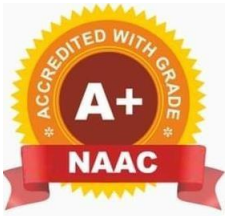


SHARDA SCHOOL OF COMPUTER SCIENCE AND ENGINEERING (SSCSE)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Software Engineering & Testing Methodologies Lab

(CSP 357)

Practical Record

Semester-V

Submitted By:

Student's Name**:** Vivek Manaee

System ID: 2023548204

Roll Number: 2301010988

Class &Section: CSE-O (G - 2)

Submitted To:

Mr. Prem Prakash Agrawal

Assistant Professor

CSE Department, SSCSE

Sharda University, Greater Noida

INDEX

S. No.	EXPERIMENT NAME	DATE OF EXPERIMENT	PAGE NUMBER	SIGNATURE
1	Design requirement analysis and develop Software Requirement Specification Sheet (SRS) for suggested system in IEEE standard.			
2	Draw the user's view analysis for the suggested system: Use case diagram, Activity Diagram and State chart Diagram.			
3	Draw the structural view diagram for the system: Class diagram, object diagram and Sequence diagram.			
4	Perform Unit testing of calculator program using Junit testing framework.			
5	Perform Integration testing of calculator program using Junit testing framework.			
6	Design Test Case for Inventory Management System based on System Specification.			
7	Design Test cases and perform manual testing of login module of Flipkart (as an example).			

VALUE ADDED EXPERIMENTS

S. No.	EXPERIMENT NAME	DATE OF EXPERIMENT	PAGE NUMBER	SIGNATURE
1	Selenium API setup and creation of Maven project.			
2	Perform automation testing using methods Maximise, minimize and window close.			
3	Identification of web elements using X-Path and CSS.			
4	Perform automation testing of checkbox, radio button of application.			
5	Perform is Displayed, isSelected and isEnabled in for validation in testing.			

EXPERIMENT NO -1

Software Requirements Specification

for

File Metadata Analyzer Tool

Version 1.0 approved

Prepared by Vivek Kumar, Vivek Manaee, Utkarsh Tyagi

Sharda University

08/18/2025

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
1.Vivek Manaee	08/08/25	Initial Draft	1.0
2. Vivek Kumar	08/08/25	Completed the changes as suggested	1.0
3. Utkarsh Tyagi	08/08/25	Completed the changes as suggested	1.0

1. Introduction

1.1 Purpose

The File Metadata Analyzer Tool is designed to help users analyze digital files and extract hidden or embedded metadata. Metadata provides contextual information such as creation time, author details, camera model, GPS location, software version, and file structure details. This tool is highly relevant in cybersecurity and forensic investigations where identifying hidden details within files can support evidence analysis, digital auditing, or threat detection.

1.2 Document Conventions

- Requirements are uniquely identified as REQ-x.
- Priorities are marked as High, Medium, or Low.
- All metadata terms are explained in the Glossary (Appendix A).

1.3 Intended Audience

This document is intended for the following audiences:

- **Developers:** To understand system features, functional requirements, and technical constraints for implementation.
- **Project Managers:** To track progress, scope, and ensure alignment with project goals.
- **Testers/QA Engineers:** To design test cases based on functional and non-functional requirements.
- **End Users/Clients:** To gain clarity on the tool's capabilities and intended use cases.
- **Documentation Writers:** To prepare user manuals and training materials based on features and workflows.

1.4 Product Scope

The **File Metadata Analyzer Tool** is a lightweight utility designed to extract, analyze, and store metadata from various digital file formats (JPEG, PNG, PDF, DOCX). It aids in cybersecurity investigations, digital forensics, and academic research by uncovering contextual information such as creation dates, authorship, software version, and GPS details.

Objectives and Goals:

- Provide a reliable, cross-platform tool for extracting metadata.
- Support both command-line (CLI) and web-based (FastAPI) interfaces.
- Enable users to save, query, and export metadata for integration with other tools.
- Ensure security by sandboxing uploaded files and preventing execution of malicious content.

Business/StrategicRelevance:

This tool aligns with organizational goals in **digital forensics, threat detection, and research support**, offering efficiency, accuracy, and extensibility through open-source libraries. It can also be adapted for enterprise auditing and compliance use cases.

1.5 References

The following resources were used or are relevant to this SRS:

1. IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications – IEEE Computer Society.
2. Python python-magic Documentation – https://github.com/ahupp/python-magic
3. Python piexif Library – Metadata extraction for images.
4. Python pypdf Library – Metadata extraction for PDF files.
5. Python python-docx Library – Metadata extraction for Word files.
6. FastAPI Documentation – https://fastapi.tiangolo.com
7. SQLite Documentation – https://sqlite.org

2. Overall Description

2.1 Product Perspective

The File Metadata Analyzer is a standalone tool that can function both as a CLI utility and as a web-based application using FastAPI. It leverages open-source Python libraries such as piexif, pypdf, and python-docx.

2.2 Product Functions

The tool detects the file type, extracts metadata, normalizes it into a standard JSON schema, displays results to the user, and saves them into a local database for later review.

2.3 User Classes and Characteristics

The tool targets basic users who only want to check metadata, developers who may extend the functionality by adding new extractors, and researchers who want to analyze metadata for forensic case studies

2.4 Operating Environment

It will run cross-platform on Windows, Linux, and macOS using Python 3.10+. It requires no high-end hardware and works with limited memory since metadata extraction is lightweight.

2.5 Design and Implementation Constraints

The MVP focuses only on core file types. Advanced features such as anomaly detection and large dataset handling are excluded. Only open-source libraries will be used.

2.6 User Documentation

A detailed README, installation guide, and usage examples will accompany the tool.

2.7Assumptions and Dependencies

The system assumes users provide valid files. Dependencies include continued support for open-source libraries like piexif, pypdf, and python-docx

3. External Interface Requirements

3.1 User Interfaces

A Command Line Interface (CLI) will allow simple execution commands such as: 'python analyze.py file.jpg'. A basic web interface will allow file uploads and return metadata in JSON format

3.2 Hardware Interfaces

The tool does not depend on specialized hardware; it only requires a standard machine running Python.

3.3 Software Interfaces

The tool will use Python's file handling libraries, metadata extractors, and SQLite as the database engine. For web-based deployment, FastAPI and Uvicorn will be used

3.4 Communications Interfaces

Web-based usage will depend on HTTP/HTTPS protocols to allow uploads and metadata retrieval securelySystem Features

- **File Upload & Detection:** The tool supports both drag-and-drop file uploads on the web interface and command-line uploads. File type is identified using python-magic to ensure accuracy.
- **Metadata Extraction:** The core feature includes extracting metadata fields. For images, EXIF data such as camera make, model, GPS location, and creation date are retrieved. For PDF files, author, title, creation time, and software used are extracted. For DOCX, author, last modified by, and document title are collected.
- **Metadata Storage:** Extracted metadata is saved in JSON for human readability and SQLite for structured querying. This ensures both simplicity and extensibility.
- **Reporting:** Users can download the analyzed results in JSON or CSV format for integration into other forensic tools or academic research.

3.5 System Feature 1

3.5.1. File Upload & Detection

3.5.2 Description and Priority

This feature enables users to upload files for analysis. The web interface supports drag-and-drop uploads, while the command-line interface (CLI) accepts file paths as input. The system uses python-magic to accurately identify the file type. **Priority**: High (critical starting point of workflow)

3.5.3 Stimulus/Response Sequences

- **Stimulus**: User uploads file via web interface (drag-and-drop) or enters file path in CLI.
- **Response**: System detects the file type and confirms the result. If the file type is unsupported, the system returns an error message.

3.5.4 Functional Requirements

- **REQ-1**: The system shall allow drag-and-drop file uploads in the web interface.
- **REQ-2**: The system shall accept file paths in the CLI.
- **REQ-3**: The system shall use python-magic to determine the file type.
- **REQ-4**: The system shall notify the user if the file type is unsupported.

Safety Requirements

The tool never executes files; it only reads metadata, ensuring safe analysis without risk of infection from malicious files.

3.6 Security Requirements

Uploaded files are sandboxed to prevent any compromise. If deployed on the web, HTTPS will ensure encrypted communication.

3.7 Software Quality Attributes

The system will emphasize usability (easy commands, clear outputs), maintainability (modular extractor functions), and portability (cross-platform support).

3.8 Business Rules

Only supported formats (JPEG, PNG, PDF, DOCX) will be fully reliable. Unsupported formats may return partial results

4. Other Requirements

Future work includes support for multimedia files, anomaly detection in metadata (e.g., mismatched timestamps), and integration with forensic analysis frameworks.

Appendix A: Glossary

EXIF: Image metadata standard. PDF Metadata: Document properties. DOCX Core Properties: Built-in metadata of Word documents.

Appendix B: Analysis Models

Workflow: User Uploads File → File Type Detection → Metadata Extraction → Normalization → JSON/SQLite Storage → Report Generation.

Fig: Flowchart for File Metadata Analyzer Tool

4. Metadata Extraction

4.1.1 Description and Priority

The core functionality of the tool is extracting metadata from supported files. Different metadata fields are extracted depending on the file type (images, PDFs, DOCX).
Priority: High (central function of the tool)

4.1.2 Stimulus/Response Sequences

- **Stimulus**: User uploads a valid supported file.
- **Response**: System parses the file, extracts metadata, and displays results in structured form (JSON).

4.1.3 Functional Requirements

- **REQ-5**: The system shall extract EXIF data (camera make, model, GPS, creation date) from image files (JPEG, PNG).
- **REQ-6**: The system shall extract author, title, creation time, and software details from PDF files.
- **REQ-7**: The system shall extract author, last modified by, and title from DOCX files.
- **REQ-8**: The system shall display extracted metadata in a normalized JSON format.

