Puissance 4

Code source ci-joint, également disponible via git : https://github.com/draialexis/connect4

Compilation :
```
javac -d build src/com/alexisdrai/connect4/*.java
src/com/alexisdrai/util/*.java
```

Exécution :
```
java -cp build com.alexisdrai.connect4.Main
```

On a réalisé un clone de *Puissance 4* en Java, sans frameworks. L'aspect orienté-objet du langage était difficile à exploiter, et ce jeu semble se prêter plus intuitivement à la programmation impérative. Cela dit, être capable d'encapsuler nos joueurs et nos cases nous a donné une perspective intéressante sur le projet et ses problématiques, et le programme développé ici avait un meilleur potentiel en termes de robustesse et d'extensibilité. Comme on le verra dans ce rapport, ce potentiel n'a pas pu être exploité complètement dans le temps imparti.

Structure

On a commencé par créer la Classe `C4Game`, qui représente le jeu de *Puissance 4*. Une instance de cette classe représentera alors une partie de *Puissance 4*. Dans cette classe, on a défini plusieurs classes internes et *enums* :

- `Cell`, privée, une cellule de la grille
  - définie par une `Color color` et une `EnumMap<Direction ,Color> allNeighbors`
  - comprend des méthodes pour nullifier sa `color`, la modifier ou l'obtenir
  - comprend des méthodes pour consulter son voisin dans une direction spécifique, ou l'ensemble de ses voisins (dans quatre directions seulement).
- `C4Player`, publique, un joueur de la partie (publique car on avait besoin d'y accéder depuis le `Main`)
  - définie par un `String name` et une `Color color`
  - comprend des getters pour `name` et `color` (non-modifiables, donc pas de setters)
  - comprend la méthode `chooseMove() : int`, qui invite l'utilisateur à choisir une colonne où insérer un jeton. Si l'utilisateur entre un charactère spécifique, tel que « s » pour *save*, un signal (en *int*) spécial sera renvoyé au `Main`, qui traitera le choix en conséquence.
- `C4Player_CPU`, privée (qui hérite de `C4Player` car c'est un `C4Player` (voir Bonus))

... deux *enums* :

- `Direction`, qui contient seulement `UP`, `RIGHT`, `DOWN`, et `LEFT`
- `Color`, qui contient seulement `RED` et `YELLOW`

Dans `C4Game`, on trouve aussi `board`, une matrice de `Cell`, qui représente la grille de jeu. `board` est rempli par la méthode `resetBoard()`, qui peuple chaque case du tableau par une `Cell` vierge (`color` nulle, `neighbor` tous nuls), avant de parcourir cette matrice une seconde fois pour désigner qui est voisin de qui, en 4-connexité ; cela, en traitant spécialement les cases en bordure de la grille.

Ses autres champs incluent :

- `topFreeCells`, un tableau de taille `TTL_COLS` qui contient l'index du rang de la prochaine case vide pour une colonne donnée
- `players`, un tableau de `C4Players`
- `tokensLeft`, le nombre de jetons restant ; c'est à dire le nombre de cases encore libres.
  - n'est pas vraiment exploité, mais pourrait rentrer en jeu dans la récursion d'un algorithme d'évaluation de la grille par un bot (arbre de décisions).
- `over`, un booléen mis à jour à chaque tour en fonction du statut du match : en cours, gagné, ou match nul.

Affichage

La méthode `displayBoard()` de `C4Game` parcourt la matrice de case en case, et examine la couleur associée à chaque case. Si la couleur est nulle, aucun jeton n'a été inséré, et elle affiche un « 0 » sans couleur spécifique. Dans les cas où un jeton de couleur rouge ou jaune est présent, elle affiche un caractère prédéfini unique à cette couleur, et dans ladite couleur, grâce aux codes d'échappement ANSI (ex : `\u001B[31m + @ + \u001B[0m` --> @).

