Continuous Integration Using Jenkins and GitHub

Aim:

To implement Continuous Integration (CI) using Jenkins for compiling (build), testing (JUnit), and running Java applications developed using Notepad, and pushing code through GitHub.

Software Requirements:

- Java JDK (version 8 or higher)
- Jenkins (version 2.516.1 LTS)
- Git
- Notepad (for coding)
- Web Browser (Chrome/Firefox)
- JUnit 4.13.2 JAR
- Hamcrest-core 1.3 JAR

Hardware Requirements:

- Processor: Intel i3 or above
- RAM: 4 GB or more
- HDD: 20 GB Free Space

Introduction to CI/CD and Jenkins:

Continuous Integration (CI) automates the building, testing, and deployment of software projects to ensure early bug detection and integration issues. Jenkins is a leading open-source CI/CD tool that fetches code from source repositories like GitHub and performs automated tasks like compiling and testing. JUnit is a unit testing framework for Java used to validate functionality via test cases.

Procedure:

1. Create Project Structure:

```
ArithmeticProject/
├── Calculator.java
├── CalculatorTest.java
├── lib/
    ├── junit-4.13.2.jar
    └── hamcrest-core-1.3.jar
```

2. Write Java Code in Notepad:

Calculator.java:

```
public class Calculator {
    public int add(int a, int b) { return a + b; }
    public int subtract(int a, int b) { return a - b; }
    public int multiply(int a, int b) { return a * b; }
    public int divide(int a, int b) { if (b == 0) throw new ArithmeticException("Cannot divide by zero"); return a / b; }
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Addition: " + calc.add(10, 5));
        System.out.println("Subtraction: " + calc.subtract(10, 5));
        System.out.println("Multiplication: " + calc.multiply(10, 5));
        System.out.println("Division: " + calc.divide(10, 5));
    }}
```

3. Write JUnit Test Case:

CalculatorTest.java:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class CalculatorTest {
    Calculator calc = new Calculator();
    @Test public void testAddition() { assertEquals(15, calc.add(10, 5)); }
    @Test public void testSubtraction() { assertEquals(5, calc.subtract(10, 5)); }
    @Test public void testMultiplication() { assertEquals(50, calc.multiply(10, 5)); }
    @Test public void testDivision() { assertEquals(2, calc.divide(10, 5)); }}
```

4. Download and place JARs in lib/ folder:

- junit-4.13.2.jar
- hamcrest-core-1.3.jar

5. Push Project to GitHub:

```
git init
git add .
git commit -m "Initial Commit"
git remote add origin <GitHub Repo URL>
git push -u origin main
```

Note:-

If your GitHub repository uses a different default branch (e.g., "master" instead of "main"), update the branch name in Jenkins job configuration accordingly.

6. Install Jenkins:

Download from https://www.jenkins.io → Jenkins 2.516.1 LTS for Windows

Run: java -jar jenkins.war

Go to http://localhost:8080 and configure Jenkins

7. Configure Git in Jenkins:

Manage Jenkins → Global Tool Configuration → Git → Add path to git.exe

8. Create Jenkins Job:

New Item → Freestyle Project → Set GitHub Repo → Add Build Step:

Windows batch command:

```
mkdir out
javac -cp ".;lib\junit-4.13.2.jar;lib\hamcrest-core-1.3.jar" -d out Calculator.java
CalculatorTest.java
cd out
java Calculator
java -cp ".;..\lib\junit-4.13.2.jar;..\lib\hamcrest-core-1.3.jar" org.junit.runner.JUnitCore
CalculatorTest
```

9. Build & Monitor:

Click Build Now and monitor console output for results.

Expected Output:

Console output should show the following:

```
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2
JUnit version 4.13.2
....
```

```
Time: 0.002
OK (4 tests)
```

Result:

Successfully created a Jenkins job to compile, test, and run a Java program using Jenkins and GitHub, demonstrating all three phases of Continuous Integration.

Inference:

The Jenkins CI pipeline automated the build, test, and execution processes effectively. The setup ensures accurate, consistent, and reliable integration of code changes through GitHub.

Version Control Operations with Git and Mercurial

Aim:

To perform version control operations on a sample software project using two version control systems: Git and Mercurial. The experiment demonstrates tracking file changes, reverting updates, collaborating locally, and pushing to GitHub for remote version control.

Software Requirements:

- Git (latest stable version)
- Mercurial (latest stable version)
- hg-git plugin (for Mercurial-GitHub integration)
- GitHub account (for remote repository)
- Notepad (for editing files)
- Web Browser: Chrome or Firefox

Hardware Requirements:

- Processor: Intel i3 or higher
- RAM: 4 GB or more
- Hard Disk: 20 GB free space

Introduction

Version control systems (VCS) are essential tools for software development. They help manage and track changes made to source code or project files.

Git and Mercurial are both distributed version control systems, meaning each developer has a full local copy of the repository. They allow:

• Saving different versions of files

• Undoing mistakes and reverting changes

• Viewing history and differences

• Isolating features or experiments with branches

• Collaborating either locally or remotely (e.g., GitHub)

Procedure:

1. Git Version Control

A. Local Versioning using Git

1. Create a folder and initialize repository:
```
mkdir git-example && cd git-example
git init
```

2. Create a file index.html with initial content:
```
<!DOCTYPE html>
<html>
<head>
   <title>My Git Page</title>
</head>
<body>
   <h1>Welcome to Git</h1>
</body>
</html>
```
3. Check status of repository:
```
git status
```
4. Track the file:
```
git add index.html
```
5. Commit the file:
```
git commit -m "Initial commit with basic HTML page"
```
6. Edit the file (Example 1: Update heading):
```
<h1>Welcome to Git Version Control</h1>
```
Commands:
```
git add index.html
git commit -m "Updated heading in index.html"
```
7. Edit the file (Example 2: Add paragraph):
```
<p>This is a demo of Git version control.</p>
```
Commands:
```
git add index.html
git commit -m "Added description paragraph"
```
8. View commit history:
```
git log
git log --oneline --graph --decorate
```
9. View file differences:
```
git diff
```

10. Revert changes:
    - Discard unstaged edits:
```
git checkout -- index.html
```
    - Reset to previous commit:
```
git reset --hard HEAD~1
```
11. Create a new branch:
```
git branch new-feature
git checkout new-feature
```
    (Example Edit: <h2>New Feature Section</h2>)
12. Merge back to main:
```
git checkout main
git merge new-feature
```
13. Collaborate via shared folders:
```
git clone /path/to/git-example
```

B. Remote Versioning using Git (GitHub)

1. Add remote repository:
```
git remote add origin https://github.com/your-username/git-example.git
git branch -M main
git push -u origin main
```
2. Pull changes from remote:
```
git pull origin main
```

2. Mercurial Version Control

A. Local Versioning using Mercurial

14. Create a folder and initialize repository:
```
mkdir hg-example && cd hg-example
hg init
```
15. Create a file index.html with initial content:
```
<!DOCTYPE html>
<html>
<head>
    <title>My Hg Page</title>
</head>
```

