Mastermind - Projet POO

Projet et rédaction réalisés par Léo Peyronnet

Le répertoire source du projet (`./src`) est composé de plusieurs packages remplissant chacun des fonctions et/ou décrivant des comportement du programme.

Le programme est lançable depuis la méthode `main` de la classe `Main` situé à la racine du répertoire source.

Nous allons au travers de cette rédaction exposer les différents packages qui composent le programme en expliquant leur fonctionnement, les choix d'implémentations qui ont été faits et les potentielles difficultés que nous avons rencontrées.

Package `entities`

Ce package regroupe l'ensemble des objets élémentaires du programme. Il est composé de trois classes : `Couleur`, `Pion` et `Result`.

`Couleur` et `Result` sont des **énumérations**, elles décrivent respectivement les couleurs admises par le programme et les différents types de réponses qui seront donné au joueur pour l'aiguiller dans sa partie.

`Pion` est à considérer comme une `Couleur` qui peut être affiché et comparé grâce à ses redéfinitions des méthodes `toString` et `equals`. Cette classe est essentielle car utilisée dans la construction de bon nombre d'autres classes du programme.

Package `listes`

Beaucoup d'élements du jeu "Mastermind" peuvent être représentés par des listes.

- L'essai d'un joueur (combinaison) peut être représenté par une liste de `Pion`.
- La combinaison à deviner peut également être représenté par une liste de `Pion`.
- Le résultat de l'essai d'un joueur peut être représenté par une liste de `Result`.

Ce package défini dans un premier temps une classe `CapedList` qui est une encapsulation et une simplification d'une `ArrayList`. Elle permet également de définir une capacité maximum, d'où son nom. `CapedList` est ensuite étendu par `EntityList` qui fourni une implémentation commune de la méthode `toString`.

Enfin, `Combinaison`, `CombinaisonSecrete` et `TentativeResult` étendent à leur tour `EntityList`. `Combinaison` est simplement une `EntityList<Pion>` et `TentativeResult` une `EntityList<Result>` avec des méthodes supplémentaires modulant la représentation de la liste (graphique ou textuelle).

Classe `CombinaisonSecrete`

`CombinaisonSecrete`, en revanche, est une `EntityList<Pion>` ayant un **rôle central** dans le programme.

La méthode `compare` permet de faire un essai. Elle compare la combinaison passée en argument avec la combinaison secrète, met à jour les pions de la combinaison secrète qui ont été découverts et retourne une `TentativeResult` qui synthétise le résultat de l'essai. Des explications plus détaillées sur le fonctionnement de la méthode peuvent être trouvés dans les commentaires du code source.

La méthode `decouverte` permet de savoir si la combinaison secrète est découverte (si le dernier essai correspondait à la combinaison secrète) et donc de savoir si la partie est gagnée (condition de victoire).

Package `rules`

L'utilisateur doit pouvoir choisir les règles du jeu avant de lancer une partie (nombre d'essais, taille des combinaisons, ...). Nous avons choisi de définir chacune de ces règles comme des objets, afin qu'elles gèrent leur état de manière **autonome**. Ainsi, chaque règle possède :

- un message de requête (à afficher à l'utilisateur pour qu'il entre la valeur)
- une contrainte d'intégrité qui permet de savoir si la valeur rentrée par l'utilisateur est valide.
- un message d'erreur dans le cas où la valeur est invalide.

Architecture

Une règle peut avoir comme valeur une entier ou un booléen, nous avons choisi une architecture comme suit :

- Interface (`Rule`) : interface commune à toutes les rules.
- Classe abstraite (`RuleBoolean` & `RuleInteger`) : implémentation commune à un type concret (boolean, int)
- Classe instanciable : implémentation spécifique à une rule.

`MapRule`

Afin de transporter (passage en argument par exemple) plus facilement les différentes règles d'une partie, nous avons défini `MapRule`, qui est est une encapsulation/simplification de `HashMap` permettant de facilement accéder ou définir une valeur de règle.

Package `jeu`

Ce package regroupe l'ensemble des structures de jeu.

- La classe `Plateau` regroupe une `CombinaisonSecrete` et une liste de `Combinaison`. (liste d'essais)
- L'interface `Partie` permet de définir les méthodes `doTour` et `launchPartie` proposant deux manière de dérouler une partie. `doTour` est manuelle, elle permet seulement de faire un tour de jeu, alors que `launchPartie` est autonome. Dans le programme principal, l'usage de `launchPartie` a été favorisé. Cependant, sous le capot, `launchPartie` fait bien appel à `doTour`.

