# Les Design Patterns

1. Design Pattern MVC

**Principe**

Le Design Pattern **MVC** pour **Modèle-Vue-Contrôleur** fait parti des plus connu et reste particulièrement presqu’incontournable dans le monde de développement logiciel.

Il permet notamment de séparer bien organiser le code source en définissant les fichiers à créer et leurs différents rôles.

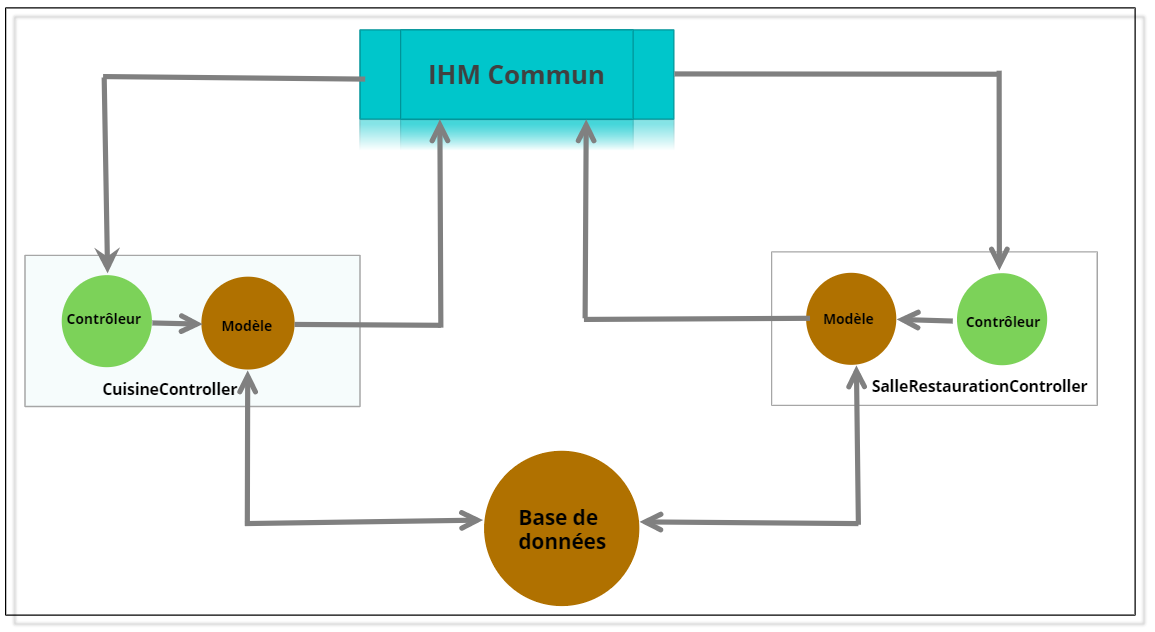
Le but de MVC est justement de séparer la logique du code en trois parties que l'on retrouve dans des fichiers distincts :

* **Vue :** La vue représente la partie de l’application que l’utilisateur a sous ses yeux. Elle est donc implémentée par une IHM. Elle s’occupe des interactions avec l’utilisateur : présentation, saisie et validation des données.
* **Modèle :** Le modèle est la partie de l’application qui exécute la logique métier, c’est-à-dire le code applicatif de l’application. Il la responsabilité de : récupérer les données, et de les manipuler conformément à l’application (le traitement, la validation, l’association et beaucoup d’autres tâches).
* **Contrôleur :** C’est la couche intermédiaire qui permet d’interagir entre les deux (2) autres couches du modèles (Vue et Modèle). Il a pour principale rôle de **Superviser** l’application. Le contrôleur se charge de savoir quelles données doivent être chargées depuis la base de données, la vérification du niveau d’authentification, appeler le **modèle** correspondant et retourner la **vue** adéquate.

Comme dit ci-dessus, un des objectifs est la **séparation des différentes couches** constituant une application interactive, de manière à simplifier la gestion de chacune. On pourra donc ainsi modifier par exemple le code applicatif sans affecté la vue de l’application, ou vis versa.

