DEERWALK INSTITUTE OF TECHNOLOGY

Tribhuvan University

Faculties of Computer Science

Bachelors of Science in Computer Science and Information Technology (BSc. CSIT)

Course: Software Engineering

Class of 2024/Semester: 6

A Lab report on:

Software Engineering

Submitted by:
Name: Namuna
Roll: 1016

Submitted to:
Shambhu Khanal
Lecturer
Department of Computer Science

Submission Date: 2025/05/22

LAB 1

Title:

Draw System Architecture Diagram, Use Case Diagram, and Class Diagram of a Library Management System.

Theory:

System Architecture Diagram

A System Architecture Diagram presents a high-level overview of a system's structure and its components. It illustrates how different modules and services interact, showcasing the flow of data and control within the system.

Key Elements:

- **Components**: Includes the user interface, catalog management, user management, borrowing and returning system, and database.
- **Connections**: Shows the communication paths between the user interface and backend services, and between backend services and the database.
- **Layers**: Represents logical divisions within the system, such as the presentation layer, business logic layer, and data layer.

Use Case Diagram

A Use Case Diagram illustrates the interactions between users (actors) and the system, focusing on the system's functionality from the user's perspective. It outlines the various operations that users can perform.

Key Elements:

- **Actors**: Roles that interact with the system, including Librarian, Member, and Administrator.
- **Use Cases**: Specific actions or functions provided by the system, such as "Search Books," "Borrow Book," "Return Book," "Add Book," "Remove Book," "Update Book Details," "Register Member," and "Generate Reports."
- **Associations**: Shows relationships between actors and their respective use cases.
- **Include/Extend**: Indicates shared or optional functionalities, such as logging actions or extending book borrowing periods.

Class Diagram

A Class Diagram represents the static structure of a system, detailing its classes, attributes, methods, and relationships among them.

Key Elements:

- **Classes**: Core entities within the system, such as Book, Member, Librarian, BorrowingTransaction, and Catalog.
- **Attributes**: Properties of each class, such as BookID, Title, Author, MemberID, Name, and TransactionDate.
- **Methods**: Functions or operations each class can perform, such as addBook(), removeBook(), borrowBook(), returnBook(), and registerMember().
- **Associations**: Relationships between classes, including one-to-many and many-to-many relationships, such as between Member and BorrowingTransaction or Book and Catalog.

