Chapter 5
Testing Techniques and Maintenance

5.0 Testing

Software testing is the process of evaluating a software application to identify and fix bugs, ensure quality, and verify that it meets the specified requirements. Testing is a critical activity in the software development lifecycle (SDLC), as it ensures the delivery of a reliable, efficient, and error-free product.

Goals:

The testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements.
2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification.

Objectives of Software Testing

1. **Verification and Validation (V&V):**
   - **Verification**: Ensures the product is being developed correctly (conforms to specifications).
   - **Validation**: Ensures the product fulfills the intended purpose (meets user needs).
2. **Defect Detection**:
   - Identify and fix bugs before deployment to ensure quality and reliability.
3. **Quality Assurance**:
   - Ensure the software performs as expected under various conditions and meets performance benchmarks.
4. **Risk Mitigation**:
   - Reduce the risk of failures by testing potential edge cases, scalability, and performance bottlenecks.

White Box and Black Box testing

White Box Testing (also called **G**lass Box Testing, Clear Box Testing, or Structural Testing) focuses on testing the internal structure, logic, and code of the software.

- The tester has full knowledge of the code.

- It verifies the internal paths, conditions, loops, and statements.
- **Objective**: To ensure that internal operations (code logic, control flow) work as expected.
- **Techniques**: Unit testing, Control flow testing, Data flow testing Path testing, Loop testing, Statement/branch testing

Black Box Testing focuses on testing the software's functionality without knowledge of its internal code.

- The tester validates the input-output behavior.
- It checks the system against requirements and specifications.
- **Objective**: To ensure the system functions as expected from the **user's perspective**.
- **Techniques**: Equivalence partitioning, BVA, Decision table testing, Error guessing

5.1 Validation and Verification

Testing is part of a broader process of software verification and validation (V & V). Verification and validation are not the same thing, although they are often confused.

Let's take a simple analogy of cooking food to understand these terms:

- **Verification:** Checking the recipe while cooking to ensure you followed the steps correctly.
- **Validation:** Tasting the food to ensure it is delicious and meets your expectations.

Figure: Verification vs Validation in Software Testing

1. Verification:

- **Definition:** Verification is the process of checking whether the software or system is being built correctly, i.e., ensuring that the product complies with its specifications and design documents.
- **Objective:** To ensure the product matches the design and requirements documentation.
- **Focus:** Process-oriented (Are we building the product correctly?).

- **Methods:** Reviews, inspections, walkthroughs, and static testing.
- Example of Verification:
    - Suppose you are developing an online shopping cart application:
    - You verify that:
        - The "Add to Cart" button is functional as per the design.
        - The database schema matches the specification.
        - The coding adheres to the development guidelines and standards.

2. Validation:

- **Definition:** Validation is the process of checking whether the software or system meets the user's needs and expectations, i.e., ensuring that the product fulfills its intended purpose.
- **Objective:** To ensure the product does what the customer expects it to do.
- **Focus:** Product-oriented (Are we building the right product?).
- **Methods:** Functional testing, user acceptance testing (UAT), and system testing.
- Example of Validation:
    - Continuing with the online shopping cart application:
    - You validate that:
        - Users can successfully add items to the cart and proceed to checkout.
        - The total price is calculated correctly, including taxes and discounts.
        - The application works seamlessly in a live environment with actual users.

Key Differences:

Aspect	Verification	Validation
Purpose	Are we building the product right?	Are we building the right product?
Focus	Adherence to specifications	User needs and expectations
Methods	Reviews, inspections, walkthroughs	Testing, UAT, simulations
When Performed	During development	After development or in the live environment

Software Inspections:

Software inspection is a formal and systematic process of examining software work products (e.g., code, design documents, test plans) to detect defects and ensure compliance with standards, requirements, and specifications. It is a peer-review process performed manually to identify issues early in the development lifecycle, before testing or execution.

There are three advantages of software inspection over testing:

1. Inspection session can discover many errors in a system.
2. Incomplete versions of a system can be inspected without additional costs.
3. Inspection help set compliance with standards, portability and maintainability.

Roles in Software Inspection

- **Moderator:** Leads the inspection process, ensures adherence to the procedure, and mediates discussions.
- **Author:** The person who created the work product under inspection.
- **Reviewers (Inspectors):** Examine the work product for defects.
- **Recorder:** Documents all findings and defects during the inspection meeting.

Software Inspection Process

1. **Planning:**
    - A moderator (leader of the inspection) schedules the inspection.
    - The work product (e.g., source code, design document) is distributed to the team.
    - Entry criteria (e.g., completeness of the work product) are verified.
2. **Overview Meeting:**
    - The author explains the work product to the inspection team.
    - Team members gain an understanding of the material being reviewed.
3. **Preparation:**
    - Each participant individually examines the work product for defects, issues, or deviations from standards.
    - Reviewers prepare notes for the inspection meeting.
4. **Inspection Meeting:**
    - **Roles:** Moderator, author, reviewers, and recorder participate.
    - Reviewers discuss the defects found during preparation.
    - The team identifies and records defects but does not solve them.
5. **Defect Reporting:**
    - The defects identified are documented in an inspection report.
    - The report includes details like defect type, severity, and location.
6. **Rework:**
    - The author addresses the identified defects based on the inspection report.
7. **Follow-Up:**
    - The moderator ensures all defects have been resolved and validates the reworked product.
    - Exit criteria are checked to confirm readiness for the next phase.

5.2 Testing Phases

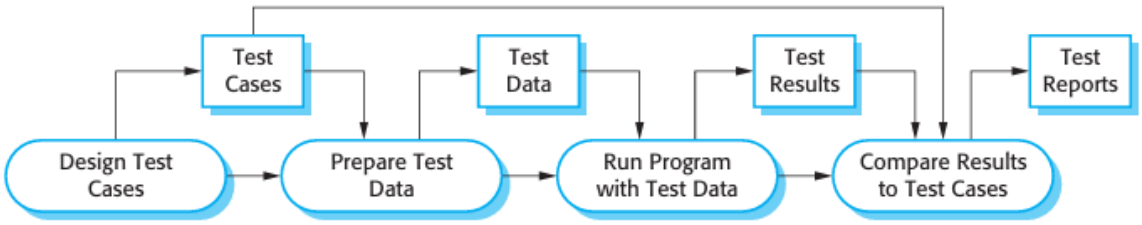


Figure: A model of the software testing process

Typically, a commercial software system has to go through three stages of testing:

a) **Development testing,** where the system is tested during development to discover bugs and defects. System designers and programmers are likely to be involved in the testing process.

b) **Release testing,** where a separate testing team tests a complete version of the system before it is released to users. The aim of release testing is to check that the system meets the requirements of system stakeholders.

c) **User testing**, where users or potential users of a system test the system in their own environment. For software products, the 'user' may be an internal market ing group who decide if the software can be marketed, released, and sold. Acceptance testing is one type of user testing where the customer formally tests a system to decide if it should be accepted from the system supplier or if further development is required.

a) Development Testing

Development testing refers to testing activities that are carried out during the development of the software to identify and fix issues early. It focuses on verifying that individual components, subsystems, and the entire system function correctly according to specifications before deployment.

