Design Patterns du Gang of Four appliqués à Java

UML

par Régis POUILLER (Home)

Date de publication : 16/09/2008

Dernière mise à jour :

Cet article a pour objectif de présenter les **Design Patterns** du "Gang of Four" avec des exemples concis appliqués à Java.
Chaque Design Pattern est présenté avec son **diagramme de classes**, ses objectifs, les motivations pour l'utiliser, les responsabilités des classes le constituant, puis une **implémentation** simple.
Un discussion a été ouverte pour la publication de cet article, pour les commentaires.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

I - INTRODUCTION

A l'origine les **Design Patterns** (**Patron de conception** en Français) sont issus des travaux de l'architecte Christopher Alexander. Ces travaux sont une capitalisation d'expérience qui ont mis en évidence des patrons en architecture des bâtiments.

Sur le même principe, en 1995, le livre "Design Patterns -- Elements of Reusable Object-Oriented Software" du GoF, Gang Of Four (**Erich Gamma**, **Richard Helm**, **Ralph Johnson** et **John Vlissides**), présente 23 Design Patterns. Dans ce livre, chacun des Design Patterns est accompagné d'exemple en C++ et Smalltalk.

En **architecture** des logiciels, un Design Pattern est la description d'une solution à un problème de conception. Pour faire l'objet d'un Design Pattern, une solution doit être réutilisable. On dit que le Design Pattern est "prouvé" s'il a pu être utilisé dans au moins 3 cas.

Les Design Patterns permettent d'améliorer la qualité de développement et d'en diminuer la durée. En effet, leur application réduit les couplages (points de dépendance) au sein d'une application, apporte de la souplesse, favorise la maintenance et d'une manière générale aide à respecter de "**bonnes pratiques**" de développement.

Les Design Patterns sont classés en trois catégories :

- créationnels : qui définissent des mécanismes pour l'instanciation et/ou l'initialisation d'objets
- structuraux : qui organisent les interfaces/classes entre elles
- comportementaux : qui définissent la communication entre les classes et leurs responsabilités

Le but de ce document est de présenter succintement chaque Design Pattern du GoF et d'y associer un exemple d'implémentation Java.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

II - BIBLIOGRAPHIE

Design Patterns -- Elements of Reusable Object-Oriented Software : il s'agit du livre écrit par le "Gang of Four". Il présente toutes les subtilités des Design Patterns. C'est sans doute le livre le plus complet sur le sujet. Le lien conduit à la critique de l'édition en Anglais. Les critiques sur l'édition en Français trouvées par ailleurs sur Internet indiquent des erreurs de traduction. J'ai utilisé le livre en Anglais pour la plupart de mes recherches et vérifications.

Design Patterns par la pratique : livre très facile à lire. Il présente comme inconvénient de ne pas couvrir tous les Design Patterns du GoF. Mis à part cela, il est très clair dans l'explication des Design Patterns qu'il présente. Et il est en Français ;-).

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

III - AUTRES RESSOURCES SUR DEVELOPPEZ.COM

Le dictionnaire des développeurs (partie conception)

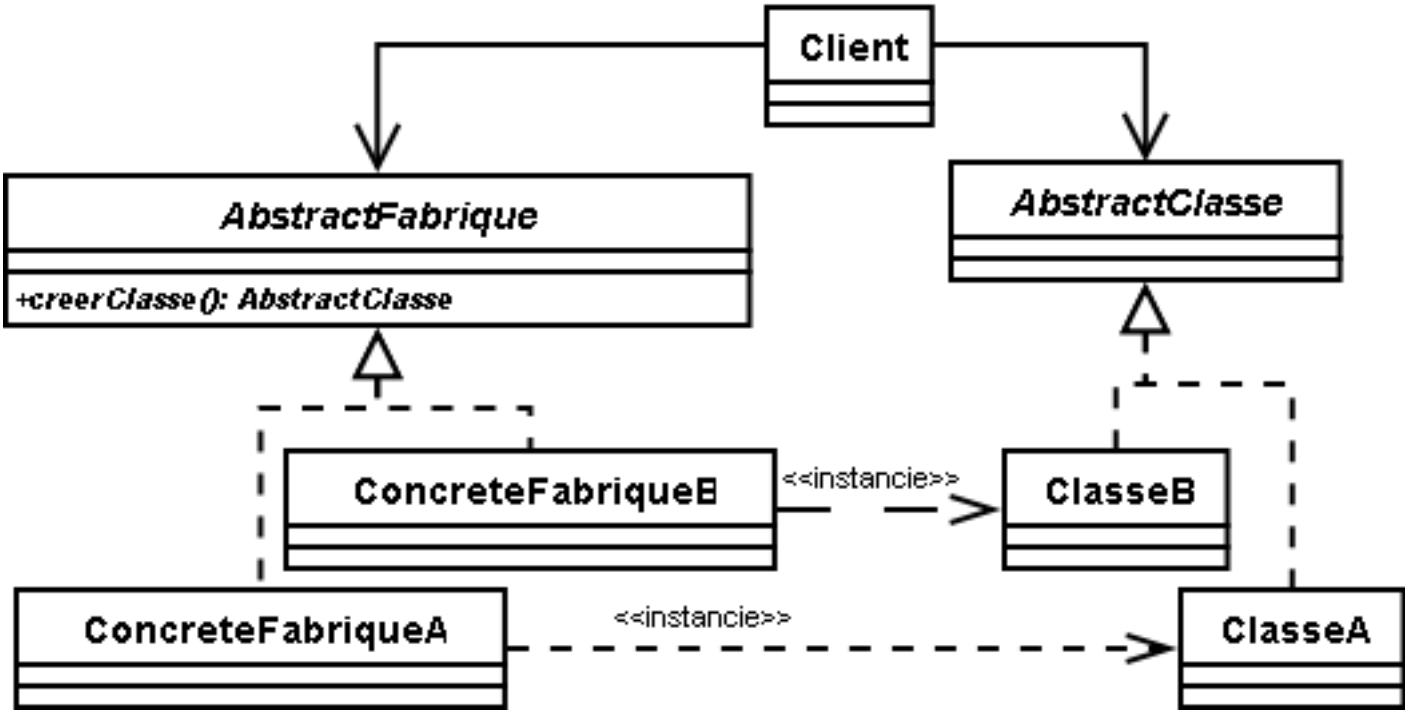
Description de design patterns par Sébastien MERIC

Souplesse et modularité grâce aux Design Patterns par Pierre Caboche

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

IV - CREATIONNELS (CREATIONAL PATTERNS)

IV-A - Fabrique abstraite (Abstract Factory ou Kit)



LIEN VERS LE DICTIONNAIRE DES DEVELOPPEURS :

Abstract Factory

AUTRE RESSOURCE SUR DEVELOPPEZ.COM :

Le kit par Sébastien MERIC

OBJECTIFS :

Fournir une interface pour créer des objets d'une même famille sans préciser leurs **classes concrètes**.

RAISONS DE L'UTILISER :

Le système utilise des objets qui sont regroupés en famille. Selon certains critères, le système utilise les objets d'une famille ou d'une autre. Le système doit utiliser ensemble les objets d'une famille.

Cela peut être le cas des éléments graphiques d'un look and feel : pour un look and feel donné, tous les graphiques créés doivent être de la même famille.

La partie cliente manipulera les **interfaces** des **objets** ; ainsi il y aura une indépendance par rapport aux classes concrètes. Chaque fabrique concrète permet d'instancier une famille d'objets (éléments graphiques du même look and feel) ; ainsi la notion de famille d'objets est renforcée.

RESULTAT :

Le Design Pattern permet d'isoler l'appartenance à une famille de classes.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

RESPONSABILITES :

- **AbstractFabrique** : définit l'interface des méthodes de création. Dans le diagramme, il n'y a qu'une méthode de création pour un objet d'une classe. Mais, le diagramme sous-entend un nombre indéfini de méthodes pour un nombre indéfini de classes.
- **ConcreteFabriqueA** et **ConcreteFabriqueB** : implèmentent l'interface et instancient la classe concrète appropriée.
- **AbstractClasse** : définit l'interface d'un type d'objet instancié.
- **ClasseA** et **ClasseB** : sont des sous-classes concrètes d'**AbstractClasse**. Elles sont instanciées par les ConcreteFabrique.
- La partie cliente fait appel à une Fabrique pour obtenir une nouvelle instance d'**AbstractClasse**. L'instanciation est transparente pour la partie cliente. Elle manipule une **AbstractClasse**.

IMPLEMENTATION JAVA :

AbstractClasse.java
```
/**
 * Définit l'interface d'un type d'objet instancié.
 */
public interface AbstractClasse {

    /**
     * Méthode d'affichage du nom de la classe.
     */
    public void afficherClasse();
}
```

ClasseA.java
```
/**
 * Sous classe de AbstractClasse
 * Cette classe est instanciée par ConcreteFabriqueA
 */
public class ClasseA implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage
     */
    public void afficherClasse() {
        System.out.println("Objet de classe 'ClasseA'");
    }
}
```

ClasseB.java
```
/**
 * Sous classe de AbstractClasse
 * Cette classe est instanciée par ConcreteFabriqueB
 */
public class ClasseB implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage
     */
    public void afficherClasse() {
        System.out.println("Objet de classe 'ClasseB'");
    }
}
```

AbstractFabrique.java
```
/**
 * Définit l'interface de la méthode de création.
 */
public interface AbstractFabrique {

    /**
     * Méthode de création d'un objet de classe AbstractClasse.
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

AbstractFabrique.java

```
     * @return L'objet créé.
     */
    public AbstractClasse creerClasse();
}
```

ConcreteFabriqueA.java

```
/**
 * Implémente l'interface "AbstractFabrique".
 */
public class ConcreteFabriqueA implements AbstractFabrique {

    /**
     * La méthode de création instancie un objet "ClasseA".
     * @return Un objet "ClasseA" qui vient d'être créé.
     */
    public AbstractClasse creerClasse() {
        return new ClasseA();
    }
}
```

ConcreteFabriqueB.java

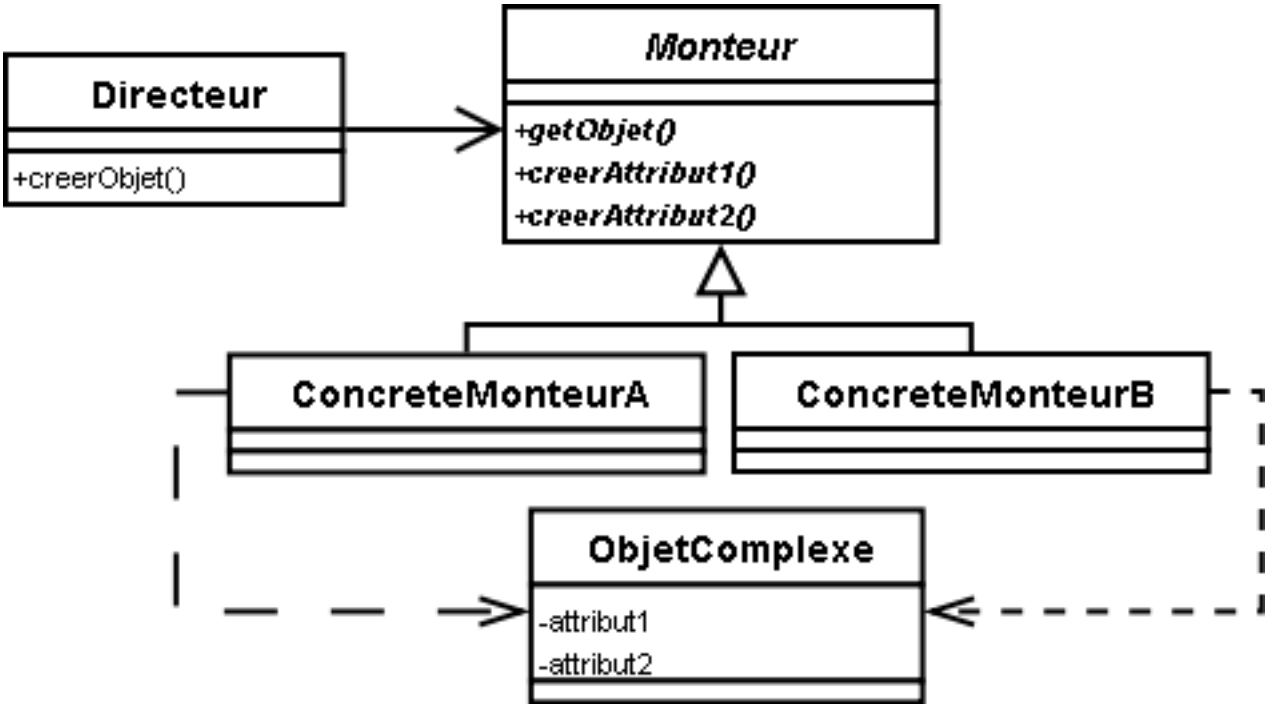
```
/**
 * Implémente l'interface "AbstractFabrique".
 */
public class ConcreteFabriqueB implements AbstractFabrique {

