



## CHAPITRE 6

### Design patterns de comportement (1 / 2)

2017/2018

Nour El Houda Ben Youssef

# Pattern de comportement

2

- ❑ Le modèle de comportement simplifie l'organisation d'exécution des objets.
- ❑ Une fonction est composée d'un ensemble d'actions qui parfois appartiennent à des domaines différents de la classe d'implémentation.
- ❑ On aimerait donc pouvoir "déléguer" certains traitements à d'autres classes.
- ❑ D'une manière générale, un modèle de comportement permet de réduire la complexité de gestion d'un objet ou d'un ensemble d'objets.

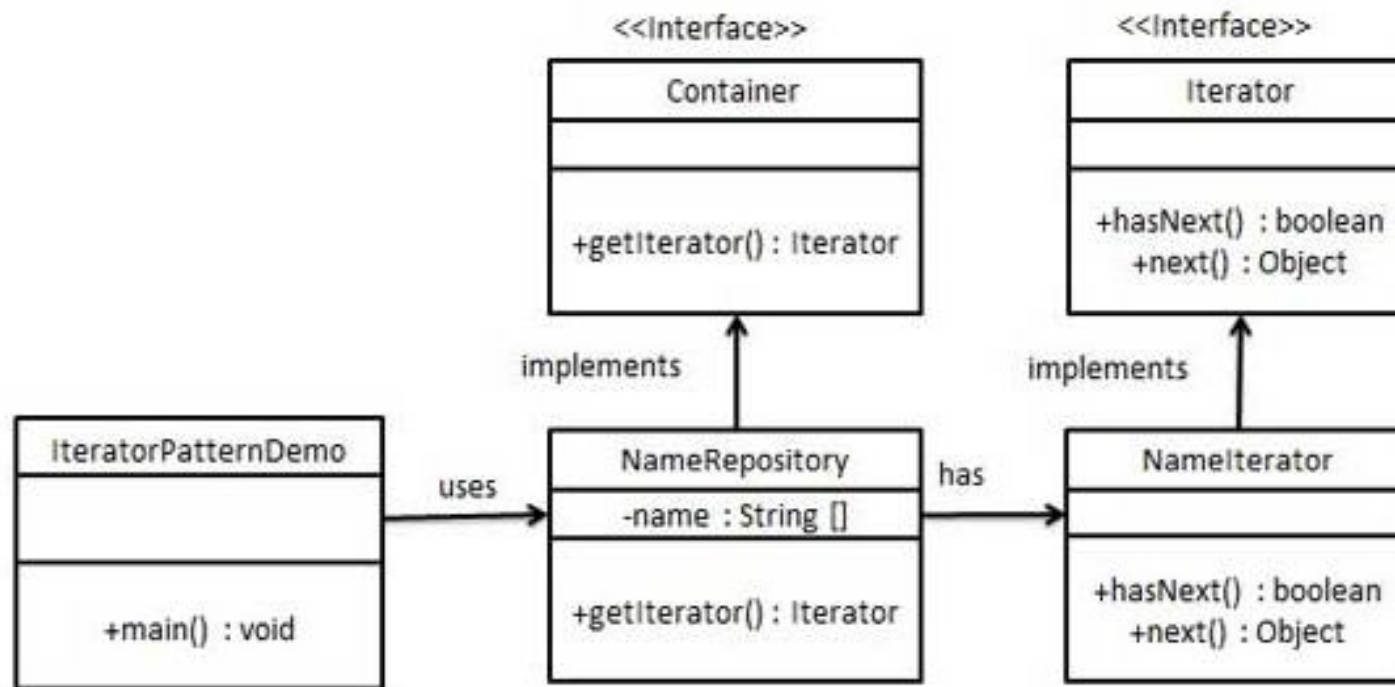
# Iterator

3

- ❑ L'itérateur ou Iterator est le plus commun des modèles de comportement.
- ❑ Une collection contient un ensemble d'objets stocké par différentes méthodes (un tableau, un vecteur...)
- ❑ L'exploitant qui accède au contenu de la collection ne souhaite pas être concerné par cette manière de gérer les objets.
- ❑ La collection offre donc un point d'accès unique sous la forme d'une interface Iterator.

# Structure

4



# Exemple

5

```
/** Classe de gestion d'un espace de dessin */
public class CanvasImpl implements GraphicsElement, Canvas {
    // Tableau pour stoker les éléments de la collection
    private GraphicsElement[] ge;
    ...
    /** Retourne un itérateur pour accéder aux objets de la collection */
    public Iterator getIterator() { return ArrayIterator( ge ); }
}

/** Interface pour toutes les collections d'objets de GraphicElement */
public interface Iterator {
    public GraphicElement getNextElement();
}
```

# Exemple

6

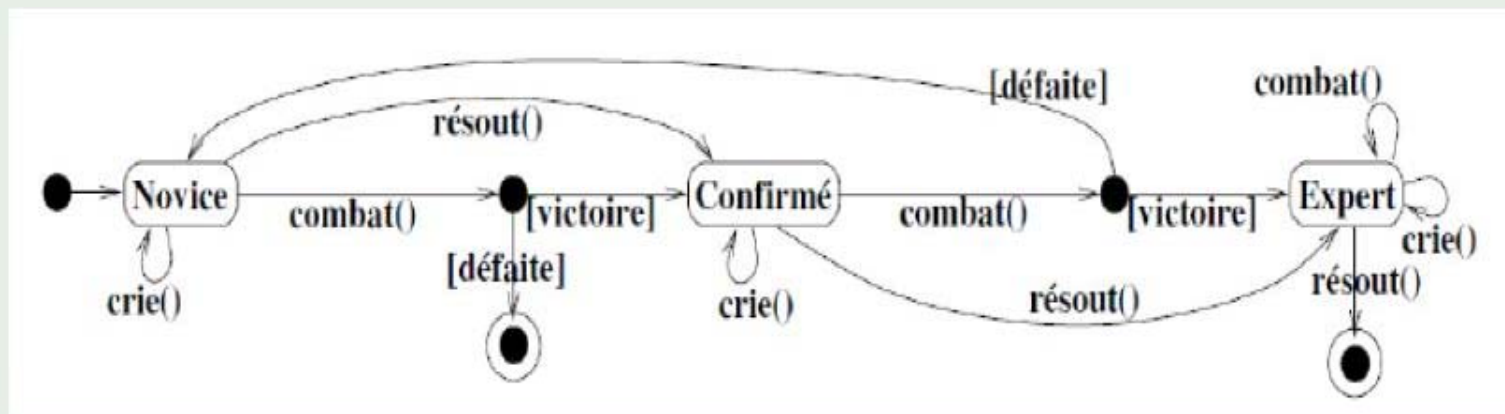
```
/** Iterateur parcourant un tableau pour retourner des objets de type GraphicElement */
public class ArrayIterator implements Iterator {
    private GraphicElement[] ge;
    private int nge;
    /** Constructeur avec un tableau de données à parcourir */
    public ArrayIterator( GraphicElement[] ge ) {
        this.ge = ge;
        nge = 0;
    }
    /** Retourne chaque élément de la collection ou null */
    public GraphicElement getNextElement() {
        if ( nge >= ge.length ) return null;
        return ge[ nge++ ];
    }
}
```

# State (état)

7

- But : Modifier les actions d'un objet selon son état.
- Contexte: L'objet passe par un ensemble connu d'états qui influencent ses méthodes.

