2 Testing Fundamentals

2.1 What is a Test Case?

A test case is a set of instructions on **HOW** to validate a particular test objective/target, which when followed will tell us if the expected behavior of the system is satisfied or not.

A test case has input, action and an expected response, to determine if a feature of an application is working correctly. Well written test cases can mean the difference between a well-tested and a poorly tested application.

An example of a software test case:

Title: Login Page – Authenticate user on Hotmail.com

Description: A user should be able to log in at hotmail.com.

Precondition: The user must have an email address and password that is previously registered.

Assumption: The browser supports hotmail.com

Test Steps:

- Navigate to hotmail.com
- Enter the email address of the registered user in the 'email' field.
- Enter the password of the registered user
- Click the 'Next' button.
- Click 'Log in'

Expected Result: The Hotmail inbox of the user should load.

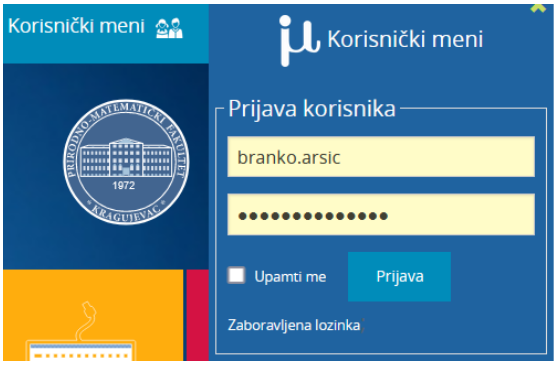
Test Scenario Vs Test Case

Test scenarios are rather vague and cover a wide range of possibilities. Testing is all about being very specific.

For a Test Scenario: Check Login Functionality there many possible test cases are:

Test Case 1: Check results on entering valid User Id & Password
Test Case 2: Check results on entering Invalid User ID & Password
Test Case 3: Check response when a User ID is Empty & Login Button is pressed
Test Case 4: and many more…

How to Write Test Cases in Manual Testing



Below is a format of a standard login Test cases example.

Test Case ID	Test Case Description	Test Steps	Test Data	Expected Results	Actual Results	Pass/Fail
TU01	Check student Login with valid Data	1. Go to site http://imi.pmf.kg.ac.rs 2. Enter Username 3. Enter Password 4. Click Submit	Username = branko.arsic Password = pass99	User should Login into an application	As Expected	Pass
TU02	Check student Login with invalid Data	1. Go to site http://imi.pmf.kg.ac.rs 2. Enter Username 3. Enter Password 4. Click Submit	Username = branko.arsic Password = qatest99	User should not Login into an application	As Expected	Pass

2.2 Test levels

Software testing levels are the different stages of the software development lifecycle where testing is conducted. There are four levels of software testing: **Unit >> Integration >> System >> Acceptance**.

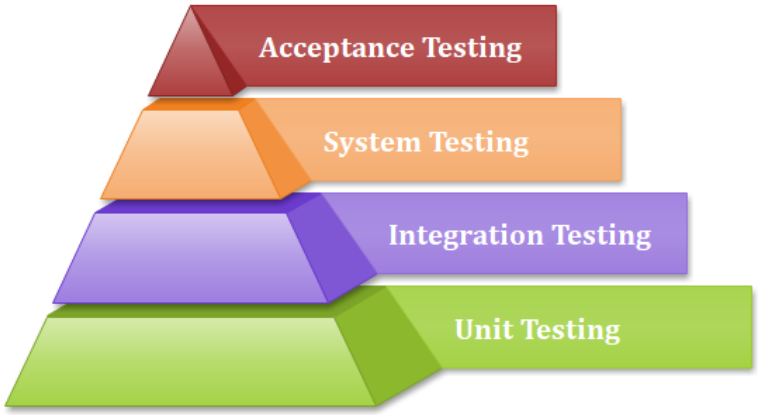


Figure 7: Different levels of testing

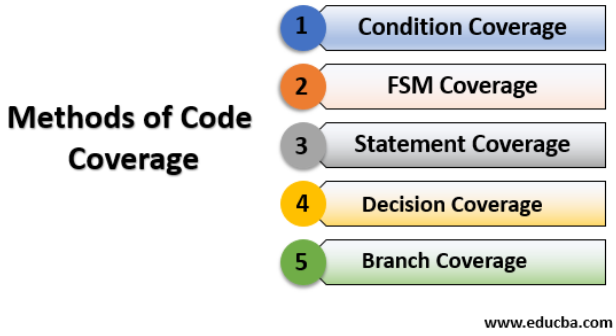
2.2.1 Component (Unit) testing

Component testing (also known as a unit or module testing) is the process of testing individual units or components of a software. It focuses on components that are separately testable.

