



## Design Patterns du Gang of Four appliqués à Java



### Table des matières

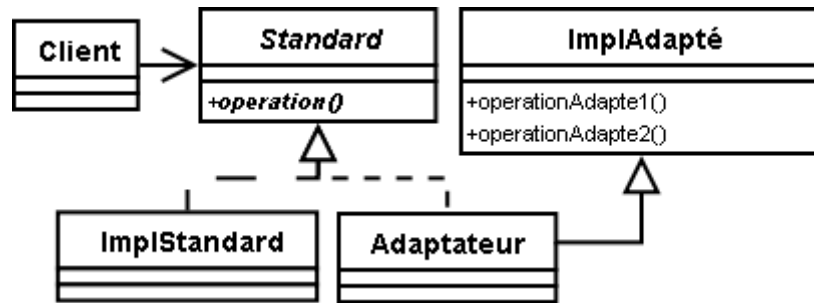
- V. STRUCTURAUX (STRUCTURAL PATTERNS)
  - V-A. Adaptateur (Adapter ou Wrapper)
  - V-B. Pont (Bridge ou Handle/Body)
  - V-C. Composite (Composite)
  - V-D. Décorateur (Decorator ou Wrapper)
  - V-E. Façade (Facade)
  - V-F. Poids-Mouche (Flyweight)
  - V-G. Proxy (Proxy ou Surrogate)

### V. STRUCTURAUX (STRUCTURAL PATTERNS) ▲

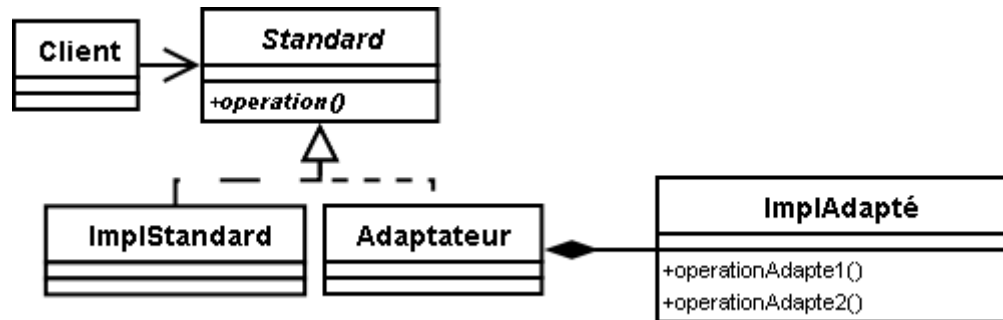
## V-A. Adaptateur (Adapter ou Wrapper) ▲

Le Design Pattern Adaptateur peut avoir deux formes :

Adaptateur avec héritage :



Adaptateur avec composition :



LIEN VERS LE DICTIONNAIRE DES DEVELOPPEURS :

Adapter

AUTRE RESSOURCE SUR DEVELOPPEZ.COM :

L'adaptateur par Sébastien MERIC

OBJECTIFS :

- Convertir l'interface d'une classe dans une autre interface comprise par la partie cliente.
- Permettre à des classes de fonctionner ensemble, ce qui n'aurait pas été possible sinon (à cause de leurs interfaces incompatibles).

RAISONS DE L'UTILISER :

Le système doit intégrer un sous-système existant. Ce sous-système a une interface non standard par rapport au système.

Cela peut être le cas d'un driver bas niveau pour de l'informatique embarquée. Le driver fourni par le fabricant ne correspond pas à l'interface utilisée par le système pour d'autres drivers.

La solution est de masquer cette interface non standard au système et de lui présenter une interface standard. La partie cliente utilise les méthodes de l'Adaptateur qui utilise les méthodes du sous-système pour réaliser les opérations correspondantes.

RESULTAT :

Le Design Pattern permet d'isoler l'adaptation d'un sous-système.

### RESPONSABILITES :

- **Standard** : définit une interface qui est identifiée comme standard dans la partie cliente.
- **ImplStandard** : implémente l'interface **Standard**. Cette classe n'a pas besoin d'être adaptée.
- **ImplAdapte** : permet de réaliser les fonctionnalités définies dans l'interface **Standard**, mais ne la respecte pas. Cette classe a besoin d'être adaptée.
- **Adaptateur** : adapte l'implémentation **ImplAdapte** à l'interface **Standard**. Pour réaliser l'adaptation, l'**Adaptateur** peut utiliser une ou plusieurs méthodes différentes de l'implémentation **ImplAdapte** pour réaliser l'implémentation de chaque méthode de l'interface **Standard**.
- La partie cliente manipule des objets **Standard**. Donc, l'adaptation est transparente pour la partie cliente.

### IMPLEMENTATION JAVA :

Standard.java

Sélectionnez

```
/**
 * Définit une interface qui est identifiée
 * comme standard dans la partie cliente.
 */
public interface Standard {

    /**
     * L'opération doit multiplier les deux nombres,
     * puis afficher le résultat de l'opération
     */
    public void operation(int pNombre1, int pNombre2);
}
```

ImplStandard.java

Sélectionnez

```
/**
 * Implémente l'interface "Standard".
 */
public class ImplStandard implements Standard {

    public void operation(int pNombre1, int pNombre2) {
        System.out.println("Standard : Le nombre est : " + (pNombre1 * pNombre2));
    }
}
```

ImplAdapte.java

Sélectionnez

```
/**
 * Fournit les fonctionnalités définies dans l'interface "Standard",
 * mais ne respecte pas l'interface.
 */
public class ImplAdapte {
```

```

public int operationAdapte1(int pNombre1, int pNombre2) {
    return pNombre1 * pNombre2;
}

/**
 * Apporte la fonctionnalité définie dans l'interface,
 * mais la méthode n'a pas le bon nom
 * et n'accepte pas le même paramètre.
 */
public void operationAdapte2(int pNombre) {
    System.out.println("Adapte : Le nombre est : " + pNombre);
}
}

```

Adaptateur.java (avec héritage)

