# Lab – Design Patterns using Java

## Objective

This exercise introduces several real-life design pattern examples. Study as many as time allows, and consider how you might apply them in your workplace.

# Singleton Pattern

## Purpose:

Ensure that a class has only one instance and provides a global point of access to it.

**Pros:**

- Controlled access to a single instance
- Saves memory for shared resources

**Cons:**

- Can hide dependencies (global state)
- Difficult to test or scale in multi-threaded environments

## Code example:

A printer spooler should only have one active instance managing the print jobs.

```
public class Singleton {
	public static void main(String[] args) {

	}
}
```

```
class PrinterSpooler {
    private static PrinterSpooler instance;

    private PrinterSpooler() {}

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

    public void print(String document) {
        System.out.println("Printing: " + document);
    }
}
```

# Factory Pattern

## Purpose:

Use a Factory to create objects without exposing the creation logic to the client, and refer to the newly created object using a common interface.

**Pros:**

- Adds flexibility in object creation
- Centralizes creation logic

**Cons:**

- Can introduce extra complexity
- Less transparent: harder to know which class is being instantiated

## Code example:

A car factory that can produce different types of cars (e.g., Sedan, SUV).

```
public class Factory {

	public static void main(String[] args) {
	}
}


interface Car {
    void drive();
}

class Sedan implements Car {
    public void drive() {
        System.out.println("Driving a Sedan");
    }
}

class SUV implements Car {
    public void drive() {
        System.out.println("Driving an SUV");
    }
}

class CarFactory {
    public static Car getCar(String type) {
        switch (type.toLowerCase()) {
            case "sedan": return new Sedan();
            case "suv": return new SUV();
            default: throw new IllegalArgumentException("Unknown car type");
        }
    }
}
```

# Composite Pattern

## Purpose:

Use the Composite pattern when you want to treat individual objects and groups of objects in the same way. It's commonly used for tree-like structures.

**Pros:**

- Simplifies client code
- Makes it easy to add new types of components

**Cons:**

- Can make the design overly general
- Might be harder to restrict component types

## Code example:

A company structure where departments can have employees or sub-departments.

```
public class Composite {

	public static void main(String[] args) {
	}
}

interface CompanyComponent {
    void showDetails();
}

class Employee implements CompanyComponent {
    private String name;
    public Employee(String name) {
        this.name = name;
    }
    public void showDetails() {
        System.out.println("Employee: " + name);
    }
}

class Department implements CompanyComponent {
    private String name;
    private List<CompanyComponent> components = new ArrayList<>();

    public Department(String name) {
        this.name = name;
    }

    public void addComponent(CompanyComponent component) {
        components.add(component);
    }

    public void showDetails() {
        System.out.println("Department: " + name);
        for (CompanyComponent component : components) {
            component.showDetails();
        }
    }
}
```

# Builder Pattern

## Purpose:

Separate the construction of a complex object from its representation, allowing different configurations of the object to be built step-by-step.

**Pros:**

- Good for creating complex objects step-by-step
- Clear and readable code for configurations

**Cons:**

- Requires more classes/code
- Not ideal for simple objects

## Code example:

Building a Laptop with optional features like graphics card, different RAM, or processors.

```
public class Builder {
	public static void main(String[] args) {
	}
}

class Laptop {
    private String processor;
    private int ram;
    private boolean hasGraphicsCard;

    private Laptop(Builder builder) {
        this.processor = builder.processor;
        this.ram = builder.ram;
        this.hasGraphicsCard = builder.hasGraphicsCard;
    }

    public static class Builder {
        private String processor;
        private int ram;
        private boolean hasGraphicsCard;

        public Builder setProcessor(String processor) {
            this.processor = processor;
            return this;
        }

        public Builder setRam(int ram) {
            this.ram = ram;
            return this;
        }

        public Builder setGraphicsCard(boolean hasGraphicsCard) {
            this.hasGraphicsCard = hasGraphicsCard;
            return this;
        }
```

```
        public Laptop build() {
            return new Laptop(this);
        }
    }

    public void showSpecs() {
        System.out.println("Laptop with " + processor + ", " + ram + "GB RAM,
                                Graphics: " + hasGraphicsCard);
    }
}
```

# Observer Pattern

## Purpose:

Define a one-to-many dependency so that when one object (subject) changes state, all its dependents (observers) are notified automatically.

**Pros:**

- Promotes loose coupling
- Good for event-driven systems

**Cons:**

- Can lead to unexpected updates if not carefully managed
- Debugging can be tricky when many observers are involved

## Code example:

A weather station that notifies multiple mobile apps when the temperature changes.

```
public class ObserverDemo {
	public static void main(String[] args) {
	}
}

interface Observer {
    void update(float temperature);
}

interface Subject {
    void register(Observer o);
    void remove(Observer o);
    void notifyObservers();
}

class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private float temperature;

    public void register(Observer o) {
        observers.add(o);
    }

    public void remove(Observer o) {
        observers.remove(o);
    }

    public void setTemperature(float temp) {
        this.temperature = temp;
        notifyObservers();
    }

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

```
class MobileApp implements Observer {
    private String name;

    public MobileApp(String name) {
        this.name = name;
    }

    public void update(float temperature) {
        System.out.println(name + " received temperature update: " + temperature +
                                        "°C");
    }
}
```

# Command Pattern

## Purpose:

Encapsulate a request as an object, thereby allowing for parameterisation and queuing of requests.

**Pros:**

- Decouples sender and receiver
- Supports undo/redo and logging

**Cons:**

- Can add a lot of boilerplate code
- Slightly harder to trace execution flow

## Code example:

A remote control that can execute commands like turning the light on or off.

```
public class CommandDemo {
	public static void main(String[] args) {
	}
}

interface Command {
    void execute();
}

class Light {
    public void turnOn() {
        System.out.println("Light ON");
    }

    public void turnOff() {
        System.out.println("Light OFF");
    }
}

class TurnOnCommand implements Command {
    private Light light;
    public TurnOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}
```

```
class TurnOffCommand implements Command {
    private Light light;
    public TurnOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOff();
    }
}

class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

## If time permits

Rresearch one of the standard design patterns.
Please investigate the pros and cons of a pattern and its UML Class diagram.
Is there a situation at work where you could use the chosen pattern?

https://refactoring.guru/design-patterns/catalog is a good place to start your research.