UNIVERSITY OF ZIMBABWE

FACULTY OF COMPUTER ENGINEERING INFORMATICS AND COMMUNICATIONS

COMPUTER ENGINEERING DEPARTMENT

BACHELOR OF SCIENCE HONOURS COMPUTER ENGINEERING

SOFTWARE TESTING AND QUALITY ASSURANCE PART 4- HSE 401

PERIOD: 2023/4

Lecturer: Mrs P Mawire

Email ID: portieville@gmail.com

Contact No: 0789728007

Preamble

The course will cover topics such as testing levels (unit, integration, system, acceptance), testing types (functional, non-functional, regression, performance, etc.), test design techniques (black-box, white-box, use case-based), test execution and defect management, test automation, performance testing, security testing, quality assurance processes, test management and documentation, test metrics and reporting, emerging trends in software testing (Agile, DevOps, cloud-based testing, etc.), and case studies/practical applications.

Course Overview

This course provides an in-depth understanding of software testing principles, techniques, and best practices, as well as the fundamentals of quality assurance in software development. Students will learn how to plan, design, and execute effective software tests, identify and report defects, and implement quality assurance processes to ensure the delivery of high-quality software products.

Course Objectives

Upon completion of the course students should be able to:

- Apply the fundamentals of software testing.
- Apply various testing techniques.
- Master test planning, documentation, and defect tracking.
- Gain hands-on experience in software testing through practical exercises.
- Learn about test automation tools and frameworks.
- Understand common challenges and best practices in software testing.
- Develop critical thinking and problem-solving skills.
- Enhance collaboration and communication skills for effective testing.

DAY 1 LECTURE NOTES

Introduction to Software Testing and Quality Assurance

- Overview of software testing and quality assurance

Software testing and quality assurance (QA) are essential components of the software development life cycle (SDLC) aimed at ensuring the delivery of high-quality software products. Software testing involves the verification and validation of software applications to identify defects, errors, or gaps between expected and actual results. The primary goal of testing is to ensure that the software meets the specified requirements, functions as intended, and is free from defects before it is released to end-users.

Since quality is the prime goal of testing and it is necessary to meet the defined quality standards, software testing should be defined keeping in view the quality assurance terms. Here, it should not be misunderstood that the testing team is responsible for quality assurance. But the testing team must be well aware of the quality goals of the software so that they work towards achieving them. Moreover, testers these days are aware of the definition that testing is to find more and more bugs. But the problem is that there are too many bugs to fi x. Therefore, the recent emphasis is on categorizing the more important bugs first. Thus, software testing can be defi ned as, Software testing is a process that detects important bugs with the objective of having better quality software.

Key aspects of software testing are:

Test Planning: Test planning involves defining the testing objectives, scope, and strategies, as well as identifying the resources, test environments, and test data required for testing.

Test Design: Test design involves creating test cases, test scenarios, and test scripts based on the requirements and design specifications of the software. Test cases outline the input data, expected outcomes, and steps to be executed during testing.

Test Execution: Test execution involves running the test cases and recording the actual results. Testers compare the actual results with the expected results to identify any discrepancies or defects.

Defect Management: Defect management includes capturing, reporting, tracking, and resolving software defects. Defects are logged into a defect tracking system, and the development team works on fixing them.

Types of Testing: Various types of testing are performed during the SDLC, including unit testing, integration testing, system testing, acceptance testing, performance testing, security testing, and more. Each type of testing aims to address specific aspects of software quality.

Test Automation: Test automation involves the use of tools and frameworks to automate the execution of test cases. It helps improve testing efficiency, repeatability, and coverage.

Quality Assurance is a broader discipline encompassing the entire software development process. It focuses on preventing defects and ensuring that the development process follows defined standards and best practices.

Key aspects of QA include:

Process Definition: QA involves defining and establishing effective development processes and standards. This includes defining coding guidelines, documentation standards, and establishing quality metrics.

Risk Management: QA teams identify potential risks and develop strategies to mitigate them. Risk analysis and mitigation techniques are employed to minimize the impact of risks on software quality.

Requirement Analysis: QA teams collaborate with stakeholders to understand and analyze requirements, ensuring they are clear, complete, and testable.

Code Reviews: QA teams perform code reviews to identify coding errors, adherence to coding standards, and to ensure code quality and maintainability.

Continuous Integration and Deployment: QA plays a crucial role in implementing continuous integration and continuous deployment practices. This includes automating build processes, version control, and ensuring the integrity of the software throughout its lifecycle.

Quality Metrics and Reporting: QA teams define quality metrics to measure the effectiveness of the development process and the quality of the software. They generate reports and provide insights to stakeholders about the status of quality and areas for improvement.

Quality Assurance: All those planned and systematic actions necessary to provide adequate confidence that a product or service will satisfy given requirements for quality. Testers and managers need to be sure that all activities of the test effort are adequate and properly executed. The body of knowledge, or set of methods and practices used to accomplish these goals, is quality assurance. Quality assurance is responsible for ensuring the quality of the product. Software testing is one of the tools used to ascertain the quality of software.

To improve a quality process, you need to examine your technology environment (hardware, networks, protocols, standards) and your market, and develop definitions for quality that suit them. First of all, quality is only achieved when you have balance-that is, the right proportions of the correct ingredients. Note Quality is getting the right balance between timeliness, price, features, reliability, and support to achieve customer satisfaction.

Traditional quality assurance principles are not a good fit for today's software projects. Further, the traditional processes by which we ensure quality in software systems is cumbersome and inflexible. Even more important, the traditional tools used by quality assurance are not able to keep up with the pace of software development today. Testers constrained to follow these outmoded practices using these cumbersome tools are doomed to failure. The quality process must be reinvented to fit the real needs of the development process. The process by which we ensure quality in a product must be improved. A company needs to write its own quality goals and create a process for ensuring that they are met. The process needs to be flexible, and it needs to take

advantage of the tools that exist today. Several new technologies exist that can be used to support quality assurance principles, such as collaboration, which allows all involved parties to contribute and communicate as the design and implementation evolve. These technologies can also help make quality assurance faster and more efficient by replacing traditional paper documentation, distribution, and review processes with instantly available single-source electronic documents, and by supplementing written descriptions with

drawings. Replacing tradition requires a culture change. And, people must change their way of working to include new tools. Changing a culture is difficult. Historically, successful

cultures simply absorb new invading cultures, adopt the new ideas that work, and get on with life. Cultures that resist the invasion must spend resources to do so, and so become distracted from the main business at hand.

Software quality is a combination of reliability, timeliness to market, price/cost, and the feature richness. The test effort must exist in balance with these other factors.

Testers need tools-that is, methods and metrics that can keep up with development-and testers need the knowledge to use those tools. Traditionally, testing has been a tool to measure the quality of the product. Today, testing needs to be able to do more than just measure; it needs to be able to add to the value of the product.

Importance of testing in the software development life cycle

Testing plays a crucial role throughout the software development life cycle (SDLC) and holds significant importance. Here are some key reasons why testing is important in the SDLC:

Error Detection: Testing helps in identifying defects, errors, or deviations from expected behavior in the software. By detecting and addressing these issues early in the development process, testing helps prevent more significant problems and reduces the risk of releasing faulty software to end-users.

Quality Assurance: Testing ensures that the software meets the specified requirements and functions as intended. It helps in verifying that the software performs its intended tasks correctly, providing a level of confidence in its quality.

Customer Satisfaction: Thorough testing helps in delivering high-quality software to end-users. By identifying and fixing bugs, usability issues, and performance bottlenecks, testing contributes to a positive user experience, enhancing customer satisfaction.

Cost Reduction: Detecting and fixing defects in the early stages of development is significantly less expensive than addressing them later in the SDLC or after the software is released. Testing helps in reducing the overall cost of software development by minimizing rework, maintenance, and support costs.

Risk Mitigation: Testing helps in identifying and mitigating risks associated with software development. It ensures that critical functionalities work as expected, security vulnerabilities are addressed, and the software is compatible with different environments and configurations.

Compliance and Standards: Testing ensures that the software complies with industry standards, regulations, and guidelines. It helps in meeting legal and regulatory requirements, ensuring data privacy and security, and maintaining the reputation and trustworthiness of the software and the organization.

Continuous Improvement: Testing provides valuable feedback and insights into the software development process. It helps in identifying areas for improvement, optimizing performance, enhancing usability, and refining the software based on user feedback and changing requirements.

Reliability and Stability: Thorough testing helps in building reliable and stable software. It increases confidence in the software's stability, robustness, and ability to handle real-world scenarios and user interactions.

Conclusion, testing is essential in the SDLC as it ensures software quality, reduces risks, enhances customer satisfaction, and contributes to cost-effective development. It is an

integral part of the software development process, enabling organizations to deliver reliable, high-performing, and user-friendly software products.

