

### Variable references

#### **Prof :**

Voiture maVoiture = new Voiture (« F488 », « Ferrari»); /\*maVoiture est une variable (non un objet) qui pointe vers l'objet en question c'est donc un pointeur\*/

En java notre variable déclarer est un pointeur vers notre objet dans la mémoire

Et notre objet est un élément qui sera accessible via notre variable

**N.B :** Prof si nous créons une autre variable avec les mêmes attributs exp

Voiture maSecondeVoiture = new Voiture (« F488 », « Ferrari»);

// Alors on aura deux objets identiques qui pointent sur deux références distinctes même s'ils ont les mêmes paramètres

Prof : Ainsi si on veut que les deux variables partagent la même mémoire, pointent sur la même adresse mémoire alors il faut :

maSecondeVoiture = maVoiture ;

**Prof :** Des lors si nous modifions le paramètre de l'une des variables cela aura un effet sur l'autre variable vice versa

maSecondeVoiture.setNom(« Peugeot »);

System.out.println(maVoiture); // Affiche Peugeot

System.out.println(maSecondeVoiture); // Affiche Peugeot

(Très important !)

**Prof :** D'autre part nous pouvons déduire de cette remarque que la condition if **ne retournera pas true** si les deux objets pointent vers deux espaces mémoire différentes.

**Prof :** Les deux pointeurs référencent le même objet

Point pointA = new Point( 2.0 , 3.0 );

Point pointB = new Point( 2.0 , 3.0 );

if (pointA==pointB) { return true; }

#### **Ramasse Miette**

Point a = new Point( 2.0 , 3.0 ); //allocation de mémoire pour un objet

Point a = new Point(10.0 , 15.0 ); // a contient la référence d'un autre objet

A l'activation du ramasse-miette, l'instance de l'objet de coordonnées (2.0, 3.0) sera effacée de la mémoire, la quantité de mémoire qui lui a été affectée est récupérée.

#### **Création d'un pointeur exp**

Voiture maVoiture ; // Pointe vers rien pour l'instant

### Création d'un objet

Voiture maVoiture = new Voiture() ; // Appel des valeurs du constructeur par défaut

/\*Note fonction du contrôle de parrain\*/

### Methode Static

En ajoutant le mot static devant une méthode, cela permet d'appeler la méthode sans la rattacher à une instance de la classe en question

```
public static void methode1(){ System.out.println("Appel de la methode 1") }
```

//Appel

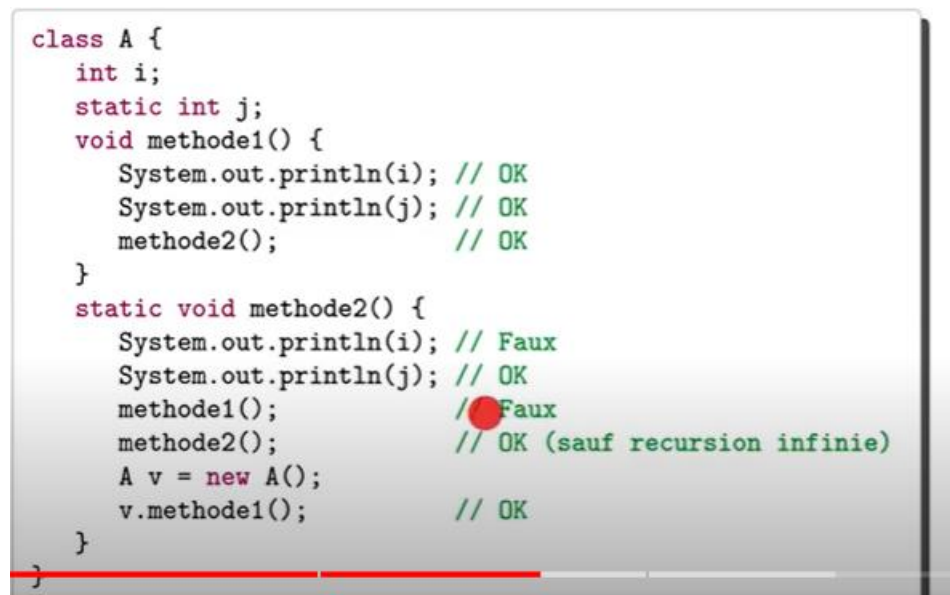
Si a est une classe contenant methode1() on a

A.methode1()

On peut aussi faire

A unObjet = new A() ;

unObjet.methode1() ;



```
class A {
    int i;
    static int j;
    void methode1() {
        System.out.println(i); // OK
        System.out.println(j); // OK
        methode2();           // OK
    }
    static void methode2() {
        System.out.println(i); // Faux
        System.out.println(j); // OK
        methode1();           // Faux
        methode2();           // OK (sauf recursion infinie)
        A v = new A();
        v.methode1();          // OK
    }
}
```

Prof : static : Veut dire qu'elle est visible au niveau de la classe et qu'elle est partagée dans toute la classe

### Attribut Static :

Si on ajoute static à la déclaration d'une variable, la valeur de la variable est partagée entre toutes les instances de la classe

Prof : Ils sont hérités aussi dans la sous-classe

```
public static int attribut1;
```

**La surcharge** ou la surdefinition est le fait de déclarer deux méthode ayant les même nom mais des paramètres différents

```
public void afficher(Eudiant) ;  
public void afficher(Professeur) ;
```

### **Methode boolean equals**

On utilise la methode equals pour comparer l'égalité du contenu des chaines(ou la méthode compareTo)

#### **Exemple :**

```
String statut = "étudiant" ;  
if ( statut.equals("étudiant") ){ return true ; }
```

Dans le cadre de l'héritage par exemple, une methode equals peut naitre et il sera question de comparer l'égalité de tout les attributs de la classe

#### **Exemple :**

```
public boolean equals(PlatsChaud unPlatChaud) {  
    if (super.equals (unPlatChaud) &&  
        unPlatChaud.getTempsCuisson() == getTempsCuisson()) {  
        return true;  
    }  
    return false;  
}
```

//

to\_string() de C++ est en java String.valueOf()

#### **Exemple :**

```
public String toString() {  
    return super.toString() + " le temps de cuisson est  
    "+String.valueOf(getTempsCuisson()) ;  
}
```

### **Différence entre classe abstraite et interface**

A la différence d'une classe abstraite, toutes les méthodes d'une interface sont des méthodes abstraites. Une interface ne peut pas être instanciée.