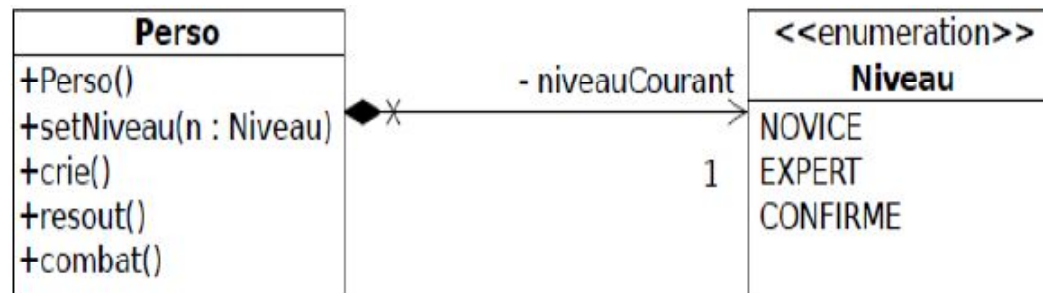
Les personnages d'un jeu peuvent avoir 3 niveaux (états) et peuvent combattre, résoudre des énigmes et crier :



# State

8

## □ Une première solution:



- Sol. 1 : Chaque méthode de Perso contient un cas par état

```
public void resout(){
    if (niveauCourant == NOVICE){ ... ; niveauCourant = CONFIRME
    ;}
    else if (niveauCourant == CONFIRME){ ... ; niveauCourant =
    EXPERT;}
    else { ... ; System.exit(0);} }
```

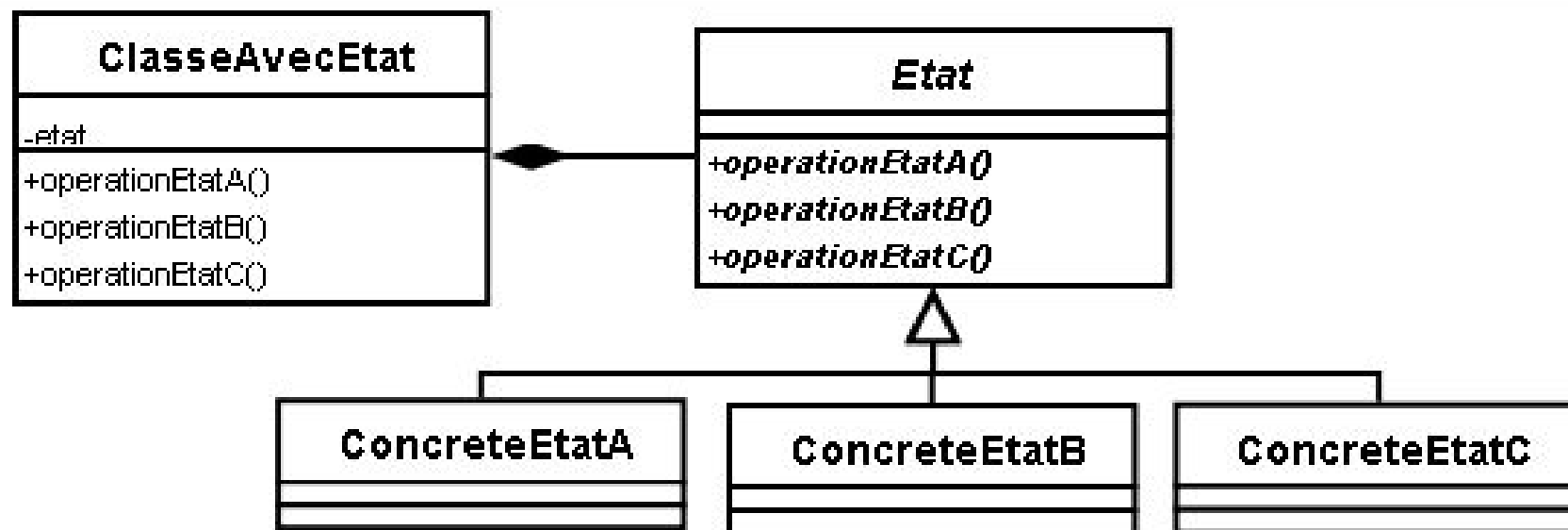
→ Coûteux d'ajouter un nouvel état



# State

9

- Bonne solution



# State: Rôles

10

- **ClasseAvecEtat** : est une classe avec état. Son comportement change en fonction de son état. La partie changeante de son comportement est déléguée à un objet **Etat**.
- **Etat** : définit l'interface d'un comportement d'un état.
- **ConcreteEtatA, ConcreteEtatB et ConcreteEtatC** : sont des sous-classes concrètes de l'interface **Etat**. Elles implémentent des méthodes qui sont associées à un **Etat**.

# Exemple

11

*State.java*

```
public interface State {  
    public void doAction(Context context);  
}
```

*StartState.java*

```
public class StartState implements State {  
  
    public void doAction(Context context) {  
        System.out.println("Player is in start state");  
        context.setState(this);  
    }  
  
    public String toString(){  
        return "Start State";  
    }  
}
```

*StopState.java*

```
public class StopState implements State {  
  
    public void doAction(Context context) {  
        System.out.println("Player is in stop state");  
        context.setState(this);  
    }  
  
    public String toString(){  
        return "Stop State";  
    }  
}
```

# Example

12

*Context.java*

```
public class Context {  
    private State state;  
  
    public Context(){  
        state = null;  
    }  
  
    public void setState(State state){  
        this.state = state;  
    }  
  
    public State getState(){  
        return state;  
    }  
}
```

```
public class StatePatternDemo {  
    public static void main(String[] args) {  
        Context context = new Context();  
  
        StartState startState = new StartState();  
        startState.doAction(context);  
  
        System.out.println(context.getState().toString());  
  
        StopState stopState = new StopState();  
        stopState.doAction(context);  
  
        System.out.println(context.getState().toString());  
    }  
}
```

Verify the output.

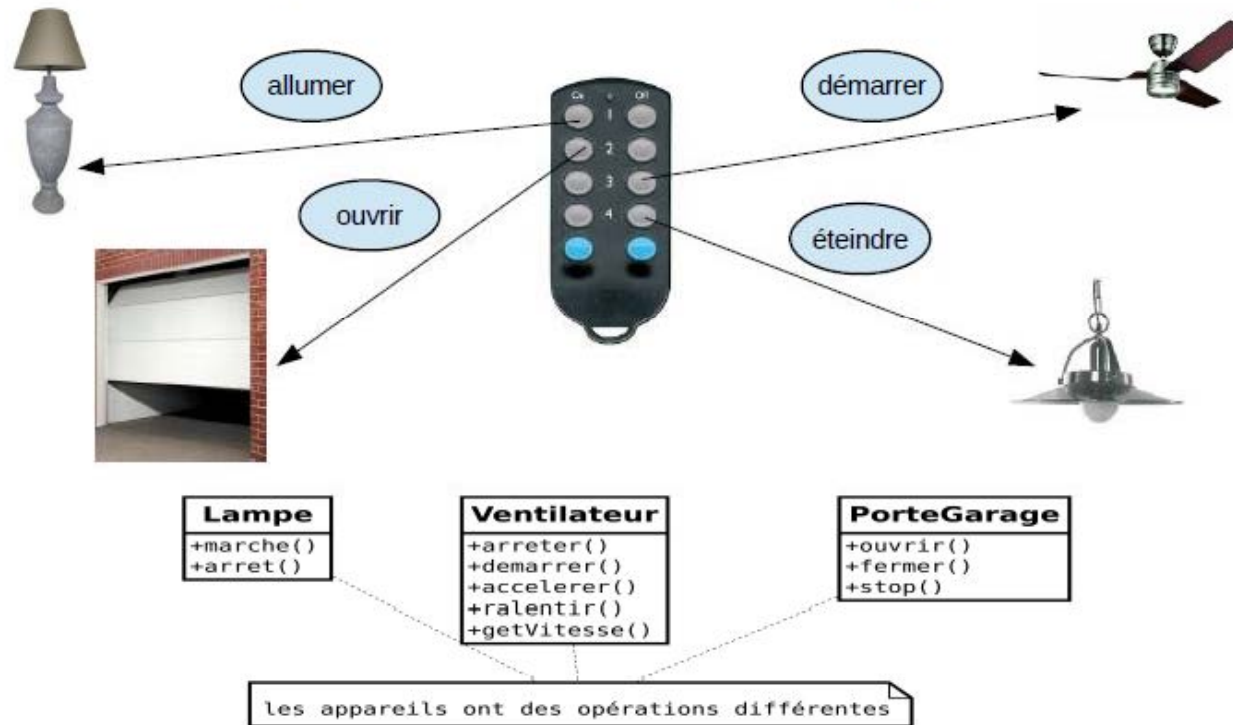
```
Player is in start state  
Start State  
Player is in stop state  
Stop State
```