Sélectionnez

```

/**
 * Adapte l'implémentation non standard avec l'héritage.
 */
public class Adaptateur extends ImplAdapte implements Standard {

    /**
     * Appelle les méthodes non standard
     * depuis une méthode respectant l'interface.
     * 1°) Appel de la méthode réalisant la multiplication
     * 2°) Appel de la méthode d'affichage du résultat
     * La classe adaptée est héritée, donc on appelle directement les méthodes
     */
    public void operation(int pNombre1, int pNombre2) {
        int lNombre = operationAdapte1(pNombre1, pNombre2);
        operationAdapte2(lNombre);
    }
}

```

Adaptateur.java (avec composition)

Sélectionnez

```

/**
 * Adapte l'implémentation non standard avec la composition.
 */
public class Adaptateur implements Standard {

    private ImplAdapte adapte = new ImplAdapte();

    /**
     * Appelle les méthodes non standard
     * depuis une méthode respectant l'interface.
     * 1°) Appel de la méthode réalisant la multiplication
     * 2°) Appel de la méthode d'affichage du résultat
     */
}

```

```

    * La classe adaptée compose l'adaptation,
    * donc on appelle les méthodes de "ImplAdapte".
    */
    public void operation(int pNombre1, int pNombre2) {
        int lNombre = adapte.operationAdapte1(pNombre1, pNombre2);
        adapte.operationAdapte2(lNombre);
    }
}

```

AdaptatorPatternMain.java  
Sélectionnez

```

public class AdaptatorPatternMain {

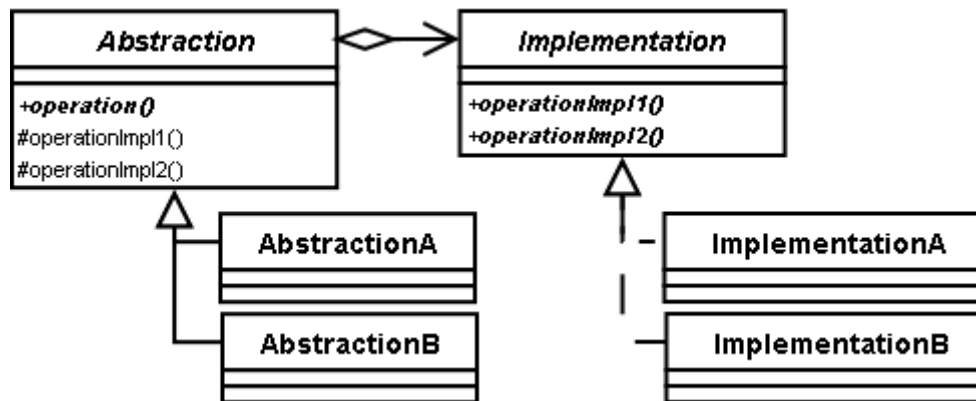
    public static void main(String[] args) {
        // Création d'un adaptateur
        final Standard lImplAdapte = new Adaptateur();
        // Création d'une implémentation standard
        final Standard lImplStandard = new ImplStandard();

        // Appel de la même méthode sur chaque instance
        lImplAdapte.operation(2, 4);
        lImplStandard.operation(2, 4);

        // Affichage :
        // Adapte : Le nombre est : 8
        // Standard : Le nombre est : 8
    }
}

```

## V-B. Pont (Bridge ou Handle/Body) ▲



OBJECTIFS :

- Découpler l'abstraction d'un concept de son implémentation.
- Permettre à l'abstraction et l'implémentation de varier indépendamment.

### RAISONS DE L'UTILISER :

Le système comporte une couche bas niveau réalisant l'implémentation et une couche haut niveau réalisant l'abstraction. Il est nécessaire que chaque couche soit indépendante.

Cela peut être le cas du système d'édition de documents d'une application. Pour l'implémentation, il est possible que l'édition aboutisse à une sortie imprimante, une image sur disque, un document PDF, etc... Pour l'abstraction, il est possible qu'il s'agisse d'édition de factures, de rapports de stock, de courriers divers, etc...

Chaque implémentation présente une interface pour les opérations de bas niveau standard (sortie imprimante), et chaque abstraction hérite d'une classe effectuant le lien avec cette interface (tracer une ligne). Ainsi les abstractions peuvent utiliser ce lien pour appeler la couche implémentation pour leurs besoins (imprimer facture).

### RESULTAT :

Le Design Pattern permet d'isoler le lien entre une couche de haut niveau et celle de bas niveau.

### RESPONSABILITES :

- **Implementation** : définit l'interface de l'implémentation. Cette interface n'a pas besoin de correspondre à l'interface de l'**Abstraction**. L'**Implementation** peut, par exemple, définir des opérations primitives de bas niveau et l'**Abstraction** des opérations de haut niveau qui utilisent les opérations de l'**Implementation**.
- **ImplementationA** et **ImplementationB** : sont des sous-classes concrètes de l'implémentation.
- **Abstraction** : définit l'interface de l'abstraction. Elle possède une référence vers un objet **Implementation**. C'est elle qui définit le lien entre l'abstraction et l'implémentation. Pour définir ce lien, la classe implémente des méthodes qui appellent des méthodes de l'objet **Implementation**.
- **AbstractionA** et **AbstractionB** : sont des sous-classes concrètes de l'abstraction. Elle utilise les méthodes définies par la classe **Abstraction**.
- La partie cliente fournit un objet **Implementation** à l'objet **Abstraction**. Puis, elle fait appel aux méthodes fournies par l'interface de l'abstraction.