Component testing is usually performed by a developer who writes different code units that could be related or unrelated to achieve a particular functionality. This usually entails writing unit tests which would call the methods in each unit and validate those when the required parameters are passed, and its return value is as expected.

Code coverage is an important part of unit testing where the test cases need to exist to cover the below three:

a) Line coverage
b) Code path coverage
c) Method coverage



Component testing may cover *functionality* (e.g. the correctness of calculations), *non-functional* characteristics (e.g. searching for memory leaks), *structural properties* (e.g. decision testing), and *changed-based testing*.

Functional testing: This involves verifying the functionality of the software against the stipulated requirements. Test data is used as input. We also check that the output given is as expected.

Non-functional testing: This involves a testing process to analyze how well the software works, for example, the number of users it can handle simultaneously.

Structural testing: This type of testing is based on the code's structure. For example, if a code is meant to calculate the average of even numbers in an array, then structure-based testing would be interested in the 'steps that lead to the average being calculated', rather than whether the final output is a correct numerical value.

Structural testing is carried out by the same people who write the code as they understand it best.

Change-based testing: This involves testing the effects of making changes to the code and then ensuring that the made changes have been implemented. It also ensures that the changes to the code do not break it.

Typical test objects for unit testing include:

- Components, units or modules
- Code and data structures
- Classes
- Database modules

Examples of typical defects and failures for unit testing include:

- Incorrect functionality (e.g., not as described in design specifications)
- Data flow problems
- Incorrect code and logic

2.2.2 Integration testing (Integration /API / Service Testing)

When the system relies on multiple functional modules that might individually work perfectly but have to work coherently when clubbed together to achieve an end to end scenario, validation of such scenarios is called Integration testing.

Integration testing focuses on interactions between components or systems.

- Component integration testing focuses on the interactions and interfaces between integrated components. Component integration testing is performed after component testing and is generally automated. In iterative and incremental development, component integration tests are usually part of the continuous integration process.
- System integration testing focuses on the interactions and interfaces between systems, packages, and microservices.

Typical test objects for integration testing include:

- Subsystems
- Databases
- Infrastructure
- Interfaces

- APIs
- Microservices

Component integration testing is often the responsibility of developers. System integration testing is generally the responsibility of testers.

The greater the scope of integration, the more difficult it becomes to isolate defects to a specific component or system, which may lead to increased risk and additional time for troubleshooting. This is One reason that continuous integration, where software is integrated on a component-by-component basis (i.e., functional integration), has become common practice. Such continuous integration often includes automated regression testing, ideally at multiple test levels.

2.2.3 System testing

System testing focuses on the behavior and capabilities of a whole system or product, often considering the end-to-end tasks

- Validating that the system is complete and will work as expected
- Preventing defects from escaping to higher test levels or production

Typical test objects for system testing include:

- Applications
- Hardware/software systems
- Operating systems
- System under test (SUT)
- System configuration and configuration data

Examples of typical defects and failures for system testing include:

- Incorrect calculations
- Incorrect or unexpected system functional or non-functional behavior
- Incorrect control and/or data flows within the system
- Failure to properly and completely carry out end-to-end functional tasks
- Failure of the system to work properly in the production environment(s)
- Failure of the system to work as described in system and user manuals

System testing should focus on the overall, end-to-end behavior of the system as a whole, both functional and non-functional.

2.2.4 Acceptance testing

Acceptance testing, like system testing, typically focuses on the behavior and capabilities of a whole system or product. Objectives of acceptance testing include:

- Establishing confidence in the quality of the system as a whole
- Validating that the system is complete and will work as expected
- Verifying that functional and non-functional behaviors of the system are as specified

Common forms of acceptance testing include the following:

- User acceptance testing
- Operational acceptance testing
- Contractual and regulatory acceptance testing
- Alpha and beta testing

2.2.4.1 UAT

User Acceptance Testing (UAT) or end-user testing, is defined as testing the software by the user or client to determine whether it can be accepted or not. This is the final testing performed once the functional, system and regression testing are completed.