Role of quality assurance in software development

Quality Assurance (QA) plays a crucial role in software development by ensuring that software products meet the required standards of quality, reliability, and usability. The primary goal of QA is to identify and rectify defects, bugs, and functional issues in the software before it is released to end-users. Here are some key aspects of the role of Quality Assurance in software development:

Defect Prevention: QA teams work proactively to prevent defects from occurring in the software development process. This involves establishing and enforcing coding standards, conducting code reviews, promoting best practices, and implementing quality control measures at every stage of development.

Requirements Analysis: QA professionals collaborate with stakeholders, including clients, business analysts, and developers, to analyze and understand the software requirements. By gaining a comprehensive understanding of the expected functionality and user experience, QA helps ensure that the software is built to meet these requirements.

QA professionals review the software requirements to ensure they are clear, complete, and testable. They collaborate with stakeholders to identify any ambiguities, inconsistencies, or gaps in the requirements, helping to ensure that the development team has a clear understanding of what needs to be built.

Test Planning and Execution: QA teams create a test strategy and test plans to systematically verify and validate the software. They design and execute various types of tests, such as functional tests, performance tests, security tests, and usability tests, to identify defects and ensure that the software functions as intended.

QA teams develop a comprehensive test plan that outlines the testing approach, test objectives, test scope, and test deliverables. The plan also includes the identification of test environments, test data requirements, and the selection of appropriate testing techniques and tools.

Defect Identification and Reporting: QA professionals meticulously track and document defects and issues discovered during testing. They use bug tracking systems to log defects, assign them to developers, and monitor their resolution. By providing detailed reports, QA helps developers understand the nature of the defects and fix them effectively. QA teams use defect tracking systems to log, manage, and track defects found during testing. They assign defects to developers, track their status, and verify fixes to ensure that issues are resolved effectively.

Process Improvement: QA teams continuously assess and improve the software development process. They identify bottlenecks, suggest process enhancements, and introduce quality control measures to enhance the overall efficiency and effectiveness of the development lifecycle. QA conducts process audits and reviews to assess adherence to established development processes, coding standards, and quality control measures. They identify areas for improvement and provide recommendations to enhance the overall development process.

Validation and Verification: QA ensures that the software meets the specified requirements and standards by validating its functionality, performance, security, and usability. They perform rigorous testing, including regression testing and user acceptance testing, to ensure that the software performs reliably in different environments and scenarios.

Customer Satisfaction: QA plays a crucial role in ensuring customer satisfaction by focusing on the quality of the software. By identifying and resolving defects before the software reaches the end-users, QA helps deliver products that meet user expectations, function correctly, and provide a positive user experience.

Overall, the role of Quality Assurance in software development is to minimize the risks associated with building software and to ensure that the final product meets the required

quality standards, resulting in reliable, high-quality software that satisfies the needs of end-users and stakeholders.

Model for software testing

Testing is not an intuitive activity, rather it should be learnt as a process. Therefore, testing should be performed in a planned way. For the planned execution of a testing process, we need to consider every element and every aspect related to software testing. Thus, in the testing model, we consider the related elements and team members involved. The software is basically a part of a system for which it is being developed. Systems consist of hardware and software to make the product run. The developer develops the software in the prescribed system environment considering the testability of the software. Testability is a major issue for the developer while developing the software, as a badly written software may be difficult to test. Testers are supposed to get on with their tasks as soon as the requirements are specified. Testers work on the basis of a bug model which classifies the bugs based on the criticality or the SDLC phase in which the testing is to be performed. Based on the software type and the bug model, testers decide a testing methodology which guides how the testing will be performed. With suitable testing techniques decided in the testing methodology, testing is performed on the software with a particular goal. If the testing results are in line with the desired goals, then the testing is successful; otherwise, the software or the bug model or the testing methodology has to be modified so that the desired results are achieved.

Software Testing as a process (Write more notes) Read Principles and Practice of Software Testing: Insights into Testing By Vijay John, Harika Done from Page 31

A misconception has prevailed through the evolution of software testing that complete testing is possible, but it is not true. Here, it has been demonstrated that complete testing is not possible. Thus, the term 'effective software testing' is becoming more popular as compared to 'exhaustive' or 'complete testing'.

Software testing has evolved through many phases, namely (i) debugging-oriented phase, (ii) demonstration-oriented phase, (iii) destruction-oriented phase, (iv) evaluation oriented phase, (v) prevention-oriented phase, and (vi) process-oriented phase. There is another classification for evolution of software testing, namely Software testing 1.0, Software testing 2.0, and Software testing 3.0.

Software testing goals can be partitioned into following categories:

1. Immediate goals
2. Long-term goals
3. Post-implementation goals

Software testing is a process that detects important bugs with the objective of having better quality software. Exhaustive testing is not possible due to the following reasons:

- It is not possible to test every possible input, as the input domain is too large.
- There are too many possible paths through the program to test.
- It is difficult to locate every design error.

Effective software testing, instead of complete or exhaustive testing, is adopted such that critical test cases are covered first. There are different views on how to perform testing which have been categorized as schools of software testing, namely (i) analytical school, (ii) standard school, (iii) quality school, (iv) context school, and (v) agile school. Software testing is a complete process like software development.

WORK TO DO

1. READ ON MODELS FOR TESTING, APPLY YOUR MODEL OF CHOICE IN A CASE STUDY.
2. REFER TO GIVEN CASE STUDIES AND DO IT IN GROUPS

Effective software testing Vs. Exhaustive testing

Effective software testing focuses on achieving sufficient test coverage and ensuring the most critical aspects of the software are thoroughly tested. Instead of aiming for complete or exhaustive testing, which is often impractical or time-consuming, the goal is to strike a balance between risk management and resource utilization. Here are some principles of effective software testing:

Risk-based Approach: Prioritize testing efforts based on the potential impact of failures and the likelihood of occurrence. Focus on high-risk areas, critical functionalities, and components that are prone to defects.

Requirements-based Testing: Align testing activities with the software's requirements. Thoroughly understand the requirements and design test cases that address each requirement. This ensures that the software functions as intended.

Test Case Design Techniques: Utilize various test case design techniques such as equivalence partitioning, boundary value analysis, and error guessing. These techniques help identify representative test cases that cover different scenarios and are likely to uncover defects.

Adopt Test Automation: Leverage test automation tools and frameworks to automate repetitive and time-consuming test cases. Automation improves efficiency, allows for faster regression testing, and enables broader test coverage.

Test Prioritization: Prioritize test cases based on criticality, impact, and business value. Start with tests that cover core functionality and critical paths. This helps identify major issues early in the testing process.

Defect Tracking and Management: Implement a robust defect tracking and management process. Thoroughly document, track, and prioritize defects to ensure they are addressed in a timely manner. Collaborate with developers to resolve issues effectively.

Continuous Testing: Integrate testing throughout the software development lifecycle, rather than treating it as a separate phase. Conduct early and frequent testing to identify issues early on and facilitate faster feedback loops.

Exploratory Testing: Supplement scripted testing with exploratory testing. Allow testers to explore the software freely, uncovering defects that might be missed by predefined test cases. Encourage creativity and critical thinking during exploratory testing.

Test Environment and Data Management: Set up and maintain test environments that closely resemble the production environment. Ensure test data is representative of real-world scenarios, covering various inputs, configurations, and edge cases.

Collaboration and Communication: Foster effective collaboration between developers, testers, and other stakeholders. Encourage open communication, sharing of knowledge, and feedback exchange to improve the overall quality of the software.

By following these principles, organizations can achieve effective software testing with a focus on risk mitigation, thorough coverage of critical areas, and efficient resource utilization.

Software Testing Fundamentals

Testing objectives and goals

The objectives and goals of software testing can vary depending on the specific project and organizational needs. However, here are some common objectives and goals of software testing:

1. Identify defects: The primary objective of software testing is to identify defects or bugs in the software. By executing various test cases, testers aim to uncover errors or discrepancies in the system's functionality, design, or performance.

2. Ensure software quality: Testing helps ensure that the software meets the desired quality standards. It involves validating that the software functions as expected, meets the specified requirements, and performs reliably under different conditions.

3. Enhance user satisfaction: Testing plays a crucial role in enhancing user satisfaction by identifying and addressing issues that could impact the user experience. By testing the software from the user's perspective, usability and accessibility issues can be identified and resolved.

4. Verify compliance: In certain domains, such as finance or healthcare, software must comply with specific regulations and standards. Testing helps verify that the software meets the required compliance criteria, ensuring legal and regulatory obligations are met.

5. Mitigate risks: Testing helps identify and mitigate risks associated with software failure. By uncovering defects early in the development cycle, testing reduces the likelihood of critical issues arising in production, which could lead to financial loss, reputational damage, or safety hazards.

6. Improve software performance: Performance testing aims to evaluate the responsiveness, scalability, and stability of the software under various workload conditions. By identifying performance bottlenecks, testing helps optimize the software's speed, resource usage, and overall efficiency.

7. Validate requirements: Testing ensures that the software meets the specified requirements. By mapping test cases to requirements, testers validate that the software behaves as intended and fulfills the stakeholders' expectations.

