Experiment#		Student ID	
Date		Student Name	[@KLWKS_bot] THANOS

5. SOLID Design Principles and Test-Driven Development.

Aim/Objective: To analyse the implementation of SOLID Principles & Test-Driven Development (TDD) for the real-time scenario.

Description: The student will understand the concept of SOLID Principles & Test-Driven Development (TDD).

Pre-Requisites: Classes and Objects in JAVA

Tools: Eclipse IDE for Enterprise Java and Web Developers

Pre-Lab:

1) Elaborate about each letter in SOLID Principles.

1. **S - Single Responsibility Principle (SRP)**
   - A class should have only **one reason to change** (one responsibility).
   - Example: Separate `SalaryCalculator` and `ReportGenerator` instead of merging them into one class.
2. **O - Open/Closed Principle (OCP)**
   - A class should be **open for extension but closed for modification.**
   - Example: Use **polymorphism** instead of modifying a class to add new functionality.
3. **L - Liskov Substitution Principle (LSP)**
   - Subtypes should be **replaceable** for their base types **without breaking functionality.**
   - Example: An `Ostrich` (a non-flying bird) should not inherit a `fly()` method.
4. **I - Interface Segregation Principle (ISP)**
   - **No class should be forced to implement unused methods.**
   - Example: Use multiple smaller interfaces (`Workable`, `Eatable`) instead of one large interface (`Worker`).
5. **D - Dependency Inversion Principle (DIP)**
   - **Depend on abstractions, not concrete classes** (loose coupling).
   - Example: Use an interface (`Database`) instead of directly depending on `MySQLDatabase`.

In-Lab:

1) Develop a Student Information System (SIS) that adheres to SOLID design principles, where you need to manage the Student Information System (SIS) with the incorporation of the concepts of Classes, Objects, Constructors, Interfaces and inheritance.

Requirements

1. Student Management:
    - Manage student details such as name, ID, and courses enrolled.
2. Course Management:
    - Manage course details such as course ID, course name, and list of enrolled students.
3. Enrollment Management:
    - Handle the enrollment of students in courses.

Applying SOLID Principles

1. Single Responsibility Principle (SRP):
    - Each class should have one responsibility. For example, a Student class handles student details, and a Course class handles course details.
2. Open/Closed Principle (OCP):
    - The system should be open for extension but closed for modification. Use interfaces and abstract classes to allow for new types of students or courses without modifying existing code.
3. Liskov Substitution Principle (LSP):
    - Subtypes must be substitutable for their base types. Ensure derived classes can be used interchangeably with their base classes.
4. Interface Segregation Principle (ISP):
    - Create specific interfaces for different functionalities, ensuring clients only depend on the interfaces they use.
5. Dependency Inversion Principle (DIP):
    - High-level modules should depend on abstractions, not concrete implementations. Use dependency injection to manage dependencies.

Procedure/Program:

```
import java.util.*;

class Student {
   private String name, id;
   private List<Course> courses = new ArrayList<>();

   public Student(String name, String id) {
    this.name = name; this.id = id;

   }
   public String getName() {
    return name;

   }
   public void enroll(Course course) {
    courses.add(course); course.addStudent(this);

   }
   public List<Course> getCourses() {
    return courses;

   }
}

class Course {
   private String courseId, courseName;
   private List<Student> students = new ArrayList<>();

   public Course(String courseId, String courseName) {
    this.courseId = courseId; this.courseName = courseName;
    }
   public void addStudent(Student student) {
    students.add(student);

   }
```

```
public String getCourseName() {
    return courseName;
    }
}

interface Enrollment {
 void enroll(Student s, Course c);

}
class EnrollmentManager implements Enrollment {
 public void enroll(Student s, Course c) {
  s.enroll(c);
 }

}

public class Main {
   public static void main(String[] args) {
      Student s1 = new Student("John Doe", "S001");
      Course c1 = new Course("C001", "Introduction to Java");
      new EnrollmentManager().enroll(s1, c1);
      System.out.println("Student: " + s1.getName() + " enrolled in: " +
s1.getCourses().get(0).getCourseName());
   }
}
```

OUTPUT

```
Student: John Doe enrolled in: Introduction to Java
```

2) During an NCC parade, a large number of cadets participated, and the leader instructed them to stand in a line sorted alphabetically by their names for easier identification. After organizing the cadets in alphabetical order, the leader wants to verify whether they are indeed standing in the correct sorted order.

