Projet Java 2020

MEMORY – Sami Khalifa

MEMORY – Sami Khalifa

Introduction

Consignes

Il nous a été confié de réaliser un projet JAVA permettant la réalisation d'une Application JAVA FX en suivant certaines contraintes. L'objectif est de mettre en pratique quelques pratiques JAVA comme la POO, Design Patterns, Types Génériques, expressions lambda, Interface Front end, Algorithmiques spécifiques etc.

Présentation du projet

Le Thème fut un Jeu de Réflexion avec des cases. De ce principe, mon choix de jeu fut contrasté, ayant d'abord essayé avec un jeu de Dame j'ai eu énormément de soucis avec l'IA adverse car le Machine Learning ne fais pas partie de mes compétences, donc après quelques heures passés dessus, j'ai finalement opté pour le jeu Memory, le concept est simple, une grille avec des couples de cases portant des numérotations, Au début de partie pendant un certain temps(qui dépends de la difficulté) toutes les cases sont dévoilés, puis elles se couvrent et le but du jeu est de mémoriser les couples de cases à dévoiler puis de les dévoiler petit à petit, le jeu prends fin lorsque toutes les cases sont dévoilés.

Structure du projet

Pour ce Projet, j'ai été amené à utiliser l'anglais et j'ai implémenté divers Design Patterns tels que MVC, Factory et Observer, mais également une Classe générique Position qui sera propre à chaque case.

Commentaire

Le jeu Démarre avec un joueur par défaut, donc aucune configuration n'est obligatoire, cependant elle reste recommandée, par défaut, le jeu démarre en Mode Facile et le joueur possède comme nom : New_Player.

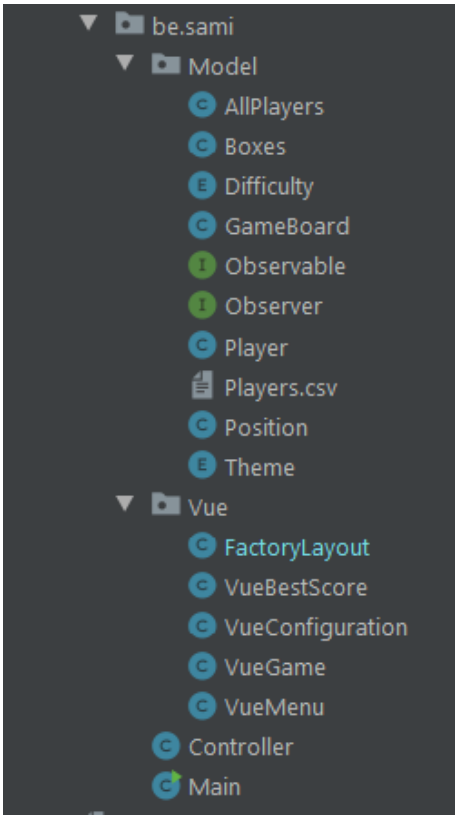
Objectif du Jeu

Le jeu se compose de paires de cartes portant des illustrations identiques. L'ensemble des cartes est mélangé, puis étalé face contre table. À son tour, chaque joueur retourne deux cartes de son choix. S'il découvre deux cartes identiques, il les ramasse et les conserve, ce qui lui permet de rejouer. Si les cartes ne sont pas identiques, il les retourne faces cachées à leur emplacement de départ.

Le jeu se termine quand toutes les paires de cartes ont été découvertes et ramassées.

1. Structure de l'Application

- *Package* be.sami : Package principal de l'Application.
- Package Model : Partie Modèle de mon application, elle comporte mes classes :



  - AllPlayers : est la Liste de tous mes joueurs, elle comporte les Méthodes Lecture et Ecriture Fichier.
  - Boxes : c'est la classe Cases, elle comporte un Button, un attribut pour valider (ne plus re cliquer dessus) ainsi qu'un attribut position.
  - Diffiulty (ENUM) : énumération qui recense le niveau de difficulté choisit par le joueur, elle permet de fixer la taille de la grille ainsi que le temps de visibilité des cases avant que le jeu commence, par ex : en mode facile, la grille est de 4x4 et le temps avant que la partie commence est de 10 sec, en normal 6x6 et le temps de 15 sec et en difficile la taille est de 8x8 et le temps de 20 sec.
  - GameBoard : Plateau de jeu, qui a comme attribut statique le premier click, la couleur à lui attribuer au click, ainsi que comme attributs non statiques la taille de la grille qui varie selon la difficulté ainsi que la liste de buttons et la liste des Observers.
  - Observable et Observer, 2 petites interfaces permettant de mettre à jour la grille en fonction des clicks de l'utilisateurs.
  - Player : Classe joueur qui a comme Attribut le nom, le niveau de difficulté, le score, le temps ainsi que si le joueur à utiliser le cheat code implémenté.
  - Player.csv : fichier CSV comprenant la liste de tous les joueurs.
  - Position : Classe comprenant une abscisse et une ordonnée qui correspondent respectivement à la position du Button dans la grille.
  - Theme : Enumération qui définit le thème de l'application, qui est par défaut en Sombre (Dark Mode). J'ai délibérément choisi de la définir moi-même au lieu d'utiliser un Color Picker car j'estimais qu'elle relevait un plus grand défi.

