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

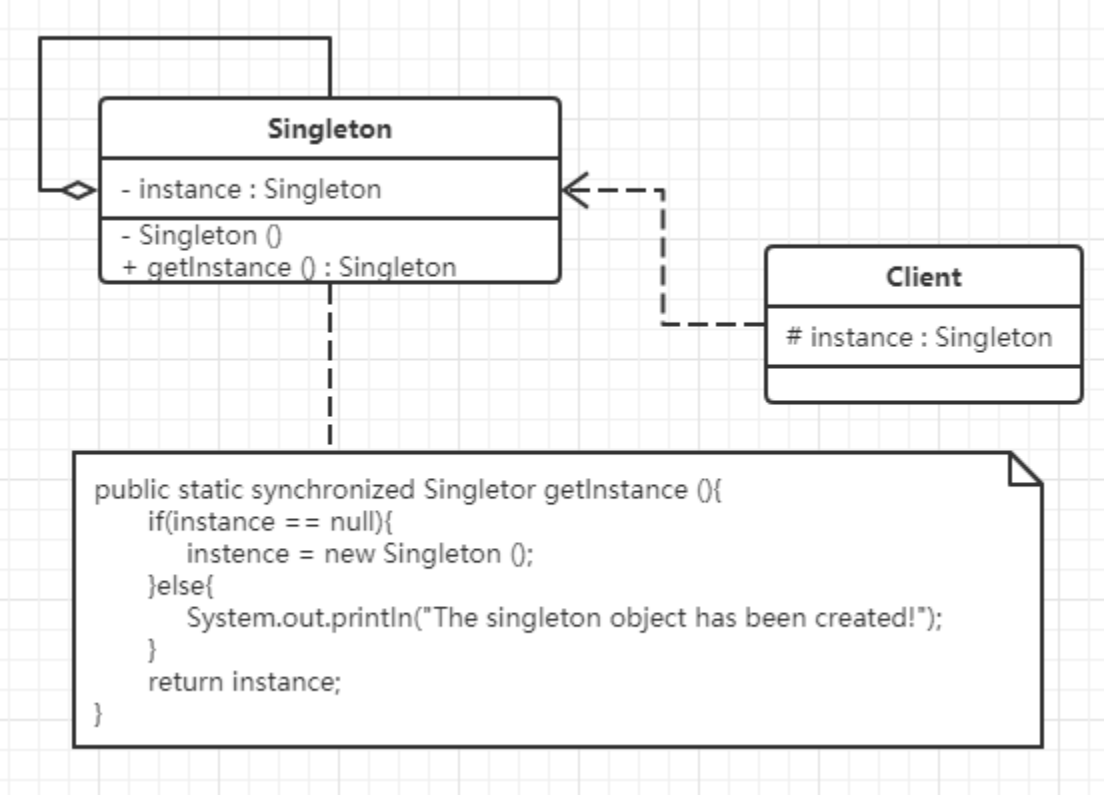
- **Structural Patterns**
  (how objects/classes can be combined to form larger structures)
  - Adapter
  - Bridge
  - *Composite*
  - *Decorator*
  - Facade
  - Flyweight
  - Proxy