8. Facilitate maintenance and future enhancements: Through testing, weaknesses and areas requiring improvement are identified, enabling developers to refine the software. Testing also establishes a baseline for future enhancements and ensures that modifications or updates do not introduce new defects.

9. Reduce overall costs: While testing incurs costs, it helps in reducing overall project costs by identifying defects early. Early detection and resolution of issues prevent them from propagating into subsequent stages of development, where they become more expensive and time-consuming to fix.

10. Build confidence: Testing provides stakeholders with confidence in the software's quality and reliability. By demonstrating that the software has been thoroughly evaluated and validated, testing instills trust in the end-users, customers, and other stakeholders.

BUT REMEMBER THESE GENERAL SOFTWARE TESTING PRINCIPLES [read more from Practical Software Testing: A Process-Oriented Approach By Chapter 2 Ilene Burnstein Page 19]

1.Testing shows presence of bugs but does not show that software is bug free

2. Exhaustive testing is impossible. On a large project it is not practically possible to test all combinations of inputs (data) and preconditions.

3. Early Testing is important. Testers do not need to wait until software is available before commencing testing activities.

4. Defect clustering. Spread of defect in software is not uniform.

5. Running the same test continually will not find new defects. Regression tests should change to reflect change in business needs.

6. Testing is context dependent. The Test Approach, tools and techniques that are used on a particular test project will not be the same as those used on a different test project. For example how you test a website application will be different from that of a database application.

7. Absence of errors fallacy. The fact that no errors were outstanding does not imply fitness for go live. Users' expectations should be met.

Reflection on basic terminology

Which of the following is a software fault?

- A programmer forgot to write the validation code for a numeric value.
- Some code that validates a month number reads: IF MONTH < 1 OR MONTH >=12 THEN Display "Invalid month" ENDIF
- A screen rejects "12/01/99" as an "invalid date"

The correct answer is the second scenario: Some code that validates a month number reads: IF MONTH < 1 OR MONTH >=12 THEN Display "Invalid month" ENDIF Yes. This would treat 12 as an INVALID month number

Which of the following is a software failure?

- The variable 'TODAY' is not initialised in a program, when started
- The business analyst failed to document all the requirements in a software requirements specification
- The software which performs the premium calculation for an insurance policy produces the wrong result.

The answer is the third option: The software which performs the premium calculation for an insurance policy produces the wrong result Yes. The software fails to produce the correct result.

Testing levels (unit, integration, acceptance)

Software testing is typically divided into multiple levels or stages to ensure comprehensive coverage and effectiveness. Here are the commonly recognized testing levels:

1. Unit Testing: Unit testing focuses on testing individual units or components of software in isolation. It verifies that each unit functions correctly as per its design and meets the specified requirements. Unit testing is usually performed by developers and is often automated. Test frameworks like JUnit and NUnit are commonly used for unit testing.

2. Integration Testing: Integration testing verifies the interactions and interfaces between different units or components of the software. It aims to uncover defects that may occur due to the integration of various units. Integration testing can be conducted at different levels, such as module integration (testing interactions between modules) or system integration (testing interactions between subsystems or modules).

3. System Testing: System testing involves testing the entire integrated system as a whole. It verifies the behavior and functionality of the system by executing end-to-end test scenarios. System testing

ensures that all the components of the software work together correctly and meet the specified requirements. It is performed from a user's perspective and includes functional and non-functional testing.

4. Acceptance Testing: Acceptance testing is conducted to determine whether the software meets the acceptance criteria and satisfies the user or customer requirements. It validates the software against business use cases and scenarios. Acceptance testing can be performed in different ways, such as User Acceptance Testing (UAT), where end-users test the software in a real or simulated environment, or by using acceptance criteria defined in the project documentation.

Additionally, there are other testing levels that are sometimes considered separate from the traditional hierarchy:

5. Regression Testing: Regression testing is not a separate level but rather a type of testing that ensures that changes or modifications to the software do not introduce new defects or regressions in existing functionality. It involves retesting previously tested features and functionalities to ensure their stability after changes.

6. Performance Testing: Performance testing evaluates the software's performance characteristics, such as responsiveness, scalability, reliability, and resource usage, under expected or simulated workload conditions. It helps identify performance bottlenecks, measure response times, and ensure the software meets the performance requirements.

7. Security Testing: Security testing focuses on identifying vulnerabilities and weaknesses in the software that may lead to security breaches or unauthorized access. It involves testing for potential threats, vulnerabilities, and verifying if the software complies with security standards and best practices.

It's important to note that the testing levels mentioned above are not strictly sequential but can overlap and be executed iteratively based on the software development lifecycle and project requirements. The specific testing levels and their scope may vary depending on the organization, project, and the software development methodology being followed.

Testing types (functional, non-functional, regression, performance, etc.)

Software testing encompasses various types of testing to address different aspects of the software's quality and behavior. Here are some common types of software testing:

1. Functional Testing: Functional testing verifies that the software functions correctly and meets the specified functional requirements. It involves testing individual features and functionalities to ensure they perform as expected.

2. Non-Functional Testing: Non-functional testing focuses on evaluating the software's non-functional aspects, such as performance, usability, security, reliability, and compatibility. It ensures that the software meets the desired quality standards beyond its functional requirements.

3. Regression Testing: Regression testing is performed to ensure that modifications or changes in the software do not introduce new defects or negatively impact existing functionality. It involves retesting previously tested features to ensure their stability.

4. Performance Testing: Performance testing evaluates the software's performance characteristics, such as responsiveness, scalability, reliability, and resource usage, under expected workload conditions. It helps identify performance bottlenecks and ensures the software meets performance requirements.

5. Usability Testing: Usability testing assesses the software's user-friendliness and ease of use. It aims to determine how well users can interact with the software, understand its features, and accomplish their tasks effectively and efficiently.

6. Security Testing: Security testing focuses on identifying vulnerabilities and weaknesses in the software that may lead to security breaches or unauthorized access. It involves testing for potential threats, verifying authentication and authorization mechanisms, and ensuring compliance with security standards.

7. Compatibility Testing: Compatibility testing verifies that the software functions correctly across different platforms, operating systems, browsers, devices, or network environments. It ensures that the software is compatible with the intended deployment configurations.

8. Integration Testing: Integration testing verifies the interactions and interfaces between different components or modules of the software. It ensures that the integrated units work together correctly and that data flows smoothly between them.

9. Acceptance Testing: Acceptance testing is conducted to determine whether the software meets the acceptance criteria and satisfies user or customer requirements. It validates the software against business use cases and scenarios.

10. Exploratory Testing: Exploratory testing is an ad-hoc testing approach where testers explore the software without predefined test cases. It involves simultaneous learning, testing, and design, allowing testers to find defects and unexpected behaviors.

11. Load Testing: Load testing evaluates the software's performance under expected or simulated high workload conditions. It assesses how the software handles concurrent users, transactions, or data volumes to identify performance bottlenecks and ensure scalability.

12. Stress Testing: Stress testing involves evaluating the software's behavior under extreme or beyond-normal workload conditions to assess its stability and robustness. It helps identify the software's breaking points and how it recovers from failures.

These are just a few examples of testing types, and there are many more specialized types of testing based on specific requirements, industry standards, or project needs. The selection of testing types depends on the software's nature, project goals, and the risks associated with it.

Test planning and system strategy

There are several common risks associated with software testing that organizations should be aware of. These risks can have an impact on the effectiveness and efficiency of the testing process, as well as on the overall quality of the software being developed. Here are some examples of common risks in software testing:

1. Incomplete Test Coverage: One of the major risks in software testing is inadequate test coverage. If the testing efforts do not cover all the critical functionalities, features, and scenarios of the software, there is a higher chance of important defects being missed.

2. Insufficient Resources: Limited resources, such as time, budget, and skilled personnel, can pose a risk to software testing. Insufficient resources may result in rushed testing, reduced test coverage, or inadequate defect tracking and reporting, which can lead to lower quality software.

3. Changing Requirements: Requirements changes during the development lifecycle can pose challenges for testing. When requirements are modified or updated, the corresponding test cases and test scripts may need to be adjusted or redeveloped, leading to additional effort and potential gaps in test coverage.

4. Time Constraints: Tight project schedules and time constraints can put pressure on the testing process. In such situations, there may be a tendency to cut corners, reduce the number of test cycles, or skip certain types of testing, compromising the thoroughness and effectiveness of testing.

5. Communication and Collaboration Issues: Poor communication and collaboration between testers, developers, and stakeholders can hinder the testing process. Lack of clarity in requirements, inadequate information sharing, or ineffective feedback loops can lead to misunderstandings, delays in issue resolution, and reduced overall testing effectiveness.

6. Technical Challenges: Software testing may encounter technical challenges related to the complexity of the software, integration issues with other systems, or dependencies on external components. These challenges can make it difficult to set up test environments, simulate realistic scenarios, or perform certain types of testing effectively.

7. Test Environment Limitations: Limited availability or instability of test environments can be a risk to testing. If the test environment does not accurately represent the production environment or lacks necessary resources, it may result in incomplete or unrealistic testing, leading to potential issues in the deployed software.