**Application**

Dans notre projet nous avons deux contrôleurs principaux dans l’application **CuisineController** et **SalleRestaurationController** possédant chacun aussi un « **modèle**» et partagent entre une même **« Vue »** et une base même base de données.



1. Design Pattern Observer

**Principe**

Dans un design pattern **observer** un [objet](https://en.wikipedia.org/wiki/Object_(computer_science)#Objects_in_object-oriented_programming) , nommé **sujet**, maintient une liste de ses dépendants, appelés **observateurs**, et les notifie automatiquement de tout changement d'état, généralement en appelant l'une de leurs [méthodes](https://en.wikipedia.org/wiki/Method_(computer_science)).

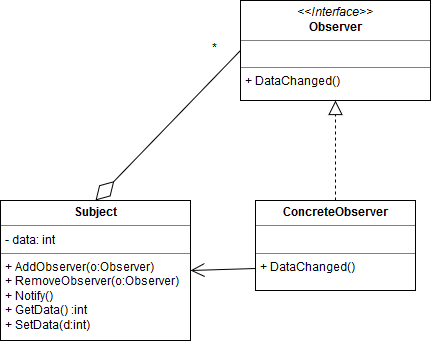


Figure 1 – Exemple Schéma explicatif du pattern Observer

**Source image :** <https://en.wikipedia.org/wiki/Observer_pattern>

**Application**

Dans ce projet nous l’utiliserons dans la partie « **vue** » du MVC. A chaque fois qu’une modification est présentée, venant du **SalleRestaurantControler** ou du **CuisineController**, la vue sera notifiée et une mise à jour sera effectuée.

1. Design Pattern Strategy

**Principe**

Ce design pattern est nécessaire lorsqu’on souhaite répondre à la question comment faire pour réaliser différentes opérations avec un seul et même objet ? C’est un design Pattern de type comportemental grâce auquel des algorithmes peuvent être sélectionnés à la volée au cours du temps d'exécution selon certaines conditions.

Le patron de conception stratégie est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application.

L'idée principale est de pouvoir mettre en place une certaine stratégie (par exemple, une méthode qui réalise une certaine action) et de pouvoir changer dynamiquement de stratégie au runtime. On créer donc une interface de base, appelée ici Strategy et on y ajoute une méthode qui sera la méthode qui applique notre stratégie.



Figure 2 – Exemple Schéma explicatif du Pattern Strategy

**Source image :** <https://fr.wikipedia.org/wiki/Strat%C3%A9gie_(patron_de_conception)>

**Application**

Dans ce projet particulièrement nous l’utiliserons dans le menu, puisqu’il y a plusieurs des recettes pour les entrées, les plats et les desserts, le design pattern strategy permettra de choisir les recettes en fonction des ingrédients et d’autres facteurs.

1. Design Pattern Builder

**Principe**

Le design pattern builder sert à séparer la construction d'un objet complet de sa représentation afin que le même processus de construction puisse créer différentes représentations. Ainsi donc on pourra avoir différentes types et représentations d'un objet en utilisant le même code de construction.

Il est particulièrement utile donc lorsqu’on souhaite faire varier la représentation interne d'un produit, encapsuler le code pour la construction et la représentation d’un objet, ou encore contrôler les étapes du processus de construction d’un objet.