```
<body>
   <h1>Hello Mercurial</h1>
</body>
</html>
```

16. Track and commit the file:

```
hg add index.html
hg commit -m "Initial commit with HTML page"
```

17. Edit file (Example 1: Update heading):

```
<h1>Hello Mercurial Version Control</h1>
```

Commands:

```
hg add index.html
hg commit -m "Updated heading in index.html"
```

18. Edit file (Example 2: Add paragraph):

```
<p>This is a demo of Mercurial version control.</p>
```

Commands:

```
hg add index.html
hg commit -m "Added description paragraph"
```

19. View log and status:

```
hg log
hg log -G
hg status
```

20. Revert file to last committed state:

```
hg revert index.html
```

21. Create a named branch:

```
hg branch new-feature
```

(Example Edit: <h2>New Feature in Hg</h2>)

22. Merge changes back:

```
hg update default
hg merge
hg commit -m "Merged new-feature branch into default"
```

23. Collaborate via shared folders:

```
hg clone /path/to/hg-example
```

B. Remote Versioning using Mercurial (GitHub via hg-git)

1. Configure hg-git in hgrc file
2. Add GitHub-compatible remote:

```
hg bookmark -r default master
hg push git+https://github.com/your-username/hg-example.git
```

3. Pull changes from remote:

```
hg pull
hg update
```

Result

Version control operations were successfully performed using Git and Mercurial. The following features were verified for both tools:

• Tracking changes and file history
• Viewing repository status and differences
• Reverting to earlier versions and discarding changes
• Branching for feature isolation and merging
• Local collaboration and cloning
• Remote backup, push, and pull operations using GitHub

Inference:

Version control operations were successfully performed using Git and Mercurial. Both tools enabled effective tracking of changes, branching, merging, and collaboration, ensuring reliable local and remote version management.

Docker and WSL: Virtualization and Container Management via CMD

PHASE 1: To Install and Configure Docker Desktop and WSL manually, and create containers of different Operating System images

Aim:

To install and configure Docker Desktop using WSL2 through **Command Prompt (CMD)** and to create containers for different OS images using Docker.

Software Requirements:

- Windows 10/11 (64-bit)
- CMD (Command Prompt)
- Docker Desktop Installer
- WSL2

Hardware Requirements:

- Processor: Intel i3 or above
- RAM: 4 GB or more
- HDD: 20 GB Free Space

Introduction:

Docker Desktop is an application that provides a user-friendly environment to build, run, and manage containerized applications on Windows or macOS. It uses container technology to package software and its dependencies into isolated units, ensuring consistent behavior across different environments.

WSL (Windows Subsystem for Linux) is a compatibility layer that allows Linux distributions to run natively on Windows. WSL2, the latest version, uses a lightweight virtual machine to provide a full Linux kernel, enabling tools like Docker to run efficiently on Windows. Together, Docker Desktop with WSL2 allows developers to create, manage, and run containers seamlessly through Windows while leveraging Linux-based environments.

Procedure:

Step 1: Check if WSL is Installed

Open **CMD** and run:

```
wsl --list --verbose
```

If WSL is not installed, install WSL and a Linux distribution.

Step 2: View Available Linux Distributions

```
wsl --list –online
```

Step 3: Install a Linux Distribution (e.g., Ubuntu)

```
wsl --install -d Ubuntu
```

This will install Ubuntu using WSL2.

Note: Restart your system if prompted.

Step 4: Install Docker Desktop

- Download from: https://www.docker.com/products/docker-desktop
- Run the installer and use **default settings**.
- Ensure the **"Use WSL 2 instead of Hyper-V"** option is checked.

Step 5: Verify Docker Installation via CMD

```
docker --version
docker info
```

You should see Docker running using the WSL 2 backend.

Create Containers for Different Operating Systems

Ubuntu Container

```
docker pull ubuntu
docker run -it ubuntu
```

Alpine Container

```
docker pull alpine
docker run -it alpine
```

Debian Container

```
docker pull debian
docker run -it debian
```

Type exit inside the container to return to CMD

Result:

Docker Desktop was successfully installed and configured using WSL2. Containers for Ubuntu, Alpine, and Debian operating systems were created and accessed through CMD.

Phase 2: To Build, Deploy, and Manage a Java Application using Docker via Ubuntu Terminal (WSL2)

Aim:

To build, deploy, and manage a basic Java application using Docker containers through Ubuntu terminal in WSL2.

Software Requirements:

- Ubuntu Terminal (Docker Ubuntu container)
- JDK Installed (Java Compiler)
- Docker Desktop with WSL2 backend

Hardware Requirements:

- Processor: Intel i3 or above
- RAM: 4 GB or more
- HDD: 20 GB Free Space

Procedure:

Step 1: Create a Java Application Directory and Source File

```
mkdir JavaApp
cd JavaApp
nano HelloWorld.java
```

Note:

If the text editor nano is not available in your Ubuntu container or WSL environment, you can install it by running:

```
apt update
apt install -y nano
```

Paste the following Java code in the nano editor:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello from Dockerized Java App!");
    }
}
```

- Save and exit nano: Press Ctrl + O, Enter to save, then Ctrl + X to exit.

Step 2: Install JDK (if not already installed)

```
apt update
apt install -y openjdk-17-jdk
```

Step 3: Compile and Package the Java Application

```
javac HelloWorld.java
jar cfe HelloWorld.jar HelloWorld HelloWorld.class
```

Step 4: Create a Dockerfile

```
nano Dockerfile
```

Paste the following content inside the Dockerfile:

```
FROM openjdk:17-jdk-slim
COPY HelloWorld.jar /app/HelloWorld.jar
WORKDIR /app
CMD ["java", "-jar", "HelloWorld.jar"]
```

- Save and exit nano (Ctrl + O, Enter, Ctrl + X).