Result

System Architechture

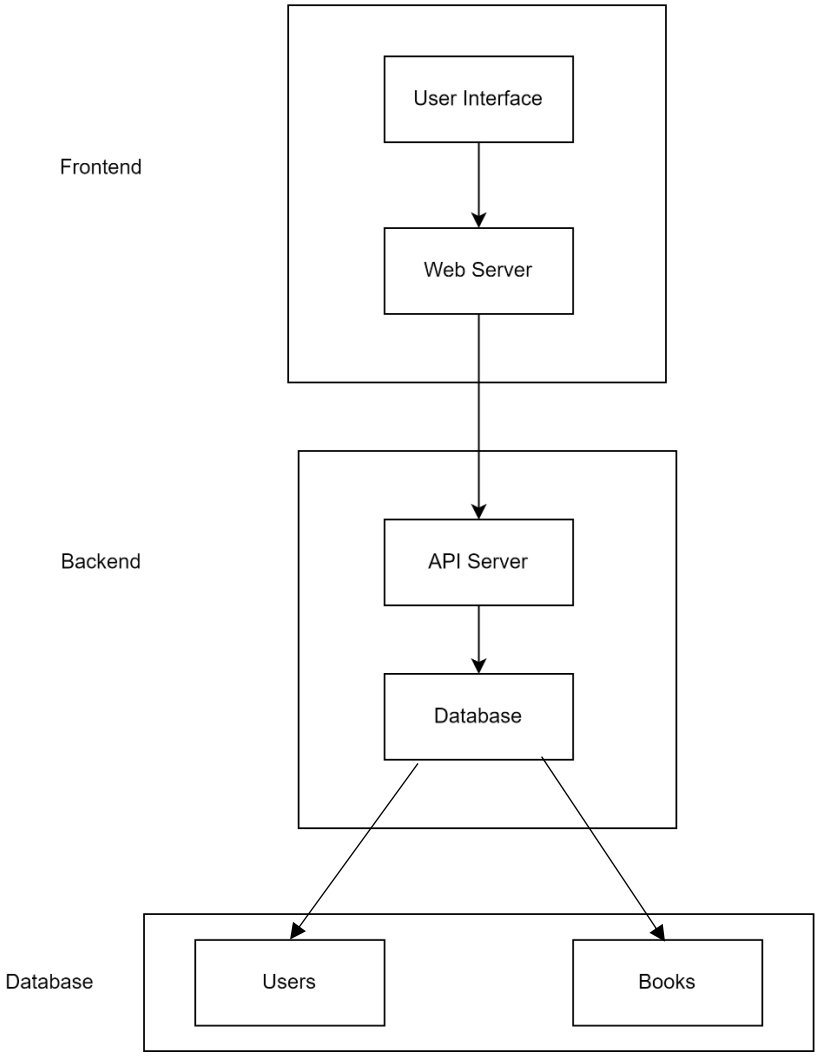


Figure 1: System Architecture Diagram of a Library Management System

Use case Diagram:

Figure 2: Use Case Diagram of a Library Management System

Class Diagram:

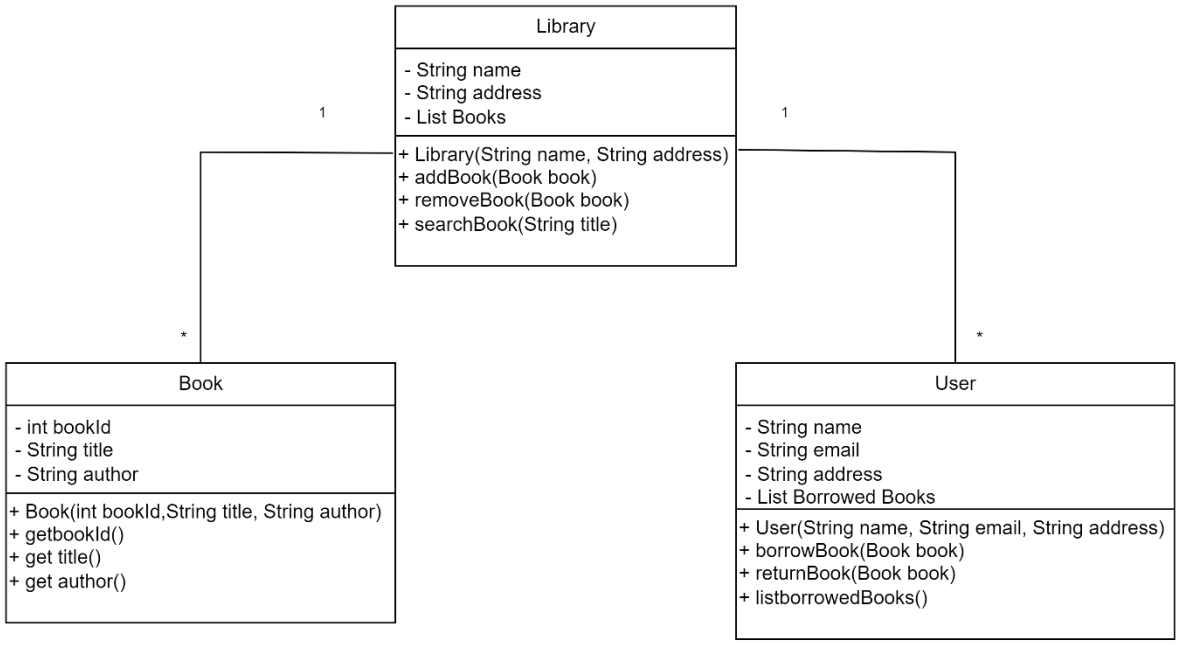


Figure 3: Class Diagram of a Library Management System

Conclusion:

In this way we created the System Architecture Diagram, Use Case Diagram and Class Diagram of Library Management System.

LAB 2

Title:

Draw Activity Diagram, Sequence Diagram and State Diagram of a Library Management System.

Theory:

Activity Diagram

An Activity Diagram is a graphical representation of workflows or processes within a system. It depicts the sequence of activities and the flow of control, showing various decision points and possible outcomes.

