Lab Report

On

Design Pattern

(Singleton, Decorator and Strategy)

Course Code: CSE - 418

Course Title: Software Engineering & Design Pattern Lab

Submitted to:

Plabon Talukder

Lecturer

Department of Computer Science & Engineering

Metropolitan University, Bangladesh

Submitted by:

Mahmud

ID: 222-115-191

Batch: 57^{th} (E)

Department of Computer Science & Engineering

Metropolitan University, Bangladesh

Date of Submission: Feb 27, 2025

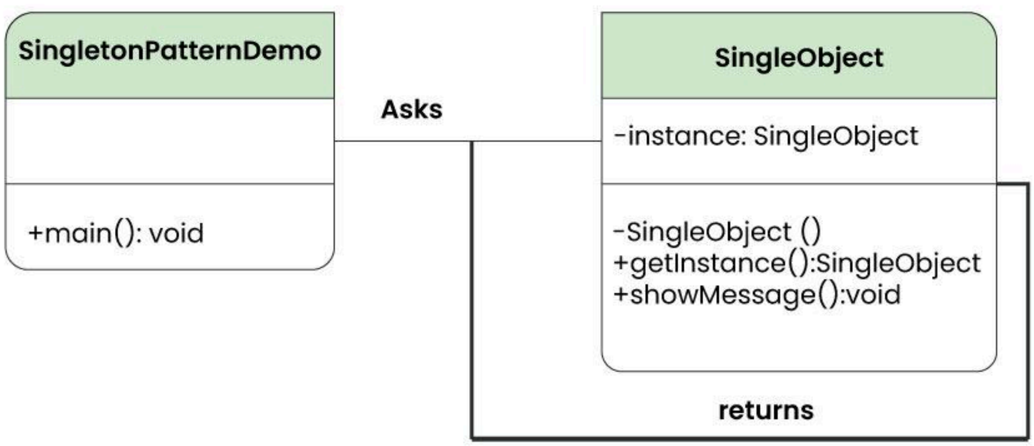
Singleton Design Pattern

The Singleton Design Pattern is a Creational pattern, whose objective is to create only one instance of a class and to provide only one global access point to that object. One commonly used example of such a class in Java is Calendar, where you cannot make an instance of that class. It also uses its own **getInstance()** method to get the object to be used.

Key Concepts of Singleton

- A private static variable, holding the only instance of the class.
- A private constructor, so it cannot be instantiated anywhere else.
- A public static method, to return the single instance of the class.

Diagram



Example

```java
public enum Singleton {
    INSTANCE;

    public void doSomething() {
        System.out.println("Doing something");
    }
}
```

1. Enum Declaration: The `Singleton` class is declared as an `enum`, which inherently ensures that only one instance of each enum constant exists.
2. INSTANCE Constant: The `INSTANCE` constant is the single instance of the `Singleton` enum.
3. Methods: You can add methods like `doSomething()` to perform actions.

```java
public static void main(String[] args) {
    Singleton.INSTANCE.doSomething(); // Output: Doing something
}
```

To access the instance, simply use Singleton.INSTANCE.

