Software Architecture
Course's Code: CSE 483
Classic Design Patterns in Java EE (JEE)
(Chapter 2)

Chapter 2

Chapter 2. Class Design Patterns in Java EE

A scenario of using Singleton Design Pattern

Robinhood is an investment platform offering commission-free trading of stocks, ETFs, cryptocurrency, and options—all through a mobile app.

In 2022, it has up to 23 million users.

When the market is at high,
the increase of trading volume
challenging the database system.

Singleton Design Pattern

Singleton is a creational design pattern,

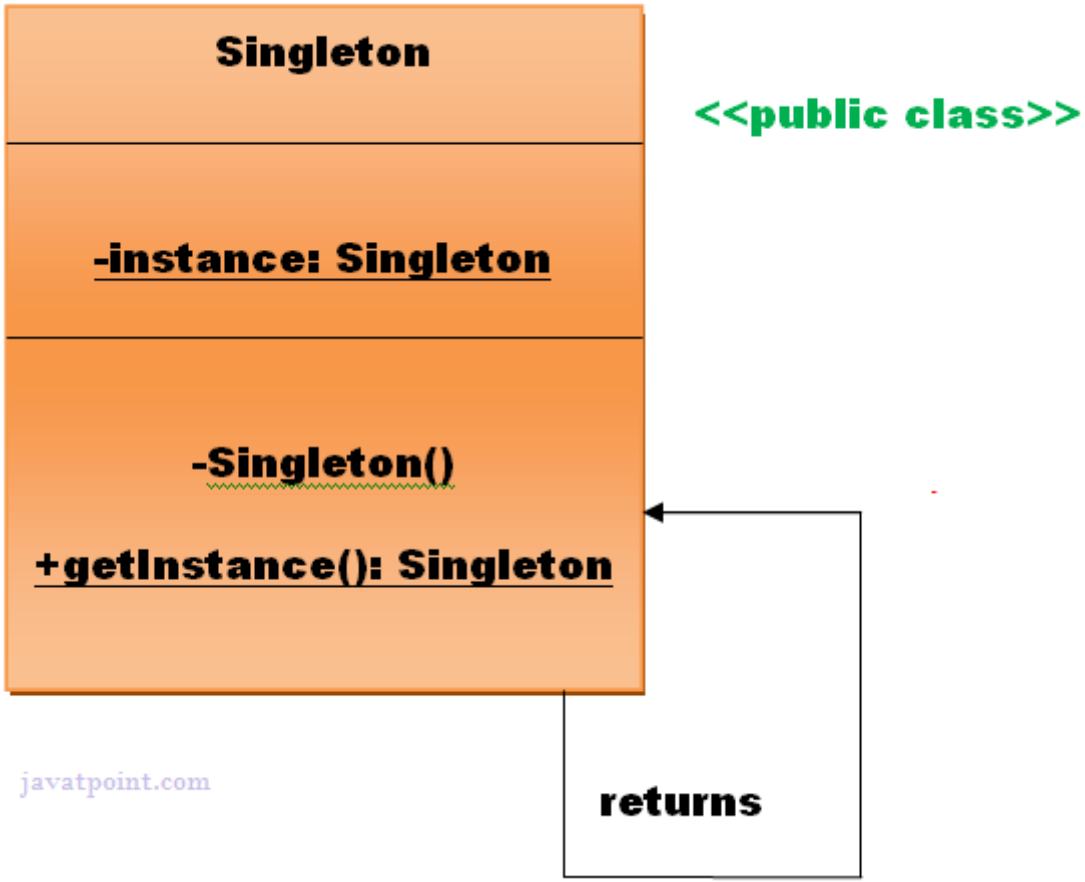
1. Only one instance of that class exists in the Java Virtual Machine

2. Provide a global access point to the instance of the class

Implement the Singleton Pattern

To create the singleton class, it will contain:

- **static member**: gets memory only once because of static
- **private constructor**: prevent to initialize the singleton class from outside class
- **static method**: provide the global point of the access to the Singleton object and return the instance



Implement the Singleton Pattern – Eager Initialization

```
public class DBConnection {

        // static member
        private static DBConnection instance=new DBConnection();

        //private constructor
        private DBConnection()
        {
                //code to set up a db connection
        }

        //static method
        public static DBConnection getInstance()
        {
                return instance;
        }

        public void getData()
        {
                //some code
        }

}
```

Implement the Singleton Pattern – Lazy Initialization

```
public class DBConnection {

        // static member
        private static DBConnection instance ;

        //private constructor
        private DBConnection()
         {
                //code to set up a db connection
        }

        //static method
        public static DBConnection getInstance()
        {
                if(instance==null)
                {
                    instance=new DBConnection();
                }
                return instance;
        }

        public void getData()
        {
                //some code
        }

}
```

Implement the Singleton Pattern – Thread Safe

```
public class DBConnection {

        // static member
        private static DBConnection instance;

        //private constructor
        private DBConnection()
         {
                //code to set up a db connection
        }

        //static method
        public static synchronized DBConnection getInstance()
        {
                if(instance==null)
                {
                    instance=new DBConnection();
                }
                return instance;
        }


        public void getData()
        {
                //some code
        }
}
```

Chapter 2. Class Design Patterns in Java EE

A scenario of using Facade Design Pattern

In the banking domain, customer comes to the bank to borrow money to purchase a vehicle.

Bank manager enters customer ‘s info in the system:

- ID number
- Car model

Computer says YES or NO to the lending application.

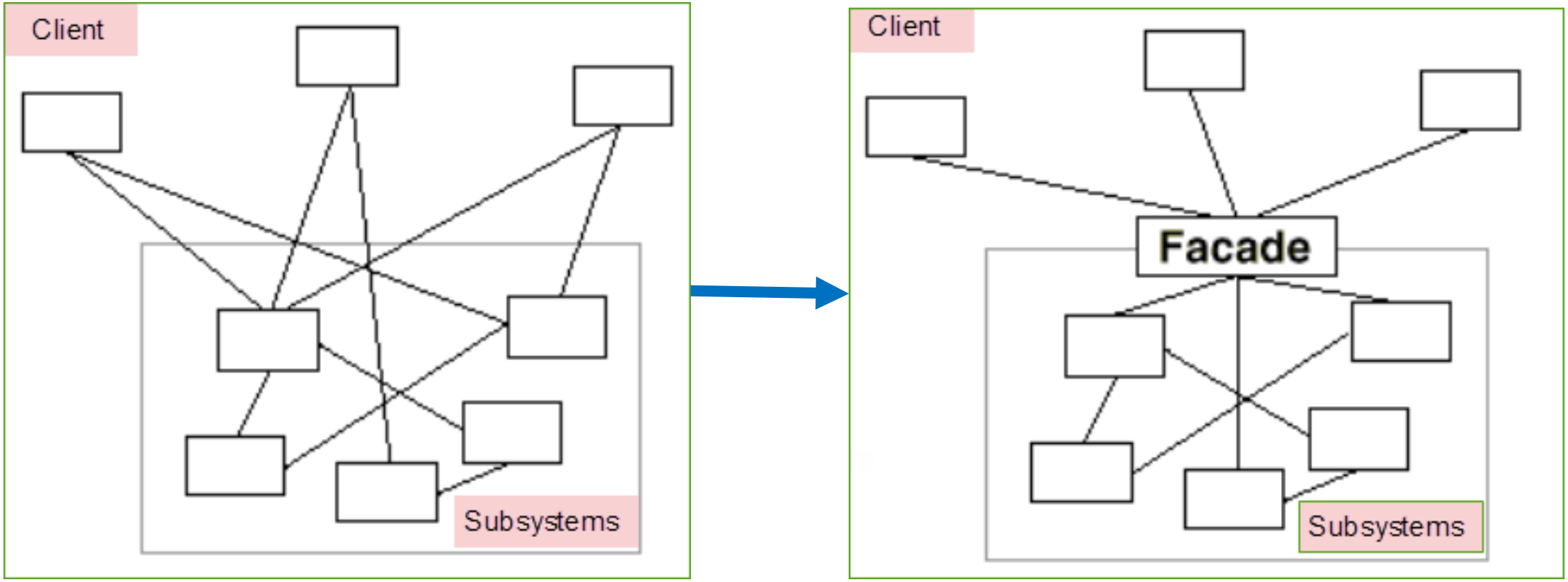
A scenario of using Facade Design Pattern

Behind the scenes, the backend system will check:

- the credit rating (Equifax, TransUnion..)
- the income (W2, IRS 1099,..)
- value of the car (Kelly Blue Book)
- issue Approval Letter if YES (Adobe)
- .....

