2020-2021

Projet de Programmation Orientée Objet - POO
« Pet Rescue Saga»

Projet réalisé par le Groupe 22:
L2 Mathématique/Informatique
Tara AGGOUN – 21961069 - taraggoun@gmail.com
Elody TANG – 21953199 – elodytang@hotmail.fr

1. DESCRIPTION DU PROJET

A. *Présentation du projet*

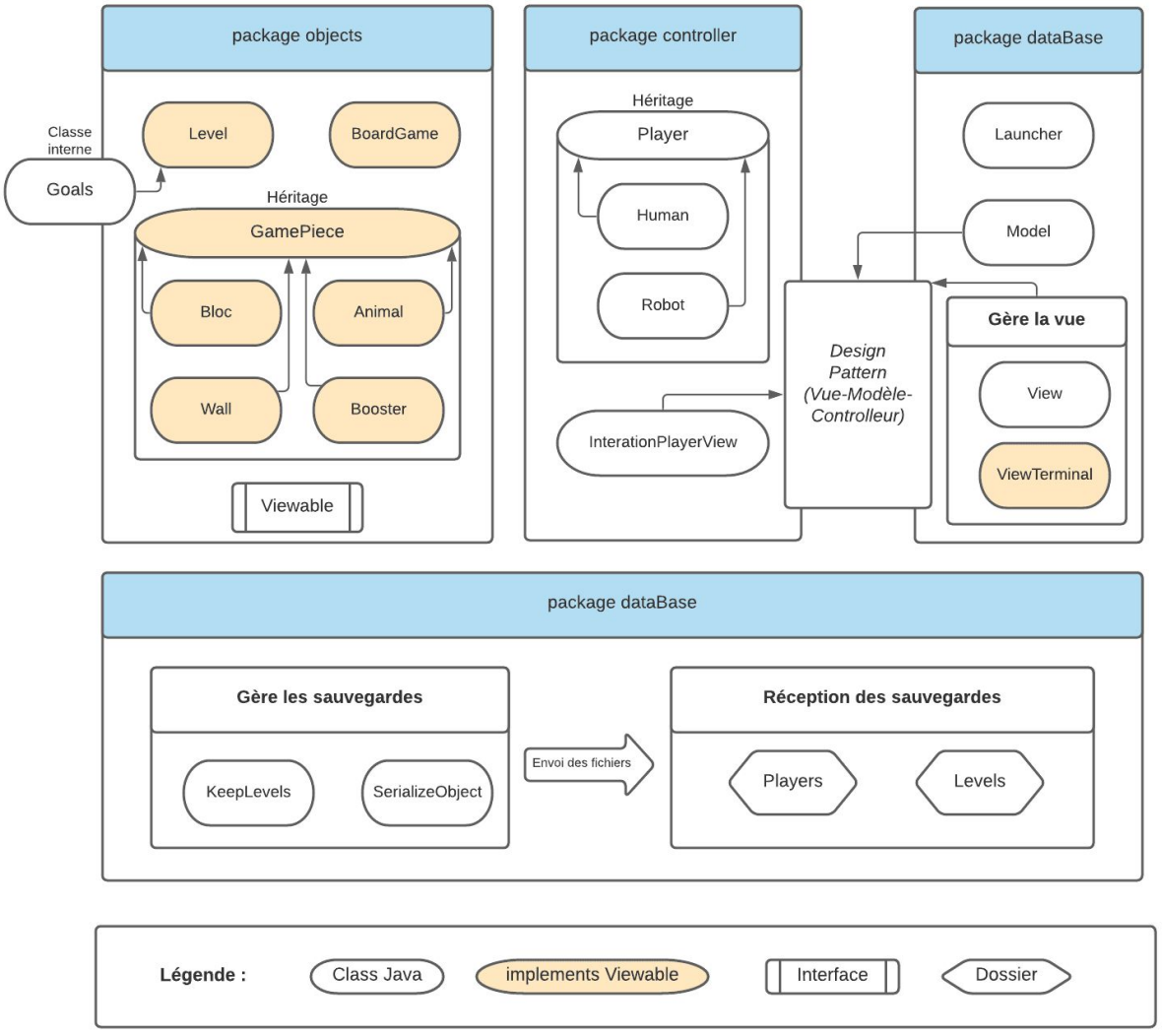
Notre projet porte sur la réalisation d'un jeu basé sur le jeu "**Pet Rescue Saga**".

