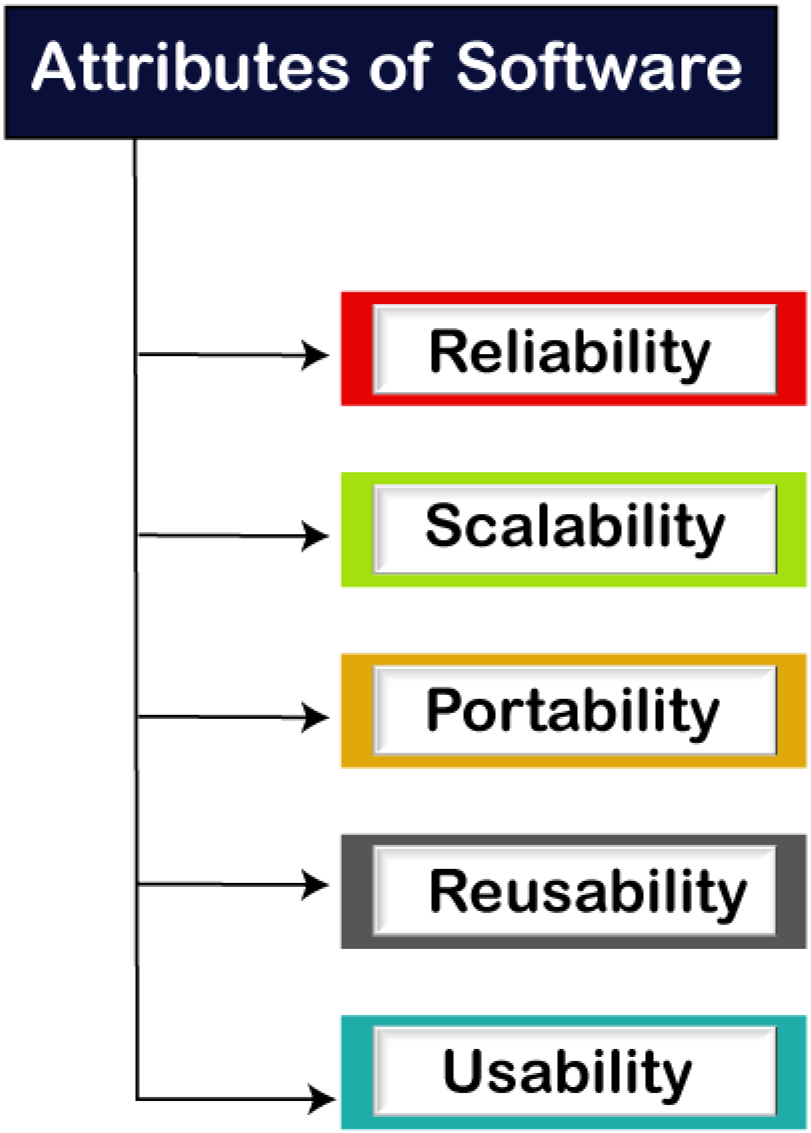
What is Software Testing

Software testing is a process of identifying the correctness of software by considering its all attributes (Reliability, Scalability, Portability, Re-usability, Usability) and evaluating the execution of software components to find the software bugs or errors or defects.



Software testing provides an independent view and objective of the software and gives surety of fitness of the software. It involves testing of all components under the required services to confirm that whether it is satisfying the specified requirements or not. The process is also providing the client with information about the quality of the software.

Testing is mandatory because it will be a dangerous situation if the software fails any of time due to lack of testing. So, without testing software cannot be deployed to the end user.

What is Testing

Testing is a group of techniques to determine the correctness of the application under the predefined script but, testing cannot find all the defect of application. The main intent of testing is to detect failures of the application so that failures can be discovered and corrected. It does not demonstrate that a product functions properly under all conditions but only that it is not working in some specific conditions.

Testing furnishes comparison that compares the behavior and state of software against mechanisms because the problem can be recognized by the mechanism. The mechanism may include past versions of the same specified product, comparable products, and interfaces of expected purpose, relevant standards, or other criteria but not limited up to these.

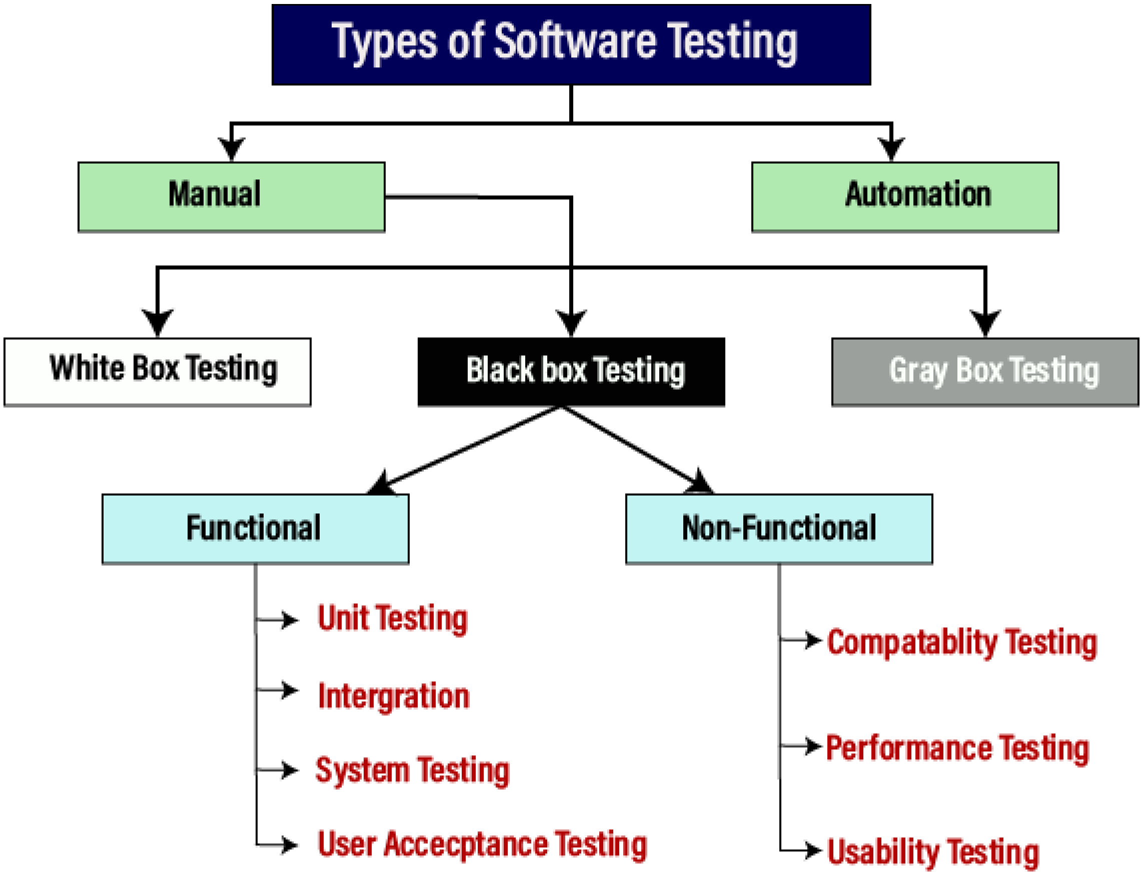
Testing includes an examination of code and also the execution of code in various environments, conditions as well as all the examining aspects of the code. In the current scenario of software development, a testing team may be separate from the development team so that Information derived from testing can be used to correct the process of software development.

The success of software depends upon acceptance of its targeted audience, easy graphical user interface, strong functionality load test, etc. For example, the audience of banking is totally different from the audience of a video game. Therefore, when an organization develops a software product, it can assess whether the software product will be beneficial to its purchasers and other audience.

Type of Software testing

We have various types of testing available in the market, which are used to test the application or the software.

With the help of below image, we can easily understand the type of software testing:



Manual testing

The process of checking the functionality of an application as per the customer needs without taking any help of automation tools is known as manual testing. While performing the manual testing on any application, we do not need any specific knowledge of any testing tool, rather than have a proper understanding of the product so we can easily prepare the test document.

Manual testing can be further divided into three types of testing, which are as follows:

- **White box testing**
- **Black box testing**
- **Gray box testing**

Automation testing

Automation testing is a process of converting any manual test cases into the test scripts with the help of automation tools, or any programming language is known as automation testing. With the help of automation testing, we can enhance the speed of our test execution because here, we do not require any human efforts. We need to write a test script and execute those scripts.

The main objectives of software testing

- To find any defects or bugs that may have been created when the software was being developed
- To increase confidence in the quality of the software
- To prevent defects in the final product
- To ensure the end product meets customer requirements as well as the company specifications
- To provide customers with a quality product and increase their confidence in the company

When it comes to testing software it's important to achieve the best possible results. But how do you know that the tests you are performing are the best ones? To help with following the right strategy for testing there are certain principles that need to be followed.

Software testing and the 7 testing principles

There are seven testing principles which are common in the software industry.

1. **Optimal testing –** it's not possible to test everything so it's important to determine the optimal amount. The decision is made using a risk assessment. This assessment will uncover the area that is most likely to fail and this is where testing should take place.

2. **Pareto Principle –** this principle states that approximately 80% of problems will be found in 20% of tests. However, there is a flaw in this principle in that repeating the same tests over and over again will mean no new bugs will be found.
3. **Review and Revise –** repeating the same tests will mean that the methods will eventually become useless for uncovering new defects. To prevent this from happens only requires the tests to be reviewed and revised on a regular basis. Adding new tests will help to find more defects.
4. **Defects that are present –** testing reduces the probability of the being a defect in the final product but does not guarantee that a defect won't be there. And even if you manage to make a product that's 99% bug free, the testing won't have shown whether the software meets the needs of clients.
5. **Meeting customer needs –** testing a product for the wrong requirements is foolhardy. Even if it is bug free it may still fail to meet customer requirements.
6. **Test early –** it's imperative that testing starts as soon as possible in the development of a product.
7. **Test in context –** test a product in accordance with how it will be used Software is not identical and will be developed to meet a certain need rather than a general one. Different techniques, methodologies, approach and type of testing can be used depending on the applications planned use.

Difference between Defect, Error, Bug, Failure and Fault!

Testing is the process of identifying defects, where a defect is any variance between actual and expected results. "A mistake in coding is called Error, error found by tester is called Defect, defect accepted by development team then it is called Bug, build does not meet the requirements then it Is Failure."

DEFECT:

It can be simply defined as a variance between expected and actual. The defect is an error found AFTER the application goes into production. It commonly refers to several troubles with the software products, with their external behavior or with its internal features. In other words, a Defect is a difference between expected and actual results in the context of testing. It is the deviation of the customer requirement.