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

Pattern: Singleton

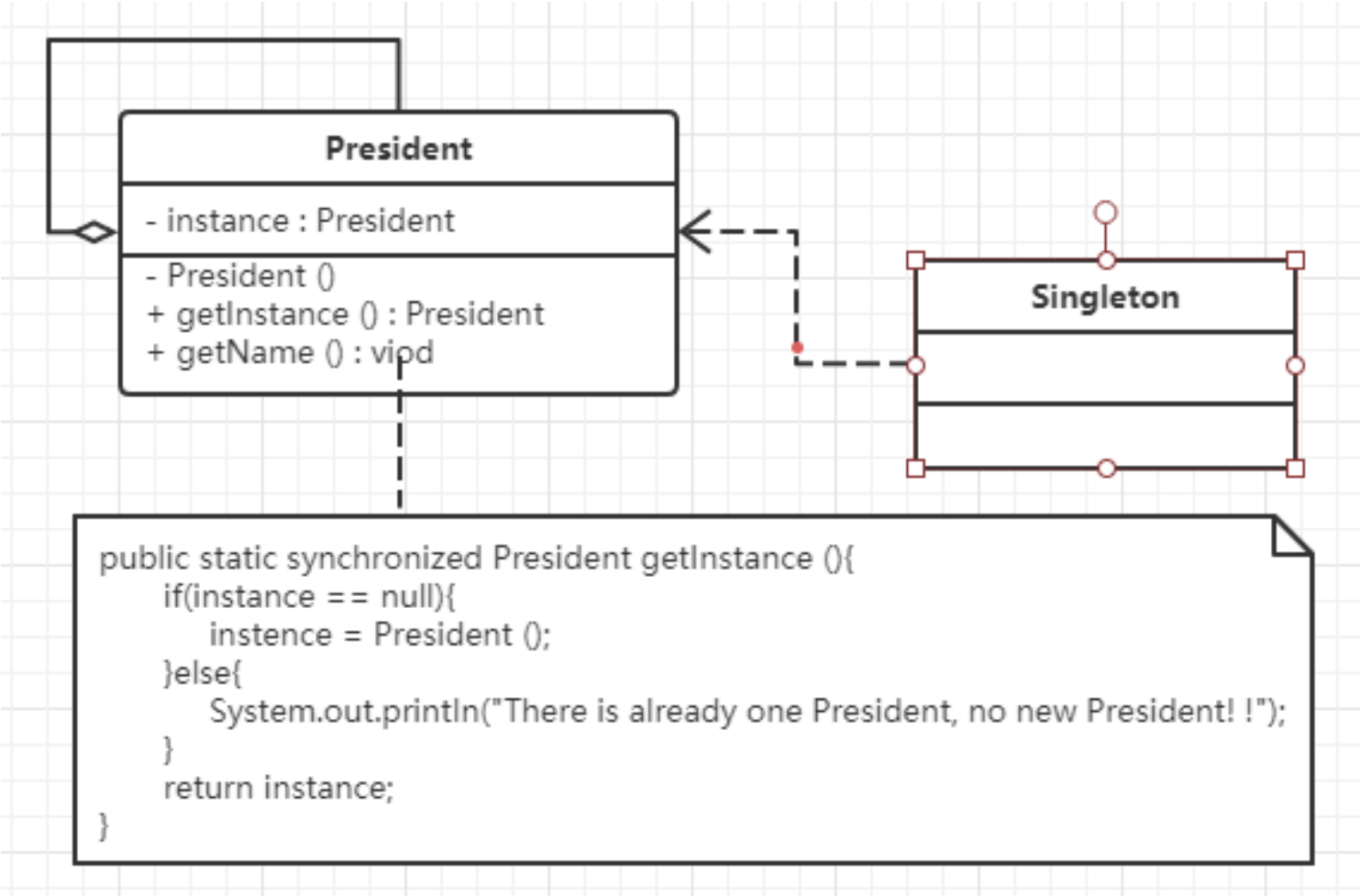
- *A pattern in which a class has only one instance and the class can create that instance itself*

