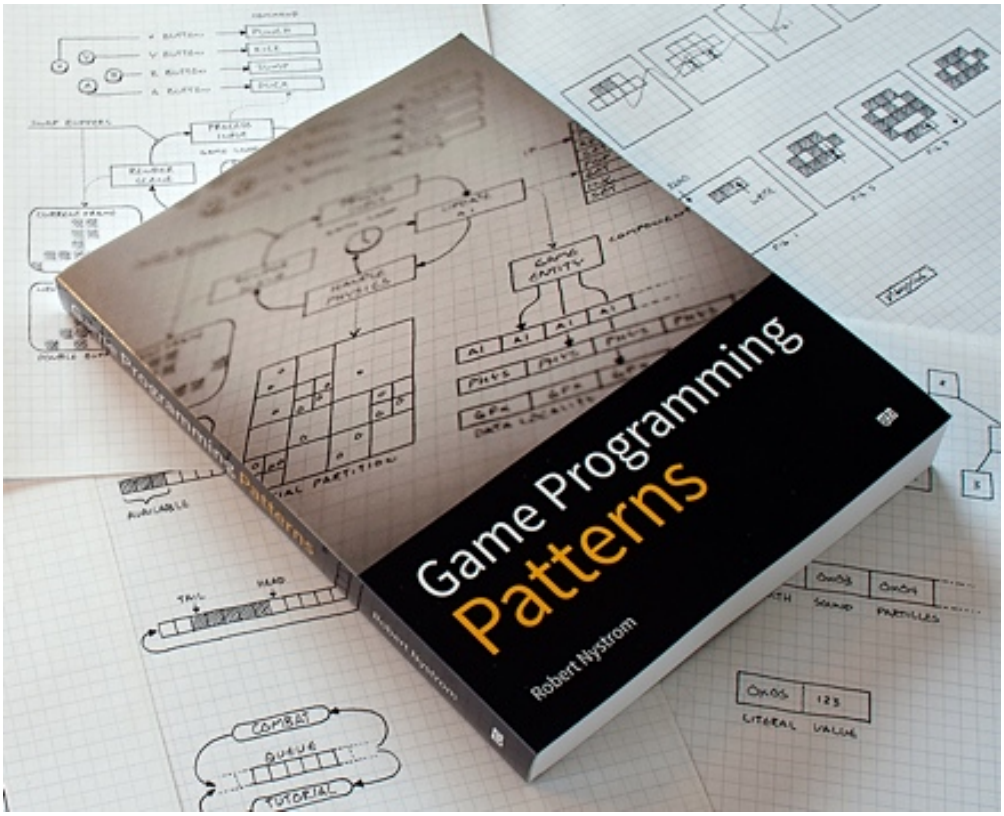
RAPPORT MVC



03/04/2016

Atelier IHM

Thibaut Terris – G1

Basil Dalié – G1

Table des matières

QUESTION 1

Le code du modèle de jeu ne suit pas l'architecture MVC ?

La structure du code

L'architecture MVC repose sur le fait que le modèle (partie fonctionnelle) du code et la vue (partie graphique) sont tous les deux dans des parties de code bien distinct et le contrôleur se charge de faire le lien entre chaque partie. Il existe des variantes au modèle MVC ou il y a un lien très étroit entre la vue et le modèle qui passe par le pattern Observer/Observable qui rajoute une couche d'abstraction à cette architecture.

Dans le code du modèle de jeu on a affaire à un contrôleur intégré. Ici toutes les couches (modèle-vue-contrôleur) peuvent être contenue dans un même fichier cependant nous pouvons faire des distinctions entre les différentes couches selon les méthodes utilisées au sein même du fichier. Il y a une méthode render() qui s'occupe uniquement de mettre à jour la vue et la méthode update() qui s'occupe uniquement de mettre à jour le modèle.

Les interactions

Une autre particularité de l'architecture MVC vient des interactions entre les différentes couches. Dans tous les cas il y a une attente d'une interaction utilisateur pour actualiser le modèle et/ou la vue via le contrôleur (ou non selon les variantes). Il y a des règles de communications qui existe. Par exemple s'il y a une action utilisateur au niveau de la vue c'est soit le modèle qui va directement être avertis (grâce à une couche d'abstraction) ou soit le contrôleur qui va avertir le modèle. Ceci est possible grâce à la mise en place du pattern observer/Observable ou la vue va s'abonner au élément du modèle qui l'intéresse et dés qu'un changement va être effectué sur un élément auxquels elle s'est abonné le modèle va l'en notifier.