Defect can be categorized into the following:

Wrong:

When requirements are implemented not in the right way. This defect is a variance from the given specification. It is Wrong!

Missing:

A requirement of the customer that was not fulfilled. This is a variance from the specifications, an indication that a specification was not implemented, or a requirement of the customer was not noted correctly.

Extra:

A requirement incorporated into the product that was not given by the end customer. This is always a variance from the specification, but maybe an attribute desired by the user of the product. However, it is considered a defect because it’s a variance from the existing requirements.

ERROR:

An error is a mistake, misconception, or misunderstanding on the part of a software developer. In the category of the developer, we include software engineers, programmers, analysts, and testers. For example, a developer may misunderstand a de-sign notation, or a programmer might type a variable name incorrectly – leads to an Error. It is the one that is generated because of the wrong login, loop or syntax. The error normally arises in software; it leads to a change in the functionality of the program.

Also Read : Specification Test Case Design Technique(Opens in a new browser tab)

BUG:

A bug is the result of a coding error. An Error found in the development environment before the product is shipped to the customer. A programming error that causes a program to work poorly, produce incorrect results or crash. An error in software or hardware that causes a program to malfunction. A bug is the terminology of Tester.

FAILURE:

A failure is the inability of a software system or component to perform its required functions within specified performance requirements. When a defect reaches the end customer it is called a Failure. During development, Failures are usually observed by testers.

FAULT:

An incorrect step, process or data definition in a computer program that causes the program to perform in an unintended or unanticipated manner. A fault is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification. It is the result of the error.

The software industry can still not agree on the definitions for all the above. In essence, if you use the term to mean one specific thing, it may not be understood to be that thing by your audience.

Difference between Bug, Defect, Error, Fault & Failure

In this section, we are going to discuss the difference between the **Bug, Defect, Error, Fault & Failure** as we understood that all the terms are used whenever the system or an application act abnormally.

Sometimes we call it an **error** and sometimes a bug or a **defect** and so on. In software testing, many of the new test engineers have confusion in using these terminologies.

Generally, we used these terms in the Software Development Life Cycle (SDLC) based on the phases. But there is a conflict in the usage of these terms.

In other words, we can say that in the era of **software testing,** the terms **bugs, defects, error, fault, and failure** come across every second of the day.

But for a beginner or the inexperienced in this field, all these terminologies may seem synonyms. It became essential to understand each of these terms independently if the software doesn't work as expected.

What is a bug?

In software testing, a bug is the informal name of defects, which means that software or application is not working as per the requirement. When we have some coding error, it leads a program to its breakdown, which is known as **a bug**. The **test engineers** use the terminology **Bug**.

If a **QA (Quality Analyst)** detect a bug, they can reproduce the bug and record it with the help of the **bug report template**.

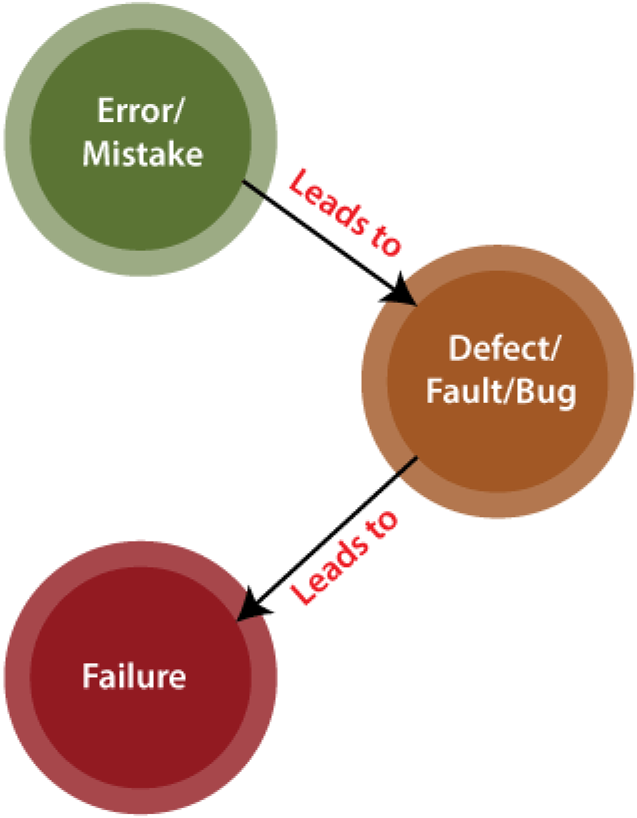
What is a Defect?

When the application is not working as per the requirement is knows as **defects**. It is specified as the aberration from the **actual and expected result** of the application or software.

In other words, we can say that the bug announced by the **programmer** and inside the code is called a **Defect.**

What is Error?

The Problem in code leads to errors, which means that a mistake can occur due to the developer's coding error as the developer misunderstood the requirement or the requirement was not defined correctly. The **developers** use the term **error**.



What is Fault?

The fault may occur in software because it has not added the code for fault tolerance, making an application act up.

A fault may happen in a program because of the following reasons:

- Lack of resources
- An invalid step
- Inappropriate data definition

What is Failure?

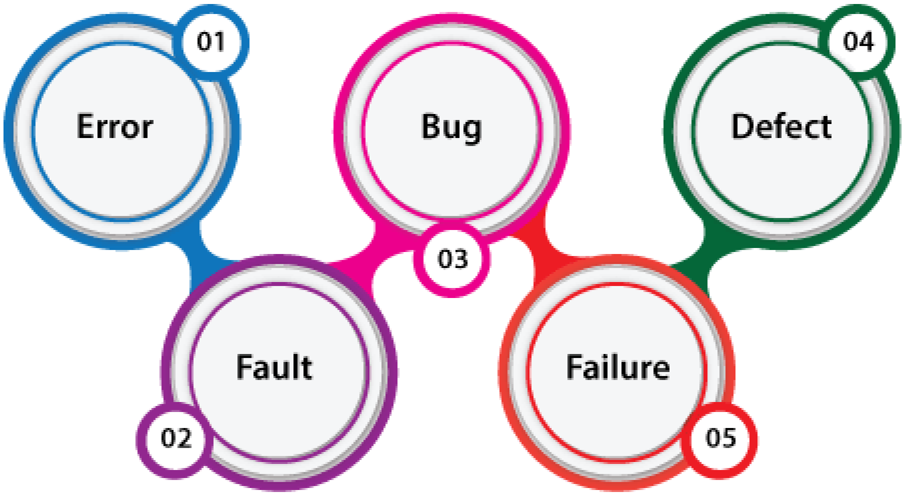
Many defects lead to the **software's failure**, which means that a loss specifies a fatal issue in software/ application or in its module, which makes the system unresponsive or broken.

In other words, we can say that if an end-user detects an issue in the product, then that particular issue is called a **failure**.

Possibilities are there one defect that might lead to one failure or several failures.

For example, in a bank application if the **Amount Transfer** module is not working for end-users when the end-user tries to **transfer money**, submit button is not working. Hence, this is a **failure**.

The flow of the above terminologies are shown in the following image:



Bug Vs. Defect Vs. Error Vs. Fault Vs. Failure

We have listed some of the vital differences between **bug, defect, error, fault, and failure in the below table**.

Comparison basis	Bug	Defect	Error	Fault	Failure
Definition	It is an informal name specified to the defect.	The **Defect** is the difference between the actual outcomes and expected outputs.	An **Error** is a mistake made in the code; that's why we cannot execute or compile code.	The **Fault** is a state that causes the software to fail to accomplish its essential function.	If the software has lots of defects, it leads to failure or causes failure.
Raised by	The **Test Engineers** submit the bug.	The **Testers** identify the defect. And it was also solved by the developer in the development phase or stage.	The **Developers and automation test engineers** raise the error.	**Human mistakes** cause fault.	The failure finds by the manual test engineer through the **development cycle**.

Different types	Different type of bugs are as follows: ○ Logic bugs ○ Algorithmic bugs ○ Resource bugs	Different type of Defects are as follows: Based on **priority**: ○ High ○ Medium ○ Low And based on the severity: ○ Critical ○ Major ○ Minor ○ Trivial	Different type of Error is as below: ○ Syntactic Error ○ User interface error ○ Flow control error ○ Error handling error ○ Calculation error ○ Hardware error ○ Testing Error	Different type of Fault are as follows: ○ Business Logic Faults ○ Functional and Logical Faults ○ Faulty GUI ○ Performance Faults ○ Security Faults ○ Software/ hardware fault	-----
Reasons behind	Following are reasons which may cause the **bugs:** Missing coding Wrong coding Extra coding	The below reason leads to the **defects**: Giving incorrect and wrong inputs. Dilemmas and errors in the outside behavior and inside structure and design. An error in coding or logic affects the software and causes it to breakdown or the failure.	The reasons for having an **error** are as follows: Errors in the code. The Mistake of some values. If a developer is unable **to compile or run a program successfully.** Confusions and issues in programming. Invalid login, loop, and syntax. Inconsistency between actual and expected outcomes. Blunders in design or requirement actions. Misperception in understanding the	The reasons behind the **fault** are as follows: A Fault may occur by an improper step in the initial stage, process, or data definition. Inconsistency or issue in the program. An irregularity or loophole in the software that leads the software to perform improperly.	Following are some of the most important reasons behind the **failure:** Environmental condition System usage Users Human error

			requirements of the application.		
Way to prevent the reasons	Following are the way to stop the **bugs**: Test-driven development. Offer programming language support. Adjusting, advanced, and operative development procedures. Evaluating the code systematically.	With the help of the following, we can prevent the **Defects**: Implementing several innovative programming methods. Use of primary and correct software development techniques. Peer review It is executing consistent code reviews to evaluate its quality and correctness.	Below are ways to prevent the **Errors**: Enhance the software quality with system review and programming. Detect the issues and prepare a suitable mitigation plan. Validate the fixes and verify their quality and precision.	The **fault** can be prevented with the help of the following: Peer review. Assess the functional necessities of the software. Execute the detailed code analysis. Verify the correctness of software design and programming.	The way to prevent **failure** are as follows: Confirm re-testing. Review the requirements and revisit the specifications. Implement current protective techniques. Categorize and evaluate errors and issues.

Conclusion

After seeing all the significant differences between **bug, defect, error, fault, and failure**, we can say that the several issues and inconsistencies found throughout software are linked and dependent on each other.

All the above terminology affects and change different parts of the software and differ from one another massively. However, all these differences between **bug, defect, errors, faults, and failures** slow down the software's excellence and performance.

Test cases and test suites

A *test case* answers the question: “What am I going to test?” You develop test cases to define the things that you must validate to ensure that the system is working correctly and is built with a high level of quality. A *test suite* is a collection of test cases that are grouped for test execution purposes.

