Introduction

Overview and Significance in the AI Era

Software testing is one of the most vital components in the modern software development lifecycle (SDLC). As applications scale and integrate with technologies like Artificial Intelligence (AI), Machine Learning (ML), cloud computing, and the Internet of Things (IoT), ensuring their correctness, reliability, and performance becomes a top priority. Testing is no longer viewed as a concluding step in the development pipeline but as a continuous, integrated process that spans across all phases of software engineering.

In AI-driven systems, where outcomes can often be non-deterministic and data quality plays a central role, software testing demands even more attention. Models must not only be accurate but also interpretable, robust, and unbiased. These additional constraints make the testing process more complex and context-dependent.

The online course titled ***"Introduction to Software Testing"*** offered a structured and in-depth walkthrough of software testing fundamentals, both from traditional and modern (agile) perspectives. It covered essential concepts such as unit testing, integration testing, system testing, and acceptance testing, along with various testing approaches like black-box, white-box, and exploratory testing. I also gained hands-on experience with tools such as JUnit5 for test-driven development (TDD), and Cucumber for behaviour-driven development (BDD), enabling me to apply what I learned in practical coding environments.

Furthermore, the course delved into automation testing frameworks, continuous integration/continuous deployment (CI/CD) pipelines, and tools like Selenium, Jenkins, and Travis CI.

The role of the Software Development Engineer in Test (SDET) was introduced as an emerging and increasingly important hybrid role in the industry—someone who possesses the coding skills of a developer and the critical eye of a tester. This aligns closely with the current trends in AI software engineering, where code correctness, testability, and automation must go hand-in-hand with performance and scalability.

Software Testing Requirements & Checklist

Testing Requirements

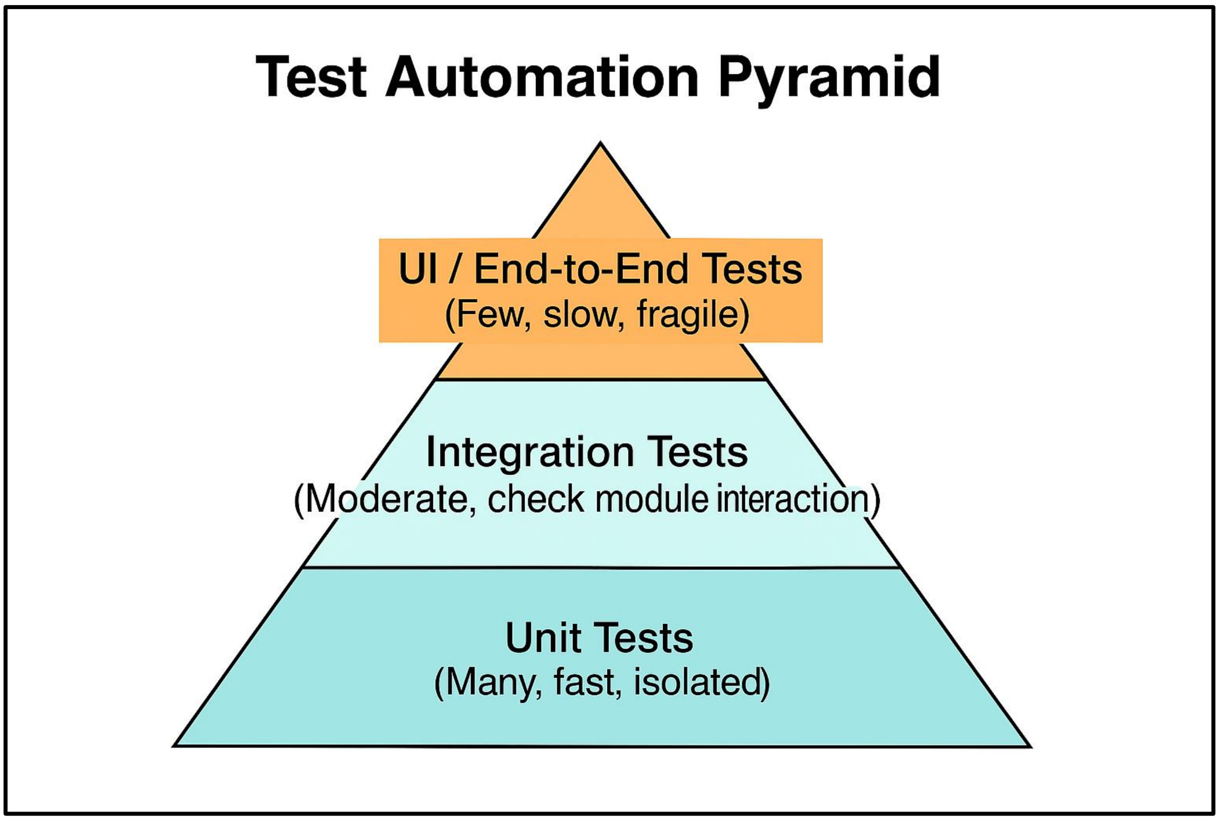
Before initiating any form of software testing, certain foundational elements must be in place to ensure that the process is effective, traceable, and aligned with project goals. These testing requirements define what needs to be tested, how it will be tested, and what outcomes are expected. Especially in AI-driven development, where dynamic behaviour and learning models are involved, clearly defined requirements are essential to manage the scope and complexity of testing.

Test Automation Pyramid

The **Test Automation Pyramid** is a conceptual framework that guides the distribution of different types of tests within a software system. The idea is to write more low-level tests (like unit tests), fewer mid-level (integration), and even fewer high-level (UI or end-to-end) tests. This structure enhances maintainability, speed, and efficiency in automated testing.

Pyramid Levels Explained:

- **Unit Tests:** Verify individual functions or components. Fast and isolated. Tools: JUnit, PyTest.
- **Integration Tests:** Ensure multiple components or systems work together. Tools: JUnit, TestNG.
- **UI Tests / End-to-End:** Simulate real user workflows. Slower and more brittle. Tools: Selenium, Cypress.



Traditional Software Testing Approach

In traditional software development models, testing is treated as a distinct phase that comes after coding is completed. This model emphasizes sequential stages—requirements, design, development, testing, and deployment—with limited flexibility for changes once development begins. However, the drawback of this approach is the late detection of bugs, which leads to higher costs of fixing defects and reduced adaptability to changing requirements.

Agile Software Testing Approach

Agile testing is integrated throughout the development process. It follows the principles of continuous feedback, iterative improvement, and collaboration between developers, testers, and stakeholders.

Benefits of Agile Testing:

- Early and Continuous Testing
- Better Collaboration
- Faster Time to Market

- Adaptability to Change
- Test Automation
- Improved Product Quality

Levels and Approaches of Software Testing

Software testing is performed at multiple levels to ensure thorough verification and validation of the application at various stages of its development.

Levels of Software Testing

1. Unit Testing
2. Integration Testing
3. System Testing
4. Operational Acceptance Testing (OAT)

Approaches to Software Testing

1. Static Testing
2. Dynamic Testing
3. Exploratory Testing
4. The Box Approaches:
   - Black-Box Testing
   - White-Box Testing
   - Grey-Box Testing

Overview of Test-Driven Development

Test-Driven Development (TDD) is a software development methodology that emphasizes writing automated test cases before writing the actual code. It follows a short and repetitive cycle often summarized as **Red-Green-Refactor**:

1. **Write a test** for a new function or feature (Red — it fails initially).
2. **Write just enough code** to make the test pass (Green).
3. **Refactor** the code while keeping all tests passing.
4. **Repeat** the process for each new feature or functionality.

This method ensures that every unit of functionality is tested right from the beginning, resulting in cleaner, more maintainable, and bug-free code.

Advantages of TDD

- Early bug detection and fast feedback loop
- Encourages modular and loosely coupled code
- Improves code quality and maintainability
- Less debugging and rework
- Better understanding of requirements through test scenarios
- Facilitates continuous integration and automation

- Enhances developer confidence when refactoring code
- Leads to better-designed interfaces and APIs

JUnit5: A Powerful TDD Framework

JUnit5 is the latest version of the widely-used Java testing framework. It offers a modern and flexible approach to writing unit tests with improved annotations and features compared to previous versions.

