\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Design patterns Java**

***Les plus utilisés.***

***Disparaissent depuis CDI.***

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Sommaire**

[I. Introduction 3](#_Toc270084765)

[II. Principes de base 4](#_Toc270084766)

[III. Pattern Stratégie 5](#_Toc270084767)

[IV. Pattern Décorateur 6](#_Toc270084768)

[V. Pattern Fabrication 7](#_Toc270084769)

[VI. Pattern Observateur 8](#_Toc270084770)

[VII. Pattern Singleton 9](#_Toc270084771)

[VIII. Pattern mixte Stratégie et Décorateur : exemple d’utilisation sur le projet PTL 10](#_Toc270084772)

[IX. Pattern command 11](#_Toc270084773)

[X. Pattern Adaptateur 13](#_Toc270084774)

[XI. Pattern adaptateur et facade 14](#_Toc270084775)

[XII. Pattern patron de methode 15](#_Toc270084776)

[XIII. Pattern iterateur et composite 16](#_Toc270084777)

[XIV. Pattern état 17](#_Toc270084778)

[XV. Pattern proxy 18](#_Toc270084779)

# Introduction

Présentation de quelques design pattern java les plus usités pour:

-Attribuer à des objets des actions facilement interchangeables. (Strategie)

- Etendre à l’infini des comportements à un objet (Decorateur)

- Créer instancer des objets sans dépendre de ces objets. (Fabrication)

- Appeler différentes méthodes dans différents objets sans avoir à modifier les signatures de ces méthodes ni la classe qui lance l’appel à ces méthodes. (Adaptateur)

- Mettre à jour les données d'un grand nombre d'objets en même temps à un instant donné. (Observateur)

- Créer des objets pour lesquels on peut modifier leur comportement à volonté et enrichir chacun de ces comportements par l’exécution préalable d’objets de même type. (Mixte entre Decoration et Stratégie)

- Exécuter des actions sans dépendre de qui et quand elles vont être exécutées en faisant du multi-threading avec un seul thread et avec possibilité d’implémenter undo-redo (Command)

# Principes de base

Ci-dessous sont des lignes directrices vers lesquelles tendre et non des règles que nous devons appliquer tout le temps :

- Identifier les aspects de votre application qui varient (en rouge dans les schémas ci-dessous) et séparez-les de ceux qui demeurent constants.

- Les classes doivent être ouvertes à l’extension mais fermées à la modification

- Programmer avec des interfaces et non avec des classes implémentant

- Dépendez d’abstraction, ne dépendez pas de classes concrète : Nos composants de hauts niveaux et de bas niveaux doivent dépendre les deux d’abstractions. (couplage faible)

- Aucune variable ne doit contenir une référence à une classe concrète

- Aucune classe ne doit sous-classer une classe concrète

- Aucune classe ne doit redéfinir une méthode de sa classe abstraite

- Préférer la composition à l’héritage : Un canard A UN comportement de vol : nous avons deux classes distinctes et non pas une seule qui hérite de comportement

# Pattern Stratégie

**Quand on veut attribuer à des objets des actions facilement interchangeables.**

Séparation entre la classe objet et les différents types d’actions que l’objet peut effectuer en fonction de tels ou tels paramètres.

Depuis java8 je peux faire :

methodeAcceuilantUnConsumer((x)->{  
 unemethodeaexecuter();  
 uneautremethodeaexecuter();  
});

