PATRONS

CREATIONNELS

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

STRUCTURELS

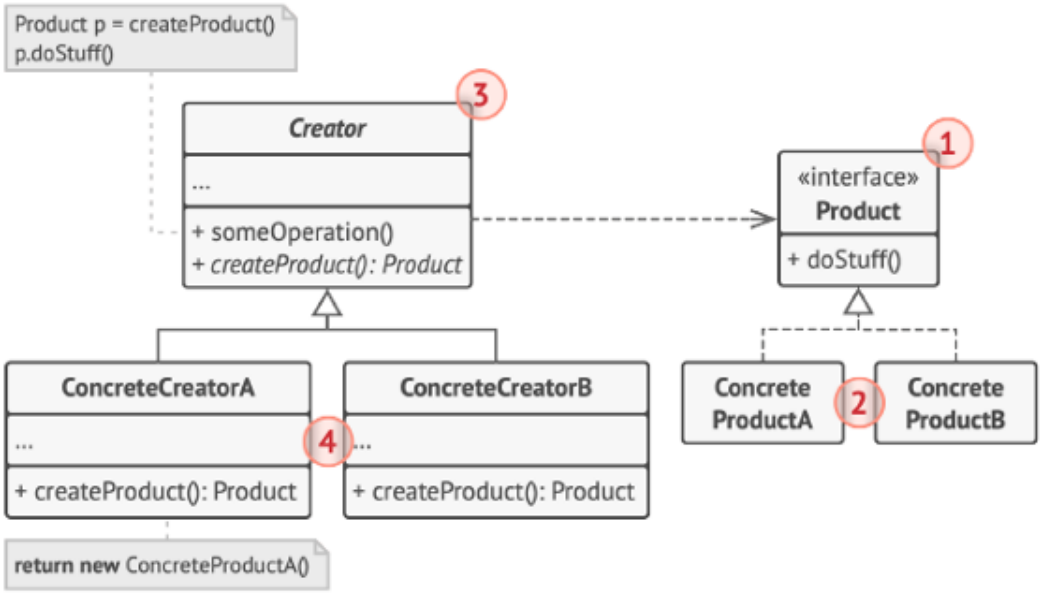
- Procuration (*Proxy*)
- Adaptateur (*Wrapper, Adapter*)
- Composite (*Arbre d'objets*)
- Décorateur (*Decorator, emballeur, wrapper)*

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**).

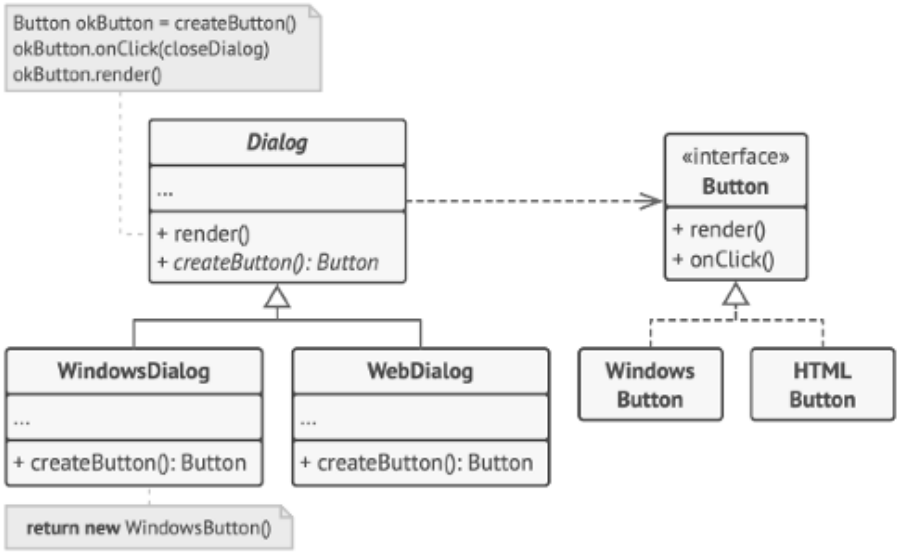
Par conséquent, la classe exacte de l'objet n'est pas connue par l'appelant (Type Product).

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).



Exemple multiplateforme pour une boîte de dialogue.

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

<u>Exemple wikipedia</u> :

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

```
public class FabriqueAnimal {

  Animal create(String typeAnimal) throws AnimalCreationException {
    if(typeAnimal.equals("chat")) {
      return new Chat();
    }
    else if(typeAnimal.equals("chien")) {
      return new Chien();
    }
    throw new AnimalCreationException("Impossible de créer un " + typeAnimal);
  }
}
```

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

```
Button okButton = createButton()
okButton.onClick(closeDialog)
okButton.render()
```

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

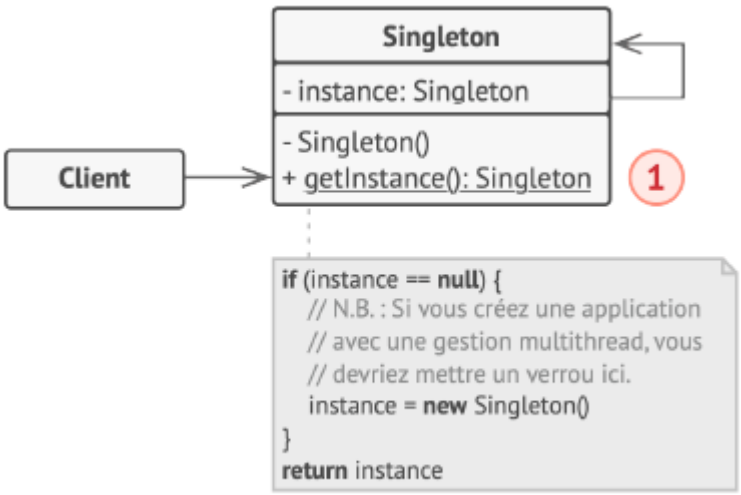
SINGLETON

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.

```java
public final class Singleton {
    private static Singleton instance;
    public String value;

    private Singleton(String value) {
        // The following code emulates slow initialization.
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        this.value = value;
    }

    public static Singleton getInstance(String value) {
        if (instance == null) {
            instance = new Singleton(value);
        }
        return instance;
    }
}
```

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.

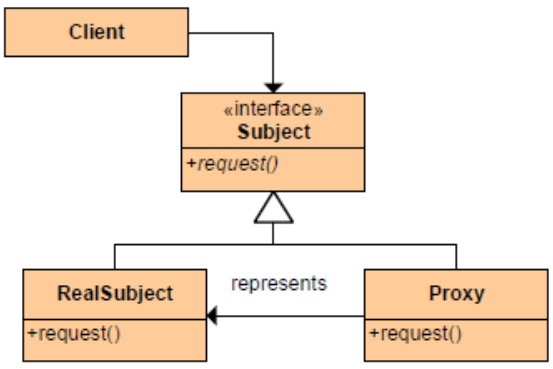
Si **P** alors *Requête traitée* => **B**

Sinon **P** retourne une erreur.