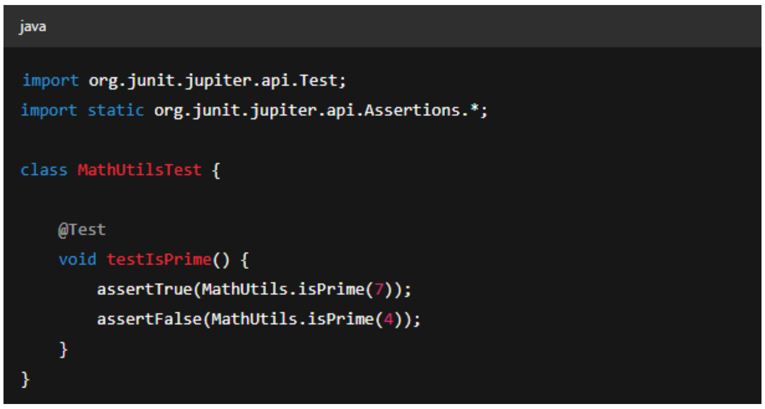
Basic JUnit5 Annotations:

- @Test — Marks a method as a test
- @BeforeEach — Runs before each test
- @AfterEach — Runs after each test
- @BeforeAll / @AfterAll — Run once before/after all tests
- @DisplayName — Describes test case in a readable format
- @Nested, @ParameterizedTest — For grouped and data-driven testing

Sample TDD Cycle with JUnit5

Let's say we're developing a simple method to check whether a number is prime.

Step 1: Write the Failing Test (Red Phase)



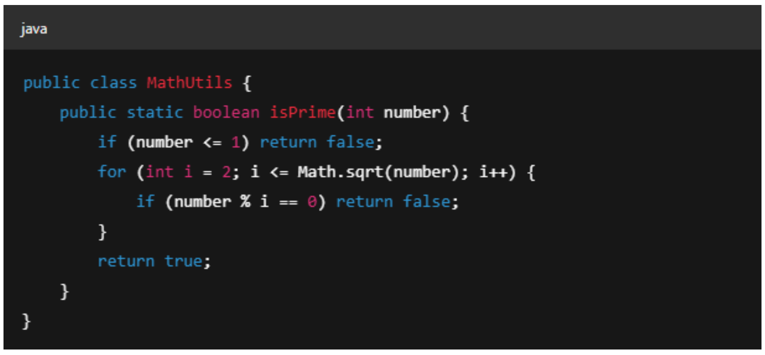
java

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class MathUtilsTest {

    @Test
    void testIsPrime() {
        assertTrue(MathUtils.isPrime(7));
        assertFalse(MathUtils.isPrime(4));
    }
}
```

Step 2: Write the Code to Make It Pass (Green Phase)



java

```java
public class MathUtils {
    public static boolean isPrime(int number) {
        if (number <= 1) return false;
        for (int i = 2; i <= Math.sqrt(number); i++) {
            if (number % i == 0) return false;
        }
        return true;
    }
}
```

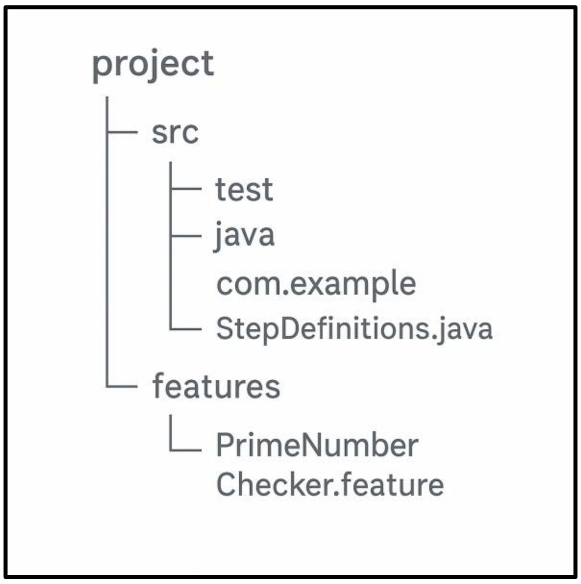
Behaviour-Driven Development (BDD) is a software development approach that bridges the gap between technical and non-technical stakeholders. It extends the principles of TDD by using natural language constructs to describe the behaviour of a system.

BDD encourages collaboration among developers, testers, business analysts, and product owners by writing test scenarios in a shared language that everyone understands.

Benefits of BDD

- Improved collaboration between developers, testers, and stakeholders
- Clear, human-readable test cases
- Ensures software behaviour aligns with business expectations
- Promotes shared understanding and documentation
- Encourages writing tests before development

Cucumber project structure diagram



