



UNIVERSITÉ
DE MONTPELLIER



Design Patterns

Conception et Programmation Objet Avancées
M3105 (cours 4)

Nadjib Lazaar (nadjib.lazaar@umontpellier.fr)

Behavioral Patterns

Les Patrons de comportement

Behavioral Patterns

- **Command Pattern / Le Patron Commande**
- State Pattern / Le Patron Etat
- Iterator Pattern / Le Patron Itérateur
- Observer Pattern / Le Patron Observateur
- Strategy Pattern / Le Patron Stratégie
- Visitor Pattern / Le Patron Visiteur
- MVC Pattern / Le Patron MVC

Behavioral Patterns

Command Pattern / Le Patron Commande

- **Command** est un pattern de comportement qui transforme « une demande » en un objet autonome contenant toutes les informations sur « la demande ».
- Principes SOLID :
 - **Single Responsibility** : La commande s'occupe d'une tâche particulière
 - **Open Close** : Une commande est ajoutée sans toucher à l'existant
 - **Liskov Substitution** : Les commandes sont interchangeables
 - **Dependency inversion** : L'invocateur ne manipule que des interfaces

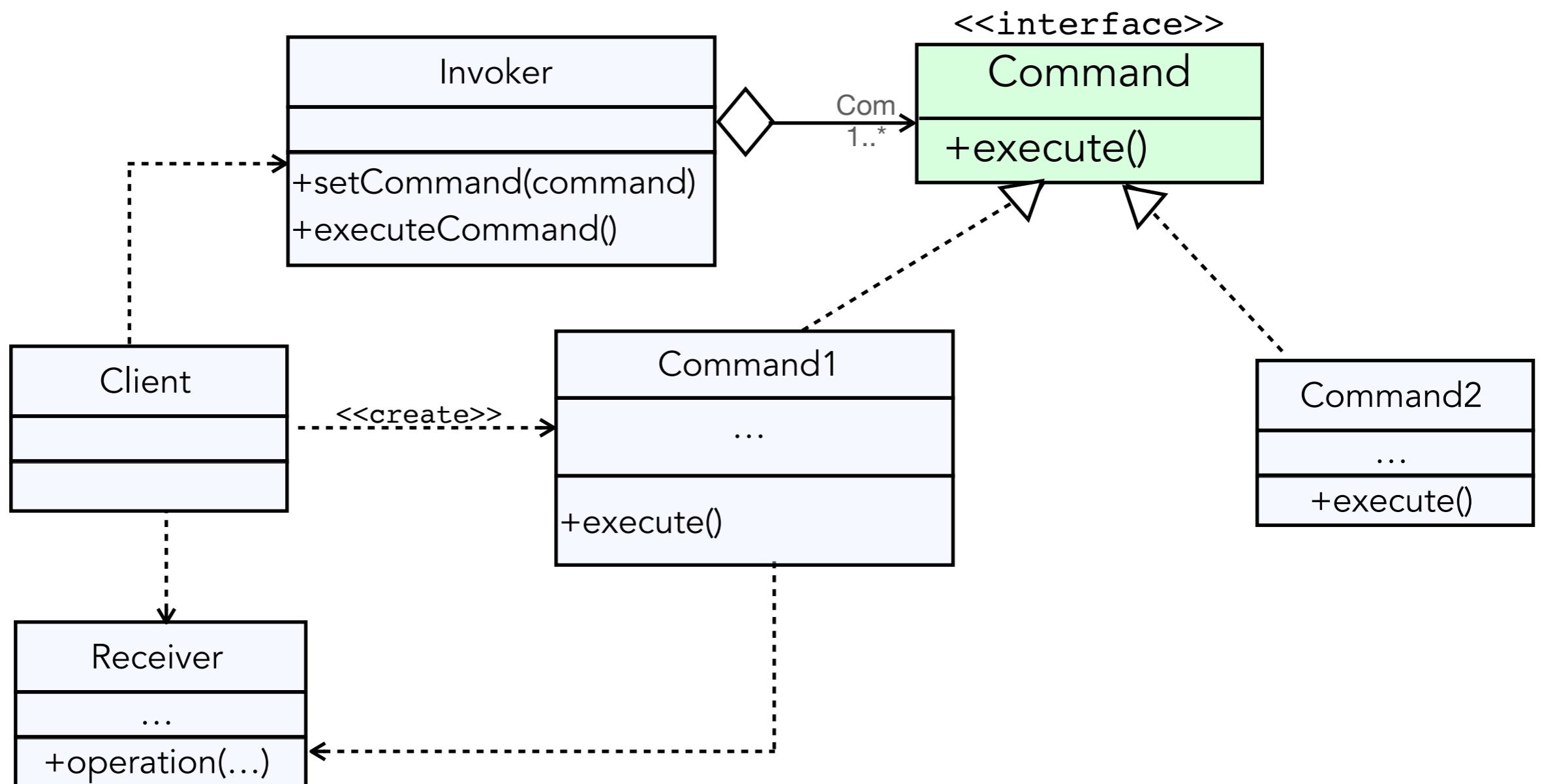
Command Pattern

Example

```
/** Permet de commander la maison connectée */
public class Telecommande {
    private Lampe lampeSalon, lampeChambre;
    private Alarme alarme;
    private Portail portail;
    private Sono sono;
    /** Active la télécommande
     * @param commande la commande vocale à exécuter */
    public void active(String commande) {
        switch(commande) { // Autorisé depuis Java 7
            case "allumeSalon" : lampeSalon.allume(); break;
            case "éteintSalon" : lampeSalon.eteint(); break;
            case "allumeChambre" : lampeChambre.allume(); break;
            case "éteintChambre" : lampeChambre.eteint(); break;
            case "allumeAlarme" : alarme.allume(); break;
            case "éteintAlarme" : alarme.eteint(); break;
            case "ouvrePortail" : portail.ouvre(); break;
            case "fermePortail" : portail.ferme(); break;
            case "plusFort" :
                if (sono.estEteinte()) sono.allume();
                sono.volumePlus(); break;
            case "moinsFort" :
                sono.volumeMoins();
                if (sono.getVolume() == 0) sono.eteint();
                break; } // accolade du switch
    } // executer(commande)
} // public class Telecommande
```

Command Pattern

UML



Command Pattern

Example

```
/** Permet de commander la maison connectée */
public class Telecommande{
    private Map<String,Commande> commandes;
    /** Associe un nom à une commande.
     * @param nom le mot déclencheur
     * @param act la commande à exécuter */
    public void setCommande(String nom, Commande act) {
        commandes.put(nom,act);
    }
    /** Active la télécommande
     * @param commande la commande vocale à exécuter */
    public void executeCommand(String commande) {
        commandes.get(commande).executer();
    } // executer(commande)
} // public class Telecommande
/** Représente une commande */
public interface Commande {
    /** Active la commande */
    public void executer();
} // public interface Commande
```

Command Pattern

Example

```
/** Commande pour monter le volume */
public class VolumeFort implements Commande{
    private Sono sono = Sono.getInstance();
    public executer() {
        if (sono.estEteinte()) sono.allumer();
        sono.volumePlus();
    }
} // public class VolumeFort
/** Commande pour allumer un appareil */
public class Allume implements Commande{
    private Allumable appareil;
    /** Construit une commande d'allumage */
    public Allume(Allumable app) {
        this.appareil = app;
    }
    @Override
    public executer() {
        appareil.allumer();
    }
} // public class Allume
```

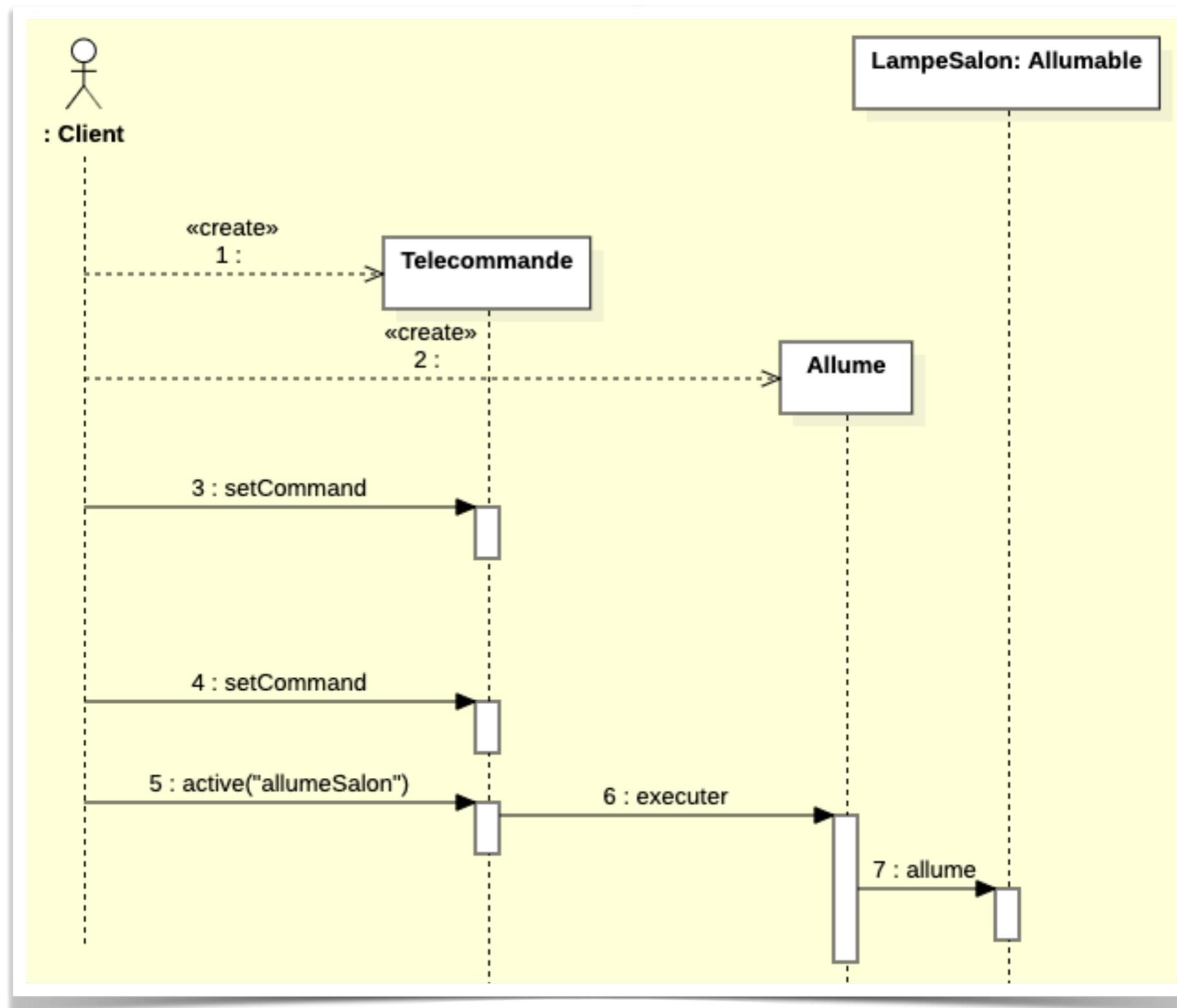
Command Pattern

Example

```
/** Représente un appareil allumable */
public interface Allumable {
    /** Allume l'objet */
    public void allumer() ;
} // public interface Allumable
/** Un extrait de Client */
public class Client {
    Allumable lampeSalon, lampeChambre, alarme;
    Telecommande tele;
    private void run() {
        tele = new Telecommande();
        tele.setCommande("allumeSalon",
                         new Allume(lampeSalon));
        tele.setCommande("allumeChambre",
                         new Allume(lampeChambre));
        tele.setCommande("allumeAlarme",
                         new Allume(alarme));
        tele.setCommande("volumePlus",
                         new VolumeFort());
        // ...
        tele.executer("allumeChambre");
    }
} // public class Client
```

