Université d'Aix-Marseille III - Licence de math-info 2ème année

I4 : Programmation objet - TD n°5 : héritage et polymorphisme

1 Héritage (début de l'examen de janvier 2005)

Nous allons créer un certain nombre de classes permettant de modéliser Freecell, un jeu de cartes solitaire bien connu des utilisateurs de Windows. Rappel : un jeu de 52 cartes contient treize *valeurs* de cartes (as, 2, 3, ... 9, 10, valet, dame et roi) apparaissant sous chacune des 4 *"couleurs"* (carreau, pique, coeur, trèfle). (Le carreau et le coeur sont de coloration rouge mais ils constituent des "couleurs" différentes, tout comme le pique et le trèfle qui sont de "couleur" différente bien que de coloration noire tous les deux). L'aire de jeu contient 16 emplacements : 8 *colonnes*, 4 *bases* et 4 *cellules*. Au début du jeu, les 52 cartes du jeu sont réparties aléatoirement sur les 8 *colonnes*. Le but est d'empiler les cartes dans l'ordre croissant et en respectant leur "couleur" sur les 4 *bases*. Les 4 *cellules* sont des emplacements pouvant accueillir chacun une carte. On ne peut déplacer qu'une seule carte à la fois, depuis une colonne ou une cellule vers une colonne, une cellule ou une base. Ainsi, on ne peut pas retirer une carte dès lors qu'elle est sur une base. Il y a des conditions pour pouvoir placer une carte au sommet d'une pile et ces conditions dépendent du type d'emplacement où se trouve la pile de cartes :

- On ne peut empiler une carte sur une base que si elle est vide et que la carte est un as ou si le sommet de la base est une carte de même "couleur" et de valeur immédiatement inférieure. Ex : si une base a le 3 de coeur en son sommet, on ne peut y poser que le 4 de coeur.
- On ne peut empiler une carte sur une colonne que si elle est vide ou si son sommet est de valeur immédiatement supérieure et de coloration différente : on ne peut placer un pique ou un trèfle que sur un carreau ou un coeur, et réciproquement. Ex : si le sommet d'une colonne est le valet de trèfle, on ne peut y déposer que le 10 de carreau ou le 10 de coeur.
- On ne peut placer une carte sur une cellule que si elle est vide.

1. *Ecrire la classe* `Carte` *ayant un constructeur à deux paramètres, le premier étant un entier (de 1 à 13) représentant la valeur de la carte (d'as à roi) et le deuxième étant de type* `String` *et représentant la "couleur" de la carte (pique, coeur, carreau, trèfle). En plus des accesseurs que vous jugerez utile de définir, vous définirez la méthode publique* `boolean estRouge()` *qui renvoie* `true` *si la carte est un carreau ou un coeur.* Remarque : en Java, on peut tester l'égalité de deux chaînes de caractères (de type `String`) grâce à l'opérateur `==`.

2. Dans un premier temps, nous allons modéliser un paquet de cartes qui accepte qu'on lui empile une carte sans condition (sans prendre en compte la carte qui se trouve en son sommet). On suppose qu'un paquet ne pourra jamais contenir plus de 52 cartes (sans qu'on ait besoin de le vérifier quand on fait une opération d'empilement sur le paquet). Le paquet est initialement vide.

Ecrire la classe `Paquet` *qui modélise une pile de cartes et contient les méthodes publiques* `boolean vide()`, `Carte sommet()`, `void empile(Carte c)` *et* `Carte depile()`.

3. *Ecrire la classe* `Colonne` *qui hérite de* `Paquet`. Elle contient la méthode supplémentaire `boolean empilable(Carte c)` permettant de déterminer si `c` peut être empilée sur la colonne en fonction des conditions données plus haut. Elle redéfinit aussi la méthode `empile` afin qu'elle vérifie que la carte à empiler est empilable et qu'elle affiche l'impossibilité d'empiler, le cas échéant.

4. *Ecrire la classe* `Cellule` *en la faisant hériter de* `Colonne`.

5. *Pourquoi la classe* `Base` *ne doit-elle pas hériter de* `Paquet` *ou d'une de ses descendantes ? Ecrire la classe* `Base` *en réutilisant (intelligemment) la classe* `Paquet` *(mais sans en hériter).*

2 Polymorphisme (début d'un exercice de l'examen de juin 2005)

Dans cet exercice, on considère l'existence de deux classes `A` et `B` dont la seule chose qu'on sache d'elles est qu'elles contiennent chacune un constructeur sans paramètre. Le but est de définir partiellement une classe pouvant mémoriser des instances de `A` et de `B` dans une liste chaînée.

2.1 Dans cette question, les classes `A` et `B` sont déjà écrites et impossibles à modifier au moment où on veut écrire la classe `ListeAouB`. Le but est d'écrire partiellement la classe `ListeAouB` qui représente une liste de longueur quelconque pouvant mémoriser uniquement des instances de `A`, de `B` ou d'une classe qui hérite de `A` ou de `B`. *Ecrivez uniquement les attributs et les constructeurs nécessaires à la classe* `ListeAouB` *afin qu'il soit possible par exemple d'écrire ceci :*

```
// Création d'une liste contenant un A puis un B.
ListeAouB liste1 = new ListeAouB(new A(),
                                 new ListeAouB(new B(),
                                               ListeAouB.VIDE));
// Création d'une liste contenant un B puis un A puis un A.
ListeAouB liste2 = new ListeAouB(new B(),
                                 new ListeAouB(new A(),
                                               new ListeAouB(new A(),
                                                             ListeAouB.VIDE)));
```

Il n'est pas demandé d'écrire les méthodes de manipulation de listes (ajout, retrait d'élement, etc) : uniquement les constructeurs. Les constructeurs permettent de créer une liste à partir d'un élément (de type `A` ou `B`) et d'une autre liste. La liste créée est la même que celle passée en paramètre mais avec le nouvel élément placé en tête de liste. On rappelle que la classe `Object` est la racine de la hiérarchie de toutes les classes de Java et donc que toutes les classes en héritent directement ou indirectement.

2.2 *Ajoutez à la classe* `ListeAouB` *la méthode* `int longueur()` *permettant de déterminer la longueur de la liste.*

2.3 *Ajoutez à la classe* `ListeAouB` *les méthodes* `boolean appartient(A)` *et* `boolean appartient(B)` *permettant de déterminer si l'élément passé en paramètre de la méthode appartient à la liste.*

2.4 Comme les constructeurs à deux paramètres définis en question **2.1** ne sont pas pratiques pour construire des listes, nous voudrions définir en plus des constructeurs à un, deux ou trois paramètres permettant de créer plus facilement des listes contenant un, deux ou trois éléments, de la manière suivante :

```
// Création d'une liste contenant un A.
ListeAouB liste1 = new ListeAouB(new A());
// Création d'une liste contenant un B puis un A.
ListeAouB liste2 = new ListeAouB(new B(), new A());
// Création d'une liste contenant un A puis un B puis un A.
ListeAouB liste3 = new ListeAouB(new A(), new B(), new A());
```

Si on veut que toutes les combinaisons de listes à un, deux ou trois éléments de type `A` *ou* `B` *(mais pas d'un autre type) puissent être créées grâce à de tels constructeurs, combien de constructeurs faut-il définir ? Justifiez.*