- Package Vue : Ce package est très simple, il contient une classe par vue ainsi que la classe Factory, qui me sert principalement à crée des Buttons, ainsi que des Hbox et Vbox plus simplement.

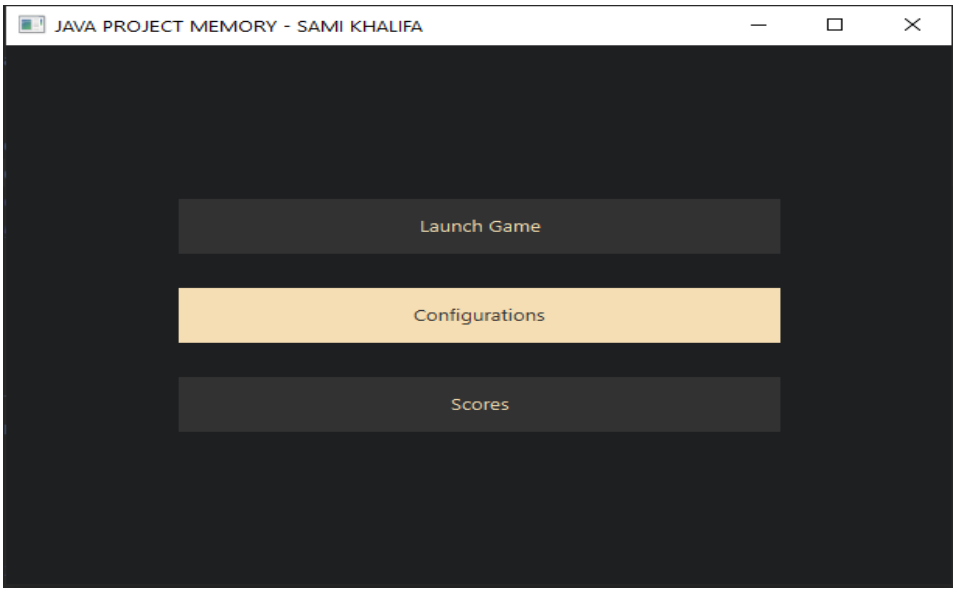
- Controller : Classe Controller qui fera toutes les actions entre la vue et le Modèle, principalement la récupération des actions réalisés par des buttons.

- Main : Point d'entrée principal de l'Application, qui Instancie le Contrôleur et affiche les méthodes grâce à la méthode showAll ().

