SET

✅ Define Software Engineering (with Objectives, Advantages & Disadvantages)

📘 Introduction

Software Engineering is a systematic, disciplined, and quantifiable approach to the **development, operation, and maintenance of software**.
According to explanations commonly provided by GeeksforGeeks, Software Engineering applies engineering principles to software so that it is **reliable, scalable, maintainable, and delivered within cost and time constraints**.

In simple words, it is the branch of engineering that focuses on building **high-quality software products** using structured processes and methodologies.

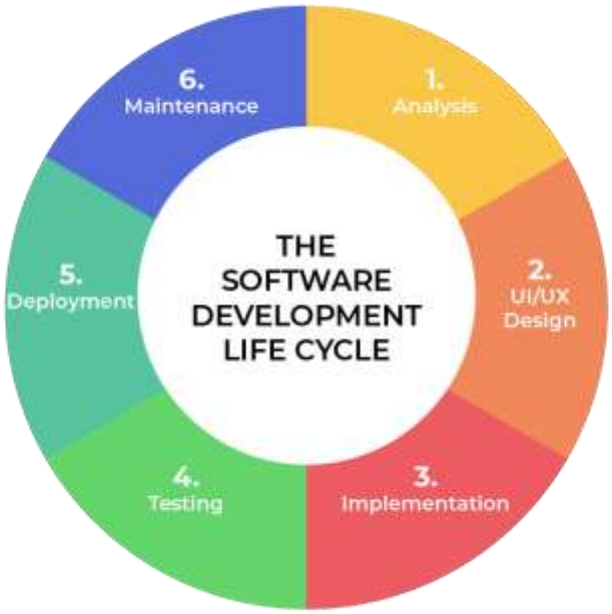
⚙️ Working (How Software Engineering Operates)

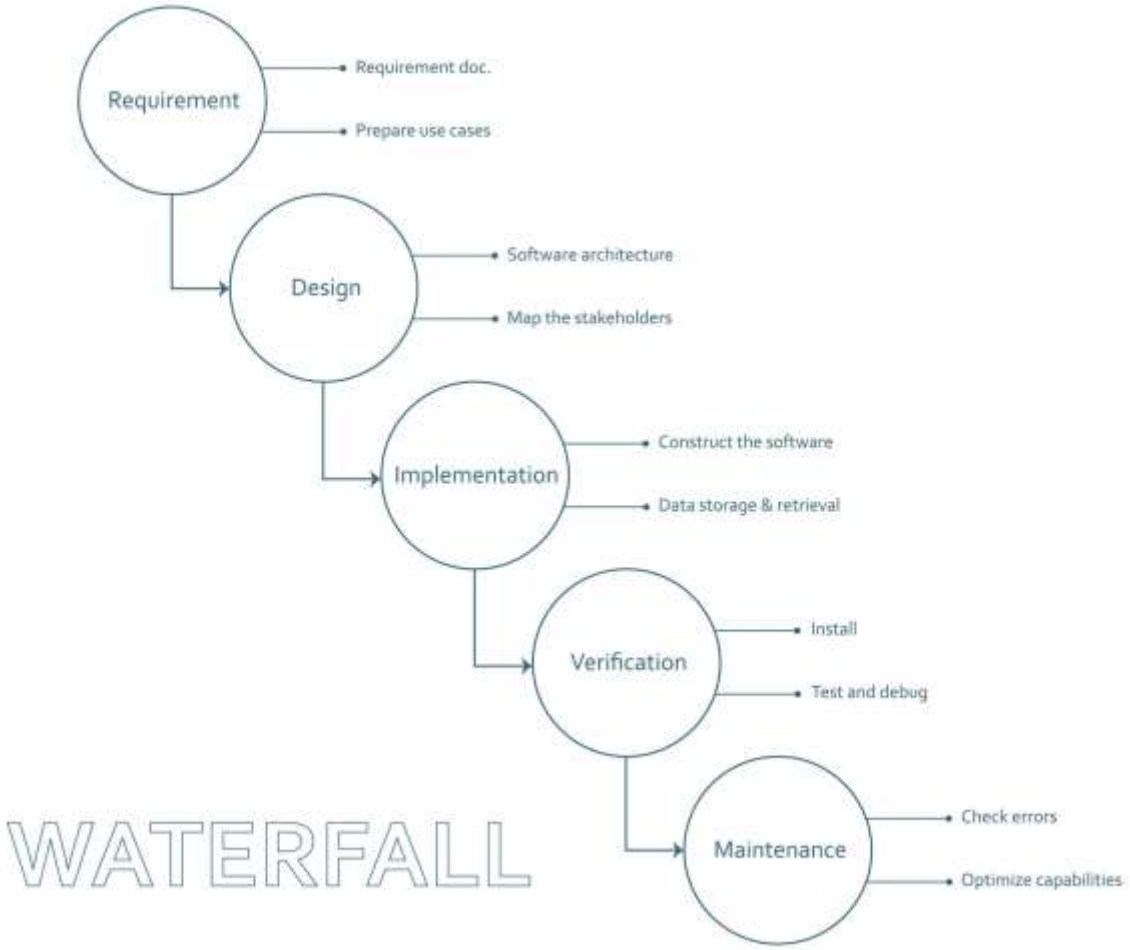
Software Engineering follows a well-defined process:

1. Requirement Analysis – understanding user needs
2. Design – creating system architecture
3. Development – coding the software
4. Testing – verifying correctness and quality
5. Deployment – releasing to users
6. Maintenance – fixing bugs and adding enhancements

Each phase is planned, documented, and reviewed to minimize risk and improve product quality.

🖼️ Diagram (Software Engineering Process Flow)





4

✨ Characteristics

- Systematic and structured development approach
- Emphasis on documentation
- Uses standard models and methodologies
- Focus on quality assurance
- Risk management at every stage
- Team-based development
- Customer requirement driven
- Supports maintainability and scalability

🧪 Example

Developing an **online banking system**:

- Requirements: account management, transfers, security
- Design: database schema, UI layouts
- Development: backend + frontend coding
- Testing: security testing, functional testing
- Deployment: hosted on cloud servers
- Maintenance: bug fixes and feature upgrades

This entire lifecycle is handled using Software Engineering principles.

✅ Objectives of Software Engineering

- Deliver **high-quality software**
- Meet **user requirements**
- Reduce **development cost**
- Ensure **on-time delivery**
- Improve **software reliability**
- Make systems **easy to maintain**
- Manage project complexity
- Increase productivity
- Minimize risks
- Support future enhancements

👍 Advantages

- Produces **reliable and efficient software**
- Better project planning and control

- Reduces overall development cost
- Improves teamwork and communication
- Easier debugging and maintenance
- Supports large and complex projects
- Higher customer satisfaction
- Ensures documentation for future reference

👎 Disadvantages

- Initial cost is high
- Requires skilled professionals
- Time-consuming documentation
- Not economical for very small projects
- Process rigidity in some models
- Tool and training expenses
- Management overhead

🚀 Applications

- Web Applications
- Mobile Apps
- Banking Systems
- Healthcare Software
- Embedded Systems
- AI & ML Platforms
- E-commerce Systems
- Government IT Projects

- Enterprise Resource Planning (ERP)

✅ **Next step:**

✅ **SDLC (Software Development Life Cycle) – Detailed Explanation**

📘 **Introduction**

The **Software Development Life Cycle (SDLC)** is a structured process used to design, develop, test, deploy, and maintain software systems in an organized manner.

According to GeeksforGeeks, SDLC defines a **series of well-planned phases** that help developers produce **high-quality, cost-effective, and reliable software** while minimizing risks.

In simple words, SDLC is a **roadmap** that guides software projects from idea to final product.

⚙️ **Working (Detailed Phase-Wise Explanation)**

SDLC works by breaking the complete software project into multiple stages:

1. Requirement Analysis

- Collect user expectations
- Understand business goals
- Prepare Software Requirement Specification (SRS)

2. Planning

- Estimate cost, time, and resources
- Identify risks
- Decide development strategy

3. Design

- Create system architecture
- Database design
- UI/UX planning
- High-Level Design (HLD) and Low-Level Design (LLD)

4. Development

- Actual coding begins
- Modules are implemented
- Unit testing is done

5. Testing

- Functional testing
- Integration testing
- System testing
- Acceptance testing
  Ensures software meets requirements and is defect-free.

6. Deployment

- Software released to production
- Users start accessing system

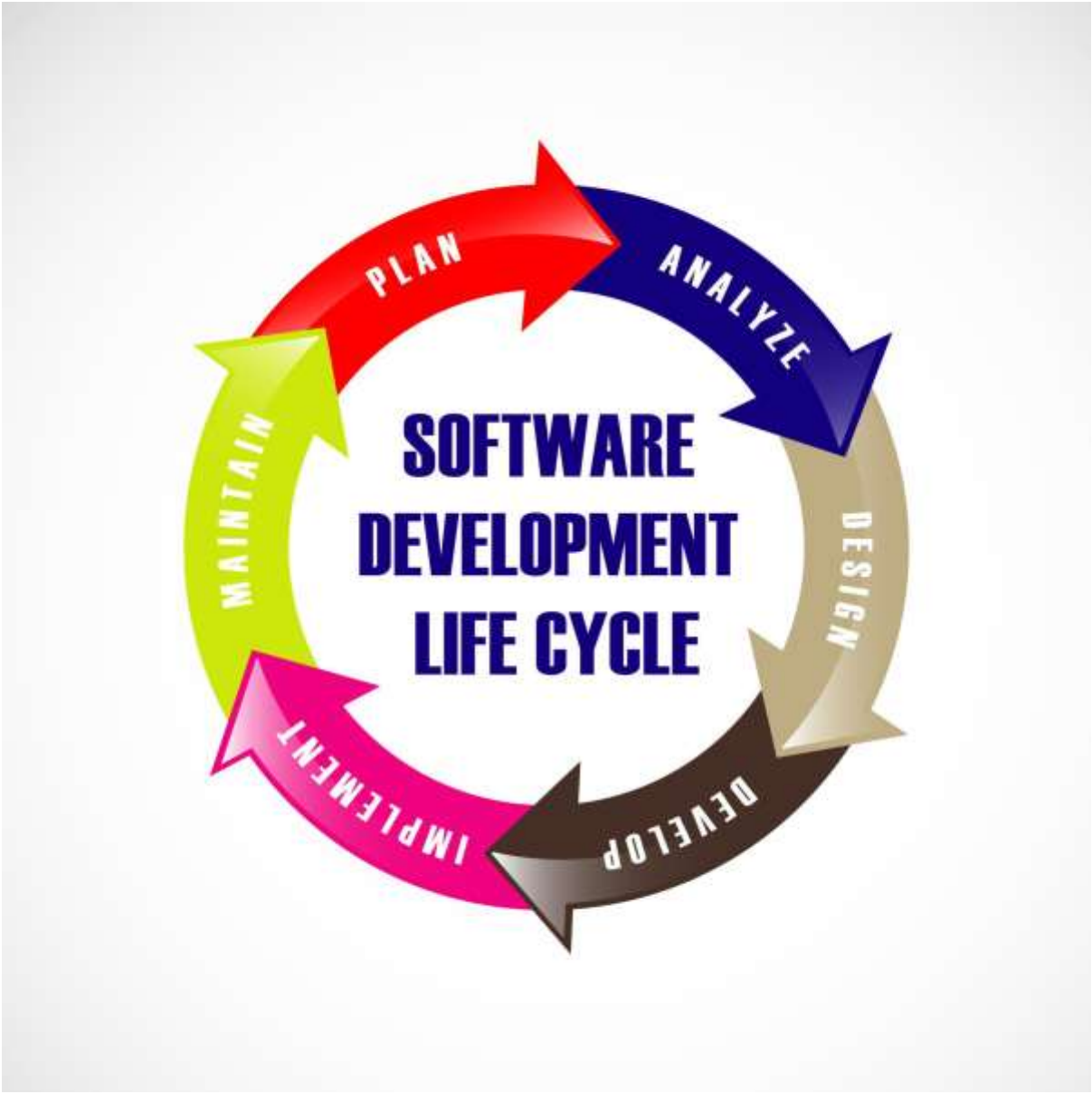
7. Maintenance

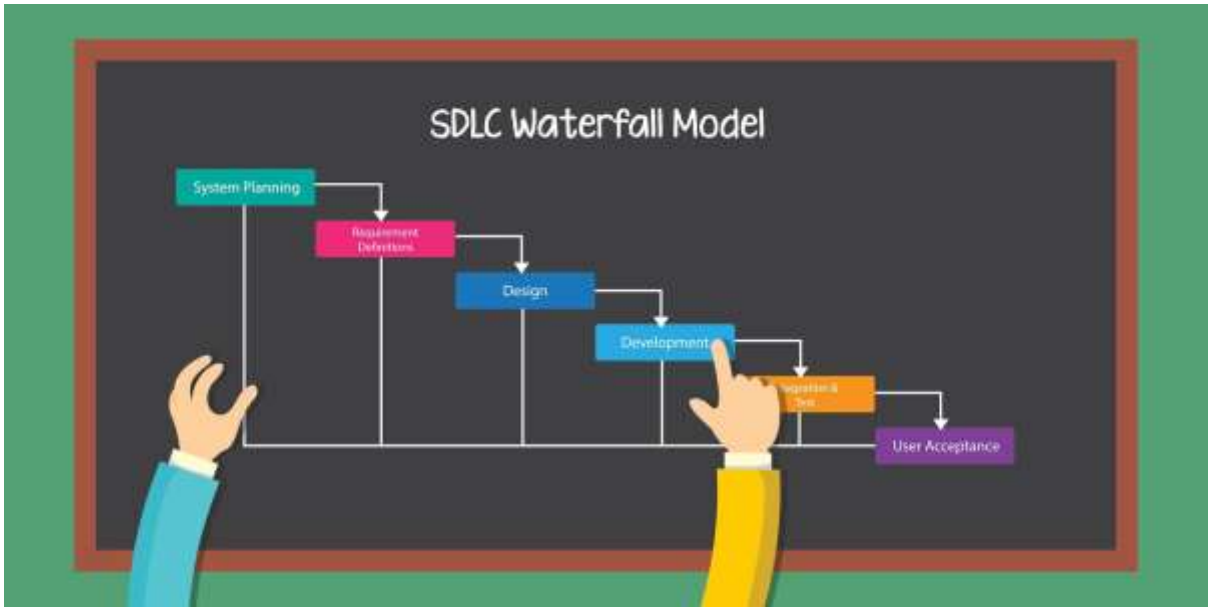
- Bug fixing
- Performance improvement
- Feature upgrades

This cycle continues throughout the software's lifetime.

🖼️ **Diagram (SDLC Flow)**


7 Stages of Software Product Development
1
Idea Generation
2
Requirement Analysis
3
Product Design
4
Product Development
5
Integration & Testing
6
Deployment
7
Maintenance


PLAN
ANALYZE
DESIGN
DEVELOP
IMPLEMENT
MAINTAIN
SOFTWARE DEVELOPMENT LIFE CYCLE



4

✨ Characteristics

- Step-by-step development approach
- Clear documentation at every phase
- Defined deliverables
- Risk management
- Quality assurance integration
- Team collaboration
- Predictable timelines
- Traceability from requirements to implementation
- Supports scalability
- Improves project visibility

🧪 Example

Hospital Management System

- Requirements: patient records, billing, appointments
- Planning: budget and timeline
- Design: database + interface
- Development: coding modules
- Testing: verify reports, login, billing
- Deployment: hosted on cloud
- Maintenance: add online consultation feature

This complete workflow follows SDLC.

👍 Advantages

- Produces high-quality software
- Reduces project failure risk
- Improves cost control
- Easy monitoring of progress
- Early detection of errors
- Better documentation
- Improves customer satisfaction
- Enhances team productivity

👎 Disadvantages

- Time-consuming for large documentation
- Less flexible in traditional SDLC models
- Late changes are expensive
- Needs skilled professionals
- Not suitable for very small projects

- Can slow innovation if too rigid

🚀 Applications

- Web Development Projects
- Mobile Applications
- Banking Systems
- Healthcare Software
- ERP Systems
- Government Applications
- Embedded Systems
- AI Platforms
- E-commerce Websites

✅ Difference Between Waterfall Model and Agile Model (12 Points)

No.	Waterfall Model	Agile Model
1	Follows a **linear and sequential** approach	Follows an **iterative and incremental** approach
2	Requirements are **fixed at the beginning**	Requirements can **change anytime**
3	Customer involvement is **minimal**	Customer involvement is **continuous**
4	Testing is done **after development phase**	Testing is done **in every iteration**
5	Working software is delivered **at the end of project**	Working software is delivered **frequently**

No.	Waterfall Model	Agile Model
6	Changes are **difficult and costly**	Changes are **easy to implement**
7	Best suited for **small and well-defined projects**	Best suited for **large and evolving projects**
8	Documentation is **heavy and detailed**	Documentation is **light and adaptive**
9	Risk is **high**, discovered late	Risk is **low**, handled early
10	Progress is measured by **phase completion**	Progress is measured by **working software**
11	Team works in **separate phases**	Team works in **close collaboration**
12	Less flexibility	Highly flexible

✅ Developing Diagrams in UML

📘 Introduction

UML (Unified Modeling Language) diagrams are standardized visual representations used in Software Engineering to **design, visualize, specify, and document software systems**.

According to GeeksforGeeks, UML diagrams help developers and stakeholders clearly understand **system structure, behavior, and interactions** before actual coding starts. They act as a **blueprint of the software system**.

In simple words, UML diagrams convert complex software ideas into easy-to-understand visuals.

⚙️ Working (How UML Diagrams Are Developed)

Developing UML diagrams generally follows these steps:

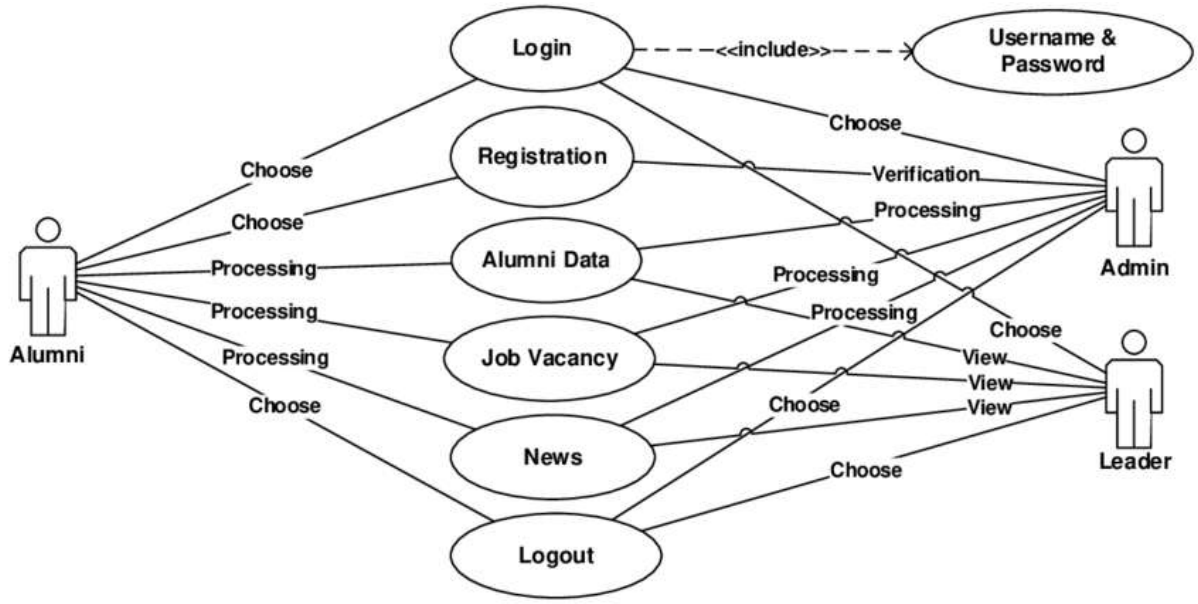
1. **Identify system requirements**
2. **Find actors and use cases**
3. **Define classes and objects**
4. **Establish relationships (association, inheritance, aggregation)**
5. **Model behavior (sequence, activity, state diagrams)**
6. **Review and refine diagrams**

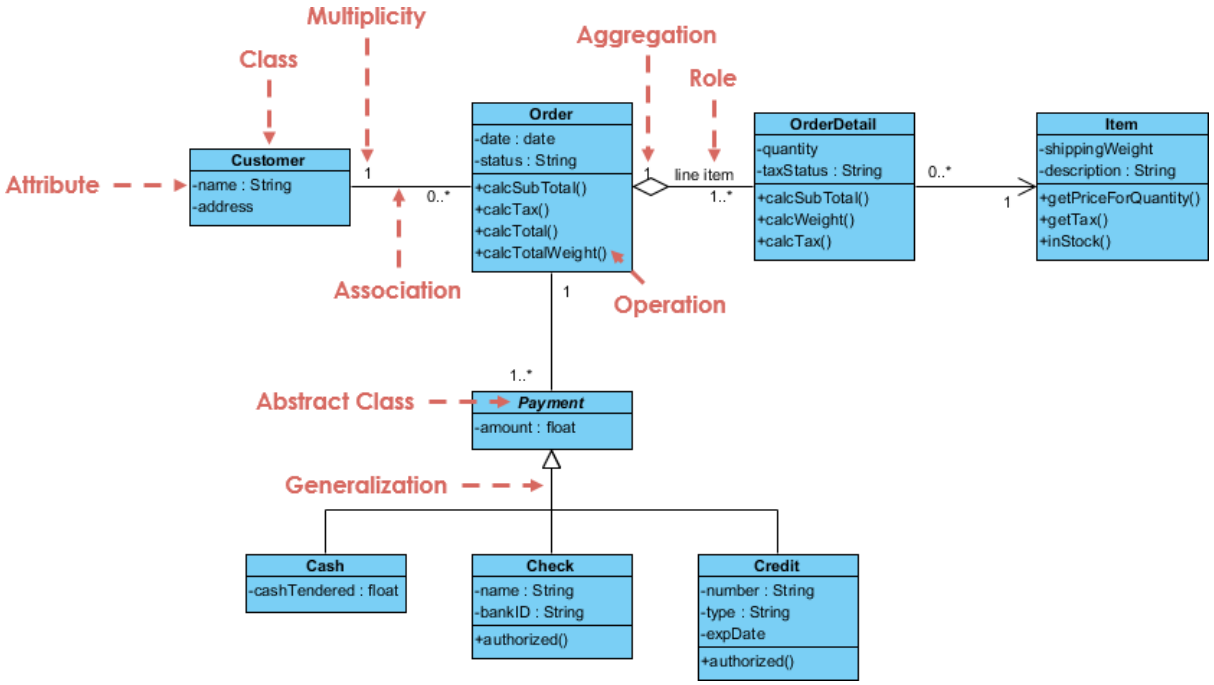
UML diagrams are divided into two main categories:

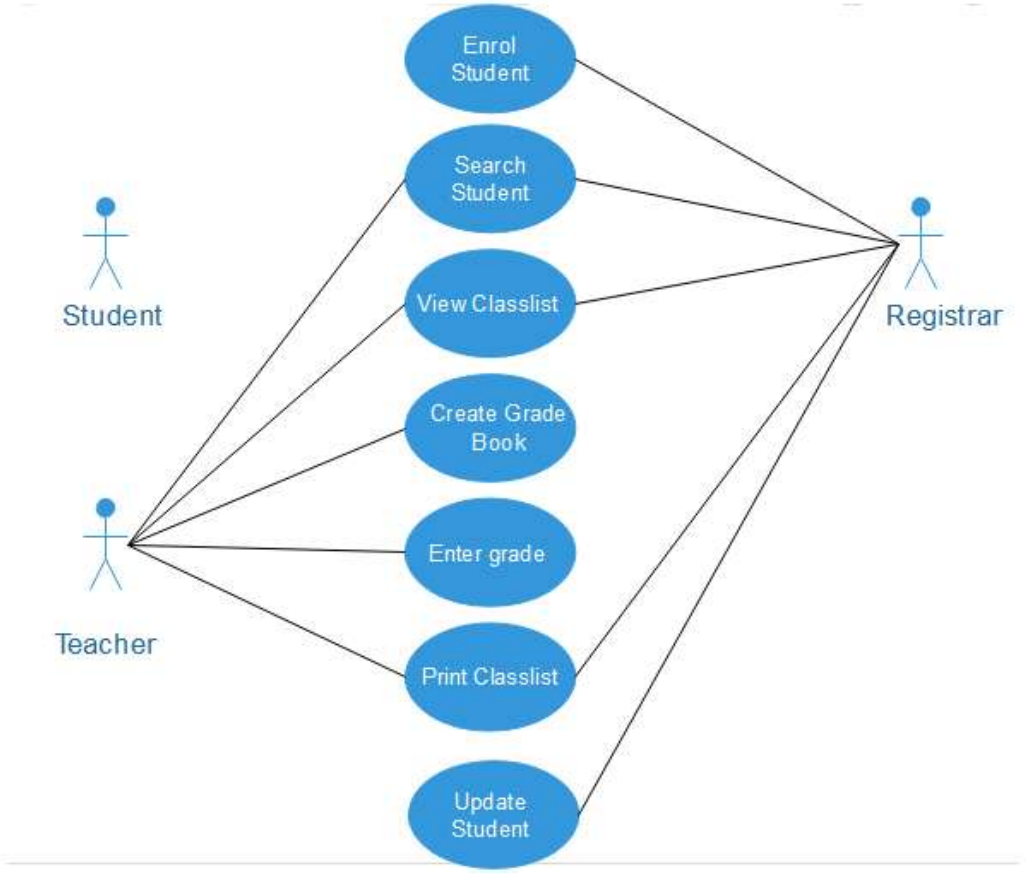
- 🔹 **Structural Diagrams**
  - Class Diagram
  - Object Diagram
  - Component Diagram
  - Deployment Diagram
- 🔹 **Behavioral Diagrams**
  - Use Case Diagram
  - Sequence Diagram
  - Activity Diagram
  - State Chart Diagram

Each diagram focuses on a **specific perspective** of the system.

🖼️ **Diagram (Common UML Diagrams)**


Login
<<include>>
Username & Password
Registration
Alumni Data
Job Vacancy
News
Logout
Alumni
Admin
Leader
Choose
Verification
Processing
View


Multiplicity
Class
Aggregation
Role
Attribute
Association
Operation
Abstract Class
Generalization
Customer
-name : String
-address
Order
-date : date
-status : String
+calcSubTotal()
+calcTax()
+calcTotal()
+calcTotalWeight()
line item
OrderDetail
-quantity
-taxStatus : String
+calcSubTotal()
+calcWeight()
+calcTax()
Item
-shippingWeight
-description : String
+getPriceForQuantity()
+getTax()
+inStock()
Payment
-amount : float
Cash
-cashTendered : float
Check
-name : String
-bankID : String
+authorized()
Credit
-number : String
-type : String
-expDate
+authorized()



4

✨ Characteristics

- Standardized notation
- Visual representation of system
- Easy to understand
- Platform independent
- Supports object-oriented design
- Improves communication between teams
- Helps detect design errors early
- Provides clear documentation
- Scalable for large systems

🧪 Example

For a **Library Management System**:

- Use Case Diagram → Borrow book, return book
- Class Diagram → Book, Student, Librarian
- Sequence Diagram → Student requests book → System verifies → Book issued
- Activity Diagram → Login → Search book → Issue book

Each diagram explains a **different view** of the same system.