Proxy

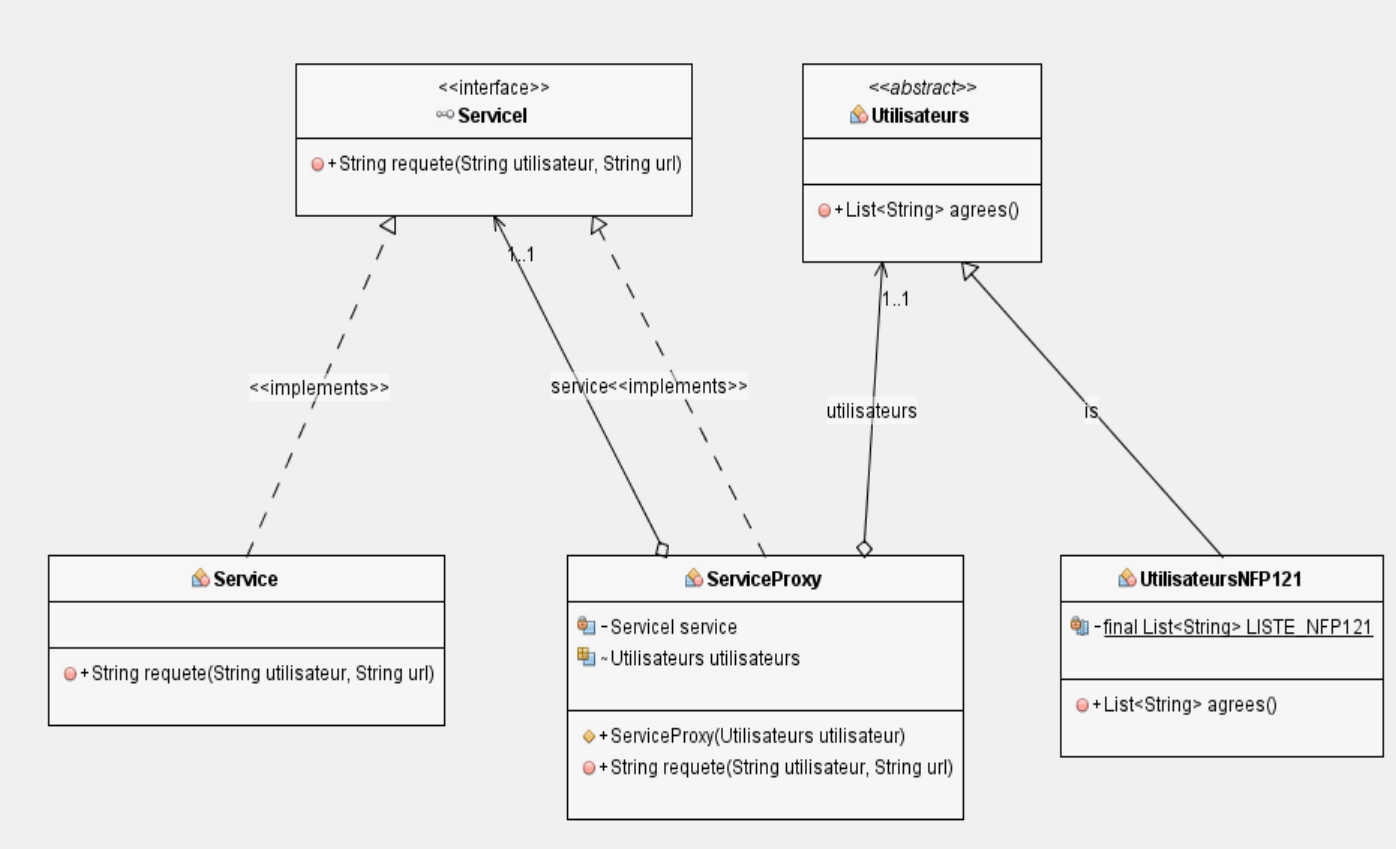
Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.



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 :

```
public class ServiceProxy implements ServiceI{
    private ServiceI service;
    Utilisateurs utilisateurs;

    public ServiceProxy(Utilisateurs utilisateur) {
        this.service = new Service();
        this.utilisateurs = utilisateur;
    }
    @Override
    public String requete(String utilisateur, String url) {
        if(this.utilisateurs.agrees().contains(utilisateur)) {
            return service.requete(utilisateur, url);
        }else {
            return "utilisateur "+utilisateur+" non agrée, requête rejetée";
        }
    }
}
```

vers RealSubject

Traitement erreur en local

Le RealSubject, Service :

```
public class Service implements ServiceI{
    @Override
    public String requete(String utilisateur, String url) {
        return ("requête de "+utilisateur+" sur l'URL "+url);
    }
}
```

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.

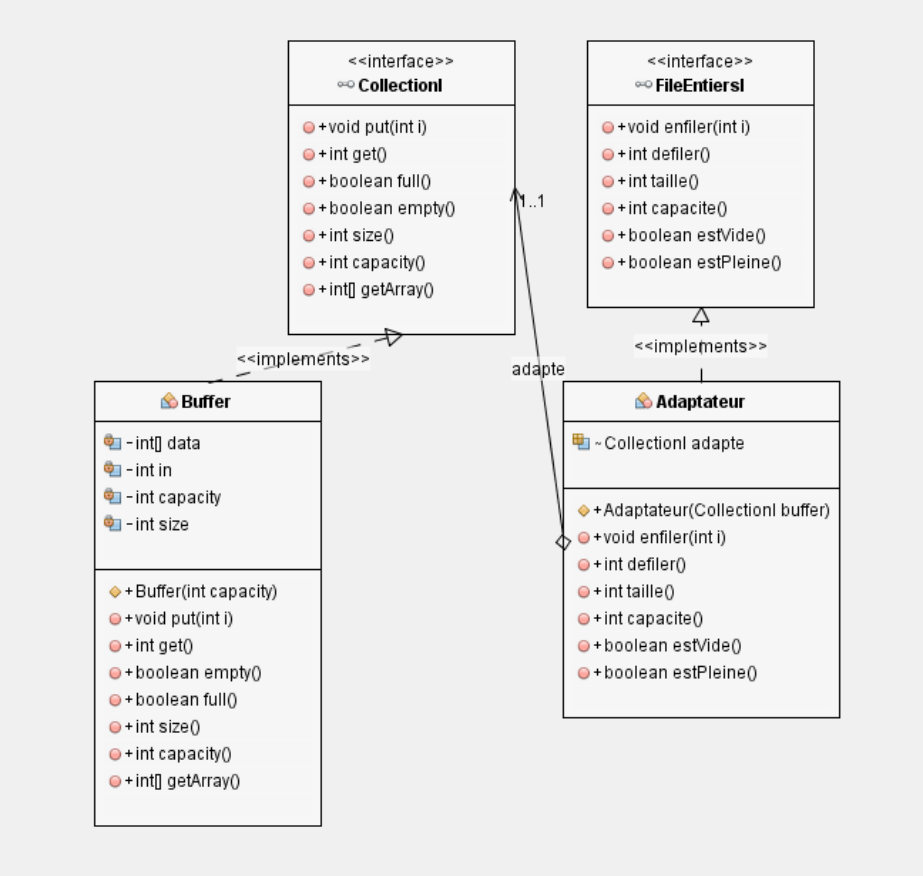
L'Adaptateur est une classe qui va implémenter l'interface vers laquelle on veut récupérer les méthodes (Prise) (Convertir / déléguer les méthodes de A dans B)

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.

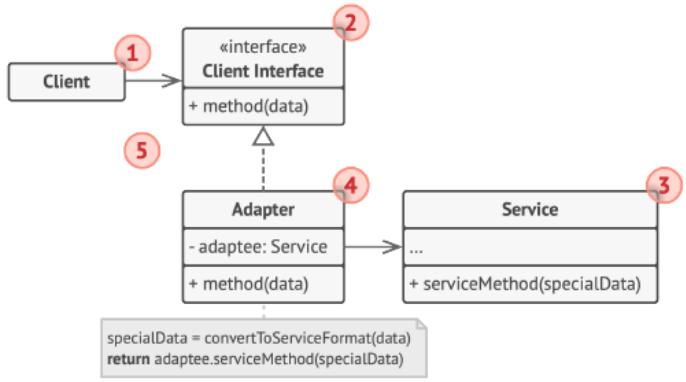
Aide à la compréhension :

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

```
Class C implements Bi {
   A a;
   C (A a) {a = a;}
   Func1B() {
      a.func1A();
   }
}
```

```
                //A           vers      B
                //CollectionI vers FileEntierI
public class Adaptateur implements FileEntiersI{
    CollectionI adapte;//A

    public Adaptateur(CollectionI buffer) {
        this.adapte = buffer;//A
    }

    @Override
    //B avec A dedans
    public void enfiler(int i) throws FilePleineException {
        try {
            adapte.put(i);
        } catch (FullException ex) {
            throw new FilePleineException();
        }
    }
```

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.

```
private String xmlToJson(String xmlString) {
    int PRETTY_PRINT_INDENT_FACTOR = 4;
    try {
        JSONObject xmlJSONObj = XML.toJSONObject(xmlString);
        String jsonPrettyPrintString = xmlJSONObj.toString(PRETTY_PRINT_INDENT_FACTOR);
        return jsonPrettyPrintString;
        //System.out.println(jsonPrettyPrintString);
    } catch (JSONException je) {
        System.out.println(je.toString());
    }
   return "";
}
```

