Projet de POOIG

Jules Cauzinille
Alice Hammel
M1 LI

6 janvier 2021

Lancement du programme Le jeu se lance en exécutant le fichier Main.java contenu dans le dossier "src" avec 1 en argument pour la vue textuelle, ou 2 pour la vue graphique.

```
cd src
javac Main.java
java Main
```

Si aucun argument n'est utilisé, l'utilisateur·rice devra répondre à une question dans le terminal pour choisir le type d'interface.

Un mot sur la répartition du travail Plutôt que de nous répartir par avance différents aspects du développement, nous avons établi une liste des fonctionnalités à développer, puis nous nous sommes relayés pour les développer, puis relire et reprendre le travail l'un de l'autre. Entre chaque séance de travail nous discutions du travail effectué et des prochaines tâches à réaliser.

1 Parties traitées

1.1 Niveaux

Les niveaux sont stockés dans le dossier `levels`, dans des fichiers dont le nom correspond au numéro du niveau, avec différents caractères pour représenter les couleurs (des chiffres), les cases vides, les obstacles, et le sol.

En dessous du sol, deux chiffres représentent le nombres de fusées et de marteaux disponibles pour ce niveau.

Une classe `Niveau` dispose d'un constructeur qui lit toutes les données de ces niveaux à l'aide de `Scanners` et les stocke dans des attributs (une matrice d'éléments, le numéro du niveau, le nombre d'animaux à sauver, etc.). Ces objets niveaux servent ensuite à initialiser un objet `Plateau`, mais ne sont pas modifiés. Les objets `Niveau` correspondent aux niveaux avant qu'ils soient joués.

1.2 Plateau de jeu

On charge dans un plateau de jeu de type `Plateau` la matrice d'éléments d'un niveau que des méthodes vont ensuite modifier selon les actions des joueurs :

- `destroy()` : fonction récursive qui détruit une boîte si elle n'est pas seule et toutes les boîtes de la même couleur voisines de cette boîte.
  - `fuseeDestroy()` : détruit une colonne.
  - `marteauDestroy()` : fait un appel à destroy() et diminue le nombre de marteaux disponibles.
- `shiftDown()` : réaménage le plateau en déplaçant les boîtes vers le bas s'il y a des cases vides sous elles.
- `shiftLeft()` : similaire à shiftDown(), mais vers la gauche : on décale une colonne si la colonne à gauche est vide.
- `animauxSauves()` : si des animaux ont atteint le sol, cette méthode les enlève de la matrice et réduit le nombre d'animaux à sauver, puis fait un appel à `shiftDown()` pour réaménager le plateau.

