Rapport de projet
Programmation Orientée Objet

Eglantine Kremer-Cochet
Finaritra Randriamitandrina

Printemps 2022

Table des matières

Chapitre 1

2048

1.1 Vue d'ensemble du travail réalisé

C'est un 2048 classique. Au début de la partie, le chronomètre se lance et deux cases sont créées aléatoirement avec pour valeur 2 ou 4. On peut déplacer l'ensemble des cases avec les flèches directionnelles du clavier. Il y a fusion de 2 cases si ces dernières se trouvent côtes à côtes après déplacement et ont la même valeur. À chaque fois que l'on clique sur une flèche, l'affichage du temps se met à jour. De plus, à chaque déplacement de cases effectué, une nouvelle case apparaît. Elle est de couleur bleu (pour être mise en évidence) et peut être de valeur 2 ou 4. La couleur des cases change selon la valeur qu'elle contient (ex : Jaune pour la valeur 2). On peut également annuler une action en appuyant sur le bouton *backspace* (retour en arrière).

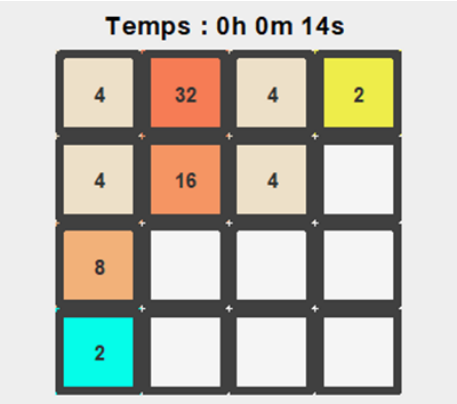

FIGURE 1.1 – Fenêtre du 2048

La partie s'arrête lorsque toutes les cases de grilles sont remplies (et qu'il n'y a plus de fusion possible) ou lorsqu'une case atteint la valeur 2048. Un pop-up apparaît alors avec un message. Lorsqu'on clique sur OK ou sur la croix, toutes les fenêtres se ferment.

FIGURE 1.2 – Pop-up si le joueur a gagné

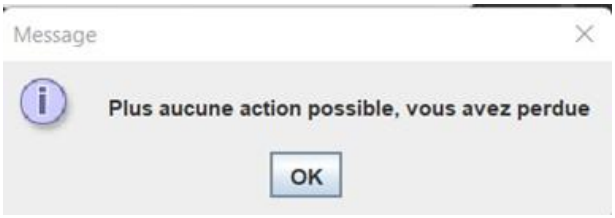

FIGURE 1.3 – Pop-up si le joueur a perdu

1.2 La liste des nouvelles fonctionnalités

Toutes les fonctions de cette liste ont été réfléchies et codées à deux.
Dans l'ordre chronologique :

- En premier lieu, nous avons créé la classe POINT (contenant les entiers x et y) qui gère les coordonnés des cases sur la grille.
- Dans la classe JEU, nous avons créé la hashmap *hmCases* qui, à partir d'un clé **Case**, nous renvoie la valeur des coordonnés **Point**. On a également créé **Case**[][] *tabCases*.
- Dans la classe JEU, nous avons codé **Case getCase(Case, Direction)** qui récupère la case à côté en fonction de la direction.
- Dans la classe JEU, dans le constructeur **Jeu()**, on augmente la dimension du tableau de 2 pour avoir des cases de bordure en plus des cases de jeu dans notre grille. Puis, on initialise les valeurs à -1 pour les bordures. Cela permettra, lors des déplacements, de savoir lorsqu'une case est contre la bordure et ne peut plus se déplacer.
- Dans la classe CASE, nous avons ajouté l'attribut **Jeu** *jeu* qui est une référence au jeu et permet la communication de JEU vers CASE. On l'initialise dans le constructeur de CASE, **Case(Jeu jeu)**.

→ *Jusqu'ici, si on compile, dans la fenêtre, sur les bordures, on voit afficher des "-1" et on utilise encore la fonction rnd().*