```
project
├─ src
│   ├─ test
│   ├─ java
│   │  com.example
│   └─ StepDefinitions.java
└─ features
    └─ PrimeNumber
       Checker.feature
```

Agile Testing Advantages

Agile testing is not a standalone phase—it is a continuous and integrated process that begins at the start of the project and continues through each iteration or sprint. It aligns closely with Agile principles of flexibility, collaboration, and customer satisfaction, providing faster feedback and higher quality releases.

Advantages of Agile Testing

- Early and Continuous Feedback
- Improved Product Quality
- Better Collaboration
- Flexibility to Change
- Automated Testing and CI/CD Integration
- Faster Time to Market
- Customer-Centric Focus

Popular Software Testing Techniques

Software testing techniques provide structured approaches for validating different aspects of an application—from functionality to usability, security, and performance. A good testing strategy often combines several of these methods to ensure comprehensive coverage.

Key Techniques in Practice

A/B Testing

Compares two versions of a feature (A and B) to determine which performs better based on user behaviour. Commonly used in UI/UX design, marketing features, and conversion optimization.

Usability Testing

Assesses how easy and intuitive the software is for end-users. Testers observe real users as they interact with the application to identify pain points or confusion.

Security Testing

Focuses on identifying vulnerabilities that could be exploited. It includes penetration testing, authentication checks, and data protection assessments.

Continuous Testing

In Agile and DevOps environments, automated tests are run frequently as part of the CI/CD pipeline. It helps detect issues early and maintain deployment readiness.

Alpha Testing

Performed internally by developers or QA teams at the end of development but before the product is released to users. Focuses on bug fixing and performance validation.

Beta Testing

Conducted by real users in a real environment. It provides feedback from actual usage and helps identify unexpected issues before full-scale release.

Regression Testing

Ensures that new code changes don't unintentionally break existing functionality. Often automated and included in CI workflows.

Smoke Testing

A quick set of high-level tests to check if the basic functionalities are working. Often referred to as "build verification testing."

Sanity Testing

Performed after receiving a new build to verify that specific bugs have been fixed and no new major issues have been introduced.

Automation Testing Frameworks and Tools

Tool	**Purpose**
Selenium	Web application testing in multiple browsers
TestNG	Advanced test configuration and parallel execution
JUnit5	Unit testing framework for Java
Appium	Mobile app automation for Android and iOS

Tool	Purpose
Coderunner	Lightweight test execution environment
Jenkins	CI/CD tool for integrating and automating tests
SoapUI	API testing (REST and SOAP)
Robot Framework	Generic automation with tabular test cases
RSpec	BDD-style testing for Ruby
Eggplant Functional	Intelligent UI testing based on AI & image recognition
Mocha	JavaScript test framework (Node.js)
Jasmine	Behaviour-driven testing for JavaScript

Types of Automation Testing Frameworks

Each framework offers a different approach depending on how tests are structured, written, and maintained.

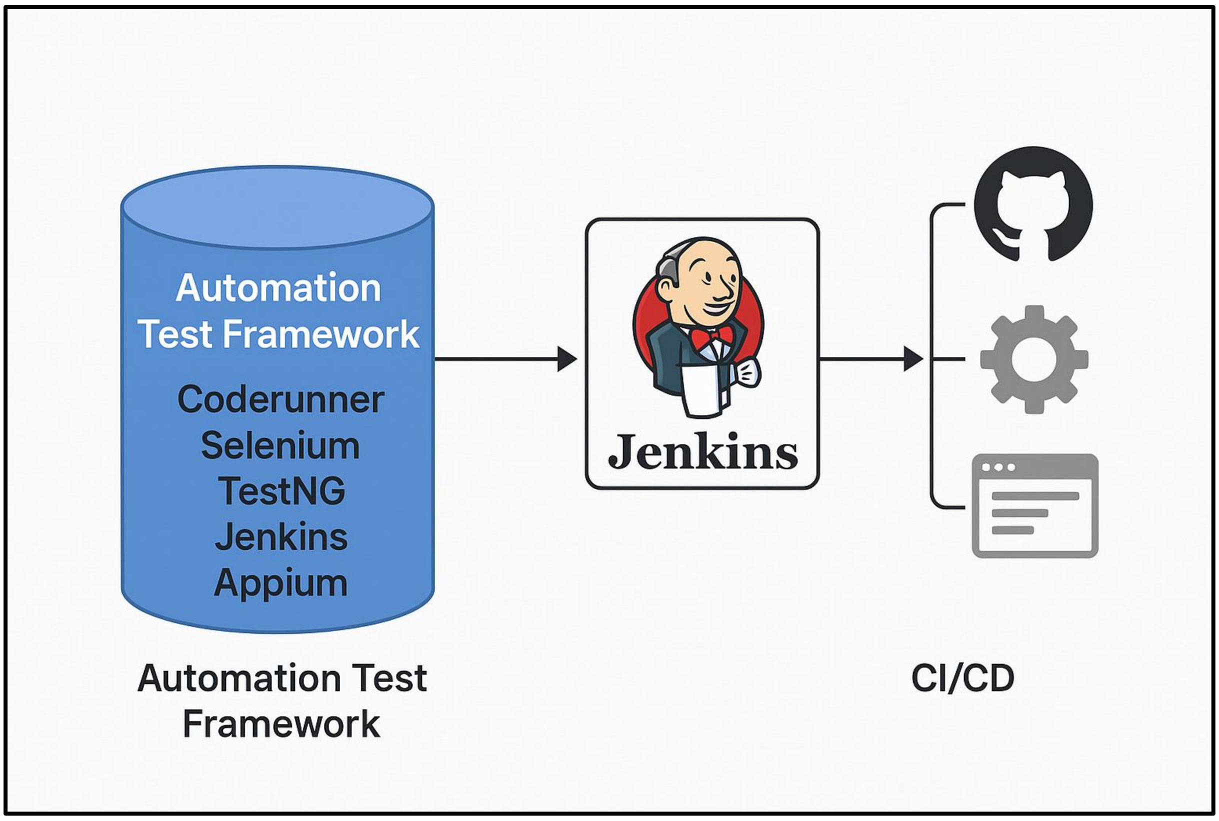
1. Linear Scripting Framework
2. Modular Testing Framework
3. Data-Driven Testing Framework
4. Keyword-Driven Testing Framework
5. Hybrid Testing Framework
6. Behaviour-Driven Development Framework

Integration with CI/CD Pipelines

Tools like **Jenkins**, **Travis CI**, and **GitHub Actions** enable the continuous execution of test cases as soon as code is committed. This ensures rapid feedback and promotes stable, deployable builds at all times.

Example Flow:

1. Developer pushes code to GitHub.
2. Jenkins triggers the build.
3. Automated tests run using Selenium + TestNG.
4. Results are reported and integrated into the pipeline.



Static vs Dynamic Testing

Static Testing

Static testing involves examining the software artifacts—such as source code, requirement documents, and design diagrams—*without executing the code*.

Purpose: To identify defects early in the development cycle before the code is run. It's focused on **prevention** rather than detection.

Advantages:

- Catches errors early and cheaply
- No need for compiled code
- Helps improve code structure and documentation
- Reduces downstream bugs

Dynamic Testing

Dynamic testing involves *executing the code* to check for functional correctness, performance, and other run-time behaviours.

Purpose: To validate that the software behaves as expected during execution and meets the business and technical requirements.

Advantages:

- Ensures actual behaviour matches expected behaviour