# Commande (command)

13

## Enregistrer des requêtes et séparer le demandeur du réalisateur d'une requête

Exemple : gérer une télécommande domotique



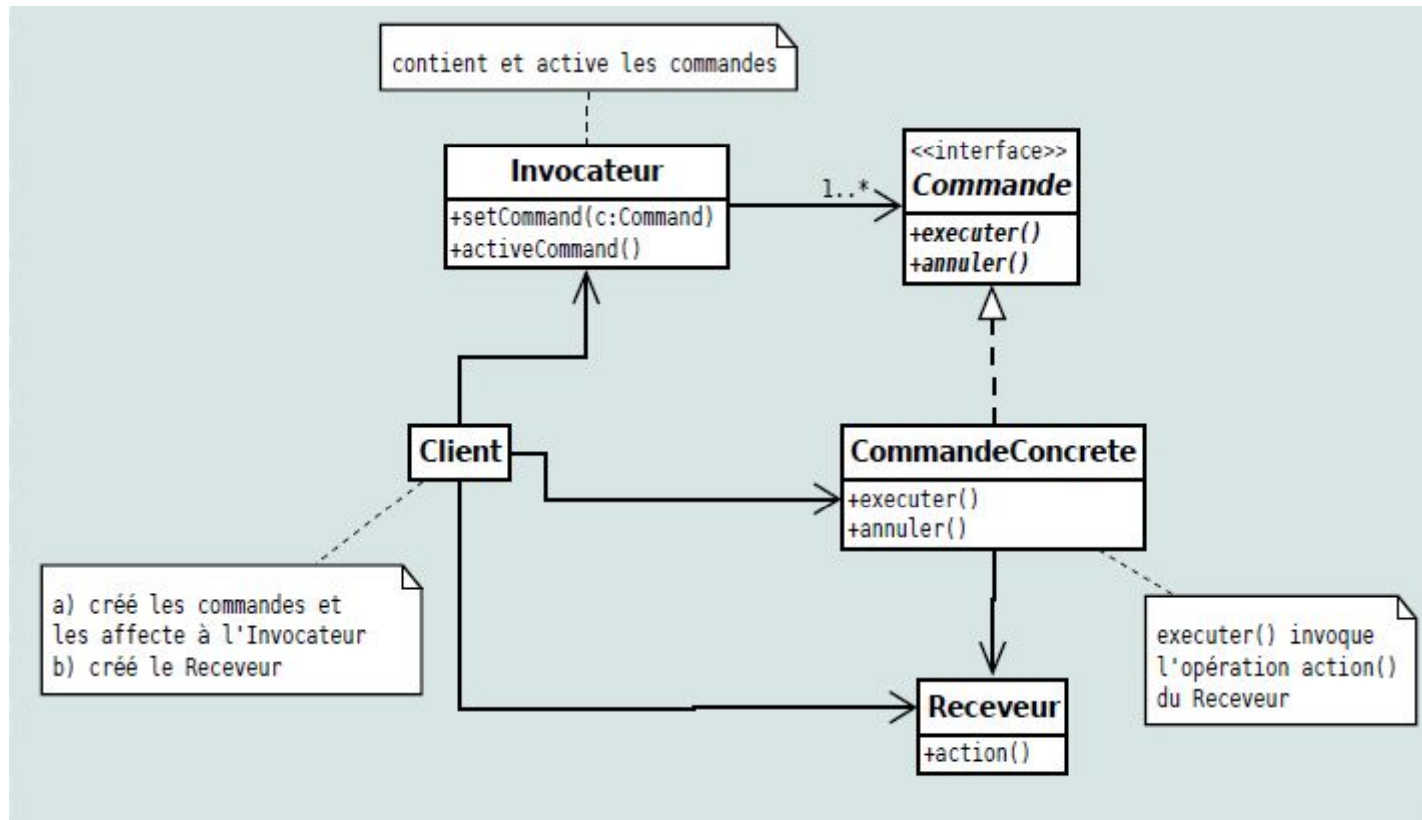
# Command

14

- ❑ Le design pattern *Commande* encapsule une requête comme un objet,
- ❑ Permet le paramétrage facile des requêtes.
- ❑ Permet la réversibilité des opérations.

# Structure

15



(dans l'exemple précédemment présenté l'invocateur est la télécommande, les receveurs sont les lampes et le ventilateur)