  `Partie` est implémentée par `PartieSolo` et `PartieMulti`.

  - `PartieSolo` fait appel à `Tableau` pour gérer son jeu. Elle se termine lorsque le `Tableau` est plein (condition de défaite) ou si la combinaison secrète est découverte (condition de victoire).
  - `PartieMulti` fait appel via une classe interne à une liste de `PartieSolo`. Elle se termine lorsque le nombre de parties désirées par les utilisateurs ont été jouées.

  Ces deux implémentations de `Partie` font appel à `MapRule` pour définir le jeu selon les volontés de (ou des) utilisateur(s).

Package `gui`

Ce package regroupe l'ensemble des affichages et demandes d'inputs que le programme contractera.

Architecture

Dans la prévision de l'apport d'une réelle GUI (comme celle demandée dans les questions bonus du sujet), nous avons défini une interface `GUI` qui défini toutes les méthodes auxquelles les autres classes du programme font appel. `GUI` est implémentée par `TerminalGUI` qui correspond à une interface graphique qui prends place dans le terminal.

Le programme principal (`Main.main()`) demande à l'utilisateur quelle type d'interface graphique il souhaite grâce à une méthode **statique** définie dans `GUI`.

Limites / Problèmes

L'emploi de cette architecture qui fait déléguer tous les affichages des objets à une seule classe permet un **polymorphisme** des différents types concrets de gui. Cependant, la programmation de cette classe "fourre-tout" à pu être éprouvant et ressemblant à certains égards à de la programmation impérative (passage par argument, les affichages d'objets différents se retrouvent dans une seule et même classe, ...).

❓ Aurait-il eu de meilleurs manières d'obtenir ce polymorphisme ?

Package `io.save`

Ce package regroupe l'ensemble des méthodes permettant à une partie d'être sauvegardé dans les fichiers du programme.

L'utilisateur, lors du choix de la combinaison pour le prochain essai, peut également choisir de sauvegarder sa partie pour la reprendre plus tard. Cette information, traitée par la GUI, doit alors remonter la pile d'appel pour influer l'état de la partie (sauvegarde puis quitter partie).

- `SaveSignal` étends `Throwable`. Il correspond à cette information remontant la pile d'appel.
- Une fois `SaveSignal` arrivé au niveau de l'implémentation de la partie (`PartieSolo` ou `PartieMulti`), elle crée une instance de `Save` qui s'occupera de créer un répertoire de sauvegarde contenant toutes les informations de la partie.

Architecture du répertoire

Le répertoire d'une sauvegarde respecte cette organisation :

```
.
└── saves/
    └── <jour-mois-annee-heures-minutes (index)>/
        ├── savedata.txt
        ├── rules.csv
        ├── 1/
        │   ├── plateau.csv
        │   └── couleurAutorisees.csv
        ├── 2/
        │   ├── plateau.csv
        │   └── couleurAutorisees.csv
        ├── 3/
        │   ├── plateau.csv
        │   └── couleurAutorisees.csv
        └── ...
```

où :

- `saves/` le répertoire parents de toutes les sauvegardes
- `<jour-mois-annee-heures-minutes (index)>/` le répertoire d'une sauvegarde avec `<jour-mois-annee-heures-minutes (index)>` la syntaxe de nommage du répertoire
- `savedata.txt` le fichier de sauvegarde principal, informations générales (type de partie, index du tour en cours ...)
- `rules.csv` : transcription de la `MapRule` de la partie.
- `1/`, `2/`, `3/`, ... : les répertoires de parties. Si PartieSolo, il n'y aura que `1/`.
  - `plateau.csv` la transcription du plateau de la partie.
  - `couleurAutorisees.csv` la liste des couleurs utilisés pour cette partie.

Limites / Problèmes

- L'usage du JSON aurait surement l'emploi d'une API allégeant la méthode `doSave`, au lieu de la création d'une structure de fichiers lourde.
- De grosses difficultés dans la conception de la classe `Save`. Même problème que pour le package `gui` (programmation impérative, classe "fourre-tout").

❓ Aurait-il été plus judicieux de laisser les parties se sauvegarder eux-mêmes ?

🚧 Le package `Save` n'a pas pu être fini. Dans l'état, si l'utilisateur souhaite sauvegarder, les fichiers se créeront et seront remplis mais il sera impossible de faire le chemin inverse car la méthode `load` n'a pas pu être implémentée.