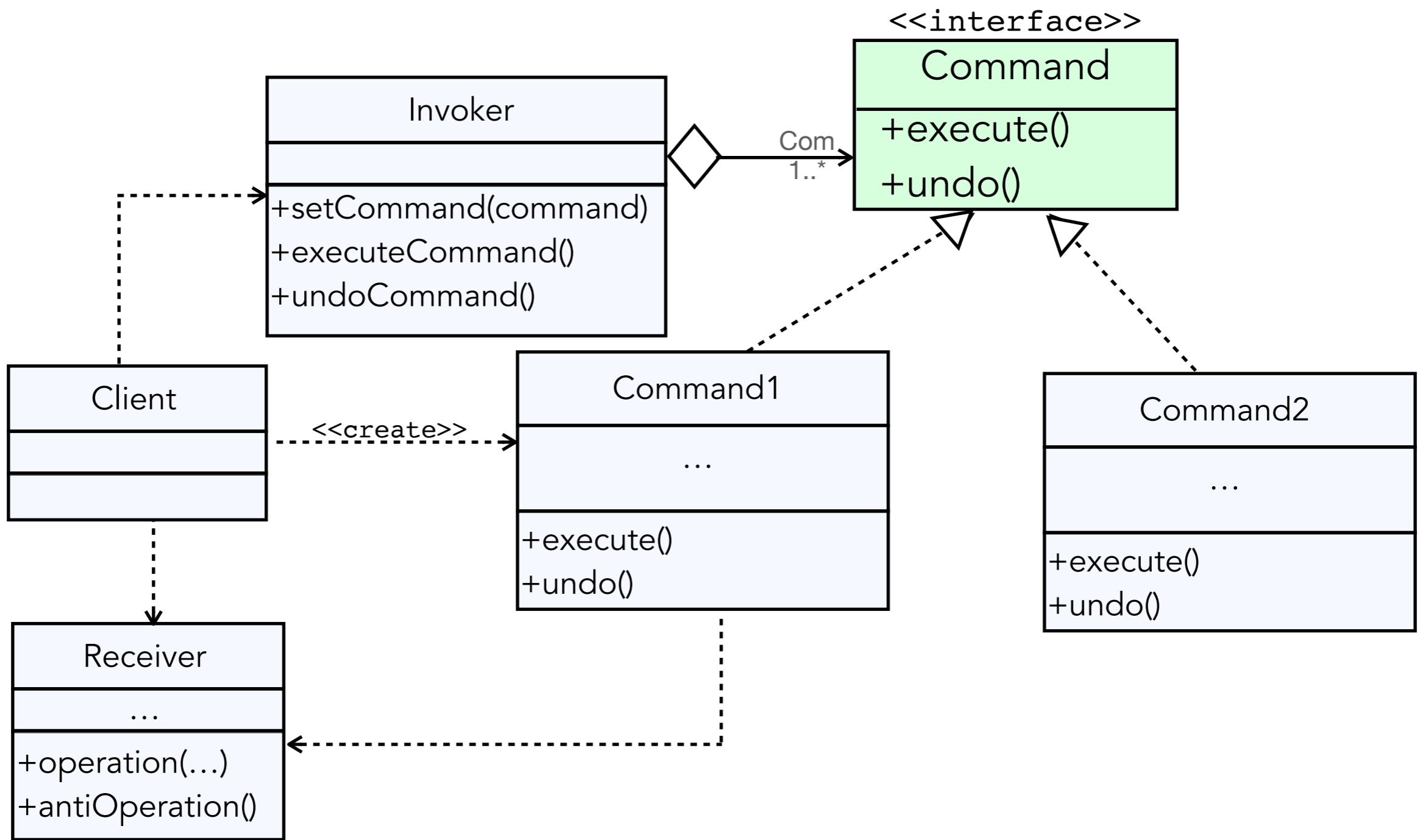
Command Pattern

Sequence Diagram



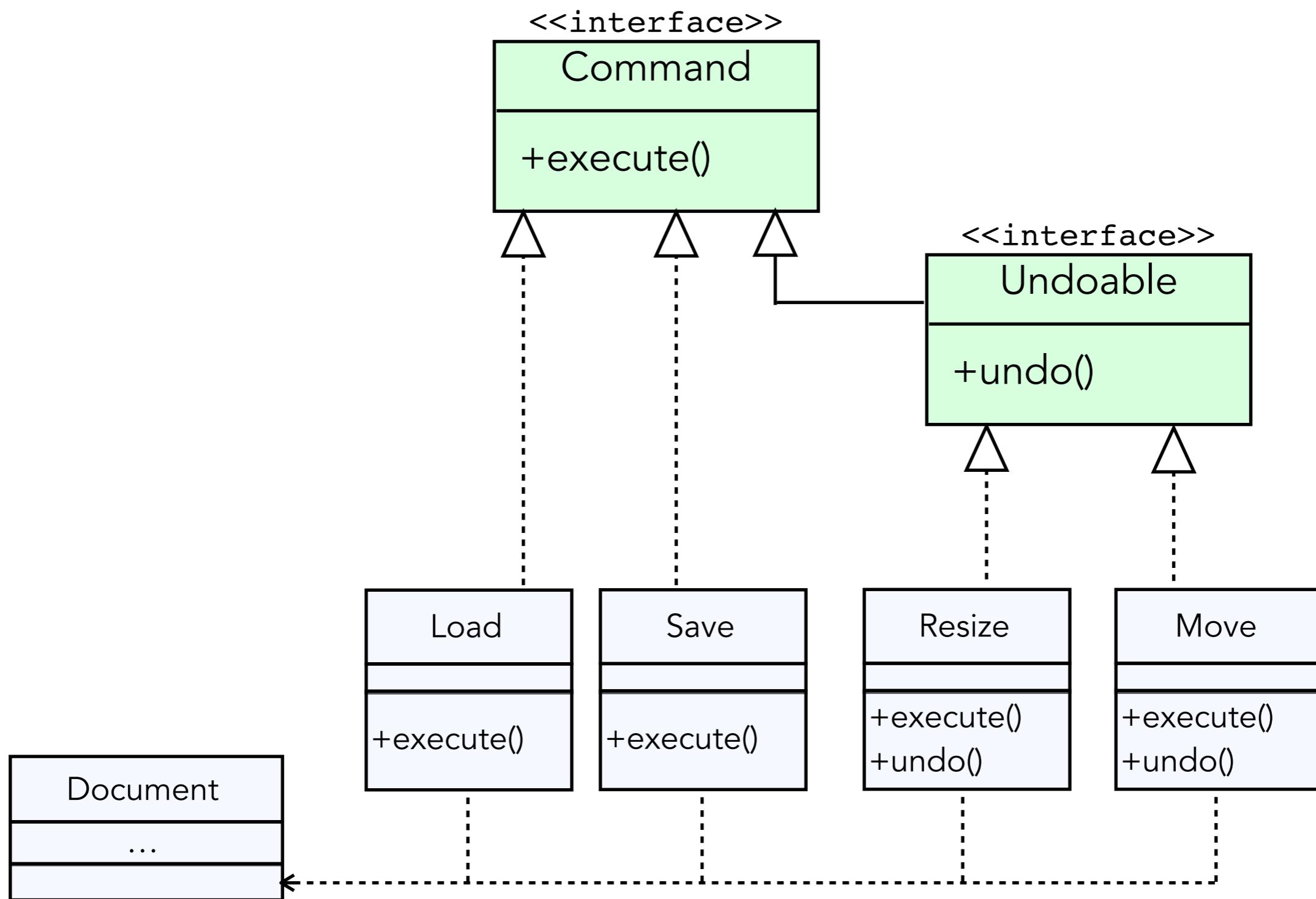
Command Pattern

UML (commandes annulables)



Command Pattern

UML (commandes annulables - exemple)



Behavioral Patterns

- Command Pattern / **Le Patron Commande**
- **State Pattern / Le Patron Etat**
- Iterator Pattern / **Le Patron Itérateur**
- Observer Pattern / **Le Patron Observateur**
- Strategy Pattern / **Le Patron Stratégie**
- Visitor Pattern / **Le Patron Visiteur**
- MVC Pattern / **Le Patron MVC**

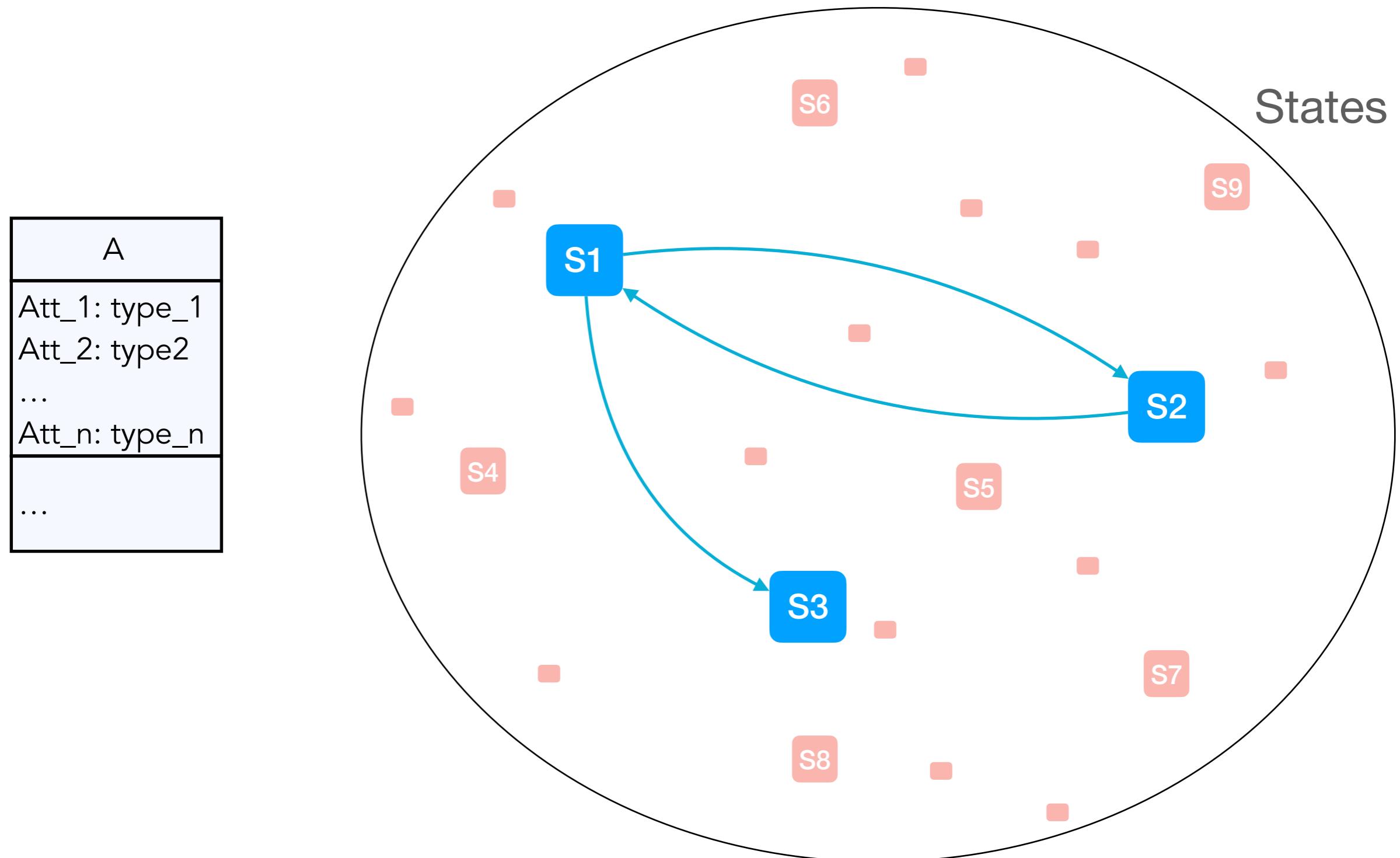
Behavioral Patterns

State Pattern / Le Patron Etat

- **State** est un pattern comportemental qui permet à un objet de modifier son comportement lorsque son état interne change.
- Principes SOLID :
 - **Open Close** : Une commande est ajoutée sans toucher à l'existant

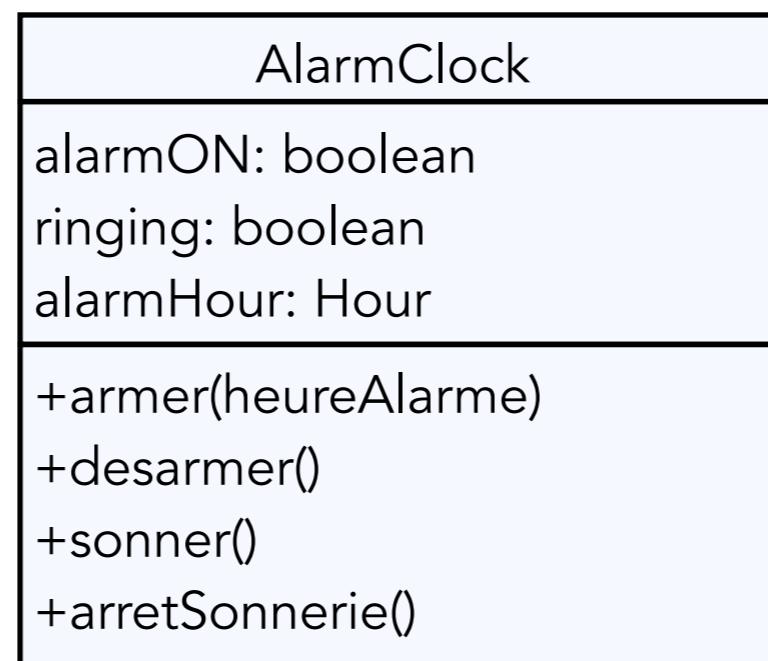
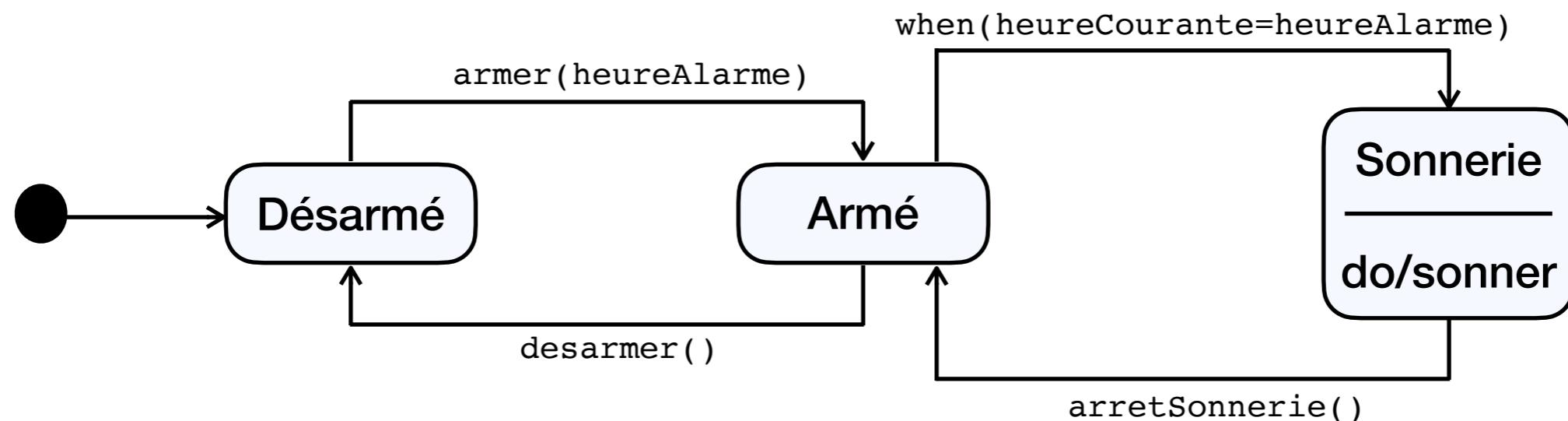
State Pattern

Motivation



State Pattern

Réveille-Matin (Exemple)



State Pattern

Réveille-Matin (Exemple)



```
public class AlarmClock implements Runnable {

    boolean alarmON;
    boolean ringing;
    Hour alarmHour;

    public AlarmClock() {
        alarmON = false;
        ringing = false;
        alarmHour = new Hour("::");
    }

    public void armer(Hour heureAlarme) {
        if (!alarmON && !ringing) {
            alarmHour = heureAlarme;
            alarmON = true;
        }
    }

    public void desarmer() {
        if (alarmON && !ringing)
            this.alarmON = false;
    }

    private void Sonner() {
        java.awt.Toolkit.getDefaultToolkit().beep();
    }

    public void arretSonnerie() {
        if (alarmON && ringing)
            ringing = false;
    }
}
```

```
@Override
public void run() {

    while (true) {
        if (alarmON && timeToRing())
            try {
                this.Sonner();
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
    }
}

private boolean timeToRing() {
    if (alarmHour == currentTime())
        ringing = true;
    return ringing;
}
```