Development testing is a vital part of the software development lifecycle, aiming to catch defects early and ensure that individual components, as well as the entire system, work as intended. Incorporating development testing with automated testing frameworks, adhering to best practices like TDD, and maintaining a focus on continuous testing are essential for building robust and reliable software.

Key Goals:

- Detect defects early in the software development process.
- Validate that each part of the system is working as intended.
- Ensure integration of components works smoothly.

Development testing can be broken down into three primary levels:

i) Unit Testing
ii) Component Testing
iii) System Testing

i) Unit Testing

- **Definition**: It is the level of testing that focuses on verifying the correctness of individual software components or modules in isolation from the rest of the system. Unit testing is the process of testing program components, such as methods or object classes. Individual functions or methods are the simplest type of component.
- **Purpose**: To verify that the smallest parts of the software work correctly in isolation.
- **Who Performs It?**: Usually done by developers who write the code.
- **Tools and Techniques**: Unit testing frameworks like JUnit (for Java), NUnit (for .NET), and PyTest (for Python) are commonly used. Stubs and mock objects are often employed to simulate interactions with other parts of the software.
- **Type of testing used**: White Box testing (Developers often have knowledge of the internal code structure.)
- **Benefits**:
    - Catches defects early in the development process.
    - Ensures that each unit works as expected before integrating it with other units.
- **Example**: In an e-commerce application, a developer writes unit tests for individual functions like calculating total order cost, applying discounts, and validating user input.
- **Another Example of Unit Testing:** Suppose we have a function calculateTax(income) in a financial application:
    - **Test Cases:**
        - For an income of $50,000, check if the tax is calculated correctly.
        - For boundary conditions, such as an income of $0, ensure the tax is $0.
        - For invalid inputs (e.g., negative income), ensure appropriate error handling.

ii) Component Testing

- **Definition**: It is a level of testing where individual components, modules, or groups of related functions are tested to ensure that they work as specified. It involves verifying the interactions between units within the component.
- **Purpose**: To test the behavior of related units when they are integrated together to form a component, ensuring that they work correctly as a cohesive unit.
- **Who Performs It?**: Usually performed by developers or testers, focusing on integrated functionality.
- **Tools and Techniques**: Component testing may use the same tools as unit testing but also includes test scripts that examine the interactions between units.
- **Type of testing used**: White Box testing (Developers often have knowledge of the internal code structure.)
- **Benefits**:
    - Helps identify defects in interactions between integrated units.
    - Validates that a set of related functionality works as a whole.
- **Example**: In the e-commerce system, after unit testing the order calculation, discount, and validation functions, the developer would run component tests to verify that these functions work together correctly as part of the "Checkout" module.

ii) System Testing

- **Definition**: It is the level of testing that tests the entire system after all components have been integrated. It focuses on verifying the system as a whole, ensuring that it meets the specified functional and non-functional requirements.
- **Purpose**: To ensure that the complete software system behaves according to the specification when all components are integrated.
- **Who Performs It?**: Generally done by dedicated testers, but in some development processes (e.g., agile), developers may also perform system testing.
- **Tools and Techniques**: Automated test suites, test case management tools, and functional testing frameworks can be used. System testing often includes functional and non-functional tests.
- **Type of Testing used**: Black Box testing used (Testers do not need to know the internal or structure of the system)

- **Benefits**:
    - Validates that the system works as expected for end-to-end functionality.
    - Detects issues that might not be evident during unit or component testing.
- **Example**: In the e-commerce application, system testing would involve running tests on the entire ordering process, from selecting items, adding them to the cart, checking out, and receiving confirmation emails, to ensure that all the integrated parts work together correctly.

b) Release Testing

Release Testing is described as a specific type of system testing performed to ensure that a software system is ready for delivery to end users or customers. It focuses on validating the software in its final version and checking if it meets the customer's expectations and business requirements.

Goal: To ensure that the complete system works as expected in its operational environment and meets both functional and non-functional requirements.

i) Requirement-based Testing

Requirement-based testing (RBT) is a software testing approach that ensures that the software system meets its specified requirements. In this approach, test cases are derived directly from the documented software requirements, ensuring a strong connection between the testing activities and the system's expected functionality.

Key Goal:

- To verify that the software system conforms to its specified requirements and behaves as expected under defined scenarios.

Key characteristics of RBT:

- Directly linked to requirements: Both functional and non-functional requirements (such as performance, security, and usability) are considered in the testing process.
- Prioritization of Test Cases: Test cases are prioritized based on the criticality of the requirement.
- Positive and Negative Testing: RBT includes **positive testing** (testing that a feature works as expected) and **negative testing** (testing that the system gracefully handles invalid or unexpected inputs).

Example of Requirement-Based Testing

Requirement:

- "The system shall send a confirmation email to the user after they complete registration."

Test Cases:

1. Verify that a confirmation email is sent after a successful registration.
2. Verify that the confirmation email is sent to the correct email address.
3. Verify that the email contains the correct user details (e.g., username).
4. Verify that no confirmation email is sent if the registration process fails.

ii) Scenario Testing

Scenario Testing is an approach where test cases are designed based on realistic user scenarios, focusing on how the system behaves under typical, real-world conditions.

Scenario testing is a software testing technique where test cases are derived from real-world use cases or scenarios that the system is likely to encounter. It focuses on testing the system in a way that reflects how users will interact with it, providing a broader view of the software's functionality and behavior under realistic conditions.

A scenario is a narrative description of how the system will be used in a particular situation. Scenarios are usually designed to represent end-to-end functionality, involving multiple steps and covering multiple user interactions.

Example of Scenario Testing

Scenario:

A user logs into an online shopping platform, browses products, adds items to their cart, proceeds to checkout, and completes a payment.

Test Steps:

1. Open the shopping site and log in with valid credentials.
2. Browse through categories and add multiple products to the shopping cart.
3. Check that the correct total price is reflected.

4. Proceed to checkout and enter shipping details.
5. Complete the payment process using a credit card.
6. Verify that the order is confirmed, and a confirmation email is sent.

iii) Performance Testing

Performance Testing is a type of non-functional testing focused on evaluating how a software system performs under certain conditions. It ensures that the system meets the required performance criteria related to speed, responsiveness, scalability, and resource usage.

Key Objective:

- To assess how well a system performs under typical and extreme conditions.

Types of Performance Testing

a) Load Testing

- **Purpose:** To evaluate how the system performs under expected user loads.
- **Goal:** To identify performance bottlenecks under normal usage.
- **Example:** Simulating 1,000 concurrent users accessing an e-commerce website to verify if the site responds within acceptable time limits.

b) Stress Testing

- **Purpose:** To test the system's performance beyond normal operational capacity, often to breaking point.
- **Goal:** To find the system's limits and how it handles extreme conditions.
- **Example:** Increasing the number of users beyond capacity to see if the system crashes or how it behaves under overload.

c) Scalability Testing

- **Purpose:** To test the system's ability to scale up or down to handle changes in load.
- **Goal:** To ensure that the system can handle increased user loads without degradation of performance.
- **Example:** Gradually increasing the number of users and data volumes to verify if the system scales appropriately.

d) Spike Testing

- **Purpose:** To assess how the system handles sudden, large increases in load.
- **Goal:** To determine whether the system can recover quickly from spikes in usage.

