O.O.P.

Steve Jobs: *Objects are like people. They're living, breathing things that know how to do things and have memories inside them so they can remember things. And rather than interacting with them at a lower level, you interact with them at a very high level of abstraction, as we're doing here.*

Here's an example: If I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes."

You have no idea how I did that. You do not know the laundry place. Maybe you speak French, and you can't even hail a taxi. You can't pay for one, you don't have dollars in your pocket. Yet I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. They encapsulate complexity, and the interfaces to that complexity are high-level.

Assessment #1

Code @ work

Here is a blog on S.O.L.I.D!

Assessment #2

Code @ work

Design Patterns

Design patterns are proven, reusable solutions to common software design problems. They provide a structured approach to problems and improve code maintainability, scalability, and reusability. The key categories of design patterns are **Creational**, **Structural**, and **Behavioral** patterns. Here's an overview of each category and some common design patterns within them:

1. Creational Design Patterns
These patterns deal with object creation mechanisms, aiming to create objects in a way that is suitable to the situation. They help manage the complexity of the instantiation process.

a. Singleton

- Purpose: Ensures that a class has only one instance and provides a global access point to that instance.
- Use Case: Typically used for classes that manage a shared resource like a configuration manager, database connection, or logging service.

Example:

```java
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

b. Factory Method

- Purpose: Defines an interface for creating an object, but lets subclasses alter the type of objects that will be made.
- Use Case: When a class cannot anticipate the type of objects it must create, or when object creation logic should be delegated to subclasses.

Example:

```java
abstract class ProductFactory {
    abstract Product createProduct();
}

class ConcreteProductFactory extends ProductFactory {
    @Override
    Product createProduct() {
        return new ConcreteProduct();
    }
}
```

c. Builder
Purpose: It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
- Use Case: When an object needs to be created step-by-step, especially if it has many optional parameters.

Example:

```java
public class Car {
    private String engine;
    private int seats;

    public static class Builder {
        private String engine;
        private int seats;

        public Builder setEngine(String engine) {
            this.engine = engine;
            return this;
        }

        public Builder setSeats(int seats) {
```

```
            this.seats = seats;
            return this;
        }

        public Car build() {
            Car car = new Car();
            car.engine = this.engine;
            car.seats = this.seats;
            return car;
        }
    }
}
```

2. Structural Design Patterns

a. Adapter

Purpose: Allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces.

Use Case: When you want to use an existing class, but its interface doesn't match the one you need.

```
Java
interface MediaPlayer {
    void play(String audioType, String fileName);
}

class MediaAdapter implements MediaPlayer {
    AdvancedMediaPlayer advancedPlayer;

    public MediaAdapter(String audioType) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedPlayer = new VlcPlayer();
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedPlayer = new Mp4Player();
        }
    }
```

```
    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedPlayer.playVlc(fileName);
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedPlayer.playMp4(fileName);
        }
    }
}
```

c. Decorator

- Purpose: Attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality.
- Use Case: You must add behaviour to individual objects without affecting others.

```
Java
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Shape: Circle");
    }
}

class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    public void draw() {
        decoratedShape.draw();
    }
}

class RedShapeDecorator extends ShapeDecorator {
    public RedShapeDecorator(Shape decoratedShape) {
```

```
        super(decoratedShape);
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape){
        System.out.println("Border Color: Red");
    }
}
```

3. Behavioral Design Patterns

These patterns deal with communication between objects, ensuring that objects interact flexibly and flexibly.

a. Observer

- Purpose: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Use Case: Useful in event-driven systems or when changes in one object need to trigger changes in others.

```
Java
interface Observer {
    void update();
}

class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
```

```
    }
}
```

Here's a guide on how to write good code - [Coding Standards]

So, at the end of Day 1, you should be able to:

1. Implement OOP while solving problems
2. Implement SOLID principles whenever you are writing code
3. Implement different design patterns in test automation based on the requirements