👍 Advantages

- Improves system visualization
- Reduces development errors
- Better team communication
- Saves development time
- Easy maintenance
- Helps in requirement clarification
- Supports documentation
- Enhances software quality

👎 Disadvantages

- Requires UML knowledge
- Time-consuming for small projects
- Can become complex for large systems
- Tool dependency

- Not executable by itself
- Over-documentation risk

✅ Use Case Diagram for DBATU University

📘 Introduction

A **Use Case Diagram** is a UML behavioral diagram that represents **interactions between users (actors) and a system**. It visually shows what functions the system provides and who can access them.

As explained by GeeksforGeeks, use case diagrams help in **requirement analysis** by clearly identifying system functionalities from an end-user perspective.

For **Dr. Babasaheb Ambedkar Technological University (DBATU)**, a use case diagram models how students, faculty, and administrators interact with the university management system.

⚙️ Working (How the Use Case Diagram Is Built)

Steps to develop the DBATU Use Case Diagram:

1. Identify **actors** (Student, Faculty, Admin)
2. Identify **use cases** (Login, View Results, Register Courses, Upload Marks, etc.)
3. Draw system boundary (DBATU System)
4. Connect actors to their respective use cases
5. Add relationships if required (include / extend)

🔹 Main Actors

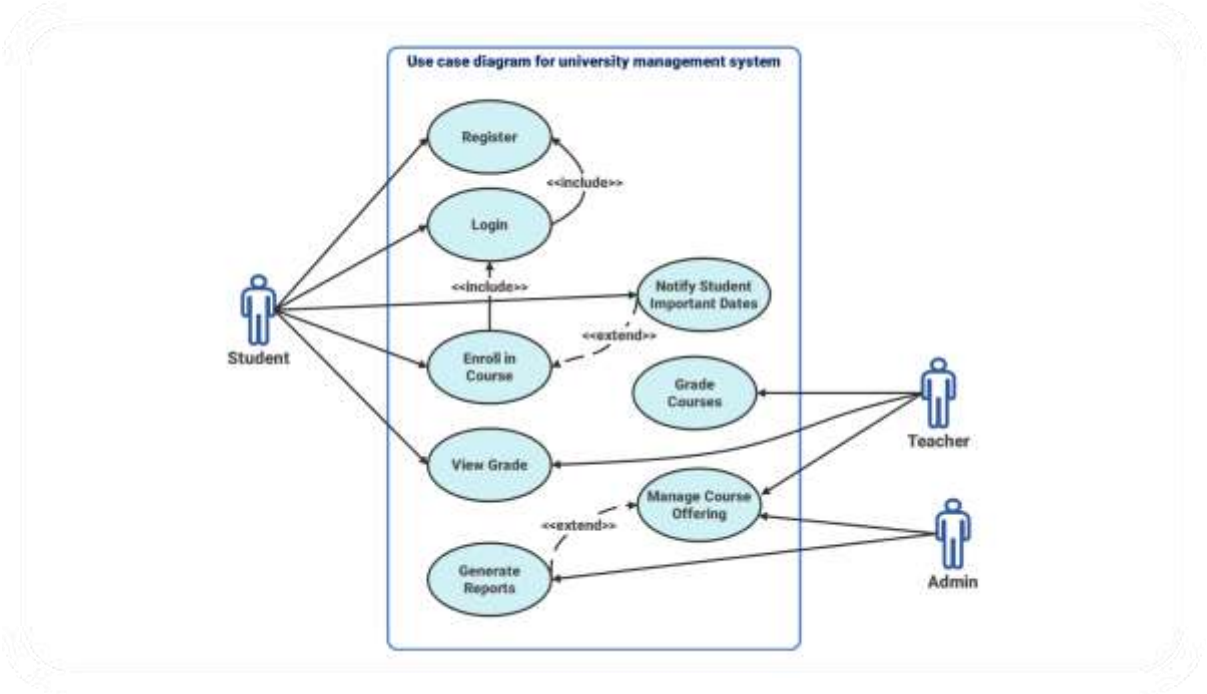
- Student
- Faculty

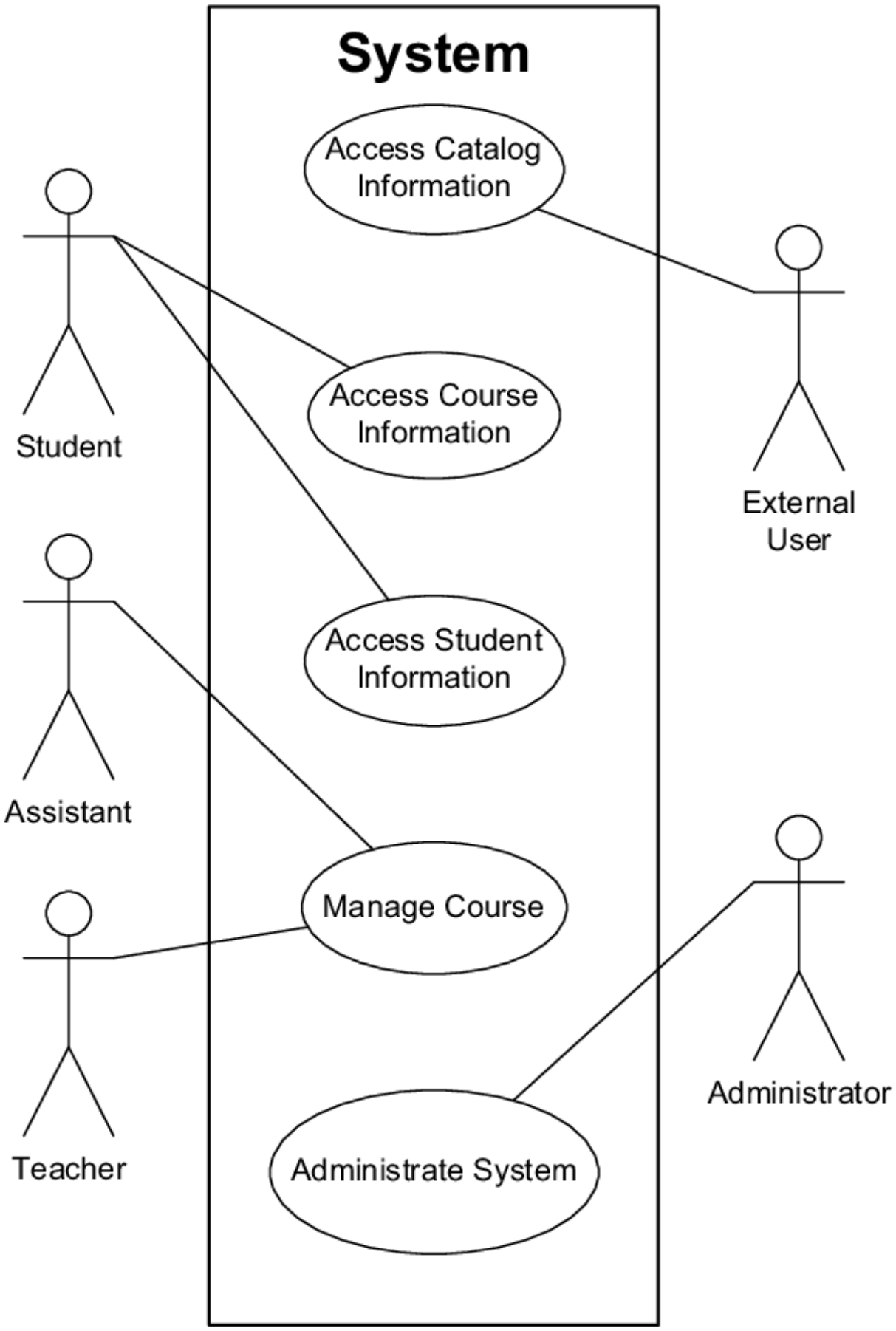
- Administrator

🔹 Main Use Cases

- Login
- Course Registration
- View Results
- Pay Fees
- Upload Marks
- Manage Students
- Generate Reports

🖼️ Diagram (Use Case Diagram – University Management System)




System
Access Catalog Information
Access Course Information
Access Student Information
Manage Course
Administrate System
Student
Assistant
Teacher
External User
Administrator

4

(These diagrams closely represent how a DBATU University system use case model is structured.)

Design Quality and Design Concepts

📘 **Introduction**

In Software Engineering, **Design Quality** refers to how well a software design meets functional requirements while remaining **simple, maintainable, scalable, and reliable**.
Design Concepts are fundamental principles that guide developers in creating high-quality software architectures.

According to GeeksforGeeks, good design quality ensures that software is **easy to understand, modify, test, and reuse**, while design concepts provide the **theoretical foundation** for achieving this quality.

In simple terms:
👉 *Design Quality = How good the design is*
👉 *Design Concepts = Rules and ideas used to make it good*

⚙️ **Working (How Design Quality Is Achieved Using Design Concepts)**

Design quality is achieved by applying key design concepts during the design phase:

🔹 **Abstraction**

Focuses on essential features and hides unnecessary details.

🔹 **Modularity**

Divides the system into independent modules.

- **Encapsulation**

Bundles data and methods together and restricts direct access.

- **Information Hiding**

Internal details of modules are hidden from other modules.

- **Separation of Concerns**

Each part of the system handles a specific responsibility.

- **Cohesion**

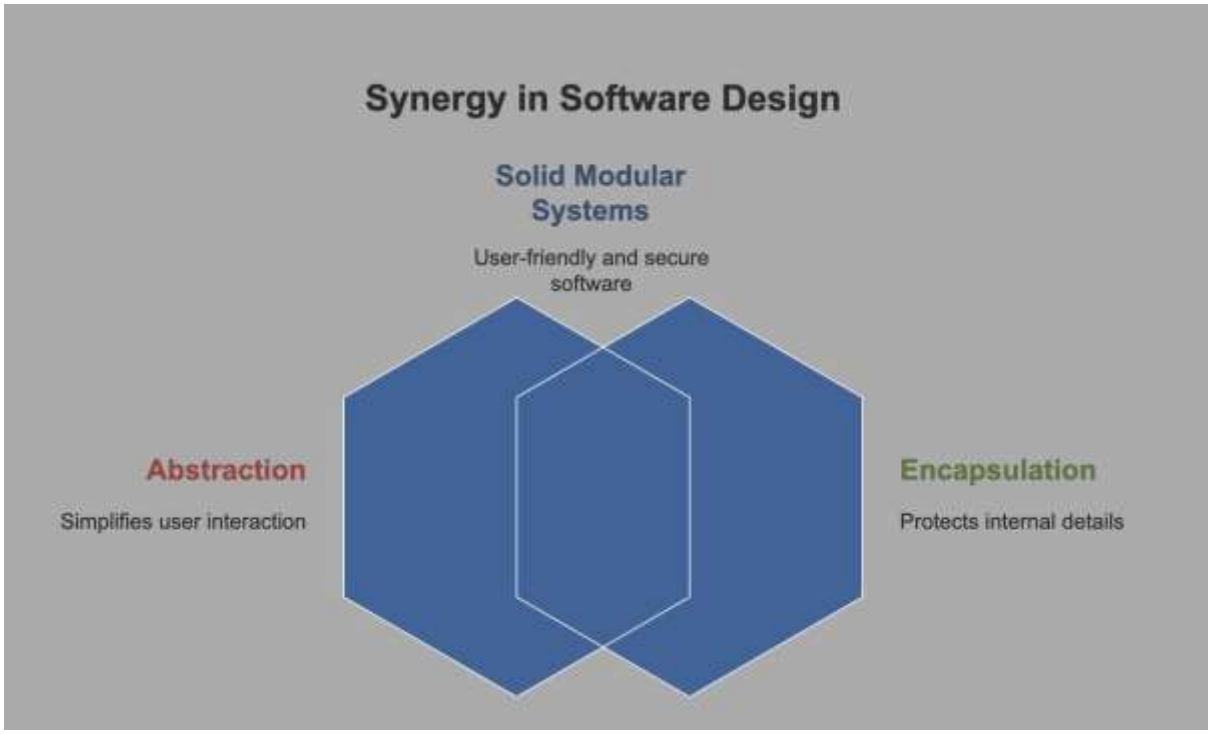
Each module should perform **only one well-defined task**.

- **Coupling**

Modules should have **minimal dependency** on each other.

By applying these concepts, developers improve readability, flexibility, and maintainability.

🖼️ **Diagram (Design Concepts Overview)**



4

✨ Characteristics of Good Design Quality

- Simple and understandable structure
- High cohesion
- Low coupling
- Easy to modify
- Reusable components
- Scalable architecture
- Well-documented
- Efficient performance
- Robust and reliable

🧪 Example

Online Shopping System

- Abstraction → Customer sees products without knowing backend logic
- Modularity → Separate modules for payment, cart, login
- Encapsulation → User data protected inside classes
- Low Coupling → Payment module independent of cart module
- High Cohesion → Each module performs a single task

This results in a clean, maintainable system.

👍 Advantages

- Improves software maintainability
- Reduces development complexity
- Makes debugging easier
- Enhances reusability
- Supports scalability
- Improves system reliability
- Enables parallel development
- Increases customer satisfaction

✅ Principles of Software Testing

📘 Introduction

The **Principles of Software Testing** are a set of fundamental guidelines that help testers perform effective testing and improve software quality.

According to GeeksforGeeks, these principles explain **how testing should be approached**, what expectations are realistic, and how defects can be discovered efficiently. They ensure testing is **systematic, meaningful, and cost-effective**.

In simple words, testing principles act as **rules that guide successful software testing**.

⚙️ Working (How Testing Principles Are Applied)

During a project, testers apply these principles throughout SDLC:

1. Testing starts early (from requirement phase)
2. Test cases are designed based on requirements

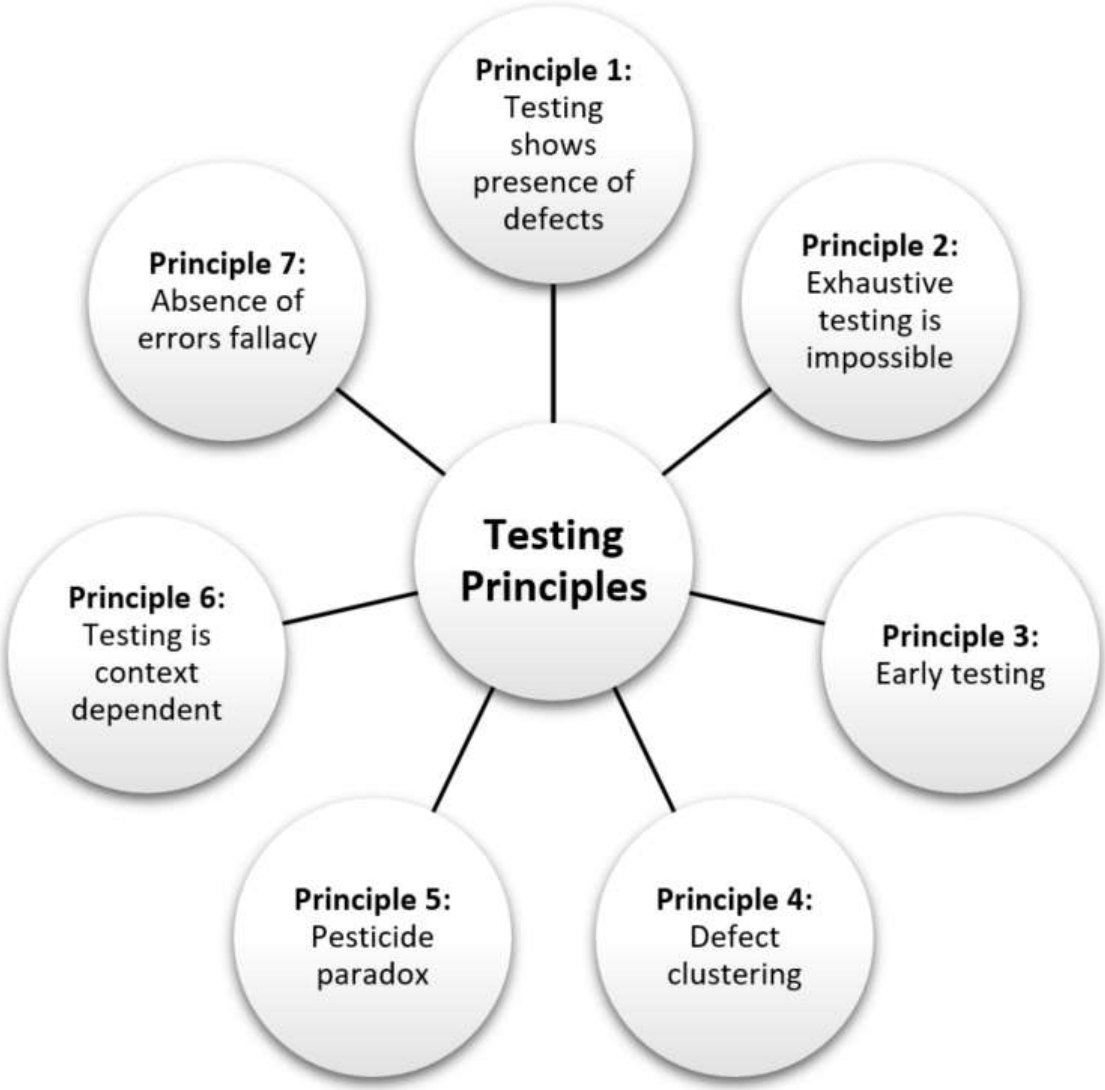
3. Critical modules are tested first
4. Defects are logged and tracked
5. Retesting and regression testing are performed
6. Testing is stopped when risk is acceptable

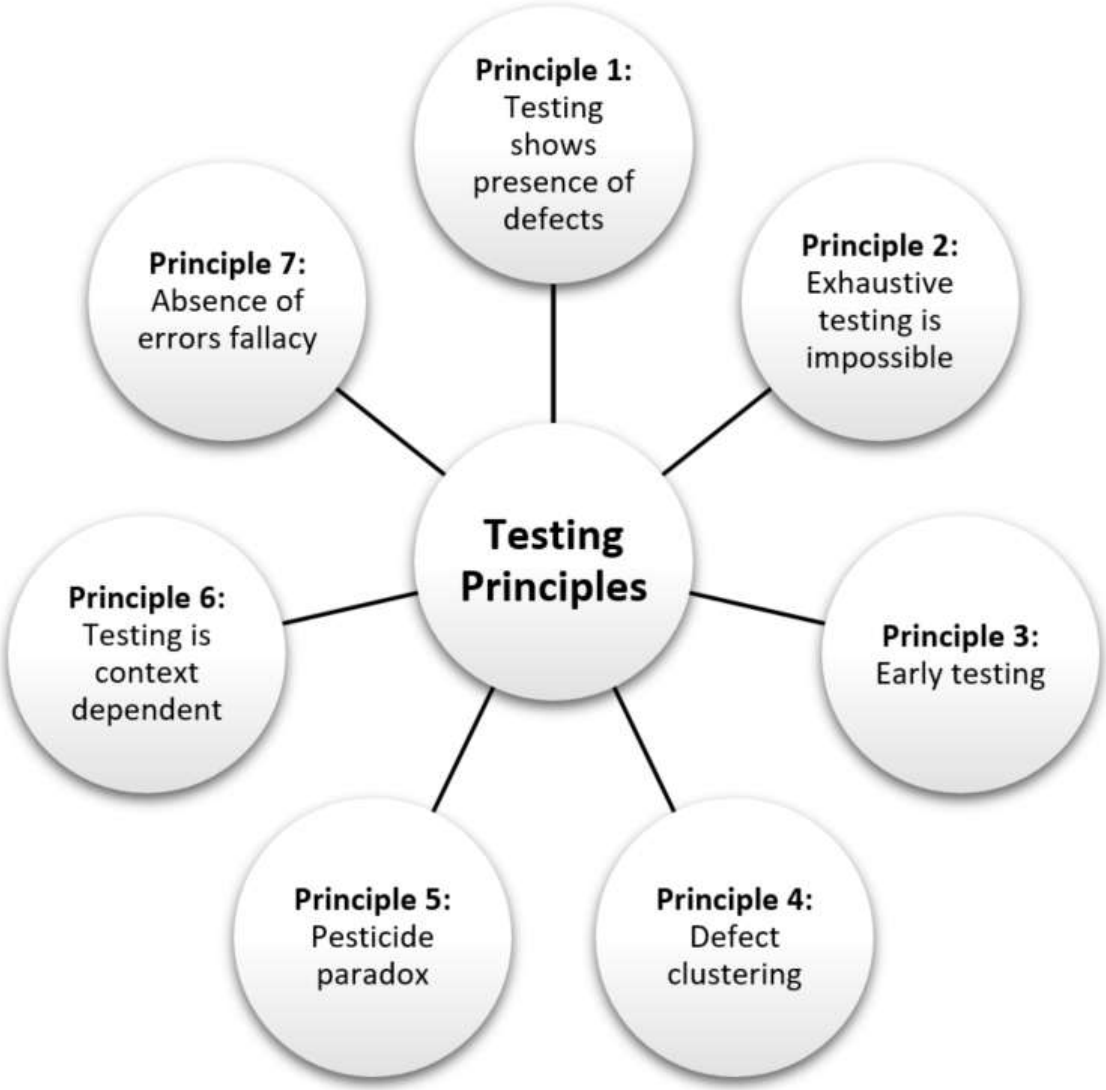
🔹 **Commonly Accepted Testing Principles:**

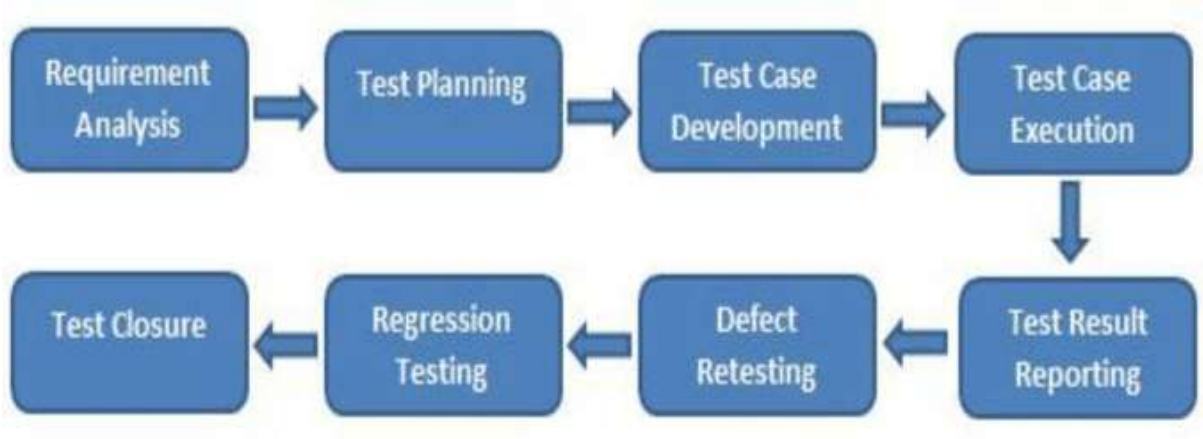
1. Testing shows presence of defects
2. Exhaustive testing is impossible
3. Early testing saves time and cost
4. Defect clustering
5. Pesticide paradox
6. Testing is context dependent
7. Absence-of-errors fallacy

These principles help testers focus on **high-risk areas** and improve product reliability.

🖼️ **Diagram (Testing Principles Overview)**


Testing Principles
Principle 1: Testing shows presence of defects
Principle 2: Exhaustive testing is impossible
Principle 3: Early testing
Principle 4: Defect clustering
Principle 5: Pesticide paradox
Principle 6: Testing is context dependent
Principle 7: Absence of errors fallacy





4

✨ Characteristics

- Defect-oriented approach
- Risk-based testing
- Continuous feedback
- Early involvement of testers
- Focus on critical areas
- Adaptive to project context
- Improves product quality
- Cost-efficient defect detection

🧪 Example

Login Module Testing

- Testing proves presence of bugs, not their absence
- Cannot test all input combinations (exhaustive testing impossible)
- Most defects found in authentication logic (defect clustering)
- Reusing same test cases misses new bugs (pesticide paradox)

This shows how testing principles work in real systems.

👍 Advantages

- Improves software reliability
- Reduces development cost
- Early defect detection
- Better test planning
- Focuses on high-risk modules
- Saves project time

- Enhances customer satisfaction
- Produces stable software

✅ STLC (Software Testing Life Cycle) with Advantages & Disadvantages

📘 Introduction

STLC (Software Testing Life Cycle) is a systematic process followed by testers to ensure that software meets quality standards before release.

According to GeeksforGeeks, STLC defines **specific testing phases**, each with clear objectives and deliverables, to detect defects early and improve overall product reliability.

In simple words, STLC explains **how testing is planned, executed, and completed step by step**.

⚙️ Working (Phases of STLC)

STLC works through the following structured phases:

1. Requirement Analysis

- Understand functional & non-functional requirements
- Identify testable features

2. Test Planning

- Prepare test strategy
- Estimate effort and resources
- Select testing tools

3. Test Case Development

- Write test cases
- Prepare test data
- Review test cases

4. Test Environment Setup

- Configure hardware/software
- Validate environment readiness

5. Test Execution

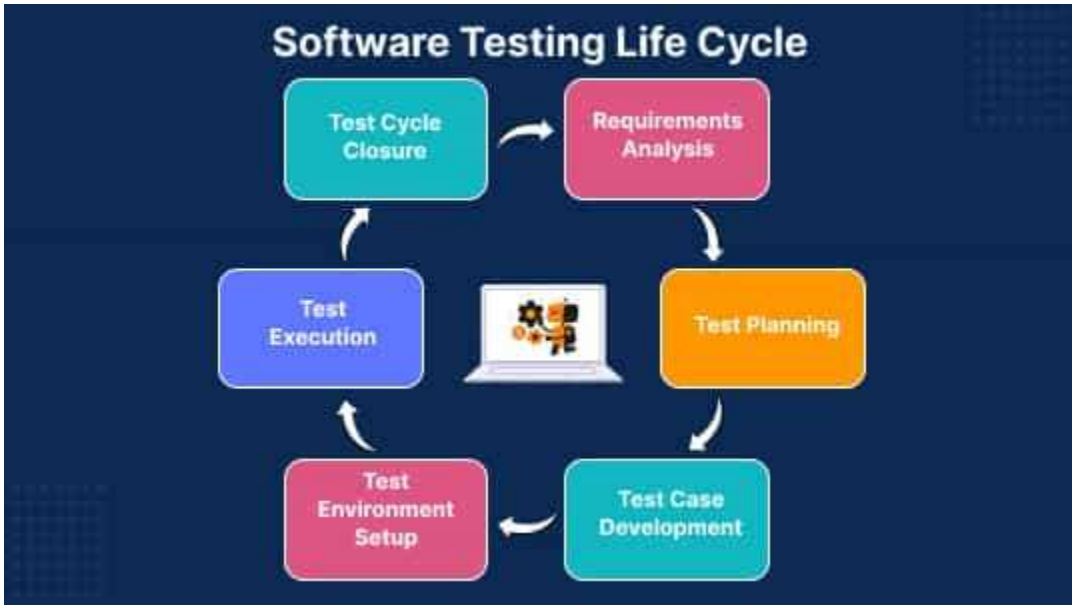
- Execute test cases
- Log defects
- Perform retesting & regression testing

6. Test Cycle Closure

- Evaluate exit criteria
- Prepare test summary report
- Analyze defect trends

Each phase has **entry and exit criteria**, ensuring disciplined testing.

