East West University

Semester: Fall-2025

Course Title: Software Architecture

Course Code: CSE423 **Sec: 01**

Assignment 1

Submitted by-

Sheikh Sarafat Hossain

2022-3-60-109

Submitted to-

Ahmed Adnan

Lecturer

Department of Computer Science & Engineering

Date of Submission: 7th November 2025

Theoretical:

1. **Briefly explain Liscov's Substitution principle with a suitable example.**
   Answer: Liskov's Substitution Principle states that a child class must be able to replace the parent class without changing the correctness of the program.
   Example:
   Suppose I have a Vehicle class which has methods like model_number(), engine_number(), and number_of_wings(). If I extend the Vehicle class with a Car class, then the Car class will have all these methods. But a car doesn't have wings. In this case, the Car class cannot fully replace the Vehicle class. So, it violates the Liskov's Substitution Principle.
   Solution:
   We can separate the number_of_wings method from the Vehicle class and create a FlyingVehicle class. Then we extend the Vehicle class in FlyingVehicle and add number_of_wings there.

2. **Briefly explain the Interface Segregation principle with a suitable example.**
   Answer: The interface segregation principle states that a client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.
   Example:
   Suppose I have an interface Worker which has methods like work(), eat(), and sleep(). Now if I implement this interface in a Robot class, then the robot will also have to implement all these methods. But a robot doesn't eat or sleep. So, it doesn't make sense and it violates the Interface Segregation Principle.
   Solution:
   We can separate the Worker interface into smaller interfaces like Workable, Eatable, and Sleepable. Now the Robot class will only implement Workable, and a Human class can implement all three if needed. This way, every class only implements what it actually needs.

3. **Differentiate between adding a function in a code and refactoring a function.**
   Answer: Adding a function in a code means adding new functionality to your program. It may produce a new output or change the existing output. It will add new test cases.
   Refactoring a function means the output must remain exactly the same, but the internal code is reorganized or cleaned up to improve readability, maintenance, and reusability. It will not add new test cases.

4. Briefly explain the following code smells and common strategies to remove those code smells-

(i) Feature envy

Answer: This happens when a method in one class is more interested in the data of another class than its own. It keeps calling methods or using variables of another class.

Solution:

Move that method to the class it is "envying" or where most of the data it uses actually belongs.

(ii) Refused bequest

Answer: This happens when a child class inherits methods or properties from a parent class but does not use them or overrides them unnecessarily. It's like the subclass "refuses" the inheritance.

Solution:

- Use composition instead of inheritance.
- Or move the unused methods to a more suitable class.

(iii) Switch-if

Answer: When a program uses a long chain of if-else or switch statements to decide behavior based on types or values.

Solution:

- Use **polymorphism** (create subclasses for each case).
- Or use **Strategy Pattern / Factory Pattern** to remove long condition checks.

(iv) Speculative generality

Answer: This occurs when code is written to support future scenarios that are not currently needed. Example: unused abstract classes, interfaces, parameters, or methods added "just in case".

Solution:

Remove the unnecessary abstractions and keep the code simple and only for what is required right now.

(v) Oddball solution

Answer: When the same problem is solved in different ways in different parts of the code. One part uses a different method instead of following the common approach used elsewhere.

Solution:

Choose the best or most used solution and make all other similar codes follow the same approach for consistency.

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

Answer: A design pattern is a general, reusable solution to a common problem that occurs in software design. It is not code, but a template or guideline on how to solve a problem in an organized and efficient way.

Design patterns are mainly divided into three categories:

1. Creational Design Patterns

Definition:

These patterns focus on how objects are created. They help make the creation process more flexible and reusable.

Examples:

Singleton: Ensures only one object of a class is created.

Example: A class for database connection that must have only one instance.

2. Structural Design Patterns

Definition:

These patterns explain how to combine or structure classes and objects to form a larger system while keeping it flexible.

Examples:

Adapter: Converts one interface into another interface the client expects.

Example: A charger adapter that lets a laptop charger fit into a different socket type.

3. Behavioral Design Patterns

Definition:

These patterns focus on how objects communicate and interact with each other.

Examples:

Observer: When one object changes, other dependent objects are automatically notified.

Example: In YouTube, when a new video is uploaded, subscribers get notifications.

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.

Answer: In this case, it is violating two SOLID principles.

1. Single Responsibility Principle (SRP)

The Order class is handling too many tasks — order details, total calculation, payment processing, and invoice generation. A class should have only one responsibility.

2. Open/Closed Principle (OCP)

Whenever a new payment method (like PayPal or crypto) is added, the Order class has to be modified. This breaks OCP because the class should be open for extension but closed for modification.

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?

Answer: In here we can apply Singleton design pattern.

The Singleton pattern ensures that only **one instance** of the Logger class exists throughout the application. This guarantees that all log messages are written to a **single log file**, avoiding conflicts caused by multiple instances trying to write logs simultaneously.

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?

Answer: The Builder design pattern should be applied in this scenario.

Reason:
The Builder pattern is used to construct a complex object step by step, allowing different representations of the object. In this case, a Car is composed of multiple parts (engine, wheels, seats, doors), and different car models (sports car, sedan) require different configurations. The Builder pattern helps assemble these parts in a controlled and flexible way according to customer choices.

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)
```

Answer: In this case, it is violating the **Liskov Substitution Principle (LSP)**. The Ostrich class, which is a subclass of Bird, has a fly() method in the parent class that it cannot use, so it raises an exception. This means the subclass cannot fully replace its parent class, which violates LSP.

Corrected Pseudo Code:

```
class Bird { }


class FlyingBird extends Bird {
   void fly() { }
}


class Ostrich extends Bird { }
```

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...");
}
}
```

Answer: Two code smells in this code are: **Large Class & Feature Envy**

1. Large Class

The Order class is doing too many things:

- Calculating order total
- Printing details
- Processing payments
- Saving to database, logging, sending notifications

One class should not handle all these responsibilities. It makes the code harder to maintain and modify.

2. Feature Envy

The Order class is directly calling other services like: DatabaseManager.saveOrderToDatabase(), Logger.logOrderSave(), NotificationService.sendOrderConfirmation().

This shows Order is too involved in other classes' work instead of only managing order-related tasks.

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.

Answer: The **Factory Method design pattern** is the most suitable for this scenario.

Reason:

- The system needs to create different types of notifications (SMS, Email, Push) based on user choice.
- Each notification type has different configurations and sending methods.
- The Factory pattern allows creating objects **without exposing the instantiation logic** to the client and makes it **easy to add new notification types** in the future without modifying existing code.

Pseudo-code using Factory Pattern:

```
interface Notification {
  void send(String message);
}

class SMSNotification implements Notification {
  void send() { }
}
class EmailNotification implements Notification {
  void send() { }
}
class PushNotification implements Notification {
  void send() { }
}
class NotificationFactory {
  static Notification create(String type) {
    if (type == "SMS") return new SMSNotification();
```

```
        if (type == "Email") return new EmailNotification();
        if (type == "Push") return new PushNotification();
        return null;
    }
}
Notification n = NotificationFactory.create("Email");
n.send("Hello User!");
```