Experiment#		Student ID	
Date		Student Name	

8. 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.

S: Single Responsibility Principle (SRP): A class should have only one reason to change.

O: Open/closed Principle (OCP): Classes should be open for extension but closed for modification.

L: Liskov Substitution Principle (LSP): Subtypes should be substitutable for their base types without altering the corectness of the program.

I: Interface segregation Principle (ISP): Integrfaces should be specific to the clients that use them.

D: Dependency Inversion Principle (DIP): High-level modules shouldnot depend on low-level modules but on abstractions.

Experiment#		Student ID	
Date		Student Name	

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:

```
public class Student {
    private String name;
    prirate String studentId;
```

```
public Student (String name, String studentId) {
    this.name = name;
    this.studentId = studentId;
} public String getName () {
    return name;
} public String getStudentId () {
    return studentId;
    }
} public class Course {
    private String courseId;
    private String courseName;
    private List<Student> enrolledstudents;
public Course (String courseId, String courseName) {
    this.courseId= courseId;
    this.courseName = courseName;
    this.enrolledstudents = new ArrayList<> ();
} public String getCourseId () {
    return courseId;
}
```

```
public String getCourseName() {
    return courseName;
} public List<Student> getEnrolledStudents() {
    return enrolledStudents;
} public void addstudent(Student student) {
    enrolledStudents.add(student);
} public void removestudent(Student student) {
    enrolledstudents.remove(students);
}} public class EnrollmentManager {
public void enrollStudent(Course course, Student student){
    course.addStudent(student);
} public void withdrawStudent(Course course, Student
                                        student) {
    course.removeStudent(student);
}} public interface Notification {
    void notifyStudent(Student student, String message);
} public class EmailNotification implements Notification{
public void notifyStudent(Student student, String
                                        message) {
```

```
System.out.println("Sending email to " + student.
getName() + ": " + message);
}}
public class SMSNotification implements Notification{
public void notifyStudent(Student student, String
                                    message) {
System.out.println("Sending SMS to " + student.
  getName()+ ": " + message);
}} public abstract class BaseNotification{
  public abstract void notifyStudent(Student student,
                                    String message);
}
public class EmailNotification extends BaseNotification
 {
public void notifyStudent(Student student, String
                                    message) {
System.out.println("Email sent to " + student.getName
() + ": " + message);
  }}
```

```
public interface StudentOperations {
void addStudent (Student student);
void removeStudent (Student student);
} public interface CourseOperations {
  void addCourse (Course course);
  void removeCourse (Course course);
} public class StudentManager implements Student
                                        Operations {
private List<Student> student=new ArrayList<> ();
public void addStudent (Student student) {
  students.add (student);
} public void removeStudent (Student student) {
  students.remove (student);
  } }
public class CourseManager implements CourseOperations {
  private List< Course> courses=new Arraylist<>();
public void addCourse (Course course) {
    courses.add (course);
} public void removeCourse (Course course) {
    courses.remove (course); } }
```

```
Public class NotificationService {
    private Notification notification;
Public NotificationService (Notification notification){
    this.notification = notification;
} public void sentNotification (Student student,
                                String message) {
    notification.notifyStudent (student, message); } }
public class StudentInfoSystemClient {
    public static void main (String[] args) {
    Student student1 = new student ("Chandana", "S001");
    Student student2 = new Student ("RajaSri", "S002");
    Course course1 = new Course ("C001", "ADOP");
    Course course2 = new Course ("C002", "DBMS");
    EnrollmentManager enrollmentManager = new
                                EnrollmentManager ();
    enrollment Manager.enrollStudent (course1, student1);
    enrollmentManager.enrollStudent (course2, student2);
    }
}
```

Experiment#		Student ID	
Date		Student Name	

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:

```
public class Cadet {
private string name;
public Cadet (String name) {
    this.name = name;
} public String getName (){
    return name;
} public String tostring () {
    return name;
} } import java.util. Collections;
    import java.util. List;
  public class CadetSorter {
public static void sortCadets (List<Cadet> cadets){
Collections.sort (cadets, (c1, c2) → c1.getName ().
                    CompareTo (c2.getName ()));
```

```
}} public class CadetSorterTest {
public void testEmptyList () {
List<Cadet> cadets=new ArrayList<>();
CadetSorter. sort Cadets (cadets);
assertTrue (cadets.isEmpty (), "List should be
                                empty");
} public void testSingle Cadet () {
List<Cadet> cadets = Arrays. aslist (new
                    Cadet ("Chandana"));
Cadet Sorter . sort Cadets (cadets);
assertEquals ("Chandana", cadets. get(0). getName(),
"The single cadet should be in the list");
    }
}
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

Running OddEvenCheckerTest

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0,

Time elapsed: 0.123 sec

Results :

Tests run: 2 Failures: 0 Errors: 0 skipped: 0

✓ **Analysis and Inferences:**

The code gives the result of the number which is even or odd, if it shows no failure then the program ran successfully.

Evaluator Remark (if Any):	Marks Secured: _____out of 50
Evaluator MUST ask Viva-voce prior to signing and posting marks for each experi	Signature of the Evaluator with Date

Course Title	Advanced Object-Oriented Programming	ACADEMIC YEAR: 2024-
Course Code	23CS2103A & 23CS2103E	

Experiment#		Student ID	
Date		Student Name	

VIVA-VOCE Questions (In-Lab):

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

Unit testing principles demand that a good test is: Easy to write. Developers typically write lots of unit tests to cover different cases and aspects of application's behaviour, so it should be easy to code all test routines.

2) Discuss about Single Responsibility Principle (SRP).

SRP is the concept that any single object in object-oriented programming should be made for one specific function.

3) Illustrate the difference between Assertions and Annotations.

Assertions: These are used for debugging and testing purposes.

Annotations: These are used for documentation, code generation, configuration and runtime behavior modification.

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:

```
public class PasswordValidator {
public static boolean isValidPassword (String
                                    password) {
int length = password.length();
  return length >=5 && length <= 10;
}} import static.org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
public class PasswordValidatorTest {
 public void testValidPassword () {
assertTrue (PasswordValidator.isValidPassword ("Abc123"));
 } public void testShortPassword () {
assertFalse (PasswordValidator.isValidPassword ("Ab1"));
 } public void testLongPassword (){
assertFalse (PasswordValidator.isValidPassword
                                ("Abcdefghijkl"));
}
```

2) Write a Java program to check whether the given number is odd or even and do unit testing in JUnit.

Procedure/Program:

```
public class OddEvenChecker {
public static String checkOddEven (int number) {
if (number % 2 == 0) {
return "Even";
} else {
return "Odd";
}}
public static void main (String[] args) {
int number = 25;
String result = checkOddEven (number);
System.out.println (number + " is " + result);
}} import static org.junit.jupiter.api.Assertion;
import org.junit.jupiter.api.Test;
Public class OddEvenCheckerTest {
public void testEvenNumber () {
assertEquals ("Even", OddEvenChecker.checkOddEven(20));
} public void testOddNumber () {
assertEquals ("odd", OddEvenChecker.checkOddEven(15));
}
}
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

testSingleCadet:

The single cadet should be in the list.

✓ **Analysis and Inferences:**

The results from these tests indicate that the 'CadeSorter' class is functioning correctly across a range of scenarios: Handling empty and single item lists properly. correctly sorts lists with unique and identical names.