Patrons de conception

Partie 1

Introduction

Un patron de conception (plus connu sous le terme anglais « Design pattern ») est une solution générique permettant de résoudre un problème spécifique.

En général, un patron de conception décrit une structure de classes utilisant des interfaces, et s'applique donc à des développements logiciels utilisant la programmation orientée objet.

Cette notion peut paraître nouvelle, mais il s'agit en fait plutôt d'un nouveau terme pour désigner les algorithmes, et les structures de données permettant de résoudre différents problèmes.

Un exemple :

Une liste chaînée permet d'avoir un groupe d'éléments dont le nombre n'est pas fixe, contrairement aux tableaux.

Cette notion ne s'applique donc pas seulement à la programmation orientée objet. Un autre exemple est l'architecture MVC (Modèle-Vue-Contrôleur) définissant une architecture où les fonctions (ou les classes en POO) ont un rôle bien défini.

Pertinence d'utilisation et implémentation

Utiliser des patrons de conception pour le développement de logiciels peut paraître compliqué ou superflu.

Dans les applications les plus simples, l'utilisation de patrons de conception peut générer une complexité dans le code source.

Cependant, les patrons de conception sont généralement utiles pour les applications ayant une taille importante et/ou dans les projets où plusieurs applications différentes interagissent entre elles (via un moyen de communication).

Il faut également bien comprendre le rôle d'un patron de conception afin de vérifier qu'il s'applique au cas rencontré, et ne pas ajouter inutilement une complexité si les avantages liés à l'utilisation d'un patron de conception ne sont pas requis.

Il y a différents ensembles de patrons de conception, créés par différents auteurs

- Les plus connus sont ceux du « Gang of Four » (ou GoF : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) décrits dans leur livre « Design Patterns -- Elements of Reusable Object-Oriented Software » (voir bibliographie) en 1995. *Les patrons de conception tirent leur origine des travaux de l'architecte Christopher Alexander dans les années 70*.

- Les patrons GRASP sont des patrons créés par Craig Larman qui décrivent des règles pour affecter les responsabilités aux classes d'un programme orienté objets pendant la conception, en liaison avec la méthode de conception BCE (pour « Boundary Control Entity » - en français MVC « Modèle Vue Contrôleur »).

- Les patrons d'entreprise (Enterprise Design Pattern) créés par Martin Fowler, décrivent des solutions à des problèmes courants dans les applications professionnelles. Par exemple, des patrons de couplage entre un modèle objet et une base de donnée relationnelle, par exemple DAO (Data Access Object).

D'autres patrons créés par divers auteurs existent et décrivent des solutions à des problèmes différents de ceux vus précédemment.

Les patrons de conception peuvent être classés en fonction du type de problème qu'ils permettent de résoudre.

Par exemple, les patrons de conception de création résolvent les problèmes liés à la création d'objets.

Patrons du « Gang of Four »

Ces patrons de conception sont classés en trois catégories :

- **Les patrons de création** décrivent comment régler les problèmes d'instanciation de classes, c'est à dire de création et de configuration d'objets (objet en unique exemplaire par exemple).
- **Les patrons de structure** décrivent comment structurer les classes afin d'avoir le minimum de dépendance entre l'implémentation et l'utilisation dans différents cas.
- **Les patrons de comportement** décrivent une structure de classes pour le comportement de l'application (répondre à un évènement par exemple).

Patrons de création

Un patron de création permet de résoudre les problèmes liés à la création et la configuration d'objets.

Par exemple, une classe nommée *AppRessources* gérant toutes les ressources de l'application ne doit être instanciée qu'une seule et unique fois.

Il faut donc empêcher la création intentionnelle ou accidentelle d'une autre instance de la classe. Ce type de problème est résolu par le patron de conception « **Singleton** ».

Les différents patrons de création sont les suivants

Singleton

Il est utilisé quand une classe ne peut être instanciée qu'une seule fois.

Prototype

Plutôt que de créer un objet de A à Z c'est à dire en appelant un constructeur, puis en configurant la valeur de ses attributs, ce patron permet de créer un nouvel objet par recopie d'un objet existant.

Fabrique

Ce patron permet la création d'un objet dont la classe dépend des paramètres de construction (un nom de classe par exemple).