➔ Règles du jeu :

- **Pop Cubes** est un jeu interactif et intuitif : vous êtes le héros de cette histoire et vous devez sauver les animaux en détruisant des groupes de couleurs supérieur à 2. Dans chaque niveau, il y a un nombre d'animaux à sauver, et un nombre de points à obtenir. En tant que **héros**, vous devez repérer les groupes et les faire exploser, afin de ramener au sol le nombre d'animaux à sauver et récupérer le plus de points possibles. Une fois les objectifs atteints, vous pourrez continuer votre aventure.

Voici la construction détaillée de toutes les classes Java et des dossiers qu'on a utilisés pour réaliser ce projet.

B. *Schéma des class Java et des dossiers présents*



C. *Description des packages*

a. *Le package objects*

Au début de l'implémentation, on a commencé à faire toutes les classes du **package "objects"** (exceptée la classe "**Booster**"), plus généralement les classes Java : "**Bloc**", "**Animal**", "**Wall**", "**GamePiece**" et "**Level**". Elle représente tous les objets différents qui sont liés à un niveau du jeu.

➔ Les classes "**Animal"**, "**Bloc**", "**Wall**", "**Booster**" et "**GamePiece**"

- Un objet typé "**Bloc**" représente un bloc de couleur à enlever. Il y a un attribut **idColor** (int) pour chaque objet créé. Une couleur est associée au numéro à chaque fois qu'un joueur ouvre un niveau, ainsi, la couleur du Bloc change à chaque fois qu'un niveau a été ouvert.
- Un objet typé "**Animal**" représente un animal à libérer. Dans nos fichiers, il y a plusieurs images d'animaux alors on a choisi de mettre des attributs **i** (int) et **imageAnimal** (Map<Integer, String>) pour faire en sorte qu'à chaque niveaux, les animaux changent et soient aléatoires. À chaque fois qu'un animal est créé, **i** est initialisé entre 1 et 7 de manière aléatoire et une image est liée à chaque **i**. Ainsi, l'image associée à cet objet est différente pour chaque animal créé.
- Un objet typé "**Wall**" représente un Mur : c'est un **"Bloc**" indestructible, qui ne peut pas être supprimé ou descendre.
- Un objet typé "**Booster**" représente un bonus. Si le joueur explose un booster, l'évènement est différent. Il y a 4 types de bonus différents :

1. Fusée (*rocket*) : enlève les blocs d'une seule colonne.
2. Bombe (*bomb*) : enlève les blocs adjacents d'un bloc.
3. Boule de Disco (*disco-ball*) : enlève tous les blocs d'une seule couleur.
4. Marteau (*hammer*) : enlève un seul bloc.

- Un objet typé "**GamePiece**" ne peut pas être initialisé : cette classe est abstraite. Elle représente une pièce qu'on peut mettre dans le tableau d'un niveau. Toutes les classes citées ci-dessus hérite de cette classe.

➔ Les classes "**Level**" et "**BoardGame**"

- Un objet typé "**Level**" représente un niveau du jeu. Elle possède une classe interne "**Goals**", qui définit tous les objectifs que le joueur doit atteindre pour terminer un niveau, comme le nombre d'animaux à sauver, le nombre de points à obtenir ou le nombre de mouvements limités. Elle possède aussi des attributs **lvl** (int) pour le numéro du niveau, **available** (boolean) pour savoir si le niveau est accessible, **gamePiece** (GamePiece[][]) pour avoir le tableau du niveau, **getAdja** (boolean[][]) pour compter ou repérer toutes les blocs contiguës à un autre bloc et **goals** (Goals) pour avoir ses propres objectifs. Toutes les méthodes pour enlever, décaler, remplacer des blocs sont faites dans cette classe.
- **BoardGame** représente le sommaire des niveaux.

➔ L'interface "**Viewable**"

Lorsque la vue sur le terminal a été implémentée, on a ajouté une interface "**Viewable**" qui ne possède qu'une seule méthode : `void displaysOnTheTerminal()`. Toutes les classes qui implémentent cette interface peuvent être affichées sur un terminal.

b. ***Le package controller***

➔ La classe "**InteractionPlayerView**"

- La classe **InteractionPlayerView** est le contrôleur qui gère l'interaction entre le joueur et la vue. À chaque fois qu'un événement a été fait, il passe par cette classe pour déterminer l'évènement d'après…

➔ Les classes "**Player**", "**Human**" et "**Robot**"

- La classe **Player** représente un joueur. Elle peut être de deux types différents : un humain ("**Human**") ou un robot ("**Robot**"). Ces classes héritent de **Player** et ont presque les mêmes méthodes, cependant, leurs événements sont différents. Pour l'humain, ses méthodes utilisent un **Scanner** pour que le programme puisse lire ses choix sur le terminal, alors que le robot utilise un **Random**.

