Université Sidi Mohamed Ben Abdellah
Faculté des Sciences Dhar El-Mehraz Fès
Département d'Informatique

Design Patterns en Java

Modèles de conception réutilisables

Par Noureddine Chenfour

2005 – 2020

Table des Matières

Chapitre 1. Introduction générale aux modèles de conception Orientés Objet

1.1 Rappel sur les aspects fondamentaux de la POO et relations entre classes

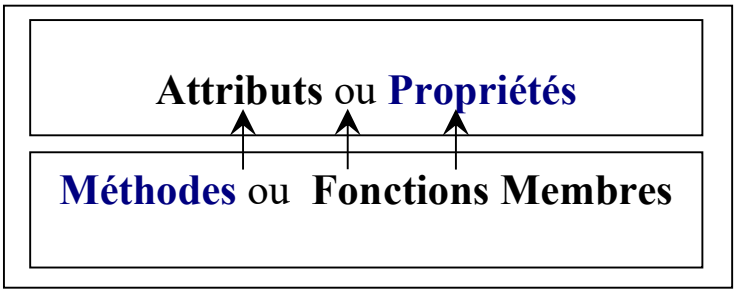
Une structuration classique d'un programme consiste en une structuration à deux niveaux : les données d'une part et le code d'une autre part. Ainsi les données qui décrivent ou caractérisent une même entité sont regroupées ensemble dans une même structure de donnée : un enregistrement ou un tableau. De la même manière des instructions réalisant ensemble une tache bien définie et complète sont regroupées dans une même procédure ou fonction.

La Programmation Orientée Objets (**POO**) consiste en une structuration de plus haut niveau. Il s'agit de regrouper ensemble les données et toutes les procédures et fonctions qui permettent la gestion de ces données. On obtient alors des entités comportant à la fois un ensemble de données et une liste de procédures et de fonctions pour manipuler ces données. La structure ainsi obtenue est appelée : **Objet**.

Un objet est alors une généralisation de la notion d'enregistrement. Il est composé de deux parties :

- Une partie statique (ou fixe), généralement privée (private) composée de la liste des données de l'objet. On les appelle : **Attributs** ou **Propriétés**, ou encore : **Données Membres**.

- Une partie dynamique, généralement publique (public) qui décrit le comportement ou les fonctionnalités de l'objet. Elle est constituée de l'ensemble des procédures et des fonctions qui permettent à l'utilisateur de configurer et de manipuler l'objet. Ainsi les données ne sont généralement pas accessibles directement mais à travers les procédures et les fonctions de l'objet. Celles-ci sont appelées : **Méthodes** ou **Fonctions Membres**.



Relations entre classes :

Les classes peuvent être reliées par l'intermédiaire des relations suivantes :

1. **Héritage**
2. **Agrégation** ou **composition**
3. **l'Implémentation** qui est une autre forme d'héritage définie entre une interface et une classe.

1.2 Interfaces et classes Abstraites

Il existe 3 principaux types de structures qu'on peut manipuler au sein d'un langage à objets :

1. Les **classes**

2. Les **classes abstraites** présentées en Java par le modificateur « **abstract** » :

```
abstract class NomClasse {
   ...
}
```

Dans une classe abstraite, le corps de quelques méthodes peut ne pas être défini (on déclare uniquement le prototype de la méthode). Ces méthodes sont dites des méthodes abstraites. Une méthode abstraite est aussi présentées par l'intermédiaire du modificateur « **abstract** » de la manière ci-après. C'est aux classes dérivées de redéfinir ces méthodes et de préciser leur comportement.

```
abstract class NomClasse {
   abstract type nomMéthode(...);
   ...
}
```

Une classe abstraite ne peut donc jamais être instanciée. Il s'agit d'une spécification devant être implémentée par l'intermédiaire d'une classe dérivée. Si cette dernière définit toutes les méthodes abstraites alors celle-ci est instanciable.

Remarque :

Une classe abstraite (présentée par le modificateur « **abstract** ») peut ne pas contenir de méthodes abstraites. Cependant, une classe contenant une méthode abstraite doit obligatoirement être déclarée « **abstract** ».

3. Les **interfaces** qui sont définies en Java par l'intermédiaire du mot clé **interface** au lieu de **class** constituent un cas particulier des classes abstraites : d'une part, ce sont des classes sans aucune structure, donc pas de propriétés (simplement des constantes si on en a besoin). D'autre part, aucune méthode n'y est définie; uniquement les prototypes de méthodes. Il faut noter que tous les membres d'une interface sont automatiquement publiques. A remarquer aussi qu'une classe dérivée d'une interface fournie une **implémentation** de celle-ci et le lien d'héritage est un héritage par implémentation réalisé par l'intermédiaire du mot clé **implements** :

- *Définition d'une interface :*

```
interface NomInterface {
    type1 methode1(...);
    type2 methode2(...);
    …
}
```

- *Implémentation d'une interface :*

```
class NomDeClasse implements NomInterface {
  …
}
```

Remarques :

- Une classe qui implémente une interface doit définir toutes les méthodes de l'interface, sinon elle doit être définie « **abstract** ».
- les attributs (ou propriétés) d'une interface sont obligatoirement des constantes. Ils sont donc par défaut **public**, **static** et **final** et nécessite donc une initialisation.
- Une interface est une structure de spécification, d'abstraction et de publication de services.

1.3 Principe de base des Design patterns

Un design pattern (ou modèle de conception) est une solution optimale robuste et réutilisable ; pouvant être associée à un problème ou un genre de problème que le développeur rencontre fréquemment lors de la réalisation des applications. Le design pattern est donc une structure de classe bien déterminée qu'il serait bénéfique d'utiliser une fois le problème détecté. La notion de design pattern est un moyen efficace permettant d'automatiser le processus de recherche de solution pour des problèmes bien connus. La conséquence d'utilisation du design pattern serait un gain considérable par rapport à un développement classique « bricolé ». Les éléments de base permettant de définir un design pattern sont :

- Nom du design pattern (ou du modèle)
- Problème traité par le modèle
- Solution
- Conséquences d'application du modèle

Cependant si on veut être plus précis et plus profond lorsqu'n est entrain de définir un « Design Pattern », nous pouvons recenser 14 caractéristiques au total. Ainsi, par exemple, il ne serait pas suffisant dans certains cas de dire :

> Voici le genre de problème qu'on peut traiter avec un certain design pattern.

On serait plus précis on disant par exemple :

> Si on a un problème nécessitant plusieurs aspects de programmation (aspect stockage, un aspect métier assez compliqué et un aspect de présentation), et que notre « **intention** » est de réaliser séparément les différents modules ou aspects ; avec comme « **motivation** » la possibilité de faire évoluer et tester séparément les différents modules sans avoir d'influences les uns sur les autres ; alors le meilleur design pattern à utiliser serait le « Bridge ». En plus, nous avons comme « **conséquence d'utilisation** » de ce design pattern l'extensibilité à l'infinie des différents modules pris séparément.

1.4 Caractéristiques d'un Design Pattern

1. Nom du design pattern
2. Alias
3. classification
4. Indication d'utilisation
5. Intention
6. Motivation
7. Conséquences
8. Structure
9. Constituants
10. Collaboration
11. Implémentation
12. Exemple de code
13. Utilisations remarquables
14. Modèle (ou design patterns) apparentés

1.5 Vue générale sur le catalogue des GOF

La classification est l'une des caractéristiques importantes des design patterns. On peut ainsi découper le catalogue des « GOF » en 3 classes selon la nature du problème à résoudre.

1- Creational Patterns : Design Patterns de Création

Nom du Pattern	En Français
1. Singleton	Singleton
2. Prototype	Prototype
3. Factory Method	Fabrique
4. Abstract Factory	Fabrique abstraite
5. Builder	Monteur

2- Architectural/Structural : design Patterns de structure

Nom du Pattern	En Français
6. Adapter	Adaptateur
7. Bridge	Pont
8. Composite	Composite
9. Decorator	Décorateur
10. Façade	Façade
11. Flyweight	Poids Mouche
12. Proxy	Procuration

3- Behavioral : Design Patterns de Comportement

Nom du Pattern	En Français
13. Chain of Responsibility	Chaine de Responsabilité
14. Command	Commande
15. Interpreter	Interpréteur
16. Iterator	Itérateur
17. Mediator	Médiateur
18. Memento	Mémento
19. Observer	Observateur
20. State	Etat
21. Strategy	Stratégie
22. Template	Patron de Méthode
23. Visitor	Visiteur

Chapitre 2. Design Patterns de création

2.1 Singleton

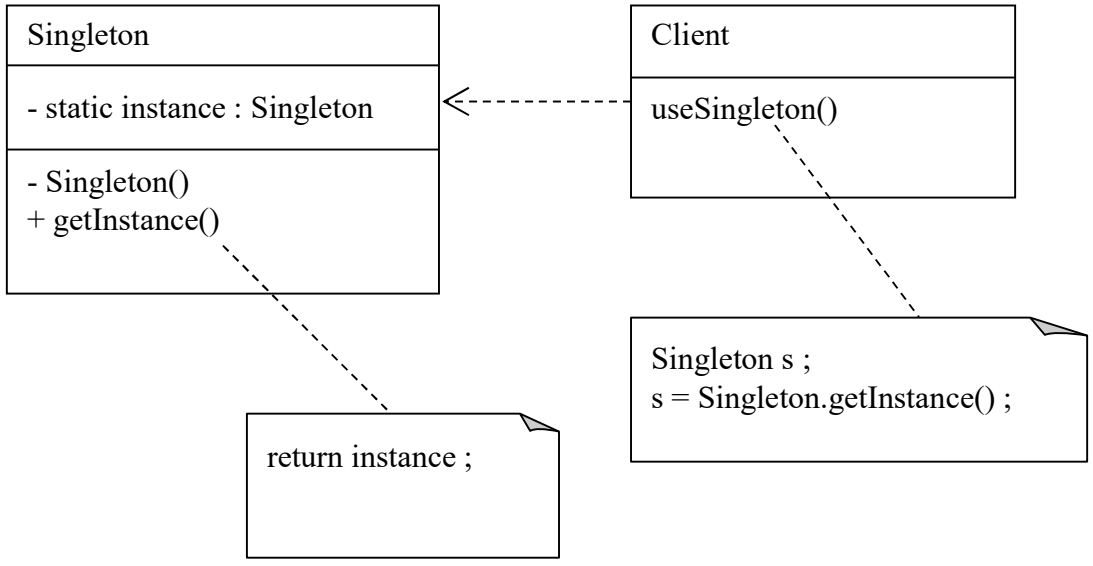
Le Singleton est un modèle de conception permettant de résoudre le Problème : Limiter l'instanciation des objets à un nombre réduit (une seule Fois : GOF). L'utilisateur du singleton n'aura pas le droit de créer plus qu'une seule instance issue d'une classe donnée nommée : Singleton. S'il le demande, il aura toujours la même instance créée préalablement dans le Singleton.

Solution Java :

- Constructeur Privé ➔ pas de possibilité d'instanciation à l'extérieur de la classe.
- Une instance « static » définie et créée à l'intérieur du Singleton.
- Une méthode « static » donnant accès à cette instance.

Remarque :

En java, la classe **Toolkit** est un singleton. Une seule instance peut être créée et c'est à l'aide de la méthode « static » **getDefaultToolkit()** de la classe Toolkit.



Implémentation :

```
public class Singleton {
     private static Singleton instance = new Singleton();

     private Singleton() {
     }

     public static Singleton getInstance() {
          return instance;
     }

}
```

```
public class Client {

     public Client() {
          useSingleton();
     }

     public void useSingleton() {
          //Singleton s1 = new Singleton(); impossible
          //Singleton s2 = new Singleton(); impossible
          Singleton s1 = Singleton.getInstance();
          Singleton s2 = Singleton.getInstance();
          System.out.println(s1);
          System.out.println(s2);
     }

     public static void main(String[] args) {
          new Client();
     }

}
```

2.2 Prototype

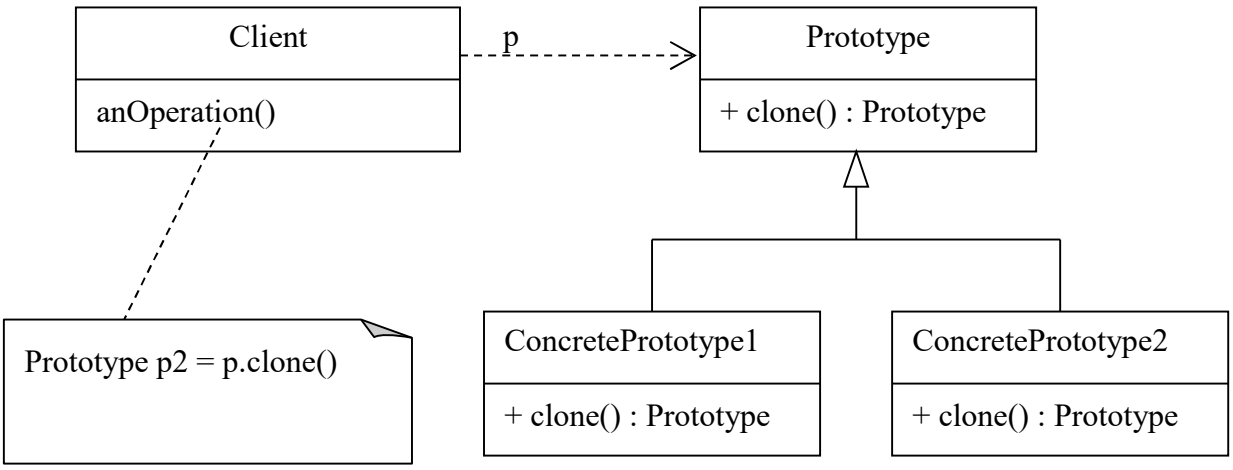
Le Design Pattern Prototype est une architecture de classes permettant le clonage des objets indépendamment de leur nature. Ainsi, dans certaines situations la création d'un objet peut être très coûteuse (prend beaucoup de temps, par exemple l'ouverture d'un objet Database qui nécessite le chargement du driver, la connexion, la création d'un objet DatabaseMetaData, etc…). Le clonage s'avère alors une solution plus rapide.