The main purpose of this testing is to validate the software against the business requirements. This validation is carried out by the end-users who are familiar with the business requirements. UAT, alpha and beta testing are different types of acceptance testing.

As the user acceptance test is the last testing that is carried out before the software goes live, obviously this is the last chance for the customer to test the software and measure if it is fit for the purpose. This is typically the last step before the product goes live or before the delivery of the product is accepted. This is performed after the product itself is thoroughly tested (i.e. after system testing).

2.2.4.2 OAT

Operational Acceptance Testing (OAT) is a type of software testing that is performed to conduct operational pre-release of software, system or application to check the quality of it. Operational Acceptance Testing is a very usual software testing whose type is non-functional and it is mainly used in software development and software maintenance projects.

Operational Acceptance Testing mainly focuses on the operational readiness of the software and to become part of the production environment. Functional testing in operational acceptance testing is limited to the tests required to verify the non-functional aspects of the system.

Operational Acceptance Testing is also known as Operational Readiness Testing (ORT) or Operations Readiness and Assurance Testing (ORAT).

The objective of Operational Acceptance Testing is:

- To determine the resiliency of the software.
- To determine the recovering ability of the software.
- To determine the integrity of the software.
- To determine software can be deployed on a network on ITIL standards. (ITIL is an acronym for Information Technology Infrastructure Library, is a set of detailed practices for IT service management)
- To determine the supportability of the software.

Types of Operational Acceptance Testing:

- Load Testing

- Performance Testing
- Installation Testing
- Backup and Restore Testing
- Security Testing
- Recovery Testing

2.2.4.3 ALPHA AND BETA TESTING

Alpha and Beta testing are the Customer Validation methodologies (Acceptance Testing types) that help in building confidence to launch the product, and thereby results in the success of the product in the market.

Even though they both rely on real users and different team feedback, they are driven by distinct processes, strategies, and goals. These two types of testing together increase the success and lifespan of a product in the market. These phases can be adapted to Consumer, Business, or Enterprise products.

2.3 Test Types
(Functional, Non-functional, White-box, Change-related, Static Testing)

Software Testing Type is a classification of different testing activities into categories, each having a defined test objective, test strategy, and test deliverables. The goal of having a testing type is to validate the Application Under Test (AUT) for the defined Test Objective.

There are many different types of testing of Software Testing such as Functional Testing, Non-Functional Testing, Automation Testing, Agile Testing, and their sub-types, etc.

For instance, the goal of *Accessibility testing* is to validate the AUT to be accessible by disabled people. So, if your Software solution must be disabled friendly, you check it against Accessibility Test Cases.

A list of 100 types of Software Testing Types along with definitions can be found here.