**public void** methodeAcceuilantUnConsumer(Utils.ThrowingConsumer o){  
 (…)  
 o.accept(**null**);

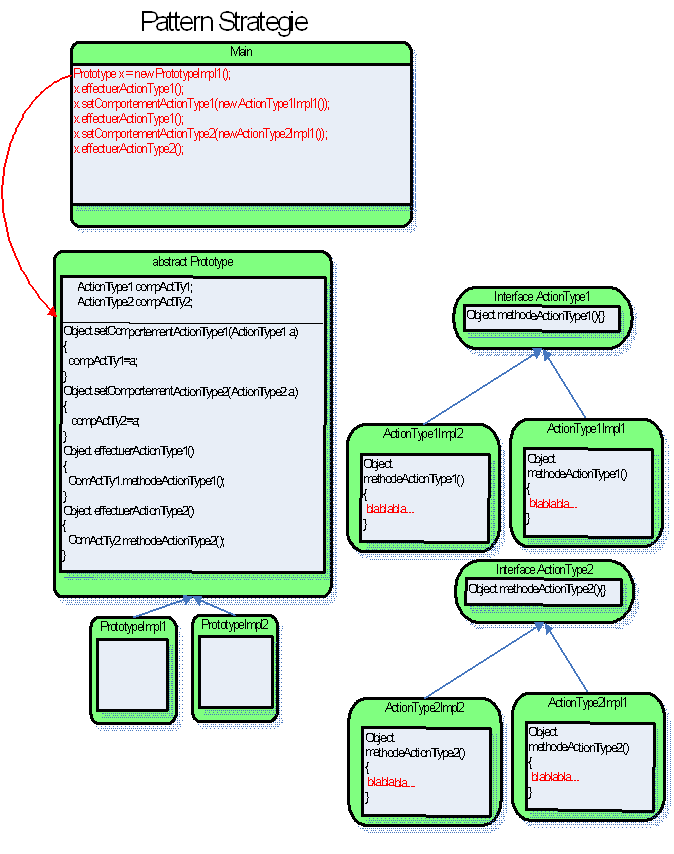
(…)  
 }

@FunctionalInterface  
**public interface** ThrowingConsumer<T, E **extends** Exception> {  
 **void** accept(T t) **throws** E;  
}

OU BIEN :

x.effectuerActionType1(ActionType1Impl2::methodeActionType1) ; (AVERIFIER)

avant :



# Pattern Décorateur

**Quand on veut étendre à l’infini des comportements à un objet**

Depuis java8 : txTemplate.executeInTransaction(()->saveRequest()) ;

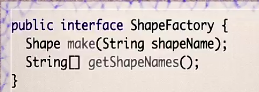
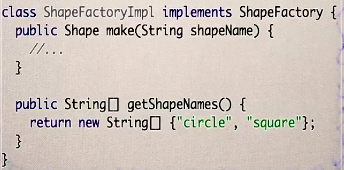
**final double** turnover = 100000;  
 **double** profit = FluentDecorator  
 .*from*(turnover)  
 .with(Expenses::*getTransportExpenses*)  
 .with(Expenses::*getOperatingExpenses*)  
 .with(Expenses::*getDeductibleTaxes*)  
 .with(Expenses::*getRemuneration*)  
 .with(Expenses::*getExceptionalExpenses*)  
 .calculate();  
 Assertions.*assertEquals*(32600,profit);  
 }  
**public static final class** FluentDecorator<T> {  
  
 **private final** T **value**;  
 **private final** Function<T, T> **function**;  
  
 **private** FluentDecorator(**final** T value, Function<T, T> function) {  
 **this**.**value** = value;  
 **this**.**function** = function;  
 }  
  
 **public static** <T> FluentDecorator<T> from(**final** T value) {  
 Objects.*requireNonNull*(value);  
 **return new** FluentDecorator<>(value, Function.*identity*());  
 }  
  
 **public** FluentDecorator<T> with(**final** Function<T, T> otherFunction) {  
 **return new** FluentDecorator<>(**this**.**value**, **function**.andThen(otherFunction));  
 }  
  
 **public** T calculate() {  
 **return this**.**function**.apply(**value**);  
 }  
 }  
  
 **private static class** Expenses  
 {  
 **public static** Double getTransportExpenses(Double aDouble){**return** 1.1;}  
 **public static** Double getOperatingExpenses(Double aDouble){**return** 1.2;}  
 **public static** Double getDeductibleTaxes(Double aDouble){**return** 3.1;}  
 **public static** Double getRemuneration(Double aDouble){**return** 4.2;}  
 **public static** Double getExceptionalExpenses(Double aDouble){**return** aDouble+5.8;}  
 }

Avant : mais avant la technique reste qd meme plus puissante que ci dessus : ci-dessus utile que pour faire des calcules simples



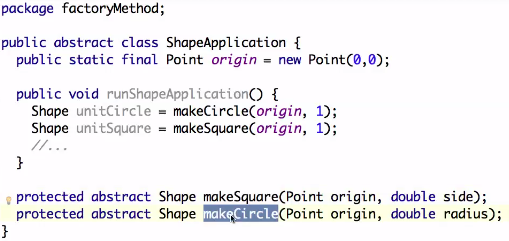
# Pattern Fabrication (Factory : ne plus utiliser. Est remplacé par @Specializes remplace Injection de dependance de xml Spring))

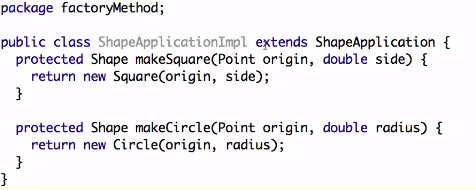
**A l’initialisation, quand on veut que des objets différents effectuent systématiquement les mêmes méthodes.**



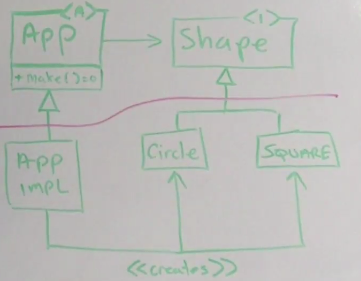
# FactoryMethod

La fabrication d’objets multiples se fait directement dans une classe applicative sans passer par une fabrique. La classe applicative ne dépend toujours pas des objets crées.

