboration*: Collaboration between the development and testing teams can help identify and address issues early, reducing the need for rework and saving time.
- *Continuous integration and delivery*: Continuous integration and delivery can help reduce testing time by automating the build and deployment process and providing frequent feedback.

Lack of Skilled Resources

Lack of skilled resources is a common challenge that organizations face in software testing. This can be due to a shortage of skilled testers, a lack of training and development programs, or a lack of experienced personnel. In this section, we will discuss the impact of lack of skilled resources on software testing and strategies for managing this challenge.

Impact of Lack of Skilled Resources on Software Testing

The following are some of the impacts of lack of skilled resources on software testing:

- *Inadequate testing*: Lack of skilled resources can lead to inadequate testing, as testers may not have the necessary skills and knowledge to test the software thoroughly.
- *Increased cost*: Inadequate testing can lead to increased costs, as defects may be discovered later in the development process, which can be costly to fix.

- *Decreased productivity*: Lack of skilled resources can lead to decreased productivity, as testers may take longer to complete testing tasks due to a lack of experience or knowledge.
- *Reduced quality*: Inadequate testing and decreased productivity can lead to reduced software quality, which can impact customer satisfaction and reputation.

Strategies for Managing Lack of Skilled Resources in Testing

The following are some strategies that can be used to manage the lack of skilled resources in testing:

- *Training and development*: Providing training and development programs for testers can help them acquire the necessary skills and knowledge to test software effectively.
- *Knowledge sharing*: Encouraging knowledge sharing among testers can help improve the overall skill level of the testing team.
- *Hiring skilled testers*: Hiring skilled testers can help address the shortage of skilled resources, but this can be a challenging task due to the high demand for skilled testers.
- *Test automation*: Test automation can help reduce the reliance on skilled resources by automating repetitive testing tasks and allowing testers to focus on more complex testing scenarios.

Automation Challenges

Automation can bring many benefits to organizations, including increased efficiency, accuracy, and cost savings. However, there are also several challenges that come with automation. In this section, we will discuss some of the challenges you might face in automation and how to address them.

Common Automation Challenges

The following are some of the common challenges that organizations face when implementing test automation:

- *Test coverage*: Automated tests can cover only the scenarios that have been explicitly programmed, and this can lead to incomplete testing coverage. Human testers can improvise and test unexpected scenarios, while automated tests cannot.
- *Maintenance*: Test automation requires maintenance, as changes in the application being tested or the test environment can cause the automated tests to fail. It can be a challenge to maintain automated tests, especially if the tests are not designed to be maintainable.
- *Integration*: Integration between test automation tools and other tools in the development and testing process can be a challenge. It requires additional effort to integrate test automation with tools such as defect tracking, test management, and continuous integration.
- *Cost*: Test automation can be expensive to implement and maintain, especially if the organization lacks the necessary expertise.

- *Expertise*: Test automation requires a high level of expertise in programming and automation tools, which can be a challenge for organizations that lack this expertise.

Strategies for Addressing Automation Challenges

The following are some strategies for addressing the common challenges of test automation:

- *Test coverage*: To address the challenge of incomplete test coverage, it is essential to ensure that the automated tests cover all critical scenarios. The automation team should work with the manual testing team to identify the critical scenarios and ensure that they are automated.
- *Maintenance*: To address the challenge of test maintenance, it is essential to design the tests to be maintainable. The automation team should use good coding practices and create automated tests that are modular and reusable.
- *Integration*: To address the challenge of integration, it is essential to select automation tools that integrate well with other tools in the development and testing process. The automation team should also ensure that the automated tests can be easily integrated with other tools.
- *Cost*: To address the challenge of cost, it is essential to select automation tools that are cost-effective and to ensure that the automation team has the necessary expertise to implement and maintain the tests.

- *Expertise*: To address the challenge of expertise, it is essential to invest in training and development programs to ensure that the automation team has the necessary skills and knowledge to implement and maintain the automated tests.

Communication and Collaboration

Effective communication and collaboration among team members are critical for successful software testing. However, it can be challenging to maintain clear and open communication channels, especially in large and distributed teams.

Effective communication is crucial in testing, but poor communication channels can hinder the process. For example, if testers are not able to communicate their findings effectively, developers may not be able to understand the issues and make the necessary changes. Testers must use effective communication channels, such as emails, instant messaging, or video conferencing, to ensure that all stakeholders are kept in the loop.

Change Management

Change management is an important aspect of software testing, as changes to software can impact its functionality, performance, and security. However, there are several challenges that testers may face when managing changes in a software testing environment. In this section, we will discuss some of the common change management challenges in testing and how to overcome them.

- *Lack of visibility*: One of the biggest challenges in change management is the lack of visibility into the changes that are being made to the software. Testers may not have access to the latest version of the software or may not

be informed about changes that have been made. This can lead to issues during testing, as testers may not be able to test the software adequately. To overcome this challenge, testers must communicate effectively with developers and other stakeholders to ensure that they are informed about changes to the software.