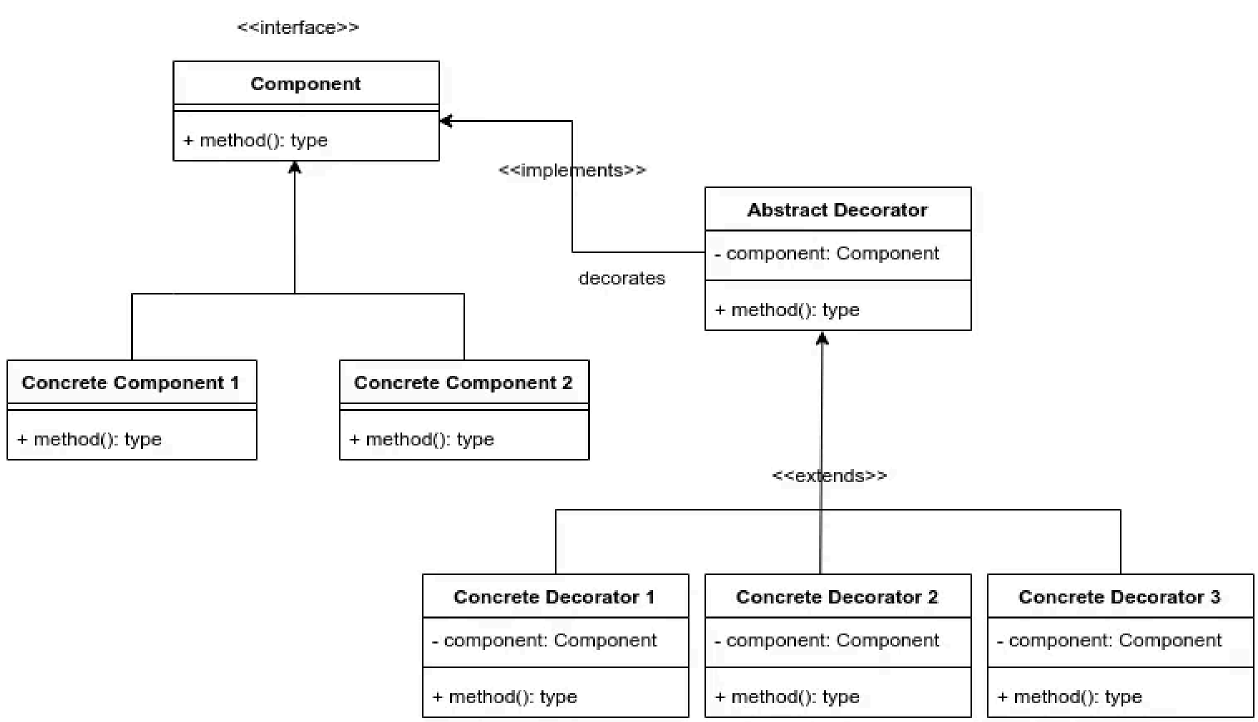
Decorator Design Pattern

The Decorator design pattern is a structural pattern used in object-oriented programming to add new functionality to objects dynamically without altering their structure. In Java, this pattern is often employed to extend the behavior of objects in a flexible and reusable way.

Components of Decorator Method Design Pattern

- **Component Interface**: An interface or abstract class that defines the core functionality. This is the base type for both concrete components and decorators.
- **Concrete Component**: A class that implements the Component interface and provides the basic behavior.
- **Decorator**: An abstract class that implements the Component interface and has a reference to a Component object. This class defines the interface for the decorators and includes a reference to a Component instance.
- **Concrete Decorators**: Classes that extend the Decorator class and add additional behavior to the Component.

Diagram



Example

```java
// Component Interface
interface Coffee {
    double cost();
    String ingredients();
}

// Concrete Component
class SimpleCoffee implements Coffee {
    @Override
    public double cost() {
        return 1;
    }

    @Override
    public String ingredients() {
        return "Normal Coffee";
    }
}
```

The `Coffee` interface defines methods for `cost()` and `ingredients()`, while `SimpleCoffee` implements these methods with a basic coffee costing $1 and containing just "Normal Coffee". This sets up the foundation for adding more complex coffee types using decorators.

```java
// Decorator
abstract class CoffeeDecorator implements Coffee {
    protected final Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }
}

// Concrete Decorators
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return coffee.cost() + 0.5;
    }

    @Override
    public String ingredients() {
        return coffee.ingredients() + " With Milk";
    }
}
```

The `CoffeeDecorator` abstract class serves as a base for adding new behaviors to coffee types. It holds a reference to a `Coffee` object and is extended by concrete decorators like `MilkDecorator`. The `MilkDecorator` adds milk to the coffee, increasing the cost by $0.5 and updating the ingredients list to include "With Milk". This pattern allows for dynamic addition of new behaviors without altering the original coffee structure.

```
public class Main {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        System.out.println("Cost: $" + coffee.cost() + ", Ingredients: " + coffee.ingredients());

        Coffee coffeeWithMilk = new MilkDecorator(coffee);
        System.out.println("Cost: $" + coffeeWithMilk.cost() + ", Ingredients: " + coffeeWithMilk.ingredients());
    }
}
```

In the Main class, a `SimpleCoffee` object is created and its cost and ingredients are printed. Then, a `MilkDecorator` is used to add milk to the coffee, creating a new `coffeeWithMilk` object. The cost and ingredients of this decorated coffee are also printed, demonstrating how the Decorator pattern dynamically adds new behaviors (in this case, adding milk) to the original coffee.

```
Output:

Cost: $1.0, Ingredients: Coffee
Cost: $1.5, Ingredients: Coffee, Milk
```