- Detects defects that can't be found through static methods
- Essential for end-to-end validation

The Exploratory and Box Approaches to Testing

Different testing methodologies provide different lenses through which to examine software. Two key categories in this context are **Exploratory Testing** and the **Box Testing Approaches** (White Box, Black Box, and Gray Box testing).

Exploratory Testing

Exploratory testing is an informal, hands-on approach where testers actively explore the software without pre-defined test cases. It relies on the tester's intuition, experience, and knowledge of the system.

Box Testing Approaches

a. White Box Testing (Clear Box Testing)

Focus: Internal logic and structure of the code.

Testers Know: How the code is written and can access it.

Use Cases:

- Verifying logic flow and loop conditions.
- Ensuring all paths and conditions are tested.
- Ideal for developers and test automation engineers.

b. Black Box Testing

Focus: Inputs and outputs of the system—*without any knowledge of the internal workings*. **Testers Know:** Only functional requirements and expected outcomes.

Use Cases:

- Validating end-user scenarios.
- Ensuring compliance with business requirements.
- Suitable for QA engineers, manual testers, and end-users.

c. Gray Box Testing

Focus: Partial knowledge of the internal structure combined with external behaviour. **Testers Know:** Some information about the system architecture or design.

Use Cases:

- When balancing between code-level insight and user-level behaviour.
- Security and data flow validations.
- **Summary Table**

Approach	Internal Knowledge	Typical Tests	Performed By
White Box	Full	Unit, Integration	Developers
Black Box	None	System, Acceptance	QA Testers
Gray Box	Partial	Security, Integration	Testers/Developers

Approach	Internal Knowledge	Typical Tests	Performed By
Exploratory	Ad-hoc/Intuitive	UI/Usability/Edge Cases	Skilled Testers

Software Testing Levels

Software testing is conducted at various levels throughout the development lifecycle to ensure every component and the system as a whole function correctly. These levels follow a bottom-up or top-down approach depending on the development methodology used.

Unit Testing

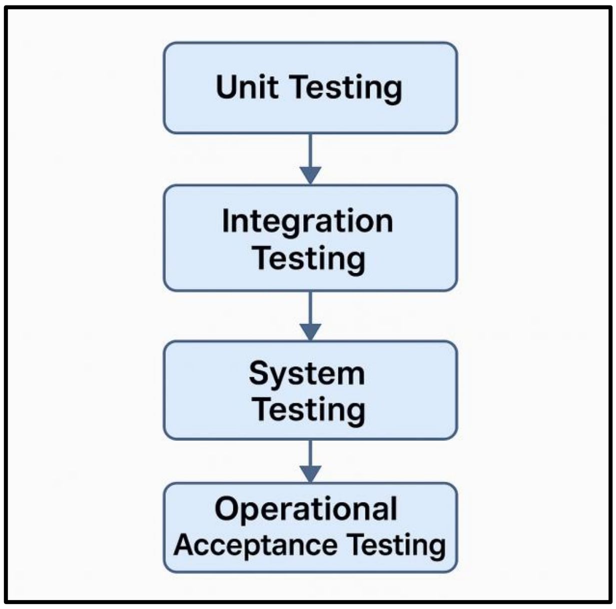
Integration Testing

System Testing

Operational Acceptance Testing (OAT)

Summary Table

Level	**Objective**	**Key Focus Area**	**Performed By**
Unit Testing	Validate individual functions/modules	Logic, internal behaviour	Developers
Integration Testing	Validate interaction between components	Data flow, APIs	Devs / Testers
System Testing	Validate end-to-end system behavior	Features, UI, performance	QA Team
Operational Acceptance	Validate production readiness	Stability, deployment	Ops / Release Team



Test-Driven Development (TDD) and Behaviour-Driven Development (BDD)

Both **TDD** and **BDD** emphasize writing tests early and using them to guide software development, but they differ in **focus, audience**, and **format**.

Test-Driven Development (TDD)

Definition:
TDD is a development process where test cases are written before the actual code. The development follows a short cycle: write a failing test, write code to pass the test, then refactor.

TDD Cycle

1. **Add a test**
2. **Run the test** – it should fail
3. **Write the minimal code** to make the test pass
4. **Run all tests**
5. **Refactor the code**
6. **Repeat**

Advantages of TDD

- Early bug detection
- Better code structure and modularity
- Fast feedback loop
- Promotes good design and documentation
- Easier refactoring with confidence
- Increases productivity and code quality

JUnit 5 for TDD

JUnit 5 is a popular testing framework used for implementing TDD in Java projects.

Behaviour-Driven Development (BDD)

Definition:
BDD extends TDD by using natural language to describe the behaviour of an application. It improves communication between developers, testers, and stakeholders using a **common language**.

Key Components:

- **Gherkin Syntax:** Uses plain English with keywords like Given, When, Then
- **Cucumber Framework:** Parses Gherkin files and connects them to code

BDD Principles:

- Focus on **business-readable scenarios**
- Encourages collaboration between technical and non-technical team members
- Tests describe **behaviour**, not implementation

Tools & Language Support:

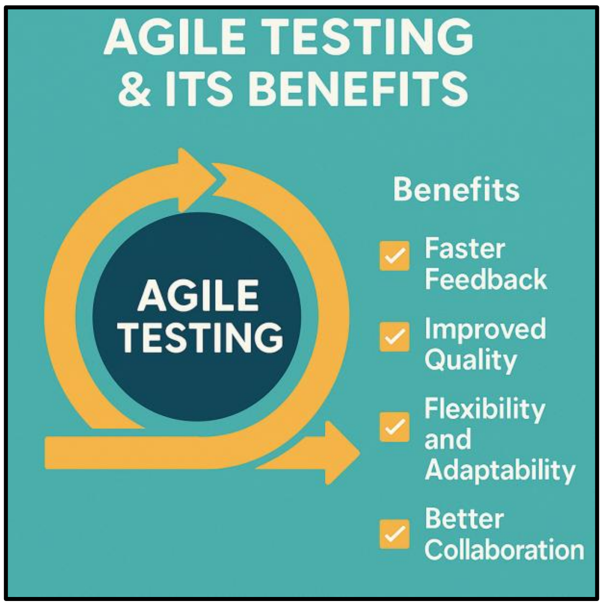
Language	Framework	Required Software
Java	Cucumber	Java SE, Maven, IntelliJ / Eclipse + Plugin
JavaScript	Cucumber.js	Node.js, VSCode / Atom
Ruby	Cucumber	Ruby, Bundler, TextMate / other editors
Kotlin	Cucumber	Java SE, Maven, IntelliJ w/ Kotlin plugin

TDD vs BDD Summary

Aspect	TDD	BDD
Focus	Code correctness	System behaviour
Language	Programming language	Natural language (Gherkin)
Stakeholder Friendly	Low	High
Tool Example	JUnit, TestNG	Cucumber, SpecFlow
Output	Unit tests	Executable specifications

Agile Testing and Its Advantages

