

Design Patterns

Réaliser par :

- *ASRIHI Yousra*
- *BEN ABDELLAH Rim*
- *BOUZMARNI Asmae*
- *EL HIRCH Imane*

ADAPTER

Un objet est un client lorsqu'il a besoin d'appeler votre code. Dans certains cas, votre code existe déjà et le développeur peut créer le client de manière à ce qu'il utilise les interfaces de vos objets. Dans d'autres, le client peut être développé indépendamment de votre code. Par exemple, un programme de simulation de fusée pourrait être conçu pour utiliser les informations techniques que vous fournissez, mais une telle simulation aurait sa propre définition du comportement que doit avoir une fusée. Si une classe existante est en mesure d'assurer les services requis par un client mais que ses noms de méthodes diffèrent, vous pouvez appliquer le pattern ADAPTER.

L'objectif du pattern ADAPTER est de fournir l'interface qu'un client attend en utilisant les services d'une classe dont l'interface est différente.

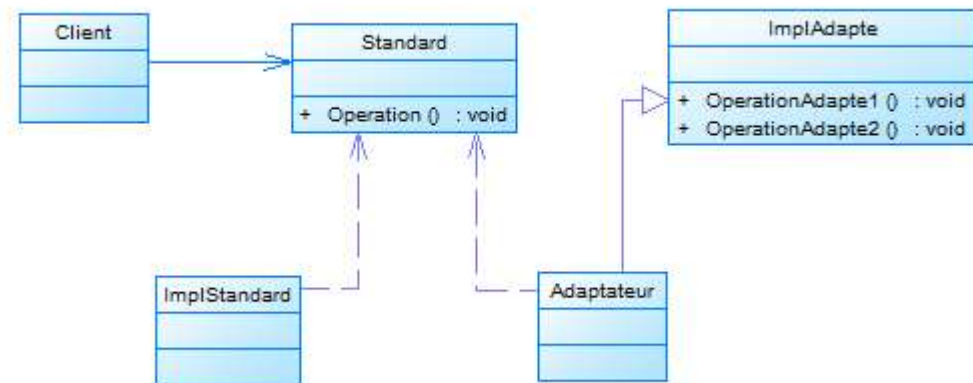
Le système doit intégrer un sous-système existant. Ce sous-système a une interface non standard par rapport au système.

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

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

- **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.

1. Diagramme de classe :



2. Implémentation Java :

```
Standard.java  ImplStandard.java  ImplAdapte.java  Adaptateur.java  AdaptatorPatternMain.java
1 package ADAPTER;
2
3 /**
4  * Définit une interface qui est identifiée
5  * comme standard dans la partie cliente.
6  */
7 public interface Standard {
8
9     /**
10     * L'opération doit multiplier les deux nombres,
11     * puis afficher le résultat de l'opération
12     */
13     public void operation(int pNombre1, int pNombre2);
14 }
```

```
Standard.java  ImplStandard.java  ImplAdapte.java  Adaptateur.java  AdaptatorPatternMain.java
1 package ADAPTER;
2
3 /**
4  * Implémente l'interface "Standard".
5  */
6 public class ImplStandard implements Standard {
7
8     public void operation(int pNombre1, int pNombre2) {
9         System.out.println("Standard : Le nombre est : " + (pNombre1 * pNombre2));
10     }
11 }
12
```

```

Standard.java ImplStandard.java ImplAdapte.java Adaptateur.java AdaptatorPatternMain.java
1 package ADAPTER;
2
3 /**
4  * Fournit les fonctionnalités définies dans l'interface "Standard",
5  * mais ne respecte pas l'interface.
6  */
7 public class ImplAdapte {
8
9     public int operationAdapte1(int pNombre1, int pNombre2) {
10         return pNombre1 * pNombre2;
11     }
12
13     /**
14     * Apporte la fonctionnalité définie dans l'interface,
15     * mais la méthode n'a pas le bon nom
16     * et n'accepte pas le même paramètre.
17     */
18     public void operationAdapte2(int pNombre) {
19         System.out.println("Adapte : Le nombre est : " + pNombre);
20     }
21 }
22

```

```

Standard.java ImplStandard.java ImplAdapte.java Adaptateur.java AdaptatorPatternMain.java
1 package ADAPTER;
2
3 /**
4  * Adapte l'implémentation non standard avec l'héritage.
5  */
6 public class Adaptateur extends ImplAdapte implements Standard {
7
8     /**
9     * Appelle les méthodes non standard
10     * depuis une méthode respectant l'interface.
11     * 1° Appel de la méthode réalisant la multiplication
12     * 2° Appel de la méthode d'affichage du résultat
13     * La classe adaptée est héritée, donc on appelle directement les méthodes
14     */
15     public void operation(int pNombre1, int pNombre2) {
16         int lNombre = operationAdapte1(pNombre1, pNombre2);
17         operationAdapte2(lNombre);
18     }
19 }
20

```

```

Standard.java ImplStandard.java ImplAdapte.java Adaptateur.java AdaptatorPatternMain.java
1 package ADAPTER;
2
3 public class AdaptatorPatternMain {
4     public static void main(String[] args) {
5         // Création d'un adaptateur
6         final Standard lImplAdapte = new Adaptateur();
7         // Création d'une implémentation standard
8         final Standard lImplStandard = new ImplStandard();
9
10        // Appel de la même méthode sur chaque instance
11        lImplAdapte.operation(2, 4);
12        lImplStandard.operation(2, 4);
13
14        // Affichage :
15        // Adapte : Le nombre est : 8
16        // Standard : Le nombre est : 8
17    }
18
19
20 }
21

```

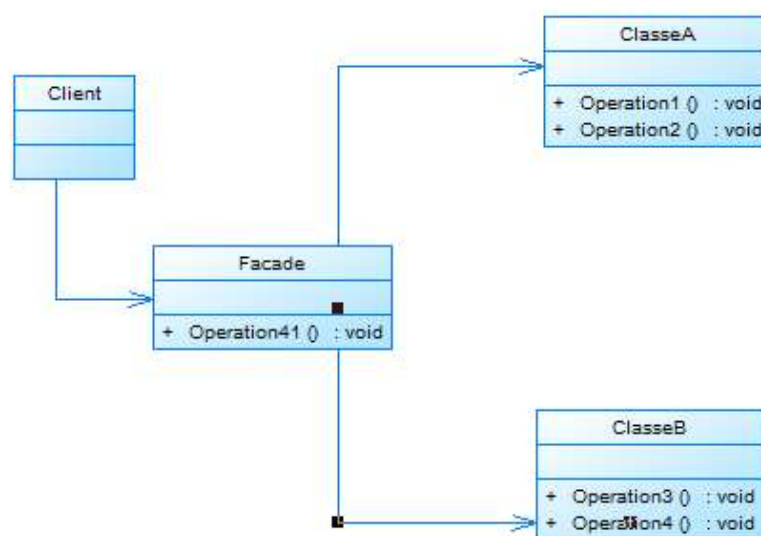
FACADE

Une approche pour simplifier l'emploi d'un kit d'outils est de fournir une façade — une petite quantité de code qui permet un usage typique à peu de frais des classes de la bibliothèque. Une façade est elle-même une classe avec un niveau de fonctionnalités situé entre le kit d'outils et une application complète, proposant un emploi simplifié des classes d'un package ou d'un sous-système.

L'objectif du pattern FACADE est de fournir une interface simplifiant l'emploi d'un sous-système.

- **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**.

1. Diagramme de classe :



2. Implémentation Java :

```
ClasseA.java ClasseB.java Facade.java FacadePatternMain.java
1 package FACADE;
2
3 /**
4  * Classe implémentant diverses fonctionnalités.
5  */
6 public class ClasseA {
7
8     public void operation1() {
9         System.out.println("Methode operation1() de la classe ClasseA");
10    }
11
12    public void operation2() {
13        System.out.println("Methode operation2() de la classe ClasseA");
14    }
15 }
16
```

```
ClasseA.java ClasseB.java Facade.java FacadePatternMain.java
1 package FACADE;
2
3 /**
4  * Classe implémentant d'autres fonctionnalités.
5  */
6 public class ClasseB {
7
8     public void operation3() {
9         System.out.println("Methode operation3() de la classe ClasseB");
10    }
11
12    public void operation4() {
13        System.out.println("Methode operation4() de la classe ClasseB");
14    }
15 }
16
```

```
ClasseA.java ClasseB.java Facade.java FacadePatternMain.java
1 package FACADE;
2
3 /**
4  * Présente certaines fonctionnalités.
5  * Dans ce cas, ne présente que la méthode "operation2()" de "ClasseA"
6  * et la méthode "operation41()" utilisant "operation4()" de "ClasseB"
7  * et "operation1()" de "ClasseA".
8  */
9 public class Facade {
10
11     private ClasseA classeA = new ClasseA();
12     private ClasseB classeB = new ClasseB();
13
14     /**
15      * La méthode operation2() appelle simplement
16      * la même méthode de ClasseA
17      */
18     public void operation2() {
19         System.out.println("--> Méthode operation2() de la classe Facade : ");
20         classeA.operation2();
21     }
22
23     /**
24      * La méthode operation41() appelle
25      * operation4() de ClasseB
26      * et operation1() de ClasseA
27      */
28     public void operation41() {
29         System.out.println("--> Méthode operation41() de la classe Facade : ");
30         classeB.operation4();
31         classeA.operation1();
32     }
33 }
34
```

```
ClasseA.java ClasseB.java Facade.java FacadePatternMain.java
1 package FACADE;
2
3 public class FacadePatternMain {
4     public static void main(String[] args) {
5         // Création de l'objet "Facade" puis appel des méthodes
6         Facade lFacade = new Facade();
7         lFacade.operation2();
8         lFacade.operation41();
9
10        // Affichage :
11        // --> Méthode operation2() de la classe Facade :
12        // Methode operation2() de la classe ClasseA
13        // --> Méthode operation41() de la classe Facade :
14        // Methode operation4() de la classe ClasseB
15        // Methode operation1() de la classe ClasseA
16    }
17 }
18 }
19
```

```
Problems Javadoc Declaration Console
<terminated> FacadePatternMain [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (3 janv. 2018 23:33:35)
[-> 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
```

Remarque :

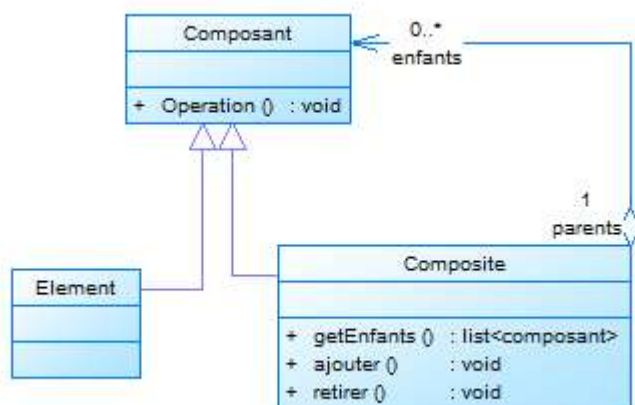
La façade peut également implémenter le Design Pattern [Singleton](#) qui sera traité ultérieurement.

COMPOSITE

Un COMPOSITE est un groupe d'objets contenant aussi bien des éléments individuels que des éléments contenant d'autres objets. Certains objets contenus représentent donc eux-mêmes des groupes et d'autres sont des objets individuels appelés des feuilles (leaf). Lorsque vous modélisez un objet composite, deux concepts efficaces émergent. Une première idée importante est de concevoir des groupes de manière à englober des éléments individuels ou d'autres groupes — une erreur fréquente est de définir des groupes ne contenant que des feuilles. Un autre concept puissant est la définition de comportements communs aux deux types d'objets, individuels et composites. Vous pouvez unir ces deux idées en définissant un type commun aux groupes et aux feuilles, et en modélisant des groupes de façon qu'ils contiennent un ensemble d'objets de ce type.

L'objectif du pattern COMPOSITE est de permettre aux clients de traiter de façon uniforme des objets individuels et des compositions d'objets.

1. Diagramme de classe :



2. Implémentation Java :

```
Composant.java  Element.java  Composite.java  CompositePatternMain.java
1 package COMPOSITE;
2
3 /**
4  * Définit l'interface d'un objet pouvant être un composant
5  * d'un autre objet de l'arborescence.
6  */
7 public abstract class Composant {
8
9     // Nom de "Composant"
10    protected String nom;
11
12    /**
13     * Constructeur
14     * @param pNom Nom du "Composant"
15     */
16    public Composant(final String pNom) {
17        nom = pNom;
18    }
19
20    /**
21     * Opération commune à tous les "Composant"
22     */
23    public abstract void operation();
24 }
25
```

```
Composant.java  Element.java  Composite.java  CompositePatternMain.java
1 package COMPOSITE;
2
3 /**
4  * Implémente un objet de l'arborescence
5  * n'ayant pas d'objet le composant.
6  */
7 public class Element extends Composant {
8
9     public Element(final String pNom) {
10         super(pNom);
11     }
12
13     /**
14     * Méthode commune à tous les composants :
15     * Affiche qu'il s'agit d'un objet "Element"
16     * ainsi que le nom qu'on lui a donné.
17     */
18     public void operation() {
19         System.out.println("Operation sur un 'Element' (" + nom + ")");
20     }
21 }
22
```

```
Composant.java  Element.java  Composite.java  CompositePatternMain.java

11 public class Composite extends Composant {
12
13     // Liste d'objets "Composant" de l'objet "Composite"
14     private List<Composant> liste = new LinkedList<Composant>();
15
16     public Composite(final String pNom) {
17         super(pNom);
18     }
19
20
21     public void operation() {
22         System.out.println("Operation sur un 'Composite' (" + nom + ")");
23         final Iterator<Composant> lIterator = liste.iterator();
24         while(lIterator.hasNext()) {
25             final Composant lComposant = lIterator.next();
26             lComposant.operation();
27         }
28     }
29
30
31     public List<Composant> getEnfants() {
32         return liste;
33     }
34
35     /**
36     * Ajoute un objet "Composant" au "Composite"
37     * @param pComposant
38     */
39     public void ajouter(final Composant pComposant) {
40         liste.add(pComposant);
41     }
42
43     ...
```

```
Composant.java  Element.java  Composite.java  CompositePatternMain.java
14      //          - lComposite4
15      //          - lComposite5
16      //          - lElement5
17      //          - lElement2
18
19      // Création des objets "Composite"
20      final Composite lComposite1 = new Composite("Composite 1");
21      final Composite lComposite2 = new Composite("Composite 2");
22      final Composite lComposite3 = new Composite("Composite 3");
23      final Composite lComposite4 = new Composite("Composite 4");
24      final Composite lComposite5 = new Composite("Composite 5");
25
26      // Création des objets "Element"
27      final Element lElement1 = new Element("Element 1");
28      final Element lElement2 = new Element("Element 2");
29      final Element lElement3 = new Element("Element 3");
30      final Element lElement4 = new Element("Element 4");
31      final Element lElement5 = new Element("Element 5");
32
33      // Ajout des "Composant" afin de constituer l'arborescence
34      lComposite1.ajouter(lElement1);
35      lComposite1.ajouter(lComposite2);
36      lComposite1.ajouter(lElement2);
37
38      lComposite2.ajouter(lComposite3);
39      lComposite2.ajouter(lComposite4);
40
41      lComposite3.ajouter(lElement3);
42      lComposite3.ajouter(lElement4);
43
44      lComposite4.ajouter(lComposite5);
45
46      lComposite5.ajouter(lElement5);
47
```

```
Problems  @ Javadoc  Declaration  Console
<terminated> CompositePatternMain [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (4 janv. 2018 14:54:47)
Operation sur un 'Composite' (Composite 1)
Operation sur un 'Element' (Element 1)
Operation sur un 'Composite' (Composite 2)
Operation sur un 'Composite' (Composite 3)
Operation sur un 'Element' (Element 3)
Operation sur un 'Element' (Element 4)
Operation sur un 'Composite' (Composite 4)
Operation sur un 'Composite' (Composite 5)
Operation sur un 'Element' (Element 5)
Operation sur un 'Element' (Element 2)
```