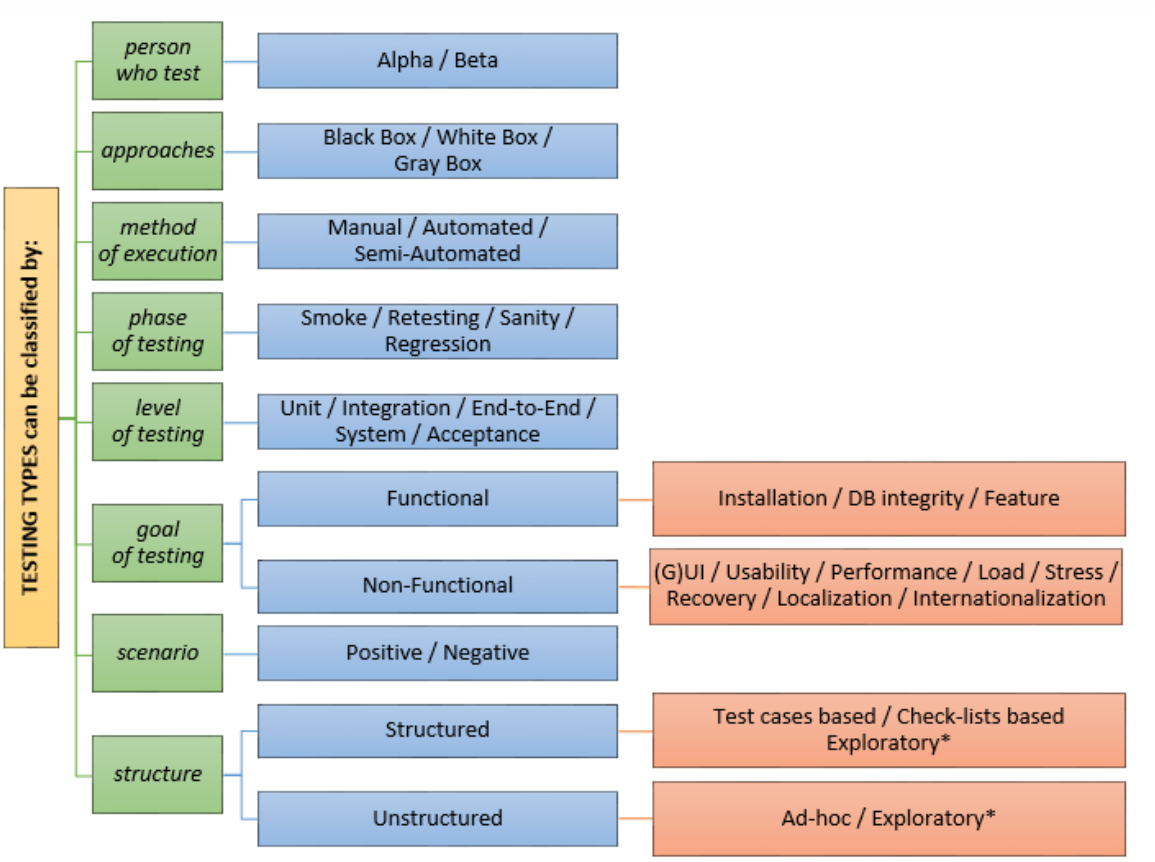


Figure 8: Classification of testing types

2.3.1 Functional testing

Functional Testing is defined as a type of testing that verifies that each function of the software application operates in conformance with the requirement specification. This testing mainly involves black box testing, and it is not concerned about the source code of the application.

Each functionality of the system is tested by providing appropriate input, verifying the output and comparing the actual results with the expected results.

This testing involves checking of User Interface, APIs, Database, security, client/server applications and functionality of the Application Under Test. Testing can be done either manually or using automation.

We usually focus on:

- Mainline functions - Testing the main functions of an application
- Basic Usability - It involves basic usability testing of the system. It checks whether a user can freely and intuitively navigate through the screens without any difficulties.
- Accessibility - Checks the accessibility of the system for the user
- Error Conditions - Usage of testing techniques to check for error conditions. It checks whether suitable error messages are displayed.

Functional Testing types include:

- Unit Testing
- Integration Testing
- System Testing
- Sanity Testing (subset of Regression Testing)
- Smoke Testing
- Interface Testing
- Regression Testing
- Beta/Acceptance Testing

2.3.2 Non-functional testing

Non-functional testing is defined as a type of Software testing to check non-functional aspects (performance, security, reliability, etc.) of a software application. It is designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing.

Non-functional Testing types include:

- Performance Testing
- Load Testing
- Stress Testing
- Volume Testing
- Security Testing
- Compatibility Testing
- Install Testing
- Recovery Testing
- Reliability Testing
- Compliance Testing
- Localization Testing

Table below shows some of the main differences between functional and non-functional testing:

Functional testing	Non-Functional testing
Functional testing is performed using the functional specification provided by the client and verifies the system against the functional requirements.	Non-Functional testing checks the Performance, reliability, scalability and other non-functional aspects of the software system.
Functional testing is executed first	Non-functional testing should be performed after functional testing
Manual or automation tools can be used for functional testing	Using tools will be effective for this testing
Business requirements are the inputs to functional testing	Performance parameters like speed, scalability are inputs to non-functional testing.
Functional testing describes what the product	Nonfunctional testing describes how good the

does	product works
Easy to do Manual Testing	Tough to do Manual Testing
Examples of Functional testing are: Unit Smoke Testing Sanity Testing Integration White box testing Black Box testing User Acceptance testing Regression Testing	Examples of Non-functional testing are: Performance Load Testing Volume Testing Stress Testing Security Testing Installation Testing Penetration Testing Compatibility Testing Migration Testing

2.3.3 Black-box Testing

Black box testing Is defined as a testing technique in which functionality of the Application Under Test (AUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software. This type of testing is based entirely on software requirements and specifications.

In Black-box testing, we just focus on inputs and output of the software system without bothering about internal knowledge of the software program.

Here are the generic steps followed to carry out any type of Black-box testing:

- Initially, the requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenarios) to check whether SUT processes them correctly. Also, some invalid inputs (negative test scenarios) are chosen to verify that the SUT can detect them.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if any are fixed and re-tested.

