# Department of Software Engineering

**SE-210: Software Design and Architecture**

**Class: BESE 12 AB**

**Lab 06: Object Oriented Design Principles (Class Level)**

**CLO 2: Select appropriate design pattern and architectural pattern for a given problem.**

# Instructor: Zunera Zahid

# 

**Lab 04: Object Oriented Design Principles (Class Level)**

**Introduction**

Students have learned the theoretical concepts of class level object oriented design principles in lectures. In this lab, students will learn how to identify and fix pieces of codes where these principles are violated.

**Objectives**

After the completion of this lab, students will be able to identify which parts of software are violating class level object oriented principles and how to fix them.

**Tools/Software Requirement**

* Papyrus/Rational Rose

**Description**

It’s a process of planning a software system where objects will interact with each other to solve specific problems. The saying goes, "Proper Object oriented design makes a developer's life easy, whereas bad design makes it a disaster."

**Lab Task**

**Task 1**

We have learned five principles of class level object oriented design formally known as SOLID principles. The five principles are:

* The Single Responsibility Principle (SRP)
* The Open Closed Principle (OCP)
* The Liskov Substitution Principle (LSP)
* Interface Segregation (ISP)
* Dependency Inversion (DIP)

Your task:

* For each of the principles, you have to give two coding examples where the principles are violating.
* Given a brief explanation of why the piece of code is violating the principle.
* A refactored version of the code, where the principles are respected.
* Given a brief explanation of why the refactored version of the code is respecting the principle.

**Important Note:** You are not required to write a fully functional code. Only write enough code which can make your point.

**Example**

**Liskov Substitution Principle**

**Violating Code:**

class Rectangle{  
 protected int width;  
 protected int height;  
 public void setWidth(int w){width = w;}  
 public void setHeight(int h){height = h;}  
 public int getWidth(){return width;}  
 public int getHeight(){return height;}  
 public int getArea(){return width \* height;}  
}

class Square extends Rectangle {  
 public void setWidth(int w){  
 width = w;  
 height = w;  
 }

public void setHeight(int h){  
 width = h;  
 height = h;   
 }  
}

**Why the code is violating LSP**

The class Square is extending the Rectangle class. Mathematically, there exist a strong relationship between a rectangle and square (square being a special form of rectangle where width and height are equal). However, behaviorally a square object is completely different from a rectangle object. In the above code, the setWidth and setHeight overridden functions of Square class are changing the behavior of the same functions in the parent class. Thus the subclass is no more substitutable for the base class, which violates Liskov Substitution Principle.

**Refactored Code**

abstract class Shape{  
 public void setWidth(int w){};   
 public void setHeight(int h){};  
}

class Rectangle extends Shape{  
 protected int width;  
 protected int height;  
 public void setWidth(int w){width = w;}  
 public void setHeight(int h){height = h;}  
 public int getWidth(){return width;}  
 public int getHeight(){return height;}  
 public int getArea(){return width \* height;}  
}

class Square extends Shape {  
 protected int width;  
 protected int height;  
 public void setWidth(int w){  
 width = w;  
 height = w;  
 }

public void setHeight(int h){  
 width = h;  
 height = h;  
 }  
 public int getWidth(){return width;}  
 public int getHeight(){return height;}  
 public int getArea(){return width \* height;}  
}

**Why the refactored code is respecting LSP**

The common behavior of setting width and height of the square and rectangle classes is generalized in the shape class. Both square and rectangle class now extend the shape class. Each class now defines it’s own behavior of setting width and height and the behaviors are no more contradicting.

