Votre Formation Sur Mesure

DESIGN PATTERNS

ELAN
202 avenue de Colmar - 67100 STRASBOURG
☎ 03 88 30 78 30 🖷 03 88 28 30 69
elan@elan-formation.fr
www.elan-formation.fr

SAS ELAN au capital de 37 000 € -
RCS Strasbourg B 390758241 – SIRET 39075824100041 – Code APE : 8559A
N° déclaration DRTEFP 42670182967 - Cet enregistrement ne vaut pas agrément de l'Etat

SOMMAIRE

I. Qu'est-ce qu'un design pattern?

Un design pattern, ou patron d'architecture en français, se définit ainsi (source Wikipedia) :

"…modèle de référence qui sert de source d'inspiration lors de la conception de l'architecture d'un système ou d'un logiciel informatique en sous-éléments plus simples."

Les développeurs d'applications informatiques ont souvent rencontrés les mêmes problématiques de conception lorsqu'il fallait optimiser la Programmation Orientée Objet (POO) : rendre leur code plus lisible, plus organisé et plus efficace en termes d'utilisation de ressources (mémoire, etc.).

Dès la fin des années 1970 apparaissent des articles mentionnant des architectures logicielles s'apparentant à ce qui deviendra les design patterns actuels. En 1995, quatre programmeurs (qui constituèrent ce qu'on a appelé le Gang of Four ou GoF) ont publié un livre devenu référence : *Design Patterns - Elements of Reusable Object-Oriented Software.* Cet ouvrage fut le précurseur de la normalisation des architectures développées en POO et donna un nom et un mode d'emploi à près de 25 patrons de conception.

Les design patterns sont alors apparus comme essentiels aux développeurs, leur apportant globalement ces quelques avantages :

- respecter des méthodes de conception professionnellement reconnues
- développer plus rapidement en suivant des modèles architecturaux ayant fait leurs preuves
- permettre au code d'être relu plus facilement par un autre développeur sensibilisé aux patterns utilisés

Il existe beaucoup de patterns, dont l'influence sur l'application va aussi bien de l'aspect purement "codage" d'une application que dans l'organisation et l'interdépendance des fichiers/dossiers de celle-ci.

II. Exemples

Voici quelques exemples (pour les plus connus et utilisés) :

1. Factory

Ce patron fournit une interface pour créer des familles d'objets sans préciser la classe dont il sera le modèle. Le patron factory (en français *fabrique*) 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.

Un exemple très utilisé de ce patron est une fabrique de l'objet ayant pour rôle de se connecter à la base de données. Si le programmeur souhaite changer le

SGBD de l'application, il devra respecter les paramètres de connexion de celui-ci, la factory instanciera alors l'objet de connexion elle-même dans une de ses méthodes, selon le SGBD voulu.

2. Composite

Ce patron permet de composer une hiérarchie d'objets, et de manipuler de la même manière un élément unique, une branche, ou l'ensemble de l'arbre. Il permet en particulier de créer des objets complexes en reliant différents objets selon une structure en arbre. Ce patron impose que les différents objets aient une même interface, ce qui rend uniformes les manipulations de la structure.

Par exemple dans une ville, les habitations peuvent être regroupées par rue, puis par quartier, puis par arrondissement, etc. Pour manipuler l'ensemble, une classe composite implémente une interface « Ville ». Cette interface est héritée par les objets qui représentent les habitations, les rues, les quartiers et les arrondissements.

3. Iterator

Ce patron permet d'accéder séquentiellement aux éléments d'un ensemble sans connaitre les détails techniques du fonctionnement de l'ensemble. C'est un des patrons les plus simples et les plus fréquents. Il consiste le plus souvent en une interface qui fournit les méthodes Précédent et Suivant, de la même manière qu'un menu de pagination.

4. Observer

Ce patron établit une relation un à plusieurs entre des objets. Lorsqu'un objet change, plusieurs autres objets sont alors mis au courant de ce changement et peuvent effectuer une action précise avant, pendant ou après celui-ci. Dans ce patron, une classe observable comporte une méthode pour inscrire des observateurs. Chaque observateur comporte une méthode « Notifier ». Lorsqu'un message est émis, la classe observable appelle la méthode « Notifier » de chaque observateur inscrit.