- Dans la classe CASE, on crée **move(Direction d)**. On l'implémente en utilisant la fonction **getCase(Case case,Direction d)** vue précédemment dans la classe JEU. Elle définit le déplacement de la case en fonction de la case à côté :
  — Si elle est vide, alors on déplace la case.
  — Si elles ont la même valeur alors on les fusionne.
  — Si la valeur est différente ou si on est en bordure (comparaison à une case de valeur -1) alors on ne fait rien.
  — On réitère tant que la case à côté est vide ou si elles ont la même valeur.
  La fusion est gérée de la manière suivante :
  On détruit la case d'à côté, on double la valeur de la case actuelle et on la déplace sur la case d'à côté.
- Dans la classe JEU, on crée la fonction **deleteCase()** qui permet de supprimer une case de la grille (pour la hashmap, on utilisera **hashmap.remove()** pour supprimer l'élément et on remplacera sa position du tableau par *Null*).
  → Elle est utilisée pour la fusion dans CASE :**move(Direction d)**.
- Dans la classe JEU, on crée une fonction **moveCase(Case c, Direction d)**, qui déplace la case dans la direction d sur la grille en changeant ses coordonnées dans la hashmap et sa position dans le tableau.
- Dans classe JEU, on crée la fonction **moveJeu(Direction d)** qui va faire une boucle de parcours sur toutes les cases de la grille existante. Le parcours est différent selon la direction :
  — Si la direction est gauche ou haut alors on parcourt le tableau de gauche à droite et de haut en bas.
  — Tandis que si d est droite ou bas, elle parcourt le tableau de droite à gauche et de bas en haut.
  Pour chacune de ces cases, on appelle la fonction CASE :**move()**.
- Dans la classe SWING2048, on a remplacé la fonction **rnd()** par **moveJeu(Direction d)**.
- Dans la classe SWING2048, on a configuré l'affichage du jeu pour ne pas faire apparaître les cases en bordure de valeur -1.

→ *À présent, dans la fenêtre, la gestion du déplacement et de la fusion des cases est fonctionnelle. Nous avons créé ces différentes fonctions de déplacement pour respecter le* ***Modèle Vue Controleur****. C'est la classe* Case *qui décide de l'action appliquée à la case.*
À ce stade-là, une case peut encore être fusionnée deux fois au même coup.

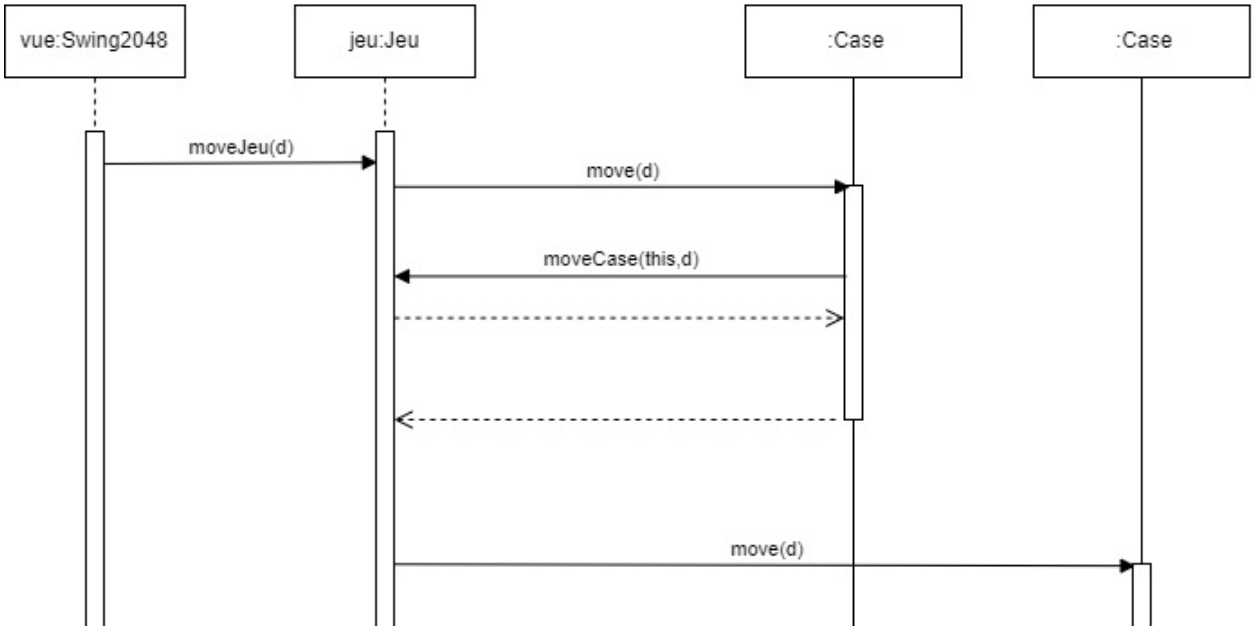

FIGURE 1.4 – Diagramme de séquence simplifié du cas : déplacement de la case

- Dans la classe JEU, on a créé la fonction **ajoutCase()**. Elle permet d'ajouter une nouvelle case aléatoirement de valeur 2 ou 4 à l'emplacement d'une case vide. On ajoute une case à chaque déplacement possible.
- Pour définir les déplacements possibles ou non, dans la classe JEU, on a créé le booléen statique *action* qui est vrai seulement s' il y a eu déplacement et/ou fusion de cases.
  On l'initialise à *false*, mais il devient *true* dans la fonction **move(Direction d)** de la classe CASE s'il y a fusion et/ou déplacement.
  Dans la fonction JEU :**moveJeu(Direction d)**, on va utiliser ce booléen. Si le booléen action est *true*t alors, on ajoute un nouvelle case.