Remarque :

Le prototypage (clonage) est aussi demandé si on a besoin de faire un passage par valeur (dans tel cas, il suffit de cloner le paramètre et de travailler sur la copie).

Solution Java :

- C'est une classe concrète qui implémente l'interface **Cloneable**.
- Elle hérite de la méthode **clone()** définie « protected » dans la classe **Object**.
- Redéfinir « public » la méthode clone().



Implémentation :

```
public class Prototype implements Cloneable {
     public Prototype clone() {
          Prototype clone = null;
          try {
               clone = (Prototype)super.clone();
          } catch (CloneNotSupportedException e) {
               e.printStackTrace();
          }
          return clone;
     }
}
```

```
public class CP1 extends Prototype {
     private int x, y;
     public CP1() {
          x = 0; y = 0;
     }
     public CP1(int x, int y) {
          this.x = x;
          this.y = y;
     }
     // getters & setters
     // ...

     public CP1 clone() {
          return (CP1)super.clone();
     }
}
```

```
package dp.creational.prototype;

public class CP2 extends Prototype {
     private String name;

     public CP2() {
          name = "";
     }
     // getters & setters
     // ...

     public CP2(String name) {
          super();
          this.name = name;
     }
}
```

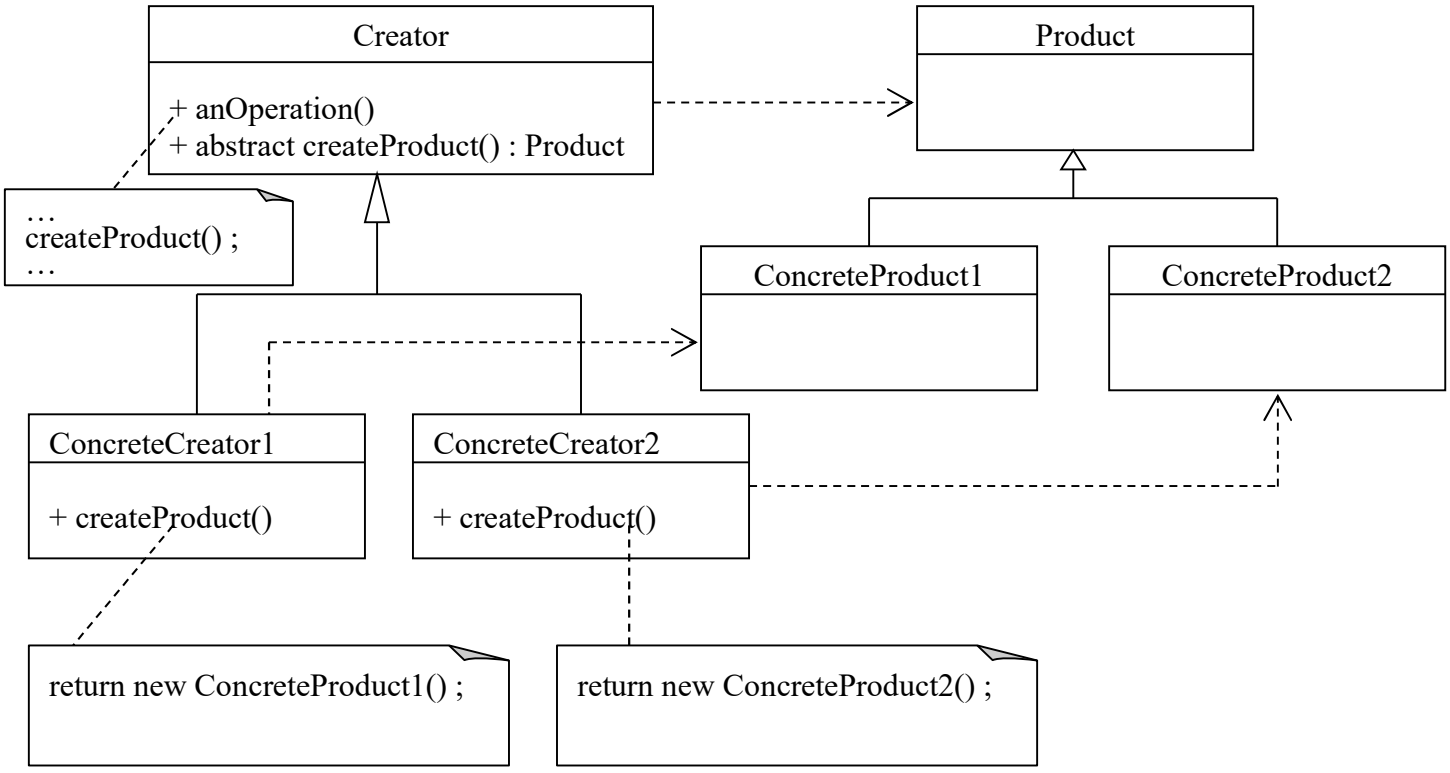
```
public class Client {

     public Client() {
          creation1();
     }

     void creation1() {
          CP1 p1 = new CP1(20, 30);
          Prototype q1 = new CP2("Prototype");

          CP1 p2 = p1.clone();
          Prototype q2 = q1.clone();

          System.out.println("&p1=" + p1 + ", p2=" + p2);
          System.out.println("p1.x=" + p1.getX() + ", p2="
                              + p2.getX());

          System.out.println("&q1=" + q1 + ", q2=" + q2);
          System.out.println("q1.name=" + ((CP2)q1).getName()
                         + ", q2=" + ((CP2)q2).getName());
     }

     void creation2() {
          Prototype tp1[] = {
                    new CP1(20, 30),
                    new CP2("Prototype")
          };
          Prototype tp2[] = new Prototype[tp1.length];

          for (int i = 0; i < tp2.length; i++) {
               tp2[i] = tp1[i].clone();
          }
     }


     public static void main(String[] args) {
          new Client();
     }

}
```

2.3 La Fabrique : Factory Method

Le Design Pattern Factory Method est une architecture de classes permettant la création d'objets complexes à partir d'objets élémentaires bien définis (par exemple, un panneau de composants). Les objets élémentaires ou les composants **(qui sont de même type)** peuvent être instances de n'importe quelle classe (des boutons, des cases à cocher, des boutons radio, etc…). La décision de la nature de la classe sera déléguée à des sous-classes qui auront la tâche de choisir les classes à instancier.

Solution :

- Utiliser une classe abstraite mère qui fait tout le travail en utilisant un objet créé par l'une de ses méthodes.
- Celle-ci est une méthode abstraite qui sera la seule méthode à redéfinir dans des classes filles pour offrir à chaque redéfinition un objet d'une classe différente.



Implémentation : Modèle

```
public abstract class AbstractCreator {
     public void anOperation() {
          //...
          Product p1 = factoryMethod();
          System.out.println("creation à base de : " + p1);
          //...
          Product p2 = factoryMethod();
          System.out.println("creation à base de : " + p2);
          //...
     }

     public void otherOperations() {
          //...
     }
     //...

     abstract public Product factoryMethod();
}
```

```
public class ConcreteCreator1 extends AbstractCreator {
     public Product factoryMethod() {
          return new ConcreteProduct1();
     }
}
```

```
public class ConcreteCreator2 extends AbstractCreator {
     public Product factoryMethod() {
          return new ConcreteProduct2();
     }
}
```

```
public class Product {
}
```

```
public class ConcreteProduct1 extends Product {
     public String toString() {
          return "ConcreteProduct1";
     }
}
```

```
public class ConcreteProduct2 extends Product {
     public String toString() {
          return "ConcreteProduct2";
     }
}
```

```
public class Client {

     public Client() {
          ConcreteCreator1 cc1 = new ConcreteCreator1();
          creation(cc1);

          ConcreteCreator2 cc2 = new ConcreteCreator2();
          creation(cc2);
     }

     public void creation(AbstractCreator creator) {
          creator.anOperation();
     }

     public static void main(String[] args) {
          new Client();
     }
}
```

Exemple :

Realisation d'un panneau :

- de Boutons
- de cases à cocher
- de boutons radio

➔ Les classes suivantes :

AbstractButtonPanel : AbstractCreator

ButtonPanel : ConcreteCreator1

CheckBoxPanel : ConcreteCreator2

RadioButtonPanel : ConcreteCreator3

Client : Client

AbstractButton : Product

JButton : ConcreteProduct1

JCheckBox : ConcreteProduct2

JRadioButton : ConcreteProduct3

Exemple :

```
public abstract class AbstractButtonPanel extends JPanel {
     public void init(String labels[]) {
          for (int i =0; i<labels.length; i++) {
               AbstractButton b = createButton(labels[i]);
               add(b);
          }
     }

     abstract public AbstractButton createButton(String label);

     public void add(String label) {
          AbstractButton b = createButton(label);
          add(b);
     }
}
```

```
public class ButtonPanel extends AbstractButtonPanel {
     public ButtonPanel(String labels[]) {
          init(labels);
     }

     public AbstractButton createButton(String label) {
          return new JButton(label);
     }
}
```

```
public class CheckBoxPanel extends AbstractButtonPanel {
     public CheckBoxPanel(String labels[]) {
          init(labels);
     }

     public AbstractButton createButton(String label) {
          return new JCheckBox(label);
     }
}
```

```
public class RadioButtonPanel extends AbstractButtonPanel {
     public RadioButtonPanel(String labels[]) {
          init(labels);
     }

     public AbstractButton createButton(String label) {
          return new JRadioButton(label);
     }
}
```

```
class Client extends JFrame {
    Client() {
        exp02();
    }

    void changeLayout(AbstractButtonPanel abp) {
       abp.setLayout(new GridLayout(abp.getComponentCount(), 1));;
    }

    void exp01() {
        JPanel p = new JPanel();
        p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
        p.add(new ButtonPanel(new String[]{"Choix 1","Choix 2",
              "Choix 3"}));
        p.add(new RadioButtonPanel(new String[]{"Choix 1",
              "Choix 2","Choix 3"}));
        p.add(new CheckBoxPanel(new String[]{"Choix 1","Choix 2",
              "Choix 3"}));

        setContentPane(p);
        pack();
        setVisible(true);
    }

    void exp02() {
        JPanel p = new JPanel();
        p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
        AbstractButtonPanel abp[] = {
            new ButtonPanel(new String[]{"Choix 1","Choix 2",
                    "Choix 3"}),
            new RadioButtonPanel(new String[]{"Choix 1",
                    "Choix 2","Choix 3"}),
            new CheckBoxPanel(new String[]{"Choix 1","Choix 2",
                    "Choix 3"})
        };
        for (int i=0; i<abp.length; i++) p.add(abp[i]);
        abp[1].add("Choix 4");
        changeLayout(abp[1]);

        setContentPane(p);
        pack();
        setVisible(true);
    }

    public static void main(String args[]) {
        new Client();
    }
}
```

2.4 Fabrique abstraite : Abstract Factory

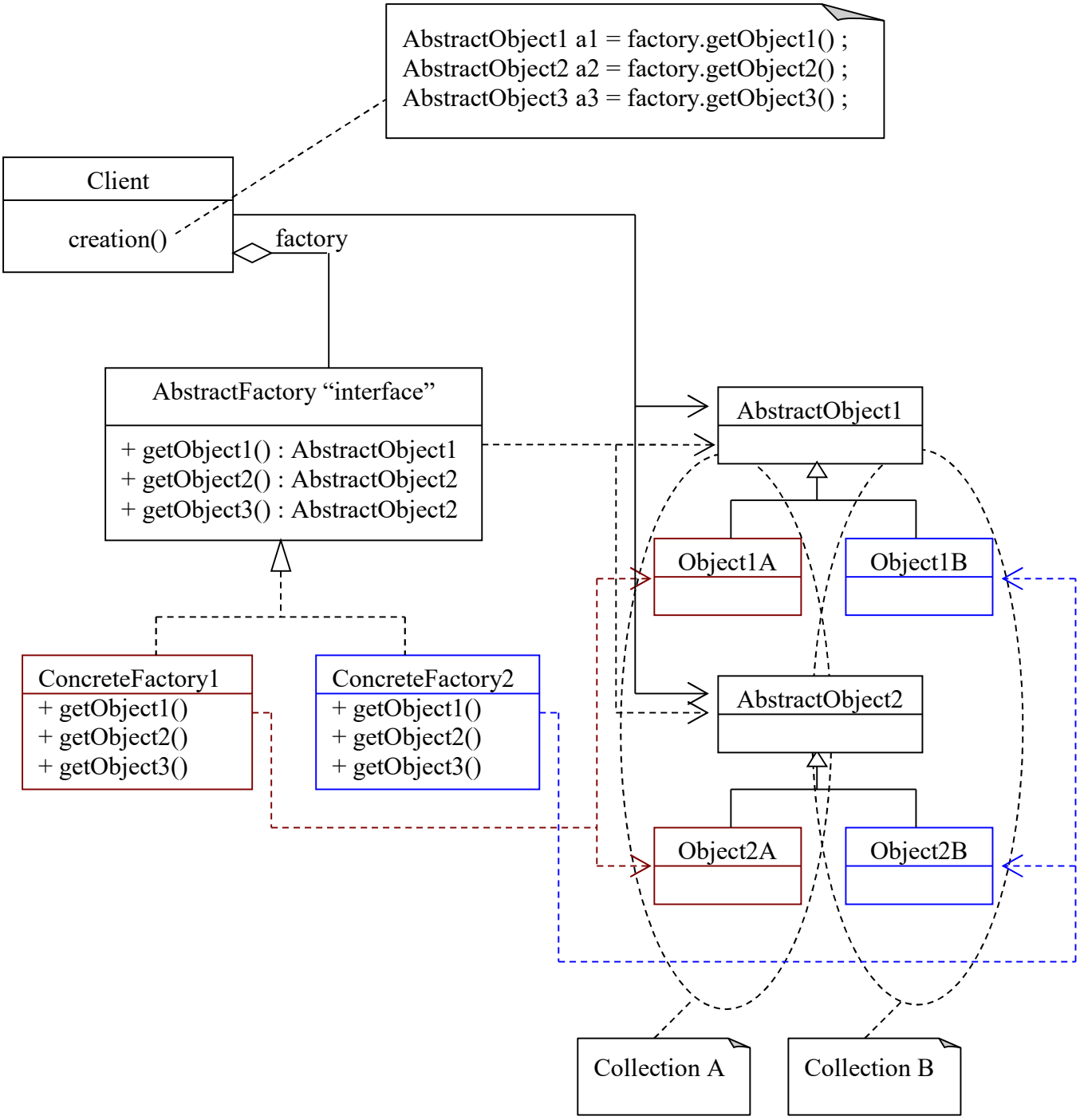
Le Design Pattern Abstract Factory permet de fournir une solution optimale pour la création de collections d'objets variées. Les collections vont contenir le même nombre de classes (d'objets) avec une correspondance entre les classes des collections. Ainsi au lieu de créer des objets à l'aide de l'opérateur new, on les créera à l'aide d'une fabrique :