Exemple courant : une classe s'occupant d'inscrire l'activité d'un utilisateur dans un fichier log se doit d'observer les événements liés au login, à la mise à jour du profil, etc.

5. Singleton

Ce patron vise à assurer qu'il n'y a toujours qu'une seule instance d'une classe précise en fournissant une interface pour la manipuler. C'est un des patrons les plus simples. L'objet qui ne doit exister qu'en un seul exemplaire comporte une méthode pour obtenir cette unique instance et un mécanisme pour empêcher la création d'autres instances. L'utilisation de ce pattern est sujette à débat depuis pas mal de temps, en ce qui concerne son efficacité et sa pertinence.

6. Strategy

Dans ce patron, une famille d'algorithmes est encapsulée de manière à ce qu'ils soient interchangeables. Les algorithmes peuvent changer indépendamment de l'application qui s'en sert. Il comporte trois rôles: le contexte, la stratégie et les implémentations. La stratégie est l'interface commune aux différentes implémentations. Le contexte est l'objet qui va associer un algorithme avec un processus.

Pour illustrer ce pattern, reprenons notre exemple de fichier log. Lorsqu'il sera nécessaire d'enregistrer les logs de l'activité utilisateur sur plusieurs supports (fichiers, base de données, etc.) et ce dans plusieurs formats (HTML, texte…). Appliquer le pattern Strategy permettra de réserver la tâche de formatage à une interface (stratégie) implémentée par des classes réservées à chaque format, puis un objet « Writer » inscrira le message formaté sur le support adéquat, en respectant les spécificités du support. L'algorithme principal, celui qui écrit le log proprement dit (le contexte), décrit cette tâche dans son algorithme mais, puisque le support peut changer, laisse une de ses implémentations s'occuper réellement de ça.

7. Template

Ce patron définit la structure générale d'un algorithme en déléguant certains passages. Permettant à des sous-classes de modifier l'algorithme en conservant sa structure générale. C'est un des patrons les plus simples et les plus couramment utilisés en programmation orientée objet. Il est utilisé lorsqu'il y a plusieurs implémentations possibles d'un calcul. Une classe d'exemple (anglais template) comporte des méthodes d'exemple, qui, utilisées ensemble, implémentent un algorithme par défaut. Certaines méthodes peuvent être vides ou abstraites. Les sous-classes de la classe template peuvent remplacer certaines méthodes et ainsi créer un algorithme dérivé.

III. Les patterns MVC et MVP

Combinaison des patrons observer, strategy et composite ;ce qui forme ainsi un patron d'architecture à part entière ; le pattern Modèle-Vue-Contrôleur (MVC) est le plus célèbre d'entre tous. Pourtant, et au vu des nombreux contributeurs de blogs et autres forums de développeurs s'écharpant sur le sujet, il est un des plus galvaudés et connait ainsi de multiples « versions ».

1. Avant-propos

Tous les frameworks orientés objet (Symfony, CakePHP, CodeIgniter, RubyOnRails…) implémentent le pattern MVC ou une de ses variantes (MVP, MVVM, MVPC…). Par ailleurs, le terme MVC est souvent utilisé à tort pour nommer une de ses variantes alors que ces dernières modifient la base de réflexion du pattern MVC originel.

Il n'est ni constructif ni obligatoire de rentrer nous-même, par le biais de ce support, dans cette « guerre des mots » à laquelle semblent se livrer de nombreux développeurs au sujet du pattern MVC, et ce depuis quelques temps. L'objectif de ce qui suit n'est pas de présenter rigoureusement l'intégralité des variantes de ce patron de conception, mais de sensibiliser à cette méthodologie d'architecture logicielle, très répandue de nos jours.

2. Modèle-Vue-Contrôleur, l' « original »

Le principal avantage de l'utilisation du pattern MVC réside dans son nom même : le code de l'application se voit divisé en trois parties, chaque partie s'octroyant un rôle précis (logique métier, présentation et traitements) sans que les deux autres n'aient à s'en occuper. L'application se voit alors, en termes de fichiers et dossiers, répartie physiquement de telle sorte à visualiser rapidement cette décomposition.

a. Le contrôleur

Le contrôleur a pour tâche de synchroniser les vues avec les modèles : contrôle des données et des paramètres passés, gestion des événements... C'est le premier acteur atteint par la demande de l'utilisateur : un bouton de déconnexion aura pour cible la méthode d'un contrôleur prévu à cet effet, celui-ci aura pour mission de vérifier le statut de l'utilisateur, d'appeler toutes les « fonctions » nécessaires à cette tâche et de transmettre au modèle les données de l'utilisateur.