# Rôles

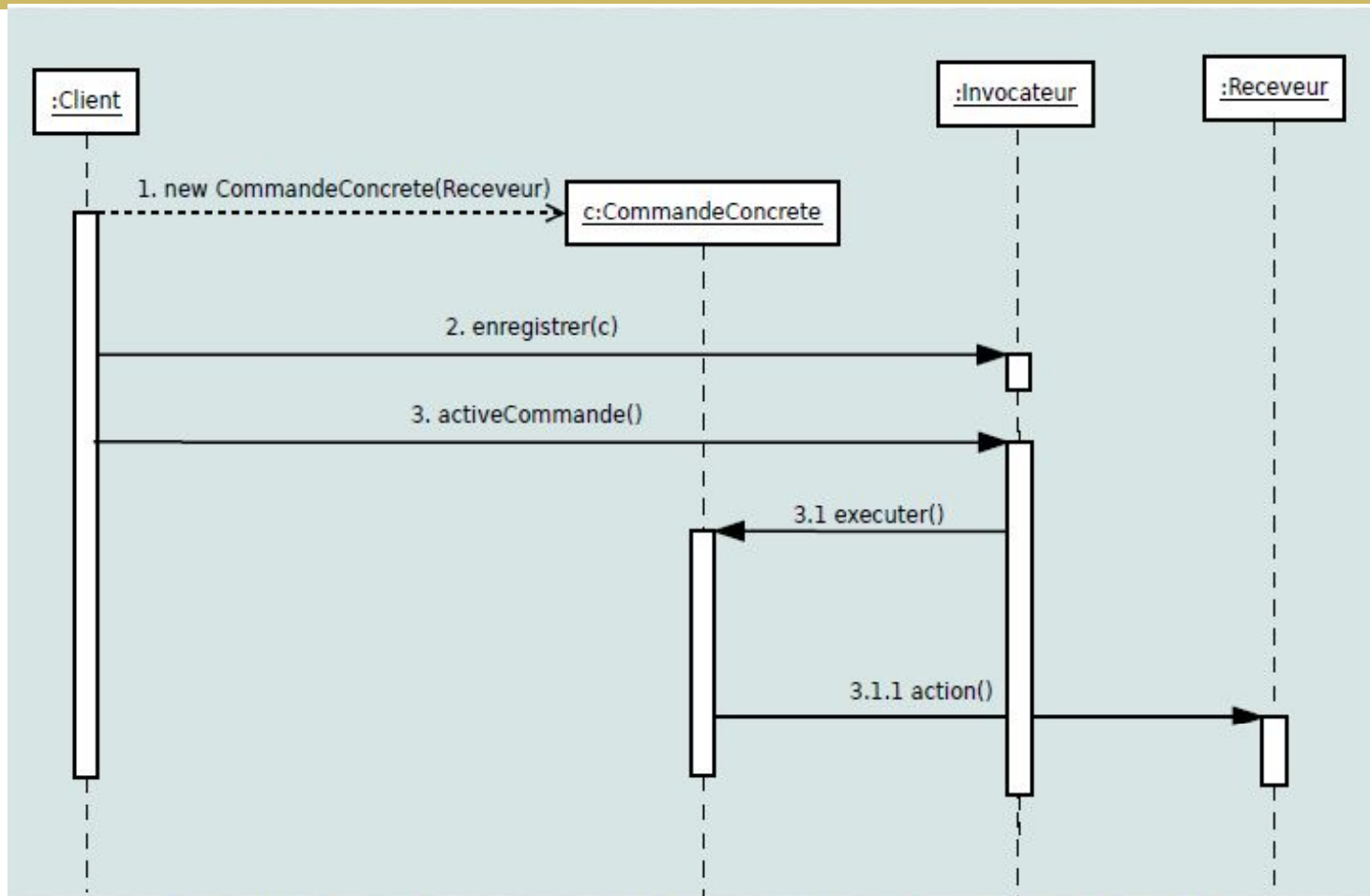
16

- **Commande** : définit l'interface d'une commande.
- **CommandeConcrete** : implémente une commande. Elle implémente la méthode **executer()** et **annuler()**, en appelant des méthodes de l'objet **Recepteur**.
- **Invocateur** : déclenche la commande. Il appelle la méthode **executer()** d'un objet **Commande**.
- **Receveur** : reçoit la commande et réalise les opérations associées. Chaque objet **Commande** concret possède un lien avec un objet **Receveur**.
- La partie cliente configure le lien entre les objets **Commande** et le **Receveur**.



# Dynamique de command

17



Le *Client* 1. crée la commande, 2. l'enregistre dans l'*Invocateur*; 3. demande à l'*Invocateur* de l'activer : l'*Invocateur* 3.1 s'adresse à la commande 3.1.1 qui s'adresse au *Receveur*

# Exemple

18

```
public interface Commande
{
    public abstract void executer() ;
    public abstract void annuler() ;
}
```

# Exemple

19

```
public class AllumerLampe implements Commande
{
    private Lampe lampe ;

    public AllumerLampe(Lampe uneLampe)
    {this.lampe = uneLampe ;}

    public void executer() {this.lampe.marche() ;}

    public void annuler() {this.lampe.arrete() ;}
}
```

*lampe.arrete() restaure l'état  
de la lampe précédant l'appel à  
lampe.marche() :*

```
public class EteindreLampe implements Commande
{
    private Lampe lampe ;

    public EteindreLampe(Lampe uneLampe)
    {this.lampe = uneLampe ;}

    public void executer() {this.lampe.arrete() ;}

    public void annuler() {this.lampe.marche() ;}
}
```

*lampe.marche() restaure l'état  
de la lampe précédant l'appel à  
lampe.arrete() :*

# Exemple

20

```
public class Telecommande
{
    private Commande lampeOn, lampeOff ;
    private Commande commandeActivee ;

    public Telecommande() {}

    public void setLampeOn(Commande c) {this.lampeOn = c ;}
    public void setLampeOff(Commande c) {this.lampeOff = c ;}

    public void activeLampeOn()
    {
        this.lampeOn.executer() ;
        this.commandeActivee = this.lampeOn ;

    }

    public void activeLampeOff()
    {
        this.lampeOff.executer() ;
        this.commandeActivee = this.lampeOff ;

    }

    public void annulation()
    {this.commandeActivee.annuler() ;}
}
```

pour mémoriser la dernière commande exécutée

On mémorise la commande qui vient d'être exécutée

Si l'annulation est demandée, la dernière commande activée exécute son opération *annuler()*

# Exemple

21

```
public class Client
{
    public static void main(String[] args)
    {
        Telecommande t = new TeleCommande() ;

        Lampe        la = new Lampe() ;

        Commande      al = new AllumerLampe(la) ;
        Commande      el = new EteindreLampe(la) ;

        t.setLampeOn(al) ;
        t.setLampeOff(el) ;

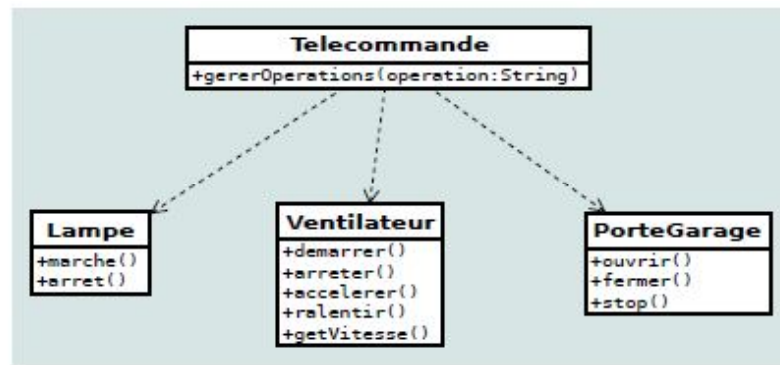
        t.activeLampeOn() ;
        t.activeLampeOff() ;

        t.annulation() ;
    }
}
```

annulation de la dernière *Commande* :  
la *Lampe* repasse à l'état "allumée"

# Mauvais exemple

22



```
public class Telecommande
{
    private Lampe l, Ventilateur v, PorteGarage p ;
    ...
    public void gererOperations(String operation)
    {
        if (operation.equals("allumerLampe")
            {l.allumer() ;}
        else if (operation.equals("eteindreVentilateur")
            {v.eteindreVentilateur() ;}
        ...
    }
}
```

il faut modifier la classe *Telecommande* dès qu'on ajoute/retire/modifie  
un appareil ou une requête

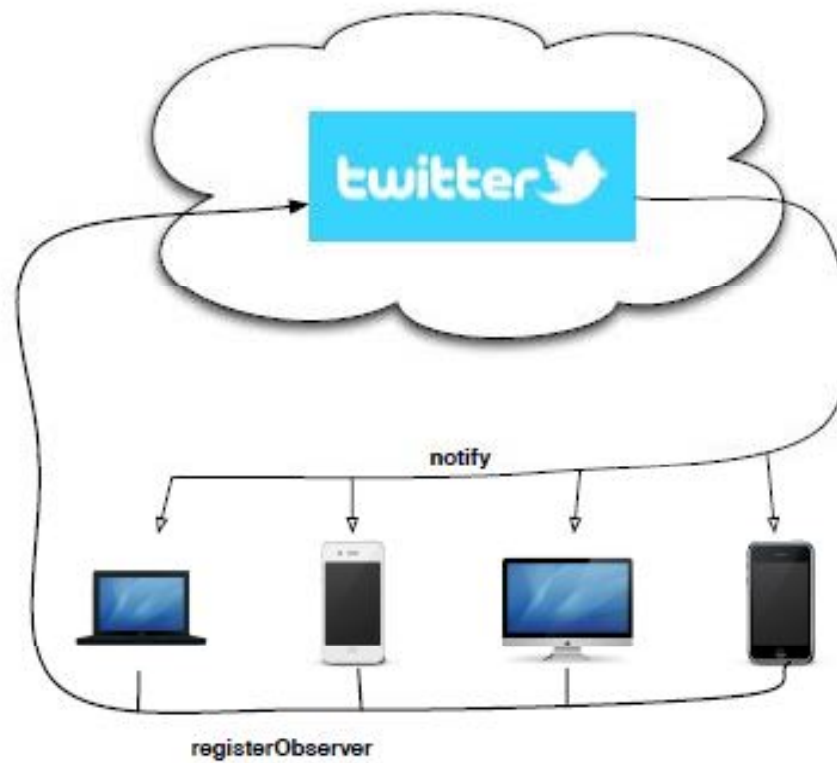
# Observer

23

- Problème : Comment faire savoir à un ensemble d'objets (observateur/abonne/observer/subscribe) qu'un autre objet (observé/sujet/publisher/observable) dont ils dépendent a été modifié?
- Définir une dépendance un-à-plusieurs (1- N) entre des objets de telle façon que si un objet change d'état tous les objets dépendants en soient notifiés et puissent se mettre à jour.
- Synonyme : publish/subscribe