BRIDGE

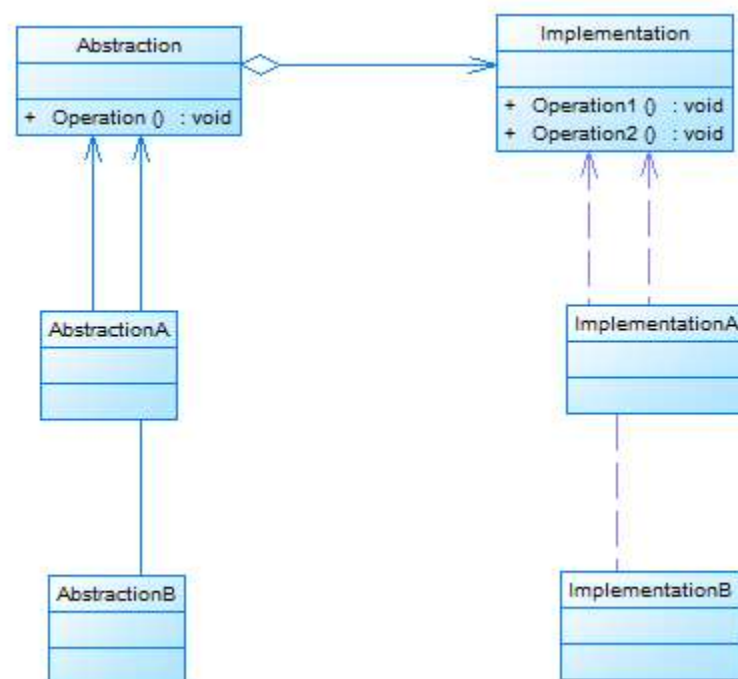
Le pattern BRIDGE, ou Driver, vise à implémenter une abstraction. Le terme abstraction se réfère à une classe qui s'appuie sur un ensemble d'opérations abstraites, lesquelles peuvent avoir plusieurs implémentations. La façon habituelle d'implémenter une abstraction est de créer une hiérarchie de classes, avec une classe abstraite au sommet qui définit les opérations abstraites. Chaque sous-classe de la hiérarchie apporte une implémentation différente de l'ensemble d'opérations. Cette approche devient insuffisante lorsqu'il vous faut dériver une sous-classe de la

hiérarchie pour une quelconque autre raison. Vous pouvez créer un BRIDGE (pont) en déplaçant l'ensemble d'opérations abstraites vers une interface de sorte qu'une abstraction dépendra d'une implémentation de l'interface.

L'objectif du pattern BRIDGE est de découpler une abstraction de l'implémentation de ses opérations abstraites, permettant ainsi à l'abstraction et à son implémentation de varier indépendamment.

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

1. Diagramme de classe :



2. Implémentation Java :

```
1 Implementation... 2 Implementation... 3 Implementation... 4 Abstraction.j... 5 AbstractionA.j... 6 AbstractionB.j... 7 BridgePattern...
DesignPatterns/src/BRIDGE/Implementation.java
2
3 /**
4  * Définit l'interface de l'implémentation.
5  * L'implémentation fournit deux méthodes
6  */
7 public interface Implementation {
8
9     public void operationImpl1(String pMessage);
10    public void operationImpl2(Integer pNombre);
11 }
12
```

```
1 Implementation... 2 Implementation... 3 Implementation... 4 Abstraction.j... 5 AbstractionA.j... 6 AbstractionB.j... 7 BridgePattern...
1 package BRIDGE;
2
3 /**
4  * Sous-classe concrète de l'implémentation
5  */
6 public class ImplementationA implements Implementation {
7
8     public void operationImpl1(String pMessage) {
9         System.out.println("operationImpl1 de ImplementationA : " + pMessage);
10    }
11
12    public void operationImpl2(Integer pNombre) {
13        System.out.println("operationImpl2 de ImplementationA : " + pNombre);
14    }
15 }
16
```

```
1 Implementation... 2 Implementation... 3 Implementation... 4 Abstraction.j... 5 AbstractionA.j... 6 AbstractionB.j... 7 BridgePattern...
1 package BRIDGE;
2
3 /**
4  * Sous-classe concrète de l'implémentation
5  */
6 public class ImplementationB implements Implementation {
7
8     public void operationImpl1(String pMessage) {
9         System.out.println("operationImpl1 de ImplementationB : " + pMessage);
10    }
11
12    public void operationImpl2(Integer pNombre) {
13        System.out.println("operationImpl2 de ImplementationB : " + pNombre);
14    }
15 }
16
```

```
1 Implementation... 2 Implementation... 3 Implementation... 4 Abstraction.j... 5 AbstractionA.j... 6 AbstractionB.j... 7 BridgePattern...
1 package BRIDGE;
2
3 /**
4  * Définit l'interface de l'abstraction
5  */
6 public abstract class Abstraction {
7
8     // Référence vers l'implémentation
9     private Implementation implementation;
10
11     protected Abstraction(Implementation pImplementation) {
12         implementation = pImplementation;
13     }
14
15     public abstract void operation();
16
17     /**
18      * Lien vers la méthode operationImpl1() de l'implémentation
19      * @param pMessage
20      */
21     protected void operationImpl1(String pMessage) {
22         implementation.operationImpl1(pMessage);
23     }
24
25     /**
26      * Lien vers la méthode operationImpl2() de l'implémentation
27      * @param pMessage
28      */
29     protected void operationImpl2(Integer pNombre) {
30         implementation.operationImpl2(pNombre);
31     }
32 }
33
```

```

1 package BRIDGE;
2
3 /**
4  * Sous-classe concrète de l'abstraction
5  */
6 public class AbstractionA extends Abstraction {
7
8     public AbstractionA(Implementation pImplementation) {
9         super(pImplementation);
10    }
11
12    public void operation() {
13        System.out.println("--> Méthode operation() de AbstractionA");
14        operationImpl1("A");
15        operationImpl2(1);
16        operationImpl1("B");
17    }
18 }
19

```

```

1 package BRIDGE;
2
3 /**
4  * Sous-classe concrète de l'abstraction
5  */
6 public class AbstractionB extends Abstraction {
7
8     public AbstractionB(Implementation pImplementation) {
9         super(pImplementation);
10    }
11
12    public void operation() {
13        System.out.println("--> Méthode operation() de AbstractionB");
14        operationImpl2(9);
15        operationImpl2(8);
16        operationImpl1("Z");
17    }
18 }
19

```

```

1 package BRIDGE;
2
3 public class BridgePatternMain {
4
5     public static void main(String[] args) {
6         // Création des implémentations
7         Implementation lImplementationA = new ImplementationA();
8         Implementation lImplementationB = new ImplementationB();
9
10        // Création des abstractions
11        Abstraction lAbstractionAA = new AbstractionA(lImplementationA);
12        Abstraction lAbstractionAB = new AbstractionA(lImplementationB);
13        Abstraction lAbstractionBA = new AbstractionB(lImplementationA);
14        Abstraction lAbstractionBB = new AbstractionB(lImplementationB);
15
16        // Appels des méthodes des abstractions
17        lAbstractionAA.operation();
18        lAbstractionAB.operation();
19        lAbstractionBA.operation();
20        lAbstractionBB.operation();
21    }
22 }

```

Problems | Javadoc | Declaration | Console

```

<terminated> BridgePatternMain [Java Application] C:\Program Files\Java\jdk-8.0_20\bin\java.exe (6 janv. 2018 23:38:31)
--> Méthode operation() de AbstractionA
operationImpl1 de ImplementationA : A
operationImpl2 de ImplementationA : 1
operationImpl1 de ImplementationA : B
--> Méthode operation() de AbstractionA
operationImpl1 de ImplementationB : A
operationImpl2 de ImplementationB : 1
operationImpl1 de ImplementationB : B
--> Méthode operation() de AbstractionB
operationImpl2 de ImplementationA : 9
operationImpl2 de ImplementationA : 8
operationImpl1 de ImplementationA : Z
--> Méthode operation() de AbstractionB
operationImpl2 de ImplementationB : 9
operationImpl2 de ImplementationB : 8
operationImpl1 de ImplementationB : Z

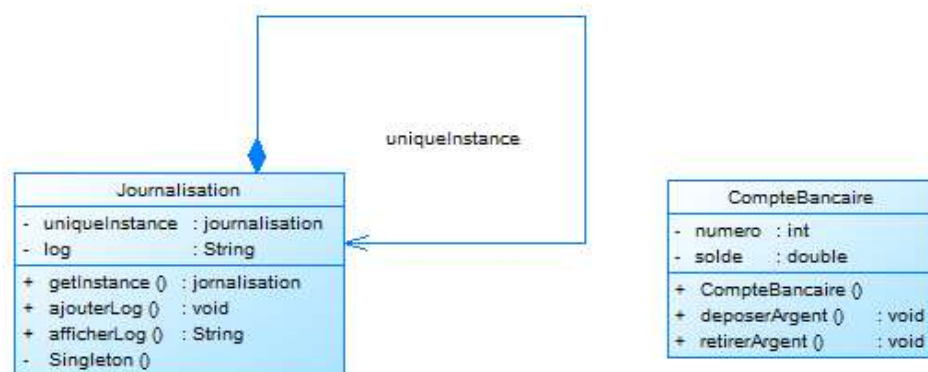
```

SINGLETON

Les objets peuvent généralement agir de façon responsable en effectuant leur travail sur leurs propres attributs, sans avoir d'autre obligation que d'assurer leur cohérence propre. Cependant, certains objets assument d'autres responsabilités, telles que la modélisation d'entités du monde réel, la coordination de tâches, ou la modélisation de l'état général d'un système. Lorsque, dans un système, un certain objet assume une responsabilité dont dépendent d'autres objets, vous devez disposer d'une méthode pour localiser cet objet. Par exemple, vous pouvez avoir besoin d'identifier un objet qui représente une machine particulière, un objet client qui puisse se construire lui-même à partir d'informations extraites d'une base de données, ou encore un objet qui initie une récupération de la mémoire système. Dans certains cas, lorsque vous devez trouver un objet responsable, l'objet dont vous avez besoin sera la seule instance de sa classe. Par exemple, la création de fusées peut se suffire d'un seul objet Factory. Dans ce cas, vous pouvez utiliser le pattern SINGLETON.

L'objectif du pattern SINGLETON est de garantir qu'une classe ne possède qu'une seule instance et de fournir un point d'accès global à celle-ci.

1. Diagramme de classe :



2. Implémentation Java :

```
Journalisation.java  CompteBancaire.java  SingletonPatternMain.java
public class Journalisation
{
    private static Journalisation uniqueInstance; // Stockage de l'unique instance de cette classe.
    private String log; // Chaîne de caractères représentant les messages de log.

    // Constructeur en privé (donc inaccessible à l'extérieur de la classe).
    private Journalisation()
    {
        log = new String();
    }

    // Méthode statique qui sert de pseudo-constructeur (utilisation du mot clef "synchronized" pour le multithread).
    public static synchronized Journalisation getInstance()
    {
        if(uniqueInstance==null)
        {
            uniqueInstance = new Journalisation();
        }
        return uniqueInstance;
    }

    // Méthode qui permet d'ajouter un message de log.
    public void ajouterLog(String log)
    {
        // On ajoute également la date du message.
        Date d = new Date();
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yy HH'h'mm");
        this.log+="["+dateFormat.format(d)+"] "+log+"\n";
    }

    // Méthode qui retourne tous les messages de log.
    public String afficherLog()
    {
        return log;
    }
}
```

```
Journalisation.java  CompteBancaire.java  SingletonPatternMain.java
package SINGLETON;
//Classe représentant un compte bancaire simpliste.
public class CompteBancaire
{
    private int numero; // Numéro du compte.
    private double solde; // Argent disponible sur le compte.

    // Constructeur d'un CompteBancaire à partir de son numéro.
    public CompteBancaire(int numero)
    {
        this.numero=numero;
        this.solde=0.0;
    }

    // Méthode qui permet de déposer de l'argent sur le compte.
    public void depoterArgent(double depot)
    {
        if(depot>0.0)
        {
            solde+=depot; // On ajoute la somme déposée au solde.
            Journalisation.getInstance().ajouterLog("Dépôt de "+depot+"€ sur le compte "+numero+".");
        }
        else
        {
            Journalisation.getInstance().ajouterLog("/!\\ Dépôt d'une valeur négative impossible (" +numero+").");
        }
    }
}
```

```
Journalisation.java  CompteBancaire.java  SingletonPatternMain.java
    else
    {
        Journalisation.getInstance().ajouterLog("/!\\ Dépôt d'une valeur négative impossible (" +numero+").");
    }
}

// Méthode qui permet de retirer de l'argent sur le compte.
public void retirerArgent(double retrait)
{
    if(retrait>0.0)
    {
        if(solde>retrait)
        {
            solde-=retrait; // On retire la somme retirée au solde.
            Journalisation.getInstance().ajouterLog("Retrait de "+retrait+"€ sur le compte "+numero+".");
        }
        else
        {
            Journalisation.getInstance().ajouterLog("/!\\ La banque n'autorise pas de découvert (" +numero+").");
        }
    }
    else
    {
        Journalisation.getInstance().ajouterLog("/!\\ Retrait d'une valeur négative impossible (" +numero+").");
    }
}
}
```




```
1 package SINGLETON;
2
3
4
5 //Classe principale de l'application.
6 public class SingletonPatternMain
7 {
8     // Méthode principale.
9     public static void main(String[] args)
10    {
11        // Création et utilisation du CompteBancaire cb1.
12        CompteBancaire cb1 = new CompteBancaire(123456789);
13        cb1.deposerArgent(100);
14        cb1.retirerArgent(80);
15        // Création et utilisation du CompteBancaire cb2.
16        CompteBancaire cb2 = new CompteBancaire(987654321);
17        cb2.retirerArgent(10);
18        // Affichage des logs en console.
19        String s = Journalisation.getInstance().afficherLog();
20        System.out.println(s);
21    }
22 }
23
24
```

```
<terminated> SingletonPatternMain [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\java.exe (7 janv. 2018 20:30:28)
[07/01/18 20h30] Dépôt de 100.0€ sur le compte 123456789.
[07/01/18 20h30] Retrait de 80.0€ sur le compte 123456789.
[07/01/18 20h30] //\ La banque n'autorise pas de découvert (987654321).
```

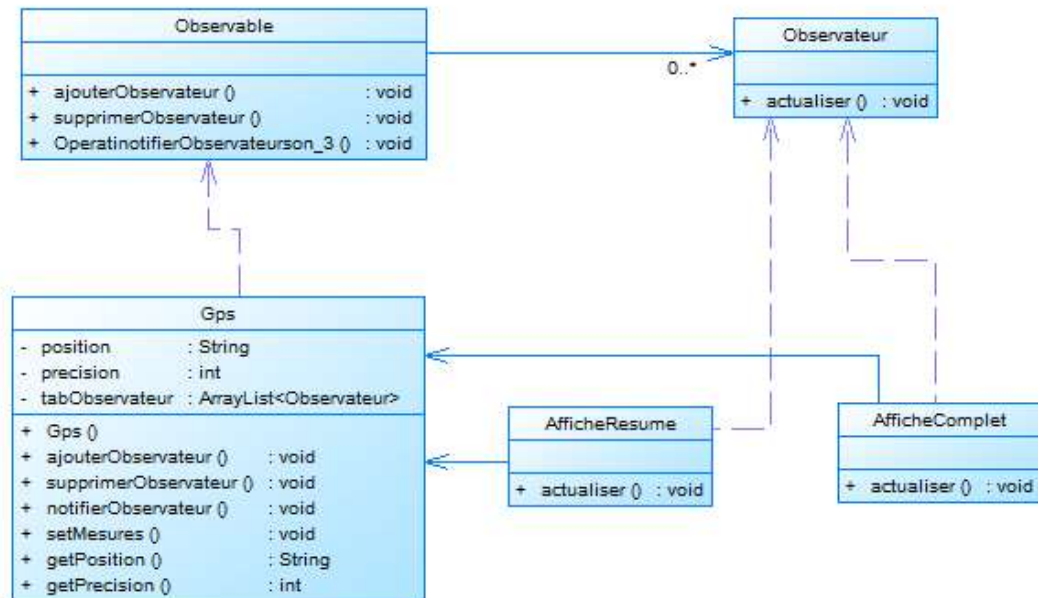
OBSERVER

D'ordinaire, les clients collectent des informations provenant d'un objet en appelant ses méthodes. Toutefois, lorsque l'objet change, un problème survient : comment les clients qui dépendent des informations de l'objet peuvent-ils déterminer que celles-ci ont changé ? Certaines conceptions imputent à l'objet la responsabilité d'informer les clients lorsqu'un aspect intéressant de l'objet change. Cette démarche pose un problème, car c'est le client qui sait quels sont les attributs qui l'intéressent. L'objet intéressant ne doit pas accepter la responsabilité d'actualiser le client. Une solution possible est de s'arranger pour que le client soit informé lorsque l'objet change et de laisser au client la liberté de s'enquérir ou non du nouvel état de l'objet.

