



Design Patterns du Gang of Four appliqués à Java

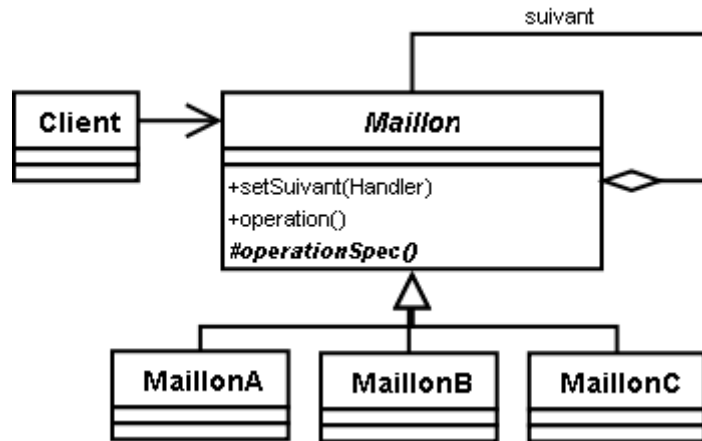


Table des matières

- VI. COMPORTEMENTAUX (BEHAVIORAL PATTERNS)
 - VI-A. Chaîne de responsabilité (Chain of responsibility)
 - VI-B. Commande (Command, Action ou Transaction)
 - VI-C. Interpréteur (Interpreter)
 - VI-D. Itérateur (Iterator ou Cursor)
 - VI-E. Médiateur (Mediator)
 - VI-F. Memento (Memento)
 - VI-G. Observateur (Observer, Dependents ou Publish-Subscribe)
 - VI-H. Etat (State ou Objects for States)
 - VI-I. Stratégie (Strategy ou Policy)
 - VI-J. Patron de méthode (Template Method)
 - VI-K. Visiteur (Visitor)

VI. COMPORTEMENTAUX (BEHAVIORAL PATTERNS) ▲

VI-A. Chaîne de responsabilité (Chain of responsibility) ▲

**OBJECTIFS :**

- Eviter le couplage entre l'émetteur d'une requête et son récepteur en donnant à plus d'un objet une chance de traiter la requête.
- Chaîner les objets récepteurs et passer la requête tout le long de la chaîne jusqu'à ce qu'un objet la traite.

RAISONS DE L'UTILISER :

Le système doit gérer une requête. La requête implique plusieurs objets pour la traiter.

Cela peut être le cas d'un système complexe d'habilitations possédant plusieurs critères afin d'autoriser l'accès. Ces critères peuvent varier en fonction de la configuration.

Le traitement est réparti sur plusieurs objets : les maillons. Les maillons sont chaînés. Si un maillon ne peut réaliser le traitement (vérification des droits), il donne sa chance au maillon suivant. Il est facile de faire varier les maillons impliqués dans le traitement.

RESULTAT :

Le Design Pattern permet d'isoler les différentes parties d'un traitement.

RESPONSABILITES :

- **Maillon** : définit l'interface d'un maillon de la chaîne. La classe implémente la gestion de la succession des maillons.
- **MaillonA**, **MaillonB** et **MaillonC** : sont des sous-classes concrètes qui définissent un maillon de la chaîne. Chaque maillon a la responsabilité d'une partie d'un traitement.
- La partie cliente appelle la méthode **operation()** du premier maillon de la chaîne.

IMPLEMENTATION JAVA :

Maillon.java
Sélectionnez

```
/**
 * Définit l'interface d'un maillon de la chaîne.
```

```

*/
public abstract class Maillon {

    private Maillon suivant;

    /**
     * Fixe le maillon suivant dans la chaine
     * @param pSuivant
     */
    public void setSuivant(Maillon pSuivant) {
        suivant = pSuivant;
    }

    /**
     * Appelle le traitement sur le maillon courant
     * Puis demande au maillon suivant d'en faire autant,
     * si le maillon courant n'a pas géré l'opération.
     * @param pNombre
     * @return Si l'opération a été gérée.
     */
    public boolean operation(int pNombre) {
        if(operationSpec(pNombre)) {
            return true;
        };

        if(suivant != null) {
            return suivant.operation(pNombre);
        }
        return false;
    }

    public abstract boolean operationSpec(int pNombre);
}

```

MaillonA.java
Sélectionnez

```

/**
 * Sous-classe concrète qui définit un maillon de la chaine.
 */
public class MaillonA extends Maillon {

    /**
     * Méthode affichant un message
     * si le nombre passé en paramètre est pair
     * @return true, si la maillon a géré l'opération
     */
    public boolean operationSpec(int pNombre) {
        if(pNombre % 2 == 0) {
            System.out.println("MaillonA : " + pNombre + " : pair");
            return true;
        }
    }
}

```

```
    }  
    return false;  
}  
}
```

MaillonB.java

Sélectionnez

```
/**  
 * Sous-classe concrète qui définit un maillon de la chaine.  
 */  
public class MaillonB extends Maillon {  
  
    /**  
     * Méthode affichant un message  
     * si le nombre passé en paramètre est inférieur à 2  
     * @return true, si la maillon a géré l'opération  
     */  
    public boolean operationSpec(int pNombre) {  
        if(pNombre < 2) {  
            System.out.println("MaillonB : " + pNombre + " : < 2");  
            return true;  
        }  
        return false;  
    }  
}
```

MaillonC.java

Sélectionnez

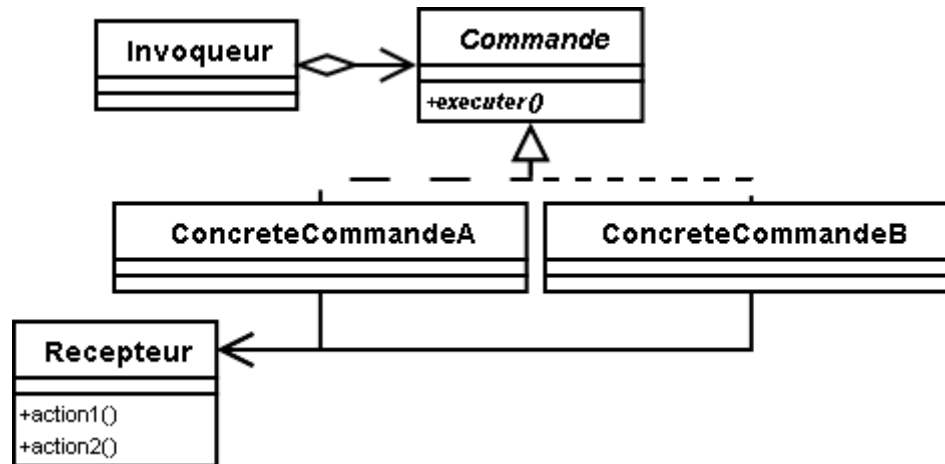
```
/**  
 * Sous-classe concrète qui définit un maillon de la chaine.  
 */  
public class MaillonC extends Maillon {  
  
    /**  
     * Méthode affichant un message  
     * si le nombre passé en paramètre est supérieur à 2  
     * @return true, si la maillon a géré l'opération  
     */  
    public boolean operationSpec(int pNombre) {  
        if(pNombre > 2) {  
            System.out.println("MaillonC : " + pNombre + " : > 2");  
            return true;  
        }  
        return false;  
    }  
}
```

ChainOfResponsibilityPatternMain.java

Sélectionnez

```
public class ChainOfResponsibilityPatternMain {  
  
    public static void main(String[] args) {  
        // Création des maillons  
        Maillon lMaillonA = new MaillonA();  
        Maillon lMaillonB = new MaillonB();  
        Maillon lMaillonC = new MaillonC();  
  
        // Définition de l'enchainement des maillons  
        lMaillonA.setSuivant(lMaillonB);  
        lMaillonB.setSuivant(lMaillonC);  
  
        // Appel de la méthode du premier maillon  
        // avec des valeurs différentes  
        System.out.println("--> Appel de la méthode avec paramètre '1' : ");  
        lMaillonA.operation(1);  
        System.out.println("--> Appel de la méthode avec paramètre '2' : ");  
        lMaillonA.operation(2);  
        System.out.println("--> Appel de la méthode avec paramètre '3' : ");  
        lMaillonA.operation(3);  
        System.out.println("--> Appel de la méthode avec paramètre '4' : ");  
        lMaillonA.operation(4);  
  
        // Affichage :  
        // --> Appel de la méthode avec paramètre '1' :  
        // MaillonB : 1 : < 2  
        // --> Appel de la méthode avec paramètre '2' :  
        // MaillonA : 2 : pair  
        // --> Appel de la méthode avec paramètre '3' :  
        // MaillonC : 3 : > 2  
        // --> Appel de la méthode avec paramètre '4' :  
        // MaillonA : 4 : pair  
    }  
}
```

VI-B. Commande (Command, Action ou Transaction) ▲



LIEN VERS LE DICTIONNAIRE DES DEVELOPPEURS :

Command

OBJECTIFS :

- Encapsuler une requête sous la forme d'objet.
- Paramétrer facilement des requêtes diverses.
- Permettre des opérations réversibles.