More subsystems required

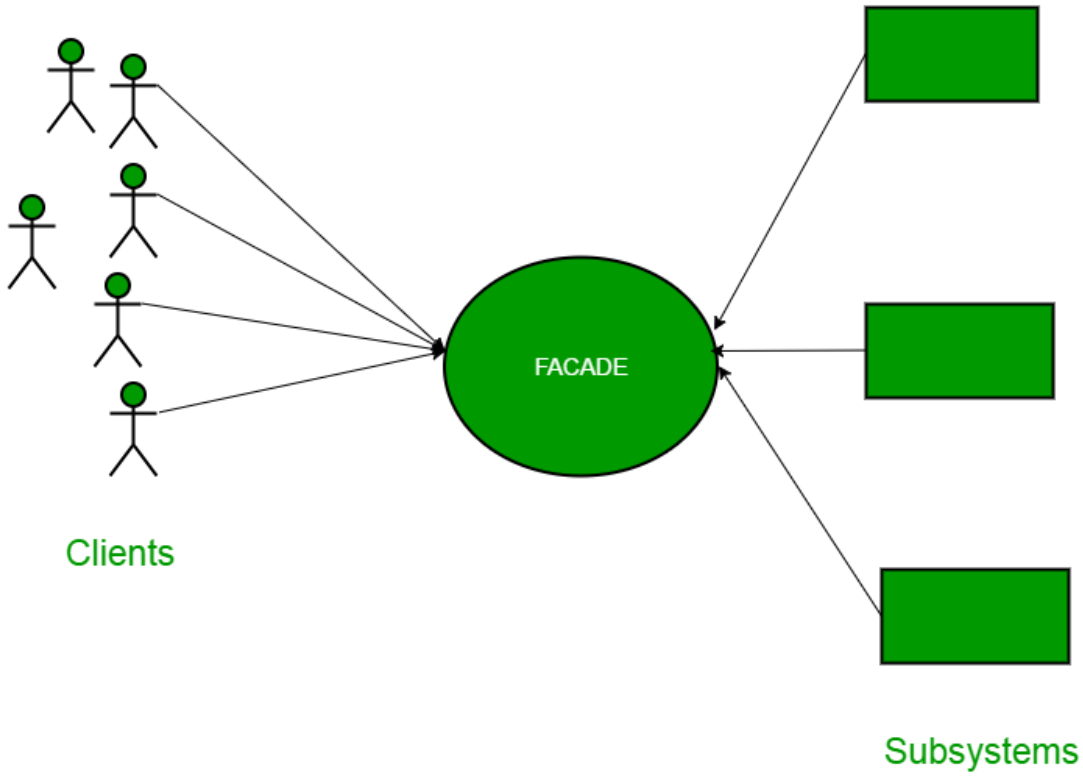
Using Facade Design Pattern



Facade Design Pattern

Facade is a Structural design pattern,

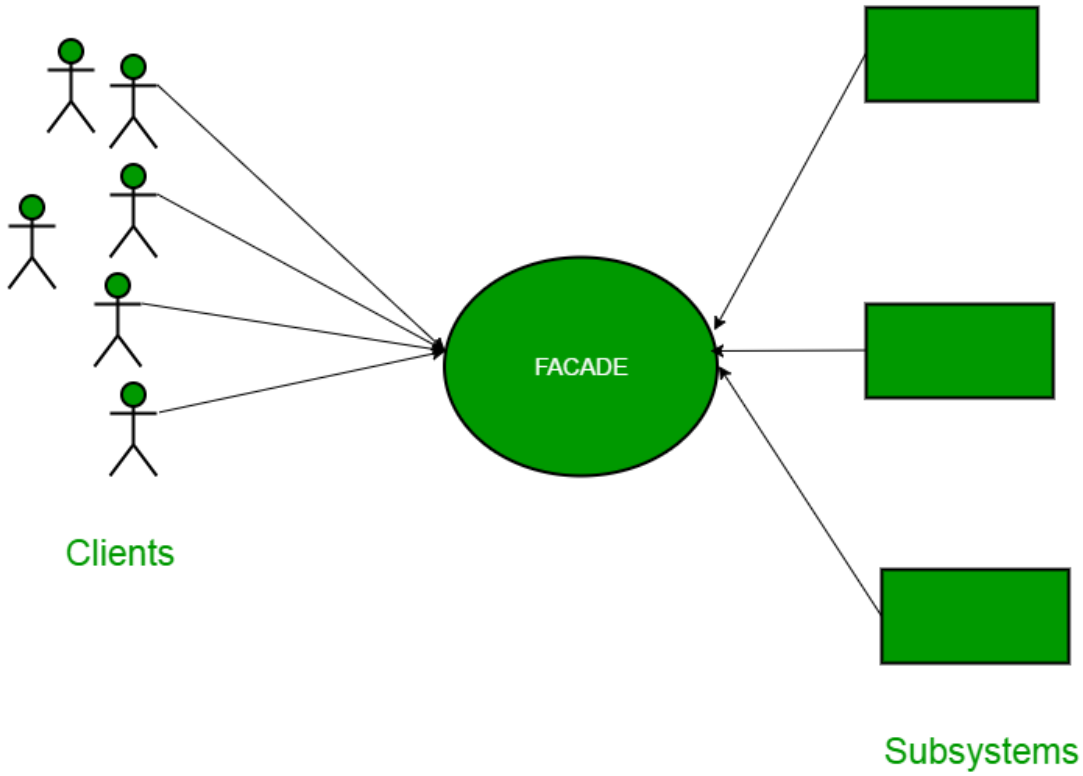
Façade pattern provides a unified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client



Facade Design Pattern

Facade pattern provides an interface to the client using which the client can access the system.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.