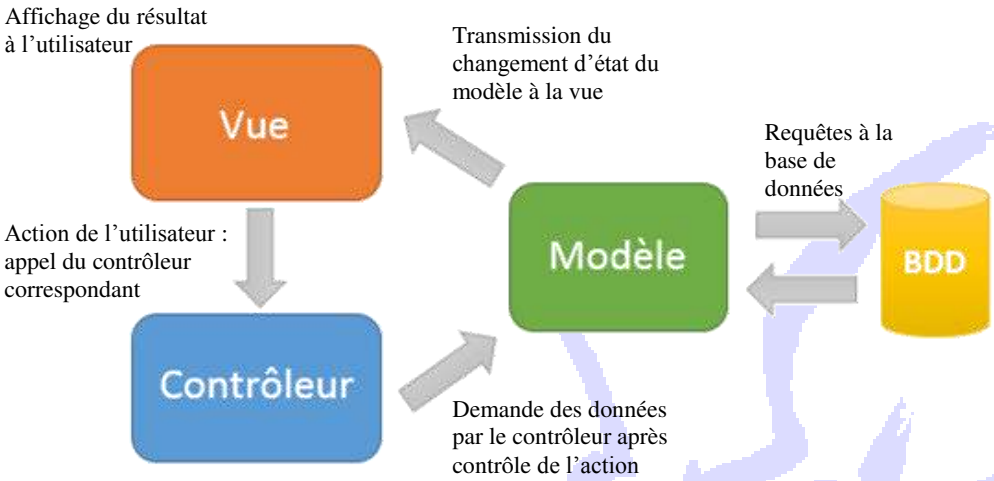
b. Le modèle

C'est la couche métier de l'application. Suite à l'action de l'utilisateur sur l'application, le modèle se charge de récupérer les données demandées par le contrôleur, effectue cette récupération selon le système de gestion de base de données, traite le ou les résultats et le retransmet.

c. La vue

La vue présente l'information visuellement mais permet également d'interagir avec. Le bouton de déconnexion cité dans la partie « contrôleur » est affiché dans une vue « déconnexion » et cette vue se charge de paramétrer le lien vers le contrôleur adéquat.

Le schéma ci-dessous illustre l'interaction des trois parties du MVC :



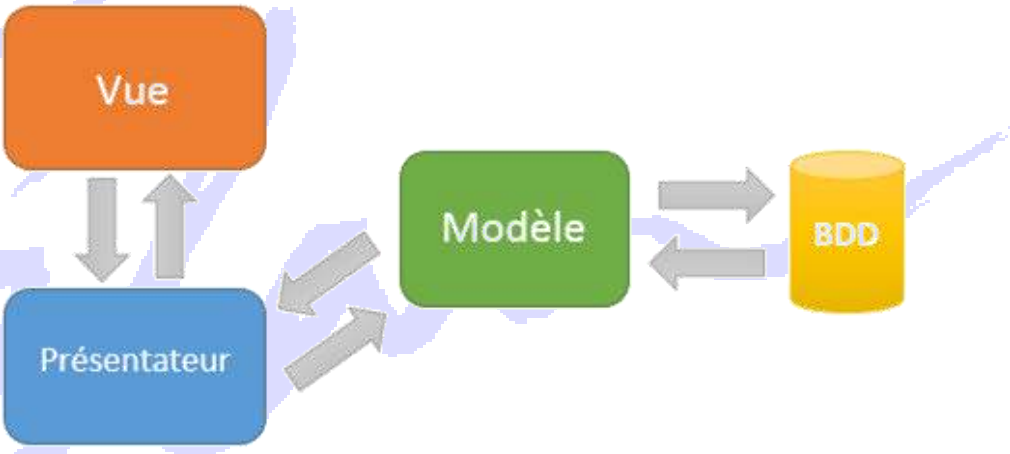
Il est à noter que, sur ce schéma, le modèle se charge de transmettre le résultat des données à la vue, sans contrôle supplémentaire ni reformatage de ces données par un contrôleur.