    /**
     * La méthode de création instancie un objet "ClasseB".
     * @return Un objet "ClasseB" qui vient d'être créé.
     */
    public AbstractClasse creerClasse() {
        return new ClasseB();
    }
}
```

AbstractFactoryPatternMain.java

```
public class AbstractFactoryPatternMain {

    public static void main(String[] args) {
        // Création des fabriques
        AbstractFabrique lFactory1 = new ConcreteFabriqueA();
        AbstractFabrique lFactory2 = new ConcreteFabriqueB();

        // Création de deux "AbstractClasse" à partir de chaque fabrique
        AbstractClasse lClasse1 = lFactory1.creerClasse();
        AbstractClasse lClasse2 = lFactory2.creerClasse();

        // Appel d'une méthode d'"AbstractClasse" qui affiche un message
        // Ce message permet de vérifier que chaque "AbstractClasse"
        // est en fait une classe différente
        lClasse1.afficherClasse();
        lClasse2.afficherClasse();

        // -------------------------
        // Affichage :
        // Objet de classe 'ClasseA'
        // Objet de classe 'ClasseB'
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

IV-B - Monteur (Builder)



LIEN VERS LE DICTIONNAIRE DES DEVELOPPEURS :

Builder

AUTRE RESSOURCE SUR DEVELOPPEZ.COM :

Le monteur par Sébastien MERIC

OBJECTIFS :

- Séparer la construction d'un **objet complexe** de sa représentation.
- Permettre d'obtenir des représentations différentes avec le même procédé de construction.

RAISONS DE L'UTILISER :

Le système doit instancier des objets complexes. Ces objets complexes peuvent avoir des représentations différentes.

Cela peut être le cas des différentes fenêtres d'une IHM. Elles comportent des éléments similaires (titre, boutons), mais chacune avec des particularités (libellés, comportements).

Afin d'obtenir des représentations différentes (fenêtres), la partie cliente passe des monteurs différents au directeur. Le directeur appelera des **méthodes** du monteur retournant les éléments (titre, bouton). Chaque implémentation des méthodes des monteurs retourne des éléments avec des différences (libellés, comportements).

RESULTAT :

Le Design Pattern permet d'isoler des variations de représentations d'un objet.

RESPONSABILITES :

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

- **ObjetComplexe** : est la classe d'objet complexe à instancier.
- **Monteur** : définit l'interface des méthodes permettant de créer les différentes parties de l'objet complexe.
- **ConcreteMonteurA** et **ConcreteMonteurB** : implémentent les méthodes permettant de créer les différentes parties. Les classes conservent l'objet durant sa construction et fournissent un moyen de récupérer le résultat de la construction.
- **Directeur** : construit un objet en appelant les méthodes d'un **Monteur**.
- La partie cliente instancie un **Monteur**. Elle le fournit au **Directeur**. Elle appelle la méthode de contruction du **Directeur**.

IMPLEMENTATION JAVA :

ObjetComplexe.java

```
/**
 * L'objet complexe
 * Dans l'exemple, on pourrait considérer
 * l'attribut1 comme un libellé
 * et l'attribut2 comme une dimension
 * La classe de l'attribut2 peut varier selon le "Monteur"
 */
public class ObjetComplexe {

    // Les attributs de l'objet complexe
    private String attribut1;
    private Number attribut2;

    // Les méthodes permettant de fixes les attributs
    public void setAttribut1(String pAttribut1) {
        attribut1 = pAttribut1;
    }

    public void setAttribut2(Number pAttribut2) {
     attribut2 = pAttribut2;
    }

    /**
     * Méthode permettant d'afficher l'état de l'objet
     * afin de permettre de mettre en évidence
     * les différences de "montage".
     */
    public void afficher() {
        System.out.println("Objet Complexe : ");
        System.out.println("\t- attribut1 : " + attribut1);
        System.out.println("\t- attribut2 : " + attribut2);
        System.out.println("\t- classe de l'attribut2 : " + attribut2.getClass());
    }
}
```

Monteur.java

```
/**
 * Définit l'interface des méthodes permettant
 * de créer les différentes partie
 * de l'objet complexe.
 */
public abstract class Monteur {

    protected ObjetComplexe produit;

    /**
     * Crée un nouveau produit
     * Aucune des parties n'est créée
     * à ce moment là.
     */
    public void creerObjet() {
        produit = new ObjetComplexe();
    }

    /**
     * Retourne l'objet une fois fini.
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Monteur.java

```
     */
    public ObjetComplexe getObjet() {
        return produit;
    }

    // Les méthodes de création des parties

    public abstract void creerAttribut1(String pAttribut1);
    public abstract void creerAttribut2(double pAttribut2);
}
```

ConcreteMonteurA.java

```
/**
 * Implémente les méthodes permettant
 * de créer les parties de l'objet complexe.
 */
public class ConcreteMonteurA extends Monteur {

    /**
     * Méthode de création de l'attribut attribut1
     * Précise que l'attibut2 représente une dimention en centimètres
     */
    public void creerAttribut1(String pAttribut1) {
        produit.setAttribut1(pAttribut1 + " (avec dimension en centimètre)");
    }

    /**
     * Méthode de création de l'attribut attribut2
     * Stocke la valeur dans un Float sans modification
     */
    public void creerAttribut2(double pAttribut2) {
        produit.setAttribut2(new Float(pAttribut2));
    }
}
```

ConcreteMonteurB.java

```
/**
 * Implémente les méthodes permettant
 * de créer les parties de l'objet complexe.
 */
public class ConcreteMonteurB extends Monteur {

    /**
     * Méthode de création de l'attribut attribut1
     * Précise que l'attibut2 représente une dimention en pouces
     */
    public void creerAttribut1(String pAttribut1) {
        produit.setAttribut1(pAttribut1 + " (avec dimension en pouces)");
    }

    /**
     * Méthode de création de l'attribut attribut2
     * Stocke la valeur dans un Doucle en le convertissant en pouces
     */
    public void creerAttribut2(double pAttribut2) {
        produit.setAttribut2(new Double(pAttribut2 * 2.54));
    }
}
```

Directeur.java

```
/**
 * Contruit un objet en appelant les méthodes d'un "Monteur".
 */
public class Directeur {
    private Monteur monteur;

    Directeur(Monteur pMonteur) {
        monteur = pMonteur;
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Directeur.java

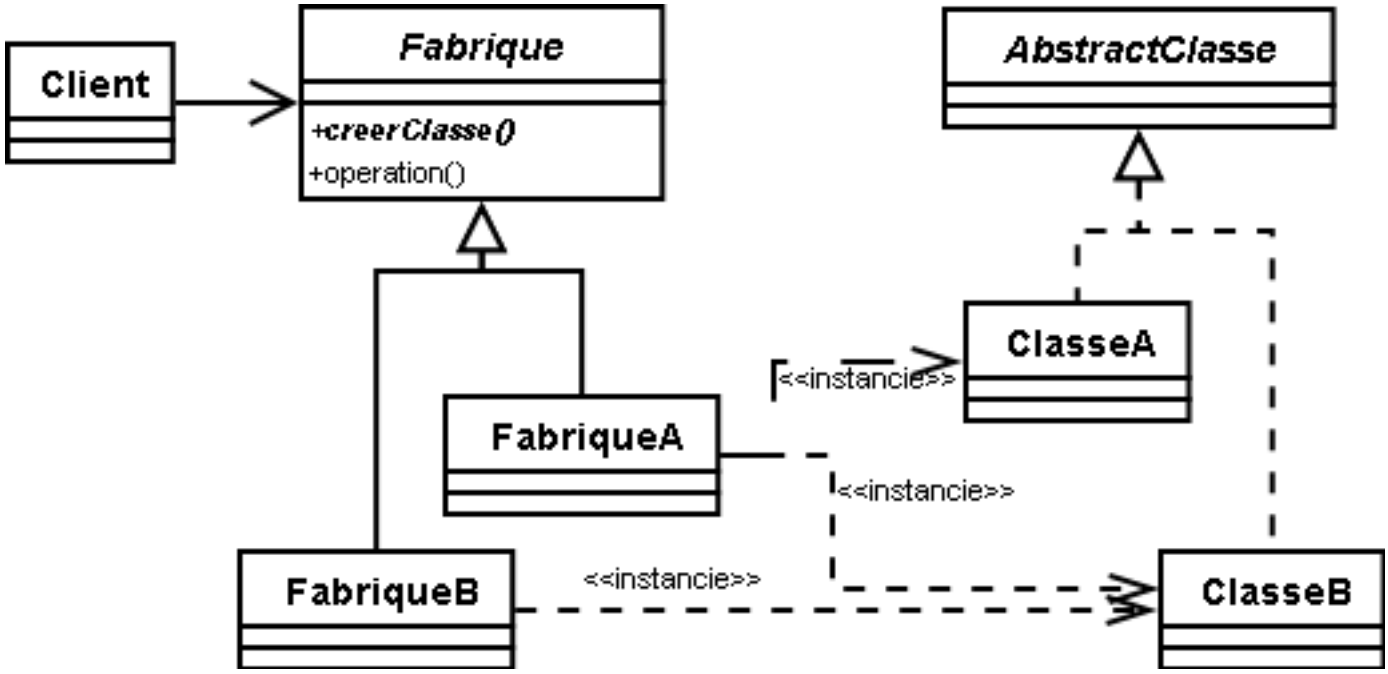
```
    }

    /**
     * Crée un objet.
     * Appelle les méthodes de création
     * des parties du "Monteur".
     */
    public ObjetComplexe creerObjet() {
        monteur.creerObjet();

        monteur.creerAttribut1("libelle de l'objet");
        monteur.creerAttribut2(12);

        return monteur.getObjet();
    }
}
```

