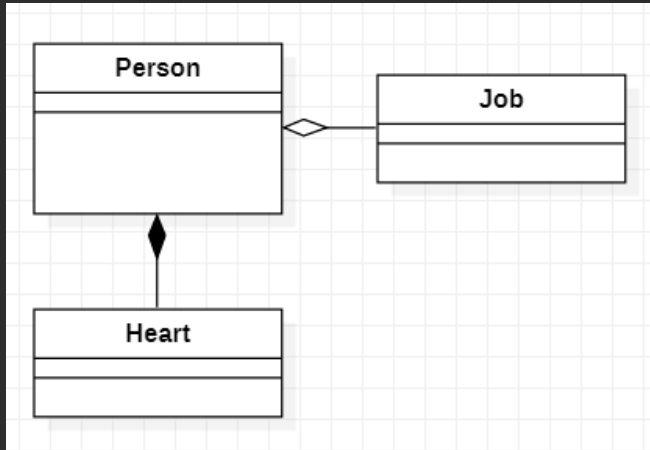


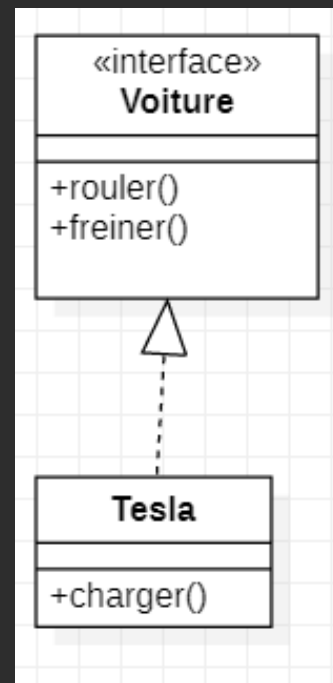
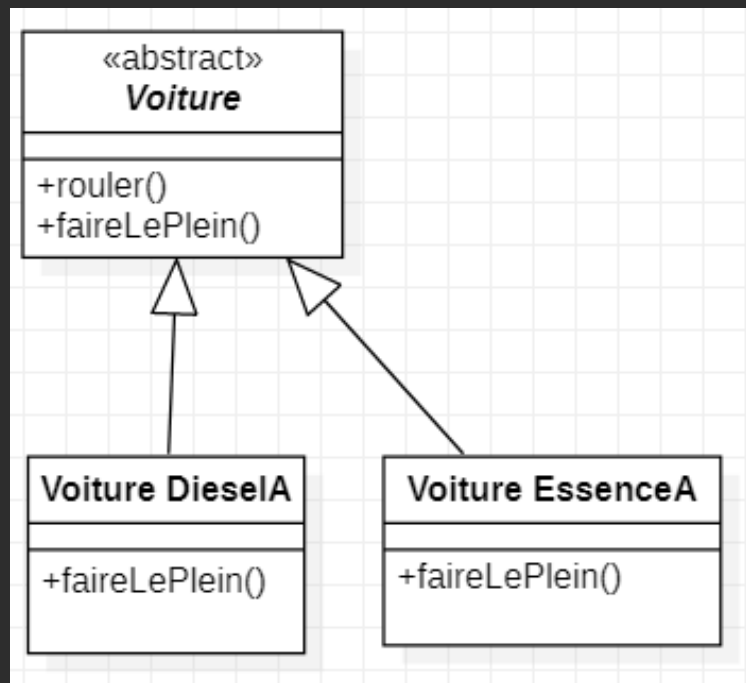
DESIGN PATTERN