Une classe abstraite peut implémenter une interface et implémenter les méthodes de l'interface mais la réciproque n'est pas possible

les classes abstraites servent à modulariser le code généralement pour les classes ayant des fonctionnalités communes tandis que les interfaces servent à implémenter les fonctionnalités métiers.

### **Design Pattern**

## Singleton

Prof : Les patrons de conceptions de conceptions sont des techniques bien précise et bonnes pratiques pour répondre à un problème de conception d'un logiciel

Pour utiliser le design pattern Singleton, il faut créer un constructeur privé pour que cette classe ne soit pas instanciée ailleurs

Prof : Par exemple pour une connexion à une base de données si on a une application qui contient 100 utilisateurs on ne va pas créer à chaque fois 1 connexion pour les 100 différents utilisateurs

Car ce ci bloquera la mémoire. Donc il va falloir utiliser ce design pattern et un seul objet sera et un seul objet sera créer pour l'ensemble des 100 utilisateurs.

Considérons une classe **MySQLDBService** (elle va implémenter l'interface DB service)

Pour pouvoir contrôler qui peut construire cette classe ; la solution est d'interdire à tout le reste de l'application de pouvoir instancier cette classe. Pour cela il suffit de préciser que le constructeur vide de cette classe soit **privée**.

Dès lors plus personne ne pourra étendre cette classe. Même les sous classe ou super classe ne pourront pas construire cette classe

Le code extérieur a cette classe ne pourra pas également

Nous pourrions créer une instance de cette classe uniquement à **l'intérieur** de la classe et pour rendre cette instance disponible il faut que ce soit nécessairement avec un accesseur statique

```
public class MySqlDBService {  
  
    private MySqlDBService (){} // Vu que son constructeur est privée les instances de cette  
                                classe ne se feront qu'à l'intérieur de la classe  
  
    static MySqlDBService service = new MySqlDBService(); // On déclare un accesseur static  
    pour rendre cette instance disponible ; On est à l'intérieur de la classe  
  
    //On déclare une méthode static pour pouvoir publier le champ service à l'extérieur  
  
    static MySqlDBService getInstance(){  
  
        return service;  
  
    } //C'est l'implémentation du pattern Singleton en java  
  
    // Pour que l'initialisation de l'instance service soit contrôlé il faut la mettre donc dans le  
    getInstance(){} Des lors il sera invoqué lors de l'appel de getInstance()
```

Prof : Cas pratique. Lors d'une connexion a une base de données, on dira que dès qu'on veut se connecter a MySql on appelle la classe DataBase qui appelle la méthode de PDO qui crée une nouvelle instance et donc une nouvelle connexion

La solution que Propose le Pattern du Singleton permet d'éviter le MAX\_CONNEXION\_LIMITE qu'on a retrouvée autrefois sur plusieurs connexions de site Web

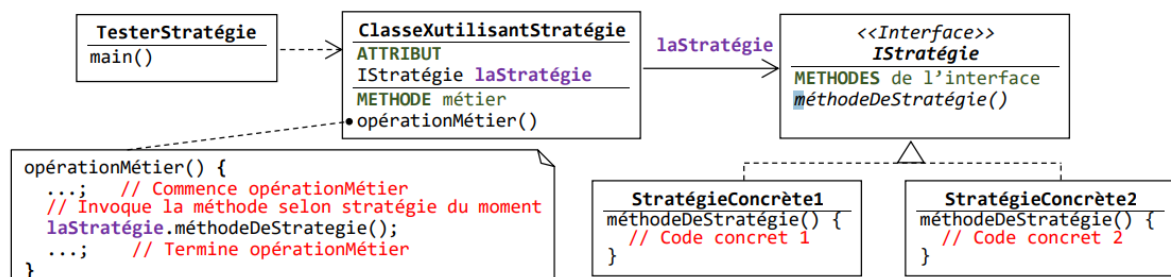
Prof : On parle de Pattern Singleton car on a une seule instance de l'objet

## Stratégie

Le design pattern Stratégie permet d'affecter dynamiquement une opérationMétier() à une classe utilisant la stratégie voulue où opérationMétier() manipule à partir d'un de ses attributs les méthodes de stratégie définie dans une ou plusieurs classe concrète

**Prof :** Un peu le cas d'une dépendance entre la classe utilisant la stratégie et une classe disposant des méthodes de stratégie

### Diagramme de classe du Design pattern Stratégie



Remarque : ClasseUtilisantStratégie définit une instance de la classe IStratégie (elle-même étant une interface spécialise deux classes pour implémenter sa méthode methodeDeStrategie)

### Etape de mise en place du Design pattern Strategie

On commence par définir une classe interface IStrategie qui définit la méthodes qu'on veut exécuter (methodeDeStrategie()) puis selon les différentes methodeDeStrategie() on les implémente dans chaque classe StrategieConcreteI cette méthode définie dans l'interface

Ensuite on créer une classe qui doit contenir un attribut (ou champ) de type IStrategie et une méthode qui fera appel aux méthodes implanter dans les classes Stratégie

Et enfin on crée une classe main pour exécuter le Design Pattern

## Composite

On organise les objets en structure arborescente.