BuilderPatternMain.java

```
public class BuilderPatternMain {

    public static void main(String[] args) {
        // Instancie les objets directeur et monteur
        Monteur lMonteurA = new ConcreteMonteurA();
        Directeur lDirecteurA = new Directeur(lMonteurA);
        Monteur lMonteurB = new ConcreteMonteurB();
        Directeur lDirecteurB = new Directeur(lMonteurB);

        // Appel des différentes méthodes de création
        ObjetComplexe lProduitA = lDirecteurA.creerObjet();
        ObjetComplexe lProduitB = lDirecteurB.creerObjet();

        // Demande l'affichage des valeurs des objets
        // pour visualiser les différences de composition
        lProduitA.afficher();
        lProduitB.afficher();

        // Affichage :
        // Objet Complexe :
        //         - attribut1 : libelle de l'objet (avec dimension en centimètre)
        //         - attribut2 : 12.0
        //         - classe de l'attribut2 : class java.lang.Float
        // Objet Complexe :
        //         - attribut1 : libelle de l'objet (avec dimension en pouces)
        //         - attribut2 : 30.48
        //         - classe de l'attribut2 : class java.lang.Double
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

IV-C - Fabrique (Factory Method ou Virtual Constructor)



OBJECTIFS :

- Définir une interface pour la création d'un objet, mais laisser les **sous-classes** décider quelle classe instancier.
- Déléguer l'instanciation aux sous-classes.

RAISONS DE L'UTILISER :

Dans le fonctionnement d'une **classe**, il est nécessaire de créer une **instance**. Mais, au niveau de cette classe, on ne connaît pas la classe exacte à instancier.

Cela peut être le cas d'une classe réalisant une sauvegarde dans un flux sortant, mais ne sachant pas s'il s'agit d'un fichier ou d'une sortie sur le réseau.

La classe possède une méthode qui retourne une instance (interface commune au fichier ou à la sortie sur le réseau). Les autres méthodes de la classe peuvent effectuer les opérations souhaitées sur l'instance (écriture, fermeture). Les sous-classes déterminent la classe de l'instance créée (fichier, sortie sur le réseau). Une variante du Pattern existe : la **méthode** de création choisit la classe de l'instance à créer en fonction de **paramètres** en entrée de la méthode ou selon des variables de contexte.

RESULTAT :

Le Design Pattern permet d'isoler l'instanciation d'une **classe concrète**.

RESPONSABILITES :

- **AbstractClasse** : définit l'interface de l'objet instancié.
- **ClasseA** et **ClasseB** : sont des sous-classes concrètes d'**AbstractClasse**. Elles sont instanciées par les classes **Fabrique**.
- **Fabrique** : déclare une méthode de création (**creerClasse**). C'est cette méthode qui a la responsabilité de l'instanciation d'un objet **AbstractClasse**. Si d'autres méthodes de la classe ont besoin d'une instance de **AbstractClasse**, elles font appel à la méthode de création. Dans l'exemple, la méthode **operation()** utilise

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

une instance de **AbstractClasse** et fait donc appel à la méthode **creerClasse**. La méthode de création peut être paramétrée ou non. Dans l'exemple, elle est paramétrée, mais le paramètre n'est significative que pour la **FabriqueA**.

- **FabriqueA** et **FabriqueB** : substituent la méthode de création. Elles implémentent une version différente de la méthode de création.
- La partie cliente utilise une sous-classe de **Fabrique**.

IMPLEMENTATION JAVA :

AbstractClasse.java
```
/**
 * Définit l'interface de l'objet instancié.
 */
public interface AbstractClasse {

    /**
     * Méthode permettant d'afficher le nom de la classe.
     * Cela permet de mettre en evidence la classe créée.
     */
    public void afficherClasse();
}
```

ClasseA.java
```
/**
 * Sous-class de AbstractClasse.
 */
public class ClasseA implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage.
     * Indique qu'il s'agit d'un objet de classe ClasseA
     */
    public void afficherClasse() {
        System.out.println("Objet de classe 'ClasseA'");
    }
}
```

ClasseB.java
```
/**
 * Sous-class de AbstractClasse.
 */
public class ClasseB implements AbstractClasse {

    /**
     * Implémentation de la méthode d'affichage.
     * Indique qu'il s'agit d'un objet de classe ClasseB
     */
    public void afficherClasse() {
        System.out.println("Objet de classe 'ClasseB'");
    }
}
```

Fabrique.java
```
/**
 * Déclare la méthode de création.
 */
public abstract class Fabrique {

    private boolean pIsClasseA = false;

    /**
     * Méthode de création
     */
    public abstract AbstractClasse creerClasse(boolean pIsClasseA);

    /**
     * Méthode appelant la méthode de création.
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Fabrique.java

```
     * Puis, effectuant une opération.
     */
    public void operation() {
        // Change la valeur afin de varier le paramètre
        // de la méthode de création
        pIsClasseA = !pIsClasseA;

        // Récupère une instance de classe "AbstractClasse"
        AbstractClasse lClasse = creerClasse(pIsClasseA);

        // Appel la méthode d'affichage de la classe
        // afin de savoir la classe concrète
        lClasse.afficherClasse();
    }
}
```

FabriqueA.java

```
/**
 * Substitue la méthode "creerClasse".
 * Instancie un objet "ClasseA".
 */
public class FabriqueA extends Fabrique {

    /**
     * Méthode de création
     * La méthode retourne un objet ClasseA, si le paramètre est true.
     * La méthode retourne un objet ClasseB, sinon.
     * @return Un objet de classe ClasseA ou ClasseB.
     */
    public AbstractClasse creerClasse(boolean pIsClasseA) {
        if(pIsClasseA) {
            return new ClasseA();
        }
        else {
            return new ClasseB();
        }
    }
}
```

FabriqueB.java

```
/**
 * Substitue la méthode "creerClasse".
 * Instancie un objet "ClasseB".
 */
public class FabriqueB extends Fabrique {

    /**
     * Méthode de création
     * La méthode ne tient pas compte du paramètre
     * et instancie toujours un objet "ClasseB"
     * @return Un objet de classe ClasseB.
     */
    public AbstractClasse creerClasse(boolean pIsClasseA) {
        return new ClasseB();
    }
}
```

FactoryMethodPatternMain.java

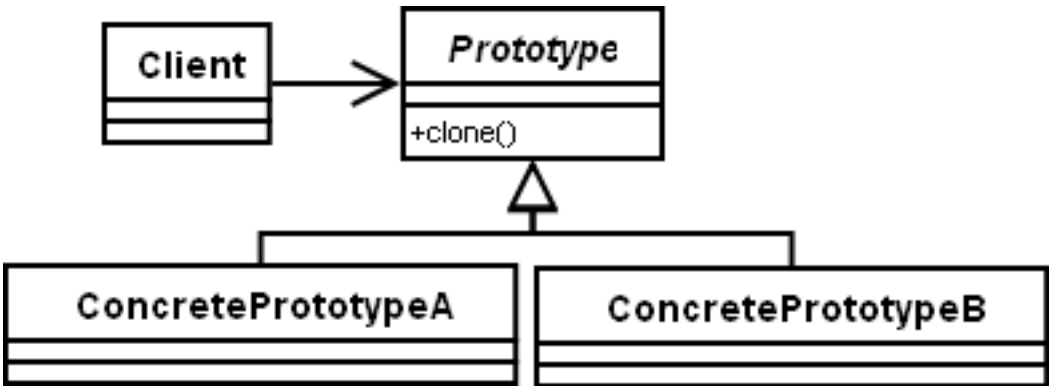
```
public class FactoryMethodPatternMain {

    public static void main(String[] args) {
        // Création des fabriques
        Fabrique lFactoryA = new FabriqueA();
        Fabrique lFactoryB = new FabriqueB();

        // L'appel de cette méthode avec FabriqueA provoquera
        // l'instanciation de deux classes différentes
        System.out.println("Avec la FabriqueA : ");
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

FactoryMethodPatternMain.java

```
        lFactoryA.operation();
        lFactoryA.operation();
        lFactoryA.operation();
        // L'appel de cette méthode avec FabriqueB provoquera
        // toujours l'instanciation de la même classe
        System.out.println("Avec la FabriqueB : ");
        lFactoryB.operation();
        lFactoryB.operation();
        lFactoryB.operation();

        // Affichage :
        // Avec la FabriqueA :
        // Objet de classe 'ClasseA'
        // Objet de classe 'ClasseB'
        // Objet de classe 'ClasseA'
        // Avec la FabriqueB :
        // Objet de classe 'ClasseB'
        // Objet de classe 'ClasseB'
        // Objet de classe 'ClasseB'
    }
}
```

IV-D - Prototype (Prototype)



OBJECTIFS :

- Spécifier les genres d'objet à créer en utilisant une instance comme prototype.
- Créer un nouvel objet en copiant ce prototype.

RAISONS DE L'UTILISER :

Le système doit créer de nouvelles instances, mais il ignore de quelle classe. Il dispose cependant d'instances de la classe désirée.

Cela peut être le cas d'un logiciel de DAO comportant un copier-coller. L'utilisateur sélectionne un élément graphique (cercle, rectangle, ...), mais la classe traitant la demande de copier-coller ne connaît pas la classe exacte de l'élément à copier.

La solution est de disposer d'une duplication des instances (élément à copier : cercle, rectangle). La duplication peut être également intéressante pour les performances (la duplication est plus rapide que l'instanciation).

RESULTAT :

Le Design Pattern permet d'isoler l'appartenance à une classe.

RESPONSABILITES :

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

- **Prototype** : définit l'interface de duplication de soi-même.
- **ConcretePrototypeA** et **ConcretePrototypeB** : sont des sous-classes concrètes de **Prototype**. Elles implémentent l'inteface de duplication.
- La partie cliente appelle la méthode **clone()** de la classe **Prototype**. Cette méthode retourne un double de l'instance.

IMPLEMENTATION JAVA :

Prototype.java
```
/**
 * Définit l'interface de l'objet à dupliquer.
 */
public abstract class Prototype implements Cloneable {

    protected String texte;

    /**
     * Constructeur de la classe.
     * @param pTexte
     */
    public Prototype(String pTexte) {
        texte = pTexte;
    }

    /**
     * La méthode clone est protected dans Object.
     * On doit la substituer pour la rendre visible.
     * De plus, il faut que la classe implémente l'interface Cloneable.
     * Depuis Java5, on peut retourner un sous-type de Object.
     */
    public Prototype clone() throws CloneNotSupportedException {
        return (Prototype)super.clone();
    }

    public void setTexte(String pTexte) {
        texte = pTexte;
    }

    /**
     * Méthode d'affichage des informations de l'objet.
     */
    public abstract void affiche();
}
```

ConcretePrototypeA.java
```
/**
 * Sous-class de Prototype.
 */
public class ConcretePrototypeA extends Prototype {

    public ConcretePrototypeA(String pTexte) {
        super(pTexte);
    }

    /**
     * Méthode d'affichage.
     * Indique que c'est un objet de classe ConcretePrototypeA
     * et la valeur de l'attribut texte.
     */
    public void affiche() {
        System.out.println("ConcretePrototypeA avec texte : " + texte);
    }
}
```

ConcretePrototypeB.java
```
/**
 * Sous-class de Prototype.
 */
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