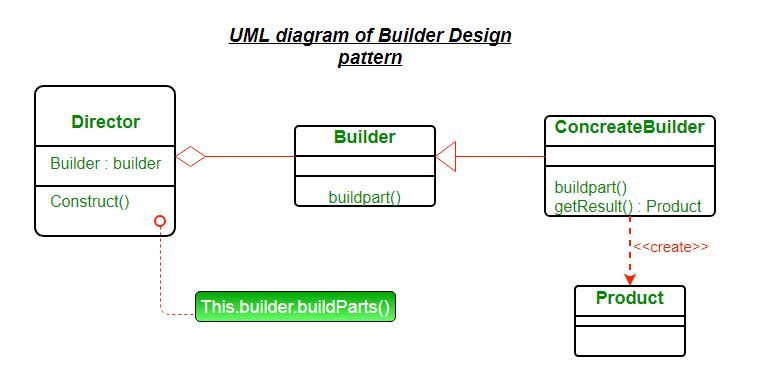


Figure 3 – Exemple Schéma explicatif du Pattern Builder

**Source Image :** <https://media.geeksforgeeks.org/wp-content/uploads/uml-of-builedr.jpg>

**Application**

Dans notre projet il sera utilisé dans la prise des commandes. Les commandes étant des process qui peuvent être représentés en différentes façons, conformément à une entrée, un plat ou un dessert, ce design pattern va donc nous aider à avoir le même process de construction pour toutes les commandes.

1. Design Pattern Singleton

**Principe :**

Le design pattern Singleton est nécessaire lorsqu’on a besoin d'une et une seule instance d'un objet, d’une initialisation tardive et un accès global sont nécessaires.

Il permet de restreindre l’instanciation d’une classe à un seul objet et lui fournir un point d’accès globale.

Son utilisation est assez simple, pour l’implémenter, il faudra juste :

* Rendre le constructeur par défaut privé, pour empêcher d'autres objets d'utiliser l’opérateur ***new*** sur la classe ;
* Créer une méthode de création statique qui agit comme un constructeur. Sous cette méthode appeler le constructeur privé pour créer un objet et l'enregistrer dans un champ statique. Tous les appels suivants à cette méthode renvoient l'objet mis en cache.

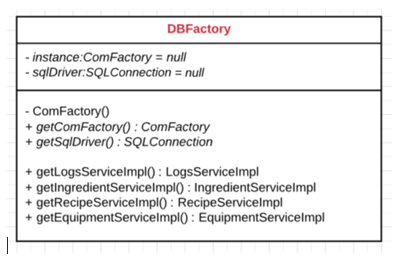


Figure 4Exemple schéma explicatif du pattern Singleton

**Source image :**  <https://design-patterns.fr/singleton>

**Application :**

Dans notre projet nous l’utiliserons dans la partie « **Modèle** » du **MVC** pour gérer l’instance de connexion avec la base de données. En effet il nous faudra instancier strictement un objet de la classe qui doit gérer la connexion avec la base de données afin de s’assurer qu’il y a bien un seul point d’entrée vers notre Base de données. On gagne ainsi en fiabilité et en sécurité de nos données.

Nous allons aussi l’utiliser dans l’implémentation du design pattern Factory

1. Design Pattern Factory

**Principe**

Le design Pattern Factory permet de fournir une classe qui va s’occuper de l’instanciation de toutes les classes du programme. Il crée donc une interface pour créer un objet, mais laisse les sous-classes décider de la classe à instancier.

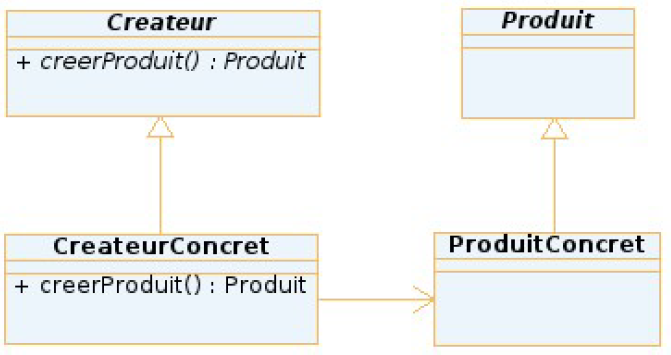


Figure 5 - Exemple Schéma explicatif du Pattern Factory

**Source image :** <http://www.c-jump.com/CIS75/Week11/lecture.html>

**Application**

Tout au long de notre projet, nous l’utiliserons donc pour simplifier le développement et la modification du code en rassemblant toutes les créations d’instances dans un unique endroit et pour réduire le couplage entre nos classes, cela simplifiera l’écriture des test unitaires ainsi que l’évolution de notre programme.