Test cases

A typical use of test case might be to use the same test script for testing multiple configurations. For instance, if you want to test a login script on three different browsers, such as Firefox, Internet Explorer, and Safari, you can create three different test case execution records in that test case. In a test case that is called Test Browsers you might include three testing scenarios:

- Test case execution record 1: Firefox and log-in test script
- Test case execution record 2: Internet Explorer and log-in test script
- Test case execution record 3: Safari and log-in test script

Test suites

If each test case represents a piece of a scenario, such as the elements that simulate a completing a transaction, use a test suite. For instance, a test suite might contain four test cases, each with a separate test script:

- Test case 1: Login
- Test case 2: Add New Products
- Test case 3: Checkout
- Test case 4: Logout

Test suites can identify gaps in a testing effort where the successful completion of one test case must occur before you begin the next test case. For instance, you cannot add new products to a shopping cart before you successfully log in to the application. When you run a test suite in sequential mode, you can choose to stop the suite execution if a single test case does not pass. Stopping the execution is useful if running a test case in a test suite depends on the success of previous test cases.

Test suites are also useful for the following types of tests:

- Build verification tests: A collection of test cases that perform a basic validation of most the functional areas in the product. The tests are executed after each product build and before the build is promoted for use by a larger audience.
- Smoke tests: A collection of test cases that ensure basic product functionality. Typically, smoke tests are the first level of testing that is performed after changes are made to the system under test.

- End-to-End integration tests: A collection of test cases that cross product boundaries and ensure that the integration points between products are exercised and validated.
- Functional verification tests: A collection of test cases that focus on a specific product function. Executing this type of test with a test suite ensures that several aspects of a specific feature are tested.
- Regression tests: A collection of test cases that are used to make a regression pass over functional product areas.

Test scripts

Each test case is typically associated with a test script, although you can run a test with no associated test script.

A *test script* is a manual or automated script that contains the instructions for implementing a test case. You can write manual test scripts to be run by a human tester, or you can automate some or all of the instructions in the test script. You can also associate automated functional test scripts, performance test scripts, and security test scripts with a test case.

Test plan

Think of the test plan as a super document that lists everything that will be required in the project. According to Software Testing Help, this will often include all of the activities to be executed, the scope, roles, entry and exit criteria and test objectives. This deliverable acts as a roadmap for teams to follow and measure their progress against. It's a critical part in knowing how they are performing and if projects are staying on track with scheduling and user expectations. Test plan will also include features that need to be tested, testing tools and environment requirements, which are all essential to plan out prior to executing any testing processes.

Test strategy

The finer details of testing are included in the test strategy. This deliverable is often a part of the test plan, and outlines the testing approach that will be used to fulfill quality standards. The test strategy gives clear direction as to what types of tests will be used, but leaves some room to ensure that anyone is able to execute the test at any time. This type of information will give clear direction about what tasks need to be completed and allows teams to determine what tools will be the most effective in helping to achieve their goals.

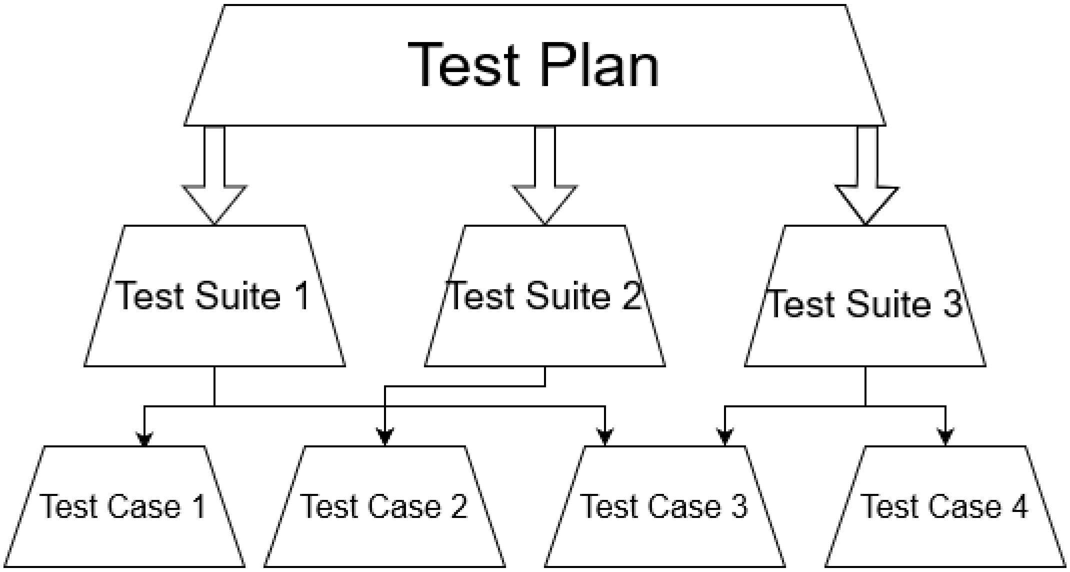
Test case

Unlike test plan and test strategy, a test case is far more specific. It's a sequence of steps that helps teams perform a test in the project. This document often includes conditions, environment, expected results, actual results and whether it passed or failed, according to Software Testing Guide. In many cases, an enterprise test management software can manage these elements and keep track of testing progress.

Testing processes have a number of deliverables that are essential, but many organizations are unable to utilize them effectively due to misunderstanding about what each asset entails. With the knowledge of the differences between these terms, quality assurance teams can create better plans and ensure that they are providing the right guidance to meet stakeholder objectives.

Test Suite, Test Case, Test Plan, What’s the Difference?

Test suites are sometimes wrongly referred to as a test plan, a test script, and sometimes a test scenario. Each of these are distinct documents that are related to one another.



What is a Test Case?

A regular use of a test case could be to leverage the same script for analyzing multiple settings. As an example, if you'd like to test an authentication script on three distinct browsers, such as Chrome, Firefox, and Edge, you could create three separate iterations within that test case. In a test case named Test Web Browsers you may include three testing situations:

- Iteration 1: login script with Chrome
- Iteration 2: login script with Firefox
- Iteration 3: login script with Edge

What is a Test Suite?

If each test case reflects a component of a scenario, like the functionality simulating completion of a sale, make use of a test suite. For example, a test suite may comprise five test cases, each having a separate test script:

- Test case 1: Login
- Test case 2: Add New Services to Cart
- Test case 3: Complete Order of Services
- Test case 4: Log-out

- Test case 5: Fulfill Service Request

Test suites could identify gaps inside a testing cycle where the successful conclusion of a single test case has to occur before starting another test case. As an example, you can't add new services to a cart until you log into the application. When you execute a test suite in a consecutive manner, you may decide to block the suite execution in case one test case doesn't pass. Preventing further execution is of good use if executing a test case in a test suite is contingent upon prior test cases passing.

Test suites are created based on the scope of the test plan, or based on where one is at in the test cycle. They could contain any kind of tests, functional or Non-Functional. Test suites can also be helpful for these kinds of tests:

- Smoke tests: A selection of test cases which ensure the application's basic functionality is working. On average, smoke tests are the initial testing that's conducted after changes are made to the application under test.
- Regression tests: A group of test cases which can be utilized to ensure breaking code changes have not occurred.
- End to end tests: A collection of test scripts which cross a product's bounds and make sure the integration between systems is exercised and confirmed.
- Functional tests: A group of test cases that are dedicated to particular requirements. Implementing this kind of test, using a test suite, means several areas of a particular feature are all tested.
- Build pipeline tests: A group of test cases that do a simple validation on all of the functional areas in the software. The scenarios need to be executed after every product build and until the build is promoted for use with a larger customer base.

In AccelaTest, you can create custom Test Suites, and nest test suites, by clicking the "Create Directory" button and giving your directory a name, such as "Regression Tests." Don't have your free AccelaTest account yet? Sign up now – did we mention Test Case Management is completely free?

What is a Test Plan?

A Test Plan a dynamic comprehensive record defining the test strategy, objectives, schedule, estimation, deliverables, and resources necessary to complete testing for a software product. Test Plans help determine the time required to confirm the quality of the software under test. The test plan acts as a blueprint to perform testing tasks in a specified process, that will be tracked and monitored with the test manager.

According to ISTQB definition: “Test Plan is a document describing the scope, approach, resources, and schedule of intended test activities.”

What is a Test Script?

Although you’ll be able to execute a test without an associated test script, it is best to associate each test case with a test script.

You may write manual test scripts to be run by an individual tester, or you can automate a few of the directions in the test script. You can link automated functional test scripts, load test scripts, and even security test scripts to a test case.

Can Test Cases have multiple Test Scripts?

In many cases, you might need to make use of test suites to combine related test cases. As an example, you may work with a suite to execute a pair of automated regression test cases, or to run a pair of test cases which comprise an end-to-end scenario. With a test suite, you’re able to organize, start, and then track the execution of associated test cases which compose the bigger scenario that’s tested by the test suite. You might even associate more than one test script with a test case.

However, you certainly should only associate multiple scripts when each test script gives another way to test the scenario in the test case. For instance, a test case may require multiple scripts to test the scenario in several different test environments. Or you have manual and automated scripts which could be utilized to run the test case.

Don’t use test cases to group test scripts which perform frequent functions as doing so may cause erroneous data regarding test results and attempted scenarios. As an example, should you execute a test case for a particular test plan, iteration, or test environment and define a unique functional script every time, the previous result of this test execution simply captures the results of the previous script execution.

In certain cases, you might also need to make use of test suites to group related test cases. As an example, you may make use of a suite to execute a few automated regression test cases, or run multiple automated E2E scenarios. With a test suite, it is possible to organize, start, and track the progress of a test case execution.

Verification vs Validation in Software: Overview & Key Differences

Verification vs Validation: Definitions

Software testing is a process of examining the functionality and behavior of the software through verification and validation.

- Verification is a process of determining if the software is designed and developed as per the specified requirements.
- Validation is the process of checking if the software (end product) has met the client's true needs and expectations.

Software testing is incomplete until it undergoes verification and validation processes. Verification and validation are the main elements of software testing workflow because they:

1. Ensure that the end product meets the design requirements.
2. Reduce the chances of defects and product failure.
3. Ensures that the product meets the quality standards and expectations of all stakeholders involved.

Most people confuse verification and validation; some use them interchangeably. People often mistake verification and validation because of a lack of knowledge on the purposes they fulfill and the pain points they address.

The software testing industry is estimated to grow from $40 billion in 2020 to $60 billion in 2027. Considering the steady growth of the software testing industry, we put together a guide that provides an in-depth explanation behind verification and validation and the main differences between these two processes.

Verification

As mentioned, verification is the process of determining if the software in question is designed and developed according to specified requirements. Specifications act as inputs for the software development process. The code for any software application is written based on the specifications document.

Verification is done to check if the software being developed has adhered to these specifications at every stage of the development life cycle. The verification ensures that the code logic is in line with specifications.

