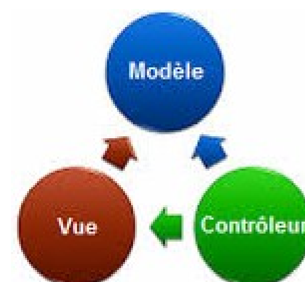
MVC

Table des matières

1. Introduction......2
2. Architecture......2
2.1. Modèle......2
2.2. Vue......3
2.3. Contrôleur......3
3. Flux de traitement......3
4. Différence avec l'architecture trois tiers......4
5. Exemple d’implémentation......5

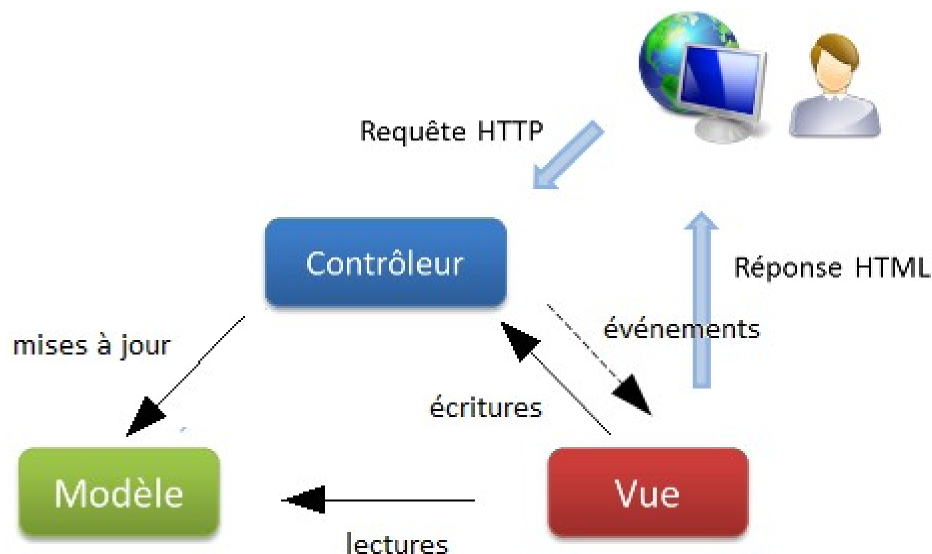
Le patron d'architecture logicielle modèle-vue-contrôleur (MVC : model-view-controller) est un modèle destiné à répondre aux besoins des applications interactives en séparant les problématiques liées aux différents composants au sein de leur architecture respective.



1. Introduction

Le patron MVC est apparu en 1978-1979. Son but principal était de proposer une solution générale aux problèmes d'utilisateurs manipulant des données volumineuses et complexes. Ce paradigme regroupe les fonctions nécessaires en trois catégories :

- un modèle (traitement, modèle de données) ;
- une vue (présentation, interface utilisateur) ;
- un contrôleur (logique de contrôle, gestion des événements, synchronisation).



Interactions entre le modèle, la vue et le contrôleur.

Les lignes pleines indiquent une dépendance et les pointillés les événements

2. Architecture

L'architecture "MVC" offre un cadre normalisé pour structurer une application et facilite le dialogue entre les concepteurs.

L'idée est de séparer les données, la présentation et les traitements. Il en résulte les trois parties énumérées plus haut : le modèle, la vue et le contrôleur.

Remarque : le modèle MVC est utilisé uniquement pour la couche présentation (côté serveur) dans l'architecture d'une application. La couche métier dans une application n'est en aucun cas concernée par ce modèle.

2.1. Modèle

Le modèle représente le cœur (algorithmique) de l'application : traitements des données, interactions avec la base de données, etc. Il décrit les données manipulées par l'application. Il regroupe la gestion de ces données et est responsable de leur intégrité. La base de données sera l'un de ses composants. Le modèle comporte des méthodes standards pour mettre à jour ces données (insertion, suppression, changement de valeur). Il offre aussi des méthodes pour récupérer ces données. Les résultats renvoyés par le modèle ne s'occupent pas de la présentation. Le modèle ne

contient aucun lien direct vers le contrôleur ou la vue.

Le modèle peut autoriser plusieurs vues partielles des données. Si par exemple le programme manipule une base de données pour les emplois du temps, le modèle peut avoir des méthodes pour avoir tous les cours d'une salle, tous les cours d'une personne ou tous les cours d'un groupe de TD.

2.2. Vue

Ce avec quoi l'utilisateur interagit se nomme précisément la vue. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toute action de l'utilisateur (clic de souris, sélection d'un bouton radio, cochage d'une case, entrée de texte, etc.). Ces différents événements sont envoyés au contrôleur. La vue n'effectue pas de traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle et d'interagir avec l'utilisateur.