2.3.4 White-box Testing

White Box testing is defined as the testing of a software solution's internal structure, design, and coding. In this type of testing, the code is visible to the tester. It focuses primarily on verifying the flow of inputs and outputs through the application, improving design and usability, strengthening security. It is usually performed by developers.

White box testing involves the testing of the software code for the following:

- Internal security holes
- Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- Expected output
- The functionality of conditional loops
- Testing of each statement, object, and function on an individual basis

There are automated tools available to perform Code coverage analysis.

Using Statement and Branch coverage you generally attain 80-90% code coverage which is sufficient.

Advantages of White Box Testing

- Code optimization by finding hidden errors.
- White box tests cases can be easily automated.
- Testing is more thorough as all code paths are usually covered.
- Testing can start early in SDLC even if a GUI is not available.

Disadvantages of White-Box Testing

- White box testing can be quite complex and expensive.
- Developers who usually execute white box test cases detest it. The white box testing by developers is not detailed can lead to production errors.
- White box testing requires professional resources, with a detailed understanding of programming and implementation.
- White box testing is time-consuming, bigger programming applications take the time to test fully.

2.3.5 Change-related testing

This is testing following a change or fix to the software or environment. This should involve two types of test:

- **Re-testing or confirmation testing**, that reruns test cases that previously failed in order to verify the success of corrective actions.
- **Regression testing** of a previously tested program following modification to ensure that effects have not been introduced or uncovered in unchanged areas of the software as a result of the changes made. An estimated 20% to 50% of changes introduce new defects.

It may be performed at all test levels and types, including functional, non-functional and structural testing, regression should take place at all stages of testing after a functional improvement or repair. Retesting and regression should be repeatable, so they can re-run every

time there is a change or fix. It's essential to automate both re-testing and regression tests. Automation is not a full substitute for manual testing as not 100% automation testing can be achieved, and it shouldn't.

2.3.6 Static testing

Static Testing is defined as a software testing technique by which we can check the defects in software without actually executing it. Under Static Testing, code is not executed. Rather it manually checks the code, requirement documents, and design documents to find errors. Hence, the name "static". Under Dynamic Testing, a code is executed. It checks for functional behavior of software system, memory/CPU usage and overall performance of the system. Hence the name "Dynamic.

Static testing is done to avoid errors at an early stage of development as it is easier to find sources of failures then failures themselves.

Static testing helps to find errors that may not be found by Dynamic Testing.

The two main types of static testing techniques are

- **Manual examinations:** Manual examinations include analysis of code done manually, also known as reviews.
- **Automated analysis using tools:** Automated analysis are basically static analysis which is done using tools.

Static testing is to find defects as early as possible. It's not a substitute for dynamic testing, both find a different type of defects. Reviews not only help to find defects but also understand missing requirements, design defects, non-maintainable code.