Depending on the complexity and scope of the software application, the software testing team uses different methods of verification, including inspection, code reviews, technical reviews, and walkthroughs. Software testing teams may also use mathematical models and calculations to make predictive statements about the software and verify its code logic.

Further, verification checks if the software team is building the product right. Verification is a continuous process that begins well in advance of validation processes and runs until the software application is validated and released.

The main advantages of the verification are:

1. It acts as a quality gateway at every stage of the software development process.
2. It enables software teams to develop products that meet design specifications and customer needs.
3. It saves time by detecting the defects at the early stage of software development.
4. It reduces or eliminates defects that may arise at the later stage of the software development process.

A walkthrough of verification of a mobile application

There are three phases in the verification testing of a mobile application development:

1. Requirements Verification
2. Design Verification
3. Code Verification

Requirements verification is the process of verifying and confirming that the requirements are complete, clear, and correct. Before the mobile application goes for design, the testing team verifies business requirements or customer requirements for their correctness and completeness.

Design verification is a process of checking if the design of the software meets the design specifications by providing evidence. Here, the testing team checks if layouts, prototypes, navigational charts, architectural designs, and database logical models of the mobile application meet the functional and non-functional requirements specifications.

Code verification is a process of checking the code for its completeness, correctness, and consistency. Here, the testing team checks if construction artifacts such as source code, user interfaces, and database physical model of the mobile application meet the design specification.

Validation

Validation is often conducted after the completion of the entire software development process. It checks if the client gets the product they are expecting. Validation focuses only on the output; it does not concern itself about the internal processes and technical intricacies of the development process.

Validation helps to determine if the software team has built the right product. Validation is a one-time process that starts only after verifications are completed. Software teams often use a wide range of validation methods, including White Box Testing (non-functional testing or structural/design testing) and Black Box Testing (functional testing).

White Box Testing is a method that helps validate the software application using a predefined series of inputs and data. Here, testers just compare the output values against the input values to verify if the application is producing output as specified by the requirements.

There are three vital variables in the Black Box Testing method (input values, output values, and expected output values). This method is used to verify if the actual output of the software meets the anticipated or expected output.

The main advantages of validation processes are:

1. It ensures that the expectations of all stakeholders are fulfilled.
2. It enables software teams to take corrective action if there is a mismatch between the actual product and the anticipated product.
3. It improves the reliability of the end-product.

A walkthrough of validation of a mobile application

Validation emphasizes checking the functionality, usability, and performance of the mobile application.

Functionality testing checks if the mobile application is working as expected. For instance, while testing the functionality of a ticket-booking application, the testing team tries to validate it through:

1. Installing, running, and updating the application from distribution channels like Google Play and the App Store
2. Booking tickets in the real-time environment (fields testing)
3. Interruptions testing

Usability testing checks if the application offers a convenient browsing experience. User interface and navigations are validated based on various criteria which include satisfaction, efficiency, and effectiveness.

Performance testing enables testers to validate the application by checking its reaction and speed under the specific workload. Software testing teams often use techniques such as load testing, stress testing, and volume testing to validate the performance of the mobile application.

Main differences between verification and validation

Verification and validation, while similar, are not the same. There are several notable differences between these two. Here is a chart that identifies the differences between verification and validation:

	Verification	**Validation**
Definition	It is a process of checking if a product is developed as per the specifications.	It is a process of ensuring that the product meets the needs and expectations of stakeholders.
What it tests or checks for	It tests the requirements, architecture, design, and code of the software product.	It tests the usability, functionalities, and reliability of the end product.
Coding requirement	It does not require executing the code.	It emphasizes executing the code to test the usability and functionality of the end product.

Activities include	A few activities involved in verification testing are requirements verification, design verification, and code verification.	The commonly-used validation activities in software testing are usability testing, performance testing, system testing, security testing, and functionality testing.
Types of testing methods	A few verification methods are inspection, code review, desk-checking, and walkthroughs.	A few widely-used validation methods are black box testing, white box testing, integration testing, and acceptance testing.
Teams or persons involved	The quality assurance (QA) team would be engaged in the verification process.	The software testing team along with the QA team would be engaged in the validation process.
Target of test	It targets internal aspects such as requirements, design, software architecture, database, and code.	It targets the end product that is ready to be deployed.

Verification and validation are an integral part of software engineering. Without rigorous verification and validation, a software team may not be able to build a product that meets the expectations of stakeholders. Verification and validation help reduce the chances of product failure and improve the reliability of the end product.

Different project management and software development methods use verification and validation in different ways. For instance, both verification and validation happen simultaneously in agile development methodology due to the need for continuous refinement of the system based on the end-user feedback.

Testers can use automation tools developed with low code development to streamline the processes of verification and validation.

Discover how the Process Director, a workflow automation software from BPLogix, can help automate your software testing process.

Verification is the process of checking that a software achieves its goal without any bugs. It is the process to ensure whether the product that is developed is right or not. It verifies whether the developed product fulfills the requirements that we have. Verification is static testing.

Verification means **Are we building the product right?**

Validation is the process of checking whether the software product is up to the mark or in other words product has high level requirements. It is the process of checking the validation of product i.e. it checks what we are developing is the right product. it is validation of actual and expected product. Validation is the dynamic testing.

Validation means **Are we building the right product?**

The difference between Verification and Validation is as follow:

Verification	Validation
It includes checking documents, design, codes and programs.	It includes testing and validating the actual product.
Verification is the static testing.	Validation is the dynamic testing.
It does *not* include the execution of the code.	It includes the execution of the code.
Methods used in verification are reviews, walkthroughs, inspections and desk-checking.	Methods used in validation are Black Box Testing, White Box Testing and non-functional testing.
It checks whether the software conforms to specifications or not.	It checks whether the software meets the requirements and expectations of a customer or not.
It can find the bugs in the early stage of the development.	It can only find the bugs that could not be found by the verification process.
The goal of verification is application and software architecture and specification.	The goal of validation is an actual product.

Verification	Validation
Quality assurance team does verification.	Validation is executed on software code with the help of testing team.
It comes before validation.	It comes after verification.
It consists of checking of documents/files and is performed by human.	It consists of execution of program and is performed by computer.

Differences between Black Box Testing vs White Box Testing

What is Black Box testing?

In Black-box testing, a tester doesn't have any information about the internal working of the software system. Black box testing is a high level of testing that focuses on the behavior of the software. It involves testing from an external or end-user perspective. Black box testing can be applied to virtually every level of software testing: unit, integration, system, and acceptance.

What is White Box testing?

White-box testing is a testing technique which checks the internal functioning of the system. In this method, testing is based on coverage of code statements, branches, paths or conditions. White-Box testing is considered as low-level testing. It is also called glass box, transparent box, clear box or code base testing. The white-box Testing method assumes that the path of the logic in a unit or program is known.

Software Testing can be majorly classified into two categories:

1. **Black Box Testing** is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester. Only the external design and structure are tested.

2. **White Box Testing** is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. Implementation and impact of the code are tested.

Differences between Black Box Testing vs White Box Testing:

S. No.	Black Box Testing	White Box Testing
1.	It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
2.	Implementation of code is not needed for black box testing.	Code implementation is necessary for white box testing.
3.	It is mostly done by software testers.	It is mostly done by software developers.
4.	No knowledge of implementation is needed.	Knowledge of implementation is required.
5.	It can be referred to as outer or external software testing.	It is the inner or the internal software testing.
6.	It is a functional test of the software.	It is a structural test of the software.

S. No.	Black Box Testing	White Box Testing
7.	This testing can be initiated based on the requirement specifications document.	This type of testing of software is started after a detail design document.
8.	No knowledge of programming is required.	It is mandatory to have knowledge of programming.
9.	It is the behavior testing of the software.	It is the logic testing of the software.
10.	It is applicable to the higher levels of testing of software.	It is generally applicable to the lower levels of software testing.
11.	It is also called closed testing.	It is also called as clear box testing.
12.	It is least time consuming.	It is most time consuming.
13.	It is not suitable or preferred for algorithm testing.	It is suitable for algorithm testing.
14.	Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.
15.	**Example:** Search something on google by using keywords	**Example:** By input to check and verify loops
16.	**Black-box test design techniques-** • Decision table testing • All-pairs testing • Equivalence partitioning • Error guessing	**White-box test design techniques-** • Control flow testing • Data flow testing • Branch testing
17.	**Types of Black Box Testing:** • Functional Testing • Non-functional testing	**Types of White Box Testing:** • Path Testing • Loop Testing

S. No.	Black Box Testing	White Box Testing
	• Regression Testing	• Condition testing
18.	It is less exhaustive as compared to white box testing.	It is comparatively more exhaustive than black box testing.

Software Testing – Boundary Value Analysis

Functional testing is a type of software testing in which the system is tested against the functional requirements of the system. It is conducted to ensure that the requirements are properly satisfied by the application. Functional testing verifies that each function of the software application works in conformance with the requirement and specification. Boundary Value Analysis(BVA) is one of the functional testings.

Boundary Value Analysis

Boundary Value Analysis is based on testing the boundary values of valid and invalid partitions. The behavior at the edge of the equivalence partition is more likely to be incorrect than the behavior within the partition, so boundaries are an area where testing is likely to yield defects.
It checks for the input values near the boundary that have a higher chance of error. Every partition has its maximum and minimum values and these maximum and minimum values are the boundary values of a partition.

Note:

- A boundary value for a valid partition is a valid boundary value.
- A boundary value for an invalid partition is an invalid boundary value.
- For each variable we check-
    - Minimum value.
    - Just above the minimum.
    - Nominal Value.
    - Just below Max value.

- Max value.

Example: Consider a system that accepts ages from 18 to 56.

Boundary Value Analysis(Age accepts 18 to 56)

Invalid	Valid	Invalid
(min-1)	(min, min + 1, nominal, max – 1, max)	(max + 1)
17	18, 19, 37, 55, 56	57

Valid Test cases: Valid test cases for the above can be any value entered greater than 17 and less than 57.

- Enter the value- 18.
- Enter the value- 19.
- Enter the value- 37.
- Enter the value- 55.
- Enter the value- 56.

Invalid Testcases: When any value less than 18 and greater than 56 is entered.

- Enter the value- 17.
- Enter the value- 57.

Single Fault Assumption: When more than one variable for the same application is checked then one can use a single fault assumption. Holding all but one variable to the extreme value and allowing the remaining variable to take the extreme value. For n variable to be checked:

Maximum of 4n+1 test cases

Problem: Consider a Program for determining the Previous Data.

Input: *Day, Month, Year with valid ranges as-*
1 ≤ Month≤12
1 ≤ Day ≤31
1900 ≤ Year ≤ 2000

Design Boundary Value Test Cases.