### IMPLEMENTATION JAVA :

Implementation.java

Sélectionnez

```
/**
 * Définit l'interface de l'implémentation.
 * L'implémentation fournit deux méthodes
 */
public interface Implementation {

    public void operationImpl1(String pMessage);
    public void operationImpl2(Integer pNombre);
}
```

ImplementationA.java

Sélectionnez

```
/**
 * Sous-classe concrète de l'implémentation
 */
public class ImplementationA implements Implementation {
```

```

    public void operationImpl1(String pMessage) {
        System.out.println("operationImpl1 de ImplementationA : " + pMessage);
    }

    public void operationImpl2(Integer pNombre) {
        System.out.println("operationImpl2 de ImplementationA : " + pNombre);
    }
}

```

ImplementationB.java

Sélectionnez

```

/**
 * Sous-classe concrète de l'implémentation
 */
public class ImplementationB implements Implementation {

    public void operationImpl1(String pMessage) {
        System.out.println("operationImpl1 de ImplementationB : " + pMessage);
    }

    public void operationImpl2(Integer pNombre) {
        System.out.println("operationImpl2 de ImplementationB : " + pNombre);
    }
}

```

Abstraction.java

Sélectionnez

```

/**
 * Définit l'interface de l'abstraction
 */
public abstract class Abstraction {

    // Référence vers l'implémentation
    private Implementation implementation;

    protected Abstraction(Implementation pImplementation) {
        implementation = pImplementation;
    }

    public abstract void operation();

    /**
     * Lien vers la méthode operationImpl1() de l'implémentation
     * @param pMessage
     */
    protected void operationImpl1(String pMessage) {
        implementation.operationImpl1(pMessage);
    }
}

```

```

/**
 * Lien vers la méthode operationImpl2() de l'implémentation
 * @param pMessage
 */
protected void operationImpl2(Integer pNombre) {
    implementation.operationImpl2(pNombre);
}
}

```

AbstractionA.java

Sélectionnez

```

/**
 * Sous-classe concrète de l'abstraction
 */
public class AbstractionA extends Abstraction {

    public AbstractionA(Implementation pImplementation) {
        super(pImplementation);
    }

    public void operation() {
        System.out.println("--> Méthode operation() de AbstractionA");
        operationImpl1("A");
        operationImpl2(1);
        operationImpl1("B");
    }
}

```

AbstractionB.java

Sélectionnez

```

/**
 * Sous-classe concrète de l'abstraction
 */
public class AbstractionB extends Abstraction {

    public AbstractionB(Implementation pImplementation) {
        super(pImplementation);
    }

    public void operation() {
        System.out.println("--> Méthode operation() de AbstractionB");
        operationImpl2(9);
        operationImpl2(8);
        operationImpl1("Z");
    }
}

```

BridgePatternMain.java



Sélectionnez

```
public class BridgePatternMain {

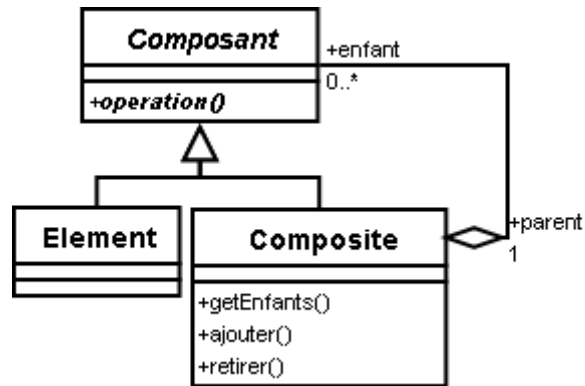
    public static void main(String[] args) {
        // Création des implémentations
        Implementation lImplementationA = new ImplementationA();
        Implementation lImplementationB = new ImplementationB();

        // Création des abstractions
        Abstraction lAbstractionAA = new AbstractionA(lImplementationA);
        Abstraction lAbstractionAB = new AbstractionA(lImplementationB);
        Abstraction lAbstractionBA = new AbstractionB(lImplementationA);
        Abstraction lAbstractionBB = new AbstractionB(lImplementationB);

        // Appels des méthodes des abstractions
        lAbstractionAA.operation();
        lAbstractionAB.operation();
        lAbstractionBA.operation();
        lAbstractionBB.operation();

        // Affichage :
        // --> Méthode operation() de AbstractionA
        // operationImpl1 de ImplementationA : A
        // operationImpl2 de ImplementationA : 1
        // operationImpl1 de ImplementationA : B
        // --> Méthode operation() de AbstractionA
        // operationImpl1 de ImplementationB : A
        // operationImpl2 de ImplementationB : 1
        // operationImpl1 de ImplementationB : B
        // --> Méthode operation() de AbstractionB
        // operationImpl2 de ImplementationA : 9
        // operationImpl2 de ImplementationA : 8
        // operationImpl1 de ImplementationA : Z
        // --> Méthode operation() de AbstractionB
        // operationImpl2 de ImplementationB : 9
        // operationImpl2 de ImplementationB : 8
        // operationImpl1 de ImplementationB : Z
    }
}
```

## V-C. Composite (Composite) ▲



### AUTRE RESSOURCE SUR DEVELOPPEZ.COM :

Le composite par Sébastien MERIC

### OBJECTIFS :

- Organiser les objets en structure arborescente afin de représenter une hiérarchie.
- Permettre à la partie cliente de manipuler un objet unique et un objet composé de la même manière.

### RAISONS DE L'UTILISER :

Le système comporte une hiérarchie avec un nombre de niveaux non déterminé. Il est nécessaire de pouvoir considérer un groupe d'éléments comme un élément unique.

Cela peut être le cas des éléments graphiques d'un logiciel de DAO. Plusieurs éléments graphiques peuvent être regroupés en un nouvel élément graphique.

Chaque élément est un composant potentiel. En plus des éléments classiques, il y a un élément composite qui peut être composé de plusieurs composants. Comme l'élément composite est un composant potentiel, il peut être composé d'autres éléments composites.

### RESULTAT :

Le Design Pattern permet d'isoler l'appartenance à un agrégat.

### RESPONSABILITES :

- **Composant** : définit l'interface d'un objet pouvant être un composant d'un autre objet de l'arborescence.
- **Element** : implémente un objet de l'arborescence n'ayant pas d'objet le composant.
- **Composite** : implémente un objet de l'arborescence ayant un ou des objets le composant.
- La partie client manipule les objets par l'interface **Composant**.