Step 4.1: Exit from the Ubuntu Docker Container

Once you've compiled your Java program and created the JAR inside the container, follow these steps to continue the Docker build on your **host machine**:

Step 4.1.1: Get the Container ID or Name

Run the following command in **CMD**:

```
docker ps -a
```

This lists all containers (running and stopped). You'll see something like:

```
CONTAINER ID   IMAGE     COMMAND       ...    NAMES
8098fc408759   ubuntu    "/bin/bash"   ...    java-ubuntu-container
```

Step 4.1.2: Copy Files from Container to Host

To build the Docker image on your host, you need the compiled files (like .jar) and the Dockerfile.

Use this command to copy them:

```
docker cp <container_id>:/JP/HW.jar .
docker cp <container_id>:/JP/Dockerfile .
```

Example:

```
docker cp 8098fc408759:/JP/HW.jar .
docker cp 8098fc408759:/JP/Dockerfile .
```

Note:

Be sure to replace 8098fc408759 with your actual container ID or name if it's different. Make sure the file paths (/JP/HW.jar and /JP/Dockerfile) are correct and the files exist in the container.

You must run these commands **from your host system (CMD)**, not inside the container.

Step 5: Build the Docker Image

```
docker build -t java-hello-world .
```

Step 6: Run the Docker Container

```
docker run --name my-java-app java-hello-world
```

Expected Output:

```
Hello from Dockerized Java App!
```

Result:

A Java application was successfully compiled and containerized using Docker. It was then run inside a Docker container through the Ubuntu terminal (WSL2) using Docker Desktop with the WSL2 backend.

Inference:

Virtualization and containerization were successfully implemented using Docker and WSL2. In **Phase 1**, Docker was configured with WSL2 to create and manage an Ubuntu container. In **Phase 2**, a Java application was built and executed inside the Dockerized Ubuntu environment, showing how containerization simplifies deployment and ensures consistency across environments.

Automating Linux Environment Provisioning with Ansible and Docker

Aim:

To install and configure Software Configuration Management (SCM) and perform provisioning using Ansible on Linux environments created with Docker.

Software Requirements:

- Docker installed on host system
- Ansible installed on control node
- Python 3 installed inside Docker containers
- Terminal / Command Prompt

Hardware Requirements:

- Minimum 4 GB RAM
- 20 GB Disk

Introduction:

Ansible is an open-source automation tool used for Software Configuration Management (SCM), provisioning, and application deployment. By combining Ansible with Docker, multiple Linux environments can be quickly simulated as containers, allowing tasks like configuration, file management, and software installation to be automated without the need for physical machines. This setup provides a lightweight, efficient, and reproducible way to practice and test SCM concepts.

Procedure:

Step 1: Pull Ubuntu Docker Image

```
docker pull ubuntu:20.04
```

Step 2: Run Ubuntu Containers

Run two containers for managed nodes:

```
docker run -dit --name node1 ubuntu:20.04
docker run -dit --name node2 ubuntu:20.04
```

- -d → detached
- -i → interactive
- -t → allocate terminal

Step 3: Access Containers and Setup SSH & Python

Node1

```
docker exec -it node1 bash
```

```
apt update
apt install -y openssh-server python3 sudo
mkdir /var/run/sshd
service ssh start
exit
```

Node2

```
docker exec -it node2 bash
apt update
apt install -y openssh-server python3 sudo
mkdir /var/run/sshd
service ssh start
exit
```

Step 4: Setup Control Node (Ubuntu WSL or native Ubuntu)

```
sudo apt update
sudo apt install -y ansible ssh nano
ansible --version
```

- Expected output: ansible [core 2.15.x]

Step 5: Create Ansible Inventory File (Use Docker connection for WSL)

```
nano ~/hosts
```

Add the following content:

```
[managed_nodes]
node1 ansible_connection=docker
node2 ansible_connection=docker
```

Save and exit: Ctrl + O → Enter → Ctrl + X

Step 6: Test Ansible Connection

```
ansible -i ~/hosts all -m ping
```

- Expected output:

```
node1 | SUCCESS => { "ping": "pong" }
node2 | SUCCESS => { "ping": "pong" }
```

Step 7: Create Ansible Playbook

```
nano createfile.yml
```

Paste the following:

```
---
- name: Create a file on managed nodes
  hosts: managed_nodes
  become: yes
  tasks:
    - name: Create a test file
      copy:
        dest: /tmp/ansible_test.txt
        content: "Hello from Ansible Controller!"
```

Save and exit: Ctrl + O → Enter → Ctrl + X

Step 8: Run Playbook

```
ansible-playbook -i ~/hosts createfile.yml
```

Step 9: Verify Provisioning

Check the file in containers:

```
docker exec -it node1 cat /tmp/ansible_test.txt
docker exec -it node2 cat /tmp/ansible_test.txt
```

- Expected output:

```
Hello from Ansible Controller!
```

Result:

Ansible was successfully used to perform Software Configuration Management and provisioning on Docker-based Linux environments.

Inference:

Ansible efficiently handled the provisioning and configuration of Linux environments running in Docker containers. The playbook executed successfully on multiple simulated nodes, showing that containerized setups simplify Software Configuration Management tasks. This method offers a lightweight, reproducible, and effective way to manage and test Linux environments.

Understanding Agile: Background and Driving Forces

Aim:

To study and understand the background, motivations, and advantages of using Agile methodology in software development.

Objectives:

- To learn why Agile is used in software development.
- To identify the driving forces that encourage the adoption of Agile.
- To understand the differences between Agile and traditional approaches like Waterfall.
- To explore how Agile improves software quality and customer satisfaction.
- To analyze real-world examples of Agile implementation.

Theory:

Agile methodology emerged as a response to the limitations of traditional software development models such as the Waterfall model, which followed a sequential, rigid process. In traditional approaches, all requirements are gathered upfront, and each phase—design, implementation, testing, and deployment—must be completed before the next phase begins. This often led to problems such as delayed feedback, high costs of changes, low flexibility, and reduced customer satisfaction when requirements evolved during the project lifecycle.

Background of Agile:

In the late 1990s, software engineers and project managers recognized that software development needed to be more adaptive and customer-centric. This led to the creation of the **Agile Manifesto in 2001**, where 17 software practitioners outlined four core values and twelve principles that guide Agile development.

The four core values of Agile:

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

These values promote teamwork, flexibility, and delivering functional software frequently.