- 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

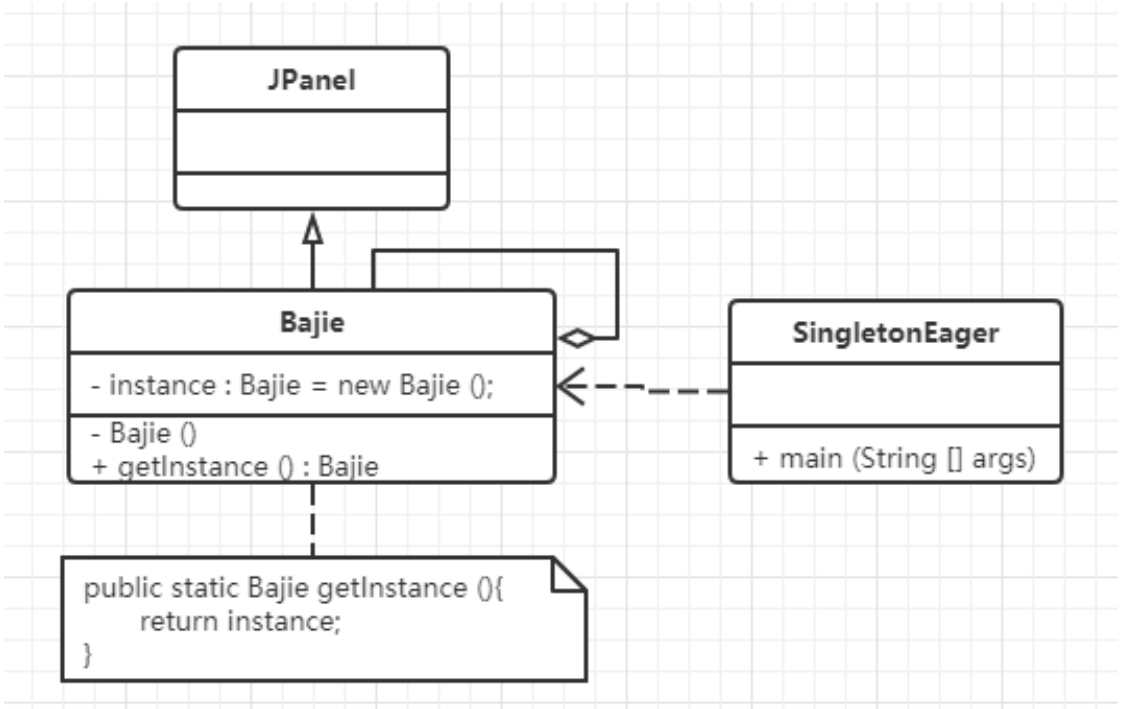


Singleton: example1

