Design Patterns

Design patterns

- **design pattern**:
  a solution to a common software problem in a context
  - recurring software structure
  - abstract from programming language
  - identifies classes and their roles in the solution to a problem
  - not code or designs; must be instantiated/applied

- example: Iterator pattern
  - The Iterator pattern defines an interface that declares methods for sequentially accessing the objects in a collection.

Gang of Four (GoF) patterns

- **Creational Patterns**
  (abstracting the object-instantiation process)
  - Factory Method
  - Abstract Factory
  - Singleton
  - Builder
  - Prototype

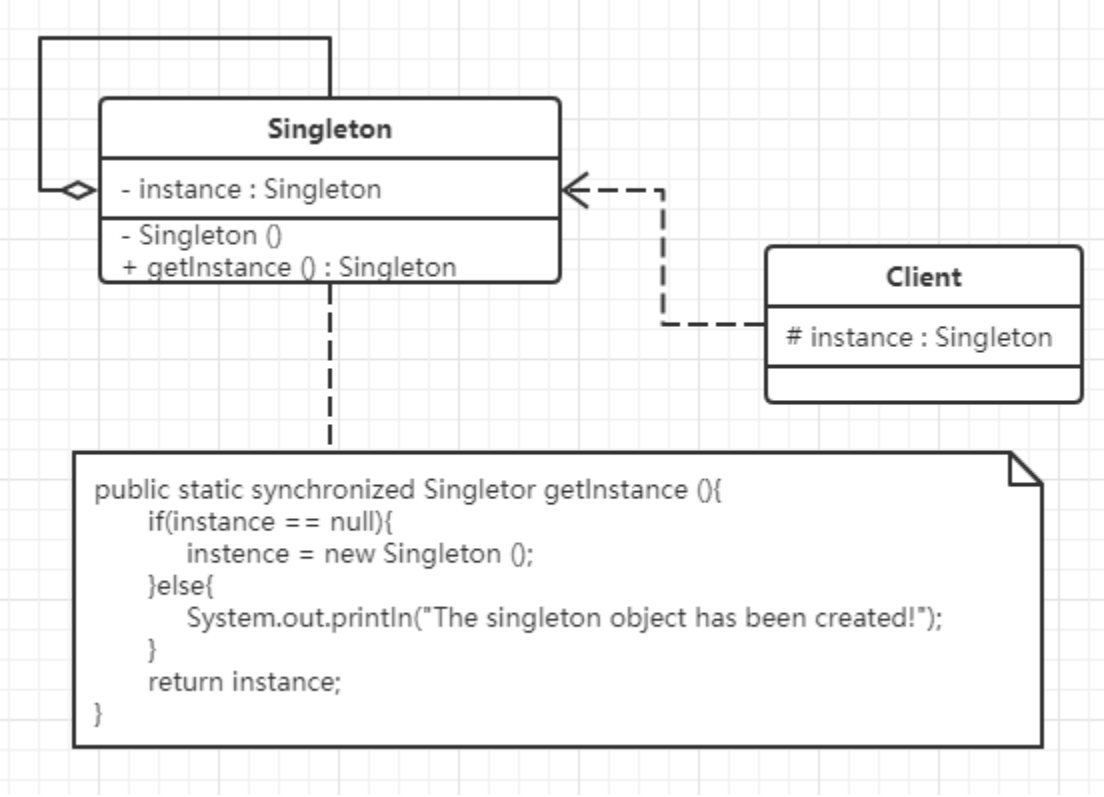
- **Structural Patterns**
  (how objects/classes can be combined to form larger structures)
  - Adapter
  - Bridge
  - *Composite*
  - *Decorator*
  - Facade
  - Flyweight
  - Proxy

- **Behavioral Patterns**
  (communication between objects)
  - Command
  - Interpreter
  - *Iterator*
  - Mediator
  - *Observer*
  - State
  - *Strategy*
  - Chain of Responsibility
  - Visitor
  - Template Method