Key Elements:

- **Activities**: Represent individual tasks or actions to be performed.
- **Transitions**: Arrows indicating the flow of control from one activity to another.
- **Decision Points**: Diamonds representing branching points in the process where decisions are made based on conditions.
- **Start and End Nodes**: Circles indicating the beginning and end of the process flow.

Sequence Diagram

A Sequence Diagram is a type of interaction diagram that illustrates how objects interact in a particular scenario or use case over time. It shows the sequence of messages exchanged between objects, focusing on the chronological order of interactions.

Key Elements:

- **Objects**: Represent entities or components participating in the interaction.
- **Lifelines**: Vertical dashed lines indicating the existence of objects over time.
- **Messages**: Arrows indicating the flow of communication between objects.
- **Activation Bars**: Horizontal bars representing the duration of an object's involvement in an interaction.

State Diagram

A State Diagram, also known as a State Machine Diagram, represents the various states and transitions that an object can undergo in response to events. It models the behavior of an object by specifying its possible states and the events that trigger state transitions.

Key Elements:

- **States**: Represent the different conditions or situations in which an object can exist.
- **Transitions**: Arrows indicating the change of state triggered by events.

- **Events**: Stimuli or occurrences that cause a transition from one state to another.
- **Actions**: Activities performed when transitioning between states.

Result:

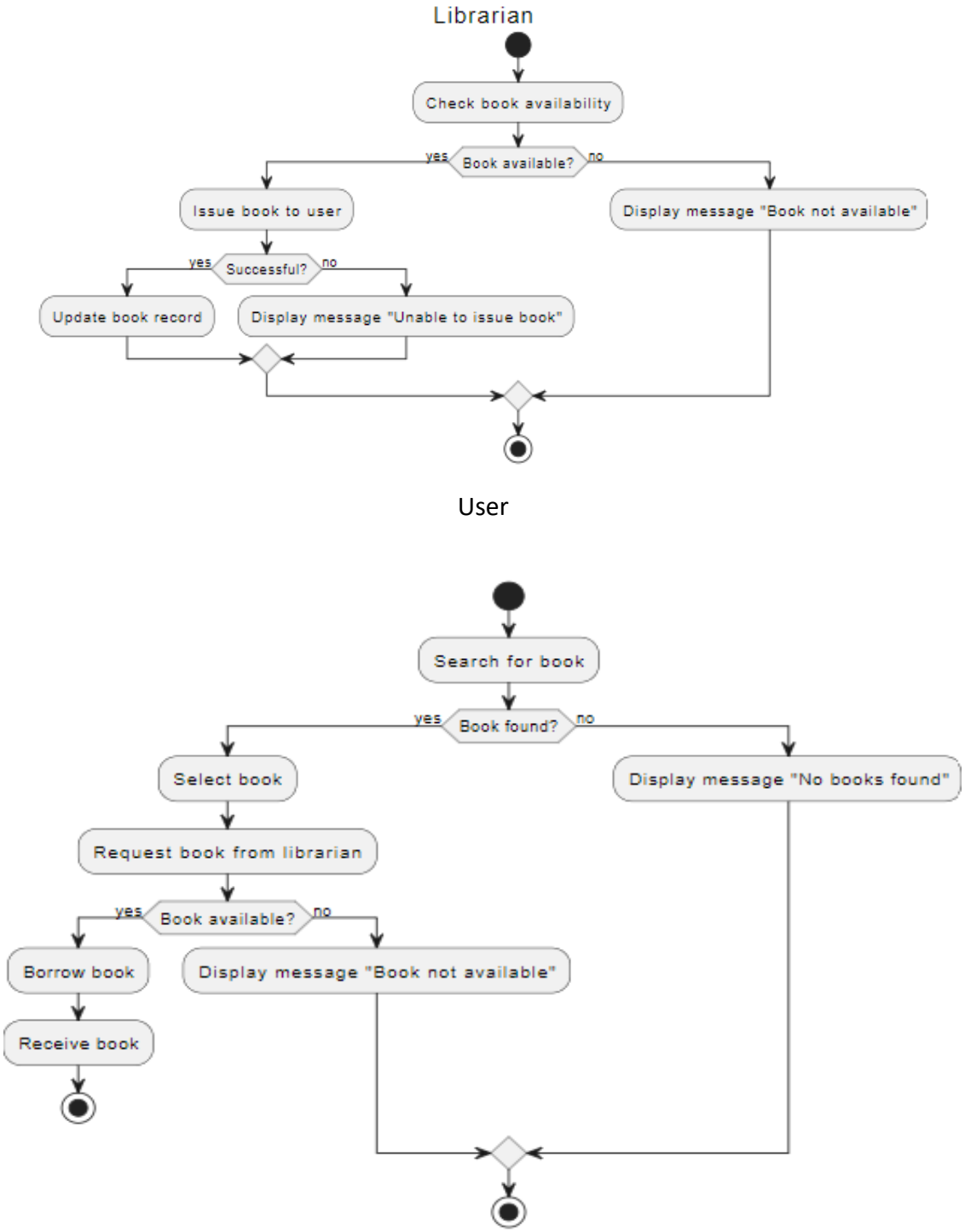


Figure 4: Activity Diagram of a Library Management System

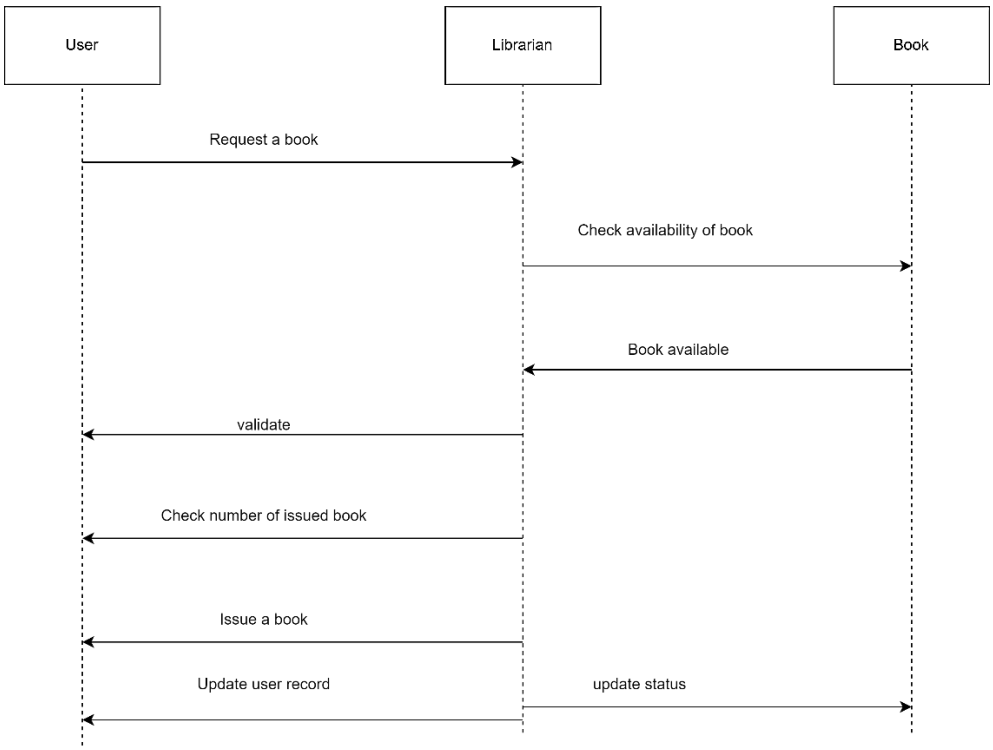


Figure: Sequence Diagram of a Library Management System

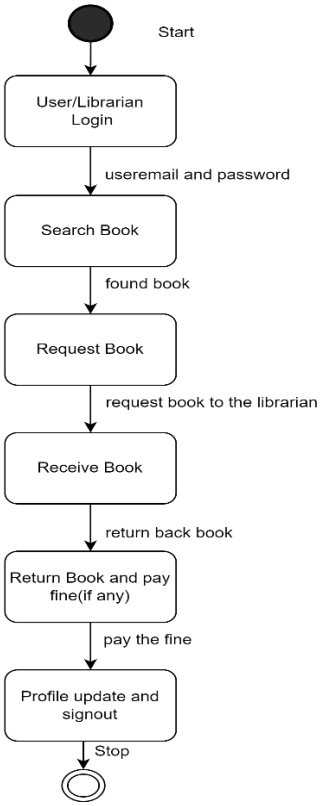


Figure: State Diagram of a Library Management System

Conclusion:

In this way we created Activity Diagram, Sequence Diagram and State Diagram of a Library Management System.