```
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

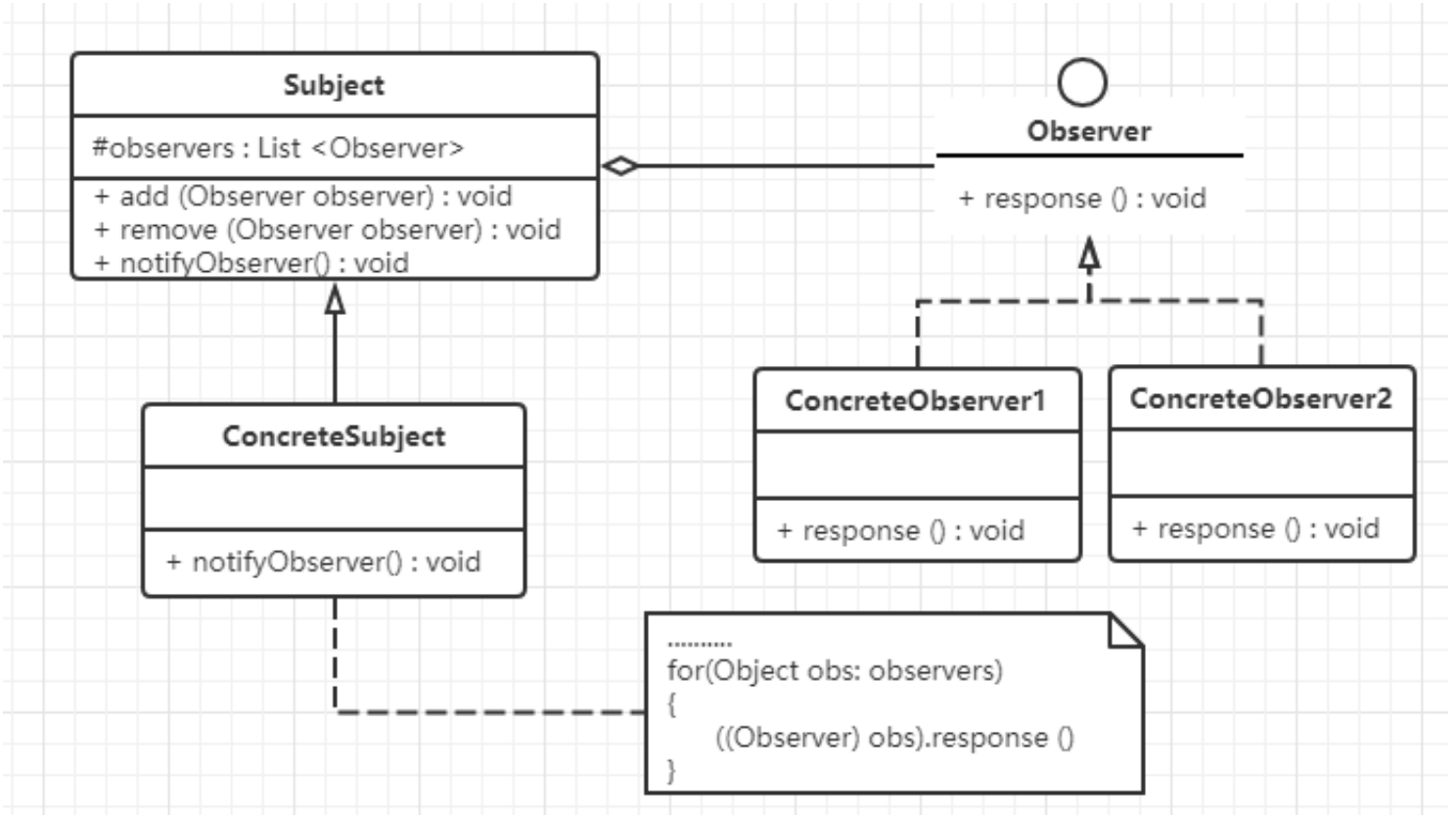