State Pattern

AlarmClock.java (v1)



```
public class AlarmClock implements Runnable {

    boolean alarmON;
    boolean ringing;
    Hour alarmHour;

    public AlarmClock() {
        alarmON = false;
        ringing = false;
        alarmHour = new Hour("::");
    }

    public void armer(Hour heureAlarme) {
        if (!alarmON && !ringing) {
            alarmHour = heureAlarme;
            alarmON = true;
        }
    }

    public void desarmer() {
        if (alarmON && !ringing)
            this.alarmON = false;
    }

    private void Sonner() {
        java.awt.Toolkit.getDefaultToolkit().beep();
    }

    public void arretSonnerie() {
        if (alarmON && ringing)
            ringing = false;
    }
}
```

```
@Override
public void run() {

    while (true) {
        if (alarmON && timeToRing())
            try {
                this.Sonner();
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
    }
}

private boolean timeToRing() {
    if (alarmHour == currentTime())
        ringing = true;
    return ringing;
}
```

State Pattern

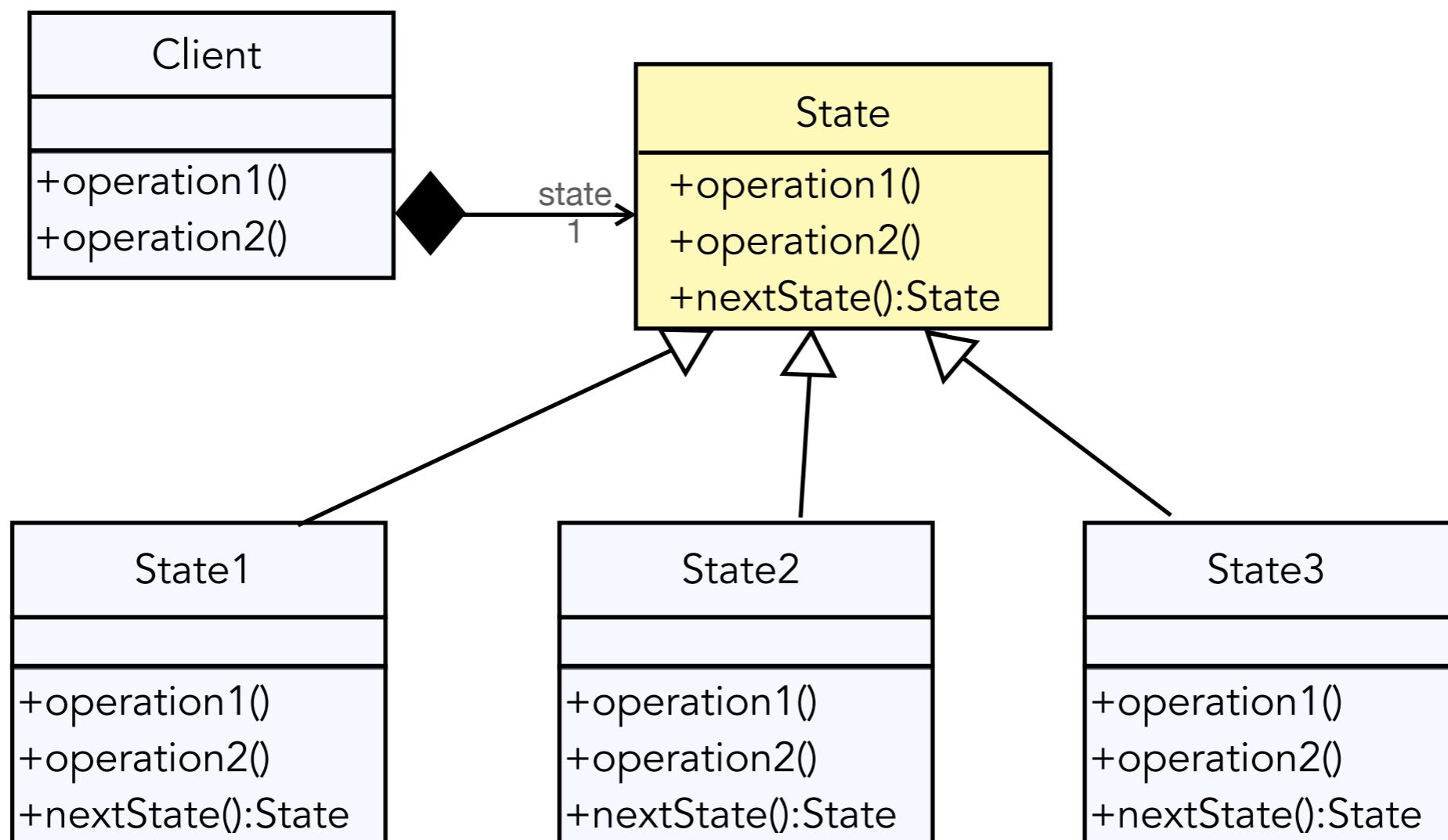
Réveille-Matin (Exemple)



- **CONS**
 - Code lourd et redondant
 - Des classes résultantes qui peuvent exploser en LOC (Lines Of Code)
 - Des structures conditionnelles imbriquées difficile à analyser et à comprendre
 - => utilisation des switch peut réduire la complexité =
 - => polymorphisme est la solution !

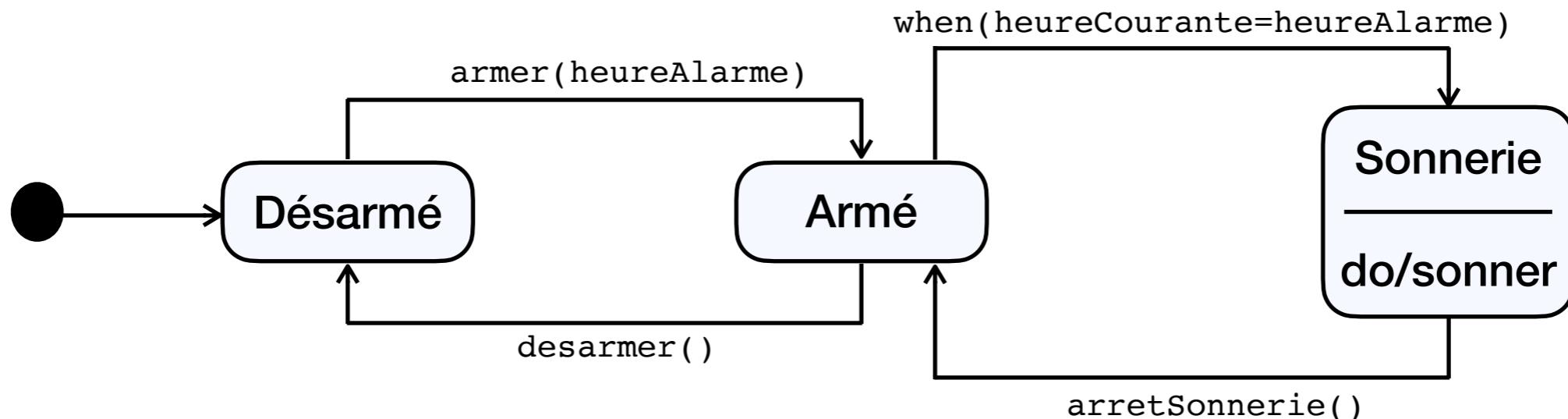
State Pattern

Réveille-Matin (Exemple)



State Pattern

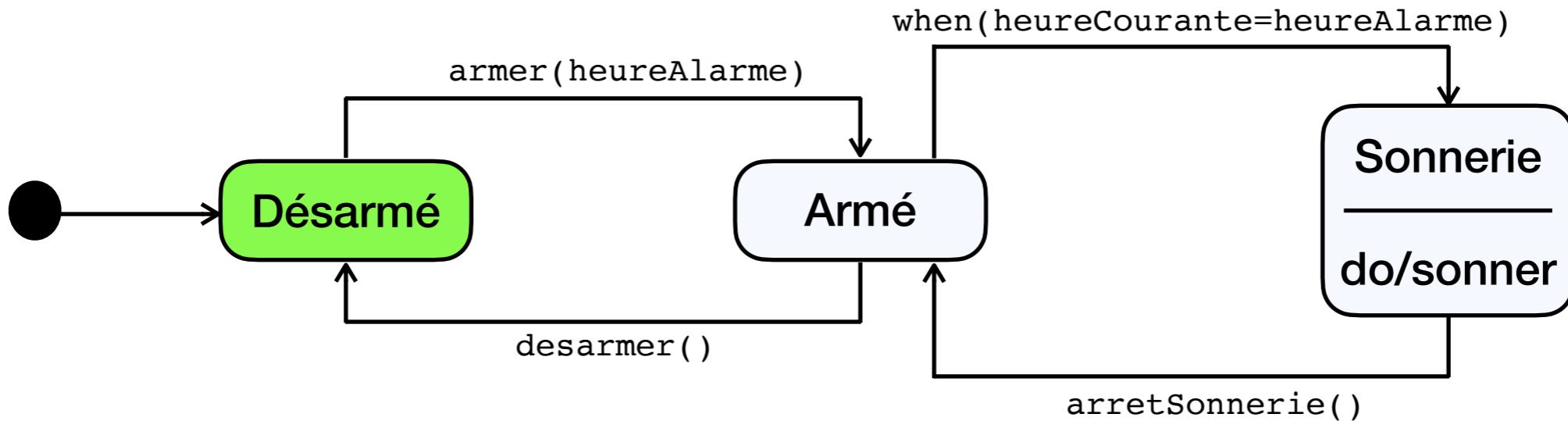
AlarmState



```
public abstract class AlarmState {  
  
    AlarmClock alarm;  
  
    AlarmState(AlarmClock ac) {  
        alarm = ac;  
    }  
  
    public abstract void armer(String heureAlarme);  
  
    public abstract void desarmer();  
  
    public abstract void Sonner() throws InterruptedException;  
  
    public abstract void arretSonnerie();  
}
```

State Pattern

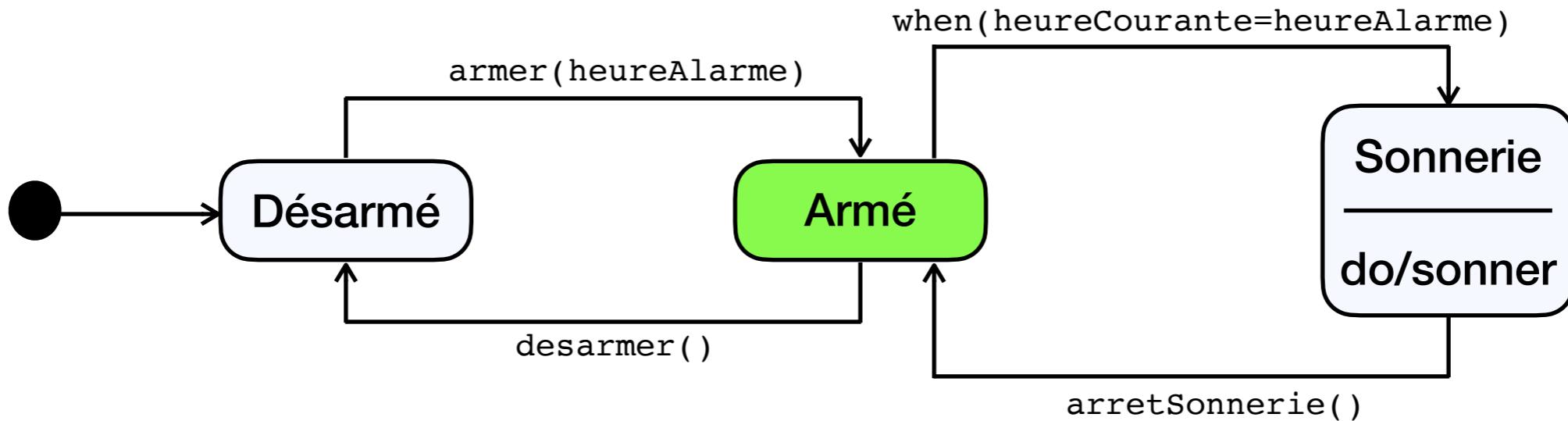
Desarme State



```
public class Desarme extends AlarmState {  
  
    Desarme(AlarmClock ac) {  
        super(ac);  
    }  
  
    @Override  
    public void armer(String heureAlarme) {  
        alarm.setHour(new Hour(heureAlarme));  
        alarm.changeState(new Arme(alarm));  
    }  
  
    @Override  
    public void desarmer() { System.out.println("disarmed state!"); }  
  
    @Override  
    public void Sonner() { System.out.println("disarmed state!"); }  
  
    @Override  
    public void arretSonnerie() { System.out.println("disarmed state!"); }  
}
```

State Pattern

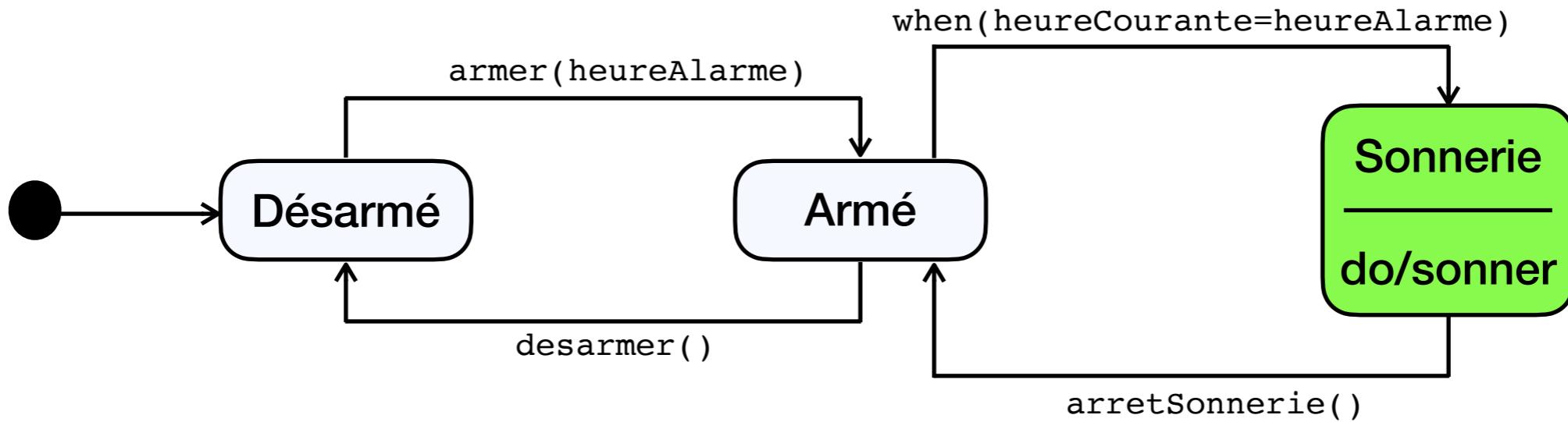
Arme State



```
public class Arme extends AlarmState {  
  
    Arme(AlarmClock ac) {  
        super(ac);  
    }  
  
    @Override  
    public void armer(String heureAlarme) { System.out.println("armed state!");}  
  
    @Override  
    public void desarmer() {  
        alarm.changeState(new Desarme(alarm));  
    }  
  
    @Override  
    public void Sonner() { System.out.println("armed state!"); }  
  
    @Override  
    public void arretSonnerie() { System.out.println("armed state!");}  
}
```

