UE Conception Orientée Objet

TD Héritage

Exercice 1 : Transport de marchandises

On souhaite modéliser en `java` le calcul de coûts de transport de marchandises. Les marchandises transportées seront des instances de la classe `Marchandise` :

Marchandise
- poids : int - volume : int
+ Marchandise (poids :int, volume :int) +getPoids() : int +getVolume() : int

Les marchandises sont transportées sous la forme de cargaisons. Les seules fonctionnalités publiques des cargaisons sont :

ajouter qui permet d'ajouter une marchandise dans cette cargaison si cela est encore possible.

cout qui retourne, sous la forme d'un nombre entier d'euros, le coût total du transport de cette cargaison.

Une cargaison est par ailleurs également caractérisée par la distance sur laquelle elle est transportée. Ce renseignement est communiqué à la construction de la cargaison sous la forme d'un nombre entier de kilomètres. On précise qu'une cargaison ne peut réunir qu'un nombre limité de marchandises qui dépend d'un encombrement total de ces marchandises à ne pas dépasser. Cet encombrement est soit le poids total, soit le volume total des marchandises, selon le type de transport utilisé. Ce dernier influe aussi sur le calcul du coût de transport de la cargaison qui, de la même façon, dépent de l'encombrement des marchandises de la cargaison. On distingue donc plusieurs types de cargaisons selon le moyen de transport utilisé. On peut cependant trouver une certain nombre de caractéristiques communes à toutes les cargaisons que vous devrez identifier. Les différents types de cargaison et leurs caractéristiques sont donnés par le tableau suivant :

type	encombrement	coût	limite
`Fluviale`	`poids`	`distance` $\times \sqrt{\texttt{encombrement}}$	`encombrement` ≤ 300000
`Routiere`	`poids`	$4 \times$ `distance` $\times$ `encombrement`	`encombrement` ≤ 38000
`Aerienne`	`volume`	$10 \times$ `distance` $+ 4 \times$ `encombrement`	`encombrement` ≤ 80000
`AerienneUrgente`	`volume`	$2 \times$ le coût d'une cargaison `Aerienne`	`encombrement` ≤ 80000

Q 1 . Proposez une hiérarchie de classes permettant de modéliser au mieux les différentes classes de cargaisons et écrivez ces classes en `java`.

Exercice 2 : Casse-Briques

On s'intéressera dans cet exercice à la modélisation d'un certain nombre d'entités d'un programme d'un jeu de casse-briques.

Rappelons brièvement le principe de ce genre de jeu : il se joue seul et le joueur contrôle une raquette qu'il peut déplacer latéralement, celle-ci lui permet de renvoyer une balle vers des briques qui sont cassées lorsqu'elles sont frappées par la balle, tout en renvoyant la balle. Le but est de détruire toutes les briques sans perdre la balle, en fait généralement le joueur dispose de quelques balles en réserve qu'il peut utiliser lorsqu'il perd une balle lui donnant ainsi quelques droits à l'erreur.

Ce jeu d'arcade fait partie des ancêtres des jeux vidéo et a connu des variantes dans lesquelles les briques sont de différents types et, donc, lorsqu'elle sont cassées déclenchent des effets variables. La représentation graphique (couleur, forme, ...) des briques dépend alors du type de celle-ci.

La classe `Joueur` modélise l'entité joueur du casse-briques, avec notamment sa raquette et ses balles en réserve.

La constante `LE_JOUEUR` désigne le joueur en train de jouer. Le paramètre `echelle` de `modifieTailleRaquette` est un facteur multiplicatif appliqué à la taille de la raquette et le paramètre `variation` de `modifieBallesEnReserve` précise combien il faut ajouter (si positif) ou retirer (si négatif) de balles au joueur.

Joueur
- nbBallesEnReserve : int - tailleRaquette : float - nbPoints : int + LE_JOUEUR : Joueur
- Joueur() + getNbBallesRestantes() : int + getTailleRaquette() : float + getNbPoints() : int + ajoutePoints(nb : int) + modifieTailleRaquette(echelle : float) + modifieBallesEnReserve(variation : int)

Les briques sont toutes de type `Brique`. Elles sont caractérisées par un nombre de points (reçus par le joueur lorsqu'il casse la brique) et par leur représentation graphique qui est un objet de type `BriqueGraphique` (voir ci-dessous). Il existe des méthodes d'accès à ces données.
Les briques disposent également d'une méthode `estCassée()` qui est déclenchée lorsqu'elles sont touchées par une balle. Cette méthode incrémente alors le nombre de points du joueur et déclenche l'effet associé à la brique.
On trouve les différents types de briques suivants avec leurs différents effets :