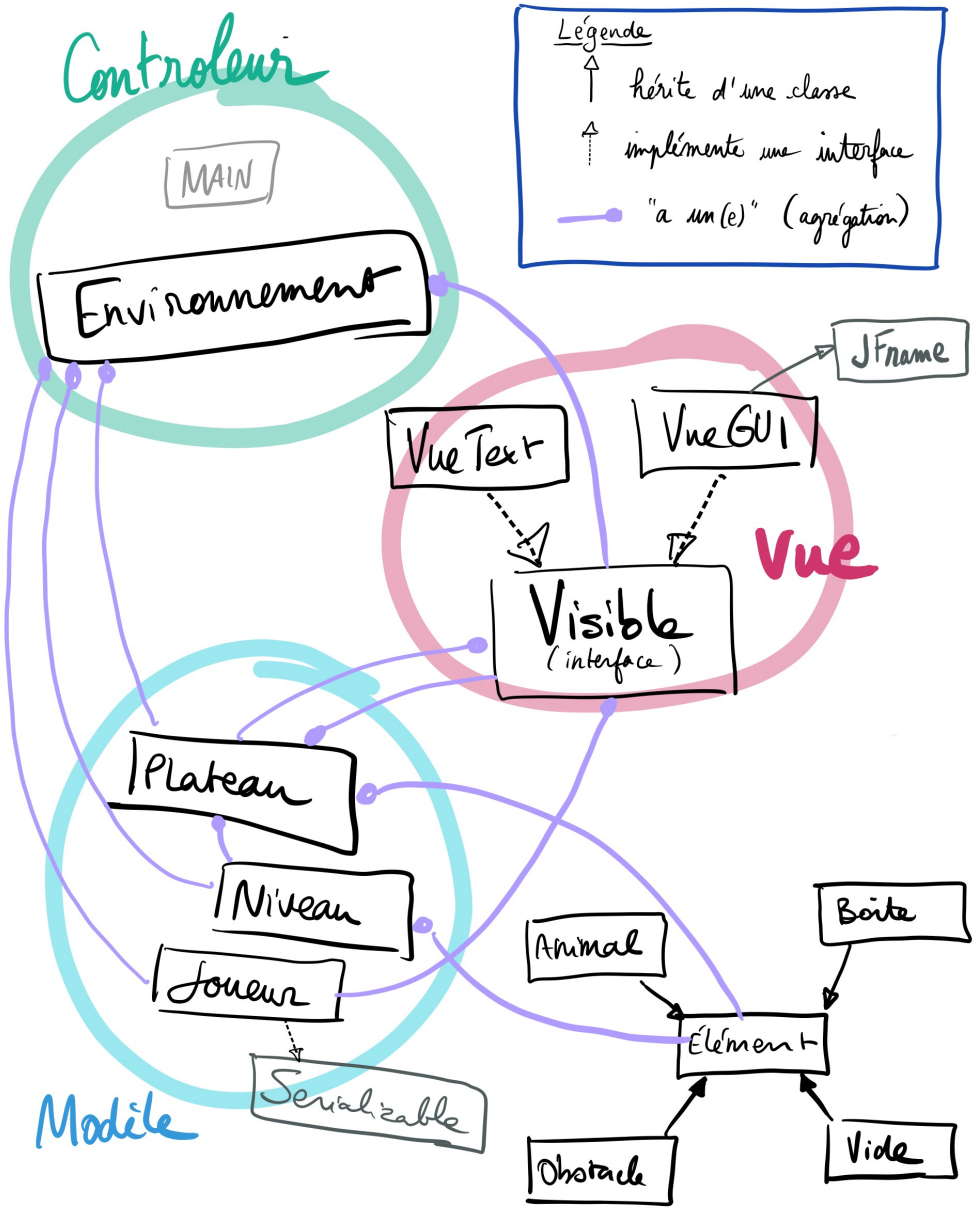

FIGURE 1 – Représentation graphique du modèle

1.3 Environnement et phases de jeu

L'`Environnement` a pour attributs une vue (de type déclaré `Visible`), attribuée dans le constructeur, un `Joueur`, un `Plateau` et un `Niveau` (ces derniers changent selon la progression du joueur).

Il dispose de méthodes pour gérer les phases du jeu :

— accueil (`welcome()`)

— choix du joueur (`choixJoueur()`)

— chargement de la sauvegarde (`loadJoueur()`)

— sauvegarde à la fin d'un niveau (`save()`)

— choix du niveau (`choixNiveau()`)

— lancement du niveau (`startNiveau()`)

— rejouer si game over (`startAgain()`)

— passer au niveau suivant ou revenir à la liste des niveaux (`choiceOrNext()`)

Nous avons parfois eu du mal à bien séparer vue, modèle et contrôleur.

C'est la méthode `main()` de la classe `Main` qui fait appel à ces méthodes afin de mettre en oeuvre le déroulé du jeu.

1.4 Sauvegardes

C'est la classe `Joueur` qui implémente l'interface `Serializable` et permet donc de sauvegarder les parties. Les attributs de cette classe permettent, pour chaque joueur, de stocker son nom, ses meilleurs scores pour chaque niveau, et la liste des niveaux débloqués, pour gérer la progression du joueur.

C'est l'environnement qui effectue la sauvegarde et les chargement des sauvegardes, qui sont stockées sous la forme de fichiers `.ser` dans le dossier `sauvegardes`).

1.5 Vues (textuelle et graphique)

Nous avons commencé par implémenter la vue textuelle (`VueText`) ainsi qu'une interface (`Visible`) contenant les méthodes de la vue appelées dans l'environnement. Notre idée était de développer l'environnement et la vue textuelle dans un premier temps, puis d'implémenter toutes les méthodes de l'interface dans une classe `VueGUI` qui étends `JFrame`. Malheureusement cela s'est avéré plus compliqué que prévu, car les vues textuelles et graphiques ne fonctionnent pas du tout de la même manière. En effet, avec une vue textuelle il était possible d'afficher plusieurs "écrans" (accueil puis choix des joueurs par exemple) les uns à la suite des autres, et qu'ils soient tous visibles pour l'utilisateur·rice, puisqu'ils s'affichent l'un après l'autre. Nous avons donc développé l'environnement sur ce principe, en faisant appel à différentes méthodes de Visible qui correspondent aux différentes phases du jeu.