State Pattern

Sonnerie State



```
public class Sonnerie extends AlarmState {  
  
    Sonnerie(AlarmClock ac) {  
        super(ac);  
    }  
  
    @Override  
    public void armer(String heureAlarme) { System.out.println("sonnerie state!"); }  
  
    @Override  
    public void desarmer() { System.out.println("sonnerie state!"); }  
  
    @Override  
    public void Sonner() throws InterruptedException {  
        while (true) {  
            java.awt.Toolkit.getDefaultToolkit().beep();  
            Thread.sleep(500);  
        }  
    }  
  
    @Override  
    public void arretSonnerie() { alarm.changeState(new Arme(alarm));}  
}
```

State Pattern

AlarmClock.java (v2)



```
public class AlarmClock implements Runnable {

    private AlarmState state;
    Hour alarmHour;

    public AlarmClock() {
        alarmHour = new Hour("--:--");
        state = new Desarme(this);
    }

    @Override
    public void run() {
        while (true)
            if (timeToRing())
                try {
                    state.Sonner();
                } catch (InterruptedException e) { e.printStackTrace(); }
    }

    private boolean timeToRing() {
        if (alarmHour == currentTime()) return true;
        return false;
    }

    public void setHour(Hour heureAlarme) { this.alarmHour = heureAlarme; }

    public void changeState(AlarmState state) { this.state = state; }

    public AlarmState getState() { return state; }

    public static void main(String[] args) {

        AlarmClock testAlarm = new AlarmClock();
        testAlarm.getState().armer("07:00");
        testAlarm.run();
    }
}
```

Behavioral Patterns

- Command Pattern / **Le Patron Commande**
- State Pattern / **Le Patron Etat**
- **Iterator Pattern / Le Patron Itérateur**
- Observer Pattern / **Le Patron Observateur**
- Strategy Pattern / **Le Patron Stratégie**
- Visitor Pattern / **Le Patron Visiteur**
- MVC Pattern / **Le Patron MVC**

Behavioral Patterns

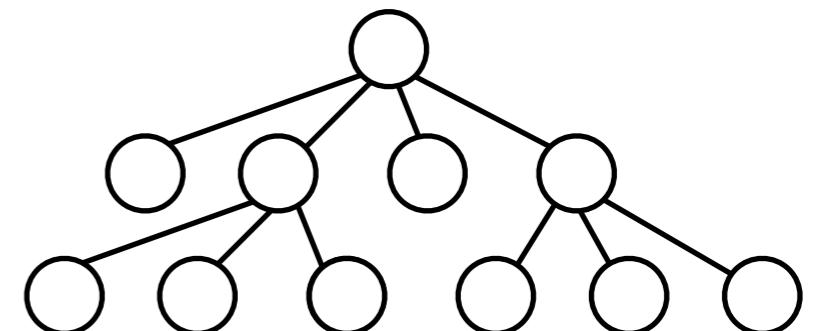
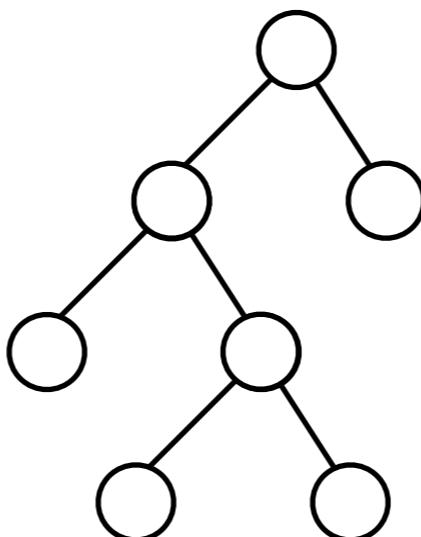
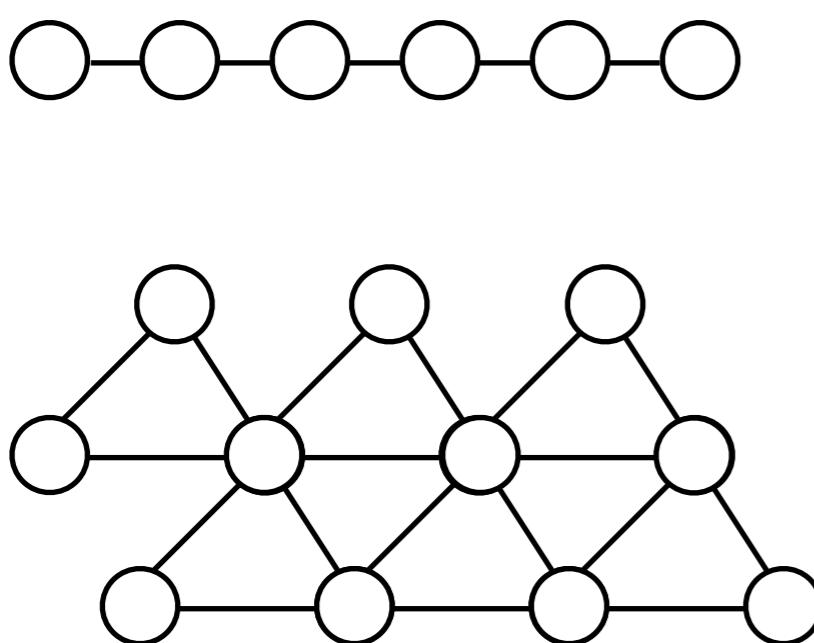
Iterator Pattern / Le Patron Itérateur

- **Iterator** est un modèle de conception comportementale qui permet de parcourir les éléments d'une collection sans exposer la représentation qui est derrière (liste, pile, arbre, etc.).
- Principes SOLID :
 - **Single Responsibility**: Unique responsabilité qui est le parcours
 - **Open/Closed**: Ouvert aux extensions / fermer aux modifications

Iterator Pattern

Why iterators

- Les collections sont l'un des types de données les plus utilisés en programmation. Ceci dit, une collection n'est qu'un conteneur pour un groupe d'objets.



Iterator Pattern

Exemple

```
public class Entreprise {  
  
    private Employe[] employes;  
  
    /** Affiche chaque employé. */  
    public void affiche() {  
        for (int i = 0; i < employes.length; ++i)  
            System.out.println(employes[i].toString());  
    }  
}
```

Iterator Pattern

Exemple

```
public class Entreprise {  
  
    private Employe[] employes;  
  
    /** Affiche chaque employé. */  
    public void affiche() {  
        for (int i = 0; i < employes.length; ++i)  
            System.out.println(employes[i].toString());  
    }  
}
```



Remplacer le tableau par une liste ⇒ révision des boucles

Iterator Pattern

Exemple

```
public class Entreprise {  
    private Employe[] employes;  
  
    private Employe getEmploye(int idx) {  
        return employes[idx];  
    }  
  
    private int nbEmployes() {  
        return employes.length;  
    }  
  
    public void affiche() {  
        for (int i = 0; i < nbEmployes(); ++i)  
            System.out.println(getEmploye(i).toString());  
    }  
}
```



Recourir à des accesseurs/mutateurs

Iterator Pattern

Exemple

```
public class Entreprise {  
    private Employe[] employes;  
  
    private Employe getEmploye(int idx) {  
        return employes[idx];  
    }  
  
    private int nbEmployes() {  
        return employes.length;  
    }  
  
    public void affiche() {  
        for (int i = 0; i < nbEmployes(); ++i)  
            System.out.println(getEmploye(i).toString());  
    }  
}
```



Recourir à des accesseurs/mutateurs



Inefficace pour plusieurs type de structures ()

Iterator Pattern

Exemple

```
public class Entreprise {  
    private Employe[] employes;  
  
    private Employe getEmploye(int idx) {  
        return employes[idx];  
    }  
  
    private int nbEmployes() {  
        return employes.length;  
    }  
  
    public void affiche() {  
        for (int i = 0; i < nbEmployes(); ++i)  
            System.out.println(getEmploye(i).toString());  
    }  
}
```



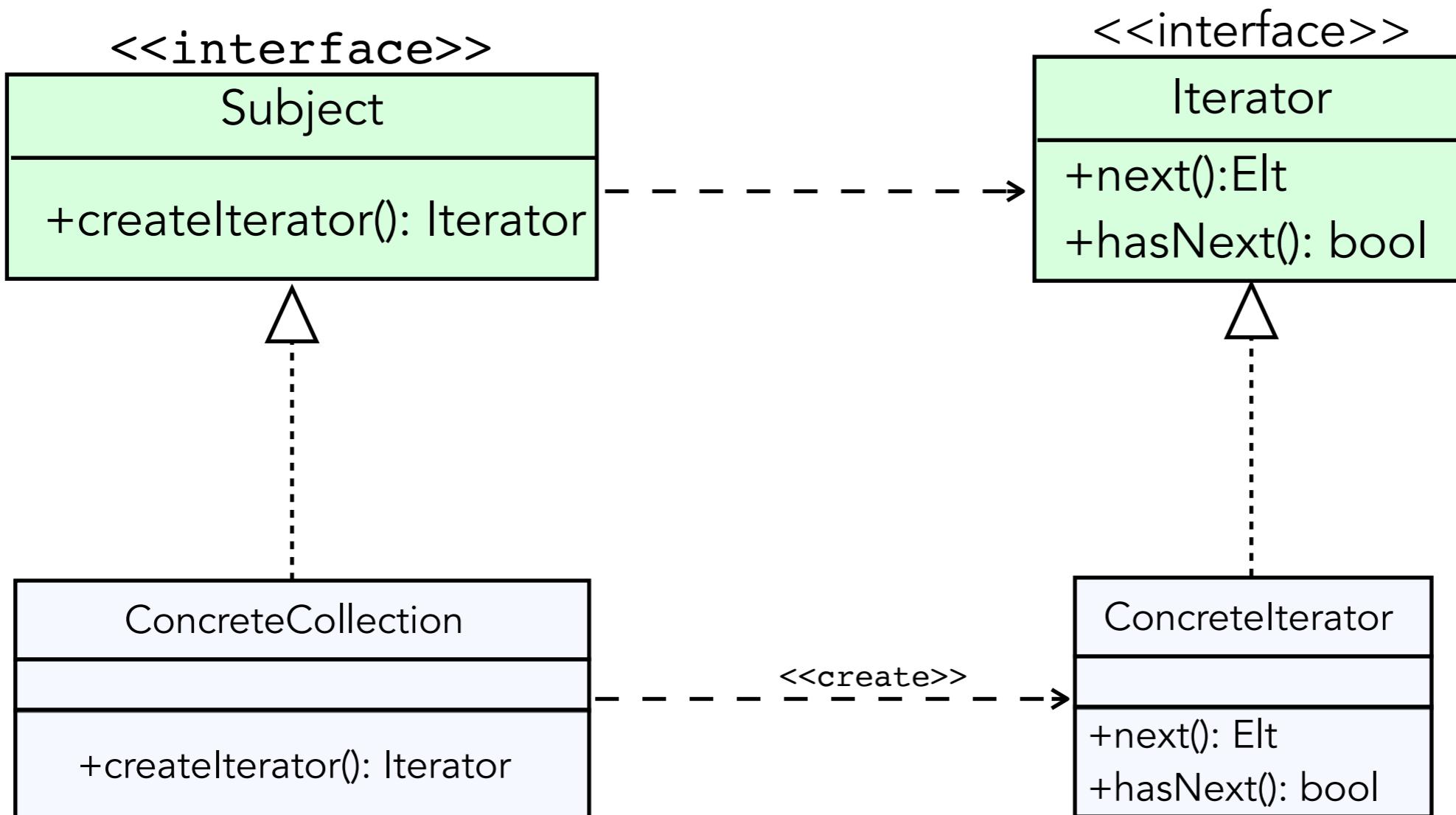
Recourir à des accesseurs/mutateurs



Inefficace pour plusieurs type de structures ()

Iterator Pattern

Pattern



Iterator Pattern

Pattern

```
public class Entreprise {
    private Employe[] employes;

    /**
     * L'itérateur de la classe Entreprise.
     * publique, car retournée par une méthode publique...
     */
    public class IteratorEntreprise implements Iterator<Employe> {
        private int position = 0; // position courante dans le tableau

        @Override
        public boolean hasNext() {
            return position < employes.length;
        }

        @Override
        public Employe next() {
            return employes[position++];
        } // position est POST-incrémenté

        /**
         * Construit l'itérateur.
         * Privé pour que seule la classe puisse y accéder !
         */
        private IteratorEntreprise() {
            position = 0;
        }
    } // public class IteratorEntreprise

    public IteratorEntreprise iterateur() {
        return new IteratorEntreprise();
    }

    public void affiche() {
        for (IteratorEntreprise ite = iterateur(); ite.hasNext();)
            System.out.println(ite.next().toString());
    }
} // public class Entreprise
```

Iterator Pattern

Itérateurs externes Java

Java supporte plusieurs types d'itérateurs différents :

- `Enumeration<E>` : interface antique (Java 1.0) :
 - `boolean hasMoreElements()` pour `hasNext()`
 - `E nextElement()` pour `next()`
- `Iterator<E>` : interface moderne (Java 1.2) :
 - `boolean hasNext()`
 - `E next()`
 - `void remove()` retire le dernier élément retourné par `next()`
- `ListIterator<E>` : ajoute à `Iterator` :
 - `boolean hasPrevious()` `hasNext()` en ordre inverse
 - `E previous()` `next()` en ordre inverse
 - `void add(E)` ajoute un élément avant le suivant.
 - `void set(E)` remplace le dernier élément retourné.