RAISONS DE L'UTILISER :

Le système doit traiter des requêtes. Ces requêtes peuvent provenir de plusieurs émetteurs. Plusieurs émetteurs peuvent produire la même requête. Les requêtes doivent pouvoir être annulées.

Cela peut être le cas d'une IHM avec des boutons de commande, des raccourcis clavier et des choix de menu aboutissant à la même requête.

La requête est encapsulée dans un objet : la commande. Chaque commande possède un objet qui traitera la requête : le récepteur. La commande ne réalise pas le traitement, elle est juste porteuse de la requête. Les émetteurs potentiels de la requête (éléments de l'IHM) sont des invoqueurs. Plusieurs invoqueurs peuvent se partager la même commande.

RESULTAT :

Le Design Pattern permet d'isoler une requête.

RESPONSABILITES :

- **Commande** : définit l'interface d'une commande.
- **ConcreteCommandA** et **ConcreteCommandB** : implémentent une commande. Chaque classe implémente la méthode **executer()**, en appelant des méthodes de l'objet **Recepteur**.
- **Invoqueur** : déclenche la commande. Il appelle la méthode **executer()** d'un objet **Commande**.
- **Recepteur** : reçoit la commande et réalise les opérations associées. Chaque objet **Commande** concret possède un lien avec un objet **Recepteur**.

- La partie cliente configure le lien entre les objets **Commande** et le **Recepteur**.

IMPLEMENTATION JAVA :

Commande.java
Sélectionnez

```
/**
 * Définit l'interface d'une commande
 */
public interface Commande {

    public void executer();

}
```

ConcreteCommandA.java
Sélectionnez

```
/**
 * Implémente une commande.
 * Appelle la méthode action1() lorsque la commande est exécutée.
 */
public class ConcreteCommandA implements Commande {

    private Recepteur recepteur;

    public ConcreteCommandA(Recepteur pRecepteur) {
        recepteur = pRecepteur;
    }

    public void executer() {
        recepteur.action1();
    }

}
```

ConcreteCommandB.java
Sélectionnez

```
/**
 * Implémente une commande.
 * Appelle la méthode action2() lorsque la commande est exécutée.
 */
public class ConcreteCommandB implements Commande {

    private Recepteur recepteur;

    public ConcreteCommandB(Recepteur pRecepteur) {
        recepteur = pRecepteur;
    }

    public void executer() {
```

```
        recepateur.action2();
    }
}
```

Invoqueur.java

Sélectionnez

```
/**
 * Déclenche les commandes.
 */
public class Invoqueur {

    // Références vers les commandes
    private Commande commandeA;
    private Commande commandeB;

    // Méthodes pour invoquer les commandes
    public void invoquerA() {
        if(commandeA != null) {
            commandeA.executer();
        }
    }

    public void invoquerB() {
        if(commandeB != null) {
            commandeB.executer();
        }
    }

    // Méthodes pour fixer les commandes
    public void setCommandeA(Commande pCommandeA) {
        commandeA = pCommandeA;
    }

    public void setCommandeB(Commande pCommandeB) {
        commandeB = pCommandeB;
    }
}
```

Recepteur.java

Sélectionnez

```
/**
 * Reçoit la commande.
 */
public class Recepteur {

    public void action1() {
        System.out.println("Traitement numero 1 effectué.");
    }
}
```



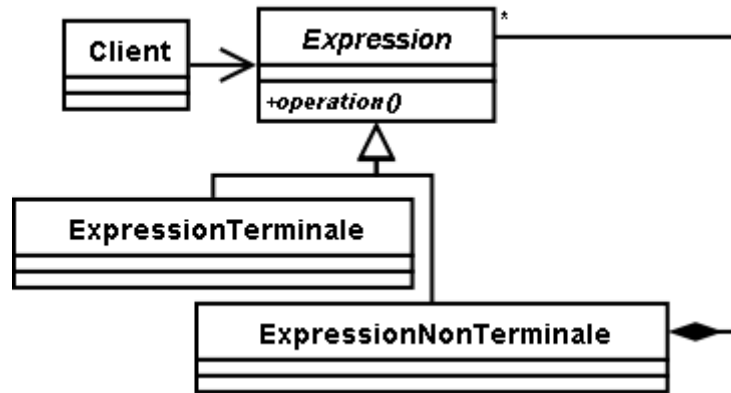
```
    public void action2() {  
        System.out.println("Traitement numero 2 effectué.");  
    }  
}
```

CommandPatternMain.java

Sélectionnez

```
public class CommandPatternMain {  
  
    public static void main(String[] args) {  
        // Création d'un récepteur  
        Recepteur lRecepteur = new Recepteur();  
  
        // Création des commandes  
        Commande lCommandeA = new ConcreteCommandA(lRecepteur);  
        Commande lCommandeB = new ConcreteCommandB(lRecepteur);  
  
        // Création et initialisation de l'invoqueur  
        Invoqueur lInvoqueur = new Invoqueur();  
        lInvoqueur.setCommandeA(lCommandeA);  
        lInvoqueur.setCommandeB(lCommandeB);  
  
        // Appel des méthodes d'invocation  
        // NB : Cette classe représente la partie cliente.  
        // Donc, normalement l'invocation  
        // ne se passe pas dans la partie cliente  
        // Dans l'exemple, elle est ici par souci de concision  
        lInvoqueur.invoquerA();  
        lInvoqueur.invoquerB();  
  
        // Affichage :  
        // Traitement numero 1 effectué.  
        // Traitement numero 2 effectué.  
    }  
}
```

VI-C. Interpréteur (Interpreter) ▲

**OBJECTIFS :**

- Définir une représentation de la grammaire d'un langage.
- Utiliser cette représentation pour interpréter les éléments de ce langage.

RAISONS DE L'UTILISER :

Le système doit interpréter un langage. Ce langage possède une grammaire prédéfinie qui constitue un ensemble d'opérations qui peuvent être effectuées par le système.

Cela peut être le cas d'un logiciel embarqué dont la configuration des écrans serait stockée dans des fichiers XML. Le logiciel lit ces fichiers afin de réaliser son affichage et l'enchaînement des écrans.

Une structure arborescente peut représenter la grammaire du langage. Elle permet d'interpréter les différents éléments du langage.

RESULTAT :

Le Design Pattern permet d'isoler les éléments d'un langage.

RESPONSABILITES :

- **Expression** : définit l'interface d'une expression.
- **ExpressionNonTerminale** : implémente une expression non terminale. Une expression non terminale peut contenir d'autres expressions.
- **ExpressionTerminale** : implémente une expression terminale. Une expression terminale ne peut pas contenir d'autres expressions.
- La partie cliente effectue des opérations sur la structure pour interpréter le langage représenté.

IMPLEMENTATION JAVA :

Expression.java
Sélectionnez

```

/**
 * Définit l'interface d'une expression
 */
public abstract class Expression {

```

```

protected static void afficherIndentation(int pIndentation) {
    for(int i=0;i<pIndentation;i++) {
        System.out.print("    ");
    }
}

public void operation() {
    operation(0);
}

public abstract void operation(int pIndentation);
}

```

ExpressionNonTerminale.java

Sélectionnez

```

/**
 * Implémente une expression non terminale.
 */
public class ExpressionNonTerminale extends Expression {

    private String libelle;
    private List<Expression> liste = new LinkedList<Expression>();

    /**
     * Constructeur permettant de fixer un attribut libelle
     * @param pTexte
     */
    public ExpressionNonTerminale(String pLibelle) {
        libelle = pLibelle;
    }

    /**
     * Permet d'ajouter des expressions a l'expression non terminale
     * @param pExpression
     */
    public void ajouterExpression(Expression pExpression) {
        liste.add(pExpression);
    }

    /**
     * Affiche l'attribut libelle sous forme de tag ouvrant et fermant
     * ainsi que les expressions contenues dans la liste
     * de l'expression non terminale
     */
    public void operation(int pIndentation) {
        afficherIndentation(pIndentation);
        System.out.println("<" + libelle + ">");
        Iterator<Expression> lIterator = liste.iterator();
        while(lIterator.hasNext()) {

```

```

        Expression lExpression = lIterator.next();
        lExpression.operation(pIndentation + 1);
    }
    afficherIndentation(pIndentation);
    System.out.println("</" + libelle + ">");
}
}

