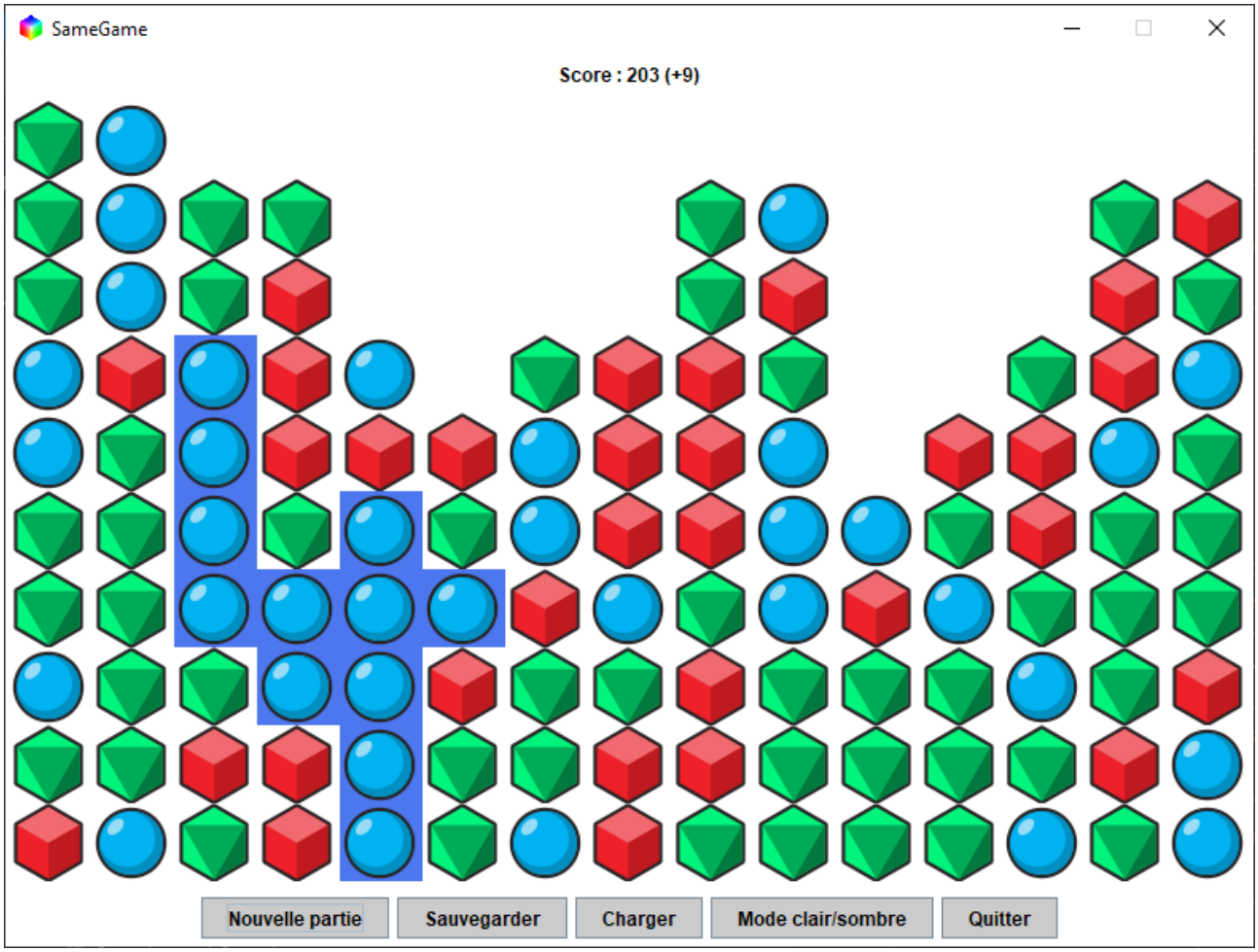
SameGame



Théo CHOLLET
Tomy DA ROCHA
1ère année IUT – Fontainebleau

Sommaire

Index des figures

1 Contexte du projet

Le projet consiste à créer un jeu nommé « **SameGame** », c'est un jeu de type puzzle des années 80. Le jeu se déroule sur une grille qui est composée de **10 lignes** et de **15 colonnes**. Des blocs sont contenus dans cette grille, ils peuvent être de **3 couleurs** différentes.

