**Writing Test Conditions & Performing the Testing of the System**

**Types of Testing:**

**1. Static Testing:**

**Static testing involves reviewing the system and its components without executing the code. The goal is to detect errors early in the development lifecycle.**

* **Reviews: Different types of reviews are performed during static testing, including:**
  + **Technical Reviews: Conducted by a team of technical experts to check if the best approach has been used for implementation.**
  + **Management Reviews: Performed by high or mid-level management to track any deviations between planned work and actual work (i.e., slippage). This helps identify whether any objectives are off-track.**
  + **Formal Reviews:**
    - **Inspection: Conducted during the task’s execution, focusing on finding defects.**
    - **Audit: Performed after task completion to check for compliance and quality.**
  + **Informal Reviews:**
    - **Code Review: A less formal inspection typically performed by peers to evaluate the quality of the code.**
    - **Peer Review: A review conducted among team members for improvement suggestions and feedback.**

**2. Dynamic Testing:**

**Dynamic testing involves executing the system and evaluating its behavior under different conditions.**

* **Unit Testing: Testing individual units or components of the system to ensure each part works as expected.**
* **Integration Testing: Testing the integration of different components or systems to ensure they work together.**
* **System Testing: Testing the complete system to verify that all parts function correctly when integrated.**
* **User Acceptance Testing (UAT): Performed by the end users to ensure the system meets their needs and requirements.**

**Implementation / Deployment:**

**1. Primary Activities in Implementation:**

* **Write Detailed User Documentation: Prepare comprehensive documentation that helps users understand how to operate the system.**
* **Provide Training for the System Users: Training sessions to ensure that end users know how to navigate the system effectively.**

**2. Implementation Methods:**

* **Parallel Implementation: Running both the old and new system simultaneously until the new system is stable.**
* **Plunge Implementation: Switching directly from the old system to the new system in one go.**
* **Pilot Implementation: Deploying the system in a small environment (one department or group) before full-scale deployment.**
* **Phased Implementation: Deploying the system in phases, focusing on different features or regions gradually.**

**Maintenance:**

**1. Primary Activities in Maintenance:**

* **Build a Help Desk: A support system where users can get assistance for troubleshooting and resolving issues.**
* **Provide Environment Support: Ensuring that the environment (software and hardware) is maintained, updated, and supported throughout the system's lifecycle.**

**Software Development Methodology**

**A software development methodology is a framework used to structure, plan, and execute the software development process. It defines the processes, techniques, and tools to be used at each stage of the software development life cycle (SDLC). Different methodologies suit different project types, team sizes, and organizational goals. Below, we'll explore the most commonly used software development methodologies.**

**1. Waterfall Model**

🔹 **Definition**: A linear and sequential development approach where each phase must be completed before moving to the next.  
🔹 **Phases**:

1. **Requirement Gathering** – Collect all project requirements.
2. **System Design** – Create architecture and technical design.
3. **Implementation (Coding)** – Developers write the code.
4. **Testing** – Verify that the software works correctly.
5. **Deployment** – Release the software to users.
6. **Maintenance** – Fix issues and provide updates.  
   🔹 **Best For**: Small projects with **fixed requirements**.  
   🔹 **Pros**: Simple, well-documented.  
   🔹 **Cons**: No flexibility, late testing may reveal major issues.

**Advantages**

✔ Simple and easy to manage.  
✔ Clear documentation.  
✔ Works well for projects with **fixed requirements**.

**Disadvantages**

❌ Not flexible—difficult to accommodate changes once a phase is completed.  
❌ Late-stage testing can reveal major issues.  
❌ Not suitable for complex or evolving projects.

**Best Use Cases**

* Government projects.
* Small-scale projects with **well-defined requirements**.
* Projects with **low risk and no frequent changes**.

**2. V-Shaped Model**

🔹 **Definition**: An extension of the Waterfall model, but testing is done in parallel with development.  
🔹 **Phases**:

* Every development phase has a corresponding **testing phase**.
* Example: Requirement Analysis → Acceptance Testing, Design → System Testing, etc.  
  🔹 **Best For**: **Critical systems** (e.g., medical, banking software).  
  🔹 **Pros**: **Early bug detection**, strict quality control.  
  🔹 **Cons**: **Rigid**, difficult to accommodate changes.