Vérification

La méthode `C4Game.check(Cell, Color)` calcule le nombre maximal de jetons de même couleur non nulle alignés résultants du dernier jeton placé. Pour ce faire, elle appelle la méthode `aligned(Cell, Color)`, et ensuite la méthode `win()` si le résultat est supérieur ou égal à la condition de victoire (par défaut, 4)

`aligned()` examine la grille dans quatre directions, en faisant d'abord appel à des méthodes [direction]`Origin(Cell, Color)`. Par exemple, `leftOrigin()`, trouve la case la plus à gauche, de même couleur non nulle que le jeton en question, par récursion grâce au réseau de voisins. `aligned()` fait ensuite appel, sur cette nouvelle case « origine », à `alignedStraight(Cell, Color)` ou `alignedDiag(Cell, Color)`, selon le cas. Ces dernières parcourent la grille recursivement et renvoient le nombre de jetons alignés. Ces résultats (en *int*) sont compilés dans un tableau à quatre cases, et `aligned()` renvoie le maximum à la méthode appelante (ici, `check()`).

On a tenu à découpler l'évaluation de la victoire et le comptage des jetons alignés, en vue de pouvoir utiliser cette fonctionnalité de comptage en réalisant notre bot.

Programme principal

Notre programme principal accepte en premier temps les commandes 'n', 'l' et 'q' pour nouvelle partie, charger une partie, ou quitter le jeu.

Tant que l'attribut `isOver` de notre instance de `C4Game` est à `false`, la partie se poursuit ainsi :

- la grille est affichée
- le joueur actuel est invité à choisir un coup
  - si le coup correspond à un signal prédéfini, voir Sauvegarde et restauration
  - si le coup est un entier positif, il est donné à `C4Game.registerMove(int)` (qui met à jour la grille, `topFreeCells`, et appelle `check()`, mais ces détails sont cachés au `Main`).
- ensuite
  - si la partie est finie (`isOver == true`), on propose à l'utilisateur de rejouer, de recharger, ou de quitter.
  - sinon, on change le joueur actuel et on réitère

Sauvegarde et restauration

Cette partie du code repose sur l'interface `Serializable` : on l'a implémentée avec `Cell`, `C4Player`, et `C4Game`. On peut alors utiliser `ObjectOutputSteam(this)` dans la méthode `C4Game.save(Path)` pour sauvegarder l'objet dans son état actuel, et la fonction `load(Path)` est juste aussi simple. Cette dernière est invoquée dans le nouveau constructeur `C4Game(Path)`, où on peuple ensuite les champs de notre nouvelle instance avec les données de l'objet chargée.

Un effet secondaire regrettable de cette démarche est que les getters sur des objets ont dû être simplifiés pour résoudre des bugs survenant dans les jeux chargés. Spécifiquement, ces getters renvoient l'adresse de ces objets, violant ainsi l'encapsulation du projet en les rendant modifiables. Toute tentative de leur faire renvoyer des copies a échoué, et, si ce projet devait se poursuivre, on tenterait d'implémenter `save()` et `load()` différemment, pour pouvoir à nouveau utiliser des copies profondes dans nos getters sensibles.

Bonus

À l'initialisation d'une partie, l'utilisateur est invité à créer deux joueurs. Les joueurs nommés « bot » seront des joueurs contrôlés par la machine, de la sous-classe `C4Player_CPU`.

Celle-ci redéfinit sa méthode `chooseMove()` pour évaluer le score potentiel associé à chacune des (sept par défaut) colonnes. Si le bot identifie un coup gagnant, l'évaluation se termine et le bot joue ce coup. Sinon, s'il identifie un coup gagnant pour l'adversaire, il le bloque. Sinon, il choisit la colonne au plus haut score (ce score additionne simplement le score pour le bot et le score potentiel pour l'adversaire – on peut modifier cette logique pour modifier son attitude : défensif, agressif, etc.). Si toutes les colonnes ont le même score, il en choisit une au hasard. Avec un peu plus de temps, on pourrait probablement implémenter une méthode d'évaluation qui couvre le reste de la partie, plutôt que l'état actuel de la grille.