UE Conception Orientée Objet

Jeu de l'oie

Exercice 1 : Jeu de l'oie
L'objectif de cet exercice est l'implémentation du "jeu de l"oie" (*goose game*), bien connu des plus jeunes.

Le jeu
Ce jeu se présente sous la forme d'un plateau de 63 cases numérotées (plus une position de départ qui peut être considérée comme une 64ème case numérotée 0). Les différents joueurs, en nombre quelconque, jouent chacun leur tour. Ils lancent 2 dés à 6 faces et avancent leur pion d'un nombre de cases égal à la somme des dés. L'arrivée sur une case peut avoir plusieurs conséquences:

- si la case d'arrivée était déjà occupée, le pion du joueur remplace le pion déjà présent et ce dernier est placé sur la case précédemment occupée par le joueur. Il n'y a donc toujours qu'au plus un joueur par case, sauf pour la case de départ qui est un cas très particulier.
- si la case est une "case oie", le joueur double le score du dé et « rebondit » donc en avançant encore son pion d'autant de cases qu'indiqué par les dés ;
- si la case est une "case piège", le joueur ne pourra plus jouer tant qu'il restera dans cette case (c'est-à-dire jusqu'à ce qu'un autre joueur tombe également sur cette case et que la première situation décrite se produise) ;
- si la case est une "case d'attente", le joueur reste bloqué sans jouer pendant un nombre de tours prédéfini pour chaque case (sauf si un autre joueur arrive sur cette case et le renvoie donc vers sa case initiale, c'est au joueur arrivant de se retrouver en attente pour le nombre de tours initialement prévu pour la case) ;
- si la case est une "case de téléportation", le pion « rebondit » jusqu'à une autre case prédéfinie ;
- pour toutes les autres cases, il ne se produit rien de particulier.

On fait le choix que les « rebonds » dus à l'arrivée sur une case oie ou une case de téléportation ne se cascadent pas. C'est-à-dire que si le premier rebond fait arriver sur une case qui devrait elle-même produire un rebond, ce second rebond n'est pas appliqué.

Pour gagner et finir la partie, il faut qu'un joueur arrive exactement sur la case 63. Si un lancer de dés lui fait "dépasser" cette case, il revient en arrière du nombre de cases en excès. Par exemple, un joueur situé sur la case 57 fait 9 aux dés, il avance jusque la case 63 en dépensant 6 points et recule ensuite des 3 points restants pour arriver sur la case 60.

Dans la version originale du jeu, les "cases oie" sont les cases 9, 18, 27, 36, 45 et 54 ; les "cases piège" sont les cases 31 (le puits) et 52 (prison) ; la case 19 est une "case d'attente" pour 2 tours ; les "cases de téléportation" sont les cases 6 (cheval) qui envoie en 12, 42 (labyrinthe) qui envoie en 30 et 58 (tête de mort) qui renvoie en 1. Cependant par la suite il ne faudra pas se focaliser sur ces cases, on veut avoir la possibilité d'éventuellement personnaliser le jeu de l'oie implémenté en modifiant les effets de chacune des cases, voire même le nombre de cases du plateau. Un exemple de partie est donné en annexe.

Mise en œuvre Nous nous intéressons donc maintenant à la conception objet de ce jeu. L'analyse a dégagé le besoin d'un certain nombre d'entités à modéliser :

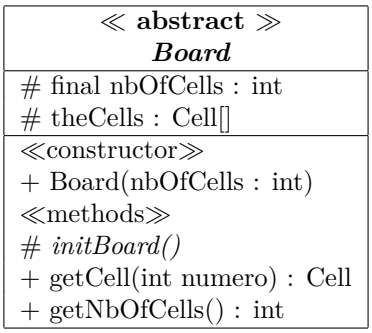
▷ la classe `Game` : elle est caractérisée par un plateau et une liste de joueurs et permet de jouer une partie du jeu de l'oie, voici son diagramme UML :

Game
thePlayers : List<Player> # board : Board
≪constructor≫ + Game(board : Board) ≪methods≫ + addPlayer(p :Player) +play() ...

La méthode `play()` déroule une partie jusqu'à son terme quand il y en a un (une partie peut être infinie si tous les joueurs sont dans des "cases piège", on ne testera pas cette situation).

Jouer une partie consiste à faire jouer successivement chaque joueur selon les règles décrites ci-dessus.

▷ La classe `Board` permet de représenter le plateau de jeu contenant les différentes cases, en voici le diagramme UML :



Un plateau est constitué d'un tableau de *nbOfCells* cases, plus une case numérotée 0 : la case de départ. On a donc en tout *nbOfCells+1* cases. Chaque case numéro i du plateau est rangée dans la case d'indice i du tableau.
Le constructeur fait appel à la méthode abstraite `initBoard()`. C'est dans cette méthode que sont créées les différentes cases (`Cell`) qui constituent le plateau.
Les sous-classes de `Board` peuvent donc personnaliser la configuration du plateau en fournissant leur implémentation de `initBoard()`.

▷ L'interface `Cell` est donnée en annexe. Quand un lancer de dés amène un joueur sur une case, c'est le résultat de l'invocation de `handleMove()` sur cette case qui indique quel est le numéro de la case réellement atteinte ce qui permet de gérer les effets des « rebonds ».

Toutes les entités seront à définir dans un paquetage appelé `goosegame`.