ConcretePrototypeB.java

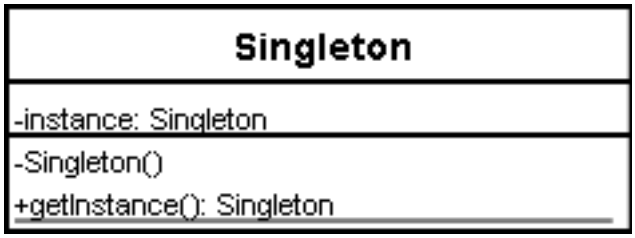
```
public class ConcretePrototypeB extends Prototype {

    public ConcretePrototypeB(String pTexte) {
        super(pTexte);
    }

    /**
     * Méthode d'affichage.
     * Indique que c'est un objet de classe ConcretePrototypeA
     * et la valeur de l'attribut texte.
     */
    public void affiche() {
        System.out.println("ConcretePrototypeB avec texte : " + texte);
    }
}
```

PrototypePatternMain.java

```
public class PrototypePatternMain {

    public static void main(String[] args) throws CloneNotSupportedException {
        // Instancie un objet de classe ConcretePrototypeA
        // et un autre de classe ConcretePrototypeB
        // de "manière traditionnelle".
        Prototype lPrototypeA = new ConcretePrototypeA("Original");
        Prototype lPrototypeB = new ConcretePrototypeB("Original");

        // Duplique les objets précédemment créés/
        Prototype lPrototypeAClone = lPrototypeA.clone();
        Prototype lPrototypeBClone = lPrototypeB.clone();

        // Affiche les objets :
        // les clones sont identiques aux originaux
        lPrototypeA.affiche();
        lPrototypeAClone.affiche();
        lPrototypeB.affiche();
        lPrototypeBClone.affiche();

        // Modifie les clones
        lPrototypeAClone.setTexte("Clone (enfait)");
        lPrototypeBClone.setTexte("Clone (enfait)");

        // Met en évidence que les clones
        // sont bien des instances à part.
        lPrototypeA.affiche();
        lPrototypeAClone.affiche();
        lPrototypeB.affiche();
        lPrototypeBClone.affiche();

        // Affichage :
        // ConcretePrototypeA avec texte : Original
        // ConcretePrototypeA avec texte : Original
        // ConcretePrototypeB avec texte : Original
        // ConcretePrototypeB avec texte : Original
        // ConcretePrototypeA avec texte : Original
        // ConcretePrototypeA avec texte : Clone (enfait)
        // ConcretePrototypeB avec texte : Original
        // ConcretePrototypeB avec texte : Clone (enfait)
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

IV-E - Singleton (Singleton)



LIEN VERS LE DICTIONNAIRE DES DEVELOPPEURS :

Singleton

AUTRES RESSOURCES SUR DEVELOPPEZ.COM :

Le singleton par Sébastien MERIC

Le Singleton en environnement Multithread par Christophe Jollivet

OBJECTIFS :

- Restreindre le nombre d'**instances** d'une classe à une et une seule.
- Fournir une **méthode** pour accéder à cette instance unique.

RAISONS DE L'UTILISER :

La classe ne doit avoir qu'une seule instance.

Cela peut être le cas d'une ressource système par exemple.

La classe empêche d'autres classes de l'instancier. Elle possède la seule instance d'elle-même et fournit la seule méthode permettant d'accéder à cette instance.

RESULTAT :

Le Design Pattern permet d'isoler l'unicité d'une instance.

RESPONSABILITES :

Singleton doit restreindre le nombre de ses propres instances à une et une seule. Son constructeur est privé : cela empêche les autres classes de l'instancier. La classe fournit la méthode statique **getInstance()** qui permet d'obtenir l'instance unique.

IMPLEMENTATION JAVA :

Singleton.java
```
public class Singleton {

    /**
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
     */
    private Singleton() {
    }
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Singleton.java

```
    /**
     * SingletonHolder est chargé à la première exécution de
     * Singleton.getInstance() ou au premier accès à SingletonHolder.instance ,
     * pas avant.
     */
    private static class SingletonHolder {
        private final static Singleton instance = new Singleton();
    }

    /**
     * Méthode permettant d'obtenir l'instance unique.
     * @return L'instance du singleton.
     */
    public static Singleton getInstance() {
        return SingletonHolder.instance;
    }
}
```

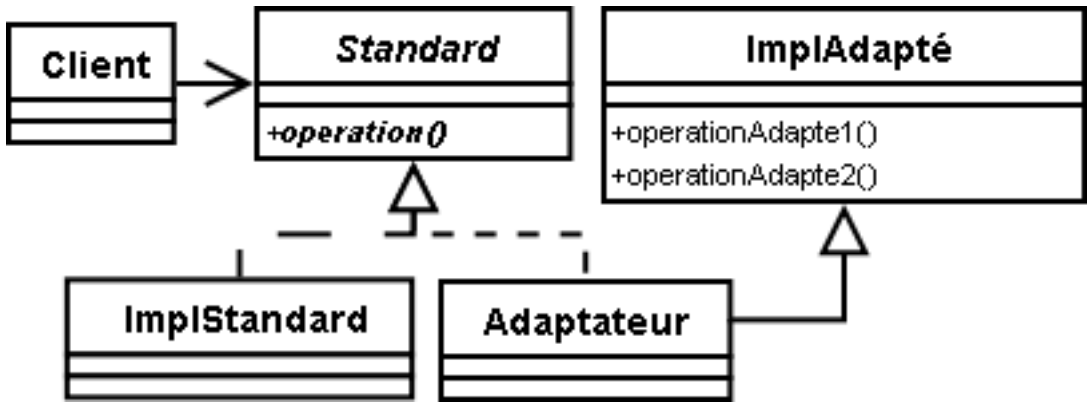

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

V - STRUCTURAUX (STRUCTURAL PATTERNS)

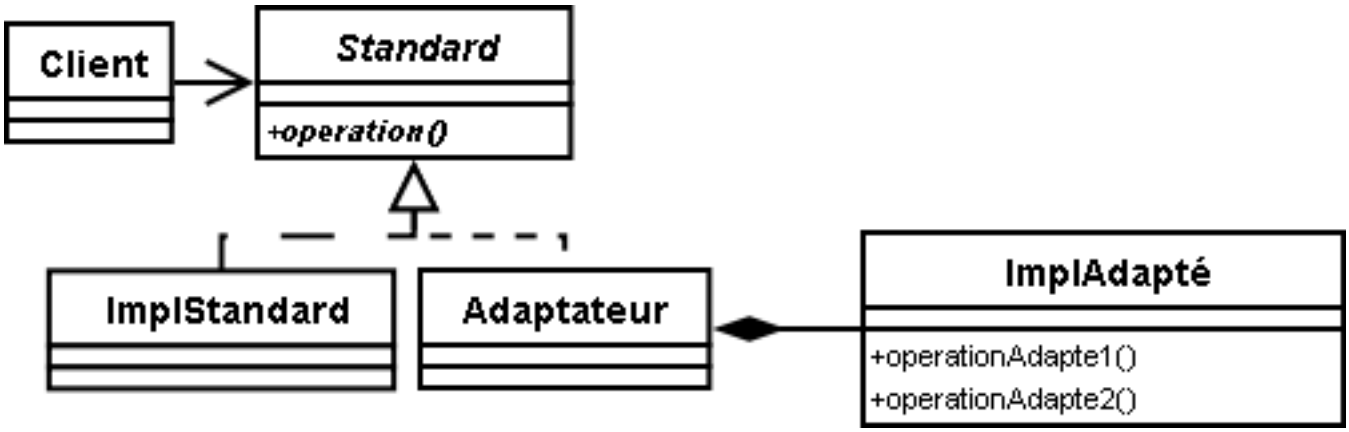
V-A - Adaptateur (Adapter ou Wrapper)

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.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

Cela peut être le cas d'un driver bas niveau pour de l'informatique embarquée. Le driver fournit 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 stantard 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
```
/**
 * 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
```
/**
 * 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
```
/**
 * 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 fonctionalité définie dans l'interface,
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

ImplAdapte.java

```
     * 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)

```
/**
 * 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)

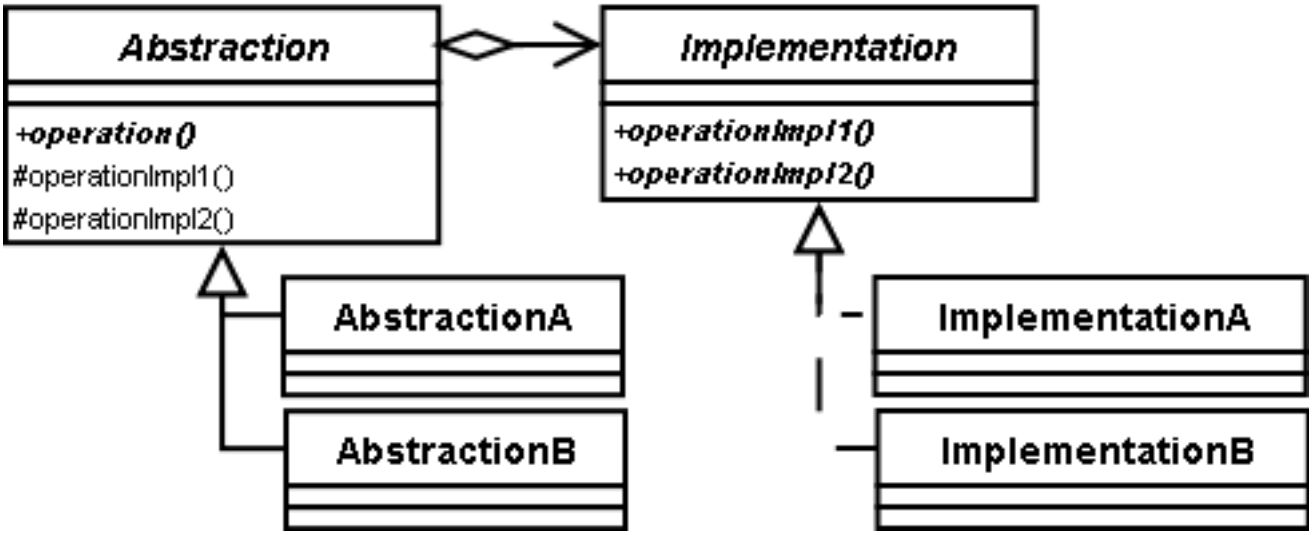
```
/**
 * Adapte l'implémentation non standard avec la composition.
 */
public class Adaptateur implements Standard {

    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

```
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
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

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 fournit par l'interface de l'abstraction.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

IMPLEMENTATION JAVA :

Implementation.java
```
/**
 * 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
```
/**
 * 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
```
/**
 * 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
```
/**
 * 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
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Abstraction.java
```
     * @param pMessage
     */
    protected void operationImpl2(Integer pNombre) {
        implementation.operationImpl2(pNombre);
    }
}
```

AbstractionA.java
```
/**
 * 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
```
/**
 * 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");
    }
}
```

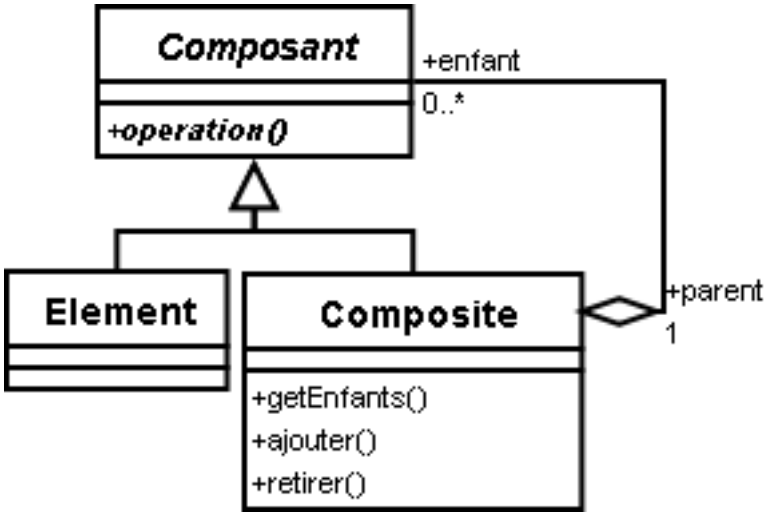
BridgePatternMain.java
```
public class BridgePatternMain {

    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
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

BridgePatternMain.java