8. Data Management: Managing test data effectively can be a challenge. Inadequate or insufficient test data can lead to incomplete testing coverage or unrealistic testing scenarios. On the other hand, using sensitive or confidential data in testing environments can pose security and privacy risks.

9. Tool Limitations: Testing tools and automation frameworks may have limitations that can impact the testing process. Incompatibility with the software under test, limited functionality, or a steep learning curve for testers can hinder the effectiveness and efficiency of testing efforts.

10. Regression Issues: Changes or fixes made in the software can inadvertently introduce new defects or regressions in previously tested functionality. Inadequate regression testing or incomplete test suites can increase the risk of undetected defects resurfacing in subsequent versions or releases.

To mitigate these risks, organizations should adopt risk management strategies, establish clear communication channels, allocate sufficient resources, prioritize test coverage, and invest in appropriate tools and technologies to support the testing process effectively.

Test Design Techniques

Test design techniques help testers systematically identify and define test cases to ensure comprehensive coverage and effectiveness in software testing. Here are some commonly used test design techniques:

1.Equivalence Partitioning: Equivalence Partitioning or Equivalence Class Partitioning is type of black box testing technique which can be applied to all levels of software testing like unit, integration, system, etc.

2. Boundary Value Analysis

3.Decision Table Testing: Decision table testing is used when a system's behavior is determined by combinations of conditions or rules. Test cases are derived from the decision table, covering all possible combinations of conditions and their corresponding outcomes.

4. State Transition Testing: State transition testing is applicable to systems with different states and transitions between them. Test cases are designed to cover the transitions from one state to another, including valid and invalid transitions, to ensure that the system behaves as expected in different states.

5. Pairwise Testing: Pairwise testing, also known as all-pairs testing, is a combinatorial technique that aims to cover all possible combinations of input parameters with minimum test cases. It selects a subset of test cases that covers all possible pairs of input values, thereby reducing the number of test cases required.

6. Use Case Testing: Use case testing focuses on testing the system's behavior and functionality based on realistic usage scenarios. Test cases are derived from use case specifications, covering the main and alternative flows of each use case.

7. Error Guessing: Error guessing is an informal technique where testers use their experience and intuition to identify potential error-prone areas in the system. Test cases are designed to target those specific areas where defects are likely to occur based on past experience or knowledge.

8. Exploratory Testing: Exploratory testing is an unscripted approach where testers dynamically explore the system, learn its behavior, and design and execute test cases on the fly. It leverages the tester's domain knowledge and experience to uncover defects and unexpected behaviors.

9. Risk-Based Testing: Risk-based testing involves identifying and prioritizing test cases based on the risks associated with the software and its impact on stakeholders. It focuses testing efforts on high-risk areas to ensure that critical functionalities and potential failure points are thoroughly tested.

10. Model-Based Testing: Model-based testing utilizes models, such as state diagrams or flowcharts, to design and generate test cases. Test cases are derived from the model, ensuring that all paths and states are covered, reducing manual effort and improving test coverage.

These are just a few examples of test design techniques. The selection of appropriate techniques depends on factors such as the nature of the system, project requirements, available resources, and the expertise of the testing team. A combination of multiple techniques is often used to achieve comprehensive test coverage.

To determine the most suitable test design technique for your project, you can consider the following factors:

1. Project Requirements: Review the project requirements and understand the nature of the software being developed. Consider whether the software has complex business rules, state-based behavior, input validation requirements, or data dependencies. Different techniques may be more suitable for different types of systems.

2. Test Objectives: Clarify the testing objectives and goals of your project. Determine what aspects of the software you want to test, such as functionality, performance, security, or usability. Some techniques may be more effective for specific objectives, such as equivalence partitioning for input validation or state transition testing for state-based systems.

3. Risk Analysis: Conduct a risk analysis to identify high-risk areas and critical functionalities of the software. Determine which parts of the system are crucial for the success of the project or have a high impact on stakeholders. Use risk-based testing to prioritize testing efforts and select appropriate techniques for those high-risk areas.

4. System Complexity: Assess the complexity of the system under test. Consider factors such as the number of inputs, dependencies between components or modules, the presence of conditional

logic, or the number of states and transitions. Techniques like pairwise testing or decision table testing can be useful for complex systems with a large number of inputs.

5. Available Resources: Evaluate the resources available for testing, including time, budget, and skill sets of the testing team. Some test design techniques may require more time or effort to implement or may require specialized skills or tools. Consider the feasibility and practicality of using certain techniques given your available resources.

6. Test Coverage: Consider the desired test coverage. Determine whether you need to cover all possible combinations of input values, focus on specific scenarios, or achieve comprehensive coverage of critical functionalities. Techniques like pairwise testing, use case testing, or state transition testing can help achieve specific coverage goals.

7. Previous Experience: Consider the experience and expertise of your testing team. If your team has prior experience with certain test design techniques and has found them effective in similar projects, it may be advantageous to leverage that experience and use those techniques again.

8. Combination of Techniques: Remember that you are not limited to using a single test design technique. In many cases, combining multiple techniques can provide better coverage and effectiveness. Consider using a mix of techniques that complement each other to achieve the desired testing goals.

By analyzing these factors and considering the specific characteristics of your project, you can make an informed decision about which test design technique or combination of techniques is most suitable for your project. It's important to adapt and tailor the techniques to fit your project's unique requirements and constraints.

Black-box testing techniques (equivalence partitioning, boundary value analysis, decision table testing, etc.)

Specification-based or Black Box Testing Techniques Black box testing is also called functional testing Treat software under test as a black box No knowledge of internals required or assumed Tests based on specification of required behaviour of the system Specifications should not define how a system should achieve the specified behaviour but what a system should do Specification is used to design tests Specification is used as the baseline for results The following are the Black Box techniques that are used for designing Test Cases:

1. Equivalence partitioning
2. Boundary value analysis
3. Decision table testing
4. State transition testing
5. Use case testing

Equivalence partitioning divides the input domain into classes or partitions, where each partition is expected to exhibit similar behavior. Test cases are then designed to cover representative values from each partition, minimizing redundancy and maximizing coverage.

The main idea behind equivalence partitioning is that if a certain input value causes a particular behavior or outcome, then any other input value within the same partition should also produce the same behavior or outcome.

The goal of equivalence partitioning is to select a representative value from each partition and use it as a test case. By doing so, you can ensure that you test a broad range of values and cover different scenarios without having to test every possible input individually.

Step-by-step process of using equivalence partitioning:

Identify the input variables: Determine the inputs that are relevant to the system or component you are testing.

Divide inputs into partitions: Divide the input range into different partitions based on the behavior of the system. Each partition represents a set of inputs that should produce the same results. For example, if you have an input field that accepts numbers between 1 and 100, you can divide it into partitions such as values less than 1, values between 1 and 50, and values greater than 50.

Select representative values: From each partition, select one or more representative values. These values should be typical or boundary values that are likely to expose defects if they exist. Using our previous example, you might choose the values 0, 25, 50, 75, and 101 as representative values.

Design test cases: Design test cases using the representative values from each partition. Each test case represents a set of inputs that belong to a specific partition. For example, you might have test cases like "Test with input value 0," "Test with input value 25," and so on.

Execute test cases: Execute the designed test cases and observe the behavior of the system. If the system behaves as expected for all the representative values within a partition, you can assume that it will work correctly for any other value within that partition.

The advantage of equivalence partitioning is that it helps to reduce the number of test cases while still providing reasonable coverage of input scenarios. By focusing on representative values, you can efficiently test the system and identify potential defects. However, it's important to note that equivalence partitioning is not a replacement for other testing techniques, such as boundary value analysis or exhaustive testing, but rather a complementary approach that can be used in combination with them.

Consider a real-world scenario of testing a login functionality for a web application. The login functionality typically requires a username and password as input.

Example-Here's how equivalence partitioning can be applied:

Identify the input variables:

Username: A string input.

Password: A string input.

Divide inputs into partitions:

Username partitions: Empty username, usernames with less than 6 characters, usernames with 6 to 20 characters, and usernames with more than 20 characters.

Password partitions: Empty password, passwords with less than 8 characters, passwords with 8 to 16 characters, and passwords with more than 16 characters.

Select representative values:

- Username representative values: "", "user", "username1234567890", "verylongusername1234567890".
- Password representative values: "", "pass", "password1", "verystrongpassword123456".

Design test cases:

- Test Case 1: Test with empty username and empty password.
- Test Case 2: Test with a username of "user" and an empty password.
- Test Case 3: Test with a username of "user" and a password of "pass".
- Test Case 4: Test with a username of "username1234567890" and a password of "password1".
- Test Case 5: Test with a username of "verylongusername1234567890" and a password of "verystrongpassword123456".

Execute test cases:

Execute each test case and observe the behavior of the login functionality. Verify if the system behaves as expected for each representative value within the respective partitions.