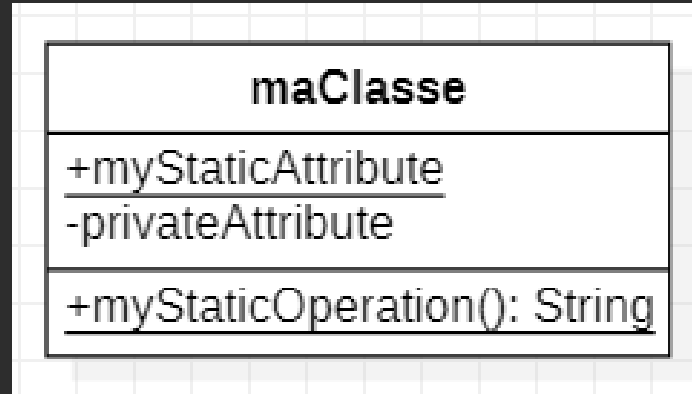
2021



```
public class Person{
    private Heart heart;
    private Job job;

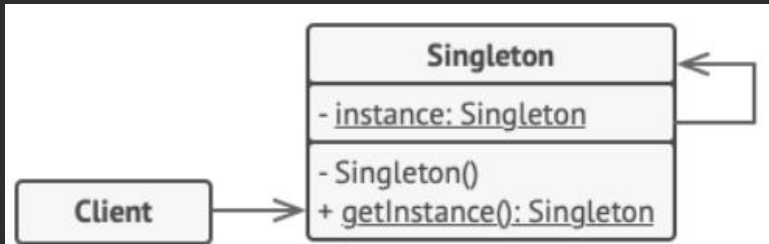
    Person(Job job){
        this.job = job;           // Agrégation
        this.heart = new Heart(); // Composition
    }
}
```





1. Singleton

Intention : garantir qu'une classe n'a qu'une seule instance et fournir un point d'accès global à cette instance



```
singleton = Singleton.getInstance()
```

```
public class Singleton {
    private static Singleton uniqueInstance = null;

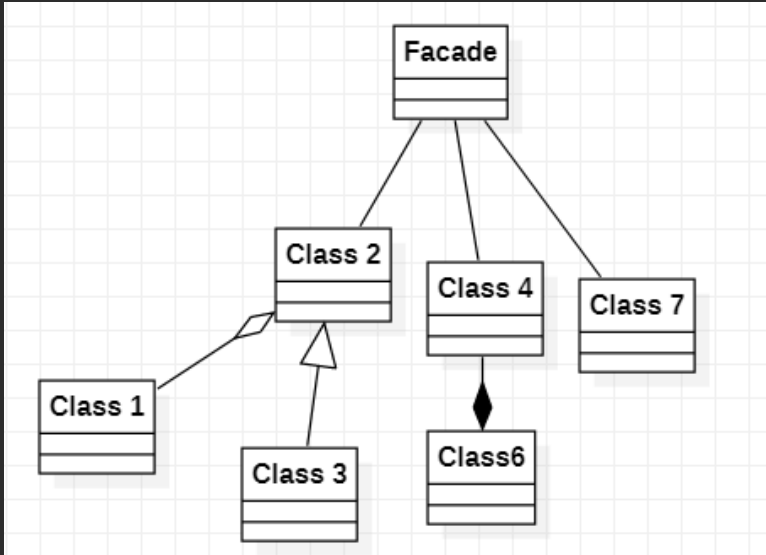
    private Singleton() {
        //TODO
    }

    private static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton()
            return new Singleton();
        }
        return uniqueInstance;
    }
}
```

```
public class Singleton{
    private static Singleton uniqueInstance = new Singleton();
    private Singleton(){
        // TODO
    }
    public static getInstance(){
        return uniqueInstance;
    }
}
```

2. Façade

Intention : fournir une **interface unifiée** et de plus haut niveau à un ensemble d'interfaces d'un sous-système (ie. un groupe de classes), afin de le rendre plus facile à utiliser



