UNIT-IV

➢ Strategic Approach to Software Testing

Testing begins at the component level and works outward toward the integration of the entire Computer-based system.

Different testing techniques are appropriate at different points in time

The developer of the software conducts testing and may be assisted by independent test groups for large projects.

The role of the independent tester is to remove the conflict of interest inherent when the builder is testing his or her own product.

Testing and debugging are different activities.

Debugging must be accommodated in any testing strategy.

Make a distinction between verification (are we building the product right?) and validation (are we building the right product?)

➢ Strategic Testing Issues

Specify product requirements in a quantifiable manner before testing starts

Specify testing objectives explicitly

Identify the user classes of the software and develop a profile for each.

Develop a test plan that emphasizes rapid cycle testing

Build robust software that is designed to test itself (e.g. uses anitbugging)

Use effective formal reviews as a filter prior to testing

Conduct formal technical reviews to assess the test strategy and test cases

Characteristics of Testing Strategies

- The process of investigating and checking a program to find whether there is an error or not and does it fulfill the requirements or not is called testing.
- When the number of errors found during the testing is high, it indicates that the testing was good and is a sign of good test case.
- Finding an unknown error that's discovered yet is a sign of a successful and a good test case.

The main objective of software testing is to design the tests in such a way that it systematically finds different types of errors without taking much time and effort so that less time is required for the development of the software.

Software Testing Myths and Facts: Just as every field has its myths, so does the field of Software Testing. Software testing myths have arisen primarily due to the following:

i. Lack of authoritative facts.
ii. Evolving nature of the industry.
iii. General flaws in human logic.

Some of the myths are explained below, along with their related facts:

1. MYTH: Quality Control = Testing.
FACT: Testing is just one component of software quality control. Quality Control includes other activities such as Reviews.

2.MYTH: The objective of Testing is to ensure a 100% defect- free product.
FACT: The objective of testing is to uncover as many defects as possible. Identifying all defects and getting rid of them is impossible.

3.MYTH: Testing is easy.
FACT: Testing can be difficult and challenging (sometimes, even more so than coding).

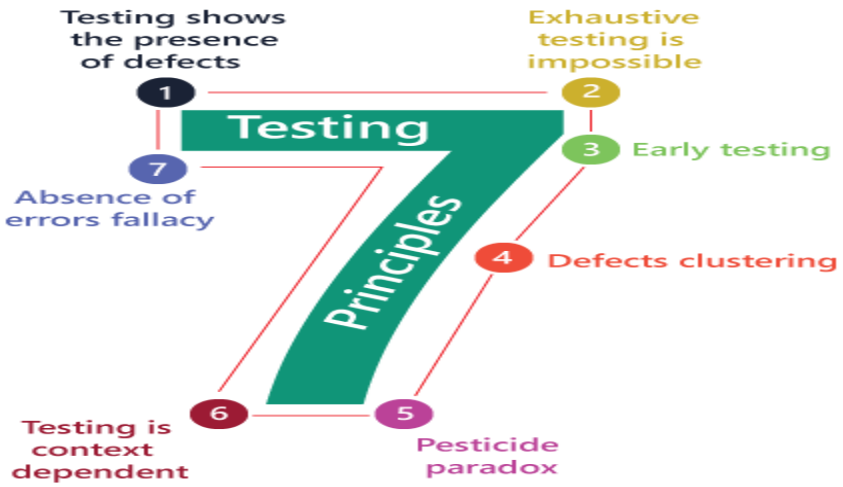
4. MYTH: Anyone can test.
FACT: Testing is a rigorous discipline and requires many kinds of skills.

5. MYTH: There is no creativity in testing.
   FACT: Creativity can be applied when formulating test approaches, when designing tests, and even when executing tests.
6. MYTH: Automated testing eliminates the need for manual testing.
   FACT: 100% test automation cannot be achieved. Manual Testing, to some level, is always necessary.
7. MYTH: When a defect slips, it is the fault of the Testers.
   FACT: Quality is the responsibility of all members/stakeholders, including developers, of a project.
8. MYTH: Software Testing does not offer opportunities for career growth.
   FACT: Gone are the days when users had to accept whatever product was dished to them; no matter what the quality.
   With the abundance of competing software and increasingly demanding users, the need for software testers to ensure high quality will continue togrow.

Software Testing Principles

Software testing is a procedure of implementing software or the application to identify the defects or bugs. For testing an application or software, we need to follow some principles to make our product defects free, and that also helps the test engineers to test the software with their effort and time. Here, in this section, we are going to learn about the seven essential principles of software testing.
Let us see the seven different testing principles, one by one:

- Testing shows the presence of defects
- Exhaustive Testing is not possible
- Early Testing
- Defect Clustering
- Pesticide Paradox
- Testing is context-dependent
- Absence of errors fallacy



Testing shows the presence of defects

The test engineer will test the application to make sure that the application is bug or defects free. While doing testing, we can only identify that the application or software has any errors. The primary purpose of doing testing is to identify the numbers of unknown bugs with the help of various methods and testing techniques because the entire test should be traceable to the customer requirement, which means that to find any defects that might cause the product failure to meet the client's needs.

By doing testing on any application, we can decrease the number of bugs, which does not mean that the application is defect-free because sometimes the software seems to be bug-free while performing multiple types of testing on it. But at the time of deployment in the production server, if the end-user encounters those bugs which are not found in the testing process.6

Exhaustive Testing is not possible

Sometimes it seems to be very hard to test all the modules and their features with effective and non-effective combinations of the inputs data throughout the actual testing process.

Hence, instead of performing the exhaustive testing as it takes boundless determinations and most of the hard work is unsuccessful. So we can complete this type of variations according to the importance of the modules because the product timelines will not permit us to perform such type of testing scenarios.

Early Testing

Here early testing means that all the testing activities should start in the early stages of the software development life cycle's **requirement analysis stage** to identify the defects because if we find the bugs at an early stage, it will be fixed in the initial stage itself, which may cost us very less as compared to those which are identified in the future phase of the testing process.

To perform testing, we will require the requirement specification documents; therefore, if the requirements are defined incorrectly, then it can be fixed directly rather than fixing them in another stage, which could be the development phase.

Defect clustering

The defect clustering defined that throughout the testing process, we can detect the numbers of bugs which are correlated to a small number of modules. We have various reasons for this, such as the modules could be complicated; the coding part may be complex, and so on.

These types of software or the application will follow the **Pareto Principle**, which states that we can identify that approx. Eighty percent of the complication is present in 20 percent of the modules. With the help of this, we can find the uncertain modules, but this method has its difficulties if the same tests are performing regularly, hence the same test will not able to identify the new defects.

Pesticide paradox

This principle defined that if we are executing the same set of test cases again and again over a particular time, then these kinds of the test will not be able to find the new bugs in the software or the application. To get over these pesticide paradoxes, it is very significant to review all the test cases frequently. And the new and different tests are necessary to be written for the implementation of multiple parts of the application or the software, which helps us to find more bugs.

Testing is context-dependent

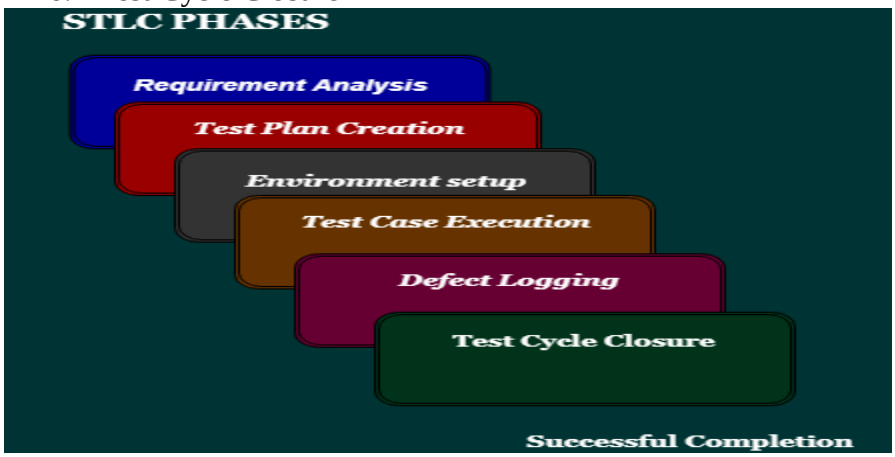
Testing is a context-dependent principle states that we have multiple fields such as e-commerce websites, commercial websites, and so on are available in the market. There is a definite way to test the commercial site as well as the e-commerce websites because every application has its own needs, features, and functionality. To check this type of application, we will take the help of various kinds of testing, different technique, approaches, and multiple methods. Therefore, the testing depends on the context of the application.

Absence of errors fallacy

Once the application is completely tested and there are no bugs identified before the release, so we can say that the application is 99 percent bug-free. But there is the chance when the application is tested beside the incorrect requirements, identified the flaws, and fixed them on a given period would not help as testing is done on the wrong specification, which does not apply to the client's requirements. The absence of error fallacy means identifying and fixing the bugs would not help if the application is impractical and not able to accomplish the client's requirements and needs

Software testing life cycle contains the following steps:

1. Requirement Analysis
2. Test Plan Creation
3. Environment setup
4. Test case Execution
5. Defect Logging
6. Test Cycle Closure



➢ Test strategies for Object-Oriented Software

Introduction:

i. Introduction: The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span.
ii. Although this fundamental objective remains unchanged for object oriented software.
iii. The nature of object-oriented software changes both testing strategy and testing tactics (Plan).

Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects.

This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data.

An encapsulated class is usually the focus of unit testing.

However, operations (methods) within the class are the smallest testable units. Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes.

Class testing for OO software is the equivalent of unit testing for conventional software.

Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface,

Class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

Integration Testing in the OO Context

Different strategies for integration testing of OO systems.

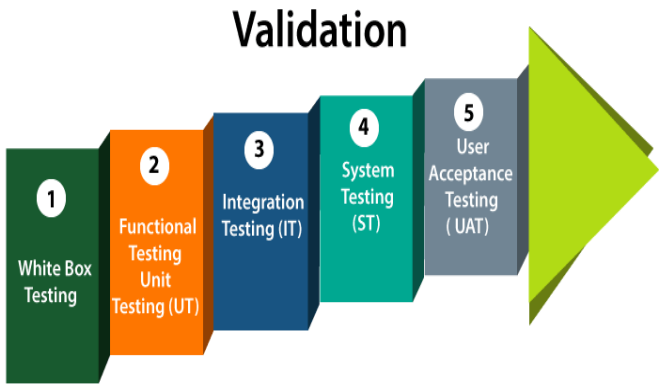
i. Thread-based testing
ii. use-based testing
iii. Cluster testing

- The first, thread-based testing, integrates the set of classes required to respond to one input or event for the system.
- Each thread is integrated and tested individually.
- Regression testing is applied to ensure that no side effects occur.
- The second integration approach, use-based testing, begins the construction of the system by testing those classes (called independent classes) that use very few (if any) server classes.
- After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested.
- This sequence of testing layers of dependent classes continues until the entire system is constructed.

Cluster testing

Cluster testing is one step in the integration testing of OO software.

Here, a cluster of collaborating classes is exercised by designing test cases that attempt to uncover errors in the collaborations.



Validation testing

Validation testing is testing where tester performed functional and non-functional testing. Here **functional testing** includes Unit Testing (UT), Integration Testing (IT) and System Testing (ST), and **non-functional** testing includes User acceptance testing (UAT).

Validation testing is also known as dynamic testing, where we are ensuring that **"we have developed the product right."** And it also checks that the software meets the business needs of the client.

Unit Testing

Unit testing involves the testing of each unit or an individual component of the software application. It is the first level of functional testing. The aim behind unit testing is to validate unit components with its performance.

A unit is a single testable part of a software system and tested during the development phase of the application software.

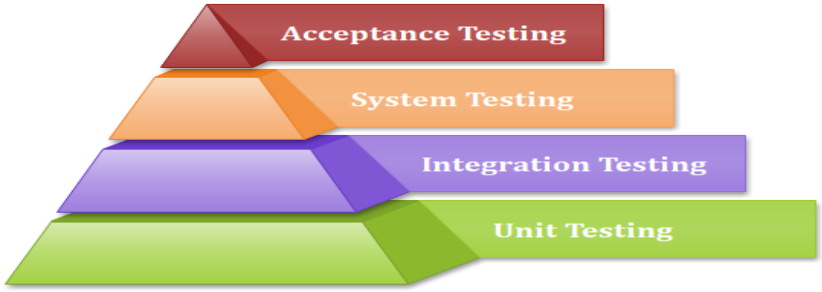
The purpose of unit testing is to test the correctness of isolated code. A unit component is an individual function or code of the application. White box testing approach used for unit testing and usually done by the developers.