Driving Forces Behind Agile Adoption:

→ **Changing Customer Requirements:** Agile accommodates evolving needs through iterative development.

→ **Faster Time-to-Market:** Agile delivers small, working increments frequently for early release.

→ **Improved Team Collaboration:** Agile encourages close communication through daily stand-ups, sprint reviews, and retrospectives.

→ **Continuous Feedback and Improvement:** Regular review and adaptation ensure continuous product enhancement.

→ **Reduced Risk and Higher Quality:** Frequent testing and integration detect defects early.

→ **Enhanced Customer Satisfaction:** Early delivery of usable software builds trust and satisfaction.

Agile vs. Traditional Approaches:

Aspect	Traditional (Waterfall)	Agile
Development Flow	Linear and sequential	Iterative and incremental
Flexibility	Rigid and difficult to change	Highly adaptive
Customer Involvement	Limited	Continuous
Delivery	At project end	Short iterations
Feedback	Late	Frequent
Documentation	Heavy	Lightweight
Testing	After implementation	Continuous

Agile Frameworks:

- **Scrum:** Sprints with defined roles (Scrum Master, Product Owner, Development Team)
- **Kanban:** Visual workflow management and continuous delivery
- **Extreme Programming (XP):** Pair programming, continuous integration, test-driven development
- **Lean Software Development:** Focus on eliminating waste and maximizing value

Real-World Importance:

Agile aligns development closely with business goals and customer needs. It improves team morale, ensures transparency, and helps teams respond quickly to market changes.

Examples:

- **Spotify:** Uses Squads, Tribes, Chapters, and Guilds. Each squad acts like a mini-startup, releasing updates frequently and improving based on user feedback.
- **Scrum Project Example:** A chat application is developed in 2-week sprints. Features are added, tested, and improved after feedback in the next sprint.

Result:

After studying Agile methodology, it is observed that Agile helps achieve:

- Faster and more frequent software delivery
- Enhanced communication and collaboration among team members
- Flexibility in handling changing requirements
- Continuous improvement through regular feedback loops
- Increased customer satisfaction and product quality

Conclusion:

Agile methodology revolutionized software development by shifting the focus from rigid processes to flexibility, collaboration, and customer satisfaction. Its iterative nature allows teams to adapt quickly to changes and deliver high-value software in shorter cycles. Understanding the background and driving forces behind Agile helps software engineers and organizations implement it effectively to achieve efficient, high-quality, and customer-driven development outcomes.

Understanding the Value of Agile in Business

Aim:

To study and understand the business benefits of adopting Agile methodology in organizations.

Objectives:

- To identify the advantages of Agile for businesses.
- To understand how Agile improves efficiency, collaboration, and customer satisfaction.
- To recognize the reasons organizations adopt Agile over traditional methods.
- To analyze real-world examples of Agile adoption in businesses.
- To explore how Agile supports competitive advantage and risk reduction.

Theory:

Agile is a flexible, iterative approach to project management and software development that emphasizes delivering work in **small, manageable increments**. Unlike traditional models like Waterfall, which are linear and rigid, Agile allows organizations to **adapt quickly to change**, respond to customer needs, and continuously improve processes and products.

Business Value of Agile:

1. **FasterTime-to-Market:**
   Agile enables organizations to release products and features in short cycles, allowing faster delivery to customers. This reduces the time between idea conception and real-world implementation.
2. **AdaptabilitytoMarketChanges:**
   Businesses face dynamic market demands and shifting customer expectations. Agile allows teams to **pivot quickly** and make incremental changes without major delays or disruptions.
3. **RiskReduction:**
   By delivering small increments and obtaining frequent feedback, Agile helps detect and correct issues early. This minimizes project failures, cost overruns, and scope creep.
4. **ImprovedCollaborationandCommunication:**
   Agile promotes **cross-functional teamwork** through daily stand-ups, sprint planning, and review meetings. Better collaboration leads to higher productivity and more innovative solutions.
5. **EnhancedCustomerSatisfaction:**
   Continuous engagement with customers ensures that products meet their evolving needs.

Early delivery of functional features builds trust and strengthens long-term customer relationships.

6. **CostEfficiency:**
   By prioritizing high-value features, reducing rework, and minimizing wasted effort, Agile can lower operational costs while maximizing ROI.
7. **CompetitiveAdvantage:**
   Organizations that adopt Agile can respond faster to market opportunities, outperform competitors, and maintain relevance in fast-paced industries.
8. **ContinuousImprovement:**
   Agile encourages teams to reflect on processes regularly through retrospectives, fostering a culture of learning and ongoing improvement.

Examples:

- **Amazon:** Uses Agile to quickly launch new website features, test them, and adapt based on customer feedback. This ensures fast innovation and responsiveness to changing user needs.
- **Startup App Development:** A startup releases a minimum viable product (MVP) early, gathers user feedback, and iterates on the product continuously instead of waiting months to launch a full version. This approach reduces risk and accelerates product-market fit.
- **Spotify:** Agile squads work independently on features, enabling rapid deployment and quick adaptation to user preferences, maintaining high customer satisfaction.

Result:

After studying Agile, it is observed that businesses adopting Agile can:

- Deliver products faster and more frequently.
- Adapt efficiently to market changes and customer demands.
- Improve team collaboration, communication, and productivity.
- Increase customer satisfaction, loyalty, and trust.
- Reduce project risks, errors, and unnecessary costs.
- Maintain a competitive edge in rapidly evolving markets.

Conclusion:

Adopting Agile provides substantial business value by enabling organizations to respond quickly to change, deliver high-quality products efficiently, and foster continuous improvement. Understanding these benefits helps organizations make informed decisions and implement Agile practices effectively, leading to increased productivity, customer satisfaction, and long-term success.

Key Practices in Agile Development

Aim:

To study and understand the key practices followed in Agile software development.

Objectives:

- To learn the methods and routines used in Agile development.
- To understand how Agile practices improve efficiency, collaboration, and software quality.
- To identify the benefits of following Agile practices in software projects.
- To explore real-world applications of Agile practices.
- To analyze how Agile practices contribute to faster delivery and customer satisfaction.

Theory:

Agile development practices are structured routines, techniques, and processes that help software teams work efficiently, adapt to change, and deliver high-quality software. These practices ensure that teams can collaborate effectively, receive continuous feedback, and iterate rapidly on their work.

