Mise en pratique du MVC sur le Jeu Memory

L'objectif de ce TP est de développer une application qui respecte le patron de conception MVC (Modèle, Vue, Contrôleur). Il s'agit d'un jeu de Memory. Ce jeu peut se jouer à plusieurs mais nous nous limiterons à un seul joueur. Nous en ferons donc un genre de réussite.

Ce jeu se joue avec un jeu de cartes composé de n familles de 2 cartes, mais pourrait être généralisé à n familles de p cartes. Les cartes sont placées sur un tapis, face cachée. Dans le cas d'un jeu de 32 cartes, on peut faire 4 rangées de 8 cartes. Le joueur retourne deux cartes, l'une après l'autre. Si elles appartiennent à la même famille (même valeur et même couleur dans un jeu de 32 cartes), le joueur les prend, sinon il les retourne de nouveau pour qu'elles soient face cachée.

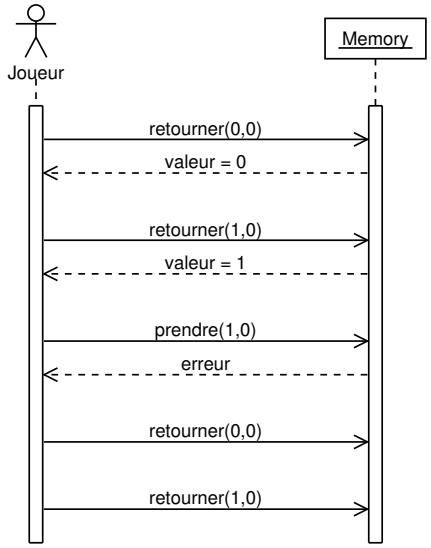
L'objectif est de reconstituer toutes les familles et donc d'enlever toutes les cartes du tapis avec le minimum d'essais.

Exercice 1 : Modélisation UML du Memory

On peut commencer par une rapide modélisation UML du Memory.

1.1 *Cas d'utilisation.*

1.2 *Diagrammes de séquence contextuels.* Voici un exemple d'un tel diagramme.



Exercice 2 : Compréhension de l'architecture MVC

La réalisation de l'application a été commencée. Elle met en œuvre le patron MVC.

Le principe du MVC est de séparer clairement le Modèle, la Vue et le Contrôleur. Le modèle correspond aux données et à la logique métier de l'application. Il est indépendant de l'interface utilisateur et doit pouvoir être réutilisé quelque soit la ou les interfaces utilisateurs développées.

Le modèle est composé des classes JeuCartes et TapisMemory qui représentent les structures de données et MemoryArbitre qui décrit la logique du jeu.

La vue constitue une présentation des données du modèle. Par exemple, voici ce que peut afficher une vue dans le cas d'une vue textuelle.

```
[ 2 ]   [---]   [XXX]   [XXX]   [---]   [XXX]
[XXX]   [---]   [XXX]   [---]   [XXX]   [XXX]
[XXX]   [---]   [XXX]   [XXX]   [---]   [XXX]
[---]   [XXX]   [---]   [---]   [XXX]   [---]
```

FIGURE 1 – Vue textuelle d'une partie de Memory

Le jeu comporte 4 rangées de 6 cartes. Les familles sont numérotées de 0 à 11. Les positions marquées « `---` » correspondent à des cartes enlevées du tapis. Celles marquées « `XXX` » sont des cartes face cachée. La position marquée `2` correspond à une carte appartenant à la famille 2.

Le contrôleur traduit les actions de l'utilisateur (saisie clavier, clique sur un bouton...) en événements de l'application. Ce sont les événements qui ont été identifiés lors de la définition des diagrammes de séquence contextuels. Ils sont alors envoyés sur le modèle qui les interprète et se met à jour.

Les modifications doivent être aussi répercutées sur les vues. Soit c'est le contrôleur qui en a la charge (le modèle est alors dit passif), soit c'est le modèle qui le fait (le modèle est alors dit actif). Dans la suite, nous considérons que le modèle est actif.