# Exemple

24





# Champ d'application

25

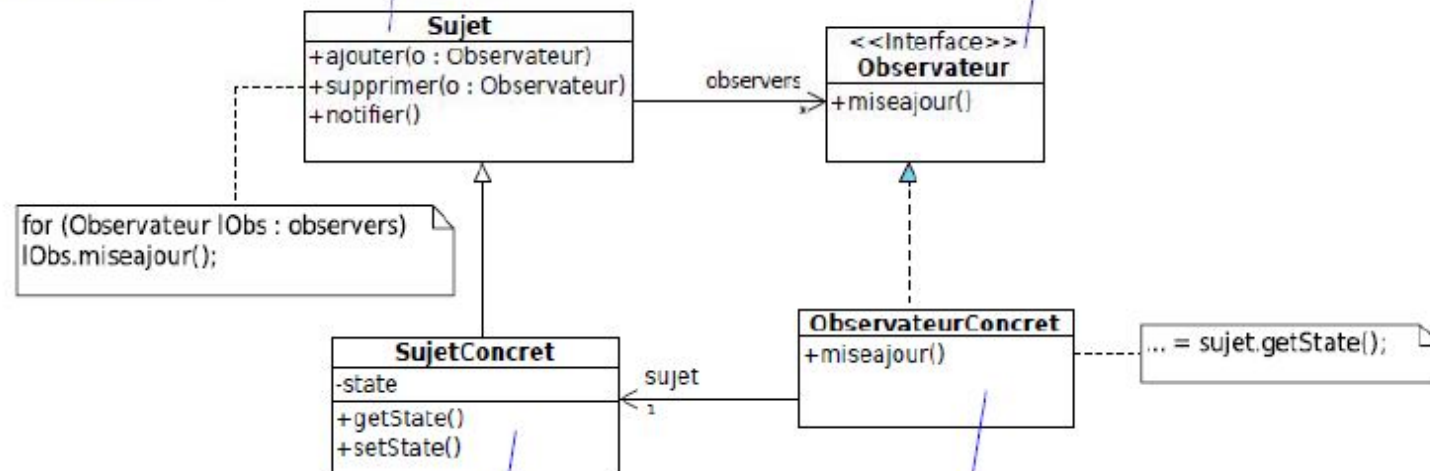
- Quand un changement sur un objet nécessite de modifier les autres et qu'on ne peut pas savoir à l'avance combien d'objets doivent être modifiés.
- Quand un objet doit être capable de notifier d'autres objets : les objets ne doivent donc pas être fortement couplés.

# Structure

26

connaît ses observateurs  
procure une interface permettant à  
un observateur de s'enregistrer  
pour être notifié

définit une interface de mise à jour  
pour les objets devant être notifiés



envoie une notification à ses  
observateurs quand son état change

maintient une référence sur l'objet  
devant être mis à jour

# Exemple

27

```
// Interface implémentée par tous les observateurs.  
public interface Observateur  
{  
    // Méthode appelée automatiquement lorsque l'état (position ou précision) du GPS change.  
    public void actualiser(Observable o);  
}  
// Interface implémentée par toutes les classes souhaitant avoir des observateurs à leur écoute.  
public interface Observable  
{  
    // Méthode permettant d'ajouter (abonner) un observateur.  
    public void ajouterObservateur(Observateur o);  
    // Méthode permettant de supprimer (résilier) un observateur.  
    public void supprimerObservateur(Observateur o);  
    // Méthode qui permet d'avertir tous les observateurs lors d'un changement d'état.  
    public void notifierObservateurs();  
}
```

# Exemple

28

```
// Classe représentant un GPS (appareil permettant de connaître sa position).
public class Gps implements Observable
{
    private String position;// Position du GPS.
    private int precision;// Précision accordé à cette position (suivant le nombre de satellites utilisés).
    private ArrayList tabObservateur;// Tableau d'observateurs.

    // Constructeur.
    public Gps()
    {
        position="inconnue";
        precision=0;
        tabObservateur=new ArrayList();
    }

    // Permet d'ajouter (abonner) un observateur à l'écoute du GPS.
    public void ajouterObservateur(Observateur o)
    {
        tabObservateur.add(o);
    }

    // Permet de supprimer (résilier) un observateur écoutant le GPS
    public void supprimerObservateur(Observateur o)
    {
        tabObservateur.remove(o);
    }

    // Méthode permettant de notifier tous les observateurs lors d'un changement d'état du GPS.
    public void notifierObservateurs()
    {
        for(int i=0;i<tabObservateur.size();i++)
        {
            Observateur o = tabObservateur.get(i);
            o.actualiser(this);// On utilise la méthode "tiré".
        }
    }
}
```

...

# Exemple

29

```
// Affiche un résumé en console des informations (position) du GPS.
public class AfficheResume implements Observateur
{
    // Méthode appelée automatiquement lors d'un changement d'état du GPS.
    public void actualiser(Observable o)
    {
        if(o instanceof Gps)
        {
            Gps g = (Gps) o;
            System.out.println("Position : "+g.getPosition());
        }
    }
}

// Affiche en console de façon complète les informations (position et précision) du GPS.
public class AfficheComplet implements Observateur
{
    // Méthode appelée automatiquement lors d'un changement d'état du GPS.
    public void actualiser(Observable o)
    {
        if(o instanceof Gps)
        {
            Gps g = (Gps) o;
            System.out.println("Position : "+g.getPosition()+" Précision : "+g.getPrecision()+"/10");
        }
    }
}
```