```

ExpressionTerminale.java
Sélectionnez

```

/**
 * Implémente une expression terminale.
 */
public class ExpressionTerminale extends Expression {

    private String texte;

    /**
     * Constructeur permettant de fixer un attribut texte
     * @param pTexte
     */
    public ExpressionTerminale(String pTexte) {
        texte = pTexte;
    }

    /**
     * Affiche l'attribut texte avec indentation
     */
    public void operation(int pIndentation) {
        afficherIndentation(pIndentation);
        System.out.println(texte);
    }
}

```

InterpreterPatternMain.java
Sélectionnez

```

public class InterpreterPatternMain {

    public static void main(String[] args) {
        // Création des expressions non terminales
        ExpressionNonTerminale lRacine =
            new ExpressionNonTerminale("RACINE");
        ExpressionNonTerminale lElement1 =
            new ExpressionNonTerminale("ELEMENT1");
        ExpressionNonTerminale lElement2 =
            new ExpressionNonTerminale("ELEMENT2");
        ExpressionNonTerminale lElement3 =
            new ExpressionNonTerminale("ELEMENT3");
    }
}

```

```

// Création des expressions terminales
ExpressionTerminale lTexte1 =
    new ExpressionTerminale("TEXTE1");
ExpressionTerminale lTexte2 =
    new ExpressionTerminale("TEXTE2");

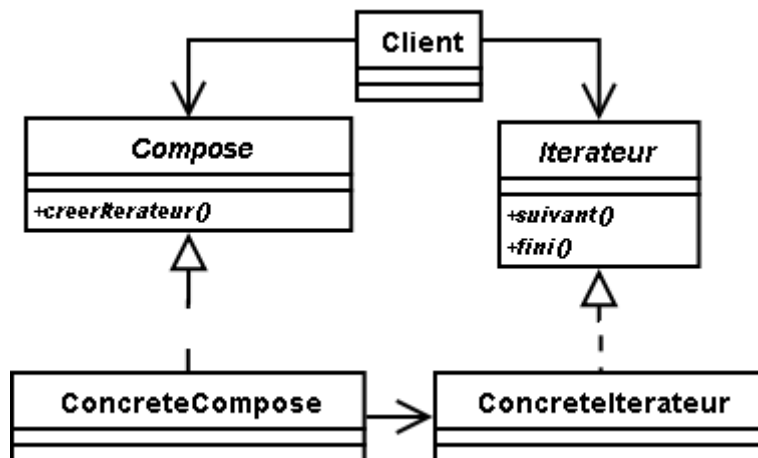
// Construit l'arborescence
lRacine.ajouterExpression(lElement1);
lRacine.ajouterExpression(lElement2);
lElement2.ajouterExpression(lElement3);
lElement1.ajouterExpression(lTexte1);
lElement3.ajouterExpression(lTexte2);

// Appel la méthode de l'expression racine
lRacine.operation();

// Affichage :
// <RACINE>
//   <ELEMENT1>
//     TEXTE1
//   </ELEMENT1>
//   <ELEMENT2>
//     <ELEMENT3>
//       TEXTE2
//     </ELEMENT3>
//   </ELEMENT2>
// </RACINE>
}
}

```

VI-D. Itérateur (Iterator ou Cursor) ▲



OBJECTIFS :

Fournir un moyen de parcourir séquentiellement les éléments d'un objet composé.

RAISONS DE L'UTILISER :

Le système doit parcourir les éléments d'un objet complexe. La classe de l'objet complexe peut varier.

Cela est le cas des classes représentant des listes et des ensembles en Java.

Les classes d'un objet complexe (listes) sont des "composés". Elles ont une méthode retournant un itérateur, qui permet de parcourir les éléments. Tous les itérateurs ont la même interface. Ainsi, le système dispose d'un moyen homogène de parcourir les composés.

RESULTAT :

Le Design Pattern permet d'isoler le parcours d'un agrégat.

RESPONSABILITES :

- **Compose** : définit l'interface d'un objet composé permettant de créer un **Iterateur**.
- **ConcreteCompose** : est une sous-classe de l'interface **Compose**. Elle est composée d'éléments et implémente la méthode de création d'un **Iterateur**.
- **Iterateur** : définit l'interface de l'itérateur, qui permet d'accéder aux éléments de l'objet **Compose**.
- **ConcreteIterateur** : est une sous-classe de l'interface **Iterateur**. Elle fournit une implémentation permettant de parcourir les éléments de **ConcreteCompose**. Elle conserve la trace de la position courante.
- La partie cliente demande à l'objet **Compose** de fournir un objet **Iterateur**. Puis, elle utilise l'objet **Iterateur** afin de parcourir les éléments de l'objet **Compose**.

IMPLEMENTATION JAVA :

Compose.java

Sélectionnez

```
/**
 * Définit l'interface d'un objet composé.
 */
public interface Compose {

    /**
     * Retourne un objet "Iterateur"
     */
    public Iterateur creerIterateur();
}
```

ConcreteCompose.java

Sélectionnez

```
/**
 * Sous-classe de l'interface "Compose".
 */
public class ConcreteCompose implements Compose {
```

```
// Elements composants l'objet "Compose"
private String[] elements = {
    "Bonjour" , "le", "monde"
};

/**
 * Retourne un objet "Iterateur" permettant
 * de parcourir les éléments
 */
public Iterateur creerIterateur() {
    return new ConcreteIterateur(elements);
}
}
```

Iterateur.java

Sélectionnez

```
/**
 * Définit l'interface de l'itérateur.
 */
public interface Iterateur {

    /**
     * Retourne l'élément suivant
     */
    public String suivant();

    /**
     * Retourne si l'itérateur
     * est arrivé sur le dernier élément
     */
    public boolean fini();
}
```

ConcreteIterateur.java

Sélectionnez

```
/**
 * Sous-classe de l'inteface "Iterateur".
 */
public class ConcreteIterateur implements Iterateur {

    private String[] elements;
    private int index = 0;

    public ConcreteIterateur(String[] pElements) {
        elements = pElements;
    }

    /**
     * Retourne l'élément
```

```
    * puis incrémente l'index
    */
    public String suivant() {
        return elements[index++];
    }

    /**
     * Si l'index est supérieur ou égal
     * à la taille du tableau,
     * on considère que l'on a fini
     * de parcourir les éléments
     */
    public boolean fini() {
        return index >= elements.length;
    }
}
```

IteratorPatternMain.java

Sélectionnez

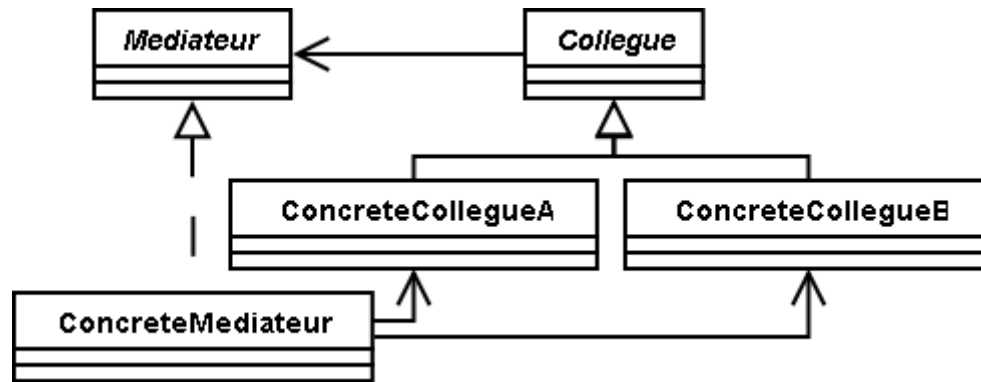
```
public class IteratorPatternMain {

    public static void main(String[] args) {
        // Création de l'objet "Compose"
        Compose lCompose = new ConcreteCompose();
        // Création de l'objet "Iterateur"
        Iterateur lIterateur = lCompose.creerIterateur();

        // Parcoure les éléments de l'objet "Compose"
        // grâce à l'objet "Iterateur"
        while(!lIterateur.fini()) {
            System.out.println(lIterateur.suivant());
        }

        // Affichage :
        // Bonjour
        // le
        // monde
    }
}
```

VI-E. Médiateur (Mediator) ▲

**OBJECTIFS :**

- Gérer la transmission d'informations entre des objets interagissant entre eux.
- Avoir un couplage faible entre les objets puisqu'ils n'ont pas de lien direct entre eux.
- Pouvoir varier leur interaction indépendamment.

RAISONS DE L'UTILISER :

Différents objets ont des interactions. Un événement sur l'un provoque une action ou des actions sur un autre ou d'autres objets.