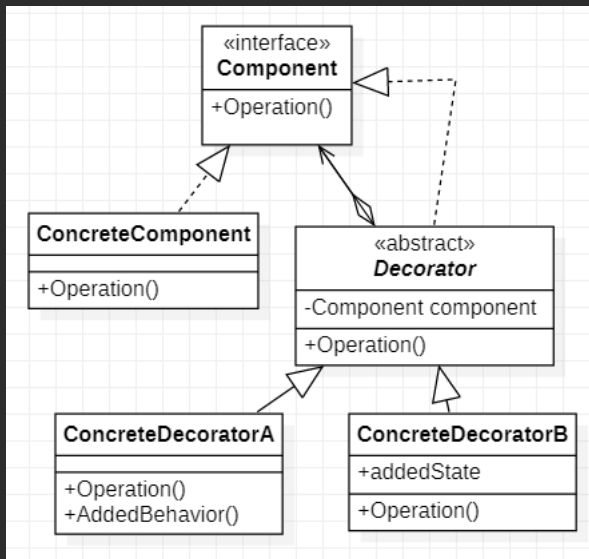
```
public class HomeTheaterFacade {
    Amplifier amp;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    DVD dvd;

    public HomeTheaterFacade(Amplifier amp, Projector
projector, Screen screen, TheaterLights lights, DVD dvd) {
        this.amp = amp;
        //pareil pour le reste
    }

    public void watchMovie(String movie) {
        lights.dim(10);
        screen.down();
        projector.on();
        projector.wideScreenMode();
        amp.on();
        amp.setSurroundSound();
        amp.setVolume(5);
        dvd.on();
        dvd.play(movie);
    }
}
```

3. Décorateur

Intention : Attacher dynamiquement des **responsabilités supplémentaires** à un objet
Fournir une alternative flexible à l'héritage pour étendre des fonctionnalités

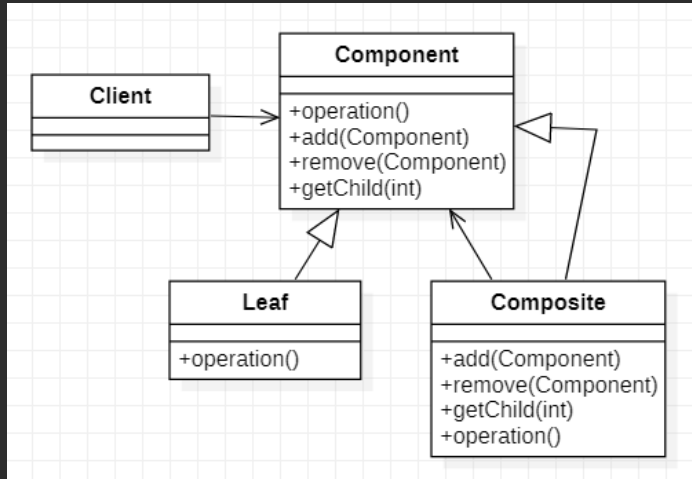


```
Boisson boisson = new Coffee();  
Soy soy = new Soy(boisson);  
Chocolate choco = new Chocolate(soy);  
int n = choco.getPrice();  
//n = new Chocolate(new Soy(new Boisson()));
```

```
//Component  
public interface Beverage {  
    public double getPrice(); //operation()  
}  
//ConcreteComponent extends Component  
public class Coffee implements Beverage {  
    public double getPrice() { //operation()  
        return 1.75;  
    }  
}  
//Decorateur extends Component  
public abstract class CondimentDecorator implements Beverage {  
    protected Beverage decorated; //d'où la flèche  
d'association  
    public CondimentDecorator(Beverage b) { //d'où l'agrégation  
        this.decorated = b;  
    }  
}  
//Concrete Decorateur extends Decorateur extends Beverage  
public class Soy extends CondimentDecorator {  
    public Soy(Beverage b) {  
        super(b);  
    }  
    public double getPrice() { //added behavior  
        return this.decorated.getPrice() + 0.4;  
    }  
}  
public class Chocolate implements CondimentDecorator {  
    public Chocolate(Beverage b) {  
        super(b);  
    }  
    public double getPrice() { //added behavior
```

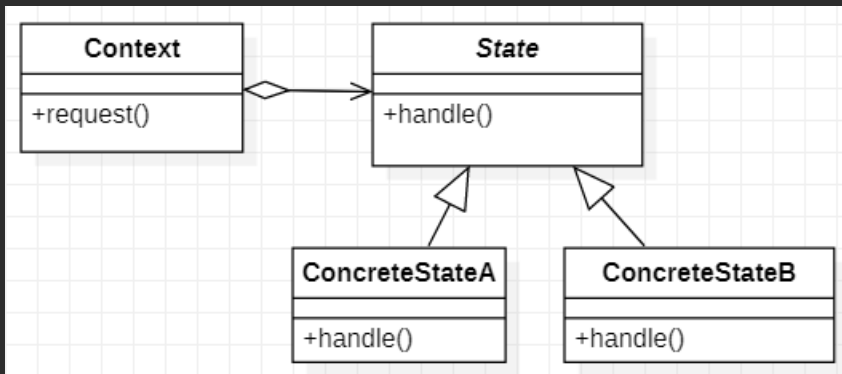
4. Composite

Intention : composer des objets dans des structures arborescentes pour représenter des hiérarchies composants/composés. Permettre aux clients de traiter de la même façon des objets individuels des groupements de ces objets