Le jeu consiste à éliminer un maximum de blocs par coup afin d'obtenir le meilleur score. Pour **éliminer un groupe de blocs**, il faut que celui-ci soit au minimum adjacent à un bloc de la même couleur. Plus il y aura de blocs dans le groupe éliminé par le joueur, plus le score sera élevé.

La partie se termine quand plus aucun groupe (2 blocs adjacents au minimum) n'est présent dans la grille.

La figure ci-dessous présente le principe de formation d'un groupe de blocs adjacents et de même couleur qui peut être éliminé par le joueur.

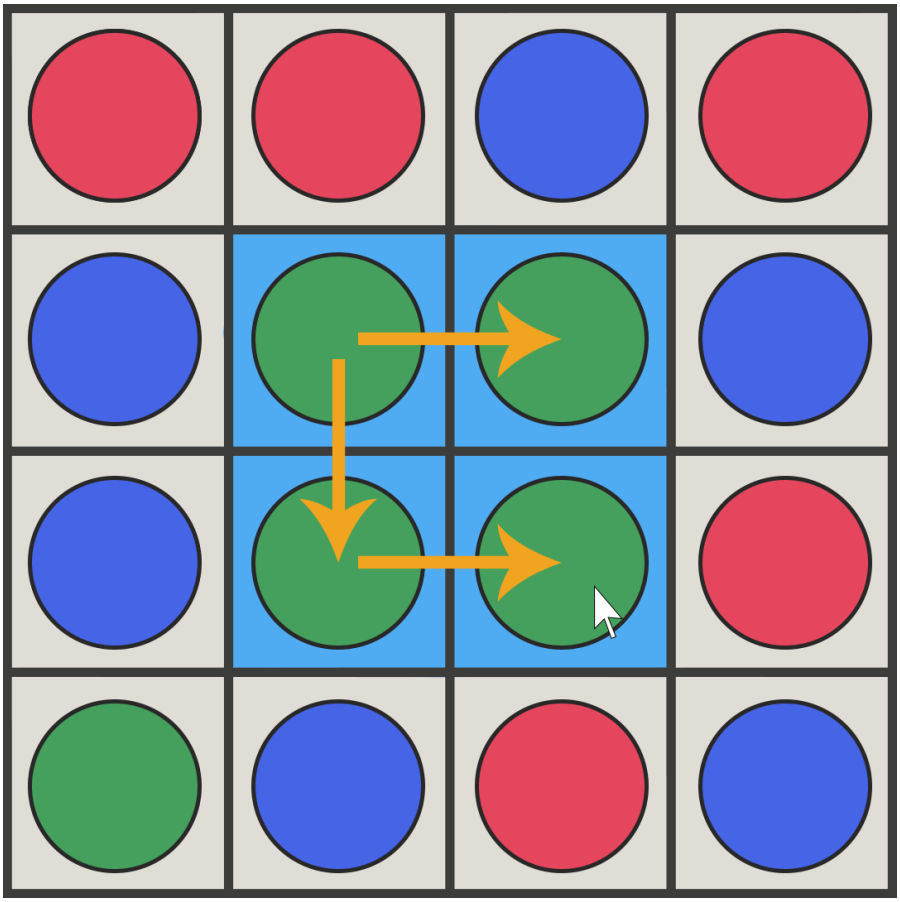


Figure 1: Principe de blocs adjacents formant un groupe.

2 Fonctionnalités

Nous décrivons dans cette partie les différentes fonctionnalités de ce jeu.

2.1 Format de fichier grille

Le fichier grille est enregistré avec un format particulier :

- le type de fichier est : **texte** ;
- l'extension du fichier est : '**.grid**' ;
- il est composé de **15 colonnes** et **10 lignes** (comme la grille) ;
- les index de colonne et de ligne dans le fichier correspondent aux index de colonne et de ligne de la grille ;
- chaque caractère enregistré correspond à une **couleur** ;
- la fin de ligne est faite avec les caractères : '**\r\n**'.