Cela peut être les éléments d'IHM. Si une case est cochée, certains éléments deviennent accessibles. Si une autre case est cochée, des couleurs de l'IHM changent.

Si les classes communiquent directement, il y a un couplage très fort entre elles. Une classe dédiée à la communication permet d'éviter cela. Chaque élément interagissant (élément de l'IHM) sont des collègues. La classe dédiée à la communication est un médiateur.

RESULTAT :

Le Design Pattern permet d'isoler la communication entre des objets.

RESPONSABILITES :

- **Colleague** : définit l'interface d'un collègue. Il s'agit d'une famille d'objets qui s'ignorent entre eux mais qui doivent se transmettre des informations.
- **ConcreteColleagueA** et **ConcreteColleagueB** : sont des sous-classes concrètes de l'interface **Colleague**. Elles ont une référence sur un objet **Mediateur** auquel elles transmettront les informations.
- **Mediateur** : définit l'interface de communication entre les objets **Colleague**.
- **ConcreteMediateur** : implémente la communication et maintient une référence sur les objets **Colleague**.

IMPLEMENTATION JAVA :

Colleague.java
Sélectionnez

```

/**
 * Définit l'interface d'un collègue.
 */

```

```
public abstract class Colleague {  
    protected Mediateur mediateur;  
  
    /**  
     * Constructeur permettant de fixer le médiateur  
     * @param pMediateur  
     */  
    public Colleague(Mediateur pMediateur) {  
        mediateur = pMediateur;  
    }  
  
    public abstract void recevoirMessage(String pMessage);  
}
```

ConcreteColleagueA.java

Sélectionnez

```
/**  
 * Sous-classe concrète de "Colleague"  
 */  
public class ConcreteColleagueA extends Colleague {  
  
    public ConcreteColleagueA(Mediateur pMediateur) {  
        super(pMediateur);  
        pMediateur.setColleagueA(this);  
    }  
  
    /**  
     * Méthode demandant de transmettre un message  
     * provenant de cette classe  
     * @param pMessage  
     */  
    public void envoyerMessageFromA(String pMessage) {  
        mediateur.transmettreMessageFromA(pMessage);  
    }  
  
    /**  
     * Méthode recevant un message  
     */  
    public void recevoirMessage(String pMessage) {  
        System.out.println("ConcreteColleagueA a reçu : " + pMessage);  
    }  
}
```

ConcreteColleagueB.java

Sélectionnez

```
/**  
 * Sous-classe concrète de "Colleague"  
 */
```

```

public class ConcreteColleagueB extends Colleague {

    public ConcreteColleagueB(Mediateur pMediateur) {
        super(pMediateur);
        pMediateur.setColleagueB(this);
    }

    /**
     * Méthode demandant de transmettre un message
     * provenant de cette classe
     * @param pMessage
     */
    public void envoyerMessageFromB(String pMessage) {
        mediateur.transmettreMessageFromB(pMessage);
    }

    /**
     * Méthode recevant un message
     */
    public void recevoirMessage(String pMessage) {
        System.out.println("ConcreteColleagueB a reçu : " + pMessage);
    }
}

```

Mediateur.java

Sélectionnez

```

/**
 * Définit l'interface d'un médiateur.
 * Réalise la transmission de l'information.
 */
public interface Mediateur {

    // Méthodes permettant les collègues
    public void setColleagueA(ConcreteColleagueA pColleagueA);
    public void setColleagueB(ConcreteColleagueB pColleagueB);

    // Méthodes permettant de transmettre des messages
    public void transmettreMessageFromA(String pMessage);
    public void transmettreMessageFromB(String pMessage);
}

```

ConcreteMediateur.java

Sélectionnez

```

/**
 * Sous-classe concrète de Mediateur
 */
public class ConcreteMediateur implements Mediateur {

    private ConcreteColleagueA colleagueA;

```

```

private ConcreteColleagueB colleagueB;

// Méthodes permettant les collègues
public void setColleagueA(ConcreteColleagueA pColleagueA) {
    colleagueA = pColleagueA;
}

public void setColleagueB(ConcreteColleagueB pColleagueB) {
    colleagueB = pColleagueB;
}

/**
 * Si le message provient de A, on le transmet à B
 */
public void transmettreMessageFromA(String pMessage) {
    colleagueB.recevoirMessage(pMessage);
}

/**
 * Si le message provient de B, on le transmet à A
 */
public void transmettreMessageFromB(String pMessage) {
    colleagueA.recevoirMessage(pMessage);
}
}

```

MediatorPatternMain.java

Sélectionnez

```

public class MediatorPatternMain {

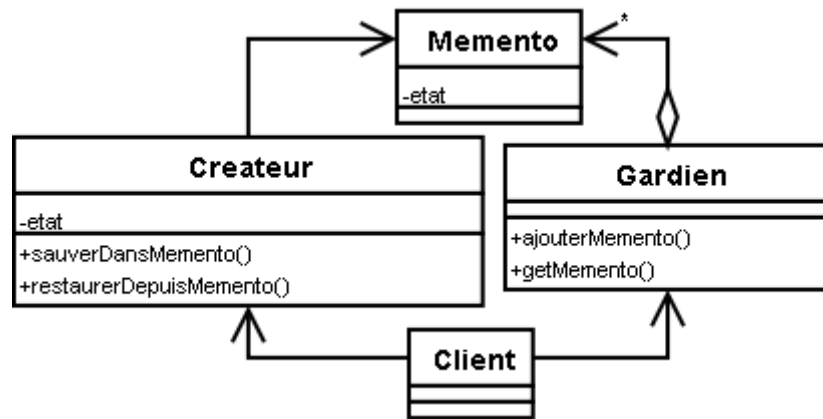
    public static void main(String[] args) {
        // Création du médiateur
        Mediateur lMediateur = new ConcreteMediateur();
        // Création des collègues
        ConcreteColleagueA lColleagueA = new ConcreteColleagueA(lMediateur);
        ConcreteColleagueB lColleagueB = new ConcreteColleagueB(lMediateur);

        // Déclenchement de méthodes qui demande
        // au médiateur de transmettre un message
        lColleagueA.envoyerMessageFromA("Coucou");
        lColleagueB.envoyerMessageFromB("Salut");

        // Affichage :
        // ConcreteColleagueB a reçu : Coucou
        // ConcreteColleagueA a reçu : Salut
    }
}

```

VI-F. Memento (Memento) ▲

**OBJECTIFS :**

Sauvegarder l'état interne d'un objet en respectant l'encapsulation, afin de le restaurer plus tard.

RAISONS DE L'UTILISER :

Un système doit conserver et restaurer l'état d'un objet. L'état interne de l'objet à conserver n'est pas visible par les autres objets.

Cela peut être un éditeur de document disposant d'une fonction d'annulation. La fonction d'annulation est sur plusieurs niveaux.

Les informations de l'état interne (état du document) sont conservées dans un memento. L'objet avec l'état interne (document) est le créateur du memento. Afin de respecter l'encapsulation, les valeurs du memento ne sont visibles que par son créateur. Ainsi, l'encapsulation de l'état interne est préservée. Un autre objet est chargé de conserver les mementos (gestionnaire d'annulation) : il s'agit du gardien.

RESULTAT :

Le Design Pattern permet d'isoler la conservation de l'état d'un objet.

RESPONSABILITES :

- **Memento** : contient la sauvegarde de l'état d'un objet. La classe doit autoriser l'accès aux informations seulement au **Createur**.
- **Createur** : sauvegarde son état dans un **Memento** ou restitue son état depuis un **Memento**.
- **Gardien** : conserve les **Memento** ou retourne un **Memento** conservé.
- La partie cliente demande au **Createur** de stocker son état dans un **Memento**. Elle demande au **Gardien** de conserver ce **Memento**. Elle peut alors demander au **Gardien** de lui fournir un des **Memento** conservés, ou bien elle demande au **Createur** de restituer son état depuis le **Memento**.