Pattern: Singleton

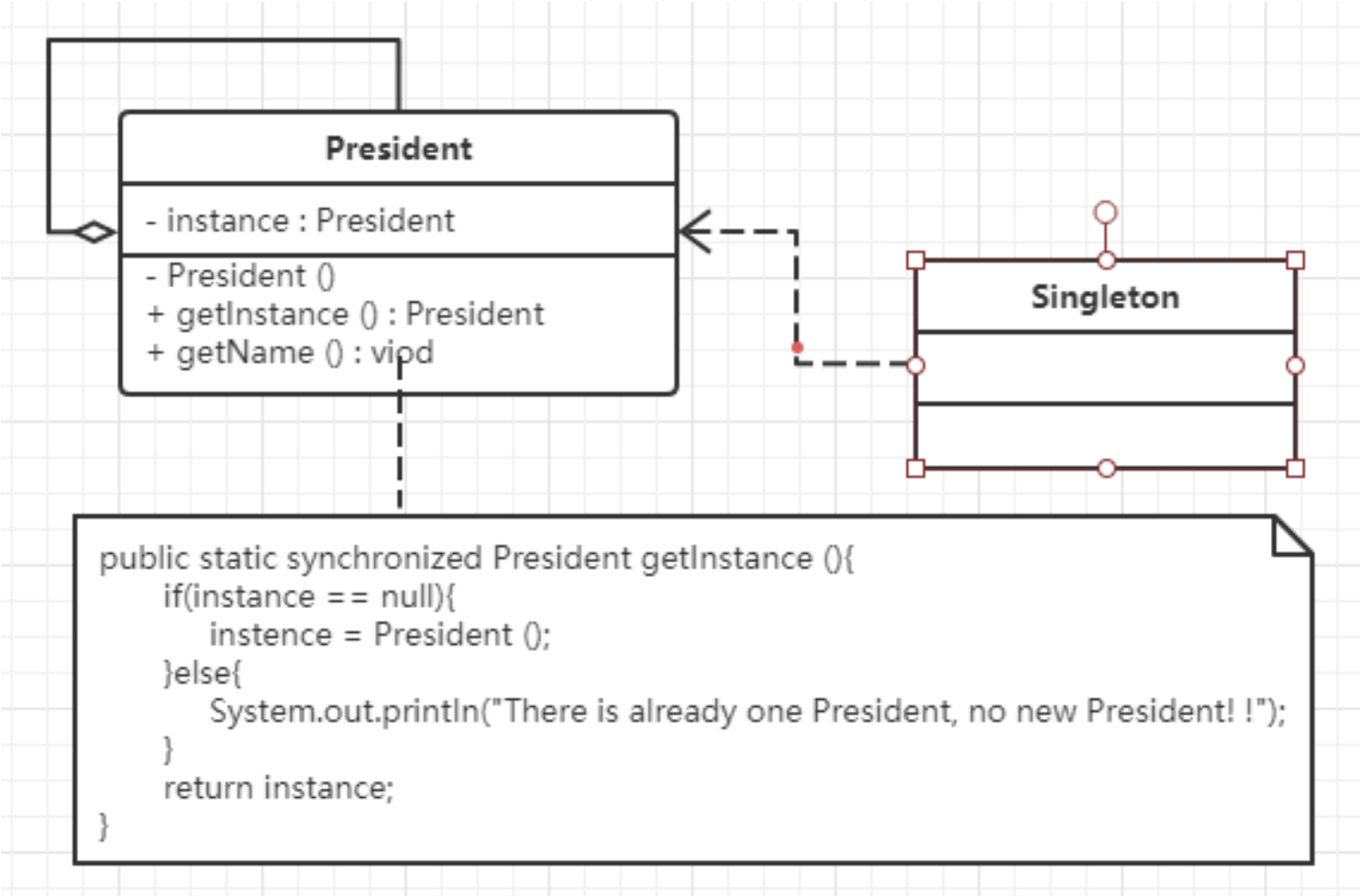
- *A pattern in which a class has only one instance and the class can create that instance itself*