Solution: Taking the year as a Single Fault Assumption i.e. year will be having values varying from 1900 to 2000 and others will have nominal values.

Test Cases	Month	Day	Year	Output
1	6	15	1990	14 June 1990

Test Cases	Month	Day	Year	Output
2	6	15	1901	14 June 1901
3	6	15	1960	14 June 1960
4	6	15	1999	14 June 1999
5	6	15	2000	14 June 2000

Taking Day as Single Fault Assumption i.e. Day will be having values varying from 1 to 31 and others will have nominal values.

Test Case	Month	Day	Year	Output
6	6	1	1960	31 May 1960
7	6	2	1960	1 June 1960
8	6	30	1960	29 June 1960
9	6	31	1960	Invalid day

Taking Month as Single Fault Assumption i.e. Month will be having values varying from 1 to 12 and others will have nominal values.

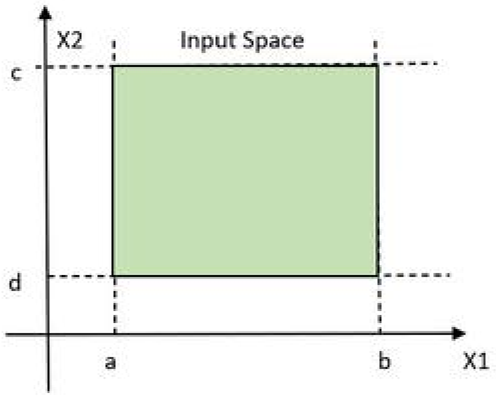
Test Case	Month	Day	Year	Output
10	1	15	1960	14 Jan 1960
11	2	15	1960	14 Feb 1960
12	11	15	1960	14 Nov 1960

Test Case	Month	Day	Year	Output
13	12	15	1960	14 Dec 1960

For the n variable to be checked Maximum of 4n + 1 test case will be required. Therefore, for n = 3, the maximum test cases are-

4 × 3 + 1 =13

The focus of BVA: BVA focuses on the input variable of the function. Let's define two variables X1 and X2, where X1 lies between a and b and X2 lies between c and d.



Showing legitimate domain

The idea and motivation behind BVA are that errors tend to occur near the extremes of the variables. The defect on the boundary value can be the result of countless possibilities.

Typing of Languages: BVA is not suitable for free-form languages such as COBOL and FORTRAN, These languages are known as weakly typed languages. This can be useful and can cause bugs also.
PASCAL, ADA is the strongly typed language that requires all constants or variables defined with an associated data type.

Limitation of Boundary Value Analysis:

- It works well when the product is under test.
- It cannot consider the nature of the functional dependencies of variables.
- BVA is quite rudimentary.

Equivalence Partitioning

It is a type of black-box testing that can be applied to all levels of software testing. In this technique, input data are divided into the equivalent partitions that can be used to derive test cases-

- In this input data are divided into different equivalence data classes.
- It is applied when there is a range of input values.

Example: Below is the example to combine Equivalence Partitioning and Boundary Value.
Consider a field that accepts a minimum of 6 characters and a maximum of 10 characters. Then the partition of the test cases ranges 0 – 5, 6 – 10, 11 – 14.

Test Scenario	Test Description	Expected Outcome
1	Enter value 0 to 5 character	Not accepted
2	Enter 6 to 10 character	Accepted
3	Enter 11 to 14 character	Not Accepted

Why Combine Equivalence Partitioning and Boundary Analysis Testing: Following are some of the reasons why to combine the two approaches:

- In this test cases are reduced into manageable chunks.
- The effectiveness of the testing is not compromised on test cases.
- Works well with a large number of variables.

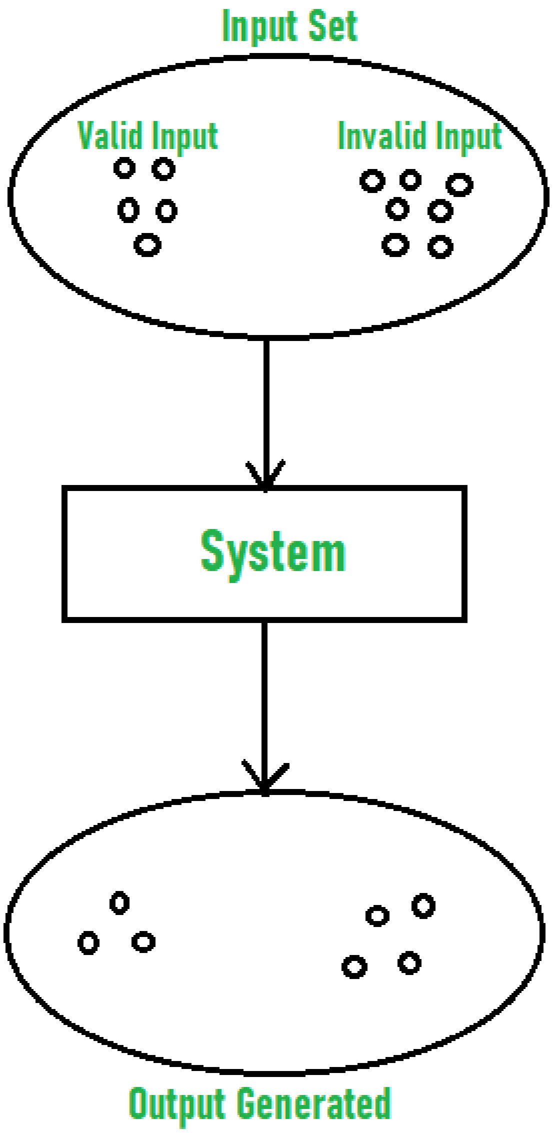
Equivalence Partitioning Method

Equivalence Partitioning Method is also known as Equivalence class partitioning (ECP). It is a software testing technique or black-box testing that divides input domain into classes of data, and with the help of these classes of data, test cases can be derived. An ideal test case identifies class of error that

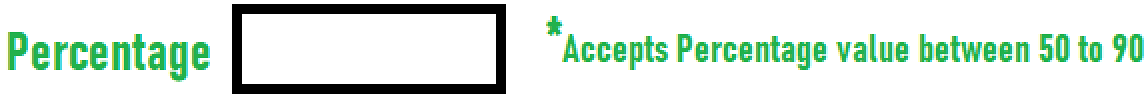
might require many arbitrary test cases to be executed before general error is observed.
In equivalence partitioning, equivalence classes are evaluated for given input conditions. Whenever any input is given, then type of input condition is checked, then for this input conditions, Equivalence class represents or describes set of valid or invalid states.

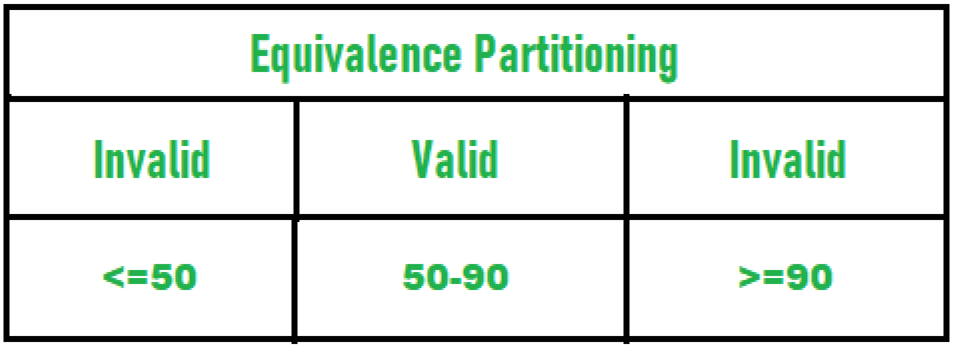
Guidelines for Equivalence Partitioning :

- If the range condition is given as an input, then one valid and two invalid equivalence classes are defined.
- If a specific value is given as input, then one valid and two invalid equivalence classes are defined.
- If a member of set is given as an input, then one valid and one invalid equivalence class is defined.
- If Boolean no. is given as an input condition, then one valid and one invalid equivalence class is defined.



Example-1:
Let us consider an example of any college admission process. There is a college that gives admissions to students based upon their percentage. Consider percentage field that will accept percentage only between 50 to 90 %, more and even less than not be accepted, and application will redirect user to an error page. If percentage entered by user is less than 50 %or more than 90 %, that equivalence partitioning method will show an invalid percentage. If percentage entered is between 50 to 90 %, then equivalence partitioning method will show valid percentage.



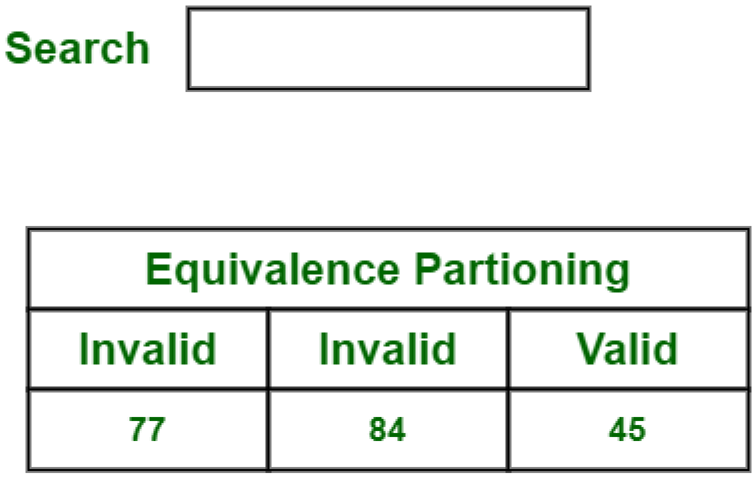


Equivalence Partitioning		
Invalid	Valid	Invalid
<=50	50-90	>=90

Example 2:
Let us consider an example of an online shopping site. In this site, each of products has a specific product ID and product name. We can search for product either by using name of product or by product ID. Here, we consider search field that accepts only valid product ID or product name.
Let us consider a set of products with product IDs and users wants to search for Mobiles. Below is a table of some products with their product Id.