IMPLEMENTATION JAVA :

Createur.java
Sélectionnez

```

/**
 * Objet dont on souhaite conserver l'état.
 */

```

```
public class Createur {  
  
    // Etat à sauvegarder  
    private int etat = 2;  
  
    /**  
     * Contient la sauvegarde d'un état d'un objet.  
     * Ses méthodes sont privées, afin que seul le "Createur"  
     * accède aux informations stockées  
     */  
    public class Memento {  
  
        // Etat sauvegardé  
        private int etat;  
  
        private Memento(int pEtat) {  
            etat = pEtat;  
        }  
  
        /**  
         * Retourne l'état sauvegardé  
         * @return  
         */  
        private int getEtat() {  
            return etat;  
        }  
    }  
  
    /**  
     * Fait varier l'état de l'objet  
     */  
    public void suivant() {  
        etat = etat * etat;  
    }  
  
    /**  
     * Sauvegarde son état dans un "Memento"  
     * @return  
     */  
    public Memento sauverDansMemento() {  
        return new Memento(etat);  
    }  
  
    /**  
     * Restitue son état depuis un "Memento"  
     * @param pMemento  
     */  
    public void restaurerDepuisMemento(Memento pMemento) {  
        etat = pMemento.getEtat();  
    }  
}
```

```

/**
 * Affiche l'état de l'objet
 */
public void afficher() {
    System.out.println("L'etat vaut : " + etat);
}
}

```

Gardien.java
Sélectionnez

```

/**
 * Conserve les "Memento".
 * Retourne un "Memento" conservé
 */
public class Gardien {

    // Liste de "Memento"
    private List<Createur.Memento> liste = new LinkedList<Createur.Memento>();

    /**
     * Ajouter un "Memento" à la liste
     * @param pMemento
     */
    public void ajouterMemento(Createur.Memento pMemento) {
        liste.add(pMemento);
    }

    /**
     * Retourne le "Memento" correspondant à l'index
     * @param pIndex
     * @return
     */
    public Createur.Memento getMemento(int pIndex) {
        return liste.get(pIndex);
    }
}

```

MementoPatternMain.java
Sélectionnez

```

public class MementoPatternMain {

    public static void main(String[] args) {
        // Création du gardien
        Gardien lGardien = new Gardien();
        // Création du créateur
        Createur lCreateur = new Createur();

        // Sauvegarde l'état (2) dans le "Gardien" par le "Memento"
        lGardien.ajouterMemento(lCreateur.sauverDansMemento());
    }
}

```

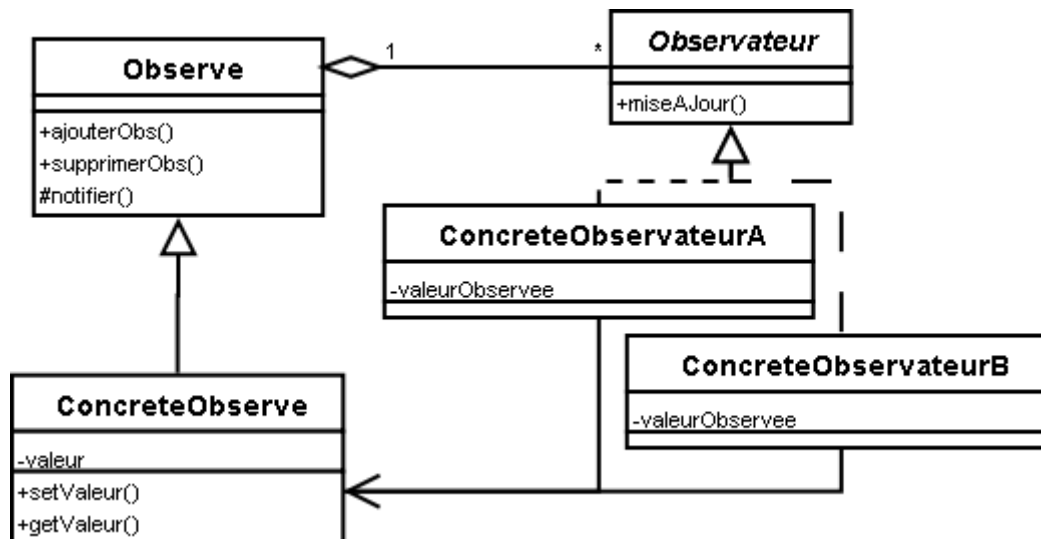
```

// Affiche l'état (2)
lCreateur.afficher();
// Change l'état (2 * 2 = 4)
lCreateur.suivant();
// Sauvegarde l'état (4) dans le "Gardien" par le "Memento"
lGardien.ajouterMemento(lCreateur.sauverDansMemento());
// Change l'état (4 * 4 = 16)
lCreateur.suivant();
// Sauvegarde l'état (16) dans le "Gardien" par le "Memento"
lGardien.ajouterMemento(lCreateur.sauverDansMemento());
// Affiche l'état (16)
lCreateur.afficher();
// Récupère l'état (4) de l'index 1 depuis le "Gardien"
Createur.Memento lMemento1 = lGardien.getMemento(1);
// Restaure l'état depuis le "Memento"
lCreateur.restaurerDepuisMemento(lMemento1);
// Affiche l'état (4)
lCreateur.afficher();

// Affichage :
// L'etat vaut : 2
// L'etat vaut : 16
// L'etat vaut : 4
}
}

```

VI-G. Observateur (Observer, Dependents ou Publish-Subscribe) ▲



OBJECTIFS :

Prévenir des objets observateurs, enregistrés auprès d'un objet observé, d'un événement.

RAISONS DE L'UTILISER :

Un objet doit connaître les changements d'état d'un autre objet. L'objet doit être informé immédiatement.

Cela peut être le cas d'un tableau affichant des statistiques. Si une nouvelle donnée est entrée, les statistiques sont recalculées. Le tableau doit être informé du changement, afin qu'il soit rafraîchi.

L'objet devant connaître le changement (le tableau) est un observateur. Il s'enregistre en tant que tel auprès de l'objet dont l'état change. L'objet dont l'état change (les statistiques) est un "observe". Il informe ses observateurs en cas d'événement.

RESULTAT :

Le Design Pattern permet d'isoler un algorithme traitant un événement.

RESPONSABILITES :

- **Observe** : est l'interface de l'objet à observer. Il possède une liste d'objets **Observateur**. Il fournit des méthodes pour ajouter ou supprimer des objets **Observateur** à la liste. Il fournit également une méthode pour avertir les objets **Observateur**.
- **ConcreteObserve** : est l'implémentation de l'objet à observer. Lorsqu'une valeur est modifiée, la méthode **notifier()** de la classe **Observe** est appelée.
- **Observateur** : définit l'interface de l'observateur. Il déclare la/les méthode(s) que l'objet **Observe** appelle en cas d'événements.
- **ConcreteObservateurA** et **ConcreteObservateurB** : sont des sous-classes concrètes de **Observateur**. Ils implémentent des comportements de mise à jour en cas d'événements.
- La partie cliente indique à l'objet **Observe** les objets **Observateur** qu'il avertira.

IMPLEMENTATION JAVA :

Observe.java

Sélectionnez

```
/**
 * Interface d'objet observé
 */
public class Observe {

    // Liste des observateurs
    private List<Observateur> listeObservateurs =
        new LinkedList<Observateur>();

    /**
     * Ajouter un observateur de la liste
     * @param pObservateur
     */
    public void ajouterObs(Observateur pObservateur) {
        listeObservateurs.add(pObservateur);
    }

    /**
     * Supprimer un observateur de la liste
     */
}
```

```

    * @param pObservateur
    */
    public void supprimerObs(Observateur pObservateur) {
        listeObservateurs.remove(pObservateur);
    }

    /**
     * Notifier à tous les observateurs de la liste
     * que l'objet à été mis à jour.
     */
    protected void notifier() {
        for(Observateur lObservateur : listeObservateurs) {
            lObservateur.miseAJour();
        }
    }
}

```

ConcreteObserve.java

Sélectionnez

```

/**
 * Implémentation d'un objet observé
 */
public class ConcreteObserve extends Observe {

    int valeur = 0;

    /**
     * Modifie une valeur de l'objet
     * et notifie la nouvelle valeur
     * @param pValeur
     */
    public void setValeur(int pValeur) {
        valeur = pValeur;
        notifier();
    }

    /**
     * Retourne la valeur de l'objet
     * @return La valeur
     */
    public int getValeur() {
        return valeur;
    }
}

```

Observateur.java

Sélectionnez

```

/**
 * Définit l'interface d'un observateur

```

```
*/  
public interface Observateur {  
  
    /**  
     * Méthode appelée par l'objet observé  
     * pour notifier une mise à jour  
     */  
    public void miseAJour();  
}
```

ConcreteObservateurA.java

Sélectionnez

```
/**  
 * Sous-classe concrète de "Observateur"  
 */  
public class ConcreteObservateurA implements Observateur {  
  
    private int valeur = 0;  
    private ConcreteObserve observe;  
  
    /**  
     * Fixe l'objet observé  
     * @param pObserve  
     */  
    public void setObserve(ConcreteObserve pObserve) {  
        observe = pObserve;  
    }  
  
    /**  
     * Méthode appelée par l'objet observé  
     * pour notifier une mise à jour  
     */  
    public void miseAJour() {  
        valeur = observe.getValeur();  
    }  
  
    /**  
     * Affiche la valeur  
     */  
    public void afficher() {  
        System.out.println("ConcreteObservateurA contient " + valeur);  
    }  
}
```