Iterator Pattern

Itérateurs externes Java

Depuis Java 5, le construct for-each est supporté :

```
LinkedList<Employe> liste = new LinkedList();
for (Employe e : liste)
    System.out.println(e.toString());
```

Qui est équivalent à écrire :

```
LinkedList<Employe> liste = new LinkedList();
for (Iterator<Employe> ite = list.iterator(); ite.hasNext())
    System.out.println(ite.next().toString());
```

Behavioral Patterns

- Command Pattern / Le Patron Commande
- State Pattern / Le Patron Etat
- Iterator Pattern / Le Patron Itérateur
- **Observer Pattern / Le Patron Observateur**
- Strategy Pattern / Le Patron Stratégie
- Visitor Pattern / Le Patron Visiteur
- MVC Pattern / Le Patron MVC

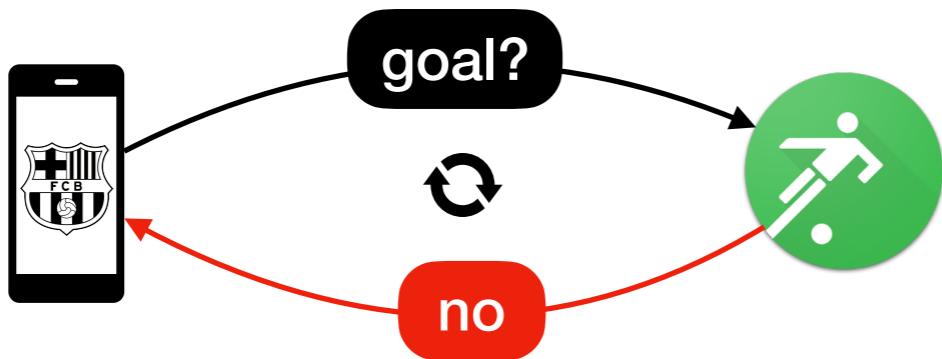
Behavioral Patterns

Observer Pattern / Le Patron Observateur

- **Observer** est un modèle de conception comportementale qui vous permet de définir un mécanisme d'abonnement pour informer plusieurs objets de tout événement qui se produit sur l'objet qu'ils observent.
- Principes SOLID :
 - **Dependency Inversion:** Promouvoir l'usage des interfaces

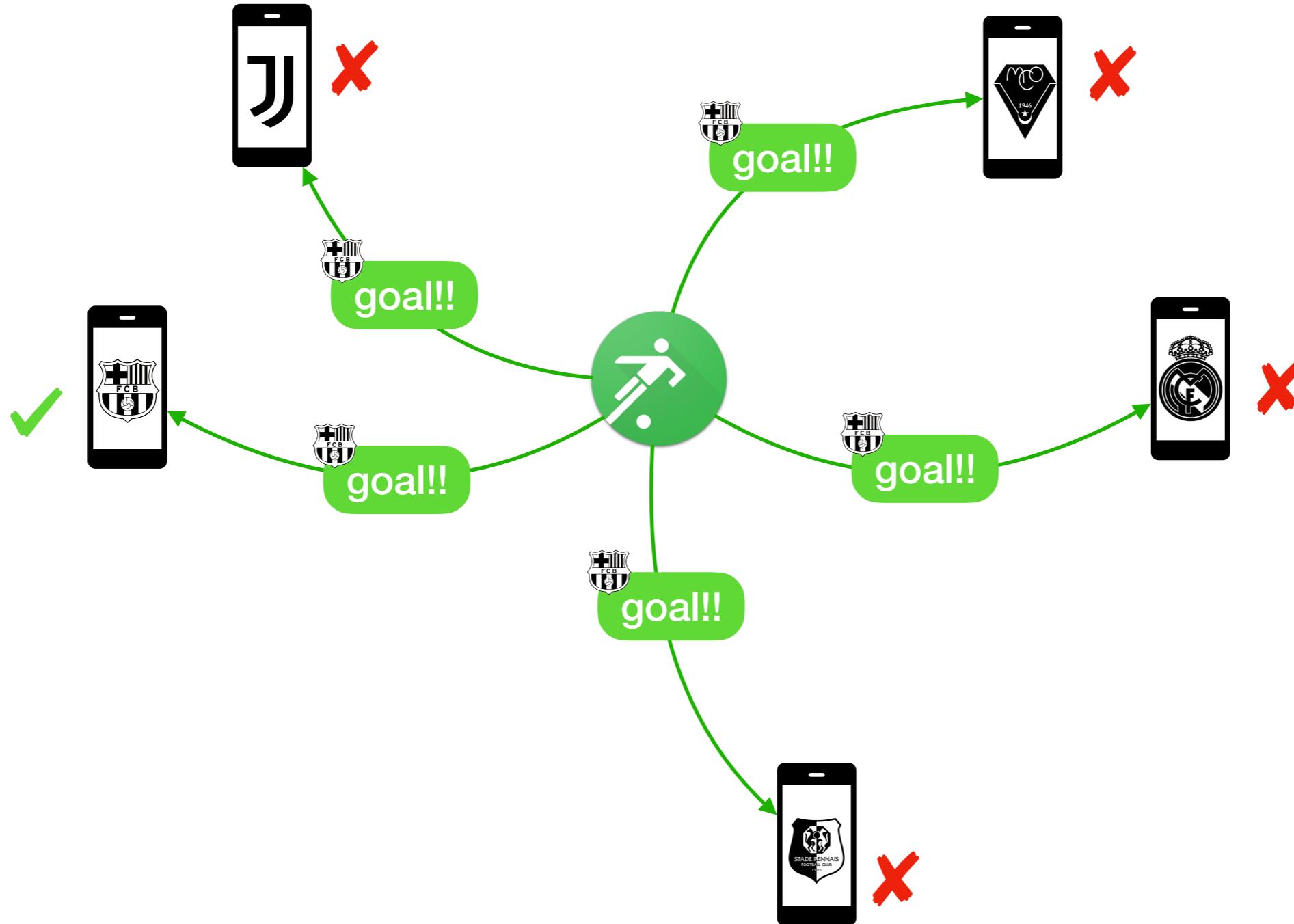
Observer Pattern

Problem



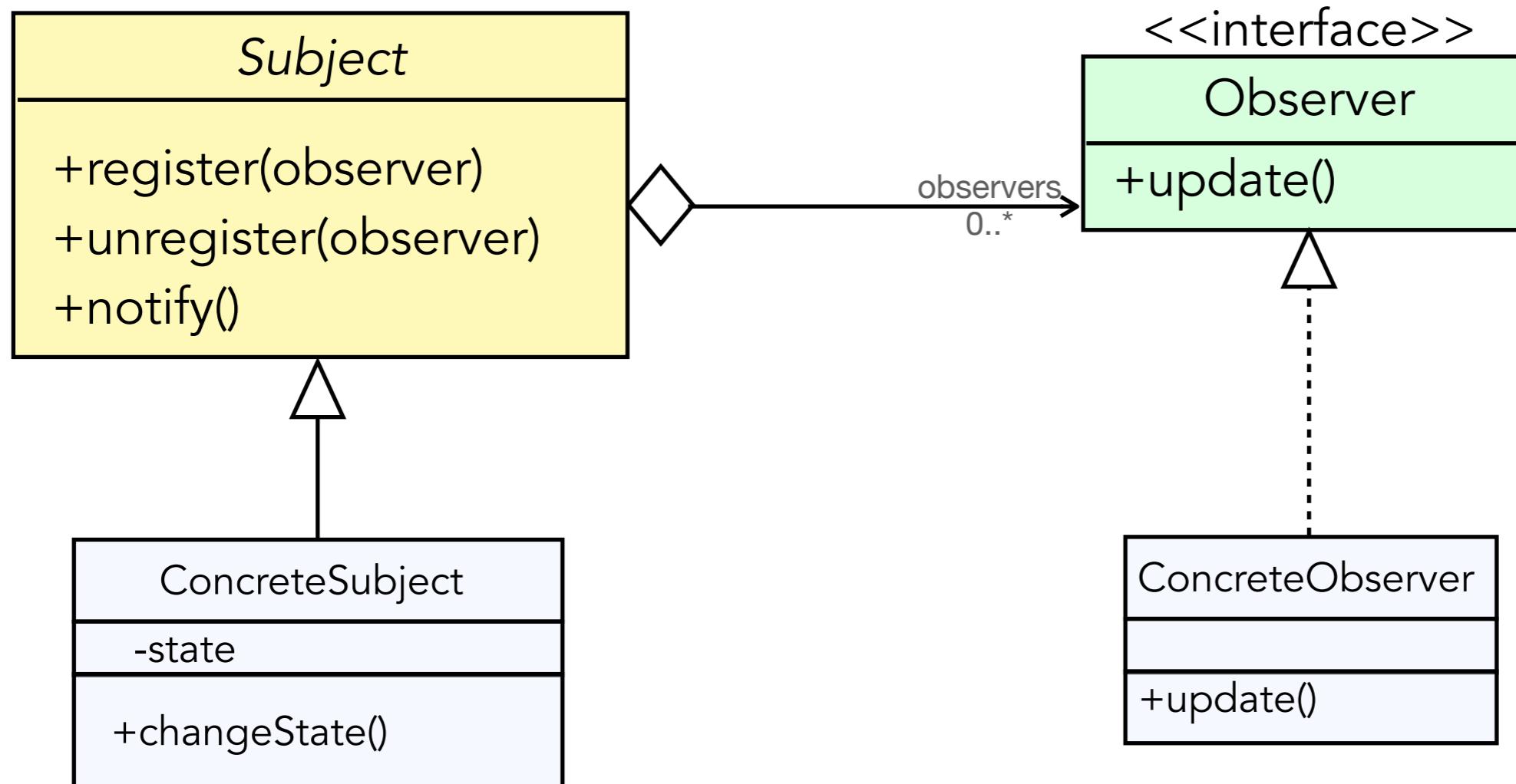
Observer Pattern

Problem



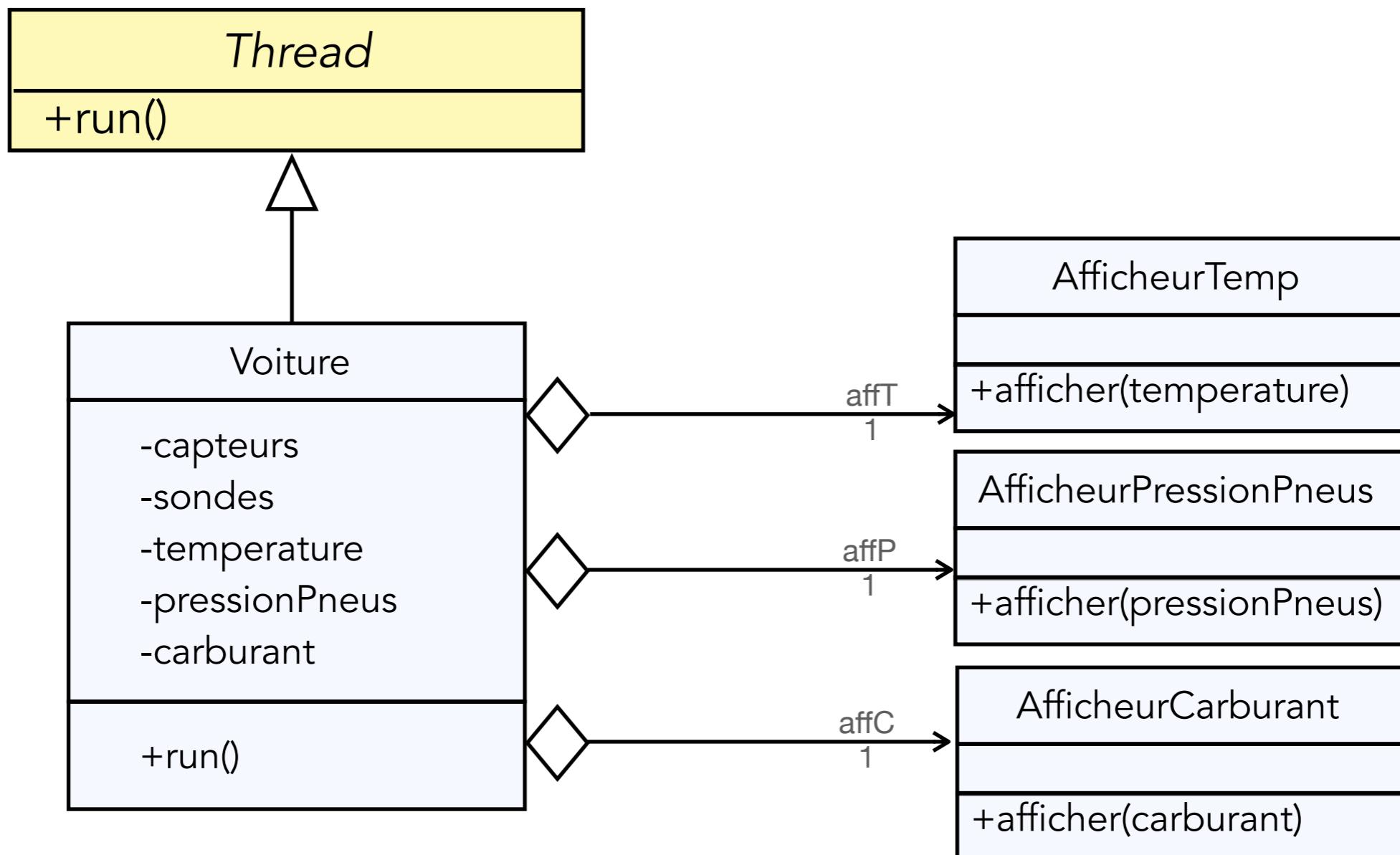
Observer Pattern

Pattern



Observer Pattern

Example



Observer Pattern

Example

```
@Override  
public void run() {  
    try {  
        while (!Thread.isInterrupted()) {  
            capteurs.wait(); // méthode blocante  
            temperature = capteurs.getTemperature();  
            pressionPneus = capteurs.getPression();  
  
            sondes.wait(); // méthode blocante  
            carburant = sondes.getCarburant();  
  
            affT.afficher(temperature);  
            affP.afficher(pressionPneus);  
            affC.afficher(carburant);  
  
        }  
    } catch (InterruptedException ex) {
```