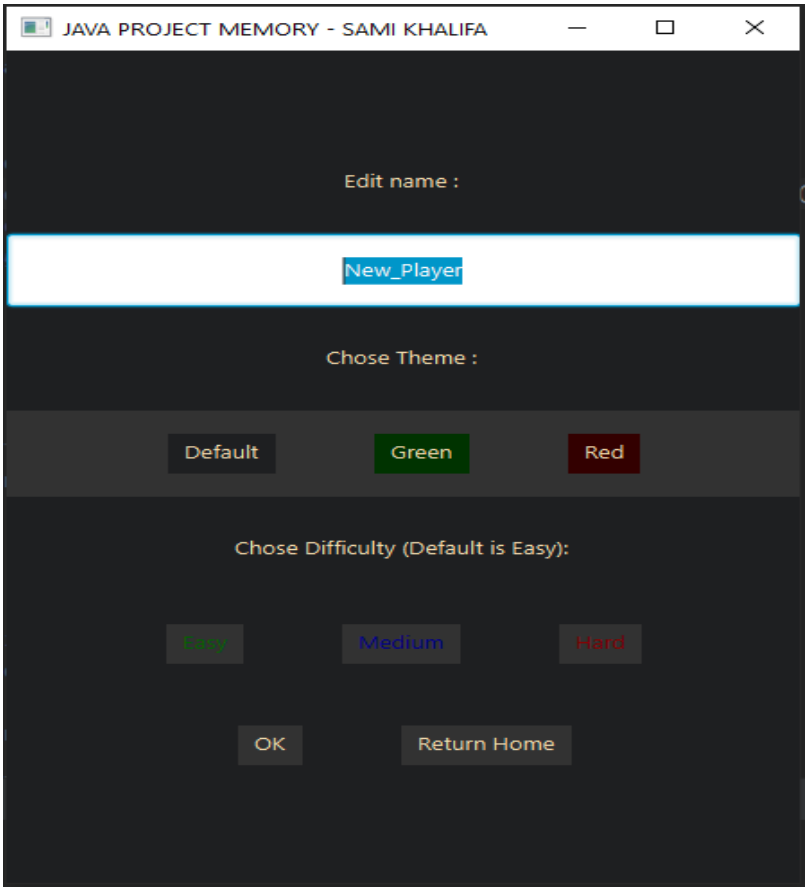
2. Fonctionnalités

Après le survol de la structure de l'application, je vais énoncer les différentes fonctionnalités de l'application.

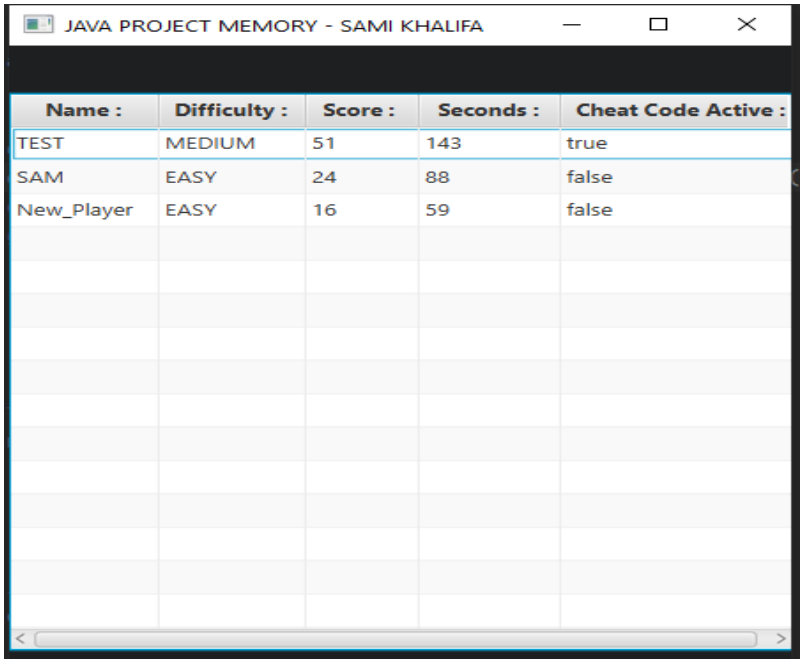
L'utilisateur rentre dans l'application, une fois dedans, il a 3 possibilités :



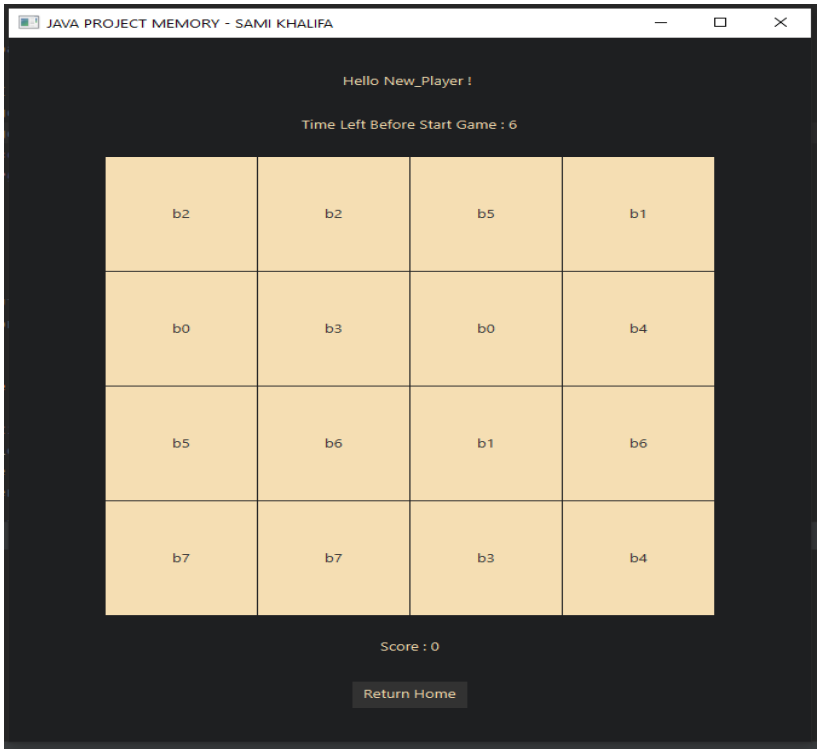
- Il peut configurer sa partie, auquel cas il pourra se renommer, changer la difficulté et le Thème de l'application, les modifications sont apportées instantanément et sont validés dès que l'utilisateur confirme l'action.