Fabrique abstraite

Ce patron permet de gérer différentes fabriques concrètes à travers l'interface d'une fabrique abstraite.

Monteur

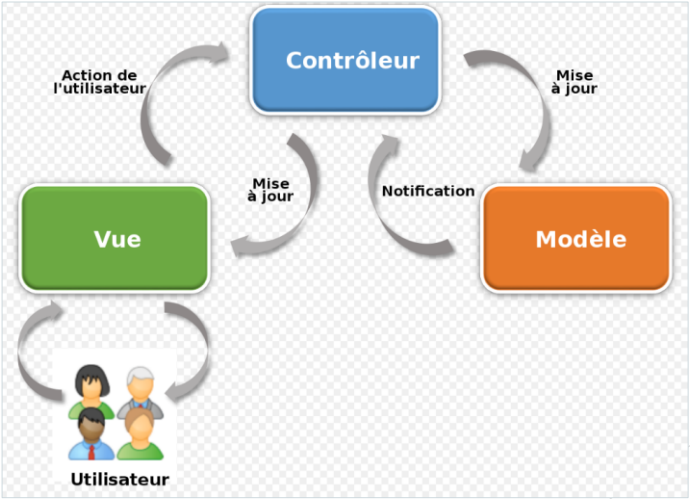
Ce patron permet la construction d'objets complexes en construisant chacune de ses parties sans dépendre de la représentation concrète de celles-ci.

Approche Modèle-Vue-Contrôleur

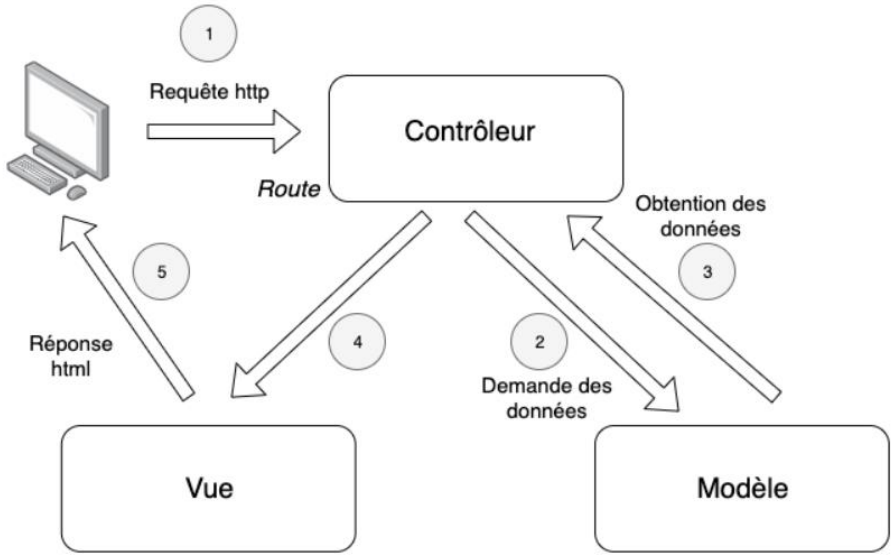
MVC

Introduction

L'approche Modèle-Vue-Contrôleur est une approche de conception de systèmes informatiques permettant de diviser en modules logiques une application comportant des interfaces graphiques, des données et de la logique (c'est-à-dire du contrôle).



Dans les applications web



Modèle

Il s'agit de la composante qui gère les données d'une application (p. ex., la modification ou la création de données).

Dans la plupart des applications, le modèle gère les accès aux bases de données dans lesquelles les données sont stockées.

Par contre, si le modèle est capable de manipuler les données, il est complètement dissocié de la manière de les présenter aux usagers.

Ainsi, le Modèle offre au reste de l'application un ensemble d'objets permettant d'accéder aux données et de les manipuler.

Vue

Il s'agit de la composante permettant d'afficher les données aux utilisateurs selon des modèles de conception établis par les concepteurs.

La Vue a pour seule tâche d'afficher les données par l'intermédiaire des composantes graphiques (champs de texte, tableaux, etc.) et de récupérer les entrées des utilisateurs par l'intermédiaire de ces mêmes composantes (boutons, champs d'écriture, etc.).