By applying equivalence partitioning, we have identified representative values that cover different scenarios for the username and password inputs. Testing these representative values helps ensure

that the login functionality is working correctly across different input ranges and potential edge cases.

Of course, in practice, there could be additional factors to consider, such as error handling, account lockout policies, and security measures. Equivalence partitioning is just one technique within a broader testing strategy, but it can significantly reduce the number of test cases while still providing reasonable coverage.

Summary-Equivalence Partitioning

Requirements documents normally state rules that the software must obey Rule A might be "if m is less than one, then do this" Rule B might be, "if m is greater than or equal to one, and less than or equal to twelve, then do this…" and so on.

Think about the rules for what values a month field can hold. If the month is less than one, that's invalid If it's between one and twelve, that's okay and the value can be accepted.

If it's greater than twelve, that's invalid and you print an error.

All the infinite range of integers that could be entered into a system must fall into one of those three criteria, one of those categories: less than one between one and twelve greater than twelve If we select one value from each range of numbers, and use this as a test case, we could say that we have covered the three rules. Does it matter what values we take? Not really! We could in fact say that every value in a partition is equivalent. This is where the name of the technique comes from By doing this, our infinite number of possible values will all be covered by the three categories, the three rules!

Business Rules	Input Partition	Output Partition
Month less than one print "invalid too low"	Month<1	invalid too low
Month between one and twelve print "valid"	(Month>=1, (Month<=12)	valid
Month greater than twelve print "invalid too high"	(Month>12)	invalid too high

Exercises-application of Equivalence Partitioning

- Design an input and output partition table to include business rules, valid input and output partitions for primary, secondary and university scenarios with respect to age of learners.
- Develop black box test cases using equivalence class partitioning and boundary value analysis to test a module that is software component of an automated teller system. The module reads in the amount the user wishes to withdraw from his/ her account. The amount must be a multiple of $5.00 and be less than or equal to $200.00. Be sure to list any assumptions you make and label equivalence classes and boundary values that you use.
- One of the fields on a form contains a text box that accepts numeric values in the range of 18 to 25. Identify equivalence classes.

The text box accepts numeric values in the range of 18 to 25 (18 and 25 are also part of the class).So this class becomes our valid class.

The classes will be as follows:

Class I: values < 18 => invalid class

Class II: 18 to 25 => valid class

Class III: values > 25 => invalid class

Another example

In a system designed to work out the taxes to be paid:

An employee has $4000 of salary tax-free.

The next $1500 is taxed at 10%.

The next $28000 after that is taxed at 22%.

Any further amount is taxed at 40%.

To the nearest whole dollar, which of these groups of numbers fall into three DIFFERENT equivalence classes?

a) $4000; $5000; $5500

b) $32001; $34000; $36500

c) $28000; $28001; $32001

d) $4000; $4200; $5600

Solution:

The classes will be as follows:

Class I: 0 to $4000 => no tax

Class II: $4001 to $5500 => 10 % tax

Class III: $5501 to $33500 => 22 % tax

Class IV: $33501 and above => 40 % tax

Select the values that fall into three different equivalence classes. Option 'd' has values from three different equivalence classes.

2. Boundary Value Analysis: Boundary value analysis focuses on testing the boundaries and extreme values of input ranges. Test cases are designed to cover values at the lower and upper boundaries, as well as just inside and outside those boundaries, as they are more likely to reveal defects.

What is Boundary Testing?

Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing".

The basic idea in normal boundary value testing is to select input variable values at their:

- Minimum
- Just above the minimum
- A nominal value
- Just below the maximum
- Maximum

Boundary Value assumes that errors tend to cluster around boundaries

Using our previous example of Date partitions, we can see that for the date to be valid, it must fall within the "valid boundary values" of 1 and 12.

Also for the date to be invalid it will fall outside these valid boundaries The principle of testing using BVA is to use the boundary value itself and one value as close as you can get either side of the boundary, using the precision that has been applied to the partition. 0,1,2 are valid values for testing the lower boundary 11,12,13 are valid values for testing the upper boundary.

Boundary testing is a software testing technique that focuses on testing the boundaries or limits of input values or conditions. The goal of boundary testing is to identify errors and vulnerabilities that may occur at the edges or boundaries of valid input ranges.

In boundary testing, test cases are designed to include both the minimum and maximum allowable values, as well as values just inside and just outside these boundaries. By testing these boundary values, it helps to ensure that the software handles them correctly and does not produce unexpected results or errors.

Boundary testing is particularly useful when dealing with input fields or parameters that have specific limits or constraints. For example, if a text field allows a maximum of 100 characters, boundary testing would involve testing inputs with exactly 100 characters, 101 characters, and 99 characters to check if the system handles these cases correctly.

The benefits of boundary testing include:

- Identifying off-by-one errors: Boundary values often expose mistakes related to off-by-one errors, where an incorrect assumption is made about the range of acceptable input values.
- Revealing system vulnerabilities: Errors and vulnerabilities are more likely to occur at the boundaries, so testing these areas helps identify potential issues that might otherwise go unnoticed.
- Enhancing test coverage: Boundary testing provides additional test coverage beyond standard functional testing by focusing on the edges of input ranges.

It's important to note that boundary testing should be used in conjunction with other testing techniques to ensure comprehensive test coverage.

Using the tax example above

To the nearest whole dollar, which of these is a valid Boundary Value Analysis test case?

a) $28000

b) $33501

c) $32001

d) $1500

Solution:

The classes are already divided above. We have to select a value which is a boundary value (start/end value). 33501 is a boundary value.

Another question

Find valid learners age boundaries and valid age values for testing the primary, secondary and university partitions from the scenario described above.

Examples of how boundary value testing and equivalence partitioning can be applied in performance testing scenarios.

Load Testing: In load testing, the goal is to evaluate the system's performance under expected or peak loads. Boundary value testing can be used to determine the maximum load the system can handle. For example, if the application is expected to handle a maximum of 1000 concurrent users, you can design test scenarios with 990, 1000, and 1010 concurrent users to assess the system's behavior at the load boundaries. Equivalence partitioning can be applied to select representative load levels, such as low (100 users), medium (500 users), and high (1000 users).

Stress Testing: Stress testing is performed to assess the system's ability to handle extreme loads or stressful conditions. Boundary value testing is crucial in stress testing as it helps identify the system's breaking point or performance degradation thresholds. For example, if the system is expected to handle up to 100 transactions per second, you can design stress test scenarios with 90, 100, and 110 transactions per second to evaluate the system's stability at the performance boundaries. Equivalence partitioning can be used to select different stress levels, such as moderate (50 transactions per second) and high (90 transactions per second).

Spike Testing: Spike testing involves subjecting the system to sudden, significant increases in load to assess its response. Boundary value testing can be applied to determine the maximum spike load the system can handle. For example, if the system is expected to handle a spike of 1000 requests within 1 minute, you can design spike test scenarios with 900, 1000, and 1100 requests within the 1-minute time frame to evaluate the system's behavior at the load boundaries. Equivalence

partitioning can be used to select different spike levels, such as low (500 requests), medium (800 requests), and high (1000 requests).

Endurance Testing: Endurance testing involves subjecting the system to a sustained load over an extended period to assess its performance and stability. Boundary value testing can be used to determine the maximum sustained load the system can handle without degradation. For example, if the system needs to handle a sustained load of 500 transactions per hour, you can design endurance test scenarios with 450, 500, and 550 transactions per hour to evaluate the system's performance at the load boundaries. Equivalence partitioning can be applied to select different sustained load levels, such as low (200 transactions per hour) and high (400 transactions per hour).

By combining boundary value testing and equivalence partitioning in performance testing, you can ensure that the system is tested for different load levels, stress levels, and performance boundaries, thereby identifying potential performance bottlenecks, scalability issues, and response time problems.

3.Decision table testing

Decision table testing, also known as cause-effect table testing, is a technique used to systematically test combinations of inputs or conditions and corresponding actions or outputs. It helps ensure comprehensive test coverage by considering different combinations of conditions and their effects. The decision table consists of four main components:

1. Conditions: These are the inputs or conditions that affect the behavior of the system or application being tested. Conditions are typically represented as column headings in the decision table.

2. Actions: These are the outputs or actions that result from the combinations of conditions. Actions are represented as the possible outcomes or actions in the decision table.

3. Rules: Each row in the decision table represents a specific combination of conditions and the associated action(s) or output(s). These rules define the behavior of the system under different conditions.

4. Conditions' Values: The possible values or states that each condition can take are listed in the decision table.

A step-by-step process to perform decision table testing:

Step 1: Identify Conditions and Actions

- Identify the relevant conditions that influence the behavior of the system.

- Determine the actions or outputs that are expected based on the combinations of conditions.

Step 2: Define Condition Values

- Determine the possible values or states for each condition.

- List these condition values in the decision table.

Step 3: Create Rules

- Generate all possible combinations of condition values.

- Create rules by associating each combination of conditions with the corresponding action(s) or output(s).

Step 4: Test Case Generation

- For each rule in the decision table, create a test case that covers that specific combination of conditions.