```
        // --> 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.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

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

```
/**
 * 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

```
/**
 * 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

```
/**
 * 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);
    }
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Composite.java

```
    /**
     * 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

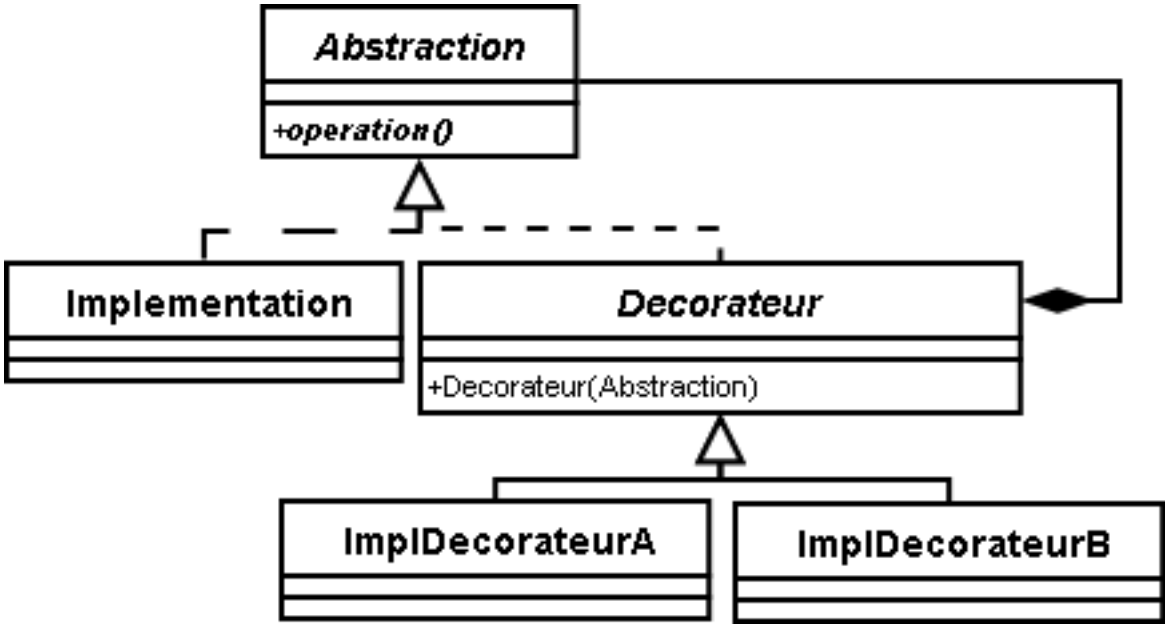
```
public class CompositePatternMain {

    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");
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

CompositePatternMain.java

```
        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 :

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

- Ajouter dynamiquement des **responsabilités** (non obligatoires) à un objet.
- Eviter 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'enchainer 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 contructeur 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 responsabiltés 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
```
/**
 * Définit l'interface générale.
 */
public interface Abstraction {

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

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

    /**
     * Implémentation de la méthode
     * pour l'opération de base
     */
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Implementation.java

```
    public void operation() {
        System.out.println("Implementation");
    }
}
```

Decorateur.java

```
/**
 * 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

```
/**
 * 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

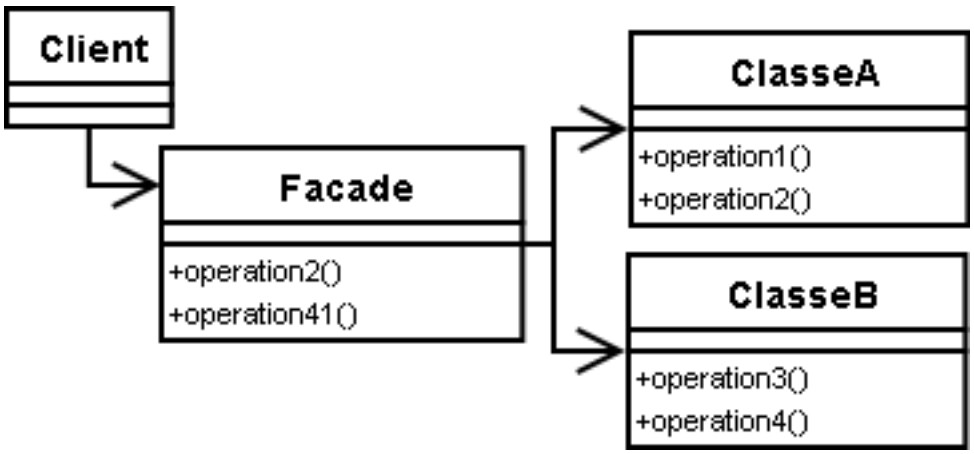
```
/**
 * Implémente un décorateur
 */
public class ImplDecorateurB extends Decorateur {

    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");
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

DecoratorPatternMain.java

```
public class DecoratorPatternMain {

    public static void main(String[] args) {
        // Création de l'implementation 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 :

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

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
```
/**
 * 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
```
/**
 * 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");
    }
}
```

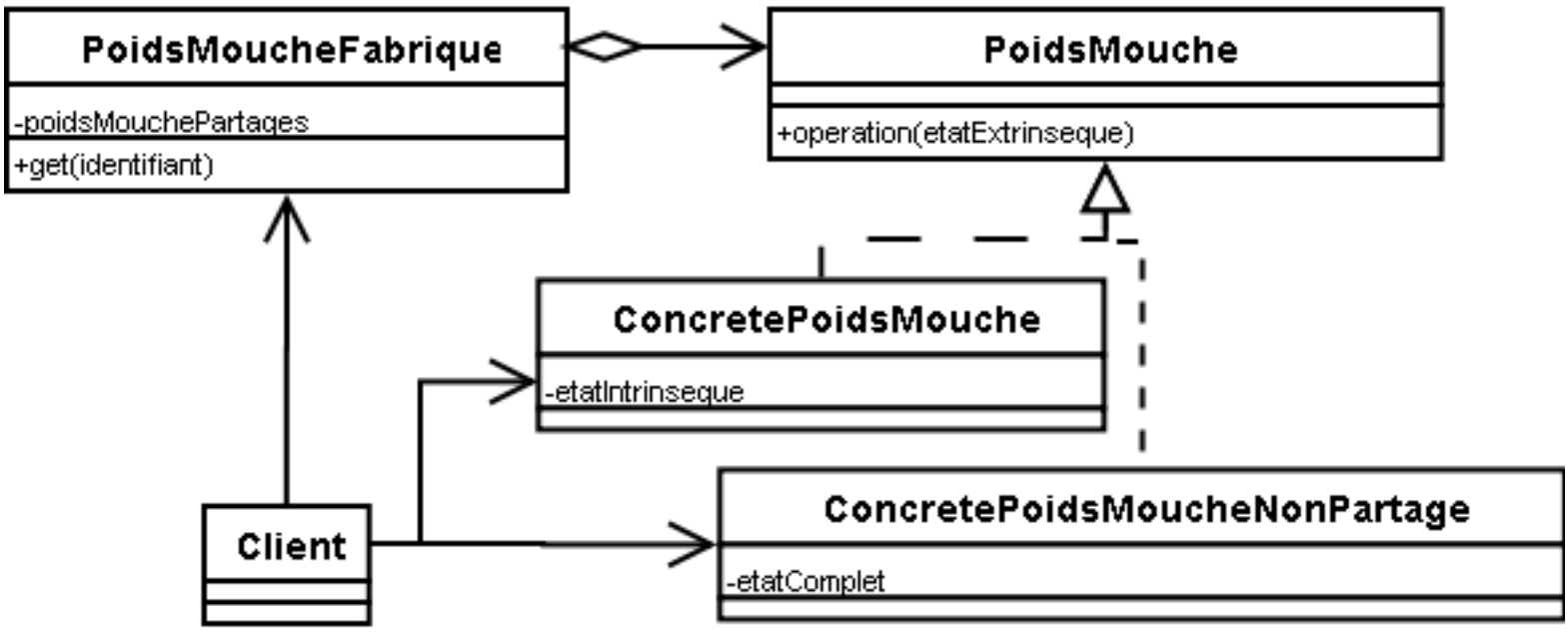
Facade.java
```
/**
 * Présente certaines fonctionnalités.
 * Dans ce cas, ne présente que la méthode "operation2()" de "ClasseA"
 * et la méthode "operation41()" utilisant "operation4()" de "ClasseB"
 * et "operation1()" de "ClasseA".
 */
public class Facade {

    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
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

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

FacadePatternMain.java
```
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).

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

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

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

    public void afficher(String pContexte);
}
```

ConcretePoidsMouche.java

```
/**
 * 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);
    }
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

ConcretePoidsMouche.java

```
}
```

ConcretePoidsMoucheNonPartage.java

```
/**
 * 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

```
/**
 * 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(pValeur);
            poidsMouchesPartages.put(pValeur, lNouveauPoidsMouche);
            return lNouveauPoidsMouche;
        }
    }

    /**
     * Retourne un "PoidsMouche" non partagé.
     * @param pValeur1
     * @param pValeur2
     * @return un "PoidsMouche"
     */
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

PoidsMoucheFabrique.java

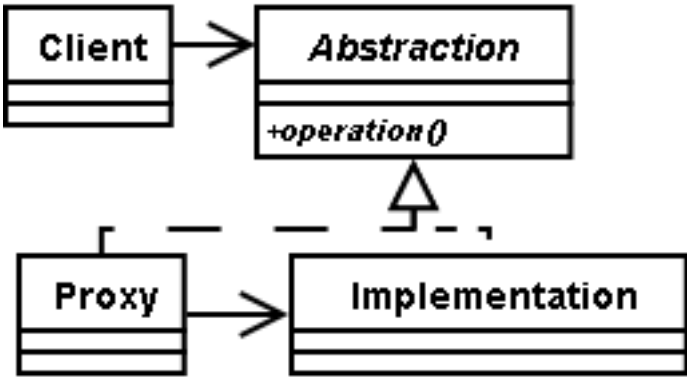
```
    public PoidsMouche getPoidsMouche(String pValeur1, String pValeur2) {
        System.out.println("--> Retourne un PoidsMouche (" + pValeur1 + ", " +
                pValeur2 + ") non partagé");
        return new ConcretePoidsMoucheNonPartage(pValeur1, pValeur2);
    }
}
```