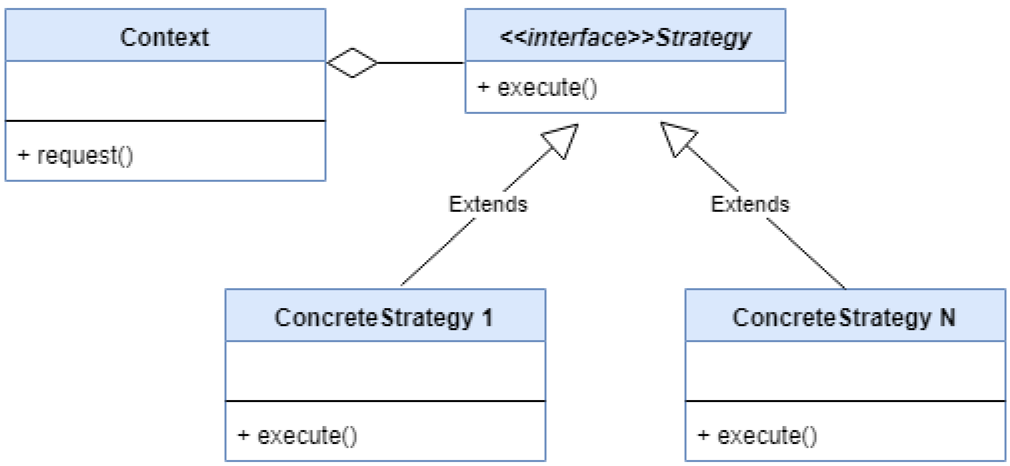
Strategy Design Pattern

The Strategy Design Pattern is a behavioral design pattern that allows you to define a family of algorithms or behaviors, put each of them in a separate class, and make them interchangeable at runtime. This pattern is useful when you want to dynamically change the behavior of a class without modifying its code.

Components of the Strategy Design Pattern

- **Context**: Holds a reference to a strategy object and delegates tasks to it, acting as the interface between the client and strategy.
- **Strategy Interface**: Defines a common interface for all concrete strategies, ensuring they are interchangeable.
- **Concrete Strategies**: Implement the Strategy Interface with specific algorithms or behaviors.
- **Client**: Selects and configures the strategy, providing it to the Context based on task requirements.

Diagram



Example

```java
// Strategy Interface
interface TravelStrategy {
    void travel(String destination);
}

// Concrete Strategies
class CarTravelStrategy implements TravelStrategy {
    @Override
    public void travel(String destination) {
        System.out.println("Traveling to " + destination + " by car");
    }
}

class FlightTravelStrategy implements TravelStrategy {
    @Override
    public void travel(String destination) {
        System.out.println("Traveling to " + destination + " by flight");
    }
}
```

This portion defines a Strategy Design Pattern for travel modes using a `TravelStrategy` interface, which declares a method `travel(String destination)`. Two concrete strategies implement this interface: `CarTravelStrategy`, which prints a message indicating travel by car to the specified destination, and `FlightTravelStrategy`, which indicates travel by flight.

```
// Context
class TravelPlanner {
    private TravelStrategy travelStrategy;

    public void setTravelStrategy(TravelStrategy travelStrategy) {
        this.travelStrategy = travelStrategy;
    }

    public void planTrip(String destination) {
        travelStrategy.travel(destination);
    }
}
```

The `TravelPlanner` class acts as the Context in the Strategy pattern. It maintains a reference to a `TravelStrategy` object and provides methods to set and use this strategy. The `planTrip(String destination)` method delegates the actual travel planning to the selected strategy, allowing for dynamic changes in travel modes without modifying the planner's logic.

```
public class Main {
    public static void main(String[] args) {
        TravelPlanner planner = new TravelPlanner();
        planner.setTravelStrategy(new CarTravelStrategy());
        planner.planTrip("Sylhet");

        planner.setTravelStrategy(new FlightTravelStrategy());
        planner.planTrip("Riyadh");
    }
}
```

In the `Main` class, a `TravelPlanner` object is created and used to plan trips. The planner's travel strategy is dynamically changed between `CarTravelStrategy` for a trip to "Sylhet" and `FlightTravelStrategy` for a trip to "Riyadh", demonstrating the flexibility of the Strategy pattern.

```
Output:

Traveling to Sylhet by car
Traveling to Riyadh by flight
```