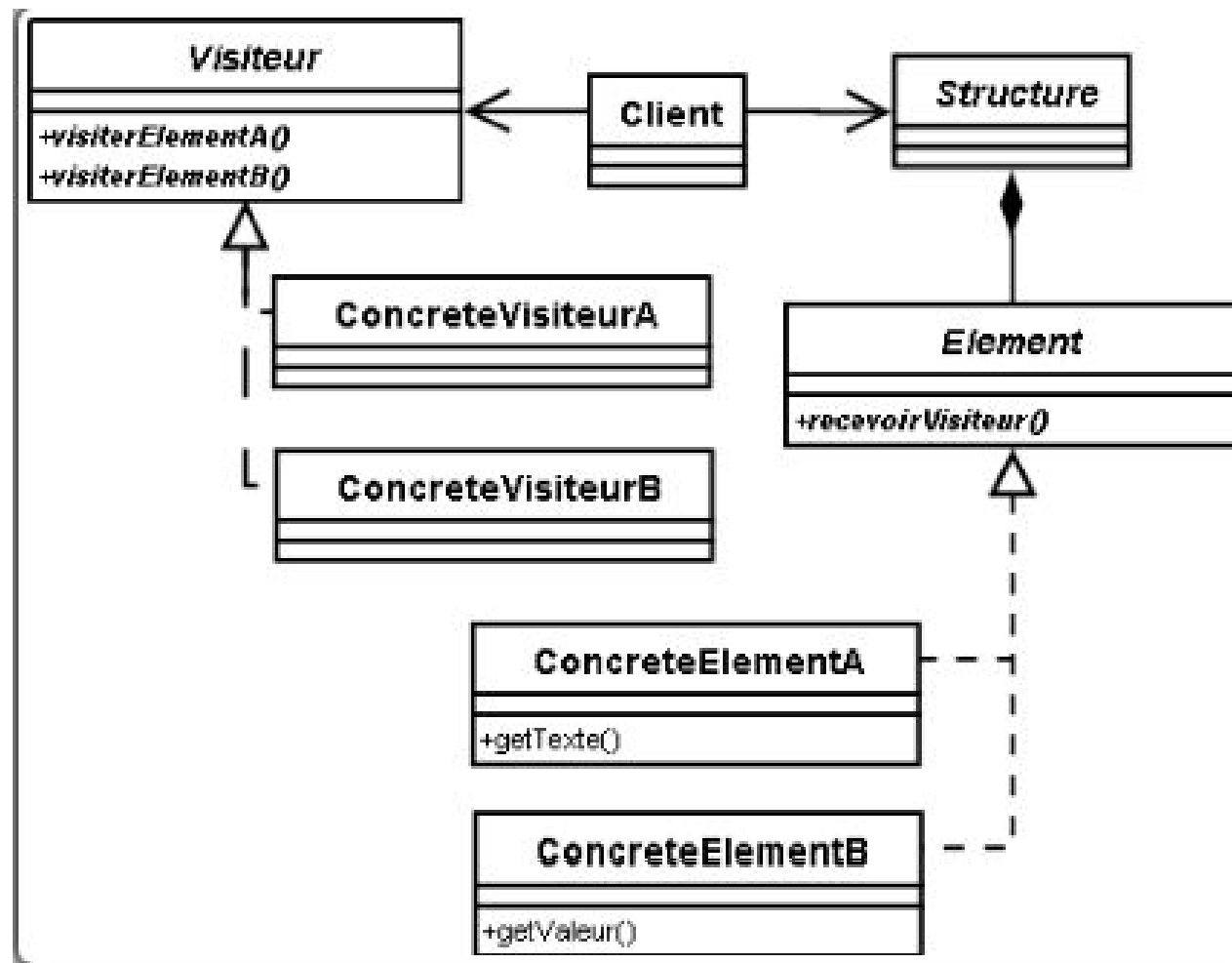
# Visitor

30

- ❑ Séparer un algorithme d'une structure de données.
- ❑ Il est nécessaire de réaliser des opérations sur les éléments d'un objet structuré
- ❑ Ces opérations varient en fonction de la nature de chaque élément et les opérations peuvent être de plusieurs types.

# Structure

31



# Rôles

32

- **Element** : définit l'interface d'un élément. Elle déclare la méthode de réception d'un objet **Visiteur**.
- **ConcreteElementA** et **ConcreteElementB** : sont des sous-classes concrètes de l'interface **Element**. Elles implémentent la méthode de réception. Elles possèdent des données/attributs et méthodes différents.
- **Visiteur** : définit l'interface d'un visiteur. Elle déclare les méthodes de visite des sous-classes concrètes de **Element**.
- **ConcreteVisiteurA** et **ConcreteVisiteurB** : sont des sous-classes concrètes de l'interface **Visiteur**. Elles implémentent des comportements de visite des **Element**.
- **Structure** : présente une interface de haut niveau permettant de visiter les objets **Element** la composant.
- La partie cliente appelle les méthodes de réception d'un **Visiteur** des **Element**.



# Exemple (les classes à visiter)

33

```
interface IVisitable {
    void accept(IVisitor visitor);
}

class Dog extends Mammal implements IVisitable
{
    public String breed = "chihuahua";

    public void accept(IVisitor visitor)
    {
        visitor.visit(this);
    }
}

class Human extends Mammal implements IVisitable
{
    public String gender = "male";

    public void accept(IVisitor visitor)
    {
        visitor.visit(this);
    }
}

class Book implements IVisitable
{
    public String color = "red";

    public void accept(IVisitor visitor)
    {
        visitor.visit(this);
    }
}
```

# Exemple (le visitor)

34

```
interface IVisitor {
    void visit(IVisitable o);
    void visit(Dog o);
    void visit(Human o);
    void visit(Book o);
}

class DebugVisitor implements IVisitor
{
    public void visit(Dog o)
    {
        System.out.println("Breed : " + o.breed);
    }

    public void visit(Human o)
    {
        System.out.println("Gender : " + o.gender);
    }

    public void visit(Book o)
    {
        System.out.println("Color : " + o.color);
    }

    public void visit(IVisitable o)
    {
        System.out.println("Not implemented yet");
    }
}
```

# Exemple (le client)

35

```
public class MainRun
{
    public static void main(String[] args)
    {
        DebugVisitor visitor = new DebugVisitor();

        Dog dog = new Dog();

        /* Display = Breed : chihuahua */
        dog.accept(visitor);

        Human human = new Human();

        /* Display = Gender : male */
        human.accept(visitor);

        Book book = new Book();

        /* Display = Color : red */
        book.accept(visitor);
    }
}
```

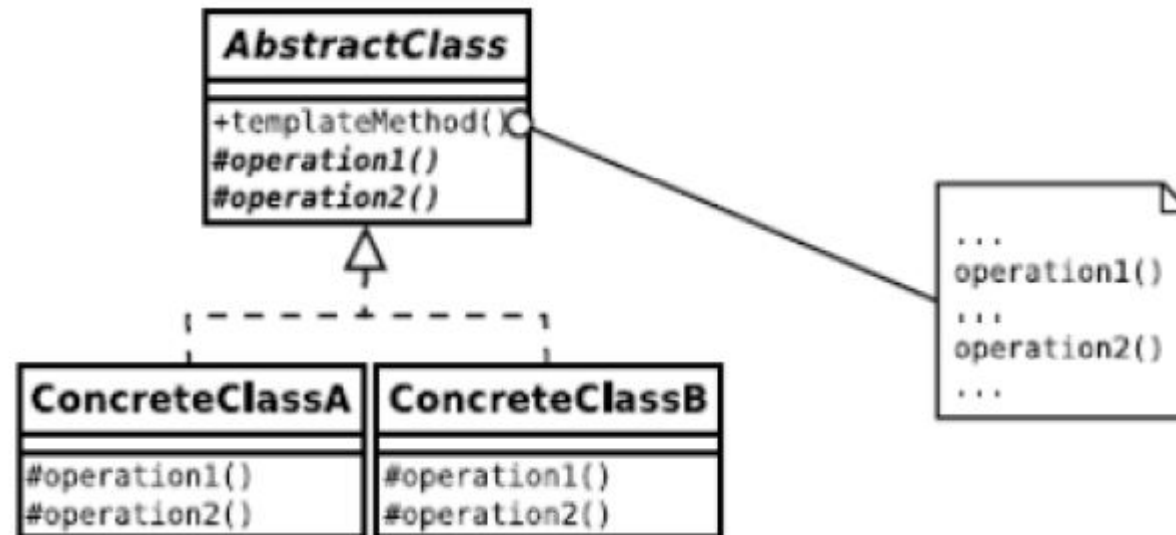
# Method template

36

- Définir le squelette d'un algorithme en déléguant certaines étapes à des sous-classes
- Une classe possède un fonctionnement global. Mais les détails de son algorithme doivent être spécifiques à ses sous-classes.

