PROJET DE POOIG
L2 INFORMATIQUE

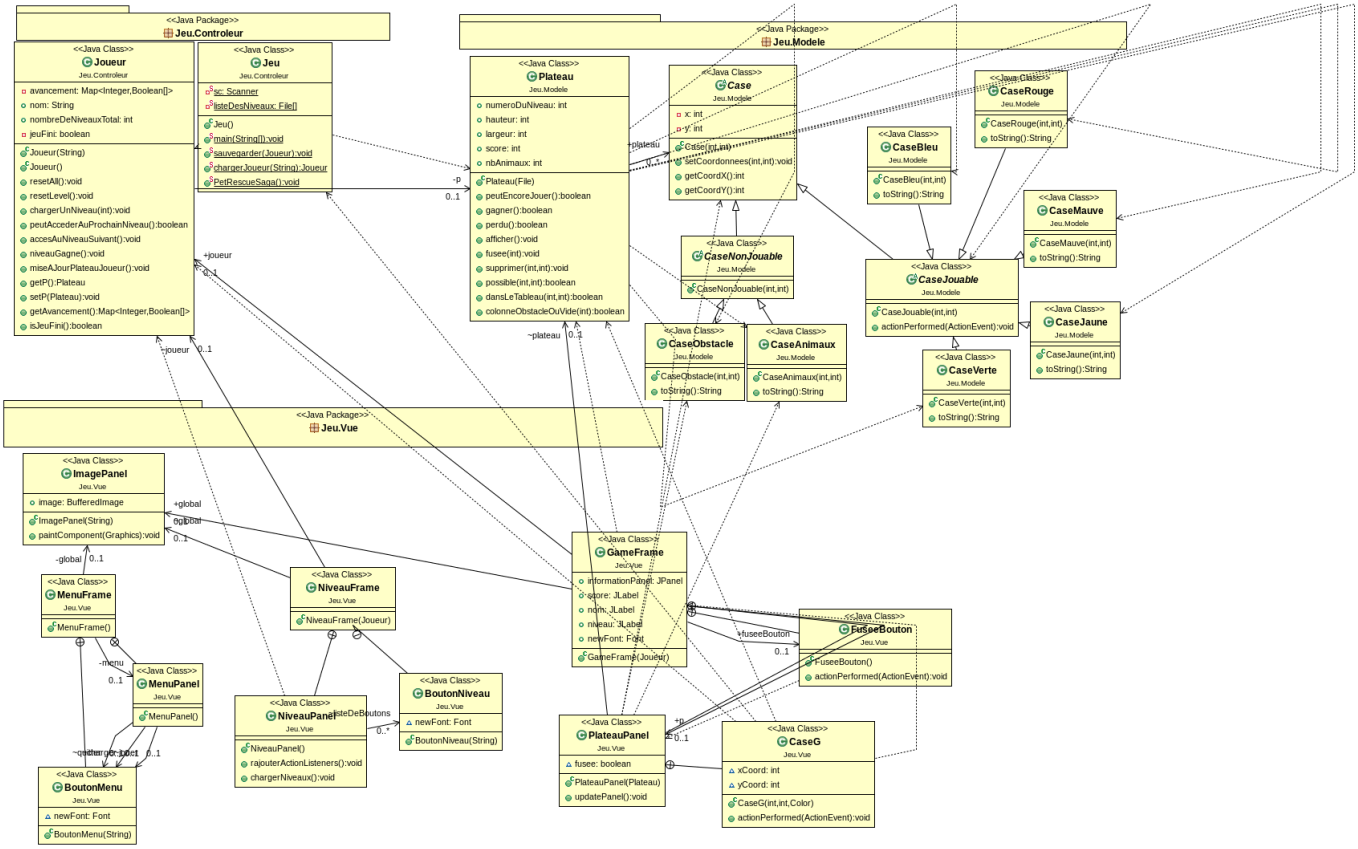
PET RESCUE SAGA

Chaima BELHOUT
Najd KACEM

Année 2020-2021

TABLE DES MATIÈRES

LE DIAGRAMME



A) MODÈLE

1) Classe Case

Bien évidemment, la classe **Case** a été la première classe implémentée, elle joue un rôle fondamental puisqu'elle mime les mouvements des blocs du jeu PetRescueSaga.
Nous avons abstractisé cette classe puis on a crée deux nouvelles classes abstraites héritant de celle-ci :