Agile testing is a software testing practice that follows the principles of agile software development. Unlike the traditional model, testing in agile happens **concurrently** with development and is deeply collaborative, iterative, and user-focused.



What is Agile Testing?

Agile testing doesn't wait for the entire codebase to be completed. Instead, it is:

- **Continuous**: Testing occurs regularly and throughout the development lifecycle.
- **Collaborative**: Involves developers, testers, and stakeholders working closely.
- **Adaptive**: Responds to changes quickly with fast feedback loops.
- **Customer-Centric**: Driven by user stories and acceptance criteria.

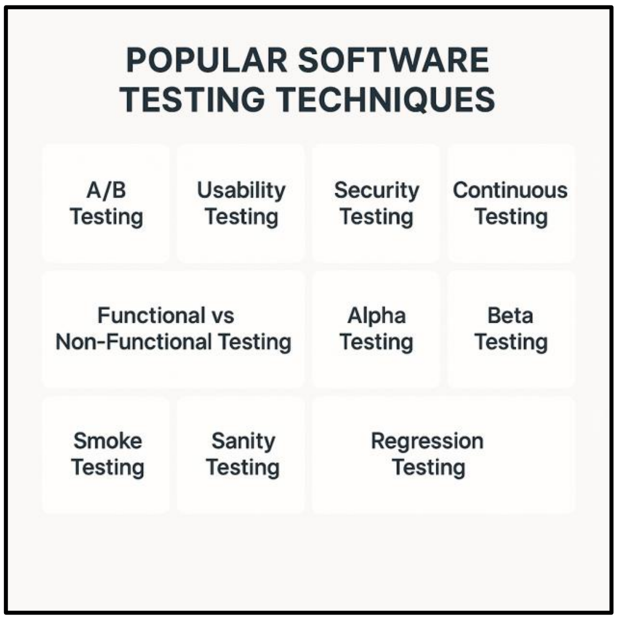
Agile testing is not a phase—it is an **ongoing activity**.

Advantages of Agile Testing

- **Faster Feedback**: Bugs are identified and resolved early.
- **Better Collaboration**: QA, devs, and business teams work together.
- **Flexibility**: Responds well to change with continuous integration.
- **Improved Product Quality**: Frequent testing ensures fewer production bugs.
- **Customer Satisfaction**: Early validation with stakeholders keeps requirements aligned.
- **Reduced Costs**: Early defect detection and less rework lower overall expenses.
- **Early Automation**: Promotes CI/CD and reduces manual effort in the long run.
- **Shift Left**: Testing is integrated into development from the start.

Popular Software Testing Techniques

Modern software development embraces a wide range of testing techniques to ensure functionality, usability, performance, and security across different environments. Here's a breakdown of the most widely used and effective techniques:



A/B Testing

Purpose**:** To compare two versions of a feature or interface (Version A and B) to determine which one performs better based on user interaction or conversion metrics.

Usability Testing

Purpose: To evaluate the software's user interface and experience by testing with real users.

Security Testing

Purpose: To identify vulnerabilities, risks, and threats in the application and ensure data protection.

Continuous Testing

Purpose: To test software continuously at every stage of development within the CI/CD pipeline.

Functional vs Non-Functional Testing

Type	**Description**	**Examples**
Functional	Validates features against requirements	Login, search, checkout
Non-Functional	Checks performance, scalability, and usability	Load, stress, security, UX

Alpha Testing

Purpose: Performed by internal employees/testers at the end of development to identify major bugs before external release.

Beta Testing

Purpose: Real users test the product in a real environment before the final release.

Regression Testing

Purpose: To ensure that new changes haven’t broken existing functionality.

Smoke Testing

Purpose: To perform basic checks to ensure the system is stable enough for further testing.

Sanity Testing

Purpose: Quick testing of specific functionalities after changes or fixes to confirm they work as expected. *Difference from Smoke Testing: Smoke is broad & shallow; sanity is narrow & deep.*

16. Automation Testing Frameworks

Automation testing frameworks provide structured ways to write, manage, and execute automated test cases efficiently. A well-designed framework improves test accuracy, reusability, and reduces maintenance overhead. An **automation testing framework** is a set of rules, guidelines, and tools used to create and manage test scripts systematically. It ensures **consistency, modularity**, and **efficiency** in test automation projects.

Types of Automation Testing Frameworks: -

1. Linear Scripting Framework
2. Modular Testing Framework
3. Data-Driven Framework
4. Keyword-Driven Framework
5. Hybrid Testing Framework
6. Behaviour-Driven Development (BDD) Framework

Planning and Preparation

Software Testing Simulation Project

For this academic simulation project, planning is essential to demonstrate not just testing theory but its practical application using real tools and frameworks. This phase defines the scope, tools, timeline, and expected outcomes.

Project Overview

Title: *Mini Software Testing Simulation using JUnit & Selenium*

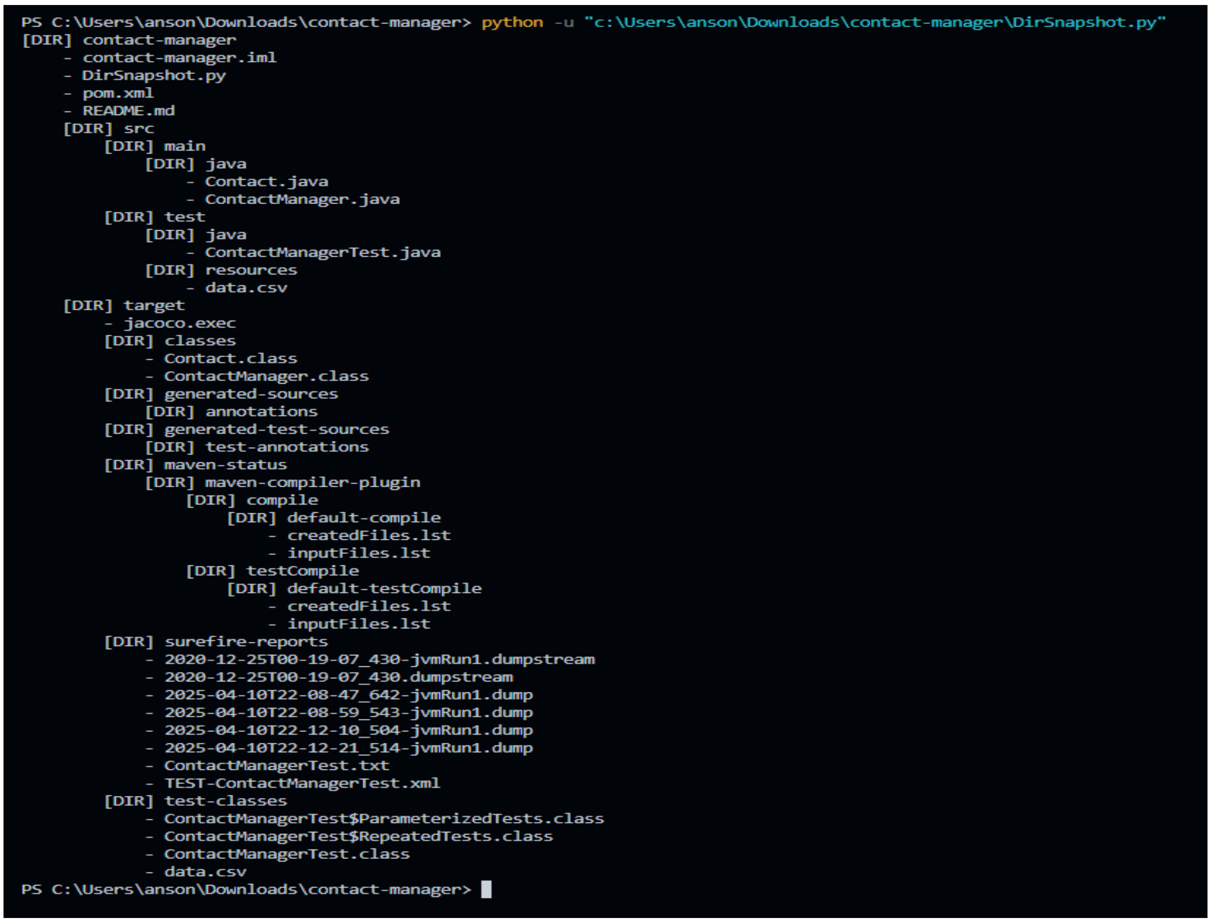
Objective: To simulate software testing in a real-world environment using automation tools, covering test planning, execution, and reporting.

Key Goals

- **Apply concepts of Test-Driven Development (TDD) and Automation Testing**
- **Use JUnit5 for unit testing**
- **Use Selenium for browser-based UI testing**
- **Integrate with Maven and optionally Jenkins**
- **Create basic reports from test runs**

Tools & Tech Stack

Tool/Technology	**Purpose**
Java (JDK 17)	**Main programming language**
JUnit5	**Unit testing framework**
Selenium WebDriver	**UI testing automation**
Maven	**Build and dependency manager**
Eclipse / IntelliJ IDEA	**IDE for writing and running tests**
Jenkins (optional)	**CI/CD and automation**