- **Example:** Simulating a flash sale on a website where thousands of users suddenly access the site at the same time.

e) Endurance (Soak) Testing

- **Purpose:** To assess how the system behaves under continuous load over an extended period.
- **Goal:** To check for memory leaks, system stability, and performance degradation over time.
- **Example:** Running the system with normal load for 48 hours straight to ensure consistent performance.

f) Latency Testing

- **Purpose:** To measure the time taken for data to travel from one point in the system to another (response time).
- **Goal:** To ensure that the system meets latency requirements for real-time applications.
- **Example:** Testing a video streaming service to check the delay between pressing "play" and the video actually starting.

c) User Testing

User Testing is a type of testing that focuses on evaluating the software from the perspective of the end users. The primary goal of user testing is to determine how well the software meets the users' needs and expectations, ensuring that the system is intuitive, usable, and fulfills the requirements it was designed for.

Key Objective:

- To ensure that the system operates as intended in a real-world environment and satisfies user needs.

Types of User Testing:

i) Alpha Testing
ii) Beta Testing
iii) Acceptance Testing

i) Alpha Testing

Alpha testing is the type of testing conducted by the development team or a selected group of users to test the software at the developer's site. Alpha testing is usually done before the software is released to a wider audience and focuses on finding bugs, usability issues, and overall user

satisfaction. It helps identify critical usability problems before the software is exposed to real users.

- **Purpose:** Initial user testing done at the developer's site by internal employees or small groups of users.
- **Goal:** To identify bugs and usability issues early in the process.
- **Characteristics:**
    - Conducted in a controlled environment.
    - Focuses on functionality and identifying major issues.
- **Conducted by:** Internal staff or users within the development organization.
- **Environment:** Simulated or controlled.
- **Objective:** To identify major bugs before the system is handed over to the client for further testing
- **Example:** An internal team of employees using a new project management tool to identify any significant issues before the software is released to customers.

ii) Beta Testing

Beta testing is the type of user testing where a release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

Beta testing takes place when an early, sometimes unfinished, release of a software system is made available to customers and users for evaluation.

Beta testers may be a selected group of customers who are early adopters of the system. Alternatively, the software may be made publicly available for use by anyone who is interested in it. Beta testing is mostly used for software products that are used in many different environments (as opposed to custom systems which are generally used in a defined environment). It is impossible for product developers to know and replicate all the environments in which the software will be used. Beta testing is therefore essential to discover interaction problems between the software and features of the environment where it is used. Beta testing is also a form of marketing— customers learn about their system and what it can do for them.

- **Purpose:** Testing conducted at the user's site, typically by external users.
- **Goal:** To gather feedback from real-world use cases and environments.
- **Characteristics:**
    - Involves a larger, more diverse group of users.
    - Users test the system in their own environment and provide feedback.

- Focuses on real-world usage and system reliability.

- **Conducted by:** Real users in their actual operational environment.
- **Environment:** Live or operational environments.
- **Objective:** To gather feedback on the system's performance, usability, and any issues that might not have been identified in controlled tests.

Example: A software company releases a beta version of a new app to a select group of users who provide feedback on usability and report bugs

iii) Acceptance Testing

Acceptance testing, where customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

Acceptance testing is an inherent part of custom systems development. It takes place after release testing. It involves a customer formally testing a system to decide whether or not it should be accepted from the system developer. Acceptance implies that payment should be made for the system.

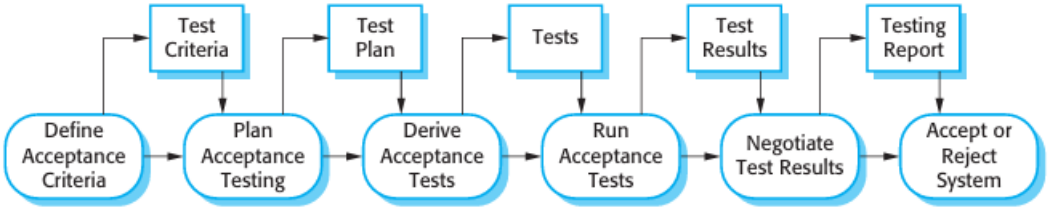


Figure: Acceptance Testing activities

- **Purpose:** Final stage of testing before system deployment, where the customer or end-user verifies that the system meets their needs and requirements.
- **Goal:** To ensure that the software is acceptable for release.
- **Characteristics:**
  - Involves customer representatives or actual users.
  - The system must pass predefined acceptance criteria.
  - Often follows a formal process with specific test cases.
- **Conducted by:** Client or user representatives.
- **Environment:** Client's environment or test environment that mimics real-world conditions.
- **Objective:** To formally validate that the system meets the business requirements and is ready for release.

5.3 Test Case Development Strategies

Test case:

A **test case** is a specific set of inputs, execution conditions, and expected outcomes that verify a program or a piece of code works as intended. It is designed to ensure that the software behaves correctly under a defined scenario.

A **test suite** is a collection of test cases that are grouped together to test a specific functionality, feature, or module of a program. It helps in systematically organizing and executing multiple test cases to ensure that a system works as intended across different scenarios.

Components of a Test Case

1. **Test Case ID**: A unique identifier for the test case.
2. **Inputs**: The data provided to the code or system.
3. **Execution Steps**: Steps to execute the test case.
4. **Expected Output**: The expected result for the given inputs.
5. **Actual Output**: The result observed after running the test case (compared with the expected output).

Example Code

Here's a simple example: A program to determine if a number is even or odd.

```
public String checkEvenOdd(int number) {
    if (number % 2 == 0) {
        return "Even";
    } else {
        return "Odd";
    }
}
```

Simple Test Case

Test Case ID	Input	Steps	Expected Output	Actual Output
TC1	number = 4	Call checkEvenOdd(4)	"Even"	"Even"
TC2	number = 7	Call checkEvenOdd(7)	"Odd"	"Odd"
TC3	number = 0	Call checkEvenOdd(0)	"Even"	"Even"

Test case Development Strategies:

a) Boundary Value Analysis

b) Equivalence Partitioning

c) Basis Path Testing

d) Control Structure Testing

a) Boundary Value Analysis

- BVA is the testing technique meant for testing the boundaries of the input domain for errors.
- Here, we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).
- For eg. A form accepts product delivery days: a minimum of 2 days and maximum of 14 days are noted on the form.

