# Les design pattern de notre projet

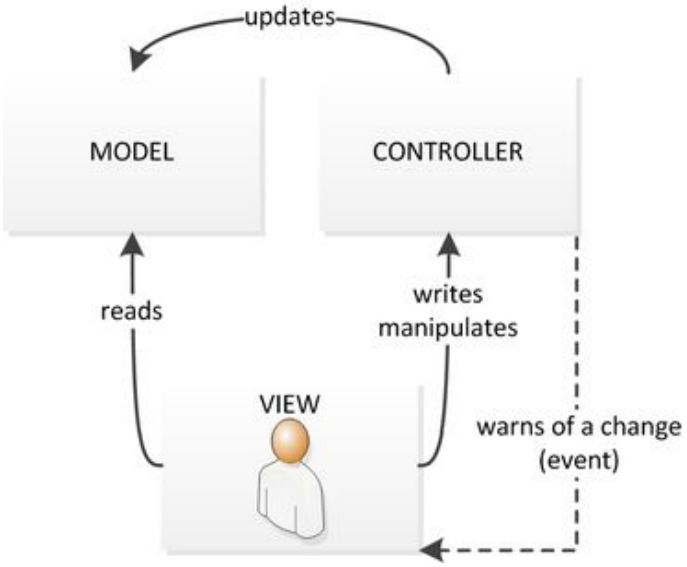
## MVC

Modèle-vue-contrôleur (MVC) est un motif d'architecture logicielle destiné aux interfaces graphiques. Le motif est composé de trois types de modules ayant trois responsabilités différentes : les modèles, les vues et les contrôleurs.

- Un modèle contient les données à afficher.
- Une vue contient la présentation de l'interface graphique.
- Un contrôleur contient la logique concernant les actions effectuées par l'utilisateur.

L'utilité de MVC est permettre de séparer l'affichage des informations, les actions de l'utilisateur et l'accès aux données.MVC permet également de concevoir des applications de manière claire et efficace grâce à la séparation des intentions. Les opérations de maintenance et de mises à jour sont fortement simplifiées.

**On utilise ce design pattern car il est obligatoire dans ce projet.**

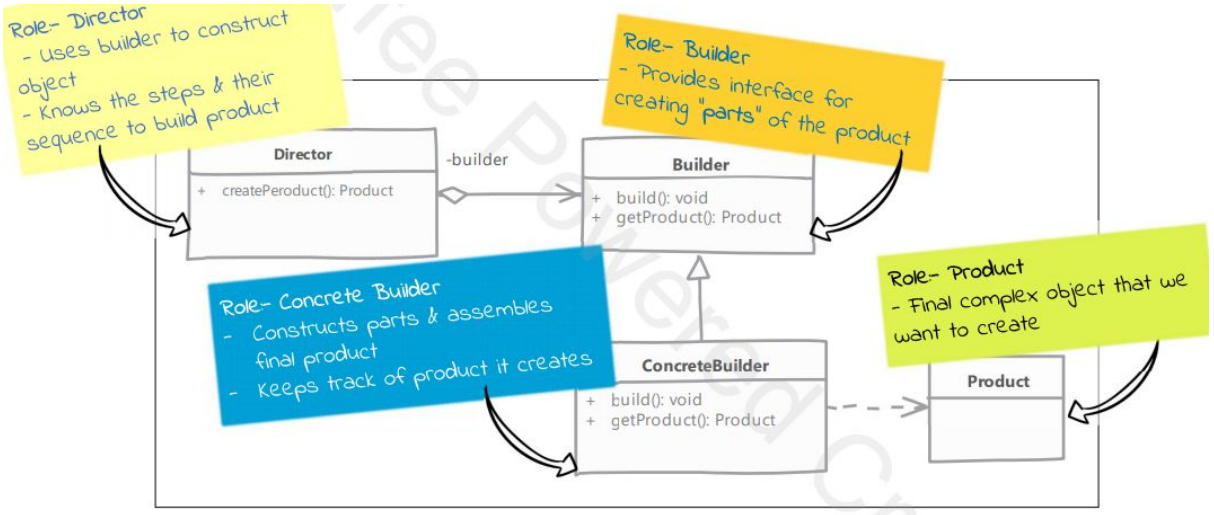



## Builder

Ce patron sépare le processus de construction d'un objet du résultat obtenu. Permet d'utiliser le même processus pour obtenir différents résultats. C'est une alternative au pattern factory. Au lieu d'une méthode pour créer un objet, à laquelle est passée un ensemble de

paramètres, la classe factory comporte une méthode pour créer un objet - le builder. Cet objet comporte des propriétés qui peuvent être modifiées et une méthode pour créer l'objet final en tenant compte de toutes les propriétés. Ce pattern est particulièrement utile quand il y a de nombreux paramètres de création, presque tous optionnels.