# Structure

37



# Rôles

38

- **AbstractClasse:** définit des méthodes abstraites primitives. La classe implémente le squelette d'un algorithme qui appelle les méthodes primitives.
- **ConcreteClasse:** est une sous-classe concrète de AbstractClasse. Elle implémente les méthodes utilisées par l'algorithme de la méthode TemplateMethod() de AbstractClasse.
- La partie cliente appelle la méthode de **AbstractClasse** qui définit l'algorithme.

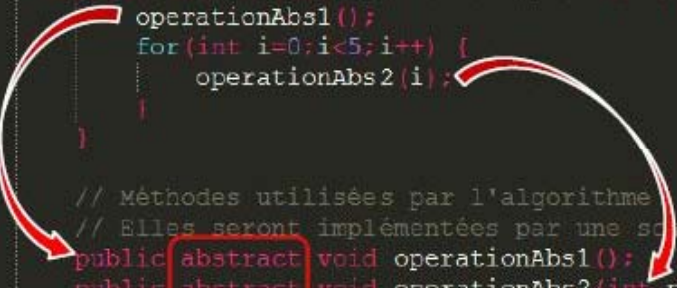
# Exemple

39

```
/* AbstractClasse.java */
public abstract class AbstractClasse {

    /**
     * Algorithme
     * La méthode est final afin que l'algorithme
     * ne puisse pas être redéfini par une classe fille
     */
    public final void operationTemplate() {
        operationAbs1();
        for(int i=0;i<5;i++) {
            operationAbs2(i);
        }
    }

    // Méthodes utilisées par l'algorithme
    // Elles seront implémentées par une sous-classe concrète
    public abstract void operationAbs1();
    public abstract void operationAbs2(int pNombre);
}
```



```
/* ConcreteClasse.java */
public class ConcreteClasse extends AbstractClasse {

    public void operationAbs1() {
        System.out.println("operationAbs1");
    }

    public void operationAbs2(int pNombre) {
        System.out.println("\toperationAbs2 : " + pNombre);
    }
}
```

```
/* MainClass.java */
public class MainClass {

    public static void main(String[] args) {
        // Création de l'instance
        AbstractClasse lClasse = new ConcreteClasse();
        // Appel de la méthode définie dans AbstractClasse
        lClasse.operationTemplate();
    }
}
```

# Strategy

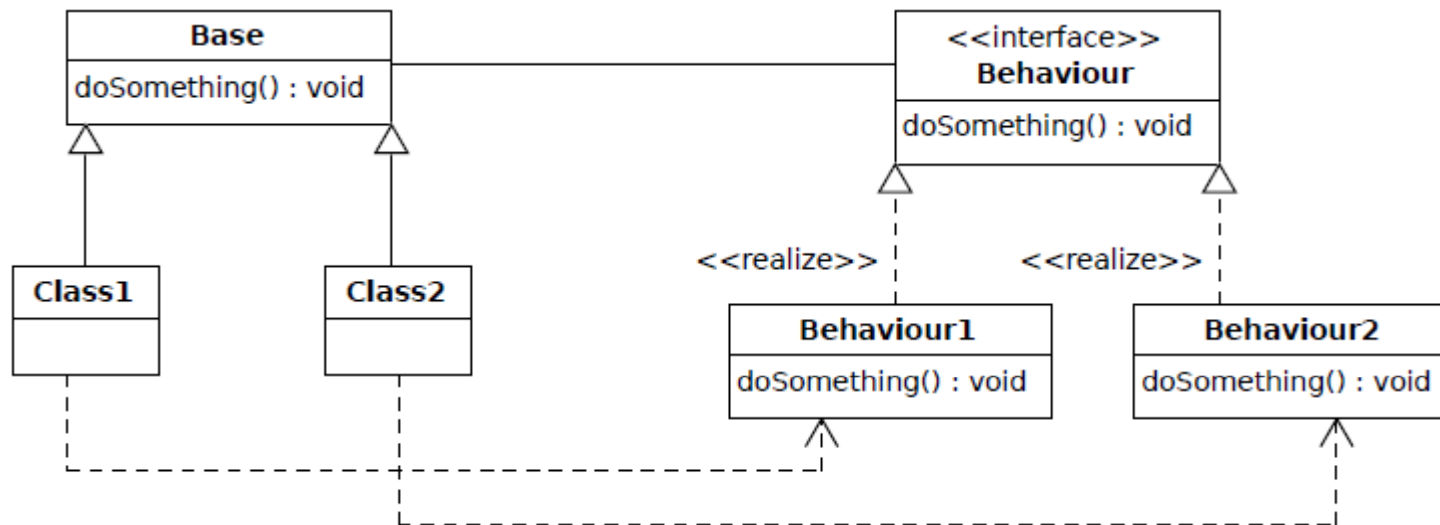
40

- Un objet doit pouvoir faire varier une partie de son algorithme.
- La partie de l'algorithme qui varie (le tri) est la stratégie. Toutes les stratégies présentent la même interface. La classe utilisant la stratégie (la liste) délègue la partie de traitement concernée à la stratégie.
- Chaque variation de comportement est une implémentation de l'interface
- Chaque classe qui a ce comportement référence une instance de la behaviour: changement dynamique possible de comportement!
- Ajout de nouveaux comportements: indolore



# Structure

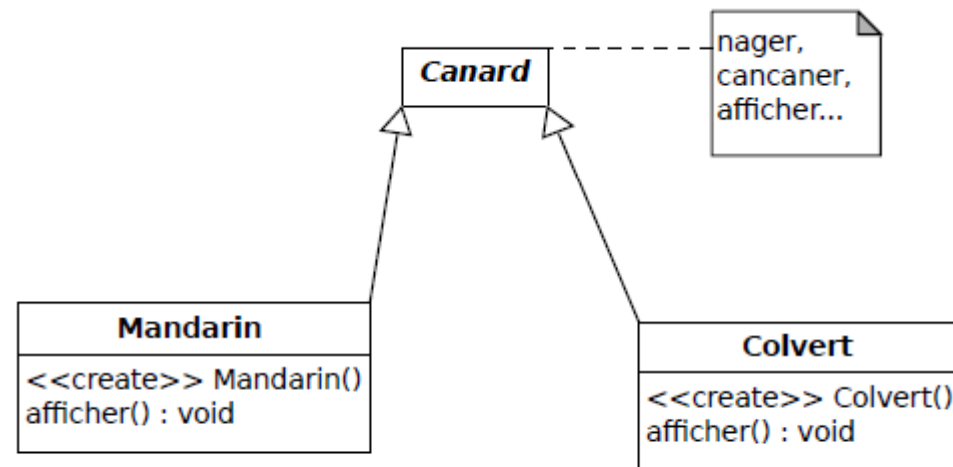
41



# Exemple

42

- Une application de jeu de simulation sur les canards
- Les canards nagent et crient:



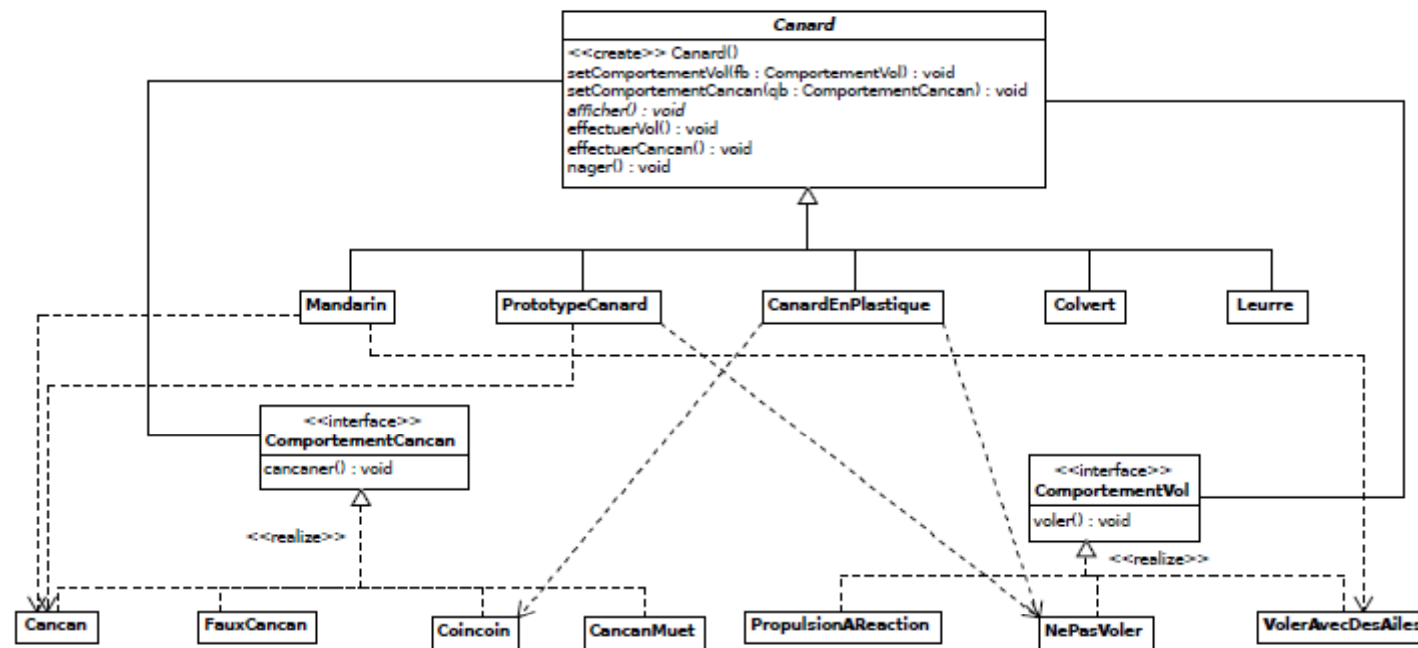
# Exemple

43

- ❑ Et si on veut faire voler les canards ?
- ❑ Simple: ajout d'une méthode voler() dans Canard ...
- ❑ ... Problème: les canards en plastique ne volent pas!
- ❑ Solution: redéfinir voler() dans CanardEnPlastique ?
- ❑ Non: le problème se pose dans d'autres classes (le canard lambda ne vole pas non plus)
- ❑ Il faut penser à séparer ce qui varie de ce qui demeure constant
- ❑ On va essayer d'encapsuler les parties variables hors du code stable

# Exemple

44



# Exemple

45

```
public abstract class Canard {
    ComportementVol comportementVol;
    ComportementCancan comportementCancan;

    public Canard() {
    }

    public void setComportementVol (ComportementVol fb) {
        comportementVol = fb;
    }

    public void setComportementCancan(ComportementCancan qb) {
        comportementCancan = qb;
    }

    abstract void afficher();

    public void effectuerVol() {
        comportementVol.voler();
    }
}
```

# Exemple

46

```
public void effectuerCancan() {  
    comportementCancan.cancaner();  
}  
  
public void nager() {  
    System.out.println("Tous les canards flottent, même les leurres!");  
}  
}
```

# Exemple

47

```
public class Mandarin extends Canard {  
  
    public Mandarin() {  
        comportementVol = new VolerAvecDesAiles();  
        comportementCancan = new Cancan();  
    }  
  
    public void afficher() {  
        System.out.println("Je suis un vrai mandarin");  
    }  
}
```

# Chaîne de responsabilités

48

- Le pattern Chain of Responsibility construit une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à son successeur et ainsi de suite jusqu'à ce que l'un des objets de la chaîne y réponde.
- Eviter le couplage entre l'expéditeur d'une requête et son destinataire en donnant à plusieurs objets la possibilité de traiter la requête.



# Chaîne de responsabilités

49

## □ Contexte

- ▣ Le système doit traiter une requête
- ▣ La requête peut être traitée de plusieurs façons (par plusieurs objets)

## □ Problème

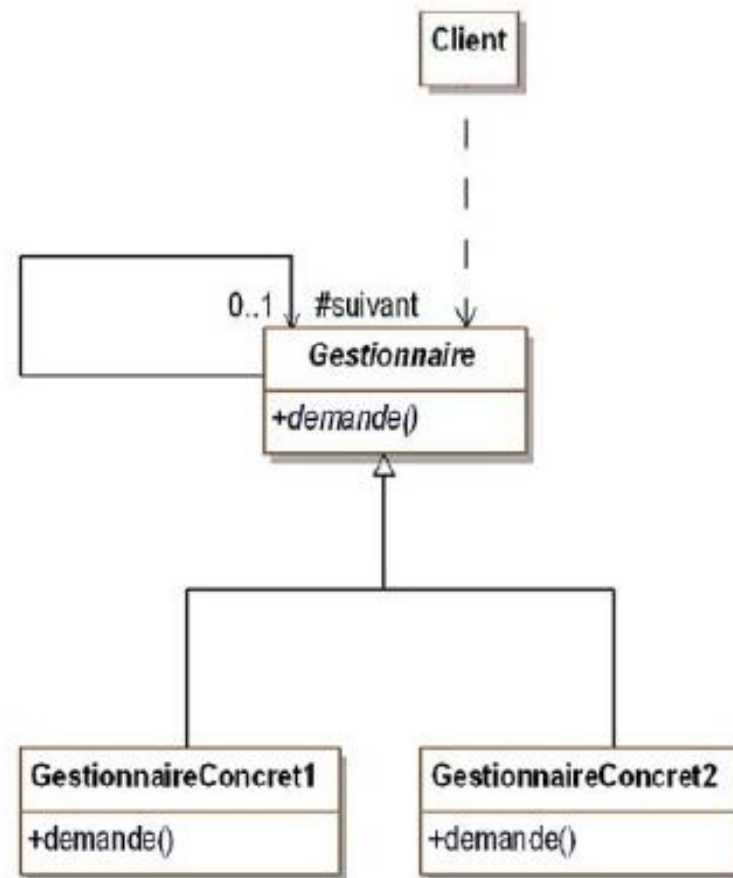
- ▣ Différents objets peuvent traiter une requête et on ne sait pas a priori lequel
- ▣ L'ensemble des objets pouvant traiter une requête doit être facilement modifiable

## □ Solution

- ▣ Isoler les différentes parties d'un traitement dans différents objets
- ▣ Faire passer la requête via une chaîne d'objets (maillons)
- ▣ Chaque maillon peut traiter la requête et/ou la faire passer au maillon suivant

# Chaîne de responsabilités

50



# Rôles

51

- Gestionnaire (ObjetBase) est une classe abstraite qui implante sous forme d'une association la chaîne de responsabilité ainsi que l'interface des requêtes ;
- GestionnaireConcret1 et GestionnaireConcret2 sont les classes concrètes qui implantent le traitement des requêtes en utilisant la chaîne de responsabilité si elles ne peuvent pas les traiter ;
- Client (Utilisateur) initie la requête initiale auprès d'un objet de l'une des classes GestionnaireConcret1 ou GestionnaireConcret2.