Prof : Le composite permet aux utilisateurs de traiter tous les composants de la structure arborescente de la même façon qu'ils soient des composants simples ou bien des objets composés d'autres composants. Donc on peut avoir plusieurs composés

**Composant :** Classe abstraite qui déclare l'interface métier

**Simple** : une classe qui n'a pas d'enfant (cad elle ne peut contenir des composant enfant) mais qui **implémente le comportement métier du composant**

On dit qu'il représente les objets feuille dans l'arborescence cad

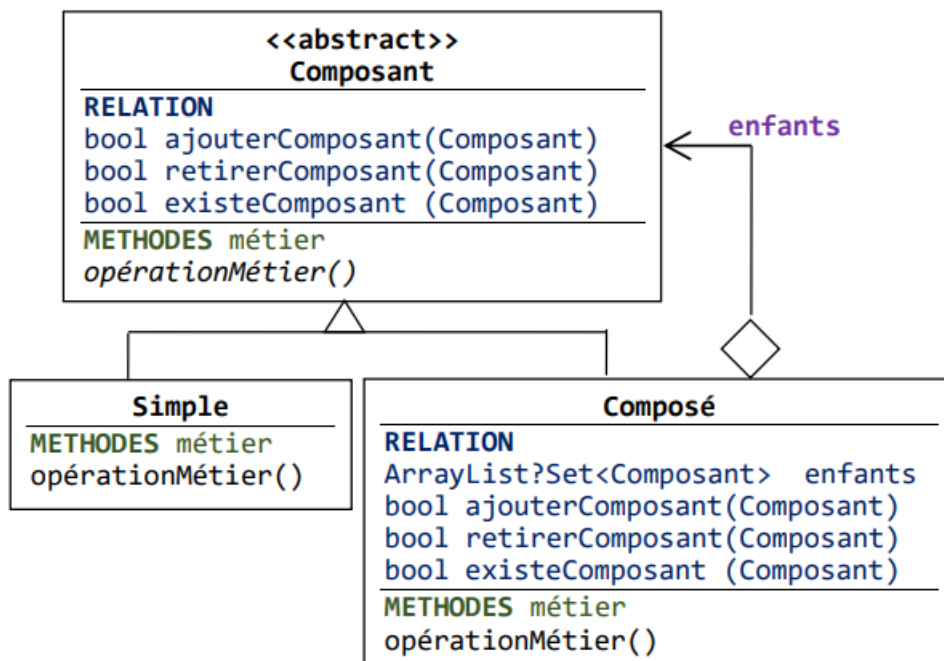
**Composé** : Un Composé **stocke les composants enfants qui le composent** afin de laisser le client solliciter l'opération métier de l'ensemble de ces composants.

On dit qu'il représente les objets nœud dans l'arborescence.

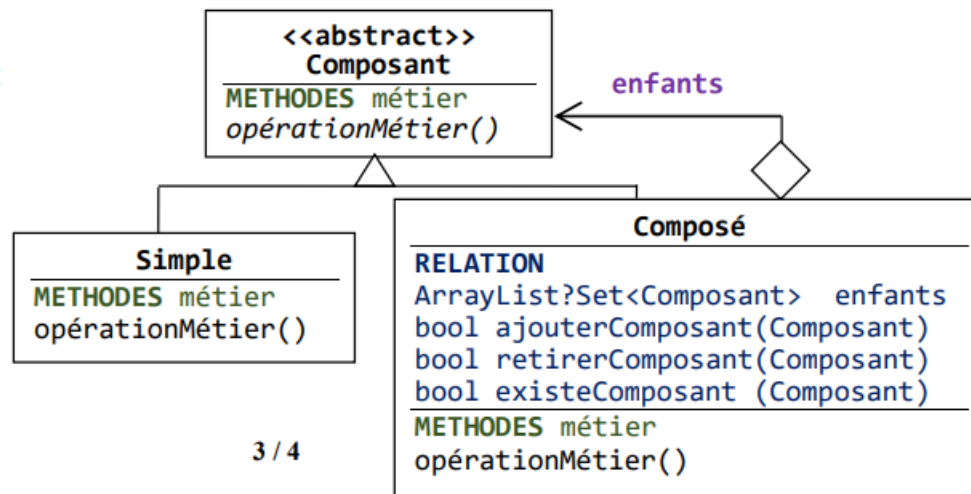
**Client** : Utilise l'interface de la classe Composant pour manipuler indifféremment les objets de la structure composite. (Toujours à partir de l'interface qu'il manipule les objets)

**Prof** : Deux approches sont proposées pour implémenter la relation enfants

La **première approche**, préconise de définir les méthodes existe/ajouter/retirerComposant dans la super-classe Composant et de définir une méthode abstraite bool estComposé() dans Composant, à spécialiser dans les sous-classes. **bool estComposé()** Retourne false dans la classe Simple ( puisqu'elle n'est pas composée des composants enfants) et true dans la classe Composé. A partir de ce moment les objets de la classe Simple ne pourront pas utiliser les méthodes non pertinentes de la classe Composant 😊



La **seconde approche**, préconise de définir les méthodes existe/ajouter/retirerComposant uniquement dans la classe Composé. Ainsi, l'interface pour tous les objets de la hiérarchie de classes de racine Composant est **limitée aux comportements métier**. Le **problème** est que le développeur **ne peut invoquer** les méthodes existe/ajouter/ retirerComposant, indifféremment, pour tout objet des sous-classes de Composant.



Rmq : Ces deux approches sont implémentées avec **Set**, ou **ArrayList**

**N.B :** Le Design pattern permet à la partie client de manipuler de la même manière un objet unique et un objet composé

*Comme l'élément composé est un élément essentiel, il peut être composé d'autres éléments composés*

**N:B Dans le pattern composite quand on appelle la méthode opération() d'un composée, automatiquement il appelle les éléments de chaque composant prioritairement**

*C'est l'objectif du pattern Decorateur*

Rmq : La différence entre ces deux approches se remarque aussi dans l'instanciation d'un objet de la classe Composant

```
Composant unComposant = newComposee("xxx", yyy);
```

```
// En ce moment unComposant ne pourra qu'a appeler ses méthodes methode
// metier ajouter/supprimer/existe -> Il faut d'on qu'ils soient déclaré dans
// la classe Composant
```

## Contrôle

**Prof :** Préparer l'essentiel sur une feuille A4

**Ce qu'il faut savoir**

-Il y aura une question sur le composite

Représenter le schéma UML et adopter les solutions qui implémente avec les attributs (cad les objets)

Revoyez donc ma solution de composite

**Prof :** Vous devez donc être alaise avec les notations graphiques

->Le prof reprend l'exemple du trait rond carré et les méthodes effacer, déplacer ..