Les classes abstraites "**CaseJouable**" et "**CaseNonJouable**".
En effet on a considéré que lors de la suppression d'une case dans un plateau, celle ci ne pouvait arriver que si la case était belle et bien une **CaseJouable.**
Cette implementation nous a beaucoup aidé à écrire nos méthodes dans la classe Plateau, notamment avec l'utilisation massive de l'opérateur "**instanceof"**.

On a crée des sous-classes de "**CaseJouable**" qui sont les cases couleurs respectives : "**CaseBleu**", "**CaseRouge**", "**CaseVert**", "**CaseJaune**", "**CaseMauve**". Ce sont les seuls objets pouvant être manipulés dans les operations de la classe Plateau.

Finalement on a étendu la classe **CaseNonJouable** à deux nouvelles sous-classes que sont "**CaseAnimal**" et "**CaseObstacle**".
Ici la distinction est importante pour l'implémentation graphique mais aussi pour les opérations de Plateau, en effet les **CaseAnimal** peuvent bouger mais les **CaseObstacle** sont immuables et ne bougeront jamais.

A) MODÈLE

2) Classe Plateau

Cette classe a été la classe fondamentale à coder afin d'obtenir un résultat un tant soit peu crédible.

La modélisation a été plutôt basique en lui octroyant un attribut plateau de type [][] Case, nous avions pensé à travailler avec des ArrayList ou des LinkedList, cependant le choix du tableau de tableau de Case a été le plus cohérent au vu de notre implémentation.

Comment avons-nous crée nos niveaux ?

Concernant l'initialisation des plateaux nous avons pensé à quelque chose de très astucieux (à notre sens) : écrire les niveaux en fichier.txt, les lire grâce à un **BufferedReader** et grâce aux données du fichier, crée un tableau tout prêt. La seule tâche un peu fastidieuse est de taper les fichiers à la main mais nous pensons que ça l'est bien moins que de le faire dans le code. Les fichiers auront la structure suivante :
La première ligne sera de la forme :

HL où H et L sont des chiffres, cela permet au **BufferedReader** de calculer la taille du plateau.

On a choisi la convention suivante :

posColonne/posLigne/Nature/Type où :
- posColonne est le numéro de la colonne;
- posLigne est la numéro de la ligne;
- Nature signifie jouable ou non jouable, on écrira 'j' pour jouable et 'n' pour non jouable;
-Type signifie deux choses :
si la Case est jouable alors on mettra la première couleur de la Case en minuscule, b pour bleu;
si la Case est non jouable alors on mettre 'o' pour obstacle,'v' pour vide, 'a' pour animal.

A) MODÈLE

Comment actualisons nous notre plateau ?

Il a fallu maintenant penser à comment implémenter toutes les méthodes de déplacement en évitant la diabolique **NullPointerException** à tout prix mais surtout en pensant à faire fonctionner ces méthodes pour n'importe quel type de plateau vu qu'avec notre système, on peut créer n'importe quel niveau !

1) La suppression

Pour supprimer une case, il a fallu vérifier trois choses :
- que les coordonnées entrées par l'utilisateur sont dans le tableau;
- que la case est une instance de **CaseJouable**.
- que cette case admette au moins une case adjacente de même couleur.

si l'une trois conditions n'était pas vrai, la suppression est rendue impossible et un message d'erreur apparaît.
Ensuite par des opérations récursives, nous supprimons les cases adjacentes de même couleur.
Rajoutons aussi le fait que lorsqu'une case est détruite, le score augmente de 10.

2) La réorganisation

La réorganisation elle aussi, se décompose en 2 étapes:
- descendre les cases qui ont des cases vides en dessous
- decaler les sous colonnes de coordonnées (k,y) vers la colonne (k,y-1). où k varie de 0 à hauteur -1

Le décalage des sous colonnes se fait de cette façon :
- Extraction d une colonne allant de (k,y) à (prochaineCaseObstacle,y) où prochaineCaseObstacle est la position où la colonne se finit (peut être hauteur -1)
- Vérification que la colonne (k,y-1) est vide
- Décalage vers la droite de toutes les cases allant de (k,y) à (prochaine obstacle,y)
- Descente des cases du plateau.