- *Inadequate testing*: Another challenge in change management is inadequate testing. Testers may not have the time or resources to test all changes thoroughly, which can lead to issues with the software. To overcome this challenge, testers must prioritize testing based on the impact of the changes on the software. They must also collaborate closely with developers to understand the changes and determine the appropriate testing strategy.
- *Inconsistent testing environments*: Testing environments must be consistent to ensure accurate and reliable results. However, this can be a challenge in change management, as different changes may require different testing environments. Testers must ensure that the testing environments are consistent across different changes and that the testing environment is appropriate for the specific change being made.
- *Resistance to change*: Resistance to change can also be a challenge in change management. Stakeholders may resist changes due to concerns about the impact on the software or the testing process. Testers must collaborate closely with stakeholders to understand their concerns and address them effectively. They must also communicate the benefits of the changes to stakeholders to gain their support.

Testing Across Platforms

Testing software across different platforms is crucial to ensure that it works seamlessly on various devices and operating systems. However, testing across platforms can be challenging, as each platform may have its unique features and requirements.

- *Platform compatibility*: One of the main challenges in testing across platforms is ensuring compatibility across different platforms. Each platform may have its specific hardware, operating system, and software configurations, which can impact the behavior of the software. Testers must ensure that the software works seamlessly across all platforms by testing it on each platform and verifying its compatibility.
- *Version compatibility*: Another challenge in testing across platforms is version compatibility. Different versions of the same platform may have different features, requirements, and configurations. Testers must ensure that the software works seamlessly across all versions of each platform by testing it on each version and verifying its compatibility.
- *User interface (UI) differences*: Each platform may have its unique user interface, which can impact the behavior of the software. Testers must ensure that the software works seamlessly on each platform's UI by testing it on each platform and verifying its functionality and appearance.
- *Performance differences*: Each platform may have different performance capabilities, such as processor speed, memory, and storage. These differences can

impact the software's performance on each platform. Testers must test the software on each platform and verify its performance and responsiveness.

- *Resource limitations*: Different platforms may have different resource limitations, such as memory, storage, and processing power. These limitations can impact the software's performance and functionality on each platform. Testers must ensure that the software works seamlessly on each platform by testing it under different resource limitations.
- *Testing tools and frameworks*: Testing tools and frameworks may not be available or may differ across different platforms. Testers must use appropriate testing tools and frameworks that are available on each platform to ensure accurate and reliable testing results.

Future of Software Testing

As technology advances and the digital landscape evolves, the future of software testing is constantly changing. Here are some potential developments that could shape the future of software testing:

- *Increased automation*: Automation is already a significant part of software testing, but it will likely become even more prevalent in the future. With the help of artificial intelligence and machine learning, software testing could become even more efficient and effective.
- *Shift-left testing*: In shift-left testing, testing is integrated earlier in the software development process, and tests are performed more frequently. This approach can

help identify and fix issues earlier in the development cycle, reducing the overall cost and time associated with testing.

- *DevOps and continuous testing*: DevOps has changed the way software is developed and delivered, and continuous testing is becoming more important to ensure that the software being delivered is high-quality and meets user requirements.
- *Mobile and IoT testing*: As more devices become connected to the Internet and more people rely on mobile devices, testing for mobile and Internet of Things (IoT) devices will become more important. Testing will need to cover not only the software but also the hardware and connectivity aspects of these devices.
- *Blockchain testing*: As blockchain technology continues to gain popularity, testing will become increasingly important to ensure the integrity and security of blockchain-based applications.
- *Cloud-based testing*: As more companies move their IT infrastructure to the cloud, testing will need to be adapted to accommodate cloud-based environments.
- *Increased focus on security testing*: With the increasing number of cyberattacks and data breaches, security testing will become even more important in the future. This will involve testing not only the software itself but also the security of the underlying infrastructure and data storage systems.

- *Virtualization and simulation testing*: Virtualization and simulation testing can help replicate real-world scenarios in a controlled environment. This can help ensure that software behaves as expected in a variety of different scenarios and environments.
- *AI-driven testing*: Artificial intelligence and machine learning can help to automatically generate test cases and optimize testing processes, making testing more efficient and effective.
- *Collaborative testing*: In the future, software testing will become more collaborative, with testers, developers, and other stakeholders working together to ensure high-quality software is delivered.

Overall, the future of software testing will be shaped by advancements in technology and a growing need for high-quality, secure, and reliable software. As such, the software testing industry will need to continue to evolve and adapt to meet changing demands and expectations.

Risk-Based Testing

Risk-based testing is an approach to software testing that prioritizes testing efforts based on the level of risk associated with the software application or system being tested. This approach involves identifying potential risks to the system and prioritizing testing efforts on the areas that pose the greatest risk to the project, rather than testing everything equally.