Les caractères pour chaque couleur sont les suivants :

- '**B**' : bloc bleu ;
- '**R**' : bloc rouge ;
- '**V**' : bloc vert ;
- ' ' (espace) : bloc vide.

Un aperçu d'un fichier enregistré est donné dans la figure ci-dessous :

```
RVVRRVRVBBBBRBV
BVVVVRBVVBRRVRB
VBBRVRBVRRBRRRR
BRBVBRBBVVBRVRV
RVBVBBBRRBRRRBV
RVVVBBRBVVBVVRB
BRBRBBBRBVRVRRV
VRRVVBBVVBBRVVV
BVRRVVBRVRRRBVV
BBRBBBBRVVRRVRB
```

Figure 2: Exemple de fichier grille.

2.2 Nouvelle partie

Au démarrage du jeu, une grille par défaut est chargée, celle-ci est stockée dans un fichier nommé : `default.grid`.
Pour créer une nouvelle partie, le jeu permet de générer une grille aléatoirement (bouton « Nouvelle partie »), la couleur des blocs dans chacune des cases de la grille est alors choisie aléatoirement parmi les trois couleurs disponibles.

2.3 Chargement

Il est possible de charger une partie grâce à un fichier qui respecte le format.

Une boîte de dialogue permet de choisir le fichier à charger. Tous les fichiers ont été stockés dans le répertoire '**save**'. Le répertoire de chargement est **relatif** par rapport au répertoire où se trouve le jeu. Les formats de fichiers sont directement filtrés avec l'extension '**.grid**'.

La boite de dialogue de choix de fichier permet de naviguer directement dans le bon répertoire et filtre aussi les extensions de fichier.

La figure ci-dessous présente cette boîte de dialogue :