🖼️ **Diagram (STLC Flow)**


Software Testing Life Cycle
Test Cycle Closure
Requirements Analysis
Test Execution
Test Planning
Test Environment Setup
Test Case Development


03
Test Case Development
04
Environment Setup
02
Test Planning
05
Test Execution
SOFTWARE TESTING LIFE CYCLE
01
Requirement Analysis
06
Test Case Closure

4

✨ Characteristics

- Structured testing approach
- Clear phase-wise activities
- Early defect detection
- Defined deliverables
- Quality-focused process
- Risk reduction
- Supports documentation
- Improves test coverage

🧪 Example

E-Commerce Website Testing

- Requirement Analysis → Product search, payment, login
- Test Planning → Choose manual + automation
- Test Case Development → Checkout scenarios
- Environment Setup → Browser + server setup
- Execution → Run tests, log payment bugs
- Closure → Prepare final test report

This complete testing flow follows STLC.

👍 **Advantages**

- Improves software quality
- Detects defects early
- Reduces project risk
- Better test planning
- Clear accountability
- Saves rework cost
- Enhances customer satisfaction
- Produces reliable software

✅ **Difference Between White Box Testing and Black Box Testing (12 Points)**

No.	White Box Testing	Black Box Testing
1	Internal code structure is **known** to tester	Internal code structure is **unknown** to tester

No.	White Box Testing	Black Box Testing
2	Also called **Structural Testing**	Also called **Functional Testing**
3	Focuses on **logic, paths, and code flow**	Focuses on **functionality and requirements**
4	Performed mainly by **developers**	Performed mainly by **testers**
5	Requires **programming knowledge**	Does **not** require programming knowledge
6	Test cases are based on **code**	Test cases are based on **requirements/specifications**
7	Used mostly for **unit testing**	Used mostly for **system and acceptance testing**
8	Finds **hidden logical errors**	Finds **missing or incorrect functionality**
9	Covers **statements, branches, and paths**	Covers **inputs and outputs**
10	Helps optimize code	Helps validate user expectations
11	Difficult to scale for large systems	Easy to apply for large systems
12	Examples: Statement coverage, Branch coverage	Examples: Equivalence partitioning, Boundary value analysis

✅ **Integration Testing**

📘 **Introduction**

Integration Testing is a level of software testing in which **individual modules or components are combined and tested as a group** to verify their interactions.

According to GeeksforGeeks, Integration Testing focuses on detecting **interface defects, data flow issues, and communication errors** between integrated units after unit testing is completed.

In simple words, Integration Testing checks whether **different parts of software work correctly together**.

⚙️ Working (How Integration Testing Is Performed)

Integration Testing is carried out after Unit Testing and before System Testing.

Main Approaches:

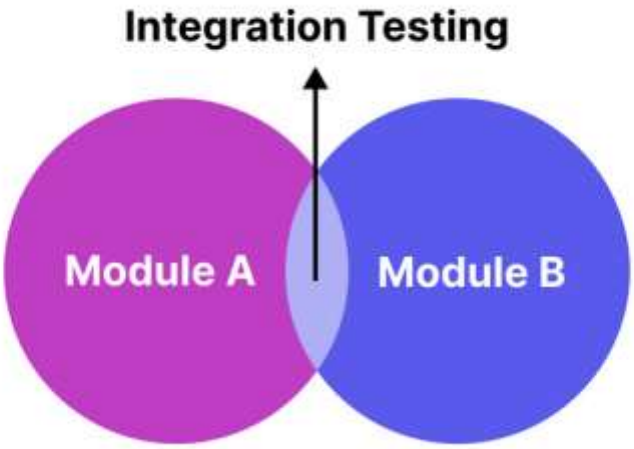
1. **Big Bang Integration**
    - All modules integrated at once
    - Tested as a whole system
2. **Top-Down Integration**
    - Top-level modules tested first
    - Lower modules added gradually (using stubs)
3. **Bottom-Up Integration**
    - Lower-level modules tested first
    - Higher modules added later (using drivers)
4. **Sandwich / Hybrid Integration**
    - Combination of Top-Down and Bottom-Up

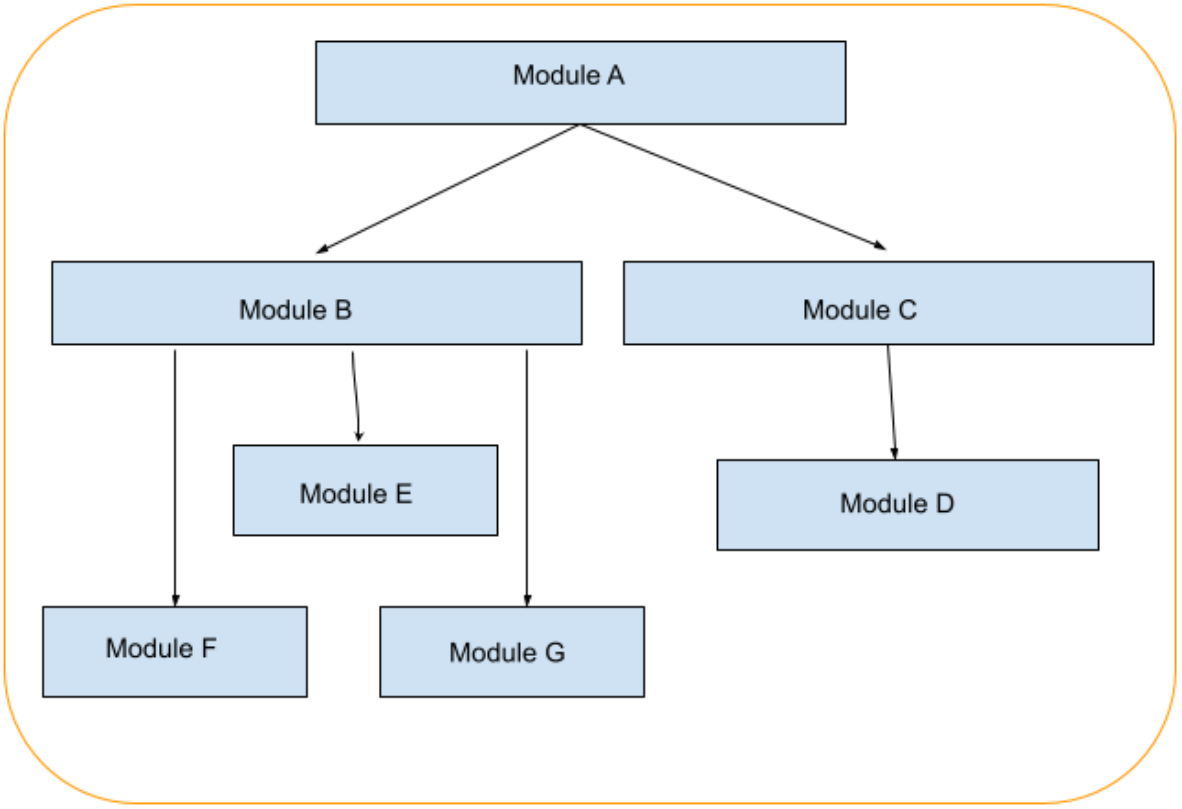
General Steps:

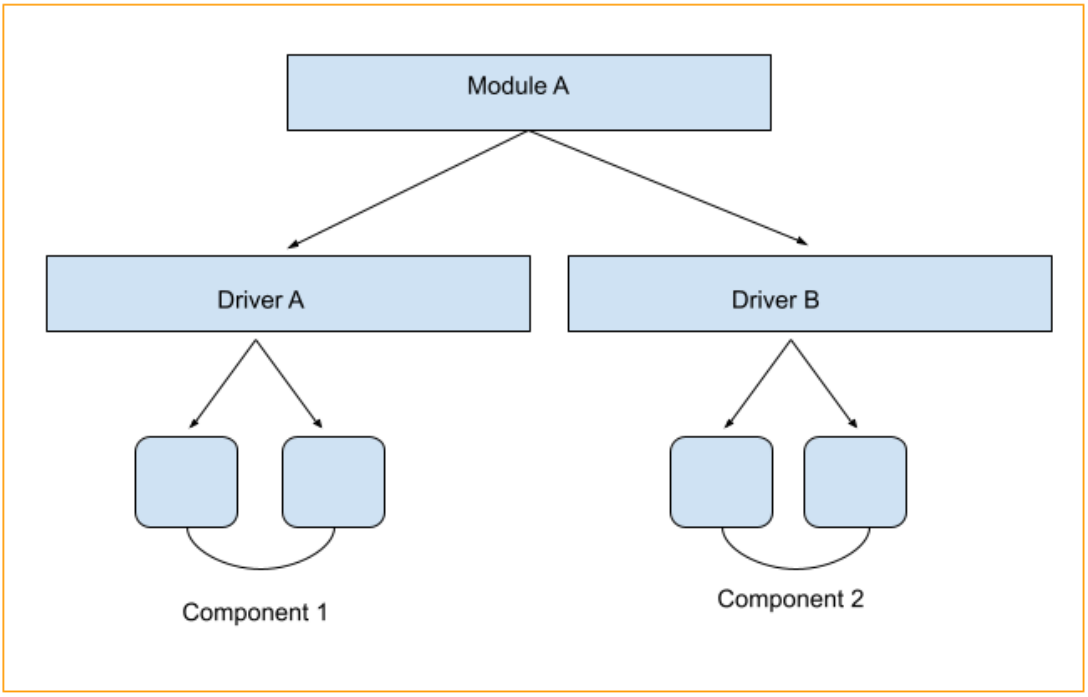
- Combine tested modules
- Execute test cases

- Verify data exchange
- Detect interface defects
- Fix issues and retest

🖼️ Diagram (Integration Testing Approaches)







4

✨ Characteristics

- Performed after unit testing
- Focuses on module interaction
- Detects interface defects
- Uses stubs and drivers
- Can be incremental or big bang
- Improves system reliability
- Early detection of integration issues
- Supports structured testing

🧪 Example

Online Shopping System

- Cart module integrated with Payment module
- Payment module integrated with Order module

Testing verifies:

- Cart passes correct data to payment
- Payment confirmation reaches order system

If payment succeeds but order fails, Integration Testing identifies this interface issue.

👍 Advantages

- Detects interface defects early
- Improves system stability
- Reduces risk in later testing stages
- Ensures smooth data flow
- Helps identify module dependency problems
- Saves debugging time
- Improves overall quality

\

✅ Difference Between System Testing and Acceptance Testing (12 Points)

No.	System Testing	Acceptance Testing
1	Tests the **entire integrated system**	Tests whether system meets **customer requirements**
2	Performed after **integration testing**	Performed after **system testing**

No.	System Testing	Acceptance Testing
3	Conducted by **QA/Test team**	Conducted by **client/end users or customer representatives**
4	Focuses on **functional and non-functional behavior**	Focuses on **business needs and user acceptance**
5	Checks system against **technical specifications**	Checks system against **user requirements**
6	Identifies **system-level defects**	Confirms **readiness for release**
7	Includes performance, security, usability testing	Mainly validates **real-world usage scenarios**
8	Executed in **testing environment**	Executed in **production-like environment**
9	Goal is to verify **correct working of system**	Goal is to get **final approval from client**
10	Uses test cases prepared by QA team	Uses acceptance criteria defined by customer
11	Mandatory internal testing phase	Final external validation phase
12	Result: bug fixes and improvements	Result: product accepted or rejected

✅ **Functional System Testing**

📘 **Introduction**

Functional System Testing is a type of **system-level testing** that verifies whether the complete integrated software system performs **according to specified functional requirements**.

As explained by GeeksforGeeks, Functional System Testing focuses on **what the system does**, not how it is internally implemented. It validates all features such as input handling, processing logic, and output generation from an end-user perspective.

In simple words, it checks whether **every function of the system works correctly**.

⚙️ Working (How Functional System Testing Is Performed)

Functional System Testing is carried out after Integration Testing and before Acceptance Testing.

Step-by-step process:

1. Study functional requirements (SRS document)
2. Identify test scenarios and test cases
3. Prepare test data
4. Execute test cases on full system
5. Compare actual results with expected results
6. Log defects if any
7. Retest after fixes
8. Perform regression testing if needed

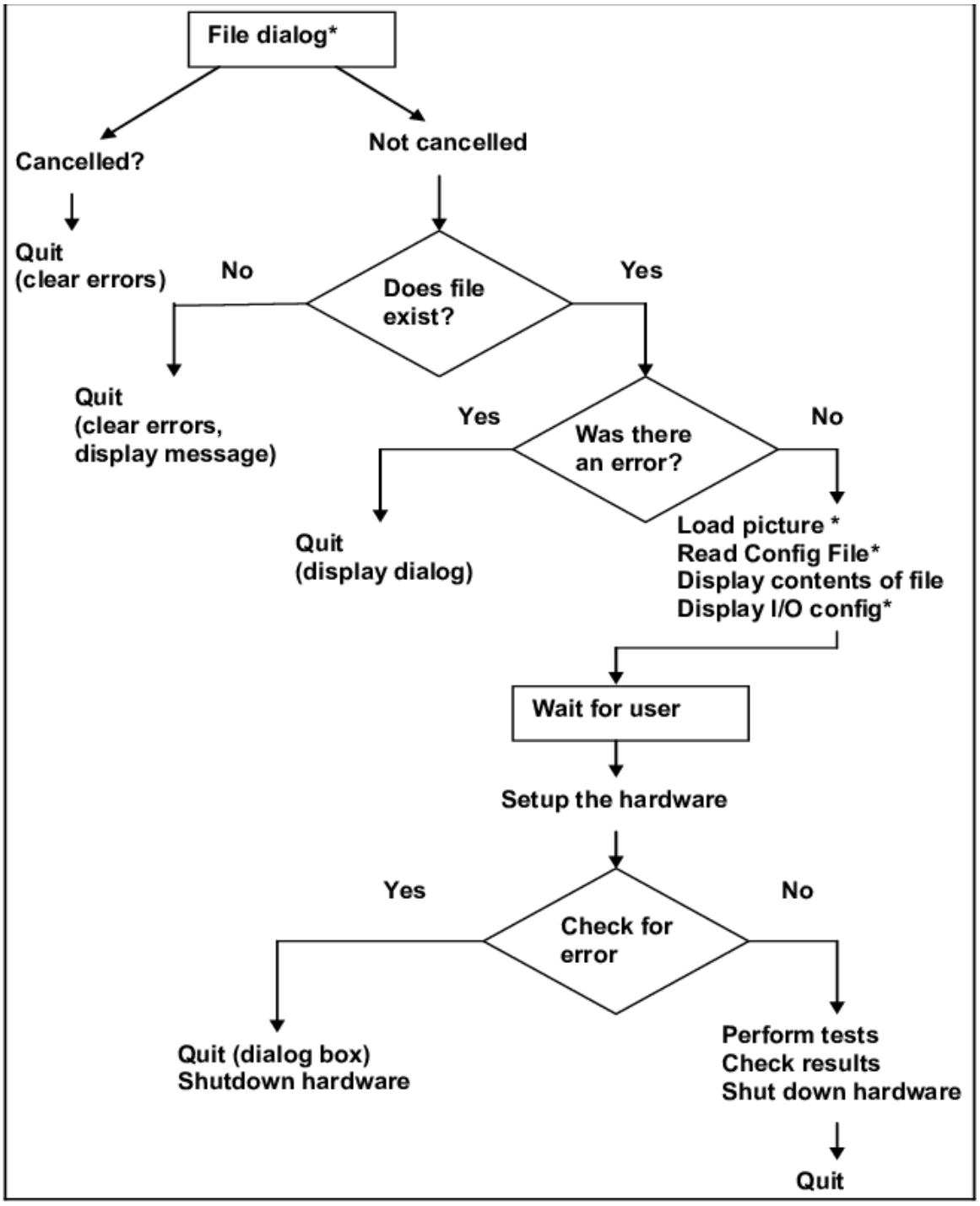
Key areas tested:

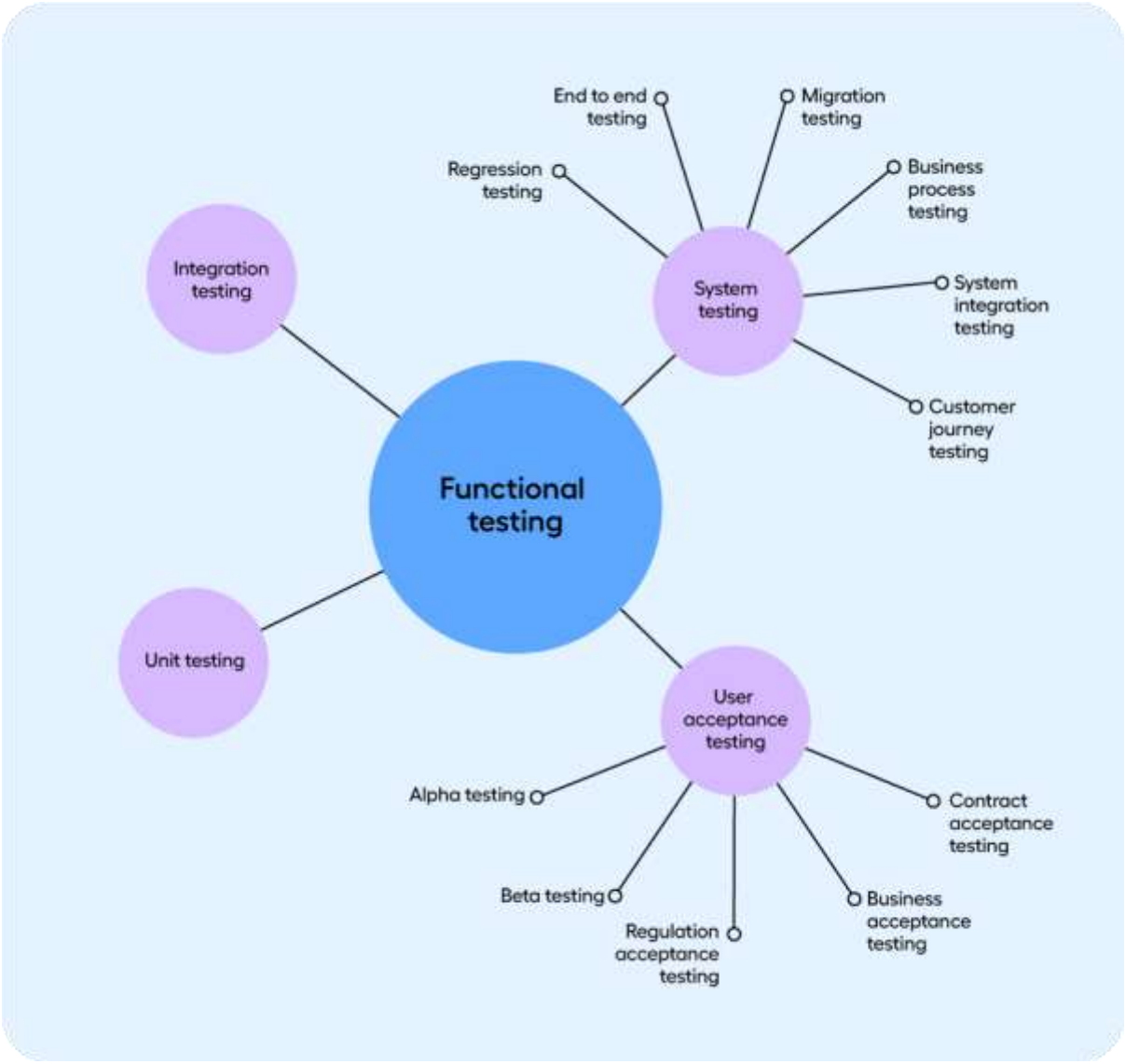
- Login / Authentication
- Data validation
- Business workflows
- Input–Output behavior

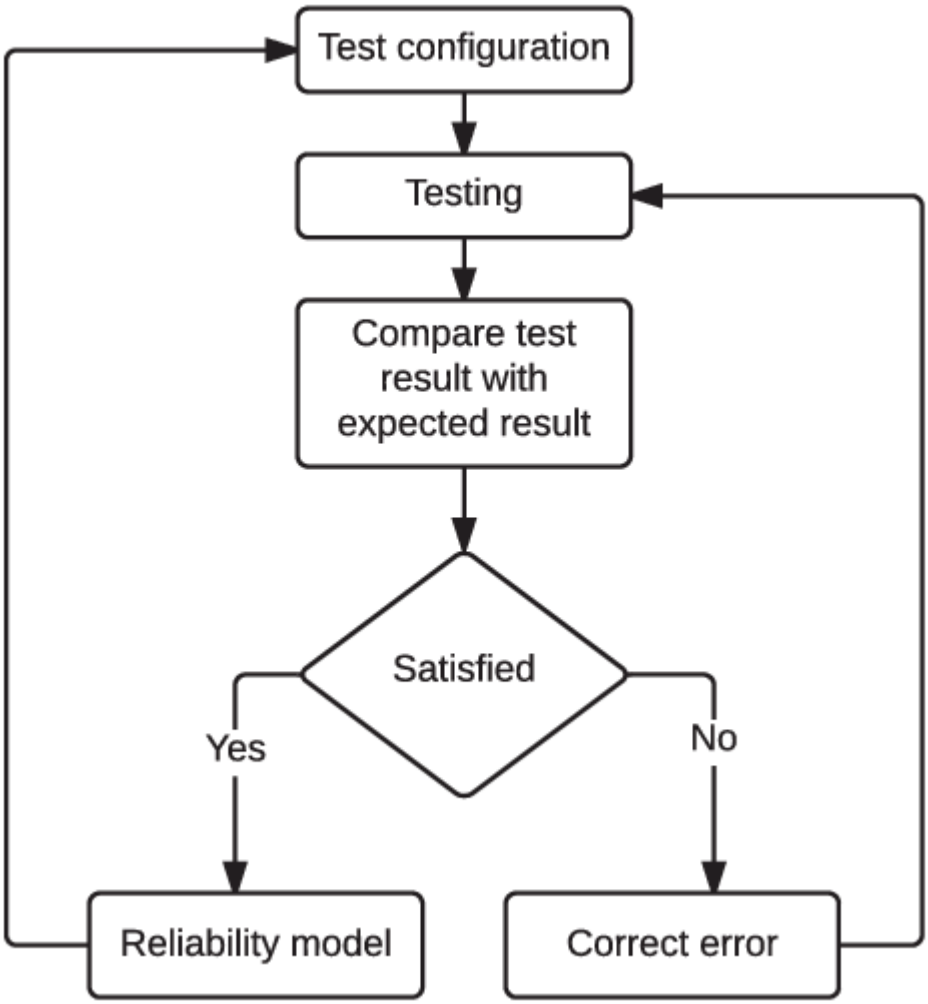
- Error messages
- User interfaces

It treats the software as a **black box** and focuses only on behavior.

🖼️ Diagram (Functional System Testing Flow)




Functional testing
Integration testing
Unit testing
System testing
End to end testing
Migration testing
Regression testing
Business process testing
System integration testing
Customer journey testing
User acceptance testing
Alpha testing
Beta testing
Regulation acceptance testing
Business acceptance testing
Contract acceptance testing



✨ Characteristics

- Requirement-based testing
- Black box testing approach
- Performed on fully integrated system
- Validates end-to-end functionality
- User-oriented testing
- Covers complete workflows
- Ensures business rules are followed
- Detects functional defects

- Independent of internal code

🧪 Example

Online Banking System

Functional System Testing checks:

- User can log in successfully
- Balance is displayed correctly
- Fund transfer works properly
- Transaction history is updated
- Error shown for invalid inputs

If fund transfer succeeds but history doesn't update, it is detected as a **functional defect**.

👍 Advantages

- Ensures system meets user requirements
- Improves product reliability
- Detects missing or incorrect functionality
- Validates complete workflows
- Easy to understand for non-technical users
- Increases customer confidence
- Reduces production defects

Difference Between Performance Testing and Regression Testing (12 Points)

No.	Performance Testing	Regression Testing
1	Checks **speed, stability, and scalability**	Checks **existing functionality after changes**
2	Focuses on **system behavior under load**	Focuses on **bug revalidation**
3	Type of **non-functional testing**	Type of **functional testing**
4	Ensures application handles **high traffic/users**	Ensures new changes didn't break old features
5	Measures response time, throughput, resource usage	Verifies previously working features
6	Performed mainly before release	Performed after every code change
7	Uses tools like JMeter, LoadRunner	Can be manual or automated
8	Finds performance bottlenecks	Finds side-effect bugs
9	Executed on stable builds	Executed whenever updates occur
10	Needs performance environment	Uses normal test environment
11	Helps optimize system capacity	Helps maintain software stability
12	Examples: Load, Stress testing	Examples: Retesting login after update

Adhoc Testing

📘 **Introduction**

Adhoc Testing is an **informal and unstructured software testing technique** performed without any predefined test cases or documentation. The main objective is to **break the application by randomly checking functionalities**.

According to GeeksforGeeks, Adhoc Testing relies heavily on the **tester's experience, intuition, and creativity** to find defects that formal testing might miss.

In simple words, Adhoc Testing means **"test without planning"**.

