**PATRONS**

**CREATIONNELS**

* + Fabrique (*Factory Method*)
  + Singleton
  + Monteur (*Builder*) (PAS VU)

**STRUCTURELS**

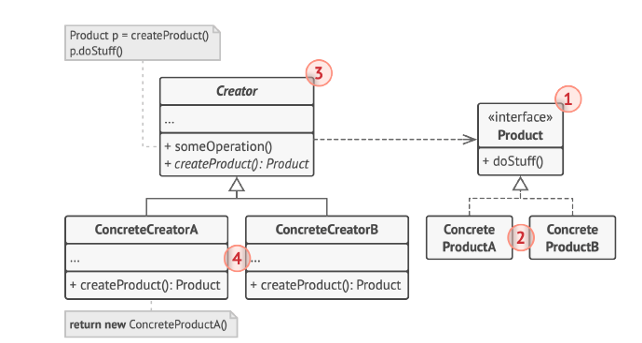
* + Procuration (*Proxy*)
  + Adaptateur (*Wrapper, Adapter*)
  + Composite (*Arbre d’objets*)
  + Décorateur (*Decorator, emballeur, wrapper)*

**COMPORTEMENTAUX**

* Visiteur (*Visitor*)
* Memento
* Commande (*Command*)
* Observateur (*Observer, Souscription*)
* MVC (***M****odel-****V****iew-****C****ontroller*)

**AUTRES**

**FABRIQUE**

La **fabrique** permet d’instancier des objets (**ConcreteProductA** et **ConcreteProductB**) qui sont d’un type *abstrait* donc d’une interface ou d’une classe abstraite (Interface **Product**).

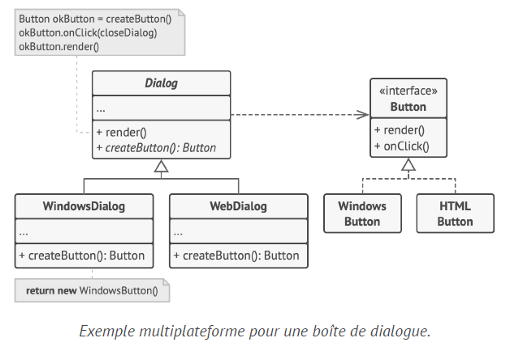
Par conséquent, la classe exacte de l’objet n’est pas connue par l’appelant (Type Product).

Cela permet d’instancier dynamiquement des sous-classes sans avoir à s’adapter par rapport à un type précis.

**Product** apporte des fonctionnalités et un type général à ses Produits concrets.

**Creator** apporte les fonctionnalités de création de Product à ses sous-classes.

Il faut préciser l’objet concret que l’on souhaite (voir ci-dessous).



On initialise un objet **Dialog** avec *new WindowsDialog* ou *new WebDialog*.

**Exemple wikipedia :**

En fonction du paramètre, on identifiera la référence à envoyer. Animal étant ici le type général des deux produits concrets Chat et Chien.



Ensuite, on déclenche **render()** sur cet objet et celui-ci exécute ce code :



Le **createButton()** s’adaptera au type de dialog.

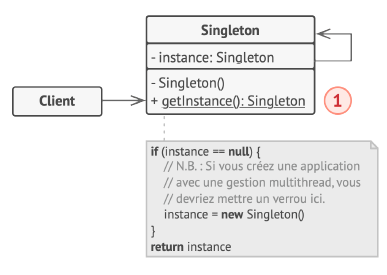
**SINGLETON**

**Singleton** garantit que l’instance d’une classe n’existe qu’en un seul exemplaire.   
C’est utile quand on veut partager l’accès à une ressource **partagée**.

Comment le mettre en place ?

-Rendre le constructeur par défaut **privé** (opérateur new impossible).

-Créer une méthode statique qui appellera le constructeur privé pour créer un objet et le sauvegarde dans un attribut statique.



Attribut instance est privé, statique et du même type que la classe ou il se trouve.

**Singleton()** est le constructeur par défaut privé.

**GetInstance()** est une méthode **publique** et **statique** qui vérifie si l’attribut a déjà été initialisé. SI ce n’est pas le cas, il l’initialise en passant par le constructeur privé. Sinon, il retourne l’attribut.



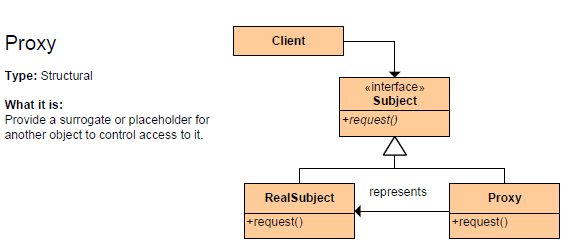
**PROXY**

**Procuration** permet d’établir un traitement préliminaire pour libérer la classe qui devait réceptionner la requête de le faire.   
S’il n’est pas validé, le proxy gère la situation lui-même sinon il donne la suite des opérations à la classe réceptrice.

**A** =>*Requête*=>**B** mais passe part **P** pour traitement préliminaire.

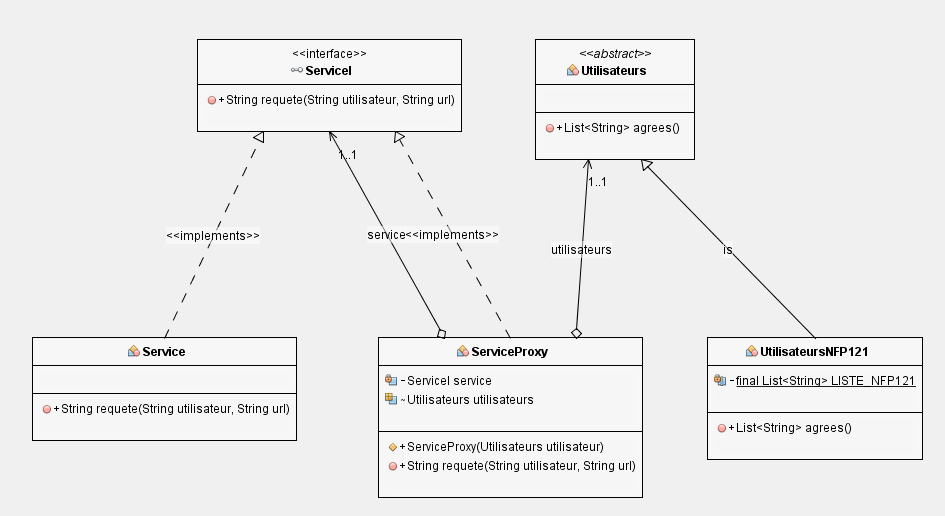
Si **P** alors *Requête* *traitée* => **B**

Sinon **P** retourne une erreur.

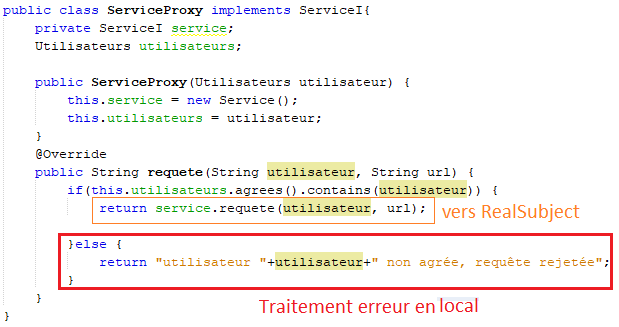


Il y a une interface **Subject** commune aux classes **Proxy** et **RealSubject** et offre la méthode *request()*.  
**Proxy** a sa création reçoit la requête à traiter et un objet RealSubject en paramètre.  
Il possède un attribut du type de l’interface pour stocker l’objet RealSubject.  
**Proxy** fait un traitement préliminaire par *request*(). Si *négatif*, il envoi le message d’erreur lui-même.  
Sinon, grâce à l’attribut realSubject, il déclenche la méthode request() de la classe **RealSubject**.

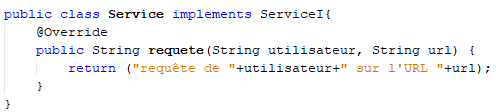
**Schéma 06\_11\_20\_ED\_Adaptateur\_Proxy** :

Utilisateur donne une liste d’utilisateurs et si l’utilisateur du request() de ServiceProxy est détecté, alors Service prend la suite et retourne le traitement final.

Le Proxy, ServiceProxy :



Le RealSubject, Service :



**ADAPTATEUR**

L’**Adaptateur** est un patron qui permet de faire collaborer des objets ayants des interfaces normalement incompatibles. Cela permet de déléguer le travail des méthodes de A dans les méthodes de B (Adapter A dans B).

J’ai deux *interfaces*, soient **Prise** et **Plug**.

**Prise** implémente la classe **Peritel**.

**Plug** implémente la classe **Rca**.

**Objectif** : pouvoir rendre possible l’utilisation des méthodes d’un Plug Rca dans les méthodes d’une prise Peritel même si elles ont des interfaces normalement incompatibles.

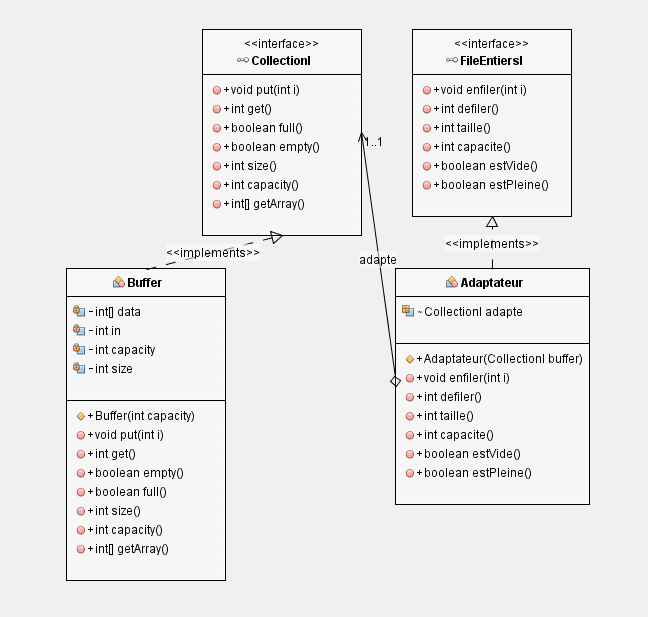
**L’Adaptateur** est une classe qui va implémenter l’interface vers laquelle on veut récupérer les méthodes (Prise) (Convertir / déléguer les méthodes de A dans B)

Le constructeur **d'Adaptateur** prendra un **Plug** en paramètre et un autre en attribut pour le stocker.