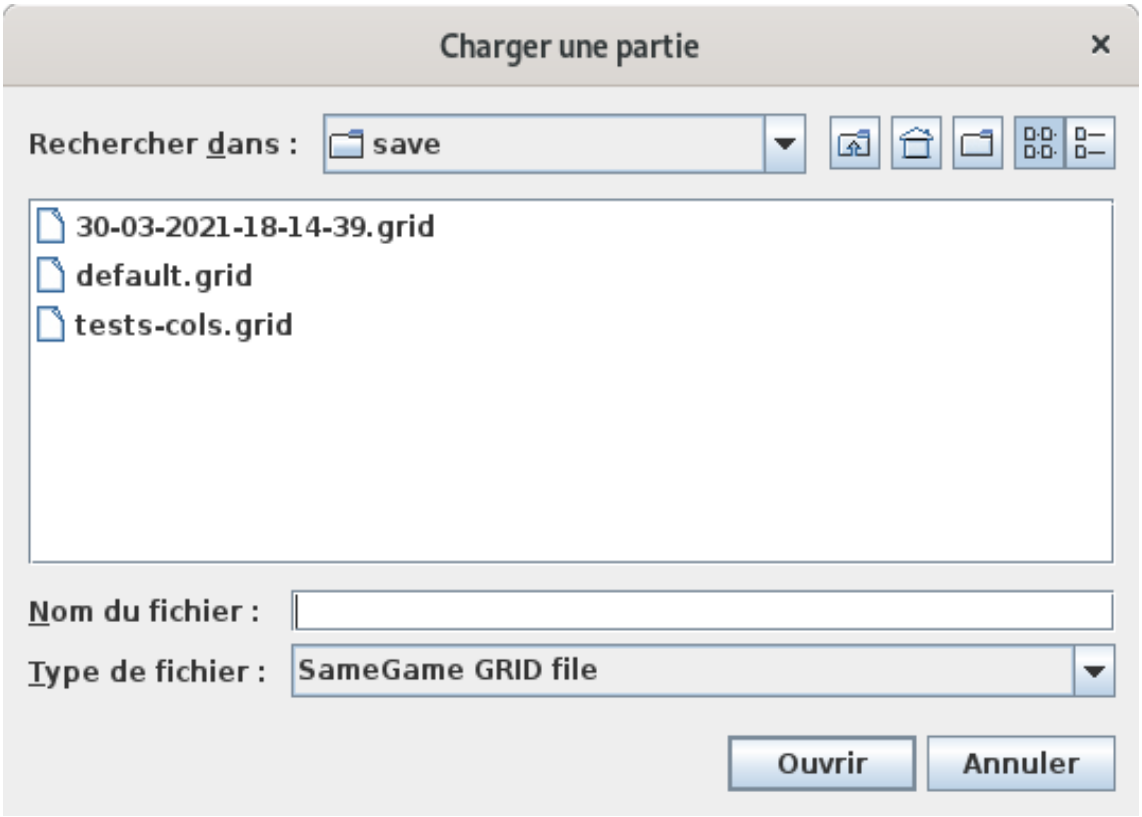


Figure 3: Boîte de dialogue de choix de fichier à charger.

2.4 Sauvegarde

Nous avons trouvé cela pertinent de proposer à l'utilisateur la possibilité de sauvegarder sa partie, il pourrait par la suite la charger.
Cette fonctionnalité permet aussi de générer plusieurs grilles aléatoires (nouvelle partie) et de les enregistrer. Cela permet de disposer de plusieurs séries de grille facilement.

La progression de la partie peut être sauvegardée, à tout moment. La sauvegarde se trouvera aussi dans le dossier '**save**', comme pour le chargement. Il y a une boîte de dialogue qui permet de choisir le nom de fichier. A l'ouverture de cette boîte de dialogue, le nom de fichier par défaut sera composé de la date et de l'heure sous le format suivant : '**dd-MM-yyyy-HH-mm-ss**'.

La figure ci-dessous présente la boîte de dialogue de choix de fichier à sauvegarder :

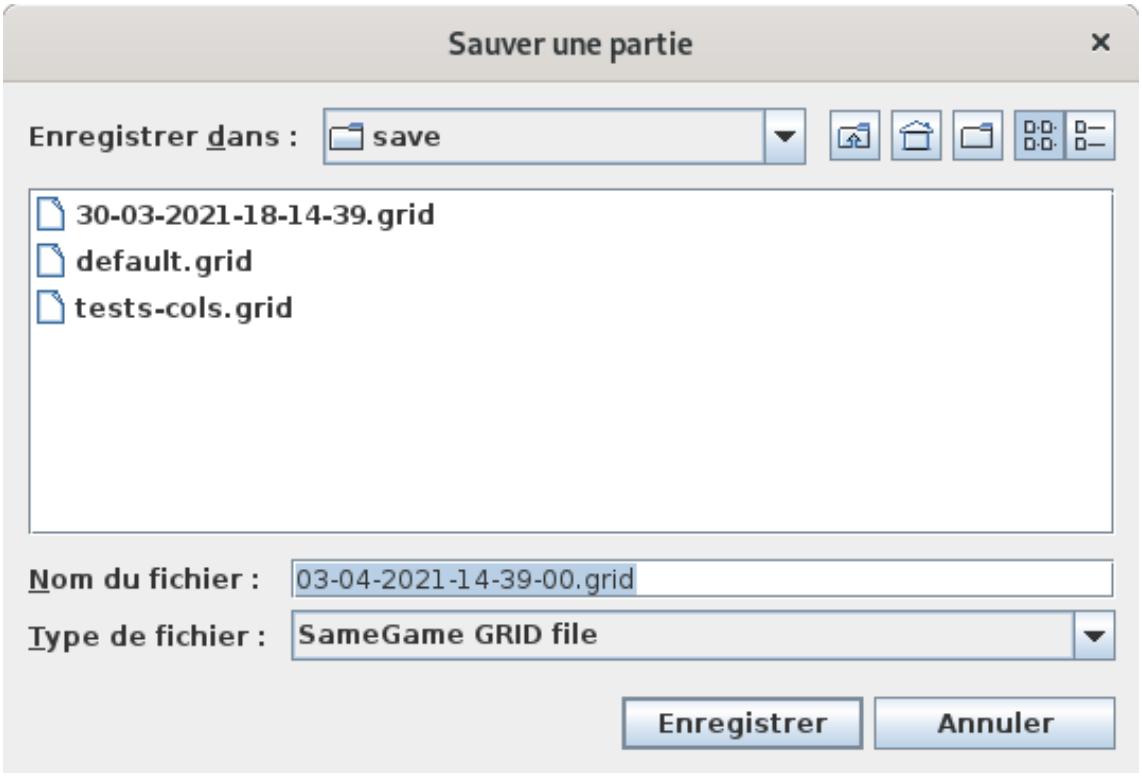


Figure 4: Boîte de dialogue de choix de fichier à sauvegarder.