Whenever the application is ready and given to the Test engineer, he/she will start checking every component of the module or module of the application independently or one by one, and this process is known as **Unit testing** or **components testing**.t tests for services, in minutes

Why Unit Testing?

In a testing level hierarchy, unit testing is the first level of testing done before integration and other remaining levels of the testing. It uses modules for the testing process which reduces the dependency

of waiting for Unit testing frameworks, stubs, drivers and mock objects are used for assistance in unit testing.



Generally, **the** software goes under four level of testing: Unit Testing, Integration Testing, System Testing, and Acceptance Testing but sometimes due to time consumption software testers does minimal unit testing but skipping of unit testing may lead to higher defects during Integration Testing, System Testing, and Acceptance Testing or even during Beta Testing which takes place after the completion of software application.

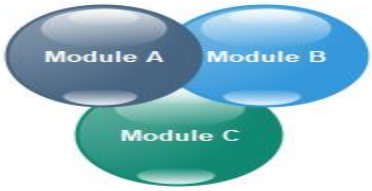
Integration testing

Integration testing is the second level of the software testing process comes after unit testing. In this testing, units or individual components of the software are tested in a group. The focus of the integration testing level is to expose defects at the time of interaction between integrated components or units.

Unit testing uses modules for testing purpose, and these modules are combined and tested in integration testing. The Software is developed with a number of software modules that are coded by different coders or programmers. The goal of integration testing is to check the correctness of communication among all the modules.

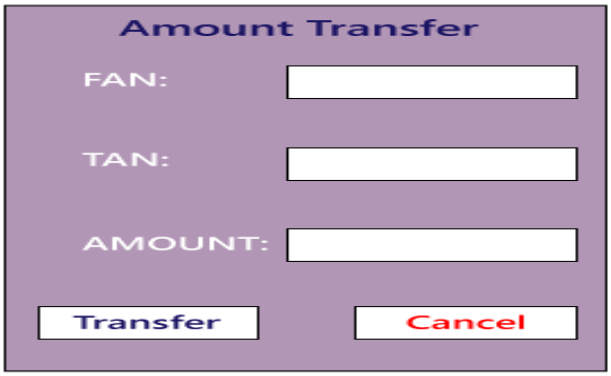
Once all the components or modules are working independently, then we need to check the data flow between the dependent modules is known as **integration testing**.

Tested in Unit Testing



Under Integration Testing

Let us see one sample example



of a banking application, as we can see in the below image of amount transfer.

- First, we will login as a user **P** to amount transfer and send Rs200 amount, the confirmation message should be displayed on the screen as **amount transfer successfully**. Now logout as

P and login as user **Q** and go to amount balance page and check for a balance in that account = Present balance + Received Balance. Therefore, the integration test is successful.

- Also, we check if the amount of balance has reduced by Rs200 in P user account.
- Click on the transaction, in P and Q, the message should be displayed regarding the data and time of the amount transfer.

Guidelines for Integration Testing

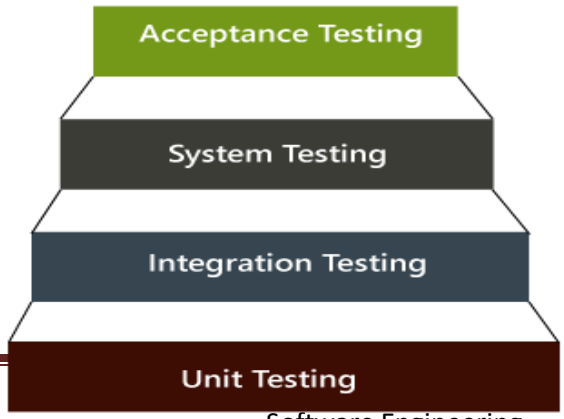
- We go for the integration testing only after the functional testing is completed on each module of the application.
- We always do integration testing by picking module by module so that a proper sequence is followed, and also we don't miss out on any integration scenarios.
- First, determine the test case strategy through which executable test cases can be prepared according to test data.
- Examine the structure and architecture of the application and identify the crucial modules to test them first and also identify all possible scenarios.
- Design test cases to verify each interface in detail.
- Choose input data for test case execution. Input data plays a significant role in testing.
- If we find any bugs then communicate the bug reports to developers and fix defects and retest.
- Perform **positive and negative integration testing**.

➢ System Testing

System Testing includes testing of a fully integrated software system. Generally, a computer system is made with the integration of software (any software is only a single element of a computer system). The software is developed in units and then interfaced with other software and hardware to create a complete computer system. In other words, a computer system consists of a group of software to perform the various tasks, but only software cannot perform the task; for that software must be interfaced with compatible hardware. System testing is a series of different type of tests with the purpose to exercise and examine the full working of an integrated software computer system against requirements.

To check the end-to-end flow of an application or the software as a user is known as **System testing**. In this, we navigate (go through) all the necessary modules of an application and check if the end features or the end business works fine, and test the product as a whole system.

It is **end-to-end testing** where the testing environment is similar to the production environment.



here are four levels of software testing
: unit testing
, integration testing
, system testing and acceptance testing
, all are used for the testing purpose. Unit Testing used to test a single software; Integration Testing used to test a group of units of software, System

Testing used to test a whole system and Acceptance Testing used to test the acceptability of business requirements. Here we are discussing system testing which is the third level of testing levels of India | List of Prime Minister **Hierarchy of Testing Levels**

There are mainly two widely used methods for software testing, one is **White box testing** which uses internal coding to design test cases and another is black box testing which uses GUI or user perspective to develop test cases?

- White box testing
- Black box testing

System testing falls under Black box testing as it includes testing of the external working of the software. Testing follows user's perspective to identify minor defects.

System Testing includes the following steps.

- Verification of input functions of the application to test whether it is producing the expected output or not.
- Testing of integrated software by including external peripherals to check the interaction of various components with each other.
- Testing of the whole system for End to End testing.
- Behavior testing of the application via a user's experience

Example of System testing

Suppose we open an application, let say **www.rediff.com**, and there we can see that an advertisement is displayed on the top of the homepage, and it remains there for a few seconds before it disappears. These types of Ads are done by the Advertisement Management System (AMS). Now, we will perform system testing for this type of field.

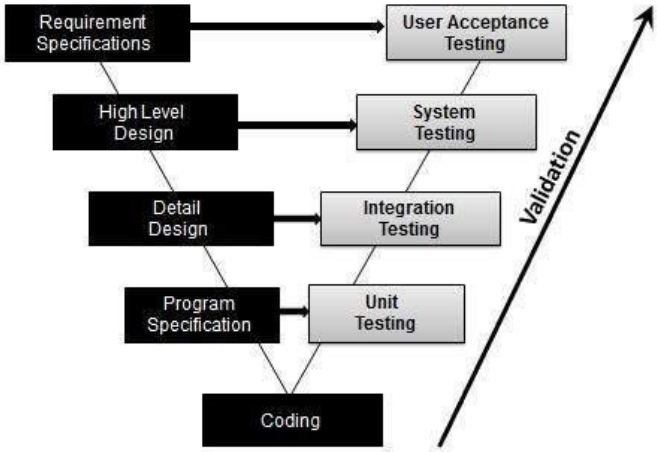
The below application works in the following manner:

- Let's say that Amazon wants to display a promotion ad on January 26 at precisely 10:00 AM on the Rediff's home page for the country India.
- Then, the sales manager logs into the website and creates a request for an advertisement dated for the above day.
- He/she attaches a file that likely an image files or the video file of the AD and applies.
- The next day, the AMS manager of Rediffmail login into the application and verifies the awaiting Ad request.
- The AMS manager will check those Amazons ad requests are pending, and then he/she will check if the space is available for the particular date and time.
- If space is there, then he/she evaluate the cost of putting up the Ad at 15$ per second, and the overall Ad cost for 10 seconds is approximate 150$.
- The AMS manager clicks on the payment request and sends the estimated value along with the request for payment to the Amazon manager.
- Then the amazon manager login into the Ad status and confirms the payment request, and he/she makes the payment as per all the details and clicks on the **Submit** and **Pay**
- As soon as Rediff's AMs manager gets the amount, he/she will set up the Advertisement for the specific date and time on the Rediffmail's home page.

➢ Software Testing - Validation Testing

The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.It answers to the question, Are we building the right product?



Validation Testing

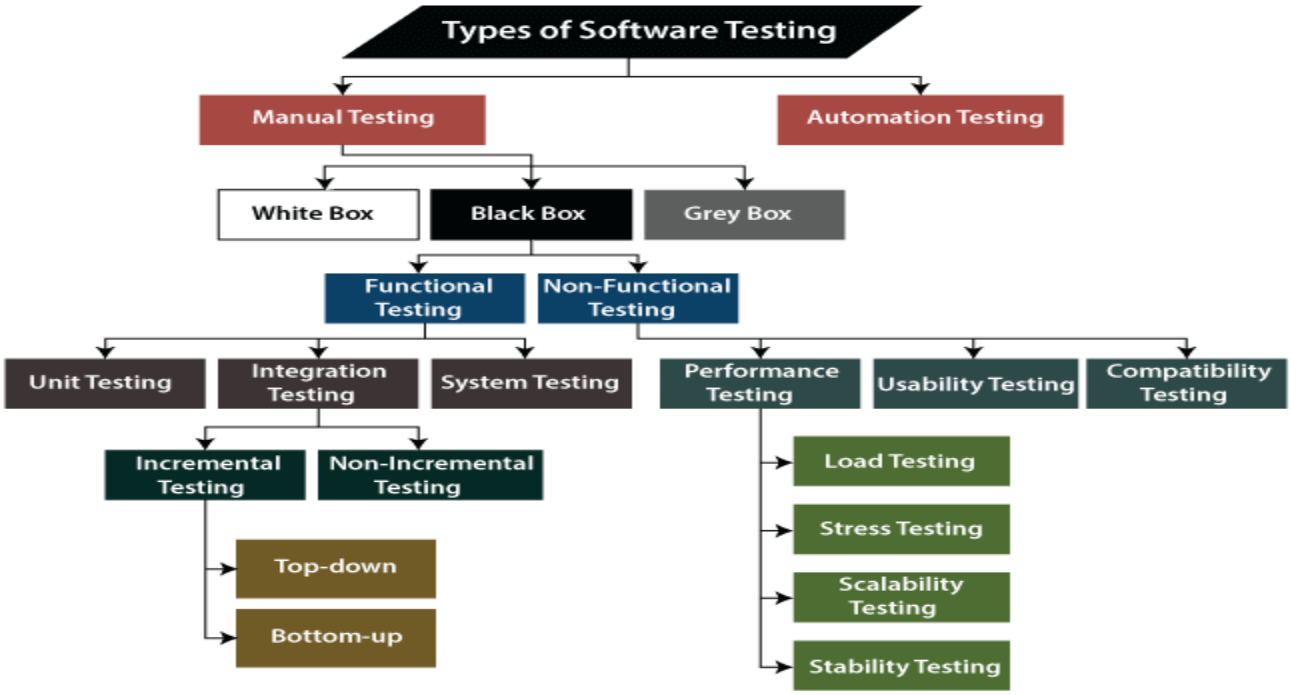
Validation testing can be best demonstrated using V-Model. The Software/product under test is evaluated during this type of testing.

Activities:

- Unit Testing
- Integration Testing
- System Testing
- User Acceptance Testing

➢ Software testing tactics:

If we want to ensure that our software is bug-free or stable, we must perform the various types of software testing because testing is the only method that makes our application bug-free.



The different types of Software Testing

The categorization of software testing is a part of diverse testing activities, such as **test strategy, test deliverables, a defined test objective, etc**. And software testing is the execution of the software to find defects.

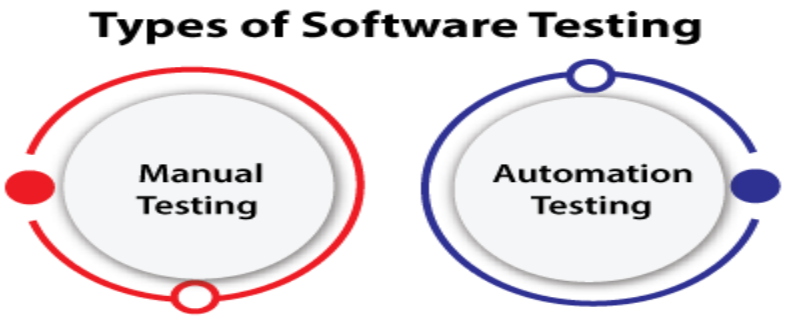
The purpose of having a testing type is to confirm the **AUT** (Application Under Test).