Comme on implémente la prise, on incorpore la méthode de l’interface : *peritel()*.  
Celle-ci contiendra un appel en partant de l’attribut vers la méthode qu’implémente Rca : rca.rca();

Maintenant, on a une classe **Adaptateur** qui accueille un **Plug** et grâce à son implémentation de la destination, il incorpore notre **Plug** dans la méthode.

**Schéma 06\_11\_20\_ED\_Adaptateur\_Proxy** :



On a *l’interface* **CollectionI** qui offre les fonctionnalités d’un tableau.  
Une deuxième *interface* **FileEntierI** qui offre les fonctionnalités d’une pile.

**Objectif** : vouloir utiliser les méthodes de **CollectionI** dans les méthodes **FileEntiersI**.  
C’est à dire, utiliser des méthodes qui fonctionne avec un tableau dans le corps d’une méthode faite pour les piles.

Notre **Adaptateur** implémente l’interface vers laquelle on veut s’adapter : FileEntiersI.  
Celui-ci accepte par son constructeur un attribut de l’interface qui devra s’adapter : CollectionI.

Aide à la compréhension :

Class A (interface Ai) =>(dans)=> **Class B (interface Bi)**

Class C **implements Bi** {

A a;

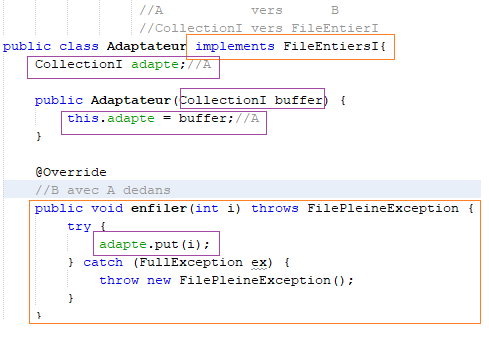
C (A a) {a = a;}

**Func1B() {**

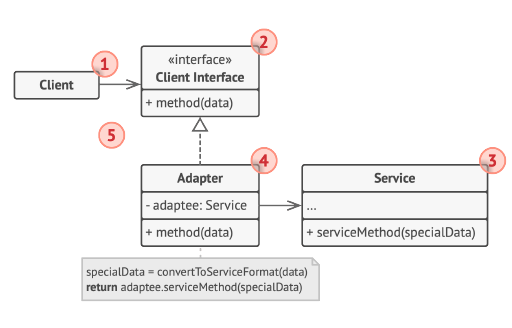
a.func1A();

**}**

}



Mise en forme différentes d’adaptateur :

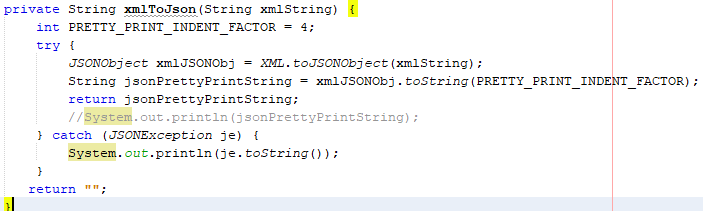


Un **Client** souhaite utiliser les fonctionnalités de Service mais il ne peut pas le faire directement.

**L’Adapter** implémente l’interface Client et encapsule l’objet service type Service.

Les méthodes de **Client** font des appels à partir de l’objet encapsulé service pour accéder au traitement de celui-ci.

Par exemple, le **Client** pourrait être du **XML** avec comme interface ayant une méthode “*convertirXMLtoJSON(Element e)*”. Et cette méthode, grâce à l’objet de type **ServiceJSON**, aurait accès à une méthode pour faire le travail.



**COMPOSITE**

Ce patron de conception comportementale permet de séparer les algorithmes et les objets sur lesquels ils opèrent.

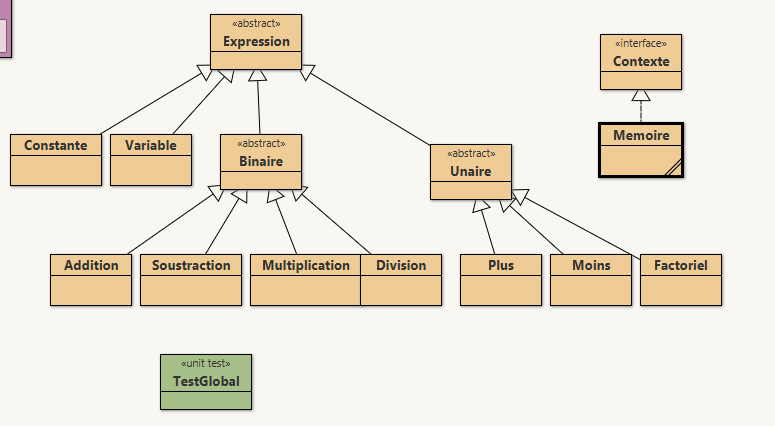
Ce-dernier est un patron de conception structurelle qui permet d'agencer les objets dans des **arborescences** afin de pouvoir traiter celles-ci comme des objets individuels.

Il y a une interface commune à toutes les classes “composant” (**componant**), les feuilles (**leafs**) et les conteneurs (**composite**) l'implémente. Un conteneur s'apparente à une **boîte** qui peut contenir soit des feuilles (ou **produits**) et éventuellement d'autres boîtes. Contrairement aux conteneurs, une feuille est une classe **terminale** car elle n'a pas de sous-éléments.

Les feuilles possèdent une méthode Interprete() qui permet d'exécuter une tâche.  
Il existe plusieurs patrons afin de se déplacer entre les différents éléments (visiteur, décorateur, ou encore interpréteur qui sera vu ci-dessous).

On verra que le patron Interprétation ne se limite qu’à une seule interprétation car sont codes est en dur (type et opération) directement dans les feuilles.  
Alors que le patron Visiteur lui déplace les opérations dans des classes spéciales et utilise la généricité pour récupérer notre type de retours qui peut très bien changer selon l’implémentation de notre visiteur.

**Exemple TP6 :**



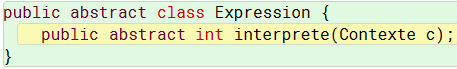
(Mettre de côté Contexte et mémoire pour l’instant)

Ce TP met en place une architecture objet qui permet d’exécuter des opérations arithmétiques imbriquées les unes ou autres. Les opérations et les constantes/variables sont des objets.

Tel que :



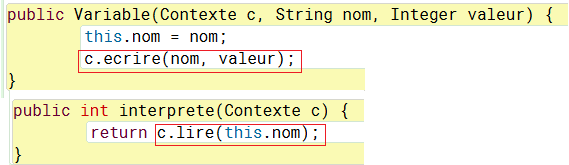
Toutes les classes étendent la classe abstraite Expression : elles sont toutes Expression.  
Elle force toutes les classes à implémenter la méthode *Interprete*() :

  
Elle possède un paramètre du type *Contexte* qui permet de chercher la valeur en mémoire de notre Variable.

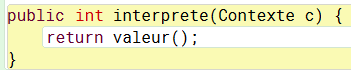
Nos classes abstract sont des **conteneurs** et elles contiennent des **feuilles**.

**Constante** et **Variable** sont nos classes terminales. Constante contient un Integer alors que le contenu de la variable, qui est une association nom/valeur, se trouve en Mémoire.

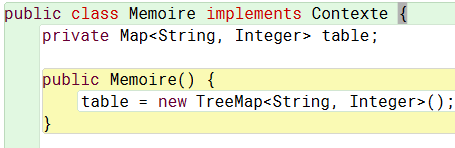
**Variable** :



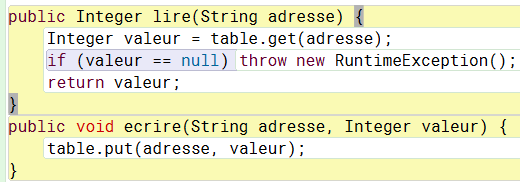
**Constante** :



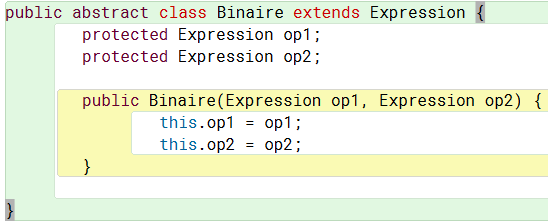
Une **Memoire** est une classe qui possède un tableau :

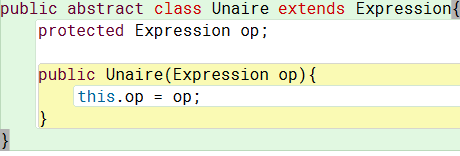


**String** est le nom de la variable et **Integer** sa valeur. On peut *y lire ses données* ou encore *en écrire :*

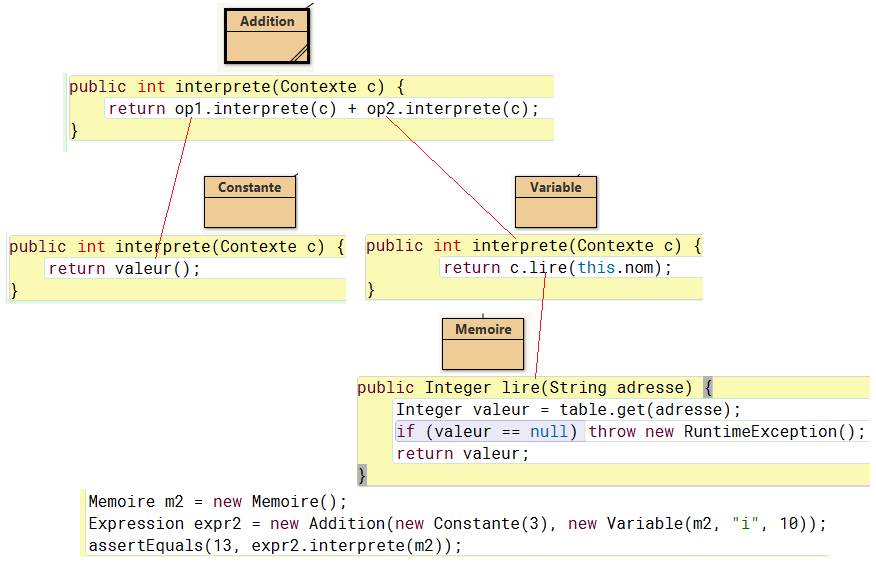


Les conteneurs **Binaire** et **Unaire** permettent de réduire la *duplication de code* et de le rendre plus logique.  
Les classes qui héritent de **Binaire** prennent dans leurs constructeurs 2 paramètres Expression :