Observer Pattern

Example

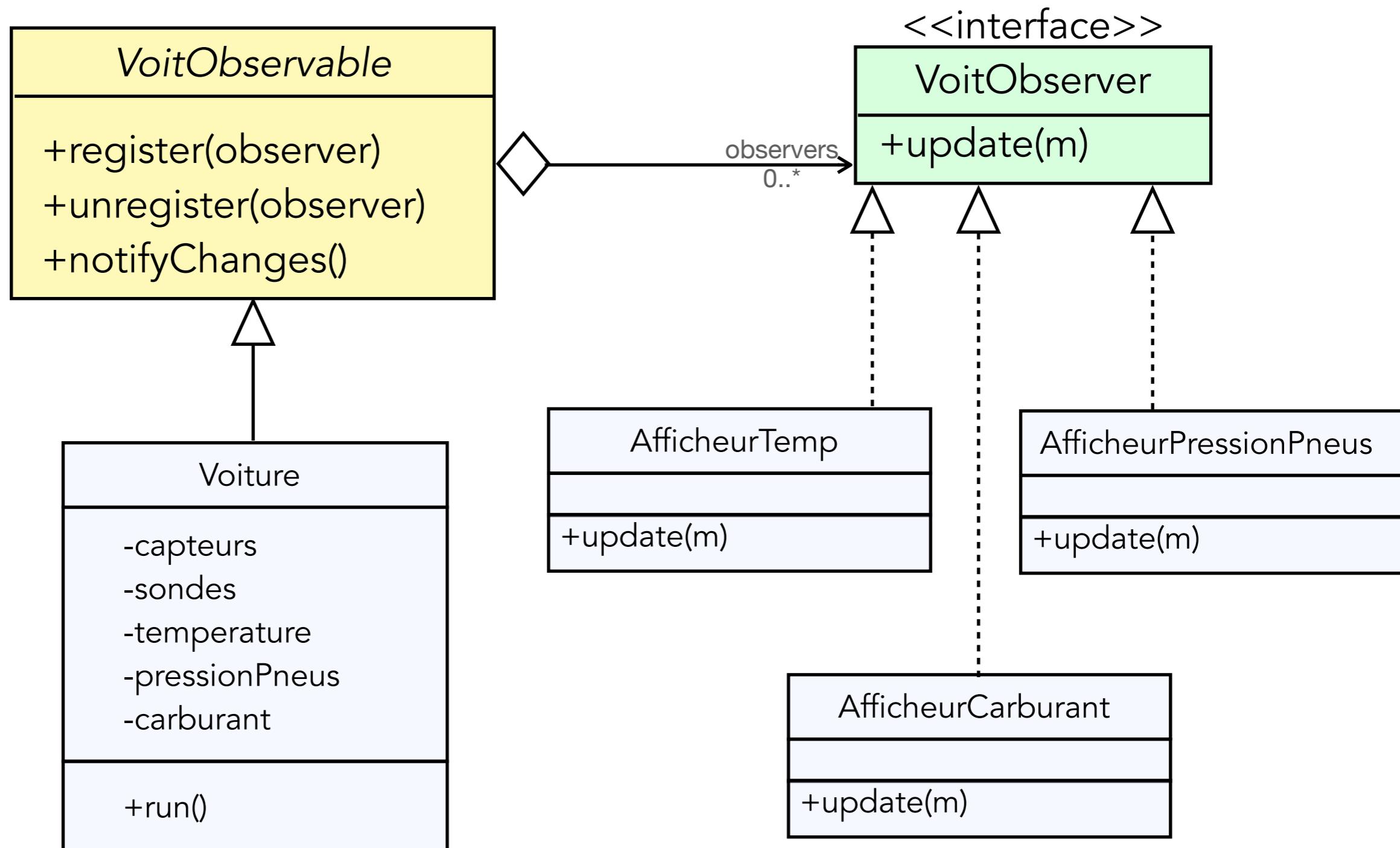
```
@Override  
public void run() {  
    try {  
        while (!Thread.isInterrupted()) {  
            capteurs.wait(); // méthode blocante  
            temperature = capteurs.getTemperature();  
            pressionPneus = capteurs.getPression();  
  
            sondes.wait(); // méthode blocante  
            carburant = sondes.getCarburant();  
  
            affT.afficher(temperature);  
            affP.afficher(pressionPneus);  
            affC.afficher(carburant);  
        }  
    } catch (InterruptedException ex) {
```



Couplage fort (ex. ajout de capteur)

Observer Pattern

Example



Behavioral Patterns

- Command Pattern / **Le Patron Commande**
- State Pattern / **Le Patron Etat**
- Iterator Pattern / **Le Patron Itérateur**
- Observer Pattern / **Le Patron Observateur**
- **Strategy Pattern / Le Patron Stratégie**
- Visitor Pattern / **Le Patron Visiteur**
- MVC Pattern / **Le Patron MVC**

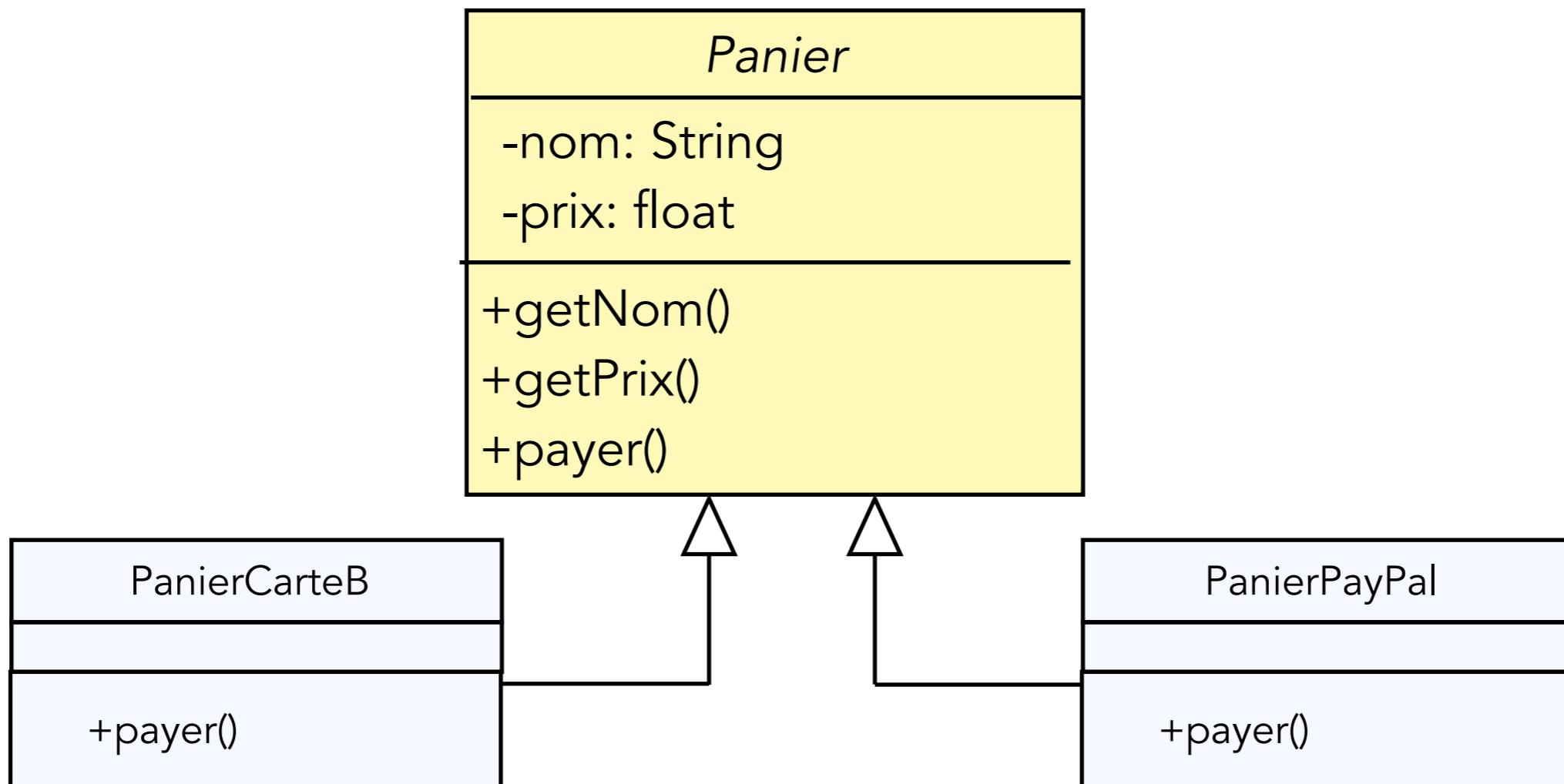
Behavioral Patterns

Strategy Pattern / Le Patron Stratégie

- **Strategy** est un pattern de conception comportementale qui vous permet de définir une famille d'algorithmes, de placer chacun d'eux dans une classe distincte et de rendre leurs objets interchangeables.
- Principes SOLID :
 - **Open/CLOSE:** Ouvert aux extensions, fermé aux modifications
 - **Liskov substitution:** comportements interchangeables
 - **Dependency Inversion:** Séparation et favorisation des interfaces

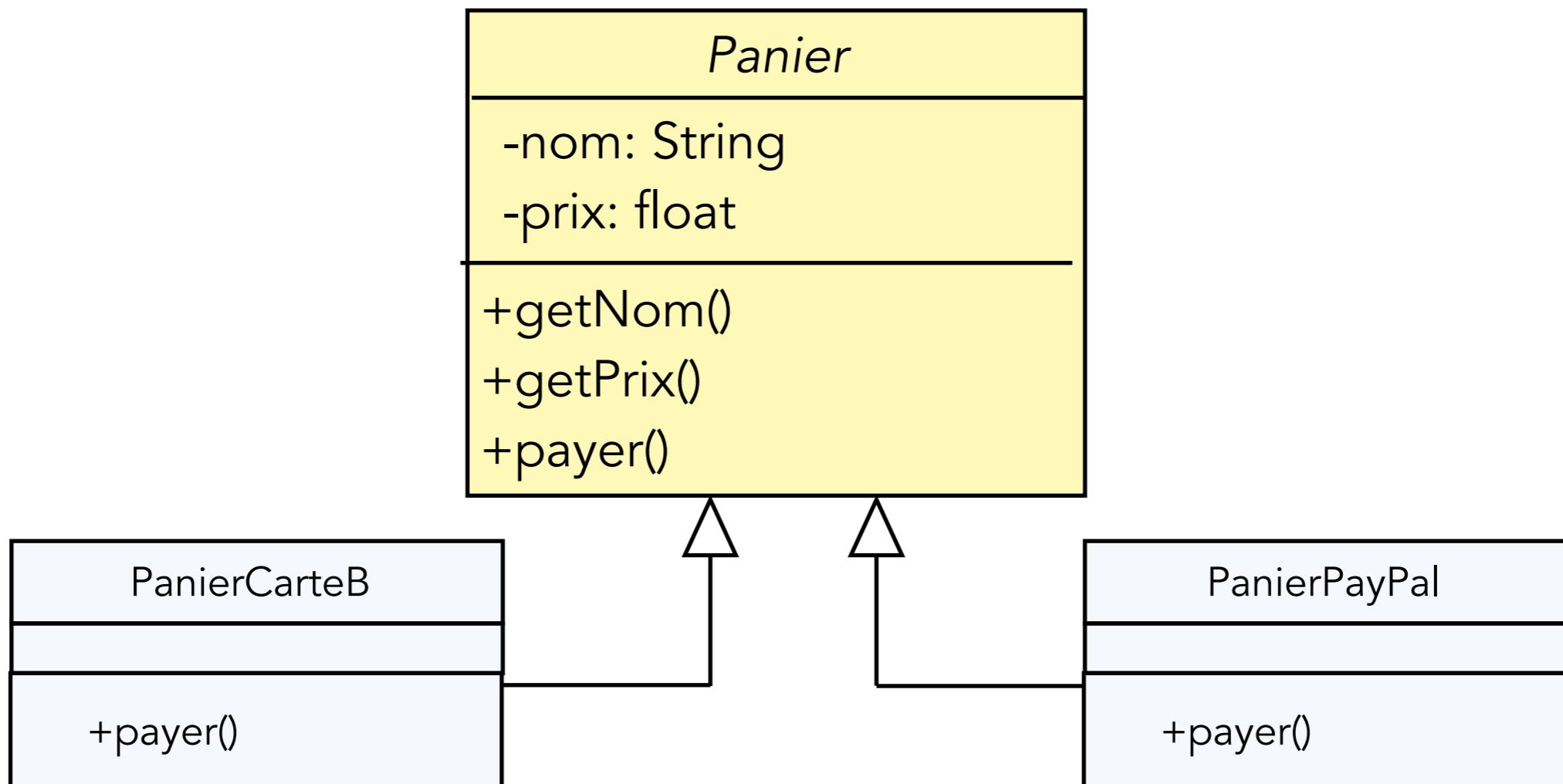
Strategy Pattern

Example



Strategy Pattern

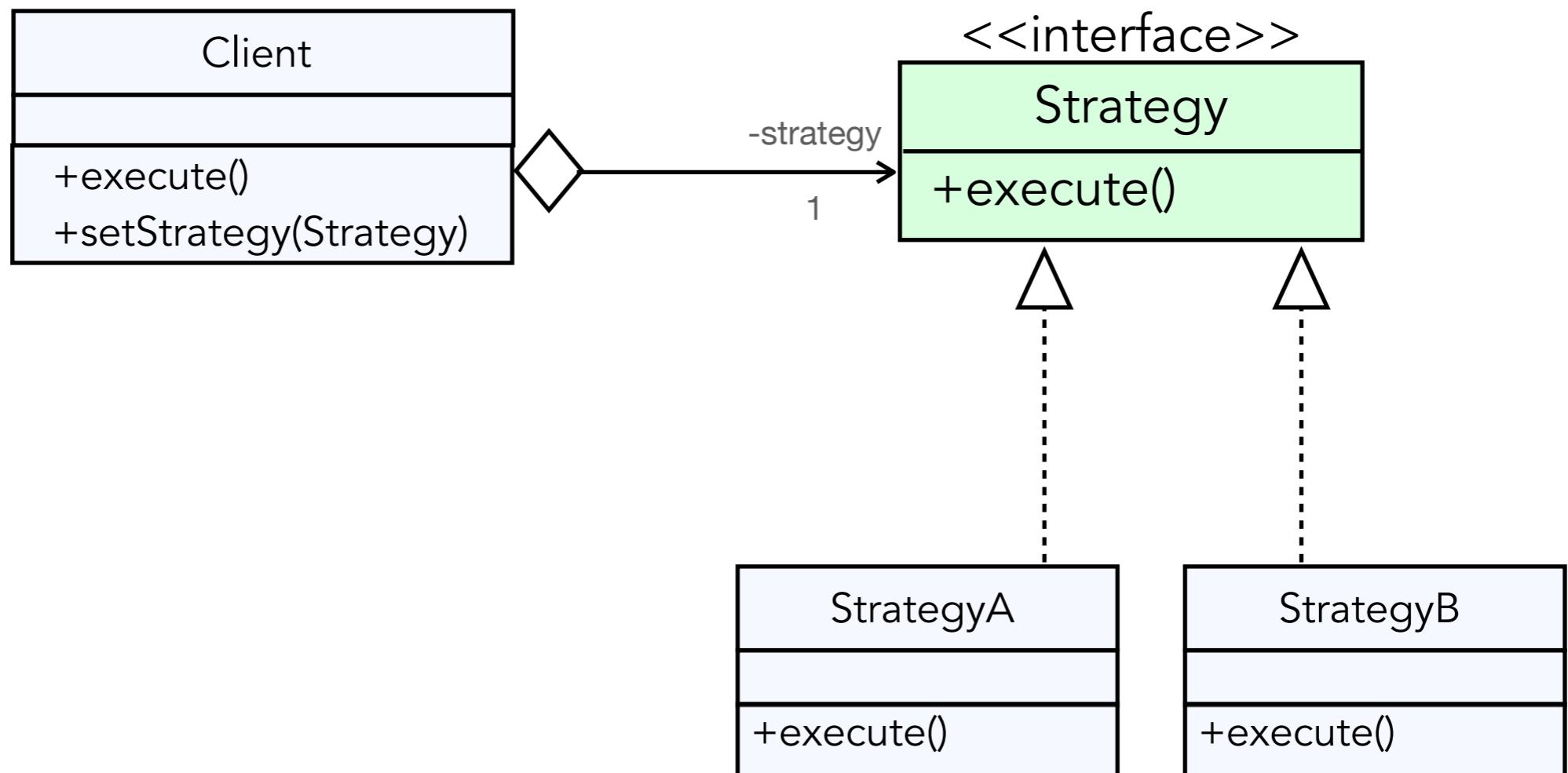
Example



Si un client change son moyen de payement ?