*Prof : Je vous donnerai la deuxième approche du Composite au devoir au devoir de*

Prof : Préparation de cuisine ;

Quel sont les types de chaque clause (classe)

### **Première implémentation**

PreparationDeCuisine tomateCuite = new PreparationSimple(« Tomate Cuite», )

Prof : Ce sera toujours de la classe X implémenter que l'objet abstract va déclarer

`ClassX objetX = new ClassImplementer(« »,) ;`

Prof : Vous devez connaître les trois façons de parcourir les ArrayList

**Maîtriser les deux solutions Composites mis sur Elearn**

Prof : Vous devez connaître inévitablement [les premiers td aussi](#)

---

### **Principe Solid**

La mise en œuvre du principe Solid vise à améliorer la cohésion, diminuer le couplage et favoriser l'encapsulation afin de maintenir la fiabilité et la robustesse d'une application

**La cohésion** : Un module est cohésif lorsqu'au haut niveau d'abstraction il ne fait qu'une seule et précise tâche

**Le couplage** : Deux modules sont dits couplés si une modification dans l'un des deux modules demande une modification dans l'autre module

**L'encapsulation** : vise à intégrer à un objet tous les éléments nécessaires à son fonctionnement, que ce soient des fonctions ou des données

**Prof** : Les cinq principes Solid sont :

#### **S : Single Responsibility Principe SRP**

Prof : Une classe doit avoir une et une seule responsabilité.

**Rmq** : Cela diminue la complexité du code, augmente la lisibilité de chaque classe et améliore l'encapsulation et la cohésion

#### **O : Open/closed principe OCP**

**Prof** : Une classe doit être ouverte à une extension mais fermée à la modification

**Rmq** : L'idée est qu'une fois qu'une classe a été approuvée elle ne doit plus être modifiée mais seulement étendue

Ainsi, on pourra ajouter une nouvelle fonctionnalité : En ajoutant des sous classes (cf. Ouvert à l'extension) . Mais sans modifier le code de la classe existante (cf. Fermé à la modification)

**Prof** : Le principe ouvert/fermé oblige à faire bon usage de l'abstraction et du polymorphisme. Du coup on préférera manipuler des objets de classes abstraites (plutôt que de classes concrètes) et on utilisera le polymorphisme



## **L : Liskov substitution Principle LSP**

Prof: Une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G soit sous-type de T

## **I : Interface segregation principle**

Prof : Préférer plusieurs interfaces spécifiques pour chaque client puisqu'une seule interface générale

## **D: Dependency Inversion Principle DIP**

Prof : Il faut dépendre des abstractions, pas des implémentations

/\*-----\*/  
/\*-----\*/

## **Observateur**

Observateur (Observer) : Définit une corrélation (ou une dépendance) du type un à plusieurs, entre objets, de façon que lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et mis à jour automatiquement.

**Prof :** La situation d'intérêt pour ce Design Pattern se présente dès lors que les changements d'état d'un objet Observable (cad. Observé) est nécessaire pour les objets Observateurs (car tous ont un œil sur l'observable)

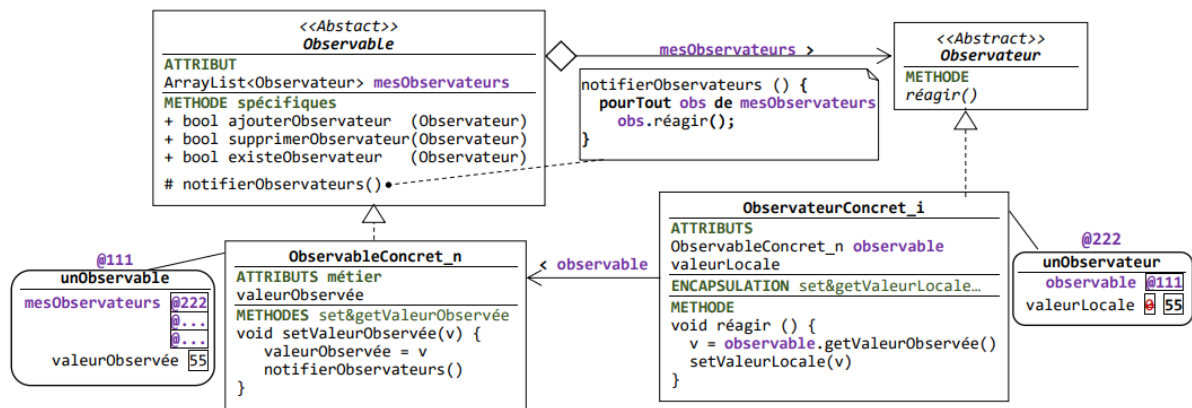
Lorsqu'un Observable change d'état il notifie tous ses observateurs (cf. méthode notifierObservateurs) afin que chacun d'eux puisse réagir à sa façon.

L'observableConcret met à jour ce changement d'état et notifie les ObservateursConcret en fonction de cette mise à jour

Chaque ObservateurConcret connaît l'ObservableConcret pour qui son état l'importe.

**Exemple :** On veut envoyer un message et que ce message soit reçu par plusieurs personnes. Pour cela on doit notifier à une liste des observateur le message arrivant

**Cas pratique :** On désire implémenter un convertisseur de la base normal vers la base binaire, la base octale et la base hexadécimal. Ainsi à chaque fois que l'utilisateur saisie un nombre, on va notifier les observateurs et calculer automatiquement les conversions.