Product	Product ID
Mobiles	45
Laptops	54
Pen Drives	67
Keyboard	76
Headphones	34

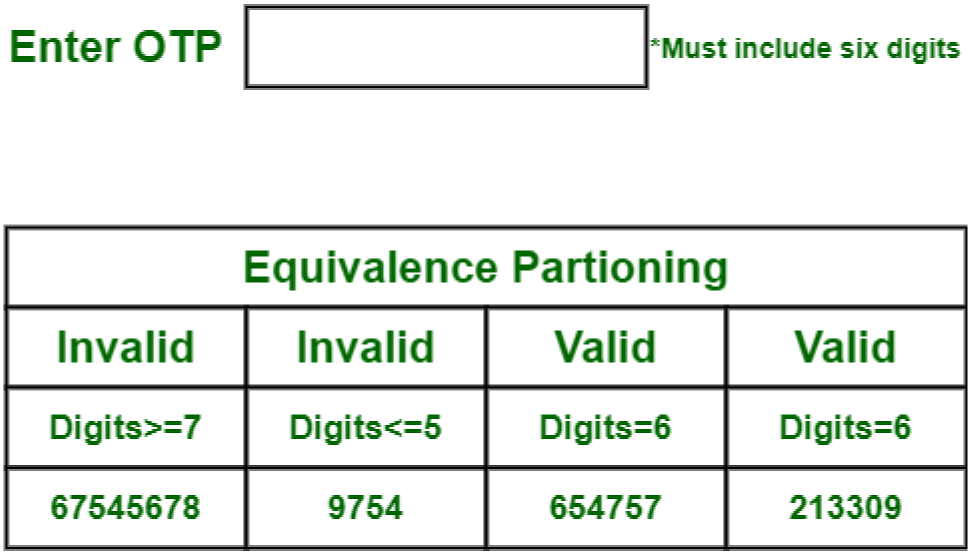
If the product ID entered by user is invalid then application will redirect customer or user to error page. If product ID entered by user is valid i.e. 45 for mobile, then equivalence partitioning method will show a valid product ID.



Search

Equivalence Partioning		
Invalid	Invalid	Valid
77	84	45

Example-3 :
Let us consider an example of software application. There is function of software application that accepts only particular number of digits, not even greater or less than that particular number.
Consider an OTP number that contains only 6 digit number, greater and even less than six digits will not be accepted, and the application will redirect customer or user to error page. If password entered by user is less or more than six characters, that equivalence partitioning method will show an invalid OTP. If password entered is exactly six characters, then equivalence partitioning method will show valid OTP.



Enter OTP *Must include six digits

Equivalence Partioning			
Invalid	Invalid	Valid	Valid
Digits>=7	Digits<=5	Digits=6	Digits=6
67545678	9754	654757	213309

Basis Path Testing in Software Testing

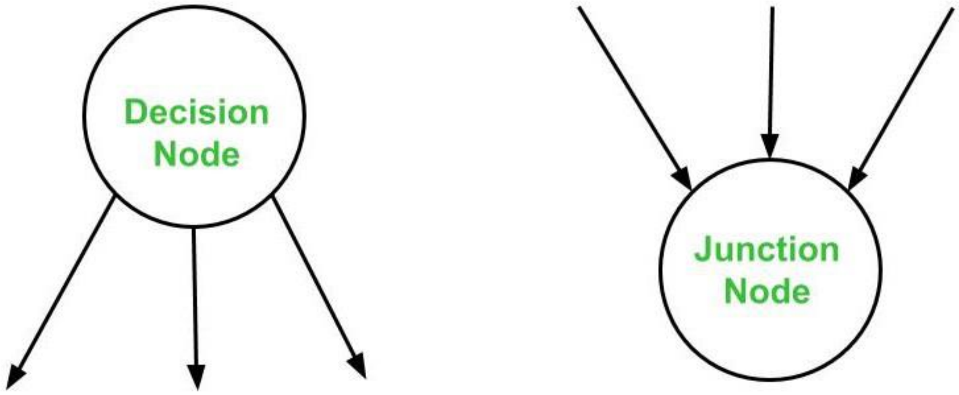
Basis Path Testing is a white-box testing technique based on the control structure of a program or a module. Using this structure, a control flow graph is prepared and the various possible paths present in the graph are executed as a part of testing. Therefore, by definition,
Basis path testing is a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

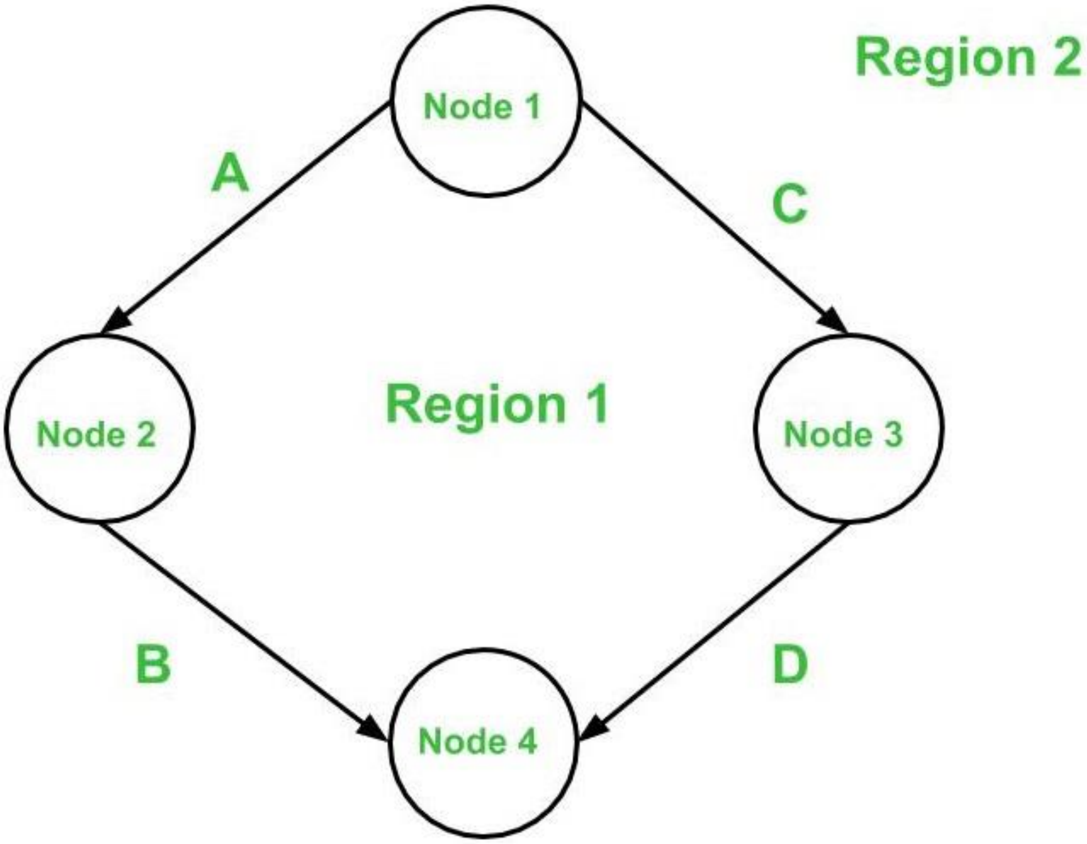
1. Construct the Control Flow Graph
2. Compute the Cyclomatic Complexity of the Graph
3. Identify the Independent Paths
4. Design Test cases from Independent Paths

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 :

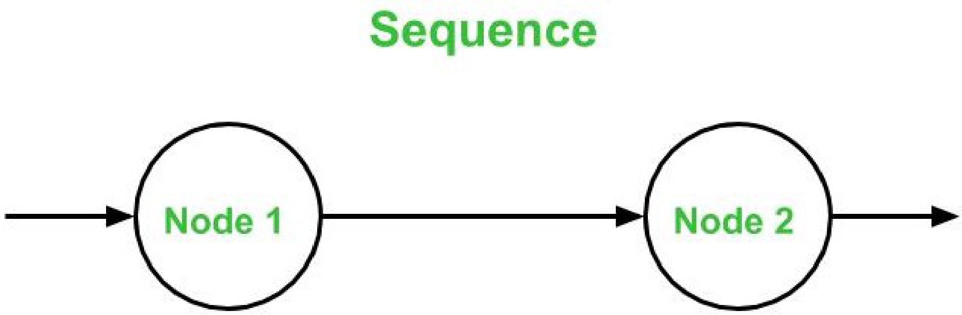
- **Junction Node –** a node with more than one arrow entering it.
- **Decision Node –** a node with more then one arrow leaving it.
- **Region –** area bounded by edges and nodes (area outside the graph is also counted as a region.).



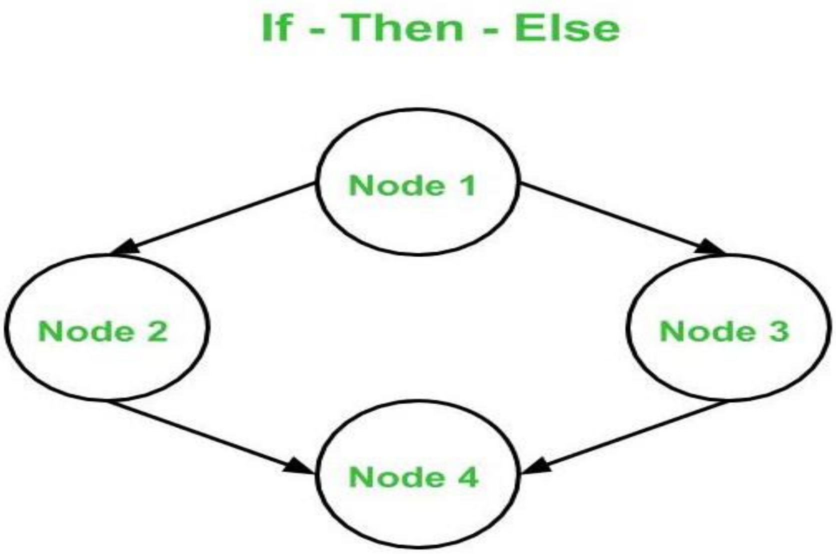


Below are the **notations** used while constructing a flow graph :