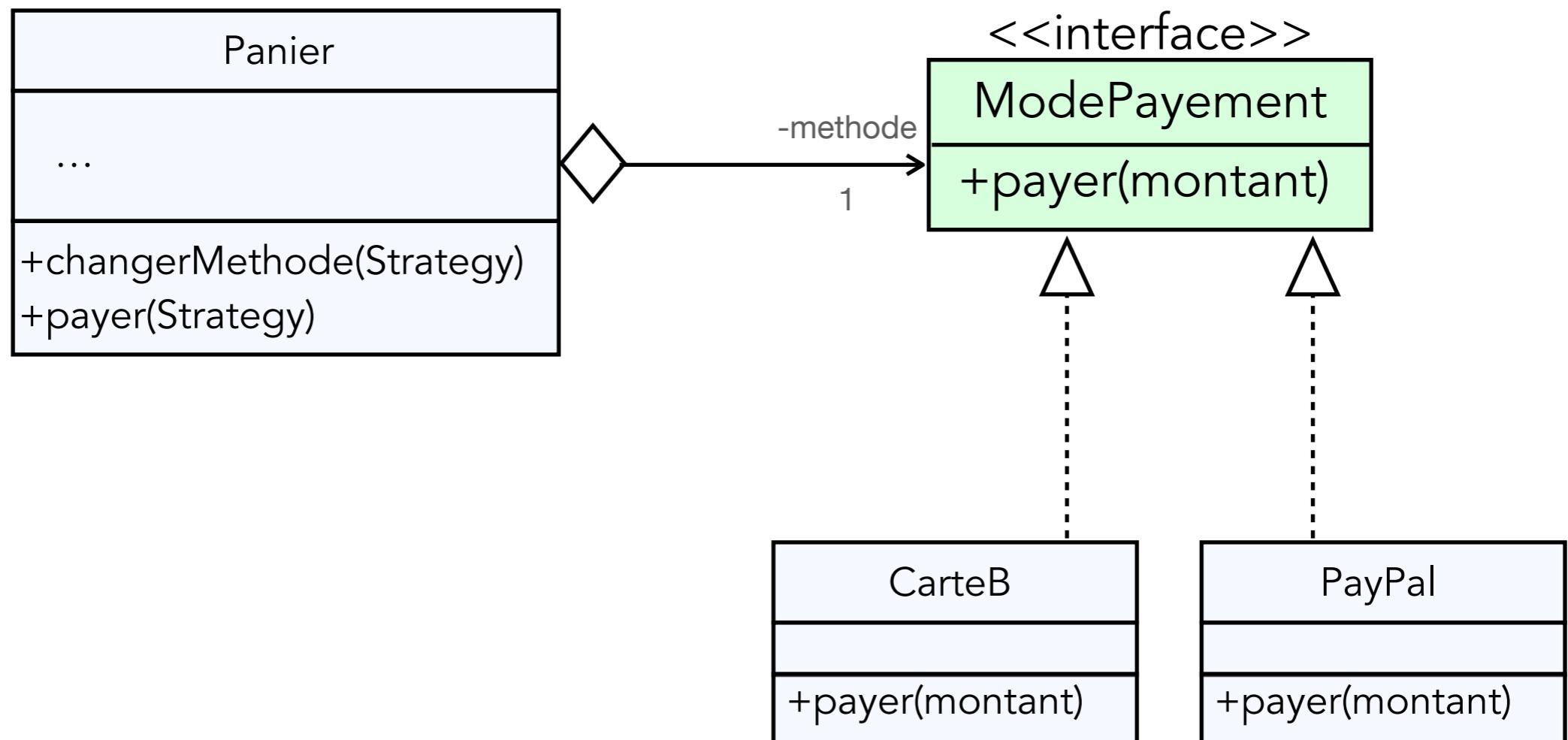
Strategy Pattern

UML Pattern



Strategy Pattern

Example



Strategy Pattern

Example

```
public interface ModePayement {  
    public void payer(int montant);  
}
```

Strategy Pattern

Example

```
public interface ModePayement {  
    public void payer(int montant);  
}
```

```
public class CarteB implements ModePayement {  
  
    private String numeroCarte;  
    private String cryptogramme;  
    private String dateExpiration;  
  
    public CarteB(String num, String crypto, String date) {  
        this.numeroCarte = num;  
        this.cryptogramme = crypto;  
        this.dateExpiration = date;  
    }  
    @Override  
    public void payer(int montant) {  
        System.out.println(montant + "€ payés par carte de crédit.");  
    }  
}
```

Strategy Pattern

Example

```
public interface ModePayement {  
    public void payer(int montant);  
}
```

```
public class CarteB implements ModePayement {  
  
    public class PayPal implements ModePayement{  
  
        private String id;  
        private String password;  
  
        public PayPal(String email, String pass){  
            this.id=id;  
            this.password=pass;  
        }  
  
        @Override  
        public void payer(int montant) {  
            System.out.println(montant + "€ payés par PayPal.");  
        }  
    }  
}
```

Strategy Pattern

Example

```
public interface ModePayement {  
    public void payer(int montant);  
}
```

```
public class Panier {  
  
    //details panier  
  
    private ModePayement methode;  
  
    public void changerMethode(ModePayement methode) {  
        this.methode= methode;  
    }  
    public void payer(ModePayement methode){  
        methode.payer(this.getPrix());  
    }  
}
```

Behavioral Patterns

- Command Pattern / **Le Patron Commande**
- State Pattern / **Le Patron Etat**
- Iterator Pattern / **Le Patron Itérateur**
- Observer Pattern / **Le Patron Observateur**
- Strategy Pattern / **Le Patron Stratégie**
- **Visitor Pattern / Le Patron Visiteur**
- MVC Pattern / **Le Patron MVC**

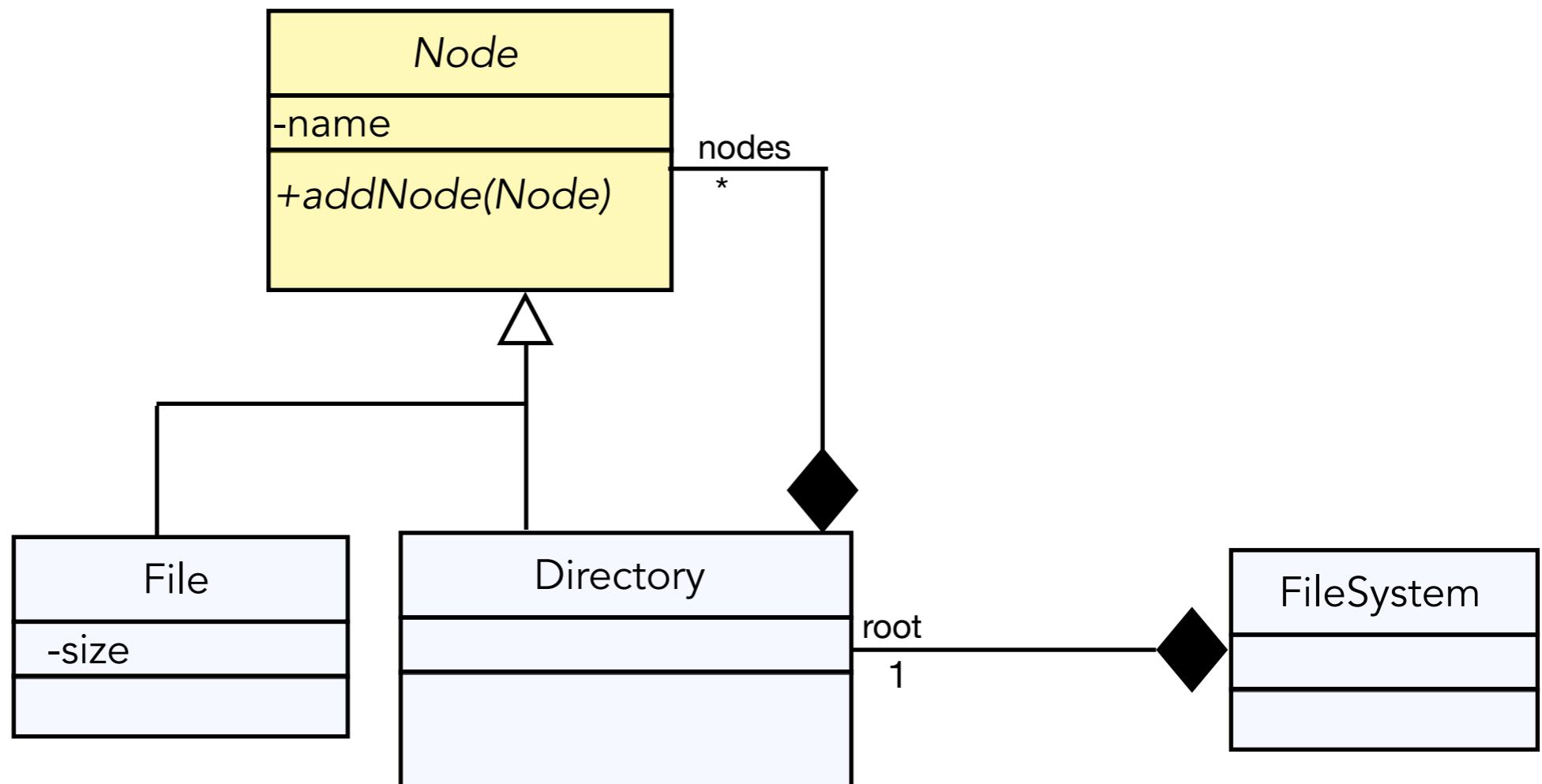
Behavioral Patterns

Visitor Pattern / Le Patron Visiteur

- **Visitor** est un modèle de conception comportementale qui vous permet de séparer les algorithmes des objets sur lesquels ils opèrent.
- Principes SOLID :
 - **Single Responsibility:** Un objet n'a qu'un seul rôle
 - **Open/Closed:** Ouvert aux extensions, fermé aux modifications
 - **Liskov substitution:** Usage du (double) polymorphisme