	Humain	Robot
Utilisation de l'interface graphique	✓	
Utilisation du jeu sur le terminal	✓	✓
Sauvegarde des données	✓	✓
Jeu automatique		✓ (mais un humain doit diriger le robot)

c. ***Le package dataBase***

➔ Les classes "**KeepLevels**" et "**SerializeObjects**"

- La classe "**SerializeObject**" s'occupe des sauvegardes des données. C'est elle qui gère si l'utilisateur veut enregistrer un joueur, des niveaux ou si elle veut récupérer ses anciennes données. Elle peut alors reprendre sa partie en cours, réinitialiser sa partie ou supprimer sa partie. Elles envoient les sauvegardes des niveaux dans le dossier **Levels** et les sauvegardes des joueurs dans le dossier **Players**.
- La classe "**KeepLevels**" gère les sauvegardes des niveaux. Grâce à elle, on peut continuer l'enregistrement des différents niveaux.

d. ***Le package launcher***

➔ La classe "**Launcher**"

- C'est la classe principale de notre jeu : elle lance les différentes vues et affiche les messages d'erreur lorsque l'utilisateur rentre une commande incorrecte.

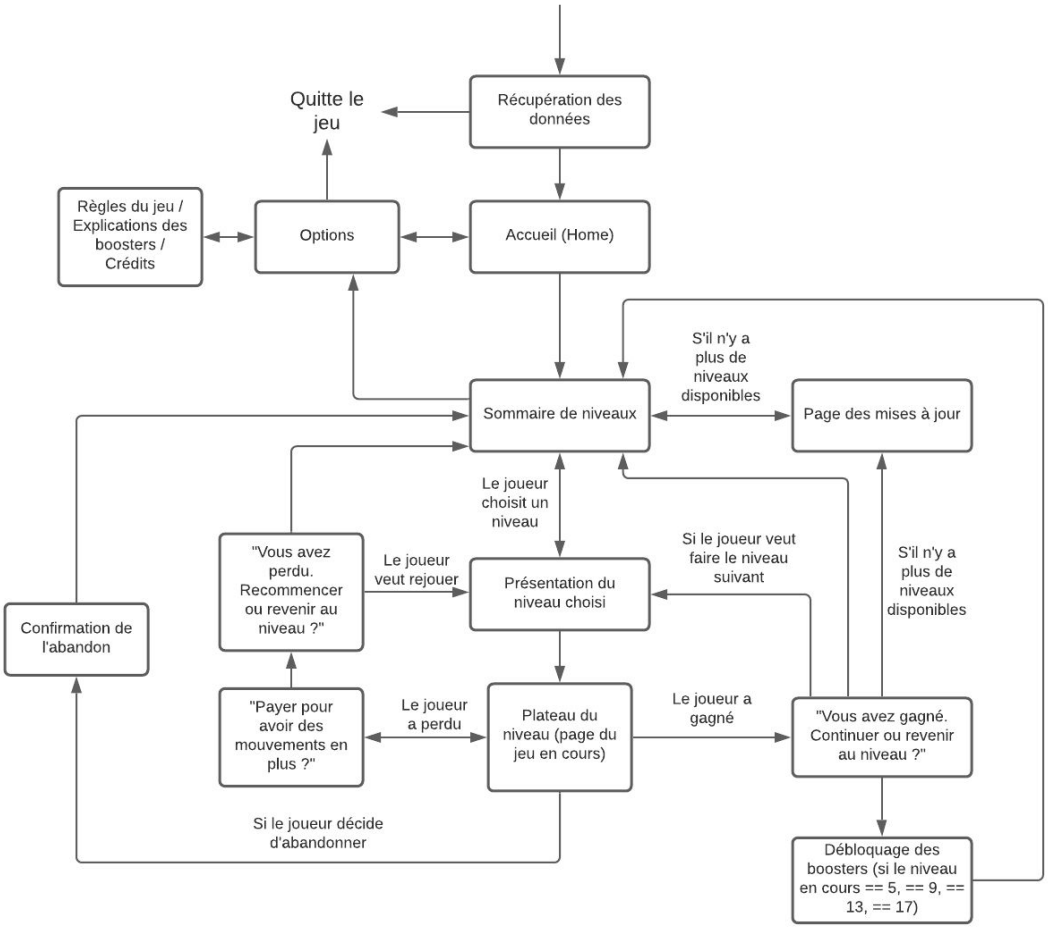
➔ La classe “**Model**”