ConcreteObservateurB.java

Sélectionnez

```
/**  
 * Sous-classe concrète de "Observateur"  
 */
```

```

public class ConcreteObservateurB implements Observateur {

    private int valeur = 0;
    private ConcreteObserve observe;

    /**
     * Fixe l'objet observé
     * @param pObserve
     */
    public void setObserve(ConcreteObserve pObserve) {
        observe = pObserve;
    }

    /**
     * Méthode appelée par l'objet observé
     * pour notifier une mise à jour
     */
    public void miseAJour() {
        valeur = observe.getValeur();
    }

    /**
     * Affiche la valeur
     */
    public void afficher() {
        System.out.println("ConcreteObservateurB contient " + valeur);
    }
}

```

ObserverPatternMain.java

Sélectionnez

```

public class ObserverPatternMain {

    public static void main(String[] args) {
        // Création de l'objet observé
        ConcreteObserve lObserve = new ConcreteObserve();

        // Création de 2 observateurs
        ConcreteObservateurA lConcreteObservateurA = new ConcreteObservateurA();
        ConcreteObservateurB lConcreteObservateurB = new ConcreteObservateurB();

        // Ajout des observateurs
        // à la liste des observateurs
        // de l'objet observé
        lObserve.ajouterObs(lConcreteObservateurA);
        lObserve.ajouterObs(lConcreteObservateurB);

        // Fixe l'objet observé des observateurs
        lConcreteObservateurA.setObserve(lObserve);
        lConcreteObservateurB.setObserve(lObserve);
    }
}

```

```

// Affiche l'état des deux observateurs
lConcreteObservateurA.afficher();
lConcreteObservateurB.afficher();

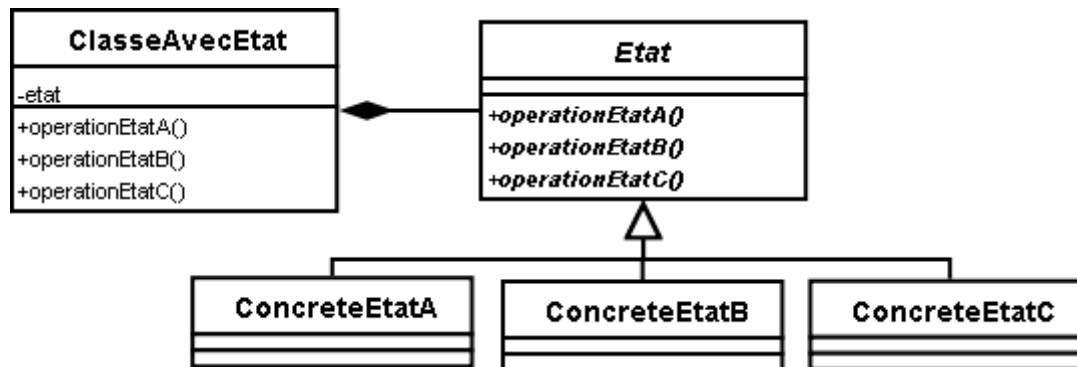
// Appel d'une méthode de mise à jour
// de l'objet observé
lObserve.setValeur(4);

// Affiche l'état des deux observateurs
lConcreteObservateurA.afficher();
lConcreteObservateurB.afficher();

// Affichage :
// ConcreteObservateurA contient 0
// ConcreteObservateurB contient 0
// ConcreteObservateurA contient 4
// ConcreteObservateurB contient 4
}
}

```

VI-H. Etat (State ou Objects for States) ▲



AUTRES RESSOURCES SUR DEVELOPPEZ.COM :

L'état par Sébastien MERIC

Améliorez vos logiciels avec le pattern Etat par Pierre Caboche

OBJECTIFS :

Changer le comportement d'un objet selon son état interne.

RAISONS DE L'UTILISER :

Un objet a un fonctionnement différent selon son état interne. Son état change selon les méthodes appelées.

Cela peut être un document informatique. Il a comme fonctions ouvrir, modifier, sauvegarder ou fermer. Le comportement de ces méthodes change selon l'état du document.

Les différents états internes sont chacun représenté par une classe état (ouvert, modifié, sauvegardé et fermé). Les états possèdent des méthodes permettant de réaliser les opérations et de changer d'état (ouvrir, modifier, sauvegarder et fermer). Certains états bloquent certaines opérations (modifier dans l'état fermé). L'objet avec état (document informatique) maintient une référence vers l'état actuel. Il présente les opérations à la partie cliente.

RESULTAT :

Le Design Pattern permet d'isoler les algorithmes propres à chaque état d'un objet.

RESPONSABILITES :

- **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**.

IMPLEMENTATION JAVA :

ClasseAvecEtat.java

Sélectionnez

```
/**
 * Classe avec état.
 * Son comportement change en fonction de son état.
 */
public class ClasseAvecEtat {

    private Etat etat;

    /**
     * Définit l'interface d'un état
     */
    public static abstract class Etat {

        /**
         * Méthode protégée permettant de changer l'état
         * @param pClasse
         * @param pEtat
         */
        protected void setEtat(ClasseAvecEtat pClasse, Etat pEtat) {
            pClasse.etat = pEtat;
        }

        // Méthodes pour changer d'état
    }
}
```

```

    public abstract void operationEtatA(ClasseAvecEtat pClasse);
    public abstract void operationEtatB(ClasseAvecEtat pClasse);
    public abstract void operationEtatC(ClasseAvecEtat pClasse);

    // Affichage de l'état courant
    public abstract void afficher();
}

/**
 * Constructeur avec initialisation à l'état A
 */
public ClasseAvecEtat() {
    etat = new ConcreteEtatA();
}

// Méthodes pour changer d'état
public void operationEtatA() {
    etat.operationEtatA(this);
}

public void operationEtatB() {
    etat.operationEtatB(this);
}

public void operationEtatC() {
    etat.operationEtatC(this);
}

/**
 * Affichage de l'état courant
 */
public void afficher() {
    etat.afficher();
}
}

```

ConcreteEtatA.java
Sélectionnez

```

/**
 * Sous-classe concrète de l'interface "Etat"
 * On peut passer de l'état A vers l'état B ou l'état C
 */
public class ConcreteEtatA extends ClasseAvecEtat.Etat {

    // Méthodes pour changer d'état
    //
    public void operationEtatA(ClasseAvecEtat pClasse) {
        System.out.println("Classe déjà dans l'état A");
    }
}

```

```

    public void operationEtatB(ClasseAvecEtat pClasse) {
        setEtat(pClasse, new ConcreteEtatB());
        System.out.println("Etat changé : A -> B");
    }

    public void operationEtatC(ClasseAvecEtat pClasse) {
        setEtat(pClasse, new ConcreteEtatC());
        System.out.println("Etat changé : A -> C");
    }

    /**
     * Affichage de l'état courant
     */
    public void afficher() {
        System.out.println("Etat courant : A");
    }
}

```

ConcreteEtatB.java

Sélectionnez

```

/**
 * Sous-classe concrète de l'interface "Etat"
 * On peut passer de l'état B vers l'état C mais pas vers l'état A
 */
public class ConcreteEtatB extends ClasseAvecEtat.Etat {

    // Méthodes pour changer d'état
    public void operationEtatA(ClasseAvecEtat pClasse) {
        System.out.println("Changement d'état (B -> A) non possible");
    }

    public void operationEtatB(ClasseAvecEtat pClasse) {
        System.out.println("Classe déjà dans l'état B");
    }

    public void operationEtatC(ClasseAvecEtat pClasse) {
        setEtat(pClasse, new ConcreteEtatC());
        System.out.println("Etat changé : B -> C");
    }

    /**
     * Affichage de l'état courant
     */
    public void afficher() {
        System.out.println("Etat courant : B");
    }
}

```

ConcreteEtatC.java

Sélectionnez


```

/**
 * Sous-classe concrète de l'interface "Etat"
 * On peut passer de l'état B vers l'état A mais pas vers l'état B
 */
public class ConcreteEtatC extends ClasseAvecEtat.Etat {

    // Méthodes pour changer d'état
    public void operationEtatA(ClasseAvecEtat pClasse) {
        setEtat(pClasse, new ConcreteEtatA());
        System.out.println("Etat changé : C -> A");
    }

    public void operationEtatB(ClasseAvecEtat pClasse) {
        System.out.println("Changement d'état (C -> B) non possible");
    }

    public void operationEtatC(ClasseAvecEtat pClasse) {
        System.out.println("Classe déjà dans l'état C");
    }

    /**
     * Affichage de l'état courant
     */
    public void afficher() {
        System.out.println("Etat courant : C");
    }
}