**On utilise builder pour cuisiner les différentes recettes.**



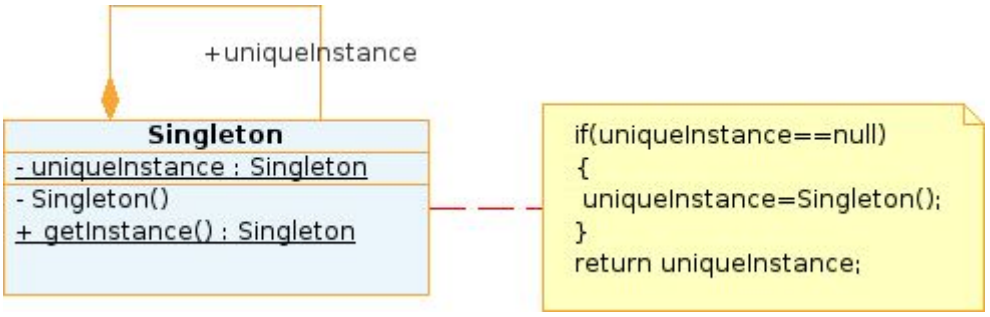

## Singleton

Ce patron vise à assurer qu'il n'y a toujours qu'une seule instance d'une classe en fournissant une interface pour la manipuler. L'objet qui ne doit exister qu'en une seule instance comporte une méthode pour obtenir cette unique instance et un mécanisme pour empêcher la création d'autres instances.

L'objectif est de restreindre l'instanciation d'une classe à un seul objet. Il est utilisé lorsqu'on a besoin exactement d'un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec peu d'objets qu'avec beaucoup d'objets similaires.

On implémente le singleton en écrivant une classe contenant une méthode qui crée une instance uniquement s'il n'en existe pas encore. Sinon elle renvoie une référence vers l'objet qui existe déjà. Dans beaucoup de langages de type objet, il faudra veiller à ce que le constructeur de la classe soit privé, afin de s'assurer que la classe ne puisse être instanciée autrement que par la méthode de création contrôlée.