COMPOSITE

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

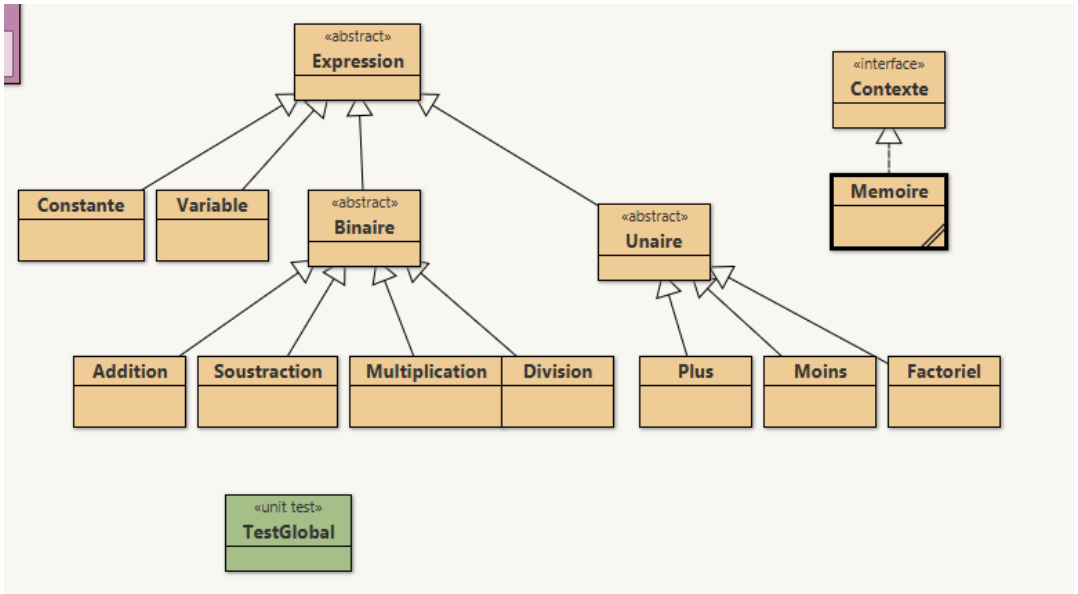
Ce-dernier est un patron de conception structurelle qui permet d'agencer les objets dans des **arborescences** afin de pouvoir traiter celles-ci comme des objets individuels.

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 :

```
Expression expr = new Addition(new Constante(3), new Constante(2));
```

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

```
public abstract class Expression {
   public abstract int interprete(Contexte c);
}
```

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 :

```
public Variable(Contexte c, String nom, Integer valeur) {
        this.nom = nom;
        c.ecrire(nom, valeur);
}
public int interprete(Contexte c) {
        return c.lire(this.nom);
}
```

Constante :

```
public int interprete(Contexte c) {
    return valeur();
}
```

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

```
public class Memoire implements Contexte {
    private Map<String, Integer> table;

    public Memoire() {
        table = new TreeMap<String, Integer>();
    }
```

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

```
public Integer lire(String adresse) {
    Integer valeur = table.get(adresse);
    if (valeur == null) throw new RuntimeException();
    return valeur;
}
public void ecrire(String adresse, Integer valeur) {
    table.put(adresse, valeur);
}
```

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 :

```java
public abstract class Binaire extends Expression {
	protected Expression op1;
	protected Expression op2;

	public Binaire(Expression op1, Expression op2) {
		this.op1 = op1;
		this.op2 = op2;
	}

}
```

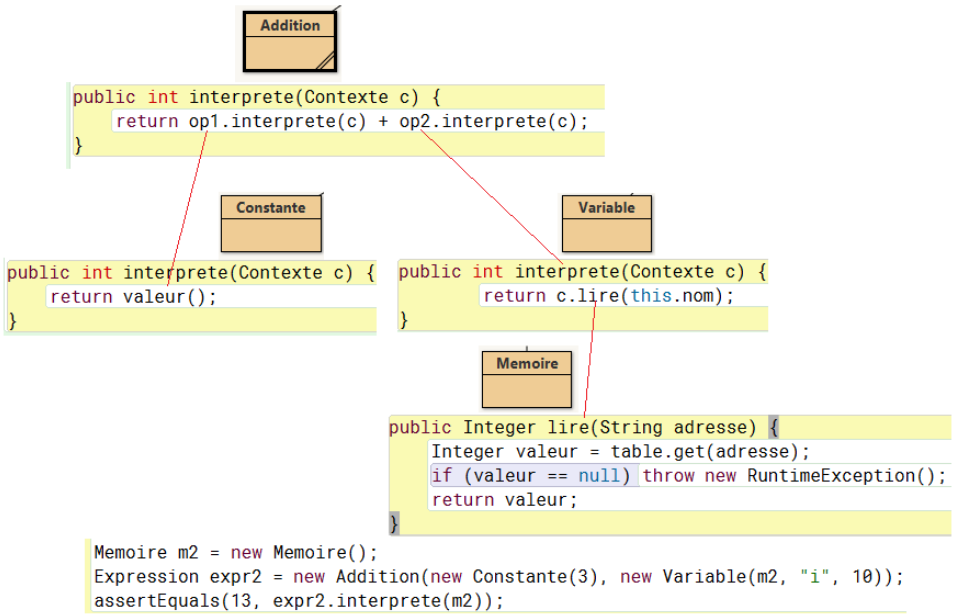
Alors que celles qui héritent d'Unaire en ont qu'une :

```java
public abstract class Unaire extends Expression{
    protected Expression op;

    public Unaire(Expression op){
        this.op = op;
    }
}
```

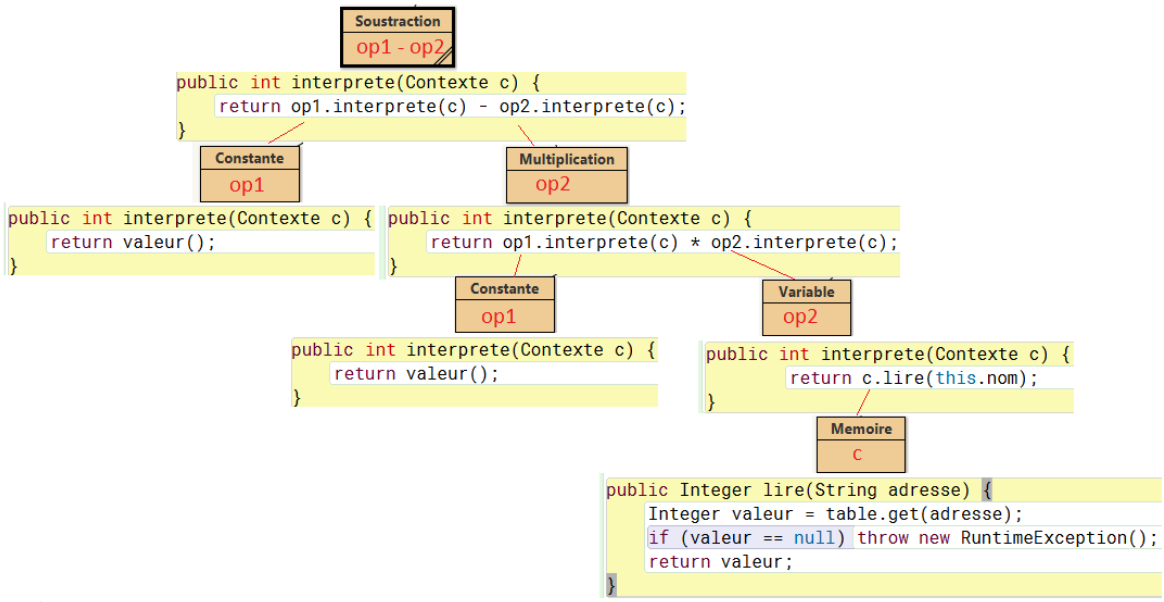
La méthode *interprete()* travail avec les **2 Expressions** enregistrées par le constructeur.