- Singleton class: a class that contains an instance and can create the instance itself.
- Access class: a class that uses the singleton class




```java
public class Singleton
{
    //Ensure instance is synchronized in all threads
    private static volatile Singleton instance=null;
    //prevents classes from being instantiated externally
    private LazySingleton(){}
    public static synchronized Singleton getInstance()
    {
        //Synchronize before the getInstance method
        if(instance==null)
        {
            instance=new LazySingleton();
        }
        return instance;
    }
}
```

Singleton: example1

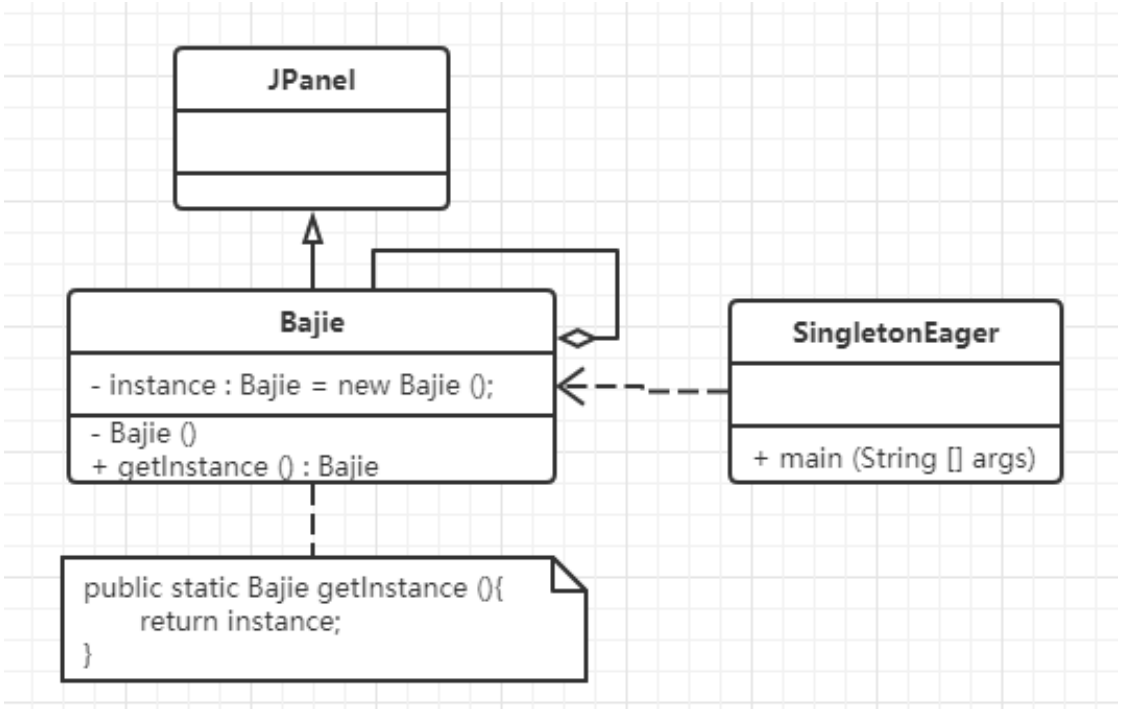


Singleton: example1

```
class President
{
    //Ensure that instance is synchronized in all threads
    private static volatile President instance=null;
    //Private prevents classes from being instantiated externally
    private President()
    {
        System.out.println("Produce a President! ");
    }
    public static synchronized President getInstance()
    {
        //Synchronize on the getInstance method
        if(instance==null)
        {
            instance=new President();
        }
        else
        {
           System.out.println("There is already one President, no new President! ");
        }
        return instance;
    }
    public void getName()
    {
        System.out.println("I'm the President of the United States: Donald trump. ");
    }
}
```

```
public class Singleton
{
    public static void main(String[] args)
    {
        President zt1=President.getInstance();
        zt1.getName();    //output the name of President
        President zt2=President.getInstance();
        zt2.getName();    //output the name of President
        if(zt1==zt2)
        {
           System.out.println("They are the same person! ");
        }
        else
        {
           System.out.println("They're not the same person! ");
        }
    }
}
```

