

Zend Framework et Design Patterns - Utiliser une façade pour accéder au modèle

par JYT ([Les expériences Zend de Sekaijin](#)) ([Blog](#))


Date de publication : 27 octobre 2007

Dernière mise à jour :

Parmi les *design patterns* que l'on trouve en programmation, il en est un que l'on oublie un peu trop à mon avis : c'est la notion de **façade**.

- I - Introduction
- II - Encore un exemple pour comprendre pourquoi
- III - Utiliser une façade avec MVC
- IV - Une première approche simple
- V - Helpers et autre plugins
- VI - Les modules et composants
- VII - Exemple d'utilisation

I - Introduction

Parmi les  **design patterns** que l'on trouve en programmation, il en est un que l'on oublie un peu trop à mon avis : c'est la notion de **façade**. Pour bien comprendre de quoi il s'agit, un petit exemple sera plus parlant. Imaginez travailler dans un centre de conception automobile. Votre boulot consiste à concevoir toute la partie commande de la voiture. Vous allez collaborer avec les designers qui conçoivent l'habitacle et le tableau de bord (la *vue*) et avec les ingénieurs qui conçoivent la partie mécanique (le *modèle* ou métier). À vous de relier le tout (faire le *contrôleur*). Si les mécanos commencent à vous parler de boulon de six ou de durite d'injection, vous allez droit à la crise de nerfs. Ce dont vous avez besoin, c'est de savoir comment on fait marcher le tout. Par exemple : comment accélérer. Que la commande d'accélération actionne un régulateur électrique, ouvre un carburateur ou actionne un injecteur de gasoil, vous importe peu.

Vous avez besoin de définir une API pour votre partie mécanique. Cette définition peut se présenter simplement par une documentation. Mais aussi par un point d'entrée unique qui vous cache la complexité du métier. Cela s'appelle une façade.

Vous ne vous êtes jamais penché sur le problème de savoir comment votre banquier traitait vos chèques ? Vous fournissez au guichet le bordereau dûment rempli comme le prévoit la façade et ce qu'il se passe après vous importe peu, du moment que votre argent va sur votre compte.

C'est donc une pratique courante qui, étrangement, se perd lorsqu'on développe des logiciels. Bien souvent, on fait ce qu'il y a à faire sans trop se demander s'il ne serait pas opportun d'isoler un sous ensemble derrière une API. En programmation à objet on encapsule ainsi des éléments. Mais on pense rarement que tout un ensemble pourrait être caché par une seule classe. La littérature sur ce sujet est pourtant vaste. Et utiliser une façade à bon escient est un gage pour l'avenir.

II - Encore un exemple pour comprendre pourquoi

Sur L'A300, les commandes de vols étaient mécaniques. Pour les commandes moteur, un levier au centre du poste de pilotage actionnait une série de cames qui finalement jouaient sur les commandes du moteur : directement. Les compagnies demandant à Airbus des moteurs différents, il était nécessaire de changer une bonne partie de cette quincaillerie pour adapter le moteur à l'avion. Sur l'A320, l'ensemble des commandes a été dé-corrélé de leur support. Pour adapter un nouveau moteur, il suffit alors de redéfinir l'interface.

Il en va de même de vos développements. Si vous arrivez à bien cloisonner vos sous-ensembles, alors en changer une partie n'impliquera que des changements mineurs sur le reste, tant que l'interface ne change pas.

III - Utiliser une façade avec MVC

Le modèle MVC propose une séparation nette et précise dans votre application. Dans Zend Framework, l'interface entre la *vue* et le *contrôleur* est bien définie. L'interaction entre la *vue* et le *modèle* est à proscrire. Mais l'interface entre le *contrôleur* et le *modèle* est complètement à votre charge. Utiliser une façade comme modèle revient à n'avoir qu'un seul et unique objet à qui le contrôleur peut s'adresser, comme pour votre banquier (le guichet). Cet objet ne va rien faire lui-même, il va juste vous mettre à disposition vos objets métiers. Mais au passage, il va vous cacher la complexité de leur organisation. Le contrôleur peut alors s'occuper de son travail (la logique applicative), pendant que le modèle s'occupe de la logique métier. D'un côté, vous parlez de clients, de facture, d'automobile, d'avion, de téléphone, de livre et que sais-je d'autre. De l'autre vous parlez de table, de base de données, d'annuaire LDAP, de ressources XML, de services Web *etc.*