⚙️ Working (How Adhoc Testing Is Performed)

Adhoc Testing does not follow any fixed procedure. The tester:

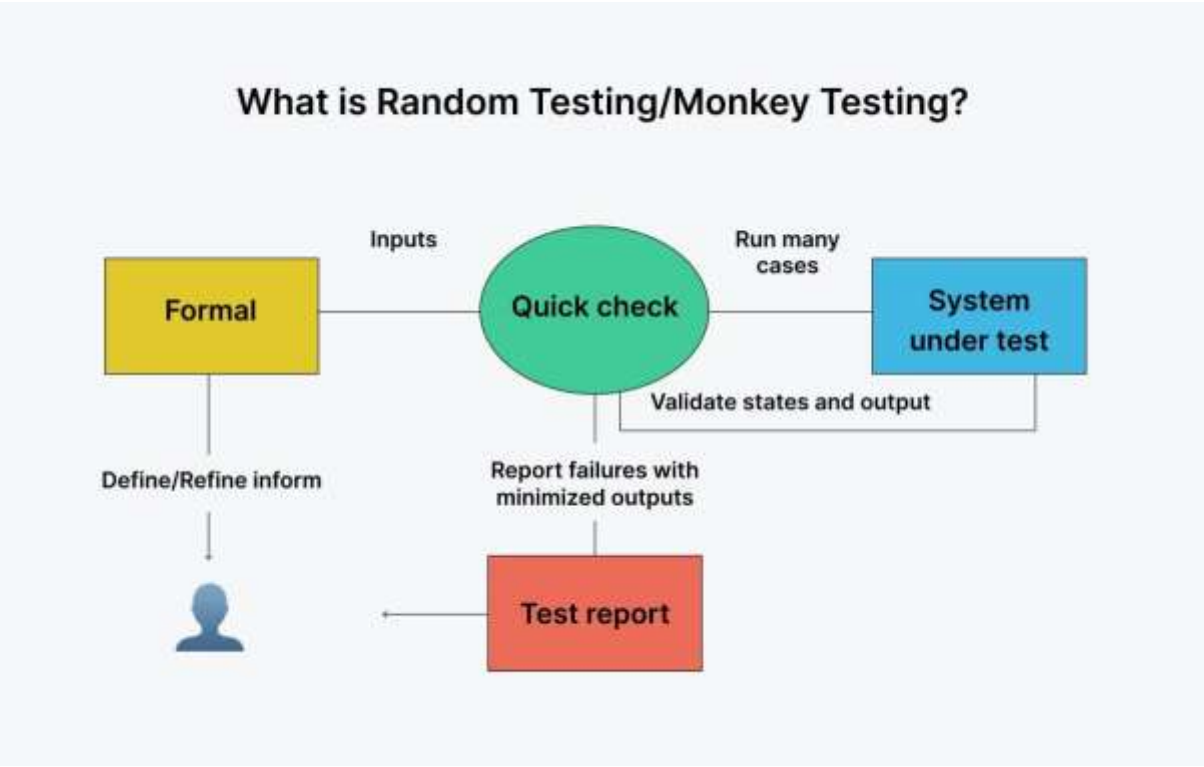
1. Understands the application briefly
2. Randomly explores different features
3. Tries unusual inputs and actions
4. Switches rapidly between modules
5. Observes unexpected behavior
6. Logs defects immediately

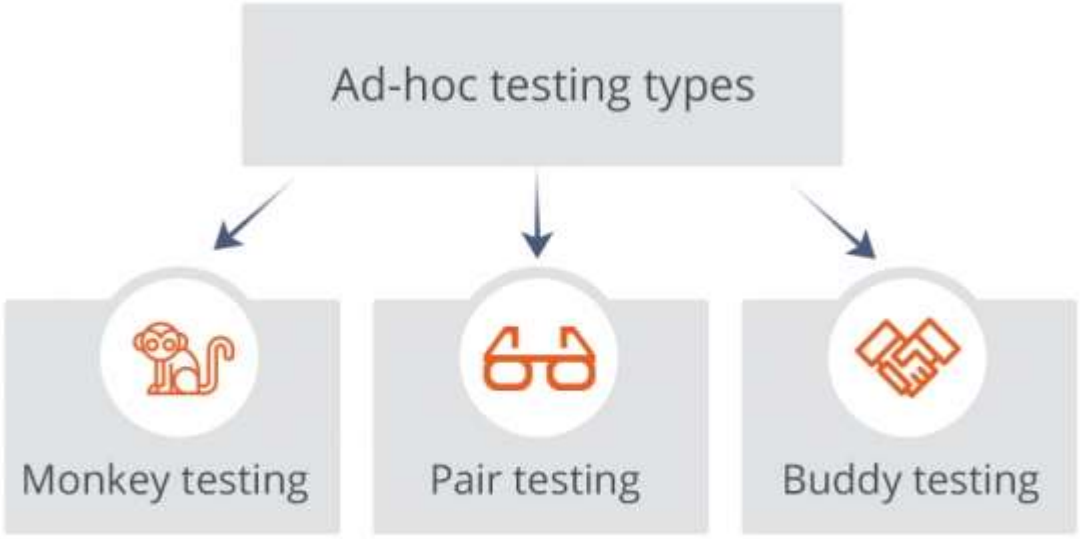
There are **no written test cases** — testing is purely experience-based.

🔹 Common Types of Adhoc Testing

- **Buddy Testing** – Developer + Tester test together
- **Pair Testing** – Two testers work simultaneously
- **Monkey Testing** – Random inputs without logic

🖼️ Diagram (Adhoc Testing Concept)





✨ Characteristics

- No formal documentation
- No predefined test cases

- Depends on tester’s skill
- Random execution
- Fast feedback
- Flexible testing
- Creative defect discovery
- Complements formal testing

🧪 Example

Login Page Testing

Tester randomly:

- Enters invalid usernames
- Pastes large text in password field
- Clicks login repeatedly
- Switches pages during login

If the system crashes or behaves oddly, the defect is recorded.
These issues may not appear in scripted testing.

👍 Advantages

- Quick to perform
- Finds critical hidden bugs
- No preparation required
- Cost-effective
- Encourages creative testing
- Improves defect detection
- Useful in early builds

- Enhances overall test coverage

Tools for Testing Object-Oriented Software

📘 Introduction

Testing Object-Oriented Software (OOS) requires specialized tools because OOP systems involve **classes, objects, inheritance, polymorphism, and encapsulation**, which make testing more complex than procedural software.

According to GeeksforGeeks, object-oriented testing tools help automate **unit testing, integration testing, regression testing, and system testing** while validating class behavior and object interactions.

In simple words, these tools ensure that **classes and objects work correctly—individually and together**.

⚙️ Working (How These Tools Are Used)

Object-oriented testing tools typically work in the following way:

1. Analyze class structure and relationships
2. Generate or execute test cases for methods
3. Validate object interactions
4. Perform regression testing after code changes
5. Produce coverage and defect reports

Commonly Used OOP Testing Tools (as referenced in GFG-style discussions):

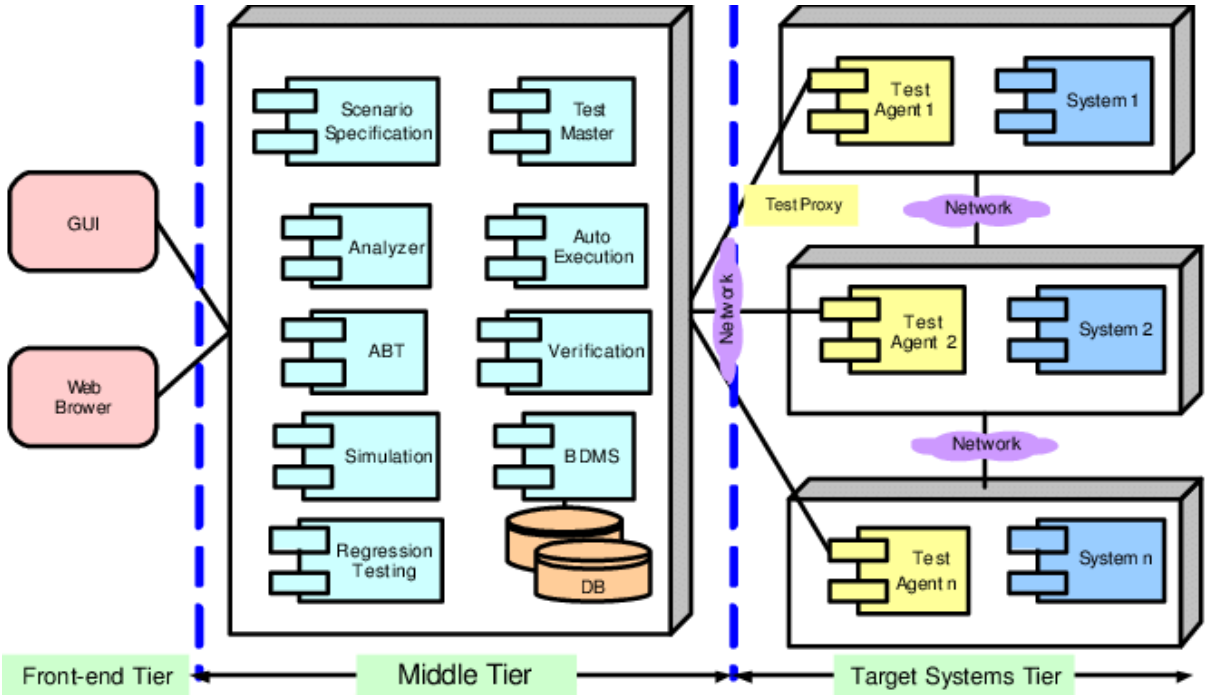
- **JUnit** – Unit testing framework for Java classes
- **TestNG** – Advanced testing framework (supports annotations & parallel execution)

- **Selenium** – Automates UI testing for OO-based web apps
- **Mockito** – Mocking framework for isolating classes
- **JUnit + Maven/Gradle** – For automated regression pipelines

These tools help test:

- Individual methods
- Class interactions
- Object states
- Inherited behavior
- Polymorphic calls

🖼️ Diagram (Object-Oriented Testing Tool Workflow)



✨ Characteristics

- Supports class-based testing
- Validates object interactions
- Enables automation
- Supports regression testing
- Provides code coverage reports
- Handles inheritance & polymorphism
- Improves test reusability
- Integrates with CI/CD pipelines
- Speeds up defect detection

🧪 Example

Online Shopping System (OOP based)

Classes:

- User
- Product
- Cart
- Payment

Using JUnit:

- Test Cart.addItem()
- Mock Payment using Mockito
- Verify order flow using Selenium UI tests

This ensures:

- Each class works correctly
- Objects interact properly

- UI reflects backend logic

👍 Advantages

- Automates repetitive testing
- Improves class-level reliability
- Early defect detection
- Supports continuous integration
- Reduces manual effort
- Enhances code quality
- Makes regression testing fast
- Handles complex OO relationships

Software Engineering Framework

📘 Introduction

A **Software Engineering Framework** provides a **structured foundation** for developing high-quality software by defining **activities, tasks, methods, and tools** required throughout the software life cycle.

According to GeeksforGeeks, the Software Engineering Framework acts as a **blueprint** that guides developers in planning, building, testing, and maintaining software in a systematic way.

In simple words, it tells us:

👉 *What to do*
👉 *When to do it*
👉 *How to do it*
👉 *Who does it*

⚙️ Working (How the Software Engineering Framework Operates)

The framework organizes software development into **five core framework activities**, supported by umbrella activities.

🔹 Core Framework Activities

1. **Communication** – Gather requirements from stakeholders
2. **Planning** – Estimate cost, schedule, and resources
3. **Modeling** – Design system architecture and components
4. **Construction** – Coding and testing
5. **Deployment** – Deliver software and collect feedback

🔹 Umbrella Activities (run throughout all phases)

- Project tracking & control
- Risk management
- Software quality assurance
- Technical reviews
- Configuration management
- Documentation
- Measurement

These ensure **quality, control, and continuous improvement**.

🖼️ Diagram – Software Engineering Framework

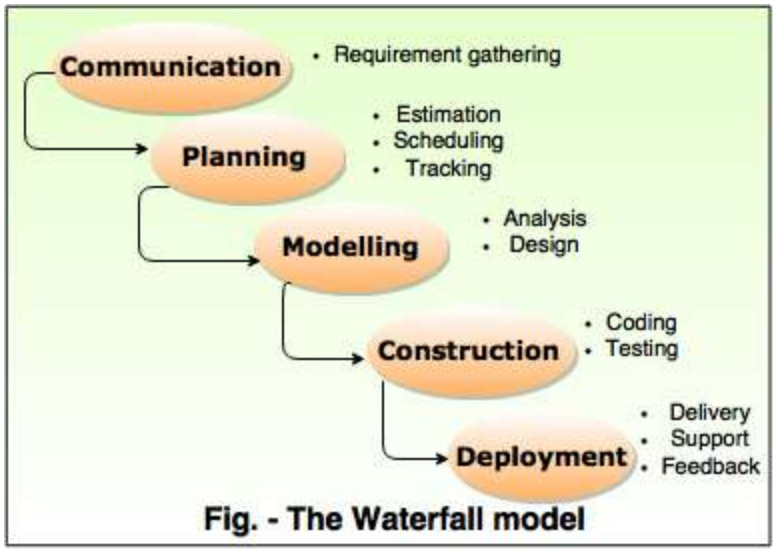
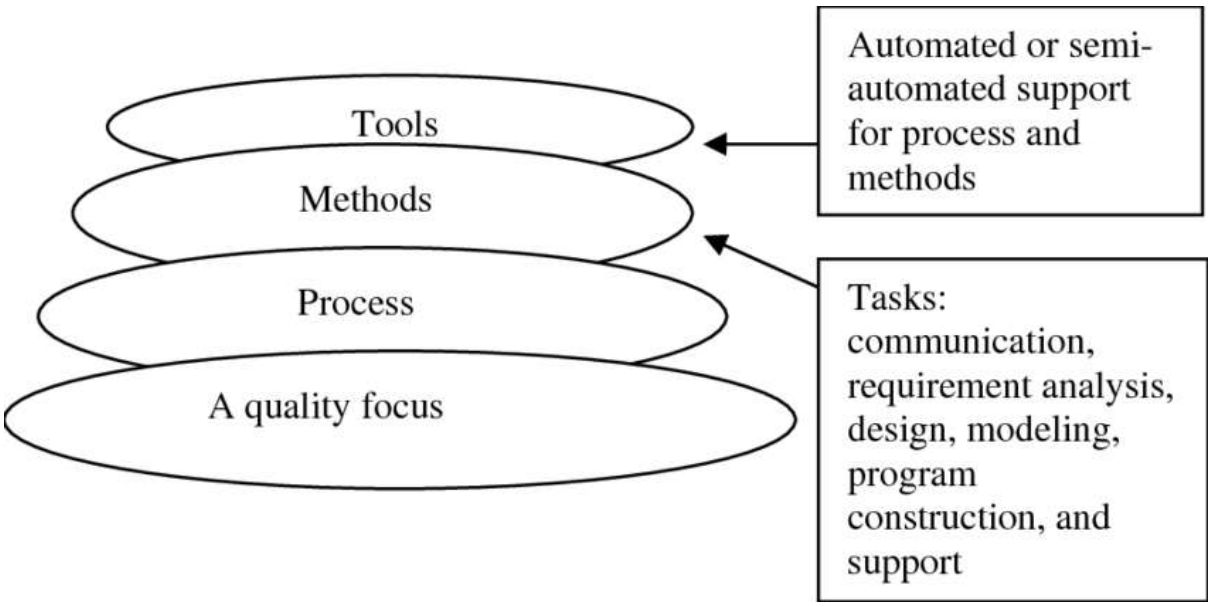


Fig. - The Waterfall model



4

(This represents the standard GFG/Pressman-style Software Engineering Framework.)

✨ Characteristics

- Systematic development approach
- Phase-wise execution
- Continuous quality monitoring
- Risk-driven planning
- Customer-focused process
- Supports documentation
- Scalable for large projects
- Encourages team collaboration
- Improves predictability

🧪 Example

Online Examination System

- Communication → Collect exam requirements
- Planning → Schedule development phases
- Modeling → Design database & UI
- Construction → Code login, test modules
- Deployment → Release system to students
- Umbrella → Track bugs, manage versions, ensure quality

Thus, the framework guides the entire project from start to finish.

👍 Advantages

- Improves software quality
- Reduces project risk
- Provides clear development structure
- Enhances team coordination
- Enables early defect detection
- Supports maintenance
- Ensures documentation
- Helps meet deadlines

Prescriptive Process Models

📘 Introduction

Prescriptive Process Models (also called traditional software process models) define a **fixed, structured sequence of activities** for software development. These models prescribe *how* software should be built step-by-step.

According to GeeksforGeeks, prescriptive models emphasize **planning, documentation, and well-defined phases**, making them suitable for projects with stable requirements.

📜 Enlist Different Prescriptive Models

Common prescriptive models are:

1. **Waterfall Model**
2. **Incremental Model**
3. **Prototyping Model**
4. **Spiral Model**
5. **V-Model (Verification & Validation Model)**
6. **RAD (Rapid Application Development) Model**

Now, let's **explain one model**.

✅ Waterfall Model (Explained)

📘 Introduction

The **Waterfall Model** is the **oldest and simplest prescriptive process model**. It follows a **linear and sequential** approach, where each phase must be completed before moving to the next.

As explained by GeeksforGeeks, this model is called *Waterfall* because progress flows **downward like a waterfall** through clearly defined stages.

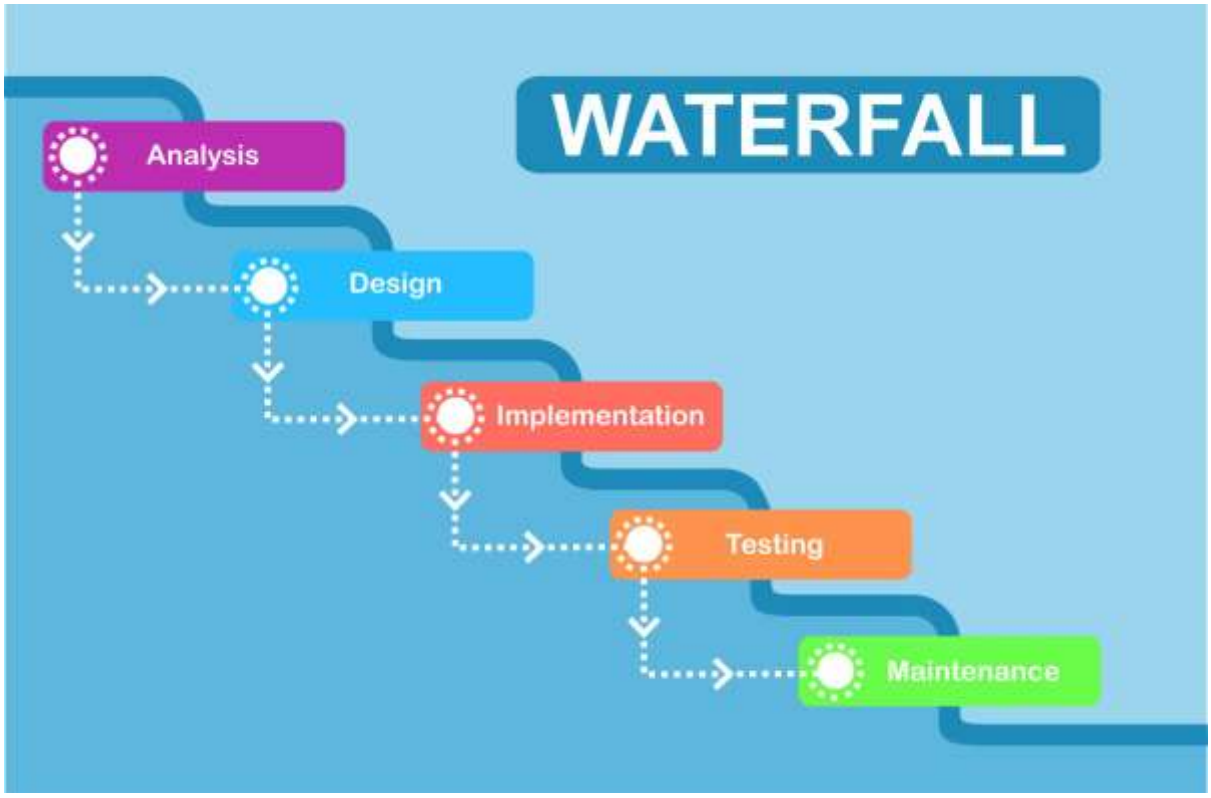
⚙️ Working (Phase-by-Phase Flow)

The Waterfall Model proceeds through these steps:

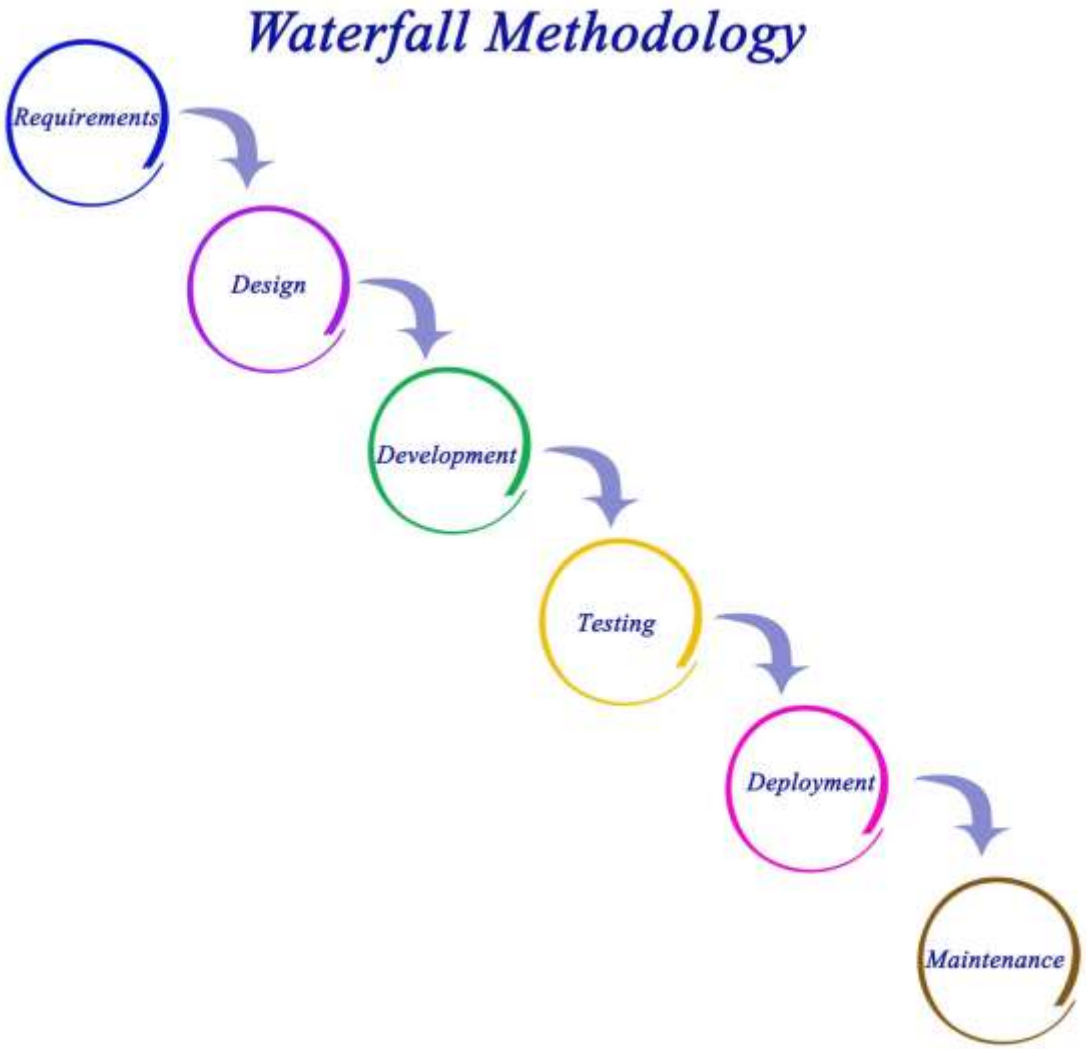
1. **Requirement Analysis** – Gather and document user needs
2. **System Design** – Create architecture and detailed design
3. **Implementation** – Convert design into code
4. **Testing** – Verify and validate the developed software
5. **Deployment** – Release software to users
6. **Maintenance** – Fix bugs and add enhancements

Each phase has **specific deliverables**, and output of one phase becomes input to the next.

🖼️ Diagram – Waterfall Model




Requirements
Design
Develop
Test
Deploy
WATERFALL



4

✨ **Characteristics**

- Linear and sequential flow
- Each phase completed only once
- Heavy documentation
- Clear milestones
- Simple to understand
- Rigid structure

- Minimal customer involvement after requirements
- Easy to manage

🧪 Example

College Management System

- Requirements → Student records, fees, attendance
- Design → Database + UI
- Implementation → Coding modules
- Testing → Validate reports and login
- Deployment → Install system in college
- Maintenance → Add new features like online exams

This project follows Waterfall because requirements are fixed.

👍 Advantages

- Simple and easy to understand
- Clear documentation
- Easy project tracking
- Suitable for small projects
- Well-defined stages
- Easy to manage
- Works well when requirements are stable

Agile View of Process

📘 Introduction

The **Agile View of Process** represents a **flexible, iterative, and customer-centric approach** to software development. Instead of following rigid sequential phases, Agile focuses on **rapid delivery of small working increments**, continuous feedback, and adaptation to change.

According to GeeksforGeeks, Agile treats software development as a **dynamic activity** where requirements evolve, and solutions grow through **collaboration between cross-functional teams and customers**.

In simple words, Agile views software development as a **cycle of plan → build → test → get feedback → improve**.

⚙️ Working (How the Agile Process Operates)

In Agile, development happens in **short iterations (called sprints)**. Each sprint delivers a **working piece of software**.