|  |
| --- |
| **Liskov’s substitution principle:**  **Violated Code:**  **public** **class** Reptiles{  **public** **void** walks();  **public** **void** eats();  **public** **void** Hunts();  }  **public** **class** crocodile **extends** Reptiles  {  *//implement all functions*  }  **public** **class** snake **extends** Reptiles  {  *//implement all functions*  }  **Violation:**  In this case parent class has the function walk but some reptiles slither. For example snakes. So, this code is incorrect logically and reused unnecessarily.  **Refactored Code:**  **public** **class** Reptiles{  **public** **void** eats();  **public** **void** Hunts();  }  **public** **class** walkingReptiles()  {  *//inherits all functions from parent class*  **public** **void** walks();  }  **public** **class** SlitheringReptiles()  {  *//inherits all functions from parent class*  **public** **void** walks();  }  **public** **class** crocodile **extends** walkingReptiles{  *//implement all functions*  }  **public** **class** snake **extends** SlitheringReptiles{  *//implement all functions*  }  **Refactoring:**  In this code, using the LSP the extra function of walking is removed and unnecessary inheritance is removed. This makes the code logically accurate and less coupling as well.  **Example 2:**  **Violated code:**  **public** **class** Vehicle{  *//functions of vehicles*  **public** **void** reFuel();  }  **public** **class** Bicycle **extends** Vehicle()  {  *//inherits all functions from parent class*  }  **public** **class** Car **extends** Vehicle()  {  *//inherits all functions from parent class*  }  **Violation:**  In this case parent class has the function that cannot be implemented In all subclasses. For example bicycles don’t need to be refueled. So, this code is incorrect logically and reused unnecessarily.  **Refactored code:**  **public** **class** Vehicle{  *//functions of vehicles*  }    **public** **class** VehiclesOnFuel **extends** Vehicle{  *//inherits all functions from class*  **public** **void** reFuel();  }  **public** **class** VehiclesOnElectricity **extends** Vehicle{  *//inherits all functions from class*  **public** **void** recharge();  }  **public** **class** HumanPoweveredVehicles **extends** Vehicle{  *//inherits all functions from class*  }  **public** **class** car **extends** VehiclesOnFuel{  *//inherits all functions from class*  }  **public** **class** bicycle **extends** HumanPoweredVehicles{  *//inherits all functions from class*  }  **Refactoring:**  In this code, using the LSP the extra function of walking is removed and unnecessary inheritance is removed. This makes the code logically accurate and less coupling as well. Now the classes can easily replace their parent classes. |
| **Open Close Princple**  **Violated code:**  public class PriceInfo  {  public double fruitPrice(Fruit fruit)  {  if (fruit instanceof Mango)  {  return fruit.getPrice ();  if (fruit instanceof Apple)  {  return fruit.getPrice();  }  }  **Violation:**  The class PriceInfo returns prices of fruits depending on the instance of certain fruit been passed. If we want to add another subclass named Banana, simply, we add one more if statement that violates the open-closed principle. As the class is easily modifiable which should not be the case instead it should be extensible.  **Refactored code:**  public class PriceInfo  {  public double fruitPrice()  {  //functionality  }  }  public class Mango extends PriceInfo  {  public double fruitPrice()  {  return this.getPrice();  }  public class Banana extends FruitInfo  {  public double fruitPrice()  {  return this.getPrice();  }  **Refactoring:**  The only way to add the subclass and achieve the goal of Open Close principle is by overriding the fruitPrice() method. Similarly, we can add more fruits by making another subclass extending from the Fruit class and this approach would not affect the existing application.  **Example 2:**  **Violated code:**  public interface CalculatorOperation {}  public class Addition implements CalculatorOperation {  private double left;  private double right;  private double result = 0.0;  public Addition(double left, double right) {  this.left = left;  this.right = right;  }  // getters and setters  }  public class Subtraction implements CalculatorOperation {  private double left;  private double right;  private double result = 0.0;  public Subtraction(double left, double right) {  this.left = left;  this.right = right;  }  // getters and setters  }  public class Calculator {  public void calculate(CalculatorOperation operation) {  if (operation == null) {  throw new InvalidParameterException("Can not perform operation");  }  if (operation instanceof Addition) {  Addition addition = (Addition) operation;  addition.setResult(addition.getLeft() + addition.getRight());  } else if (operation instanceof Subtraction) {  Subtraction subtraction = (Subtraction) operation;  subtraction.setResult(subtraction.getLeft() - subtraction.getRight());  }  }  }  **Violation:**  Although this may seem fine, it's not a good example of the OCP. When a new requirement of adding multiplication or divide functionality comes in, we've no way besides changing the calculate method of the Calculator class. Hence, we can say this code is not OCP compliant. As the class is easily modifiable which should not be the case instead it should be extensible.  **Refactored code:**  public interface CalculatorOperation {  void perform();  }  public class Addition implements CalculatorOperation {  private double left;  private double right;  private double result;  // constructor, getters and setters  @Override  public void perform() {  result = left + right;  }  }  public class Division implements CalculatorOperation {  private double left;  private double right;  private double result;  // constructor, getters and setters  @Override  public void perform() {  if (right != 0) {  result = left / right;  }  }  }  public class Calculator {  public void calculate(CalculatorOperation operation) {  if (operation == null) {  throw new InvalidParameterException("Cannot perform operation");  }  operation.perform();  }  }  **Refactoring:**  As we've seen our calculator app is not yet OCP compliant. The code in the calculate method will change with every incoming new operation support request. So, we need to extract this code and put it in an abstraction layer. Making an abstract function in the interface and then providing its implementation in each derived class resolves the problem of modifiability and introduces extensibility. |

**Deliverables**

Compile a single word document by filling in the solution part and submit this Word file on LMS. This lab grading policy is as follows: The lab is graded between 0 to 10 marks. The submitted solution can get a maximum of 5 marks. At the end of each lab or in the next lab, there will be a viva related to the tasks. The viva has a weightage of 5 marks. Insert the solution/answer in this document. You must show the implementation of the tasks in the designing tool, along with your completed Word document to get your work graded. You must also submit this Word document on the LMS. In case of any problems with submissions on LMS, submit your Lab assignments by emailing it to [nadeem.nawaz@seecs.edu.pk](mailto:nadeem.nawaz@seecs.edu.pk).