5. State

Intention : permettre de modifier le comportement d'un objet lorsque son état interne change (tout se passe comme si l'objet changeait de classe)



```
Context context = new Context();
d.fonctionnalite());
```

```
public abstract class State {
    public State handle() {
        return new StateNotOK();
    }
}

public class ConcreteState0 extends State {
    public State handle() {
        // TODO
        return new ConcreteState1();
    }
}

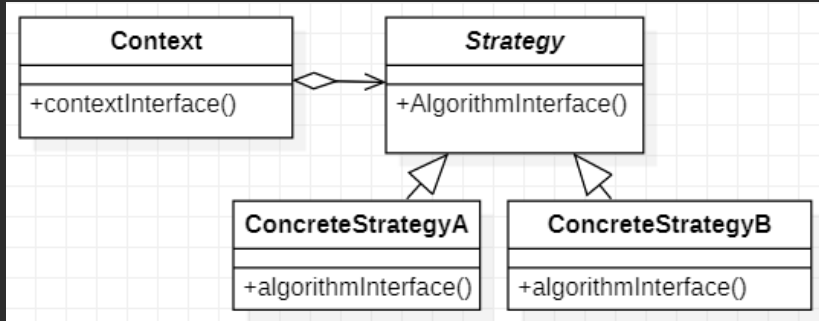
public class Context {
    private State currentState;
    private static Scanner sc;
    public Context() {
        currentState = new ConcreteState0();
    }

    public void fonctionnalite() {
        currentState = currentState.handle();
    }
}
```

6. Strategy

Intention :

- définir une famille d'algorithmes, encapsuler chacun d'eux, et les rendre interchangeables
- utiliser différents algorithmes identiques sur le plan conceptuel en fonction du contexte
- permettre aux algorithmes d'évoluer indépendamment des clients qui les utilisent



```
c = new Context(new ConcreteStrategy());
```

```
public interface Strategy{
    public void fonctionnalite();
}

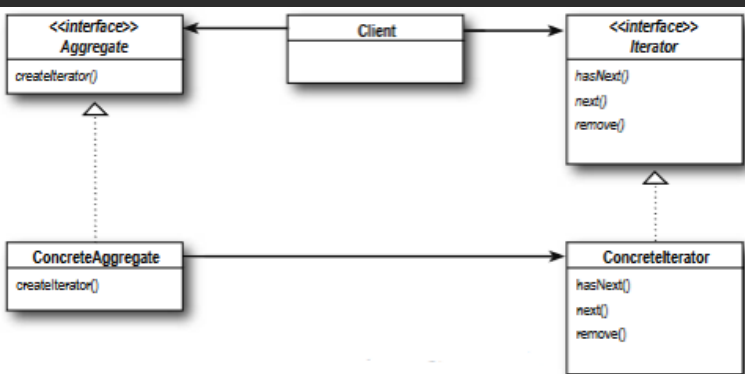
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
}

public class ConcreteStrategy implements Strategy {
    public void fonctionnalite() {
        // TODO implémente le comportement
    }
}
```

7. Iterator

Intention : fournir un moyen de parcourir séquentiellement un agrégat (collection, conteneur) d'éléments sans connaître sa structure interne



```
iterator = new
ConcreteAggregate().createIterator();
while (iterator.hasNext()) {
    MenuItem menuItem =
(MenuItem)iterator.next();
menuItem.operation();
}
```