Singleton: example2




```
class Bajie extends JPanel
{
    private static Bajie instance=new Bajie();
    private Bajie()
    {
        JLabel l1=new JLabel(new ImageIcon("Bajie.jpg"));
        this.add(l1);
    }
    public static Bajie getInstance()
    {
        return instance;
    }
}
```

```
import java.awt.*;
import javax.swing.*;
public class SingletonEager
{
    public static void main(String[] args)
    {
        JFrame jf=new JFrame("SingletonEagerTest");
        jf.setLayout(new GridLayout(1,2));
        Container contentPane=jf.getContentPane();
        Bajie obj1=Bajie.getInstance();
        contentPane.add(obj1);
        Bajie obj2=Bajie.getInstance();
        contentPane.add(obj2);
        if(obj1==obj2)
        {
            System.out.println("They are the same person! ");
        }
        else
        {
            System.out.println("They are not the same one! ");
        }
        jf.pack();
        jf.setVisible(true);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Observer Pattern

- In **Observer pattern**, the class can inform other objects which subscribe to a change of its state. Any object which is interested in this change, can subscribe to the target object, and receive this information while it occurs.
- The Pattern contains two roles:
  - **Subject**: a class that can inform others while its state has change.
  - **Observer**: a class that interested in the change of the subject.

Observer Pattern: Subject

- To implement the **Subject role,** we need to define two class:
  - **Subject** : an abstract class, which has
    - add method,
    - remove method,
    - a List, whose element type is Observer
    - a abstract method named notifyObserver.
  - **ConcreteSubject** : a class inherite from the Subject class, and implement the abstract method notifyObserver.

Observer Pattern: **Subject**

```
//abstract subject
abstract class Subject
{
    protected List<Observer> observers=new ArrayList<Observer>();
    //add
    public void add(Observer observer)
    {
        observers.add(observer);
    }
    //remove
    public void remove(Observer observer)
    {
        observers.remove(observer);
    }
    public abstract void notifyObserver(); //notify
}
```

Observer Pattern: Subject

- For the **ConcreteSubject** Class:
  - it will implement the notifyObserver method, which will notify the observer interested in it.
  - when the state has change, it will call the notifyObserver method, and call the response method of all the observers in the List to notify them.

Observer Pattern: Subject

```
//concrete subject
class ConcreteSubject extends Subject
{
    public void notifyObserver()
    {
        System.out.println("Concrete subject change");
        System.out.println("--------------");

        for(Object obs:observers)
        {
            ((Observer)obs).response();
        }

    }
}
```

Observer Pattern: Observer

- To implement the **Observer role,** we need to define two class:
  - **Observer** : an interface or abstract class, which has a notify method.
  - **ConcreteObserver** : a class inherite from the Subject class, and implement the abstract method notify.

Observer Pattern: Observer

```
//abstract observer
interface Observer
{
    void response();
}
```

```
//ConcreteObserver1
class ConcreteObserver1 implements Observer
{
    public void response()
    {
        System.out.println("ConcreteObserver1 responds! ");
    }
}
//ConcreteObserver2
class ConcreteObserver2 implements Observer
{
    public void response()
    {
        System.out.println("ConcreteObserver2 responds! ");
    }
}
```

Observer Pattern: Demo

- Consider the rate of a currency, serveral international trading companies interested in the its rate of their primary currency, and wants to notice the change of them. We can use **Observer Pattern** to implement it.

Observer Pattern: Demo

In this problem, rate represents Subject, and companies represents Observer. We define a abstract class named **Rate**, which represent the rate of the currency

```
abstract class Rate
{
    protected List<Company> companys=new ArrayList<Company>();
    //add
    public void add(Company company)
    {
        companys.add(company);
    }
    //remove
    public void remove(Company company)
    {
        companys.remove(company);
    }
    public abstract void change(int number);
}
```

Observer Pattern: Demo

- We Implement the **RMBRate,** so The Chinese company can use add method to subscribe to the **RMBRate**, and it will call the change method to notify them while changes occur.