On exécute sur ces deux Expressions leurs méthodes Interprete() respectives qui iront chercher individuellement un résultat terminal.

Exemple simple 1:



Exemple moins simple 2 :




```
Memoire m3 = new Memoire();
Expression expr3 = new Soustraction(new Constante(6), new Multiplication(new Constante(3), new Variable(m3, "i", 2)));
assertEquals(0, expr3.interprete(m3));
```

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

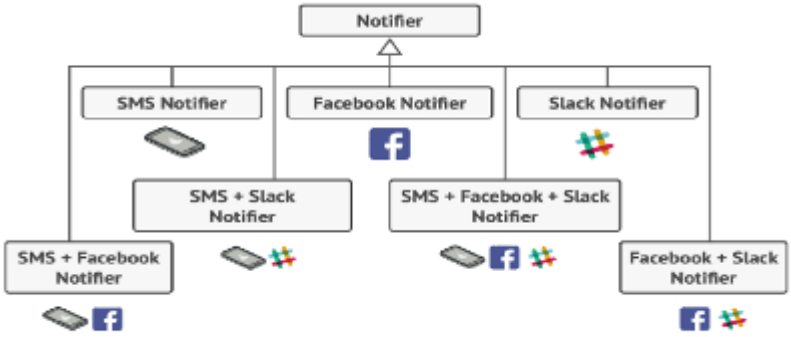
Le problème est que les évaluations et les types de retours sont en durs. Cela veut dire que s'il fallait une nouvelle interprétation, il faudrait alors dupliquer le code.

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 :



Explosion combinatoire des sous-classes.

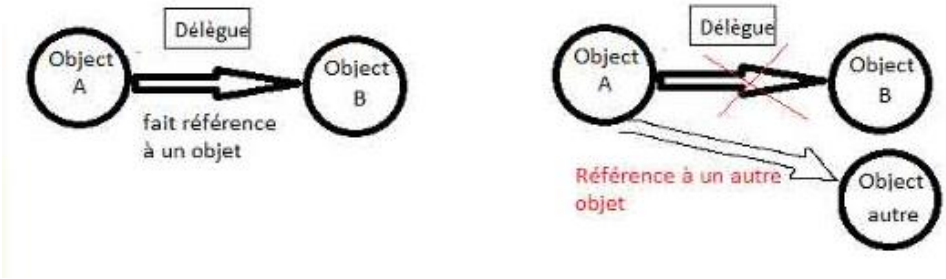
Autres problèmes de l'héritage :

-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**.

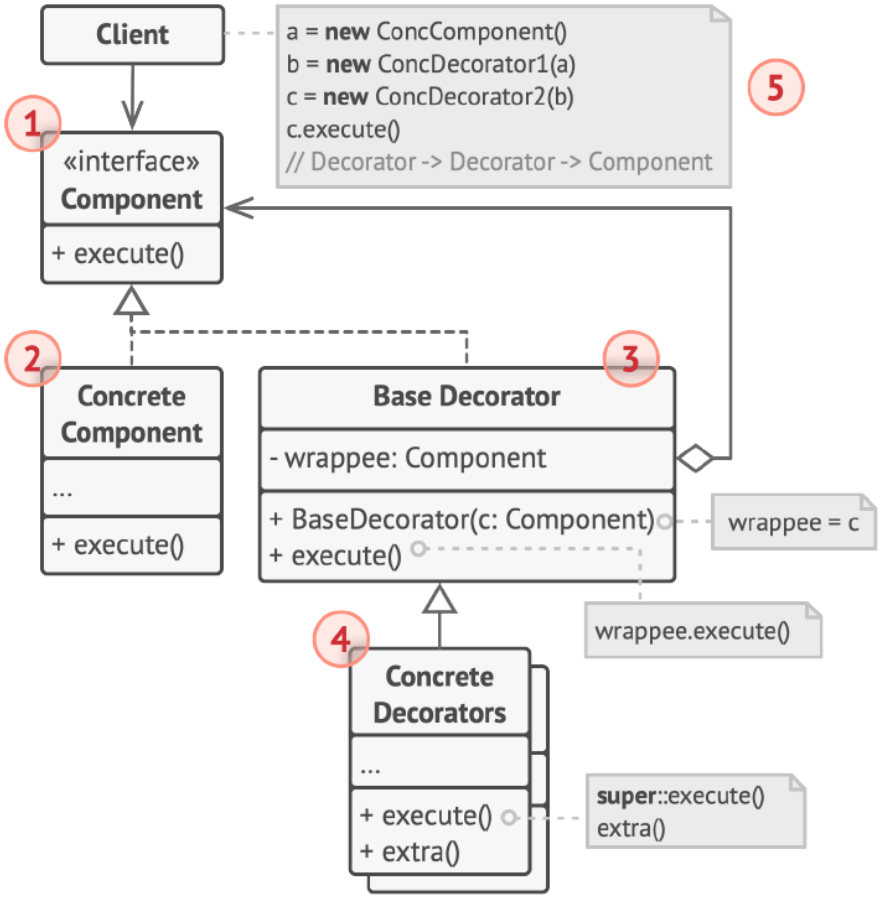
Cela donne aussi la possibilité d'agir avant ou après l'exécution à l'aide d'un **décorateur concret**.



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 :

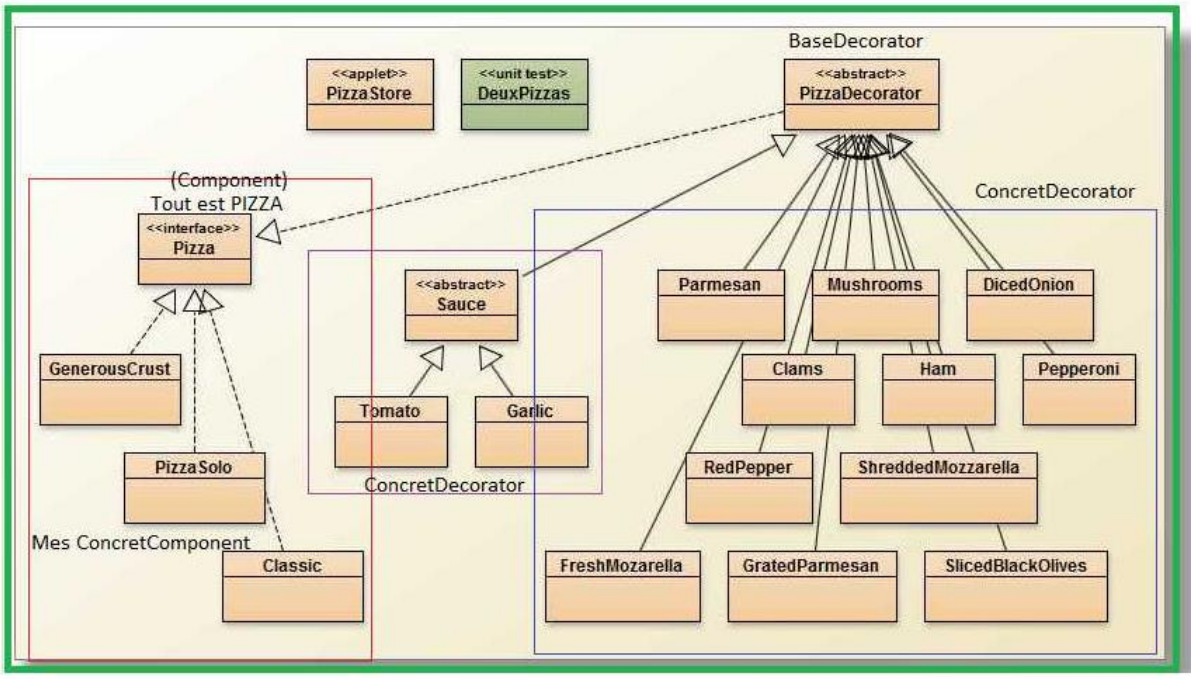


Tout est issue de l'interface **Component**. Ces méthodes sont donc communes à toutes les classes.
La suite se divise en 2 parties :

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).

```
public interface Pizza{
  /** Une description textuelle de la Pizza */
  abstract public String getDescription();
  /** le prix de vente */
  abstract public double cost();
}
```