L'objectif du pattern OBSERVER est de définir une dépendance du type un-à-plusieurs (1,n) entre des objets de manière que, lorsqu'un objet change d'état, tous les objets dépendants en soient notifiés afin de pouvoir réagir conformément.

1. Diagramme de classe :

Sur ce diagramme UML il y a le pattern Observateur, la classe Gps est l'observable et possède deux observateurs potentiels que sont AfficheResume et AfficheComple.



2. Implémentation Java :

```

1 Observable.java 2 Observable.java 3 Gps.java 4 AfficheResume.java 5 AfficheComplet.java 6 ObserverPatternMain.java
DesignPatterns/src/OBSERVER/Observateur.java
2
3 //Interface implémentée par tous les observateurs.
4 public interface Observateur
5 {
6     // Méthode appelée automatiquement lorsque l'état (position ou précision) du GPS change.
7     public void actualiser(Observable o);
8 }
9
10

```

```

1 Observable.java 2 Observable.java 3 Gps.java 4 AfficheResume.java 5 AfficheComplet.java 6 ObserverPatternMain.java
1 package OBSERVER;
2
3 //Interface implémentée par toutes les classes souhaitant avoir des observateurs à leur écoute.
4 public interface Observable
5 {
6     // Méthode permettant d'ajouter (abonner) un observateur.
7     public void ajouterObservateur(Observateur o);
8     // Méthode permettant de supprimer (réilier) un observateur.
9     public void supprimerObservateur(Observateur o);
10    // Méthode qui permet d'avertir tous les observateurs lors d'un changement d'état.
11    public void notifierObservateurs();
12 }
13

```

```

1 package OBSERVER;
2
3 import java.util.ArrayList;
4
5 //Classe représentant un GPS (appareil permettant de connaître sa position).
6 public class Gps implements Observable
7 {
8     private String position;// Position du GPS.
9     private int precision;// Précision accordé à cette position (suivant le nombre de satellites utilisés).
10    private ArrayList tabObservateur;// Tableau d'observateurs.
11
12    // Constructeur.
13    public Gps()
14    {
15        position="inconnue";
16        precision=0;
17        tabObservateur=new ArrayList();
18    }
19
20    // Permet d'ajouter (s'abonner) un observateur à l'écoute du GPS.
21    public void ajouterObservateur(Observateur o)
22    {
23        tabObservateur.add(o);
24    }
25
26    // Permet de supprimer (résilier) un observateur écoutant le GPS
27    public void supprimerObservateur(Observateur o)
28    {
29        tabObservateur.remove(o);
30    }

```

```

31
32 // Méthode permettant de notifier tous les observateurs lors d'un changement d'état du GPS.
33 public void notifierObservateurs()
34 {
35     for(int i=0;i<tabObservateur.size();i++)
36     {
37         Observateur o = (Observateur) tabObservateur.get(i);
38         o.actualiser(this);// On utilise la méthode "tiré".
39     }
40 }
41
42 // Méthode qui permet de mettre à jour de façon artificielle le GPS.
43 // Dans un cas réel, on utiliserait les valeurs retournées par les capteurs.
44 public void setMesures(String position, int precision)
45 {
46     this.position=position;
47     this.precision=precision;
48     notifierObservateurs();
49 }
50
51 // Méthode "tiré" donc c'est aux observateurs d'aller chercher les valeurs désiré grâce à un objet Gps.
52 // Pour cela on trouve un accesseur en lecture pour position.
53 public String getPosition()
54 {
55     return position;
56 }
57 // Un accesseur en lecture pour précision.
58 public int getPrecision()
59 {
60     return precision;
61 }
62

```

```

1 package OBSERVER;
2
3 //Affiche un résumé en console des informations (position) du GPS.
4 public class AfficheResume implements Observateur
5 {
6     // Méthode appelée automatiquement lors d'un changement d'état du GPS.
7     public void actualiser(Observable o)
8     {
9         if(o instanceof Gps)
10         {
11             Gps g = (Gps) o;
12             System.out.println("Position : "+g.getPosition());
13         }
14     }
15 }
16

```

```

1 package OBSERVER;
2
3 //Affiche en console de façon complète les informations (position et précision) du GPS.
4 public class AfficheComplet implements Observateur
5 {
6     // Méthode appelée automatiquement lors d'un changement d'état du GPS.
7     public void actualiser(Observable o)
8     {
9         if(o instanceof Gps)
10         {
11             Gps g = (Gps) o;
12             System.out.println("Position : "+g.getPosition()+" Précision : "+g.getPrecision()+"/10");
13         }
14     }
15 }
16
17
18

```

```

1 package OBSERVER;
2
3 //Classe principale du projet.
4 public class ObserverPatternMain{
5     // Méthode principale.
6     public static void main(String[] args)
7     {
8         // Création de l'objet Gps observable.
9         Gps g = new Gps();
10        // Création de deux observateurs AfficheResume et AfficheComplet
11        AfficheResume ar = new AfficheResume();
12        AfficheComplet ac = new AfficheComplet();
13        // On ajoute AfficheResume comme observateur de Gps.
14        g.ajouterObservateur(ar);
15        // On simule l'arrivée de nouvelles valeurs via des capteurs.
16        g.setMesures("N 39°59'993 / W 123°00'000", 4);
17        // On ajoute AfficheComplet comme observateur de Gps.
18        g.ajouterObservateur(ac);
19        // Nouvelle simulation d'arrivée de nouvelles valeurs via des capteurs.
20        g.setMesures("N 37°48'898 / W 124°12'011", 5);
21    }
22 }
23
24

```

```

<terminated> ObserverPatternMain [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (10 janv. 2018 14:26:05)
Position : N 39°59'993 / W 123°00'000
Position : N 37°48'898 / W 124°12'011
Position : N 37°48'898 / W 124°12'011 Précision : 5/10

```

MEDIATOR

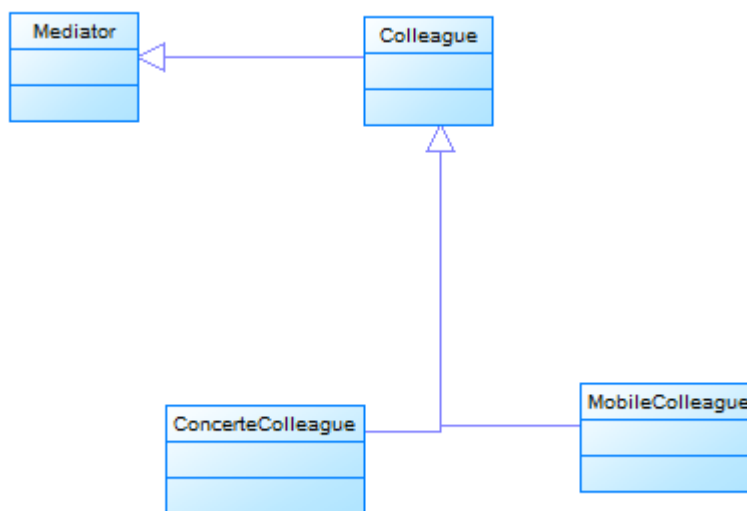
Le développement orienté objet ordinaire distribue la responsabilité aussi loin que possible, avec chaque objet accomplissant sa tâche indépendamment des autres. Par exemple, le pattern OBSERVER supporte cette distribution en limitant la responsabilité d'un objet que d'autres objets trouvent intéressant. Le pattern SINGLETON résiste à la distribution de la responsabilité et vous permet de la centraliser au niveau de certains objets que les clients localisent et réutilisent. A l'instar de SINGLETON, le pattern MEDIATOR centralise la responsabilité mais pour un ensemble spécifique d'objets plutôt que pour tous les clients dans un système. Lorsque les interactions entre les objets reposent sur une condition complexe impliquant que chaque objet d'un groupe connaisse tous les autres, il est utile d'établir une autorité centrale. La centralisation de la responsabilité est également utile lorsque la logique entourant les interactions des objets en relation est indépendante de l'autre comportement des objets.

L'objectif du pattern MEDIATOR est de définir un objet qui encapsule la façon dont un ensemble d'objets interagissent. Cela promeut un couplage lâche, évitant aux objets d'avoir à se référer explicitement les uns aux autres, et permet de varier leur interaction indépendamment.

Le **Médiateur** définit l'interface de communication entre les objets **Collègues**. Le **ConcreteMediator** implémente l'interface médiateur et coordonne la communication entre les de type objets **Collègue**. Il est au courant de tous les collègues et de leur but en ce qui concerne l'inter communication. **ConcreteColleague** communique avec d'autres collègues par l'intermédiaire du médiateur.

Sans ce modèle, tous les collègues se connaîtraient les uns les autres, conduisant à un couplage élevé. En faisant communiquer tous les collègues par un point central, nous avons un système découplé tout en gardant le contrôle sur les interactions de l'objet.

1. Diagramme de classe :



2. Implémentation Java :

```
Colleague.java ApplicationMediator.java ConcreteColleague.java MobileColleague.java MediatorPatternMain.java Mediator.java
1 package MEDIATOR;
2
3 public abstract class Colleague extends Mediator {
4     // Interface Colleague
5
6     public Colleague() {}
7
8
9     // envoie un message via le médiateur
10    public void send ( String message ) {
11        Mediator.send( message , this);
12    }
13    // avoir accès au médiateur
14    //public Mediator getMediator () { return mediateur ;}
15    public abstract void receive (String msg);
16 }
17
18
19
```

```
Colleague.java ApplicationMediator.java ConcreteColleague.java MobileColleague.java MediatorPatternMain.java Mediator.java
1 package MEDIATOR;
2
3 import java.util.ArrayList;
4
5 public class ApplicationMediator {
6
7     private ArrayList<Colleague> colleagues;
8     public ApplicationMediator() {
9         colleagues = new ArrayList<Colleague>();
10    }
11    public void addColleague(Colleague colleague) {
12        colleagues.add(colleague);
13    }
14    public void send(String message, Colleague originator) {
15        //let all other screens know that this screen has changed
16        for(Colleague colleague: colleagues) {
17            //don't tell ourselves
18            if(colleague != originator) {
19                colleague.receive(message);
20            }
21        }
22    }
23 }
24
```

```
Colleague.java ApplicationMediator.java ConcreteColleague.java MobileColleague.java MediatorPatternMain.java Mediator.java
1 package MEDIATOR;
2
3 public class ConcreteColleague extends Colleague {
4
5
6
7     public void receive(String message) {
8         super.send(message);
9         System.out.println("Colleague Received: " + message);
10    }
11
12
13
14 }
15
```

```
Colleague.java ApplicationMediator.java ConcreteColleague.java MobileColleague.java MediatorPatternMain.java Mediator.java
1 package MEDIATOR;
2
3 public class MobileColleague extends Colleague{
4
5
6     public void receive(String message) {
7         super.send(message);
8     }
9     System.out.println("Mobile Received: " + message);
10 }
11
12
13
14 }
15
```



```
1 package MEDIATOR;
2
3 public class Mediator {
4     public static void send(String message, Colleague colleague) {
5         System.out.println("send: " + message);
6     }
7 }
8
9
```

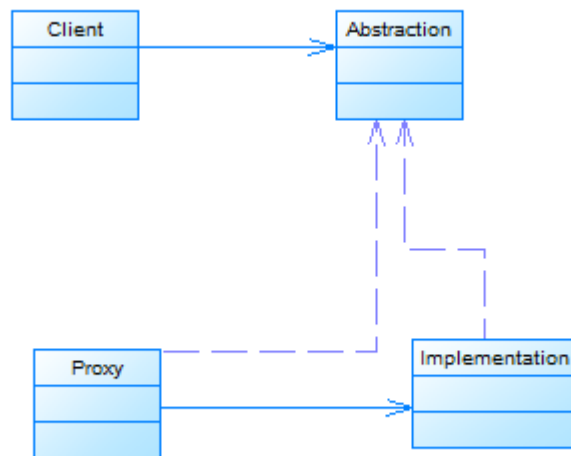
```
1 package MEDIATOR;
2
3 public class MediatorPatternMain {
4     public static void main(String[] args) {
5         ApplicationMediator mediator = new ApplicationMediator();
6         ConcreteColleague desktop = new ConcreteColleague();
7         MobileColleague mobile = new MobileColleague();
8         mediator.addColleague(desktop);
9         mediator.addColleague(mobile);
10        desktop.send("Hello World");
11        mobile.send("Hello");
12        // desktop.receive("received desk");
13        // mobile.receive("received mob");
14    }
15 }
16
17
```

Problems | Javadoc | Declaration | Console

<terminated> MediatorPatternMain [Java Application] C:\Program Files\Java\jdk1.8.0_20\bin\javaw.exe (11 janv. 2018 00:49:06)

send: Hello World
send: Hello

3. Diagramme de classe :



4. Implémentation Java :

```
Abstraction.java  Implementation.java  Proxy.java  ProxyPatternMain.java
1 package PROXY;
2
3 /**
4  * Définit l'interface
5  */
6 public interface Abstraction {
7
8     /**
9      * Méthode pour laquelle on souhaite un "Proxy"
10     */
11     public void afficher();
12 }
13
```

```
Abstraction.java  Implementation.java  Proxy.java  ProxyPatternMain.java
1 package PROXY;
2
3 /**
4  * Implémentation de l'interface.
5  * Définit l'objet représenté par le "Proxy"
6  */
7 public class Implementation implements Abstraction {
8
9     public void afficher() {
10         System.out.println("Méthode afficher() de la classe d'implémentation");
11     }
12 }
```



```
Abstraction.java  Implementation.java  Proxy.java  ProxyPatternMain.java
1 package PROXY;
2
3 /**
4  * Intermédiaire entre la partie cliente et l'implémentation
5  */
6 public class Proxy implements Abstraction {
7
8     /**
9      * Instancie l'objet "Implementation", pour appeler
10     * la vraie implémentation de la méthode.
11     */
12     public void afficher() {
13         System.out.println("--> Méthode afficher() du Proxy : ");
14         System.out.println("--> Création de l'objet Implementation au besoin");
15         Implementation lImplementation = new Implementation();
16         System.out.println("--> Appel de la méthode afficher() de l'objet Implementation");
17         lImplementation.afficher();
18     }
19 }
20
```

```
Abstraction.java  Implementation.java  Proxy.java  ProxyPatternMain.java
1 package PROXY;
2
3 public class ProxyPatternMain {
4
5     public static void main(String[] args) {
6         // Création du "Proxy"
7         Abstraction lProxy = new Proxy();
8
9         // Appel de la méthode du "Proxy"
10        lProxy.afficher();
11    }
12 }
13
14 }
15
```

Problems @ Javadoc Declaration Console

<terminated> ProxyPatternMain [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (10 janv. 2018 20:30:49)

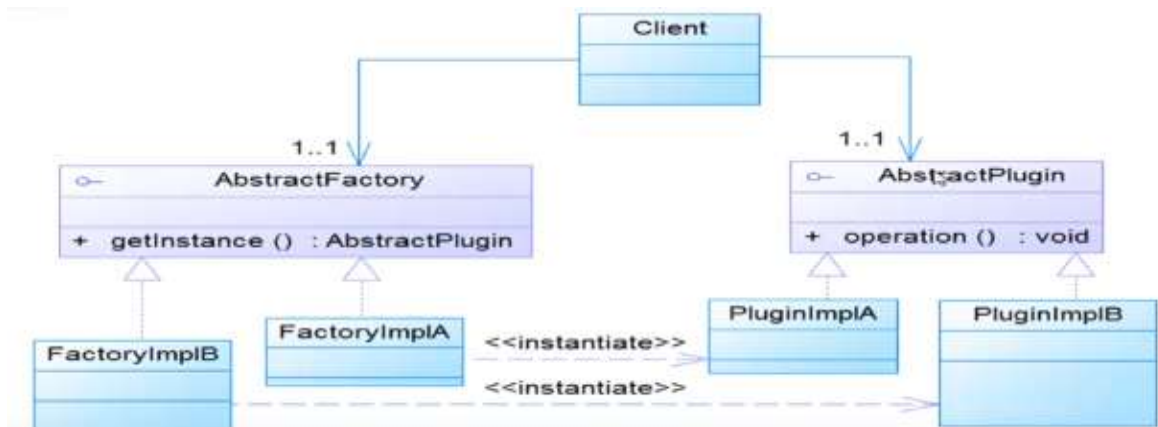
--> Méthode afficher() du Proxy :
--> Création de l'objet Implementation au besoin
--> Appel de la méthode afficher() de l'objet Implementation
Méthode afficher() de la classe d'implémentation

Abstract Factory

Objectif

Abstract Factory (Fabrique Abstraite) permet de fournir une interface pour créer des objets d'une même famille sans préciser leurs classe concrètes.