4.2 Metadata Storage

4.2.1 Description and Priority

After extraction, metadata must be stored for later review. JSON is used for human readability, while SQLite provides structured storage for querying.
Priority: Medium (important for persistence but not critical to basic function)

4.2.2 Stimulus/Response Sequences

- **Stimulus**: Metadata extraction is complete.
- **Response**: Metadata is saved automatically into both JSON and SQLite formats.

4.2.3 Functional Requirements

- **REQ-9**: The system shall save extracted metadata in JSON format.
- **REQ-10**: The system shall store extracted metadata in SQLite for structured querying.
- **REQ-11**: The system shall handle duplicate entries by preventing redundant storage.

4.3 Reporting

4.3.1 Description and Priority

Users can export analyzed results in JSON or CSV format for integration into forensic tools or academic research.
Priority: Medium (adds usability and interoperability).

4.3.2 Stimulus/Response Sequences

- **Stimulus**: User requests to download analysis results.
- **Response**: System generates a report in the chosen format (JSON or CSV) and provides it for download.

4.3.3 Functional Requirements

- **REQ-12**: The system shall allow users to download analysis results in JSON format.
- **REQ-13**: The system shall allow users to download analysis results in CSV format.
- **REQ-14**: The system shall ensure downloaded reports preserve complete metadata details.

5.Other Nonfunctional Requirements

Performance Requirements

Small files should be processed in under 5 seconds. Batch processing of up to 20 files should complete in under a minute

5. Other Requirements

Future work includes support for multimedia files, anomaly detection in metadata (e.g., mismatched timestamps), and integration with forensic analysis frameworks.

Appendix A: Glossary

EXIF: Image metadata standard. PDF Metadata: Document properties. DOCX Core Properties: Built-in metadata of Word documents.

Appendix B: Analysis Models

Workflow: User Uploads File → File Type Detection → Metadata Extraction → Normalization → JSON/SQLite Storage → Report Generation.