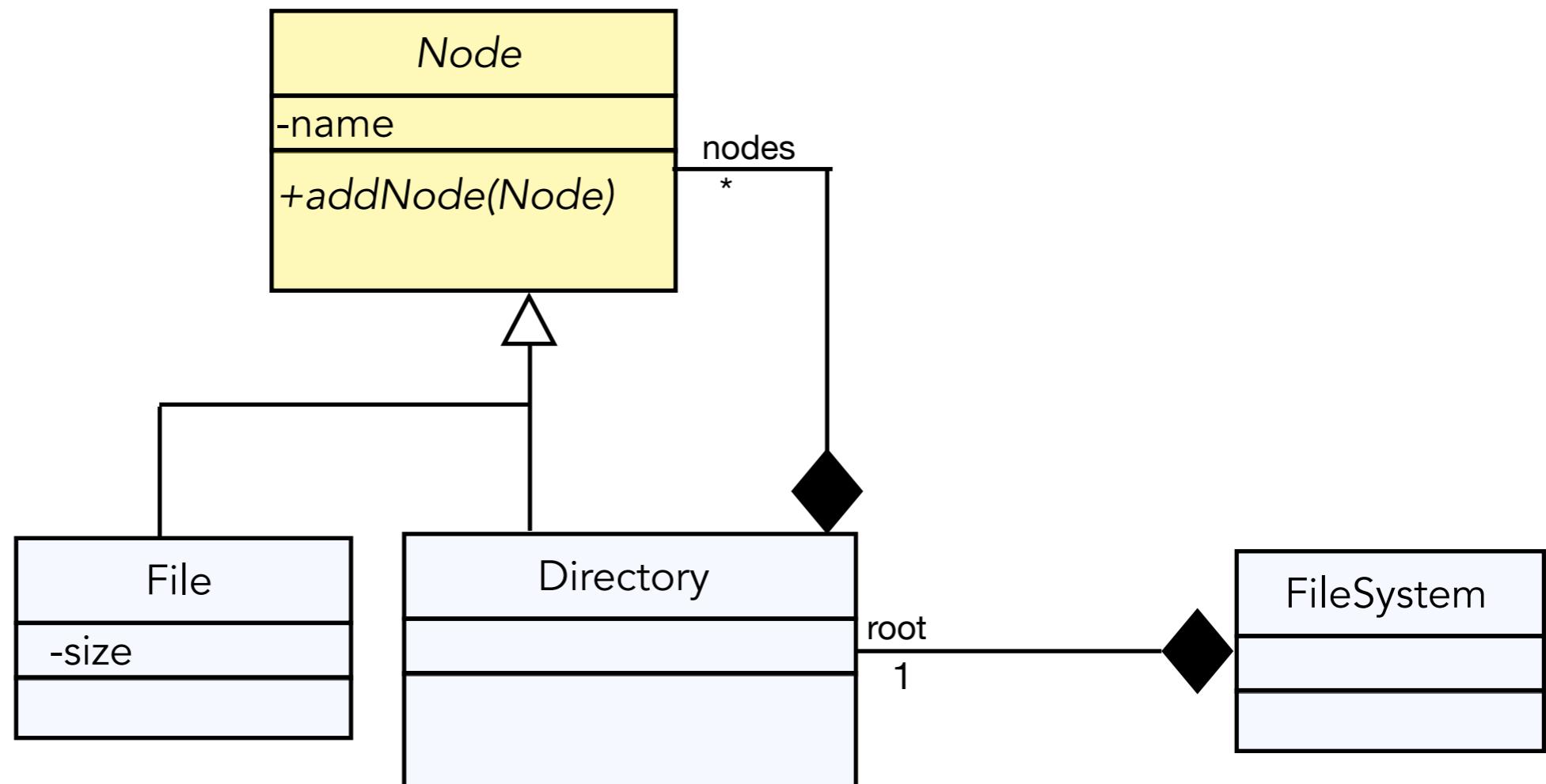
Visitor Pattern

Example



Visitor Pattern

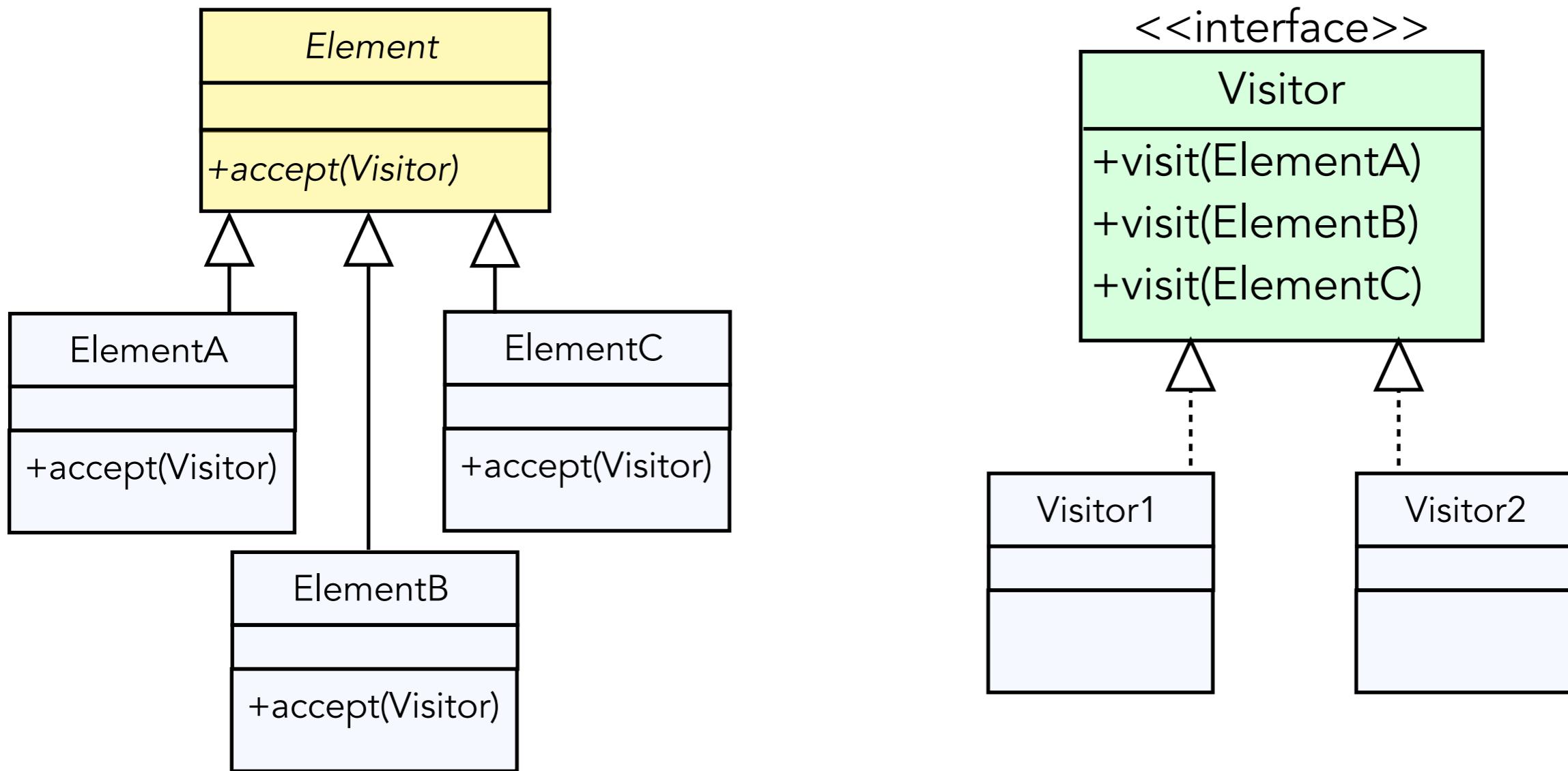
Example



Si on veut visiter le système pour calculer le nombre de fichiers ? De répertoire ? La taille ?...;

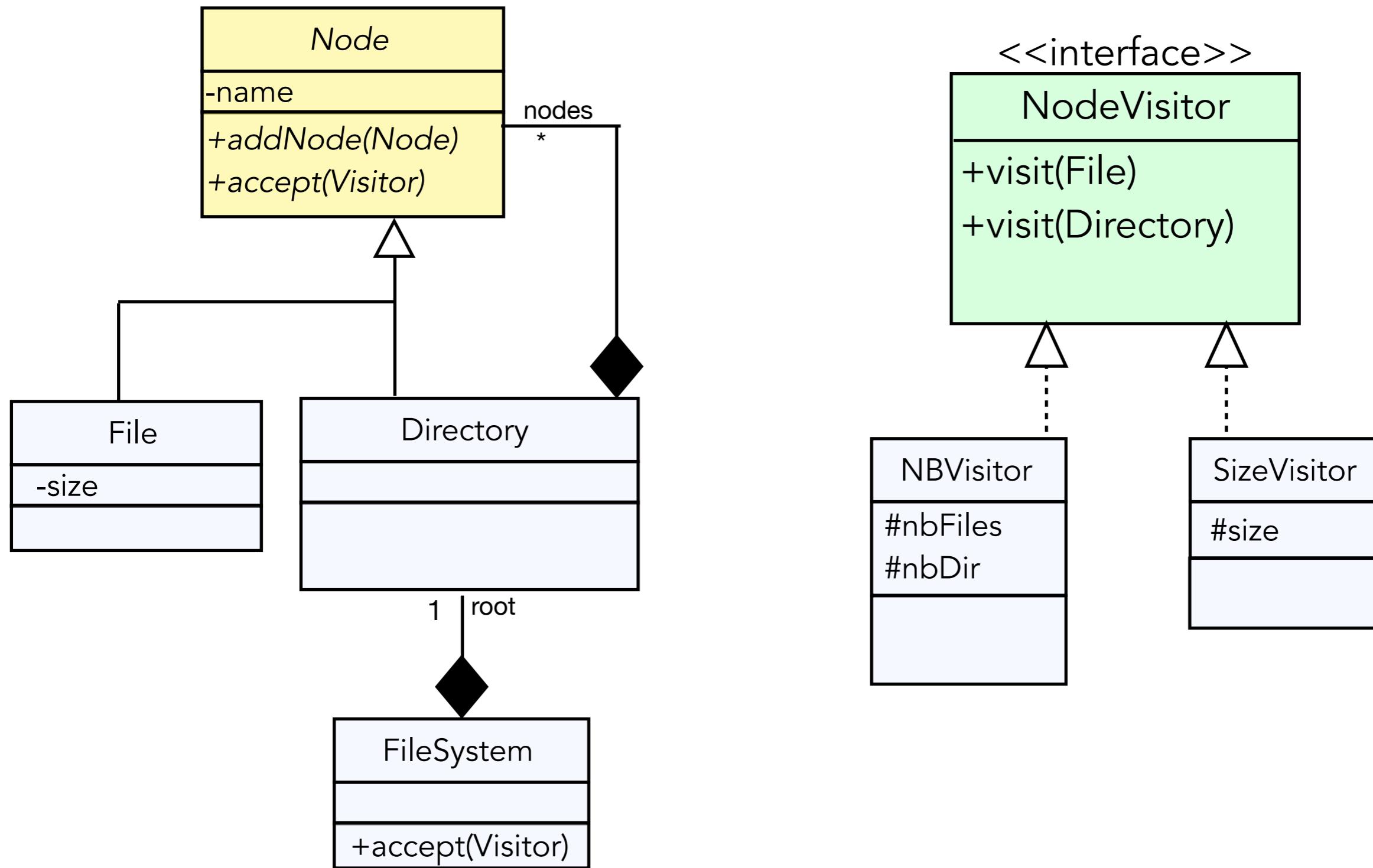
Visitor Pattern

UML Pattern



Visitor Pattern

Example



Behavioral Patterns

- Command Pattern / **Le Patron Commande**
- State Pattern / **Le Patron Etat**
- Iterator Pattern / **Le Patron Itérateur**
- Observer Pattern / **Le Patron Observateur**
- Strategy Pattern / **Le Patron Stratégie**
- Visitor Pattern / **Le Patron Visiteur**
- **MVC Pattern / Le Patron MVC**

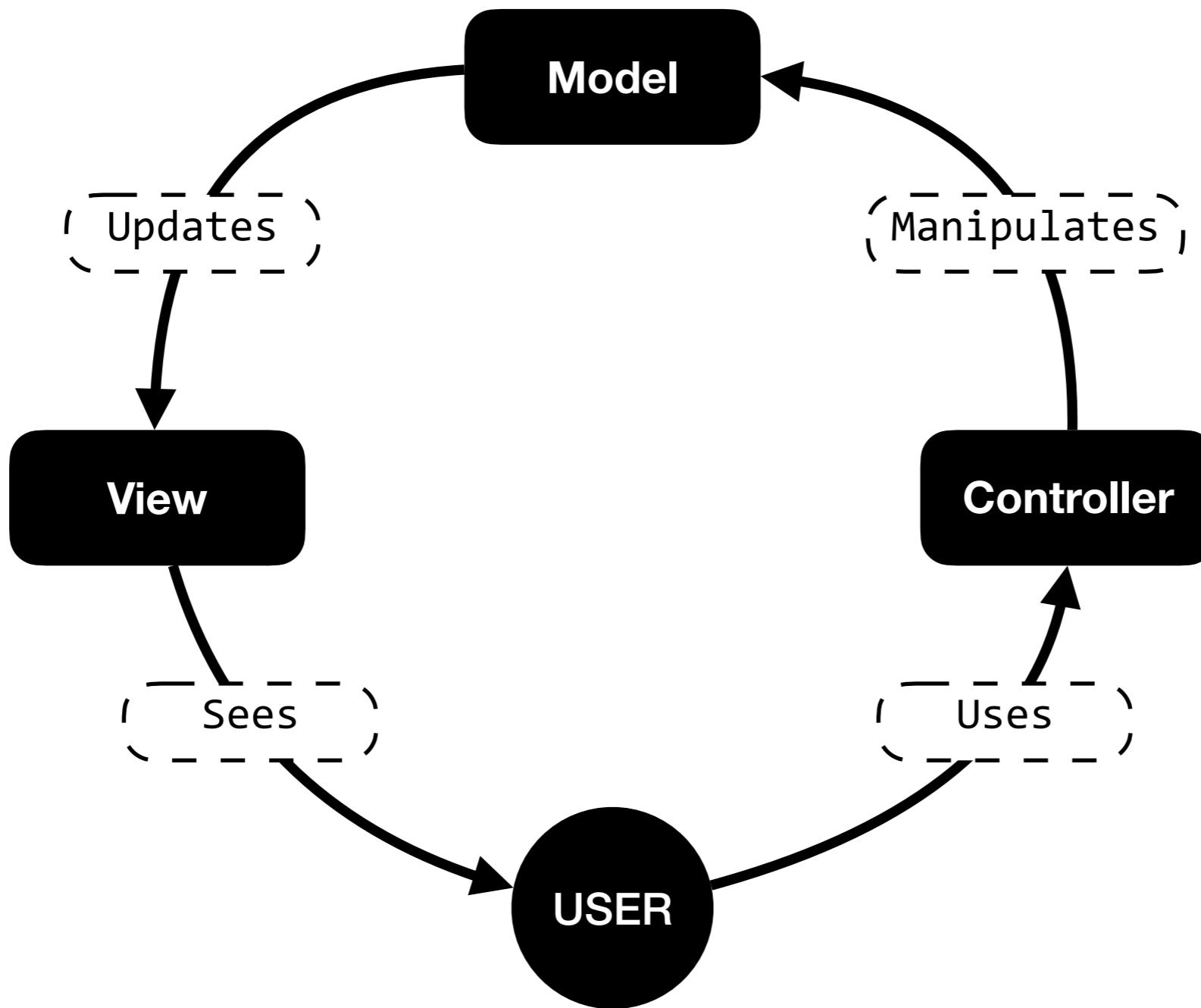
Behavioral Patterns

Model View Controller Pattern / Le Patron Modèle Vue Contrôleur

- **Model – view – controller** (MVC) est un pattern de conception comportementale couramment utilisé pour développer des interfaces utilisateur qui divise la logique de programme associée en trois éléments interconnectés.
- Principes SOLID :
 - **Single Responsibility**: Responsabilité unique des objets
 - **Interface segregation**: Séparation des interfaces
 - **Dependency Inversion**: Utilisation des interfaces

MVC Pattern

MVC Process



MVC Pattern

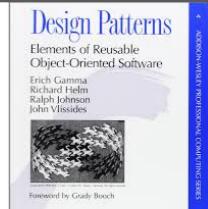
Un pattern de patterns

- Le patron MVC est en réalité un composite. Il ne définit que la **séparation des rôles**.
- En général :
 - Le modèle ne connaît ni la vue ni le contrôleur.
 - La vue et le contrôleur **observent** le modèle.
 - Le contrôleur **observe** la vue (reçoit les événements).
 - La vue peut **changer dynamiquement** de contrôleur.
 - La vue peut être **composée** de sous-vues.
- On réutilise donc les patrons **Observateur**, **Stratégie** et **Composite**.
- Les événements sont souvent gérés par un patron **Commande**.

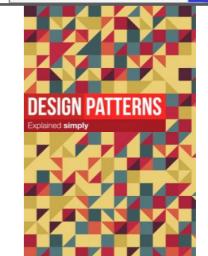
Books

Design Pattern

- **Design Patterns: Elements of Reusable Object-Oriented Software.** Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (GoF: Gang of Four).



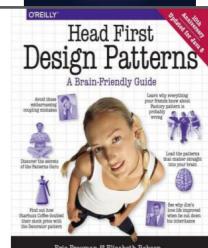
- **DESIGN PATTERNS Explained simply.** Alexander Shvets. 2013



- **Dive Into Design Patterns.** Alexander Shvets. 2019



- **Head First Design Patterns.** Freeman et al. 2014



- **Java Design Patterns.** Vaskaran Sarcar. 2019