L'extraction de sous colonne est très utile car elle nous permet de gérer toutes (à notre connaissance) les possibilités de décalage de sous colonnes d'un plateau.
Nous avons aussi implémentée la méthode fusée qui supprimera une colonne de la surface de la Terre, fonctionnalité possible que si le score atteint 100, score qui sera reinitialisé lors de l'utilisation de la méthode fusée.

B) CONTROLEUR

1) Classe Joueur

La classe **Joueur** est aussi une classe fondamentale dans l'implémentation de notre projet. C'est la
seule classe avec **Plateau** et **Case** qui implémentent l'interface **Serializable**.
Tout a été codé de façon dynamique afin d'anticiper l'ajout de niveaux par les mainteneurs du jeu.
L'idée de cette classe est de sauvegarder dans une HashMap<Integer,Boolean[]> nommée avancement, la progression du joueur.
On liera la **clé i** de type **Integer** représentant un niveau à un tableau de **Boolean** de taille 2 représentant la progression du joueur.
La première position du tableau nous dira si le joueur a débloqué le niveau 'i' et la deuxieme position si le joueur a gagné le niveau i.
On pourra donc débloquer les niveaux suivants grâce à cette HashMap, par exemple :

> Supposons qu'on initialise un joueur J et qu'on dispose de 3 niveaux au total, si on check son avancement on aura :
> avancement.get(1) = {true,false}, en effet il aura accès au niveau 1 mais il ne l'a pas encore gagné, cependant on aura aussi
> avancement.get(2) = {false,false} et avancement.get(3) = {false,false} car ces niveaux ne sont pas encore débloqués! Ils le seront si et seulement si les niveaux inférieurs sont gagnés, il faudra donc vérifier si avancement.get(i-1)[1] == true.

La classe **Joueur** dispose d'un attribut plateau et celui ci est actualisé grâce aux modifications faîtes à son avancement, c'est-à-dire qu'en fonction de son avancement, le joueur pourra jouer à un plateau et pas au suivant s'il n'a pas encore gagné celui-ci.

2) Classe Jeu

Ici la classe **Jeu** sera la classe à partir de laquelle sera lancé le jeu.
Elle proposera diverses fonctionnalités mais les plus importantes sont :

- -Le lancement du jeu en mode terminal
- -Le lancement du jeu en mode Graphique
- -Le lancement du jeu du robot en mode terminal, robot qui effectuera des suppressions sur le plateau jusqu'à la victoire finale.
- -La sauvegarde d'un joueur à partir de son nom
- -Le chargement d'un joueur à partir de son nom

Bien évidemment, le joueur étant un objet implémentant l'interface **Serializable**, il sera possible de charger/sauvegarder un joueur dans le jeu terminal et dans le jeu graphique.

Vue : Prologue

Avant de commencer de parler de ce package, nous aimerions noté le fait que dans toutes les fenêtres du jeu, nous avons importé une police spéciale grâce à un bloc d'initialisation qui permettra d'importer une police à partir du dossier ./Ressources/font

C) VUE

1) La classe ImagePanel

L'idée ici était très simple, utiliser un **CardLayout** afin de stocker tous les panels contenus dans le jeu. Cependant nous avons rencontré des difficultés à afficher une **backgroundImage** en utilisant un **CardLayout**, de ce fait on a switch sur une solution un peu plus radicale et détestable à notre goût qui est l'utilisation de plusieurs **JFrame**.
De ce fait, on a crée une classe **ImagePanel** qui est une sous classe de **JPanel**.
ImagePanel sera le **contentPane** de toutes les autres **JFrame**, et elle contiendra une image de fond.
En fonction de la fenêtre affichée, l'image de fond pourra changer, c'est pourquoi nous nous sommes dits qu'il nous fallait une classe **ImagePanel**.
Nous avons aussi choisi d'utiliser des positions absolues, en effet nous avons préféré positionner nos composants comme bon nous semblait.