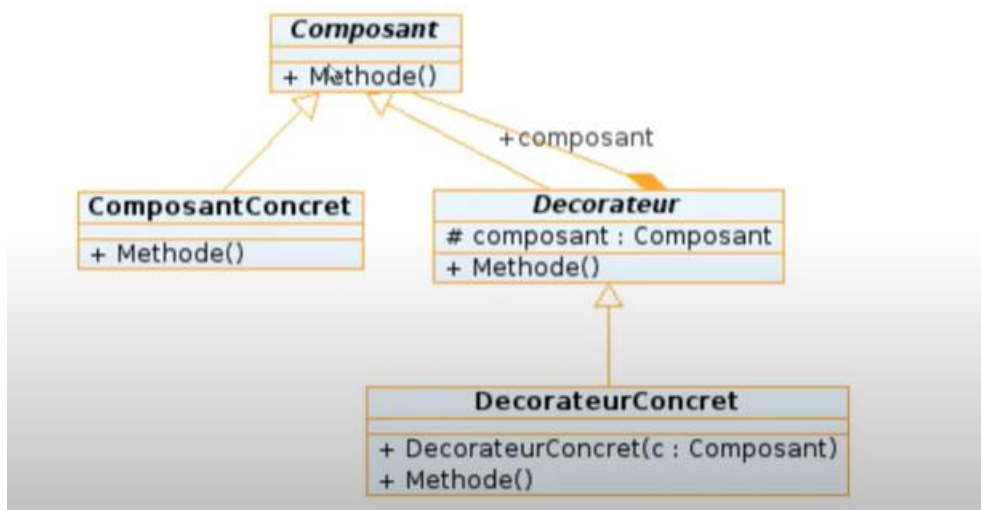
**Cas pratique :** Voir Projet DesignPatternDecorateur

Ainsi quand l'observateur change d'état, c'est l'observable qui indique aux ObservateurConcret du changement d'état de l'observateur qui pour eux, réagissent à ce changement d'état en se mettant à jour en fonction de ce changement d'état

**Prof :** L'observable est en quelque sorte le représentant de l'observateur. En effet le changement d'état des observableConcret alerte les observateurConcret

## Decorateur

**Problématique de l'héritage :** On peut se retrouver avec un graphe d'objet immense. Pour pallier ce problème, on fait intervenir le Design pattern Décorateur



Rmq : L'implémentation du Design qui consiste à définir un pointeur dans une sous-classe de Composant est très intéressante et favorise aussi une bonne construction entre les principales classe principale du Design pattern dans le main. Cas du Décorateur et Médiateur par exemple

**N.B :** A ne pas considérer cette relation d'agrégation entre Décorateur et Composant. On ne s'intéressera pas à créer une liste de composant dans Décorateur

**Composant :** Peut-être une classe abstraite ou une interface (une interface avec Lopolis). Il y a une implémentation qui est bien concrète

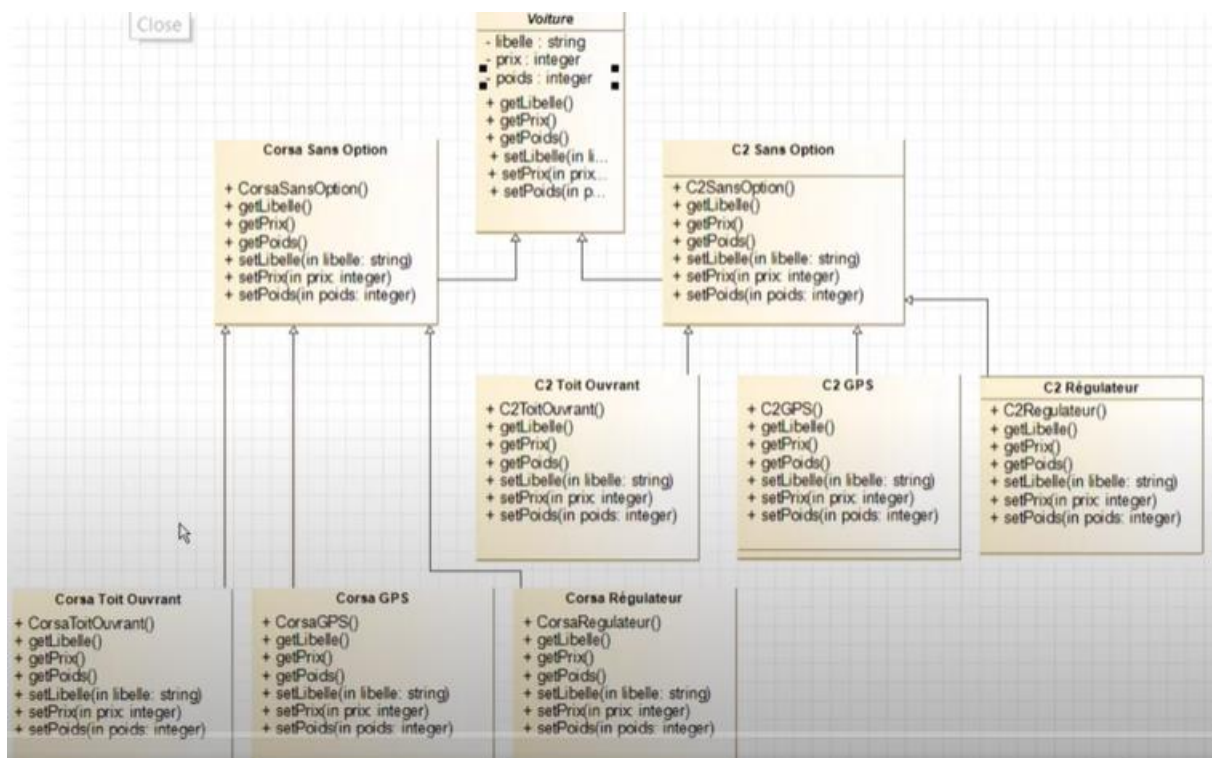
**ComposantConcret** : Ils implémentent la méthode de Composant

**Décorateur** : C'est aussi une interface ou une classe abstraite qui va étendre notre classe Composant et par la suite on peut implémenter différent décorateur