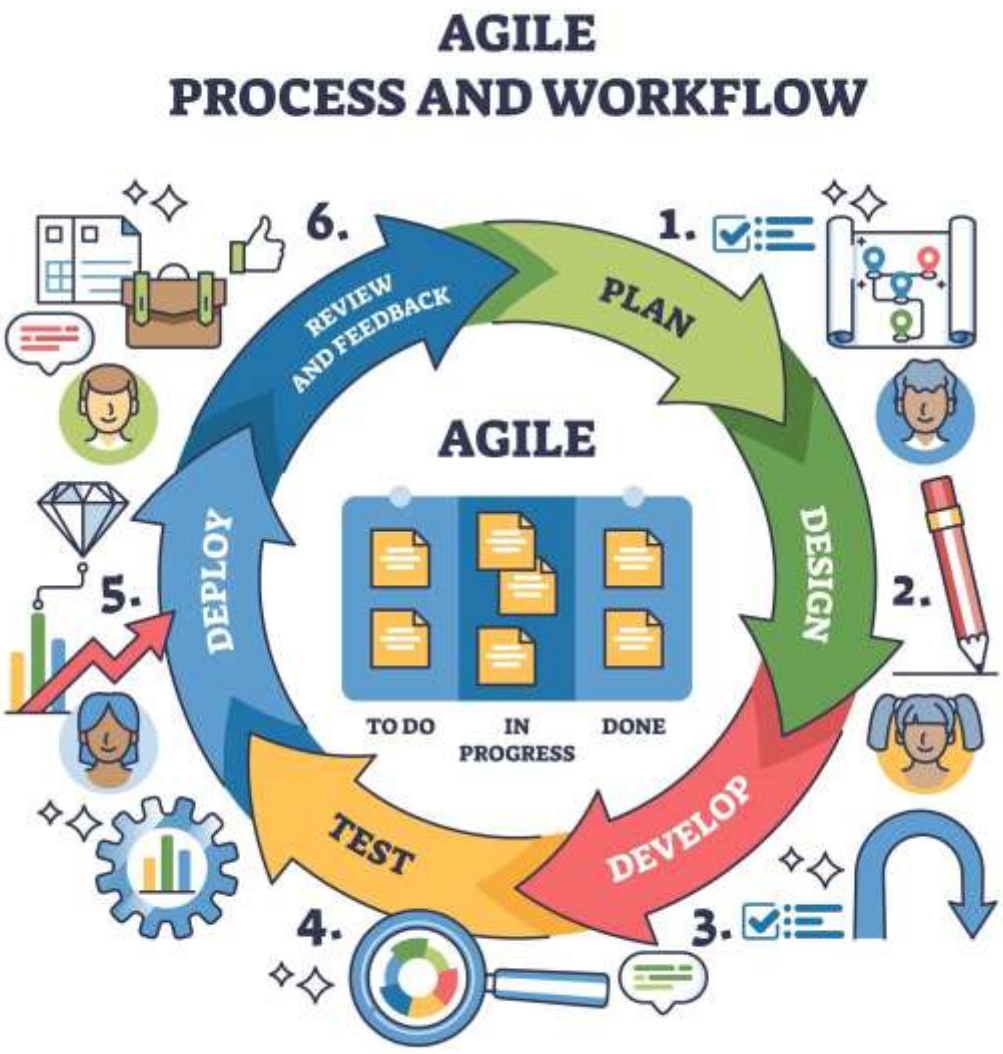
Typical Agile Flow:

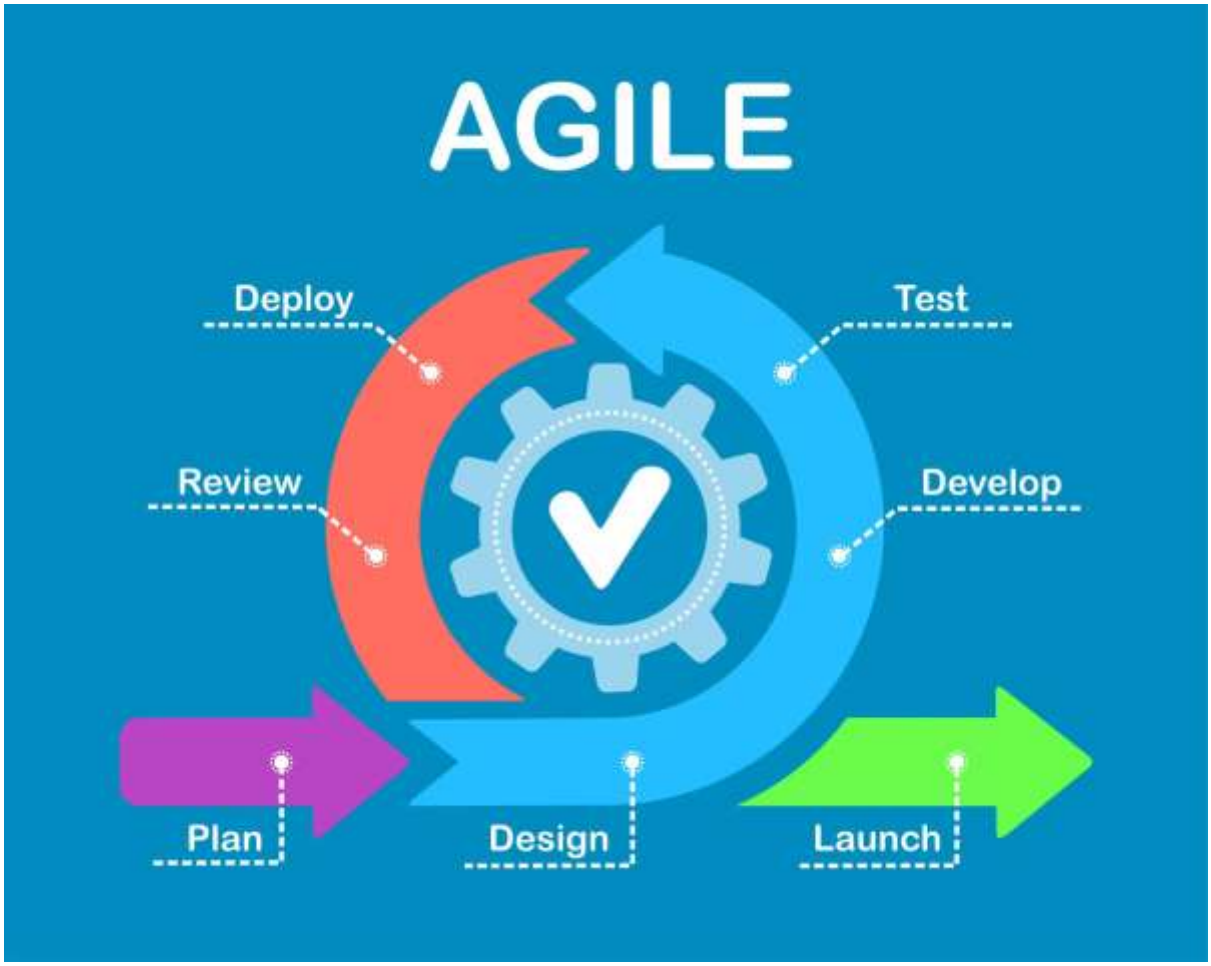
1. **Customer Communication** – Gather high-level requirements (user stories)
2. **Planning** – Select features for the current iteration
3. **Design & Development** – Build selected features
4. **Testing** – Test within the same iteration
5. **Deployment** – Deliver working software
6. **Feedback** – Collect customer input
7. **Next Iteration** – Improve and add new features

This loop repeats until the product is complete.

Key idea:
👉 *Working software is delivered frequently.*
👉 *Customer feedback drives every next step.*

🖼️ Diagram – Agile View of Process





4

(These diagrams reflect the standard GFG/Pressman-style Agile process view.)

✨ Characteristics of Agile View of Process

- Iterative and incremental development
- Continuous customer involvement
- Early and frequent delivery
- Welcomes changing requirements
- Lightweight documentation
- Close team collaboration
- Rapid feedback cycles

- Focus on working software
- Risk handled early
- Adaptive planning

🧪 Example

Food Delivery App

Iteration 1:

- Login + Registration

Iteration 2:

- Restaurant listing + Search

Iteration 3:

- Cart + Payment

After each iteration:

- App is tested
- Users give feedback
- Improvements are added in next sprint

This shows Agile's **build → review → improve** cycle.

👍 Advantages

- Faster delivery of usable software
- High customer satisfaction
- Easy to handle requirement changes
- Early defect detection
- Reduced project risk
- Better team communication

- Continuous improvement
- Suitable for complex projects

👎 Disadvantages

- Difficult cost/time estimation
- Requires experienced team
- Less documentation
- Not ideal for very small teams without discipline
- Customer availability is mandatory
- Scope creep risk

Taxonomy of Architectural Designs

📘 Introduction

The **Taxonomy of Architectural Designs** classifies software architectures into **distinct categories based on how components are organized and how data/control flows** within a system.

According to GeeksforGeeks, architectural taxonomy helps software engineers **select the most appropriate structure** for a project by understanding common architectural styles and their characteristics.

In simple words, it answers:

👉 *What are the major ways to organize a software system?*

⚙️ Working (How Architectural Taxonomy Is Organized)

Software architectural designs are commonly grouped into the following major categories:

🔹 1. Data-Centered Architecture

A central data store is accessed by multiple components.

- **2. Data-Flow Architecture**

Data moves through a series of processing steps (pipes & filters).

- **3. Call-and-Return Architecture**

Program structure follows hierarchical function calls.

- **4. Object-Oriented Architecture**

System is built around interacting objects/classes.

- **5. Layered Architecture**

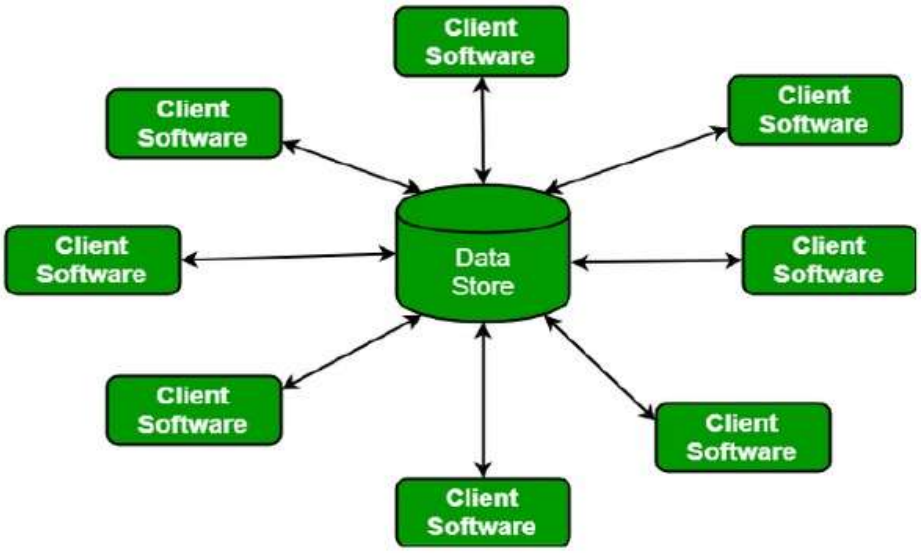
System is divided into layers, each with specific responsibility.

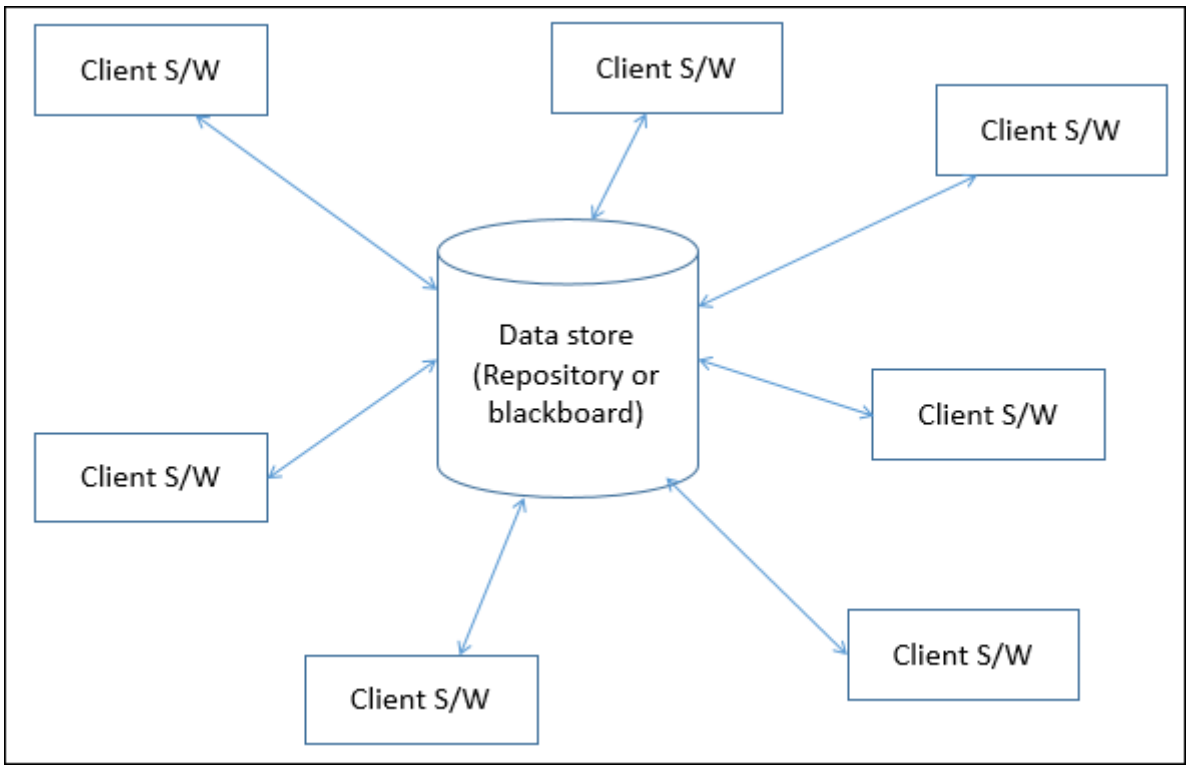
Each style addresses **different system needs** such as scalability, maintainability, or performance.

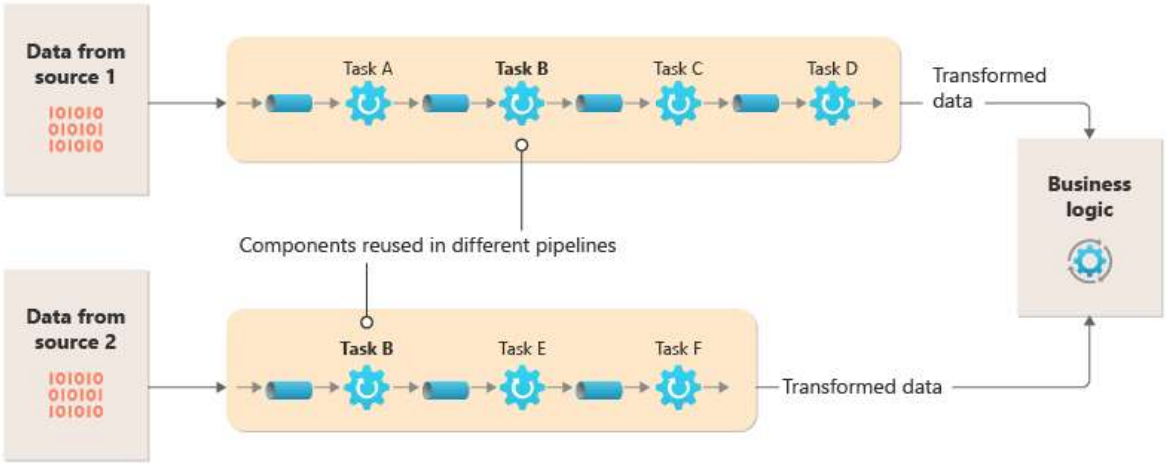
🖼️ **Diagram – Taxonomy of Architectural Designs (Common Styles)**

Taxonomy of Architectural styles:

1. **Data centred architectures:**
    - A data store will reside at the centre of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.
    - The figure illustrates a typical data centred style. The client software access a central repository. Variation of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.
    - This data-centred architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
    - Data can be passed among clients using blackboard mechanism.







5

🧩 Explanation of Each Architectural Style

🔹 1) Data-Centered Architecture

- Uses a **central repository/database**
- All components read/write from this shared data store

Example: Banking database system

🔹 2) Data-Flow Architecture

- Data passes through **processing units (filters)**
- Each unit transforms the data

Example: Compiler (lexical → syntax → semantic analysis)

🔹 3) Call-and-Return Architecture

- Main program calls subprograms in hierarchy
- Traditional procedural style

Example: Menu-driven applications

🔹 4) Object-Oriented Architecture

- System modeled as **objects communicating via methods**
- Supports encapsulation and inheritance

Example: Online Shopping System (User, Cart, Product, Payment objects)

🔹 5) Layered Architecture

- System divided into layers:
  - Presentation
  - Business Logic
  - Data Access

Each layer depends only on the layer below.

Example: Web applications (UI → Service → Database)

✨ Characteristics

- Provides structured system organization
- Improves modularity
- Encourages separation of concerns
- Enhances maintainability
- Supports scalability
- Promotes reuse
- Simplifies design decisions
- Technology independent

🧪 Example

E-Commerce Website

- Layered → UI, Backend, Database
- Object-Oriented → Product, Order, User classes
- Data-Centered → Central product database

Multiple architectural styles can coexist in one system.

👍 Advantages

- Clear system structure
- Easier maintenance
- Better scalability
- Supports parallel development
- Improves code reusability
- Simplifies debugging
- Helps architectural planning

- Reduces design complexity

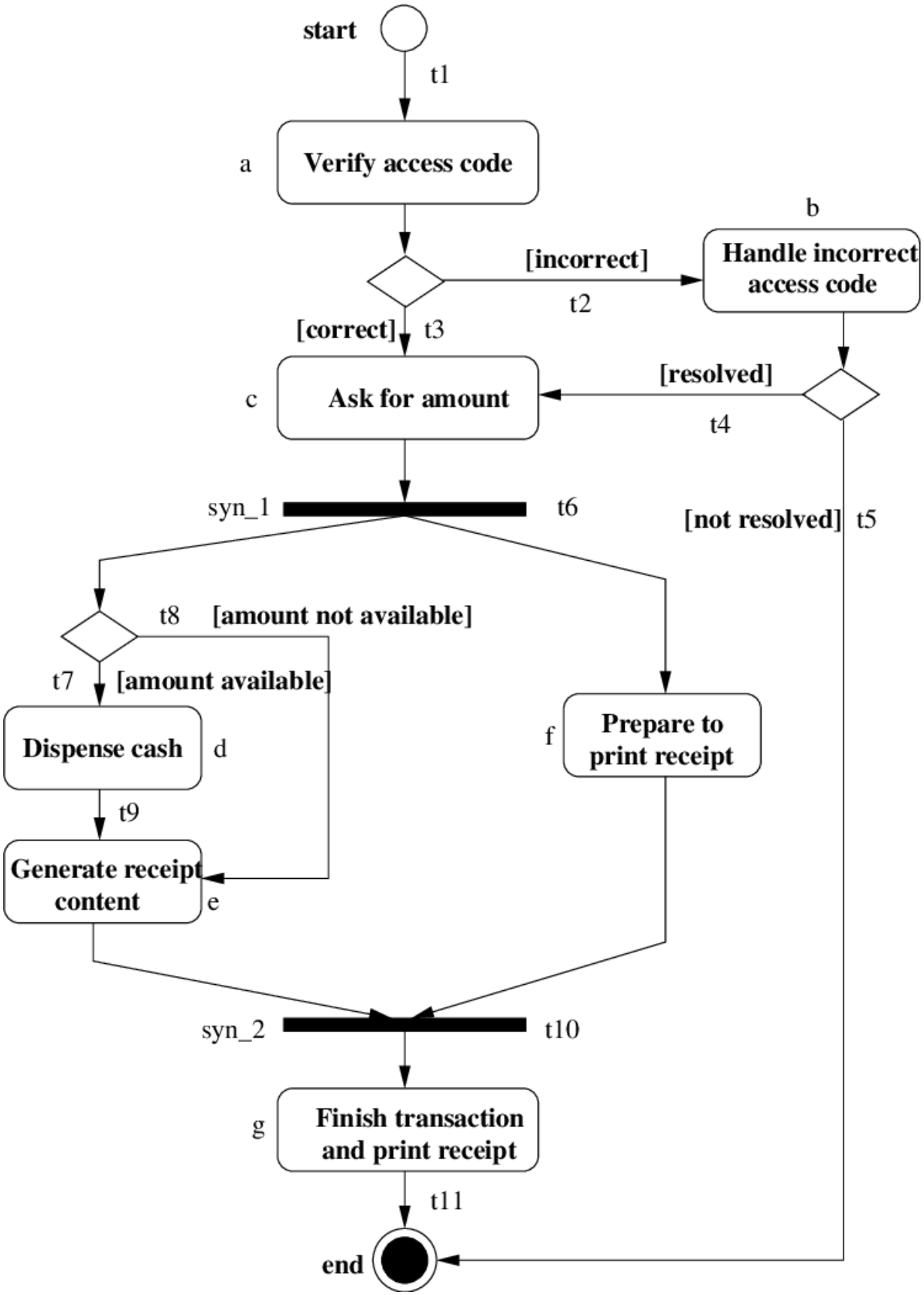
Working (ATM Activity Flow)

The ATM Activity Diagram generally follows this sequence:

1. Start
2. Insert ATM Card
3. Enter PIN
4. Validate PIN
5. Decision (PIN Valid?)
    - No → Show Error → Retry / Exit
    - Yes → Display Transaction Menu
6. Select Transaction (Withdraw / Balance Enquiry / Deposit)
7. Process Transaction
8. Dispense Cash / Show Balance / Accept Deposit
9. Print Receipt (optional)
10. Eject Card
11. End

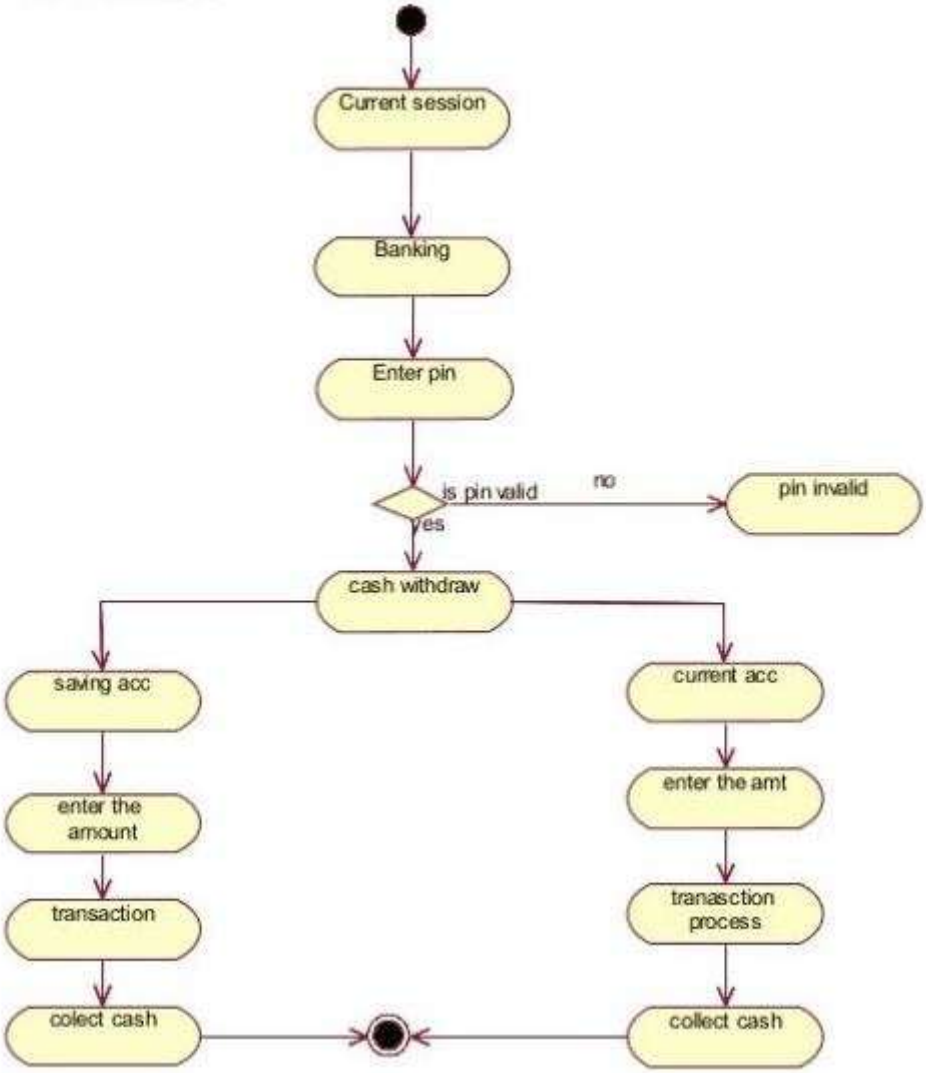
This flow includes **decision nodes** (valid/invalid PIN) and **activities** (withdraw, display balance, etc.).

🖼️ **Diagram – Activity Diagram for ATM System**


start
t1
a
Verify access code
b
Handle incorrect access code
[incorrect]
t2
[correct]
t3
c
Ask for amount
[resolved]
t4
[not resolved]
t5
syn_1
t6
t8
[amount not available]
t7
[amount available]
Dispense cash
d
f
Prepare to print receipt
t9
Generate receipt content
e
syn_2
t10
g
Finish transaction and print receipt
t11
end

Activity Diagram:

Cash Withdraw:



4

Scenario Testing

📘 Introduction

Scenario Testing is a software testing technique in which **real-life user scenarios (end-to-end workflows)** are tested to verify whether the complete system behaves correctly.

According to GeeksforGeeks, Scenario Testing focuses on **testing the application from a user's perspective**, covering multiple

functionalities together rather than testing individual features separately.

In simple words:

👉 *Scenario Testing checks complete user journeys instead of single functions.*

⚙️ Working (How Scenario Testing Is Performed)

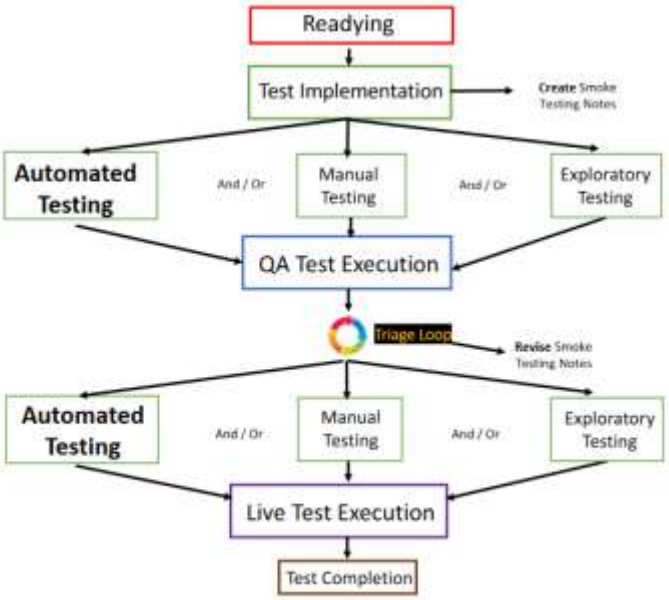
Scenario Testing works by simulating **practical user situations**.

Steps:

1. Understand application requirements
2. Identify real-world user scenarios
3. Convert scenarios into test cases
4. Execute end-to-end workflows
5. Observe system behavior
6. Log defects
7. Retest after fixes

Unlike unit or functional testing, Scenario Testing validates **multiple modules at once**.