To start testing, we should have a **requirement, application-ready, necessary resources available**.

To maintain accountability, we should assign a respective module to different test engineers.

The software testing mainly divided into two parts, which are as follows:

- **Manual Testing**
- **Automation Testing**



What is Manual Testing?

Testing any software or an application according to the client's needs without using any automation tool is known as **manual testing**.

In other words, we can say that it is a procedure of **verification and validation**.

Manual testing is used to verify the behavior of an application or software in contradiction of requirements specification.

Classification of Manual Testing

In software testing, manual testing can be further classified into **three different types of testing**, which are as follows:

- **White Box Testing**
- **Black Box Testing**
- **Grey Box Testing**

For our better understanding let's see them one by one:

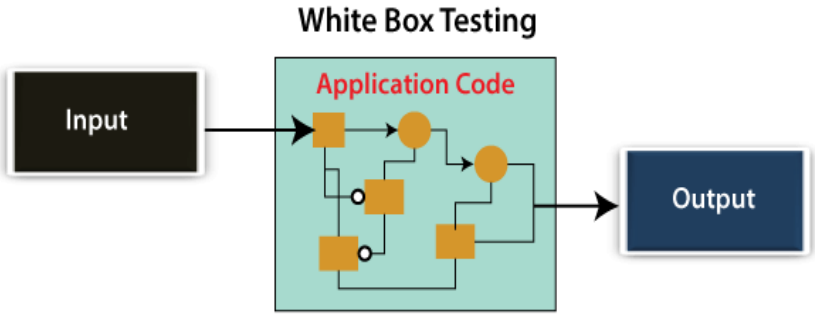
White Box Testing

In white-box testing, the developer will inspect every line of code before handing it over to the testing team or the concerned test engineers.

Subsequently, the code is noticeable for developers throughout testing; that's why this process is known as **WBT (White Box Testing)**.

In other words, we can say that the **developer** will execute the complete white-box testing for the particular software and send the specific application to the testing team.

The purpose of implementing the white box testing is to emphasize the flow of inputs and outputs over the software and enhance the security of an application.



White box testing is also known as **open box testing, glass box testing, structural testing, clear box testing, and transparent box testing**.

Black Box Testing

Another type of manual testing is **black-box testing**. In this testing, the test engineer will analyze the software against requirements, identify the defects or bug, and sends it back to the development team.

Then, the developers will fix those defects, do one round of White box testing, and send it to the testing team.

Here, fixing the bugs means the defect is resolved, and the particular feature is working according to the given requirement.

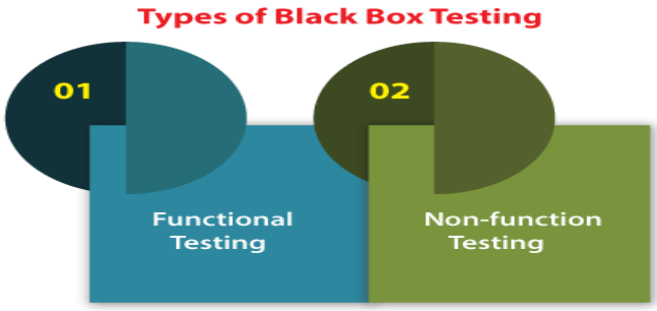
The main objective of implementing the black box testing is to specify the business needs or the customer's requirements.

In other words, we can say that black box testing is a process of checking the functionality of an application as per the customer requirement. The source code is not visible in this testing; that's why it is known as **black-box testing**.

Types of Black Box Testing

Black box testing further categorizes into two parts, which are as discussed below:

- **Functional Testing**
- **Non-function Testing**



Functional Testing

The test engineer will check all the components systematically against requirement specifications is known as **functional testing**. Functional testing is also known as **Component testing**.

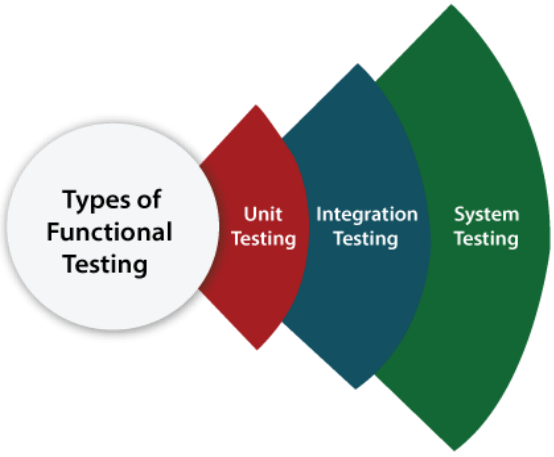
In functional testing, all the components are tested by giving the value, defining the output, and validating the actual output with the expected value.

Functional testing is a part of black-box testing as its emphases on application requirement rather than actual code. The test engineer has to test only the program instead of the system.

Types of Functional Testing

Just like another type of testing is divided into several parts, functional testing is also classified into various categories.

The diverse **types of Functional Testing** contain the following:



- **Unit Testing**
- **Integration Testing**
- **System Testing**

1. Unit Testing

Unit testing is the first level of functional testing in order to test any software. In this, the test engineer will test the module of an application independently or test all the module functionality is called **unit testing**.

The primary objective of executing the unit testing is to confirm the unit components with their performance. Here, a unit is defined as a single testable function of a software or an application. And it is verified throughout the specified application development phase.

2. Integration Testing

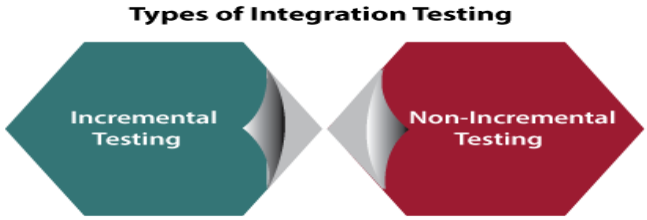
Once we are successfully implementing the unit testing, we will go integration testing

. It is the second level of functional testing, where we test the data flow between dependent modules or interface between two features is called **integration testing**.

The purpose of executing the integration testing is to test the statement's accuracy between each module.

Types of Integration Testing

Integration testing is also further divided into the following parts:



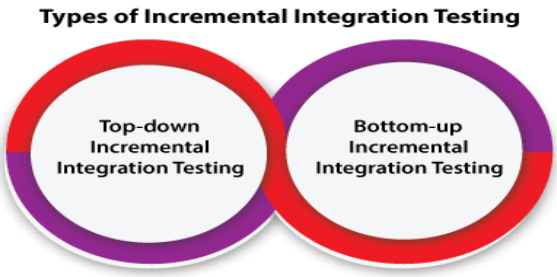
- **Incremental Testing**
- **Non-Incremental Testing**

Incremental Integration Testing

Whenever there is a clear relationship between modules, we go for incremental integration testing. Suppose, we take two modules and analysis the data flow between them if they are working fine or not.

If these modules are working fine, then we can add one more module and test again. And we can continue with the same process to get better results.

In other words, we can say that incrementally adding up the modules and test the data flow between the modules is known as **Incremental integration testing**.



Types of Incremental Integration Testing

Incremental integration testing can further classify into two parts, which are as follows:

1. **Top-down Incremental Integration Testing**
2. **Bottom-up Incremental Integration Testing**

Let's see a brief introduction of these types of integration testing:

1. Top-down Incremental Integration Testing

In this approach, we will add the modules step by step or incrementally and test the data flow between them. We have to ensure that the modules we are adding are the **child of the earlier ones**.

2. Bottom-up Incremental Integration Testing

In the bottom-up approach, we will add the modules incrementally and check the data flow between modules. And also, ensure that the module we are adding is the **parent of the earlier ones**.

Non-Incremental Integration Testing/ Big Bang Method

Whenever the data flow is complex and very difficult to classify a parent and a child, we will go for the non-incremental integration approach. The non-incremental method is also known as **the Big Bang method**.

3. System Testing

Whenever we are done with the unit and integration testing, we can proceed with the system testing.

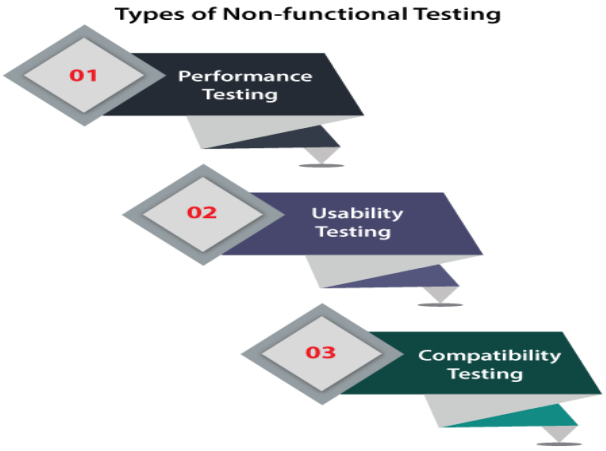
In system testing, the test environment is parallel to the production environment. It is also known as **end-to-end** testing.

Non-function Testing

The next part of black-box testing is **non-functional testing**. It provides detailed information on software product performance and used technologies.

Non-functional testing will help us minimize the risk of production and related costs of the software.

Non-functional testing is a combination of **performance, load, stress, usability and, compatibility testing**.



Types of Non-functional Testing

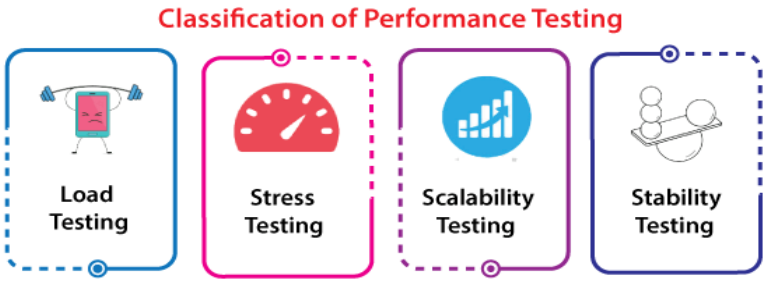
Non-functional testing categorized into different parts of testing, which we are going to discuss further:

- **Performance Testing**
- **Usability Testing**
- **Compatibility Testing**

1. Performance Testing

In performance testing, the test engineer will test the working of an application by applying some load.

In this type of non-functional testing, the test engineer will only focus on several aspects, such as **Response time, Load, scalability, and Stability** of the software or an application.



Classification of Performance Testing

Performance testing includes the various types of testing, which are as follows:

- **Load Testing**
- **Stress Testing**

- **Scalability Testing**
- **Stability Testing**
- **Load Testing**

While executing the performance testing, we will apply some load on the particular application to check the application's performance, known as **load testing**. Here, the load could be less than or equal to the desired load.

It will help us to detect the highest operating volume of the software and bottlenecks.

- **Stress Testing**

It is used to analyze the user-friendliness and robustness of the software beyond the common functional limits.

Primarily, stress testing is used for critical software, but it can also be used for all types of software applications.

Scalability Testing

To analysis, the application's performance by enhancing or reducing the load in particular balances is known as **scalability testing**.

In scalability testing, we can also check the **system, processes, or database's ability** to meet an upward need. And in this, the **Test Cases** are designed and implemented efficiently.

- **Stability Testing**

Stability testing is a procedure where we evaluate the application's performance by applying the load for a precise time.

It mainly checks the constancy problems of the application and the efficiency of a developed product. In this type of testing, we can rapidly find the system's defect even in a stressful situation.

2. Usability Testing

Another type of **non-functional testing** is **usability testing**. In usability testing, we will analyze the user-friendliness of an application and detect the bugs in the software's end-user interface.

Here, the term **user-friendliness** defines the following aspects of an application:

- The application should be easy to understand, which means that all the features must be visible to end-users.
- The application's look and feel should be good that means the application should be pleasant looking and make a feel to the end-user to use it.

3. Compatibility Testing

In compatibility testing, we will check the functionality of an application in specific hardware and software environments. Once the application is functionally stable then only, we go for **compatibility testing**.