Fig: Flowchart for File Metadata Analyzer Tool

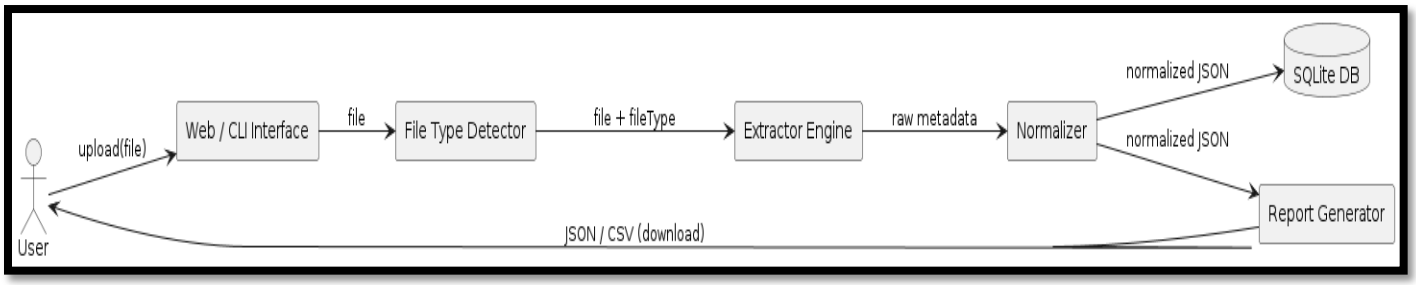
Appendix C: To Be Determined List

Future GUI design, support for video/audio metadata, ML-based anomaly 12etection

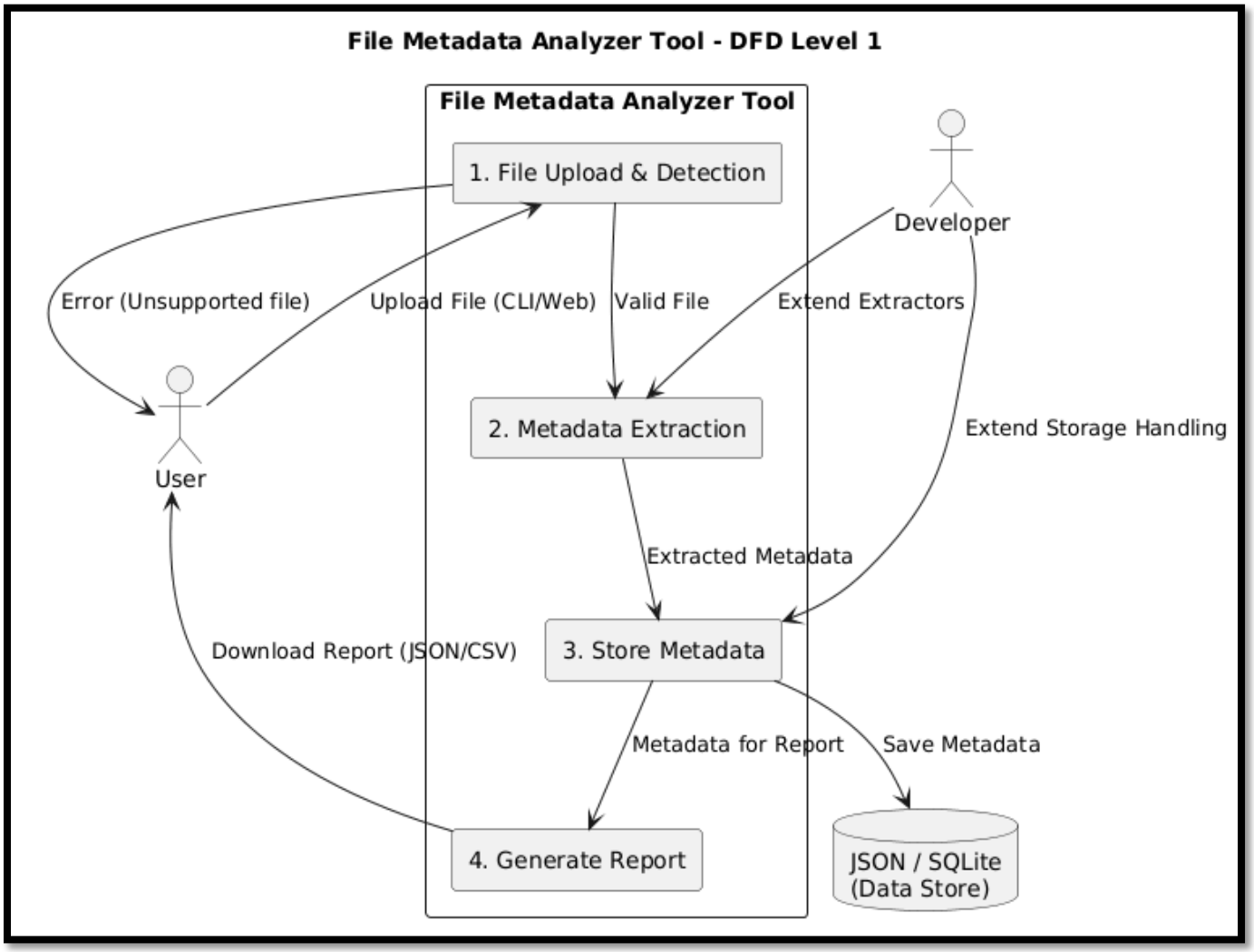
EXPERIMENT 2 &3

DESIGN PHASE

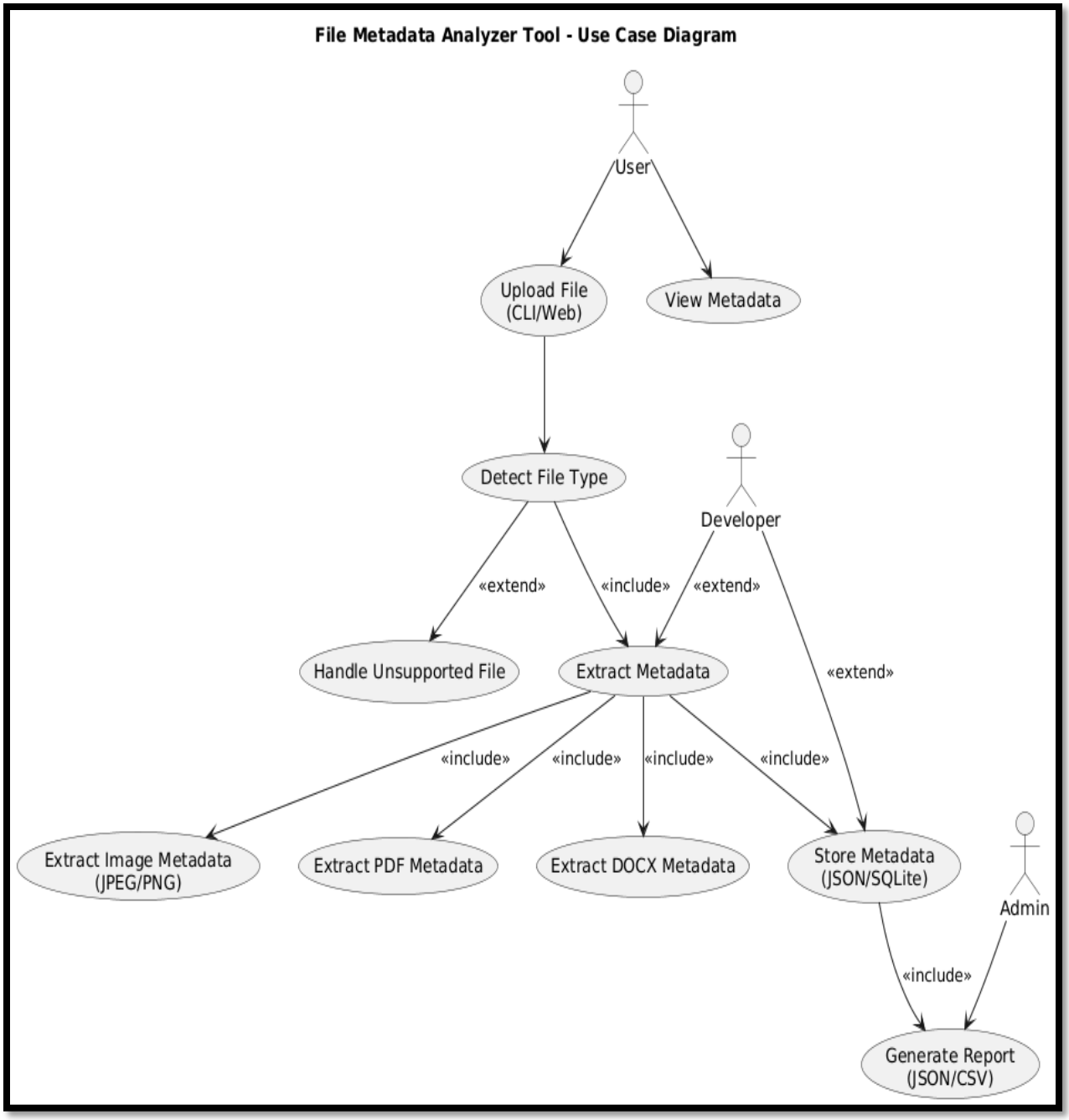
DATA FLOW DIAGRAM LEVEL 0



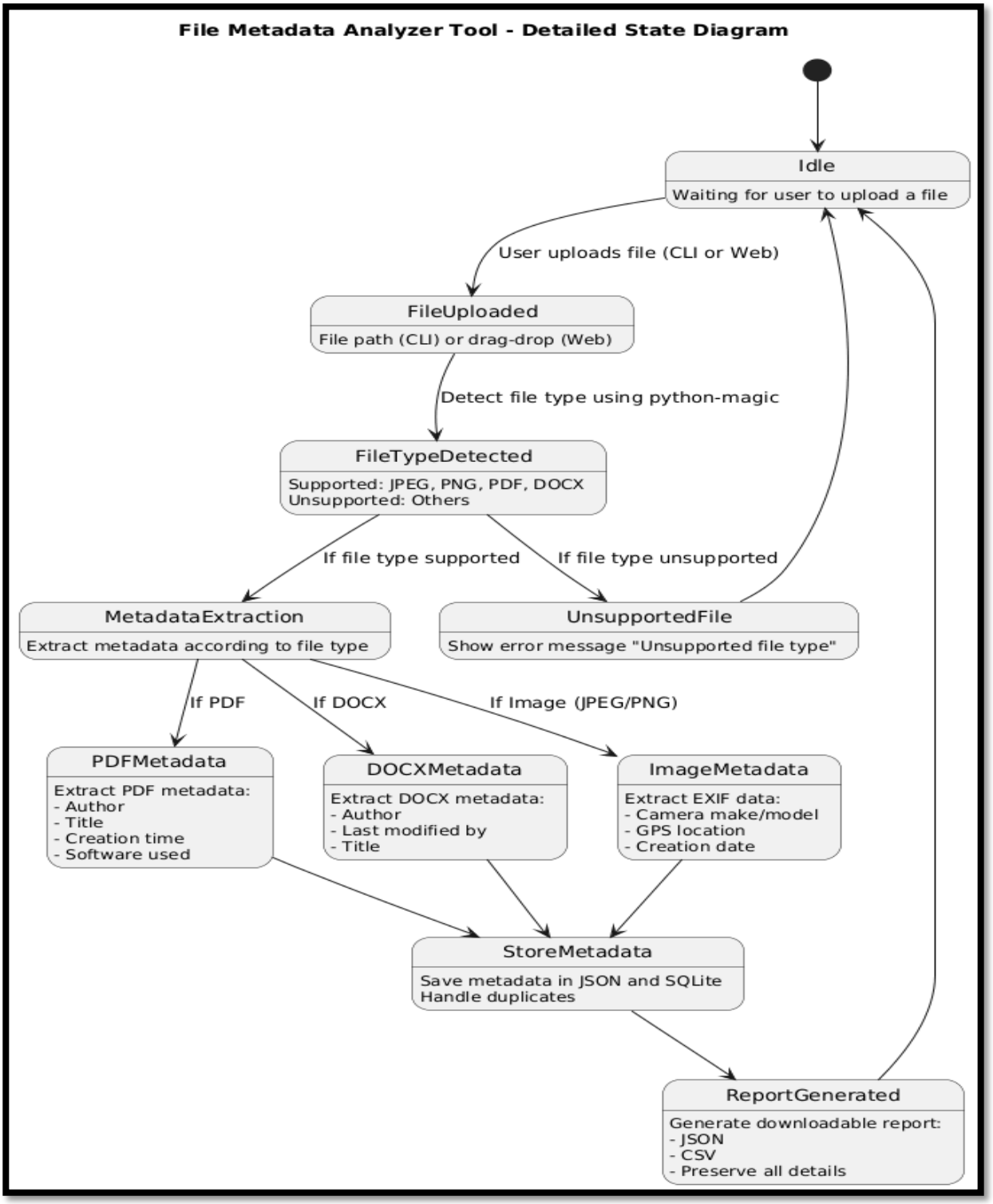
DATA FLOW DIAGRAM LEVEL 1



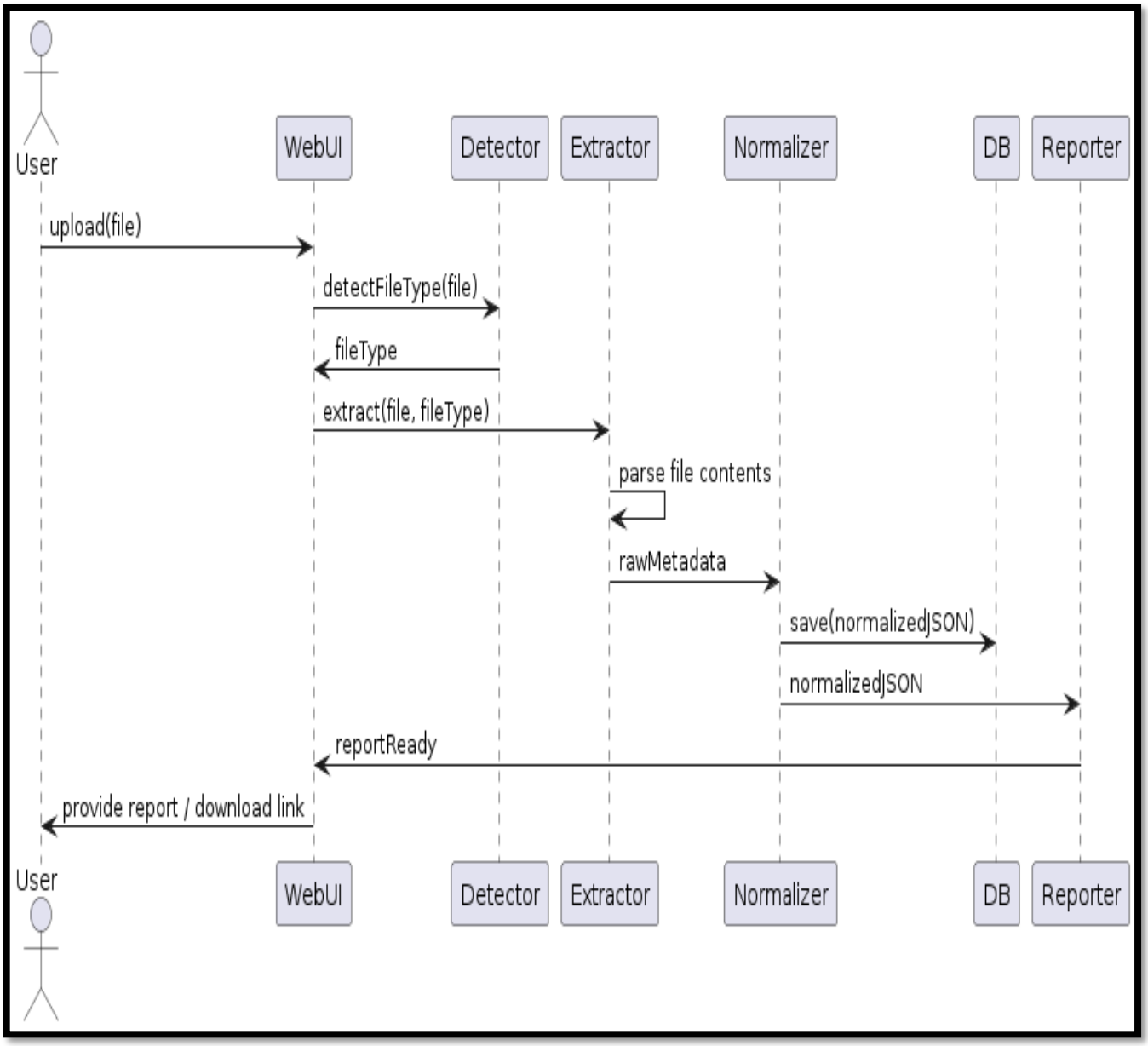