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

```
//référence objet emballé/decorateur
protected Pizza pizza;
//constructeur de base qui accepte un objet Component
public PizzaDecorator(Pizza pizza){
    //les enfants feront super(pizza)
    this.pizza = pizza;
}
//délègue aux enfants
public abstract String getDescription();
public abstract double cost();
```

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

```
//passe par la classe mère
public Parmesan(Pizza p){
    super(p);
}
//hérite des deux méthodes suivantes de la mère
public String getDescription(){
    return pizza.getDescription() + ", parmesan";
}
public double cost(){
    return pizza.cost() + .30;
}
```

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

```
Pizza pizza = new Mushrooms(new Ham(new Ham( new Classic()));
```

(notre objet)

nos décorateurs

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

```
Pizza pateClassic = new Classic();
Pizza pizza2 = new Mushrooms(new Ham(new Ham( pateClassic )));
```

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

```
public class Ham extends PizzaDecorator{
    public Ham(Pizza p){
        super(p);
    }
}
```

```
public abstract class PizzaDecorator implements Pizza{
    protected Pizza pizza;

    public PizzaDecorator(Pizza pizza){
        this.pizza = pizza;
    }
}
```

(1)

```
public String getDescription(){
    return "Pizza classic";
}
```

(2)

```
public String getDescription(){
    return pizza.getDescription() + ", ham";
}
```

(3)

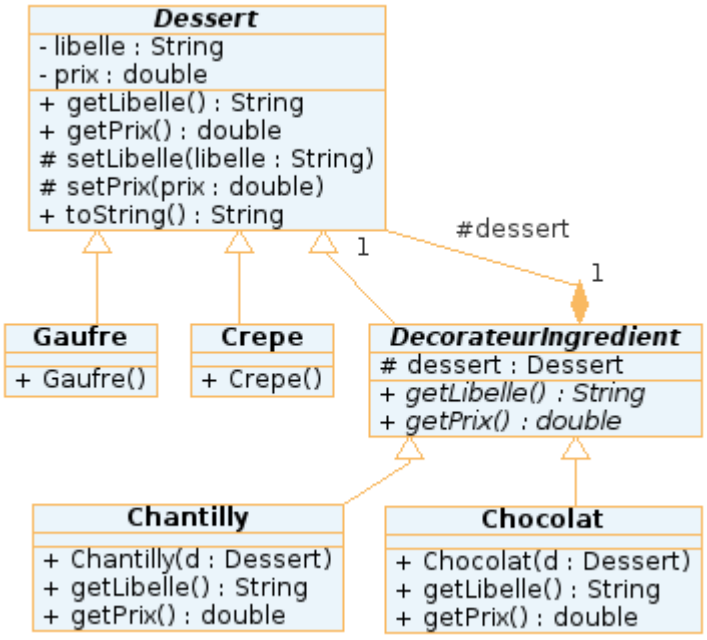
```
public String getDescription(){
    return pizza.getDescription() +
}
```

(4)

```
public String getDescription(){
    return pizza.getDescription() + ", 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() :

```
Class Vrai {
        Public boolean interprete() {
                return true;
        }

}
```

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 :

```
public abstract class VisiteurExpressionBooleenne<T> {

    public abstract T visite(Vrai v);
    public abstract T visite(Faux f);

    public abstract T visite(Non n);

    public abstract T visite(Ou ou);
    public abstract T visite(Et et);

    public abstract T visite(Sup sup);
    public abstract T visite(Egal eg);
    public abstract T visite(Inf inf);

}
```

Le visiteur se présent comme suit :

```
public Boolean visite(Ou ou) {
    return ou.bop1().accepter(this) || ou.bop2().accepter(this);
}
```

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

```
public <T> T accepter(VisiteurExpressionBooleenne<T> v) {
    return v.visite(this);
}
```

(**Retour**)

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

Si le visiteur renvoi directement une valeur finale :

```
public Boolean visite(Vrai v) {
    return true;
}
```

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)*).

```
//ALLER
Class A( {
        MethodeA(B b) {
                b.methodeB(this donc A);
         }

}
//RETOUR
Class B {
        MethodeB(A a) {
                a.methodeA(this donc B);
        }
}
```

Par exemple :

```
                                                    param1                    param2
"((5 + 3) = 8)", new Egal(new Addition(new Constante(5),new Constante(3)),new Constante(8)).accepter(vbs));
```

```
protected Contexte m;
protected VisiteurExpressionBooleenne<String> vbs;
```

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 :

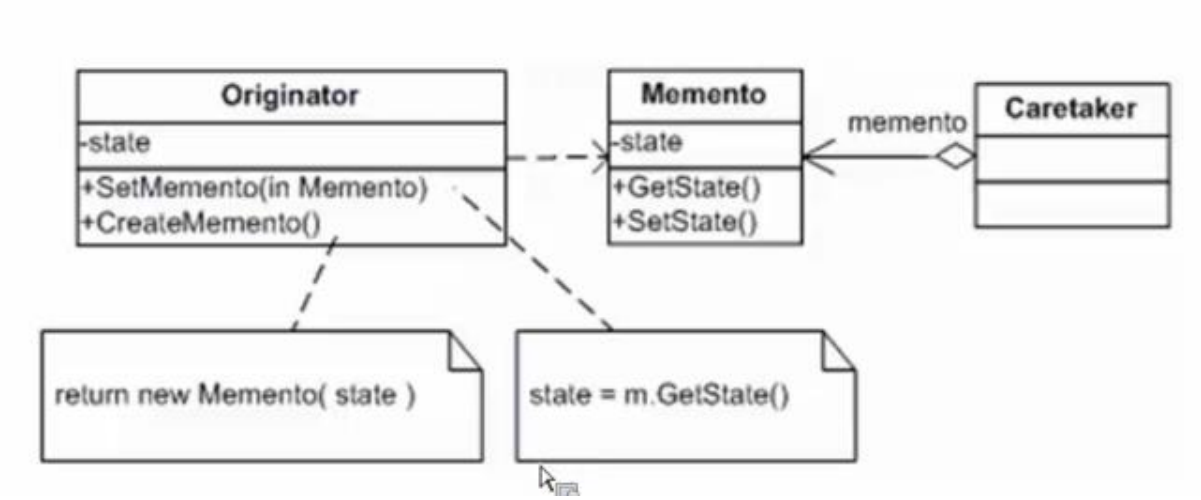
```
public void testVisiteurBoolString(){
    assertEquals("vrai", new Vrai().accepter(vbs));
    assertEquals("faux", new Faux().accepter(vbs));
    assertEquals("(5 > 8)", new Sup(new Constante(5),new Constante(8)).accepter(vbs));
    assertEquals( "((5 + 3) = 8)", new Egal(new Addition(new Constante(5),new Constante(3)),new Constante(8)).accepter(vbs));
```

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.

```java
public class Main {

    public static void main(String[] args) throws NotePadFullException {
        NotePad notes = new NotePad();
        notes.addNote("15h : il pleut");
        System.out.println("notes : " + notes);

        Caretaker gardien = new Caretaker();
        gardien.setMemento(notes.createMemento());  // sauvegarde

        notes.addNote("16h : il fait beau");
        System.out.println("notes : " + notes);
        gardien.setMemento(notes.createMemento());  // sauvegarde

        notes.addNote("17h : il neige");
        System.out.println("notes : " + notes);

        notes.setMemento(gardien.getMemento());  // restitution
        System.out.println("notes : " + notes);

        notes.setMemento(gardien.getMemento());  // restitution
        System.out.println("notes : " + notes);
    }