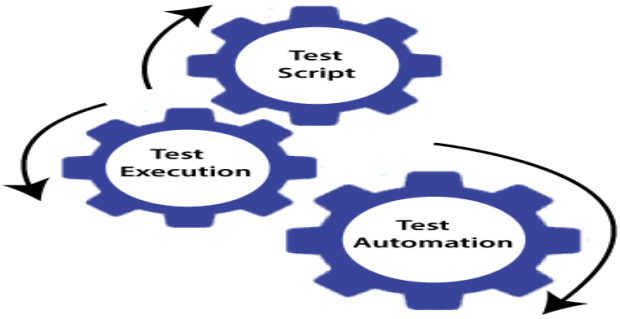
Here, **software** means we can test the application on the different operating systems and other browsers, and **hardware** means we can test the application on different sizes.

Grey Box Testing

Another part of **manual testing** is **Grey box testing**. It is a **collaboration of black box and white box testing**.

Since, the grey box testing includes access to internal coding for designing test cases. Grey box testing is performed by a person who knows coding as well as testing.

In other words, we can say that if a single-person team done both **white box and black-box testing**, it is considered **grey box testing**.



Automation Testing

The most significant part of Software testing is Automation testing. It uses specific tools to automate manual design test cases without any human interference.

Automation testing is the best way to enhance the efficiency, productivity, and coverage of Software testing.

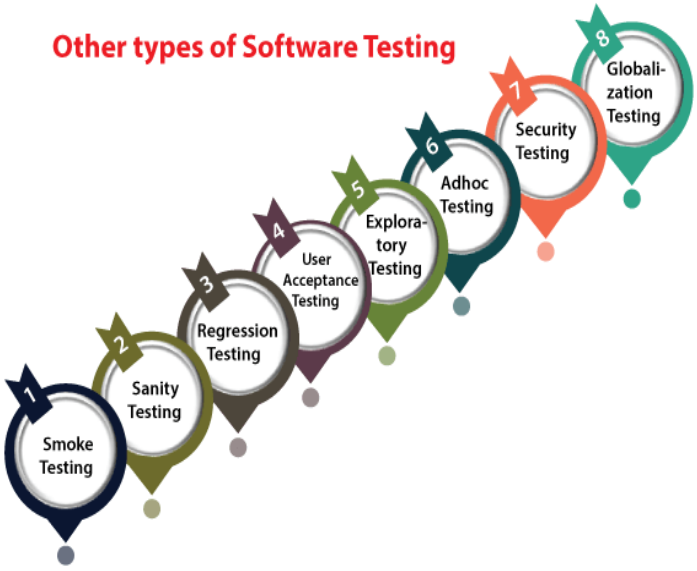
It is used to re-run the test scenarios, which were executed manually, quickly, and repeatedly.

In other words, we can say that whenever we are testing an application by using some tools is known as **automation testing**.

We will go for automation testing when various releases or several regression cycles goes on the application or software. We cannot write the test script or perform the automation testing without understanding the programming language.

Some other types of Software Testing

In software testing, we also have some other types of testing that are not part of any above discussed testing, but those testing are required while testing any software or an application.



- **Smoke Testing**
- **Sanity Testing**
- **Regression Testing**
- **User Acceptance Testing**
- **Exploratory Testing**
- **Adhoc Testing**
- **Security Testing**
- Globalization Testing

Let's understand those types of testing one by one:

In smoke testing**, we will test an application's basic and critical features before** doing one round of deep and rigorous testing.

Or before checking all possible positive and negative values is known as **smoke testing**. Analyzing the workflow of the application's core and main functions is the main objective of performing the smoke testing.

Sanity Testing

It is used to ensure that all the bugs have been fixed and no added issues come into existence due to these changes. Sanity testing is unscripted, which means we cannot documented it. It checks the correctness of the newly added features and components.

Regression Testing

Regression testing is the most commonly used type of software testing. Here, the term **regression** implies that we have to re-test those parts of an unaffected application.

Regression testing is the most suitable testing for automation tools. As per the project type and accessibility of resources, regression testing can be similar to **Retesting**.

Whenever a bug is fixed by the developers and then testing the other features of the applications that might be simulated because of the bug fixing is known as **regression testing**.

In other words, we can say that whenever there is a new release for some project, then we can perform Regression Testing, and due to a new feature may affect the old features in the earlier releases.

User Acceptance Testing

The User acceptance testing (UAT) is done by the individual team known as domain expert/customer or the client. And knowing the application before accepting the final product is called as **user acceptance testing**.

In user acceptance testing, we analyze the business scenarios, and real-time scenarios on the distinct environment called the **UAT environment**. In this testing, we will test the application before UAI for customer approval.

exploratory Testing

Whenever the requirement is missing, early iteration is required, and the testing team has experienced testers when we have a critical application. New test engineer entered into the team then we go for the **exploratory testing**.

To execute the exploratory testing, we will first go through the application in all possible ways, make a test document, understand the flow of the application, and then test the application.

Adhoc Testing

Testing the application randomly as soon as the build is in the checked sequence is known as **Adhoc testing**.

It is also called **Monkey testing and Gorilla testing**. In Adhoc testing, we will check the application in contradiction of the client's requirements; that's why it is also known as **negative testing**.

When the end-user using the application casually, and he/she may detect a bug. Still, the specialized test engineer uses the software thoroughly, so he/she may not identify a similar detection.

Security Testing

It is an essential part of software testing, used to determine the weakness, risks, or threats in the software application.

The execution of security testing will help us to avoid the nasty attack from outsiders and ensure our software applications' security.

Globalization Testing

Another type of software testing is **Globalization testing.** Globalization testing is used to check the developed software for multiple languages or not. Here, the words **globalization** means enlightening the application or software for various languages.

Globalization testing is used to make sure that the application will support multiple languages and multiple features.

In present scenarios, we can see the enhancement in several technologies as the applications are prepared to be used globally.

Levels of Testing

What are the levels of Software Testing?

Testing levels are the procedure for finding the missing areas and avoiding overlapping and repetition between the development life cycle stages. We have already seen the various phases such

as **Requirement collection, designing, coding testing, deployment, and maintenance** of SDLC (Software Development Life Cycle)

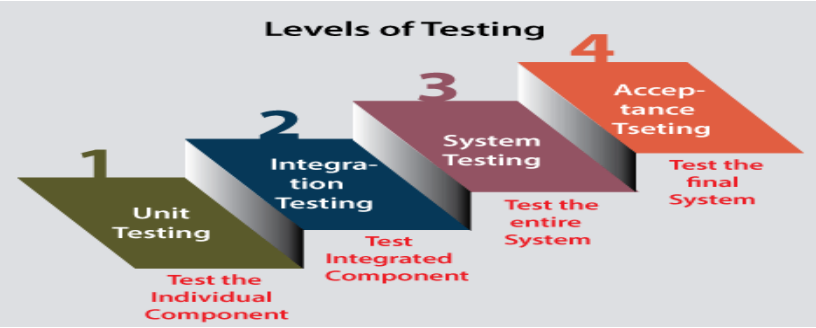
In order to test any application, we need to go through all the above phases of SDLC. Like SDLC, we have multiple levels of testing, which help us maintain the quality of the software.

Different Levels of Testing

The levels of software testing involve the different methodologies, which can be used while we are performing the software testing.

In software testing
we have four different levels of testing, which are as discussed below:

1. **Unit Testing**
2. **Integration Testing**
3. **System Testing**
4. **Acceptance Testing**



As we can see in the above image that all of these testing levels have a specific objective which specifies the value to the software development lifecycle.

For our better understanding, let's see them one by one:

Level1: Unit Testing

Unit testing is the first level of software testing, which is used to test if software modules are satisfying the given requirement or not.

The first level of testing involves **analyzing each unit or an individual component** of the software application.

Unit testing is also the first level of **functional testing**

The primary purpose of executing unit testing is to validate unit components with their performance.

A unit component is an individual function or regulation of the application, or we can say that it is the smallest testable part of the software. The reason of performing the unit testing is to test the correctness of inaccessible code.

Unit testing will help the test engineer and developers in order to understand the base of code that makes them able to change defect causing code quickly. The developers implement the unit.

Level2: Integration Testing

The second level of software testing is the **integration testing.** The integration testing process comes after **unit testing**.

It is mainly used to test the **data flow from one module or component to other modules.**

In integration testing, the **test engineer** tests the units or separate components or modules of the software in a group.

The primary purpose of executing the integration testing is to identify the defects at the interaction between integrated components or units.

When each component or module works separately, we need to check the data flow between the dependent modules, and this process is known as **integration testing**.

We only go for the integration testing when the functional testing has been completed successfully on each application module.

In simple words, we can say that **integration testing** aims to evaluate the accuracy of communication among all the modules.

Level3: System Testing

The third level of software testing is **system testing**, which is used to test the software's functional and non-functional requirements.

It is **end-to-end testing** where the testing environment is parallel to the production environment. In the third level of software testing, **we will test the application as a whole system.**

To check the end-to-end flow of an application or the software as a user is known as **System testing**.

In system testing, we will go through all the necessary modules of an application and test if the end features or the end business works fine, and test the product as a complete system.

In simple words, we can say that System testing is a sequence of different types of tests to implement and examine the entire working of an integrated software computer system against requirements.

Level4: Acceptance Testing

The **last and fourth level** of software testing is **acceptance testing**, which is used to evaluate whether a specification or the requirements are met as per its delivery.

The software has passed through three testing levels (**Unit Testing, Integration Testing, System Testing**). Some minor errors can still be identified when the end-user uses the system in the actual scenario.

In simple words, we can say that Acceptance testing is the **squeezing of all the testing processes that are previously done.**

The acceptance testing is also known as **User acceptance testing (UAT)** and is done by the customer before accepting the final product.

Usually, UAT is done by the domain expert (customer) for their satisfaction and checks whether the application is working according to given business scenarios and real-time scenarios.

➢ White Box Testing

The box testing approach of software testing consists of black box testing and white box testing. We are discussing here white box testing which also known as glass box is **testing, structural testing, clear box testing, open box testing and transparent box testing**. It tests internal coding and infrastructure of a software focus on checking of predefined inputs against expected and desired outputs. It is based on inner workings of an application and revolves around internal structure testing. In this type of testing programming skills are required to design test cases. The primary goal of white box testing is to focus on the flow of inputs and outputs through the software and strengthening the security of the software.

The term 'white box' is used because of the internal perspective of the system. The clear box or white box or transparent box name denote the ability to see through the software's outer shell into its inner workings.

Developers do white box testing. In this, the developer will test every line of the code of the program. The developers perform the White-box testing and then send the application or the software to the testing team, where they will perform the black box testing and verify the application along with the requirements and identify the bugs and sends it to the developer.

The developer fixes the bugs and does one round of white box testing and sends it to the testing team. Here, fixing the bugs implies that the bug is deleted, and the particular feature is working fine on the application.

Here, the test engineers will not include in fixing the defects for the following reasons:

- Fixing the bug might interrupt the other features. Therefore, the test engineer should always find the bugs, and developers should still be doing the bug fixes.

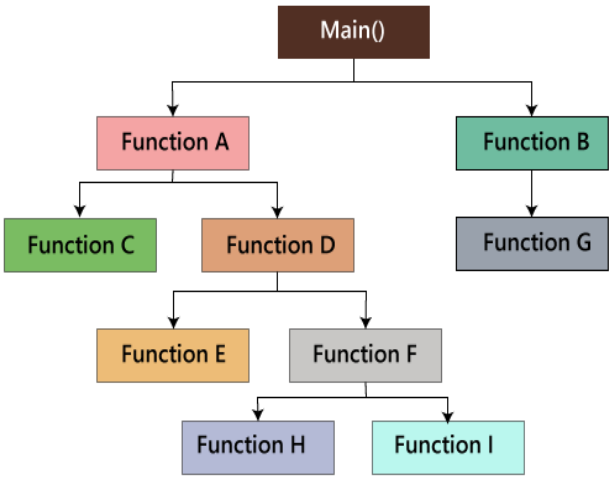
- If the test engineers spend most of the time fixing the defects, then they may be unable to find the other bugs in the application.

The white box testing contains various tests, which are as follows:

- Path testing
- Loop testing
- Condition testing
- Testing based on the memory perspective
- Test performance of the program

Path testing

In the path testing, we will write the flow graphs and test all independent paths. Here writing the flow graph implies that flow graphs are representing the flow of the program and also show how every program is added with one another as we can see in the below image:



And test all the independent paths implies that suppose a path from main() to function G, first set the parameters and test if the program is correct in that particular path, and in the same way test all other paths and fix the bugs.