Step 5: Execute Test Cases

- Execute the test cases derived from the decision table.

- Verify that the system behaves as expected for each combination of conditions.

Step 6: Analyze Coverage

- Analyze the decision table to ensure that all possible combinations of conditions have been covered by the test cases.

- Identify any missing combinations or gaps in test coverage.

By following this process, decision table testing helps ensure comprehensive test coverage by considering different combinations of conditions and their corresponding actions or outputs. It is particularly useful when there are multiple conditions and their combinations significantly affect the system's behavior.

There are several tools and frameworks that can assist with decision table testing. These tools provide functionalities to create, manage, and automate decision tables, making the testing process more efficient. Some popular tools and frameworks for decision table testing include:

1. Microsoft Excel: Excel is a widely used tool for creating decision tables. It provides features like conditional formatting, sorting, filtering, and formula-based calculations that can be utilized to manage and analyze decision tables effectively.
2. SpecFlow: SpecFlow is a behavior-driven development (BDD) framework for .NET. It allows you to define decision tables using the Gherkin language and automate the testing process by generating executable test cases from the decision table specifications.
3. FitNesse: FitNesse is an open-source testing framework that supports decision table testing. It provides a wiki-based platform where decision tables can be defined and executed as part of the testing process. FitNesse allows easy collaboration between testers, developers, and stakeholders.
4. Cucumber: Cucumber is a popular BDD testing tool that supports decision table testing. It allows you to define decision tables using Gherkin syntax and generate executable test cases. Cucumber supports integration with various programming languages and frameworks.
5. Tricentis Tosca: Tricentis Tosca is an enterprise-level test automation tool that includes support for decision table testing. It provides a graphical user interface (GUI) for creating and managing decision tables. Tosca also offers features like test case management, test data management, and test execution automation.

6. Roolie: Roolie is an open-source Java library specifically designed for decision table testing. It provides a simple and intuitive API for defining decision tables in code and executing them. Roolie supports complex decision table structures and rule evaluation.

These tools and frameworks offer different features and capabilities for decision table testing. The choice of tool depends on factors such as project requirements, programming language, level of automation needed, and team preferences. It's important to evaluate these tools based on your specific needs and select the one that best suits your requirements.

Lets look at real life application of Decision Table Testing

Decision table testing finds application in various real-life scenarios across different domains. Here are a few examples of how decision table testing can be applied.

1. Insurance Claims Processing: In the insurance industry, decision table testing can be used to verify the accuracy and consistency of claims processing systems. The decision table can capture different conditions and rules based on which claims are evaluated, such as policy coverage, claim amount, deductibles, and eligibility criteria. By testing different combinations of conditions, decision table testing helps ensure that claims are processed correctly and in accordance with the established rules.

2. Loan Approval Systems: Financial institutions often employ decision table testing to validate loan approval systems. The decision table can include various parameters like credit score, income level, loan amount, and debt-to-income ratio. By testing different combinations of conditions, decision table testing helps ensure that loan applications are evaluated accurately, and the appropriate approval or rejection decisions are made.

3. Software Configuration Management: Decision table testing is relevant in software configuration management, especially when dealing with different configurations or feature sets. The decision table can capture various conditions related to the configuration options and their dependencies. Testing different combinations of conditions helps ensure that the software behaves correctly and consistently across different configuration scenarios.

4. Traffic Signal Control Systems: Decision table testing can be used to validate the behavior of traffic signal control systems. The decision table can capture conditions such as traffic volume, time of day, and pedestrian signals. By testing different combinations of conditions, decision table testing helps ensure that the traffic signal control system responds appropriately to varying traffic situations and follows the predefined rules for signal sequencing.

5. Healthcare Diagnosis Systems: Decision table testing can be applied to healthcare diagnosis systems to validate the accuracy and consistency of the diagnostic process. The decision table can include symptoms, medical history, test results, and other relevant conditions. By testing different combinations of conditions, decision table testing helps ensure that the diagnosis system provides correct diagnoses based on the input information.

The technique is versatile and can be adapted to various domains and applications where the behavior of a system depends on multiple conditions and their combinations.

Limitations and challenges associated with decision table testing:

Combinatorial Explosion: When dealing with a large number of conditions and their possible combinations, the number of rules in the decision table can grow exponentially. This can lead to a combinatorial explosion, making it impractical to test all possible combinations. Testers must carefully select a representative set of test cases that cover the most critical combinations while keeping the testing effort manageable.

Missing Rules: The process of creating decision tables requires identifying all relevant conditions and their possible values. However, it is possible to overlook or miss certain conditions or rules during the creation of the decision table. This can result in missing test cases and incomplete coverage. It's important to ensure that the decision table is comprehensive and all possible combinations are considered.

Ambiguity and Overlapping Rules: Decision tables, especially those with complex conditions, can sometimes lead to ambiguity or overlapping rules. Ambiguity occurs when it is unclear which action or output should be associated with a particular combination of conditions. Overlapping rules occur when multiple rules cover the same combination of conditions, leading to potential conflicts. Careful analysis and review of the decision table can help identify and resolve such issues.

Maintenance and Complexity: Decision tables can become complex, especially when dealing with a large number of conditions and rules. As the system evolves or requirements change, maintaining and updating the decision table can become challenging. It's crucial to establish a process for managing and updating the decision table effectively to ensure its accuracy and relevance over time.

Limited Visibility of Interactions: Decision table testing focuses on individual combinations of conditions and their effects. However, it may not capture the interactions between conditions or the dependencies that exist. Some defects or issues may only surface when specific conditions interact in a particular way. Testers should be aware that decision table testing may not fully reveal such interactions, and additional testing techniques may be needed to address them.

Test Case Prioritization: Due to the combinatorial nature of decision table testing, it may not be feasible to test all possible combinations. Testers need to prioritize the test cases based on risk analysis, criticality, and coverage goals. This prioritization can be challenging, and there is a need for careful consideration to ensure that the most important combinations are adequately covered.

Despite these limitations and challenges, decision table testing remains a valuable technique for systematically testing the combinations of conditions and their effects. By understanding these limitations and addressing them appropriately, testers can leverage decision table testing effectively to achieve comprehensive test coverage.

3. State transition testing

State transition testing is a black-box testing technique used to validate the behavior of a system or software application based on different states or conditions. It focuses on testing the transitions between these states and ensuring that the system behaves correctly in response to those transitions.

State transition testing is useful for both procedural and object-oriented development. It is based on the concepts of states and finite-state machines, and allows the tester to view the developing software in term of its states, transitions between states, and the inputs and events that trigger state changes. This view gives the tester an additional opportunity to develop test cases to detect defects that may not be revealed using the input/output condition as well as cause-and-effect views presented by equivalence class partitioning and cause-and-effect graphing.

The basic idea behind state transition testing is to identify the various states that a system can be in, as well as the events or actions that can cause transitions between those states. These states are typically represented in a state diagram or state transition diagram.

The process of state transition testing involves the following steps:

1. Identify states: Identify the different states that the system can be in. These states represent the different conditions or modes of operation of the system.

2. Define transitions: Determine the events or actions that can cause transitions between states. These events can be user actions, system events, or time-based triggers.

3. Create a state transition diagram: Represent the identified states and transitions in a graphical form, such as a state transition diagram. This diagram helps visualize the different states and transitions within the system.

4. Define test cases: Based on the state transition diagram, define test cases that cover all possible transitions and states. Each test case should specify the initial state, the triggering event, the expected outcome, and the resulting state.

5. Execute test cases: Execute the defined test cases, following the specified sequences of states and transitions. Record the actual outcomes and compare them against the expected outcomes.

6. Validate system behavior: Verify that the system behaves correctly during state transitions. Check whether the system transitions to the expected states, and if any associated actions or conditions are correctly triggered or maintained.

State transition testing helps uncover defects related to state changes, such as incorrect transitions, missing transitions, or invalid behavior in specific states. It is particularly useful in scenarios where the behavior of a system depends on certain conditions or states, such as user authentication, order processing, or system modes.

By systematically testing the various states and transitions, state transition testing helps improve the reliability and robustness of software systems, ensuring that they handle state changes correctly and consistently.

State transition testing, like any testing technique, has its own set of challenges and limitations.

Some common challenges and limitations of state transition testing include:

1. Complexity: State transition testing can become complex, especially in systems with a large number of states and transitions. Managing and modeling all possible states and transitions can be challenging, and it may require significant effort to create an accurate and comprehensive state transition diagram.

2. State explosion problem: In systems with a high number of states and transitions, the number of possible test cases can grow exponentially. This can make it impractical or infeasible to test all possible combinations of states and transitions. Testers need to prioritize and select a representative set of test cases that cover the most critical and likely scenarios.

3. Incomplete or ambiguous specifications: State transition testing heavily relies on clear and precise specifications of the system's states and transitions. If the specifications are incomplete or ambiguous, it can lead to difficulties in identifying the correct states and transitions, resulting in incomplete or inaccurate testing.