}
```

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

public class NotePad  {

    private List<String> notes = new ArrayList<String>();

    public void addNote(String note) throws NotePadFullException {
        notes.add(note);
    }
    public String toString() {
        return notes.toString();
    }



    private List<String> getNotes() {
        return this.notes;
    }
    private void setNotes(List<String> notes) {
        this.notes = notes;
    }



    public Memento createMemento() {
        Memento memento = new Memento();
        memento.setState();   // sauvegarde du NotePad, clone, copie…
        // ici la « List<String> » notes
        return memento;
    }

    public void setMemento(Memento memento) {
        memento.getState(); // restitution du NotePad
    }

    public class Memento {

        private List<String> mementoNotes;

        public void setState() {// copie,clonage d'un notePad
            mementoNotes = new ArrayList<String>(getNotes());
        }

        public void getState() {
            setNotes(mementoNotes);
        }
    }
}
```

```java
import java.util.Stack;


public class Caretaker {

    private Stack<NotePad.Memento> mementostk;

    public Caretaker() {
        this.mementostk = new Stack<>();
    }

    public NotePad.Memento getMemento() {
        return mementostk.pop();
    }

    public void setMemento(NotePad.Memento memento) {
        this.mementostk.push(memento);
    }
}
```

Autre exemple :

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

```java
public static void main(String[] args) {
    Createur c = new Createur("1", "1");
    Caretaker ct = new Caretaker();
    ct.saveMemento(c.saveCreateur());
    System.out.println(c.getInfo());

    c.setNomPrenom("2", "2");
    System.out.println(c.getInfo());

    c.restoreCreateur(ct.restaureMemento());
    System.out.println(c.getInfo());
}
```

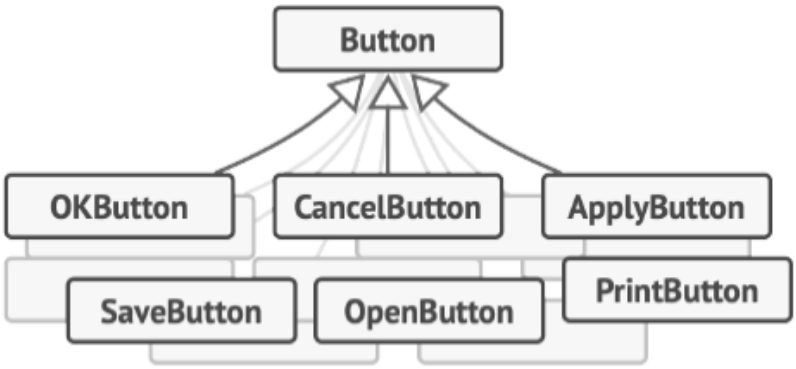
```java
public class Createur {
    private Createur createur;
    private String nomCreateur;
    private String prenomCreateur;

    public Createur(String nom, String prenom) {
        this.nomCreateur = nom;
        this.prenomCreateur = prenom;
        this.createur = this;
    }

    //sauvegarde du créateur
    public Memento saveCreateur() {
        //on doit envoyer au memento le this
        return new Memento(createur);
    }
    //restauration du createur par la récup du memento
    public void restoreCreateur(Memento m) {
        m.getState();
    }

    public void setNomPrenom(String nom, String prenom) {
        System.out.println("Modification du prenom et nom");
        this.nomCreateur = nom;
        this.prenomCreateur = prenom;
    }
    public String getInfo() {
        return this.nomCreateur+" "+this.prenomCreateur;
    }
    ////////////////////////////////
    public class Memento {
        //copies
        private Createur createurMemento;
        private String nomMemento;
        private String prenomMemento;

        //sauvegarde des paramètres pour faire une copie
        public Memento(Createur createur) {
            this.createurMemento = createur;
            this.nomMemento = createur.nomCreateur;
            this.prenomMemento = createur.prenomCreateur;
        }
        //restaure vers créateur
        private void getState() {
            createur = this.createurMemento;
            nomCreateur = this.nomMemento;
            prenomCreateur = this.prenomMemento;
        }
        /*
        public String getInfoMemento() {
            return createurMemento.getInfo();
        }*/
    }
```

```java
public class Caretaker {
    private Stack<Createur.Memento> historique;

    public Caretaker() {
        this.historique = new Stack<>();
    }

    public Createur.Memento restaureMemento() {
        //System.out.println("Caretaker(Restaure), memento restauré :"+historique.get(0).getInfoMemento());
        return historique.pop();
    }

    public void saveMemento(Createur.Memento memento) {
        this.historique.push(memento);
        //System.out.println("Caretaker(SAVE), memento ajouté :"+memento.getInfoMemento());
    }
}
```

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 :



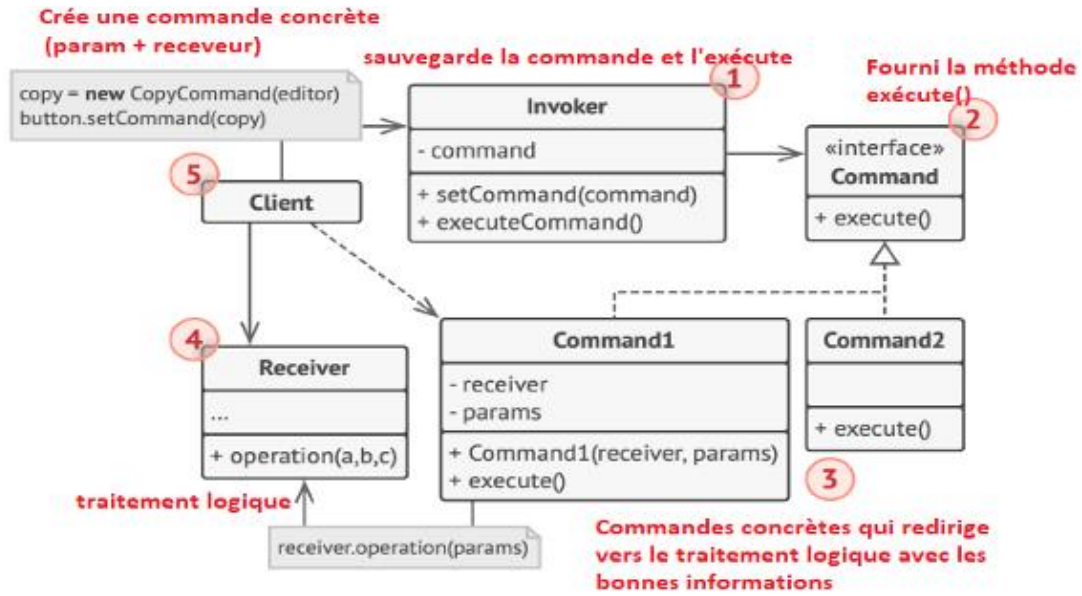
De nombreuses sous-classes de boutons. Tout semble aller pour le mieux.

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.

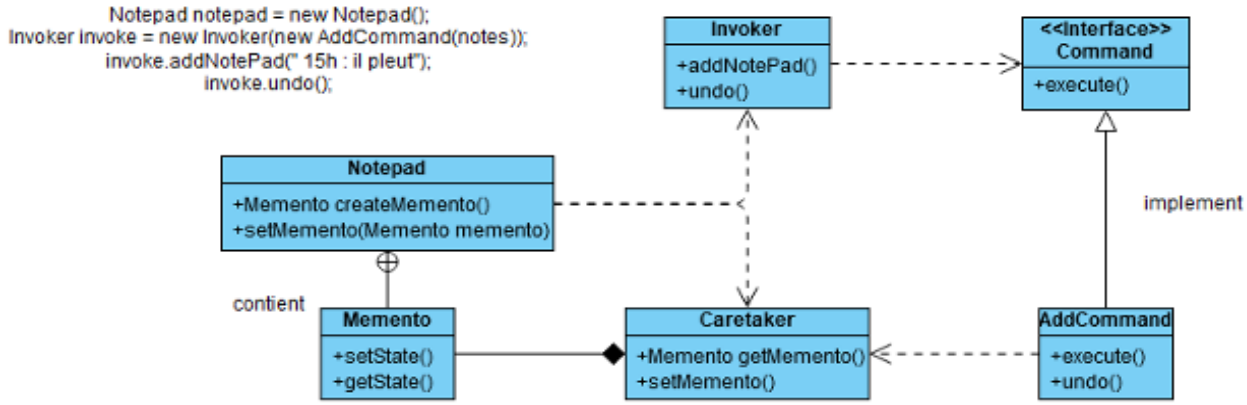


Le **Client** souhaite réaliser une action au niveau d'un clic sur un bouton : copier/coller (sushis).
Pour se faire, il crée une **Commande** à partir du constructeur de la classe concrète de la Commande à réaliser, exemple : un copier-coller. A la création, il fournit des paramètres dont celui du **Receveur**.