🖼️ Diagram – Scenario Testing Flow



Readying
Requirements / BDD
Test Analysis / Static testing
Reviews / Analysis

↓

Test Implementation:
Test script creation for different 'test types',
in alignment with test design
Code creation and Reviews

↓

QA Test Execution:
Running all new E2E tests
Running all existing E2E tests
- against relevant repo
- on all browsers (where applicable)
Comparing results; actual V expected

↓

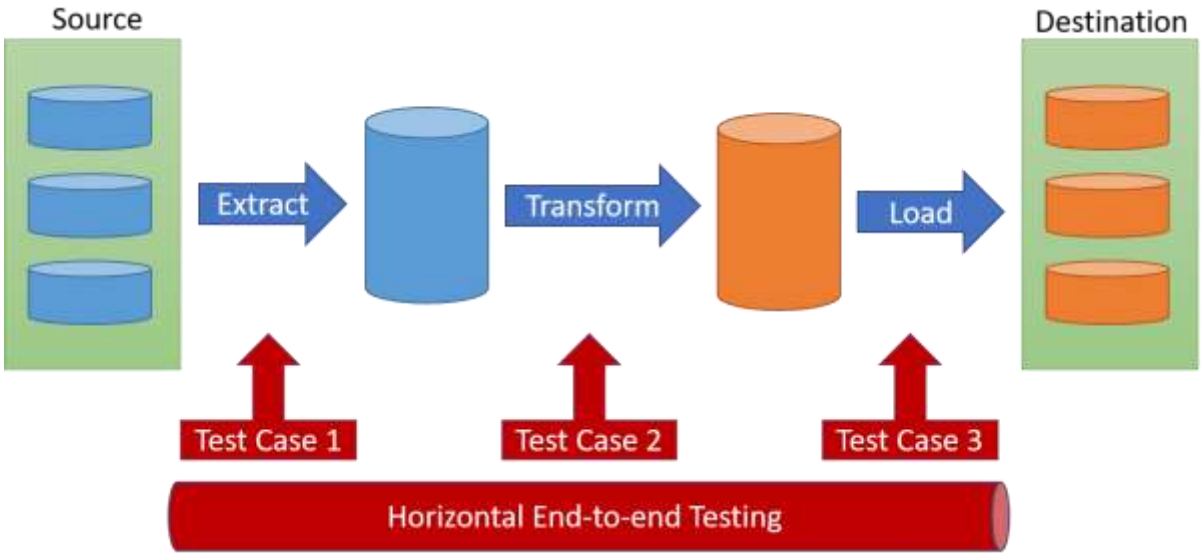
Triage Loop (fixes) - Reporting defects/changes
Updating Test basis (if changed)
Creating new tests / updating scripts
Updating test traceability
Repeating test activities – Implementation & Execution
Revising Smoke testing notes

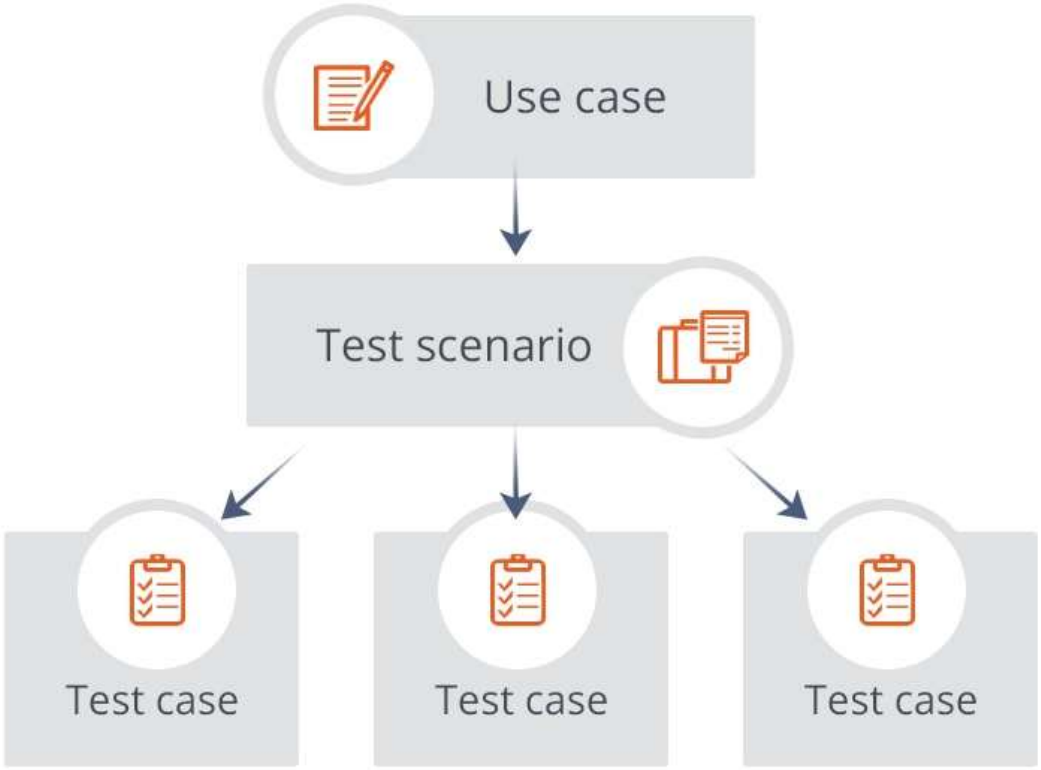
↓

Live Test Execution:
Running all E2E tests, all repos (where applicable)
on all browsers (where applicable)
Executing Smoke tests

↓

Test Completion





4

✨ Characteristics

- End-to-end testing approach
- User-centric
- Covers multiple modules together
- Based on real-life situations
- High-level testing
- Improves business flow validation
- Detects integration issues
- Complements functional testing

🧪 Example

E-Commerce Website Scenario:

1. User logs in
2. Searches product
3. Adds item to cart
4. Makes payment
5. Receives order confirmation

All these steps together form **one scenario**.

If payment succeeds but order confirmation fails, Scenario Testing identifies this workflow defect.

👍 **Advantages**

- Validates complete business flows
- Improves customer experience
- Finds integration defects
- Reduces production issues
- Easy to understand
- Realistic testing
- Enhances overall quality
- Saves rework cost

Difference Between Functional Testing and Non-Functional Testing (12 Points)

No.	Functional Testing	Non-Functional Testing
1	Verifies **what the system does**	Verifies **how the system performs**
2	Focuses on **functional requirements**	Focuses on **performance, usability, reliability, etc.**
3	Type of **functional validation**	Type of **quality validation**
4	Checks features like login, registration, payment	Checks speed, security, scalability, usability
5	Based on **SRS / business requirements**	Based on **non-functional specifications**
6	Ensures correctness of output for given input	Ensures system behavior under various conditions
7	Examples: Unit, Integration, System testing	Examples: Performance, Load, Stress testing
8	Usually executed **before non-functional testing**	Usually executed **after functional testing**
9	Can be done manually or automated	Mostly requires automation tools
10	Confirms business logic	Confirms user experience and system stability
11	Finds missing or incorrect functionality	Finds performance bottlenecks and quality issues
12	Answers: *Does it work?*	Answers: *How well does it work?*

Difference Between Top-Down Integration Testing and Bottom-Up Integration Testing (12 Points)

No.	Top-Down Integration Testing	Bottom-Up Integration Testing
1	Integration starts from **top-level modules**	Integration starts from **lowest-level modules**
2	Lower modules are added **gradually downward**	Higher modules are added **gradually upward**
3	Uses **stubs** to simulate lower modules	Uses **drivers** to simulate higher modules
4	Main control logic is tested **early**	Main control logic is tested **late**
5	High-level design errors detected **early**	Low-level design errors detected **early**
6	Development of stubs is required	Development of drivers is required
7	Difficult to observe detailed output at early stages	Detailed functionality can be tested early
8	Best suited when **top modules are critical**	Best suited when **bottom modules are critical**
9	System skeleton is available early	Complete system view appears late
10	Debugging is comparatively **more complex**	Debugging is comparatively **easier**
11	Lower-level utilities are tested later	Core utilities are tested first
12	Example focus: UI → Business Logic → Database	Example focus: Database → Business Logic → UI

ools Used for Testing Object-Oriented Software and Web Applications

📘 Introduction

Testing modern software systems—especially **Object-Oriented Software (OOS)** and **Web Applications**—requires specialized tools to handle **classes, objects, UI flows, APIs, performance, and security**.

According to GeeksforGeeks, testing tools help automate test execution, validate functionality, measure performance, and ensure reliability across complex systems.

In simple words:

👉 *Testing tools reduce manual effort and improve software quality.*

⚙️ Working (How Testing Tools Are Used)

Testing tools generally work by:

1. Executing automated test cases
2. Validating class behavior or web UI flows
3. Simulating user actions or API calls
4. Reporting defects and coverage
5. Supporting regression testing after code changes

They are selected based on:

- Type of application (OOP / Web)
- Testing level (unit / UI / performance / security)
- Project size and complexity

🔹 A) Tools Used for Testing Object-Oriented Software

These tools mainly support **class-level testing, unit testing, mocking, and regression testing**.

📜 Enlist of Common OOS Testing Tools

1. **JUnit** – Unit testing framework for Java classes
2. **TestNG** – Advanced testing framework with annotations and parallel execution
3. **Mockito** – Mocking framework to isolate classes and dependencies
4. **NUnit** – Unit testing framework for .NET applications
5. **CppUnit** – Unit testing for C++ object-oriented programs
6. **JMock** – Mock object library for Java
7. **PyTest / unittest** – OOP testing in Python
8. **JaCoCo** – Code coverage tool for Java

✨ Characteristics (OOS Testing Tools)

- Support class and method testing
- Enable mocking of objects
- Provide code coverage reports
- Help regression testing
- Integrate with CI/CD pipelines
- Improve object interaction validation

🧪 Example (OOS)

In an Online Shopping System:

- Test Cart.addItem() using JUnit
- Mock Payment class using Mockito

- Verify order logic using TestNG

This ensures **each class and object interaction works correctly**.

🔹 B) Tools Used for Testing Web Applications

Web application testing focuses on:

- UI automation
- API validation
- Performance
- Security

📜 Enlist of Common Web Application Testing Tools

🔸 Functional / UI Testing

1. **Selenium**
2. **Cypress**
3. **Playwright**
4. **TestComplete**

🔸 API Testing

5. **Postman**
6. **SoapUI**

🔸 Performance Testing

7. **Apache JMeter**
8. **LoadRunner**

🔸 Security Testing

9. **OWASP ZAP**
10. **Burp Suite**

✨ Characteristics (Web Testing Tools)

- Automate browser actions
- Validate APIs
- Measure performance under load
- Detect security vulnerabilities
- Support cross-browser testing
- Generate detailed reports

🧪 Example (Web Application)

E-commerce Website:

- Selenium → Login & checkout flow
- Postman → Payment API testing
- JMeter → Load testing with 1000 users
- OWASP ZAP → Security scan

This combination ensures **functional + performance + security quality**.

Regression Testing (with Types)

📘 Introduction

Regression Testing is a type of software testing performed to ensure that **recent code changes (bug fixes, enhancements, or new features) have not negatively affected existing functionality**.

According to GeeksforGeeks, Regression Testing re-executes previously passed test cases to confirm that the application still behaves correctly after modifications.

In simple words:

👉 *Regression Testing checks that "old features still work after new changes."*

⚙️ Working (How Regression Testing Is Performed)

Regression Testing generally follows these steps:

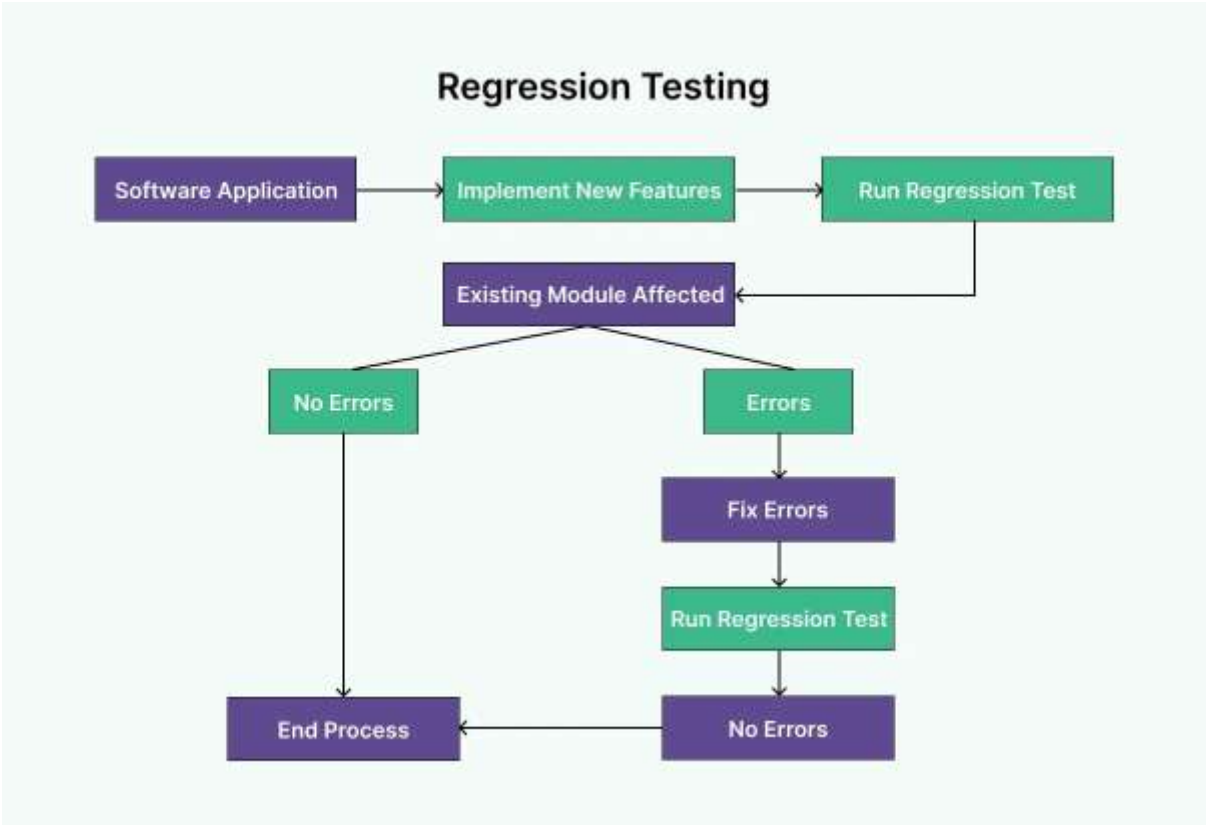
1. Make changes in code (bug fix / feature update)
2. Identify impacted areas
3. Select regression test cases
4. Execute test cases (manual or automated)
5. Compare actual vs expected results
6. Log defects (if any)
7. Fix issues and retest

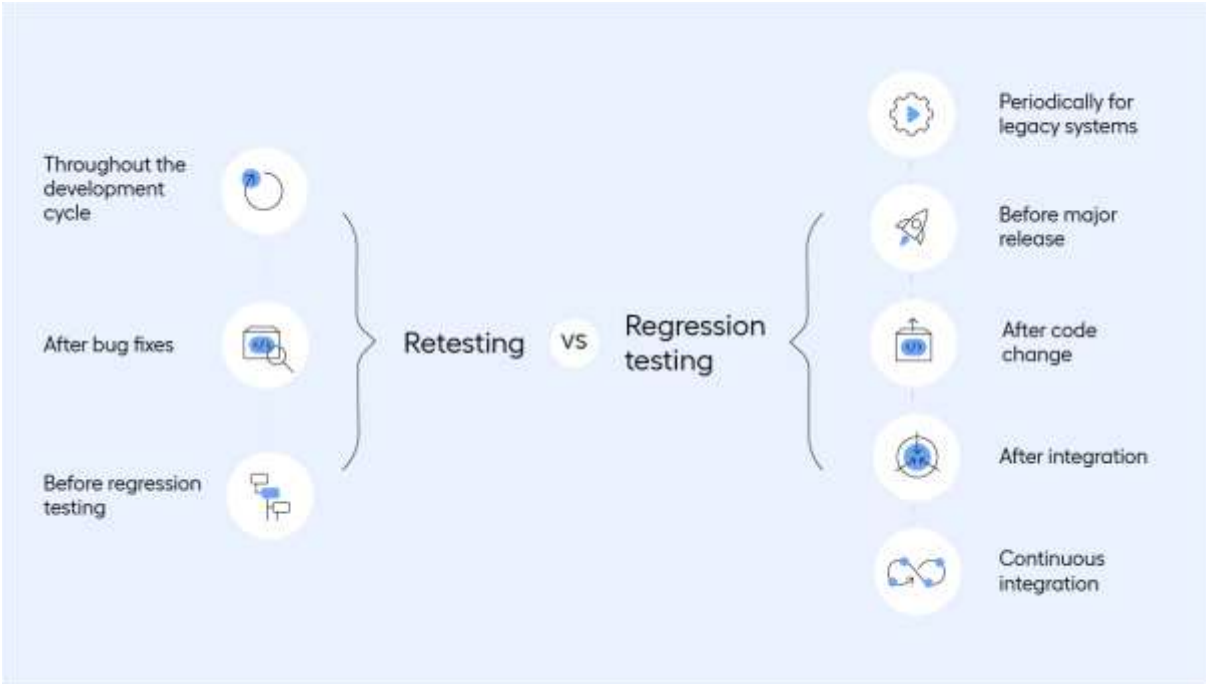
It is commonly executed:

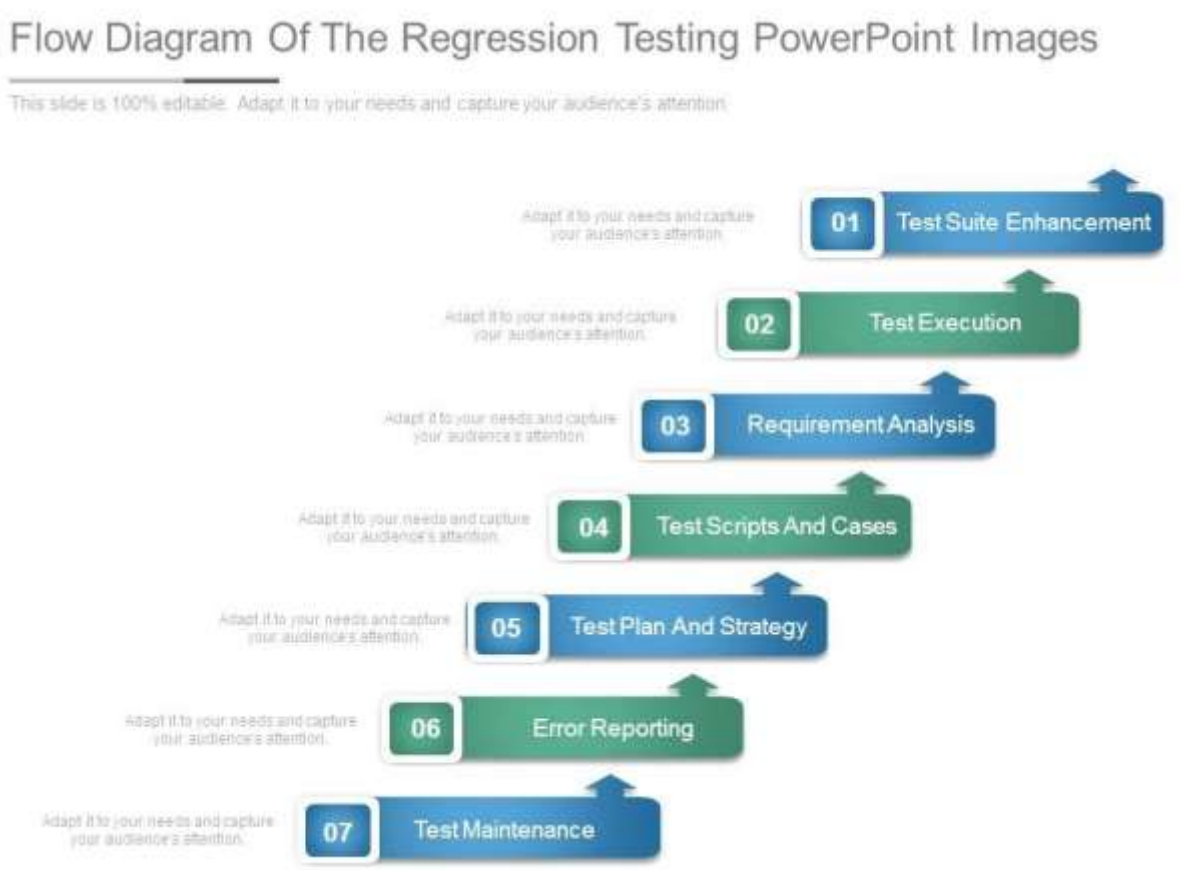
- After bug fixes
- After new feature addition
- After code refactoring
- Before every release

Automation tools are often used because regression tests are **repetitive**.

🖼️ Diagram – Regression Testing Workflow


Regression Testing
Software Application
Implement New Features
Run Regression Test
Existing Module Affected
No Errors
Errors
Fix Errors
Run Regression Test
No Errors
End Process


Throughout the development cycle
After bug fixes
Before regression testing
Retesting
vs
Regression testing
Periodically for legacy systems
Before major release
After code change
After integration
Continuous integration



4

🧩 Types of Regression Testing

🔹 1) Unit Regression Testing

- Tests individual modules after changes
- Focuses on **small code units**

Example: Retesting login function after password validation change.

🔹 2) Partial Regression Testing

- Tests only the **affected modules + related areas**

Example: After cart update, test cart + payment modules.

🔹 3) Complete Regression Testing

- Tests the **entire application**

Used when:

- Major changes are done
- Multiple modules are impacted

🔹 4) Selective Regression Testing

- Executes only **selected test cases** based on impact analysis
- Saves time and effort

🔹 5) Progressive Regression Testing

- Used when **new test cases are added** for new features
- Ensures new + old functionality works together

✨ Characteristics

- Performed after code changes
- Ensures software stability
- Reuses existing test cases
- Can be manual or automated
- Time-consuming if done fully
- Critical before production release
- Prevents reintroduction of old bugs
- Supports continuous integration

🧪 Example

E-commerce Website

Bug fixed in *payment module*.

Regression Testing checks:

- Login
- Product search
- Add to cart
- Checkout
- Order confirmation

If checkout fails after payment fix → regression defect detected.

Generic View of Software Engineering

📘 **Introduction**

The **Generic View of Software Engineering** presents a **common framework** that applies to *all* software process models. It defines a set of **core activities** supported by **umbrella activities** that guide software development from concept to delivery and maintenance.

According to GeeksforGeeks, this generic view provides a **standard roadmap** for building software, independent of whether you use Waterfall, Agile, Spiral, or any other model.

In simple words:

👉 *It shows the basic activities every software project must follow.*

⚙️ **Working (How the Generic View Operates)**

The Generic View is built around **five core framework activities**, continuously supported by **umbrella activities**.

- **Core Framework Activities**

1. **Communication**
    - Interact with stakeholders
    - Gather and clarify requirements
2. **Planning**
    - Estimate cost, effort, schedule
    - Identify risks and resources
3. **Modeling**
    - Design system architecture
    - Prepare data and interface models
4. **Construction**
    - Coding + Testing
    - Convert design into working software
5. **Deployment**
    - Deliver software to users
    - Collect feedback and support maintenance

- **Umbrella Activities (run across all phases)**

- Project tracking & control
- Risk management
- Software Quality Assurance (SQA)
- Technical reviews
- Configuration management
- Measurement
- Documentation

These ensure **quality, control, and continuous improvement** throughout the project.

🖼️ Diagram – Generic View of Software Engineering

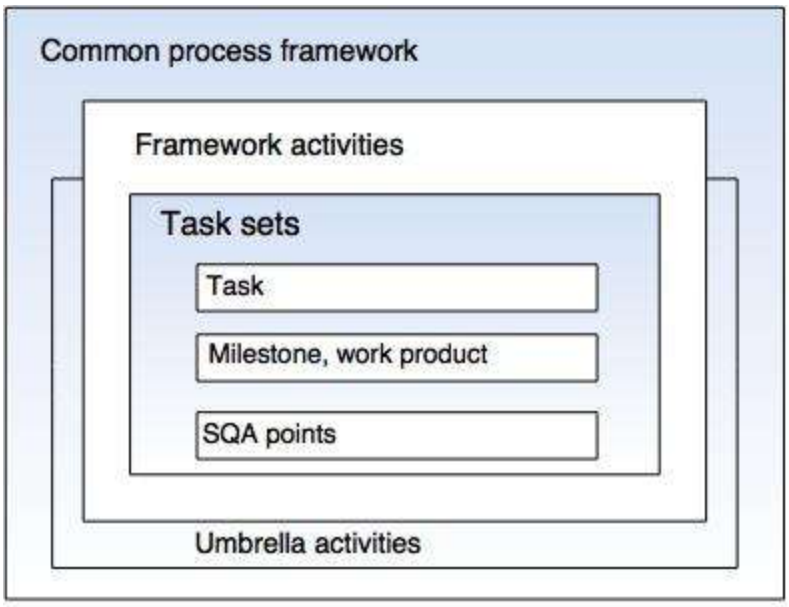


Fig.- A software process framework

4

(These represent the standard Pressman/GFG-style Generic Software Engineering Framework.)

✨ Characteristics

- Applicable to all process models
- Iterative and incremental
- Customer-focused
- Quality-driven

- Risk-aware
- Supports documentation
- Encourages feedback
- Scalable for large systems
- Promotes team collaboration

🧪 Example

Online Examination System

- Communication → Gather exam and login requirements
- Planning → Decide schedule and resources
- Modeling → Design database and UI
- Construction → Code + test modules
- Deployment → Release system to students

Umbrella activities continuously manage bugs, versions, and quality.

Requirements Engineering Activities

📘 Introduction

Requirements Engineering (RE) is a systematic process used to **identify, analyze, document, validate, and manage software requirements** throughout the project lifecycle.

According to GeeksforGeeks, Requirements Engineering ensures that the final software product **accurately reflects customer needs** and reduces the risk of costly rework caused by misunderstood or missing requirements.