Q 1 . Donnez l'algorithme de la méthode `play` de la classe `Game`.
On ne cherchera pas à détecter les parties infinies (quand tous les joueurs sont dans des cases pièges).

Q 2 . Donnez les diagrammes UML des entités nécessaires à la modélisation du jeu de l'oie.

Q 3 . Proposez un code pour la classe abstraite `Board`.
Proposez ensuite un code pour une sous-classe de `Board`.

Q 4 . Comment gérer la "case 0" ?

Q 5 . Ecrivez le programme correspondant.

Annexes

Ces fichiers sont disponibles sur le portail.

La classe `goosegame.Player`

```
package goosegame;
import java.util.Random;
/** A player in the "jeu de l'oie" game */
public class Player {
    private static Random random = new Random();
    /** current cell of the player */
    protected Cell cell;
    /** name of the player*/
    protected String name;
    /**
     * @param name the name of this player
     * @param cell the strating cell of this player
     */
    public Player (String name, Cell cell){
        this.name = name;
        this.cell = cell;
    }
    /** @see Object#toString() */
    public String toString() {
        return this.name;
    }
    /** @return the current cell of the player */
    public Cell getCell() {
       return this.cell ;
    }
    /** changes the cell of the player
     * @param newCell the new cell
```

```
     */
    public void setCell(Cell newCell) {
       this.cell = newCell;
    }
    /** @return random result of a 1d6 throw*/
    private int oneDieThrow() {
       return Player.random.nextInt(6)+ 1;
    }
    /** @return random result of a 2d6 throw */
    public int twoDiceThrow() {
        int result = oneDieThrow() + oneDieThrow();
        return result;
    }
}// Player
```

L'interface `goosegame.Cell`

```
package goosegame;

/**
 * Interface for the cells of the goose game.
 *  Note that there  can be only  one player by cell,
 *  the starting cell (index 0) excepted.
 */
public interface Cell {
        /**
         * @return true if and only if the player in this cell can freely
         *  leave the cell, else the player must wait for another player to reach
         * this cell or wait a given number of turns
         */
        public boolean canBeLeft();

        /** returns the index of this cell */
        public int getIndex();

        /**
         * returns the index of the cell really reached by a player when the player
         *    reaches this cell.  For normal cells, the returned value equals
         *   <code>getIndex()</code> and is thus independent from
         *   <code>diceThrow</code>.
         * @param diceThrow the result of the dice when the player reaches this cell
         * @return the index of the actual cell where the player eventually
         * arrives when the given throw of dice sends the player in this cell
         */
        public int handleMove(int diceThrow);

        /** @return true iff a player is in this cell */
        public boolean isBusy();

        /** handles what happens when a player arrives in this cell
        * @param player the new player in the sell
        */
        public void welcomePlayer(Player player);

        /** @return  the player in this cell null if none */
        public Player getPlayer();
}// Cell
```

Un exemple de trace de partie

```
bilbo is in cell 0, bilbo throws 9 and reaches cell 9 (goose) and jumps to cell 18 (goose)
frodo is in cell 0, frodo throws 6 and reaches cell 6 (teleport to 12) and jumps to cell 12
sam is in cell 0, sam throws 9 and reaches cell 9 (goose) and jumps to cell 18 (goose) cell is busy,...
                                                                ... bilbo is sent to  start cell
bilbo is in cell 18, bilbo throws 5 and reaches cell 23
frodo is in cell 12, frodo throws 11 and reaches cell 23 cell is busy, bilbo is sent to cell 12
sam is in cell 18, sam throws 7 and reaches cell 25
bilbo is in cell 12, bilbo throws 7 and reaches cell 19 (waiting for 2 turns )
frodo is in cell 23, frodo throws 6 and reaches cell 29
sam is in cell 25, sam throws 7 and reaches cell 32
bilbo is in cell 19, bilbo cannot play.
frodo is in cell 29, frodo throws 10 and reaches cell 39
sam is in cell 32, sam throws 8 and reaches cell 40
bilbo is in cell 19, bilbo cannot play.
frodo is in cell 39, frodo throws 9 and reaches cell 48
sam is in cell 40, sam throws 3 and reaches cell 43
bilbo is in cell 19, bilbo throws 6 and reaches cell 25
frodo is in cell 48, frodo throws 12 and reaches cell 60
sam is in cell 43, sam throws 8 and reaches cell 51
bilbo is in cell 25, bilbo throws 6 and reaches cell 31 (trap)
frodo is in cell 60, frodo throws 7 and reaches cell 59
sam is in cell 51, sam throws 4 and reaches cell 55
bilbo is in cell 31, bilbo cannot play.
frodo is in cell 59, frodo throws 7 and reaches cell 60
sam is in cell 55, sam throws 11 and reaches cell 60 cell is busy, frodo is sent to cell 55
bilbo is in cell 31, bilbo cannot play.
frodo is in cell 55, frodo throws 6 and reaches cell 61
sam is in cell 60, sam throws 8 and reaches cell 58 (teleport to 1) and jumps to cell 1
bilbo is in cell 31, bilbo cannot play.
frodo is in cell 61, frodo throws 9 and reaches cell 56
sam is in cell 1, sam throws 8 and reaches cell 9 (goose) and jumps to cell 17
bilbo is in cell 31, bilbo cannot play.
frodo is in cell 56, frodo throws 7 and reaches cell 63
frodo has won.
```