```
class Bajie extends JPanel
{
    private static Bajie instance=new Bajie();
    private Bajie()
    {
        JLabel l1=new JLabel(new ImageIcon("Bajie.jpg"));
        this.add(l1);
    }
    public static Bajie getInstance()
    {
        return instance;
    }
}
```

```
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);
    }
}
```

Pattern: Observer

- *objects whose state can be watched*



Observer: example1

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

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

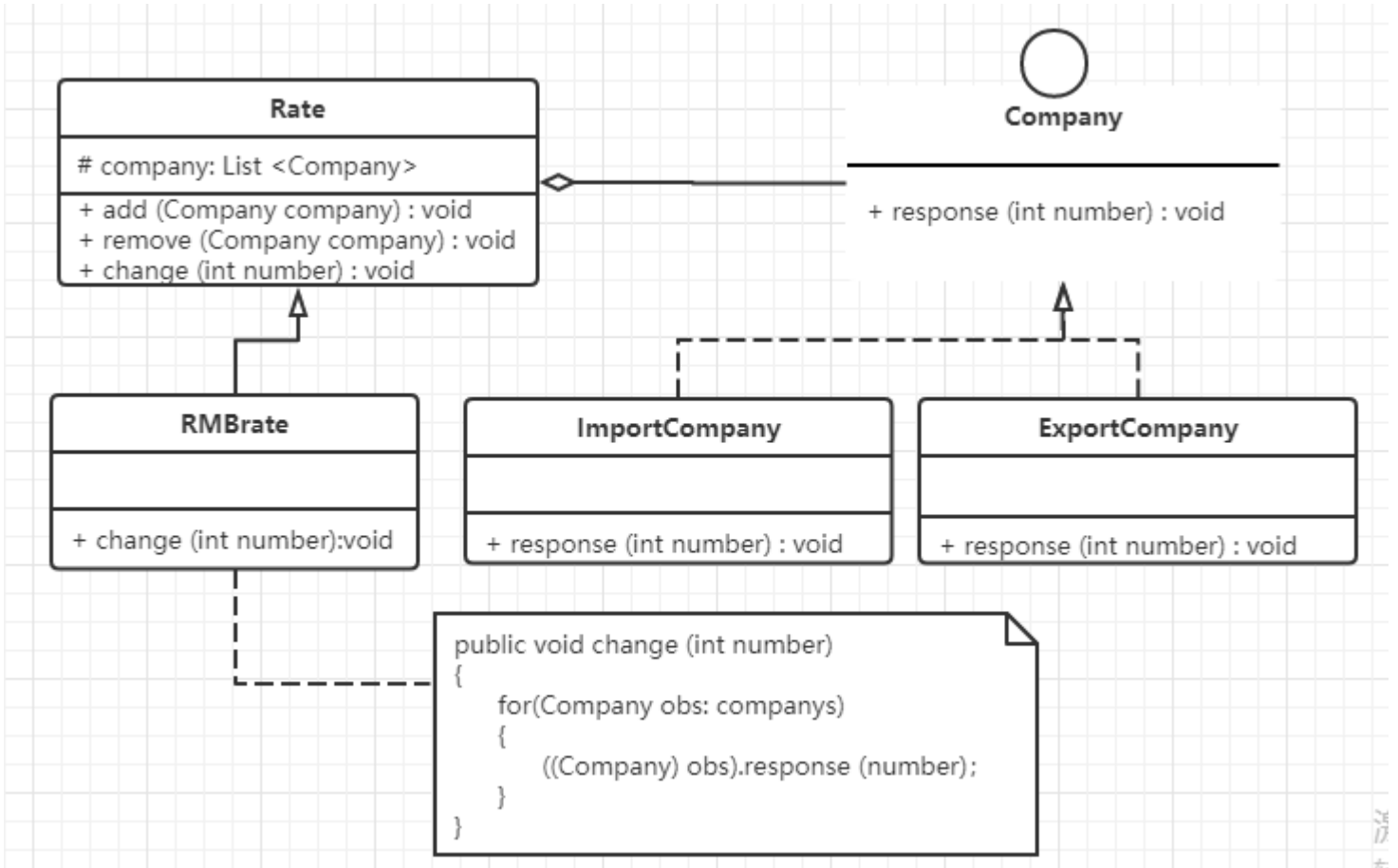
Concrete subject