Plusieurs vues peuvent afficher des informations partielles ou non d'un même modèle. Par exemple si une application de conversion de base a un entier comme unique donnée, ce même entier peut être affiché de multiples façons (en texte dans différentes bases, bit par bit avec des boutons à cocher, avec des curseurs). La vue peut aussi offrir à l'utilisateur la possibilité de changer de vue. Ceci permet une certaine récursivité du modèle.

2.3. Contrôleur

Le contrôleur prend en charge la gestion des événements de synchronisation pour que la vue ou le modèle se mettent à jour et les synchroniser. Il reçoit tous les événements de la vue et enclenche les actions à effectuer. Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle afin que les données affichées se mettent à jour.

Certains événements de l'utilisateur ne concernent pas les données mais la vue. Dans ce cas, le contrôleur demande à la vue de se modifier. Le contrôleur n'effectue aucun traitement, ne modifie aucune donnée. Il analyse la requête du client et se contente d'appeler le modèle adéquat et de renvoyer la vue correspondant à la demande.

Par exemple, dans le cas d'une base de données gérant les emplois du temps des professeurs d'une école, une action de l'utilisateur peut être l'entrée (saisie) d'un nouveau cours. Le contrôleur ajoute ce cours au modèle et demande sa prise en compte par la vue. Une action de l'utilisateur peut aussi être de sélectionner une nouvelle personne pour visualiser tous ses cours. Ceci ne modifie pas la base des cours mais nécessite simplement que la vue s'adapte et offre à l'utilisateur une vision des cours de cette personne.

Quand un même objet contrôleur reçoit les événements de tous les composants, il lui faut déterminer l'origine de chaque événement. Ce tri des événements peut s'avérer fastidieux et peut conduire à un code peu élégant (un énorme switch). C'est pourquoi le contrôleur est souvent scindé en plusieurs parties dont chacune reçoit les événements d'une partie des composants.

3. Flux de traitement

Lorsqu'un client envoie une requête à l'application :

- la requête envoyée depuis la vue est analysée par le contrôleur (par exemple un clic de souris pour lancer un traitement de données) ;
- le contrôleur demande au modèle approprié d'effectuer les traitements et notifie à la vue que

la requête est traitée (via par exemple un handler[1] ou une callback[2]) ;

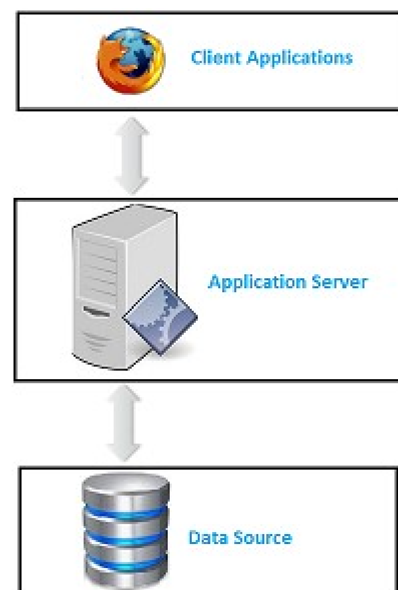
- la vue notifiée fait une requête au modèle pour se mettre à jour (par exemple affiche le résultat du traitement via le modèle).

Un avantage apporté par ce modèle est la clarté de l'architecture qu'il impose. Cela simplifie la tâche du développeur qui tenterait d'effectuer une maintenance ou une amélioration sur le projet. En effet, la modification des traitements ne change en rien la vue. Par exemple on peut passer d'une base de données de type SQL à XML en changeant simplement les traitements d'interaction avec la base, et les vues ne s'en trouvent pas affectées.

4. Différence avec l'architecture trois tiers

Son nom provient de l'anglais tier signifiant étage ou niveau. Il s'agit d'un modèle logique d'architecture qui vise à modéliser une application comme un empilement de trois couches logicielles dont le rôle est clairement défini :

1. la présentation des données : correspondant à l'affichage, la restitution sur le poste de travail, le dialogue avec l'utilisateur ;
2. le traitement métier des données : correspondant à la mise en œuvre de l'ensemble des règles de gestion et de la logique applicative ;
3. l'accès aux données persistantes : correspondant aux données qui sont destinées à être conservées sur la durée, voire de manière définitive.



exemple d'architecture en 3 couches

Chaque couche communique seulement avec ses couches adjacentes (supérieures et inférieures) et le flux de contrôle traverse le système de haut en bas. Les couches supérieures contrôlent les couches inférieures, c'est-à-dire que les couches supérieures sont toujours sources d'interaction (clients) alors que les couches inférieures ne font que répondre à des requêtes (serveurs).

Dans le modèle MVC, il est généralement admis que la vue puisse consulter directement le modèle (lecture) sans passer par le contrôleur. Par contre, elle doit nécessairement passer par le contrôleur pour effectuer une modification (écriture). Ici, le flux de contrôle est inversé par rapport au modèle en couches, le contrôleur peut alors envoyer des requêtes à toutes les vues de manière qu'elles se mettent à jour.