Le contenu du fichier sera au format déjà décrit dans le paragraphe 2.1. Il a aussi l'extension '**.grid**'.

Si la partie sauvegardée est entamée, chaque bloc « vide » (c'est-à-dire supprimé) sera enregistré comme un caractère d'espacement afin de ne pas perturber le format.

Par contre, nous n'avons pas codé l'enregistrement du score car il n'était pas prévu initialement dans le format imposé. Nous pensons cependant qu'il serait, par exemple, possible d'enregistrer tous les scores de chaque grille dans un fichier séparé. Nous n'avons pas intégré cette fonctionnalité de sauvegarde de score dans cette version de développement.

2.5 Mode clair/sombre

Les couleurs des éléments graphiques peuvent être changées afin de rendre l'expérience de jeu plus confortable, et personnalisée, selon les préférences du joueur.

Ainsi les éléments graphiques seront soit clairs (fond blanc, texte noir), ou bien sombres (fond noir, texte blanc/gris).

3 Structure du programme

3.1 Modèle-vue-contrôleur

Nous avons choisi de structurer le jeu avec une architecture du type **M.V.C** (**M**odèle, **V**ue, **C**ontrôleur), chaque classe a un rôle précis, cela permet de développer chaque classe indépendamment. Cela permet donc de travailler en groupe sur des classes différentes, ce qui est propre à la programmation orienté objet.

Le principe de ce type d'architecture a pour but de séparer les responsabilités des classes :

- les **données** sont stockées dans la partie « Modèle » ;
- la **représentation** des données est assurée par la partie « Vue » ;
- la partie « Contrôleur » permet de gérer les **actions** de l'utilisateur.

La figure ci-dessous présente cette architecture :

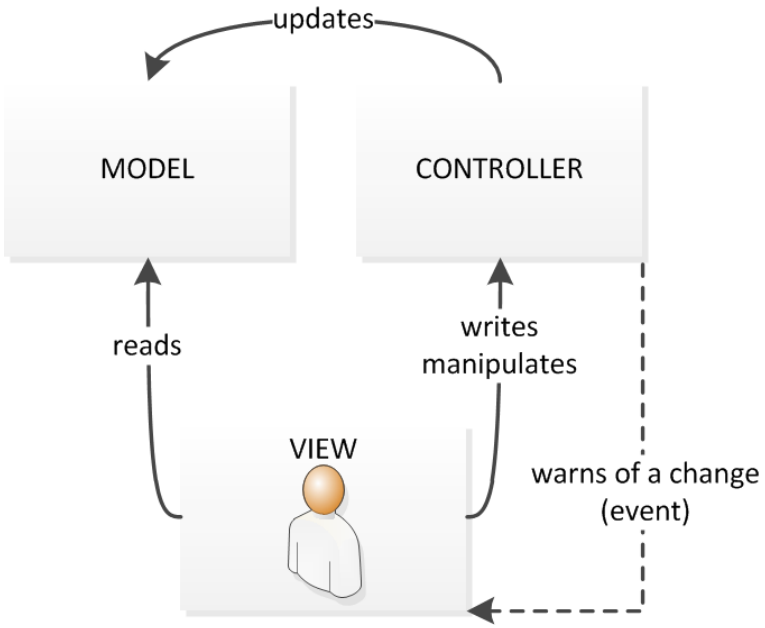


Figure 5: Interactions entre le modèle, la vue et le contrôleur.

Nous présentons dans les paragraphes suivants les différentes classes associées à cette architecture.

3.2 Modèle

La classe '**Defines**' contient toutes les définitions communes du programme, la classe ne contient que des constantes qui permettent de paramétrer les différentes options du jeu : couleurs, extensions de fichiers, intitulés des boutons, etc.