### IMPLEMENTATION JAVA :

Composant.java  
Sélectionnez

```

/**
 * Définit l'interface d'un objet pouvant être un composant
 * d'un autre objet de l'arborescence.
 */
public abstract class Composant {

    // Nom de "Composant"
    protected String nom;

    /**
     * Constructeur
     * @param pNom Nom du "Composant"
     */
    public Composant(final String pNom) {
        nom = pNom;
    }

    /**
     * Opération commune à tous les "Composant"
     */
    public abstract void operation();
}

```

Element.java  
Sélectionnez

```

/**
 * Implémente un objet de l'arborescence
 * n'ayant pas d'objet le composant.
 */
public class Element extends Composant {

    public Element(final String pNom) {
        super(pNom);
    }

    /**
     * Méthode commune à tous les composants :
     * Affiche qu'il s'agit d'un objet "Element"
     * ainsi que le nom qu'on lui a donné.
     */
    public void operation() {
        System.out.println("Op. sur un 'Element' (" + nom + ")");
    }
}

```

Composite.java  
Sélectionnez

```

/**
 * Implémente un objet de l'arborescence

```

```
* ayant un ou des objets le composant.
*/
public class Composite extends Composant {

    // Liste d'objets "Composant" de l'objet "Composite"
    private List<Composant> liste = new LinkedList<Composant>();

    public Composite(final String pNom) {
        super(pNom);
    }

    /**
     * Méthode commune à tous les composants :
     * Affiche qu'il s'agit d'un objet "Composite"
     * ainsi que le nom qu'on lui a donné,
     * puis appelle la méthode "operation()"
     * de tous les composants de cet objet.
     */
    public void operation() {
        System.out.println("Op. sur un 'Composite' (" + nom + ")");
        final Iterator<Composant> lIterator = liste.iterator();
        while(lIterator.hasNext()) {
            final Composant lComposant = lIterator.next();
            lComposant.operation();
        }
    }

    /**
     * Retourne la liste d'objets "Composant"
     * @return La liste d'objets "Composant"
     */
    public List<Composant> getEnfants() {
        return liste;
    }

    /**
     * Ajoute un objet "Composant" au "Composite"
     * @param pComposant
     */
    public void ajouter(final Composant pComposant) {
        liste.add(pComposant);
    }

    /**
     * Retire un objet "Composant"
     * @param pComposant
     */
    public void retirer(final Composant pComposant) {
        liste.remove(pComposant);
    }
}
```

CompositePatternMain.java

Sélectionnez

```
public class CompositePatternMain {

    public static void main(String[] args) {

        // On va créer l'arborescence :
        // lComposite1
        //     - lElement1
        //     - lComposite2
        //         - lComposite3
        //             - lElement3
        //             - lElement4
        //         - lComposite4
        //             - lComposite5
        //                 - lElement5
        //     - lElement2

        // Création des objets "Composite"
        final Composite lComposite1 = new Composite("Composite 1");
        final Composite lComposite2 = new Composite("Composite 2");
        final Composite lComposite3 = new Composite("Composite 3");
        final Composite lComposite4 = new Composite("Composite 4");
        final Composite lComposite5 = new Composite("Composite 5");

        // Création des objets "Element"
        final Element lElement1 = new Element("Element 1");
        final Element lElement2 = new Element("Element 2");
        final Element lElement3 = new Element("Element 3");
        final Element lElement4 = new Element("Element 4");
        final Element lElement5 = new Element("Element 5");

        // Ajout des "Composant" afin de constituer l'arborescence
        lComposite1.ajouter(lElement1);
        lComposite1.ajouter(lComposite2);
        lComposite1.ajouter(lElement2);

        lComposite2.ajouter(lComposite3);
        lComposite2.ajouter(lComposite4);

        lComposite3.ajouter(lElement3);
        lComposite3.ajouter(lElement4);

        lComposite4.ajouter(lComposite5);

        lComposite5.ajouter(lElement5);

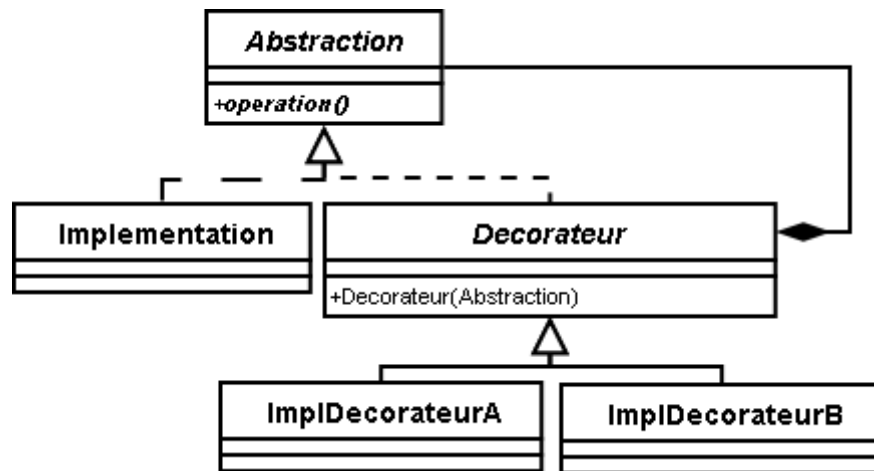
        // Appel de la méthode "operation()" de la racine
        // afin d'afficher les différents "Composant"
        lComposite1.operation();
    }
}
```

```

// Affichage :
// Op. sur un 'Composite' (Composite 1)
// Op. sur un 'Element' (Element 1)
// Op. sur un 'Composite' (Composite 2)
// Op. sur un 'Composite' (Composite 3)
// Op. sur un 'Element' (Element 3)
// Op. sur un 'Element' (Element 4)
// Op. sur un 'Composite' (Composite 4)
// Op. sur un 'Composite' (Composite 5)
// Op. sur un 'Element' (Element 5)
// Op. sur un 'Element' (Element 2)
}
}

```

## V-D. Décorateur (Decorator ou Wrapper) ▲



AUTRE RESSOURCE SUR DEVELOPPEZ.COM :

Le décorateur par Sébastien MERIC

OBJECTIFS :

- Ajouter dynamiquement des responsabilités (non obligatoires) à un objet.
- Éviter de sous-classer la classe pour rajouter ces responsabilités.

RAISONS DE L'UTILISER :

Il est nécessaire de pouvoir étendre les responsabilités d'une classe sans avoir recours au sous-classage.