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**

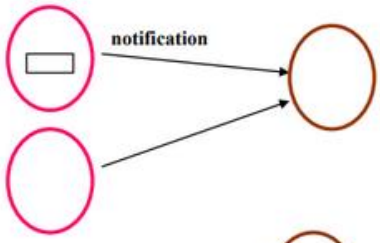
2) Je crée un **Demandeur** (invoker) auquel j'ajoute une **commande concrète** auquel j'ajoute mon **notePad**.
3) La méthode **addNotePad**() lance la méthode **execute**() de l'objet **command** contenu dans Invoker: **addCommand**() qui lui lance **gardien.setMemento(notes.createMemento())**;

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

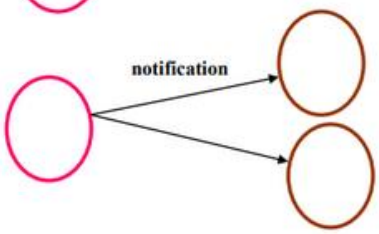
Observateur

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).*

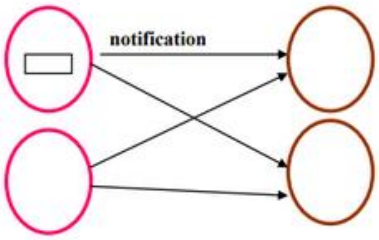
- **Plusieurs Observés / un observateur**



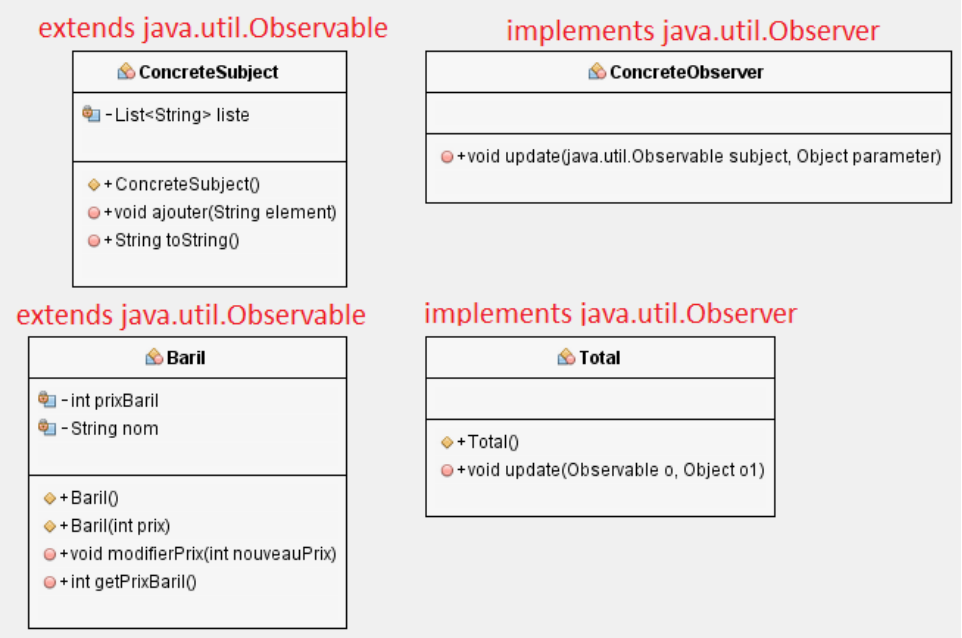
- **Un observé / plusieurs observateurs**



- **Plusieurs observés / plusieurs observateurs**



patron Observer NATIF:



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

```
public class ConcreteSubject extends java.util.Observable

        public void ajouter(String element) {
            this.liste.add(element);
            this.setChanged();
            notifyObservers();
        }
```

boolean à true ou false

//NATIFS

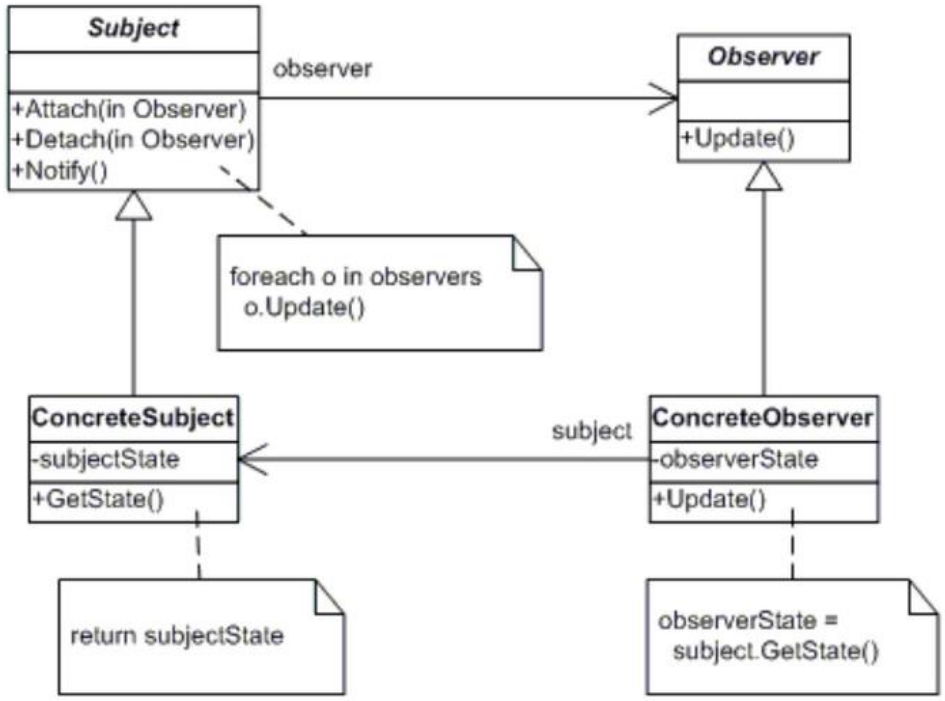
```
public class ConcreteObserver implements java.util.Observer { //Observer
    @Override
    public void update(java.util.Observable subject, Object parameter) {
        System.out.println("update de " + subject);
    }
}
```

```
public static void main(String[] args) {
    //observé/observable/subject concret
    ConcreteSubject cs = new ConcreteSubject();
    //observer/observateur concret
    java.util.Observer concreteObserver1 = new ConcreteObserver();

    cs.addObserver(concreteObserver1);

    cs.ajouter("coucou1");//modification

    java.util.Observer concreteObserver2 = new ConcreteObserver();//observateur 2
    cs.addObserver(concreteObserver2);

    cs.ajouter("coucou2");////modification

    Baril b = new Baril(50);
    Total t = new Total();
    b.addObserver(t);
    b.modifierPrix(5);////modification
}
```

Résultat :

```
update de [coucou1]
update de [coucou1, coucou2]
update de [coucou1, coucou2]
Le prix du baril vient de changer : 5
```

Le schéma du patron observer pas NATIF:



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

```
public abstract class Subject {
    //La liste des observateurs
    private List<Observer> observers;


    public Subject() {
        this.observers = new ArrayList<Observer>();
    }

    public void attach(Observer obs) {
        this.observers.add(obs);
    }
    public void detach(Observer obs) {
        this.observers.remove(obs);
    }
```

Mais aussi la méthode pour prévenir les Observer (observateur) d'un changement :

```
public void notifyObservers() {
    for(Observer obs : this.observers) {
        //on renseigne l'observé concret
        obs.update(this);
    }
}
```

Le **concreteSubject** hérite de **Subject.** Au niveau des méthodes qui apportent un changement que

l'on veut monitorer :

```
public class ConcreteSubject extends Subject {//Subject

      public void ajouter(String element) {
          this.liste.add(element);
          this.notifyObservers();
      }
```