- objet1 = fabrique.getObject1() ;
- objet2 = fabrique.getObject2() ;
- objet3 = fabrique.getObject3() ;

Si alors on veut changer de type des objets retournés, il suffira de changer de fabrique. La fabrique étant un paramètre du client de création; il suffira d'appeler la méthode de création à chaque fois avec une fabrique différente sans recompilation du client :

```
class Creator {
     AbstractFactory fabrique ;
     Creator(AbstractFactory fabrique) {
          this.fabrique = fabrique;
     }

     void build() {
          objet1 = fabrique.getObject1() ;
          objet2 = fabrique.getObject2() ;
          objet3 = fabrique.getObject3() ;
          ...
     }
}
```

```
Class client {
     public static void main(String args[]) {
          //choix de la factory
          AbstractFactory f = new ConcreteFactory1();

          Creator creator = new Creator(f);
          creator.build();
     }
}
```


AbstractObject1 a1 = factory.getObject1() ;
AbstractObject2 a2 = factory.getObject2() ;
AbstractObject3 a3 = factory.getObject3() ;
Client
creation()
factory
AbstractFactory “interface”
+ getObject1() : AbstractObject1
+ getObject2() : AbstractObject2
+ getObject3() : AbstractObject2
AbstractObject1
Object1A
Object1B
AbstractObject2
Object2A
Object2B
ConcreteFactory1
+ getObject1()
+ getObject2()
+ getObject3()
ConcreteFactory2
+ getObject1()
+ getObject2()
+ getObject3()
Collection A
Collection B

2.5 Le Monteur : Builder

Ce Design Pattern permet de décomposer un comportement complexe (dans un Director) en un ensemble de comportements élémentaires dans un objectif d'obtenir à la fin du traitement un objet : Montage d'un objet (appelé Product) sur plusieurs étapes:

- Step1() ;
- Step2() ;
- Step3() ;
- …

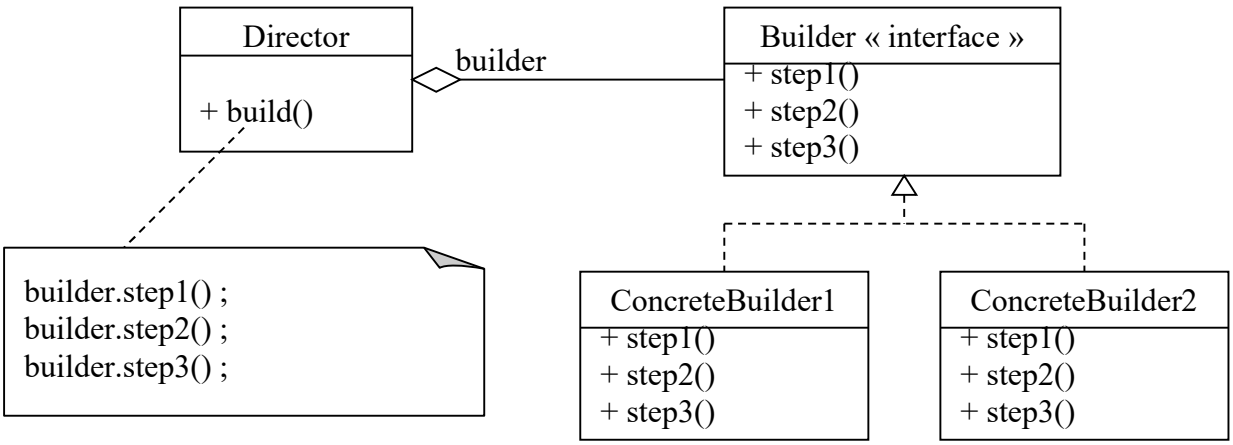
Ces comportements sont définis sous forme de méthodes externalisées, càd des méthodes d'une classe externe à la classe de construction (le Director), cette classe est appelée un Monteur (builder)

➔

- builder.step1() ;
- builder.step2() ;
- builder.step3() ;
- …

Ces méthodes ne sont pas définies (non figées) dans le builder, elles sont « abstract » en vue d'être définies dans des builders concrets et variés. A chaque fois, on pourra communiquer au Director un builder différent

➔ Résultat : changement de comportement du Director sans qu'il ne soit recompilé.



Remarques :

- Le DP Factory Method ressemble au DP Abstract Factory (dans le diagramme de classe), mais le premier permet de construire un objet complexe à base d'un ensemble d'objets élémentaires de même type. L'objectif étant la création de l'objet complexe, l'objet complexe change d'aspect sans avoir à recompiler le créateur. Le deuxième (Abstract Factory) permet de construire un objet complexe à base d'une collection d'objets variés (à chaque fois différente). Ce DP a donc pour objectif de remplacer la création des objets à la manière classique avec des new, en utilisant à chaque fois une fabrique différente permettant d'obtenir une collection différente.
- Le DP Abstract Factory et le DP Builder se ressemblent, mais l'Abstract Factory fourni des objets à l'aide de méthodes encapsulant l'opérateur New. Cependant, le Builder fourni des méthodes (et non des objets) donc du comportement sous forme de plusieurs lignes de code (des algorithmes) au choix.

Chapitre 3. Design Patterns de Structure

3.1 Adaptateur : Adapter

Le Design Pattern Adapter permet d'adapter les accès à des objets de classes différentes fournissant les mêmes services mais avec des interfaces d'accès différentes.

Exemples :

Une classe **A** fournissant la méthode **trier()**

Une classe **B** fournissant la méthode **sortById()**

Une classe **C** fournissant la méthode **sortByName()**

Pour que le client puisse utiliser ces classes indifféremment et donc fournir le même code d'accès quelque soit l'objet de classe à utiliser, il suffit de créer un adaptateur par classe fournissant le même service de la classe adaptée mais avec le même nom de méthode (même interface d'accès).

Remarque :

Il existe 2 implémentations possibles de ce DP. :

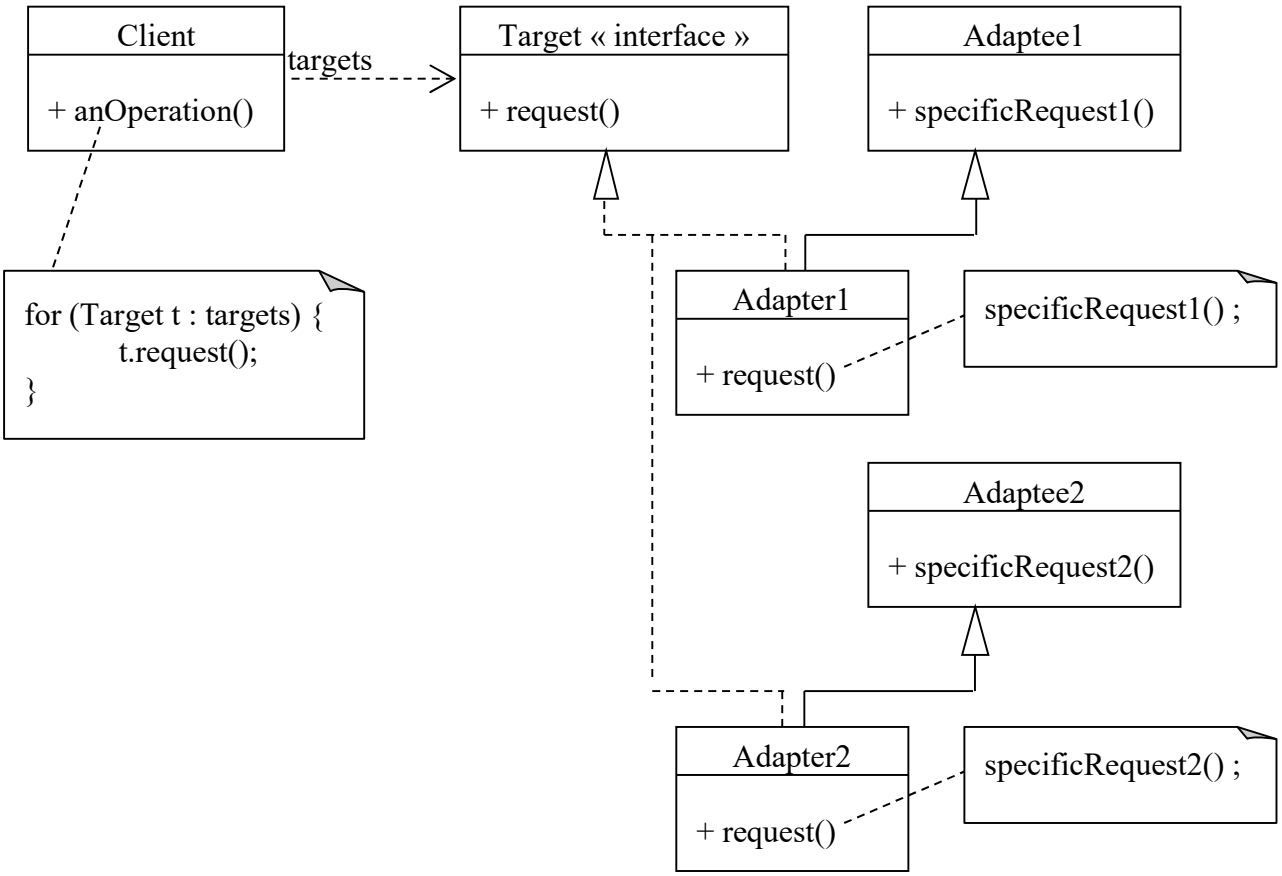
- **Adaptateur de classe** (**Class Adapter**) :

  Dans ce cas la relation entre l'adaptateur et la classe à adapter est une relation d'héritage.

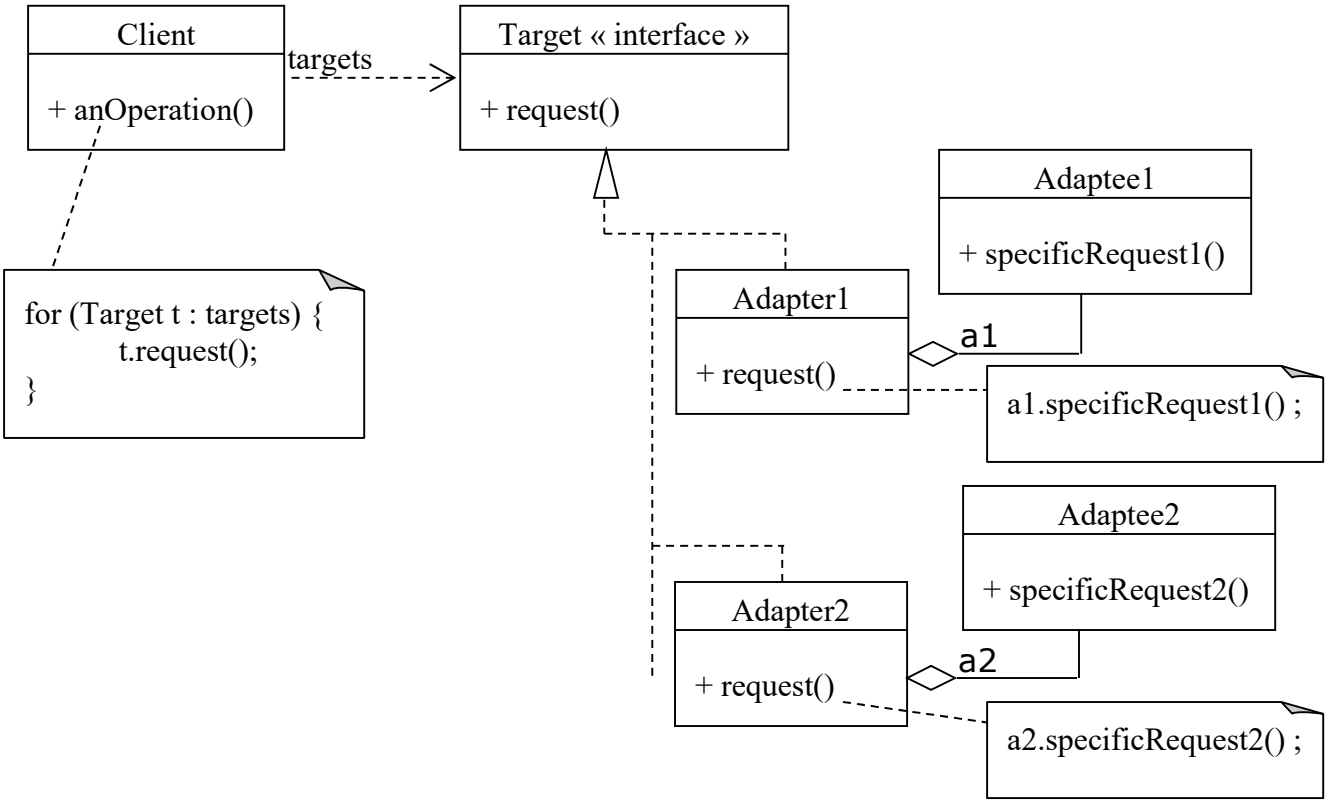
- **Adaptateur d'objet** (**Object Adapter**) :

  Un objet de la classe à adapter est contenu dans l'adaptateur (relation d'agrégation).