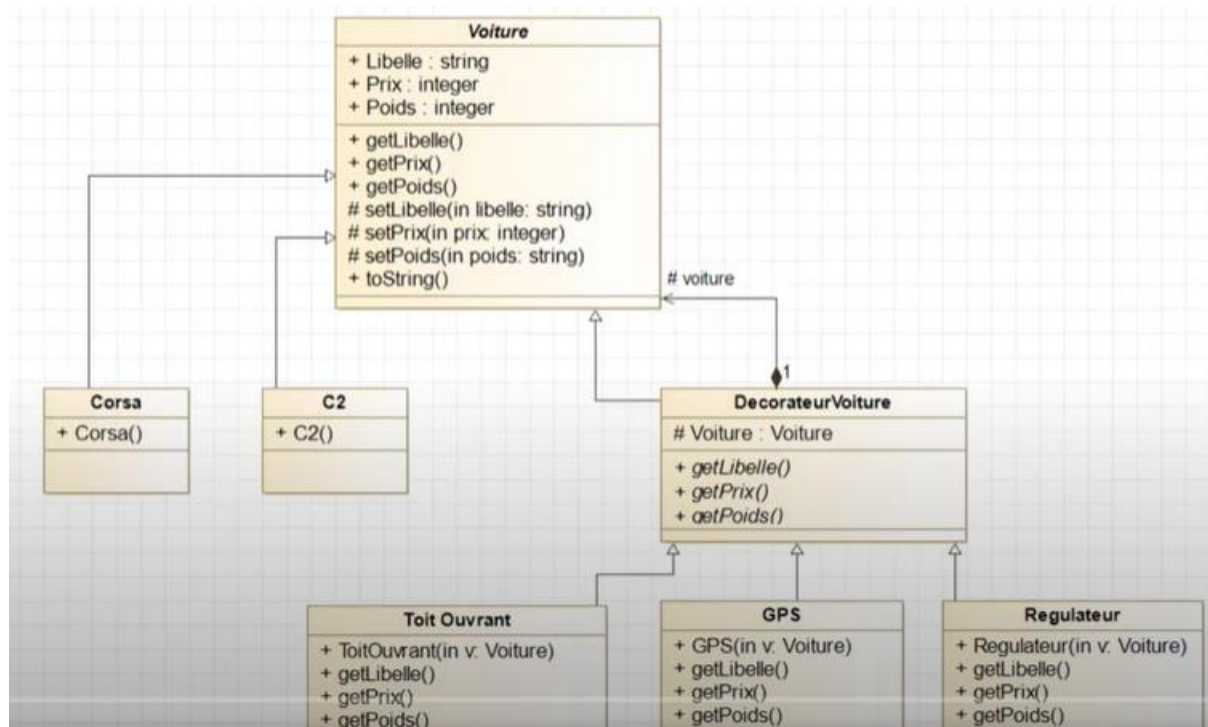
Prof : On part donc d'un héritage classique pour transposer vers un Design pattern Decorator

**Rmq** : On a une composition de Composant dans Décorateur. Ainsi Décorateur va non seulement redéfinir les méthodes de Composant mais aussi définir une *liste de Composant* dans *Décorateur* que nous pourrions implémenter à l'aide d'un Set ou d'une List (*pas dans le cas du prof*)

**Exemple :**



Application du Design pattern Décorateur



**Prof :** On peut remarquer les décorateurs offre une alternative souple à l'héritage

**Prof :** Ici Voiture devient la classe Abstraite

Corsa et C2 Deviennent les composants concrets

DécorateurVoiture : Devient le Décorateur (Toujours l'élément qui résulte de la spécialisation de chaque classe) et enfin les décorateurs Concret deviennent Toit Ouvrant, GPS et Régulateur. Au travers de ce schéma, on peut avoir donc différent Décorateur.

**Par exemple** une corsa avec un Toit ouvrant, ou un C2 avec un toit ouvrant...

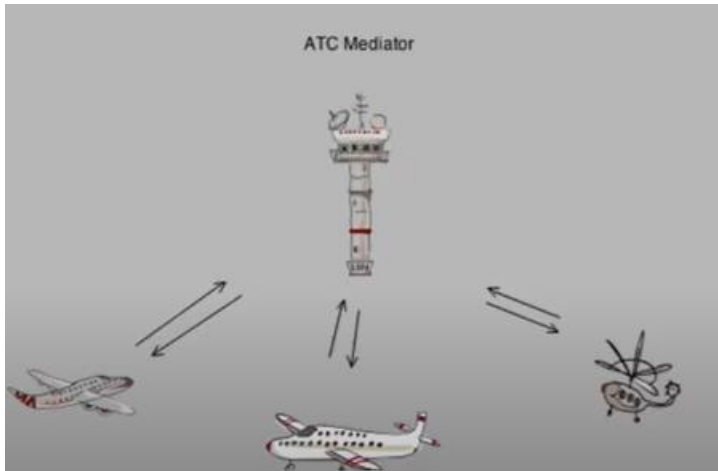
Ainsi ce Design pattern est très pratique mieux que l'héritage

**Important :** Lors de l'implémentation, il faut que le Décorateur soit lui-même abstrait (**On ne manipule pas le Décorateur mais ses classe dérivées**) pour permettre aux classes dérivées d'implanter ses méthodes à leur guise c'est le but.

## Mediateur

Le Design pattern Médiateur favorise les couplages faibles, en dispensant les objets d'avoir à faire référence explicite les uns aux autres ; de plus, il permet de modifier une relation indépendamment des autres.

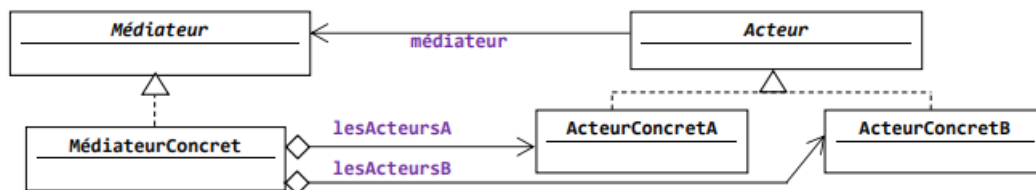
On s'appuie sur la problématique de l'aéroport. Dans lequel il devient pénible de faire communiquer chaque avion entre autres. La solution est de faire intervenir la tour de contrôle qui établira la communication entre deux avions



*Ce schéma illustre bien celui du Design pattern Médiateur*

Ainsi si on est amené à concevoir 20 objets dans le code et qu'on veut que chaque objet communique entre autres. Il va falloir implémenter un médiateur pour faire communiquer chacun de ces objets

**Rmq :** On peut remarquer que ce Design pattern restreint la communication entre ces différents objets et leur permet de communiquer via un Médiateur qui s'occupe de l'interaction entre les différents objets



**Rmq :** Pour modifier la façon dont les objets communiquent ensemble, il suffira de modifier le Médiateur

**Prof :** On peut donc déduire que le Médiateur définit les méthodes pour envoyer et recevoir des messages entre les objets.