4. Time and resource constraints: Testing all possible states and transitions can be time-consuming and resource-intensive. Organizations often have limited time and resources for testing, which may restrict the depth and coverage of state transition testing. Testers need to prioritize and allocate resources effectively to focus on the most critical aspects of the system.

5. Interactions between states: In complex systems, the behavior of the system may depend not only on individual states but also on interactions between states. Identifying and testing all possible combinations of state interactions can be challenging. Testers need to carefully consider the dependencies and interactions between states to ensure comprehensive testing.

6. Handling unexpected states or transitions: In real-world scenarios, systems may encounter unexpected states or transitions that are not explicitly defined in the state transition diagram. It is important to consider and handle such exceptional cases to ensure the robustness of the system. However, it can be challenging to anticipate and cover all possible unexpected scenarios.

7. Maintenance overhead: State transition testing requires maintaining the state transition diagram and associated test cases as the system evolves. When the system undergoes changes, such as the addition of new states or modifications to existing transitions, it requires updating the testing artifacts accordingly. This maintenance overhead can be time-consuming and may introduce additional complexity.

Despite these challenges and limitations, state transition testing remains a valuable technique for testing systems with stateful behavior. It helps uncover defects related to state transitions and provides a structured approach to validate the correctness of system behavior during state changes. From the tester’s view point the review should ensure that (i) the proper number of states are represented, (ii) each state transition (input/output/action) is correct, (iii) equivalent states are

identified, and (iv) unreachable and dead states are identified. Unreachable states are those that no input sequence will reach, and may indicate missing transitions. Dead states are those that once entered cannot be exited. In rare cases a dead state is legitimate, for example, in software that controls a destructible device.

Testers can effectively prioritize and select a representative set of test cases by considering the following factors:

1. Risk analysis: Perform a risk analysis to identify the critical or high-risk areas of the system. Focus on states or transitions that have a higher impact on the system's functionality, performance, security, or user experience. Allocate more testing effort to these areas to ensure they are thoroughly tested.
2. Requirements and business priorities: Understand the system's requirements and the priorities set by the business or stakeholders. Identify the key functionalities or features that need to be tested and prioritize test cases accordingly. Test the critical or core features first before moving on to secondary or optional features.
3. Coverage goals: Define coverage goals based on the system's states and transitions. Aim to cover the most important and common scenarios, ensuring that the selected test cases exercise a wide range of states and transitions. Consider both positive and negative test cases to cover different conditions and edge cases.
4. Usage patterns and user profiles: Analyze the typical usage patterns and user profiles of the system. Consider the most common paths or sequences of states and transitions that users are likely to follow. Prioritize test cases that cover these common usage scenarios to validate the system's behavior under realistic conditions.

5. Historical defect data: Review historical defect data and feedback from previous testing cycles. Identify recurring or high-impact defects related to state transitions. Prioritize test cases that target

these specific areas to ensure that previous issues have been addressed and that the system is more robust.

6. Collaboration and feedback: Engage in effective collaboration with stakeholders, developers, and other team members involved in the testing process. Seek their input and feedback on the critical areas that need to be tested. Consider their insights and prioritize test cases accordingly to ensure a comprehensive and well-rounded testing approach.

7. Time and resource constraints: Consider the available time and resources allocated for testing. Balance the desired level of coverage with the practical limitations. Select a representative set of test cases that provides the best coverage within the given constraints. Make strategic choices by focusing on high-risk areas and critical functionalities.

It's important to note that prioritization and selection of test cases should be an iterative process. As testing progresses and new information becomes available, continuously reassess and reprioritize test cases to adapt to changing circumstances and ensure that testing efforts remain focused on the most critical aspects of the system.

State transition testing is a widely used technique in various industries and domains. Here are a few real-life examples of its application:

1. ATM (Automated Teller Machine): State transition testing can be applied to test the behavior of an ATM system. The states can include the initial state when the ATM is idle, states for card insertion, PIN entry, selection of transaction types, and states for various transaction outcomes like withdrawal, balance inquiry, or fund transfer. By testing the transitions between these states, testers can verify that the ATM responds correctly to user actions and maintains the correct state during and after transactions.

2. E-commerce website: In an e-commerce website, state transition testing can be used to validate the ordering process. The states can include browsing products, adding items to the cart, entering shipping and billing details, and completing the purchase. Testers can focus on testing transitions

like adding items, removing items, updating quantities, and proceeding to checkout. This helps ensure that the website behaves correctly during the ordering process and handles the different states and transitions accurately.

3. Mobile application: State transition testing is applicable to mobile applications with different modes or states. For example, a messaging app can have states like offline, online, typing, and receiving messages. Testers can verify that the app transitions correctly between these states and handles events like incoming messages or network connectivity changes appropriately. State transition testing helps ensure that the app maintains the expected behavior throughout different user interactions and system events.

4. Medical device: State transition testing is crucial for medical devices with different operation modes or states. For instance, a patient monitoring system may have states like standby, monitoring, and alarm. Testers can validate that the device transitions correctly between these states based on events such as patient signal changes or alarm thresholds being exceeded. By thoroughly testing state transitions, the safety and reliability of the medical device can be ensured.

5. Automotive system: State transition testing is relevant for automotive systems that involve different driving modes or states. For example, an autonomous vehicle may have states like manual control, autonomous control, and emergency mode. Testers can focus on testing the transitions between these states, ensuring that the vehicle responds appropriately to changes in control mode and handles emergency situations correctly.

These are just a few examples of how state transition testing can be applied in real-life scenarios. The technique is versatile and can be used in various systems where behavior depends on different states and transitions. It helps identify and validate the expected behavior of the system during state changes, contributing to the overall quality and reliability of the software or hardware being tested.

In the context of state transition testing, a "dead state" refers to a state in a system where no further transitions or actions are possible. Once the system reaches a dead state, it remains in that state indefinitely without any further changes or interactions.

Dead states can pose challenges and limitations in state transition testing:

1. Lack of coverage: If a dead state is not properly identified and accounted for in the state transition diagram, it can lead to incomplete test coverage. Test cases may overlook the possibility of reaching a dead state and fail to validate the system's behavior in such scenarios.

2. Inadequate error handling: Dead states can indicate unexpected or erroneous behavior in the system. It is important to identify and handle these states appropriately to ensure graceful error recovery or system shutdown. Neglecting to address dead states can result in inadequate error handling and may lead to system instability or unpredictable behavior.

3. Test case termination: Dead states can cause test case termination, as the system cannot progress beyond these states. Testers need to recognize and handle dead states as termination conditions for test cases. Otherwise, test cases may continue indefinitely, consuming unnecessary time and resources.

To address dead states effectively in state transition testing:

1. Identify dead states: Thoroughly analyze the system and the state transition diagram to identify any dead states. These are usually states from which there are no outgoing transitions or no possible further interactions.

2. Define expected behavior: Determine the expected behavior or desired outcome when reaching a dead state. This could involve appropriate error messages, system shutdown, or other recovery procedures.

3. Test dead state handling: Develop test cases specifically to validate the behavior of the system when reaching dead states. Ensure that the system responds appropriately and handles such scenarios gracefully.

4. Error reporting and logging: Implement mechanisms for error reporting and logging to capture instances when the system reaches a dead state. This facilitates effective debugging and troubleshooting, enabling developers to identify and resolve underlying issues.

By recognizing and addressing dead states in state transition testing, testers and developers can improve the overall quality and reliability of the system by ensuring that it handles unexpected or erroneous conditions effectively.

5. Use case testing

White-box testing techniques (statement coverage, branch coverage, path coverage, etc.)

The tester's goal is to determine if all the logical and data elements in the software unit are functioning properly. This is called the white box, or glass box, approach to test case design. The knowledge needed for the white box test design approach often becomes available to the tester in the later phases of the software life cycle, specifically during the detailed design phase of development. This is in contrast to the earlier availability of the knowledge necessary for black box test design. As a consequence, white box test design follows black box design as the test efforts for a given project progress in time. Another point of contrast between the two approaches is that the black box test.

Design strategy can be used for both small and large software components, whereas white box–based test design is most useful when testing small components. This is because the level of detail required for test design is very high, and the granularity of the items testers must consider when developing the test data is very small.

Use case-based testing

Use case-based testing, also known as use case-driven testing, is an approach to software testing that centers around the use cases or scenarios that represent the interactions between users and the system. It involves designing test cases based on these use cases to ensure that the system functions correctly and fulfills the intended requirements. Here's a step-by-step process for conducting use case-based testing:

1. **Identify use cases**: Start by identifying the key use cases that represent the typical interactions between users and the system. Use cases describe specific tasks or actions performed by users to accomplish their goals. These use cases should cover a range of functionalities and represent the most important and commonly performed actions.

2. Define test objectives: Determine the specific objectives for testing each use case. This includes identifying the specific aspects of the system's behavior or functionality that need to be validated. Test objectives can include verifying inputs, outputs, system behavior, error handling, and performance requirements related to each use case.