Loop testing

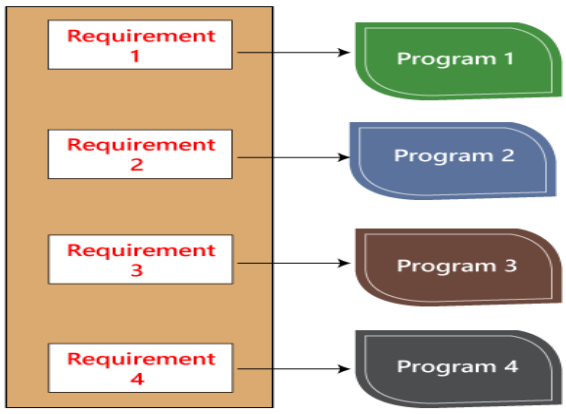
In the loop testing, we will test the loops such as while, for, and do-while, etc. and also check for ending condition if working correctly and if the size of the conditions is enough.

For example: we have one program where the developers have given about 50,000 loops.

```
1.    {
2.    while(50,000)
3.    ……
4.    ……
5.    }
```

We cannot test this program manually for all the 50,000 loops cycle. So we write a small program that helps for all 50,000 cycles, as we can see in the below program, that test P is written in the similar language as the source code program, and this is known as a Unit test. And it is written by the developers only.

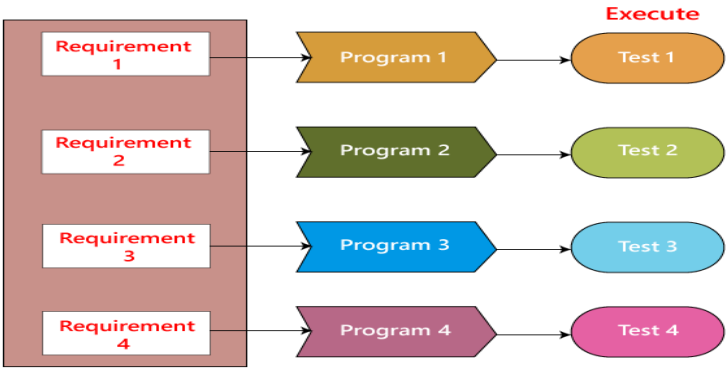
```
1.    Test P
2.    {
3.    ……
4.    …… }
```



As we can see in the below image that, we have various requirements such as 1, 2, 3, 4. And then, the developer writes the programs such as program 1,2,3,4 for the parallel conditions. Here the application contains the 100s line of codes.

The developer will do the white box testing, and they will test all the five programs line by line of code to find the bug. If they found any bug in any of the programs, they will correct it. And they again have to test the system then this process contains lots of time and effort and slows down the product release time.

Now, suppose we have another case, where the clients want to modify the requirements, then the developer will do the required changes and test all four program again, which take lots of time and efforts.



These issues can be resolved in the following ways:

In this, we will write test for a similar program where the developer writes these test code in the related language as the source code. Then they execute these test code, which is also known as **unit test programs**. These test programs linked to the main program and implemented as programs.

Therefore, if there is any requirement of modification or bug in the code, then the developer makes the adjustment both in the main program and the test program and then executes the test program.

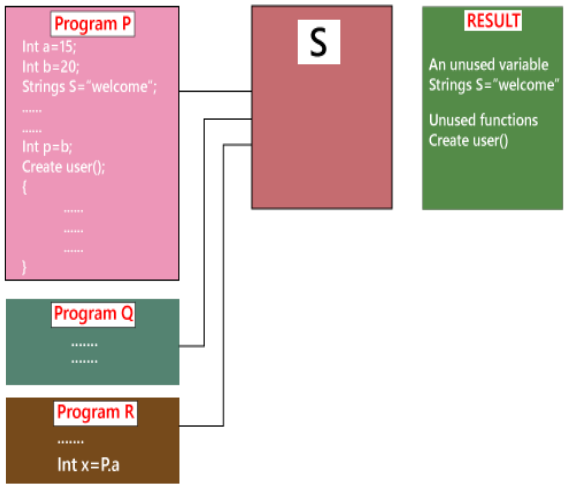
Condition testing

In this, we will test all logical conditions for both **true** and **false** values; that is, we will verify for both **if** and **else** condition.

For example:

1. if(condition) - true
2. {
3. …..
4. ……
5. ……
6. }
7. else - false
8. {
9. …..
10. ……
11. ……
12. }

The above program will work fine for both the conditions, which means that if the condition is accurate, and then else should be false and conversely.



Testing based on the memory (size) perspective

The size of the code is increasing for the following reasons:

- **The reuse of code is not there**: let us take one example, where we have four programs of the same application, and the first ten lines of the program are similar. We can write these ten lines as a discrete function, and it should be

accessible by the above four programs as well. And also, if any bug is there, we can modify the line of code in the function rather than the entire code.

- The **developers use the logic** that might be modified. If one programmer writes code and the file size is up to 250kb, then another programmer could write a similar code using the different logic, and the file size is up to 100kb.
- The **developer declares so many functions and variables** that might never be used in any portion of the code. Therefore, the size of the program will increase.

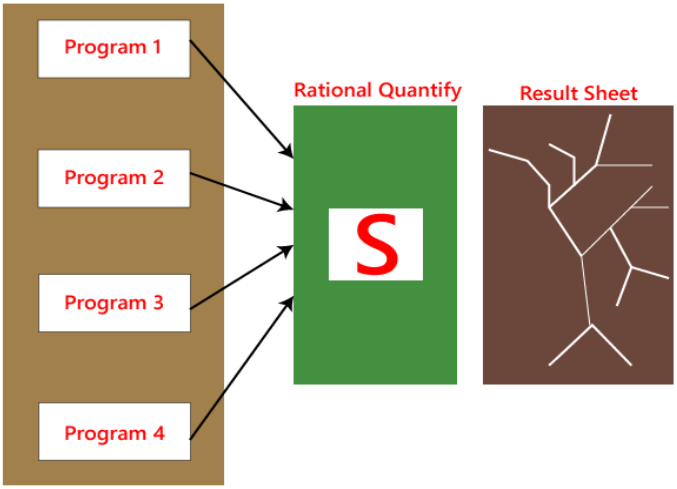
For example,

```
1.Int a=15;
2.Int b=20;
3.String S= "Welcome";
4.….
5.…..
6.…..
7.…..
8.…..
9.Int p=b;
10. Create user()
11. {
12. ……
13. ……
14. ….. 200's line of code
15. }
```

In the above code, we can see that the **integer a** has never been called anywhere in the program, and also the function **Create user** has never been called anywhere in the code. Therefore, it leads us to memory consumption.

We cannot remember this type of mistake manually by verifying the code because of the large code. So, we have a built-in tool, which helps us to test the needless variables and functions. And, here we have the tool called **Rational purify**.

Suppose we have three programs such as Program P, Q, and R, which provides the input to S. And S goes into the programs and verifies the unused variables and then gives the outcome. After that, the developers will click on several results and call or remove the unnecessary function and the variables.

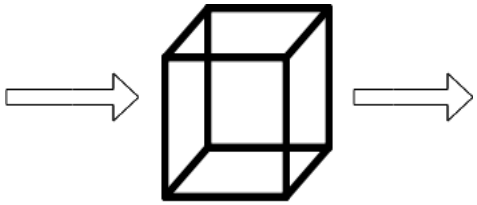


- The developer does not use the available in-built functions; instead they write the full features using their logic. Therefore, it leads us to waste of time and also postpone the product releases.

Test the performance (Speed, response time) of the program

The application could be slow for the following reasons:

- When logic is used.
- For the conditional cases, we will use **or** & **and** adequately.
- Switch case, which means we cannot use **nested if**, instead of using a switch case.



Whitebox Testing

White box testing follows some working steps to make testing manageable and easy to understand what the next task to do. There are some basic steps to perform white box testing.

Advantages of White box testing

- White box testing optimizes code so hidden errors can be identified.
- Test cases of white box testing can be easily automated.
- This testing is more thorough than other testing approaches as it covers all code paths.
- It can be started in the SDLC phase even without GUI.

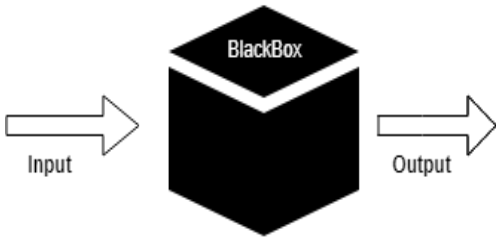
Disadvantages of White box testing

- White box testing is too much time consuming when it comes to large-scale programming applications.
- White box testing is much expensive and complex.
- It can lead to production error because it is not detailed by the developers.
- White box testing needs professional programmers who have a detailed knowledge and understanding of programming language and implementation

➢ Black box testing

Black box testing is a technique of software testing which examines the functionality of software without peering into its internal structure or coding. The primary source of black box testing is a specification of requirements that is stated by the customer.

In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed. The test team reports the result to the development team and then tests the next function. After completing testing of all functions if there are severe problems, then it is given back to the development team for correction.



Generic steps of black box testing

- The black box test is based on the specification of requirements, so it is examined in the beginning.
- In the second step, the tester creates a positive test scenario and an adverse test scenario by selecting valid and invalid input values to check that the software is processing them correctly or incorrectly.
- In the third step, the tester develops various test cases such as decision table, all pairs test, equivalent division, error estimation, cause-effect graph, etc.
- The fourth phase includes the execution of all test cases.
- In the fifth step, the tester compares the expected output against the actual output.
- In the sixth and final step, if there is any flaw in the software, then it is cured and tested again.

Test procedure

The test procedure of black box testing is a kind of process in which the tester has specific knowledge about the software's work, and it develops test cases to check the accuracy of the software's functionality.

Test cases

Test cases are created considering the specification of the requirements. These test cases are generally created from working descriptions of the software including requirements, design parameters, and other specifications. For the testing, the test designer selects both positive test scenario by taking valid input values and adverse test scenario by taking invalid input values to determine the correct output. Test cases are mainly designed for functional testing but can also be used for non-functional testing. Test cases are designed by the testing team, there is not any involvement of the development team of software.

Techniques Used in Black Box Testing

Decision Table Technique	Decision Table Technique is a systematic approach where various combinations and their respective system behavior are captured in a tabula It is appropriate for the functions that have a logical relationship between t more than two inputs.
Boundary Technique	Boundary Value Technique is used to test boundary values, boundary val those that contain the upper and lower limit of a variable. It tests, while e boundary value whether the software is producing correct output or not.
State Transition Technique	State Transition Technique is used to capture the behavior of the so application when different input values are given to the same functio applies to those types of applications that provide the specific number of a to access the application.
All-pair Technique	All-pair testing Technique is used to test all the possible discrete combinat values. This combinational method is used for testing the application th checkbox input, radio button input, list box, text box, etc.
Cause-Effect Technique	Cause-Effect Technique underlines the relationship between a given result the factors affecting the result. It is based on a collection of requirements.
Equivalence Partitioning Technique	Equivalence partitioning is a technique of software testing in which inp divided into partitions of valid and invalid values, and it is mandatory partitions must exhibit the same behavior.
Error Guessing Technique	Error guessing is a technique in which there is no specific method for ider the error. It is based on the experience of the test analyst, where the tester u experience to guess the problematic areas of the software.
Use Case Technique	Use case Technique used to identify the test cases from the beginning to the end of the system as per the usage of the system. By using this technique, the test team creates a test scenario that can exercise the entire software based on the functionality of each function from start to end.

Decision table technique in Black box testing

Decision table technique is one of the widely used case design techniques for black box testing. This is a systematic approach where various input combinations and their respective system behavior is captured in a tabular form.

That's why it is also known as a cause-effect table. This technique is used to pick the test cases in a systematic manner; it saves the testing time and gives good coverage to the testing area of the software application.

Decision table technique is appropriate for the functions that have a logical relationship between two and more than two inputs.

This technique is related to the correct combination of inputs and determines the result of various combinations of input. To design the test cases by decision table technique, we need to consider conditions as input and actions as output.

All-pairs testing

All-pairs testing technique is also known as pair wise testing. It is used to test all the possible discrete combinations of values. This combinational method is used for testing the application that uses checkbox input, radio button input (radio button is used when you have to select only one option for example when you select gender male or female, you can select only one option), list box, text box, etc.

Cause and Effect Graph in Black box Testing

Cause-effect graph comes under the black box testing technique which underlines the relationship between a given result and all the factors affecting the result. It is used to write dynamic test cases.

The dynamic test cases are used when code works dynamically based on user input. For example, while using email account, on entering valid email, the system accepts it but, when you enter invalid email, it throws an error message. In this technique, the input conditions are assigned with causes and the result of these input conditions with effects.