FlyweightPatternMain.java

```
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", "Valeur2");
        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
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

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'ojet **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

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

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


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

Abstraction.java
```
}
```

Implementation.java
```
/**
 * 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'implementation");
    }
}
```

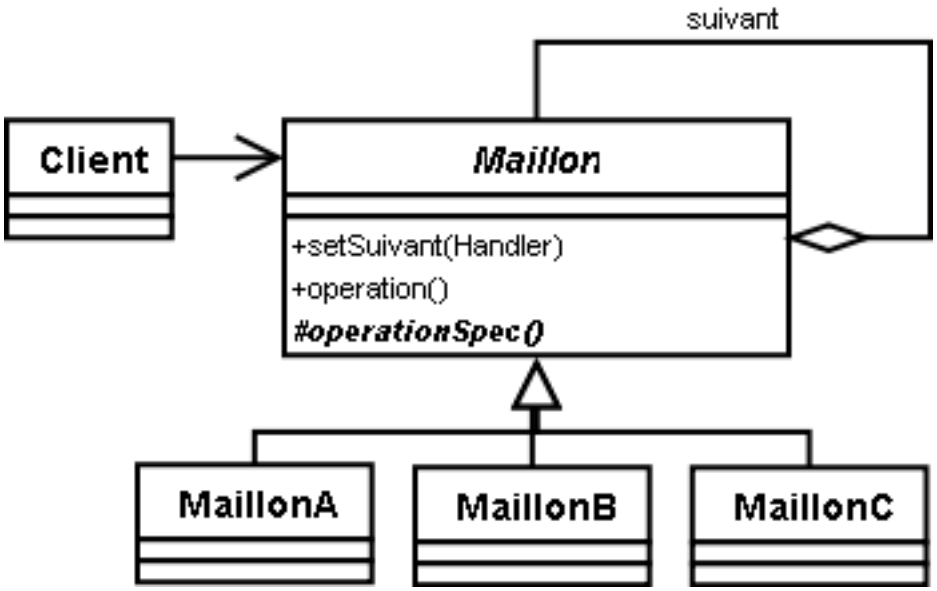
Proxy.java
```
/**
 * Intermédiaire entre la partie cliente et l'implémentation
 */
public class Proxy implements Abstraction {

    /**
     * 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 Implementation");
        lImplementation.afficher();
    }
}
```

ProxyPatternMain.java
```
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
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

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 un requête. La requête implique plusieurs objets pour la traiter.

Cela peut être le cas d'un système complexe d'habilitations possèdant 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 implèmente 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 :

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Maillon.java

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

```
/**
 * 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

```
/**
 * 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) {
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

MaillonB.java

```
            System.out.println("MaillonB : " + pNombre + " : < 2");
            return true;
        }
        return false;
    }
}
```

MaillonC.java

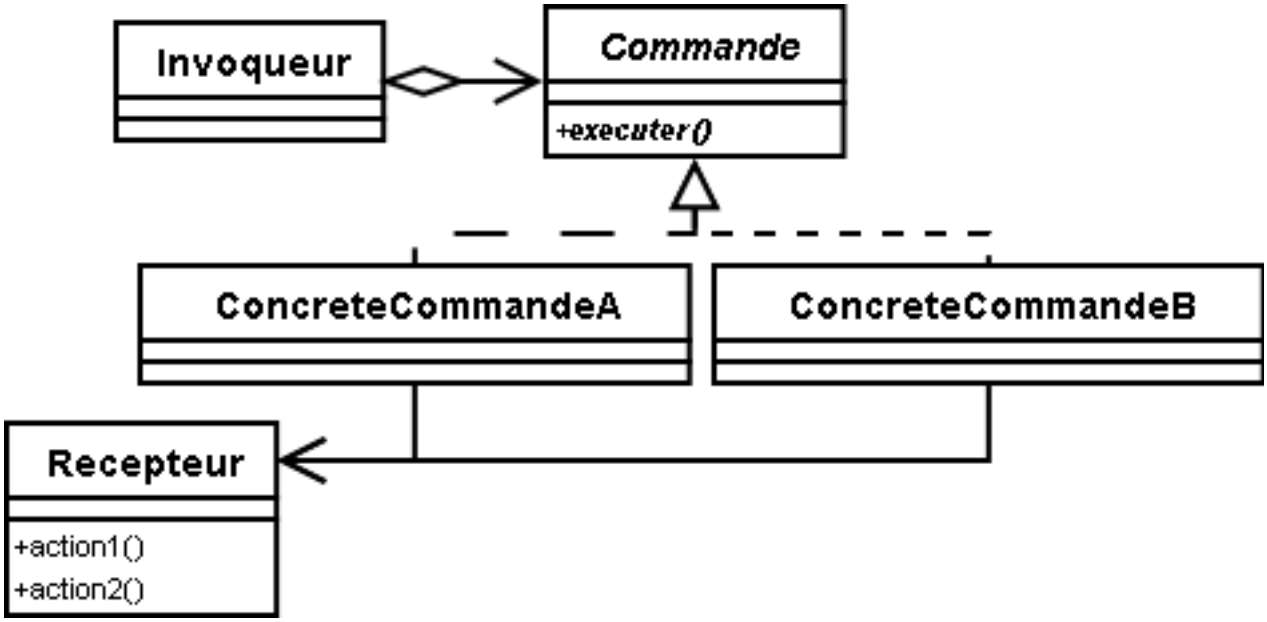
```
/**
 * Sous-classe concrète qui définit un maillon de la chaine.
 */
public class MaillonC extends Maillon {

    /**
     * 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

```
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
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

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.
- **ConcreteCommandeA** et **ConcreteCommandeB** : 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**.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

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

    public void executer();
}
```

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

    private Recepteur recepteur;

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

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

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

    private Recepteur recepteur;

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

    public void executer() {
        recepteur.action2();
    }
}
```

Invoqueur.java
```
/**
 * 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();
        }
    }
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Invoqueur.java

```
    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

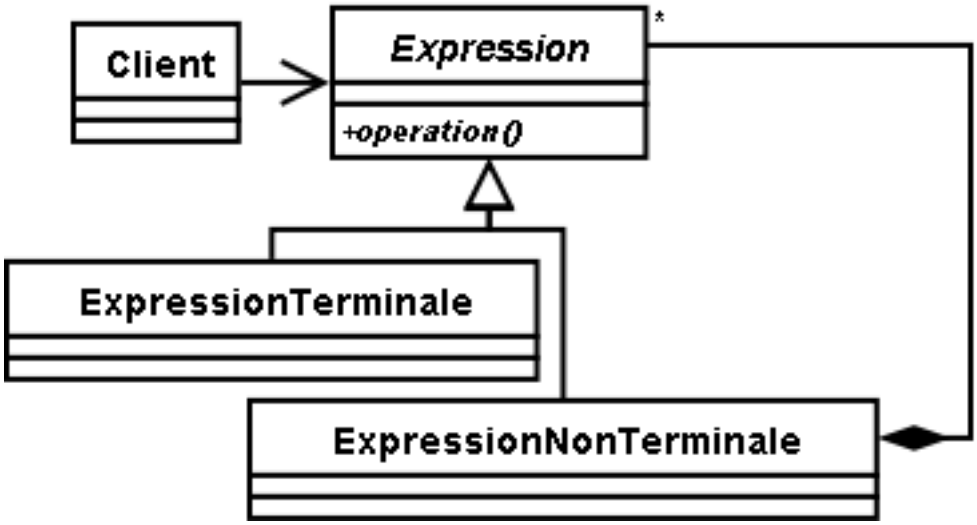
```
/**
 * Reçoit la commande.
 */
public class Recepteur {

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

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

CommandPatternMain.java

```
public class CommandPatternMain {

    public static void main(String[] args) {
        // Création d'un récepteur
        Recepteur lRecepteur = new Recepteur();

        // Création des commandes
        Commande lCommandeA = new ConcreteCommandeA(lRecepteur);
        Commande lCommandeB = new ConcreteCommandeB(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.
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

VI-C - Interpréteur (Interpreter)



OBJECTIFS :

- Définir une réprésentation 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. Un expression non terminale peut contenir d'autres expressions.
- **ExpressionTerminale** : implémente une expression terminale. Un 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

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


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

Expression.java

```
    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

```
/**
 * 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

```
/**
 * 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) {
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

ExpressionTerminale.java

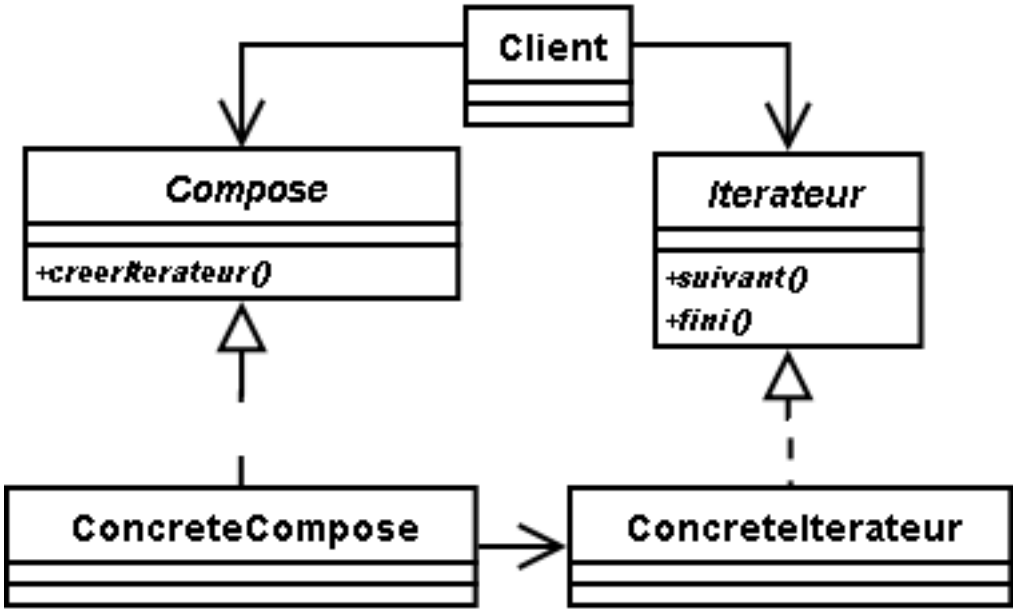
```
        texte = pTexte;
    }

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

InterpreterPatternMain.java

```
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>
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

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 "composes". Elles ont une méthode retournant un itérateur, qui permet de parcourir les éléments. Tous les iterateurs ont la même **interface**. Ainsi, le système dispose d'un moyen homogène de parcourir les composes.

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'inteface **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
```
/**
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

Compose.java

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

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

ConcreteCompose.java

```
/**
 * 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

```
/**
 * 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

```
/**
 * 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++];
    }
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

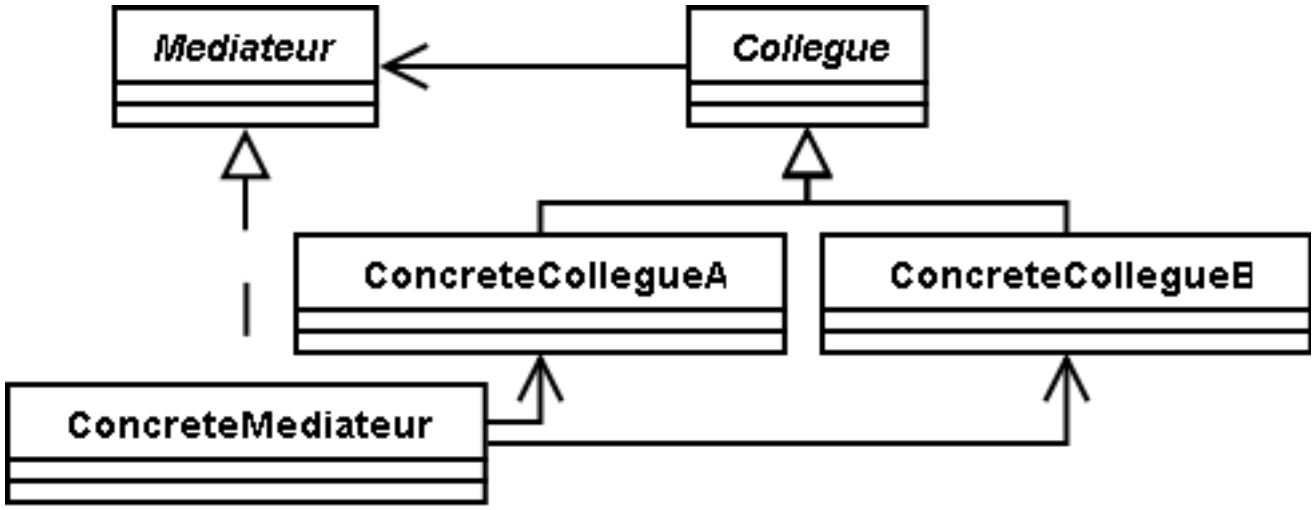
ConcreteIterateur.java