Key Agile practices include:

1. **IterativeDevelopment:**
   Work is divided into small increments called iterations or sprints (typically 1–4 weeks). Each increment produces a functional part of the software, which can be reviewed, tested, and improved in the next iteration.
2. **DailyStand-UpMeetings:**
   Short, time-boxed meetings (usually 15 minutes) where team members discuss progress, challenges, and plans for the day. Stand-ups promote transparency, alignment, and quick problem resolution.
3. **SprintPlanning:**
   Teams plan work for a sprint, breaking tasks into **user stories** with clear acceptance criteria. Responsibilities are assigned, priorities are set, and goals for the sprint are clearly defined.
4. **BacklogManagement:**
   The product backlog lists features, enhancements, and bug fixes. Agile teams continuously prioritize and refine backlog items to ensure that the most valuable work is done first.

5. **ContinuousIntegration(CI):**
   Developers frequently integrate code into a shared repository. Automated tests run on each integration to detect defects early, ensuring higher software quality and reducing integration issues.
6. **UserStoryPlanning:**
   Requirements are expressed as **user stories**, describing functionality from the end-user's perspective. This helps teams understand user needs and develop customer-focused solutions.
7. **Retrospectives:**
   At the end of each iteration, teams reflect on what went well, what could be improved, and actionable steps for the next iteration. Retrospectives drive continuous improvement and learning.
8. **PairProgrammingandTest-DrivenDevelopment(XP):**
   In practices like Extreme Programming, developers work in pairs and write automated tests before writing the actual code, improving code quality and collaboration.

Benefits of Agile Practices:

- Improved team collaboration and communication
- Frequent delivery of small, functional software increments
- Quick adaptation to changing requirements
- Enhanced software quality through continuous testing and feedback
- Greater customer satisfaction and faster feedback cycles
- Efficient project management and reduced risks

Examples:

- **DailyStand-Ups:**
  A software team meets every morning for 15 minutes to discuss progress, challenges, and the day's plan, ensuring alignment and transparency.
- **SprintPlanninginScrum:**
  A team plans a 2-week sprint to implement a new login feature, breaking work into user stories, assigning tasks, and setting sprint goals.
- **ContinuousIntegration:**
  Developers integrate code changes frequently into a shared repository. Automated tests run with each integration to catch errors early, maintaining software quality.

- **BacklogRefinement:**
  Product Owners update and prioritize the backlog regularly to ensure the team is working on the most important features.

Result:

After studying Agile development practices, it is observed that they:

- Enhance team collaboration and communication.
- Enable frequent delivery of functional software increments.
- Allow quick adaptation to changing requirements.
- Improve software quality through continuous testing and integration.
- Help teams deliver customer-focused solutions efficiently and effectively.

Conclusion:

Agile development practices provide structured methods for implementing Agile principles effectively. By following these practices, teams can work more efficiently, deliver higher-quality software, respond rapidly to changes, and maintain strong collaboration with stakeholders. Ultimately, Agile practices ensure that software development is flexible, iterative, and aligned with customer expectations.

Implementing Test-Driven Development (TDD) with Unit Testing

Aim:

To implement Test Driven Development (TDD) in Java by writing unit tests before coding and driving development through automated testing in VS Code.

Objectives:

- Understand the concept of Test Driven Development (TDD).
- Write unit tests before implementing functional code.
- Ensure code correctness by passing all test cases.
- Observe benefits of TDD such as fewer defects and better design.
- Improve software quality through continuous testing and feedback.

Tools/Software:

- **Programming Language:** Java
- **IDE:** Visual Studio Code
- **Unit Testing Framework:** JUnit 4 / 5
- **VS Code Extensions:**
    - ✓ Java Extension Pack
    - ✓ Test Runner for Java

Procedure / Steps in VS Code

Step 1: Install Required Tools

- Install Java JDK (version 8 or above).
    - ✓ Check using terminal:
    - ✓ java -version
    - ✓ javac -version
- Install Visual Studio Code.
- Install VS Code Extensions:
    - ✓ Java Extension Pack
    - ✓ Test Runner for Java

Step 2: Create Java Project

- Open VS Code → Press **Ctrl+Shift+P** → Java: Create Java Project.
- Select **No Build Tools** (or Maven if desired).
- Name project: TDDCalculator.

Folder structure will look like:

```
TDDCalculator/
 ├── src/
 │    └── main/java/Calculator.java
 │    └── test/java/CalculatorTest.java
```

Step 3: Red Phase (Write the Test First)

- Go to src/test/java/CalculatorTest.java.
- Write the test **before creating the class**:

```
import main.java.Calculator;
import static org.junit.Assert.*;
import org.junit.Test;
public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(5, result);
    }
}
```

- Run test → It **fails (RED Phase)** since Calculator class does not exist.

Step 4: Green Phase (Write Minimal Code to Pass Test)

- Go to src/main/java/Calculator.java.
- Implement only enough to pass the test:

```
package main.java;
public class Calculator {
    public int add(int a, int b) {
        return a + b;  // minimal code to pass the test
    }
}
```

- Run test → It **passes (GREEN Phase)**.

Step 5: Refactor Phase (Improve Code Quality)

- Refactor the code for better readability:

```
package main.java;
public class Calculator {
    // Refactored variable names for clarity
    public int add(int firstNumber, int secondNumber) {
        int sum = firstNumber + secondNumber;
        return sum;
    }
}
```

- Run test → It still passes (Refactor Phase).

Step 6: Repeat for Additional Functionality

- Example: Subtraction Test:

```
@Test
public void testSubtract() {
    Calculator calc = new Calculator();
    int result = calc.subtract(5, 3);  // Expect 2
    assertEquals(2, result);
}
```

- Implement minimal code to pass:

```
package main.java;
public class Calculator {
    public int add(int firstNumber, int secondNumber) {
        return firstNumber + secondNumber;
    }
    public int subtract(int firstNumber, int secondNumber) {
        return firstNumber - secondNumber;
    }
}
```

- Refactor if needed, always ensuring tests pass.
- Follow **Red → Green → Refactor** cycle for every new functionality.

Step 7: Handling Test Failures in TDD

1. **Check the Test Cases:** Ensure unit tests are written correctly.

2. **Analyze Your Code:** Look for logic errors, incorrect calculations, or missing functionality.
3. **Modify the Code:** Update/fix implementation to satisfy failing tests.
4. **Run Tests Again:** Repeat until all tests pass.
5. **Refactor if Needed:** Clean up code without breaking functionality.