Adaptateur de classe (Class Adapter)



Adaptateur d'objet (Object Adapter)



Solution classique :

```
public class A {
     public void trier() {
     }
}

public class B {
     public void sortById() {

     }
}

public class C {
     public void sortByName() {

     }
}

public class ClassicalSolution {
     public ClassicalSolution(int type) {
          //...
          switch (type) {
          case 0 : sortA();break;
          case 1 : sortB();break;
          case 2 : sortC();break;
          }
          //...
     }

     void sortA() {
          A a = new A();
          a.trier();
     }

     void sortB() {
          B b = new B();
          b.sortById();
     }

     void sortC() {
          C c = new C();
          c.sortByName();
     }
}
```

Solution basée sur le DP Adapter

```
public interface Target {// la mère
     void sort();
}

public class AAdapter extends A implements Target {
     public void sort() {
          super.trier();
     }
}

public class BAdapter extends B implements Target {
     public void sort() {
          super.sortById();
     }
}

public class CAdapter extends C implements Target {
     public void sort() {
          super.sortByName();
     }
}

public class DPSolution {
     /**
      * Les appels sont réalisés à l'aide
      * d'objets des classes :
      * AAdapter, BAdapter, CAdapter, ...
      */
     public DPSolution(Target target) {
          //...
          sortAdapter(target);
          //...
     }

     void sortAdapter(Target target) {
          target.sort();
     }
}
```

3.2 Le Pont : Bridge

Ce Design Pattern permet de faire une séparation entre 2 types de traitements différents classiquement entrelacés. On appellera cela, une séparation entre l'Abstraction et l'Implémentation. En d'autres termes : une séparation entre un premier type de comportement qui supposera que le deuxième type étant réalisé par un implémenteur externe indépendamment de la nature de ce dernier.

On prendra comme exemple, l'affichage sur une interface appropriée d'un ensemble de données issues de sources quelconques.

Solution classique :

La même instruction lit et affiche les données

➔ Les choix de la source de données et de la cible d'affichage sont figés et entrelacés.

Exemple :

Text1(i).text = recordset(i).value

➔ La source est une recordset et la cible est une zone de texte.

Solution basée sur le DP. Bridge :

Faire une abstraction de la source de données

➔ externaliser le traitement de lecture des données

➔ faire un pond entre 2 structure : la classe d'affichage et la classe (l'interface : l'implémenteur) d'extraction de la donnée.

➔ Les 2 structures peuvent être définies et ainsi évoluer séparément.

➔ Différents implémenteurs réutilisables peuvent être définis

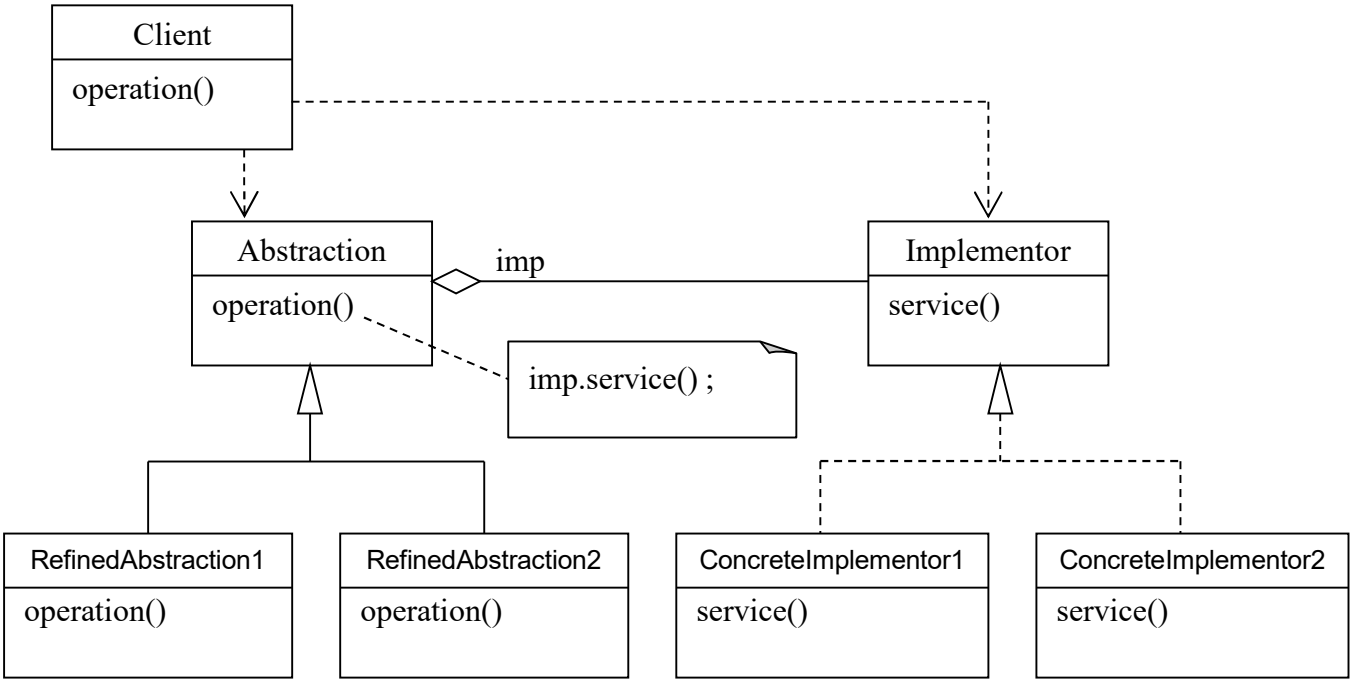
➔ On peut envisager une combinaison quelconque entre la partie Abstraction et la partie Implémentation.

Exemples Java :

1. La séparation : AbstractButton → ActionListener ➔ plusieurs ActionListeners peuvent être envisagé et ainsi associés à différents types de boutons

   ➔ Séparation : View → Controler

2. La séparation : Component → LayoutManager : permet de choisir librement pour un Component quelconque (JPanel, JLabel,…) un LayoutManager approprié (FlowLayout, BorderLayout, etc.)
3. La séparation : View → Model. Exp : JList → ListModel. Cette dernière pouvant être implémentée différemment : DefaultListModel, XMLListModel, JDBCListModel, etc.



Exemple à Implémenter :

Réaliser un afficheur de données sur des panneaux structurés :

- Afficheur sur Grille,
- Afficheur sur formulaire défilant,
- etc.

Quel que soit la source de données. On peut envisager comme source de données :

- un Parseur XML,
- un Système de connexion Base de données,
- un Ficher Excel,
- un fichier texte formaté en colonnes,
- etc.

3.3 Le Design Pattern Composite

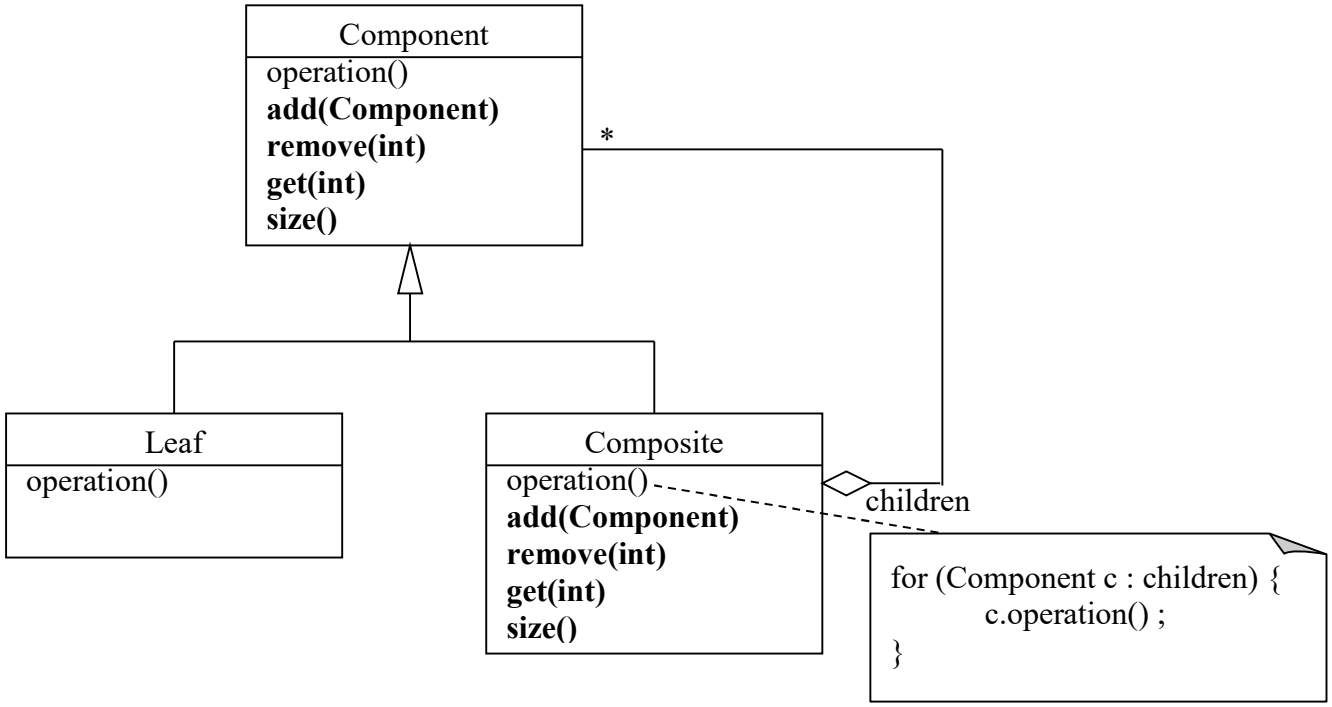
Le composite permet l'implémentation d'un schéma de structure arborescente. Toute structure nécessitant une imbrication de contenu peut faire appel à ce Design Pattern. Comme exemple d'implémentation Java, on dispose des 2 classes **Component** et **Container**. Ce schéma exploite d'une manière circulaire la force des 2 connecteurs possibles : **l'héritage et l'agrégation** :

- Le **add** du **Container** permet d'introduire dans le **Composite** (**Container**) différent Component (l'agrégation).

- Le fait que le **Composite** (étend le Component) ➔ On pourra mettre un Composite dans un autre Composite ➔ Schéma de l'arbre.

Dans le Schéma des GOF, les méthodes **add()**, **remove()** et **get()** (il faut ajouter aussi **size()**) sont définies vides dans le Component et redéfinies correctement dans le composite (mais, non redéfinie dans la classe Leaf). En langage Java, ces méthodes sont plutôt définies dans la classe **Composite** (**Container**).

L'opération proposée par les GOF dans les classes de ce Design Pattern peut être comme exemple, une opération d'affichage (explorer) : pour explorer un arbre il suffit d'afficher l'info du nœud et d'explorer (récursivement) les nœuds fils.



3.4 Le Décorateur : Decorator

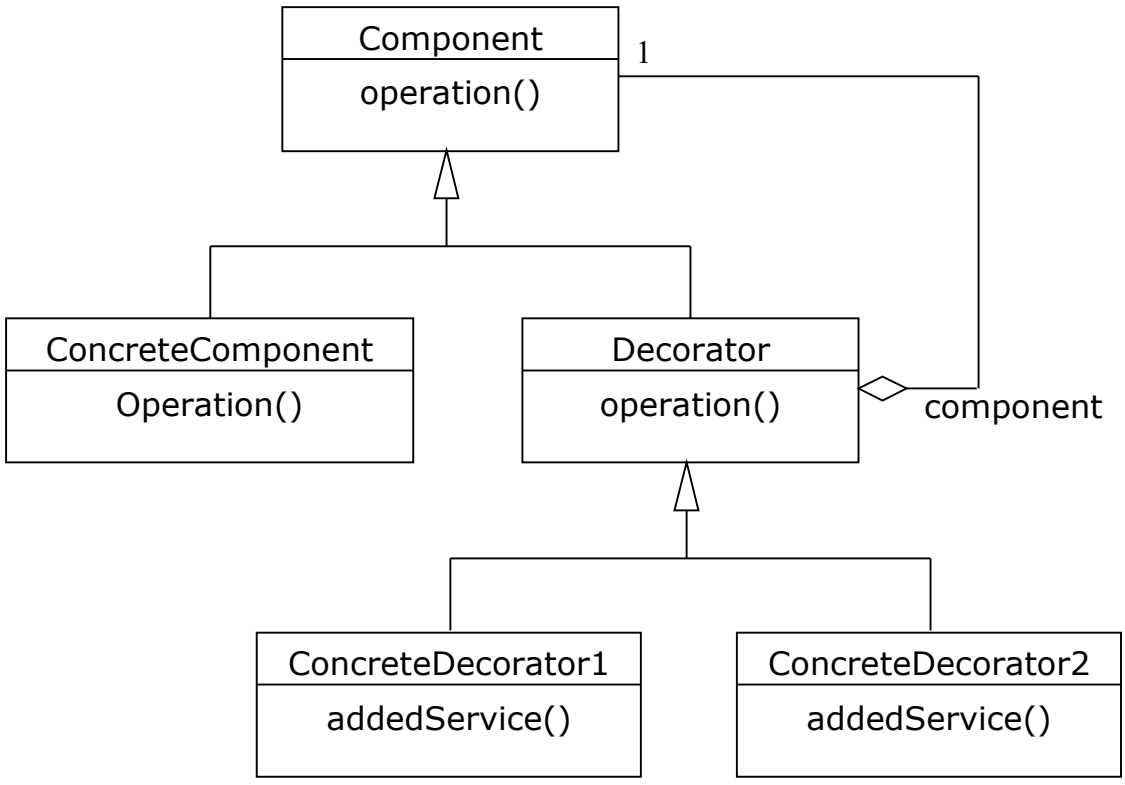
Il ressemble dans sa structure au Design Pattern Composite mais avec un objectif différent : la possibilité de superposer les Décors appliqués à un Component

➔ la règle :

Le composant décoré est lui-même un composant.