  Boundary value test might put values of 1, 2, 14, 15 to determine how the function reacts to data.
- Consider another example:

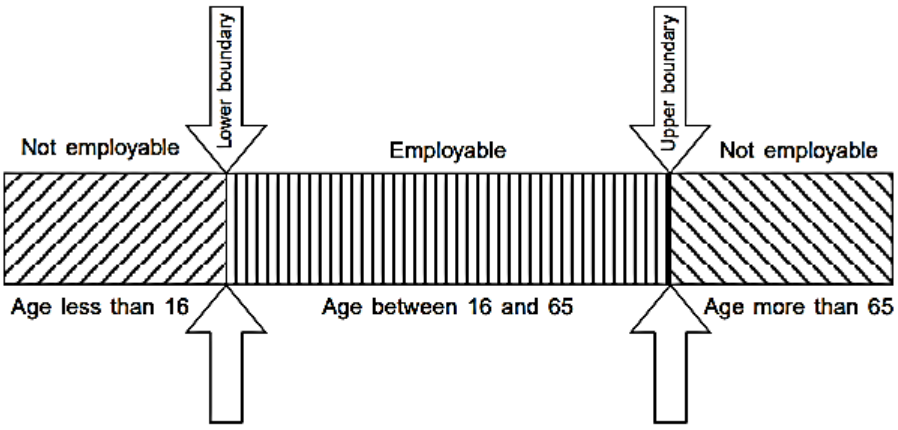


Figure: Boundary values

- There are two boundaries: the minimum employable age and the maximum employable age (retirement age).
    - These two boundaries—16 and 65—must be accepted, and all values below 16 and above 65 must be rejected.
    - Therefore, test cases are designed that combine the techniques of equivalence partitioning and boundary value analysis.
- In this example, there are five test cases:
    1. One value between 16 and 65—valid value
    2. One value at the lower boundary of 16—valid value
    3. One value just below the lower boundary (that is, less than 16)— invalid value (normally this value would be given as 1 day less than 16)
    4. One value at the upper boundary of 65—valid value
    5. One value just above the upper boundary (that is, greater than 65)— invalid value (normally this value would be given as 65 years and 1 day

Code Example

Here's a simple Java function that checks if the product delivery days are within the valid range of 2 to 14 days:

```
public String checkDeliveryDays(int days) {
   if (days >= 2 && days <= 14) {
      return "Valid Delivery Days";
   } else {
      return "Invalid Delivery Days";
   }
}
```

And the user interface is as:

Delivery date

The boundaries for the delivery days are:

- Minimum boundary: 2
- Maximum boundary: 14
- We also test just below and just above the boundaries.

Test case:

Test Case ID	Input (days)	Expected Output	Description	Actual Output
TC1	1	Invalid Delivery Days	Below the minimum boundary (invalid)	
TC2	2	Valid Delivery Days	Minimum boundary (valid)	
TC3	13	Valid Delivery Days	Just below the maximum boundary (valid)	
TC4	14	Valid Delivery Days	Maximum boundary (valid)	
TC5	15	Invalid Delivery Days	Above the maximum boundary (invalid)	

b) Equivalence Partitioning

- It is a software testing technique that divides the input data of the application under test into each partition at least once of equivalent data from which test cases can be derived.
- It reduces the time required for performing testing of a software due to less number of test cases.
- For eg.

  Consider that a form accepts an integer in the range of 100 to 999.

  Valid equivalence class partition: 100 to 999 inclusive

  Non-valid equivalence class partition : less than 100,

more than 999,

decimal no,

alphabets and non-numeric characters.

- In equivalence partitioning, the input space is partitioned into valid inputs and invalid inputs.
- The following example illustrates this technique.
    - In a human resources application, employee age can be a minimum of 16 (minimum employable age) and a maximum of 65 (retirement age).
    - The partition of valid values is between 16 and 65.
    - There are two partitions of invalid values: one below 16 and the other above 65.
    - Therefore, there are three partitions for this case—one valid and two invalid.

One test case can be designed for each partition, resulting in three test cases

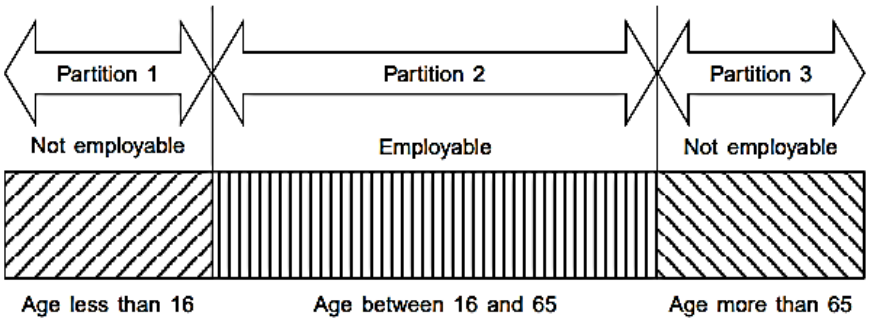


Figure: Equivalence Partitioning

Example Scenario:

Suppose we are testing a form input field that accepts integer values between 1 and 100 (inclusive).

And the user interface is as:

Enter a number:

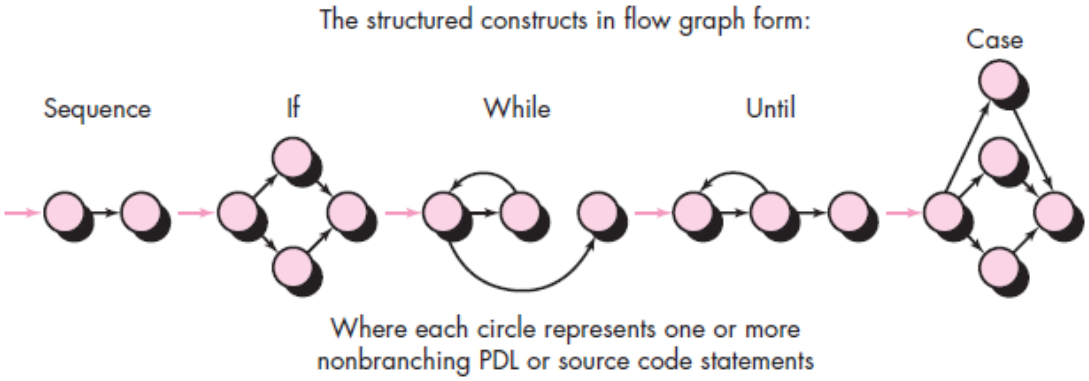
- **Step 1: Identify the Input Domain:**
  The valid input range is 1 to 100. Inputs outside this range are invalid.
- **Step 2: Divide into Equivalence Partitions:**
  - **Valid Partitions:**
    - Inputs between 1 and 100 (inclusive).
  - **Invalid Partitions:**
    - Inputs less than 1.
    - Inputs greater than 100.
    - Non-integer inputs (e.g., characters, symbols).
- **Step 3: Select Representative Values:**
  - For the valid partition (1 to 100), we can select a representative value like **50**.
  - For invalid partitions:
    - Less than 1: **0** (e.g., test the boundary condition below the valid range).
    - Greater than 100: **101** (e.g., test the boundary condition above the valid range).
    - Non-integer: **"abc"** (test for non-integer input).
- **Step 4: Create Test Cases:**
  Based on the selected representative values, the test cases could be:

Test Cases for Equivalence Testing

Test Case ID	Input (value)	Expected Output	Equivalence Class	Actual Output
TC1	"0"	Invalid Input	Invalid (Below the lower bound)	
TC2	"50"	Valid Input	Valid (Inside the range)	
TC3	"101"	Invalid Input	Invalid (Above the upper bound)	
TC4	"abc"	Invalid Input	Invalid (Non-integer input)	
TC5	"3.5"	Invalid Input	Invalid (Non-integer input)	
TC6	"a"	Invalid Input	Invalid (Non-integer input)	

c) Basis Path Testing

- Basic Path Testing is a **white-box testing technique** that focuses on analyzing the control flow of a program to identify all possible execution paths. The purpose of path testing is to ensure that every line of code and every potential decision path in the program is tested at least once.
- Path testing is a structural testing method based on the source code or algorithm and NOT based on specification.
- Used flow graphs for representing the control flow or logical flow as show below:



Cyclomatic Complexity

- Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors.
- It is calculated by developing a Control Flow Graph (CGF) of the code that measures the number of linearly- independent paths through a program module.
- Lower the cyclomatic complexity, lower the risk to modify and easier to understand.
- Cyclomatic complexity = E- N +2

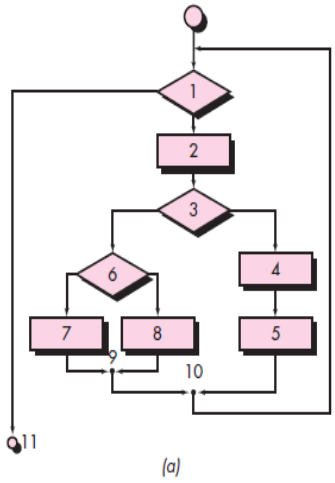
  Where:

  E = no. of edges in CGF

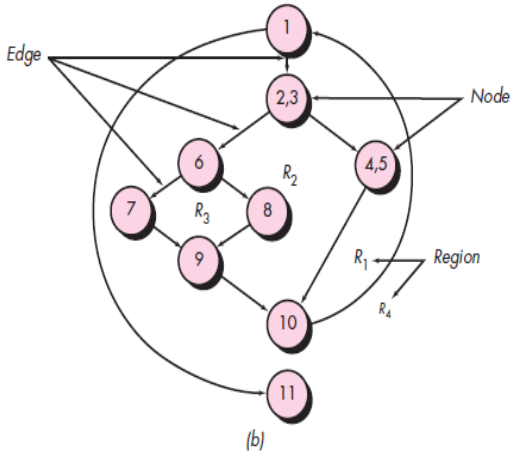
  N= no. of nodes in CGF

- Also

  Cyclomatic complexity = no of regions in the CFG



Flow chart



Flow graph

- Here, No of edges (E) =11

  No of nodes(N) = 9

  Cc = 11-9+2 = 4

Also, Cc = no of regions = 4

Note:

- Each & every nodes must be terminating on one node so we need the node – 6 on the given example.
- The Cyclomatic complexity must be equal in all these three cases.
- If there are other paths beside the Basic path then they are redundant path.
- The test case design has the highest probability of finding the most errors with a minimum amount of time and effort.

Basis Path Testing Procedure:

Step 1: Construct a **control flow graph** (CFG) for the program.

Step 2: Calculate the **cyclomatic complexity** to determine the number of independent paths.

Step 3: Identify the **independent paths** based on the graph.

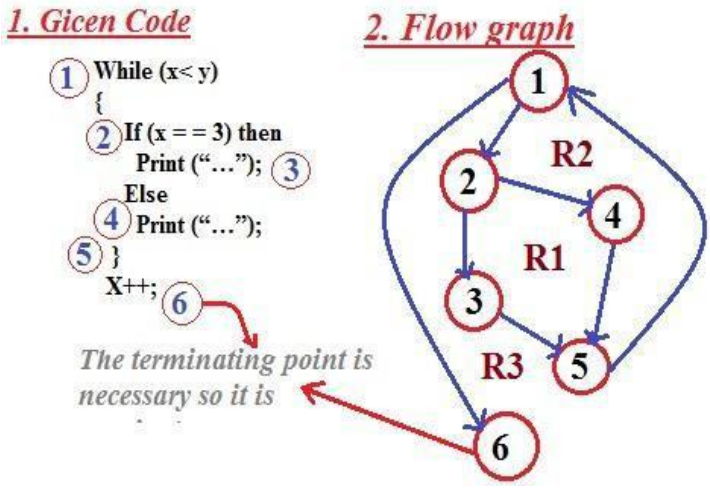
Step 4: Create **test cases** that cover all the independent paths.

Step 5: Execute the tests and analyze the results to ensure that all paths have been tested properly.

Example: **Consider a simple program with decision-making logic:**

```
while (x<y)
{
        if (x==3) then
                print ( " ….."");
        else
                print ( "…");
}
x++;
```

Step 1: Construction of a control flow graph (CFG) for the program.



Step 2: Calculation of the Cyclomatic complexity to determine the number of independent paths.

a) Here,

Number of region (R) = 3

Thus, Cc = R = 3

b) Again,

Number of Edges (E) = 7

Number of Nodes (N) = 6

Thus, Cc = E- N + 2 = 7-6 +2 = 3

Step 3: Identify the independent paths based on the graph.

The basis paths are:

Path 1: 1 – 6

Path 2: 1 – 2 – 3 – 5 – 1 – 6

Path 3: 1 – 2 – 4 – 5 – 1 – 6

Path ID	Path Sequence	Description
Path 1	1 → 6	Loop is skipped because `x >= y` initially, so the program exits.
Path 2	1 → 2 → 3 → 5 → 1 → 6	Loop iterates with `x != 3`. The program enters the `else` branch and prints `" …"`.
Path 3	1 → 2 → 4 → 5 → 1 → 6	Loop iterates with `x == 3`. The program enters the `if` branch and prints `" …..."`.

Test Cases Based on the Paths:

Test Case ID	Input (x, y)	Path Covered	Description	Expected Output	Actual Output
TC1	(x = 5, y = 5)	Path 1: `1 → 6`	Tests the scenario where `x >= y`, so the loop does not execute.	No output, as the loop is skipped.	
TC2	(x = 3, y = 5)	Path 2: `1 → 2 → 3 → 5 → 1 → 6`	Tests the scenario where `x = 3` and the loop runs.	`Print the statement inside if`	
TC3	(x = 2, y = 5)	Path 3: `1 → 2 → 4 → 5 → 1 → 6`	Tests the scenario where `x != 3` for the first iteration.	`Print the statement inside else.`	

d) Control Structure Testing

Control Structure Testing is a systematic approach used in software testing to validate the logic and flow of control within a program. This technique focuses on examining the various control structures (such as loops, branches, and conditionals) in the source code to ensure they function correctly and produce the expected results. By rigorously testing these control structures, testers can uncover defects related to logic errors and flow anomalies.

The basis path testing technique is one of a number of techniques for control structure testing but it is not sufficient on control structure testing. Some popular control structure testing techniques are described below:

i) Branch Testing (Decision Testing)

Branch Testing focuses on testing every branch of decision-making constructs (e.g., `if-else`, `switch-case`). It ensures that every possible decision outcome (True/False) is executed at least once. For example, in an `if-else` construct, both the `if` and the `else` parts must be tested.

```
if (x > 10 && y < 5) {
    z = x + y;
} else {
    z = x - y;
}
```

Here:

- Decision testing will ensure:
    - The `if` block (true branch) is executed at least once.
    - The `else` block (false branch) is executed at least once.
- It does not test the individual conditions `x > 10` and `y < 5` separately but focuses on the overall decision `(x > 10 && y < 5)`.