Le modèle de jeu à une toute autre approche. Ici nous n'attendons pas qu'il y est une interaction utilisateur. Le modèle de jeu actualise tout le temps la vue et le modèle grâce à une boucle (d'où le nom de pattern game loop) qui se lance tous les X fois par s (avec un X qui n'est souvent très grand et donc non perceptible par l'utilisateur). Ici pas de pattern Observer/Observable. Si jamais un événement se produit et que le modèle est modifié il sera actualisé au prochain tour de boucle.

Initialisation

Dans une architecture MVC nous avons une classe qui initialise chaque couche du modèle en fonction des besoins de celle ci.

Dans modèle de jeu (dans un main) on crée l'objet qui hérite du modèle de jeu (dans le cadre des projets Devint la classe abstraite Jeu) qui elle même appel une méthode init() permettant de crée les objets nécessaire à son fonctionnement (la partie graphique, le modèle) et après on lance la boucle (loop()) qui ne s'arrêtera qu'a l'arrêt du jeu.

Donnez les parties du code qui ne respectent pas ce pattern. Argumentez

```
public abstract class Jeu extends Fenetre {
public abstract class Fenetre extends JFrame {
```

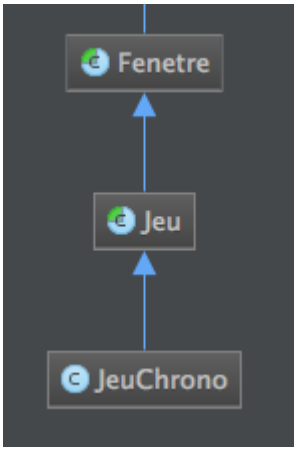
Comme vous pouvez le voir ci dessus la classe abstraite jeu qui représente le modèle de jeu extends de Fenetre qui extend elle même de JFrame. Comme nous le savons JFrame est un élément graphique.

La classe jeu présente les méthodes init() (qui entre autre implémente les listeners des touches de clavier utiles pour la classe) et reset() qui s'occupe du partiellement du modèle et la méthode update() qui s'occupe totalement du modèle.

Pour finir cette classe présente la méthode loop() qui appel à la suite update() et render() ce qui peut être considéré comme la partie contrôleur permettant de faire un lien indirect entre le modèle et la vue.

Nous avons donc au sein d'une même classe tous les éléments d'une architecture MVC qui normalement devrait être extériorisé dans plusieurs fichiers pour plus de flexibilité.

En quoi l'architecture mise en place limite l'évolution du code ?



Comme nous pouvons le voir dans le schéma ci dessus nous avons tout d'abord la classe Fenetre qui définit le "squelette graphique" puis la classe abstraite Jeu héritant de Fenetre qui met en place tous les éléments nécessaires au bon fonctionnement du jeu et enfin une classe qui hérite de Jeu (ici JeuChrono) qui elle représente vraiment le Jeu que l'on veut développer.

En partant de là nous somme figé dans un modèle car nous ne pouvons pas avoir deux vue différente pour un même modèle sans devoir recréer un fichier qui redéfinit chaque méthode (update(), render()...) et les interactions qui vont avec.

Il est beaucoup plus difficile de travailler en équipe sur modèle de jeu car le contrôleur est intégré ce qui veut dire qu'il y a un fichier commun que tout le monde doit modifier au moins une fois pour pouvoir réagir aux interactions de l'utilisateur.

Il est toujours possible de se débrouiller pour faire évoluer le code avec modèle de jeu, mais la tache est plus difficile à mettre en place qu'avec un MVC ou tous les fichiers sont déjà externalisés et donc l'architecture est plus claire. Il en est donc du travail du/des développeurs pour rendre ce modèle plus flexible.

QUESTION 2

Comment pouvez-vous modifier le code du modèle de jeu pour qu'il respecte MVC ?

Pour rendre modèle de jeu compatible avec une architecture MVC il faudrait externaliser les méthodes update() et render() dans un premier temps. On pourrait pour cela crée une classe abstraite Model ayant pour méthode abstraite update(Object object) et une classe abstraite View ayant pour méthode abstraite render(Object object) la classe Jeu jouerait le rôle de contrôleur et appellerais ces méthodes selon le contexte.

Il faudrait également remettre en place le concept d'Observer/Observable en permettant au contrôleur de pouvoir s'abonner à des éléments de la vue. Selon les intéractions utilisateurs elle pourrait appeler ainsi soit update(Object object) pour mettre à jour le modèle ou render(Object object) pour mettre à jour la vue ou encore les deux si besoin.