✓ Diagramme de classe



✓ Code Java

```
package Abstract_Factory;

public interface AbstractFactory {
    public AbstractPlugin getInstance();
}
```

```
package Abstract_Factory;

public interface AbstractPlugin {
    public void traitement();
}
```

```
package Abstract_Factory;

public class FactoryImplA implements AbstractFactory{

    public AbstractPlugin getInstance(){
        return new PluginImplA();
    }

}
```

```
package Abstract_Factory;

public class FactoryImplB implements AbstractFactory{

    public AbstractPlugin getInstance(){
        return new PluginImplB();
    }

}
```

```
package Abstract_Factory;

public class PluginImplA implements AbstractPlugin{

    public void traitement() {
        System.out.println("traitement du plugin A");
    }

}
```

```
package Abstract_Factory;

public class PluginImplB implements AbstractPlugin{

    public void traitement() {
        System.out.println("traitement du plugin B");
    }

}
```

```

package Abstract_Factory;

public class Client {

    public static void main(String[] args) {
        AbstractFactory factory=new FactoryImplA();
        AbstractPlugin plugin= factory.getInstance();
        plugin.traitemement();
    }

}

```

<terminated> Client (1) [Java Application]

60.0

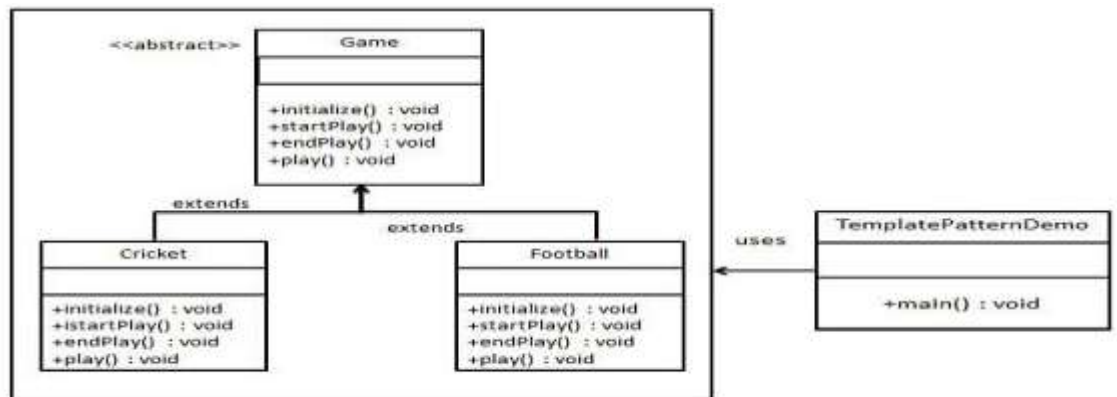
104.0

- Template Method

- ✓ Objectif

définir le squelette d'un algorithme en déléguant certaine étapes à des sous-classes, ce qui nous permet d'isoler la partie variable d'un algorithme.

- ✓ Diagramme de classe



- ✓ Code java

```

public abstract class Game {
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();

    //template method
    public final void play(){

        //initialize the game
        initialize();

        //start game
        startPlay();

        //end game
        endPlay();
    }
}

```

```

public class Cricket extends Game {

    @Override
    void endPlay() {
        System.out.println("Cricket Game Finished!");
    }

    @Override
    void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }
}

```

```

public class Football extends Game {

    @Override
    void endPlay() {
        System.out.println("Football Game Finished!");
    }

    @Override
    void initialize() {
        System.out.println("Football Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Football Game Started. Enjoy the game!");
    }
}

public class TemplatePatternDemo {
    public static void main(String[] args) {

        Game game = new Cricket();
        game.play();
        System.out.println();
        game = new Football();
        game.play();
    }
}

```

```

Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

```

```

Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!

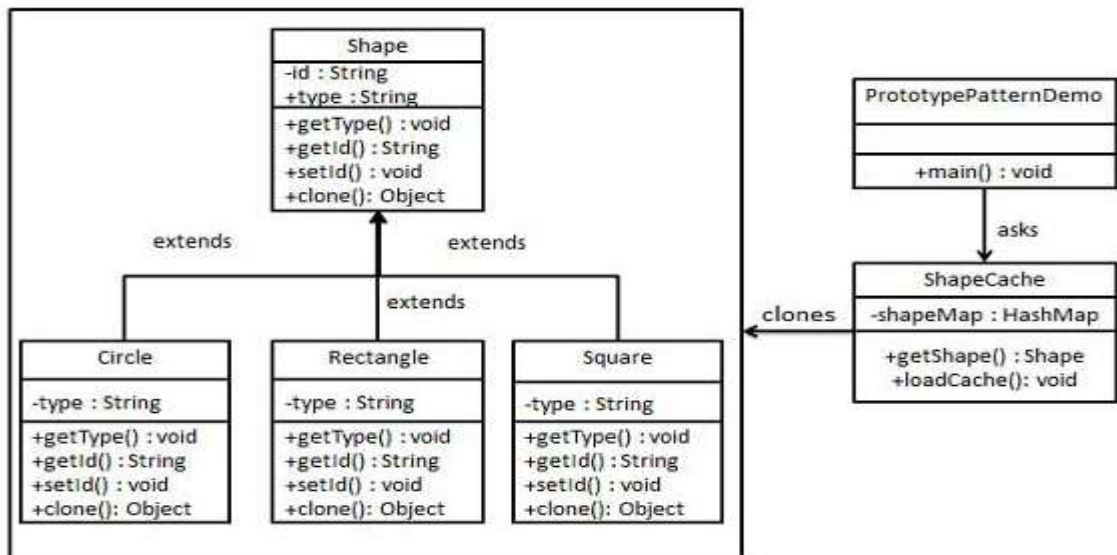
```

Pattern Prototype

✓ Objectif

Permet de fournir de nouveaux objets par la copie d'un exemple, plutôt que de produire de nouvelles instances non initialisées d'une classe

✓ Diagramme de classe



✓ Code java

```
public abstract class Shape implements Cloneable {

    private String id;
    protected String type;

    abstract void draw();

    public String getType(){
        return type;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Object clone() {
        Object clone = null;

        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }

        return clone;
    }
}
```

```

public class Rectangle extends Shape {

    public Rectangle(){
        type = "Rectangle";
    }

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}

```

```

public class Square extends Shape {

    public Square(){
        type = "Square";
    }

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}

```

```

public class Circle extends Shape {

    public Circle(){
        type = "Circle";
    }

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

```

```

import java.util.Hashtable;

public class ShapeCache {

    private static Hashtable<String, Shape> shapeMap = new Hashtable<String, Shape>();

    public static Shape getShape(String shapeId) {
        Shape cachedShape = shapeMap.get(shapeId);
        return (Shape) cachedShape.clone();
    }

    // for each shape run database query and create shape
    // shapeMap.put(shapeKey, shape);
    // for example, we are adding three shapes

    public static void loadCache() {
        Circle circle = new Circle();
        circle.setId("1");
        shapeMap.put(circle.getId(), circle);

        Square square = new Square();
        square.setId("2");
        shapeMap.put(square.getId(), square);

        Rectangle rectangle = new Rectangle();
        rectangle.setId("3");
        shapeMap.put(rectangle.getId(), rectangle);
    }
}

```

```

public class PrototypePatternDemo {
    public static void main(String[] args) {
        ShapeCache.loadCache();

        Shape clonedShape = (Shape) ShapeCache.getShape("1");
        System.out.println("Shape : " + clonedShape.getType());

        Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
        System.out.println("Shape : " + clonedShape2.getType());

        Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
        System.out.println("Shape : " + clonedShape3.getType());
    }
}

```

```

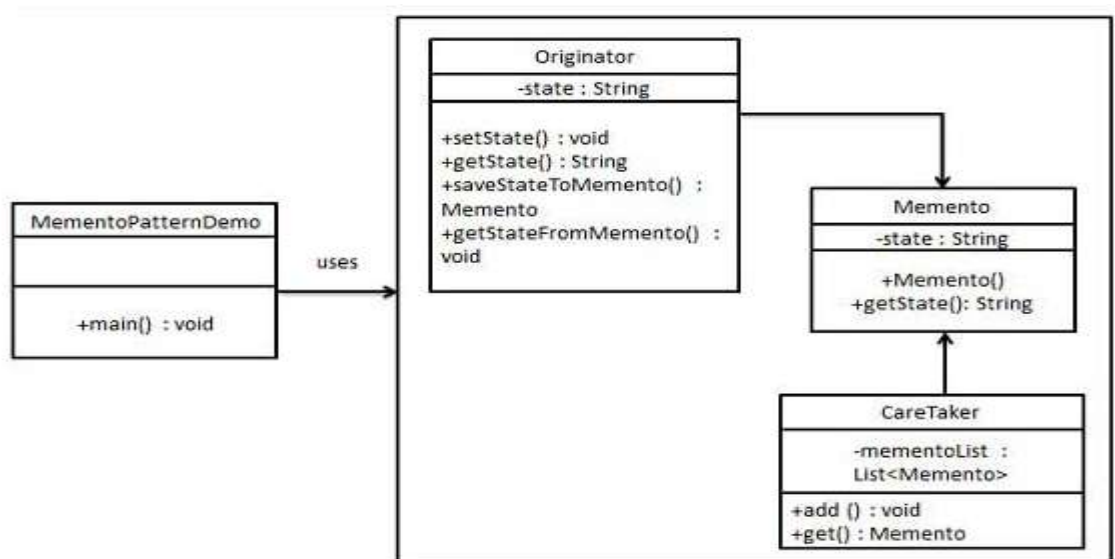
Shape : Circle
Shape : Square
Shape : Rectangle

```

Pattern Memento :

Objectif Permet de restaurer un état précédent d'un objet. Le memento est utilisé par deux objets : le *créateur* et le *gardien*. Le créateur est un objet ayant un état interne (état à sauvegarder). Le gardien agit sur le créateur de manière à conserver la possibilité de revenir en arrière. Pour cela, le gardien demande au créateur, lors de chaque action, un objet memento qui sauvegarde l'état de l'objet *créateur* avant la modification.

✓ Diagramme de classe :



✓ Code java :


```

public class Memento {
    private String state;

    public Memento(String state){
        this.state = state;
    }

    public String getState(){
        return state;
    }
}

```

```

public class Originator {
    private String state;

    public void setState(String state){
        this.state = state;
    }

    public String getState(){
        return state;
    }

    public Memento saveStateToMemento(){
        return new Memento(state);
    }

    public void getStateFromMemento(Memento memento){
        state = memento.getState();
    }
}

```

```

import java.util.ArrayList;
import java.util.List;

public class CareTaker {
    private List<Memento> mementoList = new ArrayList<Memento>();

    public void add(Memento state){
        mementoList.add(state);
    }

    public Memento get(int index){
        return mementoList.get(index);
    }
}

```

```

public class MementoPatternDemo {
    public static void main(String[] args) {

        Originator originator = new Originator();
        CareTaker careTaker = new CareTaker();

        originator.setState("State #1");
        originator.setState("State #2");
        careTaker.add(originator.saveStateToMemento());

        originator.setState("State #3");
        careTaker.add(originator.saveStateToMemento());

        originator.setState("State #4");
        System.out.println("Current State: " + originator.getState());

        originator.getStateFromMemento(careTaker.get(0));
        System.out.println("First saved State: " + originator.getState());
        originator.getStateFromMemento(careTaker.get(1));
        System.out.println("Second saved State: " + originator.getState());
    }
}

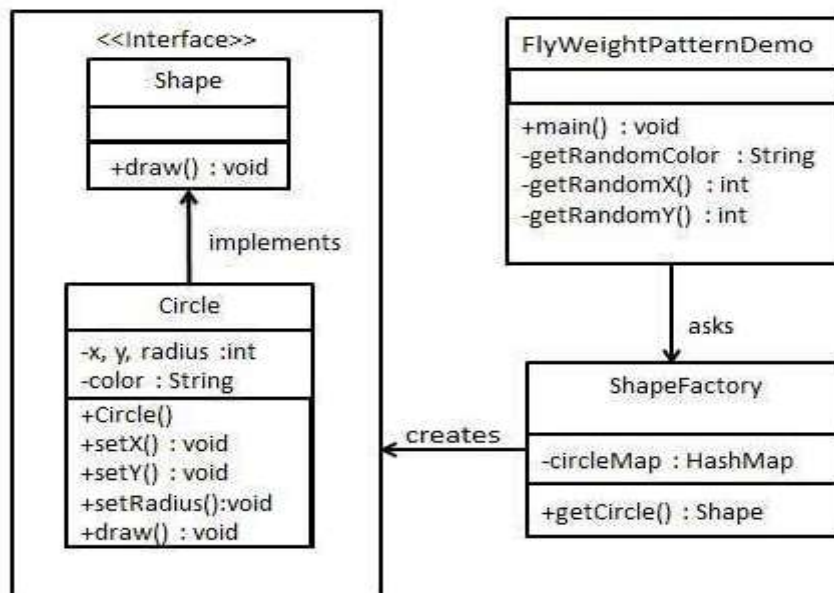
```

Flyweight Pattern

✓ Objectif

Souvent lors de l'utilisation d'un programme on se retrouve à instancier de nombreux objets, chacun prenant une certaine quantité de mémoire on peut donc se demander comment réduire la place des objets instanciés, on va donc utiliser le design pattern FlyWeight qui va en plus accélérer la vitesse d'exécution du programme.