Agissant ainsi, vous allez être contraint de définir clairement les fonctionnalités que le *modèle* met à disposition du *contrôleur*. Un peu de réflexion sur son travail n'est jamais mauvais en soi. De plus, vous allez pouvoir définir toute l'interface sans à avoir à l'implémenter. Ainsi, le développement du *contrôleur* peut commencer sans attendre que le *modèle* soit prêt. Dans un travail en équipe, c'est un bon moyen de répartir le travail. Cela permet de mettre en place toute la cinématique de l'application sans qu'elle ne fasse quoi que ce soit. La maîtrise d'ouvrage peut alors la valider. De son côté, le métier n'a pas à se préoccuper de savoir comment ses objets seront utilisés, il suffit qu'ils répondent à la définition de l'interface. Et enfin, en cours de vie de l'application, un pan entier peu changer dans le métier sans impacter le reste.

Un autre point intéressant dans cette approche est que le *contrôleur* accédant à son *modèle* au travers de la façade, alors peu importe le contenu de cette façade : il devient possible tout comme pour la *vue* de prévoir à l'avance la relation entre le *contrôleur* et le *modèle*. Pouvoir accéder à sa *vue* juste au travers d'un membre est bien pratique. Pourquoi ne pas profiter de la façade et faire de même. Le *contrôleur* demande les services à son *modèle* et fournit les valeurs à sa *vue*.

```
<?php  
$this->view->clientList = $this->model->getClientListOfCurrentUser();
```

Il serait intéressant de rendre l'attachement du membre **model** transparent pour le développeur comme l'est l'attachement de la *vue*. Afin de garder un comportement du framework le plus standard possible, j'ai décidé de rendre cet attachement optionnel et commandé par un paramètre, comme on l'a vu dans un article précédent.

J'ai choisi le paramètre : « useModel ».

IV - Une première approche simple

Définir une classe **Model** au sein de l'application. Dans `/application/Model.php`, puis dans chaque *contrôleur* dans la méthode `ini` on ajoute :

```
if ($parameters = Fast_Registry::getParameters()) {  
    if (parameters->fast->get('useModel', false)) {  
        Zend_Loader::loadClass('Model');  
        $this->model = new Model();  
    }  
}
```

Reste au développeur à ajouter les méthodes spécifiques à son métier dans la classe **Model**. Peu importe ce qu'il mettra dans la méthode `getClientListOfCurrentUser` : si elle répond au prototype, alors le *contrôleur* vu plus haut sera fonctionnel.

Mais devoir intervenir sur tous les contrôleurs pour bénéficier de l'attachement du modèle, n'est pas plus pratique que de devoir aller le chercher lorsqu'on en a besoin.

Un attachement automatique et transparent est plus pertinent. Gardons dans un premier temps la classe **Model** dans l'application et ajoutons une classe d'action générique qui aura pour but de collecter ce qui est commun aux différents contrôleurs. C'est ce que fait déjà Zend Framework avec la classe **Zend_Controller_Action**. En faisant dériver cette classe de **Zend_Controller_Action** puis nos contrôleurs de cette classe, nous aurons un endroit pour mettre toutes les parties communes.

Un petit souci : où placer cette classe ? En suivant la nomenclature ZF, je serais tenté de l'appeler **Application_Controller_Action** et elle se trouverait alors dans le dossier `/application/controller/`, or ce dossier n'existe pas. L'ajouter alourdirait la hiérarchie sans améliorer la lisibilité. Nous aurons côte-à-côte les dossiers **controller** et **controllers**. La différence sémantique entre les deux est subtile : **controller** est le dossier contenant les éléments propres au mécanisme de contrôle et **controllers** est le dossier contenant les contrôleurs de l'application. Pour ne pas alourdir la hiérarchie, j'ai décidé de mettre ma classe dans **controllers**. Elle s'appellera donc **Applications_Controllers_Action**. Je déplace les quelques lignes de la méthode `ini` de mes contrôleurs dans celle de la classe **Action** et je fais dériver mes contrôleurs de la classe **Action**.

```
class IndexController extends Application_Controllers_Action
```

Si le paramètre `useModel` est à `true`, alors tous mes contrôleurs ont le membre `model`.