**3. Spiral Model**

🔹 **Definition**: A risk-driven model that develops software in cycles with continuous risk assessment.  
🔹 **Phases (Repeated in Loops)**:

1. **Planning** – Define objectives and risks.
2. **Risk Analysis** – Identify potential risks and create solutions.
3. **Development & Testing** – Build and test a prototype.
4. **Customer Evaluation** – Get feedback and refine the product.  
   🔹 **Best For**: **Complex and high-risk projects** (e.g., aerospace, large financial systems).  
   🔹 **Pros**: **Handles risks well**, flexible.  
   🔹 **Cons**: **Expensive**, requires skilled risk management.

**4. Incremental Model**

🔹 **Definition**: The software is built and delivered in **small parts (increments)**, each adding new functionality.  
🔹 **Phases**:

* Develop the **core features** first.
* Add more features in later **increments**.
* Each increment is tested and refined before moving to the next.  
  🔹 **Best For**: **Projects needing quick releases** (e.g., SaaS, e-commerce).  
  🔹 **Pros**: Faster delivery, flexible.  
  🔹 **Cons**: Requires good **planning** from the start.

**5. Agile Model**

🔹 **Definition**: A highly flexible and **customer-focused** development approach with frequent releases.  
🔹 **Key Concepts**:

* Work is divided into **small cycles (iterations or sprints)**.
* Frequent testing and feedback.
* Collaboration between developers and customers.  
  🔹 **Best For**: **Startups, dynamic projects** with evolving requirements.  
  🔹 **Pros**: Adaptable, fast delivery, continuous improvement.  
  🔹 **Cons**: Requires **constant communication**, hard to manage for large teams.

**Comparison of Models**

| **Model** | **Flexibility** | **Risk Management** | **Speed** | **Best For** |
| --- | --- | --- | --- | --- |
| **Waterfall** | Low | Low | Slow | Fixed requirements |
| **V-Shaped** | Low | Medium | Medium | High-quality software |
| **Spiral** | High | High | Slow | Risky, large-scale projects |
| **Incremental** | Medium | Medium | Fast | Early software releases |
| **Agile** | High | Medium | Fast | Changing requirements |

**Final Thoughts**

✔ If **requirements are fixed** → Use **Waterfall or V-Model**.  
✔ If **risk is high** → Use **Spiral Model**.  
✔ If **quick releases are needed** → Use **Incremental Model**.  
✔ If **requirements change frequently** → Use **Agile Model**.

**ANOTHER WAY OF MODELS:**

**1. Waterfall Model**

**Overview**

The **Waterfall Model** is a traditional, linear, and sequential software development approach. It follows a strict **step-by-step** process where each phase must be completed before moving to the next.

**Phases of the Waterfall Model**

1. **Requirement Gathering & Analysis**
   * Collect all requirements from stakeholders.
   * Document everything in a Software Requirement Specification (SRS).
2. **System Design**
   * Plan software architecture, database structure, and UI design.
   * Specify hardware and software requirements.
3. **Implementation (Coding & Development)**
   * Developers write the actual code based on the design.
   * Each module is developed and tested separately.
4. **Testing**
   * Performed after the coding phase.
   * Includes unit testing, integration testing, system testing, and user acceptance testing.
5. **Deployment**
   * The software is released for end users.
   * Can be deployed in phases or fully at once.
6. **Maintenance**
   * Fix errors and provide updates based on user feedback.
   * Performance improvements and security patches are applied.

**Advantages**

✔ Simple and easy to manage.  
✔ Clear documentation.  
✔ Works well for projects with **fixed requirements**.

**Disadvantages**

❌ Not flexible—difficult to accommodate changes once a phase is completed.  
❌ Late-stage testing can reveal major issues.  
❌ Not suitable for complex or evolving projects.

**Best Use Cases**

* Government projects.
* Small-scale projects with **well-defined requirements**.
* Projects with **low risk and no frequent changes**.