USECASE DIAGRAM



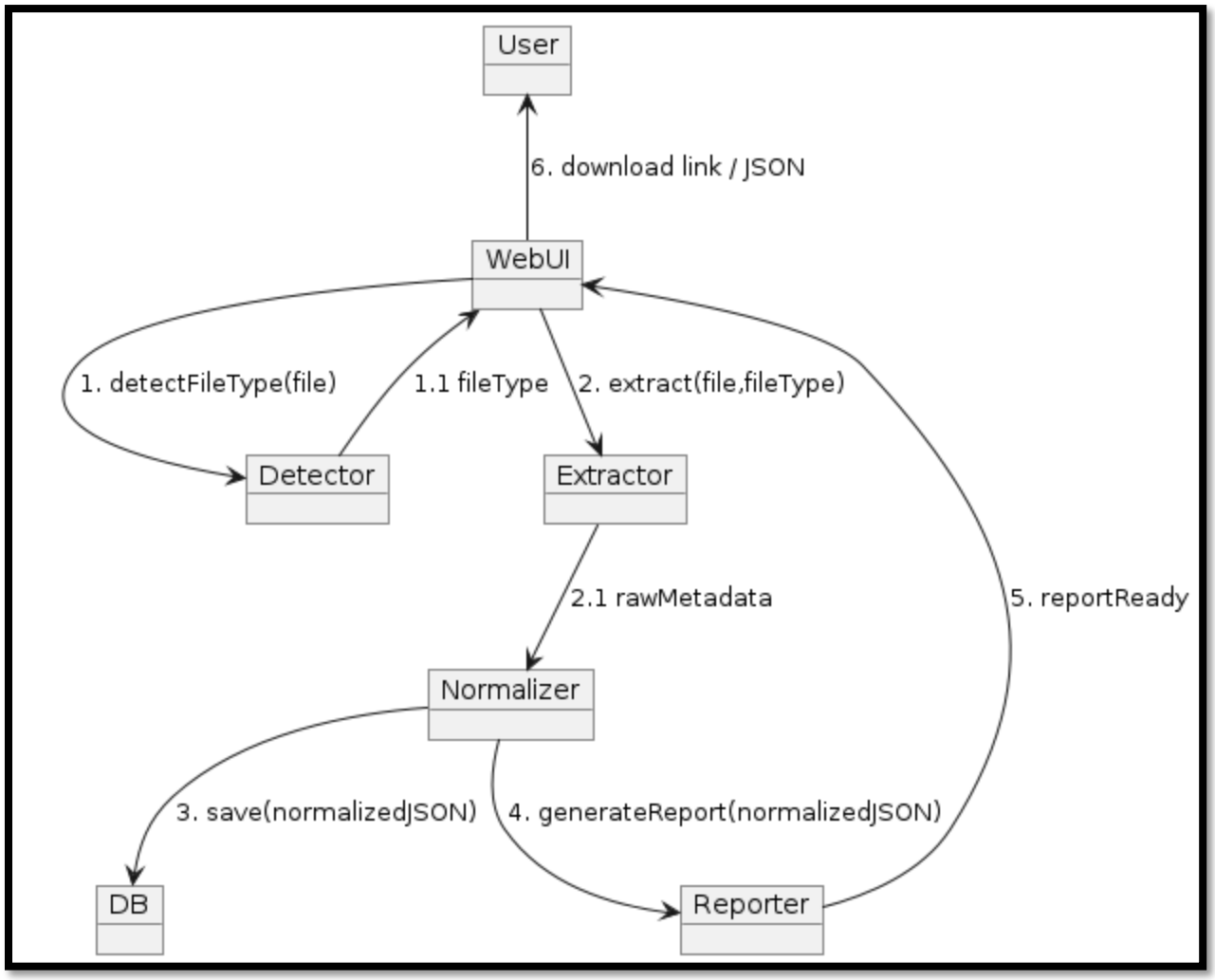
STATE DIAGRAM



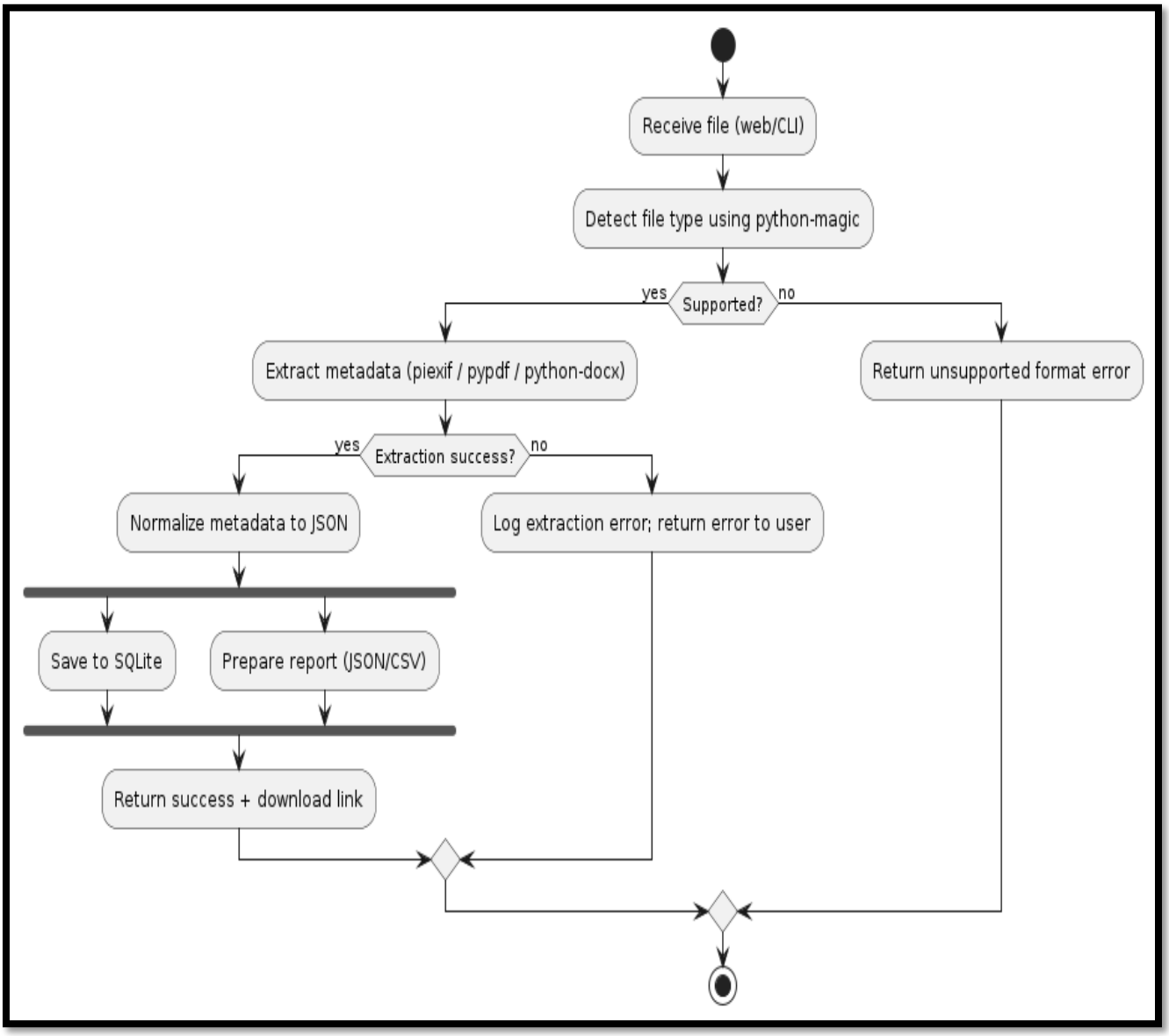
SEQUENCE DIAGRAM



COLLABORATION DIAGRAM



ACTIVITY DIAGRAM



EXPERIMENT NO – 4

- **To perform unit testing of a calculator program using the JUnit framework, you first create the Calculator class with basic arithmetic methods and then create a separate Calculator Test class to verify its functionality using JUnit annotations and assertions.**