Breakdown of the components

1. Client class:

Represents the client code that interacts with the facade

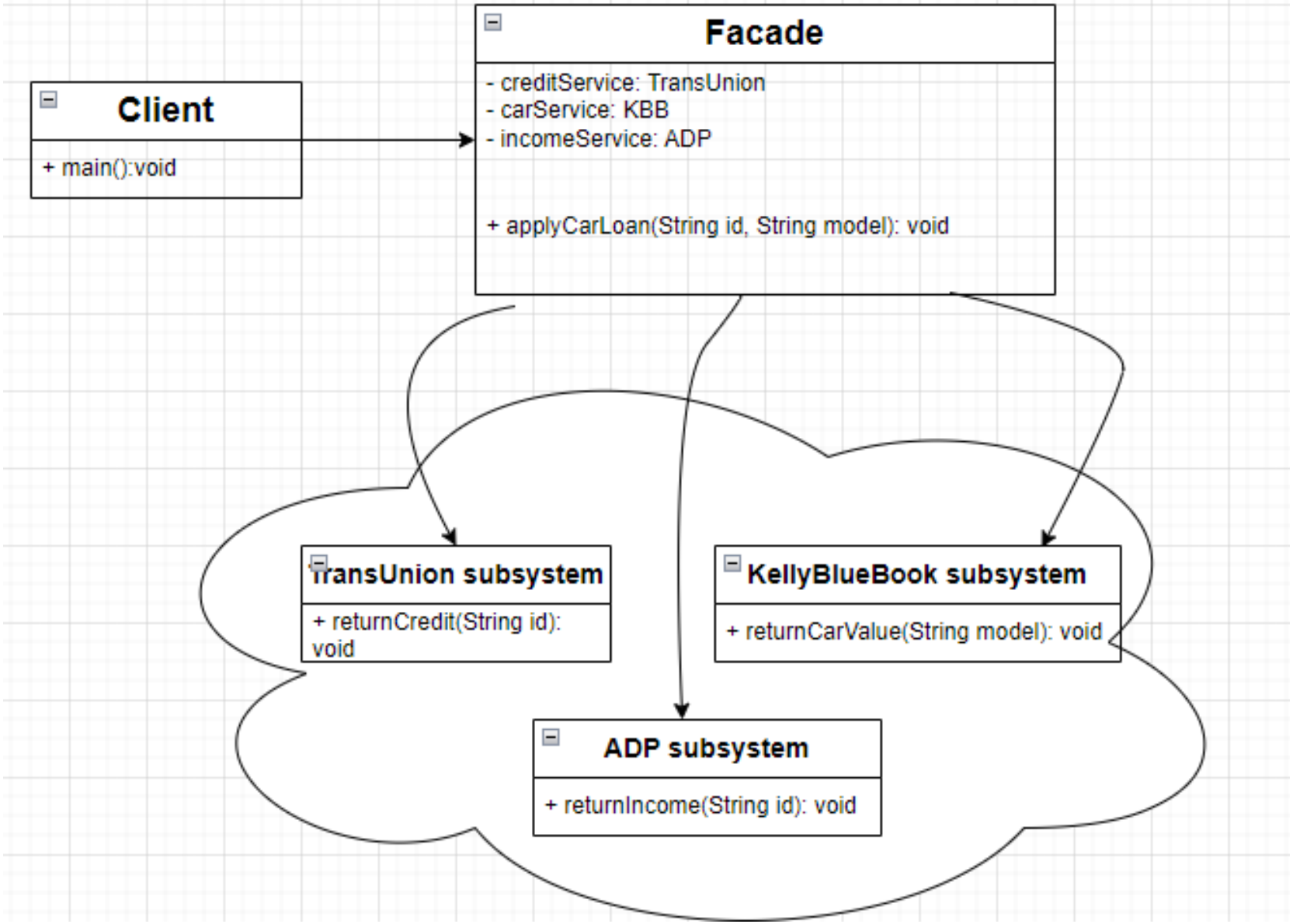
2. Facade class:

Provides a simplified interface for higher-level actions. It delegates tasks to the appropriate subsystem components

3. Subsystem Components

Simulates the functionality of a service

Structure of Facade Design Pattern



Implement the Facade Pattern

```
//Simulate TransUnion subsystem
Public class TransUnion {
        public void checkCredit(String id)
        {
                //codes to check credit
                System.out.print("Check credit of " + id+"\n");
        }
}
```

```
//Simulate KellyBlueBook subsystem
Public class KBB {
        public void checkCarValue(String model)
        {
                //codes to pull car info and value
                System.out.print("Check value of Model " + model+ "\n");
        }
}
```

Implement the Facade Pattern

```
//Simulate ADP subsystem
Public class ADP {
        public void checkIncome(String id)
        {
                //codes to retrieve the income of the customer
                System.out.print("Check income of " + id+ "\n");
        }
}

//Simulate Adobe subsystem to issue pdf file
Public class Adobe {
        public void issueApproval()
        {
                //codes to issue the pdf document
                System.out.print("issue Approval Letter ");
        }
        public void issuePreApproval()
        {
                //codes to issue the pdf document
                System.out.print("issue Pre-Approval Letter ");
        }
}
```

Implement the Facade Pattern

```
//Simulate Façade class
Public class Façade {
        private TransUnion transUnionService;
        private KBB kbbService;
        private ADP adpService;
        private Adobe adobeSevice;

        public Façade()
        {
                transUnionService=new TransUnion();
                kbbService=new KBB();
                adpService=new ADP();
                adobeService=new Adobe();
        }
        public void applyCarLoan(String id, String model)
        {
                transUnionService.returnCredit(id);
                kbbService.returnCarValue(model);
                adpService.returnIncome(id);

                //codes to return decision YES or NO
                adobeService.issueApproval();
        }
        public void applyPreApproval(String id)
        {
                transUnionService.returnCredit(id);
                adpService.returnIncome(id);

                //codes to return decision YES or NO
                adobeService.issuePreApproval();
        }
}
```

Implement the Facade Pattern

```
//Simulate Client
Public class lientDemo {

        public static void main(String[] args){
                Façade façade=new Façade();
                façade.applyCarLoan();
        }
}
```

Key points

1. The facade pattern is appropriate when you have a complex system that you want to expose to clients in a simplified way.

2. Facade design pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.

Advantages of the Facade Design Pattern

1. Simplified Interface:

Clients can interact with a single, easy-to-use interface provided by the Facade, hiding the complexity of the Subsystem.

2. Decoupling:

promotes loose coupling between the Client code and the Subsystem. Changes to the Subsystem are less likely to affect the Client code, leading to more maintainable and flexible systems

3. Encapsulation:

Clients only need to be aware of the Facade's interface, which promotes information hiding and encapsulation.

4. Maintenance and Refactoring:

Changes to the Subsystem can be made without affecting the clients. Clients interact with the Facade and not directly with the Subsystem components, modifications to the Subsystem's internal structure can be made without requiring changes to Client code.

5. Promotes Best Practices:

Facade pattern helps developers adhere to design principles by providing a clear and well-defined interface

Disadvantages of the Facade Design Pattern

1. Limited Flexibility:

Clients using the Facade may have limited control over the functionalities of the Subsystem.

2. Implicit Coupling:

Changes to the Facade or the Subsystem may inadvertently impact Clients.

3. Increased Complexity of the Facade:

As the system evolves, the Facade might become more complex. Over time, additional methods may be added to the Facade to accommodate new features or variations.

4. Reduced Visibility of System Complexity:

Developers using the Facade might not have a clear understanding of the internal workings of the subsystem. This lack of visibility can make troubleshooting and debugging more challenging.

Chapter 2. Class Design Patterns in Java EE

A scenario

YouTube channels have subscribers on their channel.

Subscribers already clicked on subscribe button of the channel to follow.

When the channel has latest videos, it will notify the subscribers to watch.