```

StatePatternMain.java

Sélectionnez

```

public class StatePatternMain {

    public static void main(String[] args) {
        // Création de la classe avec état
        ClasseAvecEtat lClasse = new ClasseAvecEtat();

        lClasse.operationEtatA();
        lClasse.operationEtatB();
        lClasse.operationEtatA();
        lClasse.afficher();
        lClasse.operationEtatB();
        lClasse.operationEtatC();
        lClasse.operationEtatA();

        // Affichage :
        // Classe déjà dans l'état A
        // Etat changé : A -> B
        // Changement d'état (B -> A) non possible
        // Etat courant : B
        // Classe déjà dans l'état B
    }
}

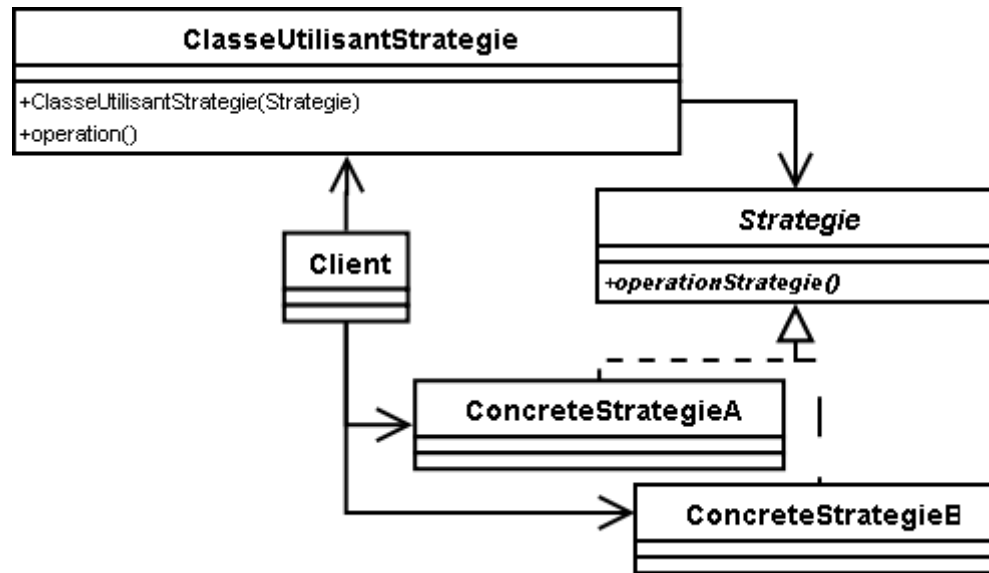
```

```

    // Etat changé : B -> C
    // Etat changé : C -> A
}
}

```

VI-I. Stratégie (Strategy ou Policy) ▲



OBJECTIFS :

- Définir une famille d'algorithmes interchangeable.
- Permettre de les changer indépendamment de la partie cliente.

RAISONS DE L'UTILISER :

Un objet doit pouvoir faire varier une partie de son algorithme.

Cela peut être une liste triée. A chaque insertion, la liste place le nouvel élément à l'emplacement correspondant au tri. Le tri peut être alphabétique, inverse, les majuscules avant les minuscules, les minuscules avant, etc...

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.

RESULTAT :

Le Design Pattern permet d'isoler les algorithmes appartenant à une même famille d'algorithmes.

RESPONSABILITES :

- **Strategie** : définit l'interface commune des algorithmes.
- **ConcreteStrategieA** et **ConcreteStrategieB** : implémentent les méthodes d'algorithme.
- **ClasseUtilisantStrategie** : utilise un objet **Strategie**.
- La partie cliente configure un objet **ClasseUtilisantStrategie** avec un objet **Strategie** et appelle la méthode de **ClasseUtilisantStrategie** qui utilise la stratégie. Dans l'exemple, la configuration s'effectue par le constructeur, mais la configuration peut également s'effectuer par une méthode "setter".

IMPLEMENTATION JAVA :

Strategie.java

Sélectionnez

```
/**
 * Définit l'interface d'une stratégie.
 */
public interface Strategie {

    public void operationStrategie();

}
```

ConcreteStrategieA.java

Sélectionnez

```
/**
 * Définit une stratégie
 */
public class ConcreteStrategieA implements Strategie {

    public void operationStrategie() {
        System.out.println("Operation de la strategie A");
    }

}
```

ConcreteStrategieB.java

Sélectionnez

```
/**
 * Définit une stratégie
 */
public class ConcreteStrategieB implements Strategie {

    public void operationStrategie() {
        System.out.println("Operation de la strategie B");
    }

}
```

ClasseUtilisantStrategie.java

Sélectionnez

```
/**
 * Utilise une stratégie.
 * La classe fait appel à la même méthode de l'objet "Strategie".
 */
```

```

    * C'est l'objet "Strategie" qui change.
    */
public class ClasseUtilisantStrategie {

    private Strategie strategie;

    /**
     * Constructeur recevant un objet "Strategie" en paramètre
     * @param pStrategie
     */
    public ClasseUtilisantStrategie(Strategie pStrategie) {
        strategie = pStrategie;
    }

    /**
     * Délègue le traitement à la stratégie
     */
    public void operation() {
        strategie.operationStrategie();
    }
}

```

StrategyPatternMain.java

Sélectionnez

```

public class StrategyPatternMain {

    public static void main(String[] args) {
        // Création d'instance des classes "Strategie"
        Strategie lStrategieA = new ConcreteStrategieA();
        Strategie lStrategieB = new ConcreteStrategieB();

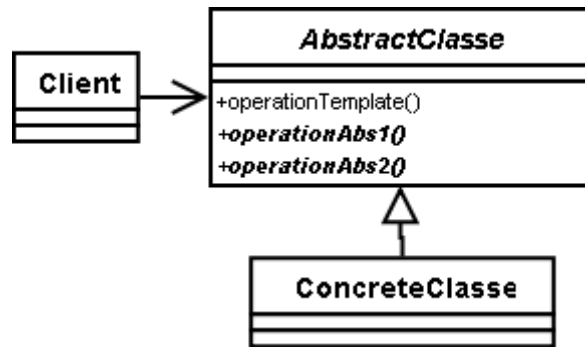
        // Création d'instance de la classe qui utilise des "Strategie"
        ClasseUtilisantStrategie lClasseA = new ClasseUtilisantStrategie(lStrategieA);
        ClasseUtilisantStrategie lClasseB = new ClasseUtilisantStrategie(lStrategieB);

        // Appel de la méthode de la classe
        // utilisant une stratégie
        lClasseA.operation();
        lClasseB.operation();

        // Affichage :
        // Operation de la strategie A
        // Operation de la strategie B
    }
}

```

VI-J. Patron de méthode (Template Method) ▲

**OBJECTIFS :**

Définir le squelette d'un algorithme en déléguant certaines étapes à des sous-classes.

RAISONS DE L'UTILISER :

Une classe possède un fonctionnement global. Mais les détails de son algorithme doivent être spécifiques à ses sous-classes.

Cela peut être le cas d'un document informatique. Le document a un fonctionnement global où il est sauvegardé. Pour la sauvegarde, il y aura toujours besoin d'ouvrir le fichier, d'écrire dedans, puis de fermer le fichier. Mais, selon le type de document, il ne sera pas sauvegardé de la même manière. S'il s'agit d'un document de traitement de texte, il sera sauvegardé en suite d'octets. S'il s'agit d'un document HTML, il sera sauvegardé dans un fichier texte.

La partie générale de l'algorithme (sauvegarde) est gérée par la classe abstraite (document). La partie générale réalise l'ouverture, fermeture du fichier et appelle une méthode d'écriture. La partie spécifique de l'algorithme (écriture dans le fichier) est définie au niveau des classes concrètes (document de traitement de texte ou document HTML).

RESULTAT :

Le Design Pattern permet d'isoler les parties variables d'un algorithme.

RESPONSABILITES :

- **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 **operationTemplate()** de **AbstractClasse**.
- La partie cliente appelle la méthode de **AbstractClasse** qui définit l'algorithme.