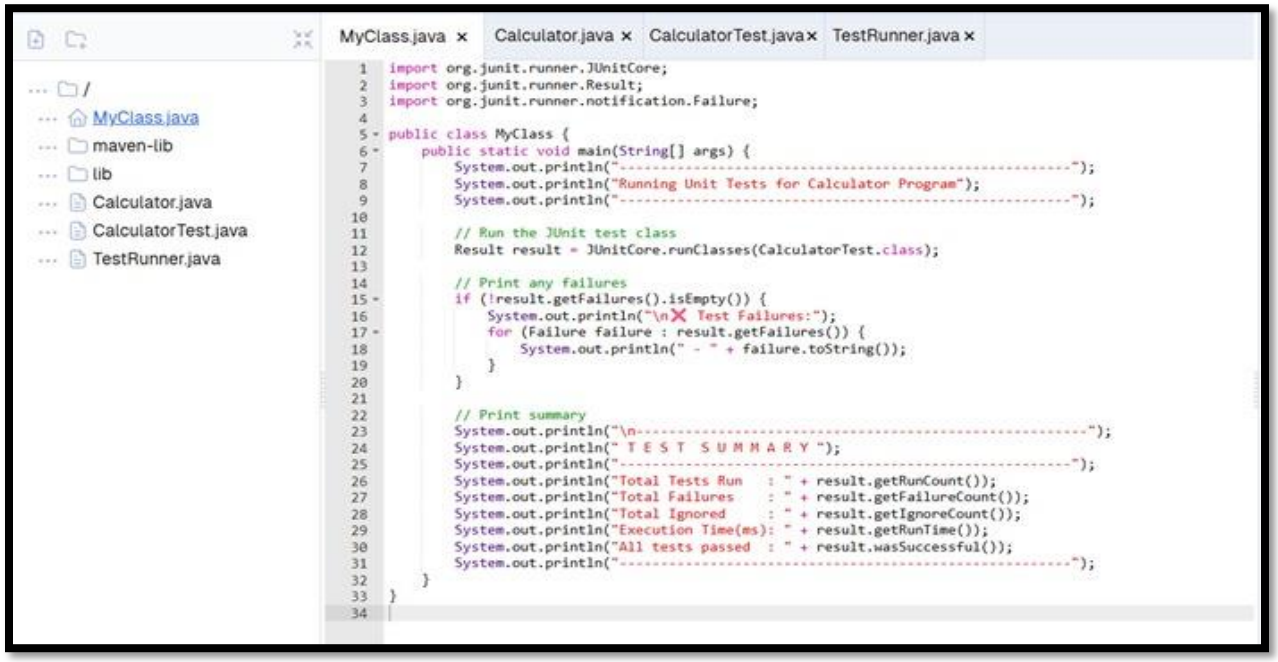


Fig.1 creating class

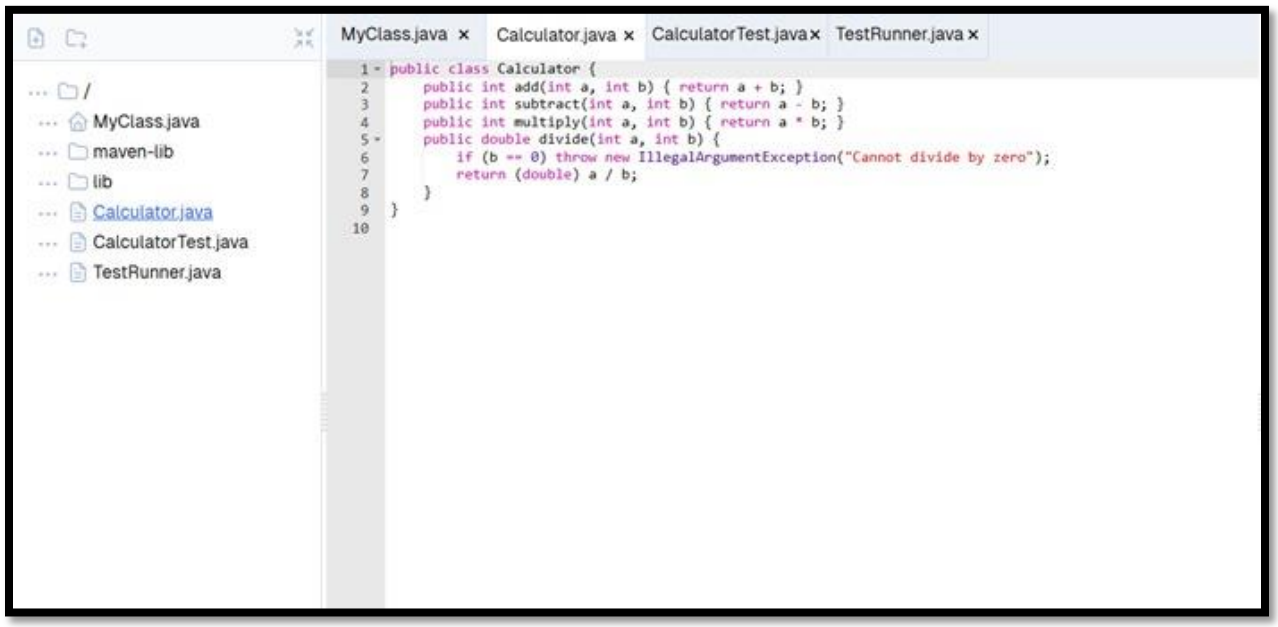
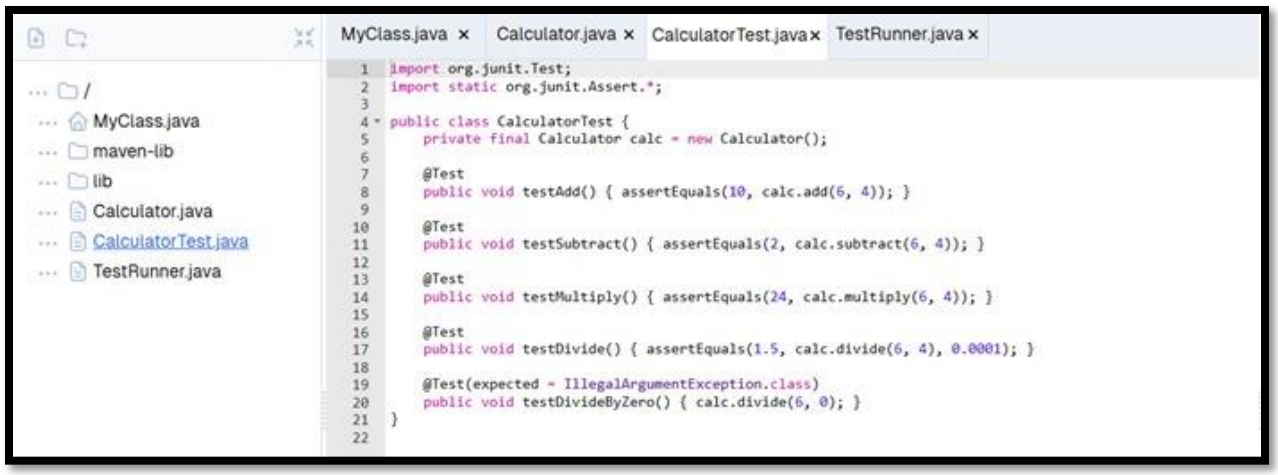
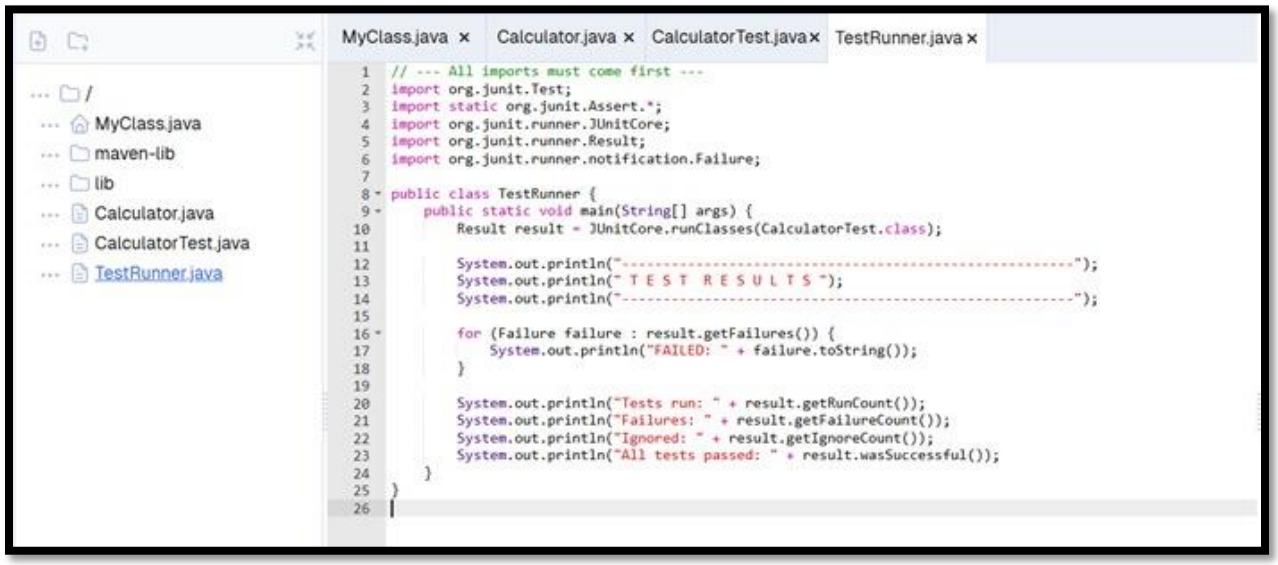
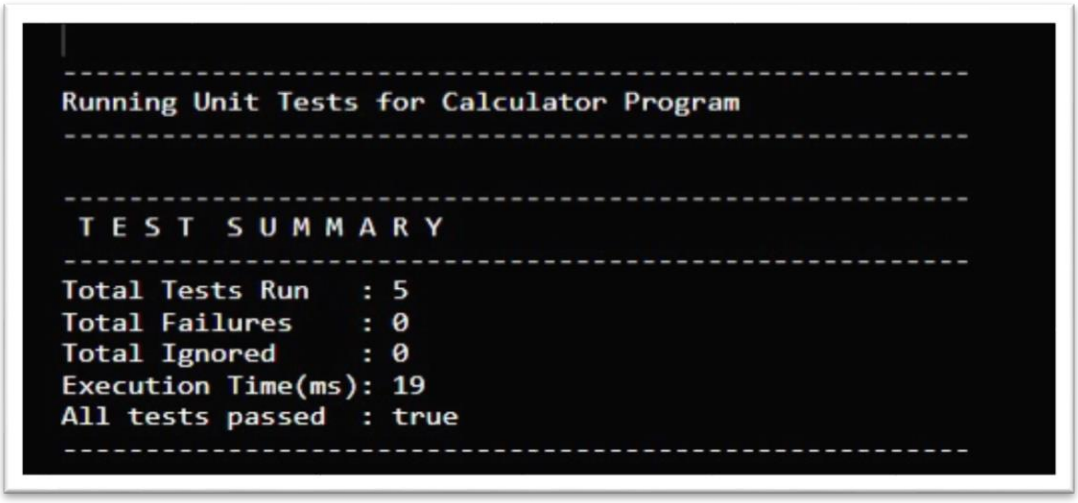