Pour cela il faut raisonner par « héritage + Agrégation » :

Ainsi, si on prend un Compsant C1 et un Décor D1, D1 appliqué à C1 ➔ D1(C1) un nouveau composant acceptant d'être décoré lui-même. Si on prend un nouveau Décor D2, on peut alors faire : D2(C1), mais plus important : D2 (D1 (C1)).



Exemples Java :

1. Un BufferedReader est un Reader (Hérirage) qui s'applique (agrégation) à un autre Reader. Ainsi si on prend un FileReader qui est un Reader, on peut lui appliquer un BufferedReader. Ensuite au BufferedReader qui est lui-même un Reader, on peut appliquer un LineNumberReader (classe fille de BufferedReader).

2. Un JScrollPane est un JComponent (Héritage) qui peut s'appliquer à un JComponent (Agrégation). Si on prend un JList (qui est un JComponent), celui là peut être décoré par un JScrollPane qui est un bon JComponent acceptant d'être ajouté à un conteneur (JPanel).

Exemple 1 :

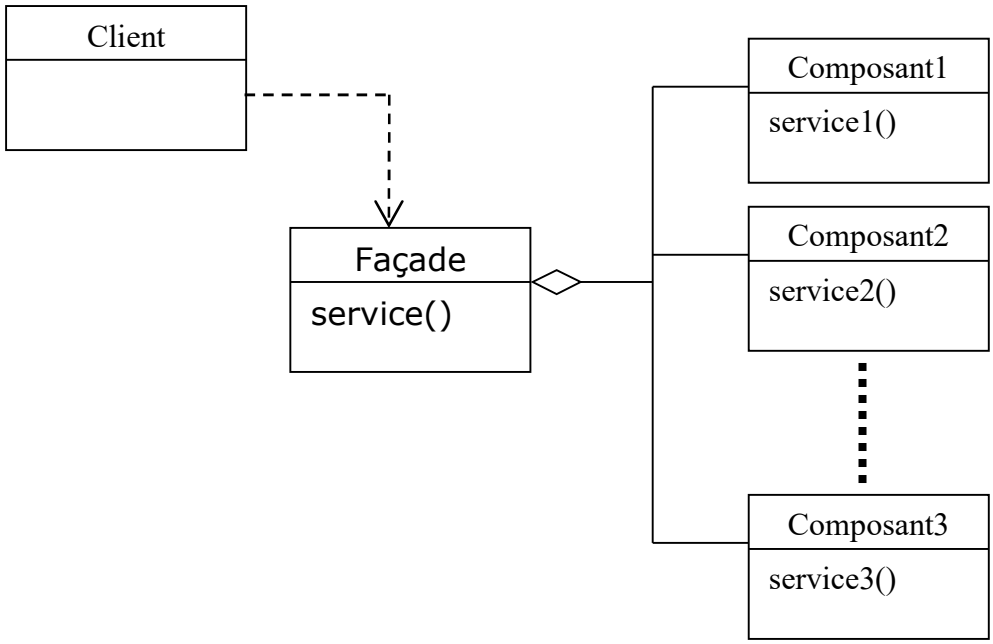
LabeledTextField (zone de texte étiquetée), BorderedTextField (zone de texte avec bordure) et BorderedLabeledTextField (zone de texte étiquetée et avec bordure) : classiquement, il s'agit de 3 classes différentes. Mais avec ce Design Pattern, on sépare le décorateur ➔ Labeled, Bordered : seulement 2 classes + la flexibilité d'appliquer Labeled à Bordered ou Bordered à Labeled (sans créer une nouvelle classe)

Exemple 2 :

Réaliser un décorateur HTML du texte (String). Les décors sont : italic (…), bold (…), etc.

3.5 Le Design Pattern Façade

C'est l'un des Design Patterns les Plus utilisés et les plus simples à mettre en œuvre (basé uniquement sur l'agrégation). Il s'agit d'encapsuler un ensemble d'objets de classes variées dans une même classe appelée façade permettant de simplifier l'accès (généralement complexe) aux différentes classes.

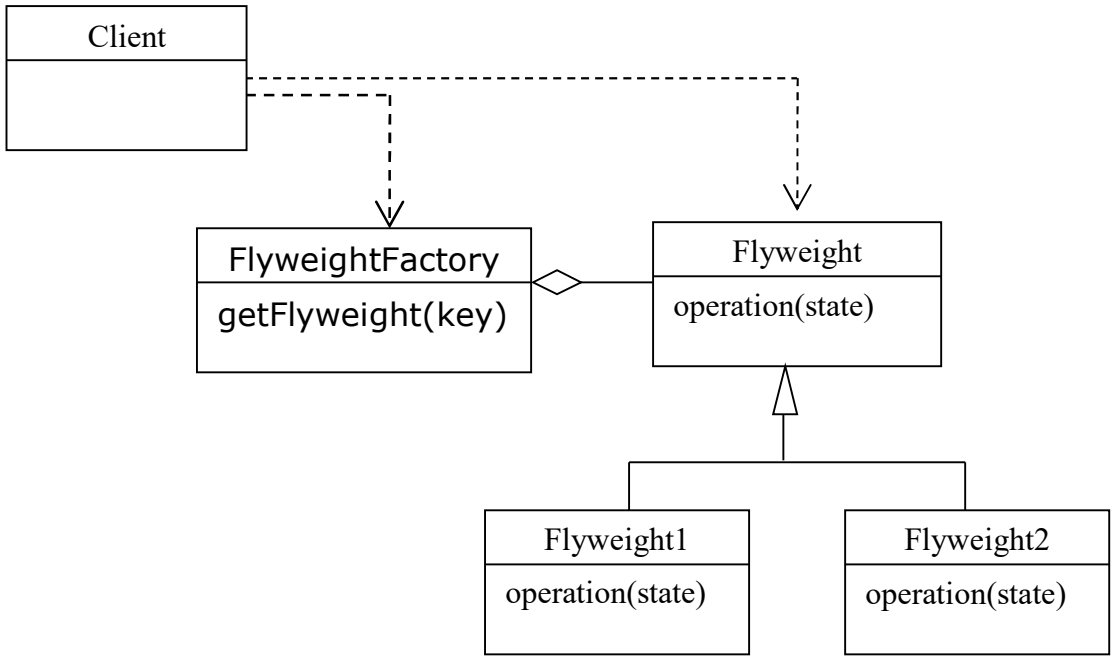


Exemples :

- Une classe Database encapsulant : Class, Connexion, DatabaseMetaData, Resultset, etc…
- Une classe Form encapsulant des JLabel, JTextField, etc..

3.6 Le Design Pattern poids mouche : Flyweight

Permet de repérer des objets utilisés plusieurs fois dans la même application. La solution est d'en créer une seule instance à enregistrer dans une fabrique (Flyweight Factory) qui se charge de délivrer ces objets aux clients demandeurs. Ces objets peuvent être identifiés à l'aide d'une clé utilisée pour leur extraction depuis la fabrique (la fabrique peut par exemple être une Hashtable). Ils peuvent aussi être configurables par l'intermédiaire de paramètres de configuration externes appelés : propriétés extrinsèques.



Exemple :

- Un bouton « Exit » avec son comportement (L'ActionListener permettant de fermer la fenêtre).
- Un bouton « OK », mais le comportement (ou l'étiquette) constitue la propriété extrinsèque sous forme d'un ActionListener (ou setText(…)).

3.7 Le Design Pattern Procuration: Proxy

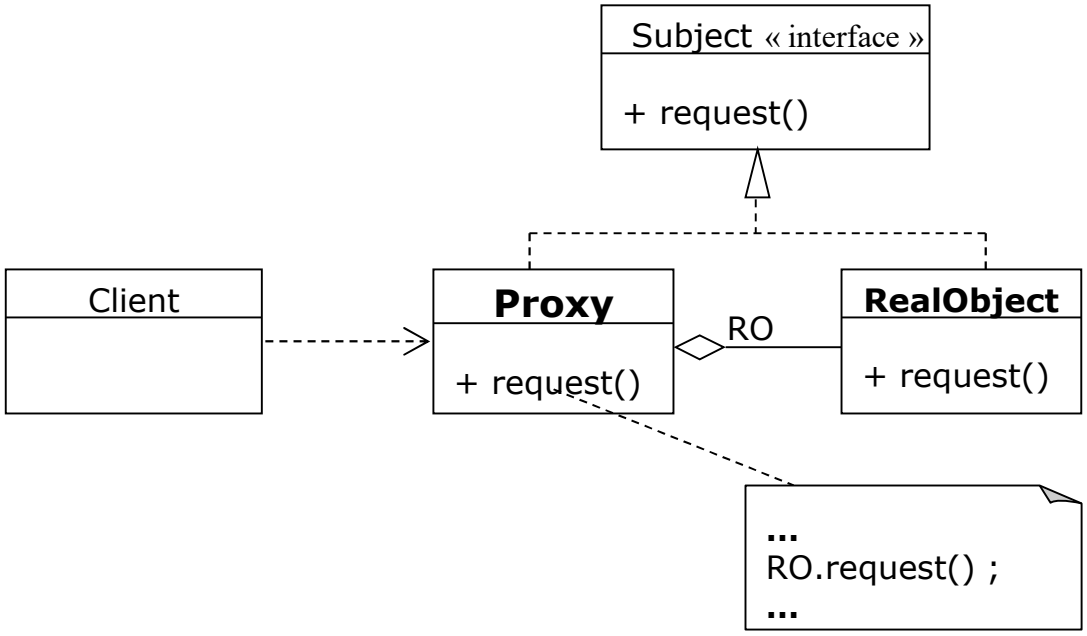
Il ressemble dans sa structure à un Adapter mais avec des objectifs différents :

- Contrôler ou Sécuriser les accès aux services d'un RealObject
- Limiter les services accessibles en fonction de la nature du client ➔ un Proxy différent par Client différent (mais, avec la même interface d'accès)
- Corriger les services du RealObject pour éviter des accès pouvant se terminer en erreur.

  Exemples :

  1- une méthode getName(…) qui retourne dans le RealObject un String pouvant être nul ➔ getName().equals(…) peut se terminer en erreur. A corriger par un Proxy retournant le cas échéant une chaîne vide.

  2- getImage() dans le Browser permettant de télécharger les images ➔ tant que l'image n'est pas prête le document est bloqué. Correction : un Proxy qui délivre le cas échéant une icône de remplacement ➔ on peut continuer l'interprétation du document en attendant l'arrivée des vraies images.
- On peut envisager de fournir de nouveaux services faisant appel à plusieurs services du RealObject



- On pourra aussi envisager un service de journalisation permettant d'améliorer les services fournis par le RealObject.

Exemple :

```
public class RealObject {
     public void insertData(…) {
          …
     }
}
```

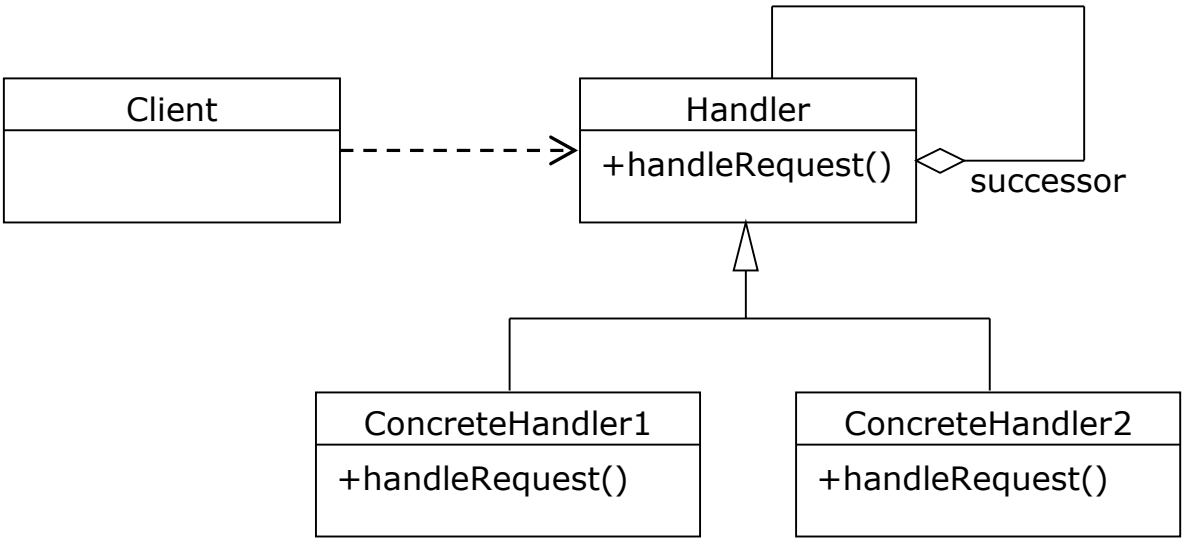
➔

```
public Proxy {
     private RealObject RO ;
     private Logger logger = new Logger() ;
     Proxy(RealObject RO) {
          This.RO = RO ;
     }
     public void insertData(…) {
          RO.insertData() ;
          logger.setLog("insert",
                         "data inserted", date, client …);
     }
}
```

Chapitre 4. Design Patterns de Comportement

4.1 Chaîne de Responsabilité : Chain Of Responsibility

C'est une structure de classes implémentée sous forme de liste chaînée polymorphe dont chaque élément a une responsabilité vis-à-vis d'un objet en entrée. Ainsi, ce dernier peut être passé au premier nœud de la chaîne. Chaque nœud se charge d'analyser l'objet en entrée ; s'il peut lui appliquer le traitement dont il est responsable il le fait, sinon il le fait passer au nœud suivant.