**Exp :**

**Avantages :** Principe de responsabilité unique et Principe ouvert/fermé

Il permet de réduire la complexité des interactions entre les objets en centralisant la communication dans un seul objet, appelé médiateur

Il peut aider à maintenir l'indépendance des objets en les empêchant de se connaître mutuellement et en leur permettant de communiquer uniquement via le médiateur.

**Inconvénient :**

Il peut ajouter de la complexité au code en introduisant un nouvel objet.

Si le médiateur est trop complexe, il peut devenir une source de problèmes à lui seul.

Prof : Nous pouvons ajouter de nouveau médiateur sans avoir à modifier les composants mis en place en centralisant la logique ce qui nous fait vérifier le principe ouvert fermé

## Méthode

1. Identifier une collection d'objets en interaction qui bénéficieraient d'un couplage faible
2. encapsuler toutes les interactions d'objets dans une classe de médiateur
3. Restructurer toutes les classes de collègues pour n'interagir qu'avec l'objet médiateur
4. Reconfigurer dynamiquement les interactions d'objets du mediator si nécessaire

### Différence entre le médiateur et l'observateur

Dans l'observateur la dépendance entre les objets est telle que lorsqu'un objet change d'état, tous ses dépendants sont notifiés et mis à jour automatiquement

Dans le Design Pattern Médiateur les objets n'ont pas la possibilité de se référer les uns aux autres sans l'intermédiaire du Médiateur

**Prof :** Pour rappel Nous avons vu que les principes SOLID tendent notamment à diminuer le couplage entre modules. Le Patron de Conception Médiateur relève de cette préoccupation

### Mise en œuvre du principe de simple responsabilité et mécanisme pour Déporter une méthode

Le principe de simple responsabilité d'une classe ("Single Responsibility Principle"), est un principe de conception selon lequel chaque classe d'un système doit avoir **une seule responsabilité** et que cette responsabilité doit être entièrement **encapsulée par la classe**. Dès lors, si une classe a plusieurs responsabilités, il est recommandé de la découper en plusieurs classes, chacune avec sa propre responsabilité.

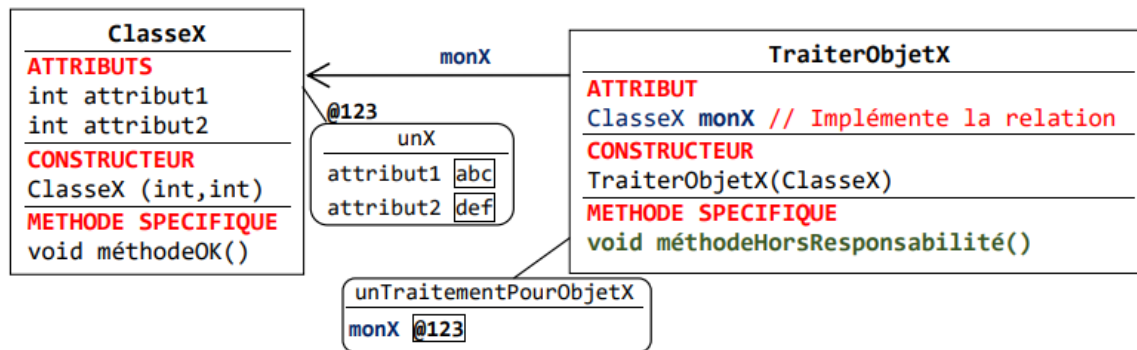
**Prof :** Il permet de créer des classes plus simples et plus facile à maintenir

Exp :

Etant donné le diagramme de classe ci-dessous dans lequel la ClasseX comporte entre autres, deux attributs et une méthodeHorsResponsabilité() qui n'a pas lieu d'être dans cette classe car elle ne caractérise pas l'objet.

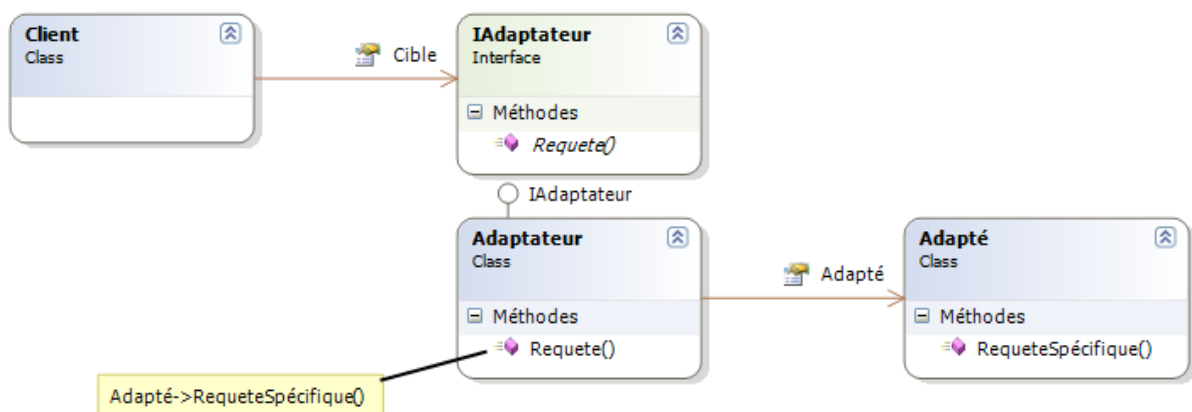
ClasseX
<b>ATTRIBUTS</b>
int attribut1
int attribut2
<b>CONSTRUCTEUR</b>
ClasseX (int,int)
<b>METHODES</b>
void méthodeOK()
void méthodeHorsResponsabilité()

Pour satisfaire au principe de Simple Responsabilité, il s'agit de retirer la **méthodeHorsResponsabilité()** et la déporter dans une classe destination **TraiterObjetX**, conformément au schéma de classe UML ci-dessous.