Cela peut être le cas d'une classe gérant des d'entrées/sorties à laquelle on souhaite ajouter un buffer et des traces de log.

La classe de départ est l'implémentation. Les fonctionnalités supplémentaires (buffer, log) sont implémentées par des classes supplémentaires : les décorateurs. Les décorateurs ont la même interface que la classe de départ. Dans leur implémentation des méthodes, elles implémentent les fonctionnalités supplémentaires et font appel à la méthode correspondante d'une instance avec la même interface. Ainsi, il est possible d'enchaîner plusieurs responsabilités supplémentaires, puis d'aboutir à l'implémentation finale.

## RESULTAT :

Le Design Pattern permet d'isoler les responsabilités d'un objet.

## RESPONSABILITES :

- **Abstraction** : définit l'interface générale.
- **Implementation** : implémente l'interface générale. Cette classe contient l'implémentation de l'interface correspondant aux fonctionnalités souhaitées à la base.
- **Decorateur** : définit l'interface du décorateur et contient une référence vers un objet **Abstraction**.
- **ImplDecorateurA** et **ImplDecorateurB** : implémentent des décorateurs. Les décorateurs ont un constructeur acceptant un objet **Abstraction**. Les méthodes des décorateurs appellent la même méthode de l'objet qui a été passée au constructeur. La décoration ajoute des responsabilités en effectuant des opérations avant et/ou après cet appel.
- La partie cliente manipule un objet **Abstraction**. En réalité, cet objet **Abstraction** peut être un objet **Implementation** ou un objet **Decorateur**. Ainsi, des fonctionnalités supplémentaires peuvent être ajoutées à la méthode d'origine. Ces fonctionnalités peuvent être par exemple des traces de log ou une gestion de buffer pour des entrées/sorties.

## IMPLEMENTATION JAVA :

Abstraction.java

Sélectionnez

```
/**
 * Définit l'interface générale.
 */
public interface Abstraction {

    /**
     * Méthode générale.
     */
    public void operation();
}
```

Implementation.java

Sélectionnez

```
/**
 * Implémente l'interface générale.
 */
public class Implementation implements Abstraction {

    /**
     * Implémentation de la méthode
     * pour l'opération de base
     */
    public void operation() {
```

```

        System.out.println("Implementation");
    }
}

```

Decorateur.java

Sélectionnez

```

/**
 * Définit l'interface du décorateur.
 */
public abstract class Decorateur implements Abstraction {
    protected Abstraction abstraction;

    /**
     * Le constructeur du "Decorateur" reçoit un objet "Abstraction"
     * @param pAbstraction
     */
    public Decorateur(final Abstraction pAbstraction) {
        abstraction = pAbstraction;
    }
}

```

ImplDecoratorA.java

Sélectionnez

```

/**
 * Implémente un décorateur
 */
public class ImplDecorateurA extends Decorateur {

    public ImplDecorateurA(final Abstraction pAbstraction) {
        super(pAbstraction);
    }

    /**
     * Implémentation de la méthode
     * pour la décoration de "ImplDecorateurA".
     * Des opérations sont effectuées avant et après
     * l'appel à la méthode de l'objet "Abstraction"
     * passé au constructeur.
     * La méthode ignore si cet objet est un autre décorateur
     * ou l'implémentation
     */
    public void operation() {
        System.out.println("ImplDecorateurA avant");
        abstraction.operation();
        System.out.println("ImplDecorateurA apres");
    }
}

```

ImplDecoratorB.java



Sélectionnez

```
/**
 * Implémente un décorateur
 */
public class ImplDecorateurB extends Decorateur {

    public ImplDecorateurB(final Abstraction pAbstraction) {
        super(pAbstraction);
    }

    /**
     * Implémentation de la méthode
     * pour la décoration de "ImplDecorateurB".
     */
    public void operation() {
        System.out.println("ImplDecorateurB avant");
        abstraction.operation();
        System.out.println("ImplDecorateurB apres");
    }
}
```

DecoratorPatternMain.java

Sélectionnez

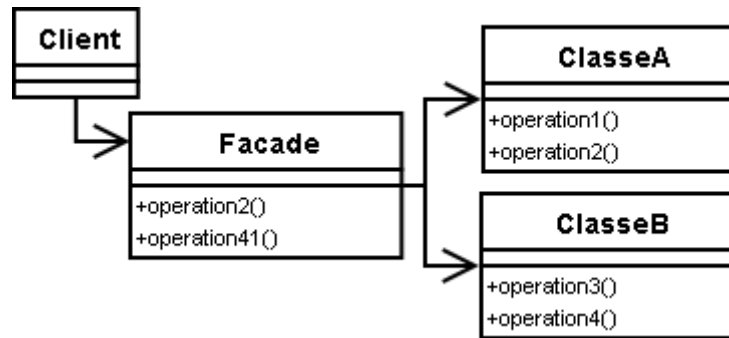
```
public class DecoratorPatternMain {

    public static void main(String[] args) {
        // Création de l'implémentation et des décorateurs
        final Implementation lImpl = new Implementation();
        final ImplDecorateurB lImplDecB = new ImplDecorateurB(lImpl);
        final ImplDecorateurA lImplDecA = new ImplDecorateurA(lImplDecB);

        // Appel de la méthode du décorateur "conteneur"
        lImplDecA.operation();

        // Affichage :
        // ImplDecorateurA avant
        // ImplDecorateurB avant
        // Implementation
        // ImplDecorateurB apres
        // ImplDecorateurA apres
    }
}
```

## V-E. Façade (Facade) ▲



### AUTRE RESSOURCE SUR DEVELOPPEZ.COM :

La façade par Sébastien MERIC

### OBJECTIFS :

- Fournir une interface unique en remplacement d'un ensemble d'interfaces d'un sous-système.
- Définir une interface de haut niveau pour rendre le sous-système plus simple d'utilisation.

### RAISONS DE L'UTILISER :

Le système comporte un sous-système complexe avec plusieurs interfaces. Certaines de ces interfaces présentent des opérations qui ne sont pas utiles au reste du système.