Cause-Effect graph technique is based on a collection of requirements and used to determine minimum possible test cases which can cover a maximum test area of the software.

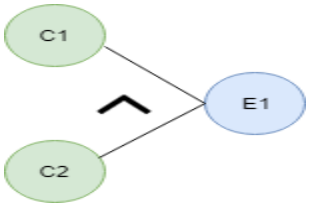
The main advantage of cause-effect graph testing is, it reduces the time of test execution and cost.

This technique aims to reduce the number of test cases but still covers all necessary test cases with maximum coverage to achieve the desired application quality.

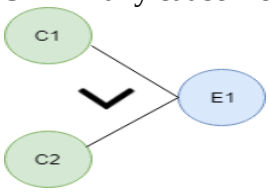
Cause-Effect graph technique converts the requirements specification into a logical relationship between the input and output conditions by using logical operators like AND, OR and NOT.

Notations used in the Cause-Effect Graph

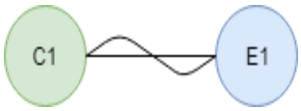
AND - E1 is an effect and C1 and C2 are the causes. If both C1 and C2 are true, then effect E1 will be true.



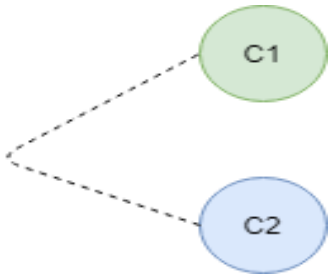
OR - If any cause from C1 and C2 is true, then effect E1 will be true.



NOT - If cause C1 is false, then effect E1 will be true.



Mutually Exclusive - When only one cause is true.



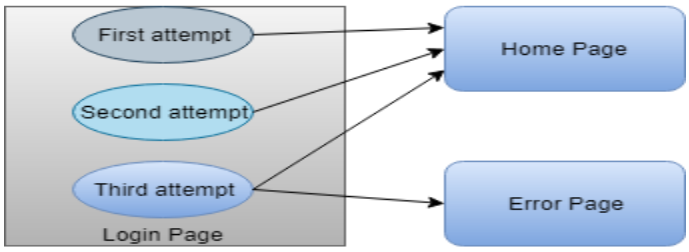
State Transition Technique

The general meaning of state transition is, different forms of the same situation, and according to the meaning, the state transition method does the same. It is used to capture the behavior of the software application when different input values are given to the same function.

We all use the ATMs, when we withdraw money from it, it displays account details at last. Now we again do another transaction, then it again displays account details, but the details displayed after the second transaction are different from the first transaction, but both details are displayed by using the same function of the ATM. So the same function was used here but each time the output was different, this is called state transition. In the case of testing of a software application, this method tests whether the function is following state transition specifications on entering different inputs.

This applies to those types of applications that provide the specific number of attempts to access the application such as the login function of an application which gets locked after the specified number of incorrect attempts. Let's see in detail, in the login function we use email and password, it gives a specific number of attempts to access the application, after crossing the maximum number of attempts it gets locked with an error message.

There is a login function of an application which provides a maximum three number of attempts, and after exceeding three attempts, it will be directed to an error page.



State transition table

STA	LOGIN	VALIDATION	REDIRECTED
S1	First Attempt	Invalid	S2
S2	Second Attempt	Invalid	S3
S3	Third Attempt	Invalid	S5
S4	Home Page		
S5	Error Page		

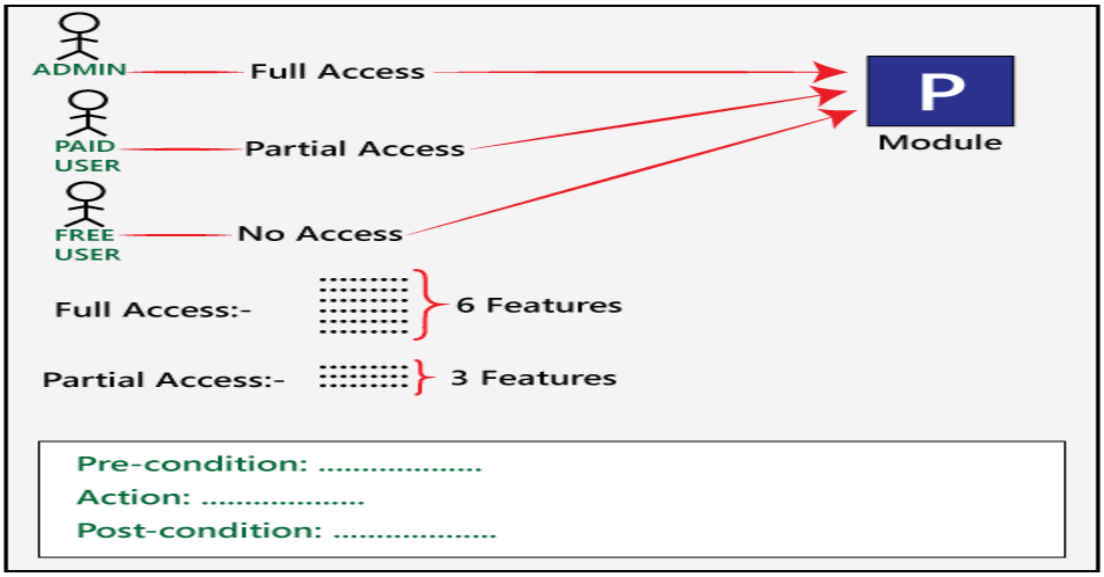
In the above state transition table, we see that state S1 denotes first login attempt. When the first attempt is invalid, the user will be directed to the second attempt (state S2). If the second attempt is also invalid, then the user will be directed to the third attempt (state S3). Now if the third and last attempt is invalid, then the user will be directed to the error page (state S5).

But if the third attempt is valid, then it will be directed to the homepage (state S4).

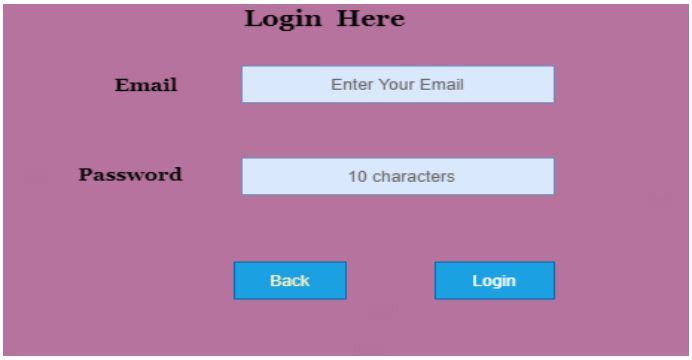
Use Case Technique

The use case is functional testing of the black box testing used to identify the test cases from the beginning to the end of the system as per the usage of the system. By using this technique, the test team creates a test scenario that can exercise the entire software based on the functionality of each function from start to end.

It is a graphic demonstration of business needs, which describe how the end-user will cooperate with the software or the application. The use cases provide us all the possible techniques of how the end-user uses the application as we can see in the below image, that how the **use case** will look like:



In the above image, we can see that a sample of a use case where we have a requirement related to the customer requirement specification (CRS).



For **module P** of the software, we have six different featuresa Tricky Program 23 - Main method

And here, **Admin** has access to all the **six features**, the **Paid user** has access to the **three features** and for the **Free user**, there is **no access** provided to any of the features.

Like for **Admin**, the different conditions would be as below:

Pre-condition→ Admin must be generated

Action→ Login as Paid user

Post-condition→ 3 features must be present

And for **Free user**, the different condition would be as below:

Pre-condition→ free user must be generated

Action→ Login as a free user

Post-condition→ no features

- **Control Structure Testing**

Control structure testing is used to increase the coverage area by testing various control structures present in the program. The different types of testing performed under control structure testing are as follows-

1. Condition Testing
2. Data Flow Testing
3. Loop Testing

1. Condition Testing

Condition testing is a test cased design method, which ensures that the logical condition and decision statements are free from errors. The errors present in logical conditions can be incorrect boolean operators, missing parenthesis in a booleans expression, error in relational operators, arithmetic expressions, and so on.

The common types of logical conditions that are tested using condition testing are-

1. A relation expression, like E1 op E2 where 'E1' and 'E2' are arithmetic expressions and 'OP' is an operator.
2. A simple condition like any relational expression preceded by a NOT (~) operator. For example, (~E1) where 'E1' is an arithmetic expression and 'a' denotes NOT operator.
3. A compound condition consists of two or more simple conditions, Boolean operator, and parenthesis.
   For example, (E1 & E2)|(E2 & E3) where E1, E2, E3 denote arithmetic expression and '&' and '|' denote AND or OR operators.
4. A Boolean expression consists of operands and a Boolean operator like 'AND', OR, NOT. For example, 'A|B' is a Boolean expression where 'A' and 'B' denote operands and | denotes OR operator.

2. Data Flow Testing

The data flow test method chooses the test path of a program based on the locations of the definitions and uses all the variables in the program.

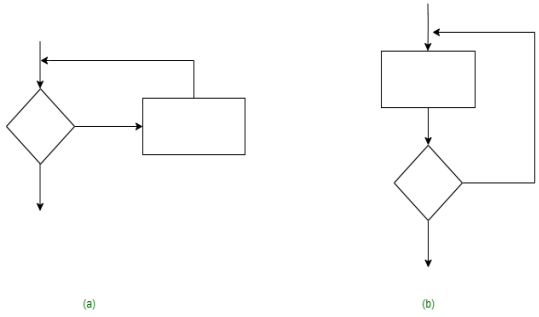
The data flow test approach is depicted as follows suppose each statement in a program is assigned a unique statement number and that theme function cannot modify its parameters or global variables. For example, with S as its statement number.

DEF (S) = {X | Statement S has a definition of X}

USE (S) = {X | Statement S has a use of X}

If statement S is an if loop statement, them its DEF set is empty and its USE set depends on the state of statement S. The definition of the variable X at statement S is called the line of statement S' if the statement is any way from S to statement S' then there is no other definition of X.

A definition use (DU) chain of variable X has the form [X, S, S'], where S and S' denote statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is line at statement S'.



SIMPLE LOOPS

A simple data flow test approach requires that each DU chain be covered at least once. This approach is known as

the DU test approach. The DU testing does not ensure coverage of all branches of a program.

However, a branch is not guaranteed to be covered by DU testing only in rar cases such as then in which the other construct does not have any certainty of any variable in its later part and the other part is not present. Data flow testing strategies are appropriate for choosing test paths of a program containing nested if and loop statements.

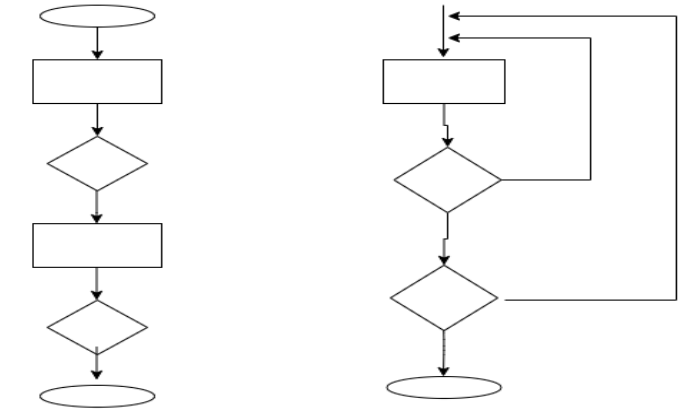
3. Loop Testing:

Loop testing is actually a white box testing technique. It specifically focuses on the validity of loop construction.

Following are the types of loops.

1. **Simple Loop –** The following set of test can be applied to simple loops, where the maximum allowable number through the loop is n.
   1. Skip the entire loop.
   2. Traverse the loop only once.
   3. Traverse the loop two times.
   4. Make p passes through the loop where p<n.
   5. Traverse the loop n-1, n, n+ 1 time.

2 Concatenated Loops – **If loops are not dependent on each other, contact loops can be tested using the approach used in simple loops. if the loops are interdependent, the steps are followed in nested loops.**

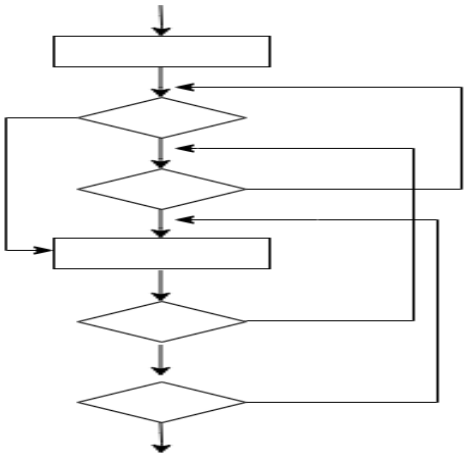


Concatenated Loops

3 Nested Loops – Loops within loops are called as nested loops. when testing nested loops, the number of tested increases as level nesting increases. The following steps for testing nested loops are as follows-

1. Start with inner loop. set all other loops to minimum values.
2. Conduct simple loop testing on inner loop.
3. Work outwards.
4. Continue until all loops tested.