```
//consrete: RMBrate
class RMBrate extends Rate
{
    public void change(int number)
    {
        for(Company obs:companys)
        {
            ((Company)obs).response(number);
        }
    }

}
```

Observer Pattern: Demo

Then, we define the abstract class : company.

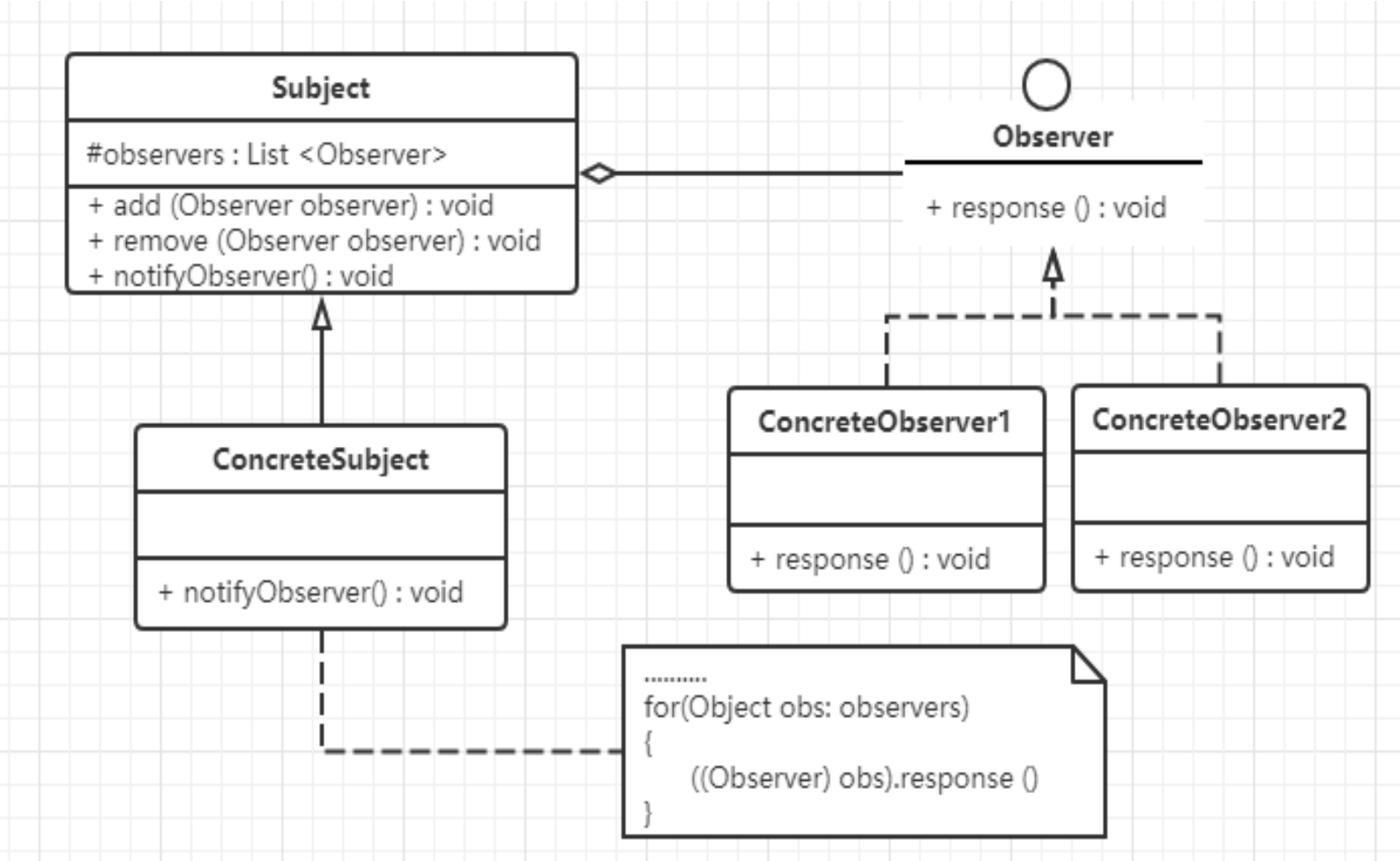
```
//abstrct observer: Company
interface Company
{
    void response(int number);
}
```

Observer Pattern: Demo

After receiving the change, the companies can deal with their own business.