**2. V-Shaped Model (Validation & Verification Model)**

**Overview**

The **V-Shaped Model** is an extension of the Waterfall model but with an emphasis on **testing at each stage**. Each development stage has a **corresponding testing phase**, forming a "V" shape.

**Phases in the V-Shaped Model**

1. **Requirement Analysis** → **Acceptance Testing**
   * Requirements are gathered & analyzed.
   * Acceptance testing ensures all requirements are met.
2. **System Design** → **System Testing**
   * High-level design of software architecture.
   * System testing checks the overall system’s functionality.
3. **High-Level Design** → **Integration Testing**
   * Define major software components & their interactions.
   * Integration testing verifies proper communication between modules.
4. **Low-Level Design** → **Unit Testing**
   * Detailed design of individual modules.
   * Unit testing ensures each module functions correctly.
5. **Implementation (Coding)**
   * Develop the actual software based on the design.

**Advantages**

✔ Catches defects early.  
✔ Testing is done in **parallel** with development.  
✔ Works well for **projects requiring high reliability**.

**Disadvantages**

❌ Rigid structure—difficult to change requirements later.  
❌ Can be **time-consuming** due to extensive testing.

**Best Use Cases**

* **Healthcare & banking software** (where failures are unacceptable).
* Software requiring **strict quality control**.

**3. Spiral Model**

**Overview**

The **Spiral Model** combines **iterative development** with **risk management**. It is best for complex projects where risks need to be analyzed continuously.

**Phases of the Spiral Model**

Each cycle (spiral) contains four phases:

1. **Planning**
   * Gather customer requirements.
   * Identify risks and create mitigation strategies.
2. **Risk Analysis**
   * Identify technical and business risks.
   * Prototype potential solutions.
3. **Development & Testing**
   * Develop a version of the product.
   * Test it for defects and usability.
4. **Review & Refinement**
   * Gather customer feedback.
   * Prepare for the next cycle.

**Advantages**

✔ **Risk handling** at every phase.  
✔ Flexible—changes can be made during development.  
✔ Best for **large, high-risk projects**.

**Disadvantages**

❌ Expensive and time-consuming.  
❌ Requires skilled risk analysts.

**Best Use Cases**

* **Aerospace & defense software**.
* **Large, expensive projects with evolving requirements**.

**4. Incremental Model**

**Overview**

The **Incremental Model** delivers the software in **small parts (increments)**, where each increment adds functionality.

**Phases of the Incremental Model**

1. **Requirement Gathering**
   * Define only the core requirements.
   * Additional requirements are handled in later increments.
2. **Design & Implementation (Increment 1)**
   * Develop the first version with basic features.
3. **Testing & Feedback**
   * Test & review before adding new features.
4. **Next Increment Development**
   * Build the next version with additional features.

**Advantages**

✔ Faster software delivery.  
✔ Easier debugging and testing.  
✔ More **flexibility** compared to the Waterfall model.

**Disadvantages**

❌ Requires good **planning** from the start.  
❌ Modules must be well-defined early.

**Best Use Cases**

* **E-commerce and SaaS applications**.
* **Projects needing early releases with gradual improvements**.

**5. Agile Model**

**Overview**

The **Agile Model** is a modern, **iterative**, and **collaborative** approach that focuses on **continuous feedback & improvement**.

**Key Principles of Agile**

1. **Individuals & interactions over processes & tools**.
2. **Working software over comprehensive documentation**.
3. **Customer collaboration over contract negotiation**.
4. **Responding to change over following a plan**.

**Agile Frameworks**

* **Scrum** → Iterative development in short cycles (sprints).
* **Kanban** → Workflow visualization with a task board.
* **Extreme Programming (XP)** → Focuses on quality & fast iterations.

**Advantages**

✔ High **flexibility** to changes.  
✔ Continuous delivery & feedback.  
✔ Involves **customers** throughout development.

**Disadvantages**

❌ Requires frequent meetings & collaboration.  
❌ Not suitable for **small teams or projects with fixed scope**.

**Best Use Cases**

* **Startups & innovative projects**.
* **Software with evolving requirements**.