```
    /**
     * 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

```
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();

        // Parcours 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 :

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

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 :

- **Collegue** : 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.
- **ConcreteCollegueA** et **ConcreteCollegueB** : sont des sous-classes concrètes de l'interface **Collegue**. Elles ont une référence sur un objet **Mediateur** auquel elles transmettront les informations.
- **Mediateur** : définit l'interface de communication entre les objets **Collegue**.
- **ConcreteMediateur** : implémente la communication et maintient une référence sur les objets **Collegue**.

IMPLEMENTATION JAVA :

Collegue.java
```
/**
 * Définit l'interface d'un collègue.
 */
public abstract class Collegue {

    protected Mediateur mediateur;

    /**
     * Constructeur permettant de fixer le médiateur
     * @param pMediateur
     */
    public Collegue(Mediateur pMediateur) {
        mediateur = pMediateur;
    }

    public abstract void recevoirMessage(String pMessage);
}
```

ConcreteCollegueA.java
```
/**
 * Sous-classe concrète de "Collegue"
 */
public class ConcreteCollegueA extends Collegue {

    public ConcreteCollegueA(Mediateur pMediateur) {
        super(pMediateur);
        pMediateur.setCollegueA(this);
    }

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


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

ConcreteCollegueA.java

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

ConcreteCollegueB.java

```
/**
 * Sous-classe concrète de "Collegue"
 */
public class ConcreteCollegueB extends Collegue {

    public ConcreteCollegueB(Mediateur pMediateur) {
        super(pMediateur);
        pMediateur.setCollegueB(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("ConcreteCollegueB a reçu : " + pMessage);
    }
}
```

Mediateur.java

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

    // Méthodes permettant les collègues
    public void setCollegueA(ConcreteCollegueA pCollegueA);
    public void setCollegueB(ConcreteCollegueB pCollegueB);

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

ConcreteMediateur.java

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

    private ConcreteCollegueA collegueA;
    private ConcreteCollegueB collegueB;

    // Méthodes permettant les collègues
    public void setCollegueA(ConcreteCollegueA pCollegueA) {
        collegueA = pCollegueA;
    }

    public void setCollegueB(ConcreteCollegueB pCollegueB) {
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

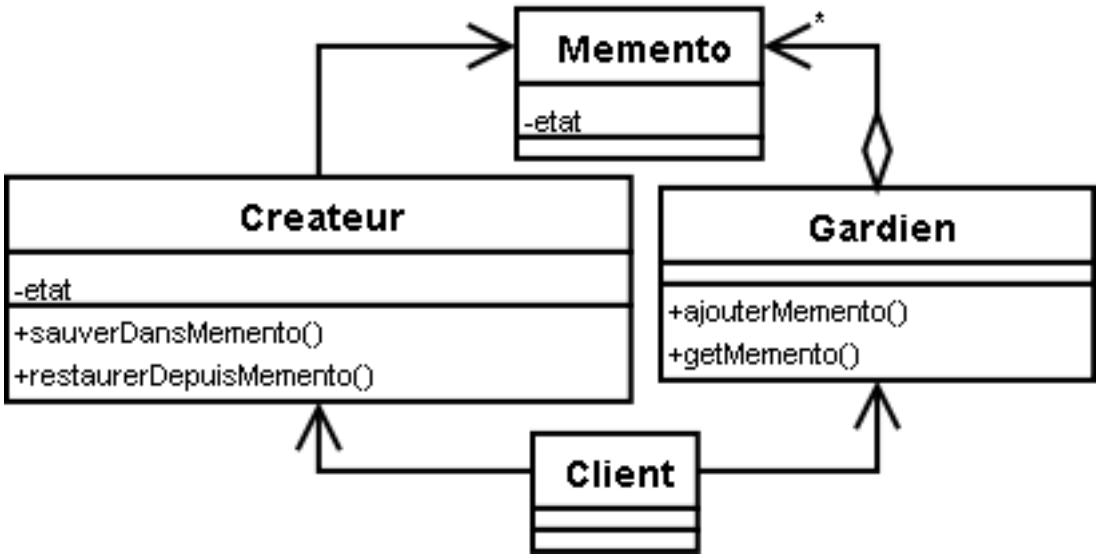
ConcreteMediateur.java

```
        collegueB = pCollegueB;
    }

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

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

MediatorPatternMain.java

```
public class MediatorPatternMain {

    public static void main(String[] args) {
        // Création du médiateur
        Mediateur lMediateur = new ConcreteMediateur();
        // Création des collègues
        ConcreteCollegueA lCollegueA = new ConcreteCollegueA(lMediateur);
        ConcreteCollegueB lCollegueB = new ConcreteCollegueB(lMediateur);

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

        // Affichage :
        // ConcreteCollegueB a reçu : Coucou
        // ConcreteCollegueA 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.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

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 le 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

```
/**
 * 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"
     * accéde 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
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Createur.java

```
     */
    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

```
/**
 * 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

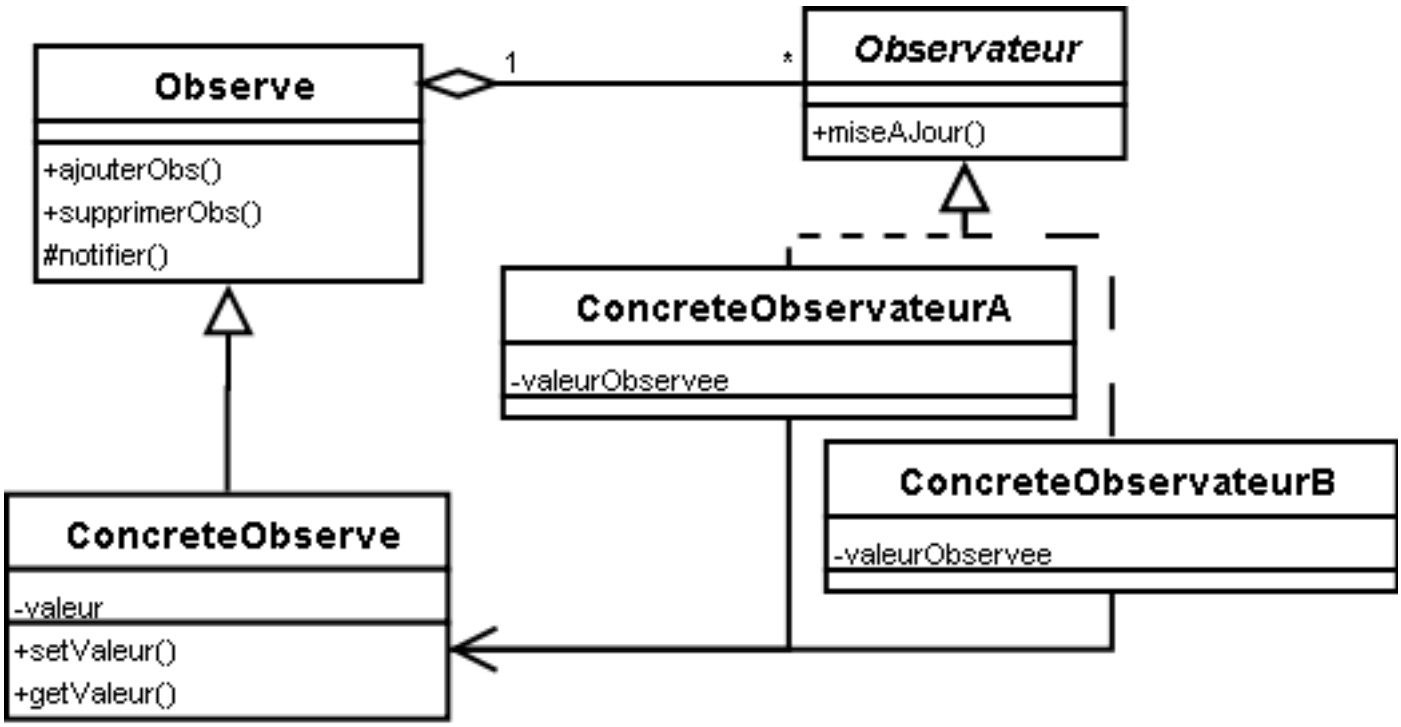
```
public class MementoPatternMain {

    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)
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

MementoPatternMain.java

```
            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 connaitre 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.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

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 un 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

```
/**
 * 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

```
/**
 * Implémentation d'un objet observé
 */
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

ConcreteObserve.java

```java
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

```java
/**
 * Définit l'inteface d'un observateur
 */
public interface Observateur {

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

ConcreteObservateurA.java

```java
/**
 * 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);
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

ConcreteObservateurB.java

```java
/**
 * 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

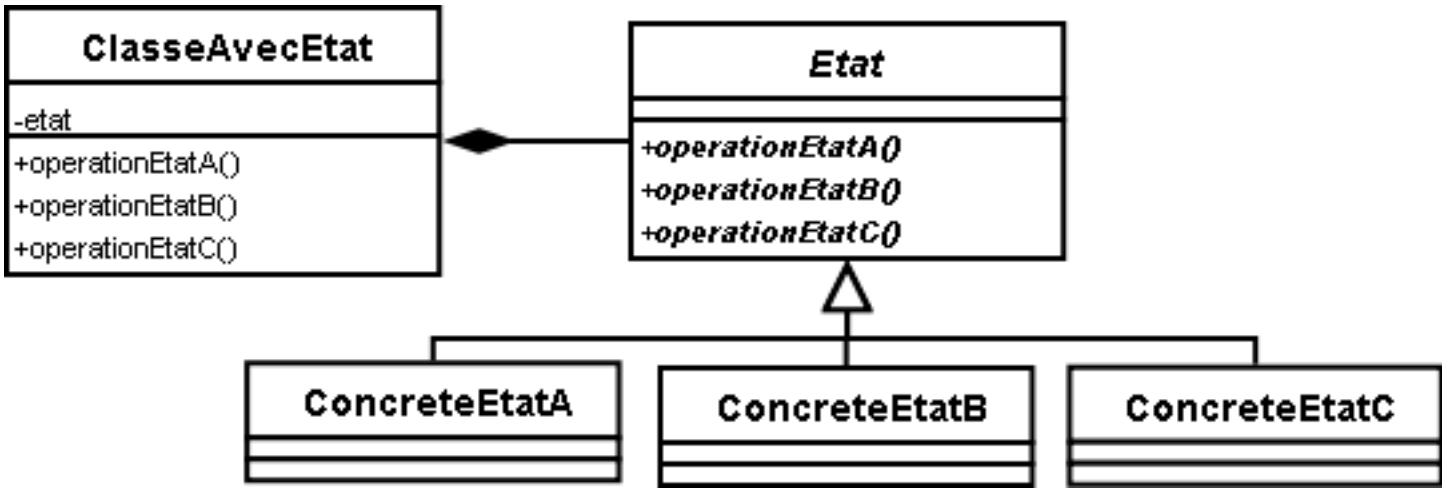
```java
public class ObserverPatternMain {