**On utilisera Singleton pour mettre en place le DP Factory.**



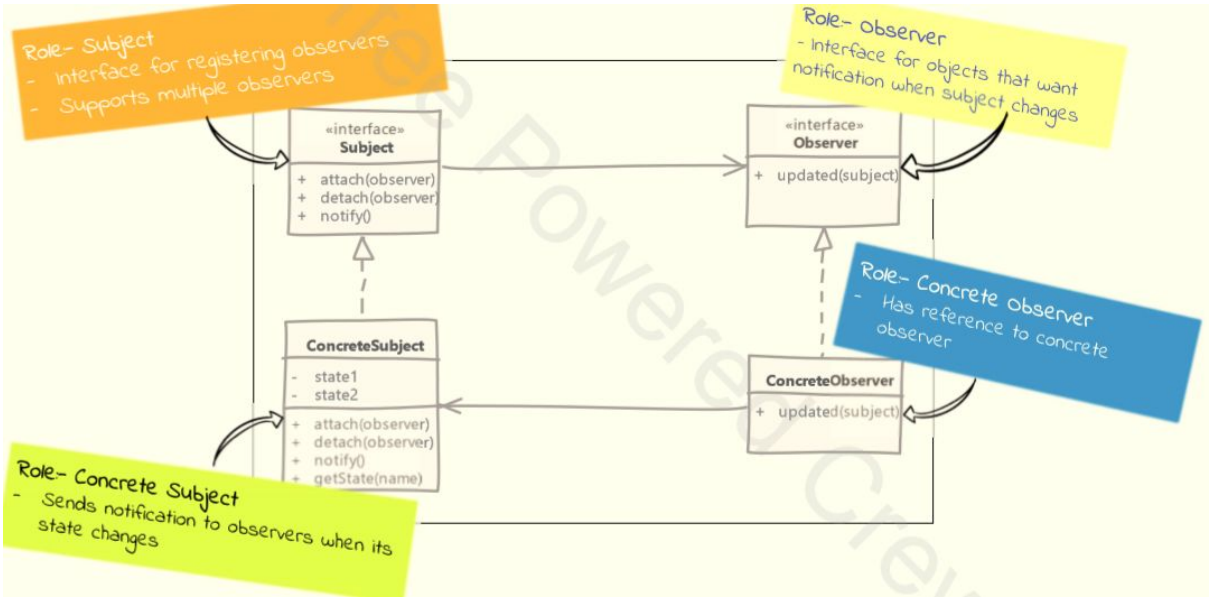

## Observer

Ce patron établit une relation un à plusieurs entre des objets, où lorsqu'un objet change, plusieurs autres objets sont avisés du changement. Dans ce patron, un objet le sujet tient une liste des objets dépendants des observateurs qui seront avertis des modifications apportées au sujet. Quand une modification est apportée, le sujet émet un message aux différents observateurs. Le message peut contenir une description détaillée du changement. Dans ce patron, un objet observer comporte une méthode pour inscrire des observateurs. Chaque observateur comporte une méthode Notify. Lorsqu'un message est émis, l'objet appelle la méthode Notify de chaque observateur inscrit.

Les notions d'observateur et d'observable permet de limiter le couplage entre les modules aux seuls phénomènes à observer. Il permet aussi une gestion simplifiée d'observateur multiples sur un même objet observable. Il est recommandé dès qu'il est nécessaire de gérer des évènements, quand une classe déclenche l'exécution d'une ou plusieurs autres.

Dans ce patron, le sujet observable se voit attribuer une collection d'observateurs qu'il notifie lors de changements d'états. Chaque observateur concret est chargé de faire les mises à jour adéquates en fonction des changements notifiés. Ainsi, l'observé n'est pas responsable des changements qu'il impacte sur les observateurs.

**On utilisera le design pattern observer pour notifier lorsque le matériel est sale et a besoin d'être ramassé pour être lavé par le plongeur.**




## Strategy

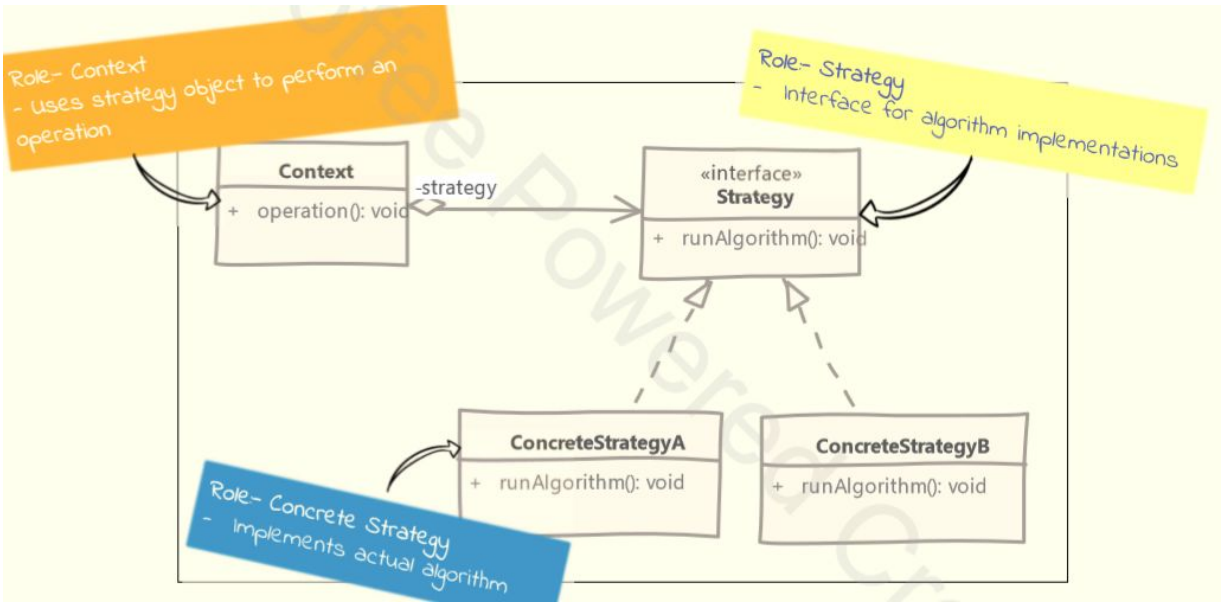
Design Pattern de type comportemental grâce auquel des algorithmes peuvent être sélectionnés à la volée au cours du temps d'exécution selon certaines conditions.