Cela peut être le cas d'un sous-système communiquant avec des outils de mesure ou d'un sous-système d'accès à la base de données.

Il serait plus judicieux de passer par une seule interface présentant seulement les opérations utiles. Une classe unique, la façade, présente ces opérations réellement nécessaires. Remarque : La façade peut également implémenter le Design Pattern Singleton.

### RESULTAT :

Le Design Pattern permet d'isoler les fonctionnalités d'un sous-système utiles à la partie cliente.

### RESPONSABILITES :

- **ClasseA** et **ClasseB** : implémentent diverses fonctionnalités.
- **Facade** : présente certaines fonctionnalités. Cette classe utilise les implémentations des objets **ClasseA** et **ClasseB**. Elle expose une version simplifiée du sous-système **ClasseA-ClasseB**.
- La partie cliente fait appel aux méthodes présentées par l'objet **Facade**. Il n'y a donc pas de dépendances entre la partie cliente et le sous-système **ClasseA-ClasseB**.

### IMPLEMENTATION JAVA :

ClasseA.java  
Sélectionnez

```
/**
 * Classe implémentant diverses fonctionnalités.
 */
public class ClasseA {

    public void operation1() {
        System.out.println("Methode operation1() de la classe ClasseA");
    }

    public void operation2() {
        System.out.println("Methode operation2() de la classe ClasseA");
    }
}
```

ClasseB.java

Sélectionnez

```
/**
 * Classe implémentant d'autres fonctionnalités.
 */
public class ClasseB {

    public void operation3() {
        System.out.println("Methode operation3() de la classe ClasseB");
    }

    public void operation4() {
        System.out.println("Methode operation4() de la classe ClasseB");
    }
}
```

Facade.java

Sélectionnez

```
/**
 * Présente certaines fonctionnalités.
 * Dans ce cas, ne présente que la méthode "operation2()" de "ClasseA"
 * et la méthode "operation4()" utilisant "operation4()" de "ClasseB"
 * et "operation1()" de "ClasseA".
 */
public class Facade {

    private ClasseA classeA = new ClasseA();
    private ClasseB classeB = new ClasseB();

    /**
     * La méthode operation2() appelle simplement
     * la même méthode de ClasseA
     */
    public void operation2() {
        System.out.println("--> Méthode operation2() de la classe Facade : ");
    }
}
```

```

        classeA.operation2();
    }

    /**
     * La méthode operation41() appelle
     * operation4() de ClasseB
     * et operation1() de ClasseA
     */
    public void operation41() {
        System.out.println("--> Méthode operation41() de la classe Facade : ");
        classeB.operation4();
        classeA.operation1();
    }
}

```

FacadePatternMain.java

Sélectionnez

```

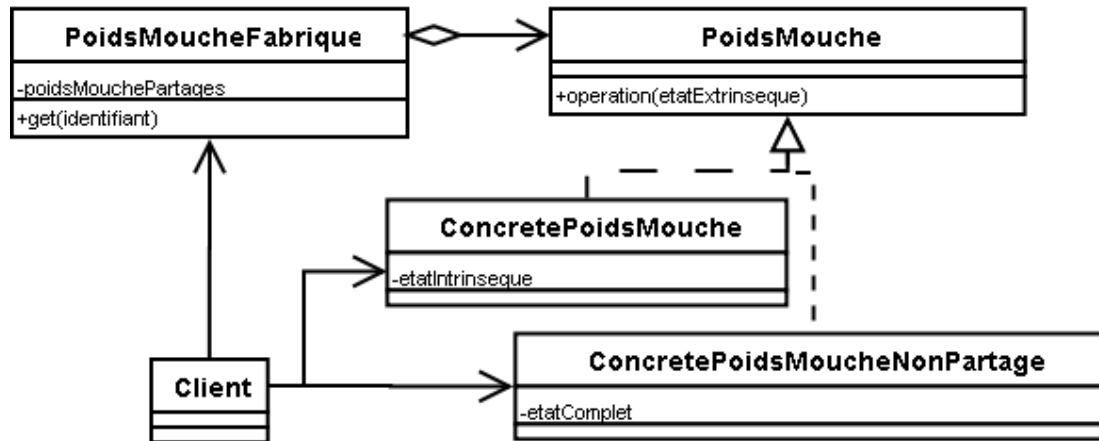
public class FacadePatternMain {

    public static void main(String[] args) {
        // Création de l'objet "Facade" puis appel des méthodes
        Facade lFacade = new Facade();
        lFacade.operation2();
        lFacade.operation41();

        // Affichage :
        // --> Méthode operation2() de la classe Facade :
        // Methode operation2() de la classe ClasseA
        // --> Méthode operation41() de la classe Facade :
        // Methode operation4() de la classe ClasseB
        // Methode operation1() de la classe ClasseA
    }
}

```

## V-F. Poids-Mouche (Flyweight) ▲



### OBJECTIFS :

Utiliser le partage pour gérer efficacement un grand nombre d'objets de faible granularité.

### RAISONS DE L'UTILISER :

Un système utilise un grand nombre d'instances. Cette quantité occupe une place très importante en mémoire. Or, chacune de ces instances a des attributs extrinsèques (propre au contexte) et intrinsèques (propre à l'objet).

Cela peut être les caractéristiques des traits dans un logiciel de DAO. Le trait a une épaisseur (simple ou double), une continuité (continu, en pointillé), une ombre ou pas, des coordonnées. Les caractéristiques d'épaisseur, de continuité et d'ombre sont des attributs intrinsèques à un trait, tandis que les coordonnées sont des attributs extrinsèques. Plusieurs traits possèdent des épaisseurs, continuité et ombre similaires. Ces similitudes correspondent à des styles de trait.