V - Helpers et autre plugins

J'avoue sincèrement ne pas avoir eu le temps de me pencher sur cette approche. Il me semble possible de reproduire un comportement similaire de cette façon. Ce sera peut être une évolution future.

VI - Les modules et composants

Si j'utilise des modules, je vais pouvoir reproduire dans le module le fonctionnement décrit ci-dessus. Mes modules auront alors leur propre façade sur la partie métier qui les concerne. C'est plutôt pas mal. Mais comment faire si un module a besoin de tout ou partie du modèle d'un autre ? S'il n'utilise que le modèle d'un autre module, je peux m'arranger pour qu'il prenne comme façade celle de ce module. Il suffit pour cela de bien définir le chargement de la classe. Mais s'il doit utiliser son propre modèle plus des morceaux de celui d'un autre, je suis obligé de recopier une partie de cette façade. Dupliquer du code n'est jamais une bonne chose, surtout lorsqu'il n'y a aucun changement entre les deux copies. De plus, on perd alors l'intérêt d'une façade dont le but est justement de n'avoir qu'un point d'accès.

Pour répondre à cette problématique tout comme Zend l'a fait pour les vues, j'ai défini une classe modèle générale qui accepte des plugins, qui dans mon cas seront des Composants de modèle, des partie de modèle, permettant ainsi aux contrôleurs de ne charger que les composants dont ils ont besoin, chaque composant étant une façade sur un sous ensemble du modèle. Le remplacement d'un composant par un autre est ainsi facilité et l'ajout de composant dans l'application aussi. De plus, les modules peuvent toujours définir leurs propres modèles, tout en laissant la possibilité à d'autres de les utiliser.

Reprenant le fonctionnement décrit plus haut, j'ai cette fois défini une classe Action générale : **Fast_Controller_Action** dérive de **Zend_Controller_Action**. Mes contrôleurs ou **Application_Controller_Action** peuvent maintenant en dériver. C'est elle qui embarque le chargement de la classe **Fast_Model** dont une instance est attachée aux contrôleurs. Elle dispose d'une méthode **addComponent** qui prend comme paramètres le nom d'une classe composant et d'un chemin optionnel. Elle charge alors cette classe et ajoute ses méthodes au modèle. La classe modèle à fournir doit dériver de **Fast_Model_Component**. Un composant pouvant avoir besoin d'interroger le modèle, un membre protégé est disponible.

VII - Exemple d'utilisation

Parameters.ini

```
useModel = true
```

Classe client

```
<?php
Zend_Loader::loadClass('Fast_Model_Component');
class Model_Client extends Fast_Model_Component
{
    public function getClientList($nameStart) {
        #
    }
}
```

Il y a fort à parier que le **ClientController** devra utiliser ce composant dans la majorité de ses méthodes. Le plus simple est alors de le charger dans la méthode **ini** mais il est parfaitement possible de charger un composant pour tout le module et en même temps d'en avoir un autre qui n'est chargé que pour une action.

Contrôleur

```
Zend_Loader::loadClass('Fast_Controller_Action');
class ClientController extends Fast_Controller_Action
{
    public function showListAction(){
        //charger le composant model_Client du dossier model de ce module
        $this->model->addComponent('Model_Client' , dirname(dirname(__FILE__)));
        $this->view->clientList = $this->model->getClientList() ;
    }
}
```

Notez au passage qu'il est très simple ici de faire un bouchon. Il suffit dans la méthode **getClientList** de retourner une valeur du type attendu sans brancher réellement celle-ci sur le vrai modèle. Lorsque le modèle est prêt, les branchements sont relativement simples puis qu'il n'y a plus à chercher où sont utilisés les objets métier. Ils ne le sont qu'au travers de la façade.

A+JYT