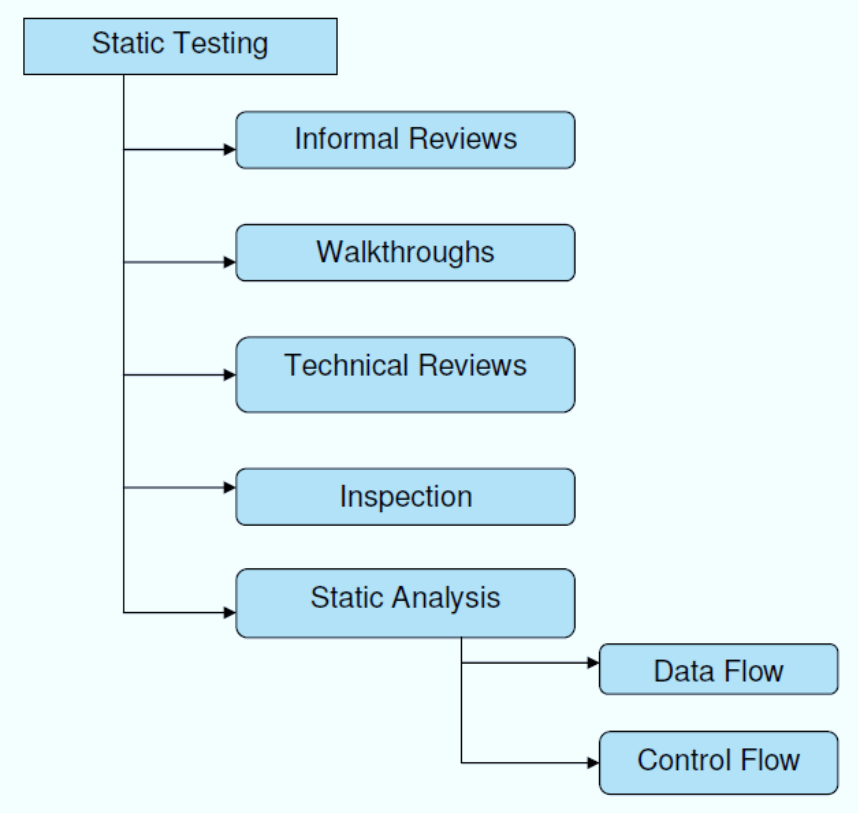


Figure 9: static testing techniques

2.4 Test Techniques

Software testing techniques help you design better test cases. Since exhaustive testing is not possible, Manual Testing Techniques help reduce the number of test cases to be executed while increasing test coverage. They help identify test conditions that are otherwise difficult to recognize.

Test design techniques can be Static or Dynamic. Static techniques test software without executing while Dynamic involves the execution of the software of a component system.

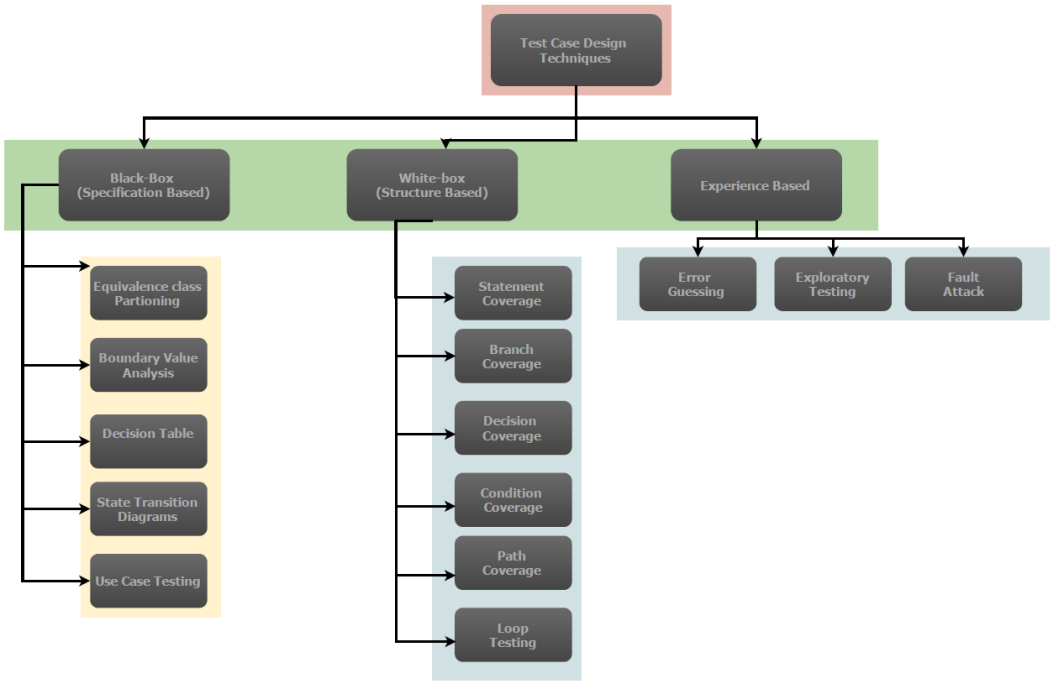


Figure 10: Dynamic test techniques

2.4.1 Boundary Value Analysis (BVA)

Boundary value analysis is based on testing at the boundaries between partitions. It includes maximum, minimum, inside or outside boundaries, typical values and error values.
It is generally seen that a large number of errors occur at the boundaries of the defined input values rather than the center. It is also known as BVA and gives a selection of test cases that exercise bounding values.
This black box testing technique complements equivalence partitioning. This software testing technique base on the principle that, if a system works well for these particular values then it will work perfectly well for all values which come between the two boundary values.

Guidelines for Boundary Value analysis

If an input condition is restricted between values x and y, then the test cases should be designed with values x and y as well as values which are above and below x and y.