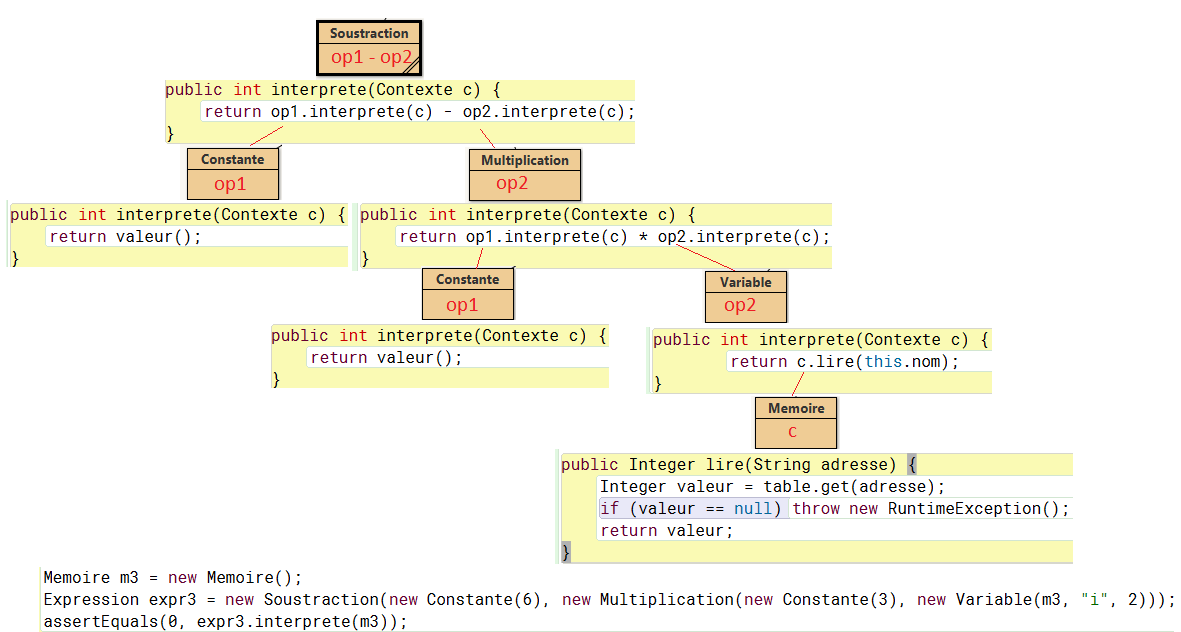
  
Alors que celles qui héritent d'Unaire en ont qu‘une :



La méthode *interprete()* travail avec les **2 Expressions** enregistrées par le constructeur.  
  
On exécute sur ces deux Expressions leurs méthodes Interprete() respectives qui iront chercher individuellement un résultat terminal.   
Exemple simple 1:



Exemple moins simple 2 :

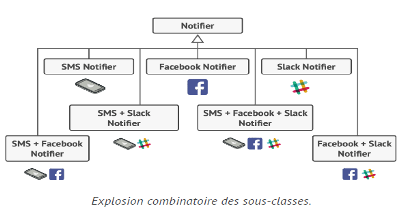
  
On peut jouer à ce petit jeu à l’infini.

Le problème est que les évaluations et les types de retours sont en durs. Cela veut dire que s’il fallait une nouvelle interprétation, il faudrait alors dupliquer le code.

**Décorateur**

Ce patron permet d’apporter de nouveaux traitements / comportements (emballeur/conteneurs/décorateurs) à un objet (emballer/concretComponent) en l’emballant dans un ou plusieurs emballeurs.

Cela permet de répondre aux limites de l’héritage car on se retrouve vite limité plus il y a de combinaison de traitements possibles.  
  
Par exemple dans un système de notification, j’aimerais pouvoir m’adapter en fonction du support :



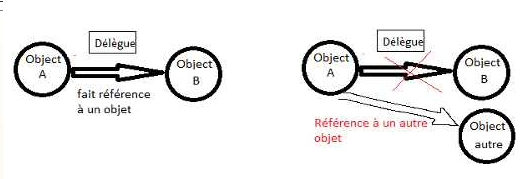
Autres problèmes de l’héritage :

-on ne peut pas modifier le comportement d'un objet au moment de l'exécution.

-l'héritage multiple n'existe pas (la plupart du temps).

La **composition** ou **agrégation**, nous permet de déléguer le travail par référence en passant par une classe abstraite : **PizzaDecorator**.

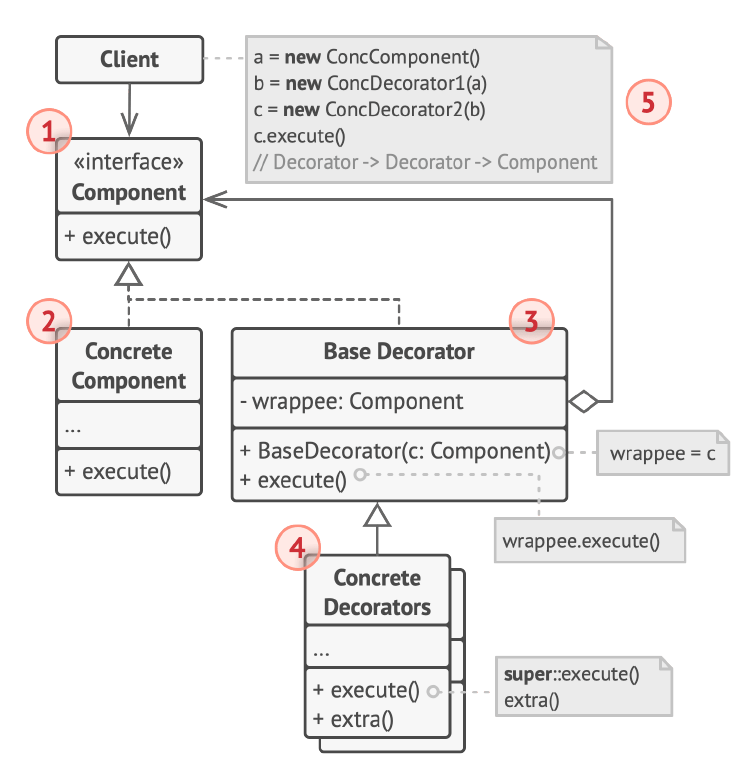
Cela donne aussi la possibilité d'agir avant ou après l'exécution à l'aide d'un **décorateur concret**.



Une analogie :

La pâte de ma pizza pourrait être une personne et les ingrédients ses habilles. Ce personnage en aurait de base et en fonction du temps extérieur, il pourrait en enfiler davantage : pullover, doudoune, écharpe, bonnet...

Schéma de base du patron décorateur :

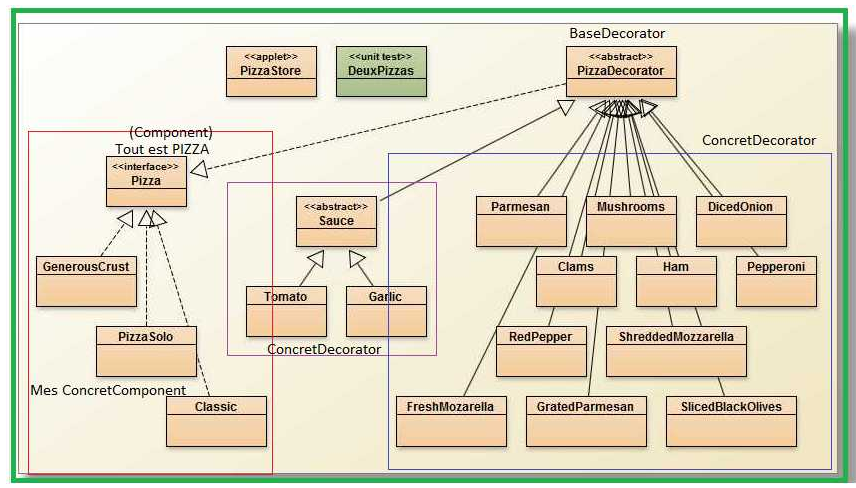


Tout est issue de l’interface **Component**. Ces méthodes sont donc communes à toutes les classes.  
La suite se divise en 2 parties :

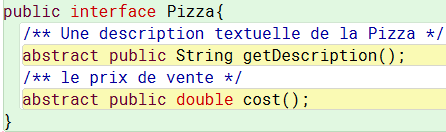
Le **ConcreteComponent** est notre objet qui sera emballé par nos différents **décorateurs**.  
Et **Base Decorator** qui peut accepter dans son constructeur un objet ou un **Concrete Decorators :**   
Elle est commune aux **decorateurs**.

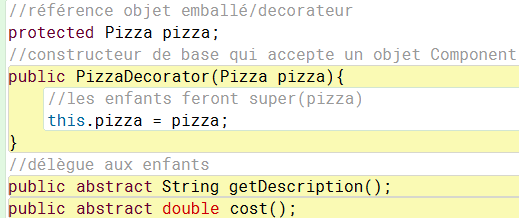
Cette classe a un *attribut* du type du **Component** pour stocker la *référence* de l’objet ou du décorateur récupéré par son *constructeur* afin de déléguer le travail aux **Concrete Decorators**.  
Les classes **Concrete Decorators** hérite de **Base Decorator** et applique un traitement sur l’objet.

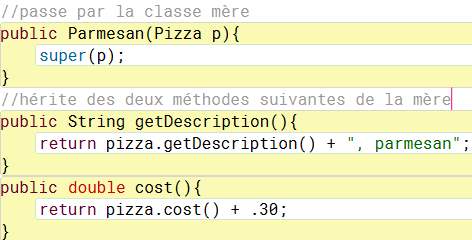
Un premier exemple concret :

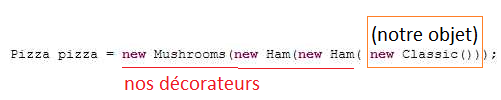


Tout est **Pizza** (Interface), ensuite à un niveau plus bas, tout est **PizzaDecorator** (abstract).



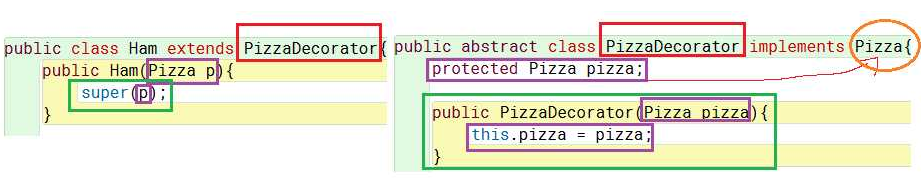
Ce décorateur de base est abstrait et contient une référence du type de mon interface pour pouvoir prendre n'importe quoi sur ce schéma.  