La classe '**Grid**' permet de stocker les éléments de la grille de jeu et de gérer tous les calculs nécessaires pour pouvoir progresser dans la partie.
Elle gère :

- le chargement ou la sauvegarde d'un fichier,
- la génération aléatoire d'une grille,
- la détection des blocs voisins, des groupes,
- le décalage et la suppression des blocs.

La classe '**ScorePanel**' a un double rôle, elle vérifie, enregistre et affiche le score obtenu par le joueur. Nous n'avons pas séparé, pour cette classe, la partie vue de la partie modèle, car de notre point de vue, il n'y avait que le score à gérer (partie modèle) et à afficher (partie vue).

3.3 Vue

Toutes les classes de la partie vue sont des classes pour la représentation graphique.

Les classes '**ScorePanel**', '**GridPanel**', '**GameMenuPanel**' et '**EndPanel**' sont responsables de l'affichage d'une partie de l'écran de jeu. Respectivement, elles gèrent l'affichage : du score, de la grille, du menu et du panneau de fin.
Elles héritent toutes de la classe '**GamePanel**' (dérivée de la classe JAVA 'JPanel'). Celle-ci contient les attributs et méthodes communs à tous les panneaux.

La classe '**MainWindow**' dérive de la classe JAVA 'JFrame', c'est la fenêtre principale qui contient tous les panneaux. C'est elle qui crée ces panneaux et permet d'interagir avec eux.

La classe '**GridPanel**' est le panneau d'affichage de la grille, elle est constituée d'un tableau de blocs (classe '**Block**').

La classe '**Block**' sert à générer l'image correspondante à la couleur du bloc à afficher dans la grille.

3.4 Contrôleur

Pour répondre aux actions de l'utilisateur, les classes '**GameMenuEvents**' et '**BlockEvents**' s'occupent de répondre visuellement, et mécaniquement aux requêtes du joueur.

Par exemples, elles gèrent les actions pour répondre au clique sur les boutons du menu, ou bien au survol de la souris sur la grille de jeu, en vérifiant si le groupe doit être ou non graphiquement visible.

Ces classes interagit avec la fenêtre principale '**MainWindow**' qui connaît tous les panneaux.

En effet, cette dernière classe 'MainWindow' connaît tous les panneaux, elle implémente plusieurs méthodes qui permettent de :

- créer, sauver ou charger une partie,
- changer de mode (clair ou sombre),
- détecter la fin de la partie.

3.5 Liste des classes

Le tableau ci-dessous résume toutes les classes du programme avec leur type.

Classe	Type	Résumé
Block	Vue	Blocs à dessiner.
BlockEvents	Contrôleur	Gestionnaire de clics sur les blocs.
Defines	Modèle	Définitions communes.
EndPanel	Vue	Panneau de fin.
GameMenuEvents	Contrôleur	Gestionnaire des clics dans le menu.
GameMenuPanel	Vue	Panneau pour le menu.
GamePanel	Vue	Classe mère des panneaux.
Grid	Modèle	Grille de jeu.
GridPanel	Vue	Panneau pour la grille.
Main	Contrôleur	Programme principal.
MainWindow	Vue	Fenêtre principale.
ScorePanel	Modèle-Vue	Panneau pour le score.

3.6 Diagramme de classes

Le diagramme de classes ci-dessous permet de résumer l'architecture du programme et l'organisation des classes :