```
PS C:\Users\anson\Downloads\contact-manager> python -u "c:\Users\anson\Downloads\contact-manager\DirSnapshot.py"
[DIR] contact-manager
    - contact-manager.iml
    - DirSnapshot.py
    - pom.xml
    - README.md
    [DIR] src
        [DIR] main
            [DIR] java
                - Contact.java
                - ContactManager.java
        [DIR] test
            [DIR] java
                - ContactManagerTest.java
            [DIR] resources
                - data.csv
    [DIR] target
        - jacoco.exec
        [DIR] classes
            - Contact.class
            - ContactManager.class
        [DIR] generated-sources
            [DIR] annotations
        [DIR] generated-test-sources
            [DIR] test-annotations
        [DIR] maven-status
            [DIR] maven-compiler-plugin
                [DIR] compile
                    [DIR] default-compile
                        - createdFiles.lst
                        - inputFiles.lst
                [DIR] testCompile
                    [DIR] default-testCompile
                        - createdFiles.lst
                        - inputFiles.lst
        [DIR] surefire-reports
            - 2020-12-25T00-19-07_430-jvmRun1.dumpstream
            - 2020-12-25T00-19-07_430.dumpstream
            - 2025-04-10T22-08-47_642-jvmRun1.dump
            - 2025-04-10T22-08-59_543-jvmRun1.dump
            - 2025-04-10T22-12-10_504-jvmRun1.dump
            - 2025-04-10T22-12-21_514-jvmRun1.dump
            - ContactManagerTest.txt
            - TEST-ContactManagerTest.xml
        [DIR] test-classes
            - ContactManagerTest$ParameterizedTests.class
            - ContactManagerTest$RepeatedTests.class
            - ContactManagerTest.class
            - data.csv
PS C:\Users\anson\Downloads\contact-manager> 
```

Phases of Simulation

Phase	Description
Requirement Analysis	Define what the app should do
Test Case Design	Write unit test cases using JUnit5
Development	Implement the app in Java
Test Execution	Run and validate all test cases
Automation Integration	Add Selenium and optionally Jenkins
Documentation	Include test results and summary report

Key Components of the Identified Skill Sets

In the domain of **Software Testing**, especially within the **AI and Agile ecosystem**, mastering certain core skills is critical for both manual and automated testing roles. These skill sets are not just technical—they also include analytical, organizational, and collaborative abilities.

Technical Skills

Skill	**Description**
Test Case Design	Ability to design robust test cases covering functional and non-functional aspects.
Programming Proficiency	Writing and understanding code in languages like **Java**, **Python**, **JavaScript**, etc., is essential for writing unit and automation tests.
Test Automation Tools	Hands-on experience with tools like **JUnit, Selenium, Appium, TestNG**, and **Cucumber**.
Version Control Systems	Knowledge of **Git** for managing code and test versions.
CI/CD Integration	Familiarity with tools like **Jenkins**, **Travis CI**, and **GitHub Actions** for automated test execution.
Bug Tracking Tools	Experience with tools such as **Jira, Bugzilla, Mantis** to report and manage defects.
IDE and Build Tools	Working with **Eclipse, IntelliJ, Maven, Gradle** for test execution and project management.

Analytical and Problem-Solving Skills

- **Root Cause Analysis (RCA)**
- **Critical Thinking**
- **Scenario Modelling**

Process and Methodology Understanding

- Understanding **SDLC** and **STLC** phases.
- Knowledge of **Agile**, **Scrum**, and **DevOps** workflows.
- Experience with **Test-Driven Development (TDD)** and **Behavior-Driven Development (BDD)** practices.

Communication and Collaboration Skills

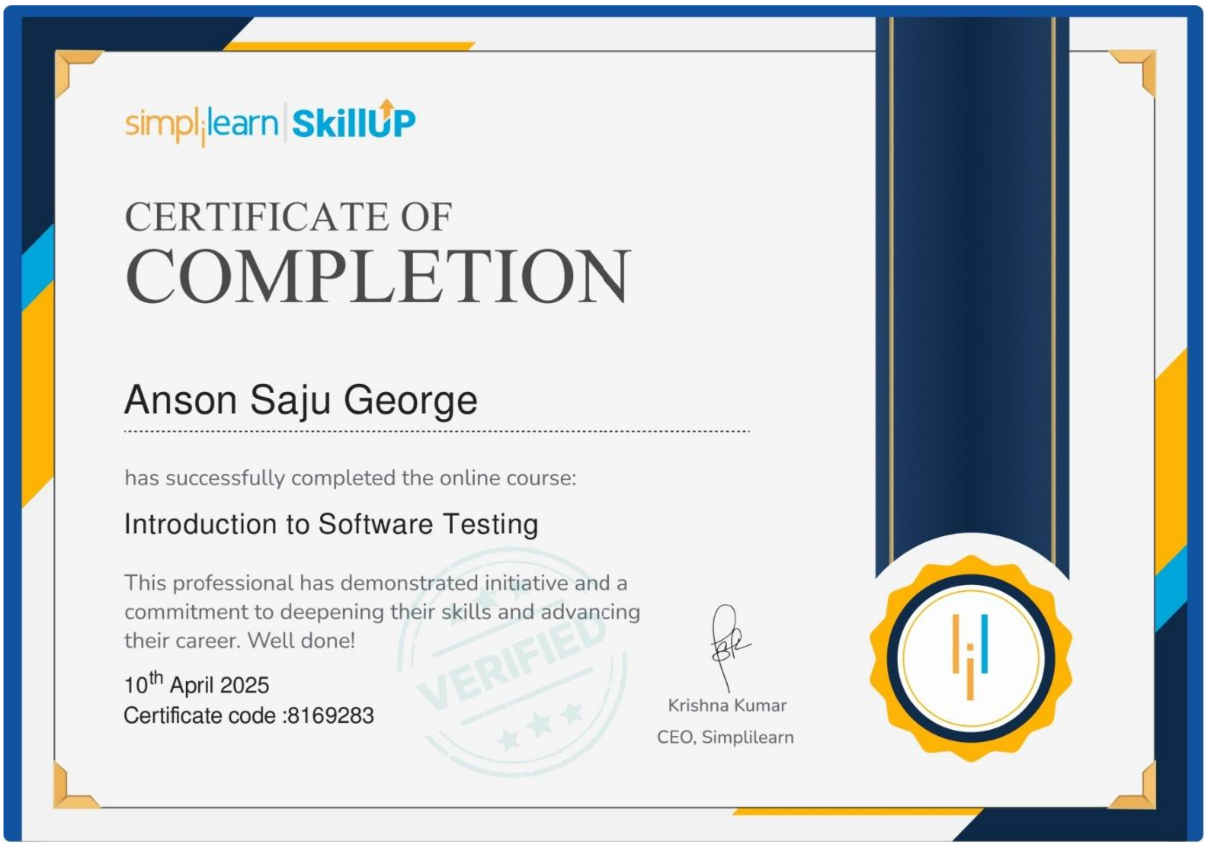
- Clear Bug Reporting
- Team Communication

- Documentation

Adaptability & Continuous Learning

- Willingness to **learn new tools**, languages, and testing frameworks.
- Keeping up with **trends in AI testing**, cloud-based test platforms, and new automation practices.

Proofs of outcome and feedback



simplilearn | SkillUP

CERTIFICATE OF
COMPLETION

Anson Saju George

has successfully completed the online course:

Introduction to Software Testing

This professional has demonstrated initiative and a commitment to deepening their skills and advancing their career. Well done!

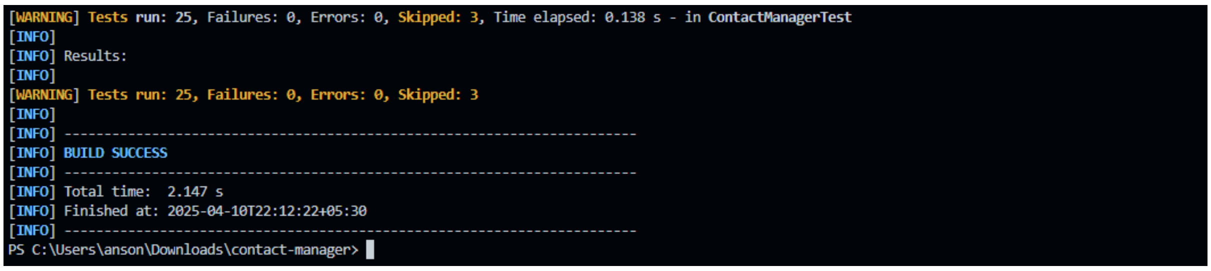
10th April 2025
Certificate code :8169283

VERIFIED

[illegible]

Krishna Kumar
CEO, Simplilearn

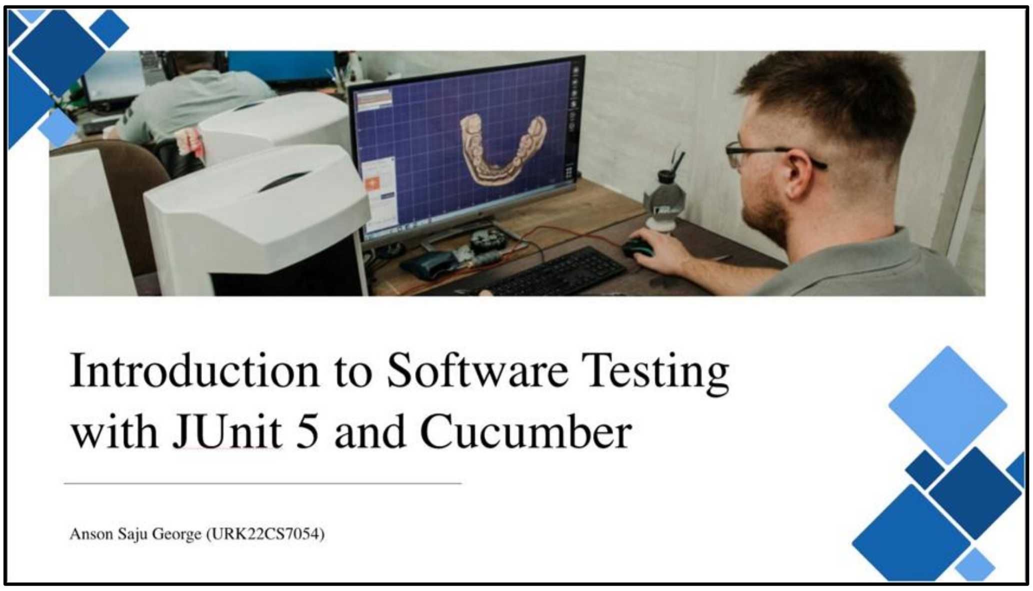
Result