Another scenario

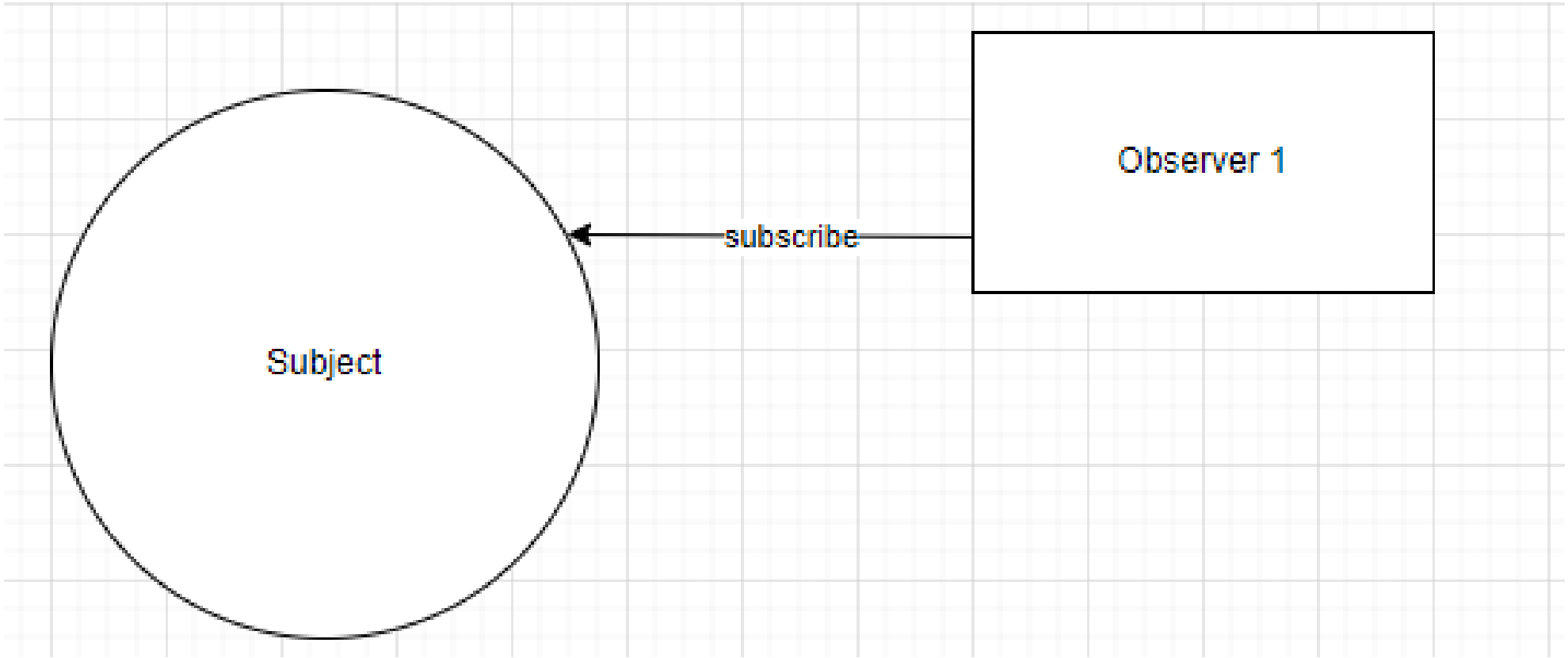
> 2000 stores

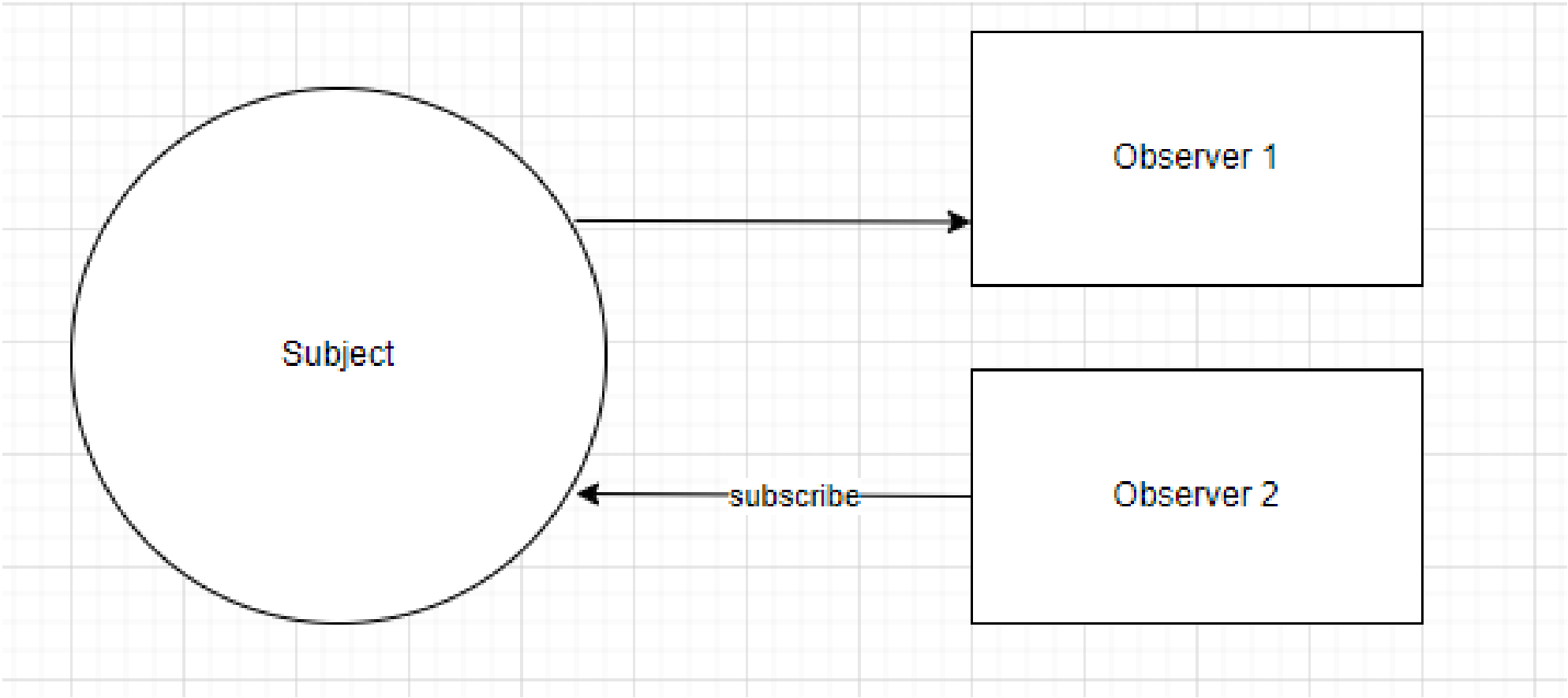
> 3000 stores

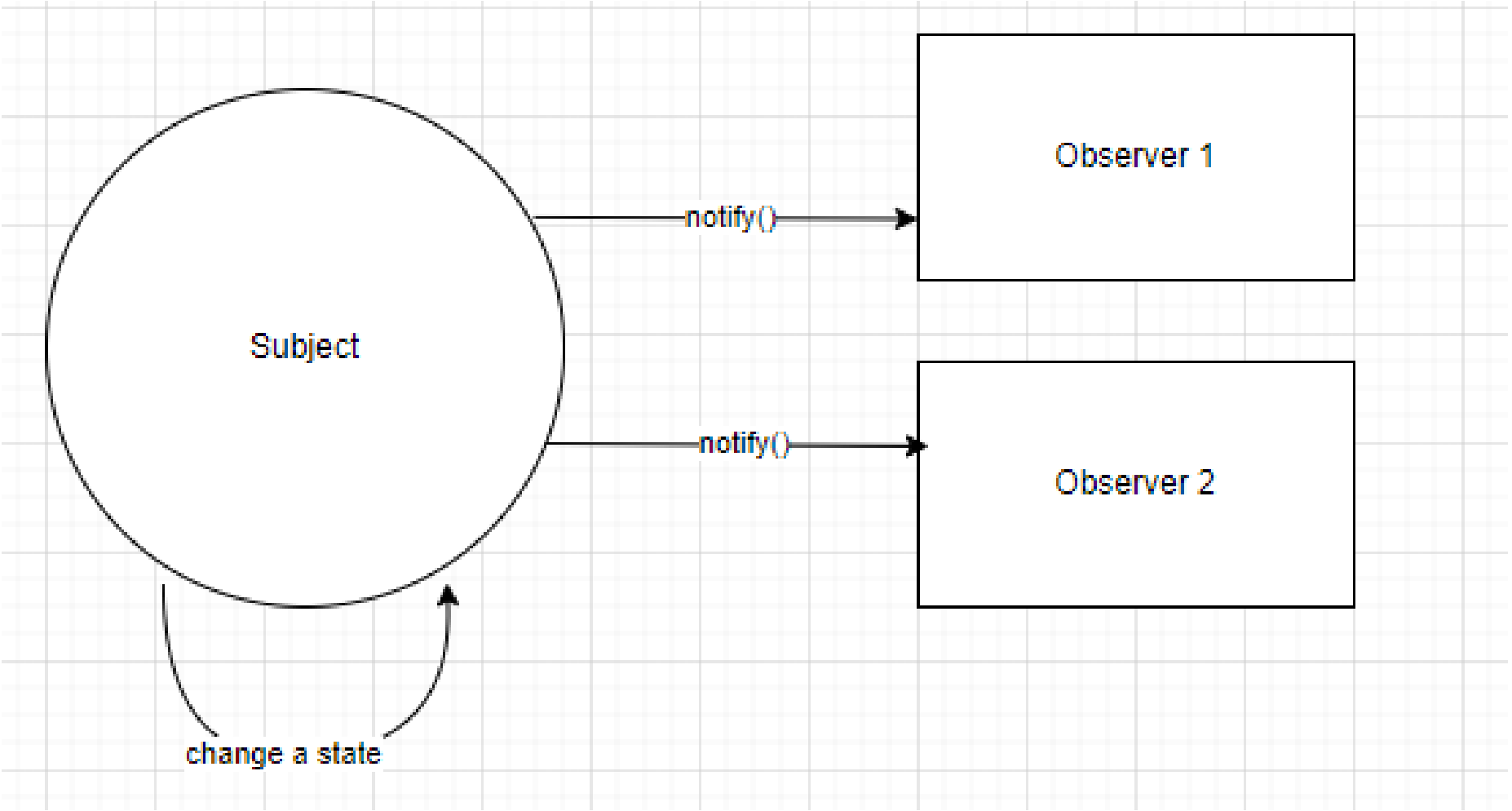
> 2000 stores

How they manage to display the selling prices?

How they manage to update the selling prices?

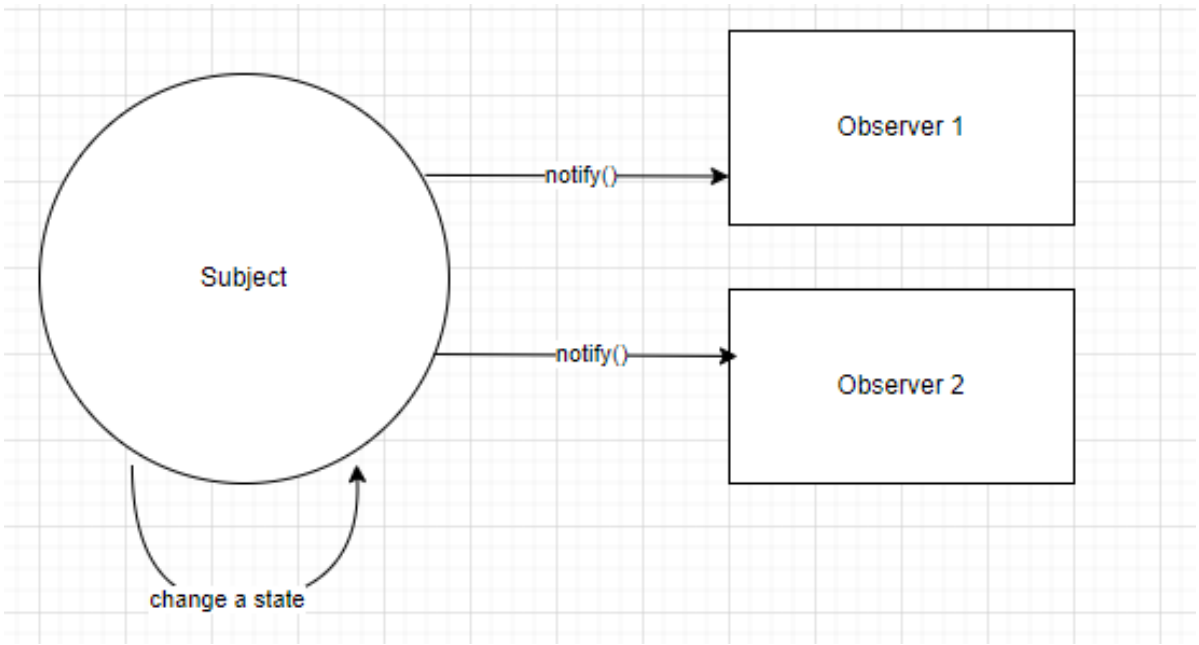

Observer 1
subscribe
Subject


Observer 1
Subject
subscribe
Observer 2


Observer 1
notify()