Les données affichées proviennent de la composante Modèle et les entrées sont envoyées vers le Contrôleur.

Contrôleur

Il s'agit de la pièce maîtresse de l'architecture MVC.

Le Contrôleur connaît les besoins de la Vue et du Modèle et sait quel moyen la Vue utilise pour modifier ou demander des informations au Modèle.

Il transforme les requêtes de la Vue en action, ou suite d'actions, compréhensibles par le Modèle et informe la Vue de changements d'état du Modèle.

Cette dynamique peut s'exécuter de façon synchrone ou non. Très souvent, la Vue et le Contrôleur forment une paire.

Singleton

Le singleton est un patron de conception dont l'objet est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement).

Il est utilisé lorsque l'on a besoin d'exactement 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 ou bien protégé, afin de s'assurer que la classe ne puisse être instanciée autrement que par la méthode de création contrôlée.

Le singleton doit être implémenté avec précaution dans les applications multi-thread

Si deux processus légers exécutent en même temps la méthode de création alors que l'objet unique n'existe pas encore, il faut absolument s'assurer qu'un seul créera l'objet, et que l'autre obtiendra une référence vers ce nouvel objet.

La solution classique à ce problème consiste à utiliser l'exclusion mutuelle pour indiquer que l'objet est en cours d'instanciation.

Diagramme de classes UML

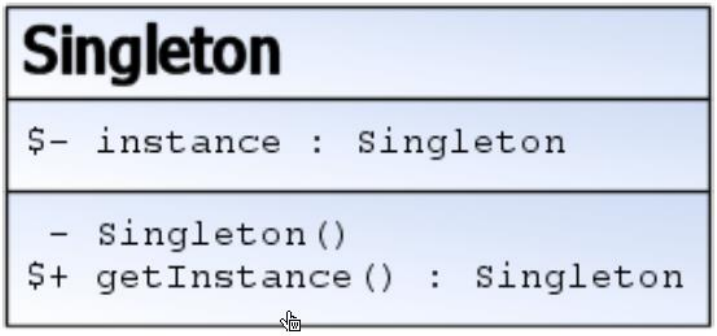


Diagramme de classes UML du patron de conception Singleton

JAVA

Voici une solution écrite en Java (il faut écrire un code similaire pour chaque classe-singleton)

```
public class Singleton
{
    private static Singleton INSTANCE = null;

    /*
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
     */
    private Singleton() {}
```

```
/*
 * Le mot-clé synchronized sur la méthode de création
 * empêche toute instanciation multiple même par
 * différents threads.
 * Retourne l'instance du singleton.
 */
public synchronized static Singleton getInstance()

     {
          if (INSTANCE == null)
              INSTANCE = new Singleton();
          return INSTANCE;
     }
 }
```

Une solution variante existe cependant. Elle consiste à alléger le travail de la méthode **getInstance** en déplaçant la création de l'instance unique au niveau de la déclaration de la variable référant l'instance unique.

```
public class Singleton
{
    /*
     * Création de l'instance au niveau de la variable.
     */
    private static final Singleton INSTANCE = new Singleton();

    /*
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
     */
    private Singleton() {}

    /*
     * Dans ce cas présent, le mot-clé synchronized n'est pas utile.
     * L'unique instanciation du singleton se fait avant
     * l'appel de la méthode getInstance(). Donc aucun risque d'accès concurrents.
     * Retourne l'instance du singleton.
     */
    public static Singleton getInstance()
    {
        return INSTANCE;
    }
}
```

PHP

```php
class Singleton {

    private $instance;

    /**
     * Empêche la création externe d'instances.
     */
    private function __construct () {}

    /**
     * Empêche la copie externe de l'instance.
     */
    private function __clone () {}

    /**
     * Renvoi de l'instance et initialisation si nécessaire.
     */
    public static function getInstance() {
        static $instance;

        if ($instance === null) {
            $instance = new self();
        }

        return $instance;
    }

    /**
     * Méthodes dites métier
     */
    public function uneAction () {}
}

// Utilisation
Singleton::getInstance()->uneAction();
```

Exercice

Créer une class Connexion en Node (à voir) pour obtenir une connexion à votre base de données MySQL.

D'autres patrons de conception à venir …