En externalisant les attributs intrinsèques des objets (style de trait), on peut avoir en mémoire une seule instance correspondant à un groupe de valeurs (simple-continu-sans ombre, double-pointillé-ombre). Chaque objet avec des attributs extrinsèques (trait avec les coordonnées) possède une référence vers une instance d'attributs intrinsèques (style de trait). On obtient deux types de poids-mouche : les poids-mouche partagés (style de trait) et les poids-mouche non partagés (le trait avec ses coordonnées). La partie cliente demande le poids-mouche qui l'intéresse à la fabrique de poids-mouche. S'il s'agit d'un poids-mouche non partagé, la fabrique le créera et le retournera. S'il s'agit d'un poids-mouche partagé, la fabrique vérifiera si une instance existe. Si une instance existe, la fabrique la retourne, sinon la fabrique la crée et la retourne.

### RESULTAT :

Le Design Pattern permet d'isoler des objets partageables.

### RESPONSABILITES :

- **PoidsMouche** : déclare l'interface permettant à l'objet de recevoir et d'agir en fonction de données extrinsèques. On externalise les données extrinsèques d'un objet **PoidsMouche** afin qu'il puisse être réutilisé.
- **ConcretePoidsMouche** : implémente l'interface poids-mouche. Les informations contenues dans un **ConcretePoidsMouche** sont intrinsèques (sans lien avec son contexte). Puisque, ce type de poids-mouche est obligatoirement partagé.
- **ConcretePoidsMoucheNonPartage** : implémente l'interface poids-mouche. Ce type de poids-mouche n'est pas partagé. Il possède des données intrinsèques et extrinsèques.

- **PoidsMoucheFabrique** : fournit une méthode retournant une instance de **PoidsMouche**. Si les paramètres de l'instance souhaitée correspondent à un **PoidsMouche** partagé, l'objet **PoidsMoucheFabrique** retourne une instance déjà existante. Sinon l'objet **PoidsMoucheFabrique** crée une nouvelle instance.
- La partie cliente demande à PoidsMoucheFabrique de lui fournir un PoidsMouche.

#### IMPLEMENTATION JAVA :

PoidsMouche.java

Sélectionnez

```
/**
 * Classe dont on souhaite limiter le nombre d'instance en mémoire.
 */
public interface PoidsMouche {

    public void afficher(String pContexte);

}
```

ConcretePoidsMouche.java

Sélectionnez

```
/**
 * Classe dont on souhaite limiter le nombre d'instance en mémoire.
 */
public class ConcretePoidsMouche implements PoidsMouche {

    private String valeur;

    public ConcretePoidsMouche(String pValeur) {
        valeur = pValeur;
    }

    public void afficher(String pContexte) {
        System.out.println("PoidsMouche avec la valeur : " + valeur +
            " et contexte : " + pContexte);
    }

}
```

ConcretePoidsMoucheNonPartage.java

Sélectionnez

```
/**
 * Sous-classe de Poids-Mouche dont on ne partage pas les instances.
 */
public class ConcretePoidsMoucheNonPartage implements PoidsMouche {

    private String valeur1;
    private String valeur2;

    public ConcretePoidsMoucheNonPartage(String pValeur1, String pValeur2) {
        valeur1 = pValeur1;
        valeur2 = pValeur2;
    }

}
```

```

    }

    public void afficher(String pContexte) {
        System.out.println("PoidsMouche avec la valeur1 : " + valeur1 +
            " avec la valeur2 : " + valeur2);
    }
}

```

PoidsMoucheFabrique.java  
Sélectionnez

```

/**
 * Fabrique de PoidsMouche
 */
public class PoidsMoucheFabrique {

    // Tableau des "PoidsMouche" partagés
    private Hashtable<String, ConcretePoidsMouche> poidsMouchesPartages =
        new Hashtable<String, ConcretePoidsMouche>();

    PoidsMoucheFabrique() {
        poidsMouchesPartages.put("Bonjour", new ConcretePoidsMouche("Bonjour"));
        poidsMouchesPartages.put("le", new ConcretePoidsMouche("le"));
        poidsMouchesPartages.put("monde", new ConcretePoidsMouche("monde"));
    }

    /**
     * Retourne un "PoidsMouche" partagé
     * Si la valeur passé en paramètre
     * correspond à un "PoidsMouche" partagé déjà existant,
     * on le retourne.
     * Sinon on crée une nouvelle instance,
     * on la stocke et on la retourne.
     * @param pValeur Valeur du "PoidsMouche" désiré
     * @return un "PoidsMouche"
     */
    public PoidsMouche getPoidsMouche(String pValeur) {
        if(poidsMouchesPartages.containsKey(pValeur)) {
            System.out.println("--> Retourne un PoidsMouche (" + pValeur +
                ") partagé déjà existant");
            return poidsMouchesPartages.get(pValeur);
        }
        else {
            System.out.println("--> Retourne un PoidsMouche (" + pValeur +
                ") partagé non déjà existant");
            final ConcretePoidsMouche lNouveauPoidsMouche = new ConcretePoidsMouche(
                poidsMouchesPartages.put(pValeur, lNouveauPoidsMouche));
            return lNouveauPoidsMouche;
        }
    }
}

```

```

/**
 * Retourne un "PoidsMouche" non partagé.
 * @param pValeur1
 * @param pValeur2
 * @return un "PoidsMouche"
 */
public PoidsMouche getPoidsMouche(String pValeur1, String pValeur2) {
    System.out.println("--> Retourne un PoidsMouche (" + pValeur1 + ", " +
        pValeur2 + ") non partagé");
    return new ConcretePoidsMoucheNonPartage(pValeur1, pValeur2);
}
}