✓ Diagramme de classe



✓ Code java

```
public interface Shape {
    void draw();
}
```

```

public class Circle implements Shape {
    private String color;
    private int x;
    private int y;
    private int radius;

    public Circle(String color){
        this.color = color;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void setRadius(int radius) {
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y : " + y + "]");
    }
}

```

```

import java.util.HashMap;

public class ShapeFactory {

    // Uncomment the compiler directive line and
    // javac *.java will compile properly.
    // @SuppressWarnings("unchecked")
    private static final HashMap circleMap = new HashMap();

    public static Shape getCircle(String color) {
        Circle circle = (Circle)circleMap.get(color);

        if(circle == null) {
            circle = new Circle(color);
            circleMap.put(color, circle);
            System.out.println("Creating circle of color : " + color);
        }
        return circle;
    }
}

```

```

public class FlyweightPatternDemo {
    private static final String colors[] = { "Red", "Green", "Blue", "White", "Black" };
    public static void main(String[] args) {

        for(int i=0; i < 20; ++i) {
            Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());
            circle.setX(getRandomX());
            circle.setY(getRandomY());
            circle.setRadius(100);
            circle.draw();
        }
    }
    private static String getRandomColor() {
        return colors[(int)(Math.random()*colors.length)];
    }
    private static int getRandomX() {
        return (int)(Math.random()*100 );
    }
    private static int getRandomY() {
        return (int)(Math.random()*100);
    }
}

```

```

Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100

```

Factory_pattern

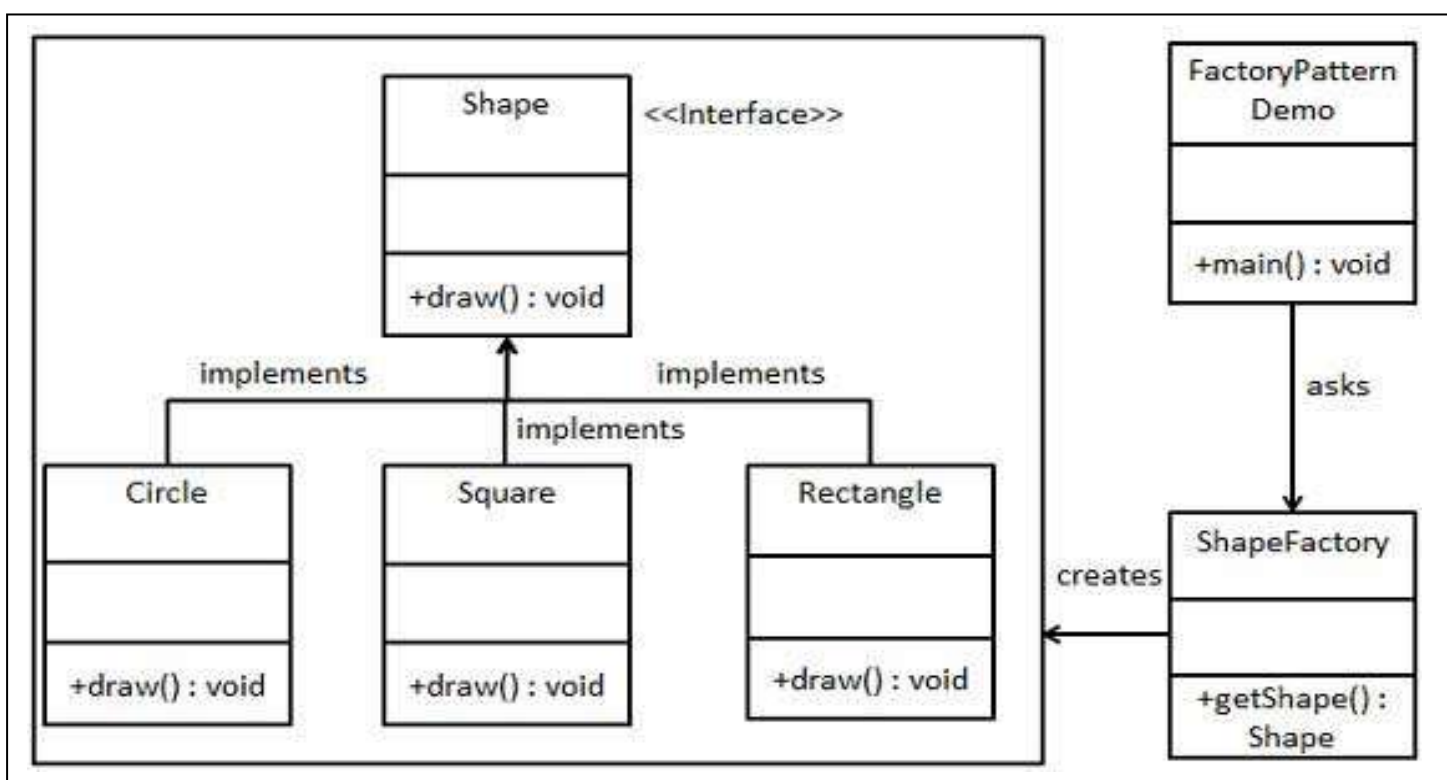
Factory pattern est l'un des motifs de conception les plus utilisés en Java. Ce type de modèle de conception est sous un modèle de création car ce modèle fournit l'une des meilleures façons de créer un objet.

Dans le modèle Factory pattern, nous créons un objet sans exposer la logique de création au client et nous référons à l'objet nouvellement créé en utilisant une interface commune.

Implémentation

Nous allons créer une interface Shape(forme) et des classes concrètes implémentant l'interface Shape. Une classe pattern ShapeFactory est définie comme une étape suivante.

FactoryPatternDemo, notre classe de démonstration utilisera ShapeFactory pour obtenir un objet Shape. Il transmettra les informations (CIRCLE / RECTANGLE / SQUARE) à ShapeFactory pour obtenir le type d'objet dont il a besoin.



Classe shape

```
Shape.java x Rectangle.java Square.java Circle.java ShapeFactory.java FactoryPatternDemo.java
1 package DesignPatternFactory;
2
3 public interface Shape {
4     void draw();
5 }
```

Classe Rectangle

```
Shape.java Rectangle.java x Square.java Circle.java ShapeFactory.java FactoryPatternDemo.java
1 package DesignPatternFactory;
2
3 public class Rectangle implements Shape{
4     public void draw() {
5         System.out.println("Inside Rectangle::draw() method.");
6     }
7 }
```

Classe Square

```
Shape.java Rectangle.java Square.java x Circle.java ShapeFactory.java FactoryPatternDemo.java
1 package DesignPatternFactory;
2
3 public class Square implements Shape {
4     public void draw() {
5         System.out.println("Inside Square::draw() method.");
6     }
7 }
```

Classe Circle

```
Shape.java Rectangle.java Square.java Circle.java x ShapeFactory.java FactoryPatternDemo.java
1 package DesignPatternFactory;
2
3 public class Circle implements Shape {
4     public void draw() {
5         System.out.println("Inside Circle::draw() method.");
6     }
7 }
8
```

Classe shapeFactory

```
Shape.java  Rectangle.java  Square.java  Circle.java  ShapeFactory.java  FactoryPatternDemo.java

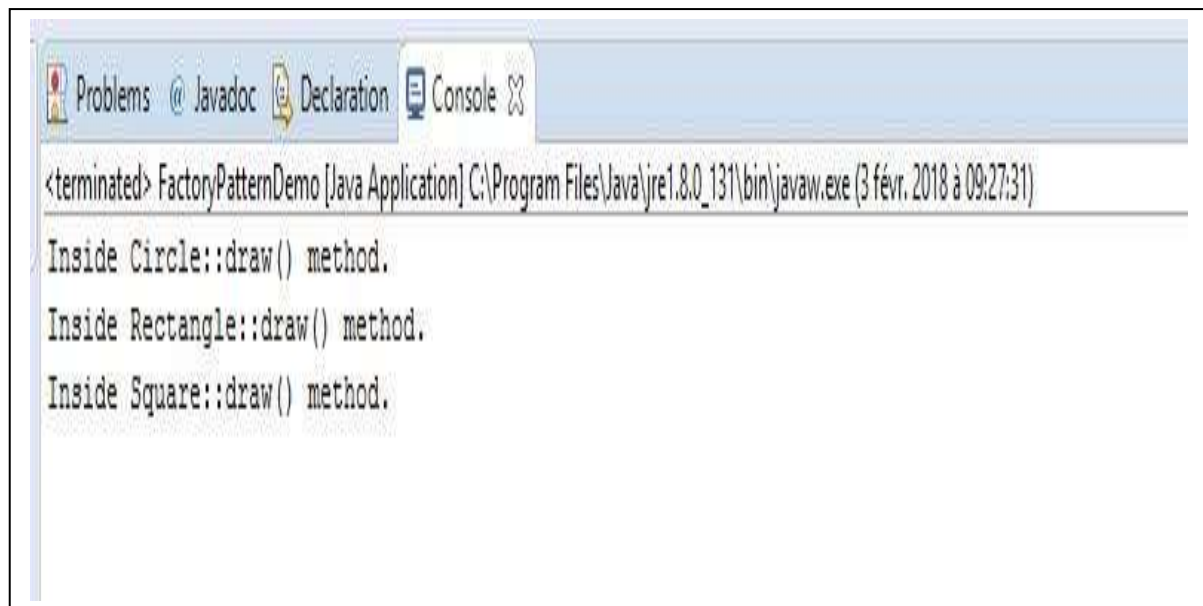
1 package DesignPatternFactory;
2
3 public class ShapeFactory {
4
5     //use getShape method to get object of type shape
6     public Shape getShape(String shapeType){
7         if(shapeType == null){
8             return null;
9         }
10        if(shapeType.equalsIgnoreCase("CIRCLE")){
11            return new Circle();
12        }
13        else if(shapeType.equalsIgnoreCase("RECTANGLE")){
14            return new Rectangle();
15        }
16        else if(shapeType.equalsIgnoreCase("SQUARE")){
17            return new Square();
18        }
19
20        return null;
21    }
22 }
23
```

Classe FactoryPatternDemo

```
Shape.java  Rectangle.java  Square.java  Circle.java  ShapeFactory.java  FactoryPatternDemo.java

1 package DesignPatternFactory;
2
3 public class FactoryPatternDemo {
4
5     public static void main(String[] args) {
6         ShapeFactory shapeFactory = new ShapeFactory();
7
8         //get an object of Circle and call its draw method.
9         Shape shape1 = shapeFactory.getShape("CIRCLE");
10
11        //call draw method of Circle
12        shape1.draw();
13
14        //get an object of Rectangle and call its draw method.
15        Shape shape2 = shapeFactory.getShape("RECTANGLE");
16
17        //call draw method of Rectangle
18        shape2.draw();
19
20        //get an object of Square and call its draw method.
21        Shape shape3 = shapeFactory.getShape("SQUARE");
22
23        //call draw method of circle
24        shape3.draw();
25    }
26 }
```


Exécution



The screenshot shows a console window from an IDE. The title bar includes tabs for 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console output shows the application has terminated and lists the draw() methods for Circle, Rectangle, and Square classes.

```
<terminated> FactoryPatternDemo [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (3 févr. 2018 à 09:27:31)  
Inside Circle::draw() method.  
Inside Rectangle::draw() method.  
Inside Square::draw() method.
```


Builder_pattern

Le modèle Builder construit un objet complexe en utilisant des objets simples et en utilisant une approche pas à pas. Ce type de modèle de conception est sous un modèle de création car ce modèle fournit l'une des meilleures façons de créer un objet.

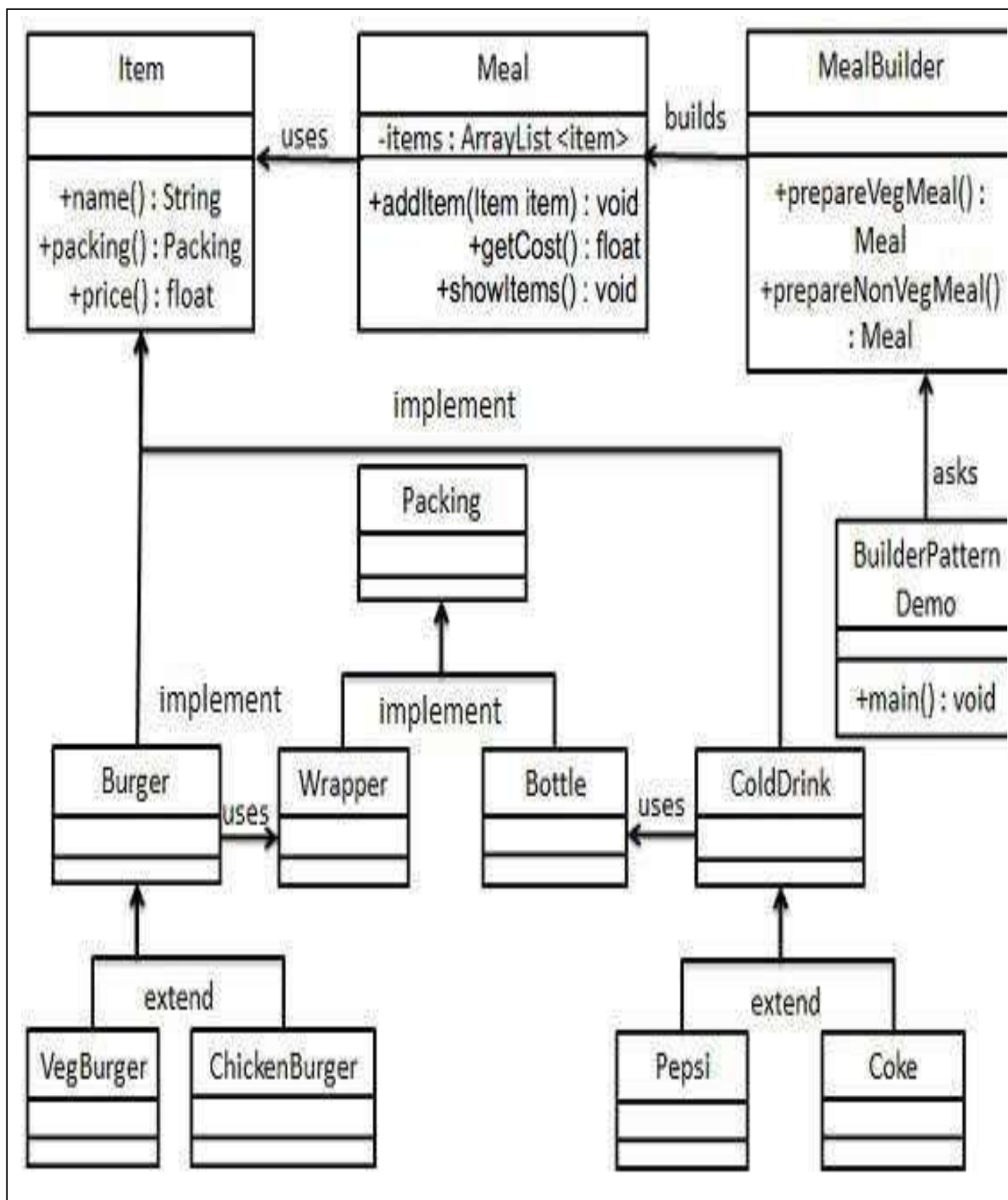
Une classe Builder génère l'objet final étape par étape. Ce générateur est indépendant des autres objets.

Implémentation

Nous avons examiné une analyse de rentabilisation d'un restaurant-minute où un repas typique pourrait être un hamburger et une boisson froide. Burger pourrait être soit un hamburger de légumes ou un hamburger de poulet et sera emballé par un emballage. La boisson froide peut être un coke ou un pepsi et sera emballée dans une bouteille.

Nous allons créer une interface Item représentant des produits alimentaires tels que des hamburgers et des boissons froides et des classes de béton mettant en œuvre l'interface Item et une interface Packing représentant des emballages de produits alimentaires et des classes de béton implémentant l'interface Packing. serait emballé comme une bouteille.

Nous créons ensuite une classe Meal ayant ArrayList of Item et un MealBuilder pour construire différents types d'objets Meal en combinant Item. BuilderPatternDemo, notre classe de démonstration utilisera MealBuilder pour construire un repas.