Test Case

Test Case ID	Input Values	Condition Evaluation	Branch Executed	Expected Value of z
TC1	x = 15, y = 2	x > 10 = true, y < 5 = true	True branch (z = x + y)	z = 15 + 2 = 17
TC2	x = 8, y = 2	x > 10 = false, y < 5 = true	False branch (z = x - y)	z = 8 - 2 = 6
TC3	x = 15, y = 8	x > 10 = true, y < 5 = false	False branch (z = x - y)	z = 15 - 8 = 7
TC4	x = 8, y = 8	x > 10 = false, y < 5 = false	False branch (z = x - y)	z = 8 - 8 = 0

ii) Condition Testing

Conditional testing involves testing a program's functionality by evaluating different conditions (e.g., if, else if, else, switch-case statements). The goal is to ensure that the program executes correctly under various scenarios.

Condition testing is a test-case design method that exercises the logical conditions contained in a program module. A relational expression takes the form: ***E*1 <relational-operator> *E*2**, Where *E*1 and *E*2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠, >, or ≥.

A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&), and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.

Example:

```
if (x > 10 && y < 5) {
   z = x + y;
}
else {
   z = x - y;
}
```

Conditional Testing focuses on testing each condition in the if statement **individually**:

1. **Condition 1:** `x > 10`
2. **Condition 2:** `y < 5`

Test Cases for Conditional Testing:

Test Case ID	Input (x, y)	Condition `x > 10`	Condition `y < 5`	Expected Output (z)	Description
TC1	(12, 3)	True	True	15	Both conditions are true; `z = x + y`.
TC2	(8, 3)	False	True	5	`x > 10` is false; `z = x - y`.
TC3	(12, 7)	True	False	5	`y < 5` is false; `z = x - y`.
TC4	(8, 7)	False	False	1	Both conditions are false; `z = x - y`.

iii) Data Flow Testing

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.

Data Flow Testing focuses on the lifecycle of variables in a program, specifically their **definitions** (where variables are assigned values) and **uses** (where variables are read). This type of testing examines paths from the **definition** of a variable to its **use**.

Specifically, it checks:

1. **Where variables are defined (assigned a value)**.

   Example: `int x = 5;` (definition of `x`).

2. **Where variables are used**:
   - **C-use** (Computational Use): Used in calculations

     Example: `y = x + 2;` (`x` is used here).

   - **P-use** (Predicate Use): Used in conditions.

Example: `if (x > 5)` (`x` is used in the condition).

Data Flow Testing ensures:

- Every variable **defined** is eventually **used**.
- There are no unnecessary or incorrect uses of variables.

Example:

```
public class SimpleDataFlow {
    public static int calculateSum(int a, int b) {
        int sum = 0;            // Definition of sum (D1)
        if (a > b) {            // Predicate use of a and b (P1, P2)
            sum = a + b;        // Definition of sum (D2), computational use of a and b (C1, C2)
        }
        return sum;             // Computational use of sum (C3)
    }

    public static void main(String[] args) {
        // Test cases
        System.out.println("Case 1: " + calculateSum(5, 3));
            // Expected: 8 (a > b, sum = a + b)
        System.out.println("Case 2: " + calculateSum(2, 5));
            // Expected: 0 (a <= b, sum remains 0)
    }
}
```

Test Cases

Test Case ID	Input (a, b)	DU Pair Covered	Description	Expected Output	Actual Output
TC1	(5, 3)	D2 → C3	Covers the true branch of `if (a > b); sum = a + b.`	8	
TC2	(2, 5)	D1 → C3	Covers the false branch of `if (a > b); sum = 0.`	0	

iv) Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.

- **Simple loops:** They are sequentially tested for the data of desired domain.
- **Nested loop:** They are started to test from innermost loop.
- **Concatenated loop:** Multiple loops are tested simultaneously if there is no dependency exists.
- **Unstructured loops:** Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

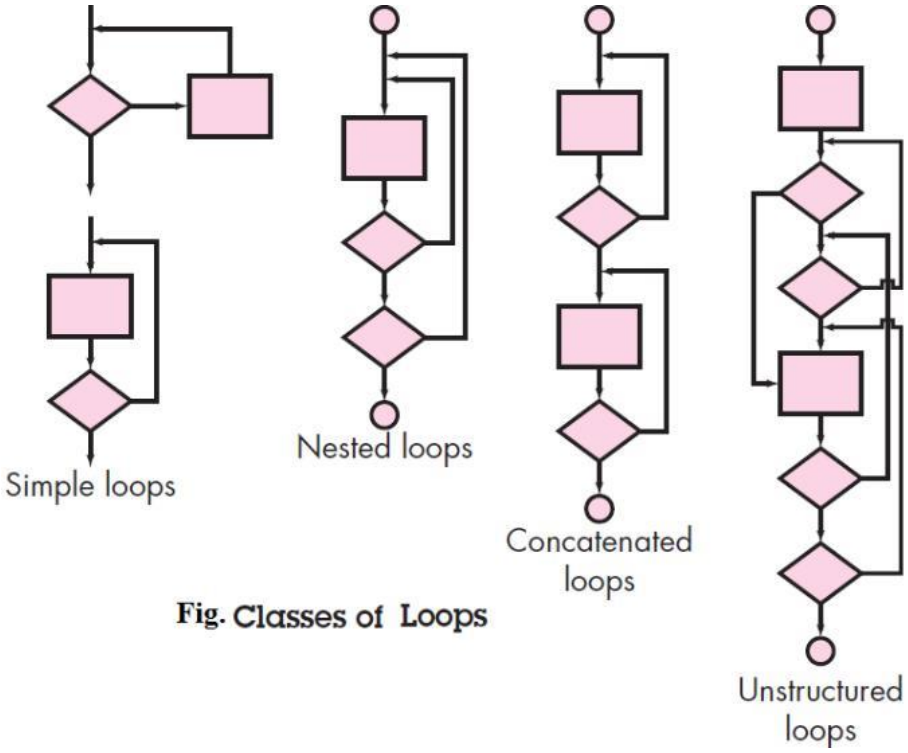


Fig. Classes of Loops

- **Loop Testing** focuses on validating loops in a program to ensure they work correctly under different conditions. It tests:

    1. **Zero iterations**: The loop is skipped entirely.
    2. **One iteration**: The loop executes once.
    3. **Multiple iterations**: The loop runs several times.
    4. **Boundary conditions**: Testing at the edge of loop limits.