→ *À présent, à chaque action possible, une nouvelle case se crée sur la grille. Néanmoins, par exemple, si toutes les cases se trouvent sur la bordure de droite et qu'on veut faire un déplacement à droite, aucune case ne va se créer car c'est un déplacement perdant.*

- Dans la classe JEU, on va créer une pile *deja_fusionne* à qui, à chaque fusion, on ajoute une référence à une case déjà fusionnée. Le but ici est de ne pas fusionner une case à une autre case déjà fusionnée. Si la case courante a déjà été fusionnée ou si la case à côté a déjà été fusionnée (c.a.d. si elle se trouve dans la pile), on ne fait rien. Sinon, on fusionne. Il ne faut pas oublier de vider la pile au début du coup suivant.
- Pour coder la fin de la partie, on crée la fonction **testEndGame()** dans la classe JEU qui vérifie les conditions de fin de Jeu perdante (si aucun déplacement n'est possible).
  On crée aussi la fonction **endGame(boolean gagnante)** qui met fin au jeu selon la fin si elle est gagnante ou perdante.
  — Dans CASE :**move(Direction d)** si la nouvelle valeur lors d'une fusion est égale à 2048, on appelle **endGame(true)**.
  — Dans JEU :**testEndGame()**, si toutes les positions du tableau sont remplies, alors on appelle **endGame(false)**.
  Pour cela, on a créé la fonction JEU :**numberCurentCase()** qui renvoie le nombre de cases sur

le plateau.

→ À présent, nous avons le jeu 2048 de base complet. Il prend en compte les déplacements dits "perdants" et le fait qu'une case ne puisse fusionner qu'une seule fois par coup.

1.3 Les extensions

1.3.1 Améliorations graphiques : *Finaritra & Eglantine*

Couleurs des cases

Dans la fonction **rafraichir()** de La classe Swing2048, on code pour chaque valeur de case une couleur.

Les nouvelles cases sont ajoutées dans une pile dans la classe Jeu nommée *addedCase*. La classe Swing2048 va l'utiliser pour colorier en bleu les nouvelles cases. Il ne faut pas oublier de vider la pile dans Jeu :**moveJeu(Direction d)**.

Arrondie des cases

Plusieurs idées ont été tentées. Première idée : modifier la bordure en créant une classe RoundedBorder
→ non concluant, le background reste carré et on a juste une fine bordure noire autour

Seconde idée : créer une classe RoundedLabel pour faire des JLabel arrondis
→ non concluant pour la même raison. J'ai cependant laissé le fichier existant pour conserver une trace du travail effectué.

Dans les deux situations, j'ai utilisé l'objet **RoundedRect**

Finalement, lors de la création de l'objet **LineBorder**, j'ai ajouté le boolean *true* en paramètre pour avoir un arrondi.

1.3.2 Affichage du temps : *Finaritra*

Dans la classe Jeu, on a rajouté l'attribut Timer *timer* et la hashmap <String, Integer> *gameTime* pour la gestion du temps.

On a créé la classe CustomTime dans le dossier modele pour faire la conversion en heure, en minute et en seconde du temps et on la stocke dans *gameTime* . La classe Swing2048 va faire l'affichage du temps en utilisant *gameTime* .

1.3.3 L'historique retour en arrière : *Eglantine*

Création de la classe LogFile, avec les fonctions **createLogFile()**, **writeLogFile()**, **deleteLogFile()** et **readLastLineLogFile()**. Ainsi, au lancement du jeu, un fichier *log.txt* est créé et à chaque coup, la table est enregistrée dans une nouvelle ligne du fichier. Pour revenir au coup précédent, on appelle **readLastLine-LogFile()** pour récupérer la table du coup précédent, on met à jour le jeu, et on affiche. Je voulais rajouter en plus le fait de supprimer le coup annulé, mais supprimer une ligne d'un fichier n'est pas très simple et le temps à manquer.

Si nous avions eu le temps, nous aurions aussi pu utiliser ce fichier pour permettre des sauvegardes, et par exemple fermer la fenêtre et reprendre le jeu plus tard.

1.3.4 Les pop ups : *Finaritra*

On a créé la classe PopUp dans le dossier vue_controleur qui affiche avec la fonction **show(String sms)** un pop-up avec le message sms. On teste dans la fonction Swing2048 :**ajouterEcouteurClavier()** la fin de la partie. Si c'est le cas, alors un affiche un pop-up avec un message gagnant ou perdant.