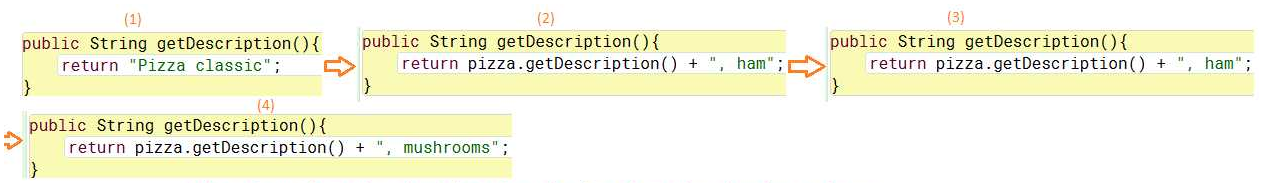
Mes **ConcretComponent** sont mes couleurs primaires, c'est-à-dire mes objets qui seront emballés par mes décorateurs. Ils s’occupent du traitement :  


On cherche agrémenter notre pâte de base avec différents ingrédients, par exemple :

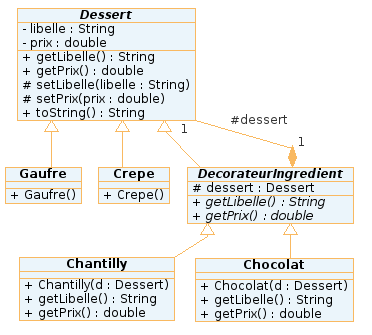
La notion d’emballement est plus lisible quand l’objet est sous la forme d’un objet et non d’une référence :  


Le résultat prendra forme de l’intérieur (notre pâte) jusqu’à l’extérieur (new Mushrooms()).



Le fonctionnement ressemble à la cumulation des résultats quand il y a récursion.

Autre schéma :

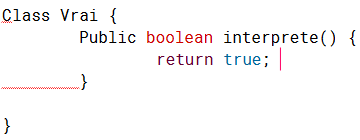


**Visiteur**

Le patron visiteur permet d’aller chercher la référence, le “this”, d’un objet rentré en paramètre.

Associé avec un composite, il permet de récupérer la référence d’un objet et d’y appliquer le traitement que l’on souhaite.

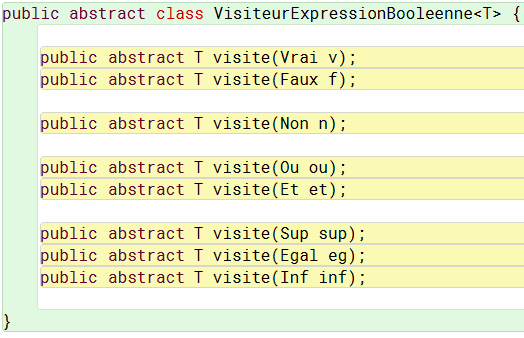
La différence avec l’interpreter s’explique par la présence de plusieurs interprétations qui ne sont pas liées aux feuilles.  
Là où une feuille qui se nomme “Vrai” devra directement retourner une valeur si on passe par un interpreter() :



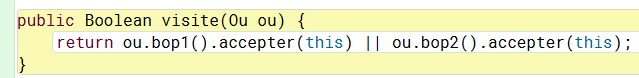
Le visiteur, lui récupèrera la référence et fera un traitement dans une ou plusieurs autres classes selon l’interprétation que l’on voudra. En effet, on pourrait très bien vouloir un boolean (true/false), un String (“vrai”/”faux”), ou encore du langage java mais en String (“true”/”false”) …

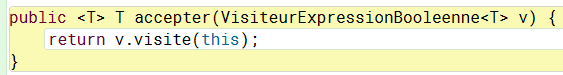
Le fonctionnement d’une visite se confond avec une ancre : faire un aller-retour.

Il faut deux méthodes qui fonctionneront dans un sens à l’aller et dans l’autre au retour.  
Le visiteur “Visite” et le Visiter accepte cette visite avant de renvoyer une référence de lui-même vers le Visiteur.

L’interface ou abstract de notre visiteur :  


Le visiteur se présent comme suit :

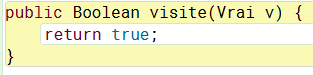
  
(**Aller**)  
visite(destination)=> destination.accepter(départ).



(**Retour**)

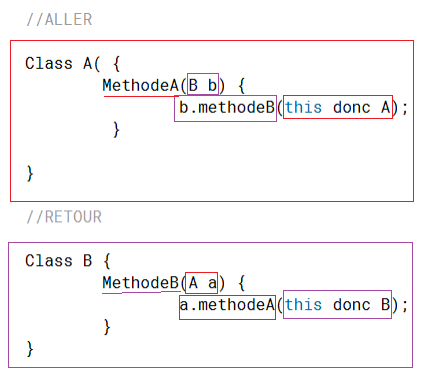
Accepter(destination)=> destination.visite(départ).

Si le visiteur renvoi directement une valeur finale :

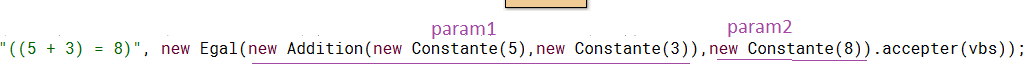


La classe de départ A aura une méthode (visite) qui permettra d’aller à un point B.  
Cette méthode possède en paramètre le point B (visite(B b))

Et le **point B** a une méthode qui prend en paramètre **A** (*accepter(A a)*) et ce qui permettra de revenir vers **A** avec la *référence* de **B**.  
On déclenche cette méthode dans la méthode de **A** sur **B** (*b.accepter(A a)*).  
On arrive dans B et avec la méthode de A et son paramètre **A**, on retourne vers **A** avec la *référence* de **B** (*a.visite(this de b*)).



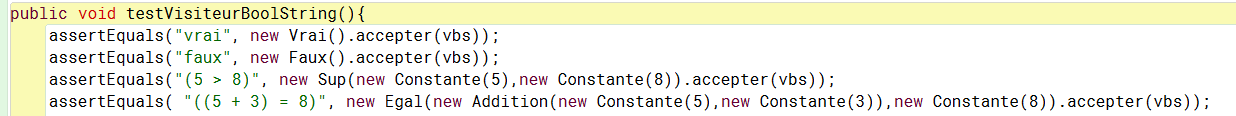
Par exemple :



A partir du visiteur vbs, je vais utiliser la méthode “accepter()” sur les différents éléments d’égal puis sur égal lui-même quand ses paramètres seront décomposés.

Pour mettre à bien se fonctionnement, il faut que notre relation entre la méthode *accepter* et *visite*, utilise des **génériques**. Comme cela, en fonction du traitement voulu, donc du visiteur, on pourra retourner ce que l’on voudra.

Autres exemples :

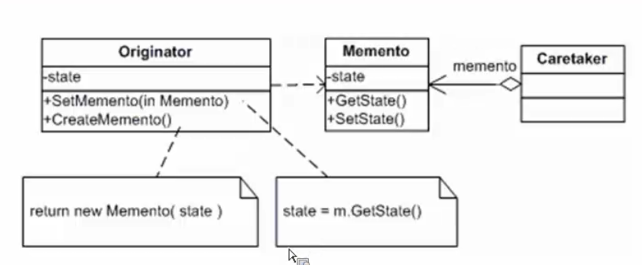


**Memento**

Ce patron permet la sauvegarde (instantané) et la restitution de l’état d’un objet via des méthodes sans violer le principe d’encapsulation et sans connaître le type.

Pour respecter ce principe, il faut suivre une règle : ne pas envahir l’espace personnel des autres.  
C’est à dire que le programme se divisera en plusieurs classes ayant un rôle bien déterminé.  
C’est logique puisqu’une classe extérieure causera des problèmes de sécurité si celle-ci avait directement accès aux informations par elle-même.

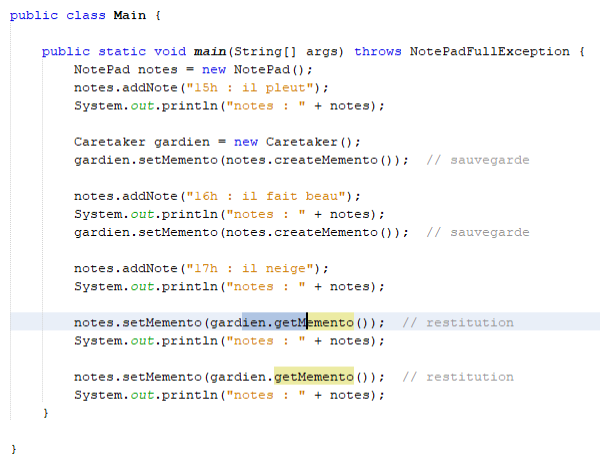
Il existe plusieurs techniques pour implémenter un Memento.  
La méthode classique est basée sur des classes imbriquées (memento dans créateur) :



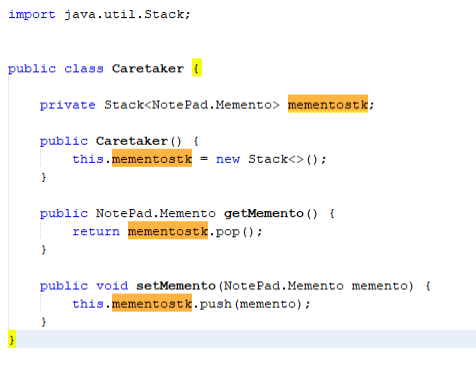
La classe dont on veut sauvegarder ou restituer l’état devra le faire elle-même (createur/originator) via deux méthodes :  
**restauration** (in) => *setMemento*(Memento in)   
**création** (out)=> *createMemento*()

Quand on sauvegarde, le **createur** instancie son attribut ***state*** à partir de son “this”.  
Quand on restaure, la méthode prend en param le memento qui changera l’attribut ***state***.  
  
Ensuite, la classe **Memento** représente cette sauvegarde, la copie de notre référence.  
Puis CareTaker contient les différents Mementos via une Collection.  
  
1) on crée un gardien  
2) on crée un créateur  
3) le créateur fait une sauvegarde : un Memento.  
4) le gardien accueil le memento dans une collection de mementos.

Dans cette implémentation, la classe mémento est imbriquée à l’intérieur du créateur. Ceci permet au créateur d’accéder aux attributs et méthodes du mémento, même s’ils sont privés. Le gardien quant à lui n’a qu’un accès limité aux attributs et méthodes du mémento : on le laisse entasser les mémentos dans la pile, mais il ne peut pas les modifier.