```
//concreteObserver1: ImportCompany
class ImportCompany implements Company
{
    public void response(int number){
        if(number>0){
            System.out.println("RMBrate increase, improved the profit margin of import companies");
        }
        else if(number<0){
            System.out.println("RMBrate decrease, reduced the profit margin of import companies");
        }
    }
}
//concreteObserver2: ExportCompany
class ExportCompany implements Company
{
    public void response(int number){
        if(number>0){
            System.out.println("RMBrate increase, reduced the profit margin of export companies");
        }
        else if(number<0){
            System.out.println("RMBrate increase, improved the profit margin of export companies");
        }
    }
}
```

Observer Pattern : Class Diagram

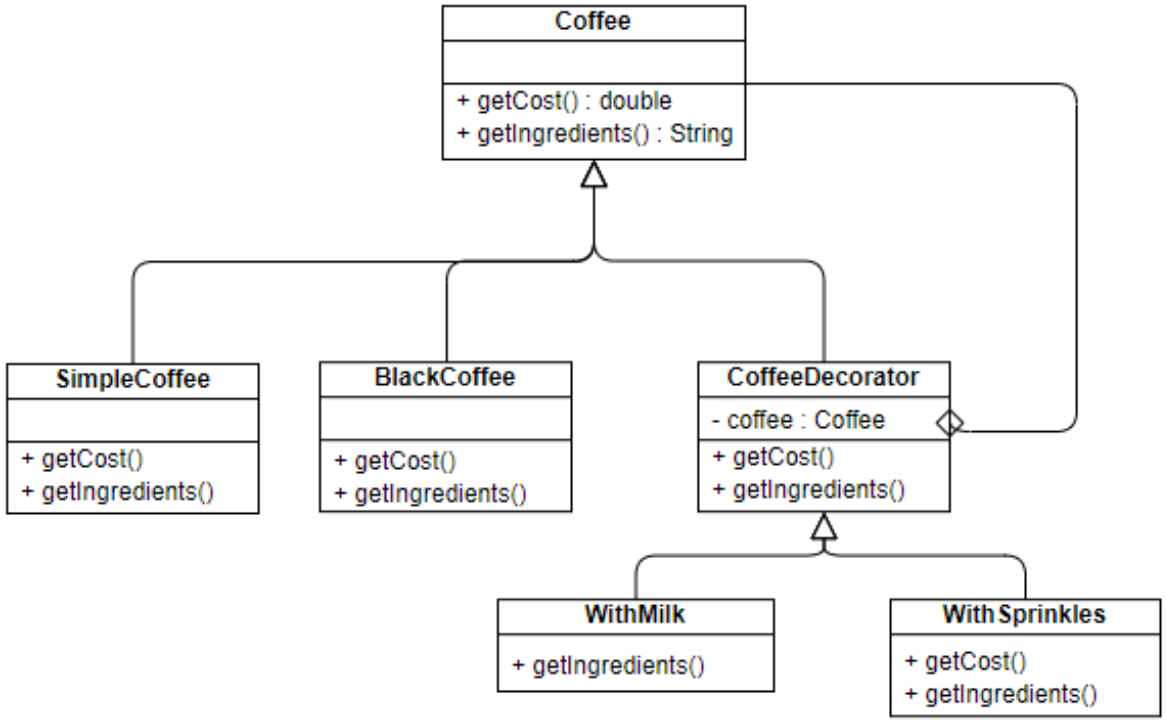


Decorator Pattern

- In object-oriented programming, the **Decorator Pattern** is a design pattern that allows behavior to be added to an individual object, without affecting the behavior of other objects from the same class.
- To implement the **Decorator pattern**, we just need to implement the interface of the extended (decorated) object (Component) transparently by forwarding all requests to it.
- The **Decorator Class** has a member in Component type, and we will initialize it with the ConcreteComponent.
- For the operation inherited from the Component class, we will use this member's method to implement it, and then we can extend the Component with extra method

Decorator Pattern: Demo

- Suppose we have a coffee shop, the coffee may be added with milk or sprinkles, each of them have a different cost, now we can use the **Decorator Pattern** to implement it.



Decorator Pattern: Demo

- First, we define the Coffee Class, SimpleCoffee Class and the BlackCoffee Class.