Example:

```
public class LoopTestingExample {
    public static int sumOfNumbers (int n) {
        int sum = 0; // Initialize sum
        for (int i = 1; i <= n; i++) {  // Loop from 1 to n
            sum += i; // Add i to sum
        }
        return sum; // Return the result
    }

    public static void main(String[] args) {
        // Test cases
        System.out.println("Case 1: " + sumOfNumbers(0)); // Expected: 0
        System.out.println("Case 2: " + sumOfNumbers(1)); // Expected: 1
        System.out.println("Case 3: " + sumOfNumbers(3)); // Expected: 6
        System.out.println("Case 4: " + sumOfNumbers(100)); // Expected: 5050
    }
}
```

Test Cases

Test Case ID	Input (n)	Condition Tested	Description	Expected Output	Actual Output
TC1	0	Zero iterations	Loop does not execute.	0	
TC2	1	One iteration	Loop executes once (`i = 1`).	1	
TC3	3	Multiple iterations	Loop executes 3 times (`i = 1, 2, 3`).	6	
TC4	100	Boundary condition	Loop executes 100 times (`1 to 100`).	5050	

5.4 Test Automation

Test automation is the use of software tools and scripts to automatically execute tests on a software application, instead of performing manual testing. It helps speed up the testing process, increase accuracy, and ensure consistent results.

Some of the popular test tools for automation are:

- **Java**: JUnit, TestNG, Selenium, Mockito, Cucumber.
- **Python**: pytest, unittest, Selenium, Robot Framework, nose2.
- **C++**: Google Test, Catch2, CppUnit, Boost.Test, TUT.
- **PHP**: PHPUnit, Behat, Codeception, PHPSpec, SimpleTest.

5.5 Software Changes and Evolution

Software is not static; it evolves over time to meet changing requirements, fix defects, and adapt to new technologies. Understanding software changes and evolution is crucial for maintaining software quality and ensuring that it continues to meet user needs.

Reasons for Software Changes:

Software changes can be driven by various factors, including:

1. **Changing Requirements:** As businesses evolve, so do their requirements. New features may need to be added, or existing features may need to be modified or removed.
2. **Defect Fixes:** Software may have defects or bugs that require correction to improve functionality and user experience.
3. **Technological Advancements:** The introduction of new technologies can necessitate updates to existing software to ensure compatibility and take advantage of improvements.
4. **Performance Improvements:** Optimizing performance may involve changes to algorithms, data structures, or system configurations.
5. **Regulatory Compliance:** Changes in laws and regulations may require software updates to ensure compliance.
6. **User Feedback:** Direct feedback from users can lead to enhancements in usability, functionality, and overall user satisfaction.

Types of Software Changes

Software changes can be categorized into several types:

- **Corrective Changes:** Modifications made to fix defects or issues in the software.
- **Adaptive Changes:** Changes made to adapt the software to a new environment or to integrate with new hardware or software.
- **Perfective Changes:** Enhancements made to improve performance or add new features without altering the existing functionality.
- **Preventive Changes:** Updates aimed at preventing future problems or issues, often involving refactoring or re-engineering components.

Software Evolution:

Software evolution refers to the process of developing and maintaining software systems over time, encompassing all the changes and adaptations made to the software in response to changing requirements, technologies, and environments. This process involves modifying existing software to enhance its functionality, performance, or usability, ensuring that it continues to meet the needs of users and stakeholders.

Software evolution refers to the process of changing and adapting software to:

- Fix issues.
- Add new features.
- Adapt to new environments or technologies.

Software Evolution Process:

The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers.

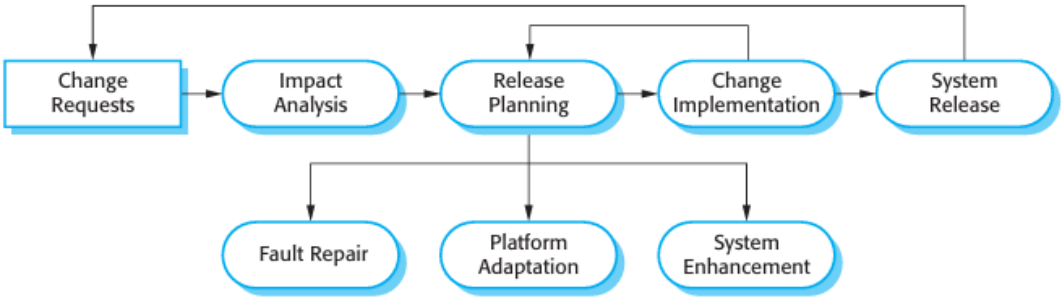


Figure: The Software Evolution Process

- The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.
- If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered.
- A decision is then made on which changes to implement in the next version of the system. The changes are implemented and validated, and a new version of the system is released.
- The process then iterates with a new set of changes proposed for the next release.

Phases in Software Evolution:

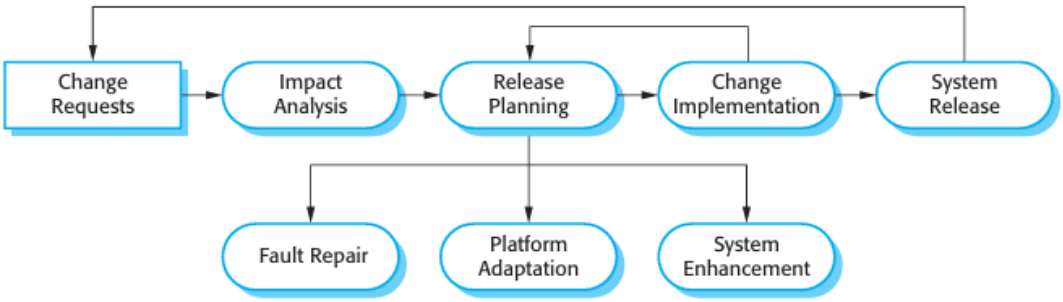


Figure: The Software Evolution Process

1. Change Requests

- Change Requests are formal proposals for modifications to the software system. They arise from users, stakeholders, or internal assessments.
- Each request is documented, specifying the nature of the change, its justification, and anticipated benefits.
- Effective management of change requests is crucial for prioritizing updates based on urgency and impact.

2. Impact Analysis

- Impact Analysis evaluates the potential consequences of implementing change requests.
- This phase includes identifying affected components, estimating required resources, and assessing risks associated with the change.

- It ensures that the proposed modifications are feasible and aligned with project goals, helping to inform decision-making about whether to proceed.

3. Release Planning

- **Release Planning** is the strategic phase where the approach for deploying changes is outlined. It encompasses several key activities:
    - **Fault Repair:** The process of identifying and fixing defects in the software. This includes prioritizing bugs based on their severity and impact, followed by testing to ensure the fixes are effective without introducing new issues.
    - **Platform Adaptation:** Modifying the software to ensure compatibility with different hardware or software environments. This involves assessing compatibility, modifying the code as needed, and conducting tests to validate functionality across various platforms.
    - **System Enhancement:** Adding new features or improving existing functionalities based on user and stakeholder feedback. This requires gathering requirements, designing enhancements, and implementing and testing these changes to ensure they integrate smoothly with the existing system.

4. Change Implementation

- Change Implementation is the execution phase where planned changes are developed and integrated into the software system.
- This includes coding, testing, and documenting the changes.
- Adhering to established quality assurance practices during this phase is essential to minimize defects and ensure the new features function as intended.

5. System Release

- System Release is the final phase where the updated software is deployed to users.
- This involves transferring the new version into the production environment and making it available to end-users.
- Along with the release, user training and support may be provided, and ongoing monitoring is essential to address any post-release issues and gather feedback for future improvements.