1.3.5 Animations : *Eglantine*

Glissé des cases lors du déplacement

Création de la fonction **animationGlisseCase()** dans Swing2048, puisque cette fonction fait parti de la vue. Cependant, je ne sais pas comment faire appel à cette fonction, et j'ai arrêté de chercher à cause d'un autre problème que l'on va voir au prochain point. J'ai cependant laissé la fonction commentée dans le code source pour prouver que le sujet a été traité.

Apparition des cases en fondu

Cela se fait directement dans la fonction **rafraichir()**. On part de la couleur de la case nulle pour arriver en fondue sur la couleur de la nouvelle case apparue. Pour rendre le dégradé visible, je fais des *sleep()* sur le thread en cours.

Problème : l'affichage du code après compilation ne se fait pas ligne par ligne mais tout est d'abord chargé et ensuite affiché d'un coup. Par conséquent, les *sleep()* créent seulement un temps de latence durant lequel il n'y a aucun affichage, puis l'affichage terminé apparaît.

1.4 Diagramme de classe

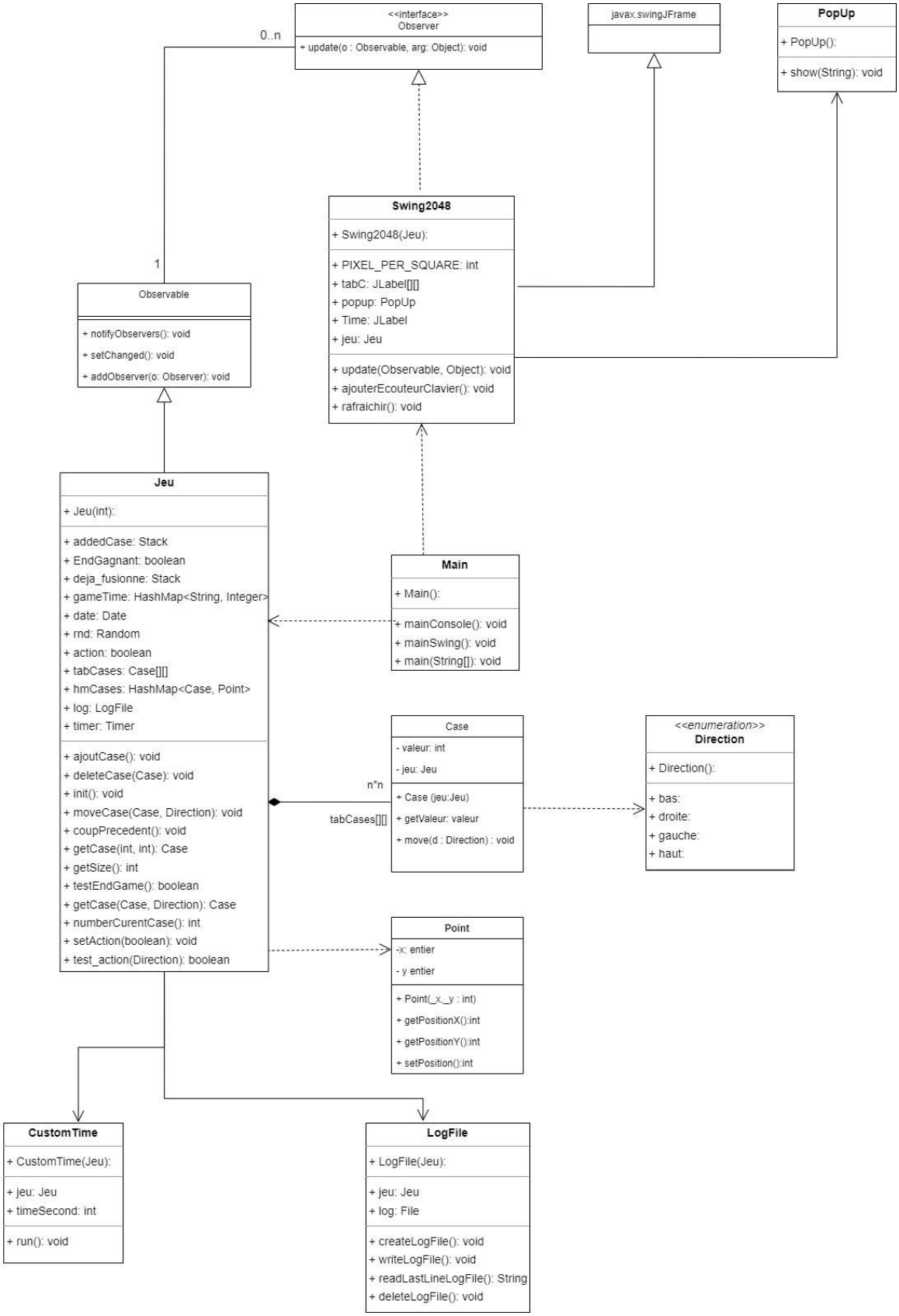

FIGURE 1.5 – Diagramme de classe final