- Il peut également consulter les Scores via le Button Best Score

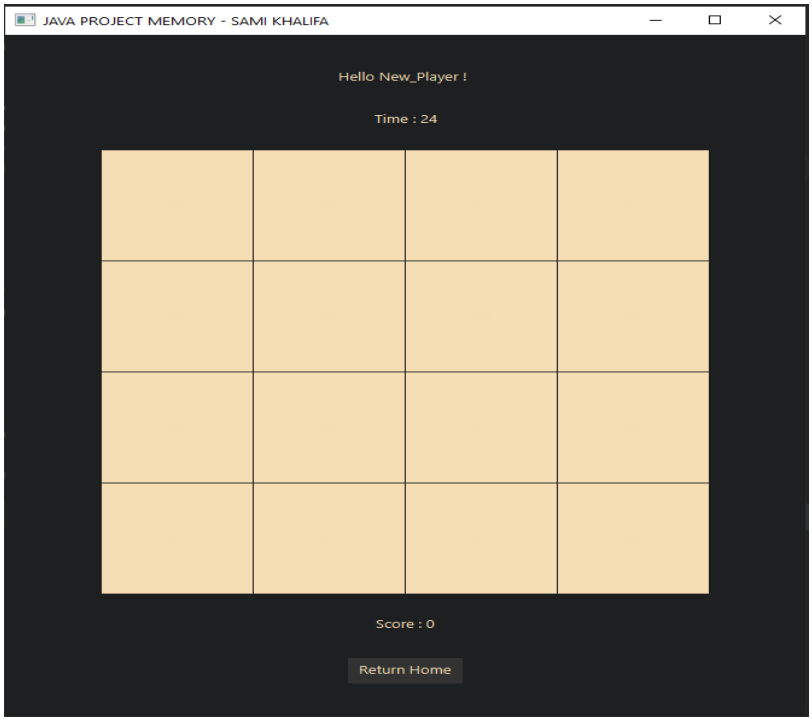


- Lancer directement le jeu, son nom sera NEW_PLAYER et son thème sera Sombre.
  N.B : Si la liste de joueur est vide, la grille affichera « No rows to display ».