Step 8: Running Tests in VS Code

- Open CalculatorTest.java.
- Hover over @Test → Click **Run Test**.
- OR use **Testing Panel** (sidebar) → Run all tests.
- Observe the **Red** → **Green** → **Refactor** cycle for each feature.

Program

Red Phase:

```
import main.java.Calculator;
import static org.junit.Assert.*;
import org.junit.Test;
public class CalculatorTest {
   @Test
   public void testAdd() {
      Calculator calc = new Calculator();
      int result = calc.add(2, 3);
      assertEquals(5, result);
   }
}
```

- Fails initially because class/method don’t exist.

Green Phase:

```
package main.java;
public class Calculator {
   public int add(int a, int b) {
      return a + b;  // minimal code to pass the test
   }
}
```

- Test passes.

Refactor Phase:

```
package main.java;
public class Calculator {
    // Refactored for readability
    public int add(int firstNumber, int secondNumber) {
        int sum = firstNumber + secondNumber;
        return sum;
    }
}
```

- Test still passes.

Observations / Results

- Initial test failed (Red Phase) → confirmed requirement not implemented.
- Minimal code (Green Phase) passed test → functionality verified.
- Refactored code improved readability and maintainability while keeping tests passing.
- Benefits observed: fewer defects, clearer design, and improved understanding of requirements.

Conclusion:

The experiment successfully demonstrated the Test Driven Development (TDD) approach in Java using JUnit within Visual Studio Code. By following the Red–Green–Refactor cycle, tests guided the implementation process, resulting in more reliable, maintainable, and bug-free code.

TDD improved design clarity, ensured correctness through automated testing, and enhanced overall software quality and agility.

Enhancing Code Agility through Design Principles and Refactoring

Aim:

To apply Java design principles and refactoring techniques in Visual Studio Code (VS Code) to improve code quality, maintainability, and agility.

Objectives:

- Understand key design principles: SOLID, DRY, and KISS.
- Refactor existing code to remove shared state and improve modularity.
- Enhance agility by making code easier to extend or modify.
- Reduce technical debt and improve overall software quality.
- Observe the benefits of applying design principles in practical coding.

Tools / Software:

- Programming Language: Java
- IDE: Visual Studio Code
- Extensions / Tools:
    - ✓ Extension Pack for Java (includes IntelliSense, Debugger, and Refactoring tools)
    - ✓ Code Runner (for running code easily)
    - ✓ Java Development Kit (JDK)

Theory / Design Principles:

1. SOLID Principles:

- **S – Single Responsibility:** Each class should have only one responsibility.
- **O – Open/Closed:** Classes should be open for extension, closed for modification.
- **L – Liskov Substitution:** Subtypes must be substitutable for their base types.
- **I – Interface Segregation:** Many small interfaces are better than one large interface.
- **D – Dependency Inversion:** Depend on abstractions, not concrete implementations.

2.DRY(Don'tRepeatYourself):

Avoid duplicating code; extract reusable logic into methods or helper classes.

3.KISS(KeepItSimple,Stupid):

Keep code simple, readable, and maintainable — avoid unnecessary complexity.

Procedure

Step 1: Set Up the Environment

1. Install Java JDK (verify using java -version and javac -version).
2. Open Visual Studio Code.

3. Install the following extensions:
    - Extension Pack for Java
    - Code Runner
    - IntelliCode (optional)
4. Create a new Java project:
    - Press Ctrl + Shift + P → Select “Java: Create Java Project” → Choose **No Build Tools**.
    - Name it DesignPrinciplesLab.

Step 2: Create Original Code

Calculator.java

```
class Calculator {
    int result;
    int add(int a, int b) { result = a + b; return result; }
    int subtract(int a, int b) { result = a - b; return result; }
    int multiply(int a, int b) { result = a * b; return result; }
    int divide(int a, int b) {
        if (b == 0) result = 0;
        else result = a / b;
        return result;
    }
}
```

MainOriginal.java

```
public class MainOriginal {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Addition 5 + 3 = " + calc.add(5, 3));
        System.out.println("Subtraction 10 - 4 = " + calc.subtract(10, 4));
        System.out.println("Multiplication 6 * 7 = " + calc.multiply(6, 7));
        System.out.println("Division 20 / 5 = " + calc.divide(20, 5));
    }}
```

Output:

```
Addition 5 + 3 = 8
Subtraction 10 - 4 = 6
Multiplication 6 * 7 = 42
Division 20 / 5 = 4
```

Step 3: Identify Problems

- All operations are in one class → violates Single Responsibility Principle.
- Shared state (result variable).
- Difficult to maintain and extend.
- Violates DRY and KISS principles.

Step 4: Refactor Using VS Code Refactoring Tools

1. Highlight each operation method → Right-click → Refactor → Extract to New Class.
2. Rename classes using F2 (e.g., Addition, Subtraction, etc.).
3. Move each class to its own file: Move Class command or manually create new .java files.
4. Create reusable printing logic for DRY principle.

Step 5: Refactored Code (After Applying SOLID, DRY, and KISS)

Addition.java

```
public class Addition {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Subtraction.java

```
public class Subtraction {
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

Multiplication.java

```
public class Multiplication {
    public int multiply(int a, int b) {
        return a * b;
    }
}
```

Division.java

```
public class Division {
    public int divide(int a, int b) {
        if (b == 0) throw new ArithmeticException("Cannot divide by zero");
        return a / b;
    }
}
```

ResultPrinter.java

```
public class ResultPrinter {
    public static void print(String operation, int result) {
        System.out.println(operation + " Result: " + result);
    }
}
```

MainRefactored.java

```
public class MainRefactored {
    public static void main(String[] args) {
        Addition add = new Addition();
        Subtraction sub = new Subtraction();
        Multiplication mul = new Multiplication();
        Division div = new Division();
        int a = 10, b = 5;
        ResultPrinter.print("Addition", add.add(a, b));
        ResultPrinter.print("Subtraction", sub.subtract(a, b));
        ResultPrinter.print("Multiplication", mul.multiply(a, b));
        ResultPrinter.print("Division", div.divide(a, b));
    }
}
```

Step 6: Run and Test Refactored Code

- Right-click MainRefactored.java → Run Java

Output:

```
Addition Result: 15
Subtraction Result: 5
Multiplication Result: 50
Division Result: 2
```

Observations / Results

Aspect	Original Code	Refactored Code
Structure	Single class	Modular classes
Responsibility	Multiple responsibilities	One per class (SRP)
Shared State	Yes (result variable)	None
Extensibility	Difficult	Easy to extend
Readability	Low	High
Maintainability	Poor	Excellent

Final Output:

```
Addition Result: 15
Subtraction Result: 5
Multiplication Result: 50
Division Result: 2
```

Advantages of Refactoring:

- Code becomes modular, reusable, and clean.
- Follows SOLID, DRY, and KISS principles.
- Easier to extend (e.g., add modulus or power operations).
- Eliminates shared state → fewer side effects.
- Improves agility and maintainability.