Le développement de l'interface textuelle s'est passé sans problèmes majeurs. L'une de nos principales interrogations concernait les Scanners : de nombreuses fonctionnalités demandant un input de l'utilisateur·rice utilisent des scanners qui lisent l'input système (`Scanner(System.in)`), et font appel les unes aux autres. Si nous fermions ces scanners à la fin de ces fonctions, il était impossible de rouvrir des scanners lisant l'input système car la fermeture du scanner (`sc.close()`) cause la fermeture de la ressource associée (`System.in`). Nous avons donc décidé de ne pas fermer ces scanners et de laisser le *garbage collector* les gérer. Étonnamment, cela pose plus de problème lorsque le code est exécuté dans Eclipse que dans un terminal.

Quand nous avons commencé à travailler sur l'interface graphique, après avoir fini la vue textuelle, nous avons réalisé qu'il serait compliqué de garder le même déroulé dans l'`Environnement` avec une vue graphique. En effet, il nous était de premier abord impossible d'afficher différents "écrans" (accueil puis choix du joueur par exemple) les uns après les autres en les appelant dans l'`Environnement`, puisque la deuxième méthode "cacherait" immédiatement la première. Une interaction avec le joueur était nécessaire (cliquer sur "start" sur l'écran d'accueil avant d'ouvrir la fenêtre du choix des niveaux, par exemple). Contrairement à la vue textuelle, c'est le joueur qui déclenche les différentes phases du jeu. Dans la vue textuelle, c'est seulement quand l'utilisateur·rice doit entrer une instruction (nom du joueur, coordonnées du bloc, etc.) que l'exécution des phases de jeu se met en "pause".

Nous avons donc envisagé d'appeler les différentes méthodes de la vue graphique directement dans les `ActionListener` assignés aux boutons, mais cela aurait posé plusieurs problèmes. Nous aurions du modifier l'`Environnement` pour ce type de vues, ce qui aurait réduit la modularité de notre contrôleur,

et tout le déroulé du jeu aurait été implémenté dans la vue, ce qui n'aurait pas respecté le design pattern MVC que nous avons essayé de respecter.

Nous avons donc décidé de recourir à une solution annexe permettant, comme pour la vue textuelle, de mettre en "pause" le déroulement des différentes étapes. Pour ce faire, nous avons attribué un booléen `next` à la classe VueGUI, initialisé à `False` à l'ouverture d'une fenêtre et passant à `True` une fois que l'une des actions disponibles est exécutée par l'utilisateur·rice. Ainsi, notre contrôleur, qui n'a pas vraiment accès aux évènements de l'interface graphique, peut tout de même savoir quand passer à l'étape suivante en allant vérifier l'état du booléen `next` avec des boucles `while` "vides" qui ne s'arrêtent que l'orsque ce dernier passe à `True`. Cette méthode un peu "artisanale" s'est avérée très efficace et nous a permis d'éviter d'implémenter des patterns compliqués comme observable/observer qui nous aurait permis de notifier l'environnement à chaque action détectée sur l'interface... Au final, tout ce qui concerne l'interface en elle-même est géré localement et notre contrôleur Environnement n'a besoin que de ce booléen `next` pour lancer les différentes étapes du jeu au bon moment.

Une autre difficulté que nous avons rencontré est l'exécution de certaines tâches dans des `ActionListener`. Il semblerait que les seules variables modifiables par une expression lambda (des `actionListener` sur les boutons de notre interface dans le cas présent) soit des variables de type `final`. Dans ce cas, comment faire pour renvoyer l'indice d'un niveau sélectionné ou le nom d'un nouvel utilisateur ? Nous avons choisi de multiplier les attributs de la classe `vueGUI` pour que nos actionListeners viennent directement modifier ces derniers et que l'environnement puisse récupérer les valeurs dont il a besoin. Ainsi, la classe `vueGUI` possède, par exemple, un attribut `String playerName` et un attribut `boolean useFusee` permettant respectivement de récupérer le nom du joueur dans le contrôleur ou de savoir si le coup spécial "fusée" doit être utilisé.

2 Problèmes connus

— Il arrive que la réorganisation du plateau ne se fasse pas correctement, en particulier dans les niveaux ayant des obstacles. Le problème étant difficile à répliquer, nous n'avons pas encore pu suffisamment le comprendre pour modifier les méthodes de Plateau.