LAB 3

Title:

Make a Test Case and show the output of any one of the features of a system in Junit or

Selenium testing framework.

Theory:

Unit testing is a crucial practice in software development that involves the examination of individual components within a software application. These components, which can include methods, functions, classes, or modules, are tested in isolation to ensure they perform as intended and produce the expected outputs for specific inputs. By conducting unit tests, developers can identify and rectify any issues or bugs within these components early in the development cycle, thereby enhancing the overall quality and reliability of the software system. Unit testing helps to verify the correctness of each component's behaviour, improves code maintainability, facilitates easier debugging, and ultimately contributes to the successful delivery of high-quality software products.

JUnit is a widely used testing framework for Java, offering annotations, assertions, and test runners to streamline the creation and execution of unit tests, thereby automating, and standardizing the testing process, and promoting efficient software development practices.

Test cases representing specific scenarios or conditions, are organized into test suites, which group related tests together, facilitating efficient testing management and organization**.** This approach enhances the maintainability and reliability of the software system.

The steps involved are:

1. Ensure that a working development environment with JUnit installed. Set up project and

include the necessary JUnit libraries.

2. Choose the specific system feature or functionality to test. Define the expected behavior and outcomes for this feature.

3. Create a new JUnit test class for the feature for testing. Write test methods (test cases) that cover different scenarios related to the feature.

4. Set up any necessary preconditions. Execute the feature or functionality being tested. Verify that the actual results match the expected outcomes.

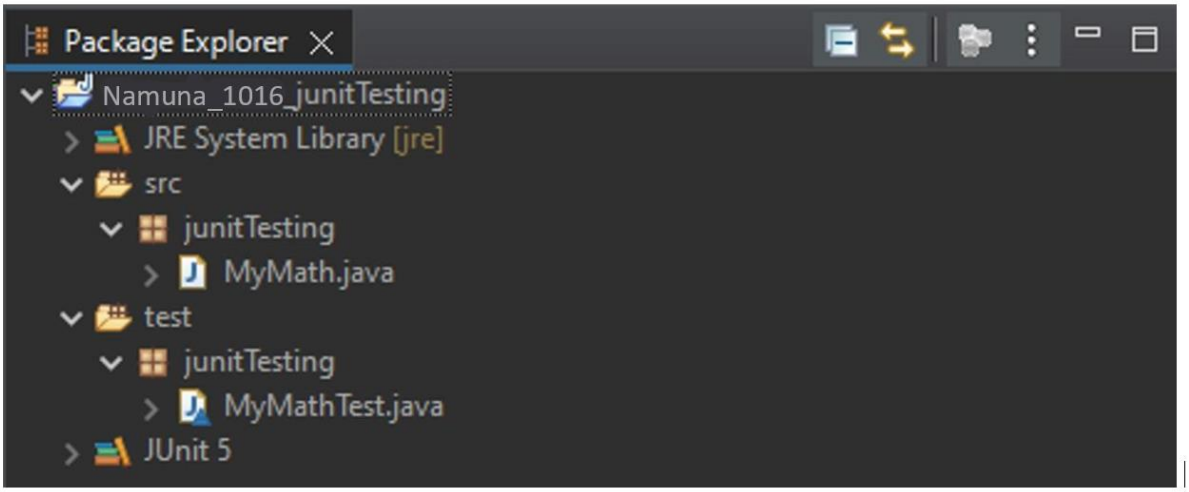
5. Use JUnit assertions to validate the results. Compare actual values with expected values.