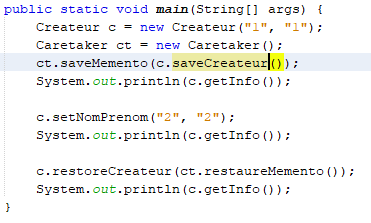


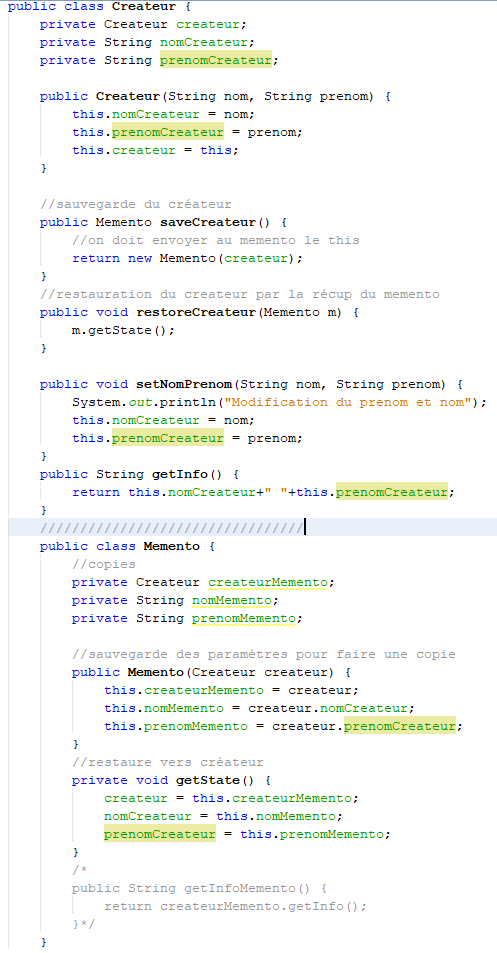


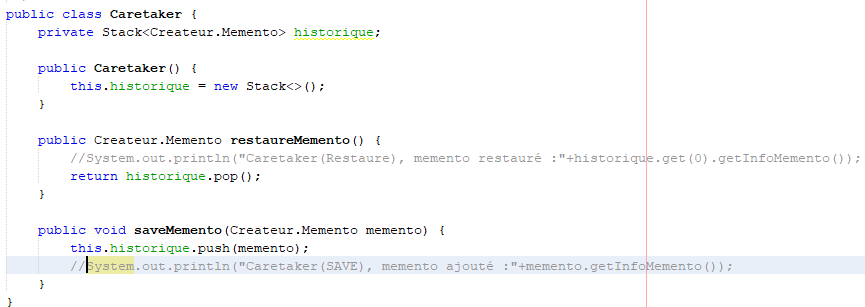


Autre exemple :

On a un créateur qui a un nom et un prénom. On souhaite sauvegarder les modifications de ces 2 attributs.



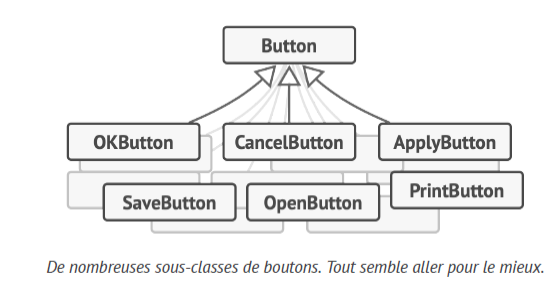




**Commande**

Le but est de prendre **une action à effectuer** et de la transformer en un **objet autonome** qui contient **tous les détails de cette action**.

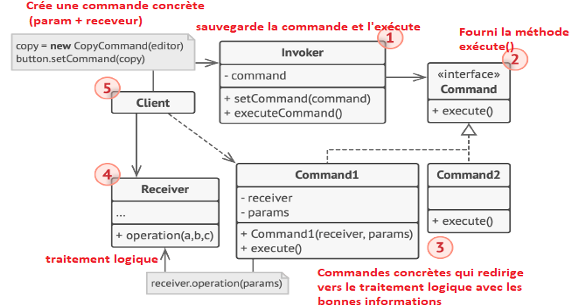
Sans patron, on pourrait avoir un fonctionnement comme celui-ci :

  
Mes sous-classes sont nombreuses et le Button ne joue plus son rôle de GUI:

On doit **séparer la logique métier** (fonctionnement) du **code de présentation** (GUI).

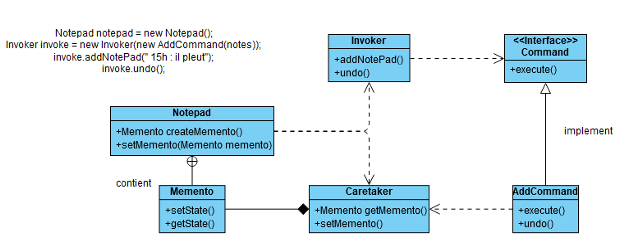
(Autre terminologie "séparation des préoccupation")

**Analogie** : le client passe commande auprès d’un serveur : il veut des sushis. Celui-ci dépose ensuite la note en cuisine avant qu’un cuisinier expert en sushi la prenne en charge.

  
Le **Client** souhaite réaliser une action au niveau d’un clic sur un bouton : copier/coller (sushis).   
Pour se faire, il crée une **Commande** à partir du constructeur de la classe concrète de la Commande à réaliser, exemple : un copier-coller. A la création, il fournit des paramètres dont celui du **Receveur**.

Le **Demandeur** récupère cette demande de commande du **Client** et l'exécute (serveur).

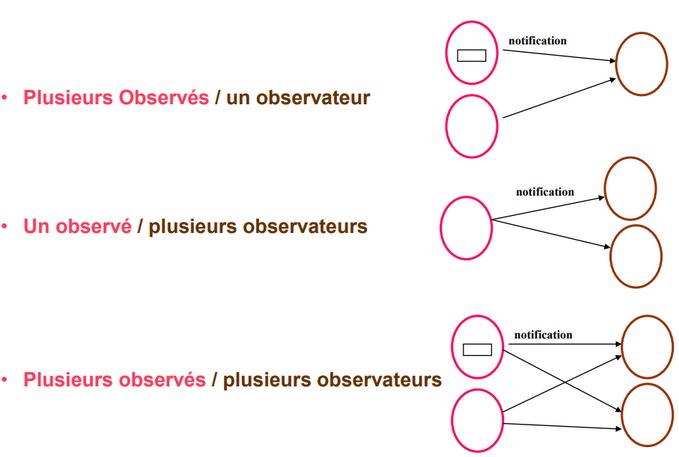
Ensuite, la **commande concrète** désignée par le **Client** lors de la création d'un objet commande réceptionne la demande (dépôt en cuisine de la note). Elle sauvegarde les paramètres et redirige le traitement de l'opération logique vers le bon **Receveur** (expert en sushi).

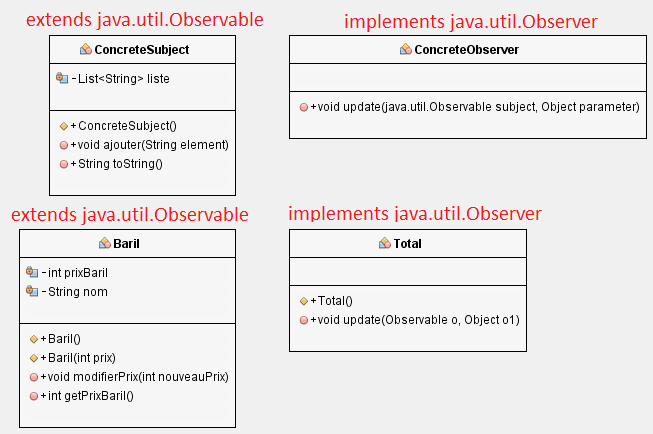
Exemple association des patrons command et memento :1) Je crée un objet **Notepad : notepad**

2) Je crée un **Demandeur** (invoker) auquel j'ajoute une **commande concrète** auquel j'ajoute mon **notePad**.  
3) La méthode **addNotePad**() lance la méthode **execute**() de l'objet **command** contenu dans Invoker:  
**addCommand**() qui lui lance **gardien.setMemento(notes.createMemento())**;

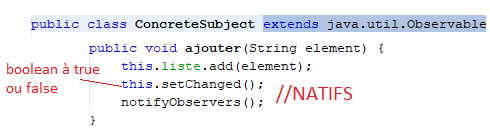
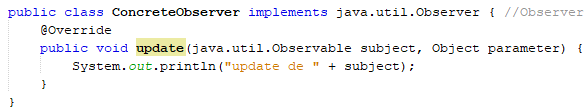
4) Pour faire un retour en arrière : **undo** d'invoker => undo() d'AddCommand => notes.setMemento(gardien.getMemento());

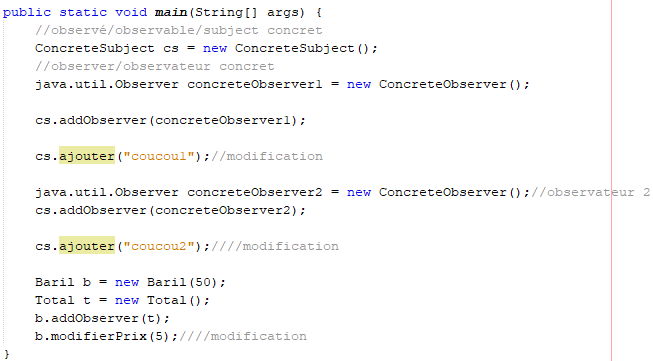
**Observateur**

Ce patron permet d’observer les changements et de prévenir quand cela arrive.  
Il y a des **Observable/subject** (*observés*) qui sont observés par un ou plusieurs **Observer** *(observateurs).*  


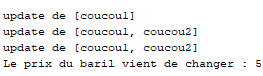
patron Observer NATIF:  


**ConcretSubject** et **Baril** sont des Subject / Observable (Observés)  
**ConcretObserver** et **Total** sont des Observer (Observateurs)

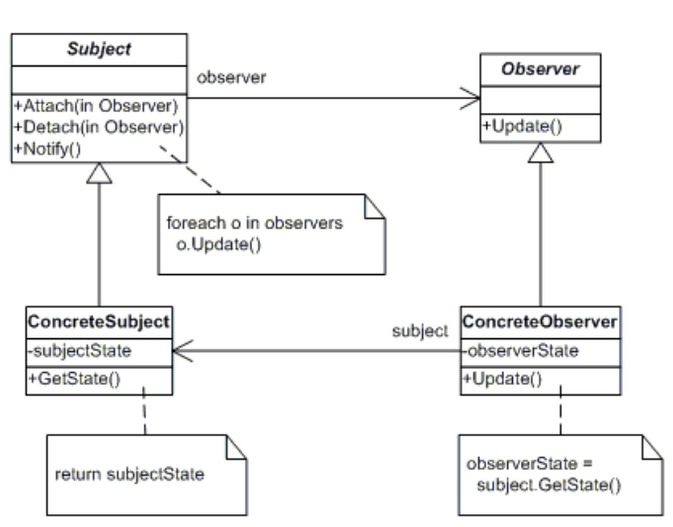
  




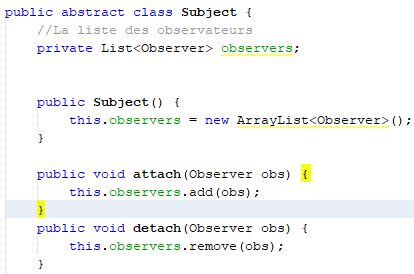
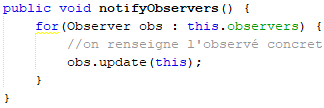
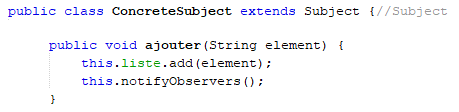
Résultat :