    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'un 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
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

ObserverPatternMain.java

```
        }
    }
```

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**.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

IMPLEMENTATION JAVA :

ClasseAvecEtat.java

```
/**
 * 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 permattant 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

```
/**
 * 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 {
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

ConcreteEtatA.java

```
    // 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

```
/**
 * 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

```
/**
 * 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");
    }
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

ConcreteEtatC.java

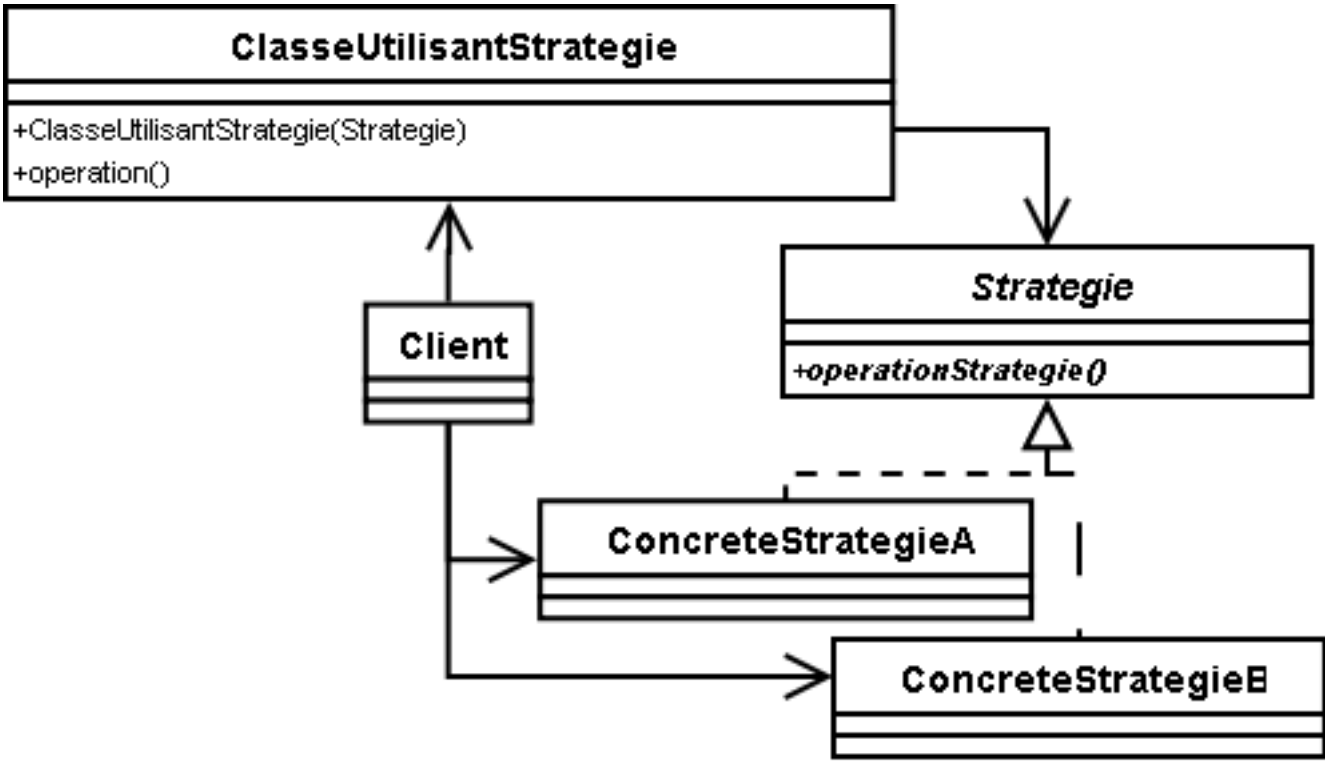
```
    public void operationEtatC(ClasseAvecEtat pClasse) {
        System.out.println("Classe déjà dans l'état C");
    }

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

StatePatternMain.java

```
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
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

VI-I - Stratégie (Strategy ou Policy)



OBJECTIFS :

- Définir une famille d'**algorithmes** interchangeables.
- 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 miniscules, 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 :

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

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

    public void operationStrategie();
}
```

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

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

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

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

ClasseUtilisantStrategie.java
```java
/**
 * 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();
    }
}
```

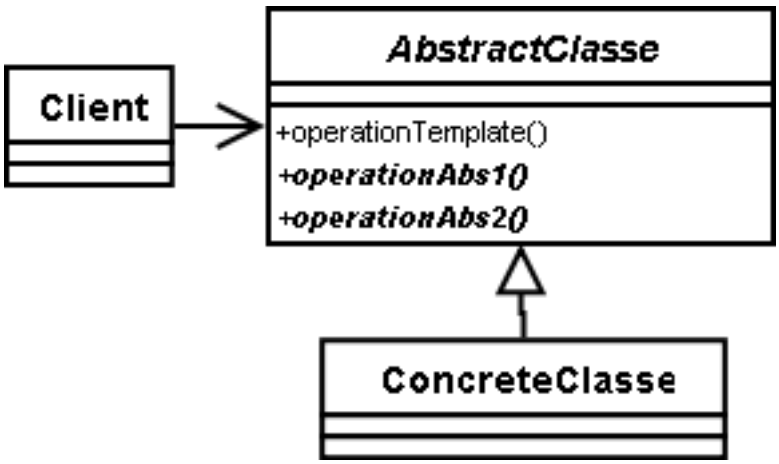
StrategyPatternMain.java
```java
public class StrategyPatternMain {

    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);
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

StrategyPatternMain.java

```
        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 un méthode d'écriture. La partie spécifique de l'algorithme (écriture dans la 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.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

- **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 :

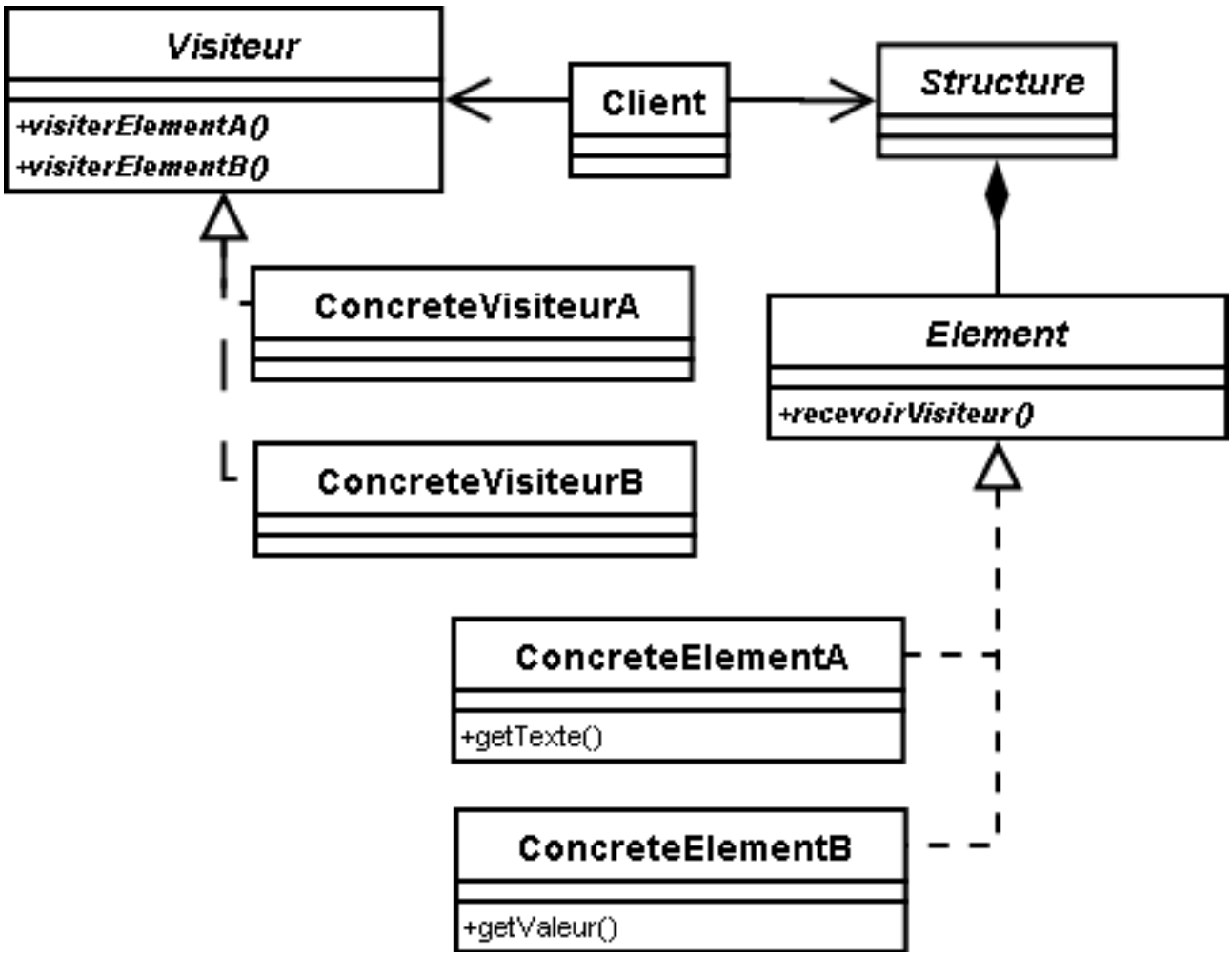
AbstractClasse.java
```
/**
 * Définit l'algorithme
 */
public abstract class AbstractClasse {

    /**
     * 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
```
/**
 * 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
```
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
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

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 un 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.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/

RESPONSABILITES :

- **Element** : définit l'interface d'une 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'une 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
```
/**
 * Définit l'interface d'un élément
 */
public interface Element {

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

ConcreteElementA.java
```
/**
 * 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
```
/**
 * 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
     */
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

ConcreteElementB.java

```
    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

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

    public void visiterElementA(ConcreteElementA pElementA);
    public void visiterElementB(ConcreteElementB pElementB);
}
```

ConcreteVisiteurA.java

```
/**
 * 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

```
/**
 * 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

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


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

Structure.java

```
    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

```
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
    }
}
```


Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

VII - REMERCIEMENTS

Je remercie sincèrement par ordre plus ou moins chronologique :

- **www.developpez.com** me permettant de publier cet article.
- **Nono40** pour ses outils.
- **Baptiste Wicht** pour son accueil lors de ma candidature.
- **Ricky81** pour son accueil et sa disponibilité.
- **djo.mos** pour ses corrections.
- **Matthieu Brucher** pour ses encouragements.
- **SpiceGuid** pour ses suggestions pertinentes et encouragements.
- **hed62** pour ses encouragements.
- **_Mac_** pour sa relecture orthographique rigoureuse.

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.

VIII - ANNEXES

Document résumé sur les Design Patterns (Miroir)

Copyright 2008 - Régis POUILLER. Aucune reproduction, même partielle, ne peut être faite de ce site et 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 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Droits de diffusion permanents accordés à developpez LLC.