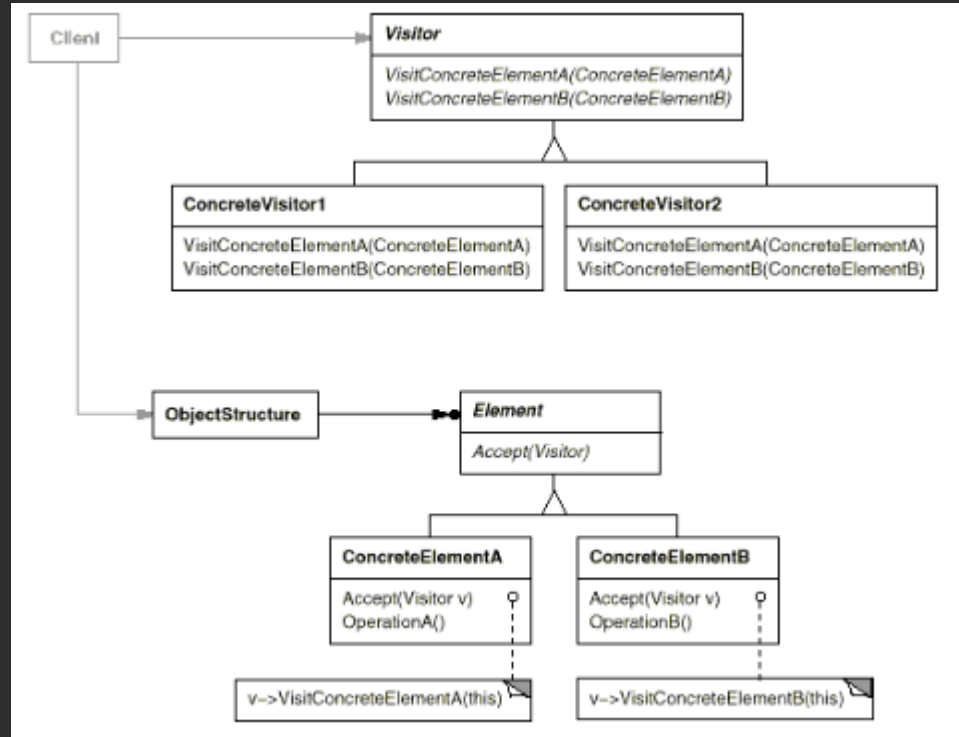
```
public class ConcreteIterator implements Iterator { // Item[] est ici un Array
    Item[] items;
    int position = 0;
    public ConcreteIterator(Item[] items) {
        this.items = items;
    }
    public Object next() {
        Item item = items[position];
        position = position + 1;
        return item;
    }
    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}

public interface Aggregate {
    public Iterator createIterator();
}

public class ConcreteAggregate implements Aggregate {
    Item[] items;
    int numberOfItems = 0;
    static final int MAX_ITEMS = 10;
    public ConcreteAggregate(){
        items = new Item[MAX_ITEMS];
    }
    public Iterator createIterator() {
        return new ConcreteIterator(items);
        // return collection.iterator();
    }
    public operation(){
        // TODO}}
}
```

8. Visitor

Intention : représenter une opération à effectuer sur les éléments d'une structure et permettre de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère

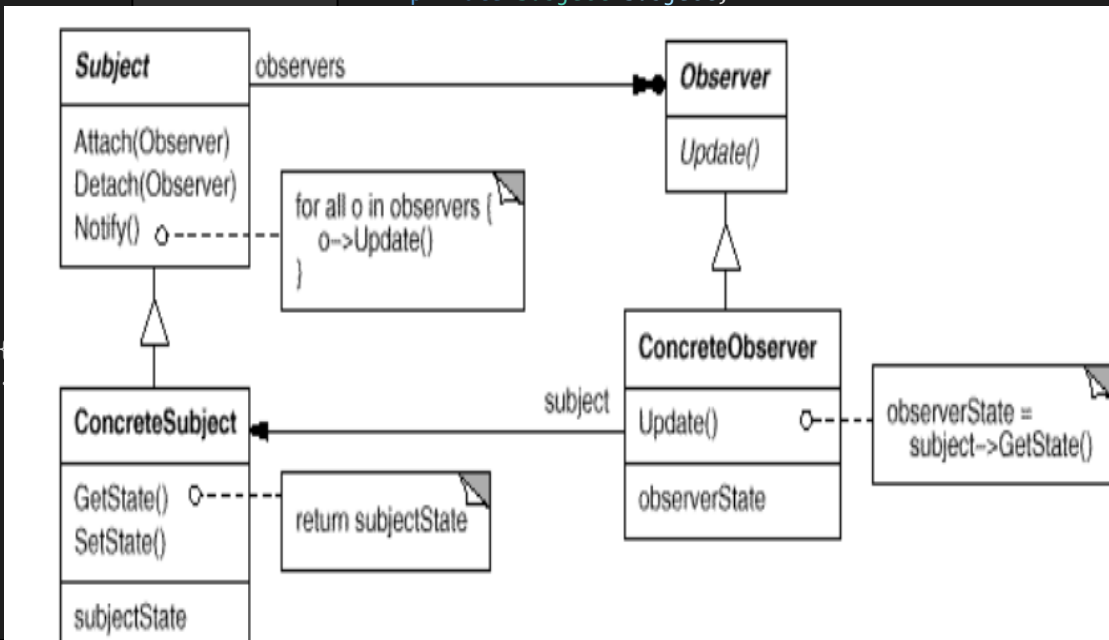


9. Observateur

Intention : définir une interdépendance de type un à plusieurs, de telle sorte que si un objet change d'état, tous **ceux qui en dépendent soient notifiés** et mis à jour automatiquement

```
public class ConcreteSubject implements Subject {
    private ArrayList observers;
    private float parametre;
    public ConcreteSubject() {
        observers = new ArrayList();
    }
    public void registerObserver(Observer o) {
        observers.add(o);
    }
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }
    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(parametre);
        }
    }
    public void setState(float parametre) {
        this.parametre = parametre;
        notifyObservers();
    }
}
```

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
public interface Observer {
    public void update(float parametre);
}
public class ConcreteObserver implements Observer {
    private float parametre;
    private Subject subject;
```

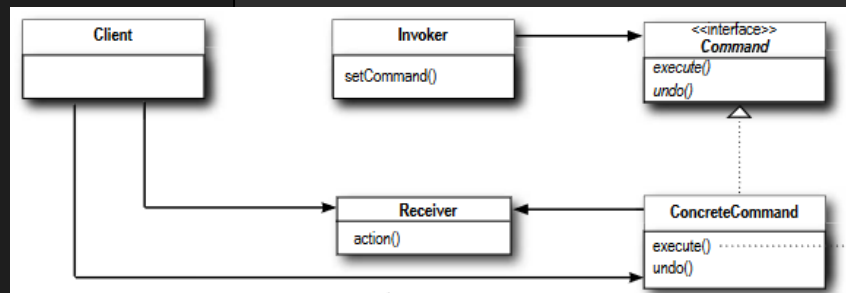


10. Commande

Intention: encapsuler une requête comme un objet, pour permettre de gérer une file d'attente ou un historique de requêtes et d'assurer le traitement des opérations réversibles
Promouvoir l'invocation d'une méthode d'un objet comme un objet à part entière

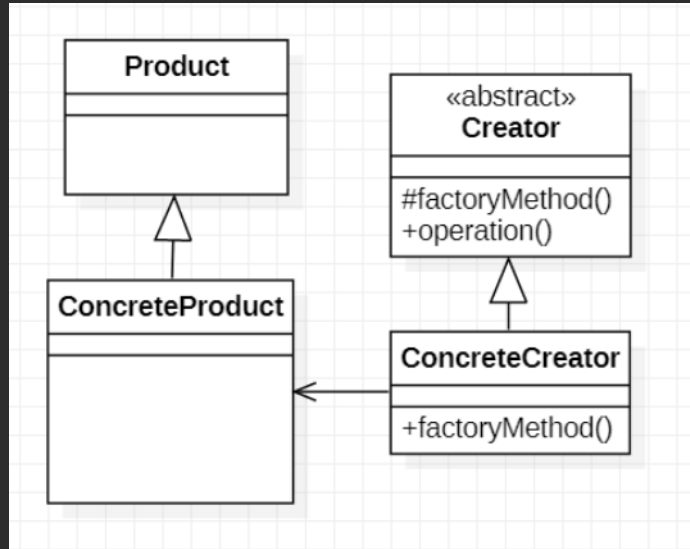
```
public class Invoker {
    Command[] onCommands;
    Command[] offCommands;
    public Invoker() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }
    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }
    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }
    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }
}
```