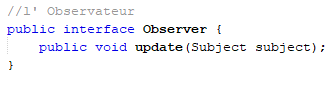
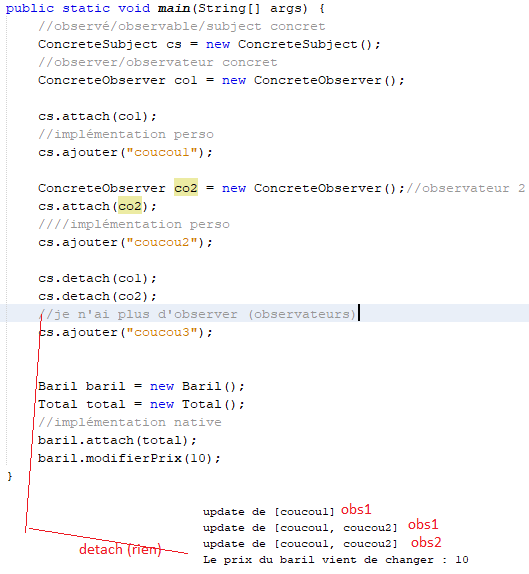


Le schéma du patron observer pas NATIF:

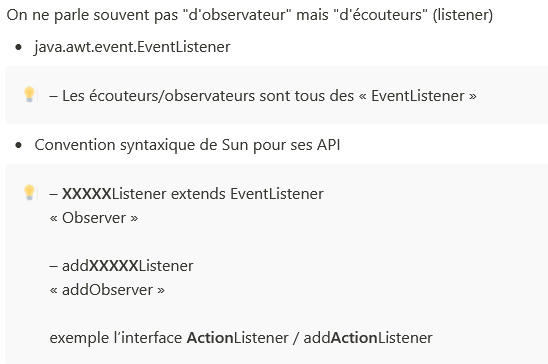


Un **Subject** (Observable, observé) est une classe abstraite qui possède une liste d’Observer (observateurs).

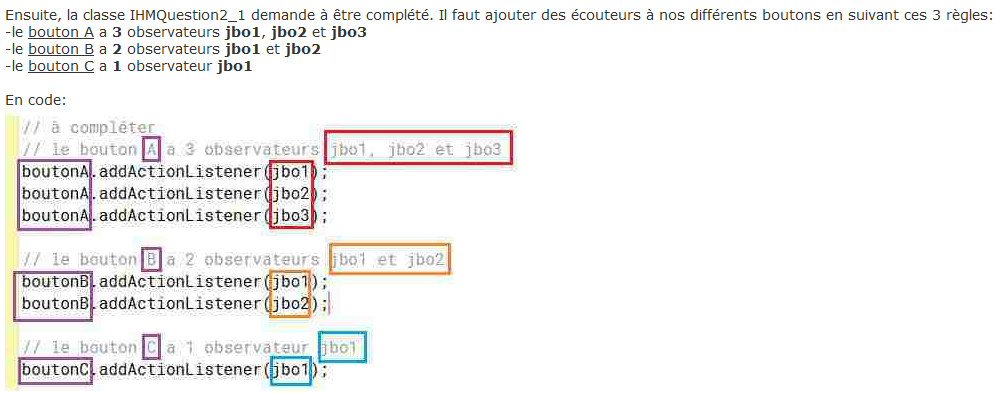
  
Mais aussi la méthode pour prévenir les Observer (observateur) d’un changement :  
  
Le **concreteSubject** hérite de **Subject.** Au niveau des méthodes qui apportent un changement que l’on veut monitorer :  


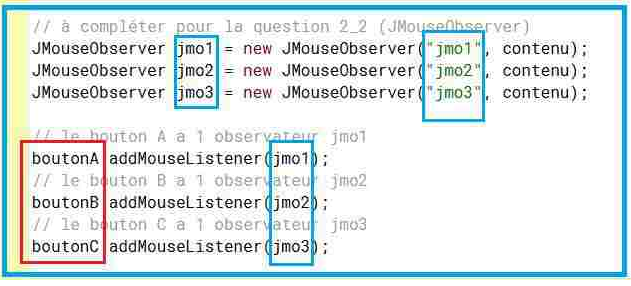
L’Interface **Observer** fourni une méthode **update**(Observer o) qui est utilisé dans la méthode **notifyObservers** de **Subject (personnelle).**   
  


Les observateurs sont utiles dans les GUIS quand il faut gérer différents boutons :

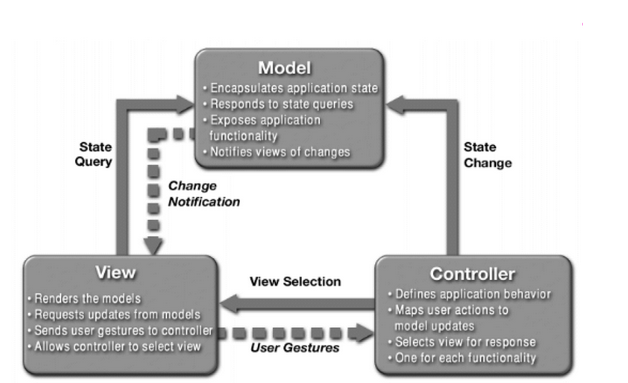
  
Nos boutons sont nos **ConcretObservable** et au lieu d’hériter d’Observable, on utilise “**javax.swing.JButton**”.  
L'interface **Observer** qui est implémentée par le **ConcreteObserver**, est remplacé par l'interface "**java.awt.event.ActionListener**".

Quant aux méthodes **addObserver** et **notifyObservers**, elles seront remplacées :  
**-addXXXXListener(ActionListener l) =**> add**Action**Listener(ActionListener l)  
**-actionPerformed(ActionEvent ae)**

Exemple :Autre exemple, quand l’action n’est plus un clic mais le survol de notre souris :  

**MVC (Model-View-Controller)**

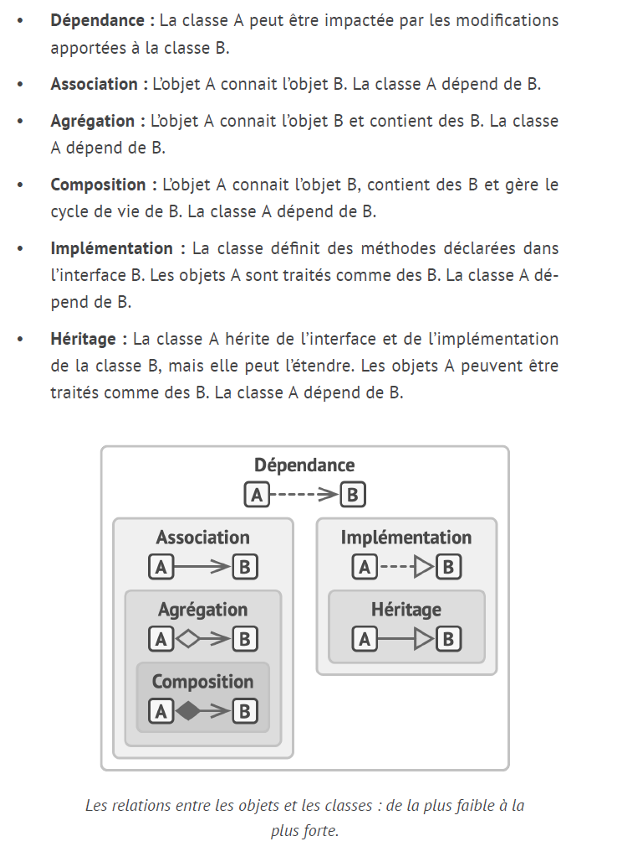


* Le **Modèle** (Observ**é**/Observable/Subject) contient la logique et l'état de l'application, il prévient ses observateurs lors d'un changement d'état.
* La **Vue** représente l'interface utilisateur, elle sert d’Observer (Observ**er**/Observateur)
* Le **Contrôleur** assure la synchronisation entre la vue et le modèle.

In a MVC (Model View Controller), the Controller provides the communication between the Model and View objects. The Controller may be a separate class or within the model or view.

Mon **Modèle** ajoute donc une **Vue** qui l’observe. Quand le modèle change, ses Vues sont prévenues.  
Le **Modèle** contiendra le fonctionnement de l’application (donc il aura NotifyObservers et setChanged).  
La **Vue** contiendra les différents boutons (extends JFrame + implements Observer)  
Le **Contrôleur** sera l’intermédiaire entre le Modèle et la Vue (actionPerformed et addActionListener) (implements ActionListener).

**UML**



**HASHMAP (Clef unique = pas homonyme)**

import java.util.Map;

import java.util.HashMap;

import java.util.List;

import java.util.ArrayList;

**Déclarer un hashmap:**

Map<Type1, Type2> map = new Hashmap<>(); //diamant <>

**Hashmap qui contient un autre tableau :**

Map<String, List<String>> map = new Hashmap<> () ;

**Comment vérifier si une clé existe ?**

If(map.containsKey (“nomClef”)) {…  
Où inversement **(!)** :  
If (**!** map.containsKey (“nomClef”)) {…

**Quand on rajoute une clé :**

map.put(“maClef”, new ArrayList<> ()) ;

**Récupérer la valeur d’une clé :**

map.get(“nomClef”) ;

**Comment vider un hashmap ou la valeur liée à une clef :**

map.clear() ;  
map.get(“maClef”).clear() ;

**Quand on veut rajouter une valeur à notre tableau pour une clé :**  
**//vérifie si la clef existe**  
 if (map.containsKey(“maClef”)) {

**//récupère la valeur (copie) liée à la clef**

ArrayList<String> copieArr = map.get(“maClef”) ;

**//ajoute la nouvelle valeur à notre copie**

copieArr.add(“une valeur”) ;

**//attribut notre copie comme valeur pour notre clé dans le map**

map.put(“maClef”, copieArr) ;

}

**Comment parcourir les clefs d'un hashmap :**  
for ( String key : map.keySet() ) {

System.out.println( key );

}

**Comment récupérer valeurs/clefs :**

HashMap<String, ArrayList<String>> map;

map = new HashMap<> () ;

map.put("abcd", new ArrayList<> ()) ;

map.get("abcd").add("10");map.get("abcd").add("20") ;

for(HashMap.Entry<String, ArrayList<String>> entry : map.entrySet()) {

String key = entry.getKey() ;

ArrayList<String> value = entry.getValue() ;

**//Key : abcd Value : [10, 20]**

System.out.println("Key : "+key+" Value : "+value);

}

**Comment faire un iterator à partir du tableau précédent :**

//HashMap<String, ArrayList<String>> map;

public Iterator<String> iterator() {

return new TreeSet(boiteMail.keySet()).iterator() ;

}

**Parcourir le contenu de ma valeur (exemple) :**

ArrayList<String> copieArr = map.get(“maClef”) ;  
for(String value : copieArr) {  
 System.out.println(value);  
}

**Reflexion**

**getDeclaredMethods()** affiche les méthodes déclarées par la classe elle-même (private','protected' et'default).  
**getMethods()** affiche toutes les méthodes publiques y compris celles hérités (interface inclus).

