Design Pattern : nos différents choix

Ce document est aperçu des choix que nous avons effectué concernant les différents Design Pattern. Etant donné leur nombre important, nous nous devions de choisir les plus adaptées, et ceci en fonction des besoins du projet. Nous avons donc choisi les suivants :

Strategy : En informatique le patron *stratégie* est un *design pattern* 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. Le patron stratégie est prévu pour fournir le moyen de définir une famille d'algorithmes, encapsuler chacun d'eux en tant qu'objet, et les rendre interchangeables. Ce patron laisse les algorithmes changer indépendamment des clients qui les emploient.

Il est à utiliser dès lors qu'un objet peut effectuer plusieurs traitements différents, dépendant d'une variable ou d'un état. Dans notre cas il nous servira beaucoup lors de ce projet afin de générer d'automatique des éléments divergeant mais reposant sur une base commune. (les clients par exemple.)

Builder : Le monteur (*builder*) est un design pattern utilisé pour la création d'une variété d'objets complexes à partir d'un objet source. L'objet source peut consister en une variété de parties contribuant individuellement à la création de chaque objet complet grâce à un ensemble d'appels à l'interface commune de la classe abstraite *Monteur*. Son but est de séparer la construction d'un objet complexe de la représentation afin que le même processus de construction puisse créer différentes représentations.

Factory : La fabrique (*factory method*) est un design pattern utilisé en POO. 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 design pattern singleton pour les implémenter.

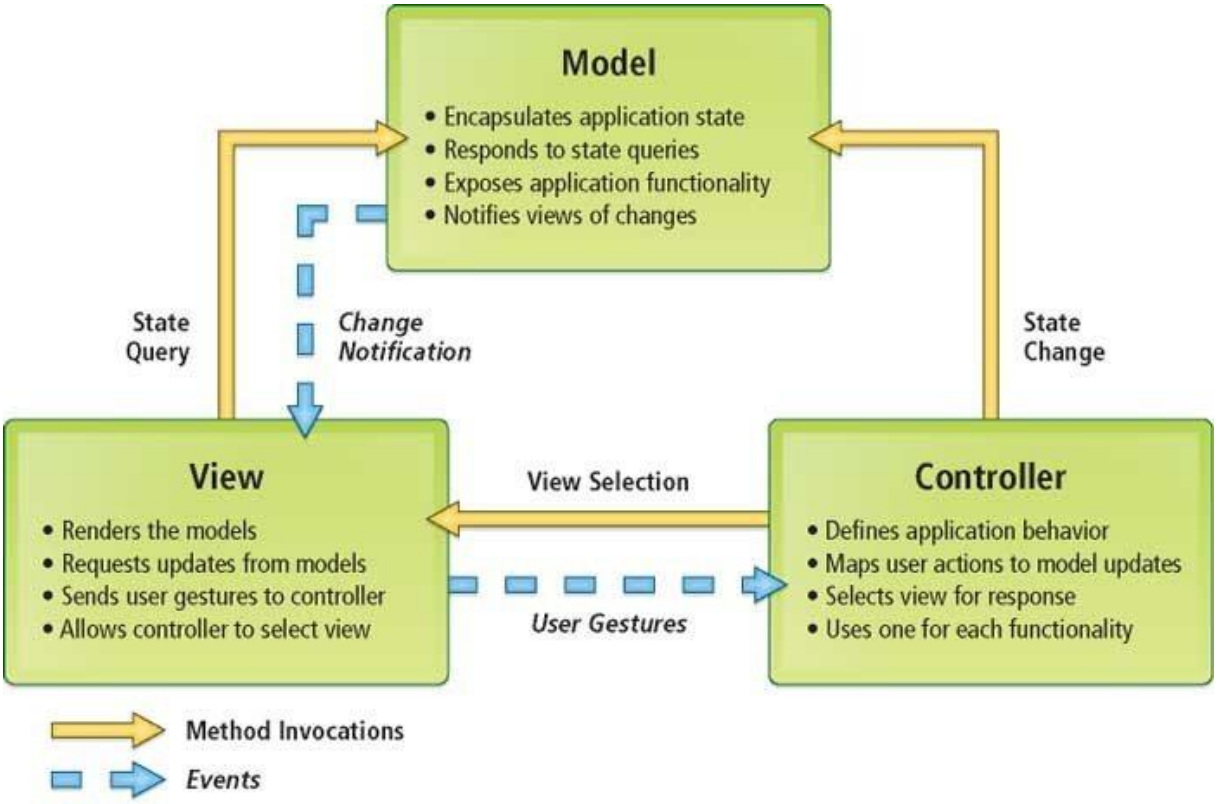
Singleton : Le singleton est un *design pattern* dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement). Il est utilisé lorsqu'on a besoin exactement d'un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec peu d'objets qu'avec beaucoup d'objets similaires.

Decorator : Un décorateur permet d'attacher dynamiquement de nouvelles responsabilités à un objet. Les décorateurs offrent une alternative assez souple à l'héritage pour composer de nouvelles fonctionnalités.
Il résout les problématiques suivantes :

- L'ajout (et la suppression) des responsabilités à un objet dynamiquement au moment de l'exécution.
- Il constitue une alternative aux sous-classes pour une surcharge flexible des fonctionnalités.

Quand on utilise l'héritage, les différentes sous-classes étendent une classe mère en différentes manières. Mais une extension est attachée à la classe au moment de la compilation, et ne peut pas changer à l'exécution.
On l'utilisera surement pour l'interface

MVC
:



Il s'agit d'un style d'architecture de logiciel très populaire vous permettant d'être efficace & structuré lors du développement d'un projet.

Le M, modèle (model) représente les données et ne fait rien d'autre, il ne dépend donc pas des vues & des contrôleurs.

Le V, vue (view) affiche les données que le modèle contient à l'utilisateur, ex : formulaire, bouton, etc... (Un modèle peut contenir plusieurs vues)

Le C, contrôleur (controller) traite les données de l'utilisateur, il dépend et interagit avec le modèle ainsi que la vue (Dans certains cas il est possible que la vue & le contrôleur soient le même objet).

Le modèle MVC vous permet d'être plus efficient en vous donnant la possibilité de réutiliser le même code plusieurs fois dans d'autres applications similaires à celle que vous avez et ainsi que de séparer votre structure en multiple calques. Cette combinaison vous offre une meilleure maintenance de votre projet et vous facilite la tâche durant les tests unitaires & vous permet de travailler à plusieurs sur un même projet (ex : Le développeur back-end peut mettre en place la structure sans intervenir sur le front-end, vice-versa).

Un autre avantage est le fait qu'il y a peu d'accrochage entre les vues, les modèles et les contrôleurs, ce qui vous permet d'avoir un code propre où les fonctions & les classes peuvent être facilement réécrite & optimisé.