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

```
//l' Observateur
public interface Observer {
    public void update(Subject subject);
}

public static void main(String[] args) {
    //observé/observable/subject concret
    ConcreteSubject cs = new ConcreteSubject();
    //observer/observateur concret
    ConcreteObserver col = new ConcreteObserver();

    cs.attach(col);
    //implémentation perso
    cs.ajouter("coucoul");

    ConcreteObserver co2 = new ConcreteObserver();//observateur 2
    cs.attach(co2);
    ////implémentation perso
    cs.ajouter("coucou2");

    cs.detach(col);
    cs.detach(co2);
    //je n'ai plus d'observer (observateurs)
    cs.ajouter("coucou3");


    Baril baril = new Baril();
    Total total = new Total();
    //implémentation native
    baril.attach(total);
    baril.modifierPrix(10);
}
```

detach (rien)

```
update de [coucoul] obs1
update de [coucoul, coucou2] obs1
update de [coucoul, coucou2] obs2
Le prix du baril vient de changer : 10
```

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

On ne parle souvent pas "d'observateur" mais "d'écouteurs" (listener)

- java.awt.event.EventListener

> – Les écouteurs/observateurs sont tous des « EventListener »

- Convention syntaxique de Sun pour ses API

> – **XXXXX**Listener extends EventListener
> « Observer »
>
> – add**XXXXX**Listener
> « addObserver »
>
> exemple l'interface **Action**Listener / add**Action**Listener

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 :

Ensuite, la classe IHMQuestion2_1 demande à être complété. Il faut ajouter des écouteurs à nos différents boutons en suivant ces 3 règles:
-le bouton A a **3** observateurs **jbo1**, **jbo2** et **jbo3**
-le bouton B a **2** observateurs **jbo1** et **jbo2**
-le bouton C a **1** observateur **jbo1**

En code:

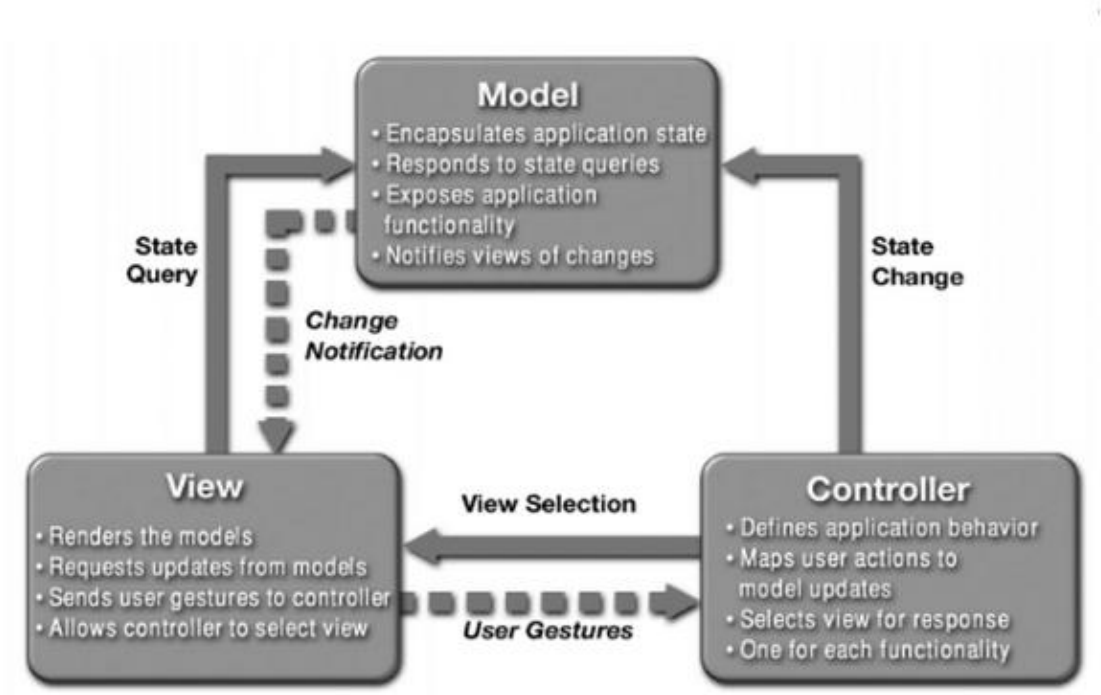
```
// à compléter
// le bouton A a 3 observateurs jbo1, jbo2 et jbo3
boutonA.addActionListener(jbo1);
boutonA.addActionListener(jbo2);
boutonA.addActionListener(jbo3);

// le bouton B a 2 observateurs jbo1 et jbo2
boutonB.addActionListener(jbo1);
boutonB.addActionListener(jbo2);

// le bouton C a 1 observateur jbo1
boutonC.addActionListener(jbo1);
```

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

-L'interface **Observer** est remplacée par l'interface **java.awt.event.MouseListener**.
-La méthode **notifyObservers()** est remplacée par **mouseEntered(MouseEvent ae)**.
-La méthode **addObserver** est remplacée par **java.awt.event.addMouseListener**.

```
// à compléter pour la question 2_2 (JMouseObserver)
JMouseObserver jmo1 = new JMouseObserver("jmo1", contenu);
JMouseObserver jmo2 = new JMouseObserver("jmo2", contenu);
JMouseObserver jmo3 = new JMouseObserver("jmo3", contenu);

// le bouton A a 1 observateur jmo1
boutonA.addMouseListener(jmo1);
// le bouton B a 1 observateur jmo2
boutonB.addMouseListener(jmo2);
// le bouton C a 1 observateur jmo3
boutonC.addMouseListener(jmo3);
```

MVC (Model-View-Controller)

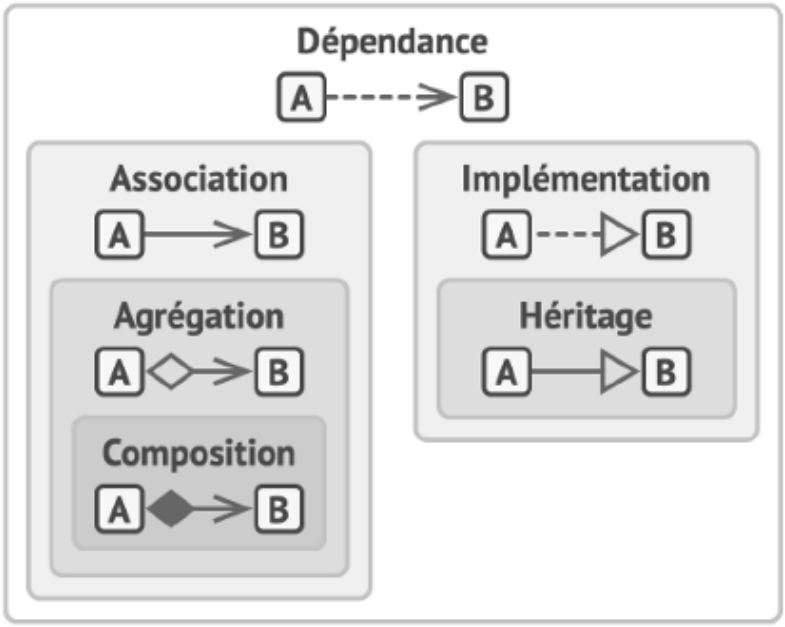


- Le **Modèle** (Observ**é**/Observable/Subject) contient la logique et l'état de l'application, il prévient ses observateurs lors d'un changement d'état.
- La **Vue** représente l'interface utilisateur, elle sert d'Observer (Observ**er**/Observateur)
- Le **Contrôleur** assure la synchronisation entre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

- **Dépendance :** La classe A peut être impactée par les modifications apportées à la classe B.
- **Association :** L'objet A connait l'objet B. La classe A dépend de B.
- **Agrégation :** L'objet A connait l'objet B et contient des B. La classe A dépend de B.
- **Composition :** L'objet A connait l'objet B, contient des B et gère le cycle de vie de B. La classe A dépend de B.
- **Implémentation :** La classe définit des méthodes déclarées dans l'interface B. Les objets A sont traités comme des B. La classe A dépend de B.
- **Héritage :** La classe A hérite de l'interface et de l'implémentation de la classe B, mais elle peut l'étendre. Les objets A peuvent être traités comme des B. La classe A dépend de B.



Les relations entre les objets et les classes : de la plus faible à la plus forte.