**Topic:** SOLID Principles

**S. Single Responsibility Principle (SRP):**

This principle states that a class should have only one reason to change. In other words, a class should have only one responsibility or job to do. If a class has multiple responsibilities, changes to one responsibility may unintentionally affect other responsibilities.

This class's sole function is to process orders. It has three private methods, each of which handles a different aspect of the order processing logic. The class is easier to maintain and alter because the many responsibilities are separated into their own methods.

|  |
| --- |
| public class OrderProcessor {      public void processOrder(Order order) {          validateOrder(order);          saveOrder(order);          sendConfirmationEmail(order);      }        private void validateOrder(Order order) {          // validation logic      }        private void saveOrder(Order order) {          // save to database logic      }        private void sendConfirmationEmail(Order order) {          // email sending logic      }  } |

**O. Open/Closed Principle (OCP):**

This principle states that a class should be open for extension but closed for modification. This means that new functionality should be added to a class by creating new classes that inherit from the original class, rather than modifying the original class itself.

In this case, we have a Shape interface that offers a method for determining the area of a shape. Then we have two classes, Circle and Rectangle, that implement the Shape interface and implement the area() method. We might add a new shape, like as a triangle, by creating a new class that implements the Shape interface rather than modifying the current Circle or Rectangle classes.

|  |
| --- |
| public interface Shape {      double area();  }  public class Circle implements Shape {      private double radius;      public Circle(double radius) {          this.radius = radius;      }      public double area() {          return Math.PI \* radius \* radius;      }  }  public class Rectangle implements Shape {      private double width;      private double height;      public Rectangle(double width, double height) {          this.width = width;          this.height = height;      }      public double area() {          return width \* height;      }  } |

**L. Liskov Substitution Principle (LSP):**

This principle states that any instance of a parent class should be able to be replaced by an instance of a child class without affecting the correctness of the program. In other words, child classes should be able to be used in place of their parent class without causing unexpected behavior.

In this case, We have a Bird class with a fly() function in this example. Because penguins cannot fly, we have a Penguin class that extends Bird but overrides the fly() method to throw an error. The watchBird() method of the BirdWatcher class takes a Bird object and calls its fly() method. This method can be called with either a Bird or a Penguin object, and the behaviour is the same for both.

|  |
| --- |
| public class Bird {      public void fly() {          System.out.println("I'm flying!");      }  }  public class Penguin extends Bird {      public void fly() {          throw new UnsupportedOperationException("Penguins can't fly!");      }  }  public class BirdWatcher {      public void watchBird(Bird bird) {          bird.fly();      }  } |

**I. Interface Segregation Principle (ISP)**

The Interface Segregation Principle states that clients should not be forced to depend on interfaces they do not use. No code should be forced to depend on methods it does not use. This means that interfaces should be designed in such a way that they are specific to the needs of their clients.

In this case, each class only implements the methods it needs to, and the interface is designed specifically for the needs of its clients.

|  |
| --- |
| interface EditableDocument {      void edit();  }  interface PrintableDocument {      void print();  }  interface ShareableDocument {      void share();  }  class WordDocument implements EditableDocument, PrintableDocument, ShareableDocument {      void edit() {          // ...      }      void print() {          // ...      }      void share() {          // ...      }  }  class PdfDocument implements PrintableDocument, ShareableDocument {      void print() {          // ...      }      void share() {          // ...      }  } |

**D. Dependency Inversion Principle (DIP)**

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules. Instead, both should depend on abstractions. Abstractions should not depend on details, but details should depend on abstractions.

Consider a scenario where we have a user class that needs to send an email when a new account is created. This code allows us to easily switch to a different email sending implementation in the future, without having to modify the User class.

|  |
| --- |
| interface EmailService {      void sendEmail(String message);  }  class User {      private EmailService emailService;      User(EmailService emailService) {          this.emailService = emailService;      }      void createAccount() {  // Create account logic          emailService.sendEmail("New account created.");      }  }  class EmailSender implements EmailService {      void sendEmail(String message) {  // ...      } |