Comme exemples d'application de ce Design Pattern, on pourra citer :

1. Une chaîne de production, dans laquelle chaque nœud apporte un traitement à l'objet en cours de construction.
2. Une chaîne d'employés (chef de services) qui apportent chacun un traitement à un papier administratif : la secrétaire rédige un papier, le chef corrige le papier, le directeur signe, …
3. Une structure d'héritage dans laquelle chaque constructeur d'une classe appelle le constructeur de la classe mère. Dans ce cas, tous les constructeurs peuvent apporter du nouveau à l'objet à

construire. Le mot clé super permettra de faire passer une information à traiter d'un constructeur à l'autre.

4. Une chaîne d'automates et dont un seul sera capable d'extraire la prochaine unité lexicale.

Remarque :

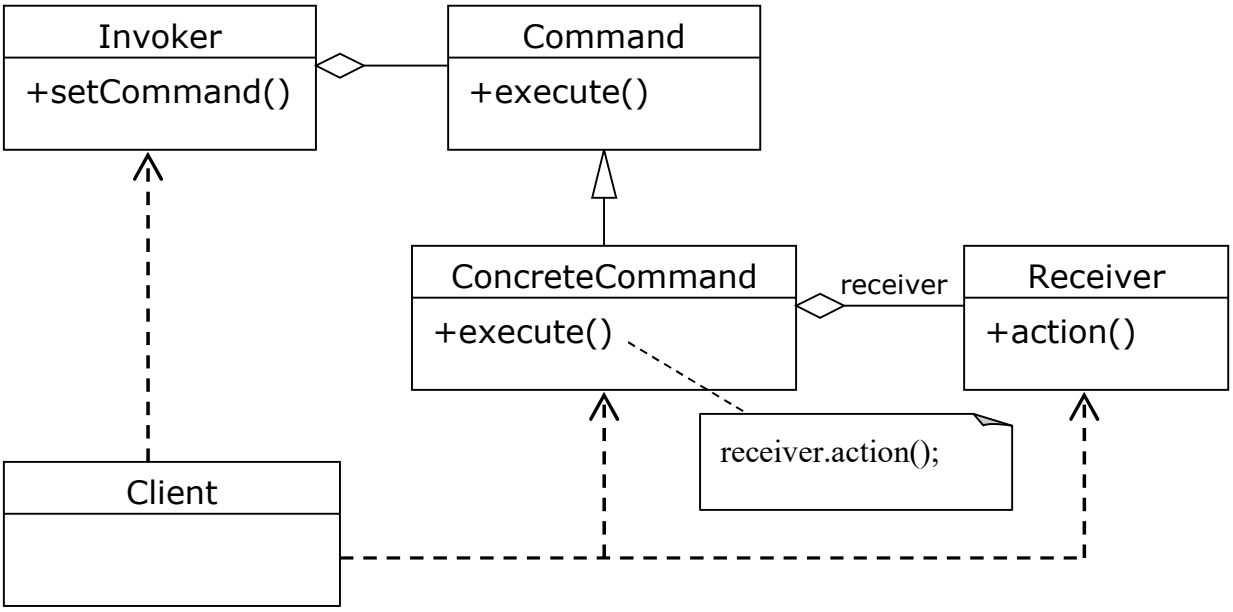
Il est nécessaire dans ce cas de considérer un traitement par défaut, le cas où aucun élément de la chaîne n'arrive à traiter l'objet.

4.2 Le Design Pattern Commande: Command

Il s'agit d'associer une commande « Command » différente à chaque objet différent. L'objet est appelé « **Invoker** », ce qui permettra d'avoir l'exécution de la commande une fois l'Invoker connu et sans avoir à procéder par des « **if...else** ».

L'exemple le plus approprié en Java est l'implémentation des écouteurs d'événements : à chaque contrôle de l'interface (« l'Invoker ») on associe, par l'intermédiaire d'une méthode addXListener, un écouteur (« **ConcreteCommand** ») qui implémente l'interface XListener (« **Command** »).

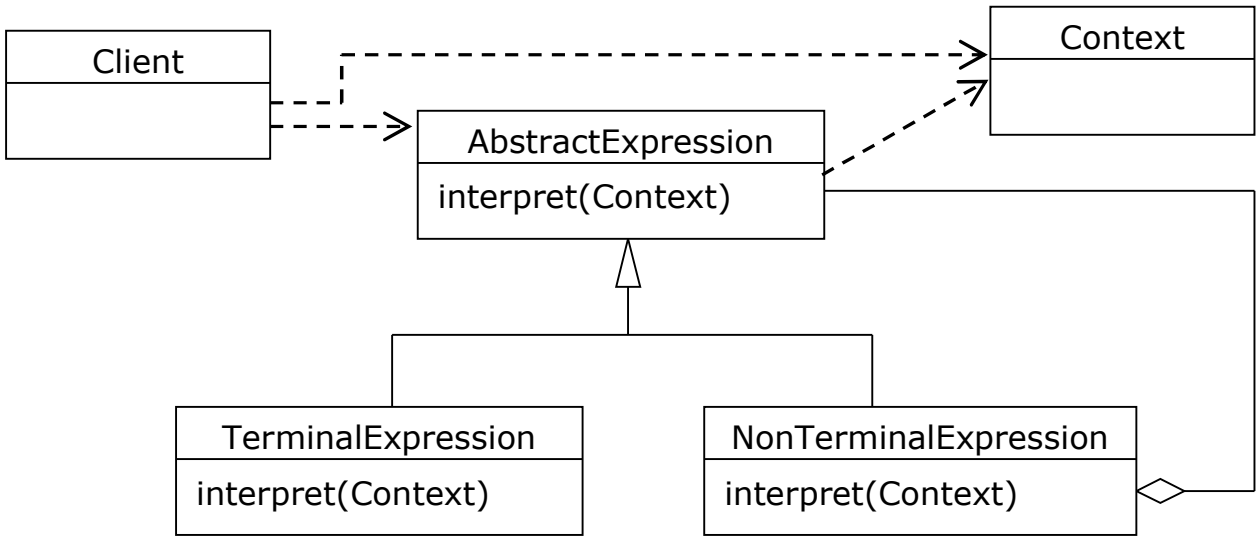
Le « **Receiver** » est dans ce cas le contrôle qui va être utilisé par l'écouteur (ou qui va subir l'action de l'écouteur).



4.3 Le Design Pattern Interpréteur: Interpreter

Il permet d'implémenter sous forme de Composite un schéma de traduction dirigée par la syntaxe : C'est une implémentation de l'analyseur syntaxique d'un compilateur en vue d'interpréter un programme ou une expression en entée.

Le schéma est implémenté d'une manière arborescente, où chaque nœud de l'arbre est soit un terminal (dans tel cas c'est une feuille) ou un non-terminal et dans ce cas il représente une sous expression à interpréter.

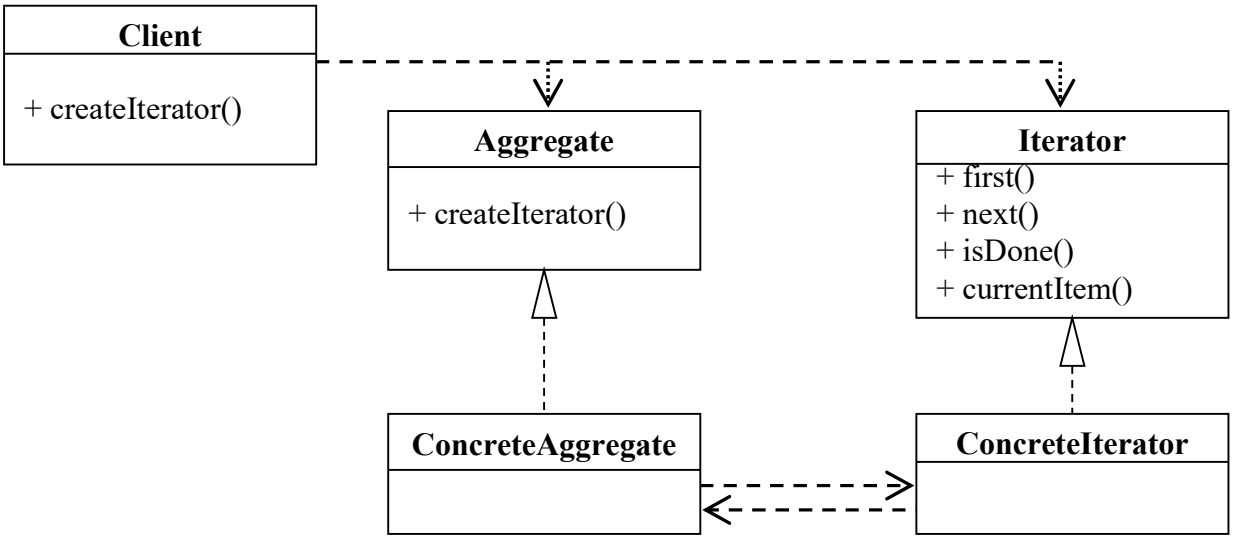


Exemple :

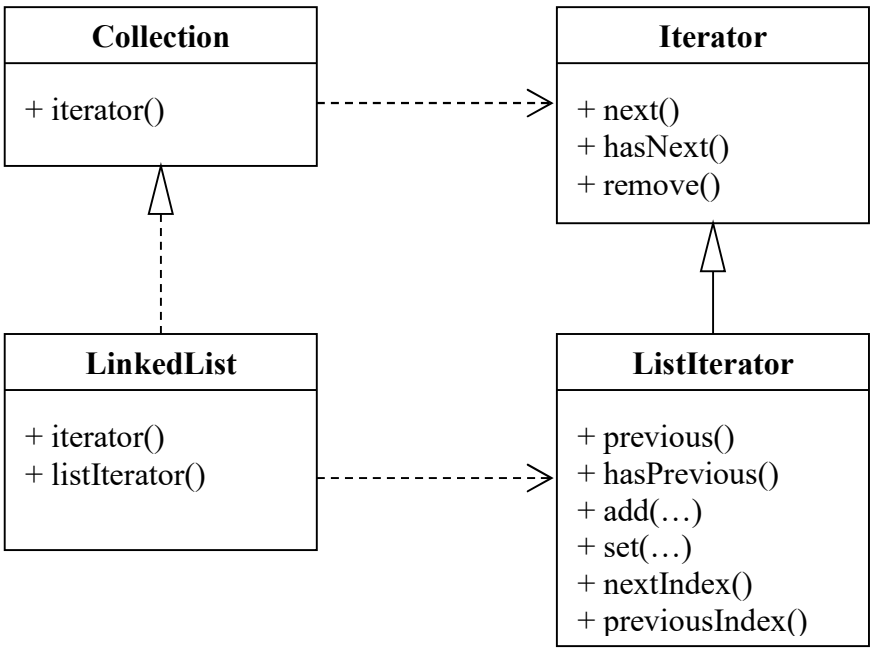
Un évaluateur d'expressions arithmétiques.

4.4 Le Design Pattern Itérateur: Iterator

Ce Design Pattern donne une solution permettant de garder un pointeur sur les nœuds consécutifs d'une liste chaînée (appelée aussi aggrégat d'objets : **aggregate**). Celui-ci (le pointeur) sera accessible par l'utilisateur final sans exposer la structure interne de la liste. La soluton fournit les différents services permettant de parcourir la liste et de récupérer les éléments de celle-ci. Il s'agit des méthodes : **first()**, **next()**, **currentItem()** ainsi que **isDone()** pour détecter la fin de la liste.



Il est à remarquer que l'implémentation de l'itérateur dépendra de la structure de la liste implémentée. C'est le cas des collections en Java :

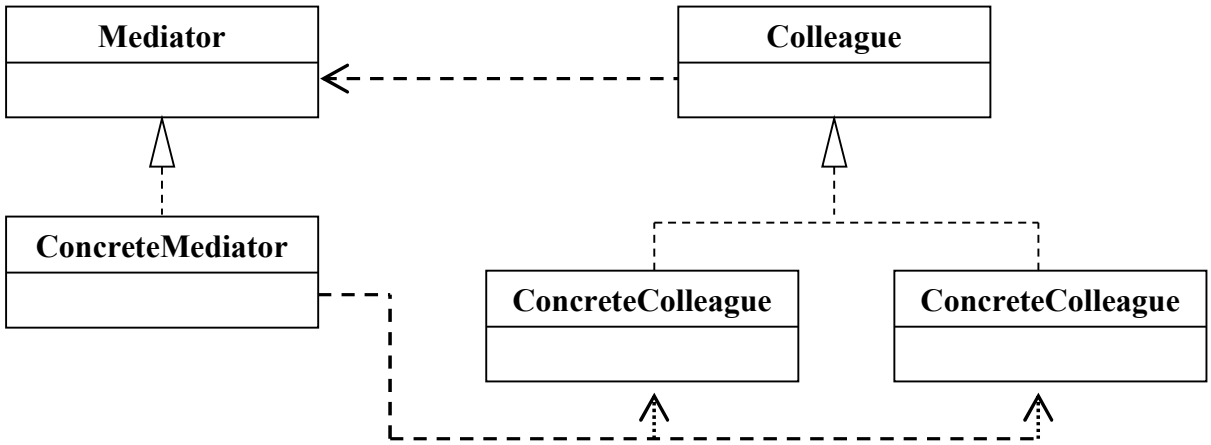


Il est à Remarquer que :

- La méthode next() remplace les 3 méthodes proposées par les GOF, qui sont : first(), next() et currentItem(). En effet, le premier appel à next() donne le même résultat que first() ; d'autre part, la fonction next() retourne la référence de l'objet courant.
- Iterator et ListIterator sont des interfaces implémentées localement dans « LinkedList » (en réalité, elle sont implémentées dans AbstractList qui est la classe abstraite mère de LinkedList).