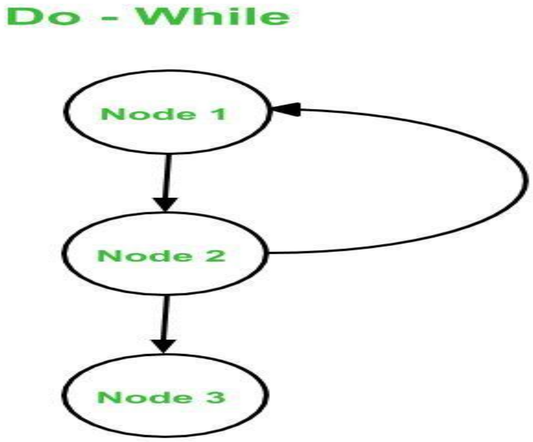
- **Sequential Statements –**



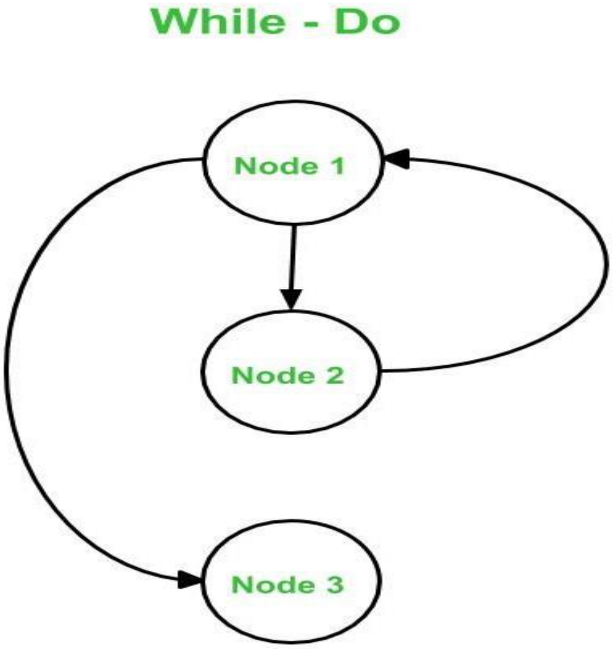
- **If – Then – Else –**



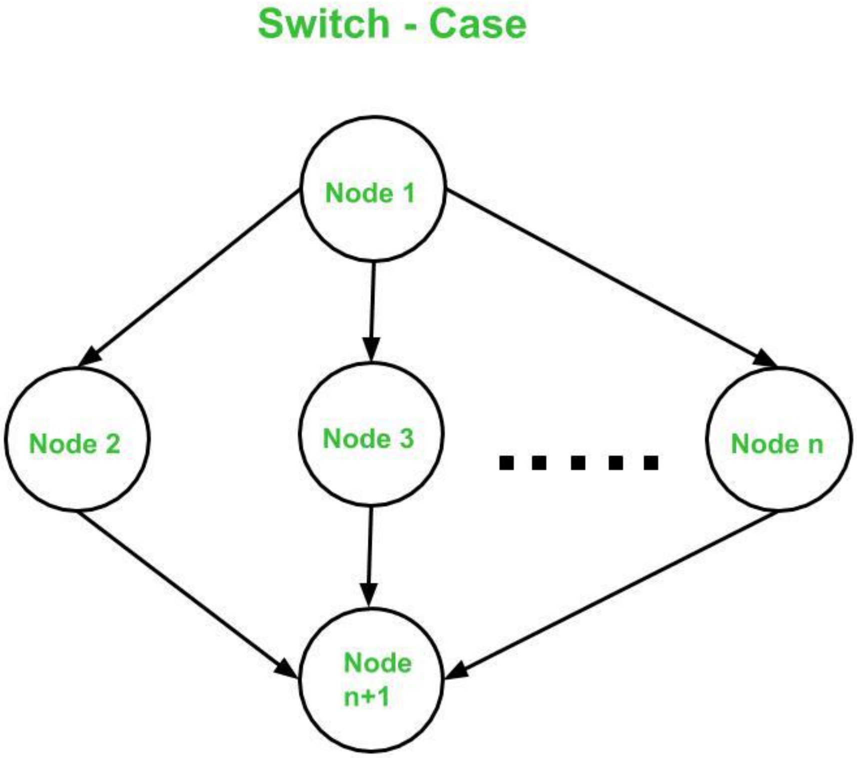
- **Do – While –**



- **While – Do –**



- **Switch – Case –**



Cyclomatic Complexity

Cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program. It is computed using the Control Flow Graph of the program. The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if second command might immediately follow the first command.

For example, if source code contains no control flow statement then its cyclomatic complexity will be 1 and source code contains a single path in it. Similarly, if the source code contains one **if condition** then cyclomatic complexity will be 2 because there will be two paths one for true and the other for false.

Mathematically, for a structured program, the directed graph inside control flow is the edge joining two basic blocks of the program as control may pass from first to second.
So, cyclomatic complexity M would be defined as,

$$M = E - N + 2P$$

where,
E = the number of edges in the control flow graph
N = the number of nodes in the control flow graph
P = the number of connected components

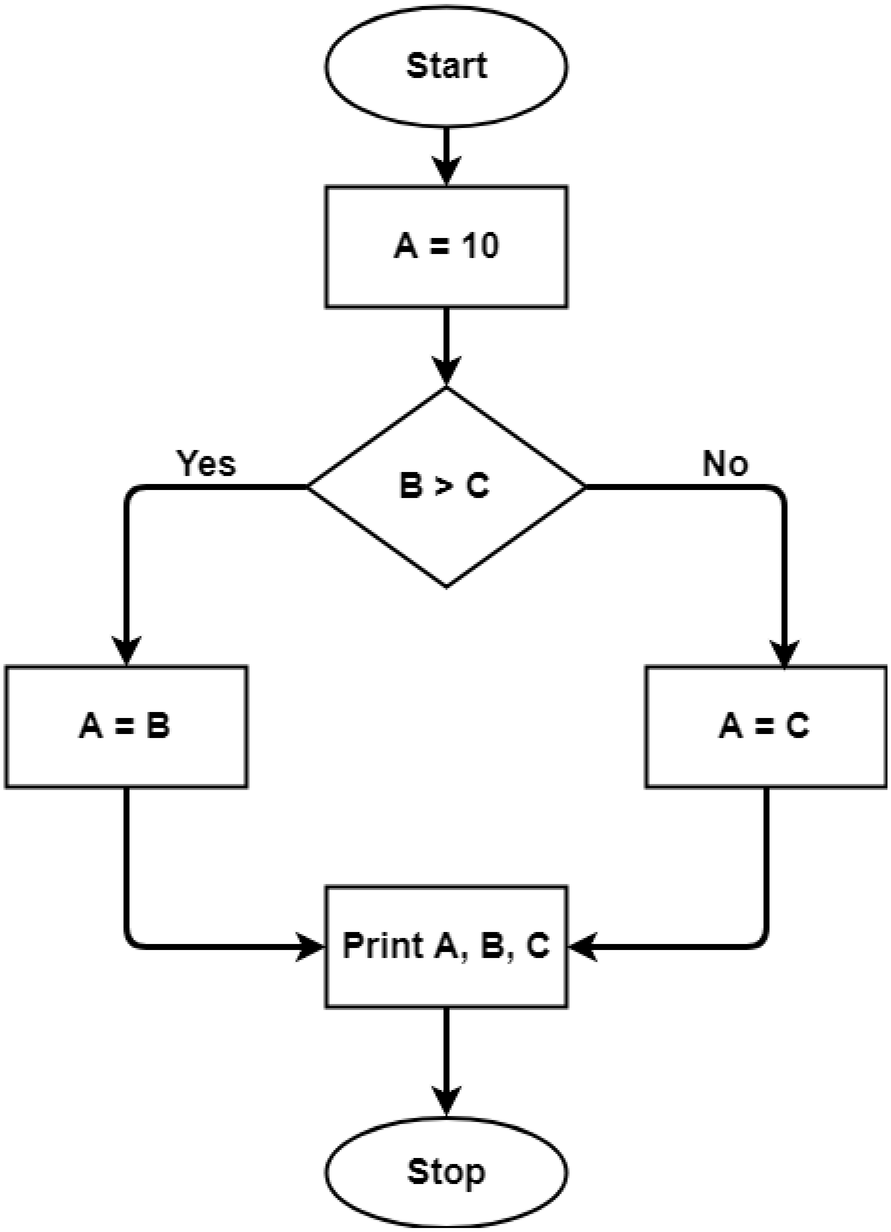
Steps that should be followed in calculating cyclomatic complexity and test cases design are:

- Construction of graph with nodes and edges from code.
- Identification of independent paths.
- Cyclomatic Complexity Calculation
- Design of Test Cases

Let a section of code as such:

```
A = 10

   IF B > C THEN

      A = B

   ELSE

      A = C

   ENDIF

Print A

Print B

Print C
```

Control Flow Graph of above code



The cyclomatic complexity calculated for above code will be from control flow graph. The graph shows seven shapes(nodes), seven lines(edges), hence cyclomatic complexity is 7-7+2 = 2.

Use of Cyclomatic Complexity:

- Determining the independent path executions thus proven to be very helpful for Developers and Testers.
- It can make sure that every path have been tested at least once.
- Thus help to focus more on uncovered paths.
- Code coverage can be improved.
- Risk associated with program can be evaluated.

- These metrics being used earlier in the program helps in reducing the risks.

Advantages of Cyclomatic Complexity:.

- It can be used as a quality metric, gives relative complexity of various designs.
- It is able to compute faster than the Halstead's metrics.
- It is used to measure the minimum effort and best areas of concentration for testing.
- It is able to guide the testing process.
- It is easy to apply.

Disadvantages of Cyclomatic Complexity:

- It is the measure of the programs's control complexity and not the data complexity.
- In this, nested conditional structures are harder to understand than non-nested structures.
- In case of simple comparisons and decision structures, it may give a misleading figure.

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
```

3. **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:

1. **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.
2. **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.
3. **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.
4. **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.
5. **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.

Cyclomatic Complexity | Calculation | Examples

Cyclomatic Complexity-

Cyclomatic Complexity may be defined as-

- It is a software metric that measures the logical complexity of the program code.
- It counts the number of decisions in the given program code.
- It measures the number of linearly independent paths through the program code.

Cyclomatic complexity indicates several information about the program code-

Cyclomatic Complexity	**Meaning**
1 – 10	• Structured and Well Written Code • High Testability • Less Cost and Effort
10 – 20	• Complex Code • Medium Testability • Medium Cost and Effort
20 – 40	• Very Complex Code • Low Testability • High Cost and Effort
> 40	• Highly Complex Code • Not at all Testable • Very High Cost and Effort

Importance of Cyclomatic Complexity-

- It helps in determining the software quality.
- It is an important indicator of program code's readability, maintainability and portability.
- It helps the developers and testers to determine independent path executions.
- It helps to focus more on the uncovered paths.
- It evaluates the risk associated with the application or program.
- It provides assurance to the developers that all the paths have been tested at least once.

Properties of Cyclomatic Complexity-

- It is the maximum number of independent paths through the program code.
- It depends only on the number of decisions in the program code.
- Insertion or deletion of functional statements from the code does not affect its cyclomatic complexity.
- It is always greater than or equal to 1.

Calculating Cyclomatic Complexity-

Cyclomatic complexity is calculated using the control flow representation of the program code.

In control flow representation of the program code,

- Nodes represent parts of the code having no branches.
- Edges represent possible control flow transfers during program execution

There are 3 commonly used methods for calculating the cyclomatic complexity-

Method-01:

Cyclomatic Complexity = Total number of closed regions in the control flow graph + 1

Method-02:

Cyclomatic Complexity = E – N + 2

Here-

- E = Total number of edges in the control flow graph
- N = Total number of nodes in the control flow graph

Method-03:

Cyclomatic Complexity = P + 1

Here,

P = Total number of predicate nodes contained in the control flow graph

Note-

- Predicate nodes are the conditional nodes.
- They give rise to two branches in the control flow graph.

PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY-

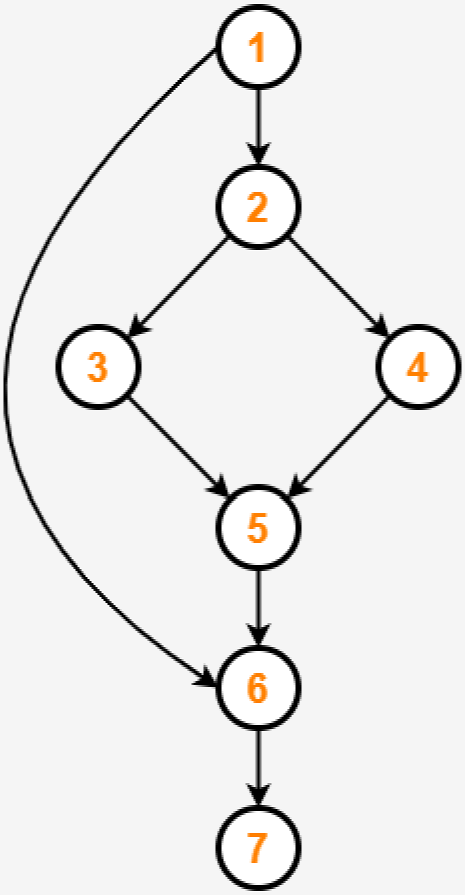
Problem-01:

Calculate cyclomatic complexity for the given code-