Dans l'architecture trois tiers, si une vue modifie les données, toutes les vues concernées par la modification doivent être mises à jour, d'où l'utilité de l'utilisation du MVC au niveau de la couche

1 module gérant une situation particulière comme une exception dans un processus.
2 une fonction de rappel ou de post-traitement : fonction passée en argument à une autre fonction.

de présentation. La couche de présentation permet donc d'établir des règles du type « mettre à jour les vues concernant X si Y ou Z sont modifiés ». Mais ces règles deviennent rapidement trop nombreuses et ingérables si les relations logiques sont trop élevées. Dans ce cas, un simple rafraîchissement des vues à intervalle régulier permet de surmonter aisément ce problème. Il s'agit d'ailleurs de la solution la plus répandue en architecture trois tiers, l'utilisation du MVC étant moderne et encore marginale.

5. Exemple d'implémentation

Une balle est lâchée d'une hauteur H (en m) et rebondit sur le sol. À chaque rebond, elle remonte d'un certain ratio de la hauteur du précédent rebond. Elle s'arrête lorsque la hauteur du rebond devient inférieure à 1 mm.

Déterminer le nombre N de rebonds effectués.

```
#!/usr/bin/python
# -*- coding: utf-8 *-*

"""
    module   : mvc.py
    projet   : Interface graphique, basée sur Tk

    version  : 1.0
    auteur   : profs SI
    creation : 2/10/2014
    modif    :
"""

from Tkinter import *
from tkFont import *

### la vue ###
class Dialogue (Frame):
    """
    Classe implementant lemasque de saisie de la vue
    """
    def __init__(self, master = None):
        Frame.__init__(self, Tk())
        self.fall     = Fall()         # instanciation du modele
        self.fall_err = FallError()    # instanciation du controleur
        self.setup()

    def setup(self):
        normal_font = Font(self, family = 'helvetica', size = 12)
        bold_font   = Font(self, family = 'Helvetica', size = 12, weight = BOLD)

        r = 0 # Compteur de ligne

        # Les entrees
        for (name, label) in [("high", "Hauteur"), ("rate", "Amortissement")]:
            self.texte = Label(self, text = label, font = bold_font)
            self.texte.pack()

            self.entry = Entry(self, relief = SUNKEN, highlightthickness = 0, font = 
normal_font, background = "white")
            self.entry.grid(row = r, column = 1, padx = 10, sticky = "we")
            self.entry.pack()

            setattr(self, name + '_entry', self.entry)
            r += 1
```

```
        # Les boutons
        bouton = Button(self, text="Calculer", anchor = "center", font = bold_font,
command=self.action)
        bouton.pack()

        # Les messages
        self.result = Label(self, anchor = "w", text = "", font = normal_font)
        self.result.pack()

    def action(self):
        try:
            # appel au controleur
            self.check_entries()
            # appel au modele
            self.result['text'] = "Rebond : %d" % self.fall.compute()
        except FallError, e:
            self.message(str(e))

    def check_entries(self):
        for (name, fall_func, fall_err_func, desc) in [
              ("high", self.fall.setHigh, self.fall_err.high, "Hauteur de chute"),
              ("rate", self.fall.setRate, self.fall_err.rate, "amortissement sur rebond")
            ]:
            entry = getattr(self, name + '_entry')

            # affectation des donnees du modele si entrees correctes
            fall_func(fall_err_func(entry.get()))

    def message(self, msg):
        self.result['text'] = msg

### le modele ###
class Fall:
    """
    Classe implementant les methodes pour les calculs de la chute de la balle
    """
    def __init__(self, limit = 1, high = None, rate = None):
        self._limit = limit     # en mm
        self._high  = high      # en m
        self._rate  = rate      # en %

    # les accesseurs
    def setHigh(self, high):
        self._high = high

    def setRate(self, rate):
        self._rate = rate

    # le traitement
    def compute(self):
        """
        Calcul du nombre de rebonds
        """
        rebond = 0
        h      = self._high * 1000
        while h > self._limit:
            h *= (1 - self._rate)
            if h > self._limit:
                rebond += 1
        return rebond

### le controleur ###
class FallError:
    """
    Classe de validation des donnes du masque de saisie de la vue
```

```
    """
    def __init__(self, msg = None):
        self.msg = msg

    def __str__(self):
        return self.msg

    def high(self, high = None):
        """
        consultation de la hauteur de chute
        """
        h = high
        if high is not None:
            try:
                h = float(high)
                if h <= 0:
                    raise FallError("La hauteur de chute doit être > 0")
            except ValueError:
                raise FallError("La hauteur de chute doit être un nombre réel")
        return h

    def rate(self, rate = None):
        """
        consultation de l'amortissement au rebond
        """
        r = rate
        if rate is not None:
            try:
                r = float(rate)
                if r < 0 or r > 1:
                    raise FallError("L'amortissement doit être sur un interval [0..1]")
            except ValueError:
                raise FallError("L'amortissement doit être un nombre réel")
        return r

##### programme principal ####
dialogue = Dialogue()
dialogue.pack()
dialogue.mainloop()
```