   You as the leader, construct a JUnit test program to validate the cadet's arrangement and ensure it aligns with the expected alphabetical sorting order. The test program should include various scenarios such as an empty list of cadets, a single cadet, multiple cadets with different names, and cadets with identical names and also assertions within the unit test to verify the correctness of the cadet's alphabetical arrangement.

   Procedure/Program:

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

import java.util.Arrays;
import java.util.List;

public class CadetSortingTest {

   @Test
   public void testEmptyList() {
      List<String> cadets = Arrays.asList();
      assertTrue(isSorted(cadets));
   }

   @Test
   public void testSingleCadet() {
      List<String> cadets = Arrays.asList("Alice");
      assertTrue(isSorted(cadets));
   }

   @Test
   public void testMultipleCadetsSorted() {
      List<String> cadets = Arrays.asList("Alice", "Bob", "Charlie");
      assertTrue(isSorted(cadets));
   }
```

```
@Test
  public void testMultipleCadetsUnsorted() {
    List<String> cadets = Arrays.asList("Charlie", "Alice", "Bob");
    assertFalse(isSorted(cadets));
  }

  @Test
  public void testCadetsWithIdenticalNames() {
    List<String> cadets = Arrays.asList("Alice", "Alice", "Bob");
    assertTrue(isSorted(cadets));
  }

  private boolean isSorted(List<String> cadets) {
    for (int i = 1; i < cadets.size(); i++) {
      if (cadets.get(i - 1).compareTo(cadets.get(i)) > 0) {
        return false;
      }
    }
    return true;
  }
}
```

✓ **Data and Results:**

Data

Cadet names are arranged alphabetically to verify correct sorting order.

Result

The JUnit test confirms the correctness of alphabetical cadet arrangement.

✓ **Analysis and Inferences:**

Analysis

Various test cases ensure sorted, unsorted, empty, and duplicate scenarios.

Inferences

The implemented sorting verification method effectively validates cadet ordering.

VIVA-VOCE Questions (In-Lab):

1) State about "Write Unit Tests" principle in Clean Coding Techniques.

Write Unit Tests ensures small code units are tested automatically, improving code quality and preventing bugs.

2) Discuss about Single Responsibility Principle (SRP).

Single Responsibility Principle (SRP) states that a class should have only one reason to change, making it easier to maintain.

3) Illustrate the difference between Assertions and Annotations.

Assertions validate conditions in tests (`assertEquals()`, `assertTrue()`), while **Annotations** provide metadata (`@Test`, `@BeforeEach`).

4) List the Assertions in JAVA Test Driven Development (TDD).

Java TDD Assertions: `assertEquals()`, `assertNotNull()`, `assertTrue()`, `assertFalse()`, `assertThrows()`, etc.

5) What is the problem of not having Interface segregation principle in SOLID principles>

Without **Interface Segregation Principle (ISP)**, interfaces become bloated, forcing classes to implement unnecessary methods, reducing flexibility.

Post-Lab:

1) Write a Test-Driven Development program to accept the password when the length of it should be between 5 to 10 characters (“Password validator”)

Input: Abc123

Output: Valid

password: accepted

Procedure/Program:

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

public class PasswordValidator {

   public static String validatePassword(String password) {
      if (password.length() >= 5 && password.length() <= 10) {
         return "Valid password: accepted";
      } else {
         return "Invalid password: rejected";
      }
   }

   @Test
   public void testValidPassword() {
      assertEquals("Valid password: accepted", validatePassword("Abc123"));
   }
```

```
@Test
   public void testShortPassword() {
      assertEquals("Invalid password: rejected", validatePassword("Abc"));
   }

   @Test
   public void testLongPassword() {
      assertEquals("Invalid password: rejected",
validatePassword("Abc1234567"));
   }

   @Test
   public void testExactLength() {
      assertEquals("Valid password: accepted", validatePassword("Abc12"));
      assertEquals("Valid password: accepted",
validatePassword("Abc1234567"));
   }
}
```

Experiment#		Student ID	
Date		Student Name	[@KLWKS_bot] THANOS

✓ **Data and Results:**

Data

The program validates passwords based on length constraints between 5 to 10 characters.

Result

Passwords meeting length criteria are accepted; others are rejected as invalid.

✓ **Analysis and Inferences:**

Analysis

The test cases cover valid, short, long, and exact-length password scenarios.

Inferences

The implementation correctly enforces password length restrictions using TDD principles.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.