```
[WARNING] Tests run: 25, Failures: 0, Errors: 0, Skipped: 3, Time elapsed: 0.138 s - in ContactManagerTest
[INFO]
[INFO] Results:
[INFO]
[WARNING] Tests run: 25, Failures: 0, Errors: 0, Skipped: 3
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.147 s
[INFO] Finished at: 2025-04-10T22:12:22+05:30
[INFO] ------------------------------------------------------------------------
PS C:\Users\anson\Downloads\contact-manager>
```

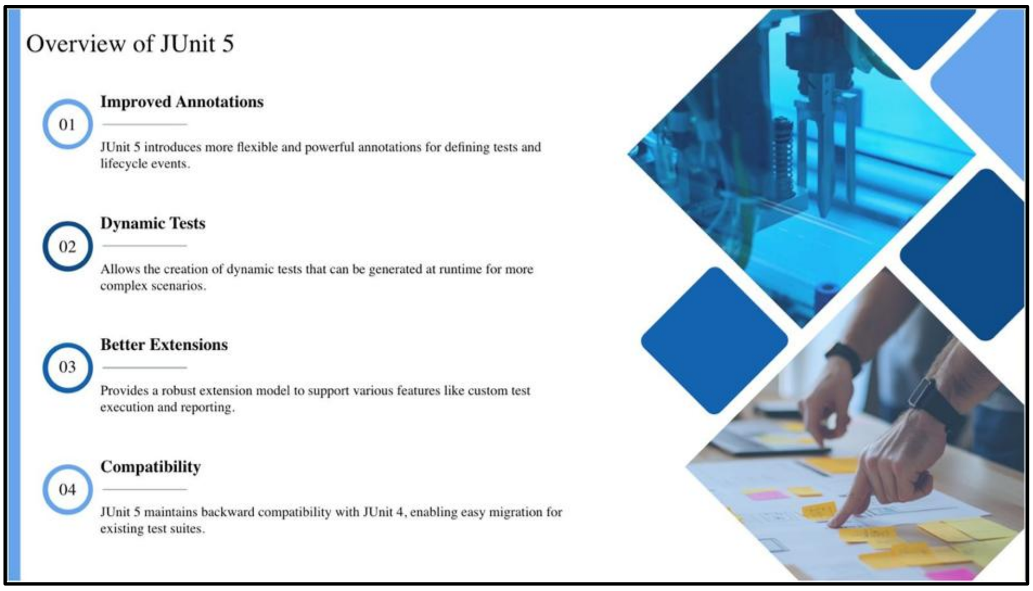
The JUnit 5 test suite for the Contact Manager application was successfully executed using Apache Maven in a local development environment. All unit tests, including standard, repeated, and parameterized tests, passed without any failures or errors. Tests that were conditionally disabled—such as OS-specific or explicitly marked disabled tests—were appropriately skipped. The successful execution confirmed the correctness and reliability of the contact creation logic, validating both expected inputs and exceptional cases. The test logs and Maven build report further verified the stability and test coverage of the project.

PowerPoint Presentation



Introduction to Software Testing with JUnit 5 and Cucumber

Anson Saju George (URK22CS7054)



Overview of JUnit 5

01 **Improved Annotations**

JUnit 5 introduces more flexible and powerful annotations for defining tests and lifecycle events.

02 **Dynamic Tests**

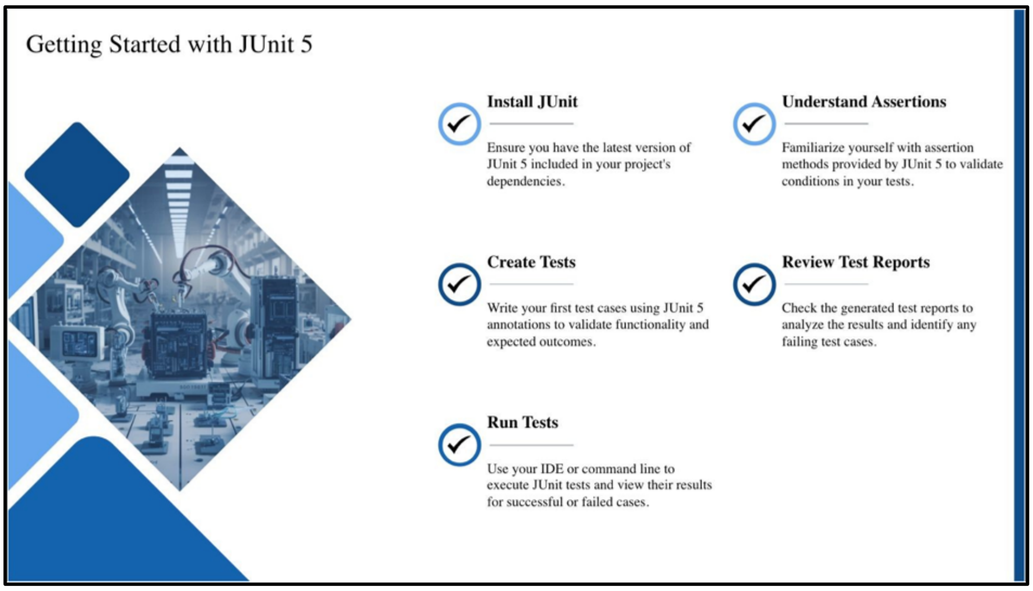
Allows the creation of dynamic tests that can be generated at runtime for more complex scenarios.

03 **Better Extensions**

Provides a robust extension model to support various features like custom test execution and reporting.

04 **Compatibility**

JUnit 5 maintains backward compatibility with JUnit 4, enabling easy migration for existing test suites.



Getting Started with JUnit 5

Install JUnit

Ensure you have the latest version of JUnit 5 included in your project's dependencies.

Understand Assertions

Familiarize yourself with assertion methods provided by JUnit 5 to validate conditions in your tests.

Create Tests

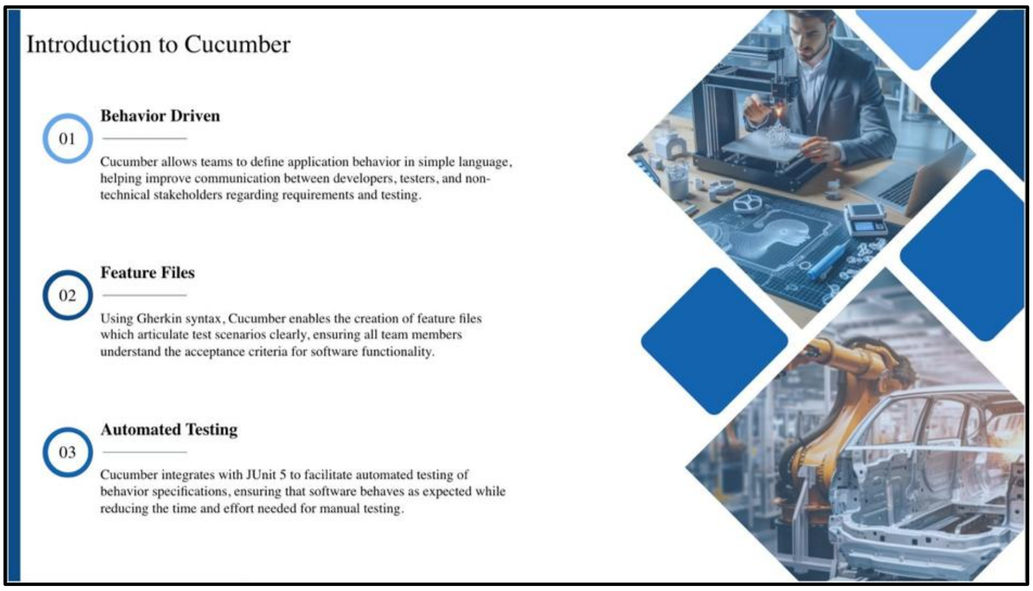
Write your first test cases using JUnit 5 annotations to validate functionality and expected outcomes.

Review Test Reports

Check the generated test reports to analyze the results and identify any failing test cases.