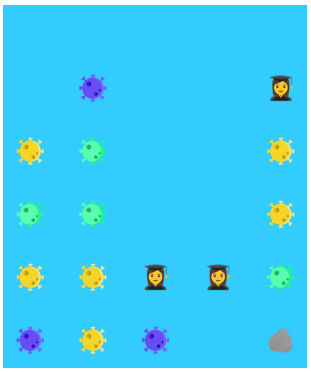

FIGURE 2 – Erreur de réorganisation

— Au niveau de la syntaxe du code en lui-même, certaines fonctions semblent être un peu trop longues : nous avons essayé d'écrire des fonctions les moins étendues possible pour rendre le code plus clair en séparant les méthodes en plusieurs étapes. Or la construction de certains éléments, dans la vue graphique notamment, ne semblait pas permettre un tel arrangement du code. La fonction `afficherPlateau()`, dans la `vueGUI` est par exemple très longue (elle fait presque 150 lignes) parce qu'elle demande la création de nombreux éléments à ajouter à la fenêtre (un menu sur la gauche avec plusieurs boutons, un titre en haut, le plateau en lui même, etc). Nous avons donc décidé de garder ces "grandes" fonctions tout en sachant qu'il aurait fallu subdiviser la création de la fenêtre en plusieurs étapes distinctes.

3 Pistes d'extension

Nature des joueurs

L'implémentation d'un joueur "robot" demanderait peut-être des modifications à notre modèle. Par exemple, dans l'état actuel de notre programme, les coordonnées sont récupérées par les vues, soit via l'input système soit via un clic sur un bouton, dans la méthode `move()` qui fait appel à la bonne méthode de `Plateau` selon l'input.

Pour pouvoir utiliser un joueur "robot", il aurait peut-être fallu que la méthode `move()` renvoie simplement les coordonnées et que le plateau les interprète, et avoir une méthode `giveCoordo()` dans `Joueur` qui donnerait les coordonnées à la vue. Une classe `JoueurRobot` étendrait `Joueur`, et générerait des coordonnées avec `giveCoordo()`.

Cependant, la récupération de coordonnées pour un joueur humain dépend forcément du type de vue utilisée, ce qui demanderait de faire appel à des méthodes de la vue dans la classe `Joueur`.

Amélioration de l'interface graphique

- Affichage du plateau : nous avons commencé par afficher les plateaux en entier (de la première case jusqu'à la dernière). C'est d'ailleurs ce qu'il se passe dans la vue textuelle qui, n'étant qu'un démonstrateur, ne possède pas exactement le mêmes traitement du jeu que la vue graphique. Dans la vue graphique, le plateau n'est pas affiché intégralement. Seule la partie en bas à gauche du plateau est effectivement visible, et certaines cases apparaissent à la suite de la destruction d'autres cases. Cela semble poser quelques problèmes sur les niveaux les plus étendus (le 6 par exemple), puisqu'on ne voit pas où sont les animaux à sauver avant d'avoir détruit assez des cases pour qu'ils apparaissent. Nous aurions donc pu utiliser des `JScrollPane` qui permettraient d'aller voir le haut et la droite du plateau pour choisir la prochaine case à détruire en connaissance de cause. Cela aurait permit à notre plateau d'avoir un `scroller` et de laisser la possibilité à l'utilisateur de se déplacer dans celui-ci.
- Nouvelle vue graphique avec des animations : plusieurs éléments du jeu pourraient être "animés" dans la vue graphique. Par exemple, le plateau affiché pourrait permettre de voir les cases descendre plutôt que de disparaître et de réapparaître ailleurs. Malheureusement, nous avons choisi d'utiliser des `JButton` dans des `GridLayout` ce qui ne nous permet pas d'implémenter de telles fonctionnalités.
- Ajout des boutons exit et help.

Sauvegarde de l'avancement dans chaque niveau

Actuellement la sauvegarde ne se fait qu'une fois qu'un niveau est terminé, il suffit donc de sauvegarder les meilleurs scores du joueur et les niveaux débloqués. Si nous voulions que la classe `Plateau` implémente également `Serializable`, nous pourrions également sauvegarder l'état du plateau pour chaque joueur. Cette sauvegarde devrait alors être effectuée après chaque réorganisation du plateau et permettrait à un joueur de reprendre sa partie là où elle s'était arrêtée.