Dans ce cas présent, c'est au modèle qu'incombe la tâche de formater ces données pour la vue, ce qui lui octroie des tâches qui, pour des raisons de logique, incomberaient plus pertinemment à un contrôleur.

3. Modèle-Vue-Présentateur, l'« alternative »

La constatation précédente, somme toute logique mais subtile, d'un "non-sens" théorique sur le patron de conception constitue le principal point d'achoppement qui a donné naissance à de nombreuses variantes du MVC.

Le pattern MVP (Modèle-Vue-Présentateur) est la réponse au problème soulevé précédemment : les échanges directs entre le modèle et la vue.



L'action transmise de la vue au présentateur (qui possède la même fonction qu'un contrôleur, à peu de choses près) subit un premier contrôle puis le présentateur fait une demande au modèle. Le modèle ne se chargeant ici que de

la récupération et la transmission des données vers ou depuis la base de données. Le contrôleur les récupère et les formate (voire les recontrôle) pour enfin les transmettre à la vue correspondante.

Avec ce pattern, chaque partie a un rôle qui lui est clairement défini. Attention toutefois, comme dit plus haut, il est souvent rencontré sur Internet des schémas identiques mais sous le nom MVC, mais c'est un abus de langage malheureusement assez répandu.

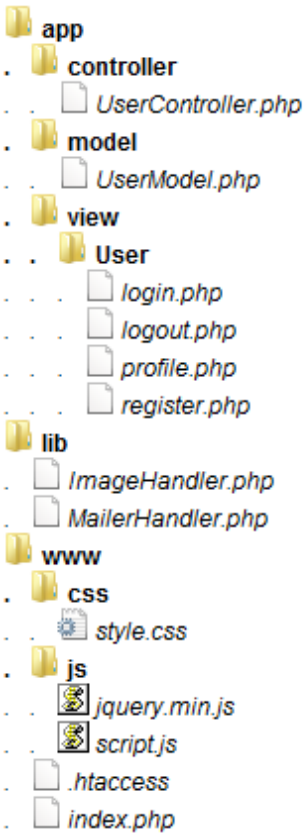
4. Le Front-Controller

Au sein d'une application web organisée en MVC, un fichier précis représente en quelque sorte "la porte d'entrée" de l'application : cet élément s'occupe de diriger la requête de l'utilisateur aux contrôleurs correspondants, fournissant un point d'entrée centralisé qui orchestre les chemins (routes) à atteindre pour traiter les demandes et atteindre leurs résultats.

Le Front-Controller est reconnu comme un design pattern en soi, son implantation n'est aucunement obligatoire, mais dans le cadre d'un site Internet il apporte par son action un avantage indispensable au besoin de découplage et de lisibilité du code.

5. Exemple

Voici un exemple d'arborescence MVC très simple, parmi tant d'autres :



Prenons l'exemple d'une gestion d'utilisateur permettant l'inscription, la connexion, la mise à jour du profil et la déconnexion.

Ici, le dossier app est réellement celui qui subit l'organisation en MVC de l'application. Le dossier view est subdivisé en sous-dossiers pour des raisons de lisibilité.

Le dossier lib contient les librairies dites « third-party », c'est-à-dire les algorithmes se chargeant de fonctionnalités annexes à l'application (ici envoi de mails de confirmation et traitement des images du profil utilisateur). Les librairies sont généralement totalement découplées de l'application, donc interchangeables de projet en projet.

Le dossier www (nommé parfois *public*) contient les fichiers « front » de l'application, autrement dit ceux que l'utilisateur est autorisé à manipuler.

Le fichier index.php constitue la porte d'entrée de tout ceci, il fait figure de FrontController : index.php va assembler les vues nécessaires à la demande du visiteur, tout en disposant du layout du site, c'est-à-dire des éléments communs à toutes les pages pouvant être visitées (logo, menu, pied de page, etc.).

Ce dernier va router les demandes et s'occuper de l'appel des vues adéquates ou, bien souvent, travaille de concert avec un routeur qui génèrera les chemins à atteindre.

IV. Références en ligne

https://openclassrooms.com/courses/programmez-en-oriente-objet-en-php/les-design-patterns

http://baptiste-wicht.developpez.com/tutoriels/conception/mvc/

http://design-patterns.fr/

http://www.albertzuurbier.com/index.php/programming/84-mvc-vs-mvp-vs-mvvm (en anglais)