6. Test both positive and negative scenarios.

7. Execute JUnit test suite. Observe the test results (pass/fail).

Result:

TID	Test Case	Input	Expected Output	Actual Output	Result
1.	Enter valid array of integer numbers	{50, 10, 30}	900	90	Fail
2.	Enter a valid array of integer numbers	{50, 10, 30}	90	900	Pass



MyMath.java: -

```
package junitTesting;

public class MyMath {
 public int calculateSum(int [] numbers) {
          int sum = 0;
          for(int number:numbers) {
          sum += number;
          }
            return sum;
   }
}
```

1. MyMathTest.java: -

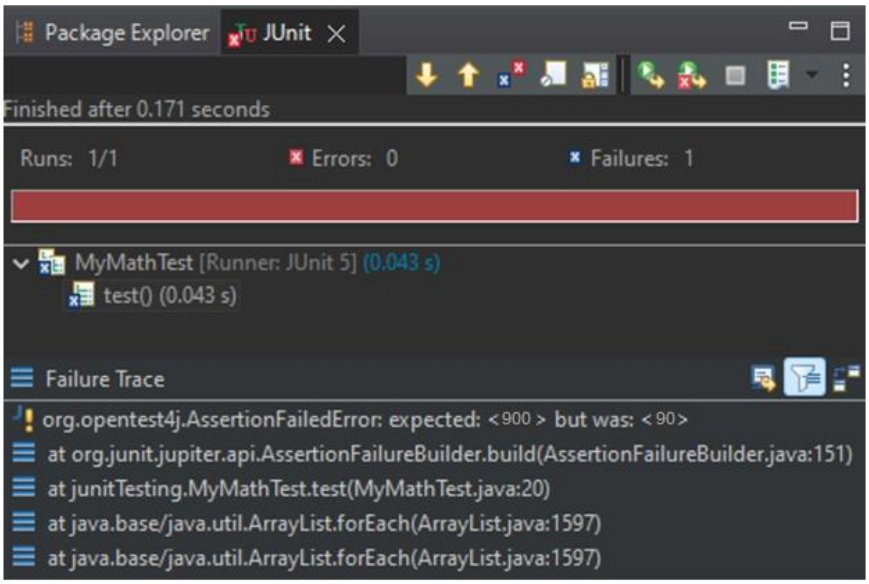
```
package junitTesting;

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

```
class MyMathTest {

	@Test
	    void test() {
	        int[] numbers = {50,10,30};
	       MyMath math = new MyMath();
	          int result = math.calculateSum(numbers);
	         System.out.println(result);
	        int expectedResult = 900;
	           assertEquals(expectedResult, result);
	}
}
```

Test output: -



2. MyMathTest.java: -

```
package junitTesting;

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

class MyMathTest {

 @Test
 void test() {
```

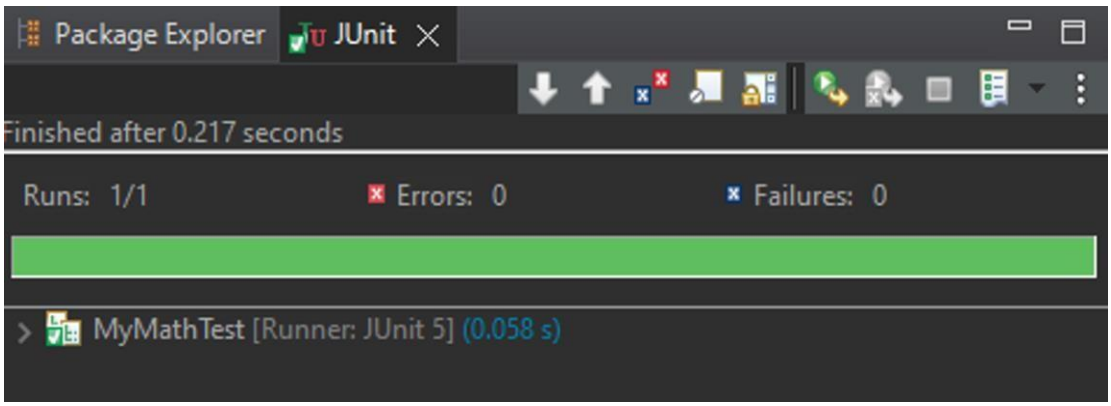
```
        int[] numbers = {50,10,30};

    MyMath math = new MyMath();
        int result = math.calculateSum(numbers);

        System.out.println(result);

    int expectedResult = 90;
        assertEquals(expectedResult, result);
  }
}
```

Test output: -



Conclusion:

In this way, test cases were made, Junit testing framework is used to test a system feature.