Conclusion:

By applying SOLID, DRY, and KISS principles along with refactoring techniques, the code was transformed into a more modular, readable, and maintainable structure. Shared state was eliminated, responsibilities were clearly separated, and code duplication was reduced. These improvements made the system easier to extend and modify, enhanced agility, reduced technical debt, and ultimately improved the overall quality, reliability, and maintainability of the software.

Automating Builds with Jenkins Integrated with GitHub

Aim:

To understand how Jenkins can be used as an **automated build tool** connected with GitHub to automatically fetch, compile, and build code whenever changes are made to the repository.

Objectives:

1. To understand the concept of build automation in Agile development.
2. To learn how Jenkins automates the build process.
3. To integrate Jenkins with GitHub for automatic code fetching.
4. To demonstrate an automated build process using a sample Java project.

Theory:

An **Automated Build Tool** is a software utility that automates the process of converting source code into executable applications. It performs actions like compilation, testing, and packaging automatically — removing manual effort and reducing human error.

In this experiment, **Jenkins** acts as an **automated build tool**. Jenkins connects to a **GitHub repository**, automatically fetches the latest source code when a change is committed, and runs the build script automatically.

Key Features of Jenkins (as a Build Tool):

- Automatically pulls source code from version control (e.g., GitHub).
- Executes build commands such as javac, make, or npm run build.
- Generates build reports and logs.
- Can be triggered manually or automatically through GitHub webhooks.

Architecture Diagram:

Developer → GitHub Repository → Jenkins Server → Build Automation → Console Output

Procedure:

Step 1: Install Jenkins

1. Download Jenkins from https://www.jenkins.io/download.
2. Install and start the Jenkins service.
3. Open Jenkins Dashboard using http://localhost:8080.

Step 2: Create a New Freestyle Project

1. Click on **“New Item”** → select **Freestyle Project**.
2. Enter the project name: AutomatedBuildDemo.

3. Click **OK**.

Step 3: Connect to GitHub Repository

1. In the **Source Code Management** section, choose **Git**.
2. Paste your repository URL (e.g., https://github.com/username/BuildDemo.git).
3. Add credentials if required.

Step 4: Configure Build Trigger

- Select **“GitHub hook trigger for GITScm polling”** → this ensures Jenkins automatically builds the project when new code is pushed to GitHub.

Step 5: Add Build Step

1. Scroll to **Build → Add Build Step → Execute Windows Batch Command** (or Shell Script).
2. Enter the following build script:
```
echo "Starting automated build..."
javac -d bin src/*.java
echo "Build completed successfully!"
```

Step 6: Save and Run

1. Click **Save**.
2. Click **Build Now** to run the build manually.
3. Observe the **Console Output** for results.

Step 7: Test Automation

- Commit and push a small change to GitHub.
- Jenkins automatically detects the new commit and triggers another build.

Output:

Jenkins automatically pulls the latest code from GitHub and runs the build script.

Console Output Example:

```
Starting automated build...
Compiling Java files...
Build completed successfully!
Finished: SUCCESS
```

Result:

Successfully studied and implemented an **Automated Build Process** using Jenkins integrated with GitHub. Jenkins automatically fetched, compiled, and built the source code whenever updates were pushed to the GitHub repository.

Conclusion:

The experiment was successfully conducted, and it was concluded that **Jenkins** is an efficient automated build tool that can fetch source code directly from **GitHub** and execute build tasks automatically.

By integrating Jenkins with GitHub, the manual effort required for compiling and building code is eliminated. This ensures faster, more reliable, and consistent builds in Agile development environments.

The experiment demonstrated how continuous automation simplifies the software build process and enhances productivity.

Managing Source Code with Git and GitHub

Aim:

To study and understand how Git and GitHub manage and track source code versions, allowing multiple developers to collaborate efficiently in Agile software development.

Objectives:

1. To understand the concept of version control.
2. To learn how to initialize, commit, and push code using Git.
3. To use GitHub for online collaboration and repository management.
4. To demonstrate practical version control operations.

Theory:

A **Version Control System (VCS)** tracks and manages changes to source code, allowing developers to revert to previous versions, collaborate safely, and maintain history.

- **Git**: A distributed VCS that maintains local repositories for each developer.
- **GitHub**: A cloud-based hosting service for Git repositories.

Advantages of Git & GitHub:

- Full project history
- Collaboration among multiple developers
- Branching and merging support
- Rollback to previous versions
- Continuous integration compatibility with Jenkins

Procedure:

Step 1: Install and Configure Git

```
git config --global user.name "YourName"
git config --global user.email "youremail@example.com"
```

Step 2: Initialize Repository

```
git init
```

Step 3: Add and Commit Files

```
git add .
git commit -m "Initial commit"
```

Step 4: Create Remote Repository on GitHub

1. Log in to GitHub → Click **New Repository** → Name it VersionDemo.
2. Copy the remote URL.

Step 5: Link Local Repo to Remote

```
git remote add origin https://github.com/username/VersionDemo.git
```

Step 6: Push Code to GitHub

```
git push -u origin main
```

Step 7: Create and Merge Branch

```
git branch feature
git checkout feature
git merge feature
```

Output:

All commits and branches are visible on the GitHub repository. Developers can collaborate, merge changes, and track the full version history.

Result:

Successfully studied and demonstrated version control operations using Git and GitHub. Code changes were tracked, committed, pushed, and managed efficiently.

Conclusion:

The experiment was successfully completed, and it was concluded that **Git** and **GitHub** provide a powerful and distributed version control system for managing source code in Agile projects. Through Git commands such as init, commit, push, and branching operations, developers can easily track changes, collaborate on shared repositories, and maintain code history. This ensures efficient team collaboration, transparency, and smooth integration with other Agile tools like Jenkins for automated builds and continuous integration.

Continuous Integration Using Jenkins Freestyle Project

Aim:

To understand and demonstrate Continuous Integration (CI) using Jenkins Freestyle Project, where code from GitHub is automatically built and tested after every commit.