IMPLEMENTATION JAVA :

AbstractClasse.java

Sélectionnez

```

/**
 * Définit l'algorithme
 */
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

Sélectionnez

```

/**
 * Sous-classe concrète de AbstractClasse
 * Implémente les méthodes utilisées par l'algorithme
 * de la méthode operationTemplate() de AbstractClasse
 */
public class ConcreteClasse extends AbstractClasse {

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

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

```

TemplateMethodPatternMain.java

Sélectionnez

```

public class TemplateMethodPatternMain {

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

        // Affichage :
        // operationAbs1
        // operationAbs2 : 0
    }
}

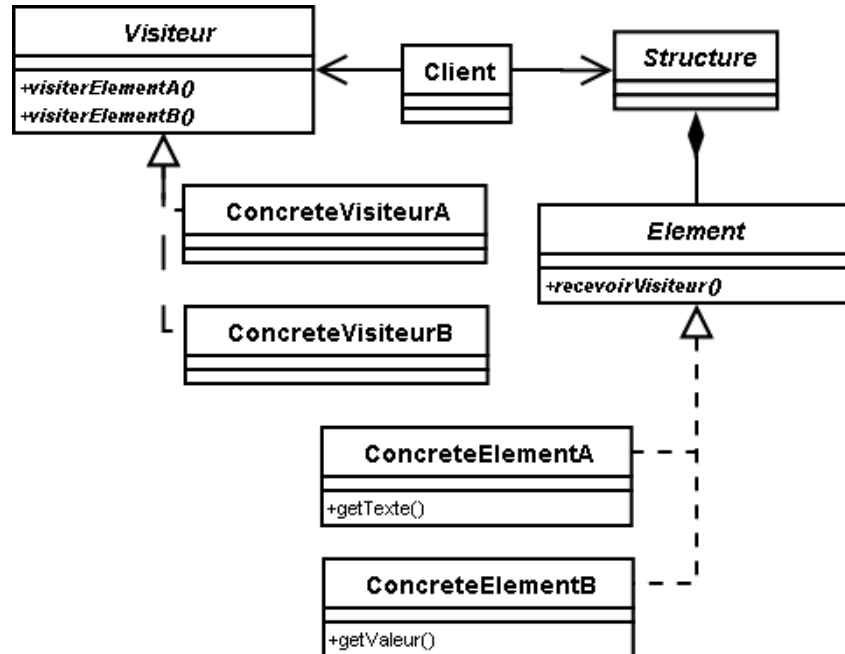
```

```

//      operationAbs2 : 1
//      operationAbs2 : 2
//      operationAbs2 : 3
//      operationAbs2 : 4
    }
}

```

VI-K. Visiteur (Visitor) ▲



OBJECTIFS :

Séparer un algorithme d'une structure de données.

RAISONS DE L'UTILISER :

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.

Cela peut être le cas d'un logiciel d'images de synthèse. L'image est composée de plusieurs objets : sphère, polygone, personnages de la scène qui sont constitués de plusieurs objets, etc... Sur chaque élément, il faut effectuer plusieurs opérations pour le rendu : ajout des couleurs, effet d'éclairage, etc...

Chaque type d'opération (ajout des couleurs, effet d'éclairage) est implémenté par un visiteur. Chaque visiteur implémente une méthode spécifique (visiter un sphère, visiter un polygone) pour chaque type d'élément (sphère, polygone). Chaque élément (sphère) implémente une méthode d'acceptation de visiteur où il appelle la méthode spécifique de visite.

RESULTAT :

Le Design Pattern permet d'isoler les algorithmes appliquées sur des structures de données.

RESPONSABILITES :

- **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**.

IMPLEMENTATION JAVA :

Element.java
Sélectionnez

```
/**
 * Définit l'interface d'un élément
 */
public interface Element {

    public void recevoirVisiteur(Visiteur pVisiteur);
}
```

ConcreteElementA.java
Sélectionnez

```
/**
 * Sous-classe concrète d'un élément.
 * Contient un donnée texte
 */
public class ConcreteElementA implements Element {

    public String texte;

    /**
     * Constructeur initialisant la donnée texte
     * @param pTexte
     */
    public ConcreteElementA(String pTexte) {
        texte = pTexte;
    }

    /**
     * Méthode retournant la donnée texte
     * @return
     */
}
```



```
    */
    public String getTexte() {
        return texte;
    }

    public void recevoirVisiteur(Visiteur pVisiteur) {
        pVisiteur.visiterElementA(this);
    }
}
```

ConcreteElementB.java

Sélectionnez

```
/**
 * Sous-classe concrète d'un élément.
 * Contient un donnée numérique
 */
public class ConcreteElementB implements Element {

    public Long valeur;

    /**
     * Constructeur initialisant la donnée numérique
     * @param pValeur
     */
    public ConcreteElementB(Long pValeur) {
        valeur = pValeur;
    }

    /**
     * Méthode retournant la donnée numérique
     * @return
     */
    public Long getValeur() {
        return valeur;
    }

    public void recevoirVisiteur(Visiteur pVisiteur) {
        pVisiteur.visiterElementB(this);
    }
}
```

Visiteur.java

Sélectionnez

```
/**
 * Définit l'interface d'un visiteur
 */
public interface Visiteur {

    public void visiterElementA(ConcreteElementA pElementA);
```

```
    public void visiterElementB(ConcreteElementB pElementB);  
}
```

ConcreteVisiteurA.java
Sélectionnez

```
/**  
 * Sous-classe concrète d'un visiteur.  
 */  
public class ConcreteVisiteurA implements Visiteur {  
  
    public void visiterElementA(ConcreteElementA pElementA) {  
        System.out.println("Visiteur A : ");  
        System.out.println("    Texte de l'element A : " + pElementA.getTexte());  
    }  
  
    public void visiterElementB(ConcreteElementB pElementB) {  
        System.out.println("Visiteur A : ");  
        System.out.println("    Valeur de l'element B : " + pElementB.getValeur());  
    }  
}
```

ConcreteVisiteurB.java
Sélectionnez

```
/**  
 * Sous-classe concrète d'un visiteur.  
 */  
public class ConcreteVisiteurB implements Visiteur {  
  
    public void visiterElementA(ConcreteElementA pElementA) {  
        System.out.println("Visiteur B : ");  
        System.out.println("    Texte de l'element A : " + pElementA.getTexte());  
    }  
  
    public void visiterElementB(ConcreteElementB pElementB) {  
        System.out.println("Visiteur B : ");  
        System.out.println("    Valeur de l'element B : " + pElementB.getValeur());  
    }  
}
```

Structure.java
Sélectionnez

```
/**  
 * Présente une interface de haut niveau permettant  
 * de visiter les objets "Element" la composant.  
 */  
public class Structure {
```

```
    private Element[] elements = new Element[] {
```

```

        new ConcreteElementA("texte1"),
        new ConcreteElementA("texte2"),
        new ConcreteElementB(new Long(1)),
        new ConcreteElementA("texte3"),
        new ConcreteElementB(new Long(2)),
        new ConcreteElementB(new Long(3))
    };

    /**
     * Méthode de visite
     */
    public void visiter(Visiteur pVisiteur) {
        for(int i=0;i<elements.length;i++) {
            elements[i].recevoirVisiteur(pVisiteur);
        }
    }
}

```

VisitorPatternMain.java

Sélectionnez

```

public class VisitorPatternMain {

    public static void main(String[] args) {
        // Création des visiteurs
        Visiteur lVisiteurA = new ConcreteVisiteurA();
        Visiteur lVisiteurB = new ConcreteVisiteurB();


        // Création de la structure
        Structure lStructure = new Structure();

        // Appels des méthodes de réception des visiteurs
        lStructure.visiter(lVisiteurA);
        lStructure.visiter(lVisiteurB);

        // Affichage :
        // Visiteur A :
        // Texte de l'element A : texte1
        // Visiteur A :
        //   Texte de l'element A : texte2
        // Visiteur A :
        //   Valeur de l'element B : 1
        // Visiteur A :
        //   Texte de l'element A : texte3
        // Visiteur A :
        //   Valeur de l'element B : 2
        // Visiteur A :
        //   Valeur de l'element B : 3
        // Visiteur B :
        //   Texte de l'element A : texte1
        // Visiteur B :
    }
}

```

```
// Texte de l'element A : texte2
// Visiteur B :
// Valeur de l'element B : 1
// Visiteur B :
// Texte de l'element A : texte3
// Visiteur B :
// Valeur de l'element B : 2
// Visiteur B :
// Valeur de l'element B : 3
}
}
```

Vous avez aimé ce tutoriel ? Alors partagez-le en cliquant sur les boutons suivants :  Partager

Copyright © 2008 Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site ni de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Droits de diffusion permanents accordés à Developpez LLC.

TrollDi : comment écrire du code non maintenable et qui vous assurera un travail à vie ?

Quels sont les langages de programmation les plus utilisés par les développeurs ?

Oracle annonce la sortie officielle de Java 10, ce qui signifie la fin des mises à jour et correctifs de sécurité gratuits pour Java 9

GPDR : un guide pratique pour les développeurs

Responsables bénévoles de la rubrique Java : Mickael Baron - Robin56 - Contacter par email

[Nous contacter](#)

[Participez](#)

[Hébergement](#)

[Informations légales](#)

[Partenaire : Hébergement Web](#)

© 2000-2019 - www.developpez.com