Ce mécanisme est décrit sur la figure 2 qui illustre l'interaction entre l'acteur Joueur et le système Memory dans le cas d'une interface graphique. Le joueur clique sur des boutons correspondant aux cases ou à la prise de toutes les cartes retournées. Ces événements sont traduits par le contrôleur en événements « *retourner* » et « *prendre* » émis vers le modèle du Memory qui indique ses changements à la vue.

Bien entendu, ce serait une mauvaise conception de considérer que le modèle doit connaître toutes les vues possibles et la manière de les mettre à jour. La solution est alors d'utiliser le patron de conception Observateur. Les vues sont alors des observateurs du modèle. Le modèle avertit les observateurs inscrits des changements qu'il subit. Les observateurs, ici les vues, mettent à jour l'affichage soit en exploitant les informations fournies lors de l'avertissement, soit en interrogeant explicitement le modèle. Ainsi, la vue doit connaître le modèle mais pas l'inverse.

Pour séparer les caractéristiques de chaque composant, des interfaces ont été écrites pour spécifier le modèle, la vue et le contrôleur du jeu ainsi que l'observateur du modèle.

2.1 *Le modèle.* Le modèle est découpé en deux parties. La première modélise les données métiers et les opérations associées. Elle est composée de deux classes l'une représentant un jeu de cartes (JeuCartes), l'autre le tapis du Memory (TapisMemory). Notons que toutes les règles du jeu ne sont pas définies dans cette partie. Par exemple, dans la classe TapisMemory, s'il est nécessaire d'avoir retourné une carte face visible pour pouvoir accéder à sa valeur, rien ne dit combien de cartes il est possible de retourner. On peut par exemple les rendre toutes visibles.

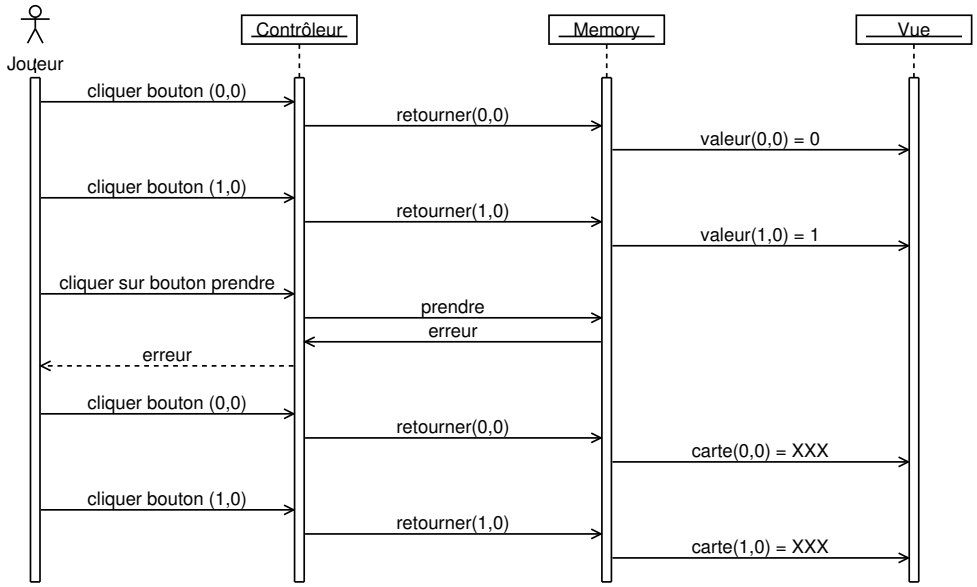


FIGURE 2 – Le scénario de la question 1.2 en détaillant l'interaction Joueur et Memory.

La deuxième partie décrit la logique de l'application, donc les règles du jeu en fonction des actions de l'utilisateur. Ici, c'est la classe MemoryArbitre. On retrouve les opérations identifiées dans le diagramme de séquence contextuel (retourner une carte, prendre une carte) mais aussi des opérations de plus haut niveau comme retourner sans paramètre qui retourne ou enlève toutes les cartes visibles.