Run Tests

Use your IDE or command line to execute JUnit tests and view their results for successful or failed cases.



Introduction to Cucumber

01 **Behavior Driven**

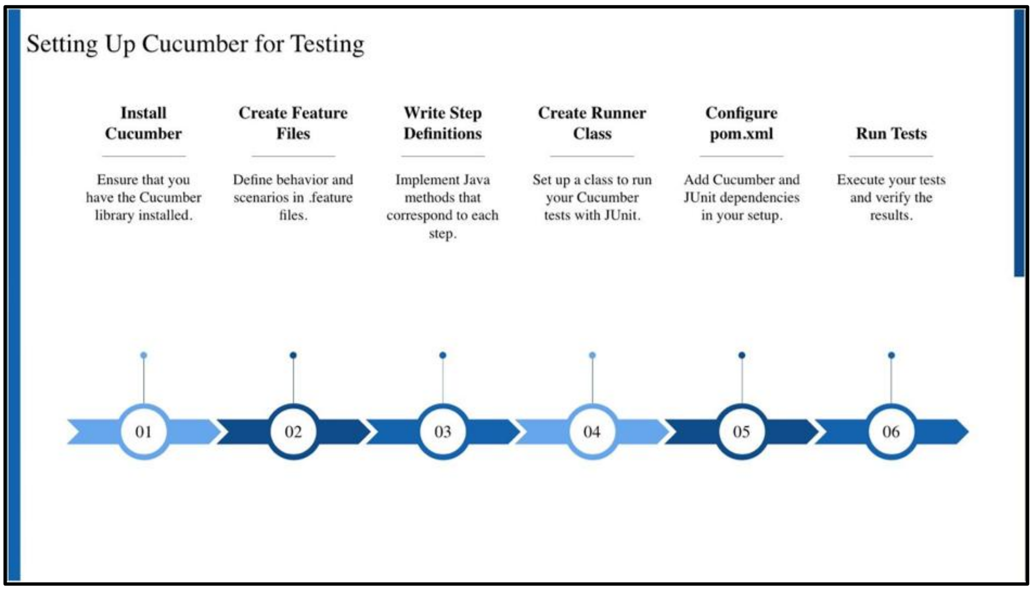
Cucumber allows teams to define application behavior in simple language, helping improve communication between developers, testers, and non-technical stakeholders regarding requirements and testing.

02 **Feature Files**

Using Gherkin syntax, Cucumber enables the creation of feature files which articulate test scenarios clearly, ensuring all team members understand the acceptance criteria for software functionality.

03 **Automated Testing**

Cucumber integrates with JUnit 5 to facilitate automated testing of behavior specifications, ensuring that software behaves as expected while reducing the time and effort needed for manual testing.



Setting Up Cucumber for Testing

01 **Install Cucumber**

Ensure that you have the Cucumber library installed.

02 **Create Feature Files**

Define behavior and scenarios in .feature files.

03 **Write Step Definitions**

Implement Java methods that correspond to each step.

04 **Create Runner Class**

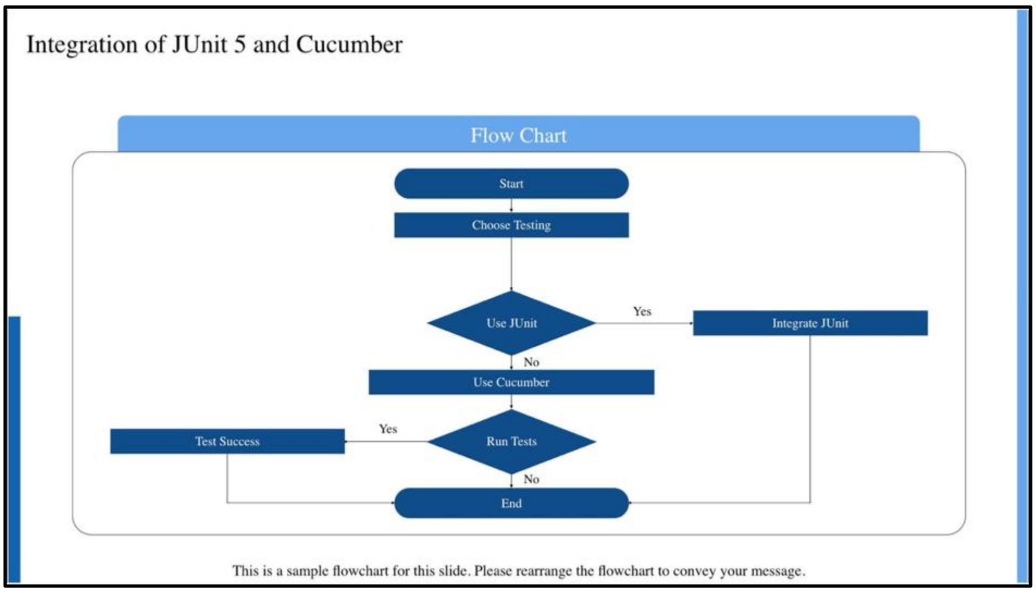
Set up a class to run your Cucumber tests with JUnit.

05 **Configure pom.xml**

Add Cucumber and JUnit dependencies in your setup.

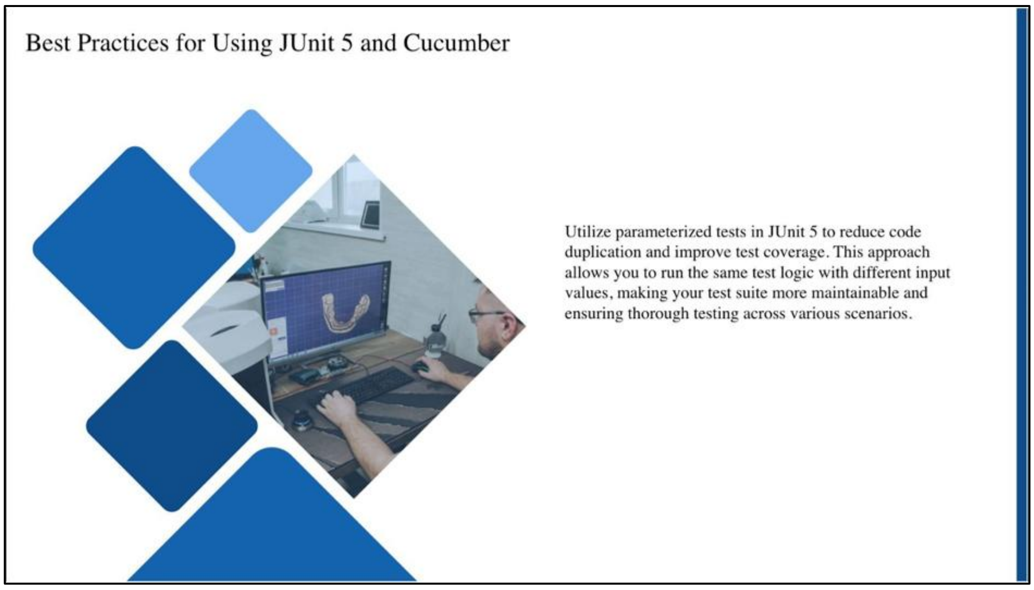
06 **Run Tests**

Execute your tests and verify the results.



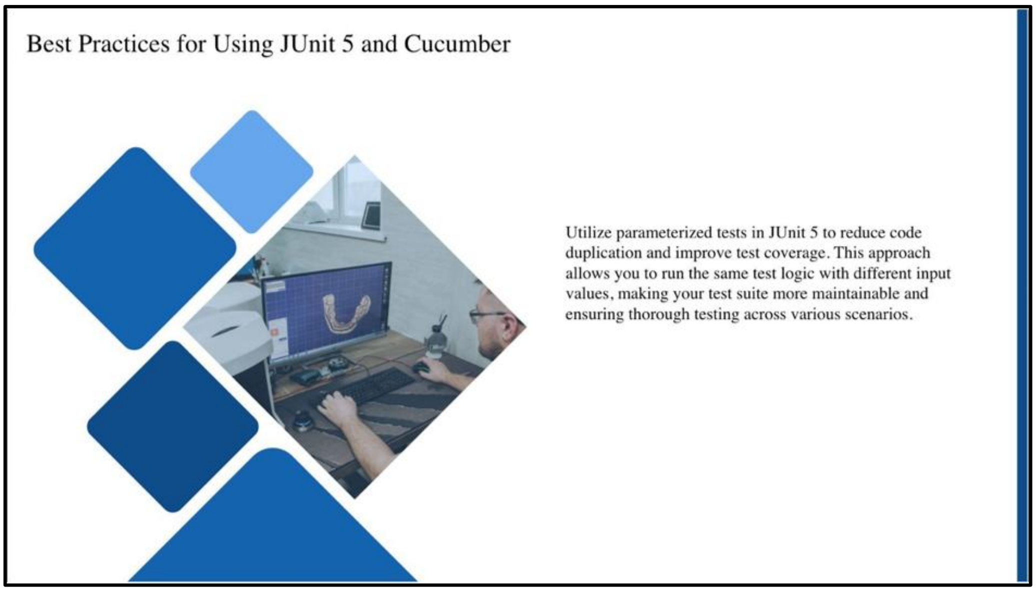
Integration of JUnit 5 and Cucumber

This is a sample flowchart for this slide. Please rearrange the flowchart to convey your message.



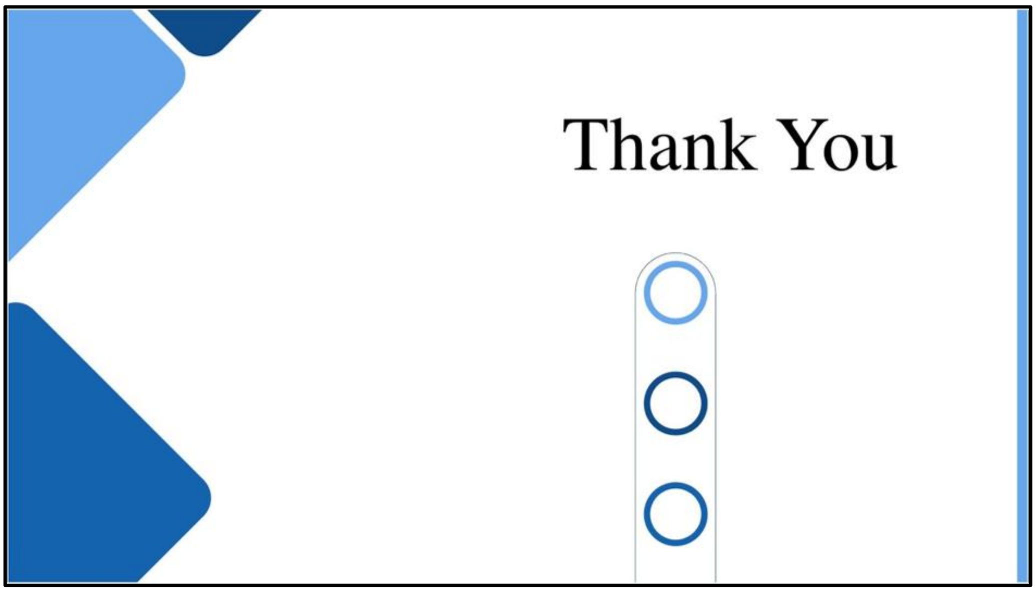
Best Practices for Using JUnit 5 and Cucumber

Utilize parameterized tests in JUnit 5 to reduce code duplication and improve test coverage. This approach allows you to run the same test logic with different input values, making your test suite more maintainable and ensuring thorough testing across various scenarios.



Best Practices for Using JUnit 5 and Cucumber

Utilize parameterized tests in JUnit 5 to reduce code duplication and improve test coverage. This approach allows you to run the same test logic with different input values, making your test suite more maintainable and ensuring thorough testing across various scenarios.



Thank You