**Récupérer une classe à partir de son nom :**

Class< ? > cl = Class.forName("nonClasse”) ;

**Récupérer et instancier une classe avec son constructeur par défaut :**

Object newInstance = cl.newInstance() ;  
this.instanceSpecification = (SpecificationI) newInstance ;

**Récupérer les paramètres d’une méthode :**

String target = "isSatisfiedBy";

for(Method m : instanceSpecification.getClass().getMethods()) {

if(target.equals(m.getName())) {

//récupère son paramètre

params = m.getParameterTypes();

}

}

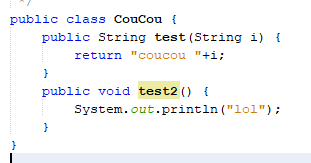
**Récupérer dans un tableau les méthodes publiques d’une classe :**

Class<?> maClasse = Class.forName("nonClasse”) ;  
Method[] methods = maClasse.getMethods();

Récupérer une méthode si on connait son nom + paramètre :

Class<?> maClasse = Class.forName("nonClasse”) ;  
Method method = maClasse.getMethod("nomMethode", String.class);

**Récupérer et instancier des méthodes quand on connait nom+param :**



Class<?> maClasse = Class.forName("Main.CouCou");

Method m = maClasse.getMethod("test", String.class);**//méthode test avec param**

**//public java.lang.String Main.CouCou.test(java.lang.String)**

System.out.println(m);

Object newInstance = maClasse.newInstance() ;   
 **//m = méthode; newInstance = “Coucou c = new Coucou()” ; c.methode(“param”)**

Object returnValue = m.invoke(newInstance, "parameter-value1");

System.out.println(returnValue);**//coucou parameter-value1**

Method m2 = maClasse.getMethod("test2", null);**//méthode test2 sans param**

**//public void Main.CouCou.test2()**

System.out.println(m2);

m2.invoke(newInstance, null);

**Invoquer une méthode static :**

public static void methStatic() {

System.out.println("coucou");

}

Class<?> maClasse2 = Class.forName("Main.Main");

Method method = maClasse2.getMethod("methStatic", null); //pas de param

method.invoke(null); //static donc null

Récupérer le type de retour d’une méthode :

Class returnType = m.getReturnType();

System.out.println(returnType);**//class java.lang.String**

**Serialization**

[**http://blog.paumard.org/cours/java/chap10-entrees-sorties-serialization.html**](http://blog.paumard.org/cours/java/chap10-entrees-sorties-serialization.html)[**https://www.jmdoudoux.fr/java/dej/chap-serialisation.htm#serialisation-1**](https://www.jmdoudoux.fr/java/dej/chap-serialisation.htm#serialisation-1)

Ce procédé permet de rendre un objet **persistant** pour stockage, échange et vice versa.  
On fait une copie à l’identique d’un objet que l’on pourra stocker sur un disque dur ou sur le réseau.

Java propose ce mécanisme nativement pour différents besoins comme ;

* rendre persistants des objets (dans un fichier ou une base de données par exemple)
* créer une copie intégrale d'un objet (deep copy)

Cette technique s’utilise principalement quand la durée est courte (persistance temporaire, échange réseau...).  
Les éléments qui auront "**transient” ne seront pas récupérés.**

**Méthode avec des objets** :

-Il faut que la classe implémente Serializable

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(new File(fileName)));  
oos.writeObject(mb);  
oos.close();

ObjectInputStream ois = new ObjectInputStream(

new FileInputStream(new File(fileName)));

IMailBox mb = (IMailBox)ois.readObject();

return mb;

**XML**

**Qu-est-ce-que XML (Markup Language) ?**C’est un langage de balisage pour stocker et transmettre des informations. Celui-ci suit des normes d’écriture (grammaire DTD) qui peuvent changer en fonction de son utilisation.

Un fichier XML commence par un prologue :  
*<?xml version="1.0" encoding="UTF-8"?>*

Peut avoir des commentaires :   
*<!-- blabla -->*

**Element**: balise



**Attributs :** une balise peut avoir des attributs  
  
**NE** et **MORT** : *clefs* ; Elle on des *valeurs*.

**Racine :** un document XML contient une balise dite « racine » qui englobe toutes les autres et qui est unique.

Une balise peut-être **vide** (balise **auto-fermante** : *<balise****/>***) ou **non-vide** (balise **ouvrante** suivis d’une balise fermante : *<balise>…</balise>*).

**Donnée** : Ce qui contient un *élément (cf : Victor Hugo)*

**Qu-est-ce-que JDOM (J** comme Java**) ?**

API JAVA de manipulation des données XML. JDOM n’étant pas un parseur, il utilise soit **SAX** pour *parser* (analyser/parcourir) un fichier XML (JDOM ->SAX->ParserXML) ou bien **DOM** qui possède plus de fonctionnalités**.**

**SAX** utilise un modèle de traitement « *par événements* » et ne permet pas de modifier   
ni naviguer comme on le souhaite. Cet outil permet de lire une portion ou la totalité d’un fichier **XML** dans l’ordre.   
On y traite un document *élément par élément, au fur et à mesure*. Alors que **DOM** charge l’intégralité d’un document XML dans une structure de données qui peut être manipulée puis reconvertie en XML.

Par conséquent SAX permet uniquement de « **Lire au fur et à mesure** ».  
DOM de faire un peu ce que l’on veut car il y a le chargement de la totalité du fichier XML dans un arbre.  
Et JDOM de faciliter la « **manipulation** » de ces données XML en utilisant SAX ou DOM dans un contexte JAVA.

**JDOM** utilise **DOM** pour manipuler les éléments d’un DOM spécifique.

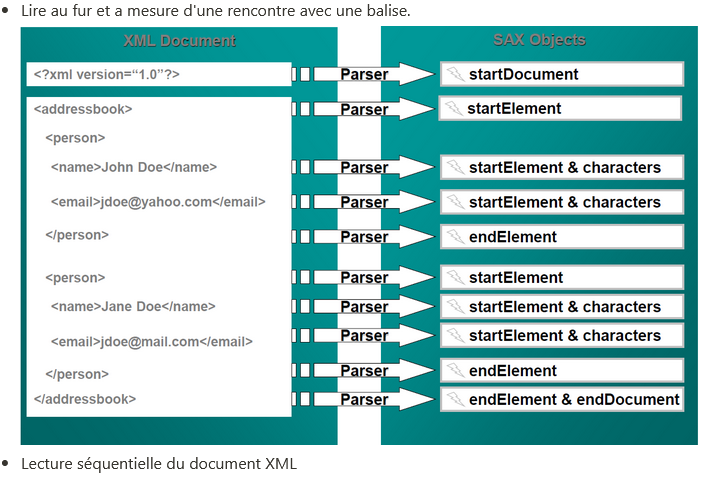
Comme **DOM**, **JDOM** encapsule un document XML dans un « arbre d’objets » sauf que lui fonctionne à partir de « classes ».

Utiliser **DOM** à la place de **JDOM** permet d’encapsuler un fichier XML de manière « universaliste » (pas propre à Java).

JDOM propose plusieurs fonctionnalités :

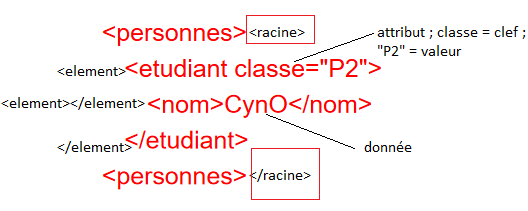
* Création, navigation, modification ou suppression de contenu d’un documents XML
* Encapsulation d’un document XML sous la forme d’un objets Java
* Exportation d’un document dans un fichier, un flux SAX ou un arbre DOM.

**SAX** :



**Créer un fichier XML avec JDOM :**

Exemple de fichier XML :


Afficher notre arborescence :

**try**

{

//On utilise ici un affichage classique avec getPrettyFormat()

XMLOutputter sortie = **new** XMLOutputter(Format.*getPrettyFormat*());

sortie.output(*document*, System.*out*);

}

**catch** (java.io.IOException e){}

}

Enregistrer notre arborescence :

**try**

{  
String nomFichier =  «….» ;

File fichier = new File(nomFichier) ;

//On utilise ici un affichage classique avec getPrettyFormat()

XMLOutputter sortie = **new** XMLOutputter(Format.*getPrettyFormat*());

//Remarquez qu'il suffit simplement de créer une instance de FileOutputStream

//avec en argument le nom du fichier pour effectuer la sérialisation.

sortie.output(*document*, **new** FileOutputStream(fichier));

}

**catch** (java.io.IOException e){}

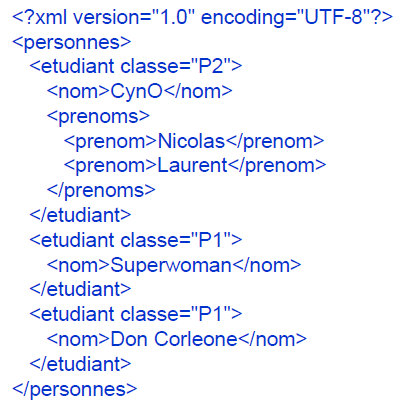
}

Parcourir un fichier JDOM :

On transforme un fichier XML en une arborescence JDOM.

Comme il n’y a aucune modification à appliquer, on utilise SAX (**SAXBuilder**).