```
public interface Command {
    void execute();
    void undo();
}
public class Receiver {
    // TODO
    public void Action() {
        // TODO
    }
}
public class ConcreteCommand implements Command {
    Receiver receiver;
    public ConcreteCommand(Receiver receiver) {
        this.receiver = receiver;
    }
    public void execute() {
        receiver.action();
    }
}
```



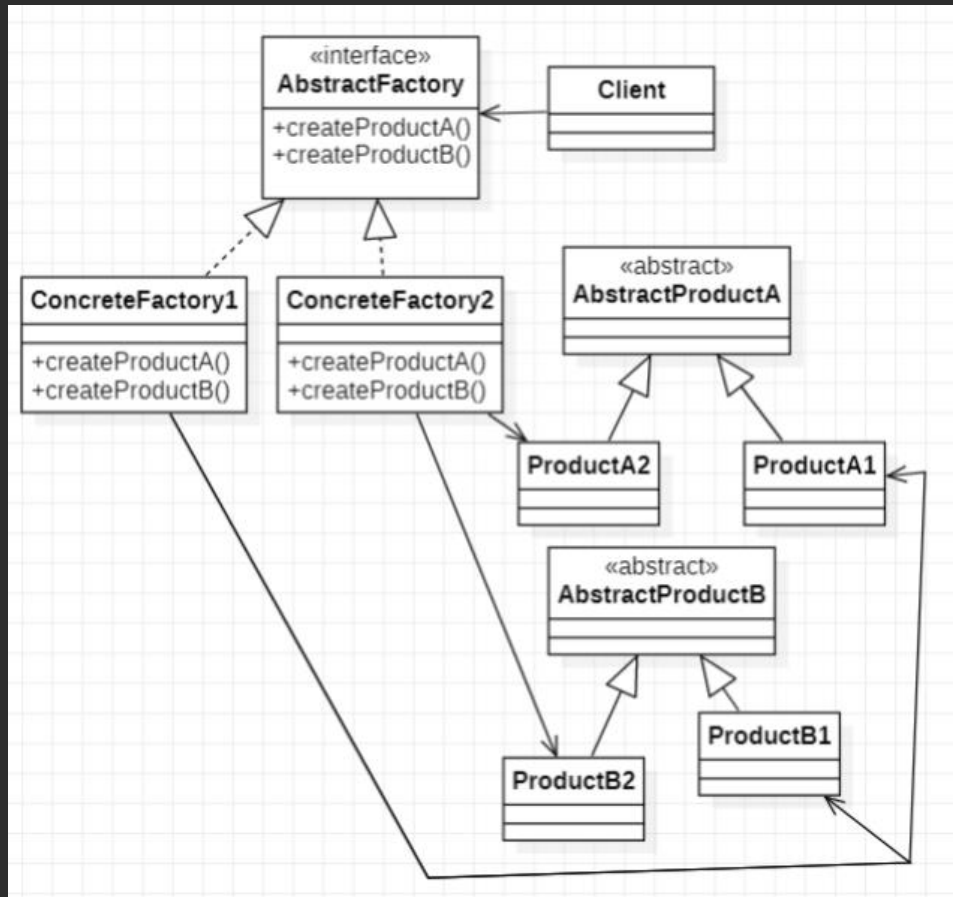
11. Fabrique

Intention: définir une interface pour la création d'un objet, en laissant à ses sous-classes le soin de choisir la classe concrète de l'objet à instancier



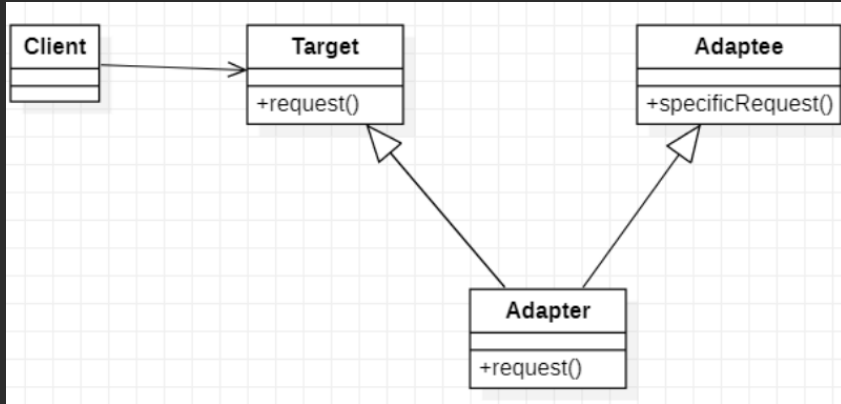
12. Fabrique abstraite

Intention: fournir une interface pour la création de familles d'objets apparenté ou interdépendants, sans qu'il soit nécessaire de spécifier leur classe concrète



13. Adaptateur

Intention: faire correspondre à une interface donnée un objet existant qu'on ne contrôle pas



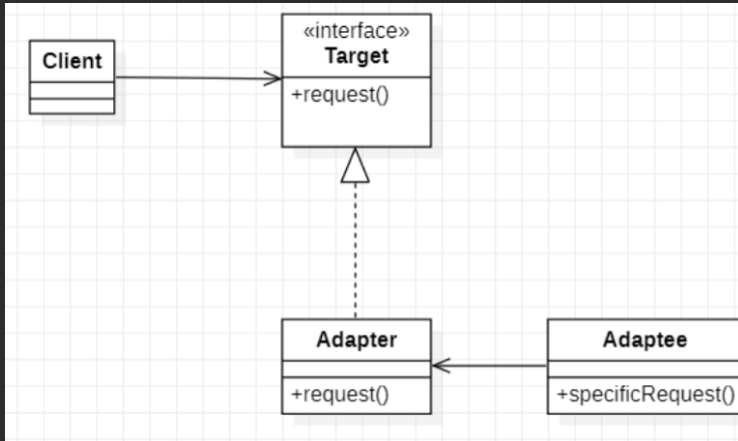
```
public interface Target {
    public void request();
}