2) La classe MenuFrame

Cette fenêtre dispose de 3 boutons:
Jouer,**Charger**,**Quitter**.
Si le joueur clique sur **Jouer** alors une fenêtre de dialogue apparaîtra et permettra au joueur d'entrer un nom. Ce nom servira à initialiser un nouveau joueur et ouvrira une nouvelle fenêtre contenant les niveaux accessibles.
De la même façon si le joueur clique sur **Charger**, une fenêtre de dialogue se créera et proposera au joueur d'écrire le nom avec lequel il s'est enregistré auparavant, cela permettra de **désérialiser** le joueur et de charger une fenêtre de niveau avec son avancement personnel.
Si le joueur clique sur **Quitter** alors la fenêtre se fermera.

3) La classe niveauFrame

Ici on rajoutera aussi dynamiquement les niveaux qu'on a stocké sous forme de **JButton**.
Nous avons fait en sorte de griser les boutons des niveaux qui ne sont pas accessibles par le joueur actuel!
En cliquant sur un bouton on pourra accéder à la fenêtre du Jeu

4) La classe GameFrame

Ici est où se passe le jeu : on a une classe **PlateauPanel** qui contiendra toutes les opérations à faire sur le plateau actuel,
Le score sera actualisé en fonction des cases supprimées, le plateau sera mis à jour grâce aux méthodes de la classe interne **PlateauPanel**
la fusée sera disponible quand le score sera de 100 minimum.
Si le joueur gagne alors un pop-up apparaîtra pour l'informer de sa victoire puis on lui demandera s'il veut continuer ou s'il veut sauvegarder.

PROBLÈMES RENCONTRÈS & POINTS À AMÉLIORER

Les problèmes rencontrés

Le premier problème auquel nous avons dû faire face a été d"établir un cahier des charges cohérent et efficace.

N'ayant que très peu d'expérience dans ce type d'exercice, nous avons dû apprendre à être méthodique et à gérer les imprévus que pouvaient engendrer certaines de nos lacunes. De plus, malgré les travaux pratiques de ce semestre, nous avons éprouvé des difficultés à choisir les outils adaptés et à les utiliser.

Concernant les problèmes d'ordre technique, nous avons fait un mauvais choix en imbriquant un actionListener dans un autre, nous faisant perdre un temps non négligeable.

Par ailleurs, la partie Swing a été la partie où nous avons ressenti le plus de difficulté puisque nous n'avions pas les connaissances nécessaires afin de s'en servir efficacement.

Néanmoins, nous nous sommes épanouis dans ce projet. En effet, cet exercice nous a permis d'améliorer nos compétences en Java. Nous avions pour ambition de rendre ce projet présentable, nous avons ainsi pris le temps de découvrir et d'utiliser les fonctionnalités des outils graphiques Java.

Les points à améliorer

Concernant la **Serialisation**, nous écrivons actuellement dans un fichier qui se situe dans le dossier ./Ressources/Sauvegarde, nous pensons que cela peut poser des problèmes de permission d'écriture sur certaines machines.

Aussi, nous n'avons pas eu le temps nécessaire pour pouvoir essayer de remplir les plateaux dynamiquement : dans le niveau 6 du **PetRescueSaga**, lorsque l'on supprime la colonne tout à droite, celle ci se ré-remplit automatiquement.

Il aurait été judicieux de rajouter plus de niveaux et plus de bonus comme le marteau et, surtout, définir une limite de coups possibles accordés au joueur.

De plus, la maintenabilité du code est potentiellement un problème, notamment si les dossiers contenant les ressources venaient à changer. Même si la génération de niveaux a été conceptualisée de façon dynamique, l'utilisation des chemins d'accès aux différents fichiers ne l'est que partiellement. Ce point est ainsi essentiel à corriger.

Pour finir, nous souhaiterions insister sur notre absence de compétence concernant le CardLayout et l'utilisation des positions absolues des Components. Ne possédant pas les qualifications et le savoir requis pour cette manipulation, nous avons effectué celle-ci selon notre propre expérience sans avoir pu y accorder un temps suffisant.