## Design Pattern Adapteur

Le patron de conception **Adaptateur** ou **Adapter** ) permet de faire fonctionner ensemble des classes qui n'auraient pas pu fonctionner ensemble en raison de leur interface **incompatible**. Le patron Adapter consiste à créer une **classe d'adaptateur** qui implémente l'interface souhaitée et qui contient une référence à l'objet qui possède l'**interface incompatible** cad (Adaptation). Lorsque l'adaptateur est appelé, il redirige les appels vers l'objet incompatible.



Prof : Il est question de permettre à la classe adaptateur qui permet à un objet de la classe Adaptation de fonctionner comme si il était de la classe Adapteur.

Lorsque la méthode **operation()** de **Adaptateur** est appelé, elle redirige l'appel vers la méthode **noperationAdaptate()** de la classe **Adaptation**.

**Prof :** Que fait le client

// Il demande à un objet qui implémente la cible, d'exécuter l'opération

**Adaptation** uneAdaptation = new **Adaptation**() ;

**ICible** cible = new **Adaptateur**(uneAdaptation) ; // Prof c'est l'injection de dépendance



cible.operation() ;

**Prof :** Note du 09/01/2023

IArchiver <<Abstract >>

Sauvegarder()

ArchiverMySql

Sauvegarder()

ArchiverMongoDB

Sauvegarder()

ReservationVoiture

```
Void archiverReservation(){
ArchiverMySql ar = new ArchiverMySql() ;
ar.sauvegarder() ; // Appel de la methode sauvegarde de la classe ArchiverMySql
}

Void archiverReservation2(){
ArchiverMongoDB ar = new ArchiverMongoDB() ;
ar.sauvegarder() ; // Appel de la methode sauvegarde de la classe ArchiverMongoDB
}
```

Main

```
ReservationVoiture uneReservation = new ReservationVoiture();
uneReservation.archiverReservation() //
```

**Prof :** Rappel Composite, Singleton et Mediateur



Mettez sur votre feuille A4 quelque chose de cibles très efficaces

Composite, ArrayList et injection de dépendance

**Prof :** On ne verra pas le Patron de Methode .

Je vais vous présenter des bidules et ce serait à vous de me répondre par vos code java

Prof : Cette série de concept rentre en complexité

### Dépendance

Relation entre deux classe . Dans laquelle une classe declare un attribut de l'autre classe pour pouvoir l'utiliser dans

### Injection de dependance

Transmission de parametre via un constructeur

### Inversion de dependance

Le fait de transmette un parametre a un objet via un set . Mais le mieux c'est l'injection cad via un constructeur

---

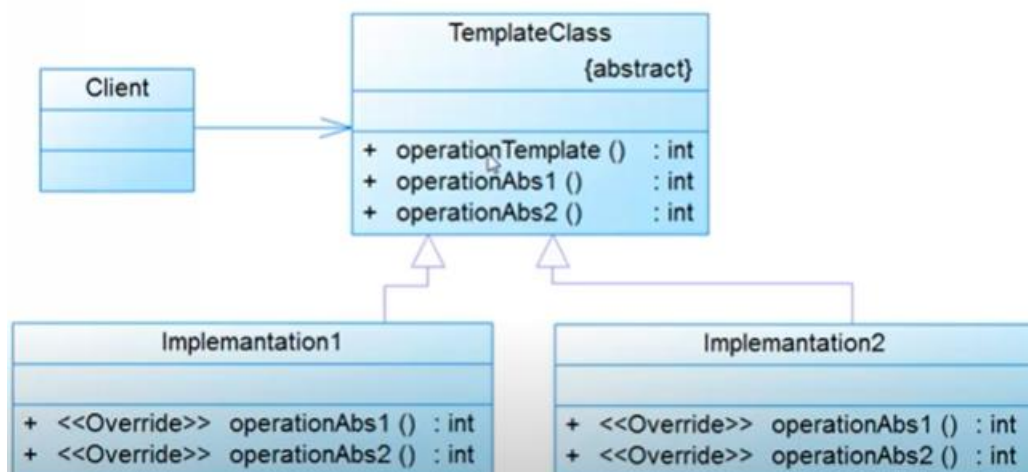
**Prof :** Ca ca ne concerne pas le devoir

Conteneur d'injection de dependance

Auto Wiring

### Design Pattern PATRON DE METHODE (Hors Programme)

Patron de méthode (Template method) : Définit le squelette de l'algorithme d'une opération, en déléguant le traitement de certaines étapes à des sous-classes. Le patron de méthode permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de l'algorithme



TemplateClass : est une classe abstraite dans laquelle on définit le operationTemplate() (qui est une Template méthode contenant le squelette d'algorithme dont les parties complémentaires seront implémentés par les classes filles Implementation1 et Implementation2

**Prof :** En d'autres termes pour que le opeationTemplate() fonctionne, il faudrait que les classes fille implémentent operationAbs1() et operationAbs2()

**N.B :** Les méthodes etape1() et etape2() n'ont pas de vocation en dehors de la méthode operationTemplate() donc ce serait important de ne pas permettent aux client d'utiliser ses méthodes là C'est pourquoi il faut les déclarer proctected

**Prof :** Des lors l'exécution du même algorithme dans le main produira des résultats différents en sortie

Rmq : Le but du design pattern Template de méthode est de permettre à une classe de définir le "squelette" d'un algorithme, en déléguant certaines étapes à ses sous-classes.

Le design pattern Stratégie, quant à lui permet de définir une famille d'algorithmes de les encapsuler chacun dans une classe distincte et de rendre ces algorithmes interchangeables à l'intérieur d'une même classe