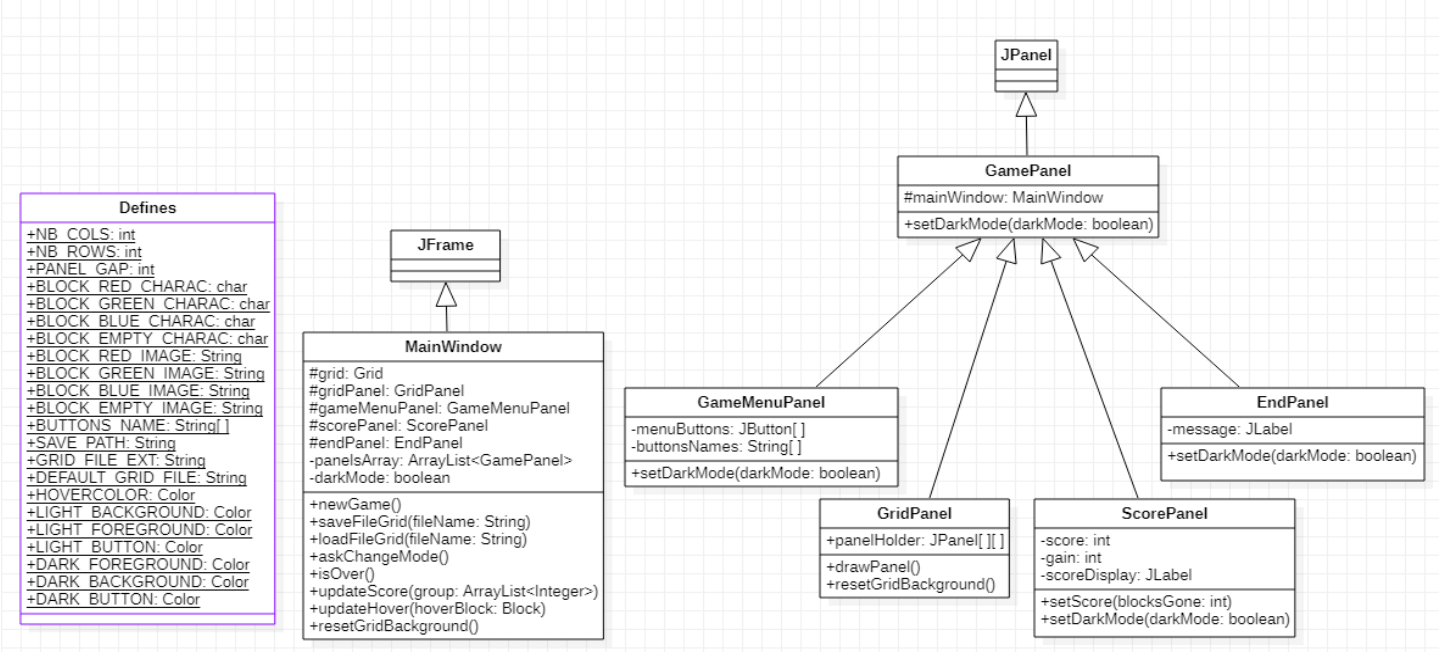


Figure 6: Diagramme de classes (1/2).