4 Unstructured loops – This type of loops should be



Unstructured Loops

redesigned, whenever possible, to reflect the use of unstructured the structured programming constructs.

➢ Basis Path Testing in Software Testing

Basis Path Testing is a white-box testing technique based on the control structure of a program or a module. Using this structure, a control flow graph is prepared and the various possible paths present in the graph are executed as a part of testing. Therefore, by definition,

Basis path testing is a technique of selecting the paths in the control flow graph, that provide a basis set of execution paths through the program or module.

Since this testing is based on the control structure of the program, it requires complete knowledge of the program's structure. To design test cases using this technique, four steps are followed:

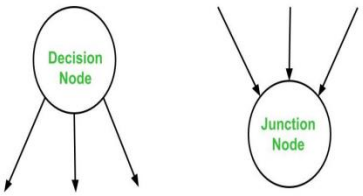
1. Construct the Control Flow Graph
2. Compute the Cyclomatic Complexity of the Graph
3. Identify the Independent Paths
4. Design Test cases from Independent Paths

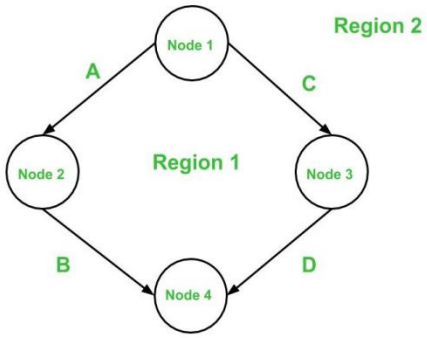
Let's understand each step one by one.

1. Control Flow Graph

A control flow graph (or simply, flow graph) is a directed graph which represents the control structure of a program or module. A control flow graph (V, E) has V number of nodes/vertices and E number of edges in it. A control graph can also have :

- **Junction Node –** a node with more than one arrow entering it.
- **Decision Node –** a node with more then one arrow leaving it.
- **Region –** area bounded by edges and nodes (area outside the graph is also counted as a region.).

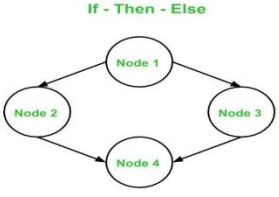


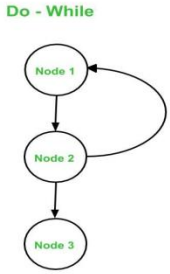


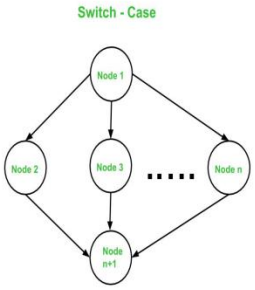
Below are the **notations** used while constructing a flow graph :

- **Sequential Statements**

If – Then – Else –



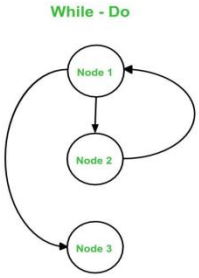




Do – While

While – Do

Switch – Case



Cyclomatic Complexity

The cyclomatic complexity V(G) is said to be a measure of the logical complexity of a program. It can be calculated using three different formulae:

1. **Formula based on edges and nodes :**

   V (G) = e - n + 2*P

   Where,

   e is number of edges

   n is number of vertices

   P is number of connected components.

   For example, consider first graph given above,

   where, e = 4, n = 4 and p = 1

   So,

   Cyclomatic complexity V(G)

   = 4 - 4 + 2 * 1

   = 2

2. **Formula based on Decision Nodes :**

   V (G) = d + P

   where,

   d is number of decision nodes

   P is number of connected nodes.

   For example, consider first graph given above,

   where, d = 1 and p = 1

   So,

   Cyclomatic Complexity V(G)

   = 1 + 1

   = 2

1. **Formula based on Regions :**

   V(G) = number of regions in the graph

   For example, consider first graph given above,

   Cyclomatic complexity V(G)

   = 1 (for Region 1) + 1 (for Region 2)

   = 2

Hence, using all the three above formulae, the cyclomatic complexity obtained remains same. All these three formulae can be used to compute and verify the cyclomatic complexity of the flow graph.

Note –

1. For one function [e.g. Main ( ) or Factorial ( )], only one flow graph is constructed. If in a program, there are multiple functions, then a separate flow graph is constructed for each one of

them. Also, in the cyclomatic complexity formula, the value of 'p' is set depending of the number of graphs present in total.

2. If a decision node has exactly two arrows leaving it, then it is counted as one decision node. However, if there are more than 2 arrows leaving a decision node, it is computed using this formula
   d = k - 1
   Here, k is number of arrows leaving the decision node.

Independent Paths

An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined. The cyclomatic complexity gives the number of independent paths present in a flow graph. This is because the cyclomatic complexity is used as an upper-bound for the number of tests that should be executed in order to make sure that all the statements in the program have been executed at least once.

Consider first graph given above here the independent paths would be 2 because number of independent paths is equal to the cyclomatic complexity.
So, the independent paths in above first given graph :

- **Path 1:**
  A -> B
- **Path 2:**
  C -> D

Note –

Independent paths are not unique. In other words, if for a graph the cyclomatic complexity comes out be N, then there is a possibility of obtaining two different sets of paths which are independent in nature.

Design Test Cases:

Finally, after obtaining the independent paths, test cases can be designed where each test case represents one or more independent paths.

Advantages:

Basis Path Testing can be applicable in the following cases:

More Coverage

Basis path testing provides the best code coverage as it aims to achieve maximum logic coverage instead of maximum path coverage. This results in an overall thorough testing of the code.

Maintenance Testing

When a software is modified, it is still necessary to test the changes made in the software which as a result, requires path testing.

Unit Testing

When a developer writes the code, he or she tests the structure of the program or module themselves first. This is why basis path testing requires enough knowledge about the structure of the code.

Integration Testing

When one module calls other modules, there are high chances of Interface errors. In order to avoid the case of such errors, path testing is performed to test all the paths on the interfaces of the modules.

Testing Effort

Since the basis path testing technique takes into account the complexity of the software (i.e., program or module) while computing the cyclomatic complexity, therefore it is intuitive to note that

testing effort in case of basis path testing is directly proportional to the complexity of the software or program.

Object –Oriented Testing Methods

Test case design methods for OO software are still evolving. However, an overall approach to OO test case design has been defined by Berard :

1. Each test case should be uniquely identified and explicitly associated with the class to be tested.
2. The purpose of the test should be stated.
3. A list of testing steps should be developed for each test and should contain:

a. A list of specified states for the object that is to be tested.
b. A list of messages and operations that will be exercised as a consequence of the test.
c. A list of exceptions that may occur as the object is tested.
d. A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test).
e. Supplementary information that will aid in understanding or implementing the test.

➢ The Test Case Design Implications of OO Concepts

The OO class is the target for test case design. Because attributes and operations are encapsulated, testing operations outside of the class is generally unproductive. Although encapsulation is an essential design concept for OO, it can create a minor obstacle when testing. As Binder notes, "Testing requires reporting on the concrete and abstract state of an object." Yet, encapsulation can make this information somewhat difficult to obtain. Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

Inheritance also leads to additional challenges for the test case designer. We have already noted that each new context of usage requires retesting, even though reuse has been achieved. In addition, multiple inheritance complicates testing further by increasing the number of contexts for which testing is required. If subclasses instantiated from a super class are used within the same problem domain, it is likely that the set of test cases derived for the super class can be used when testing the subclass. However, if the subclass is used in an entirely different context, the super class test cases will have little applicability and a new set of tests must be designed.

Applicability of Conventional Test Case Design Methods

The white-box testing methods can be applied to the operations defined for a class. Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested. However, the concise structure of many class operations causes some to argue that the effort applied to white-box testing might be better redirected to tests at a class level.

Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods. Use-cases can provide useful input in the design of black-box and state-based tests.

Fault-Based Testing

The object of fault-based testing within an OO system is to design tests that have a high likelihood of uncovering plausible faults. Because the product or system must conform to

customer requirements, the preliminary planning required to perform fault based testing begins with the analysis model. The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.

Consider a simple example.5 Software engineers often make errors at the boundaries of a problem. For example, when testing a SQRT operation that returns errors for negative numbers, we know to try the boundaries: a negative number close to zero and zero itself. "Zero itself" checks whether the programmer made a mistake like

if (x > 0) calculate_the_square_root();

instead of the correct

if (x >= 0) calculate_the_square_root();

As another example, consider a Boolean expression:

if (a && !b || c)

Multicondition testing and related techniques probe for certain plausible faults in this expression, such as

&& should be ||! was left out where it was needed There should be parentheses around !b || c

For each plausible fault, we design test cases that will force the incorrect expression to fail. In the previous expression, (a=0, b=0, c=0) will make the expression as given evaluate false. If the && should have been ||, the code has done the wrong thing and might branch to the wrong path.

Of course, the effectiveness of these techniques depends on how testers perceive a "plausible fault." If real faults in an OO system are perceived to be "implausible," then this approach is really no better than any random testing technique. However, if the analysis and design models can provide insight into what is likely to go wrong, then fault-based testing can find significant numbers of errors with relatively low expenditures of effort.

Integration testing looks for plausible faults in operation calls or message connections. Three types of faults are encountered in this context: unexpected result, wrong operation/message used, incorrect invocation. To determine plausible faults as functions (operations) are invoked, the behavior of the operation must be examined.

Integration testing applies to attributes as well as to operations. The "behaviors" of an object are defined by the values that its attributes are assigned. Testing should exercise the attributes to determine whether proper values occur for distinct types of object behavior.

It is important to note that integration testing attempts to find errors in the client object, not the server. Stated in conventional terms, the focus of integration testing is to determine whether errors exist in the calling code, not the called code. The operation call is used as a clue, a way to find test requirements that exercise the calling code.

The Impact of OO Programming on Testing

There are several ways object-oriented programming can have an impact on testing. Depending on the approach to OOP,

- Some types of faults become less plausible (not worth testing for)
- Some types of faults become more plausible (worth testing now)
- Some new types of faults appear.

When an operation is invoked, it may be hard to tell exactly what code gets exercised. That is, the operation may belong to one of many classes. Also, it can be hard to determine the exact type or class of a parameter. When the code accesses it, it may get an unexpected value. The difference can be understood by considering a conventional function call:

x = func (y);

For conventional software, the tester need consider all behaviors attributed to func and nothing more. In an OO context, the tester must consider the behaviors of base::func(), of derived::func(), and so on. Each time func is invoked, the tester must consider the union of all distinct behaviors. This is easier if good OO design practices are followed and the difference between superclasses and subclasses (in C++ jargon, these are called base classes and derived classes) are limited. The testing approach for base and derived classes is essentially the same.

Testing OO class operations is analogous to testing code that takes a function parameter and then invokes it. Inheritance is a convenient way of producing polymorphic operations. At the call site, what matters is not the inheritance, but the polymorphism. Inheritance does make the search for test requirements more straightforward.

By virtue of OO software architecture and construction, are some types of faults more plausible for an OO system and others less plausible? The answer is, "Yes." For example, because OO operations are generally smaller, more time tends to be spent on integration because there are more opportunities for integration faults. Therefore, integration faults become more plausible.

Test Cases and the Class Hierarchy

Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.

Consider the following situation. A class base contains operations inherited and redefined. A class derived redefines redefined to serve in a local context. There is little doubt the derived::redefined () has to be tested because it represents a new design and new code. But does derived::inherited() have to be retested?

If derived::inherited() calls redefined and the behavior of redefined has changed, derived::inherited () may mishandle the new behavior. Therefore, it needs new tests even though the design and code have not changed. It is important to note, however, that only a subset of all tests for derived::inherited() may have to be conducted. If part of the design and code for inherited does not depend on redefined (i.e., that does not call it nor call any code that indirectly calls it), that code need not be retested in the derived class.