- les briques «*simple*» rapportent 100 points et n'ont aucun effet particulier,
- les briques «*jackpot*» rapportent aléatoirement de 1 à 10 fois plus de points qu'une brique de base et n'ont aucun effet particulier,
- les briques «*grande raquette*» rapportent autant qu'une brique simple et doublent la taille de la raquette du joueur,
- les briques «*bonus balle*» rapportent 3 fois plus qu'une brique simple et augmentent de 1 le nombre de balles en réserve du joueur,
- les briques «*piège*» rapportent 0 point et ont pour effet d'enlever un nombre de points variable, précisé à la construction de la brique, et de supprimer une balle de la réserve du joueur (si il lui en reste).

D'autres types de briques doivent pouvoir être envisagés et ajoutés facilement.
A chaque type de brique correspond un type de `BriqueGraphique`, voir Figure 1.

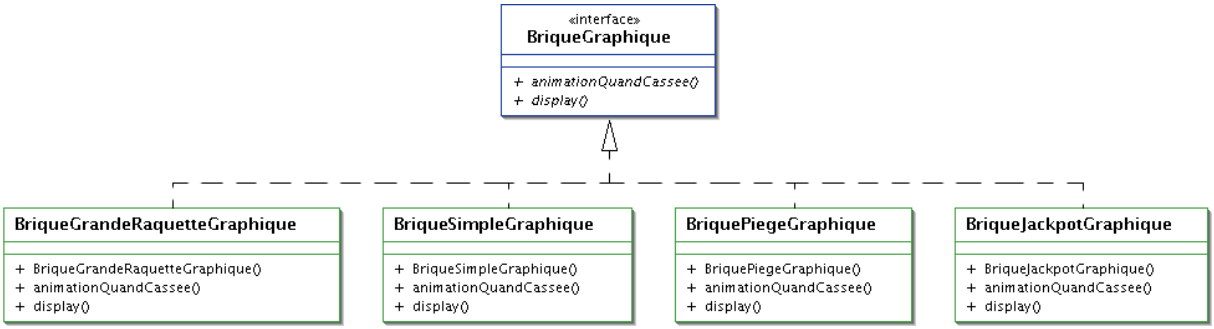

FIG. 1: Une hiérarchie de types pour les briques graphiques

Il permet d'afficher la brique (méthode `display`) ou de déclencher une animation graphique (méthode `animationQuandCassee`) quand la brique est cassée.

Q 1 . Donnez des diagrammes UML détaillés (attributs, constructeurs et méthodes avec paramètres et valeurs de retours) de la hiérarchie des types `Brique`.

Q 2 . Donnez le code java correspondant à votre solution pour représenter des briques *grande raquette* et des briques *bonus balle*. Le cas échéant vous donnerez également le cas des «super-types» nécessaires. (Aucun code n'est à fournir pour les classes de `BriqueGraphique` associées).

On propose d'enrichir les briques possibles : pour chaque type de brique déjà proposé, une variante est créée. Cette variante appelée *double effet* a comme conséquence de doubler le nombre de points attribués pour chacun des types ainsi que l'effet appliqué. Ainsi :

▷ une brique *bonus balle* «*double effet*» rapportera 6 fois plus qu'une brique simple et 2 nouvelles balles pour le joueur,

▷ une brique *jackpot* «*double effet*» rapportera aléatoirement de 2 à 20 fois plus de points qu'une brique de base et n'aura pas d'autre effet.

Evidemment pour chaque nouveau type de brique que l'on peut imaginer on veut pouvoir en avoir une version «double effet».
Graphiquement, une brique *double effet* se traduit par un objet graphique dont l'affichage est le même que celui de la brique de base doublée avec simplement l'affichage qui est cerné d'un trait rouge épais ; l'animation, elle, reste la même.

Q 3 . Faites une proposition de conception sous forme de diagramme UML pour gérer ces briques *double effet*, ainsi que leur vue graphique.
Vous indiquerez comment cette proposition se positionne par rapport à votre réponse à la question 1.

Q 4 . Donnez le code java pour les briques *double effet* (pas l'aspect composant graphique) conformément à votre proposition.

Q 5 . Donnez le code java permettant de créer une *brique bonus balle double effet*.