In simple words:

👉 *Requirements Engineering answers: What should the system do, and why?*

⚙️ Working (How Requirements Engineering Is Performed)

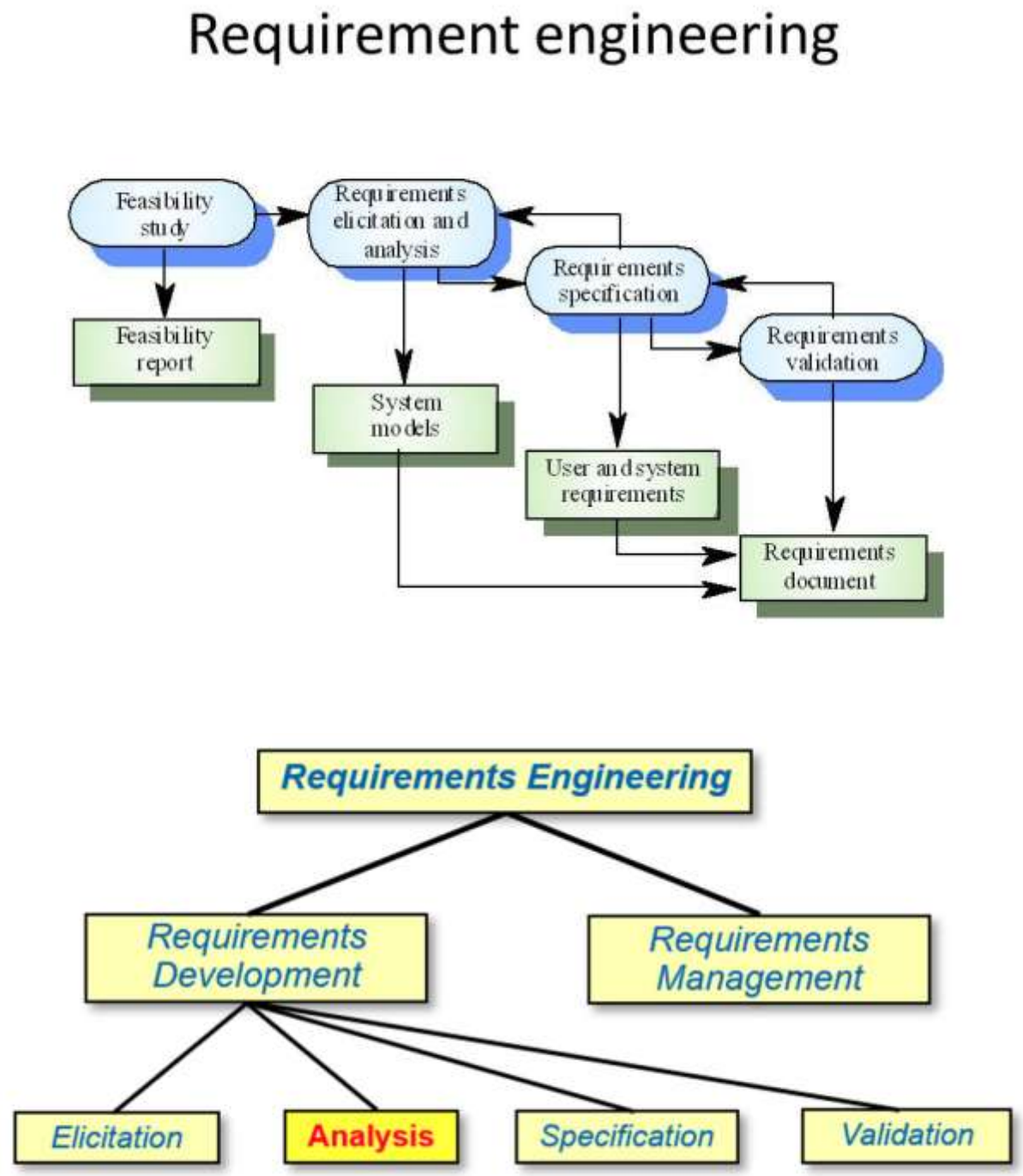
Requirements Engineering is carried out through a set of **well-defined activities** that are iterative in nature.

🔹 Main Requirements Engineering Activities

1. **Feasibility Study**
    - Checks technical, economic, and operational feasibility
    - Decides whether the project is viable
2. **Requirements Elicitation**
    - Collects requirements from stakeholders (interviews, surveys, workshops)
3. **Requirements Analysis**
    - Resolves conflicts
    - Prioritizes requirements
    - Models system behavior
4. **Requirements Specification**
    - Documents requirements formally in SRS (Software Requirement Specification)
5. **Requirements Validation**
    - Ensures requirements are correct, complete, and consistent
6. **Requirements Management**
    - Handles changes
    - Maintains traceability throughout development

These activities repeat whenever requirements evolve.

🖼️ Diagram – Requirements Engineering Process



4

(This represents the standard GFG-style Requirements Engineering workflow.)

✨ Characteristics

- User-centric approach
- Iterative process
- Supports change management
- Emphasizes documentation
- Improves requirement clarity
- Reduces project risk
- Enhances communication
- Ensures traceability
- Quality-driven

🧪 Example

Online Admission System

- Feasibility → Check infrastructure availability
- Elicitation → Gather student & admin needs
- Analysis → Prioritize login, form submission, fee payment
- Specification → Prepare SRS document
- Validation → Review with stakeholders
- Management → Update requirements when new features are added

Software Design Attributes (Concepts Explained in Detail)

📘 Introduction

Software Design Attributes are quality characteristics that determine **how well a software system is designed**. They describe properties such as maintainability, reliability, efficiency, and usability.

According to GeeksforGeeks, software design attributes help engineers evaluate whether a design is **robust, flexible, reusable, and easy to evolve**. These attributes guide architects and developers in creating **high-quality software systems**.

In simple words:

👉 *Design attributes define how good a software design really is.*

⚙️ Working (How Design Attributes Are Applied)

During the design phase:

1. Requirements are analyzed
2. Architecture is proposed

3. Design attributes are evaluated (modularity, cohesion, coupling, etc.)
4. Weak areas are improved
5. Final design is optimized for quality

These attributes influence:

- Architecture decisions
- Module decomposition
- Interface design
- Data structures
- Error handling

They are not implemented directly in code — instead, they **shape how the system is structured**.

🖼️ **Diagram – Software Design Attributes Overview**

4

🧩 Major Software Design Attributes

Below are the most important design attributes explained in detail:

🔹 1) Maintainability

Ability of software to be **easily modified, corrected, or enhanced**.

Key points

- Simple structure
- Proper documentation
- Modular design

Example: Updating tax rules in a billing system without affecting other modules.

🔹 2) Reliability

Ability of software to **perform correctly under stated conditions for a specified time**.

Key points

- Fewer failures
- Proper exception handling
- Consistent output

Example: Banking system processing transactions without crashes.

🔹 3) Efficiency (Performance)

How well the software uses **CPU, memory, and time**.

Key points

- Optimized algorithms
- Minimal resource usage

Example: Search feature returning results quickly even with large data.

🔹 4) Usability

Ease with which users can **learn and operate the system**.

Key points

- Simple UI
- Clear navigation
- Helpful error messages

Example: ATM screens guiding users step-by-step.

🔹 5) Modularity

System is divided into **independent modules**, each performing a specific task.

Key points

- Easier debugging
- Parallel development
- Better organization

🔹 6) Reusability

Ability to reuse components in **other projects or modules**.

Key points

- Generic design
- Well-defined interfaces

Example: Login module reused across multiple applications.

🔹 7) Flexibility

Ease with which software can **adapt to new requirements**.

Key points

- Low coupling
- Configurable components

🔹 8) Scalability

Ability of software to **handle growth in users or data**.

Example: E-commerce site supporting increasing traffic.

🔹 9) Cohesion

Degree to which elements inside a module are **closely related**.

👉 High cohesion = good design.

🔹 10) Coupling

Degree of dependency between modules.

👉 Low coupling = good design.

✨ Characteristics of Good Software Design (via Attributes)

- High cohesion
- Low coupling
- Easy maintenance
- Reliable execution
- Efficient performance
- User-friendly
- Reusable components
- Scalable architecture
- Flexible structure
- Clear documentation

🧪 Example

Online Shopping System

- Modularity → Separate Cart, Payment, Product modules
- Reusability → Authentication reused in admin panel
- Reliability → Payment confirmation handled safely
- Usability → Simple checkout flow
- Scalability → Supports more users during sales

All these come from applying **design attributes correctly**

a) Quality

📘 **Definition**

Quality refers to the degree to which a software product **meets specified requirements and satisfies customer expectations**.

In software engineering, quality means the product is:

- Correct
- Reliable
- Efficient
- Usable
- Maintainable

👉 In simple words: *Quality = building the right product in the right way.*

✅ **b) Quality Control (QC)**

📘 **Definition**

Quality Control is a **product-oriented process** that focuses on **identifying defects in the developed software** through testing and inspection.

It answers:

👉 *Is the product correct?*

🔹 **Key Points**

- Reactive activity
- Finds defects after development
- Uses testing and reviews
- Focuses on **product**

🧪 **Example**

Executing test cases to detect bugs in a login module.

✅ **c) Quality Assurance (QA)**

📘 **Definition**

Quality Assurance is a **process-oriented approach** that focuses on **preventing defects** by improving development processes and standards.

It answers:

👉 *Are we following the correct process?*

🔹 **Key Points**

- Proactive activity
- Prevents defects
- Defines standards and procedures
- Focuses on **process**

🧪 **Example**

Defining coding standards and review procedures before development starts.

✅ d) Difference Between Verification and Validation (12 Points)

No.	Verification	Validation
1	Checks **documents, design, and code**	Checks **final product behavior**
2	Ensures product is built **correctly**	Ensures right product is built
3	Static process	Dynamic process
4	Done **without executing code**	Done **by executing software**
5	Performed during development	Performed after development
6	Includes reviews, inspections	Includes testing activities
7	Process-oriented	Product-oriented
8	Finds defects early	Finds defects late
9	Carried out by developers/QA	Carried out by testers/users
10	Examples: Code review, design review	Examples: System testing, UAT
11	Prevents defects	Detects defects
12	Question: *Are we building it right?*	Question: *Are we building the right thing?*

A) Integration Testing – Definition & Any Three Approaches

📘 Introduction

Integration Testing is a software testing level where **individual modules are combined and tested as a group** to verify correct interaction between components.

According to GeeksforGeeks, Integration Testing mainly focuses on **interface defects, data flow issues, and communication errors** that occur when modules work together.

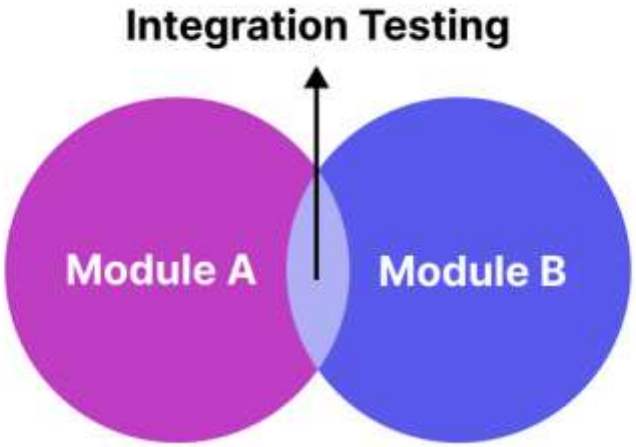
In simple words:

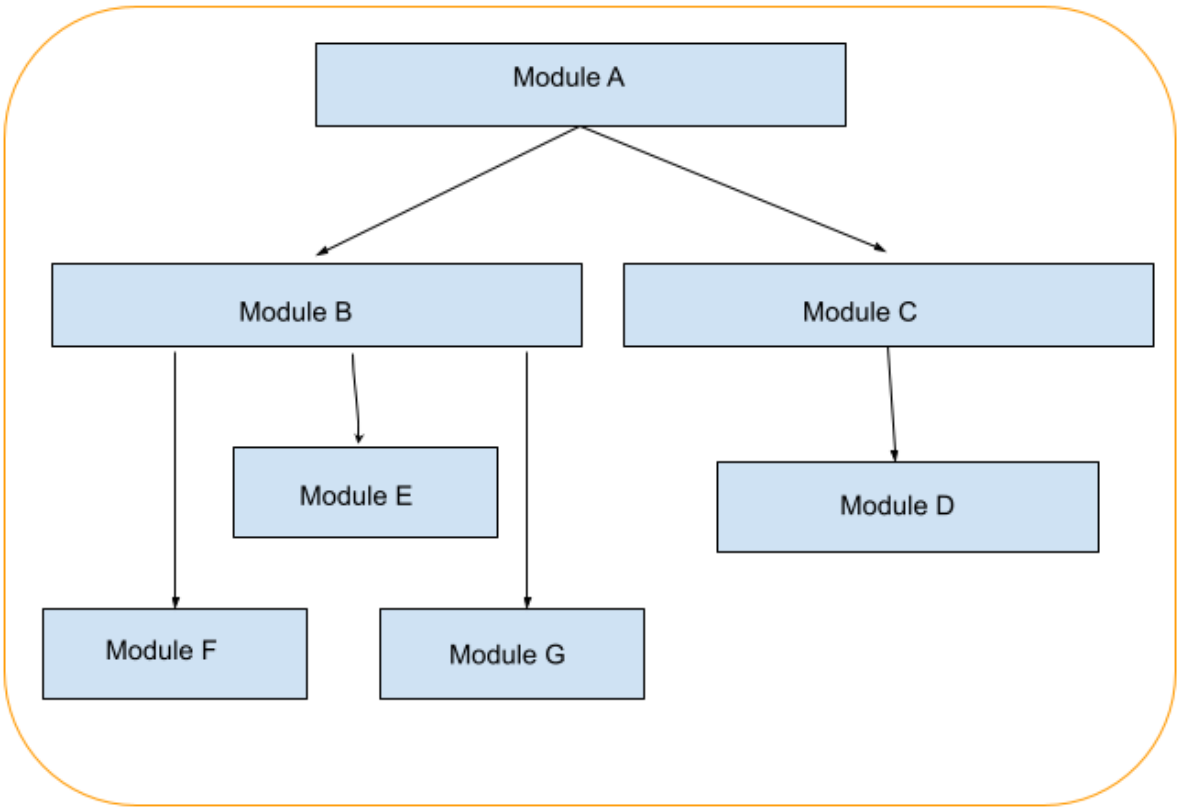
👉 *Integration Testing checks whether different modules work properly together.*

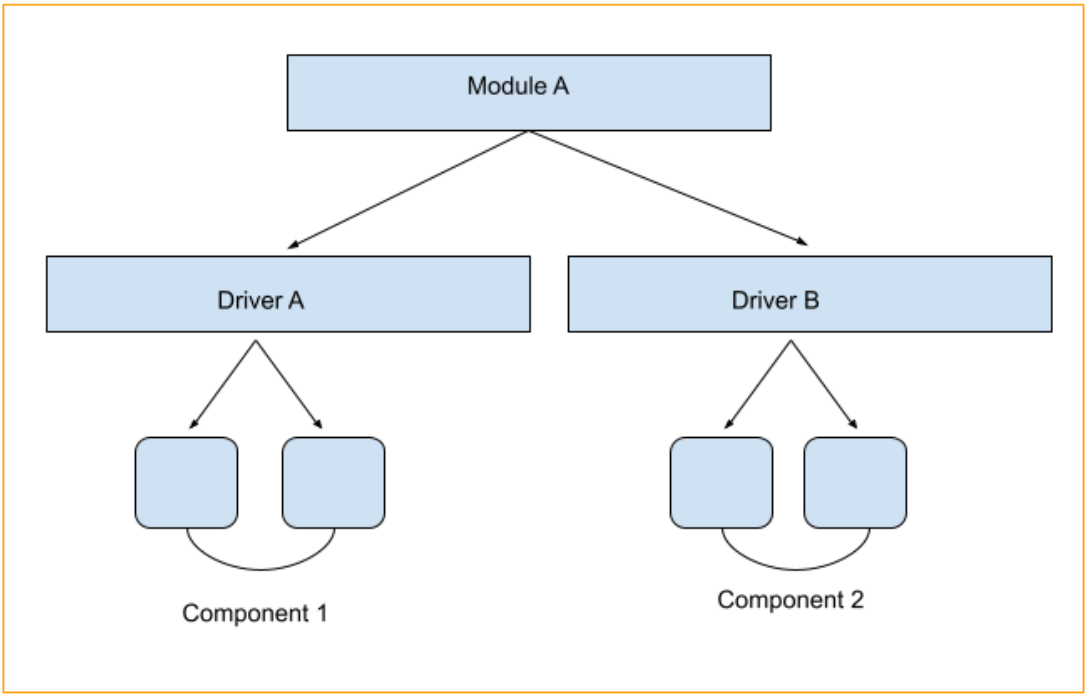
⚙️ Working

- Performed **after Unit Testing** and before System Testing
- Integrated modules are tested using predefined test cases
- Interface defects are detected and fixed
- Retesting is done after corrections

🖼️ Diagram – Integration Testing Approaches







4

🧩 Any Three Approaches of Integration Testing

🔹 1) Top-Down Integration Testing

- Starts from **top-level modules**
- Lower modules added gradually
- Uses **stubs**

Advantage: High-level logic tested early
Disadvantage: Low-level utilities tested late

🔹 2) Bottom-Up Integration Testing

- Starts from **lowest-level modules**
- Higher modules added later
- Uses **drivers**

Advantage: Core utilities tested early
Disadvantage: Main control logic tested late

🔹 3) Sandwich (Hybrid) Integration Testing

- Combination of Top-Down and Bottom-Up
- Middle layer used as starting point

Advantage: Balanced testing
Disadvantage: Complex to manage

👍 Advantages

- Detects interface defects early
- Improves system reliability
- Ensures smooth data flow

✅ B) System Testing + Load, Stress & Performance Testing

📘 System Testing – Definition

System Testing is a level of testing where the **entire integrated system is tested as a whole** to verify that it meets specified requirements.

As per GeeksforGeeks, System Testing validates both **functional and non-functional behavior** of complete software.

👉 *It checks whether the full system works correctly.*

🧩 Types of System Testing

🔹 a) Load Testing

Tests system behavior under **expected user load**.

Purpose:

- Measure response time
- Identify performance bottlenecks

Example:
Testing website with 500 simultaneous users.

🔹 b) Stress Testing

Tests system beyond **normal capacity** to find breaking point.

Purpose:

- Determine maximum load
- Check recovery capability

Example:
Suddenly increasing users from 500 to 5000.

🔹 c) Performance Testing

Evaluates **speed, scalability, and stability**.

Purpose:

- Measure response time
- Monitor CPU/memory usage

Example:
Checking how fast search results appear.

✨ Characteristics

- Performed after Integration Testing
- Covers end-to-end workflows
- Includes functional + non-functional testing

✅ C) Differences

a) Test Case vs Test Scenario (12 Points)

No	Test Case	Test Scenario
1	Detailed step-by-step instructions	High-level functionality
2	Specific input & output	General user flow
3	Low-level	High-level
4	Used for execution	Used for planning
5	Contains test data	No test data
6	Written after scenario	Written first
7	More in number	Fewer

No	Test Case	Test Scenario
8	Technical	Business oriented
9	Used by testers	Used by stakeholders
10	Easy automation	Hard to automate
11	Focuses on validation	Focuses on coverage
12	Example: login steps	Example: user login flow

---

b) Functional vs Non-Functional Testing (12 Points)

No	Functional Testing	Non-Functional Testing
1	Tests *what* system does	Tests *how* system performs
2	Based on requirements	Based on quality attributes
3	Validates features	Validates performance/usability
4	Login, payment testing	Load, security testing
5	Business logic focused	System behavior focused
6	Manual/automation	Mostly automation
7	Executed first	Executed later
8	Finds functional bugs	Finds performance issues
9	User requirements	System standards
10	Product correctness	Product quality
11	Examples: unit/system	Examples: load/stress
12	“Does it work?”	“How well does it work?”

✅ **Conclusion**

- **Integration Testing** ensures modules work together
- **System Testing** validates full application
- **Load, Stress, Performance** ensure stability under varying conditions
- Proper differentiation between **test cases/scenarios** and **functional/non-functional testing** improves overall test strategy.

Waterfall Model (with Advantages & Limitations)

📘 **Introduction**

The **Waterfall Model** is one of the **earliest and simplest prescriptive process models** used in Software Engineering. It follows a **linear and sequential approach**, where each phase must be completed before the next phase begins.

According to GeeksforGeeks, the Waterfall Model is called *"waterfall"* because development flows **downward step by step**, just like water falling from one level to another.

In simple words:

👉 *Work moves phase by phase, with no overlap.*

⚙️ **Working (Phase-by-Phase Explanation)**

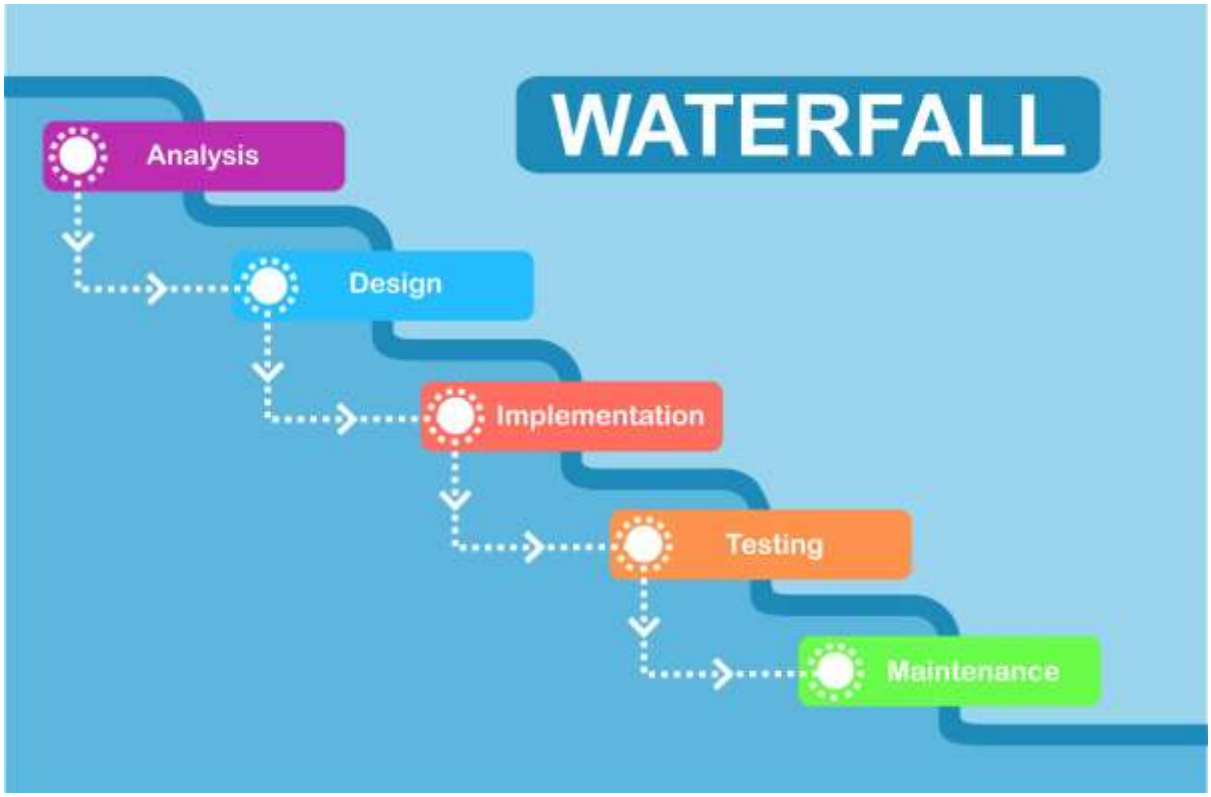
The Waterfall Model proceeds through these fixed stages:

1. **Requirement Analysis**
    - Collect and document user requirements
2. **System Design**
    - Prepare architecture, database design, and interfaces

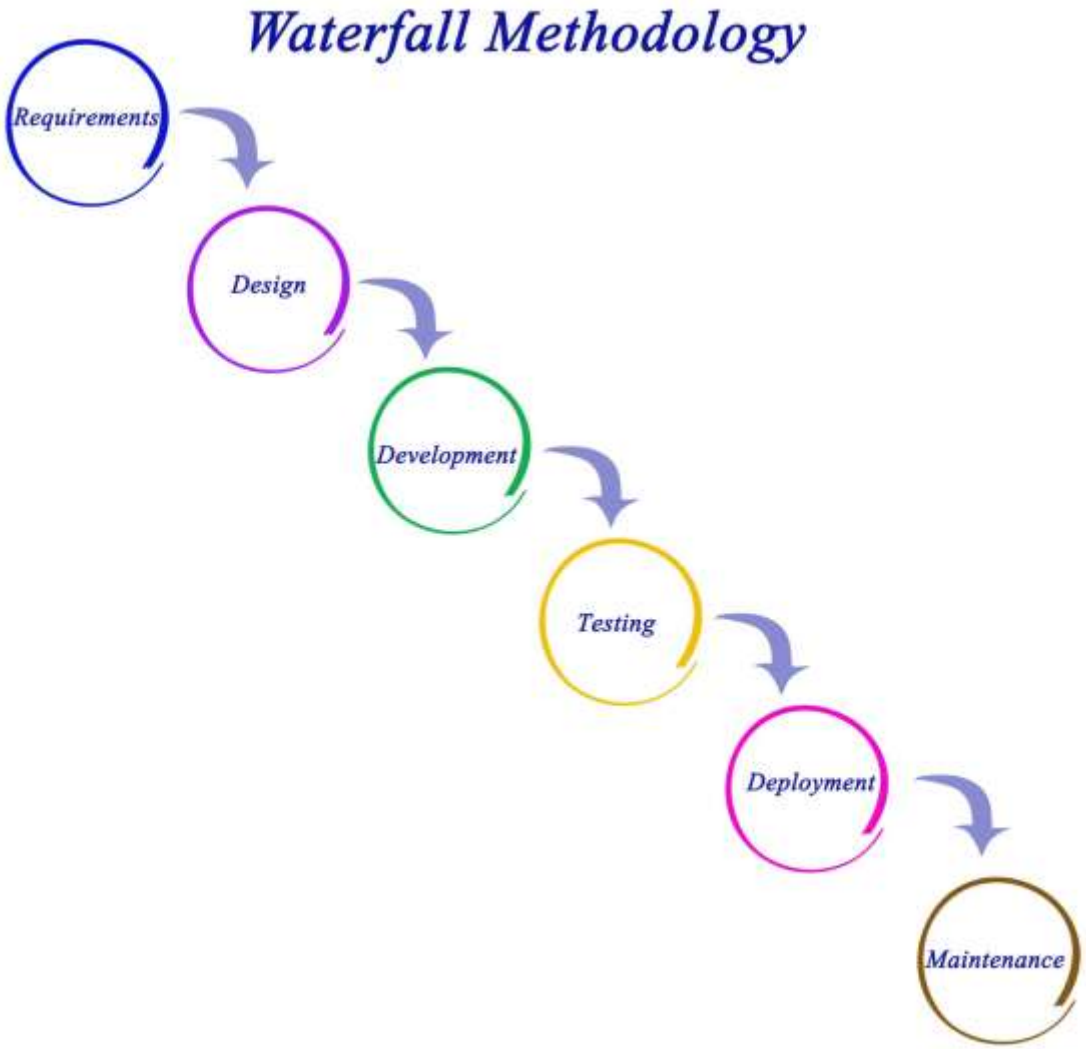
3. **Implementation (Coding)**
    - Convert design into source code
4. **Testing**
    - Verify and validate the developed software
5. **Deployment**
    - Deliver software to users
6. **Maintenance**
    - Fix bugs and add enhancements after release

Each phase starts **only after** the previous phase is fully completed.

🖼️ Diagram – Waterfall Model




Requirements
Design
Develop
Test
Deploy
WATERFALL



4

✨ Characteristics

- Linear and sequential flow
- One phase at a time
- Heavy documentation
- Clear milestones
- Simple to understand
- Rigid structure

- Customer involvement mainly at requirement phase
- No working software until late stages

🧪 Example

College Management System

- Requirements → Student records, fees, attendance
- Design → Database + UI
- Coding → Implement modules
- Testing → Verify login, reports
- Deployment → Install system in college
- Maintenance → Add online exam feature

Since requirements are fixed, Waterfall works well here.