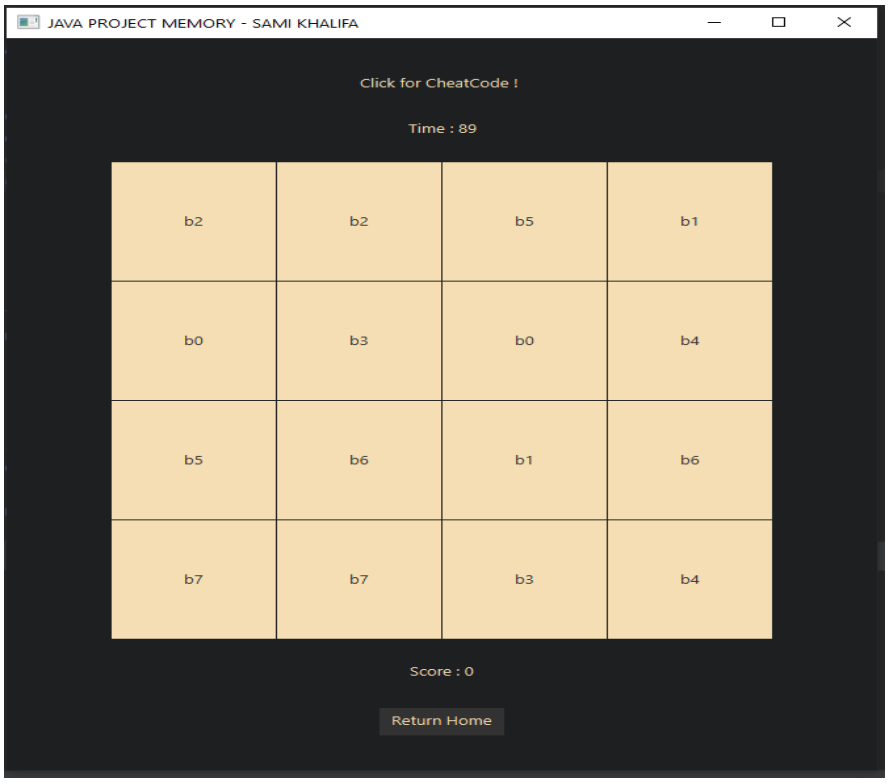


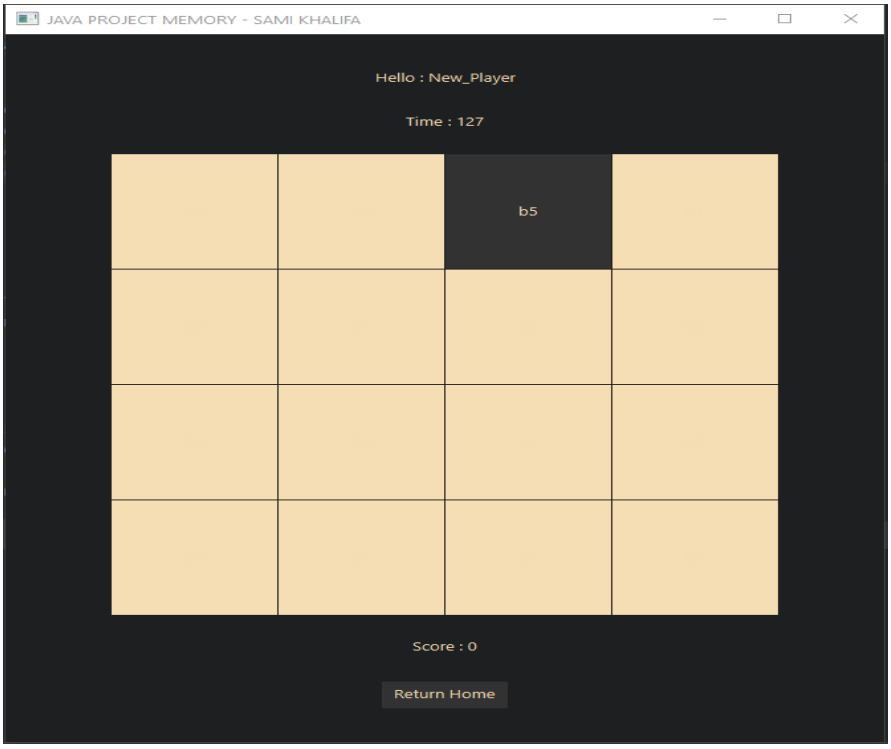
Lors du lancement de la partie :

- La grille qui comporte tous les buttons du jeu.
- Un Timer qui, au terme d'un laps de temps défini (selon la difficulté, dans l'exemple ci-dessous 10 secondes) bloquera la partie et affichera les cases puis les camouflera au terme de celui-ci, le temps ainsi que le score servent de repère pour identifier les meilleurs joueurs.



- Un label comprenant le nom du joueur, qui lorsqu'on le survole, active un cheat code qui change le label en « Click for Cheatcode » (sur l'image ci-dessous la souris survole le label) et affiche temporairement les cases au survol, et lorsqu'on click sur ce label, permet désormais au curseur d'afficher le numéro des cases qu'il survol, il faut savoir cependant que l'utilisation de cette fonctionnalité sera inscrite dans la classe Joueur et sera visible par l'ensemble des joueurs.