Objectives:

1. To understand the concept of Continuous Integration.
2. To learn how Jenkins automates build and testing after each commit.
3. To configure and run a Jenkins Freestyle Project.
4. To observe CI in action through automatic build triggering.

Theory:

Continuous Integration (CI) is an Agile practice where developers integrate their code changes frequently, and every change is automatically tested and built. **Jenkins**, a CI tool, automates this process using pipelines or freestyle jobs.

A **Freestyle Project** in Jenkins is a flexible job type that allows defining custom build steps and triggers without coding a pipeline.

Key Features of Jenkins CI:

- Automatically detects changes from GitHub.
- Builds and tests source code.
- Provides success/failure build reports.
- Ensures code stability across teams.

Procedure:

Step 1: Create a Freestyle Project

1. Open Jenkins → Click **New Item → Freestyle Project** → Name it CIBuildDemo.
2. Select **OK**.

Step 2: Connect GitHub Repository

- Under **Source Code Management**, select **Git**.
- Enter your GitHub repository URL.

Step 3: Configure Build Trigger

- Enable: **"GitHub hook trigger for GITScm polling."**
  → This triggers a build whenever new code is pushed to GitHub.

Step 4: Add Build Command

Under **Build Steps → Execute Batch Command** (or Shell Script):

```
echo "Running CI Build..."
javac -d bin src/*.java
java -cp bin HelloWorld
echo "CI Build Completed Successfully!"
```

Step 5: Save and Run

1. Save configuration.
2. Click **Build Now** to test manually.
3. Push a new commit to GitHub — Jenkins will automatically start the CI build.

Output:

Console output shows:

```
Running CI Build...
Compiling project files...
Hello, Agile World!
CI Build Completed Successfully!
Finished: SUCCESS
```

Each GitHub commit automatically triggers a Jenkins build, ensuring real-time CI integration.

Result:

Successfully demonstrated **Continuous Integration** using Jenkins Freestyle Project connected with GitHub. Jenkins automatically built and tested the code after every commit.

Conclusion:

The experiment was successfully demonstrated, and it was concluded that **Jenkins Freestyle Project** plays a crucial role in implementing **Continuous Integration (CI)**. By connecting Jenkins with a GitHub repository, the CI process automatically triggers builds and tests after each code commit, ensuring early detection of errors and maintaining code stability. This experiment highlighted how Continuous Integration supports Agile principles by providing rapid feedback, automation, and seamless integration across the development cycle.

Executing Testing Processes in an Agile Project

Aim:

To understand and implement testing activities in an Agile software development project, ensuring software quality through continuous testing and feedback.

Objectives:

1. Understand the concept of Agile testing.
2. Perform unit testing, integration testing, and acceptance testing.
3. Learn how testing integrates into Agile sprints.
4. Observe the benefits of continuous testing and feedback.
5. Demonstrate practical testing activities with a simple project.

Tools / Software:

- Programming Language: Java
- IDE: Visual Studio Code
- Testing Framework: JUnit 4 / 5
- Optional: Maven for dependency management

Theory:

Agile testing is a software testing practice that follows the principles of Agile development. Testing is **continuous, iterative, and collaborative** with development.

Key Agile Testing Practices:

1. **Unit Testing:** Test individual units of code (methods, classes).
2. **Integration Testing:** Test interactions between multiple units or modules.
3. **Acceptance Testing:** Validate that the software meets user requirements.
4. **Continuous Testing:** Testing occurs frequently during development, often automated.

Benefits:

- Early detection of defects
- Faster feedback and corrections
- Ensures quality during each sprint
- Supports customer-focused development

Procedure / Steps (Simple Example — Calculator Project)

Step 1: Set Up Environment

1. Install Java JDK (verify: java -version and javac -version)
2. Install VS Code and extensions:

- ✓ Java Extension Pack
- ✓ Test Runner for Java

3. Create a new Java project: AgileTestingDemo

Project structure:

```
AgileTestingDemo/
├── src/
│   ├── main/java/Calculator.java
│   └── test/java/CalculatorTest.java
```

Step 2: Create Original Code (Minimal Functionality)

Calculator.java

```
package main.java;
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

Step 3: Write Unit Tests (Red Phase)

CalculatorTest.java

```
package main.java;
import static org.junit.Assert.*;
import org.junit.Test;
public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(5, result);
    }
    @Test
    public void testSubtract() {
        Calculator calc = new Calculator();
        int result = calc.subtract(5, 3);
        assertEquals(2, result);
    }
}
```

- Run tests → should **pass** because methods exist.
- If methods were missing, tests would fail (Red Phase).

Step 4: Integration Testing

1. Create a new module if multiple classes exist (e.g., Addition.java, Subtraction.java)
2. Test combined functionality:

```
@Test
public void testCalculatorOperations() {
    Calculator calc = new Calculator();
    int sum = calc.add(10, 5);
    int diff = calc.subtract(10, 5);
    assertEquals(15, sum);
    assertEquals(5, diff);
}
```

- Ensures modules interact correctly.

Step 5: Acceptance Testing (User Requirements)

1. User Story: "User can perform addition and subtraction correctly."
2. Acceptance Test:

```
@Test
public void testUserStory() {
    Calculator calc = new Calculator();
    assertEquals(7, calc.add(4,3));
    assertEquals(2, calc.subtract(5,3));
}
```

- Simulates end-user validation.

Step 6: Continuous Testing in Agile Sprint

1. Add new features (e.g., multiplication).
2. Write tests **before coding** (TDD approach).
3. Implement minimal code → run tests → refactor → repeat.

ExampleforMultiplication:

Test:

```
@Test
public void testMultiply() {
    Calculator calc = new Calculator();
    assertEquals(12, calc.multiply(3,4));
}
```

Code:

```
public int multiply(int a, int b) {
    return a * b;
}
```

Step 7: Observations / Results

Aspect	**Observation**
Unit Testing	Tests individual methods, detects early defects
Integration Testing	Ensures multiple modules interact correctly
Acceptance Testing	Confirms software meets user requirements
Continuous Testing	Frequent automated testing ensures quality

Step 8: Conclusion

The experiment successfully demonstrated Agile testing activities using a simple Java calculator project. Testing within Agile sprints ensures **early detection of defects, faster feedback, and continuous improvement**. Unit tests, integration tests, and acceptance tests collectively maintain software quality, validate requirements, and support iterative development.