👍 Advantages

- Simple and easy to understand
- Well-structured phases
- Clear documentation
- Easy project management
- Defined deliverables at each stage
- Suitable for small projects
- Works well when requirements are stable
- Easy to track progress

Characteristics of Software (Brief)

📘 Introduction

Software has unique properties that make it **different from hardware**. Understanding these characteristics helps engineers design better systems.

✨ Key Characteristics of Software

1. Software is Developed, Not Manufactured

- Cost is mainly in development, not production.

2. Software Does Not Wear Out

- It doesn't degrade physically; failures occur due to bugs or changes.

3. Software is Mostly Custom-Built

- Many applications are tailored to specific user needs.

4. Highly Complex

- Even small programs can have many logical paths.

5. Easy to Modify (but risky)

- Changes can introduce new defects.

6. Intangible Product

- Cannot be physically touched or seen.

7. Requires Continuous Maintenance

- Needs updates, fixes, and enhancements.

8. Evolves Over Time

- New features are added as user needs change.

Quality Assurance (QA) and Quality Control (QC)

📘 Introduction

Quality Assurance (QA) and **Quality Control (QC)** are essential activities to ensure **high-quality software products**.

As explained by GeeksforGeeks:

- **QA focuses on preventing defects (process-oriented)**
- **QC focuses on detecting defects (product-oriented)**

🧩 Quality Assurance (QA)

📌 Definition

Quality Assurance is a **proactive process** that ensures the **right development procedures and standards** are followed to prevent defects.

✨ Key Points

- Process-oriented
- Prevents defects
- Defines standards and methodologies
- Includes audits and process reviews

💉 Example

Setting coding standards and review guidelines before development.

🧩 Quality Control (QC)

📌 Definition

Quality Control is a **reactive activity** that identifies defects in the **actual software product** through testing and inspection.

✨ Key Points

- Product-oriented
- Detects defects
- Uses testing and inspections

- Ensures output correctness

🧪 Example

Executing test cases to find bugs in login module.

✅ Summary (Short Note)

- **QA → Process focused → Prevents defects**
- **QC → Product focused → Finds defects**

Both together ensure **reliable and high-quality software delivery**.

System Integration

📘 Introduction

System Integration is the process of **combining different subsystems or modules into a single, complete system** and ensuring that they work together correctly.

According to GeeksforGeeks, system integration focuses on verifying **data flow, interfaces, and interactions** among components after individual modules have been developed and unit tested.

In simple words:

👉 *System Integration connects separate parts of software into one working system.*

⚙️ Working (How System Integration Is Done)

System Integration typically follows these steps:

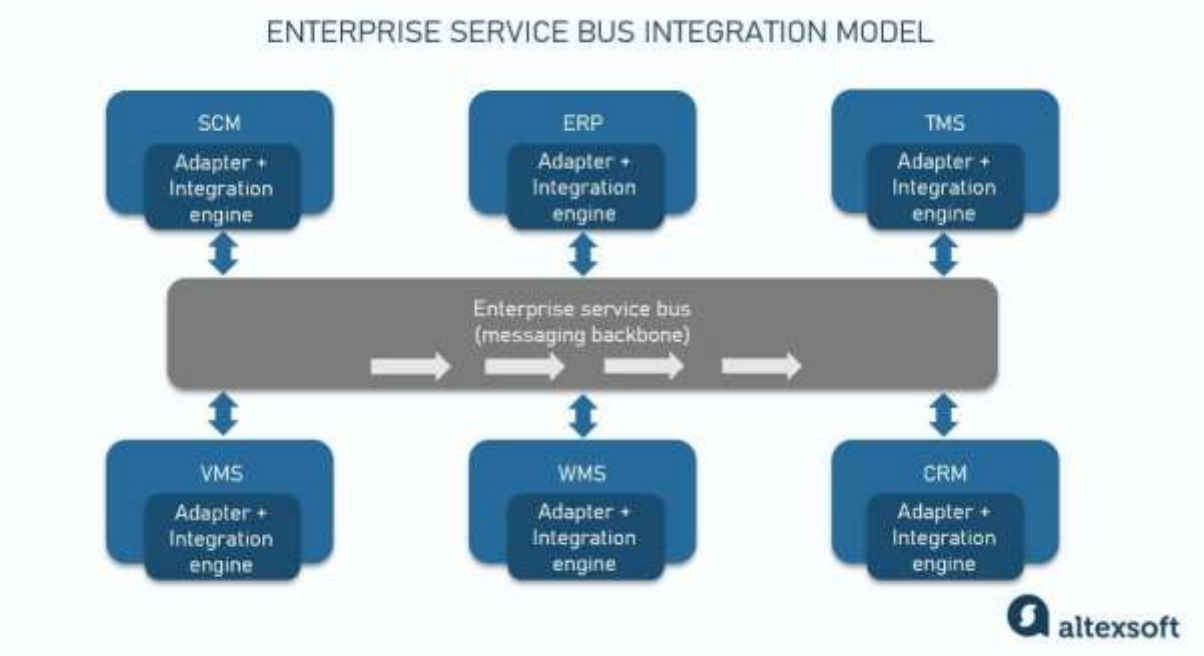
1. Identify all independent modules/subsystems
2. Define interfaces between components

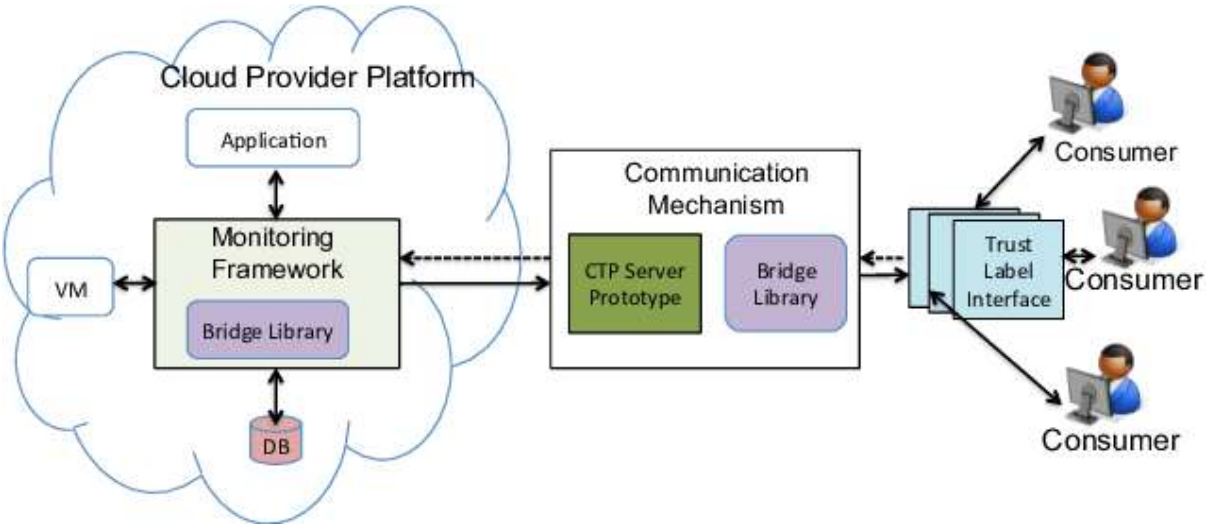
3. Integrate modules incrementally or all at once
4. Verify communication and data exchange
5. Detect interface defects
6. Fix issues and re-integrate
7. Prepare the system for **System Testing**

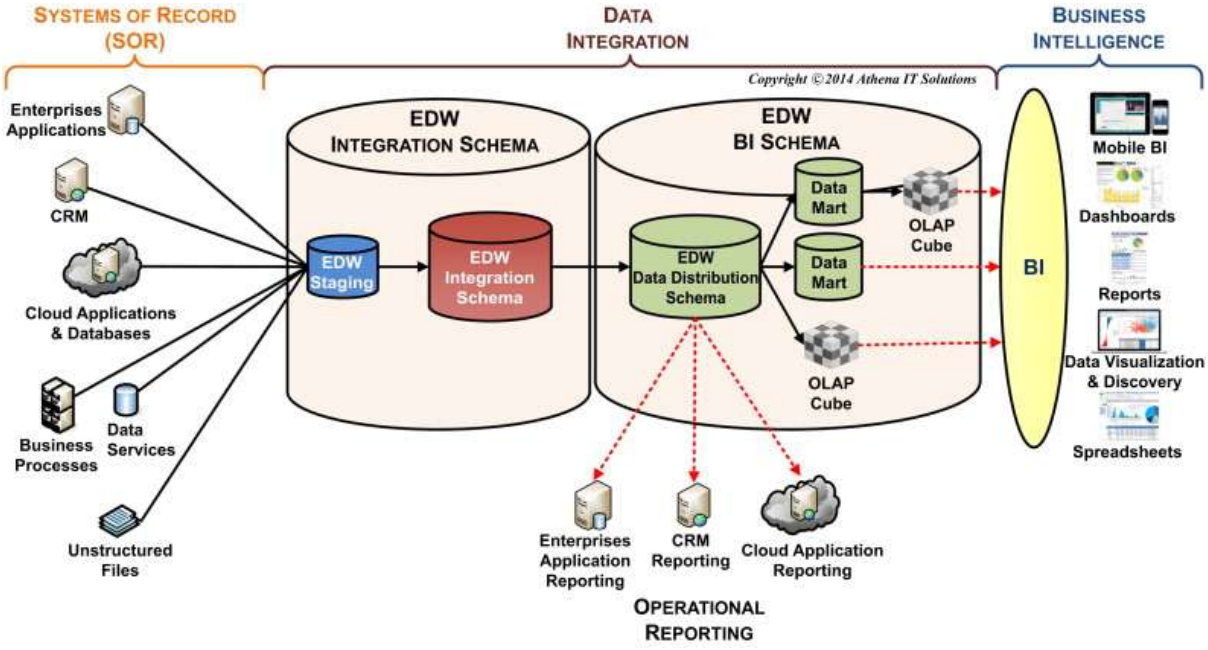
Common strategies used during integration:

- Big Bang Integration
- Incremental Integration (Top-Down / Bottom-Up / Hybrid)

🖼️ Diagram – System Integration Overview







4

✨ Characteristics

- Combines multiple modules into one system
- Focuses on interface correctness
- Ensures smooth data flow
- Performed after unit development
- Prepares system for system testing

- Detects interaction defects early
- Improves overall system reliability

🧪 **Example**

Online Shopping System

Separate modules:

- User Login
- Product Catalog
- Cart
- Payment
- Order Management

During System Integration:

- Login connects with Cart
- Cart connects with Payment
- Payment connects with Order module

If payment succeeds but order is not created, System Integration identifies this interaction defect.

1. Which of the following is a common software myth?

a) Software maintenance is easy
b) Software requirements are always clear at the start
c) Software testing is unnecessary
d) Software can always be perfect

✅ **Correct Answer: b) Software requirements are always clear at the start**

---

2. What does the term *“stakeholder”* refer to?

a) Developers only
b) Testers only
c) Any person affected by the software
d) Users only

✅ **Correct Answer: c) Any person affected by the software**

---

3. Which of the following is a behavioral UML diagram?

a) State Diagram
b) Activity Diagram
c) Use Case Diagram
d) All of these

✅ **Correct Answer: d) All of these**

---

4. Which of the following is NOT a key attribute of design quality?

a) Efficiency
b) Complexity
c) Modularity
d) Maintainability

✅ **Correct Answer: b) Complexity**

---

5. What does the acronym STLC stand for?

a) Software Testing Life Cycle
b) Software Development Life Cycle
c) Software Documentation Life Cycle
d) Software Technical Life Cycle

✅ **Correct Answer: a) Software Testing Life Cycle**

---

6. What is “Quality” in software development?

a) Meeting only the basic requirements
b) Free of bugs
c) Meeting specified and implied needs
d) Having good documentation

✅ **Correct Answer: c) Meeting specified and implied needs**

---

7. Load Testing falls under which category?

a) Functional testing
b) Non-functional testing
c) Usability testing
d) Security testing

✅ **Correct Answer: b) Non-functional testing**

---

8. What is the main focus of System Testing?

a) Individual components
b) Integrated modules
c) End-to-end functionality
d) API performance

✅ **Correct Answer: c) End-to-end functionality**

---

9. Which tool is used for unit testing in Python?

a) JMeter
b) PyTest
c) JUnit
d) Selenium

✅ **Correct Answer: b) PyTest**

---

10. What is the main objective of Exploratory Testing?

a) To follow test scripts strictly
b) To explore the system with minimal planning
c) To test performance of system
d) To test usability

✅ **Correct Answer: b) To explore the system with minimal planning**

---

11. What is the primary purpose of Static Testing?

a) To find runtime defects
b) To find defects without executing code
c) To measure performance
d) To test the system completely

✅ **Correct Answer: b) To find defects without executing code**

---

12. Equivalence Partitioning is a:

a) White box testing technique
b) Black box testing technique
c) Static analysis technique
d) Structural testing technique

✅ **Correct Answer: b) Black box testing technique**

How to Write a Use Case Diagram for an Online Airline Reservation System

📘 **Introduction**

A **Use Case Diagram** is a UML behavioral diagram that shows **actors (users)** and their **interactions with system functionalities**.

As explained by GeeksforGeeks, it focuses on **what the system does** from a user's perspective.

For an **Online Airline Reservation System**, it models how passengers and admins interact with booking services.

⚙️ Steps to Draw the Use Case Diagram

Step 1 – Identify Actors

- Passenger / Customer
- Admin
- Payment Gateway (external system)

Step 2 – Identify Use Cases

- Register / Login
- Search Flights
- View Flight Details
- Book Ticket
- Select Seats
- Make Payment
- Generate Ticket
- Cancel Booking
- View Booking History
- Manage Flights (Admin)

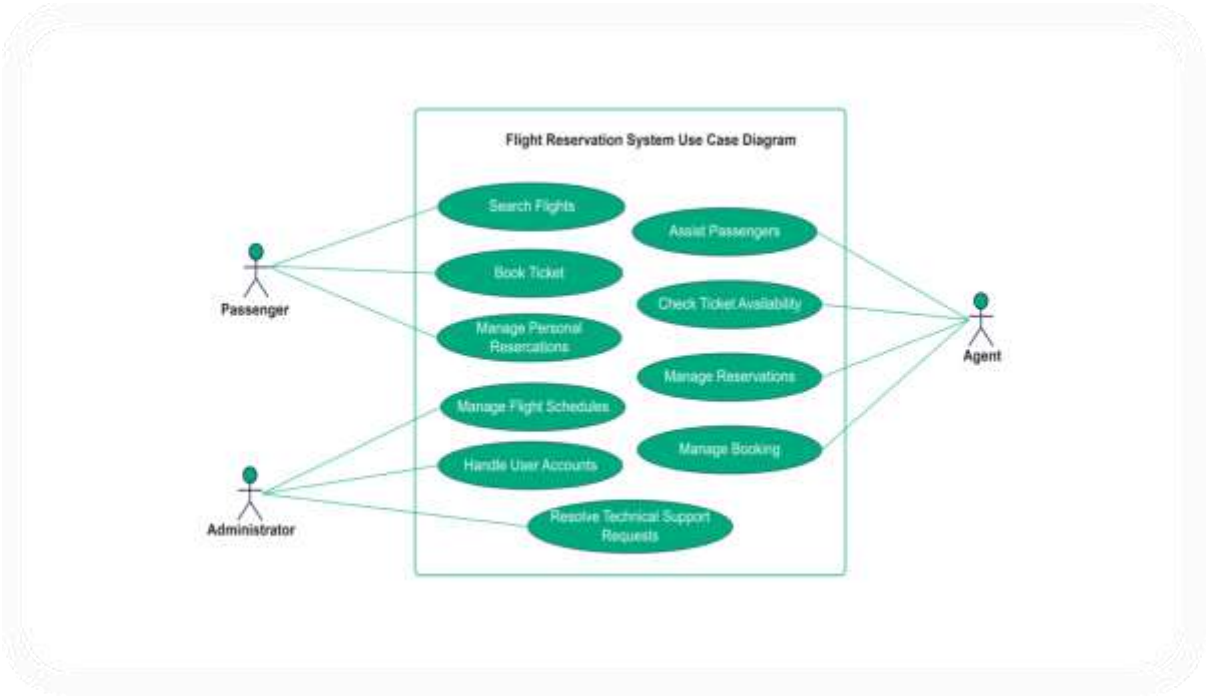
Step 3 – Draw System Boundary

Label it: **Online Airline Reservation System**

Step 4 – Connect Actors to Use Cases

Use association lines.

🖼️ Diagram – Use Case Diagram (Online Airline Reservation System)



4

✨ Characteristics

- User-centered view
- Shows system scope clearly
- Easy to understand
- Helps requirement analysis
- High-level system representation
- Technology independent

Defect Bash (Bug Bash)

📘 Introduction

Defect Bash (also called **Bug Bash**) is a **collaborative, time-boxed testing activity** where people from different roles—developers, testers, product managers, designers, and even support teams—come together to **explore the application simultaneously and find defects**.

According to GeeksforGeeks, Defect Bash is an **informal, exploratory testing session** conducted near the end of development to uncover **hidden bugs, usability issues, and edge cases** that scripted testing may miss.

In simple words:

👉 *Defect Bash = many people testing the product at the same time to quickly find maximum defects.*

⚙️ Working (How Defect Bash Is Conducted)

A Defect Bash typically follows these steps:

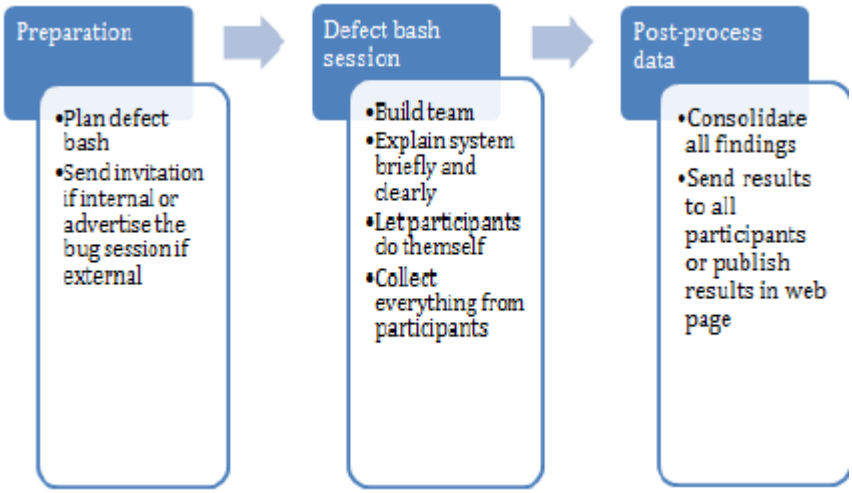
1. **Planning**
    - Decide scope (modules/features)
    - Fix duration (usually 1–4 hours)
    - Invite participants from multiple teams
2. **Briefing**
    - Explain objectives and rules
    - Share build URL, credentials, and bug-logging process
3. **Execution**
    - Everyone explores the system freely
    - Try unusual inputs, workflows, and edge cases

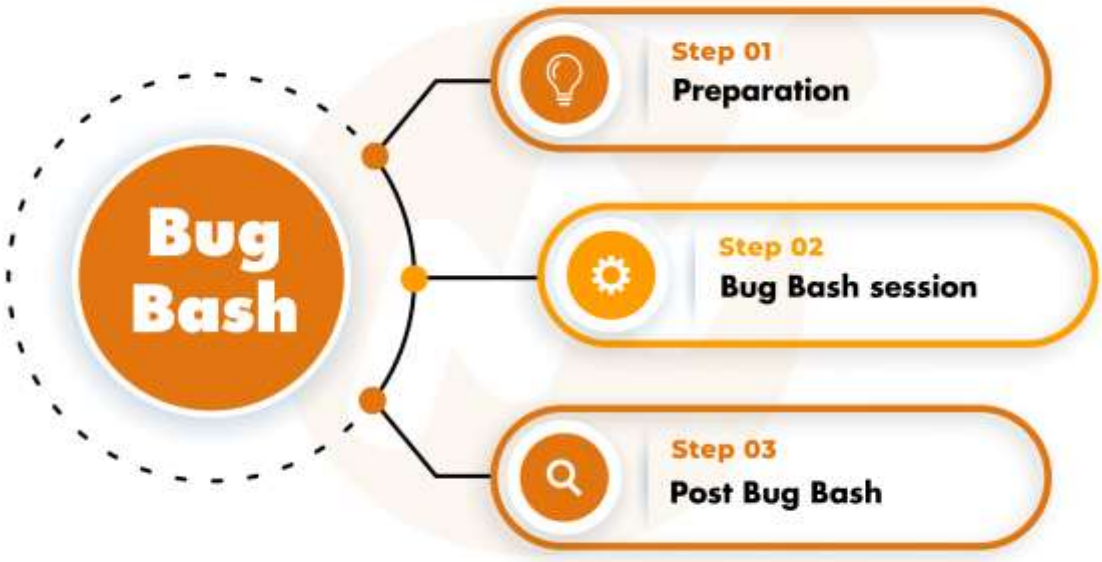
- Log defects with screenshots/steps

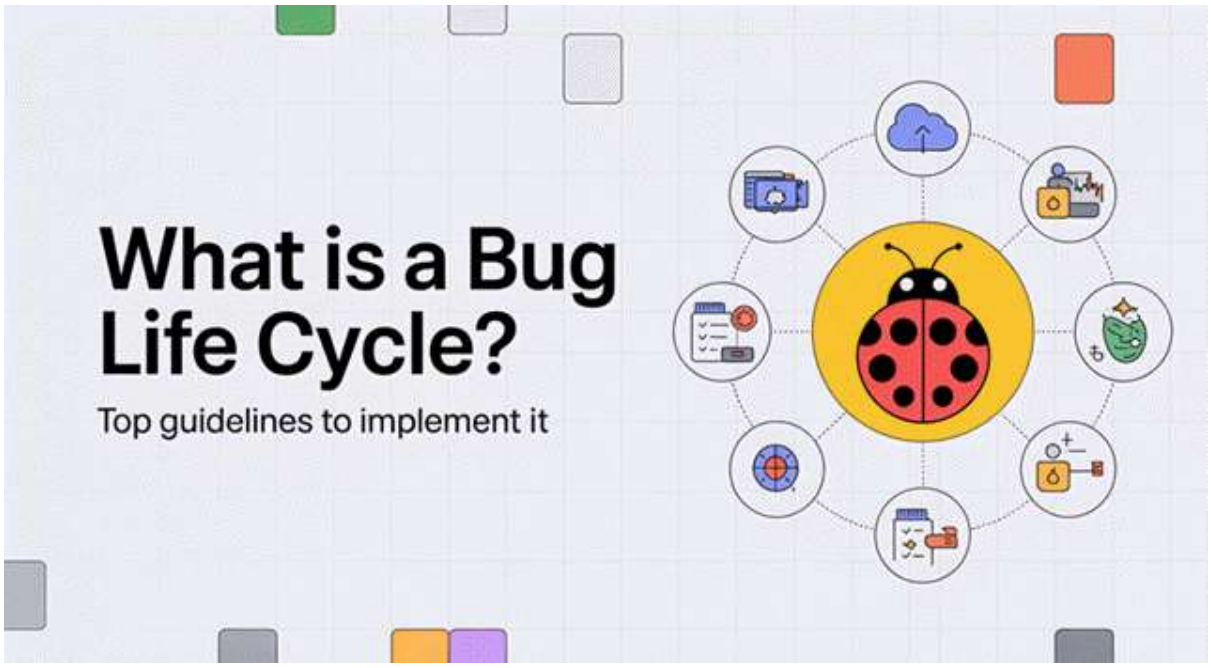
4. **Defect Review**
    - Remove duplicates
    - Prioritize bugs (Critical/High/Medium/Low)
5. **Closure**
    - Share summary (total bugs, major issues)
    - Assign fixes to developers

Key idea: **Exploratory + collaborative testing in a short time window.**

🖼️ Diagram – Defect Bash Workflow







4

✨ Characteristics

- Time-boxed activity
- Exploratory (no fixed test scripts)
- Involves cross-functional teams

- Fast defect discovery
- Focus on real user behavior
- Finds usability + functional issues
- High engagement and collaboration
- Usually done before release

🧪 **Example**

E-Commerce Website Defect Bash

Participants: QA, Developers, UI Designers, Product Manager

Everyone tests simultaneously:

- Login with invalid emails
- Add/remove items rapidly from cart
- Apply wrong coupon codes
- Switch browsers/devices
- Interrupt payment process

Results:

- Cart total miscalculation
- UI breaks on long product names
- Payment error message unclear

These issues might not appear in scripted tests but are caught during **Defect Bash**.

Major Factors Governing Performance Testing

🔹 **1) Response Time**

Time taken by the system to respond to a user request.

- **2) Throughput**

Number of transactions or requests processed per unit time.

- **3) Latency**

Initial delay between sending a request and receiving the first response.

- **4) Concurrent Users**

Number of users accessing the system simultaneously.

- **5) Workload / Load Pattern**

Type and volume of operations performed (login, search, checkout, etc.).

- **6) Resource Utilization**

Usage of CPU, memory, disk I/O, and network bandwidth.

- **7) Scalability**

Ability of the system to handle increased load by adding resources.

- **8) Network Conditions**

Bandwidth, packet loss, and network delays affect performance.

🔹 9) Test Environment

Hardware, OS, database, and server configuration.

🔹 10) Application Architecture

Monolithic vs microservices, caching, load balancers, etc.

✨ Characteristics

- Non-functional in nature
- Metrics driven
- Environment dependent
- Iterative process
- Helps capacity planning
- Identifies bottlenecks early

🧪 Example

E-commerce Website

- 1000 concurrent users
- Checkout response < 3 seconds
- CPU spikes to 90% during payment

Here, **concurrent users, response time, and CPU utilization** are key governing factors.

👍 Advantages

- Improves application speed
- Prevents production failures

- Enhances user experience
- Supports scalability planning
- Detects performance bottlenecks early

👎 **Limitations**

- Requires skilled engineers
- Tool and environment cost
- Time-consuming setup
- Results depend on test realism

✅ **Distinguish Between Object-Oriented and Procedural Software (12 Points)**

(Based on standard concepts summarized by GeeksforGeeks)

No.	Object-Oriented Software	Procedural Software
1	Based on **objects and classes**	Based on **functions and procedures**
2	Follows **bottom-up approach**	Follows **top-down approach**
3	Data is **encapsulated** with methods	Data and functions are **separate**
4	Supports **inheritance**	Does not support inheritance
5	Supports **polymorphism**	No polymorphism
6	Easier **code reusability**	Limited reusability

No.	Object-Oriented Software	Procedural Software
7	More secure due to **data hiding**	Less secure
8	Better for **large & complex systems**	Better for **small applications**
9	Examples: Java, C++, Python	Examples: C, Pascal
10	Easy maintenance and scalability	Maintenance becomes difficult as size grows
11	Real-world modeling is easy	Real-world modeling is difficult
12	Modular and flexible design	Rigid program structure