Subject
notify()
Observer 2
change a state

Observer Design Pattern

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



Breakdown of the components

1. Subject:

An interface providing methods to add, remove and notify Observers.

2. Observer:

An interface to provide an update() method for objects that will be notified when their state changes

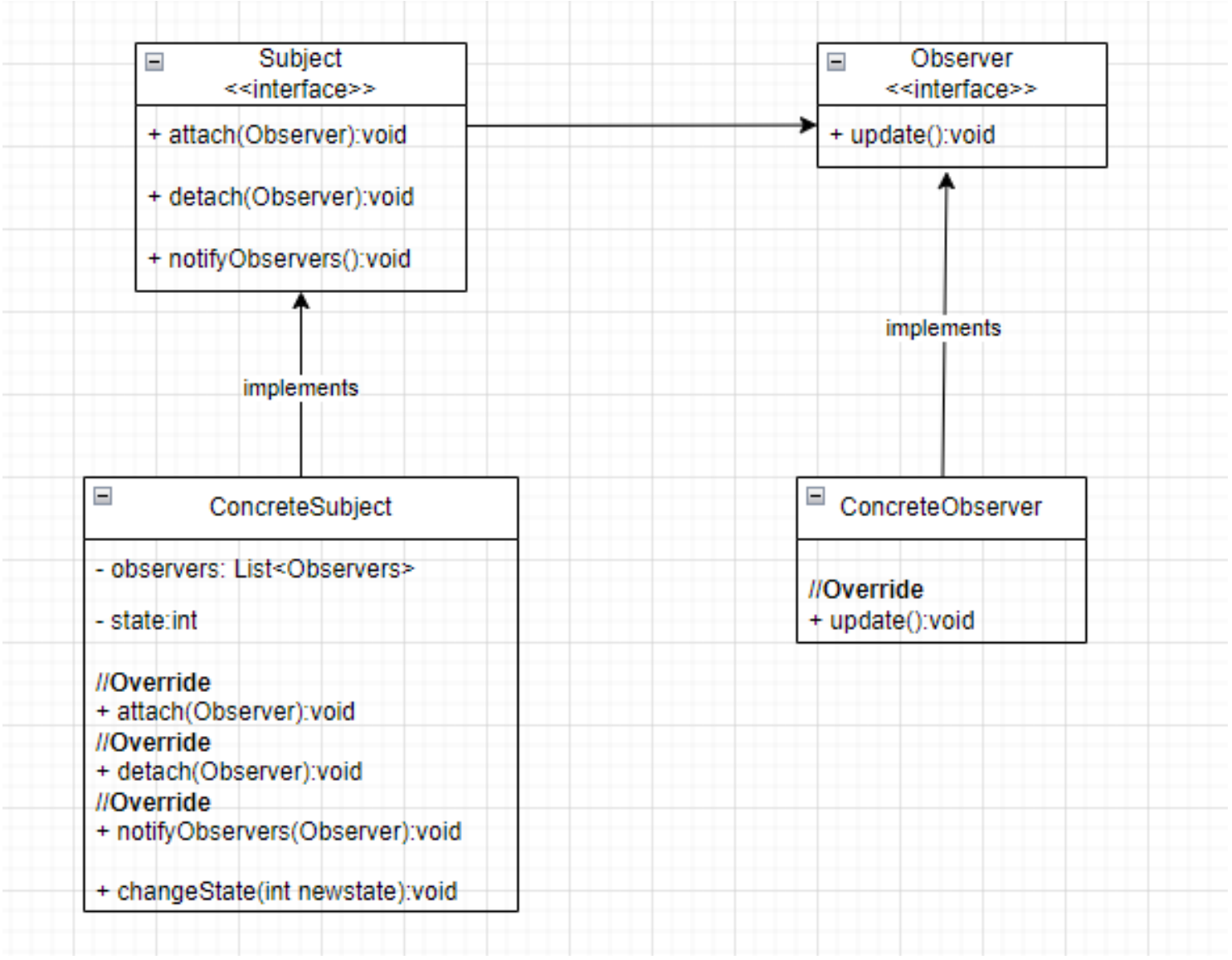
3. ConcreteSubject:

contains a list of Observers, implements the Subject's methods, sends notifications to its observers when the state changes.

4. ConcreteObserver:

implements Observer methods, stores the topic's state.

Structure of Observer Design Pattern



A review of polymorphism

2 types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.

A. compile-time polymorphism:

- method overloading
- method overriding

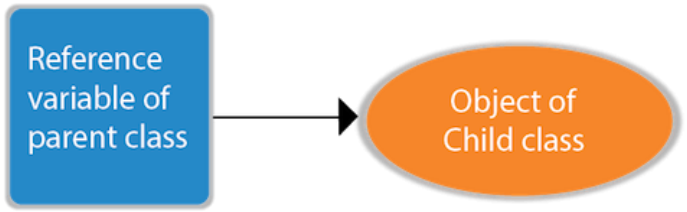
```
public class A
{
        public void displaySomething()
        {
                System.out.print(“print class A”);
        }
}
public class B extends A
{
        @Override
         public void displaySomething()
        {
                System.out.print(“print class B”);
        }
}
```

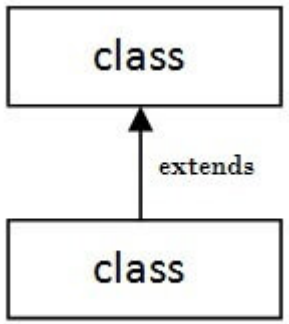
A review of polymorphism

2 types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.

B. runtime polymorphism: overridden method is called through the reference variable of a superclass.

Upcasting






```
public class A
{
        public void displaySomething()
        {
                System.out.print("print class A");
        }
}
public class B extends A
{
        @Override
         public void displaySomething()
        {
                System.out.print("print class B");
        }
}
```

A a=new B(); //upcasting

~~B b=new A();~~

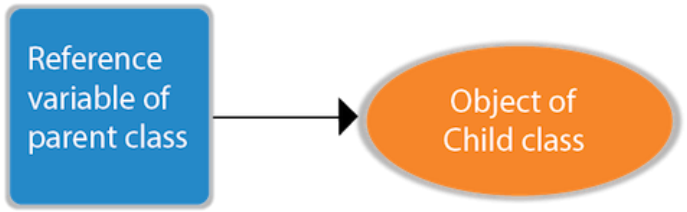
a.displaySomething(); //print class B

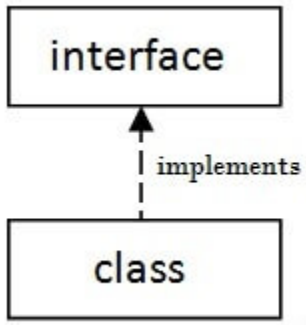
A review of polymorphism

2 types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.

B. runtime polymorphism: overridden method is called through the reference variable of a superclass.

Upcasting






```
public interface A
{
        public void displaySomething();
}
public class B implements A
{
        public void displaySomething()
        {
                System.out.print("print class B");
        }
}
```

A a=new B(); //upcasting

a.displaySomething(); //print class B

Implementation of Observer Design Pattern

1. Subject component

```
public interface Subject {
    public void attach(Observer o);
    public void detach(Observer o);
    public void notifyObservers();
}
```

Implementation of Observer Design Pattern

2. Observer component

```
public interface Observer {
    public void update(int state);
}
```

Implementation of Observer Design Pattern

3. ConcreteSubject component

```
public class ConcreteSubject implements Subject {
    private List<Observer> observers=new ArrayList<Observer>();
    private int state;

    @Override
    public void attach(Observer o)
    {
        if(!observers.contains(o))
        {
            observers.add( e:o);
        }
    }
    @Override
    public void detach(Observer o)
    {
        if(observers.contains(o))
        {
            observers.remove(o);
        }
    }
    @Override
    public void notifyObservers()
    {
        for(Observer o : observers)
        {
            o.update(state);
        }
    }

    public void changeState(int newstate)
    {
        this.state=newstate;
        System.out.print( s:"Subject has new State change\n");
        notifyObservers();
    }
}
```

Implementation of Observer Design Pattern

4. ConcreteObserver component

```
public class ObserverA implements Observer {

    @Override
    public void update(int state)
    {
        System.out.print( s:"Observer A update\n");
        System.out.print("Observer A get the new state: "+state+"\n");
        System.out.print("Observer A adds 3 to state:"+(state+3) +"\n");
    }
}
```

Implementation of Observer Design Pattern

4. ConcreteObserver component

```
public class ObserverB implements Observer {

    @Override
    public void update(int state)
    {
        System.out.print( s:"Observer B update\n");
        System.out.print("Observer B get the new state: "+state+"\n");
        System.out.print("Observer B adds 10 to state:"+(state+10) +"\n");
    }
}
```