Fig.1 Calculator class


MyClass.java
Calculator.java
CalculatorTest.java
TestRunner.java
/
MyClass.java
maven-lib
lib
Calculator.java
CalculatorTest.java
TestRunner.java
import org.junit.Test;
import static org.junit.Assert.*;
public class CalculatorTest {
private final Calculator calc = new Calculator();
@Test
public void testAdd() { assertEquals(10, calc.add(6, 4)); }
@Test
public void testSubtract() { assertEquals(2, calc.subtract(6, 4)); }
@Test
public void testMultiply() { assertEquals(24, calc.multiply(6, 4)); }
@Test
public void testDivide() { assertEquals(1.5, calc.divide(6, 4), 0.0001); }
@Test(expected = IllegalArgumentException.class)
public void testDivideByZero() { calc.divide(6, 0); }
}


MyClass.java
Calculator.java
CalculatorTest.java
TestRunner.java
/
MyClass.java
maven-lib
lib
Calculator.java
CalculatorTest.java
TestRunner.java
// --- All imports must come first ---
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class TestRunner {
public static void main(String[] args) {
Result result = JUnitCore.runClasses(CalculatorTest.class);
System.out.println("--");
System.out.println(" T E S T R E S U L T S ");
System.out.println("--");
for (Failure failure : result.getFailures()) {
System.out.println("FAILED: " + failure.toString());
}
System.out.println("Tests run: " + result.getRunCount());
System.out.println("Failures: " + result.getFailureCount());
System.out.println("Ignored: " + result.getIgnoreCount());
System.out.println("All tests passed: " + result.wasSuccessful());
}
}

Results:


Running Unit Tests for Calculator Program
T E S T S U M M A R Y
Total Tests Run : 5
Total Failures : 0
Total Ignored : 0
Execution Time(ms): 19
All tests passed : true

EXPERIMENT NO – 5

- **Performing integration testing of a calculator program using JUnit typically involves creating test cases that verify the interaction and combined functionality of different units**

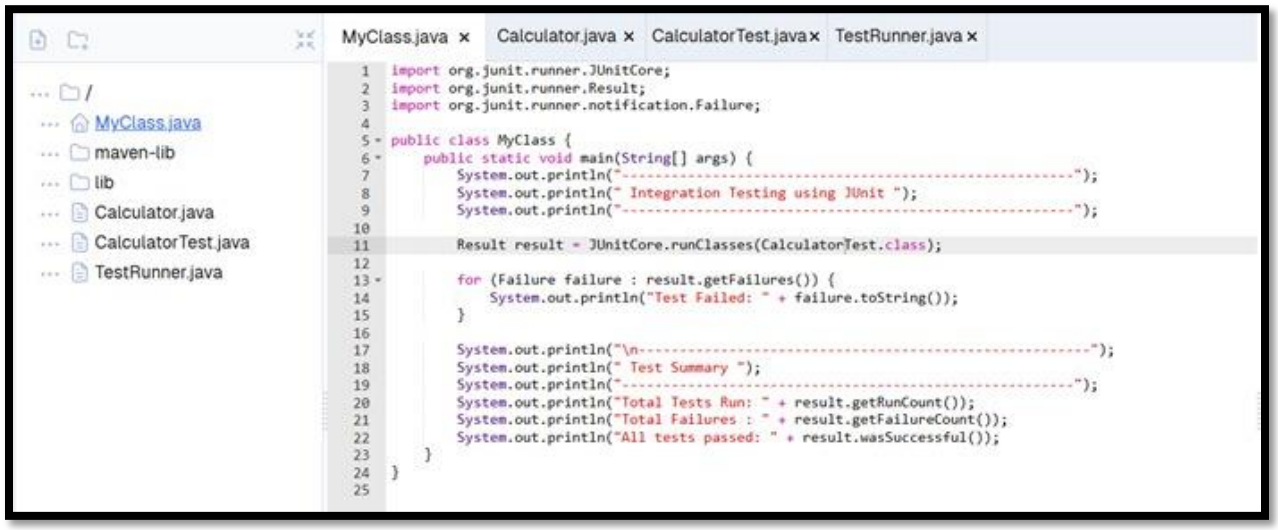


Fig.1 creating class

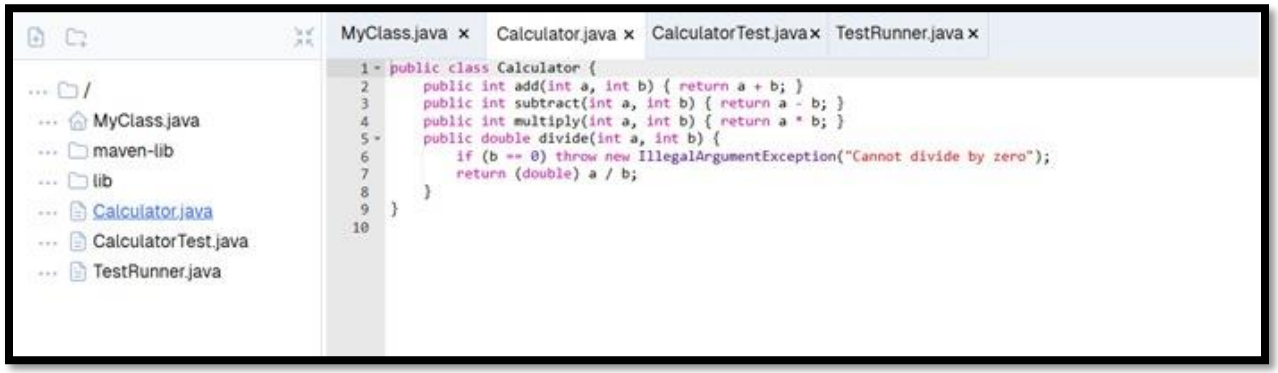


Fig.1 Calculator class

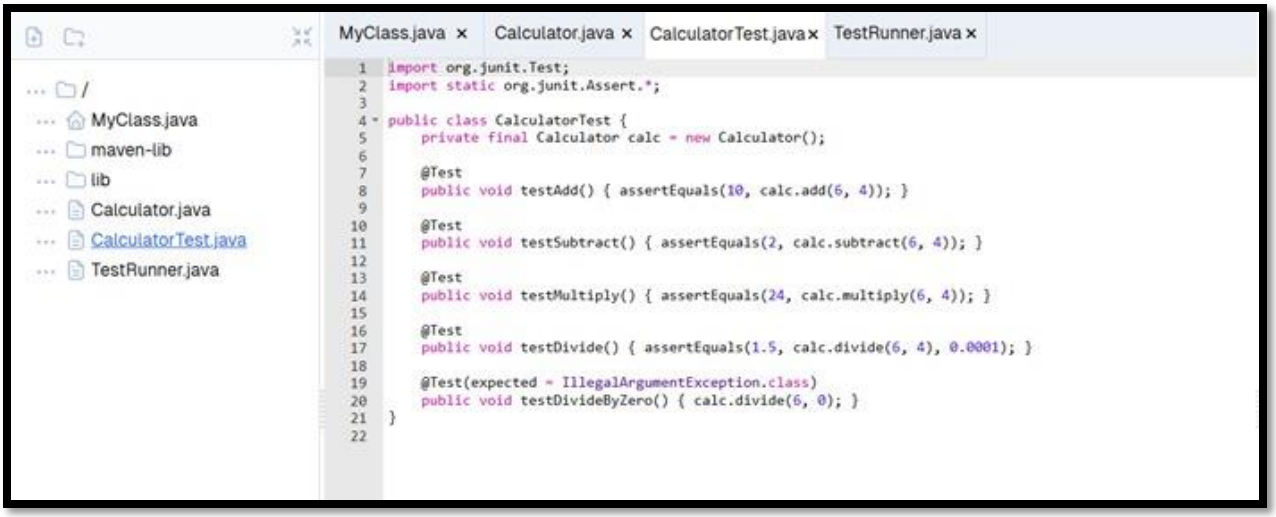
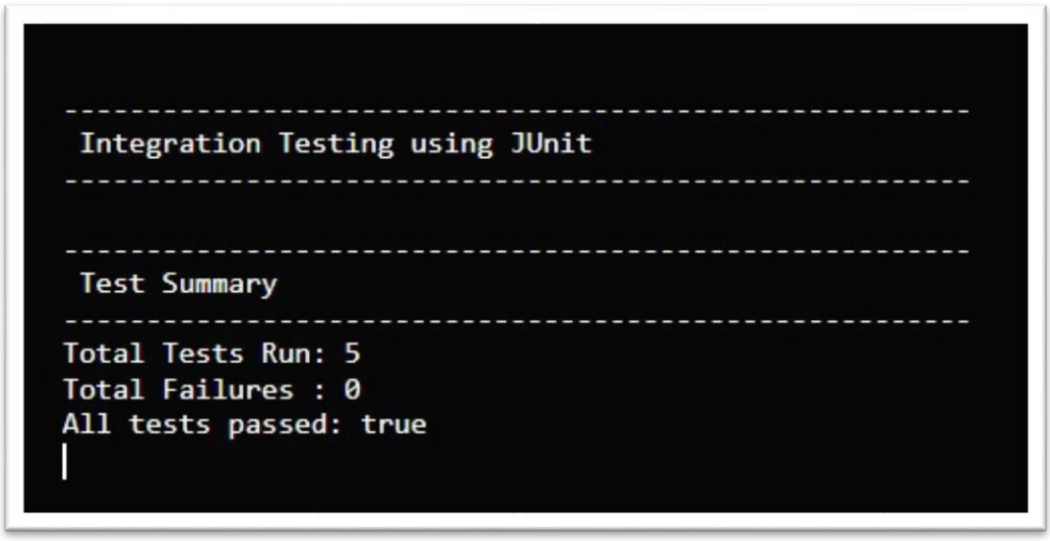


Fig.1 Calculator Test class

Results:



EXPERIMENT-06

Aim:

To design and document test cases for an Inventory Management System based on its system specifications.