Ainsi chaque classe View aurait une méthode notify(Observer observer, Object object) qui enverrais les données modifiées à chacun de ses observers ajouter grâce à une méthode addObserver().

```
/**
 * Project : modele-de-jeu
 * Created by Thibaut/Basil on 03/04/2016
 */
abstract class AbstractView  extends Observable{



    abstract void render(Object object);

    @Override
    public synchronized void addObserver(Observer o) {
        super.addObserver(o);
    }

    @Override
    public synchronized void deleteObserver(Observer o) {
        super.deleteObserver(o);
    }

    @Override
    public void notifyObservers() {
        super.notifyObservers();
    }


}
```

La méthode init() serait en faite remplacée par le constructeur du contrôleur abstrait et permettrait d'initialiser la vue et le modèle.

```java
/**
 * Project : modele-de-jeu
 * Created by Thibaut on 03/04/2016
 */
abstract class AbstractController implements Observer{

    private AbstractView abstractView;
    private AbstractModel abstractModel;

    public AbstractController(AbstractView abstractView, AbstractModel abstractModel) {
        this.abstractView = abstractView;
        this.abstractModel = abstractModel;
    }


    @Override
    public void update(Observable o, Object arg) {
        abstractView.render(abstractModel.update(arg));
    }
}
```

```java
/**
 * Project : modele-de-jeu
 * Created by Thibaut/Basil on 03/04/2016
 */
abstract class AbstractModel extends Observable {

    abstract Objects update(Object arg);
}
```

Avec cette nouvelle architecture, la Vue est complètement indépendante du modèle et le contrôleur Jeu permet de faire le lien avec une couche d'abstraction permettant de faire communiquer le modéle et la vue entre eux. Il est donc tout à fait possible dans cette configurantion là de créer deux vues pour un même modèle par exemple en réécrivant seulement la couche d'abstraction à chaque fois. Le code devient à ce moment-là beaucoup plus flexible.

QUESTION 3

Qu'elles sont les spécificités des jeux DEVINT ?

La principale caractéristique des jeux DeVINT est qu'ils s'adressent à un public de personne en situation de handicap visuel et/ou cognitif, des aménagements doivent être entrepris en fonction du type de handicap et du profil des utilisateurs afin de les rendre plus accessibles. Par exemple, chaque jeu doit avoir une consigne claire et concise, et toutes les indications doivent être prononcées oralement (par enregistrement ou synthèse vocale). Du point graphique, le jeu est limité à une seule fenêtre en plein écran, les contrastes doivent être accentués, et les textes doivent être suffisamment grands. L'utilisateur peut aussi avoir le choix de changer les couleurs à tout moment durant le jeu.

Ensuite, les raccourcis clavier doivent respecter un certain nombre de contraintes : certaines touches sont réservées pour des fonctions prédéfinies : par exemple, la touche F1 doit répéter l'objectif du jeu, la touche F2 doit donner une aide, etc.

Est-ce que la solution MVC respecte les besoins et si oui comment ? Comparez les 2 architectures et leurs avantages par rapport aux besoins de DEVINT.

L'utilisation d'une architecture MVC pourrait parfaitement convenir pour la création des jeux DeVINT, en effet aucune des caractéristiques précédemment citées n'est incompatible avec cette architecture. Seulement, on peut se demander quels en seraient les bénéfices par rapport au modèle de jeu fourni. En effet, un des principaux avantages de l'architecture MVC est qu'elle permet de proposer différentes présentations pour un même ensemble de données du modèle, or les jeux DeVINT étant composés d'une unique fenêtre, cette caractéristique aurait un intérêt limité. De plus, le code obtenu serait plus complexe, et n'apporterait pas nécessairement d'avantage.

De plus, dans le cas d'une architecture MVC, les changements d'état du modèle et de la vue sont déclenchés par des actions utilisateurs, or selon le type de jeu développé on peut avoir besoin d'un rafraîchissement en continu(par exemple pour le déplacement d'un objet sur l'écran), pour cela, on peut considérer que le patron de conception game loop est plus adapté. Par ailleurs, on pourrait décider de déclencher une aide automatique lorsque l'utilisateur ne fait aucune action durant un temps déterminé, avec le patron game loop, on peut se contenter de compter le nombre d'itérations de la boucle principale et déclencher l'aide quand ce dernier atteint un certain seuil. Avec une architecture MVC, on aurait besoin d'utiliser une minuterie ce qui entraînerait une complexité plus importante.

Néanmoins, un avantage non négligeable avec l'architecture MVC est une plus grande modularité, ce qui aboutit à un code plus facilement maintenable. En effet, avec le patron de conception Game Loop, si on décide d'améliorer l'interface graphique, on sera peut-être amené à modifier des fichiers contenant la logique algorithmique du jeu. Au contraire, avec un modèle MVC le code relatif à l'initialisation et la mise à jour de l'IHM est isolé dans des fichiers séparés, on peut ainsi travailler sur l'interface graphique sans craindre une régression dans la partie algorithmique du Jeu . Ce qui est plus en accord avec le principe SOLID ouvert/fermé.