Interface Item

```
Item.java X Pepsi.java Coke.java MealBuilder....
1 package Builder_Pattern;
2
3 public interface Item {
4     public String name();
5     public Packing packing();
6     public float price();
7 }
8
```

Interface Packing

```
Packing.java X Meal.java Coke.java MealBuilder.java
1 package Builder_Pattern;
2
3 public interface Packing {
4     public String pack();
5 }
6
```

Classe Wrapper

```
Wrapper.java X Meal.java Coke.java MealBuilder....
1 package Builder_Pattern;
2
3 public class Wrapper implements Packing{
4     public String pack() {
5         return "Wrapper";
6     }
7 }
8
```

Classe Bottle

```
Bottle.java Meal.java Coke.java MealBuilder... BuilderPatt...
1 package Builder_Pattern;
2
3 public class Bottle implements Packing {
4
5     public String pack() {
6         return "Bottle";
7     }
8 }
9
```

Classe burger

```
Burger.java ColdDrink.java VegBurger.java ChickenBurg... Coke.java
1 package Builder_Pattern;
2
3 public abstract class Burger implements Item {
4
5     public Packing packing() {
6         return new Wrapper();
7     }
8     public abstract float price();
9 }
10
```

Classe ColdDrink

```
Burger.java ColdDrink.java VegBurger.java ChickenBurg... Coke.java
1 package Builder_Pattern;
2
3 public abstract class ColdDrink implements Item {
4
5     public Packing packing() {
6         return new Bottle();
7     }
8
9     public abstract float price();
10 }
```


Classe VegBurger

```
Burger.java  ColdDrink.java  VegBurger.java  ChickenBurg...  Coke.java

1 package Builder_Pattern;
2
3 public class VegBurger extends Burger {
4
5
6     public float price() {
7         return 25.0f;
8     }
9
10
11     public String name() {
12         return "Veg Burger";
13     }
14 }
```

Classe Chicken Burger

```
Burger.java  ColdDrink.java  VegBurger.java  ChickenBurg...  Coke.java

1 package Builder_Pattern;
2
3 public class ChickenBurger extends Burger {
4
5     @Override
6     public float price() {
7         return 50.5f;
8     }
9
10     @Override
11     public String name() {
12         return "Chicken Burger";
13     }
14 }
15
```

Classe Coke

```
Burger.java ColdDrink.java VegBurger.java ChickenBurg... Coke.java
1 package Builder_Pattern;
2
3 public class Coke extends ColdDrink {
4
5
6     public float price() {
7         return 30.0f;
8     }
9
10
11     public String name() {
12         return "Coke";
13     }
14 }
```

Classe Pepsi

```
Pepsi.java Meal.java Coke.java MealBuilder.... BuilderPatt..
1 package Builder_Pattern;
2
3 public class Pepsi extends ColdDrink {
4
5     @Override
6     public float price() {
7         return 35.0f;
8     }
9
10     @Override
11     public String name() {
12         return "Pepsi";
13     }
14 }
```

Classe meal

```
MealBuilder.java BuilderPatternDemo.java Meal.java ✖
1 package Builder_Pattern;
2
3 import java.util.ArrayList;
4
5 public class Meal {
6     private List<Item> items = new ArrayList<Item>();
7
8     public void addItem(Item item){
9         items.add(item);
10    }
11
12    public float getCost(){
13        float cost = 0.0f;
14
15        for (Item item : items) {
16            cost += item.price();
17        }
18        return cost;
19    }
20
21    public void showItems(){
22        for (Item item : items) {
23            System.out.print("Item : " + item.name());
24            System.out.print(", Packing : " + item.packing().pack());
25            System.out.println(", Price : " + item.price());
26        }
27    }
28 }
29
30 }
```

Classe mealBuilder

```
MealBuilder.java ✖ BuilderPatternDemo.java
1 package Builder_Pattern;
2
3 public class MealBuilder {
4
5     public Meal prepareVegMeal (){
6         Meal meal = new Meal();
7         meal.addItem(new VegBurger());
8         meal.addItem(new Coke());
9         return meal;
10    }
11
12    public Meal prepareNonVegMeal (){
13        Meal meal = new Meal();
14        meal.addItem(new ChickenBurger());
15        meal.addItem(new Pepsi());
16        return meal;
17    }
18 }
19 }
```


Classe BuilderPatternDemo

```
MealBuilder.java  BuilderPatternDemo.java X
1 package Builder_Pattern;
2
3 public class BuilderPatternDemo {
4     public static void main(String[] args) {
5
6         MealBuilder mealBuilder = new MealBuilder();
7
8         Meal vegMeal = mealBuilder.prepareVegMeal();
9         System.out.println("Veg Meal");
10        vegMeal.showItems();
11        System.out.println("Total Cost: " + vegMeal.getCost());
12
13        Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
14        System.out.println("\n\nNon-Veg Meal");
15        nonVegMeal.showItems();
16        System.out.println("Total Cost: " + nonVegMeal.getCost());
17    }
18 }
19
```

Execution

```
Problems @ Javadoc Declaration Console X
<terminated> BuilderPatternDemo [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (3 févr. 2018 à 20:32:47)
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0

Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```


Chain_of_responsabiliy_pattern

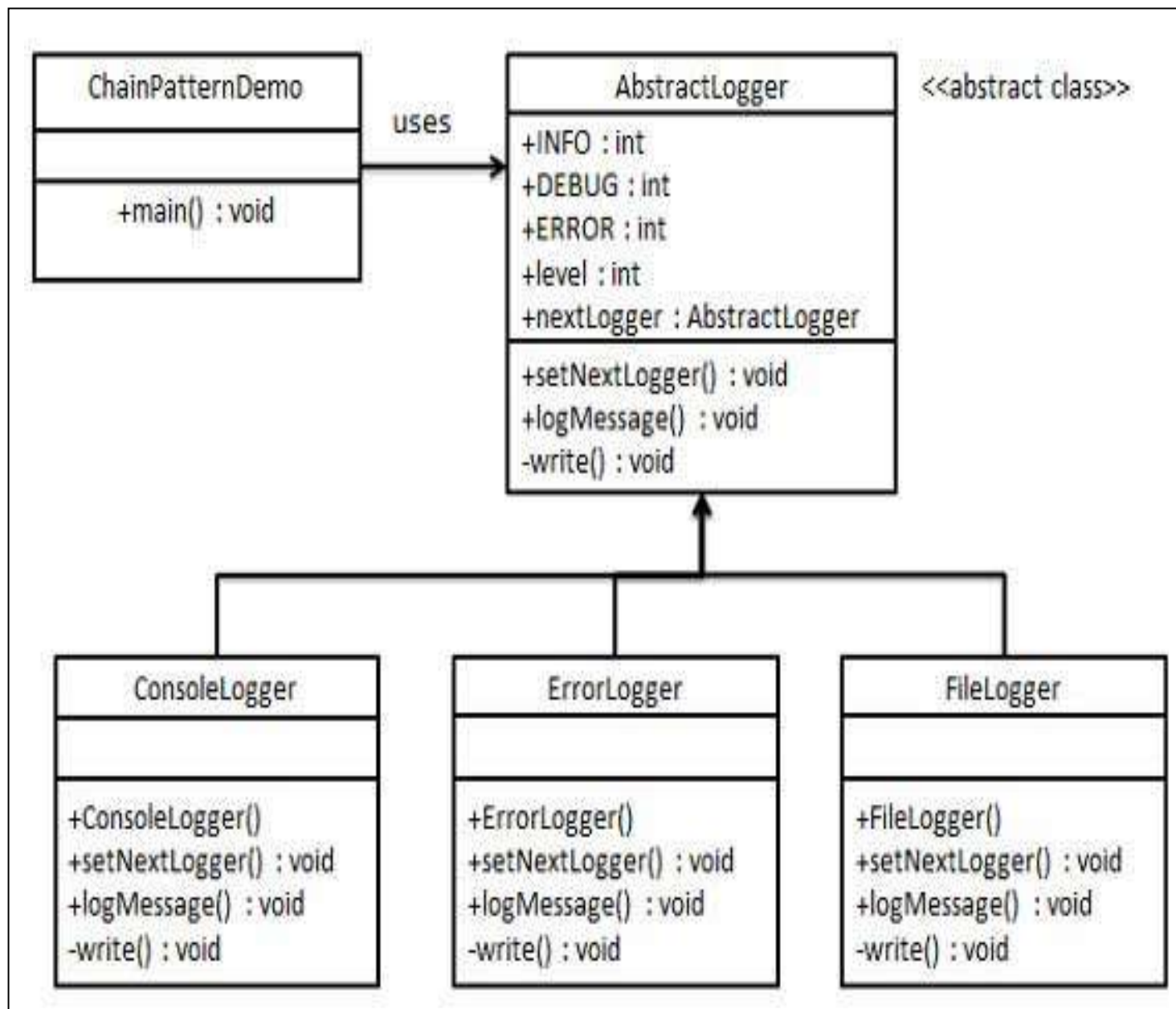
Comme son nom l'indique, le modèle de chaîne de responsabilité crée une chaîne d'objets récepteurs pour une requête. Ce motif découple l'expéditeur et le destinataire d'une requête en fonction du type de **demande**. Ce modèle relève de modèles comportementaux.

Dans ce modèle, normalement chaque récepteur contient une référence à un autre récepteur. Si un objet ne peut pas gérer la requête, il passe la même chose au destinataire suivant et ainsi de suite.

Implementation

Nous avons créé une classe abstraite AbstractLogger avec un niveau de journalisation. Ensuite, nous avons créé trois types d'enregistreurs qui étendent AbstractLogger. Chaque enregistreur vérifie le niveau du message à son niveau et imprime en conséquence autrement ne pas imprimer et passer le message à son prochain enregistreur.

Diagramme de classe :



Classe abstractlogger

```
AbstractLogger.java  ErrorLogger.java  ConsoleLogger.java  FileLogger.java  ChainPatternDemo.java
1 package chain_of_responsability;
2
3 public abstract class AbstractLogger {
4     public static int INFO = 1;
5     public static int DEBUG = 2;
6     public static int ERROR = 3;
7
8     protected int level;
9
10    //next element in chain or responsibility
11    protected AbstractLogger nextLogger;
12
13    public void setNextLogger(AbstractLogger nextLogger) {
14        this.nextLogger = nextLogger;
15    }
16
17    public void logMessage(int level, String message) {
18        if(this.level <= level) {
19            write(message);
20        }
21        if(nextLogger != null) {
22            nextLogger.logMessage(level, message);
23        }
24    }
25
26    abstract protected void write(String message);
27
28 }
29
```

Classe consolelogger

```
AbstractLogger.java  ErrorLogger.java  ConsoleLogger.java  FileLogger.java  ChainPatternDemo.java
1 package chain_of_responsability;
2
3 public class ConsoleLogger extends AbstractLogger {
4
5     public ConsoleLogger(int level) {
6         this.level = level;
7     }
8
9     @Override
10    protected void write(String message) {
11        System.out.println("Standard Console::Logger: " + message);
12    }
13 }
14
```

Classe errorlogger

```
AbstractLogger.java  ErrorLogger.java  ConsoleLogger.java  FileLogger.java  ChainPatternDemo.java

1 package chain_of_responsability;
2
3 public class ErrorLogger extends AbstractLogger {
4
5     public ErrorLogger(int level){
6         this.level = level;
7     }
8
9     @Override
10    protected void write(String message) {
11        System.out.println("Error Console::Logger: " + message);
12    }
13 }
14
```

classefilelogger

```
AbstractLogger.java  ErrorLogger.java  ConsoleLogger.java  FileLogger.java  ChainPatternDemo.java

1 package chain_of_responsability;
2
3 public class FileLogger extends AbstractLogger {
4
5     public FileLogger(int level){
6         this.level = level;
7     }
8
9     @Override
10    protected void write(String message) {
11        System.out.println("File::Logger: " + message);
12    }
13 }
14
```


Classe chainepatterndemo

```
AbstractLogger.java  ErrorLogger.java  ConsoleLogger.java  FileLogger.java  ChainPatternDemo.java X

1 package chain_of_responsability;
2
3 public class ChainPatternDemo {
4
5     private static AbstractLogger getChainOfLoggers(){
6
7         AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
8         AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
9         AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);
10
11         errorLogger.setNextLogger(fileLogger);
12         fileLogger.setNextLogger(consoleLogger);
13
14         return errorLogger;
15     }
16
17     public static void main(String[] args) {
18         AbstractLogger loggerChain = getChainOfLoggers();
19
20         loggerChain.logMessage(AbstractLogger.INFO,
21             "This is an information.");
22
23         loggerChain.logMessage(AbstractLogger.DEBUG,
24             "This is an debug level information.");
25
26         loggerChain.logMessage(AbstractLogger.ERROR,
27             "This is an error information.");
28     }
29 }
```

Execution

Problems @ Javadoc Declaration Console

<terminated> BuilderPatternDemo [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (3 févr. 2018 à 20:57:05)

Veg Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Veg Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5

state_pattern

Dans le modèle state, un comportement de classe change en fonction de son état. Ce type de modèle de conception est soumis à un modèle de comportement.

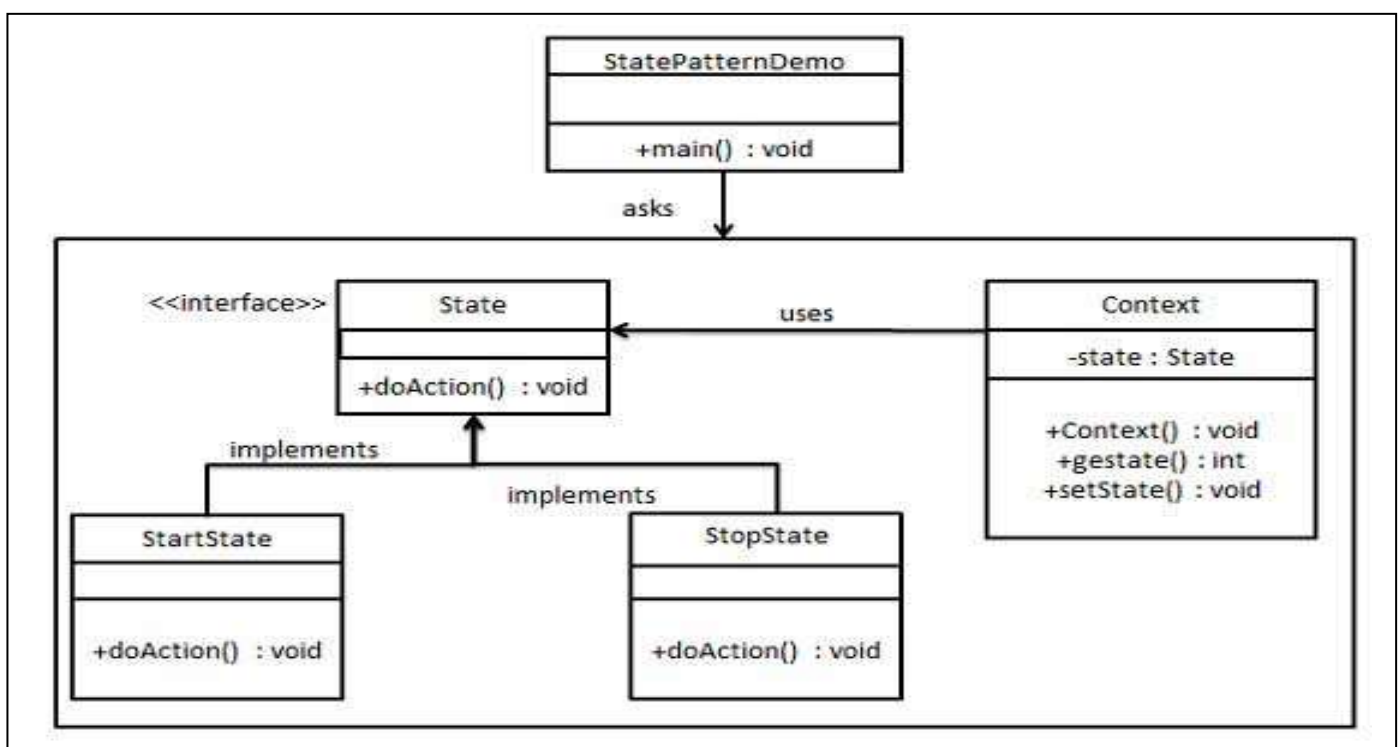
Dans le modèle d'état, nous créons des objets qui représentent différents états et un objet de contexte dont le comportement varie à mesure que son objet d'état change.

implementation

Nous allons créer une interface state définissant une action et des classes d'états concrètes implémentant l'interfacestate. Le contexte est une classe qui porte un État.

StatePatternDemo, notre classe de démonstration, utilisera les objets Context et state pour démontrer les changements de comportement du contexte en fonction du type d'état dans lequel il se trouve.

Diagramme de classe :



Classe state

```
State.java StartState.java StopState.java Context.java
1 package state_pattern;
2
3 public interface State {
4     public void doAction(Context context);
5 }
6
```

Classe stateState