C'est ici que sont gérées les deux phases du jeu. La première consiste à retourner les deux cartes. La seconde consiste soit à prendre les deux cartes si elles constituent une famille, soit à les retourner de nouveau.

2.1.1 Indiquer à quoi correspondent les opérations qui apparaissent dans l'interface MemoryModele.

2.1.2 Écrire un programme TestMemoryScenario1 qui correspond au diagramme de séquence contextuel donné à la question 1.2.

Remarque : pour voir ce qui se passe après chaque événement envoyé au (opération appliquée sur le) MemoryArbitre, on pourra utiliser la méthode afficherTapis qui affiche le tapis de la manière présentée en figure 1.

2.1.3 Ajouter une trace après chaque opération sur la classe MemoryArbitre est fastidieux et rend le texte du programme de test peu lisible. Pour éviter ce défaut, on peut définir un observateur (réalisation de MemoryObservateur) qui affiche le tapis après chaque modification.

Indiquer à quoi correspondent les opérations qui sont déclarées dans l'interface MemoryObservateur.

2.1.4 Écrire cet observateur du MemoryArbitre et modifier le programme de test précédent pour le prendre en compte. On peut noter que l'on peut directement afficher le Memory car la méthode `toString` de `Object` a été redéfinie pour afficher le tapis de jeu.

2.2 *Les vues.* Une vue présente à l'utilisateur une partie du modèle. À ce titre, il est logique que la vue ait un accès sur le modèle pour récupérer les informations à afficher. Pour savoir quand les informations ont changé et modifier en conséquence la vue, il est intéressant de s'appuyer sur les observateurs proposés par MemoryArbitre.

2.2.1 *Comprendre l'interface MemoryVue.* Expliquer l'interface MemoryVue et son lien avec MemoryObservateur.

2.2.2 *Intérêt de la classe MemoryVueClasse.* Expliquer l'intérêt de la classe MemoryVueClasse. Expliquer l'intérêt de définir à la fois la classe MemoryVueClasse et l'interface MemoryVue.

2.2.3 *Ajout d'une vue.* Indiquer ce qu'il faut faire pour initialiser une vue en cours de partie.

2.2.4 *Ajout d'une vue texte.* Transformer l'observateur texte précédent en une vue texte et tester. Dans le cas de la vue texte, désactiver la vue consiste à la supprimer de la liste des observateurs et l'activer consiste à l'ajouter de nouveau.

2.2.5 *Ajout d'une vue graphique.* Définir une vue graphique qui affiche les cartes en utilisant les images fournies (séries animaux ou musique). On utilisera des boutons pour représenter chaque carte, ce bouton contiendra l'image (`#.gif` où `#` est la valeur de la carte) de la carte ou le dos de la carte (`dos.gif`).

On ne vérifiera pas s'il y a suffisamment de cartes pour la taille du tapis considéré.

2.2.6 Compléter le programme de test précédent pour qu'il propose les deux vues : la vue graphique et la vue textuelle.

2.3 *Les contrôleurs.* Un contrôleur permet de traduire les ordres de l'utilisateur en événements sur le modèle. Un exemple de contrôleur est fourni. Il s'agit de la classe MemoryControleurSwingLigneColonne.

2.3.1 Écrire un programme de test avec un tapis de taille 3 par 6 qui propose la vue texte, une vue graphique avec les animaux, une seconde vue graphique avec les instruments de musique et, enfin, le contrôleur graphique.

2.3.2 Expliquer la classe MemoryControleurSwingLigneColonne.

2.3.3 Il serait plus pratique pour l'utilisateur de pouvoir directement cliquer sur la vue graphique pour retourner les cartes. Ajouter un contrôleur graphique sur cette vue.

2.4 *Statistiques.* On veut compléter le jeu précédent en y intégrant un compteur du nombre de coups joués, ceux qui ont été infructueux et ceux qui ont permis de prendre des cartes.

2.4.1 Définir le modèle correspondant.

2.4.2 Définir un observateur sur ce modèle.

2.4.3 Définir une vue texte.

2.4.4 Définir une vue graphique qui pourra être intégrée à la vue graphique du Memory.