3. Design test cases: Based on the identified use cases and test objectives, design test cases that cover different scenarios and conditions. Each test case should describe the steps to execute the use case, the expected inputs, and the expected outputs or system behavior. It should also consider variations in inputs, boundary conditions, and error conditions to ensure comprehensive coverage.

4. Execute test cases: Execute the designed test cases by following the steps outlined in each test case. Provide the specified inputs and observe the system's behavior or outputs. Record the actual results and compare them against the expected outcomes. Ensure that the system behaves as expected for each use case.

5. Report and track defects: If any discrepancies are found between the actual results and the expected outcomes, report them as defects or issues in a tracking system. Include detailed information about the test case, the observed behavior, and any supporting logs or screenshots to aid in the understanding and resolution of the issues.

6. Iterative process: Use case-based testing is an iterative process. As the system evolves, new use cases are identified, or requirements change, update the test cases accordingly. Continuously review and refine the testing approach to ensure comprehensive coverage of the system's functionality.

Use case-based testing provides a user-centric perspective on testing, focusing on the system's behavior and functionality in real-life scenarios. It helps ensure that the system meets the needs and expectations of its intended users and functions correctly in practical usage situations. By aligning testing efforts with use cases, this approach enhances the effectiveness and relevance of the testing process.

Website-E-commerce case study

Use Case: User Registration

Description: This use case involves a new user signing up for an account on the e-commerce website. Steps:

1. User navigates to the website's registration page.

2. User enters their personal information, including name, email address, and password.

3. User submits the registration form.

4. System validates the entered information.

5. If the information is valid, the system creates a new user account and displays a success message.

6. If the information is invalid, the system displays appropriate error messages and prompts the user to correct the input.

Based on this use case, we can design several test cases to validate different aspects of the user registration functionality. Here are a few examples:

Test Case 1: Successful Registration

- Test Objective: Verify the system can register a new user successfully.

- Steps:

  1. Navigate to the registration page.

2. Enter valid information for name, email address, and password.

3. Submit the form.

- Expected Outcome: The system creates a new user account and displays a success message.

Test Case 2: Invalid Email Address

- Test Objective: Verify that the system detects and handles an invalid email address.

- Steps:

1. Navigate to the registration page.

2. Enter a valid name and password.

3. Enter an invalid email address (e.g., missing '@' symbol).

4. Submit the form.

- Expected Outcome: The system displays an error message indicating that the email address is invalid.

Test Case 3: Missing Name Field

- Test Objective: Verify that the system detects and handles a missing name field.

- Steps:

1. Navigate to the registration page.

2. Enter a valid email address and password.

3. Leave the name field blank.

4. Submit the form.

- Expected Outcome: The system displays an error message indicating that the name field is required.

Test Case 4: Weak Password

- Test Objective: Verify that the system enforces password strength requirements.

- Steps:

1. Navigate to the registration page.

2. Enter a valid name and email address.

3. Enter a weak password (e.g., fewer than 6 characters).

4. Submit the form.

- Expected Outcome: The system displays an error message indicating that the password does not meet the strength requirements.

Test Case 5: Duplicate Email Address

- Test Objective: Verify that the system detects and handles a duplicate email address.

- Steps:

1. Navigate to the registration page.

2. Enter a valid name, email address, and password.

3. Submit the form.

4. Repeat steps 1-3 with the same email address.

- Expected Outcome: The system displays an error message indicating that the email address is already in use.

User interface testing

User Interface (UI) testing is a software testing technique that focuses on validating the visual elements, functionality, and usability of the user interface of an application. It involves testing the graphical user interface (GUI) components, such as menus, buttons, forms, screens, and their interactions with users. The objective of UI testing is to ensure that the interface is intuitive, responsive, visually appealing, and functions correctly. Here's an overview of UI testing:

1. Functional Testing:

- Validate UI elements: Test the various UI elements, such as buttons, links, checkboxes, radio buttons, dropdowns, etc., to ensure they are present, visible, and functional.

- Interactions: Test the interactions between UI elements and user actions, such as clicking buttons, selecting options, entering data, and validating the expected behavior.

- Navigation: Test the navigation flow within the application, ensuring that menus, tabs, navigation bars, and links function correctly and enable users to move between screens or pages.

- Form Validation: Test the validation of user input in forms, such as required fields, data formats (e.g., email addresses, phone numbers), and error handling when invalid data is entered.

- Error Handling: Test error messages and notifications displayed to users when they encounter errors or submit incorrect data. Verify that the messages are clear, helpful, and provide guidance on resolving the issues.

2. Visual Testing:

- Layout and Alignment: Test the layout and alignment of UI elements, ensuring they are properly positioned, aligned, and consistent across different screens or resolutions.

- Typography: Verify the font styles, sizes, and colors used in the user interface, ensuring readability and adherence to design guidelines.

- Images and Icons: Test the display of images, icons, and graphics, ensuring they appear correctly and are not distorted or misaligned.

- Color and Theme: Validate the color scheme and theme of the UI, ensuring consistency, accessibility, and compliance with design guidelines.

- Responsive Design: Test the responsiveness of the UI across different devices and screen sizes to ensure that it adapts and displays correctly.

3. Usability Testing:

- User Interaction: Test the ease of use and intuitiveness of the UI, ensuring that users can perform tasks efficiently and without confusion.

- User Input: Validate the ease of entering data, selecting options, and interacting with forms, ensuring that the UI provides appropriate feedback and guidance.

- Accessibility: Test the accessibility features of the UI, such as support for assistive technologies, screen readers, keyboard navigation, and color contrast for visually impaired users.

- Performance: Assess the UI's performance in terms of loading times, responsiveness to user actions, and handling of large data sets or complex operations.

4. Cross-Browser and Cross-Platform Testing:

- Test the UI across different web browsers (e.g., Chrome, Firefox, Safari) and ensure consistent behavior and appearance.

- Validate the UI's compatibility and functionality on different platforms (e.g., Windows, macOS, iOS, Android).

5. Localization and Internationalization Testing:

- Test the UI with different languages, character sets, and locales to ensure proper translation, text display, and support for international users.

UI testing plays a crucial role in ensuring that the application's user interface meets the desired standards of functionality, usability, and visual appeal. By thoroughly testing the UI, you can enhance the user experience, identify and fix issues, and deliver a high-quality software product.

White-Box Testing

The goal for white box testing is to ensure that the internal components of a program are working properly. A common focus is on structural elements such as statements and branches. The tester develops test cases that exercise these structural elements to determine if defects exist in the program structure. The term exercise is used in this context to indicate that the target structural elements are executed when the test cases are run. By exercising all of the selected structural elements the tester hopes to improve the chances for detecting defects. Testers need a framework for deciding which structural elements to select as the focus of testing, for choosing the appropriate test data, and for deciding when the testing efforts are adequate enough to terminate the process with confidence that the software is working properly. Such a framework exists in the form of test adequacy criteria. Formally a test data adequacy criterion is a stopping rule [1,2]. Rules of this type can be used to determine whether or not sufficient testing has been carried out. The criteria can be viewed as representing minimal standards for testing a program. The application scope of adequacy criteria also includes:

(i) helping testers to select properties of a program to focus on during test;

(ii) helping testers to select a test data set for a program based on the selected properties;

(iii) supporting testers with the development of quantitative objectives for testing;

(iv) indicating to testers whether or not testing can be stopped for that program

A program is said to be adequately tested with respect to a given criterion if all of the target structural elements have been exercised according to the selected criterion. Using the selected adequacy criterion a tester can terminate testing when he/she has exercised the target structures, and have some confidence that the software will function in manner acceptable to the user. If a test data adequacy criterion focuses on the structural properties of a program it is said to be a program-based adequacy criterion. Program-based adequacy criteria are commonly applied in white box testing. They use either logic and control structures, data flow, program text, or faults as the focal point of an adequacy evaluation [1]. Other types of test data adequacy criteria focus on program specifications. These are called specification-based test data adequacy criteria. Finally, some test data adequacy criteria ignore both program structure and specification in the selection and evaluation of test data. An example is the random selection criterion [2]. Adequacy criteria are usually expressed as statements that depict the property, or feature of interest, and the conditions under which testing can be stopped (the criterion is satisfied). For example, an adequacy criterion that focuses on statement/branch properties is expressed as the following: A test data set is statement, or branch, adequate if a test set T for program P causes all the statements, or branches, to be executed respectively.

The concept of test data adequacy criteria, and the requirement that certain features or properties of the code are to be exercised by test cases, leads to an approach called "coverage analysis," which in practice is used to set testing goals and to develop and evaluate test data.

Under some circumstances, the planned degree of coverage may be less than 100% possibly due to the following:

• The nature of the unit —Some statements/branches may not be reachable. —The unit may be simple, and not mission, or safety, critical, and so complete coverage is thought to be unnecessary.

• The lack of resources —The time set aside for testing is not adequate to achieve 100% coverage. —There are not enough trained testers to achieve complete coverage for all of the units. —There is a lack of tools to support complete coverage. •

Other project-related issues such as timing, scheduling, and marketing constraints.