It is common that during the development of a project there is not enough time or resources to mitigate all the potential risks. By applying risk-based testing, the tester can ensure that the most critical foreseeable risks are mitigated and the product can go live with minimum risk.

The most common way to apply risk-based testing is to create a list with all the possible risks the team can identify and decide the probability and the impact of every risk. It does not have to be very accurate, a simple "low" and "high" separation should be enough.

Table 9-1 shows what this matrix would then look like.

Table 9-1. *Risk Matrix*

Risk	Probability	Impact
Risk 1	LOW	HIGH
Risk 2	LOW	LOW
Risk 3	HIGH	LOW
Risk 4	HIGH	HIGH
Risk 5	HIGH	HIGH
…		
Risk N	LOW	HIGH

The risks with a high probability and high impact must be tested first, and they must be mitigated either by eliminating the impact or by minimizing their probability.

Next, the risks with a low probability and high impact should be tested to make sure that the probability is indeed low, and then the team should have a mitigation plan about the impact.

After that, if there is still time, the risks with a high probability and a low impact could be tested to verify that the impact is indeed low, and then the team could reduce the probability of the risk.

Finally, the risks with low probability and low impact, always depending on the time left and on the business, can be ignored, since they are not likely to happen, and even if they happen, the impact might be minimal.

Figure 9-1 shows a graph to better visualize the connection between the impact and the probability of the risks.

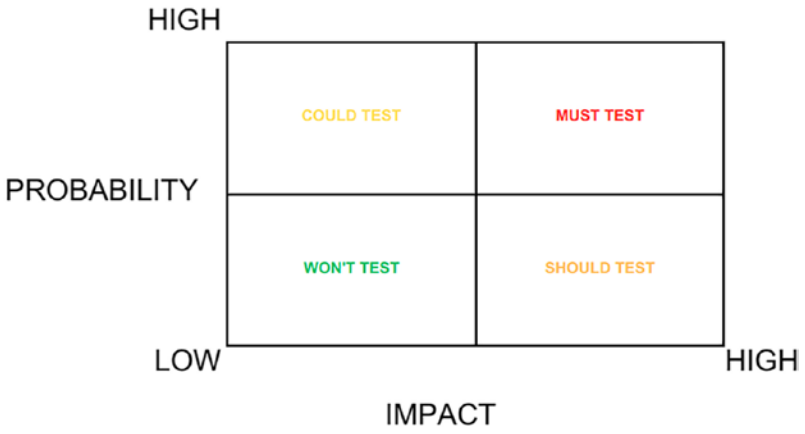

Figure 9-1. *Risk graph*

Summary

There are several challenges in the software world, and testing is no exception. Although every challenge is manageable, we should be proactive and mitigate all the critical risks in time; otherwise, it is certain that we will have to pay a higher price later. A structured way to do this is to create a risk assessment session at the start of every project and revisit risk every now and then to keep track of all the risks and their impacts.

Afterword

In conclusion, software testing is an essential process that ensures software products are of high quality and meet the requirements of the end users. It involves different activities such as test planning, test design, test execution, and test reporting, among others.

There are different types of software testing, including functional testing, performance testing, security testing, and user acceptance testing, among others. Each type of testing is critical and helps to identify defects or issues that could impact the quality of the software product.

In recent years, agile development has become a popular approach to software development, and testing is an essential component of this approach. Agile development allows teams to work collaboratively, respond to changes quickly, and deliver high-quality software products.

Continuous testing is also becoming increasingly important, especially with the rise of DevOps and the need to deliver software products faster. Continuous testing involves integrating testing into the software development process, which helps to identify issues earlier and reduce the overall cost of fixing defects.

In the future, software testing is expected to become more automated, with a focus on artificial intelligence and machine learning. This will help to improve the accuracy and efficiency of testing, reduce the time taken to perform testing, and improve the overall quality of software products.

Automated testing is an emerging trend in software testing that can help to speed up the testing process and improve the overall efficiency of the testing process. However, automation alone cannot guarantee a successful software release, and it is necessary to combine automated testing with manual testing to achieve the best results.

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4

The main message I want you to take from this book is that these are guidelines and best practices and in no way are they absolute truths about software testing. There are multiple ways to perform software testing, and the decisions you will make during a project will be different every time. The tools and the guidelines are there to help you and assist you in this journey, but at the end of the day, the human factor, your intuition, will be the most important asset you bring to the table. Quality assurance starts and ends with every person in an organization.

There is always room for improvement and learning. Staying up-to-date with the latest technologies, tools, and best practices can help you become a better software tester and contribute to the development of high-quality software. Remember that testing is a collaborative effort that involves communication, teamwork, and a commitment to quality. So, keep learning and exploring the exciting and ever-changing world of software testing!

Index

A

B

C

P. Leloudas, *Introduction to Software Testing*,
https://doi.org/10.1007/978-1-4842-9514-4

D

E

F

G, H

I, J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z