Implementation of Observer Design Pattern

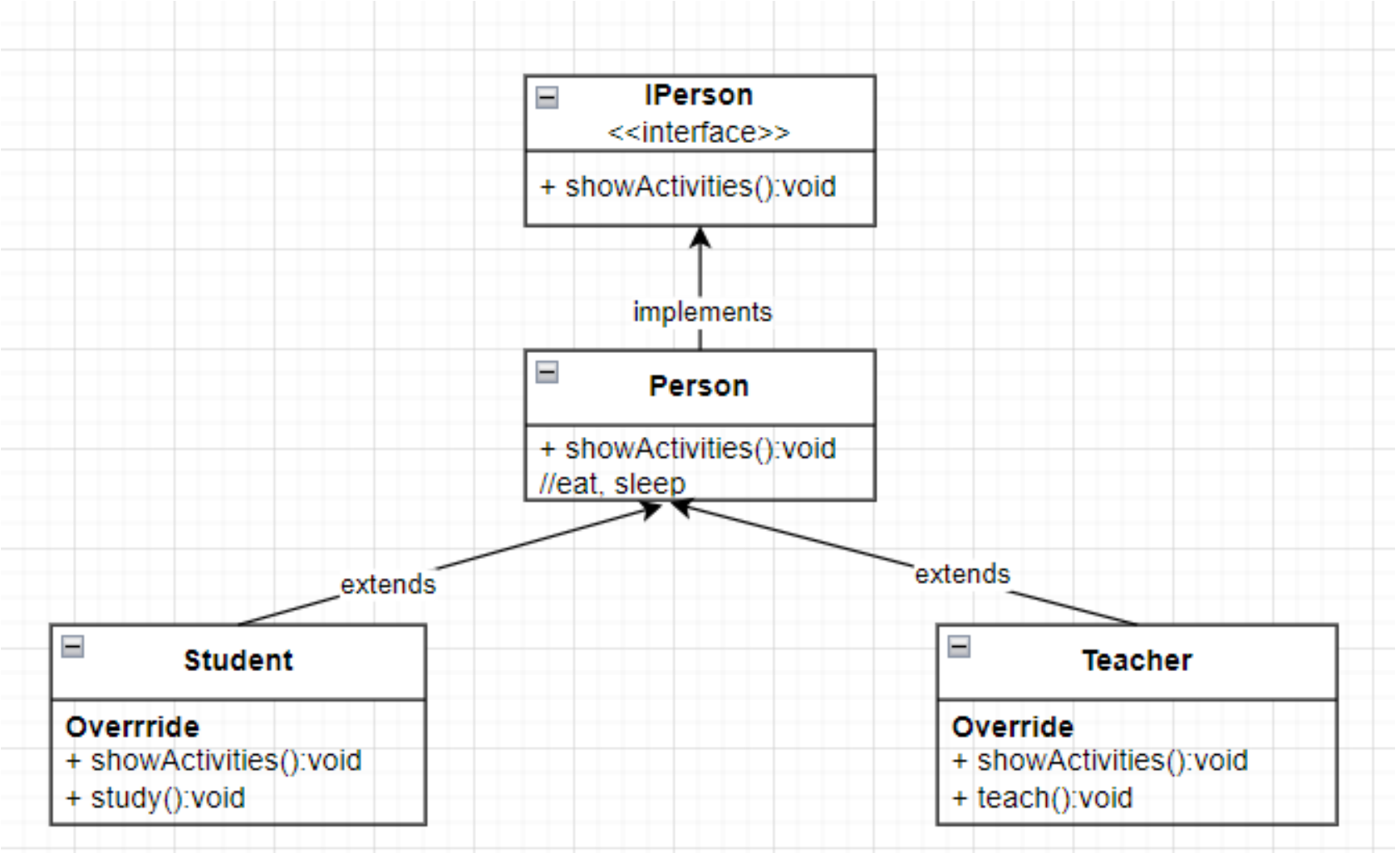
```
public static void main(String[] args) {
    // TODO code application logic here

    ConcreteSubject s=new ConcreteSubject();

    //initialize 2 new observers
    Observer a=new ObserverA();
    Observer b=new ObserverB();

    //register observer A to Subject
    s.attach( o:a);

    //register observer B to Subject
    s.attach( o:b);

    //Subject s change the state and expect observers A,B to act accordingly
    s.changeState( newstate:10);
}
```

Applicability of Observer Design Pattern

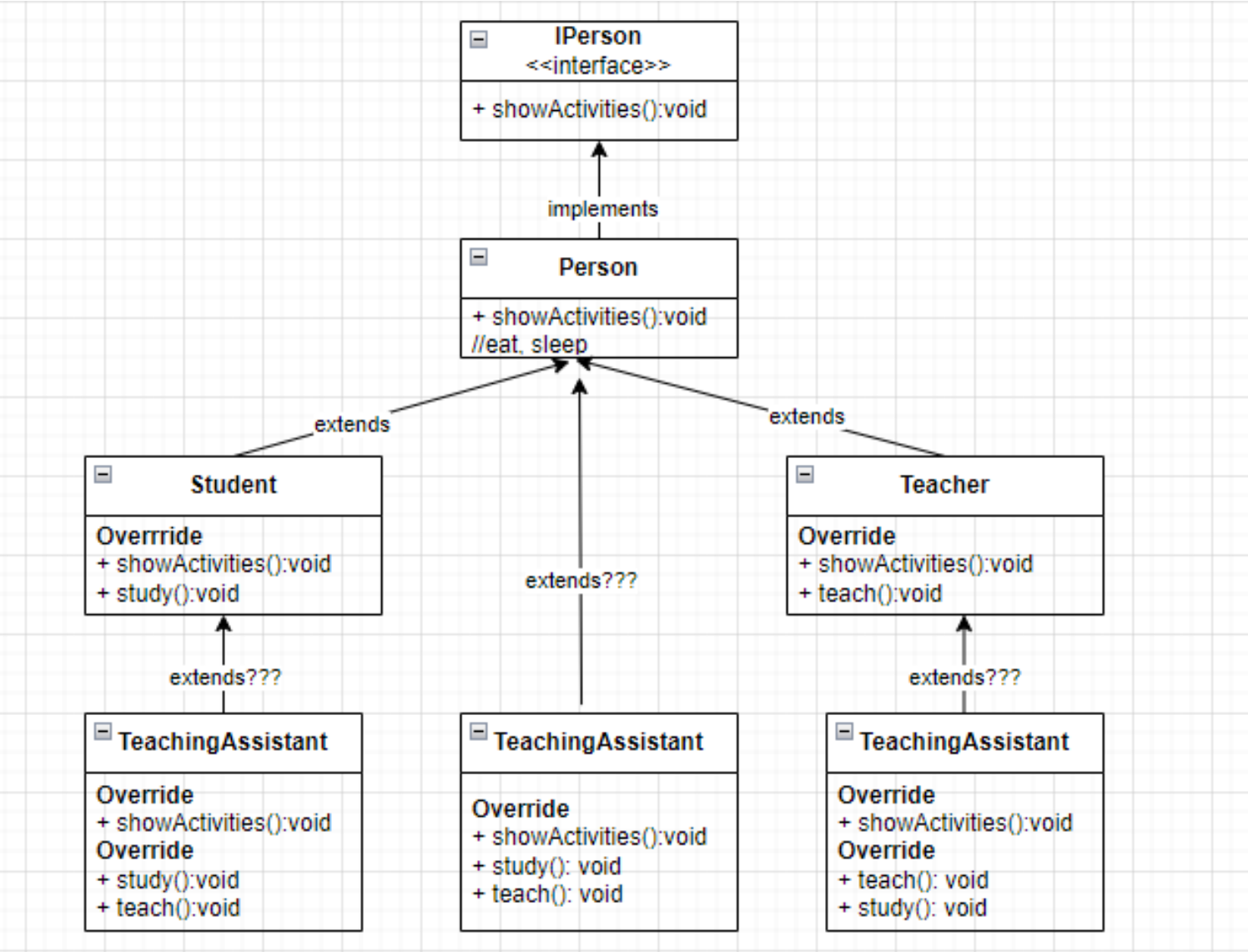
1. used if there are multiple dependencies on the state of one object
2. used to send notifications, emails, messages, etc..
3. subscribers of news or events

Chapter 2. Class Design Patterns in Java EE

Scenario


IPerson
<<interface>>
+ showActivities():void
implements
Person
+ showActivities():void
//eat, sleep
extends
extends
Student
Overrride
+ showActivities():void
+ study():void
Teacher
Override
+ showActivities():void
+ teach():void