- Cette classe possède beaucoup d’attributs : elle sert à stocker toutes les variables du jeu, comme la partie en cours, les couleurs dédiées à chaque bloc, les boosters utilisées pendant une partie (ou au début de partie), le temps d’attente pour la réinitialisation des vies, le nombre de pages totales dans le sommaire...

➔ La Vue Graphique (GUI) = la classe “**View**” qui étend de JFrame

- Elle possède 4 attributs : **ipv** (InteractionPlayerView), **model** (Model), **cardLayout** (CardLayout) pour changer facilement de page et **mainPanelBold** (JPanel) qui est le JPanel principale de la classe. Elle sert exclusivement à afficher les différentes parties du jeu sur une interface graphique.

- Les liens entre les différentes pages de notre projet sont représentées par ce schéma :



➔ La Vue Terminal = la classe **ViewTerminal**

Tout comme la classe **View**, elle possède deux attributs : **ipv** (InteractionPlayerView) et **model** (Model). Elle sert exclusivement à afficher les différentes parties du jeu sur le terminal.

➔ Les différences entre les deux Vue(s)

- Ce que la VueGraphique (**View**) peut faire comparé à la VueTerminal (**ViewTerminal**) :

1. Système de vie, système de points et système de pièces

2. Utilisation des boosters en début de partie
3. Si le joueur perd, il peut avoir 5 mouvements en plus s'il paie

- Ce que la VueTerminal (**ViewTerminal**) peut faire comparé à la VueGraphique (**View**) :

1. Utilisation du Robot

2. LISTE DES FONCTIONNALITÉS

A. Demandées et réalisées

➔ Les parties obligatoires :

- Le principe du jeu général : le joueur doit enlever des groupes de blocs pour sauver des animaux, une suite de niveaux (appelée "**Level**") avec une difficulté croissante.
- Des niveaux enregistrés que le programme peut utiliser.
- Un environnement de jeu accessible aux joueurs et qui manipule les niveaux, qui présente les règles et la progression des joueurs.
- Un plateau avec une partie visible et une partie cachée (appelée "**GamePiece[][]**").
- Un design Pattern "Vue-Modèle-Contrôleur" (appelée "**View(Terminal)-Model-InteractionPlayerView**").
- La VueTerminal est séparée de la Vue pour l'interface graphique, le joueur peut choisir ce qu'il veut utiliser en début de jeu, ou à partir des commandes déjà implémentées.
- Des sauvegardes pour que le joueur puisse continuer sa partie.

➔ Les différents rôles :

- Deux rôles différents : un humain et un robot (appelée "**Human**" et "**Robot**") qui ont deux implémentations différentes.

B. Réalisées mais pas demandées

➔ Sauvegarde des données :

- Si les données d'un joueur sont sauvegardées, le joueur peut continuer sa partie sur le terminal ou sur l'interface graphique. Les deux vues sont connectées.

➔ Sur l'interface graphique :

- Des pages supplémentaires : les crédits, les explications des boosters, les règles du jeu, la page des mises à jour, le déblocage des boosters, le verrouillage des niveaux lorsque le joueur n'a plus de vies ou de mouvements possibles pendant une partie, deux pages lorsque le joueur a gagné/a perdu le niveau.
- Les boosters peuvent être utilisés dès le début de jeu.
- Un système de nombre de vies, un système de nombre de points, un système de nombre de pièces pour les joueurs.
- Lorsque le joueur perd ses 5 vies, il doit attendre 10 minutes pour rejouer.
- Le déblocage des boosters est progressif (et peuvent se jouer en début de partie).

➔ Des options supplémentaires :

- Un éditeur de sommaire où les joueurs peuvent choisir le niveau qu'ils veulent faire à plusieurs pages.