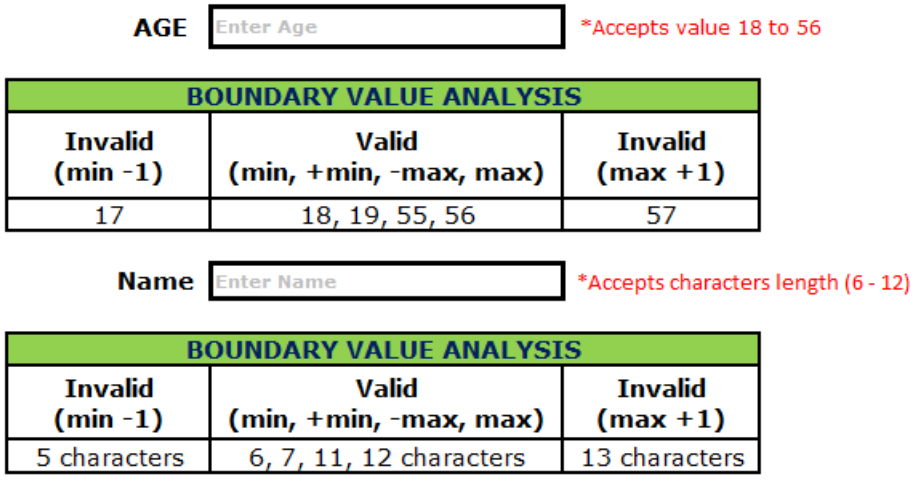


Figure 11: Boundary value Analysis sampling

If an input condition is a large number of values, the test case should be developed which need to exercise the minimum and maximum numbers. Here, values above and below the minimum and maximum values are also tested.
Apply guidelines 1 and 2 to output conditions. It gives an output that reflects the minimum and the maximum values expected. It also tests the below or above values.

2.4.2 Equivalence Class Partitioning

Equivalent Class Partitioning is a black box technique (code is not visible to tester) which can be applied to all levels of testing like unit, integration, system, etc. It allows you to divide a set of test conditions into a partition that should be considered the same. This software testing method divides the input domain of a program into classes of data from which test cases should be designed.

The concept behind this technique is that the test case of a representative value of each class is equal to a test of any other value of the same class. It allows you to Identify valid as well as invalid equivalence classes.

Example: Demonstrating Equivalence Class Partitioning

a) Test field which accepts age:

Enter age _____

EQUIVALENCE PARTITIONING		
Invalid	Valid	Invalid
<= 17	18-65	>= 66
Partition 1	Partition 2	Partition 3

Table 1: Equivalence partitioning class example for valid and invalid age samples

The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing (e.g. 15, 45 and 75). The hypothesis behind this technique is that if one condition/value in a partition passes all others will also pass. Likewise, if one condition in a partition fails, all other conditions in that partition will fail.

b) If a customer pays up to $ 500 to their account, they will receive a 5% discount, if they make a payment of $ 500 to $ 1000 they will receive a 15% discount. Payments of $1000 and up do not receive a discount.

Invalid	Valid for 5% discount	Valid for 15% discount	Invalid
$0	$1 $499	$500 $999	$1000
Partition 1	Partition 2	Partition 3	Partition 4

Table 2: Equivalence partitioning class example for valid and invalid discount samples

2.4.3 Decision Table Based Testing

A decision table is also known as to Cause-Effect table. This software testing technique is used for functions that respond to a combination of inputs or events. For example, a submit button should be enabled if the user has entered all required fields.

The first task is to identify functionalities where the output depends on a combination of inputs. If there are large input set of combinations, then divide it into smaller subsets which are helpful for managing a decision table.

For every function, you need to create a table and list down all types of combinations of inputs and their respective outputs. This helps to identify a condition that is overlooked by the tester.

Following are steps to create a decision table:

- Enlist the inputs in rows
- Enter all the rules in the column
- Fill the table with the different combination of inputs
- In the last row, note down the output against the input combination.

Example: A submit button in a contact form is enabled only when all the inputs are entered by the end user.

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Input								
Name	F	T	F	T	F	T	F	T
Email	F	F	T	T	F	F	T	T
Message	F	F	F	F	T	T	T	T
Output								
Submit	F	F	F	F	F	F	F	T

Table 3: Decision table sample for contact form

2.4.4 State Transition

In the State Transition technique changes in input conditions change the state of the Application Under Test (AUT). This testing technique allows the tester to test the behavior of an AUT. The tester can perform this action by entering various input conditions in a sequence. In the State transition technique, the testing team provides positive as well as negative input test values for evaluating the system behavior.