5.6 Maintenance Process and Reengineering

Software maintenance refers to the process of modifying and updating software applications after their initial deployment. It is a critical aspect of the software development lifecycle (SDLC), as it ensures that the software continues to meet user needs and adapts to changing environments.

Types of Software Maintenance

Software maintenance can be classified into several categories:

1. **Corrective Maintenance:**
    - Involves fixing defects or bugs in the software.
    - Aims to restore the software to its intended functionality.
    - Examples include patching security vulnerabilities or addressing user-reported issues.
2. **Adaptive Maintenance:**
    - Involves modifying the software to operate in a new environment or to work with new hardware/software.
    - This may include updates for new operating systems or changes in database management systems.
3. **Perfective Maintenance:**
    - Focuses on enhancing the software's performance or adding new features based on user feedback.
    - Examples include improving the user interface, optimizing algorithms, or introducing new functionalities.
4. **Preventive Maintenance:**
    - Aims to anticipate and prevent future problems by making improvements or refactoring the code.
    - This could involve updating libraries, rewriting parts of the code, or improving documentation.

Software Re-engineering

Software reengineering as a structured process of improving or transforming existing software to enhance its maintainability, performance, or adaptability without significantly changing its functionality. Reengineering is seen as a cost-effective alternative to complete redevelopment.

Software reengineering involves analyzing, modifying, and updating a legacy system to make it more efficient and easier to maintain while preserving its core functionalities. It includes activities such as code restructuring, documentation improvement, and migration to modern technologies.

Key Objectives of Software Reengineering

1. **Improved Maintainability**: Making the software easier to understand, update, and debug.
2. **Enhanced Performance**: Optimizing the software to work efficiently with modern hardware and systems.
3. **Reduced Cost**: Minimizing the costs associated with maintaining outdated systems.
4. **Preserving Investment**: Retaining the value of existing software while extending its lifespan.
5. **Migration to Modern Platforms**: Adapting the software to newer environments or technologies.

The Reengineering Process

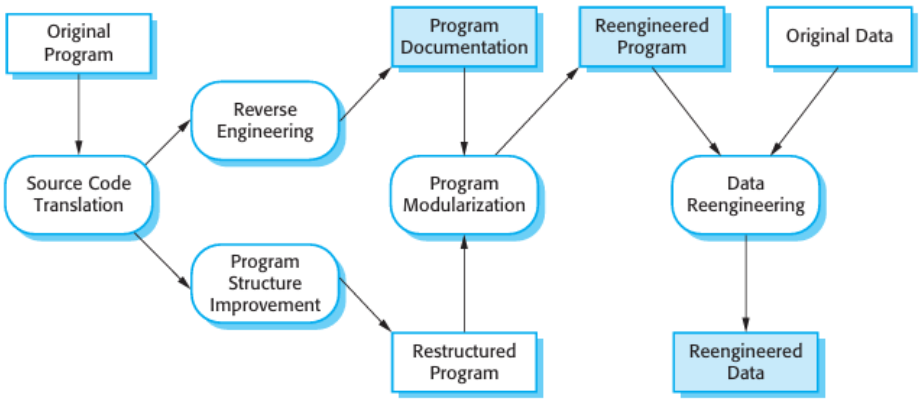


Figure: The reengineering process

The activities in this reengineering process are as follows:

1. **Source Code Translation**
   - The original program's source code is translated into a new programming language or platform.
   - For example, legacy systems written in COBOL could be translated into Java
2. **Reverse Engineering**
   - Reverse engineering analyzes the **existing system** to extract: Design information, Functionality, Business logic, and other essential details that are often poorly documented.
   - This step helps understand the existing system thoroughly before modifications.

- Proper documentation is created or updated to reflect the improved system. This documentation ensures that future developers or maintainers can understand the system effectively.

3. **Program Structure Improvement**
   - The code structure is improved without changing its functionality.
   - Redundant, obsolete, or inefficient code is refactored for better readability and maintainability.
   - This provides a restructured program.
4. **Program Modularization**
   - The code structure is improved without changing its functionality.
   - Redundant, obsolete, or inefficient code is refactored for better readability and maintainability.
   - After modularization and structure improvement, the program becomes more efficient and maintainable. This provides reengineered program.

5. **Data Reengineering**
   - The original data associated with the system is restructured or migrated to modern databases or formats.
   - This step ensures that the data is consistent, optimized, and compatible with the reengineered program.
   - Data associated with the reengineered program is updated to ensure seamless integration and alignment with the modernized system.

Reverse and Forward Engineering

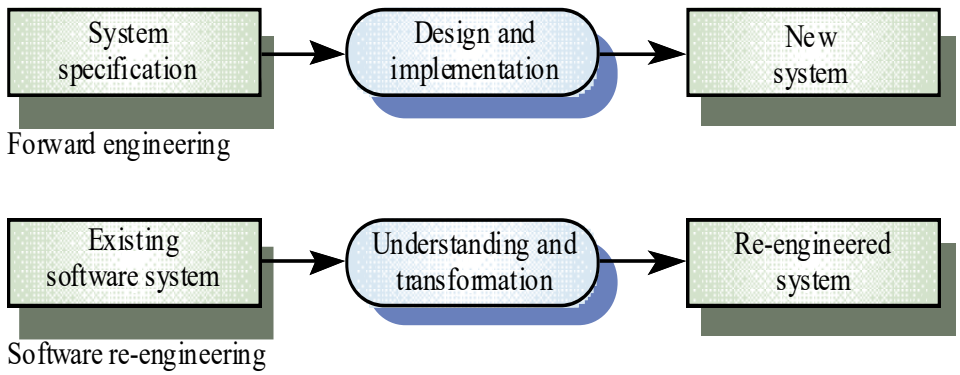


Fig. Forward Engineering and Reengineering

Forward Engineering is a method of creating or making an application with the help of the given requirements. Forward engineering is also known as **Renovation and Reclamation**. Forward engineering is required high proficiency skill. It takes more time to construct or develop an application.

Reverse Engineering is also known as backward engineering, is the process of forward engineering in reverse. In this, the information are collected from the given or exist application. It takes less time than forward engineering to develop an application. In reverse engineering the application are broken to extract knowledge or its architecture.

Difference between Forward Engineering and Reverse Engineering:

S.NO	FORWARD ENGINEERING	REVERSE ENGINEERING
1.	In forward engineering, the application are developed with the given requirements.	In reverse engineering or backward engineering, the information are collected from the given application.
2.	Forward Engineering is high proficiency skill.	Reverse Engineering or backward engineering is low proficiency skill.
3.	Forward Engineering takes more time to develop an application.	While Reverse Engineering or backward engineering takes less time to develop an application.
4.	The nature of forward engineering is Prescriptive.	The nature of reverse engineering or backward engineering is Adaptive.
5.	In forward engineering, production is started with given requirements.	In reverse engineering, production is started by taking existing product.
6.	The example of forward engineering are construction of electronic kit, construction of DC MOTOR etc.	The example of backward engineering are research on Instruments etc.

***