**Comparison of Models**

| **Model** | **Flexibility** | **Risk Management** | **Speed** | **Best For** |
| --- | --- | --- | --- | --- |
| **Waterfall** | Low | Low | Slow | Fixed, well-defined projects |
| **V-Shaped** | Low | Medium | Medium | High-quality software |
| **Spiral** | High | High | Slow | Risky, large-scale projects |
| **Incremental** | Medium | Medium | Fast | Early software releases |
| **Agile** | High | Medium | Fast | Dynamic, evolving projects |

**Final Thoughts**

🔹 If **requirements are fixed** → Use **Waterfall or V-Model**.  
🔹 If **risk is high** → Use **Spiral Model**.  
🔹 If **you need quick releases** → Use **Incremental Model**.  
🔹 If **requirements change frequently** → Use **Agile Model**.

TDD

````

// LoginService.java

public class LoginService {

  public boolean authenticate(String username, String password) {

       // Mock authentication logic

       return "admin".equals(username) && “admin@123".equals(password);

   }

}

// LoginServiceTest.java

import org.junit.Test;

import static org.junit.Assert.\*;

public class LoginServiceTest {

   @Test

   public void testAuthenticate() {

       LoginService loginService = new LoginService();

      assertTrue(loginService.authenticate("admin", "admin@123"));

   }

   @Test

   public void testInvalidCredentials() {

       LoginService loginService = new LoginService();

      assertFalse(loginService.authenticate("user", "wrongpassword"));

   }

}

BDD

Calculator.feature

```````````````````

Feature: Calculator

 As a user

 I want to perform basic arithmetic operations

 So that I can verify the correctness of calculations

 Scenario: Addition

   Given I have entered 50 into the calculator

   And I have entered 70 into the calculator

   When I press add

   Then the result should be 120 on the screen

 Scenario: Subtraction

   Given I have entered 70 into the calculator

   And I have entered 30 into the calculator

   When I press subtract

   Then the result should be 40 on the screen

CalculatorSteps.java ( Step Definition File)

`````````````````````````````````````````````

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

import org.junit.Assert;

public class CalculatorSteps {

   private int number1;

   private int number2;

   private int result;

   @Given("I have entered {int} into the calculator")

   public void iHaveEnteredNumberIntoTheCalculator(int number) {

       if (number1 == 0)

           number1 = number;

       else

           number2 = number;

   }

   @When("I press add")

   public void iPressAdd() {

       result = number1 + number2;

   }

   @When("I press subtract")

   public void iPressSubtract() {

       result = number1 - number2;

   }

   @Then("the result should be {int} on the screen")

   public void theResultShouldBeExpectedResultOnTheScreen(int expectedResult) {

      Assert.assertEquals(expectedResult, result);

   }

}

TestRunner.java

````````````````

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

import org.junit.runner.RunWith;

@RunWith(Cucumber.class)

@CucumberOptions(features = "src/test/resources",

       glue = "stepdefinitions")

public class TestRunner {

}

**SCRUM VS. KANBAN VS. LEAN: KEY DIFFERENCES**

| **Aspect** | **Scrum** | **Kanban** | **Lean** |
| --- | --- | --- | --- |
| **Structure** | Time-boxed sprints, roles, ceremonies | No time-boxed sprints, flexible flow | Focus on eliminating waste, improving flow |
| **Flexibility** | Less flexible, with fixed sprints and ceremonies | Highly flexible, adapts to changing priorities | Flexible, continuous improvement over time |
| **Work Flow** | Iterative with regular review and feedback | Continuous flow, work pulled as needed | Optimized for efficiency, focuses on value |
| **Focus** | Deliver a shippable increment at the end of each sprint | Manage work-in-progress and flow | Maximize value, reduce waste |
| **WIP Limits** | Not explicitly defined | Explicit WIP limits to improve flow | Focus on streamlining processes, minimizing waste |
| **Roles** | Scrum Master, Product Owner, Development Team | No specific roles, just teams working collaboratively | No specific roles; applies to all processes |
| **When to Use** | Projects with clear, structured goals and timelines | Work that flows continuously and needs flexible prioritization | Improving overall efficiency, reducing waste |