Strategy est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application. Une famille d'algorithmes est encapsulée de manière qu'ils soient interchangeables.

On utilise Strategy dès lors qu'un objet peut effectuer plusieurs traitements différents.

Il comporte trois rôles : le contexte, la stratégie et les implémentations. La stratégie est l'interface commune aux différentes implémentations (généralement une classe abstraite). Le contexte est l'objet qui va associer un algorithme avec un processus.

**On utilise stratégie car on a plusieurs acteurs avec des comportements différents.**




## Factory

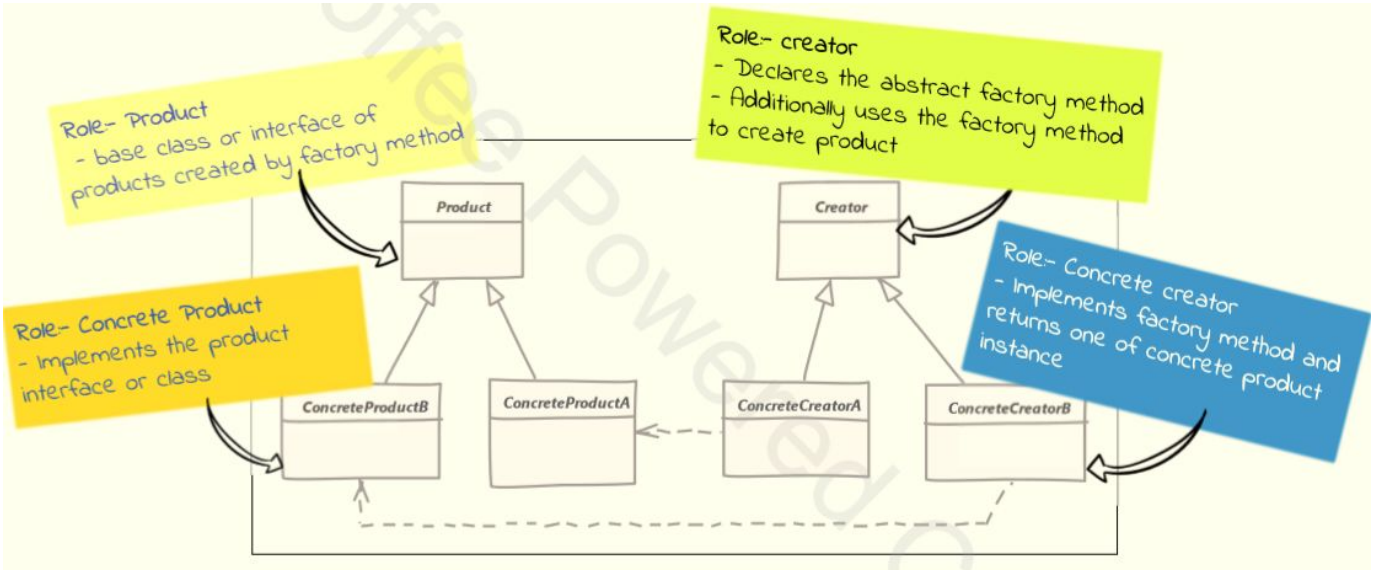
La fabrique (factory method) est un patron de conception créationnel utilisé en programmation orientée objet. Elle permet d'instancier des objets dont le type est dérivé d'un type abstrait. La classe exacte de l'objet n'est donc pas connue par l'appelant.

Plusieurs fabriques peuvent être regroupées en une fabrique abstraite permettant d'instancier des objets dérivant de plusieurs types abstraits différents.

Les fabriques étant en général uniques dans un programme, on utilise souvent le patron de conception singleton pour les implémenter.

Le patron factory (en français fabrique) est un patron récurrent. Une fabrique simple retourne une instance d'une classe parmi plusieurs possibles, en fonction des paramètres qui ont été fournis. Toutes les classes ont un lien de parenté, et des méthodes communes, et chacune est optimisée en fonction d'une certaine donnée

**On utilise Factory pour mettre en place différentes instanciations à partir d'une classe abstraite (acteur).**


Role:- Product
- base class or interface of
products created by factory method
Role:- creator
- Declares the abstract factory method
- Additionally uses the factory method
to create product
Product
Creator
Role:- Concrete creator
- Implements factory method and
returns one of concrete product
instance
Role:- Concrete Product
- Implements the product
interface or class
ConcreteProductB
ConcreteProductA
ConcreteCreatorA
ConcreteCreatorB

.