Base::redefined () and **derived::redefined ()** are two different operations with different specifications and implementations. Each would have a set of test requirements derived from the specification and implementation. Those test requirements probe for plausible faults: integration faults, condition faults, boundary faults, and so forth. But the operations are likely to be similar. Their sets of test requirements will overlap. The better the OO design, the greater is the overlap. New tests need to be derived only for those derived::redefined () requirements that are not satisfied by the base::redefined () tests.

To summarize, the base::redefined () tests are applied to objects of class derived. Test inputs may be appropriate for both base and derived classes, but the expected results may differ in the derived class.

Scenario-Based Test Design

Fault-based testing misses two main types of errors: **(1)** incorrect specifications and **(2)** interactions among subsystems. When errors associated with incorrect specification occur, the product doesn't do what the customer wants. It might do the wrong thing or it might omit important functionality. But in either circumstance, quality (conformance to requirements) suffers. Errors associated with subsystem interaction occur when the behavior

of one subsystem creates circumstances (e.g., events, data flow) that cause another subsystem to fail.

Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests .

Scenarios uncover interaction errors. But to accomplish this, test cases must be more complex and more realistic than fault-based tests. Scenario-based testing tends to exercise multiple subsystems in a single test (users do not limit themselves to the use of one subsystem at a time).

As an example, consider the design of scenario-based tests for a text editor. Use cases follow:

Use-Case: Fix the Final Draft

Background: It's not unusual to print the "final" draft, read it, and discover some annoying errors that weren't obvious from the on-screen image. This use-case describes the sequence of events that occurs when this happens

1. Print the entire document

2. Move around in the document, changing certain pages

3. As each page is changed, it's printed

4. Sometimes a series of pages is printed.

This scenario describes two things: a test and specific user needs. The user needs are obvious: **(1)** a method for printing single pages and **(2)** a method for printing a range of pages. As far as testing goes, there is a need to test editing after printing (as well as the reverse). The tester hopes to discover that the printing function causes errors in the editing function; that is, that the two software functions are not properly independent.

Use-Case: Print a New Copy

Background: Someone asks the user for a fresh copy of the document. It must be printed.

1. Open the document

2. Print it .

3. Close the document.

Again, the testing approach is relatively obvious. Except that this document didn't appear out of nowhere. It was created in an earlier task. Does that task affect this one?

In many modern editors, documents remember how they were last printed. By default, they print the same way next time. After the Fix the Final Draft scenario, just selecting "Print" in the menu and clicking the "Print" button in the dialog box will cause the last corrected page to print again. So, according to the editor, the correct scenario should look like this:

Use-Case: Print a New Copy

1. Open the document

2. Select "Print" in the menu

3. Check if you're printing a page range; if so, click to print the entire document.

4. Click on the Print button

5. Close the document.

A test case designer might miss this dependency in test design, but it is likely that the problem would surface during testing. The tester would then have to contend with the probable response, "That's the way it's supposed to work!"

Testing Surface Structure and Deep Structure

Surface structure refers to the externally observable structure of an OO program. That is, the structure that is immediately obvious to an end-user. Rather than performing functions,

the users of many OO systems may be given objects to manipulate in some way. But whatever the interface, tests are still based on user tasks. Capturing these tasks involves understanding, watching, and talking with representative users (and as many non representative users as are worth considering).

There will surely be some difference in detail. For example, in a conventional system with a command-oriented interface, the user might use the list of all commands as a testing checklist. If no test scenarios existed to exercise a command, testing has likely overlooked some user tasks (or the interface has useless commands). In a object based interface, the tester might use the list of all objects as a testing checklist.

The best tests are derived when the designer looks at the system in a new or unconventional way. For example, if the system or product has a command-based interface, more thorough tests will be derived if the test case designer pretends that operations are independent of objects. Ask questions like, "Might the user want to use this operation—which applies only to the Scanner object—while working with the printer?" Whatever the interface style, test case design that exercises the surface structure should use both objects and operations as clues leading to overlooked tasks.

Deep structure refers to the internal technical details of an OO program. That is, the structure that is understood by examining the design and/or code. Deep structure testing is designed to exercise dependencies, behaviors, and communication mechanisms that have been established as part of the system and object design of OO software.

➢ Testing Methods applicable at the Class level

We noted that software testing begins "in the small" and slowly progresses toward testing "in the large." Testing in the small focuses on a single class and the methods that are encapsulated by the class. Random testing and partitioning are methods that can be used to exercise a class during OO testing.

Random Testing for OO Classes

To provide brief illustrations of these methods, consider a banking application in which an account class has the following operations: open, setup, deposit, withdraw, balance, summarize, creditLimit, and close. Each of these operations may be applied for account, but certain constraints (e.g., the account must be opened before other operations can be applied and closed after all operations are completed) are implied by the nature of the problem. Even with these constraints, there are many permutations of the operations. The minimum behavioral life history of an instance of account includes the following operations:

open•setup•deposit•withdraw•close

This represents the minimum test sequence for account. However, a wide variety of other behaviors may occur within this sequence:

open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]n•withdraw•close

A variety of different operation sequences can be generated randomly. For example:

Test case r1: **open•setup•deposit•deposit•balance•summarize•withdraw•close**

Test case r2: **open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close**

These and other random order tests are conducted to exercise different class instance life histories.

Partition Testing at the Class Level

Partition testing reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning for conventional software. Input and output are categorized and test cases are designed to exercise each category. But how are the partitioning categories derived?

State-based partitioning categorizes class operations based on their ability to change the state of the class. Again considering the account class, state operations include deposit and withdraw, whereas no state operations include balance, summarize, and creditLimit. Tests are designed in a way that exercises operations that change state and those that do not change state separately. Therefore,

Test case p1: open•setup•deposit•deposit•withdraw•withdraw•close

Test case p2: open•setup•deposit•summarize•creditLimit•withdraw•close

Test case p1 changes state, while test case p2 exercises operations that do not change state (other than those in the minimum test sequence).

Attribute-based partitioning categorizes class operations based on the attributes that they use. For the account class, the attributes balance and creditLimit can be used to define partitions. Operations are divided into three partitions: (1) operations that use creditLimit, (2) operations that modify creditLimit, and (3) operations that do not use or modify creditLimit. Test sequences are then designed for each partition.

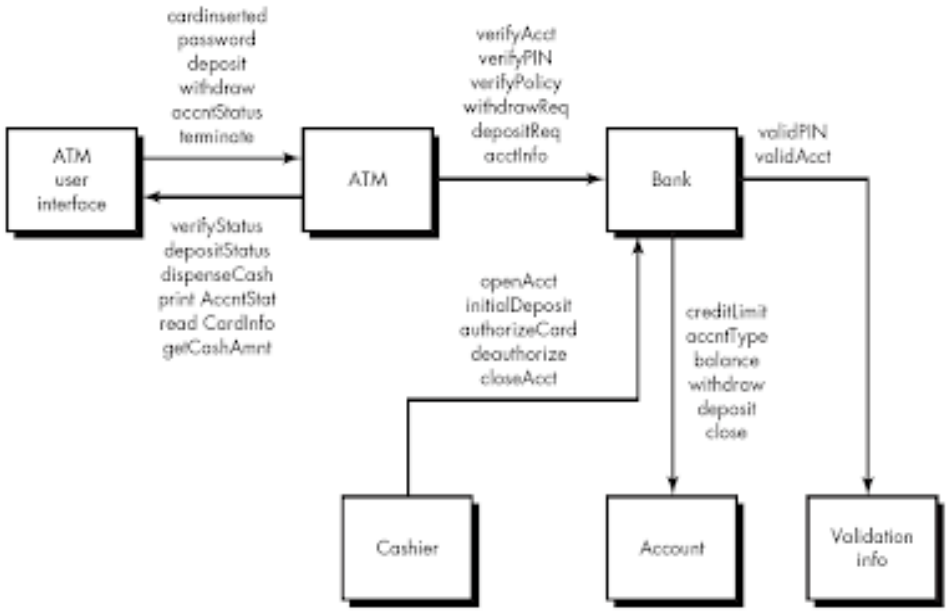
Category-based partitioning categorizes class operations based on the generic function that each performs. For example, operations in the account class can be categorized in initialization operations (open, setup), computational operations (deposit, withdraw). queries (balance, summarize, creditLimit) and termination operations (close).

➢ Interclass Test case Design

Test case design becomes more complicated as integration of the OO system begins. It is at this stage that testing of collaborations between classes must begin. To illustrate "interclass test case generation" , we expand the banking example to include the classes and collaborations noted in Figure below The direction of the arrows in the figure indicates the direction of messages and the labeling indicates the operations that are invoked as a consequence of the collaborations implied by the messages.

Like the testing of individual classes, class collaboration testing can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioral testing.

Multiple Class Testing



Kirani and Tsai suggest the following sequence of steps to generate multiple class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.
3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

To illustrate, consider a sequence of operations for the bank class relative to an ATM class :

verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]n

A random test case for the bank class might be

test case r3 = **verifyAcct•verifyPIN•depositReq**

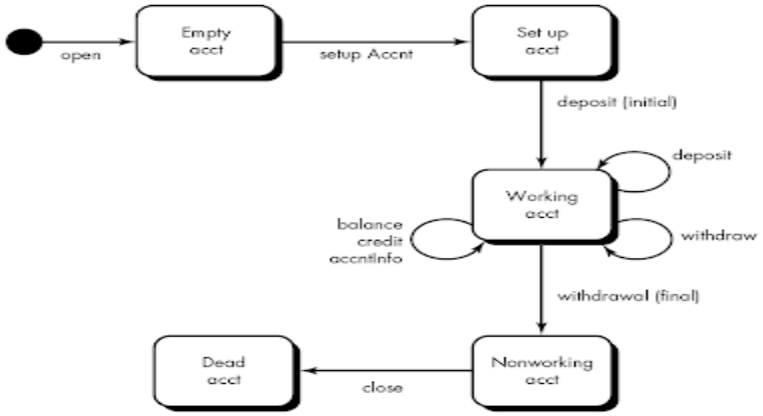
In order to consider the collaborators involved in this test, the messages associated with each of the operations noted in test case r3 is considered. Bank must collaborate with ValidationInfo to execute the verifyAcct and verifyPIN. Bank must collaborate with account to execute depositReq. Hence, a new test case that exercises these collaborations is The approach for multiple class partition testing is similar to the approach used for partition testing of individual classes. However, the test sequence is expanded to include those operations that are invoked via messages to collaborating classes. An alternative approach partitions tests based on the interfaces to a particular class. Referring to figure, the bank class receives messages from the ATM and cashier classes. The methods within bank can therefore be tested by partitioning them into those that serve ATM and those that serve cashier. State-based partitioning can be used to refine the partitions further.

Tests Derived from Behavior Models

We already discussed the use of the state transition diagram as a model that represents the dynamic behavior of a class. The STD for a class can be used to help derive a sequence of

tests that will exercise the dynamic behavior of the class (and those classes that collaborate with it). Figure below illustrates an STD for the account class Referring to the figure, initial transitions move through the empty acct and setup acct states. The majority of all behavior for instances of the class occurs while in the working acct state. A final withdrawal and close cause the account class to make transitions to the nonworking acct and dead acct states, respectively.

test case r4 = **verifyAcctBank[validAcctValidationInfo]•verifyPINBank• [validPinValidationInfo]•depositReq• [depositaccount]**



The tests to be designed should achieve all state coverage. That is, the operation sequences should cause the account class to make transition through all allowable states:

test case s1: **open•setupAccnt•deposit (initial)•withdraw (final)•close**

It should be noted that this sequence is identical to the minimum test sequence . Adding additional test sequences to the minimum sequence,

test case s2: **open•setupAccnt•deposit(initial)•deposit•balance•credit•withdraw (final)•close**

test case s3: **open•setupAccnt•deposit(initial)•deposit•withdraw•accntInfo•withdraw (final)•close**

Still more test cases could be derived to ensure that all behaviors for the class have been adequately exercised. In situations in which the class behavior results in a collaboration with one or more classes, multiple STDs are used to track the behavioral flow of the system.

The state model can be traversed in a "breadth-first" manner. In this context, breadth first implies that a test case exercise a single transition and that when a new transition is to be tested only previously tested transitions are used.

Consider the credit card object discussed earlier. The initial state of credit card is undefined (i.e., no credit card number has been provided). Upon reading the credit card during a sale, the object takes on a defined state; that is, the attributes card number and expiration date, along with bank specific identifiers are defined. The credit card is submitted when it is sent for authorization and it is approved when authorization is received. The transition of credit card from one state to another can be tested by deriving test cases that cause the transition to occur.