Scenario


IPerson
<<interface>>
+ showActivities():void
implements
Person
+ showActivities():void
//eat, sleep
extends
extends
extends???
Student
Overrride
+ showActivities():void
+ study():void
Teacher
Override
+ showActivities():void
+ teach():void
extends???
extends???
TeachingAssistant
Override
+ showActivities():void
Override
+ study():void
+ teach():void
TeachingAssistant
Override
+ showActivities():void
+ study(): void
+ teach(): void
TeachingAssistant
Override
+ showActivities():void
Override
+ teach(): void
+ study(): void

Problems need to be resolved

1. Dynamically add a behavior at run-time

2. More flexible than inheritance

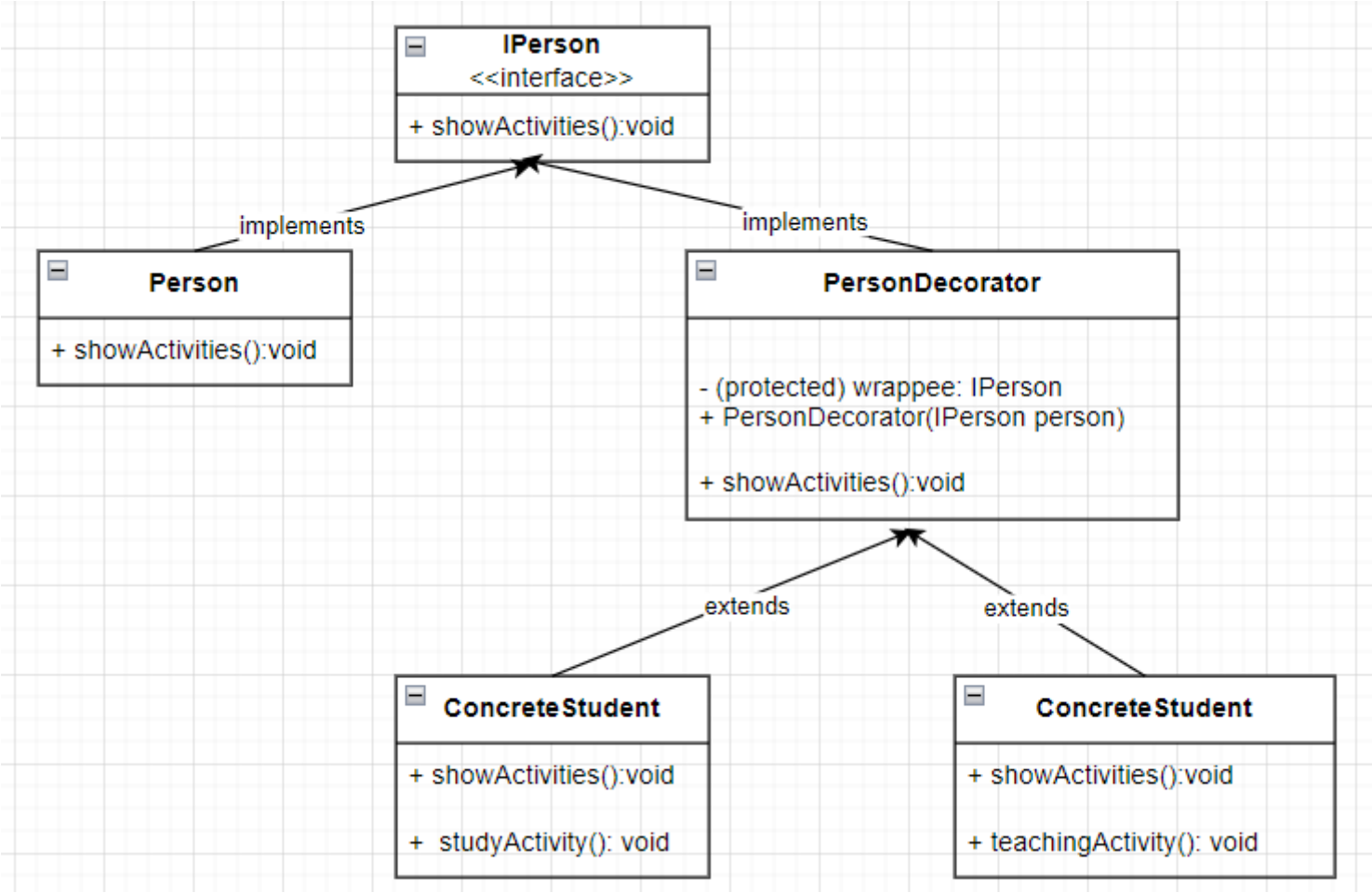
Decorator Design Pattern

A Structural Design Pattern

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

A breakdown of components

1. **Component** (IPerson): It is an interface for objects that can have responsibilities added to them dynamically
2. **ConcreteComponent** (Person): It is a concrete class of component interface and it defines an object to which additional responsibilities can be attached
3. **Decorator** (PersonDecorator): It has a reference to a Component object and defines an interface that conforms to the interface of the component
4. **ConcreteDecorator** (ConcreteStudent and ConcreteTeacher): It is a concrete implementation of Decorator and it adds responsibilities to the component



Implementation of Observer Design Pattern

1. Component

```
public interface IPerson {

    public void showActivities();

}
```

Implementation of Observer Design Pattern

2. ConcreteComponent

```
public class Person implements IPerson {

    @Override
    public void showActivities()
    {
        System.out.print( s:"Eat and sleep\n");
    }

}
```

Implementation of Observer Design Pattern

3. Decorator

```
public class PersonDecorator implements IPerson {

    protected IPerson wrappee;

    public PersonDecorator(IPerson person)
    {
        this.wrappee=person;
    }

    @Override
    public void showActivities()
    {
        wrappee.showActivities();
    }
}
```

Implementation of Observer Design Pattern

4. ConcreteDecorator (ConcreteStudent and ConcreteTeacher)

```
public class ConcreteStudent extends PersonDecorator {
    public ConcreteStudent(IPerson person)
    {
        super(person);
    }

    @Override
    public void showActivities()
    {
        super.showActivities();
        this.studyActivity();
    }

    public void studyActivity()
    {
        System.out.print( s:"Study\n");
    }

}
```

Implementation of Observer Design Pattern

4. ConcreteDecorator (ConcreteStudent and ConcreteTeacher)

```
public class ConcreteTeacher extends PersonDecorator{
    public ConcreteTeacher(IPerson person)
    {
        super(person);
    }

    @Override
    public void showActivities()
    {
        super.showActivities();
        this.teachingActivity();
    }

    public void teachingActivity()
    {
        System.out.print( s:"Teaching\n");
    }
}
```

Implementation of Observer Design Pattern

ClientDemo

```
public class Demo {
    public static void main(String[] agrs)
    {
        //Start an instance of Person object
        IPerson person=new Person();

        //Start new Student object which passes person object as a wrappee
        ConcreteStudent student=new ConcreteStudent(person);
        student.showActivities();

        //Start new Teacher onject which passes person onject as a wrappee
        ConcreteTeacher teacher =new ConcreteTeacher(person);
        teacher.showActivities();

        //Trying to create activities of a TeachingAssistant
        ConcreteTeacher teachingassistant=new ConcreteTeacher( person: student);
        teachingassistant.showActivities();


    }

}
```

Benefits of Decorator Design Pattern

1. This pattern allows you to extend functionality dynamically and statically without altering the structure of existing objects.

2. By using this pattern, you could add a new responsibility to an object dynamically.

3. This pattern is also known as Wrapper.

4. This pattern uses the compositions for object relationships to maintain SOLID principles.

5. This pattern simplifies coding by writing new classes for every new specific functionality rather than changing the existing code of your application