```
State.java StartState.java StopState.java Context.java StatePatternDemo.java
1 package state_pattern;
2
3 public class StartState implements State {
4
5     public void doAction(Context context) {
6         System.out.println("Player is in start state");
7         context.setState(this);
8     }
9
10    public String toString(){
11        return "Start State";
12    }
13 }
```

Classe stopstate

```
State.java StartState.java StopState.java Context.java StatePatternDemo.java
1 package state_pattern;
2
3 public class StopState implements State {
4
5     public void doAction(Context context) {
6         System.out.println("Player is in stop state");
7         context.setState(this);
8     }
9
10    public String toString(){
11        return "Stop State";
12    }
13 }
14
```


Classe Contexte

```
State.java StartState.java StopState.java Context.java StatePatternDemo.java
1 package state_pattern;
2
3 public class Context {
4     private State state;
5
6     public Context() {
7         state = null;
8     }
9
10    public void setState(State state) {
11        this.state = state;
12    }
13
14    public State getState() {
15        return state;
16    }
17 }
```

Classe statePatternDemo

```
State.java StartState.java StopState.java Context.java StatePatternDemo.java
1 package state_pattern;
2 public class StatePatternDemo {
3     public static void main(String[] args) {
4         Context context = new Context();
5
6         StartState startState = new StartState();
7         startState.doAction(context);
8
9         System.out.println(context.getState().toString());
10
11        StopState stopState = new StopState();
12        stopState.doAction(context);
13
14        System.out.println(context.getState().toString());
15    }
16 }
```

Execution

Problems Javadoc Declaration Console

<terminated> StatePatternDemo [Java Application] C:\Program Files\Java\

Player is in start state

Start State

Player is in stop state

Stop State

Strategy_pattern

Dans le modèle Stratégie, un comportement de classe ou son algorithme peut être modifié au moment de l'exécution. Ce type de modèle de conception est soumis à un modèle de comportement.

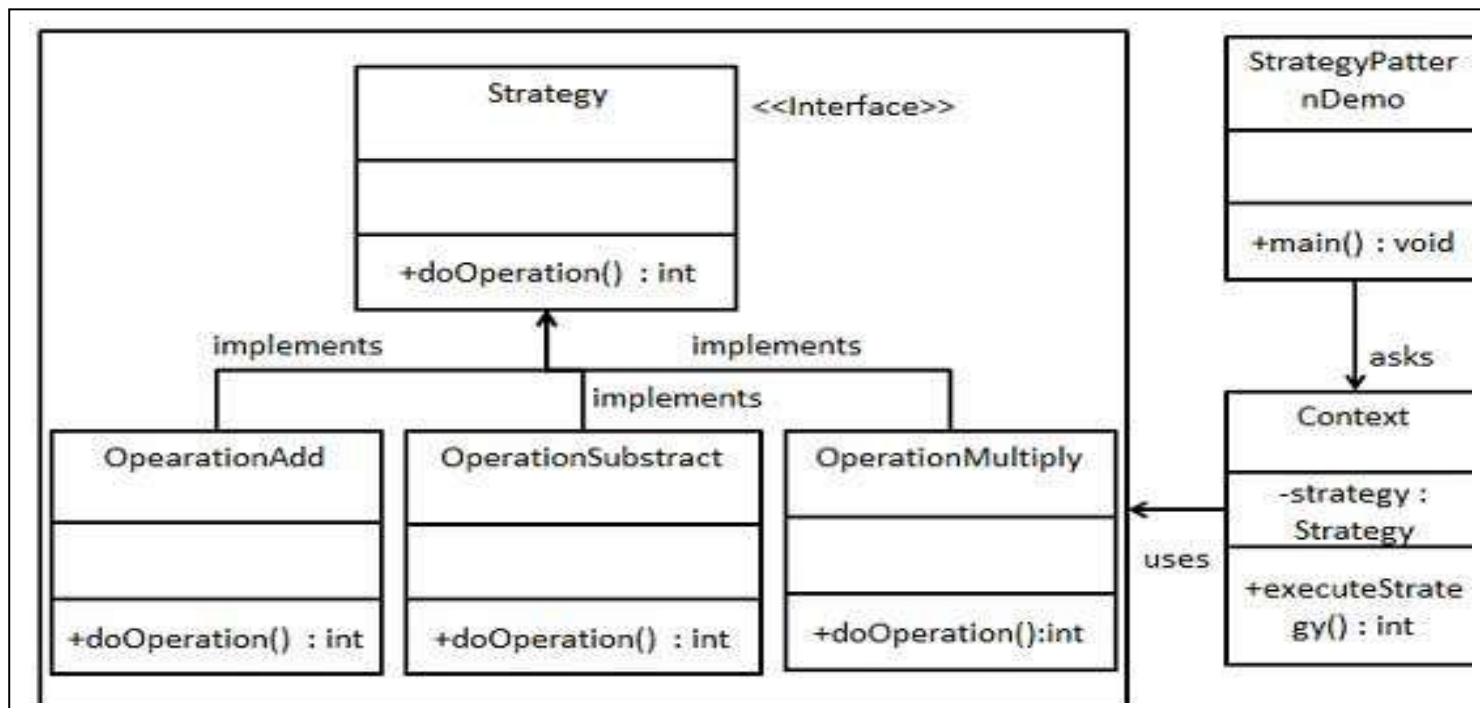
Dans le modèle Stratégie, nous créons des objets qui représentent différentes stratégies et un objet contexte dont le comportement varie selon son objet stratégie. L'objet de stratégie modifie l'algorithme d'exécution de l'objet de contexte.

implementation

Nous allons créer une interface Stratégie définissant une action et des classes de stratégies concrètes implémentant l'interface Stratégie. Le contexte est une classe qui utilise une stratégie.

StrategyPatternDemo, notre classe de démonstration, utilisera les objets Contexte et Stratégie pour démontrer les changements de comportement du Contexte en fonction de la stratégie déployée ou utilisée.

Diagramme de classe



Classestrategy

```
Strategy.java  OperationAdd.java  OperationSubtract.java  Context.java
1 package strategiePattern;
2
3 public interface Strategy {
4     public int doOperation(int num1, int num2);
5 }
6
```

ClasseOperationAdd

```
OperationAdd.java  OperationSubtract.java  Context.java  StrategyPatternDemo.java
1 package strategiePattern;
2
3 public class OperationAdd implements Strategy{
4     @Override
5     public int doOperation(int num1, int num2) {
6         return num1 + num2;
7     }
8 }
9
```

Classeoperationsubstract

```
OperationSubstract.java Context.java StrategyPatternDemo.java
1 package strategiePattern;
2
3 public class OperationSubstract implements Strategy{
4     @Override
5     public int doOperation(int num1, int num2) {
6         return num1 - num2;
7     }
8 }
9
```

ClasseOperationMultiply

```
Strategy.java OperationAdd.java OperationSubstract.java OperationMultiply.java
1 package strategiePattern;
2
3 public class OperationMultiply implements Strategy{
4     @Override
5     public int doOperation(int num1, int num2) {
6         return num1 * num2;
7     }
8 }
9
```

ClasseContext

```
Context.java StrategyPatternDemo.java
1 package strategiePattern;
2
3 public class Context {
4     private Strategy strategy;
5
6     public Context(Strategy strategy){
7         this.strategy = strategy;
8     }
9
10    public int executeStrategy(int num1, int num2){
11        return strategy.doOperation(num1, num2);
12    }
13 }
14
```


ClassStrategyPatternDemo

```
StrategyPatternDemo.java X
1 package strategiePattern;
2
3 public class StrategyPatternDemo {
4     public static void main(String[] args) {
5         Context context = new Context(new OperationAdd());
6         System.out.println("10 + 5 = " + context.executeStrategy(10, 5));
7
8         context = new Context(new OperationSubstract());
9         System.out.println("10 - 5 = " + context.executeStrategy(10, 5));
10
11        context = new Context(new OperationMultiply());
12        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
13    }
14 }
15
```

Execution

Problems @ Javadoc Declaration Console X

<terminated> StrategyPatternDemo [Java Application] C:\Program Files\Java\jre1.8.0_13

10 + 5 = 15

10 - 5 = 5

10 * 5 = 50

|

ITERATOR

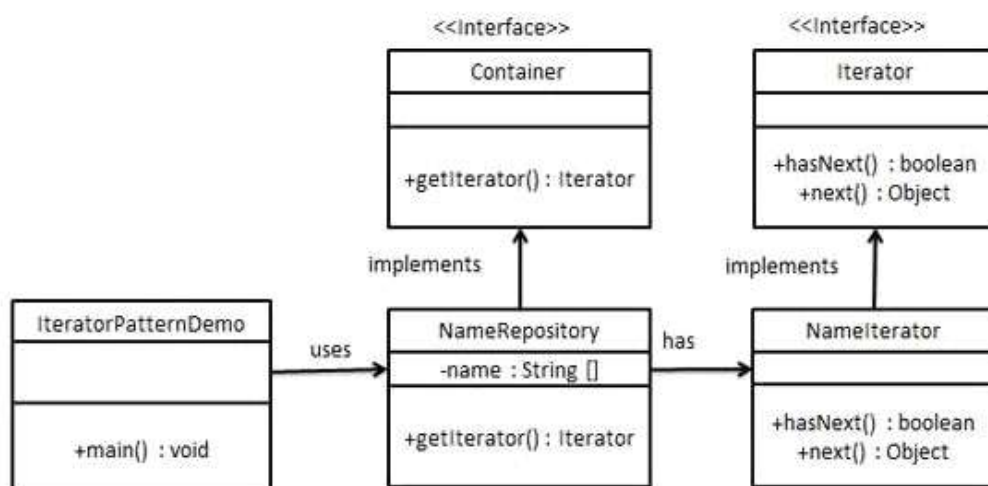
Le modèle Iterator est un mécanisme de conception très utilisé dans Java et dans l'environnement de programmation .Net. Ce modèle est utilisé pour obtenir un moyen d'accéder aux éléments d'un objet de collection de manière séquentielle sans avoir besoin de connaître sa représentation sous-jacente.

Le modèle d'itérateur tombe sous la catégorie de modèle comportemental.

Implementation

Nous allons créer une interface Iterator qui décrit la méthode de navigation et une interface Container qui retrace l'itérateur. Les classes concrètes implémentant l'interface Container seront responsables de l'implémentation de l'interface Iterator et de l'utiliser

IteratorPatternDemo, notre classe de démonstration utilisera NamesRepository, une implémentation de classe concrète pour imprimer un nom stocké en tant que collection dans NamesRepository.



Etape1 :

Creer les interfaces *Iterator.java* et *Container.java*

```
public interface Iterator {  
    public boolean hasNext();  
    public Object next();  
}
```

```
public interface Container {  
    public Iterator getIterator();  
}
```

Etape2 :

Créez une classe concrète implémentant l'interface Container. Cette classe a une classe interne NameIterator implémentant l'interface Iterator.

NameRepository.java

```
public class NameRepository implements Container {  
    public String names[] = {"Robert" , "John" , "Julie" , "Lora"};  
  
    @Override  
    public Iterator getIterator() {  
        return new NameIterator();  
    }  
  
    private class NameIterator implements Iterator {  
  
        int index;  
  
        @Override  
        public boolean hasNext() {  
  
            if(index < names.length){  
                return true;  
            }  
            return false;  
        }  
  
        @Override  
        public Object next() {  
  
            if(this.hasNext()){  
                return names[index++];  
            }  
            return null;  
        }  
    }  
}
```

Etape3 :

Utilisez le NameRepository pour obtenir l'itérateur et imprimer les noms.

IteratorPatternDemo.java


```

public class IteratorPatternDemo {

    public static void main(String[] args) {
        NameRepository namesRepository = new NameRepository();

        for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
            String name = (String)iter.next();
            System.out.println("Name : " + name);
        }
    }
}

```

Etape4 :

Vérifiez la sortie.

```

Name : Robert
Name : John
Name : Julie
Name : Lora

```

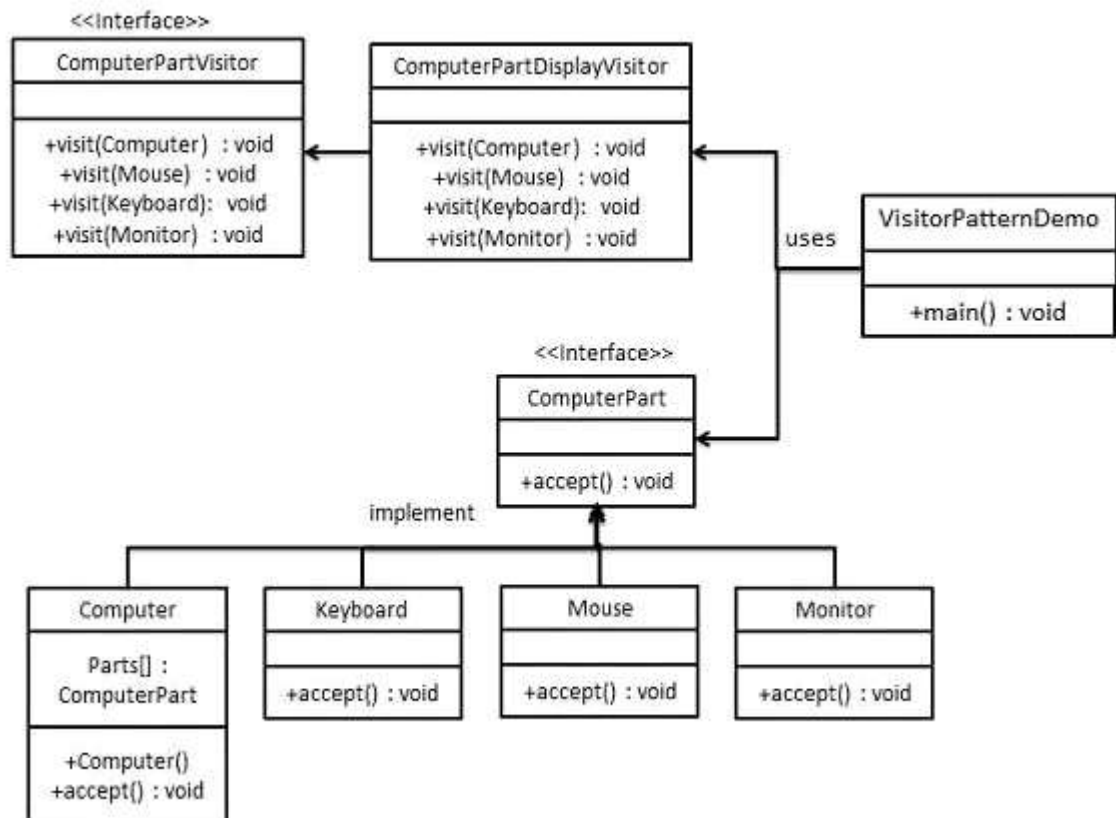
Visitor Pattern

Dans le modèle Visitor, nous utilisons une classe de visiteurs qui modifie l'algorithme d'exécution d'une classe d'éléments. De cette façon, l'algorithme d'exécution de l'élément peut varier au fur et à mesure que le visiteur varie. Ce modèle relève de la catégorie de modèle de comportement. Conformément au modèle, l'objet élément doit accepter l'objet visiteur afin que l'objet visiteur gère l'opération sur l'objet élément.

Implementation

Nous allons créer une interface ComputerPart définissant l'acceptation de l'opération. Le clavier, la souris, le moniteur et l'ordinateur sont des classes concrètes implémentant l'interface ComputerPart. Nous allons définir une autre interface ComputerPartVisitor qui va définir une opération de classe visiteur. L'ordinateur utilise le visiteur concret pour faire l'action correspondante.

VisitorPatternDemo, notre classe de démonstration, utilisera les classes Computer et ComputerPartVisitor pour démontrer l'utilisation du modèle de visiteur.



Etape1 :

Définir une interface pour représenter l'élément.

ComputerPart.java

```

public interface ComputerPart {
    public void accept(ComputerPartVisitor computerPartVisitor);
}
  
```

Etape2 :

Créer des classes concrètes étendant la classe ci-dessus.

Keyboard.java