```
//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();
        }

    }
}
```

Concrete observer

```
//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: example2



Observer: example2

subject

```
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

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

Concrete observer

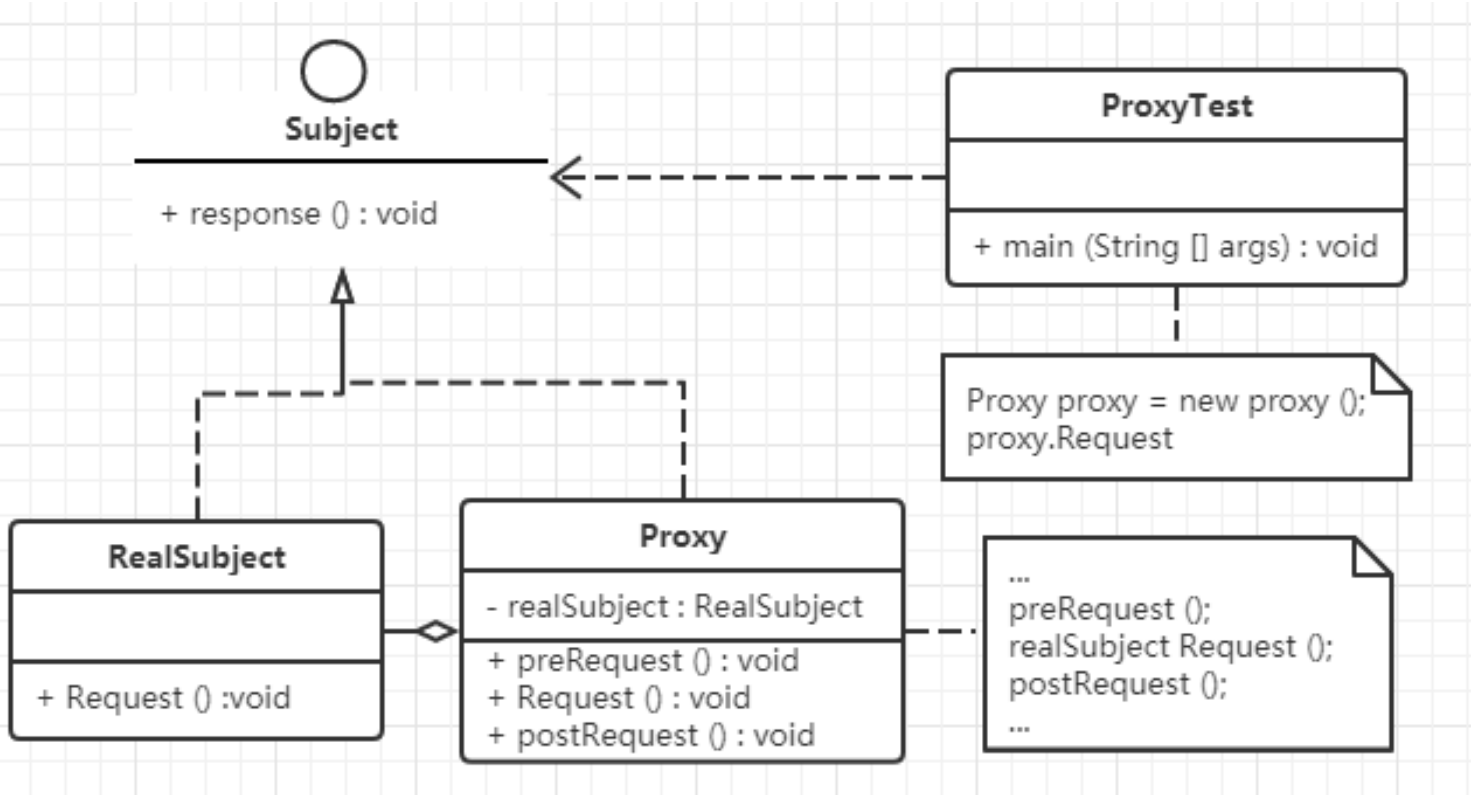
```
//concreteObserver1: ImportCompany
class ImportCompany implements Company
{
    public void response(int number){
        if(number>0){
            System.out.println("RMBrate increase, improved the profit margin of import companies");
        }
        else if(number<0){
            System.out.println("RMBrate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

Concrete subject

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

}
```

Pattern: Proxy

- an object with a proxy to control access to that object.



Proxy: example1

subject