public class Adaptee {
    public void specificRequest() {
        // TODO
    }
}

public class Adapter implements Target {
    Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

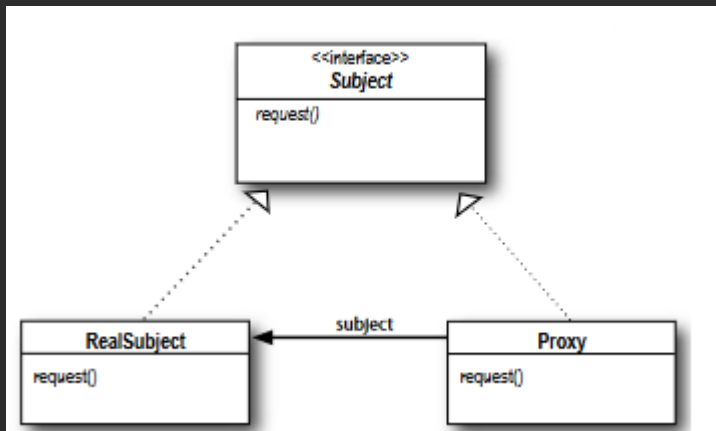
    public void request() {
        // TODO
        adaptee.specificRequest();
    }
}
```



```
Adaptee adaptee = new Adaptee();
Target adapteeAdapter = new
    Adapter(adaptee);
adapteeAdapter.request();
```

14. Procuration

Intention: fournir un **substitut** (le proxy) à un autre objet afin d'en contrôler les accès (ie. les opérations qui lui sont appliquées)



```
Proxy proxy = new Proxy();
proxy.service();
```

```
public interface Subject {
    void service();
}
public class RealSubject implements
Subject {
    public void service() {
        // TODO
    }
}
public class Proxy implements Subject {
    private Subject subject;
    // TODO

    public Proxy() {
        this.subject = new RealSubject();
        // TODO
    }
    public void service() {
        this.subject.service();
        // TODO
    }
}
```

Conclusion

> ~ / ATILLA



atilla.org



t.me/ATILLAnnonce



Ce document sera sur le Drive D'Atilla