Exemple de fichier XML :



**public static void** main(

//On crée une instance de SAXBuilder

SAXBuilder sxb = **new** SAXBuilder();

Document *document* = null ;

Element *racine* = null ;

**try**

{

//On crée un nouveau document JDOM avec en argument le fichier XML

//Le parsing est terminé ;)

*document* = sxb.build(**new** File("Exercice2.xml"));

System.out.println(« XML chargé correctement ») ;

racine = document.getRootElement() ;

}

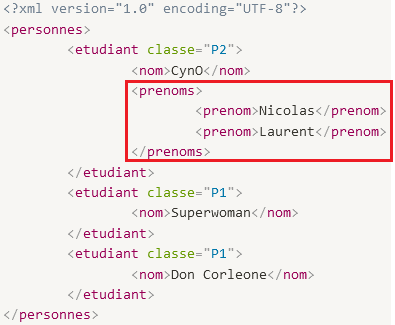
**catch**(Exception e){

System.out.println(« Erreur chargement XML ») ;

}

Modifier un fichier XML avec JDOM :

Je souhaite supprimer les balises/éléments « **prenoms** » :

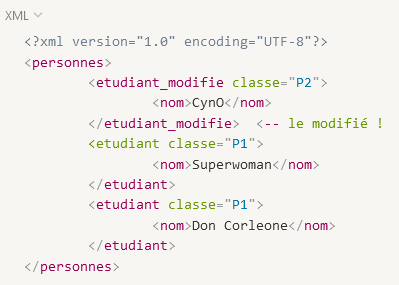


Dans un premier temps, on lit le fichier.  
Ensuite, on récupère les éléments étudiant dans un tableau.

On parcours se tableau et on supprime dès que l’on trouve « **prénoms** ».

On enregistre les modifications dans le fichier.



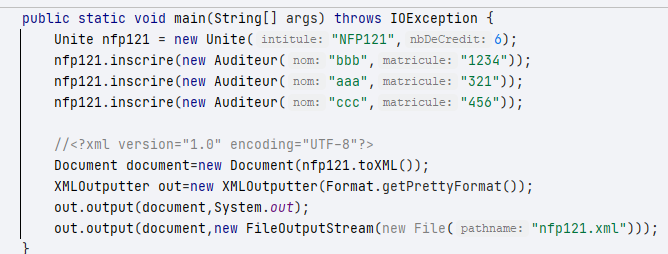


Passer de JDOM à XML  exemple :

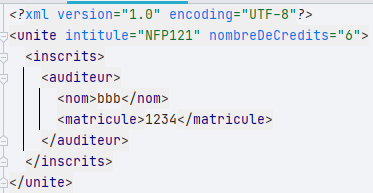


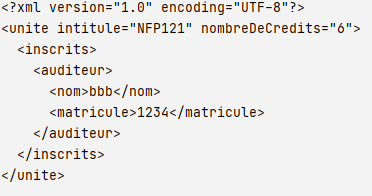
(/ ! \ : il faut implémenter Comparable<nomClasse>)





Résultat :





On crée dans un premier temps les balises <unite> et <inscrits>.(1er toXML)  
Ensuite, on crée nos balises <auditeur> (2ème toXML) .

**INJECTION DE DEPENDANCE** (IDD)

Qu’est-ce qu’une **dépendance** ?

Celle-ci est liée au couplage qui est un moyen de mesurer le niveau d’interaction entre deux ou plusieurs composants logiciels (classe, …).

Plus la dépendance est **forte** plus il y a d’échangent d’information. A contrario, il y a le couplage **faible**.  
  
**Résultat** : maintenance compliqué, composant difficilement réutilisable, interblocage…

L’**injection de dépendance** consiste à **injecter dynamiquement les dépendances**.  
Ainsi ces dépendances entre les classes qui implémentent et les interfaces ne sont plus exprimées dans le code de manière **statique** mais déterminées **dynamiquement**.  
Cela peut passer par un **fichier de configuration** (cf femtocontainer).

Il existe différents types d’IDD :

* Injection par **interface**
* Injection par **constructeur**
* Injection par **mutateur**
* Injection par **champs**

Exemple **dépendance directe (ou composition):**

La classe A possède un attribut du type de la classe B.

public class A {  
 private B b;  
  
 public A() {  
 this.b = new B();  
 }  
  
 private void callBdoSomething() {  
 this.b.bDoSomething();  
 }  
}

public class B {  
 public B(){}  
 protected void bDoSomething(){}  
}

**Inconvénients** : solution statique et disperse les dépendances (A dépend de B et inversement).

Exemple **dépendance par interface :**

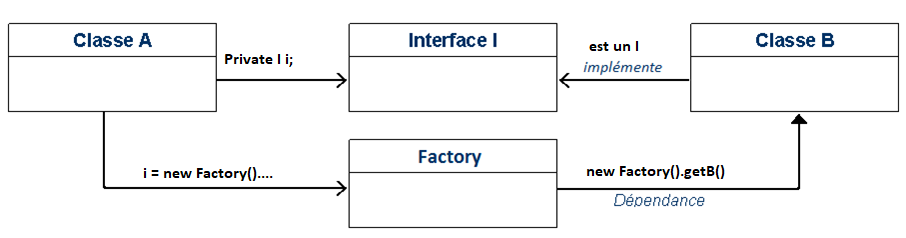
A prend un attribut du type de l’interface. Comme B l’implémente, cela marche.

public class A{  
 private I i;  
  
 public A() {  
 this.i = new B();  
 }

public class B implements I{  
 public B() {}  
}

**Inconvénients** : La dépendance est toujours là (this.i = **new B()**)

Exemple **dépendance par l’utilisation d’un Design pattern « Factory » :**

****

public class A {  
 private I i;  
  
 public A() {  
 this.i = new Factory().getDependance();  
 i.doSomething();  
 }  
}

public class B implements I {  
 public B(){}  
 public void doSomething() {System.*out*.println("coucou");}  
}

public class Factory {  
 public Factory(){}  
  
 public I getDependance() {  
 return new B();  
 }  
}

public interface I {  
 void doSomething();  
}

Factory est notre intermédiaire. L’inconvénient ici est que « new B() » est écrit en dur.

Exemple **qui ressemble à factory :**

public class A {  
 private I i;  
  
 public A(I i) {  
 this.i = i;  
 i.doSomething();  
 }  
}

public class Assembleur {  
 public Assembleur() {  
 I reference = new B();  
 A a = new A(reference);  
 }  
}

public interface I {  
 public void doSomething();  
}

On est toujours dépendant de new B().

C’est-à-dire que si jamais je veux un autre type de classe, je dois changer le code.

Comment ne plus être dépendant dans le code ?

C’est là qu’intervient l’IDD par **fichier de configuration**.

Un exemple simple :

public class A {  
 private I i;  
  
 public A(I i) {  
 this.i = i;  
 i.doSomething();  
 }  
}

import java.io.FileReader;  
import java.io.Reader;  
import java.util.Properties;  
  
public class Assembleur {  
 public Assembleur() throws Exception {  
 Properties props = new Properties();  
 Reader reader = new FileReader("src/dependance\_avec\_fichierConf/assembleur.props");  
 props.load(reader);  
 String className = props.getProperty("implementation\_i\_class");  
 Class<?> c = Class.forName(className);  
  
 I reference = (I) c.newInstance();  
 A a = new A(reference);  
 }  
}

assembleur.props : **implementation\_i\_class**=**dependance\_avec\_fichierConf.B1**

Notre code ne fait référence à aucun type car celui-ci se trouve à l’extérieur.

Exemple d’une **injection par interface** :

public interface Injecter {  
 void inject(I i);  
}

public class A implements Injecter{  
 private I i;  
  
 public A() {}  
  
 @Override  
 public void inject(I i) {  
 this.i = i;  
 this.i.doSomething();  
 }  
}

public class Assembleur {  
 public Assembleur() {  
 A a = new A();  
 a.inject(new B1());  
 }  
}

Il est possible de coupler cette solution avec un fichier config :

public class Assembleur {  
 public Assembleur() throws Exception {  
 Properties props = new Properties();  
 Reader reader = new FileReader("src/injection\_par\_interface/assembleur.props");  
 props.load(reader);  
 String className = props.getProperty("implementation\_i\_class");  
 Class<?> c = Class.forName(className);  
  
 A a = new A();  
 I reference = (I) c.newInstance();  
 a.inject(reference);  
 }  
}

**RULE et ses patrons (spécification…)**

Le **patron spécification**

Un programme doit suivre des règles pour fonctionner. Mais certaines peuvent changer dans le temps. La maintenabilité peut donc se détériorer due à un couplage croissant.  
Ce patron joue le rôle d’un **moteur de règle**.

Exemple d’un programme mêlant **composite** + **spécification**:

Le composite permet de **cumuler** les **spécifications** (emboitement de constructeur) car tout est l’interface **SpecificationI<E>** et les spécifications sont nos **feuilles**.

**Rappel** : cela permet de créer des combinaisons dynamiquement (choix de l’utilisateur).

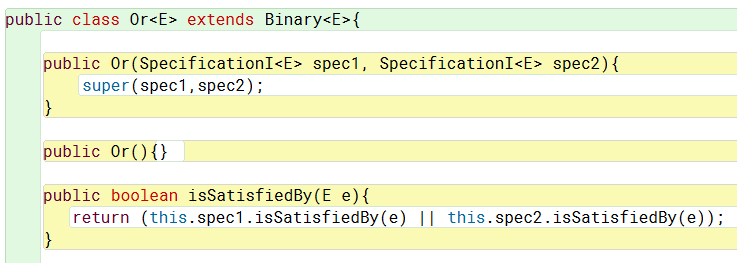
Nos feuilles implémentent 2 méthodes :

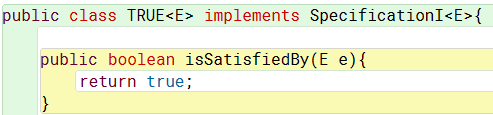
* Public abstract boolean **isSatisfiedBy**(E e)
* Public abstract String **toString()**

La première est la **condition que doit satisfaire** notre classe et la deuxième joue le rôle **d’interpreter**.

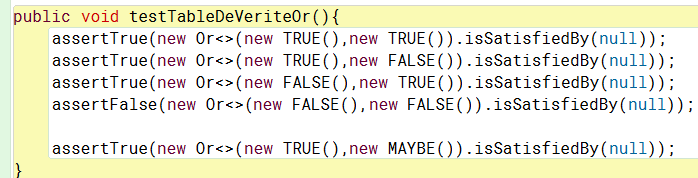
Notre composite est constitué de **feuilles terminales** (TRUE, FALSE et MAYBE) qui ont une methode **isSatisfiedBy** qui renvoi les valeurs logiques true, false ou l’un ou l’autre.  
  
Ensuite, les autres classes sont des règles à respecter qui implémentent elles aussi leurs **isSatisfiedBy.** Ces classes ont 2 attributs **Specifications** qui sont utilisés par leurs isSatisfiedBy pour faire appel aux isSatisfiedBy de ces attributs.

Or forme une règle à partir de 2 attributs ayant une valeur logique true ou false.





Application :



Autre règle :

Je souhaite n’avoir aucune true :