Guideline for State Transition:

- State transition should be used when a testing team is testing the application for a limited set of input values.
- The technique should be used when the testing team wants to test a sequence of events that happen in the application under test.

In the following example, if the user enters a valid password in any of the first three attempts the user will be able to log in successfully. If the user enters the invalid password in the first or second try, the user will be prompted to re-enter the password. When the user enters the password incorrectly 3rd time, the action has taken, and the account will be blocked.

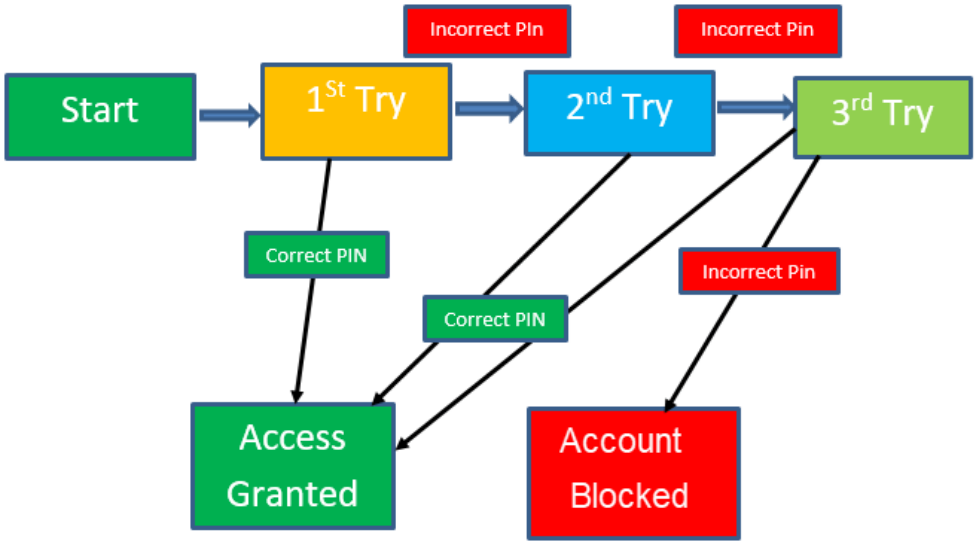


Figure 12: State transition diagram for PIN authentication

In this diagram when the user gives the correct PIN number, he or she is moved to Access granted State. Following Table is created based on the diagram above:

State Transition Table

	Correct PIN	Incorrect PIN
S1) Start	**S5**	**S2**
S2) 1st attempt	**S5**	**S3**
S3) 2nd attempt	**S5**	**S4**
S4) 3rd attempt	**S5**	**S6**
S5) Access Granted	**–**	**–**
S6) Account blocked	**–**	**–**

Table 4: State transition for PIN authentication

In the above-given table when the user enters the correct PIN, the state is transitioned to Access granted. And if the user enters an incorrect password, he or she is moved to the next state. If he does the same 3rd time, he will reach the account blocked state.

2.4.5 Error Guessing

Error Guessing is a software testing technique based on guessing the error which can prevail in the code. The technique is heavily based on the experience where the test analysts use their

experience to guess the problematic part of the testing application. Hence, the test analysts must be skilled and experienced for better error guessing.

The technique counts a list of possible errors or error-prone situations. Then tester writes a test case to expose those errors. To design test cases based on this software testing technique, the analyst can use the past experiences to identify the conditions.

Guidelines for Error Guessing:

- The test should use the previous experience of testing similar applications
- Understanding of the system under test
- Knowledge of typical implementation errors
- Remember previously troubled areas
- Evaluate Historical data & Test results

Error Guessing technique requires skilled and experienced tester. It is mainly based on intuition and experience.

Examples:

- Try to enter blank space input fields
- Try to enter a value greater than expected
- Press submit entering values
- Try to send invalid parameter

Conclusion

- Software testing Techniques allow you to design better cases. There are five primarily used techniques.
- Boundary value analysis is testing at the boundaries between partitions.
- Equivalent Class Partitioning allows you to divide set of test condition into a partition which should be considered the same.
- Decision Table software testing technique is used for functions which respond to a combination of inputs or events.
- In State Transition technique changes in input conditions change the state of the Application Under Test (AUT)
- Error guessing is a software testing technique which is based on guessing the error which can prevail in the code.