4.5 Le Design Pattern Médiateur: Mediator

Ce Design Pattern permet de centraliser (comme un Hub ou un concentrateur) le traitement dans une seule classe appelée « **médiateur** ». Ainsi au lieu de faire connaître un objet X à tous les autres objets de l'application (les collègues), on le fait connaître uniquement au médiateur. Tout collègue désirant demander un service à l'objet X, il le fait à travers le Médiateur. Il serait alors très simple de mettre à jour le collègue X ou ses services ou le remplacer complètement, aucune mise à jours n'est alors effectuée au niveau d'aucun collègue. Le seul code modifié est celui du médiateur.



Exemple :

Soit une zone de texte T1 déstinée à contenir des dates. On considère un médiateur permettant l'accès à cette zone de texte :

```
public class Mediator {
	private JTextField date ;

	public Mediator() {
	}

	public void setDateColleague(JComponent date) {
		this.date = (JTextField)date;
	}

	public void setDate(String text) {
		date.setText(text);
	}

	public String getDate() {
		return date.getText();
	}
}
```

Dans les collègues, au lieu de faire :

T1.setText(uneDate) ;

On procède comme suit :

m.setDate(uneDate) ;

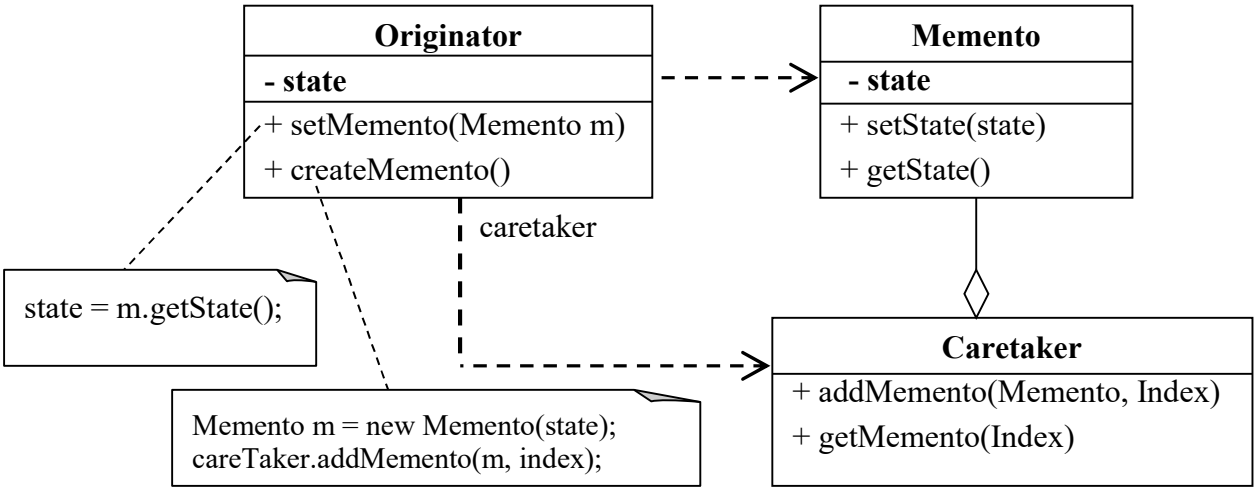
avec m étant un objet de type Mediator.

Si on veut changer la zone de texte par un autre contrôle, il suffira de mettre à jour le code du médiateur, les autres collègues ne seraient aucunement modifiés.

4.6 Le Design Pattern Mémento: Memento

C'est une solution permettant l'enregistrement des états d'un objet bien déterminé, appelé « **Originator** », à des moments particuliers, en vue de retrouver ces états dans le futur. On considère ainsi un objet dont la valeur de l'une de ses propriétés évolue tout au long de l'existence de l'objet. Par exemple un éditeur de texte tel que le contenu du document en cours de rédaction change d'un instant à l'autre, on pourra donc envisager des opérations « undo » (annuler). Un autre exemple : un salarié tel que la propriété « salaire » peut changer et on voudrais se rappeler des valeurs antérieurs, etc.

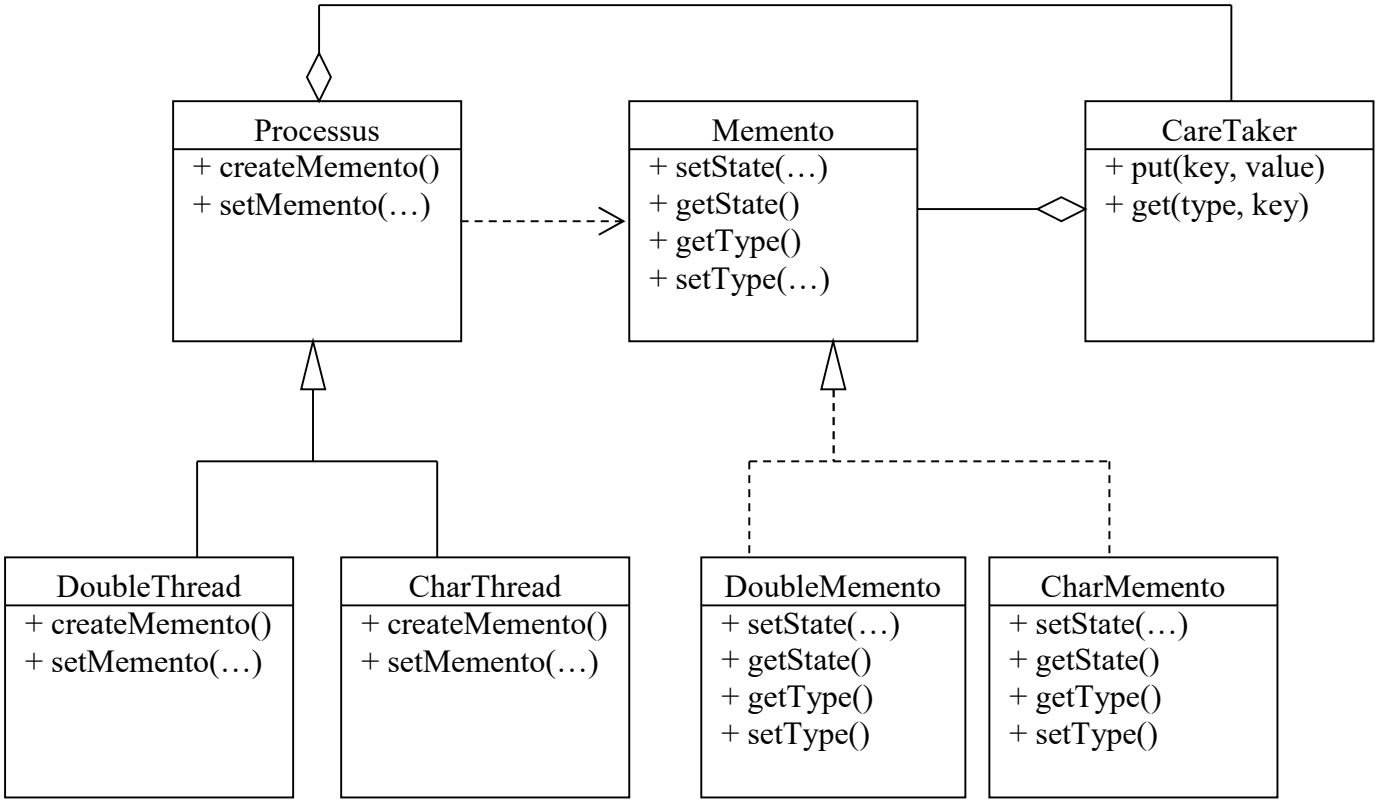
La solution du Mémento permettra de sauvegarder sous forme d'un objet appelé « **Memento** » uniquement la valeur de cet état (classiquement, on pourrait penser à enregistrer tout l'objet : le Salarié par exemple). La sauvegarde de ces Mémentos est réalisée dans une structure de données permettant de prendre soin des Mémentos enregistrés, appelée « **Caretaker** ».



Exercice :

Réalisation d'un Thread disposant d'un état de type réel (double) qui s'incrémente systématiquement dans le run() (par pas de 0,01). Le Thread s'occupe lui-même de sauvegarder ces Memento, avec comme « Index » l'instant : date et heure, dans un Caretaker basé sur une Hashtable. Le client pourra demander de retrouver l'état associé à un instant donné.

On pourra aussi réaliser un autre « originator » ; ce sera une classe Thread qui gère des lettres. L'état est une lettre qui change aléatoirement (ou par pas de 1). On aura donc une autre classe Memento mais un même CareTaker.



4.7 Le Design Pattern Observateur : Observer

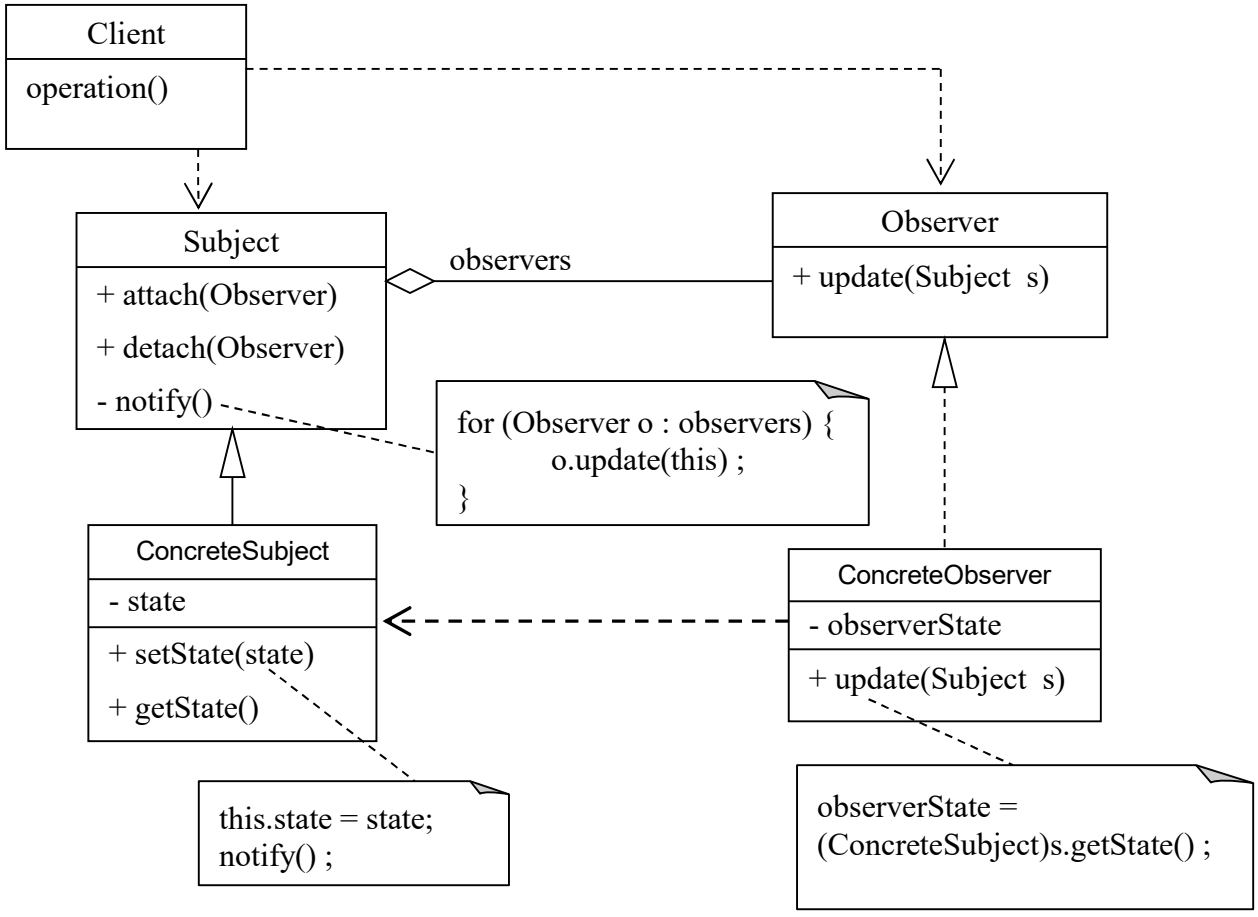
Le design pattern « Observer » permet de résoudre le problème suivant :

Faire en sorte que : à chaque modification d'un objet donné (un sujet), un ou plusieurs autres objets d'autres classes (appelés observateurs) prennent conscience de cette modification ce qui leur permettra de se mettre à jour par rapport au changement du « sujet » d'origine.

En réalité ce n'est pas l'observateur qui reste à l'écoute de tout changement du sujet, mais c'est le sujet lui-même qui à chaque modification il avise les observateurs un par un.

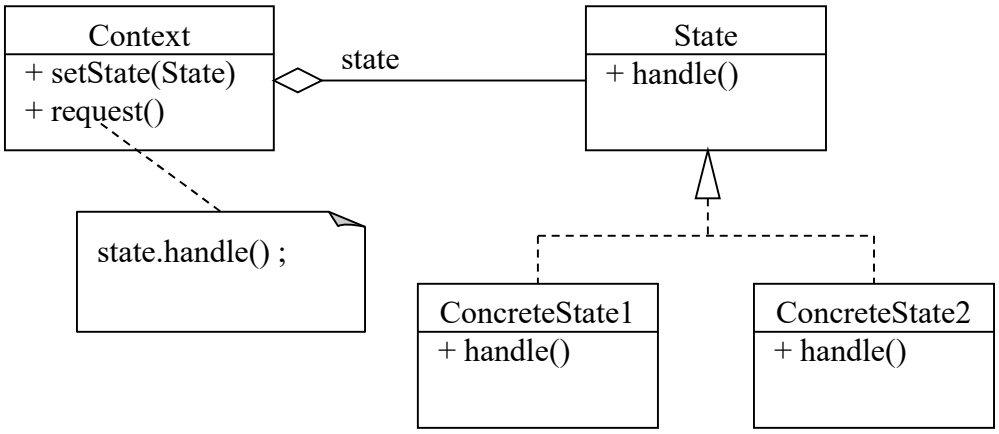
Solution :

+ La classe « Subject » doit contenir une structure permettant d'y enregistrer tous les observateurs possibles. Donc deux méthodes pour attacher et détacher des observateurs.
+ Une méthode « notify » permettant d'informer les observateurs du changement. Celle-ci doit être appelée à chaque changement du « subject »



4.8 Le Design Pattern Etat : State

Ce design pattern permet à un processus, ou généralement à un objet, de changer de comportement lorsque son état change. L'opération de changement de comportement sera réalisée sans avoir à faire des tests sur l'état (if … else if…). La solution consiste à transformer l'état en un objet encapsulant au sein de lui-même le comportement qui lui est associé. Ainsi, pour fait changer d'état à un processus, on lui communique le nouvel état sous forme d'un objet « State ».