```

public class Keyboard implements ComputerPart {

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}
  
```

Monitor.java

```

public class Monitor implements ComputerPart {

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}

```

Mouse.java

```

public class Mouse implements ComputerPart {

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}

```

Computer.java

```

public class Computer implements ComputerPart {

    ComputerPart[] parts;

    public Computer(){
        parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};
    }

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        for (int i = 0; i < parts.length; i++) {
            parts[i].accept(computerPartVisitor);
        }
        computerPartVisitor.visit(this);
    }
}

```

Etape3 :

Définir une interface pour représenter le visiteur

ComputerPartVisitor.java

```

public interface ComputerPartVisitor {
    public void visit(Computer computer);
    public void visit(Mouse mouse);
    public void visit(Keyboard keyboard);
    public void visit(Monitor monitor);
}

```

```

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

    @Override
    public void visit(Computer computer) {
        System.out.println("Displaying Computer.");
    }

    @Override
    public void visit(Mouse mouse) {
        System.out.println("Displaying Mouse.");
    }

    @Override
    public void visit(Keyboard keyboard) {
        System.out.println("Displaying Keyboard.");
    }

    @Override
    public void visit(Monitor monitor) {
        System.out.println("Displaying Monitor.");
    }
}

```

```

public class VisitorPatternDemo {
    public static void main(String[] args) {

        ComputerPart computer = new Computer();
        computer.accept(new ComputerPartDisplayVisitor());
    }
}

```

```

Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.

```

Decorator

Le motif décorateur permet à un utilisateur d'ajouter de nouvelles fonctionnalités à un objet existant sans modifier sa structure. Ce type de motif de conception est structuré car ce motif agit comme une enveloppe à la classe existante.

Ce modèle crée une classe de décorateur qui enveloppe la classe d'origine et fournit des fonctionnalités supplémentaires préservant la signature des méthodes de classe.

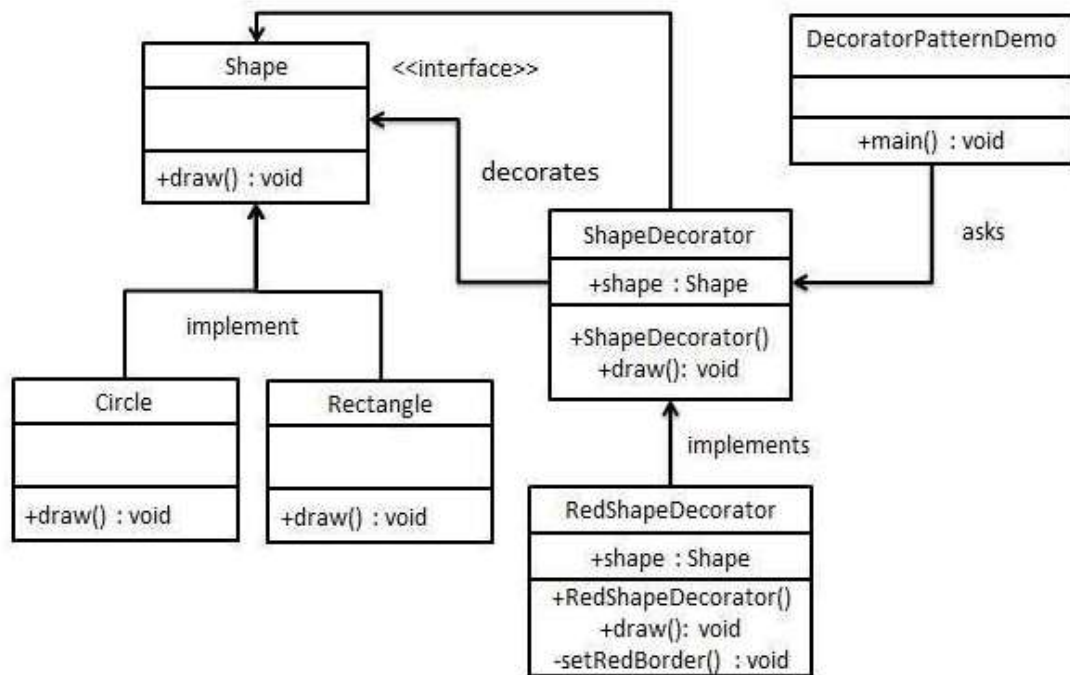
Nous démontrons l'utilisation du modèle décorateur par l'exemple suivant dans lequel nous allons décorer une forme avec une certaine couleur sans altérer la classe de forme.

Implementation

Nous allons créer une interface Shape et des classes concrètes implémentant l'interface Shape. Nous allons ensuite créer une classe décorative abstraite ShapeDecorator implémentant l'interface Shape et ayant l'objet Shape comme variable d'instance.

RedShapeDecorator est une classe concrète implémentant ShapeDecorator.

DecoratorPatternDemo, notre classe de démonstration utilisera RedShapeDecorator pour décorer les objets Shape.



```

public interface Shape {
    void draw();
}

```

```

public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Shape: Rectangle");
    }
}

```

```

public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Shape: Circle");
    }
}

```

```
public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape){
        this.decoratedShape = decoratedShape;
    }

    public void draw(){
        decoratedShape.draw();
    }
}
```

```
public class RedShapeDecorator extends ShapeDecorator {

    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape){
        System.out.println("Border Color: Red");
    }
}
```

```
public class DecoratorPatternDemo {
    public static void main(String[] args) {

        Shape circle = new Circle();

        Shape redCircle = new RedShapeDecorator(new Circle());

        Shape redRectangle = new RedShapeDecorator(new Rectangle());
        System.out.println("Circle with normal border");
        circle.draw();

        System.out.println("\nCircle of red border");
        redCircle.draw();

        System.out.println("\nRectangle of red border");
        redRectangle.draw();
    }
}
```

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

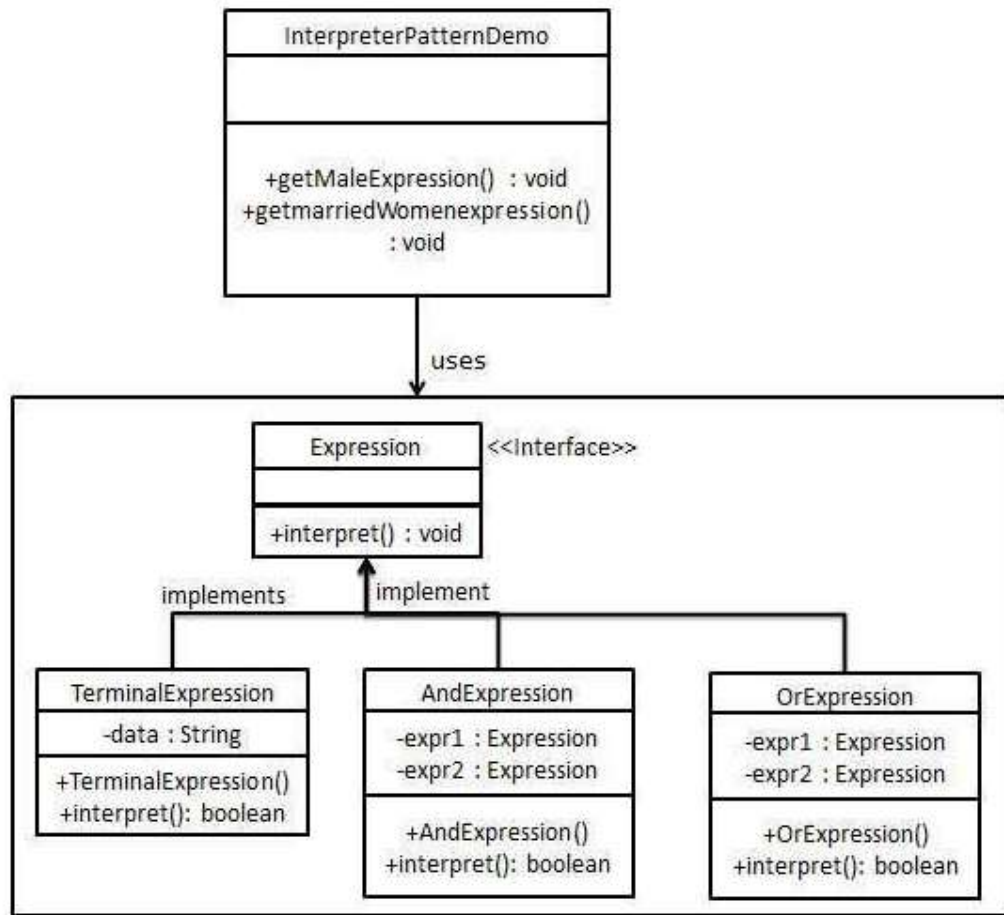
Interpreter

Le modèle d'interprète fournit un moyen d'évaluer la grammaire ou l'expression du langage. Ce type de modèle relève du modèle comportemental. Ce modèle implique la mise en œuvre d'une interface d'expression qui dit d'interpréter un contexte particulier. Ce modèle est utilisé dans l'analyse SQL, le moteur de traitement de symboles, etc.

Implementation

Nous allons créer une interface `Expression` et des classes concrètes implémentant l'interface `Expression`. Une classe `TerminalExpression` est définie qui agit comme un interpréteur principal du contexte en question. Les autres classes `OrExpression`, `AndExpression` sont utilisées pour créer des expressions combinatoires.

`InterpreterPatternDemo`, notre classe de démonstration, utilisera la classe `Expression` pour créer des règles et démontrer l'analyse des expressions.



```

public interface Expression {
    public boolean interpret(String context);
}

```

```

public class TerminalExpression implements Expression {

    private String data;

    public TerminalExpression(String data){
        this.data = data;
    }

    @Override
    public boolean interpret(String context) {

        if(context.contains(data)){
            return true;
        }
        return false;
    }

}

```

```

public class OrExpression implements Expression {

    private Expression expr1 = null;
    private Expression expr2 = null;

    public OrExpression(Expression expr1, Expression expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }

    @Override
    public boolean interpret(String context) {
        return expr1.interpret(context) || expr2.interpret(context);
    }
}

```

```

public class AndExpression implements Expression {

    private Expression expr1 = null;
    private Expression expr2 = null;

    public AndExpression(Expression expr1, Expression expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }

    @Override
    public boolean interpret(String context) {
        return expr1.interpret(context) && expr2.interpret(context);
    }
}

```

```

public class InterpreterPatternDemo {

    //Rule: Robert and John are male
    public static Expression getMaleExpression(){
        Expression robert = new TerminalExpression("Robert");
        Expression john = new TerminalExpression("John");
        return new OrExpression(robert, john);
    }

    //Rule: Julie is a married women
    public static Expression getMarriedWomanExpression(){
        Expression julie = new TerminalExpression("Julie");
        Expression married = new TerminalExpression("Married");
        return new AndExpression(julie, married);
    }

    public static void main(String[] args) {
        Expression isMale = getMaleExpression();
        Expression isMarriedWoman = getMarriedWomanExpression();

        System.out.println("John is male? " + isMale.interpret("John"));
        System.out.println("Julie is a married women? " + isMarriedWoman.interpret("Ma
    }
}

```

```
John is male? true  
Julie is a married women? true
```

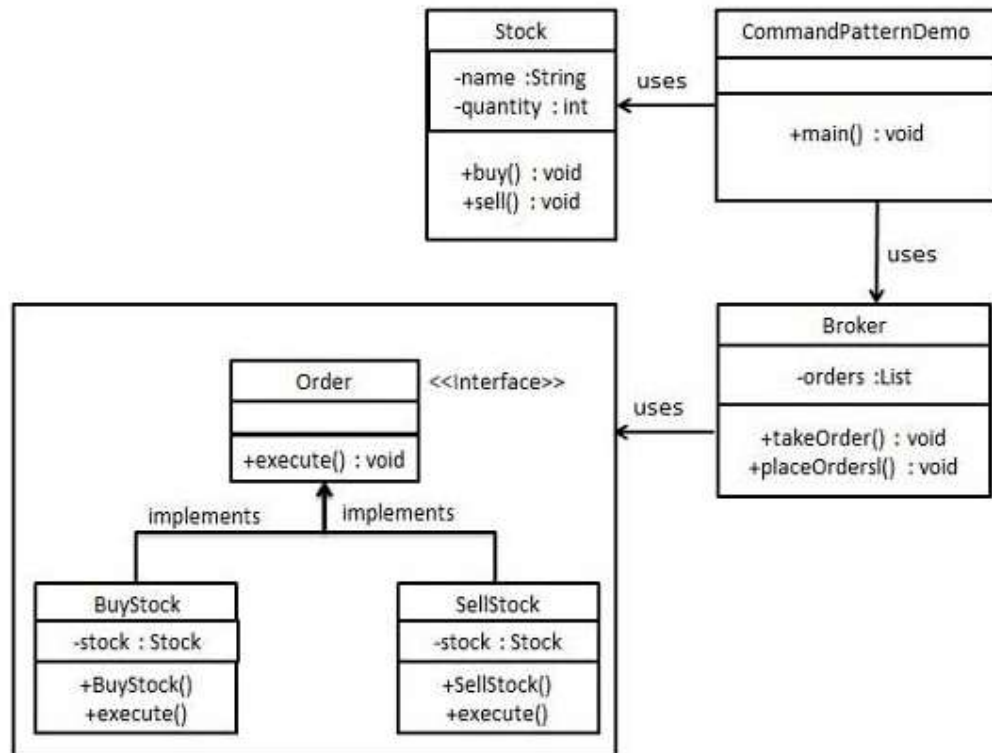
Command

Le modèle de commande est un modèle de conception piloté par les données et relève de la catégorie de modèle comportemental. Une requête est enveloppée sous un objet en tant que commande et transmise à l'objet invocateur. L'objet Invoker recherche l'objet approprié qui peut gérer cette commande et passe la commande à l'objet correspondant qui exécute la commande.

Implementation

Nous avons créé une interface Commande qui agit comme une commande. Nous avons créé une classe Stock qui agit comme une demande. Nous avons des classes de commande concrètes BuyStock et SellStock implémentant l'interface d'ordre qui effectuera le traitement de commande réel. Un courtier de classe est créé qui agit comme un objet invocateur. Il peut prendre et passer des commandes.

L'objet Broker utilise un modèle de commande pour identifier quel objet exécutera quelle commande en fonction du type de commande. CommandPatternDemo, notre classe de démonstration, utilisera la classe Broker pour démontrer le modèle de commande.



```

public interface Order {
    void execute();
}

```

```

public class Stock {

    private String name = "ABC";
    private int quantity = 10;

    public void buy(){
        System.out.println("Stock [ Name: "+name+",
            Quantity: " + quantity +" ] bought");
    }
    public void sell(){
        System.out.println("Stock [ Name: "+name+",
            Quantity: " + quantity +" ] sold");
    }
}

```

```

public class BuyStock implements Order {
    private Stock abcStock;

    public BuyStock(Stock abcStock){
        this.abcStock = abcStock;
    }

    public void execute() {
        abcStock.buy();
    }
}

```

```

public class SellStock implements Order {
    private Stock abcStock;

    public SellStock(Stock abcStock){
        this.abcStock = abcStock;
    }

    public void execute() {
        abcStock.sell();
    }
}

```

```

import java.util.ArrayList;
import java.util.List;

public class Broker {
    private List<Order> orderList = new ArrayList<Order>();

    public void takeOrder(Order order){
        orderList.add(order);
    }

    public void placeOrders(){
        for (Order order : orderList) {
            order.execute();
        }
        orderList.clear();
    }
}

```

```

public class CommandPatternDemo {
    public static void main(String[] args) {
        Stock abcStock = new Stock();

        BuyStock buyStockOrder = new BuyStock(abcStock);
        SellStock sellStockOrder = new SellStock(abcStock);

        Broker broker = new Broker();
        broker.takeOrder(buyStockOrder);
        broker.takeOrder(sellStockOrder);

        broker.placeOrders();
    }
}

```

```

Stock [ Name: ABC, Quantity: 10 ] bought
Stock [ Name: ABC, Quantity: 10 ] sold

```