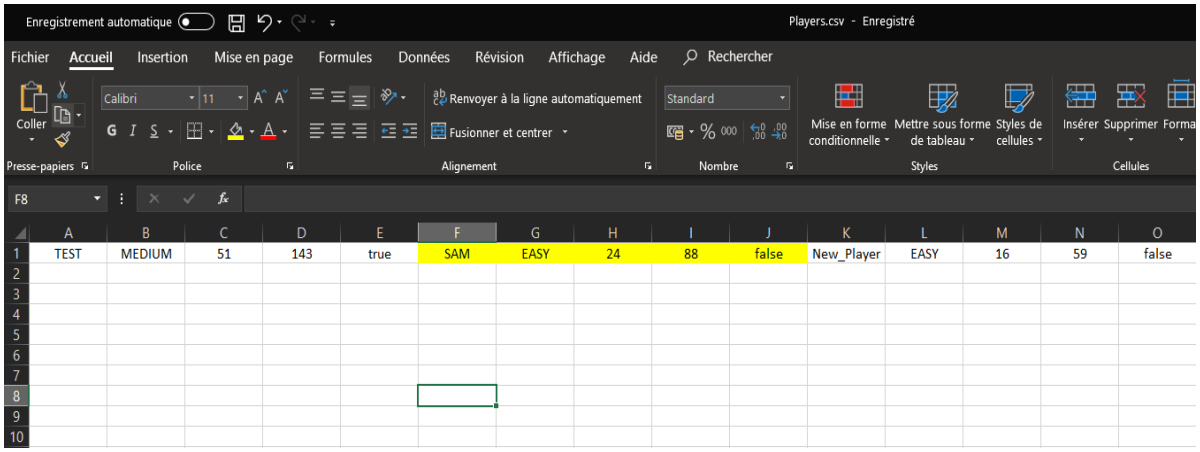


- Un score un peu spécial, qui décrémente de 1 en cas d'échec de correspondance des buttons et incrémente du (total de la grille divisée par 2), ce choix fut opté pour équilibrer la difficulté. Par Ex : Dans une grille de 6x6 (donc de longueur 6), une correspondance entrainera une incrémentation de 6/2 donc 3 car plus difficile à trouver que dans une grille de longueur 4 par exemple.

- Un Button ReturnHome qui, comme dans chaque scene reviens au menu Principal.

3. Persistance de données

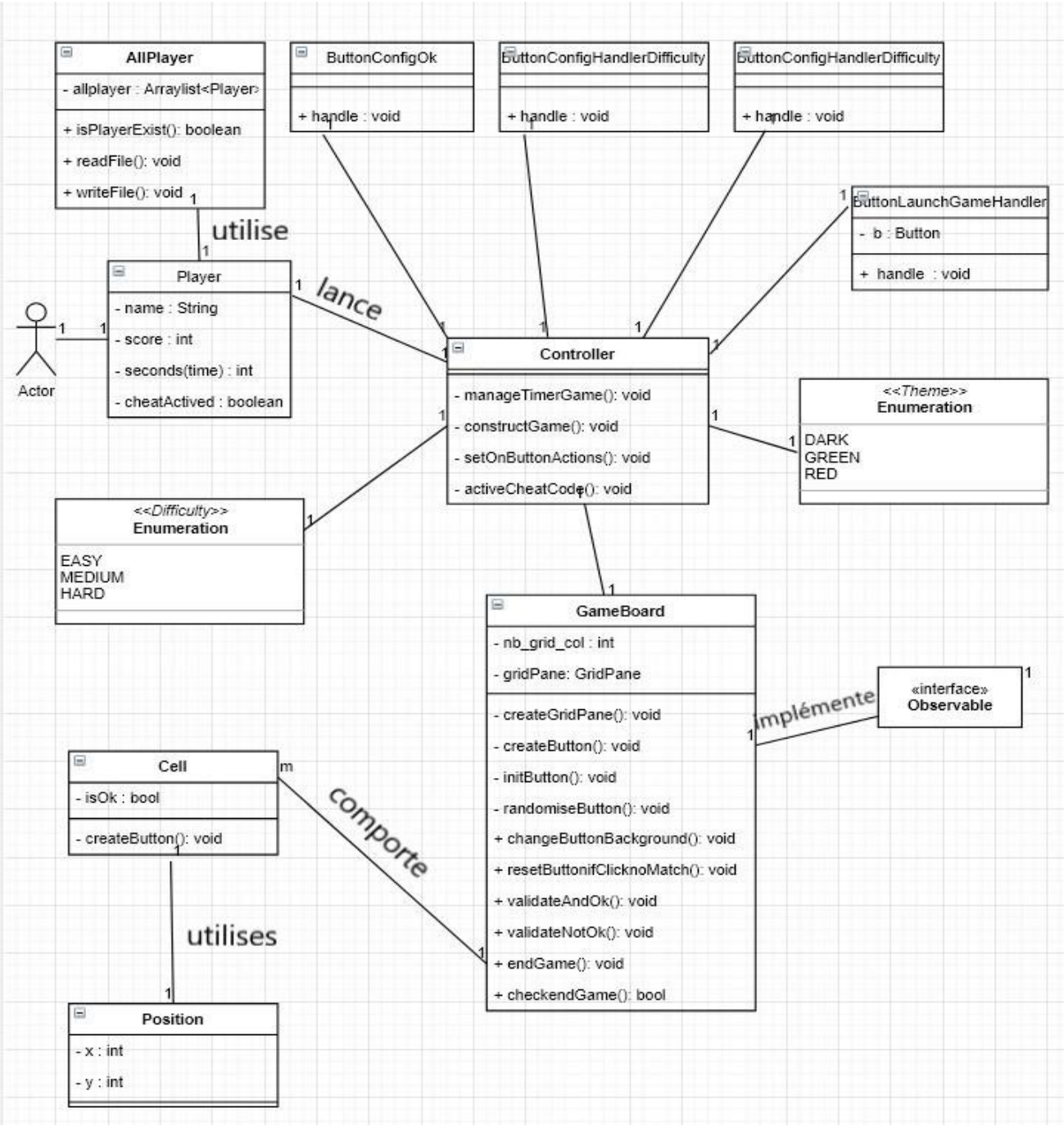
Pour la persistance des données, j'ai décidé d'opter pour un fichier CSV, facile à lire et à écrire et plus aisé à ouvrir avec Excel. J'aurai pu également utiliser un fichier texte ou LOG, qui étaient prévus. J'y inscrit les joueurs de ma classe Player répartis comme cis : nom, difficulté, score, temps (en sec) ainsi que s'il a utilisé un code de triche ou non.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	TEST	MEDIUM	51	143	true	SAM	EASY	24	88	false	New_Player	EASY	16	59	false
2															
3															
4															
5															
6															
7															
8															
9															
10															