```
//subject : Specialty
interface Specialty
{
    void display();
}
```

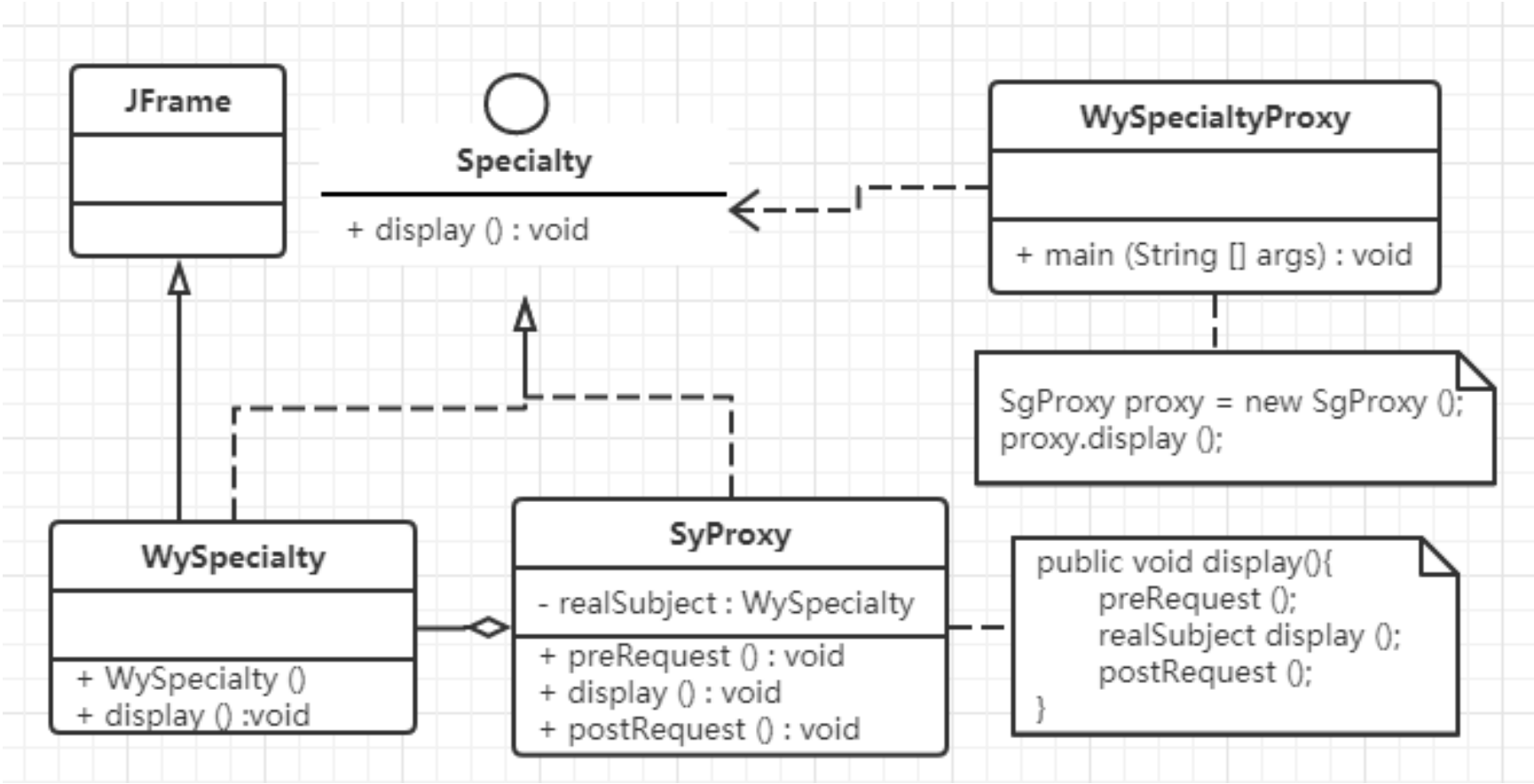
Real Subject

```
//RealSubject
class RealSubject implements Subject
{
    public void Request()
    {
        System.out.println("Access realSubject method");
    }
}
```

proxy

```
//proxy
class Proxy implements Subject
{
    private RealSubject realSubject;
    public void Request()
    {
        if (realSubject==null)
        {
            realSubject=new RealSubject();
        }
        preRequest();
        realSubject.Request();
        postRequest();
    }
    public void preRequest()
    {
        System.out.println("preRequest");
    }
    public void postRequest()
    {
        System.out.println("postRequest");
    }
}
```

Proxy: example2



Proxy: example2

subject

```
//subject : Specialty
interface Specialty
{
    void display();
}
```

Real Subject

```
//realSubject: WySpecialty
class WySpecialty extends JFrame implements Specialty
{
    private static final long serialVersionUID=1L;
    public WySpecialty()
    {
        super("SgProxt WySpecialty Test");
        this.setLayout(new GridLayout(1,1));
        JLabel l1=new JLabel(new ImageIcon("WuyuanSpecialty.jpg"));
        this.add(l1);
        this.pack();
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public void display()
    {
        this.setVisible(true);
    }
}
```

proxy

```
//Proxy: SgProxy
class SgProxy implements Specialty
{
    private WySpecialty realSubject=new WySpecialty();
    public void display()
    {
        preRequest();
        realSubject.display();
        postRequest();
    }
    public void preRequest()
    {
        System.out.println("SgProxy WyProxy starts");
    }
    public void postRequest()
    {
        System.out.println("SgProxy WyProxy ends");
    }
}
```