- Les joueurs peuvent voir les crédits, les explications des boosters et les règles du jeu.
- Les couleurs des niveaux apparaissent aléatoirement.
- Le joueur peut réinitialiser/supprimer sa partie ou quitter le jeu.
- Un système de nombre de mouvements.
- Le joueur possède 4 boosters (la fusée, le marteau, la bombe et la boule de Disco) et peut les utiliser.
- Les boosters peuvent être utilisés au milieu de la partie.

➔ Une Javadoc

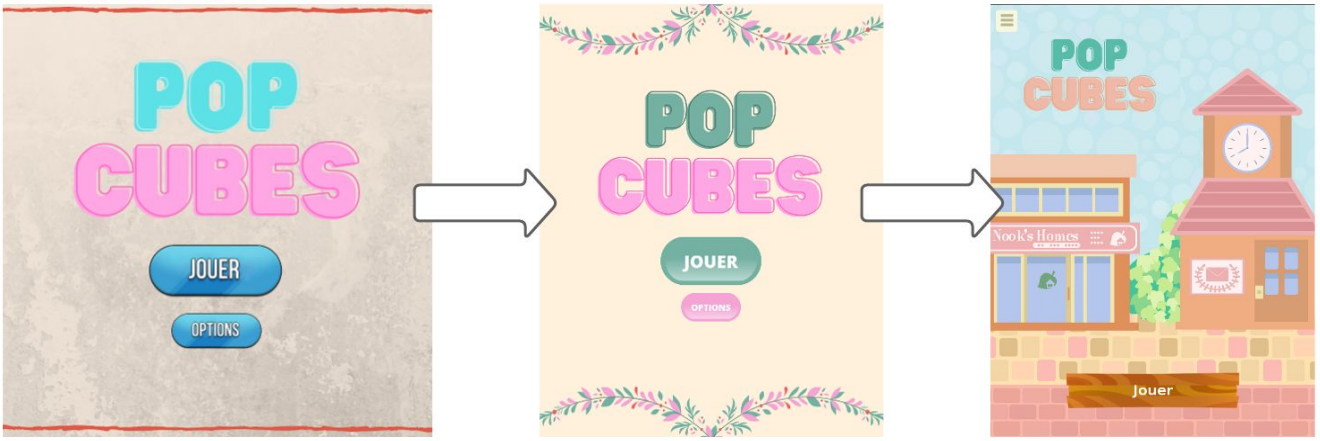
3. INFORMATION SUPPLÉMENTAIRE

A. *Problèmes rencontrés*

➔ Un début difficile

Au début de l'implémentation, le sujet était compliqué à comprendre. On a eu du mal à comprendre l'interface **Serializable** et comment l'utiliser. De plus, on avait pas très bien compris, dans l'énoncé, "Les niveaux doivent être stockés sur le disque" et ce que ça impliquait.

C'était aussi la première fois qu'on implémente une interface graphique : on a pris beaucoup de temps pour la faire et on a souvent changé d'avis.



Ensuite, on avait mal implémenté notre VueTerminal, une fois qu'on a implémenté la VueGraphique, rien n'allait. On a dû supprimer la plupart de nos classes pour les réécrire. On a aussi vu que les accents peuvent être une source d'erreur de compilation, alors on a décidé d'enlever tous les accents.

B. *Piste d'extension*

➔ Liste exhaustive des extensions qu'on pourrait ajouter :

- **Un nombre de boosters limité** : actuellement, le nombre de boosters est illimité, le joueur peut utiliser autant de boosters qu'il veut.
- **Ajouter des animations** : pour le déplacement de toutes les pièces du tableau des niveaux, le déplacement se fait en instantané : les blocs apparaissent tout de suite à leur nouvel emplacement. Il faudrait ralentir le

mouvement et faire en sorte que les joueurs puissent voir la progression des déplacements (les blocs devraient tomber un à un, puis se décaler à gauche ensuite). Évènement qu'on ne voit pas !

- **Des contraintes de niveaux** : actuellement, il n'y a que des blocs "classiques", on pourrait créer des blocs "plus résistants", où il faudrait plusieurs coups pour les détruire. Ou on pourrait créer des cadenas et il faudrait sauver une clé pour pouvoir ouvrir les cadenas.
- **Des options supplémentaires** :
    1. Annuler le coup, et revenir au coup précédent.
    2. Faire un Booster supplémentaire qui supprimer tous les blocs visibles.
    3. Faire une demande d'aide, qui renverrait une zone intéressante à jouer.