4. Analyse

Voici le **diagramme de classe** que j'ai réalisé :



Commentaires :

- Le point d'entrée de l'Application est la classe joueur, en effet afin d'obtenir la liste de tous les utilisateurs la classe Player va être aidée par AllPlayers.
- La classe joueur va donc faire appel au Controller qui va utiliser ses sous classes handler afin de l'aider à gérer les utilisations des boutons qu'ils comportent.

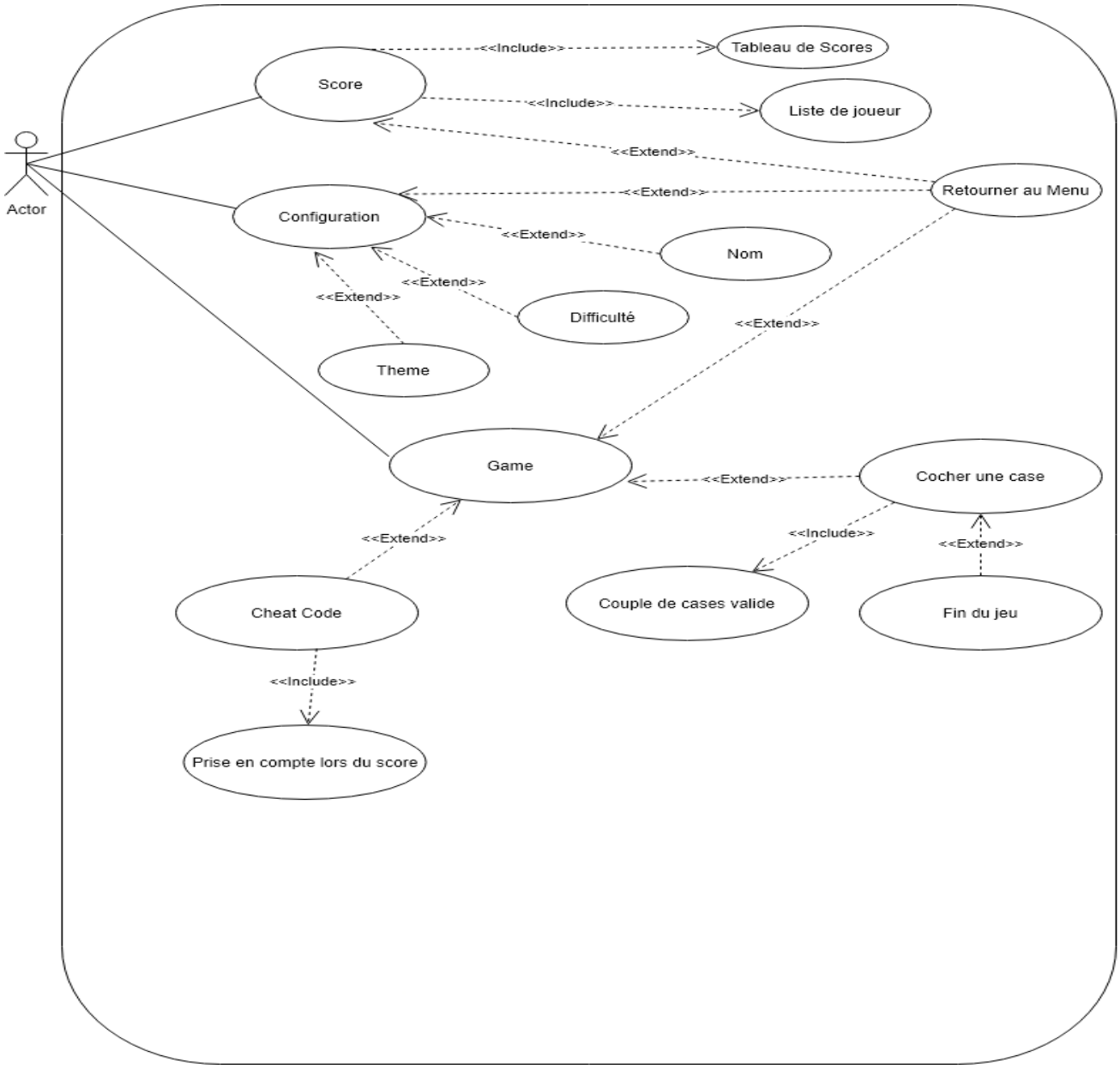
- J'ai choisi de ne pas mettre les Getters et Setters pour ne pas surcharger le diagramme.