```
public interface Coffee {
    public double getCost();
    public String getIngredients();
}
```

```
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1;
    }

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

```
public class BlackCoffee implements Coffee {
    @Override
    public double getCost() {
        return 2;
    }

    @Override
    public String getIngredients() {
        return "BlackCoffee";
    }
}
```

Decorator Pattern: Demo

- Then, we define a **CoffeeDecorator** Class.

```
public abstract class CoffeeDecorator implements Coffee {
    private final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }

    @Override
    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}
```

a Coffee class member

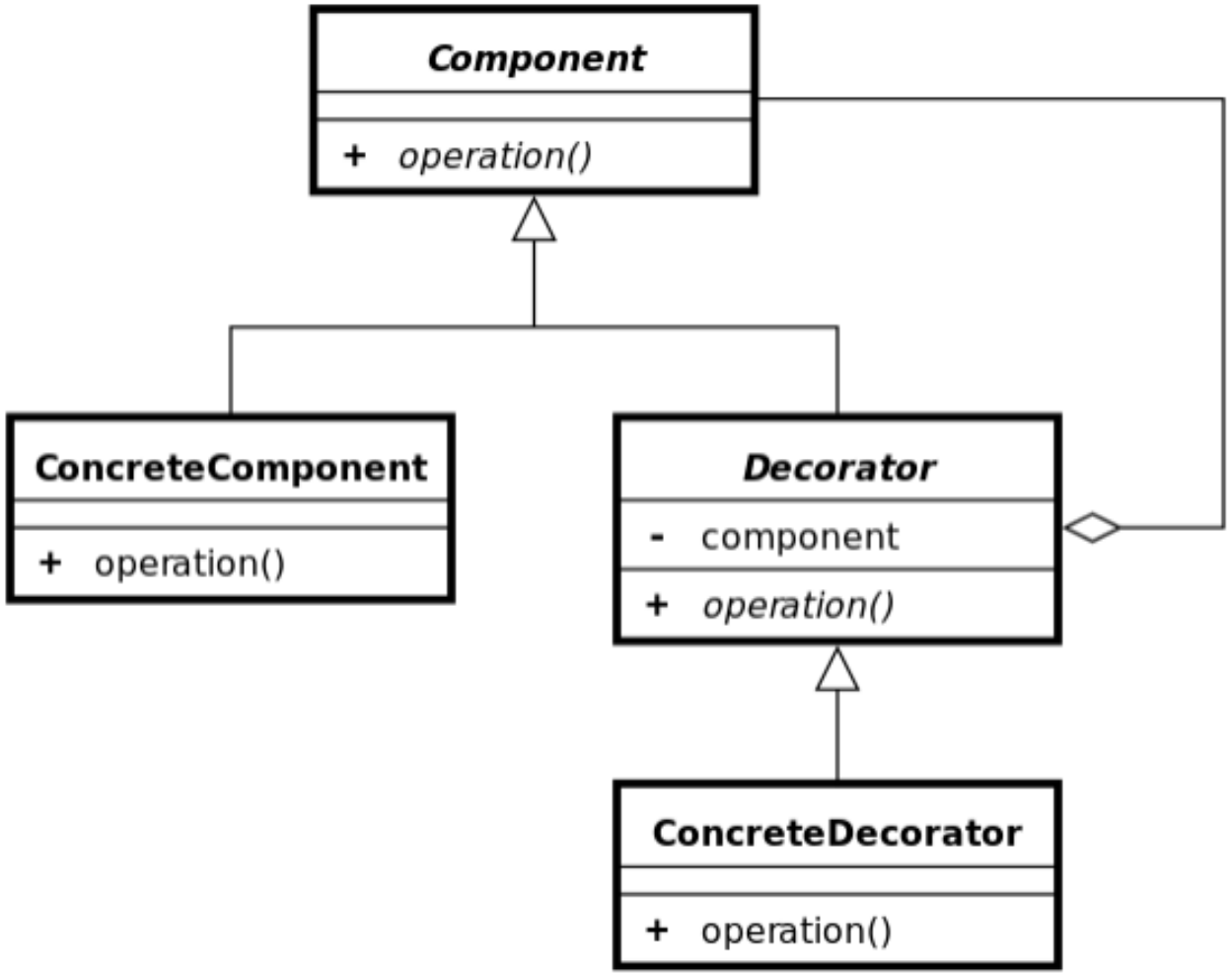
Override methods by transfering them to decorated coffee member

