Theoretical:

1. Briefly explain Liscov’s Substitution principle with a suitable example.
2. Briefly explain the Interface Segregation principle with a suitable example.
3. Differentiate between adding a function in a code and refactoring a function.
4. Briefly explain the following code smells and common strategies to remove those code smells-

   (i) Feature envy

   (ii) Refused bequest

   (iii) Switch-if

   (iv) Speculative generality

   (v) Oddball solution

5. What is a design pattern? Explain the categories of design patterns (with definitions and examples).

Analytical:

1. In an e-commerce platform, the **Order** class is responsible for handling the order details, calculating the order total, processing payments, and generating invoices. Whenever a new payment method (like PayPal or cryptocurrency) is introduced, the **Order** class must be modified to accommodate the new payment logic.

   Identify the SOLID principles that were violated here. Justify your answers.

2. In a logging system of a large web application, a **Logger** class is used throughout the application to record logs in a file. The system ensures that all log messages are written in a single log file to avoid issues like multiple instances creating conflicting log files.

   Which design pattern would you apply in this scenario and why?

3. In a car manufacturing application, a **Car** object is composed of several parts, including the engine, wheels, seats, and doors. Depending on the model, different configurations of these parts are assembled (e.g., a sports car vs. a sedan). The construction process involves multiple steps, and the parts may vary depending on the customer’s choice.

Which design pattern would you apply in this scenario and why?

Code-based:

1. Identify which SOLID principle(s) are violated in the following code and write the corrected pseudo code.

```
class Bird:
    def fly(self):
        print("Flying")

class Ostrich(Bird):
    def fly(self):
        raise Exception("Ostriches can't fly!")

def make_bird_fly(bird: Bird):
    bird.fly()

bird = Bird()
make_bird_fly(bird)

ostrich = Ostrich()
make_bird_fly(ostrich)
```

2. Identify 2 code smells from the following code snippet. Justify your selections.

```
class Order {
    private double price;
    private int quantity;
    private String customerName;

     public Order(double price, int quantity, String
customerName) {
        this.price = price;
        this.quantity = quantity;
        this.customerName = customerName;
    }

    public double calcTotalAmount() {
        return price * quantity;
    }
```

```
      public void printOrderDetails() {
          double sum = calcTotalAmount();
       System.out.println("Order for"+customerName+": "+sum);
     }

     public void processPayment() {
         double total = calcTotalAmount();
         System.out.println("Processing payment of: " + total);
     }

     public void saveToDatabase() {
         DatabaseManager.saveOrderToDatabase(this);
         Logger.logOrderSave(this);
         NotificationService.sendOrderConfirmation(this);
     }
 }


 class DatabaseManager {
     public static void saveOrderToDatabase(Order order) {
         System.out.println("Saving order to the database...");
     }
 }
```

3. You are building a notification system for a mobile application that supports different types of notifications, such as SMS, Email, and Push notifications. Each notification type requires different configurations and sending methods. The system needs to create the appropriate notification object based on the user's chosen notification type.

   Which creational design pattern will be the most suitable for this scenario? Explain your answer and write a pseudo-code for your design.