- J'ai choisi également de ne pas mettre en Attribut les classes et les énumérations que j'ai moi-même crée, ainsi que les Scènes (pour le contrôleur).

- Mon programme implémente le design pattern Observer et Observable, ici j'ai représenté uniquement Observable car elle concerne les Model et pas Observer car elle concerne la vue.

- Les Cardinalités sont présentes même si parfois elles ne sont pas visibles.

Voici le **diagramme d'activité** que j'ai réalisé :



Commentaires :

- Tableau de Scores et Liste de joueurs représentent une même entité car le tableau s'affiche même si la liste est vide.

- Les éléments du cas « Configuration » sont en extends car rien n'oblige à l'utilisateur de valider ses modifications, car un joueur existe déjà par défaut.

- Les couples de cases sont « include » car elles dépendent du click et c'est lui qui appellera la fin du jeu, même si je l'ai mis en extends de « Cocher une case » il s'agit en réalité d'un trio qui dépendent chacun l'un de l'autre.

5. Designs Patterns

En ce qui concerne les Design Patterns, j'ai choisi d'en utiliser 3 :

- MVC ; Le plus important concernant la bonne répartition des classes, le Modèle reprend toutes les classes que j'ai créé et qui me servent de support ainsi qu'une partie de la logique, la Vue corresponds aux différentes interfaces de mon Application et le Controller gère toute la correspondance entre modèle et vue, tout en implémentant la logique du jeu. J'ai essayé de l'implémenter le mieux possible.

- Factory : cette classe me fut très utile, je l'ai nommée FactoryLayout et elle me permet de crée aisément des buttons, des labels, des Hbox ainsi que des Vbox.

- Observable & Observer : Ces interfaces, très difficiles à implémenter pour moi, car il y'a plusieurs façons de le faire, j'ai d'abord choisi d'utiliser l'interface propertyListenner, mais je n'y arrivais pas donc je me suis basé sur les Observer et Observable. Enfin m'ont en fin de compte beaucoup servi pour notifier la vue lorsqu'un couple de Button sont validés et qu'il faut changer le score en conséquence.

- (Design Pattern Avorté) Singleton : Cette classe m'aurait permis de contrôler qu'uniquement une seule instance de Player serait instanciée, toutefois j'ai trouvé qu'elle ne serait pas vraiment utile pour mon projet.

6. Conclusion

La raison pour laquelle je n'aborde pas plus mes méthodes est parce que je les ai pas mal commentés dans mon code. J'avais également un commentaire concernant la classe générique position, je l'ai facilement implémenté mais cependant c'était difficile de lui trouver une utilité dans ma logique de jeu car je n'utilise pas d'attributs de position.

Concernant la difficulté du projet, il était très difficile de faire la part des choses entre l'implémentation du design pattern, l'apprentissage de Java Fx, l'aisance entre les classes utilisées etc. cependant les compétences acquises m'ont permis d'être plus a l'aise avec java et voir petit a petit la portée immense de ce langage de programmation.