Decorator Pattern: Demo

- Finally, we define the WithMilk Class and the WithSprinkles Class, extends the function we want.

```
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

    @Override
    public String getIngredients() {
        return super.getIngredients() + ", Milk";
    }
}

class WithSprinkles extends CoffeeDecorator {
    public WithSprinkles(Coffee c) {
        super(c);
    }

    @Override
    public double getCost() {
        return super.getCost() + 0.2;
    }

    @Override
    public String getIngredients() {
        return super.getIngredients() + ", Sprinkles";
    }
}
```

Decorator Pattern: Class Diagram



Proxy Pattern

- In **Proxy Pattern**, the class can provide a sub-interface to another class, which provide other object with a proxy to control access to that object.
- It can deal with the problems that arise when directly accessing objects makes the system get into trouble.
- Its main ambition is: Providing a similar Object, and optimization of it, or a constaint because of potential security problem.

Proxy Pattern: Role

- To Provide a similar object, and limit the method that can be use, The **Role of Proxy Pattern** should be designed with characteristic below:
  - **Base Class** : a class that will be inherited by the RealObject Class and the Proxy Class. The methods in Base Class will not be limited by the Proxy.
  - **RealObject Class** : the class inherited from the Base Class, and maybe have extra method.
  - **Proxy Class** : the class inherited from the Base Class, and it will have an object of **RealObject Class**.
- The **Proxy Class** will initialize the **RealObject Class** object in it , and use the object to implement the method of the Base Class.

Proxy Pattern: Demo

- Suppose we have a RealImage Class, when we initialize a object of it, it will load the picture from disk immediately. When the size of picture is big, it will lead to a big comsume of memory. We can use Proxy Pattern to provide a optimization.
- In this demo, you can know how to use Proxy Pattern to optimize the problem we meet : how to adapt the class such that it loads the picture only if we when need.

Proxy Pattern: Demo

```java
public interface Image {
    void display();
}
```

```java
public class RealImage implements Image {

    private String fileName;

    public RealImage(String fileName){
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + fileName);
    }

    private void loadFromDisk(String fileName){
        System.out.println("Loading " + fileName);
    }
}
```

Proxy Pattern: Demo

- We Define a **ProxyImage Class**, which contains:
  - a member of Real Image
  - display method inherited from the Image Interface.

```
public class ProxyImage implements Image{

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}
```

Proxy Pattern: Demo

- The optimization provided by the proxy class is:
  - When we Initialize the ProxyImage Class, it just save the file name, but will not load the picture immediaty.
  - When the display method is called, we will initialize the RealImage member in it, and use the display method to show it.
  - So we can provide a optimization that: we provide a optimized class which will load the picture when we need it, without change the original class.

Proxy Pattern: Class Diagram