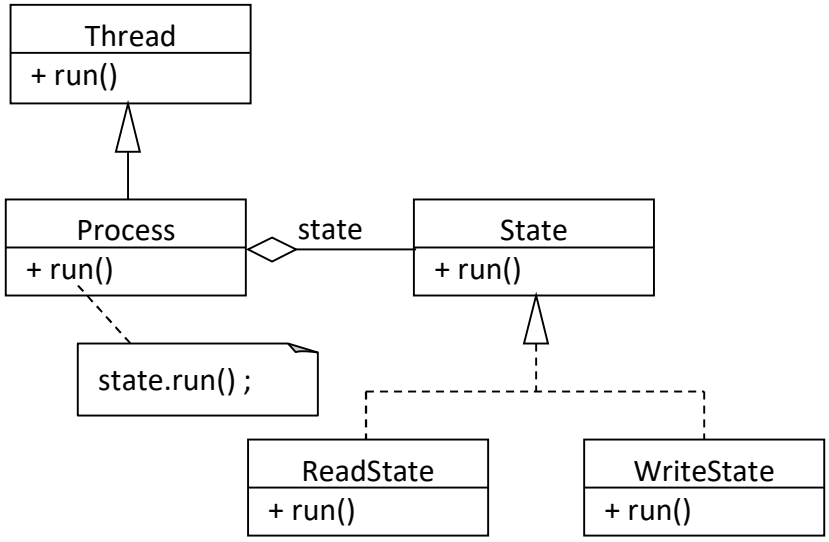


Remarque :

Le processus « Context » peut évoluer (changer d'état) d'une manière autonome, comme il peut être piloté depuis l'extérieur par un autre processus qui lui fait changer d'état selon le besoin.

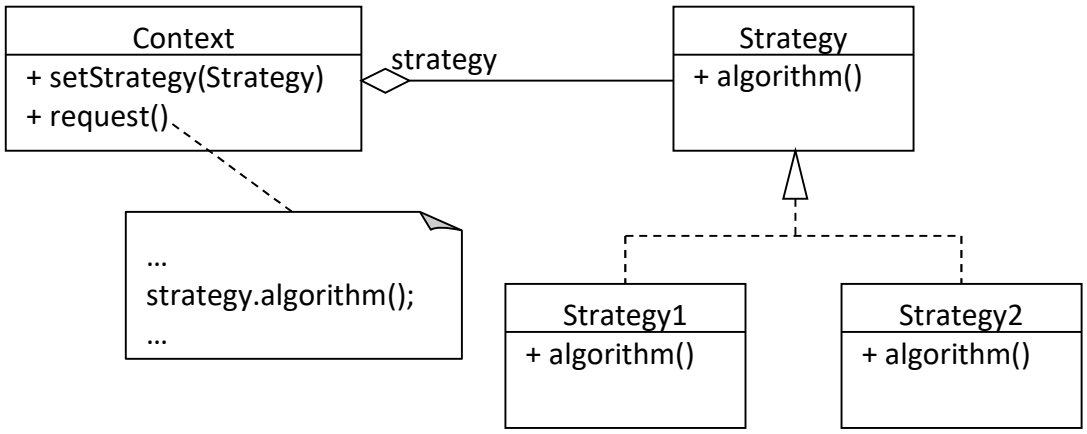
Exemple :

On voudrait réaliser un processus de Lecture/Ecriture. Le processus change d'état (lecture/écriture) aléatoirement (ou commandé par le client).



4.9 Le Design Pattern Stratégie : Strategy

Il permet à un « context » de changer de stratégie (d'algorithme), ou encore de « manière de faire », sans avoir à le recompiler. La stratégie se trouve externalisée et peut être injectée dynamiquement dans le contexte. on aura alors en conséquence un contexte paramétrable. Le design pattern « Strategy » est donc un design pattern de paramétrage (ou encore de configuration)



Exemple :

+ Algorithmes de tri.
+ Un Parseur XML

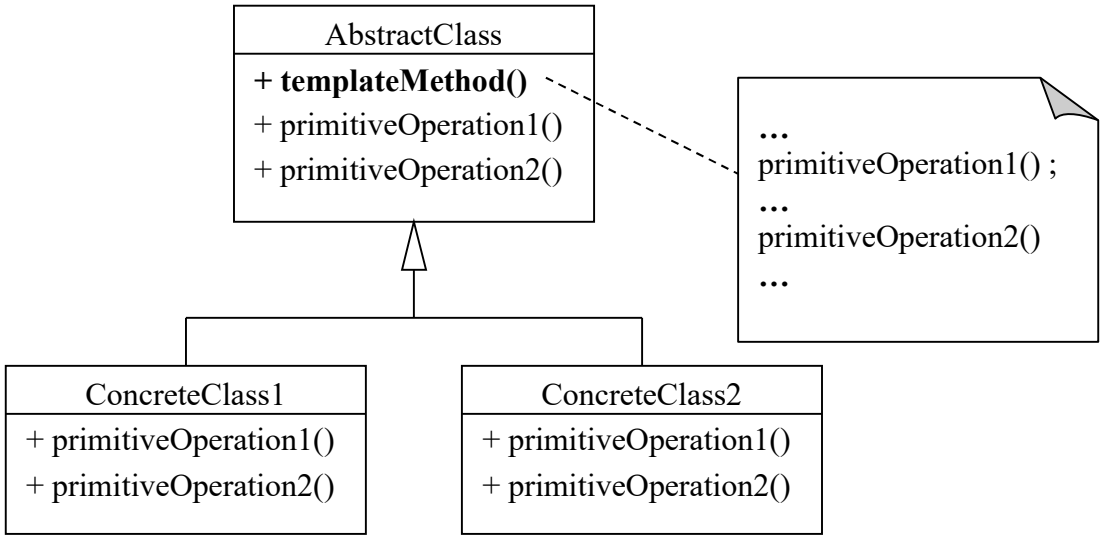
Remarque :

Ce design pattern ressemble au design pattern State dans sa structure mais avec la différence suivante :

Le design pattern State résout le problème de changement d'état d'un processus (pouvant être fréquent ou le long de son cycle de vie) qui entraîne un changement de comportement. Par contre, le design pattern « strategy » permet juste de paramétrer ou configurer « initialement » le processus pour le choix d'un seul algorithme à appliquer dans un contexte donné sans changement ensuite durant son cycle de vie.

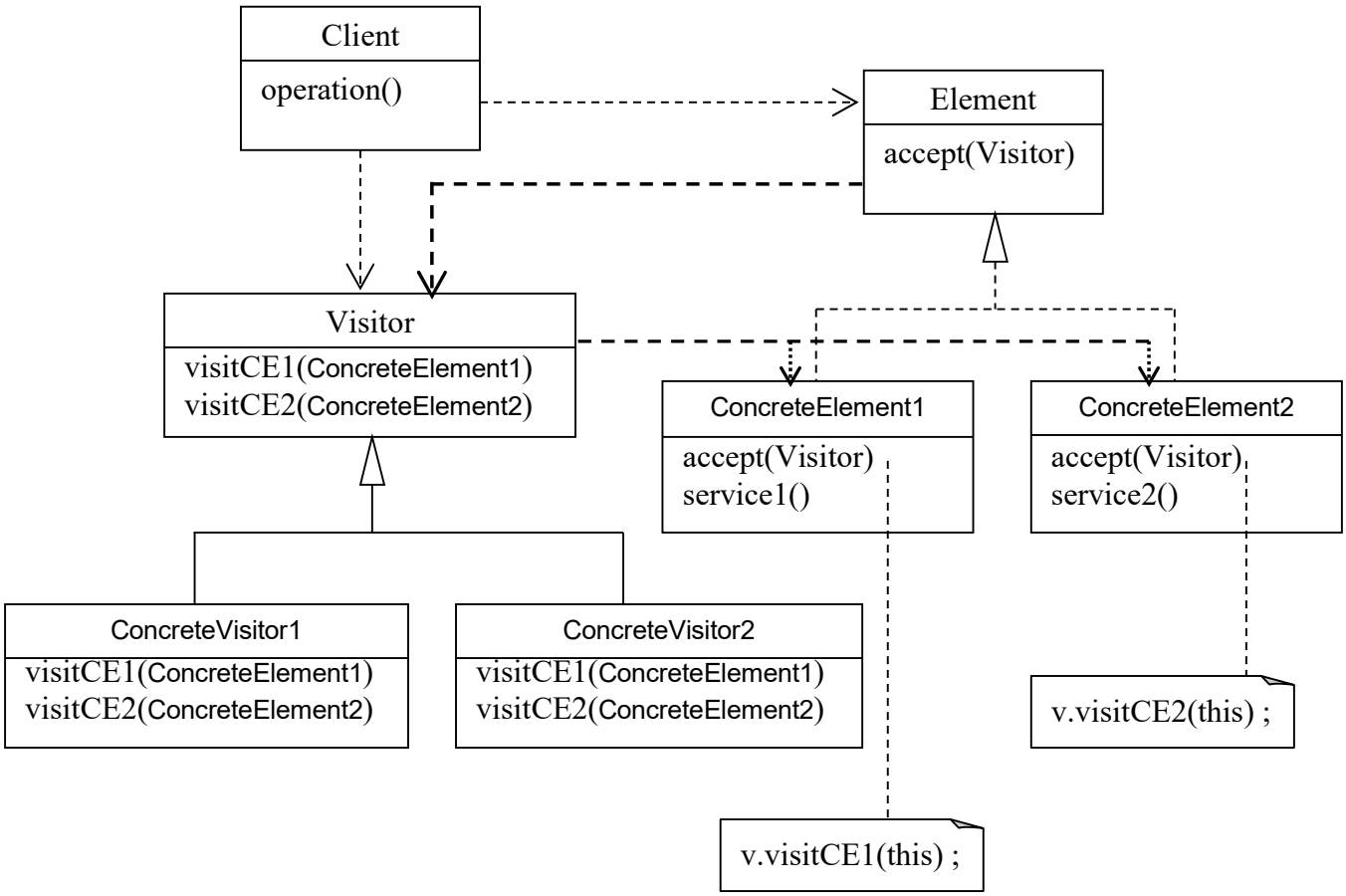
4.10 Patron de méthode : Template Method

Ce design pattern permet de réaliser un algorithme dont certaines étapes (primitives) ne sont pas figées et peuvent changer selon le besoin. Le squelette ou la structure générale de l'algorithme reste constante, mais le développeur aura la possibilité, à l'aide de l'héritage, de redéfinir (ou définir) certains blocs de l'algorithme qui sont détachés sous forme de méthodes abstraites.



4.11 Le Design Pattern Visiteur : Visitor

Le visiteur est une structure de classes permettant à une classe (« Element ») d'être extensible en termes de services sans avoir à la modifier ou à l'étendre (par héritage). C'est par l'intermédiaire d'une autre classe, le visiteur, autorisée à visiter celle-ci (« Element ») que des services supplémentaires seront rajoutés. Il s'agit d'une extensibilité **dynamique**, puisqu'il est possible de prévoir autant de visiteurs que l'on souhaite.



Ce design pattern permet d'éviter de « *polluer* » des classes avec une multitude de services de natures variées et qui change d'une implémentation à une autre. Il s'agira généralement de nouveaux services qui ne rentrent pas dans le cadre du métier implémenté par l'élément à étendre. Comme utilisation remarquable de ce principe, on pourra évoquer la programmation orientée aspects (AOP). Il est en fait préférable de réunir les aspects dans des classes séparées. Ce sera alors une séparation entre le métier de l'application et les aspects couverts par celle-ci.

Exemples : Un visiteur de journalisation, un visiteur d'authentification, un visiteur d'affichage, etc.

Remarques :

1. Le design pattern « Visitor » permet d'éviter 2 principaux problèmes :

- Retoucher à chaque nouveau besoin une classe « Element » avec des services supplémentaires.
- Utiliser des « switch » pour faire des traitements adaptés à la nature de l'élément à traiter.
- Ajouter à la classe des services de nature étrangère par rapport à celle des services déjà existants.

2. Il y a donc autant de visiteurs (exp. Afficheur, Lecteur) à ajouter que de nouveaux services à programmer (afficher, lire).
3. Un visiteur s'occupe d'un et un seul type de service ou aspect (exp. Affichage, Journalisation, transformation XML ou JSON, calcul, etc.)
4. Un visiteur utilise des services déjà existants dans l'élément à visiter (service1(), service2(), …) pour fournir un nouveau service à travers ses méthodes « visit… ».
5. Il y a autant de méthodes dans le visiteur que d'éléments à visiter (exp. A, B, C ➔ vistA, visitB, visitC). Toutes les méthodes font la même chose (exp. Affichage), mais elles portent des noms différents, des paramètres différents et un contenu ou une manière de faire différente. En fait, le paramètre de chaque méthode est l'élément sur lequel le service sera appliqué (visitA(A), visitB(B), visitC(C)).