 \*



Mais plutôt que de faire plein de méthodes faire comme Patern Fabrication : un argument non typé défini l’objet à retourner.



# Pattern Observateur

**Quand on veut qu’une infinité d’objets mette à jour leurs données en même temps à un instant donné.**



# Pattern Singleton

**Quand on veux que plusieurs objets/threads ne se réfèrent qu’à une seule et même classe (contenant par exemple des données.)**



**Nous pouvons utiliser une classe statique au lieu d’implémenter ce pattern. Mais cela peut générer des bogues subtils liés à l’ordre d’initialisation, difficiles à détecter. Attention, si il y a plusieurs chargeurs de classe, il peut y avoir plusieurs instances de singleton créées. Une façon de contourner ce problème est de spécifier soi même le chargeur de classe. Sous classer un singleton est déconseillé. Ne pas faire trop de singletons dans une application. (Voir plutôt technique du Holder dans apprentissage J2ee)**

# Pattern mixte Stratégie et Décorateur : exemple d’utilisation sur le projet PTL

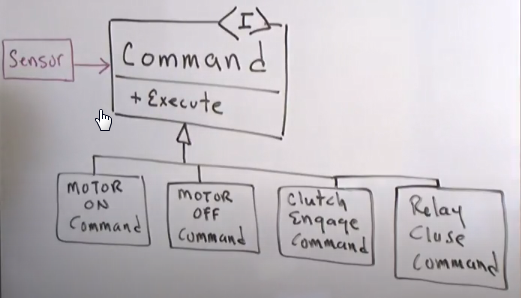
**Ce pattern permet de créer des objets pour lesquels on peut modifier leur comportement à volonté et enrichir chacun de ces comportements par l’exécution préalable d’objets de même type.**

Exemple de diagramme mixte pour l’application PTL:



# Pattern command : remplacé par MessageDrivenBeans et CDI events

**Ce pattern permet de placer une abstraction entre un objet qui fait des commandes et les objets qui executent la commande : possibilité d’annuler la dernière commande passée. (und, redo et peut servir aussi a executer plein de threads sans risque de stackoverflow)**

****

Sensor est mon interactor(Controller)

UNDO REDO :

Command peut avoir un champ privé contenant ce qu’il a executé.

Ajouter une methode Undo et Redo à <I>Command et stocker les commandes executées dans une liste permet d’ajouter la fonctionnalité Undo et Redo.

A LA CHAINE DANS UN SEUL THREAD

Dans le main, je peux mettre mes commandes dans une liste :

Controller

List<Command> commands = new ArrayList() ;

Commands.add(new CommandX()) ;

Commands.add(new CommandY()) ;

Commands.add(new CommandZ()) ;

While (commands.size()>0)

{

Command cmd = commands.get(0);

commands.remove(0) ;

cmd.execute();

}

Il execute toutes les commandes de la liste. Si on ne met qu’une seule commande OU PLUSIEURS DIFFERENTES CommandX() dans la liste et on rajoute ça dans cette commande :

execute()

{

If (x){ // si j’ai appuyé sur un bouton par exemple

Main.commands.add(new CommandZ() ;

}

Else {

Main.commands.add(this) ;

}

}

permet d’executer les commandes à la suite dans un seul thread. Pas de stackoverflow possible. Execution asynchrone. (Actor’s Model)

# Pattern Adaptateur

**Pour « brancher » une classe sur tout type d’objets.**

Comme pour les différents modèles de prises électriques de différents pays. Il y a un adaptateur par pays pour brancher son portable partout dans le monde.



# Pattern adaptateur et facade

**Non fait.**

Non Fait

# Pattern patron de methode

**Non fait.**

Non Fait

# Pattern iterateur et composite

**Non fait.**

Non Fait

# Pattern état

**Non fait.**

Non Fait

# Pattern proxy

**Non fait.**

Non Fait

# CleanCoder Pattern pour app type webapp