Objective:

- To understand how to create test cases from functional requirements.
- To ensure that all critical features of the system are covered through testing.
- To identify valid and invalid input conditions for the system.

Theory:

A **test case** is a set of conditions or variables used to determine whether a system under test satisfies requirements and works correctly.

The **Inventory Management System (IMS)** helps track products, stock levels, sales, and reports. It typically includes modules such as:

1. User Login
2. Product Management (Add/Edit/Delete Product)
3. Stock Update
4. Report Generation

Designing test cases ensures each of these modules works as expected before integration or deployment.

Software and Hardware Requirements:

- System with Windows/Linux OS
- Java or Python environment (for backend testing)
- Browser (for web-based IMS)
- Excel / Google Sheets for recording test cases

Procedure:

1. Identify the key modules of the system.
2. Analyze the functional requirements for each module.
3. Prepare test cases in tabular format.
4. Execute each test case manually or through automation.

5. Record the results and mark as Pass or Fail.

Test Case Table:

Case ID	Test Scenario	Input Data	Expected Output	Actual Output	Status
TC01	Verify login with valid credentials	Username: **admin**Password: **1234**	Login successful, Dashboard displayed	Dashboard displayed	**Pass**
TC02	Verify login with invalid credentials	Username: **admin**Password: **xyz**	Error message should appear	Error message displayed	**Pass**
TC03	Add new product to inventory	Product ID: **P001**Name: **Mouse**Quantity: **50**	Product added successfully	Product added	**Pass**
TC04	Update product quantity	Product ID: **P001**Quantity: **+20**	Updated stock should be **70**	Stock updated	**Pass**
TC05	Generate monthly report in PDF format	Select option: **Monthly Report**	PDF report generated successfully	Report generated	**Pass**

Output:

A structured set of test cases covering all modules of the Inventory Management System.

Result:

Test cases were successfully designed and documented for the Inventory Management System. The system was tested against each case and performed as expected.

EXPERIMENT-07

Aim:

To design test cases and perform manual testing of the login module of Flipkart.

Objective:

- To understand the process of manual testing in a real-world web application.
- To verify that the Flipkart login functionality behaves correctly under various conditions.

Theory:

Manual Testing is the process of testing software manually without automation tools. The tester acts as an end-user and verifies all features.

The **Login Module** is a critical component of any application. It verifies user identity and grants access to authorized users only. Testing ensures that valid credentials allow access, and invalid ones are rejected.

Software Requirements:

- Web Browser (Chrome, Edge, Firefox)
- Stable Internet Connection
- Access to https://www.flipkart.com

Procedure:

1. Open the Flipkart login page.
2. Observe available fields and actions (Email, Password, Login button, Forgot Password).
3. Identify valid and invalid input scenarios.
4. Execute test cases one by one.
5. Record actual outcomes.

Test Case Table:

Test Case ID	Test Scenario	Input Data	Expected Output	Actual Output	Status
TC01	Valid login	Valid Email & Correct Password	Redirect to homepage	Redirected to homepage	**Pass**
TC02	Invalid password	Valid Email & Wrong Password	Error message displayed	Error message displayed	**Pass**
TC03	Empty fields	None	Validation message: “Enter Email/Password”	Message displayed	**Pass**
TC04	Click “Forgot Password?”	Click on Forgot?	Redirects to password recovery page	Redirected successfully	**Pass**
TC05	Invalid email format	Email: **user@123**	Validation message displayed	Validation message displayed	**Pass**

output:

Manual test execution screenshots and results confirming Flipkart’s login validation system works correctly.

Result:

Manual testing successfully validated all functionalities of the Flipkart login module

VALUE ADDED EXPERIMENT

EXPERIMENT - A

Aim:

Selenium API Setup and creation of Maven Project

Objective:

- To understand how to install and configure Selenium WebDriver.
- To learn how to create a Maven project for automation testing.
- To add Selenium dependencies using pom.xml.
- To verify successful setup by running a sample automated script.

Theory:

Selenium WebDriver is a widely used open-source automation tool for testing web applications. It allows automation of browser interactions such as clicking, navigation, form submission, etc. Maven is a build automation tool used mainly for Java projects. It simplifies dependency management (like adding Selenium JARs), project structure, and test execution.

Key Concepts:

- Selenium WebDriver: Automates browsers like Chrome, Firefox, Edge.
- Maven Project: Includes standard folder structure and pom.xml for dependencies. pom.xml: XML file where we add Selenium libraries and other plugins.

Software and Hardware Requirements:

- Windows / Linux / macOS
- JDK 8 or above
- Apache Maven
- IDE: IntelliJ IDEA / Eclipse / VS Code
- Browser Drivers: ChromeDriver / GeckoDriver
- Stable Internet Connection

Procedure:

1. Install Java Development Kit (JDK)

1. Download JDK from Oracle/OpenJDK.
2. Install and set environment variable:
   - JAVA_HOME → JDK installation path

- Add to PATH → %JAVA_HOME%\bin

Verify: java –version

2. Install Apache Maven

1. Download Maven from https://maven.apache.org
2. Extract the folder and set environment variable:

   - MAVEN_HOME → Maven folder
   - Add to PATH → %MAVEN_HOME%\bin

Verify: mvn –version

3. Create Maven Project

1. Open Eclipse/IntelliJ.
2. Go to File → New → Maven Project.
3. Select Archetype: maven-archetype-quickstart

1. Enter:

   - Group Id: com.selenium.demo
     Artifact Id: SeleniumTestProject

2. Click Finish.

Maven project structure will be created automatically:

src/main/java
src/test/java pom.xml

4. Add Selenium Dependency in pom.xml

Open pom.xml and add:

```
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.18.1</version>
  </dependency>
</dependencies>
```

Save the file → Maven downloads the Selenium JARs automatically.

5. Download Browser Driver (Example: ChromeDriver)

1. Check Chrome version → Help → About Chrome 2. Download matching driver: https://chromedriver.chromium.org/
3. Extract and place chromedriver.exe in project folder or system PATH.

6. Write Sample Selenium Script

Create a new class under src/test/java:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class FirstSeleniumTest {     public
static void main(String[] args) {
      System.setProperty("webdriver.chrome.driver", "chromedriver.exe");

      WebDriver driver = new ChromeDriver();        driver.get("https://www.google.com");

      System.out.println("Title: " + driver.getTitle());        driver.quit();
   } }
```

Output:

- Maven project created successfully.
- Selenium dependencies downloaded and configured.
- Sample WebDriver script executed and browser launched automatically. Google homepage opened and title printed on console.

Result:

Selenium WebDriver API was successfully set up, and a Maven project was created using Eclipse/IntelliJ. A sample automation script was executed to verify the successful configuration of Selenium in the Maven environment.

EXPERIMENT – B

Aim:

Perform Automation Testing using methods Maximise, Minimise and Window close **Objective:**

- To learn how to control browser windows using Selenium WebDriver.
- To understand the use of basic window handling methods.
- To write and execute a Selenium script that performs maximize, minimize, and close operations.
- To verify correct browser behavior through automation.

Theory:

Selenium WebDriver provides built-in methods to control web browser windows. These methods help in adjusting the browser size during automated testing for better visibility, screenshot handling, and layout verification.

Important Selenium Window Methods:

1. **driver.manage().window().maximize()** ○ Maximizes the browser to full screen. ○ Commonly used at the start of test execution.
2. **driver.manage().window().minimize()** ○ Minimizes the browser window to the taskbar. ○ Helps in background execution of scripts.
3. **driver.close()** ○ Closes the currently active browser window. ○ If multiple windows exist, only the focused one is closed.
4. **driver.quit()** ○ Closes all browser windows and ends WebDriver session.

These functions are useful for UI testing, responsive checks, and controlling visual screen space during automation.

Software Requirements:

- JDK 8 or above
- Apache Maven
- Selenium WebDriver
- ChromeDriver / GeckoDriver
- IDE: Eclipse / IntelliJ / VS Code
- Internet Connection

Procedure:

1. Create Maven Project

Use an existing Maven project or create a new one using: mvn

archetype:generate

2. Add Selenium Dependency in pom.xml

```
<dependency>
   <groupId>org.seleniumhq.selenium</groupId>
   <artifactId>selenium-java</artifactId>
   <version>4.18.1</version>
</dependency>
```

3. Download & Configure Browser Driver

- Download ChromeDriver from the official website.
- Add the driver executable path in your script.

4. Write Automation Script

Create a Java file under src/test/java:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class WindowMethodsTest {
   public static void main(String[] args) {

      System.setProperty("webdriver.chrome.driver", "chromedriver.exe");

      WebDriver driver = new ChromeDriver();

      // Open Website
      driver.get("https://www.google.com");

      // Maximize Window
driver.manage().window().maximize();
System.out.println("Window Maximized");

      // Wait for 3 seconds
      try { Thread.sleep(3000); } catch (Exception e) {}

      // Minimize Window
driver.manage().window().minimize();
System.out.println("Window Minimized");
```

```
    try { Thread.sleep(3000); } catch (Exception e) {}

    // Close Current Window
    driver.close();
    System.out.println("Window Closed");
} }
```

Output:

- Browser launched and navigated to Google.
- Window maximized successfully.
- Window minimized successfully.
- Browser window closed using Selenium WebDriver.