```

FlyweightPatternMain.java

Sélectionnez

```

public class FlyweightPatternMain {

    public static void main(String[] args) {
        // Instancie la fabrique
        PoidsMoucheFabrique lFlyweightFactory = new PoidsMoucheFabrique();

        // Demande des "PoidsMouche" qui sont partagés
        PoidsMouche lFlyweight1 = lFlyweightFactory.getPoidsMouche("Bonjour");
        PoidsMouche lFlyweight1Bis = lFlyweightFactory.getPoidsMouche("Bonjour");

        // Affiche ces deux "PoidsMouche"
        lFlyweight1.afficher("Contexte1");
        lFlyweight1Bis.afficher("Contexte1Bis");

        // Affiche si les références pointent sur la même instance
        // Cela est logique puisque c'est le principe de l'instance partagée.
        System.out.print("lFlyweight1 == lFlyweight1Bis : ");
        System.out.println(lFlyweight1 == lFlyweight1Bis);

        // Demande un "PoidsMouche" qui ne fait pas partie des existants
        PoidsMouche lFlyweight2 = lFlyweightFactory.getPoidsMouche("BonjouR");
        PoidsMouche lFlyweight2Bis = lFlyweightFactory.getPoidsMouche("BonjouR");

        // Affiche ces deux "PoidsMouche"
        lFlyweight2.afficher("Contexte2");
        lFlyweight2Bis.afficher("Contexte2Bis");

        // Demande et affiche un "PoidsMouche" non partagé
        PoidsMouche lFlyweight3 = lFlyweightFactory.getPoidsMouche("Valeur1", "Valeu
        lFlyweight3.afficher(null);

        // Affichage :
        // --> Retourne un PoidsMouche (Bonjour) partagé déjà existant
        // --> Retourne un PoidsMouche (Bonjour) partagé déjà existant
    }
}

```

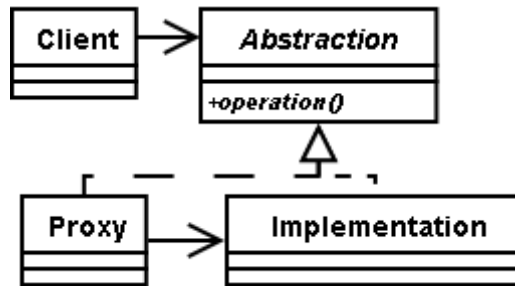


```

// PoidsMouche avec la valeur : Bonjour et contexte : Contexte1
// PoidsMouche avec la valeur : Bonjour et contexte : Contexte1Bis
// lFlyweight1 == lFlyweight1Bis : true
// --> Retourne un PoidsMouche (Bonjour) partagé non déjà existant
// --> Retourne un PoidsMouche (Bonjour) partagé déjà existant
// PoidsMouche avec la valeur : Bonjour et contexte : Contexte2
// PoidsMouche avec la valeur : Bonjour et contexte : Contexte2Bis
// --> Retourne un PoidsMouche (Valeur1, Valeur2) non partagé
// PoidsMouche avec la valeur1 : Valeur1 avec la valeur2 : Valeur2
}
}

```

## V-G. Proxy (Proxy ou Surrogate) ▲



### OBJECTIFS :

Fournir un intermédiaire entre la partie cliente et un objet pour contrôler les accès à ce dernier.

### RAISONS DE L'UTILISER :

Les opérations d'un objet sont coûteuses en temps ou sont soumises à une gestion de droits d'accès. Il est nécessaire de contrôler l'accès à un objet.

Cela peut être un système de chargement d'un document. Le document est très lourd à charger en mémoire ou il faut certaines habilitations pour accéder à ce document.

L'objet réel (système de chargement classique) est l'implémentation. L'intermédiaire entre l'implémentation et la partie cliente est le proxy. Le proxy fournit la même interface que l'implémentation. Mais il ne charge le document qu'en cas de réel besoin (pour l'affichage par exemple) ou n'autorise l'accès que si les conditions sont satisfaites.

### RESULTAT :

Le Design Pattern permet d'isoler le comportement lors de l'accès à un objet.

### RESPONSABILITES :

- **Abstraction** : définit l'interface des classes **Implementation** et **Proxy**.

- **Implementation** : implémente l'interface. Cette classe définit l'objet que l'objet **Proxy** représente.
- **Proxy** : fournit un intermédiaire entre la partie cliente et l'objet **Implementation**. Cet intermédiaire peut avoir plusieurs buts (synchronisation, contrôle d'accès, cache, accès distant, ...). Dans l'exemple, la classe **Proxy** n'instancie un objet **Implementation** qu'en cas de besoin pour appeler la méthode correspondante de la classe **Implementation**.
- La partie cliente appelle la méthode **operation()** de l'objet **Proxy**.

#### IMPLEMENTATION JAVA :

Abstraction.java

Sélectionnez

```
/**
 * Définit l'interface
 */
public interface Abstraction {

    /**
     * Méthode pour laquelle on souhaite un "Proxy"
     */
    public void afficher();
}
```

Implementation.java

Sélectionnez

```
/**
 * Implémentation de l'interface.
 * Définit l'objet représenté par le "Proxy"
 */
public class Implementation implements Abstraction {

    public void afficher() {
        System.out.println("Méthode afficher() de la classe d'implémentation");
    }
}
```

Proxy.java

Sélectionnez

```
/**
 * Intermédiaire entre la partie cliente et l'implémentation
 */
public class Proxy implements Abstraction {

    /**
     * Instancie l'objet "Implementation", pour appeler
     * la vraie implémentation de la méthode.
     */
    public void afficher() {
        System.out.println("--> Méthode afficher() du Proxy : ");
        System.out.println("--> Création de l'objet Implementation au besoin");
    }
}
```

```
        Implementation lImplementation = new Implementation();
        System.out.println("--> Appel de la méthode afficher() de l'objet Implementa
        lImplementation.afficher();
    }
}
```

ProxyPatternMain.java


Sélectionnez

```
public class ProxyPatternMain {

    public static void main(String[] args) {
        // Création du "Proxy"
        Abstraction lProxy = new Proxy();

        // Appel de la méthode du "Proxy"
        lProxy.afficher();

        // Affichage :
        // --> Méthode afficher() du Proxy :
        // --> Création de l'objet Implementation au besoin
        // --> Appel de la méthode afficher() de l'objet Implementation
        // Méthode afficher() de la classe d'implementation
    }
}
```

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.

**Quels sont les frameworks que vous aimeriez apprendre en 2019 ?**

**Quels sont les langages de programmation que vous voulez apprendre en 2019 ?**

**Quels sont vos environnements de développement intégrés (EDI) préférés en 2018 ? Et pourquoi ?**

**Apprendre 23 principes pour écrire du code lisible, un tutoriel d'Artur Smiarowski**

---

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](http://www.developpez.com)