```
IF A = 354

THEN IF B > C

THEN A = B

ELSE A = C

END IF

END IF

PRINT A
```

Solution-

We draw the following control flow graph for the given code-



Control Flow Graph

Using the above control flow graph, the cyclomatic complexity may be calculated as-

Method-01:

Cyclomatic Complexity

= Total number of closed regions in the control flow graph + 1

= 2 + 1

= 3

Method-02:

Cyclomatic Complexity

= E – N + 2

= 8 – 7 + 2

= 3

Method-03:

Cyclomatic Complexity

= P + 1

= 2 + 1

= 3

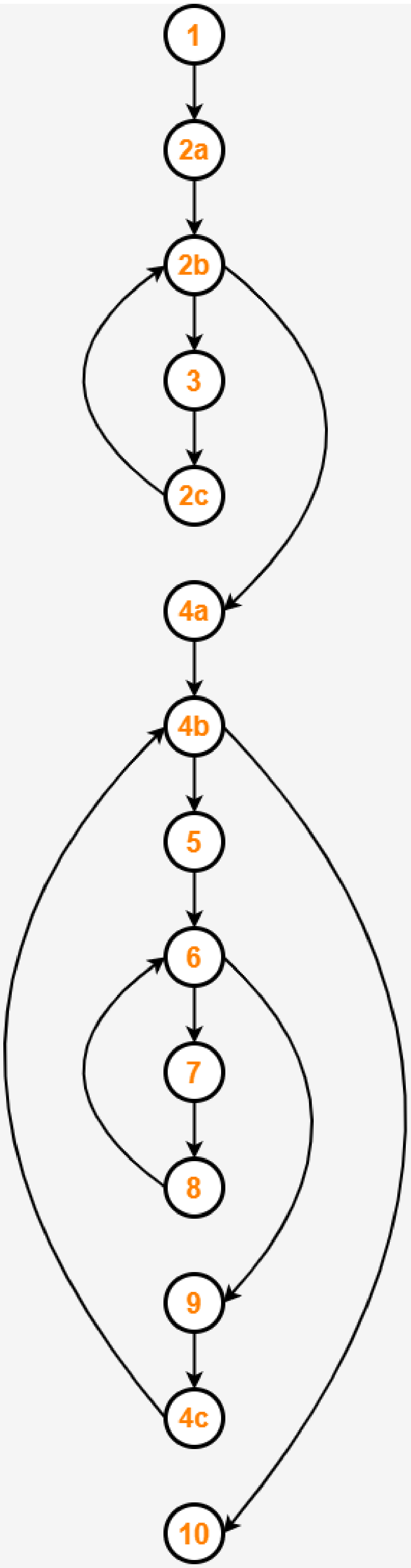
Problem-02:

Calculate cyclomatic complexity for the given code-

```
{ int i, j, k;

for (i=0 ; i<=N ; i++)

p[i] = 1;

for (i=2 ; i<=N ; i++)

{

k = p[i]; j=1;

while (a[p[j-1]] > a[k] {

p[j] = p[j-1];

j--;

}

p[j]=k;

}
```

Solution-

We draw the following control flow graph for the given code-



Control Flow Graph

Using the above control flow graph, the cyclomatic complexity may be calculated as-

Method-01:

Cyclomatic Complexity

= Total number of closed regions in the control flow graph + 1

= 3 + 1

= 4

Method-02:

Cyclomatic Complexity

= E – N + 2

= 16 – 14 + 2

= 4

Method-03:

Cyclomatic Complexity

= P + 1

= 3 + 1

= 4

Problem-03:

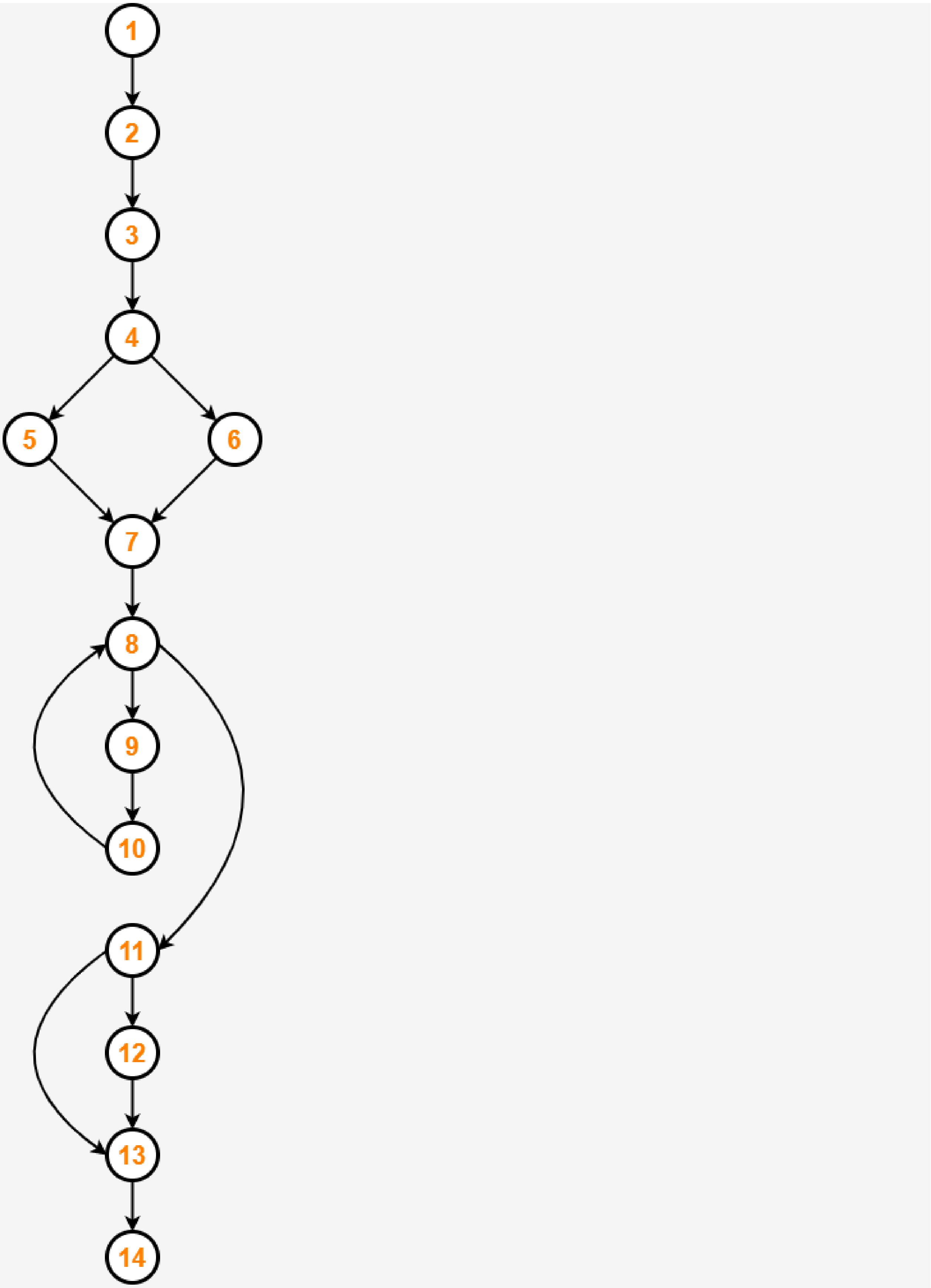
Calculate cyclomatic complexity for the given code-

```
begin int x, y, power;

float z;

input(x, y);

if(y<0)
```

```
power = -y;

else power = y;

z=1;

while(power!=0)

{ z=z*x;

power=power-1;

} if(y<0)

z=1/z;

output(z);

end
```

Solution-

We draw the following control flow graph for the given code-


1
2
3
4
5
6
7
8
9
10
11
12
13
14

Using the above control flow graph, the cyclomatic complexity may be calculated as-

Method-01:

Cyclomatic Complexity

= Total number of closed regions in the control flow graph + 1

= 3 + 1

= 4

Method-02:

Cyclomatic Complexity

= E – N + 2

= 16 – 14 + 2

= 4

Method-03:

Cyclomatic Complexity

= P + 1

= 3 + 1

= 4

To gain better understanding about Cyclomatic Complexity,

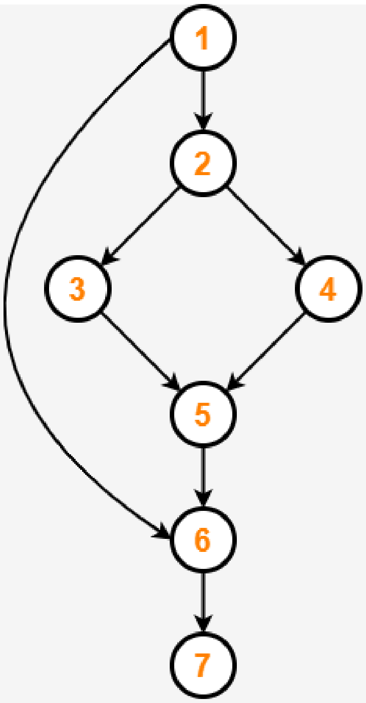
Watch this Video Lecture

Next Article- Cause Effect Graph Technique

Get more notes and other study material of **Software Engineering**.

Watch video lectures by visiting our YouTube channel **LearnVidFun**.

Summary



Control Flow Graph