Result:

Automation testing was successfully performed using Selenium WebDriver window control methods. The script executed maximize, minimize, and close operations correctly, demonstrating effective browser manipulation through automation.

EXPERIMENT – C

Aim:

Identification of Web elements using X-Path and CSS.

Objective:

- To understand how to locate web elements using XPath and CSS selectors.
- To learn the difference between **Absolute XPath**, **Relative XPath**, and **CSS selector syntax**.
- To create a Selenium script that identifies and interacts with elements using these locators.
- To perform automation testing using the most commonly used locator strategies.

Theory:

Locators are used to identify elements on a webpage for automated testing. Selenium WebDriver supports multiple locators like ID, Name, Class, XPath, CSS Selector, LinkText, etc. Among these, **XPath** and **CSS Selectors** are the most powerful and widely used.

1. XPath (XML Path Language)

XPath is a syntax used to navigate through the HTML DOM.

Types of XPath:

A. Absolute XPath

- Starts from the root element (/html/...)
- Example:
- /html/body/div[1]/input
- Not recommended because it breaks if layout changes.

B. Relative XPath

- Starts with // and can search anywhere in DOM.
- Example:
- //input[@id='email'] ***Useful XPath Formats:***
- By attribute: //tagname[@attribute='value']
- By text: //button[text()='Login']
- Using contains(): //input[contains(@placeholder,'Email')]

- Multiple conditions:

//input[@type='text' and @name='username']

2. CSS Selectors

CSS selectors identify web elements faster than XPath in most browsers.

CSS Syntax:

- By ID:

#idname

- By class:

.classname

- By attribute:

input[name='email']

- Contains:

input[placeholder*='Email']

- Starts-with:

input[id^='user']

- Ends-with:

input[id$='name']

Software Requirements:

- Selenium WebDriver
- Java JDK
- Maven
- ChromeDriver / GeckoDriver
- IDE: Eclipse / IntelliJ **Procedure:**

1. Create Maven Project / Use Existing Selenium Project

2. Add Selenium Dependency in pom.xml

```
<dependency>
   <groupId>org.seleniumhq.selenium</groupId>
   <artifactId>selenium-java</artifactId>
   <version>4.18.1</version>
</dependency>
```

3. Write Selenium Script Using XPath & CSS

Create Java file under src/test/java:

```
import org.openqa.selenium.By; import
org.openqa.selenium.WebDriver; import
org.openqa.selenium.WebElement; import
org.openqa.selenium.chrome.ChromeDriver;

public class XPathCSSDemo {
```

```
    public static void main(String[] args) {

        System.setProperty("webdriver.chrome.driver", "chromedriver.exe");

        WebDriver driver = new ChromeDriver();
driver.get("https://www.facebook.com");

        driver.manage().window().maximize();

        // Using XPath
        WebElement emailXPath = driver.findElement(By.xpath("//input[@id='email']"));
emailXPath.sendKeys("testuser@example.com");

        // Using CSS Selector
        WebElement passCSS = driver.findElement(By.cssSelector("input#pass"));
passCSS.sendKeys("password123");

        // Login button using contains in XPath
        WebElement loginBtn = driver.findElement(By.xpath("//button[contains(@name,'login')]"));
loginBtn.click();

        driver.close();
    } }
```

Output:

- Webpage opened successfully.
- Email field identified using XPath.
- Password field identified using CSS Selector. Login button identified using advanced XPath ("contains").
- Script executed and browser closed successfully.

Result:

The experiment was successfully performed. Web elements were identified using **XPath** and **CSS Selectors** and interacted with through Selenium WebDriver. This demonstrates the ability to locate elements accurately for automation testing.

EXPERIMENT – D

Aim:

Perform Automation Testing of checkbox, radio button of application **Objective:**

- To identify and interact with checkbox and radio button elements using Selenium.
- To understand different WebDriver methods such as click(), isSelected(), and isEnabled().
- To write an automation script that selects and validates checkbox and radio button behavior.
- To verify UI selection states through automated testing.

Theory:

Checkboxes and radio buttons are common input elements in web applications.

1. Checkbox

- Allows **multiple selections**.
- Located using locators like **ID, Name, XPath, CSS Selector**.
- Selenium functions used:
  - click() → checks/unchecks
  - isSelected() → returns true if checkbox is checked

2. Radio Button

- Allows **only one option** to be selected from a group.
- Selenium functions used:
  - click() ○ isSelected()

Common Use Cases:

- Selecting user preferences
- Choosing gender
- Accepting terms & conditions
- Choosing payment options

Selenium WebDriver can easily automate selection and verification of these elements.

Software Requirements:

- Java JDK
- ChromeDriver / GeckoDriver
- Selenium WebDriver

- Apache Maven
- IDE (Eclipse / IntelliJ)
- Internet Connection

Procedure:

1. Create / Open Selenium Maven Project

2. Add Selenium Dependency in pom.xml

```
<dependency>
   <groupId>org.seleniumhq.selenium</groupId>
   <artifactId>selenium-java</artifactId>
   <version>4.18.1</version>
</dependency>
```

3. Write Automation Script for Checkbox & Radio Button

Create Java class under src/test/java:

```
import org.openqa.selenium.By; import
org.openqa.selenium.WebDriver; import
org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class CheckboxRadioTest {

   public static void main(String[] args) {

      System.setProperty("webdriver.chrome.driver", "chromedriver.exe");

      WebDriver driver = new ChromeDriver();
      driver.manage().window().maximize();

      driver.get("https://www.ironspider.ca/forms/checkradio.htm");

      // Checkbox example
      WebElement red = driver.findElement(By.xpath("//input[@value='red']"));
red.click();
      System.out.println("Red checkbox selected: " + red.isSelected());

      WebElement yellow = driver.findElement(By.xpath("//input[@value='yellow']"));
yellow.click();
      System.out.println("Yellow checkbox selected: " + yellow.isSelected());


      // Radio button example
      WebElement water = driver.findElement(By.xpath("//input[@value='water']"));
water.click();
      System.out.println("Water radio selected: " + water.isSelected());
```

```
        WebElement milk = driver.findElement(By.xpath("//input[@value='milk']"));
milk.click();
        System.out.println("Milk radio selected: " + milk.isSelected());

        driver.close();
    } }
```

Output:

- Browser launched successfully.
- Checkbox "Red" and "Yellow" were selected and verified using isSelected().
- Radio button "Water" and "Milk" were selected one after another.
- Selenium script executed and validated selection behavior of both elements. Browser closed successfully.

Result:

Automation testing of checkbox and radio button elements was successfully performed. The script demonstrated the ability to locate, select, and validate selection states using Selenium WebDriver functions.

EXPERIMENT – E

Aim:

Perform isDisplayed, isSelected and isEnabled in for validation in testing.

Objective:

- To understand the use of WebDriver validation methods in Selenium.
- To check whether elements are visible, enabled, or selected.
- To perform automated validation on checkboxes, buttons, and text fields.

- To write a Selenium script that uses all three validation methods.

Theory:

Selenium WebDriver provides validation methods to verify the state of elements during automation testing.

1. isDisplayed()

- Checks if the element is **visible on the webpage**.
- Returns **true** if displayed, else **false**.
- Useful for validating labels, buttons, forms, etc.

2. isEnabled()

- Checks if an element is **enabled** for interaction.
- Returns **true** if you can type/click on it.
- Common for testing disabled buttons, inputs, etc.

3. isSelected()

- Checks if a **checkbox or radio button** is selected.

- Returns **true** if the item is selected, else **false**.

These functions help validate UI behavior and ensure correct application functionality.

Software Requirements:

- Java JDK
- Selenium WebDriver
- ChromeDriver / GeckoDriver
- Maven
- Eclipse / IntelliJ
- Internet Connection

Procedure:

1. Create / Open Selenium Maven Project

2. Add Selenium Dependency in pom.xml

```
<dependency>
   <groupId>org.seleniumhq.selenium</groupId>
   <artifactId>selenium-java</artifactId>
   <version>4.18.1</version>
</dependency>
```

3. Write Selenium Script Using Validation Methods

Create a file under src/test/java:

```
import org.openqa.selenium.By; import
org.openqa.selenium.WebDriver; import
org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class ValidationMethodsTest {

   public static void main(String[] args) {

      System.setProperty("webdriver.chrome.driver", "chromedriver.exe");

      WebDriver driver = new ChromeDriver();
driver.manage().window().maximize();

      driver.get("https://www.ironspider.ca/forms/checkradio.htm");

      // Validate isDisplayed()
      WebElement header = driver.findElement(By.xpath("//h2[text()='Checkboxes']"));
      System.out.println("Header displayed: " + header.isDisplayed());

      // Checkbox - isSelected()
      WebElement red = driver.findElement(By.xpath("//input[@value='red']"));
System.out.println("Before clicking, red selected: " + red.isSelected());        red.click();
```

```
        System.out.println("After clicking, red selected: " + red.isSelected());

        // Radio button - isEnabled()
        WebElement water = driver.findElement(By.xpath("//input[@value='water']"));
        System.out.println("Water radio enabled: " + water.isEnabled());

        driver.close();
    } }
```

Output:

- isDisplayed() validated that page header is visible.
- isSelected() confirmed checkbox selection before and after click.
- isEnabled() verified radio button availability.
- Browser opened, validations executed, and window closed successfully.

Result:

Automation testing using **isDisplayed**, **isSelected**, and **isEnabled** methods was successfully performed. These validation functions confirmed the visibility, availability, and selection state of elements, ensuring correct UI behavior during testing.