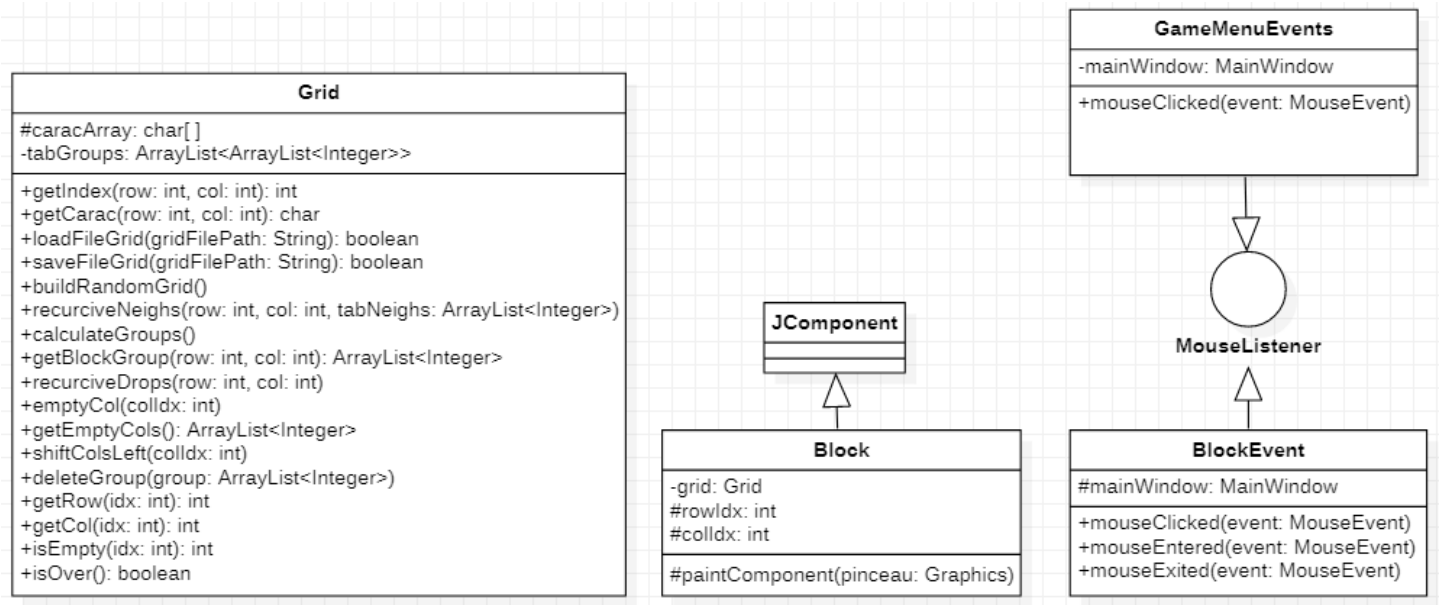


Figure 7: Diagramme de classes (2/2).

4 Description des algorithmes

4.1 Détection des groupes

La classe '**Grid**' est un objet qui permet de travailler sur la grille de jeu, mais sans représentation graphique.

Elle possède un attribut '**caracArray**' qui stocke la grille sous forme de caractères dans un tableau à une dimension. Nous avons choisi de travailler avec un tableau à une dimension plutôt que 2 pour faciliter les développements, notamment pour les boucles.

Pour transformer l'indice (index) de ce tableau en coordonnées ligne-colonne, nous avons implémenter une méthode '**getIndex()**' qui calcule l'index dans le tableau à partir des coordonnées, selon l'équation suivante :

index = index_ligne * nb_cols + index_col

où :

- index_ligne : index de ligne du bloc,
- index_col : index de colonne du bloc,
- nb_cols : nombre de colonnes dans la grille,
- index : index dans le tableau de caractères.

Afin de déterminer si un bloc possède un ou plusieurs voisins, nous avons utilisé une méthode récursive qui prend en arguments les coordonnées du bloc et un tableau de voisins à remplir par la fonction.

Dans la classe '**Grid**', la méthode '**recurciveNeighs()**' recherche un voisin de bloc à droite, à gauche, en haut et en bas. Pour chaque voisin trouvé, elle va s'appeler (récursivité).

Au fur et à mesure des appels de cette méthode, elle remplit le tableau dynamique initialement passé en paramètre qui contiendra alors tous les blocs voisins.

Le choix de prendre un tableau dynamique pour stocker les index de blocs voisins est justifié par le fait que pour chaque bloc, le nombre de voisins peut varier.

Pour éviter les doublons dans les tableaux de voisins de blocs, lors de la recherche, nous testons lors de l'insertion dans le tableau dynamique, si le bloc n'est pas déjà existant.
Ce tableau dynamique des voisins peut ensuite être consulté à partir d'une autre méthode (getter).

Cela permet également à la méthode '**calculateGroups()**' de faire l'inventaire de tous les groupes présents dans la grille qui sont eux-mêmes stockés dans un tableau dynamique : '**tabGroups**' qui est un attribut de la classe 'Grid'.

4.2 Génération aléatoire de la grille

Pour assurer la génération aléatoire de la grille, la méthode '**buildRandomGrid()**' située dans la classe '**Grid**' va dans un premier temps, générer un entier entre : **0** et **2** inclus.
Chaque chiffre dans cet intervalle correspond à un caractère qui se réfère à une couleur : 'R' pour rouge, 'V' pour vert ou 'B' pour bleu.

La couleur choisie par cette valeur aléatoire est ajoutée dans le tableau de caractères qui définit la couleur de chacune des cases de la grille. La méthode génère autant de chiffres aléatoires qu'il y a de cases dans la grille de jeu.

4.3 Déplacements des blocs

Lors de la suppression d'un groupe de blocs, les blocs adjacent peuvent être affectés. Si ils se trouvaient au dessus de la suppression alors ils se doivent de tomber et remplacer l'espace vide.
S'il y a une colonne vide sur leur gauche alors ce sera toute la colonne de droite qui prendra l'espace vide.
Les figures ci-dessous présentent le décalage des blocs qui tombent et ceux qui se décalent à droite.

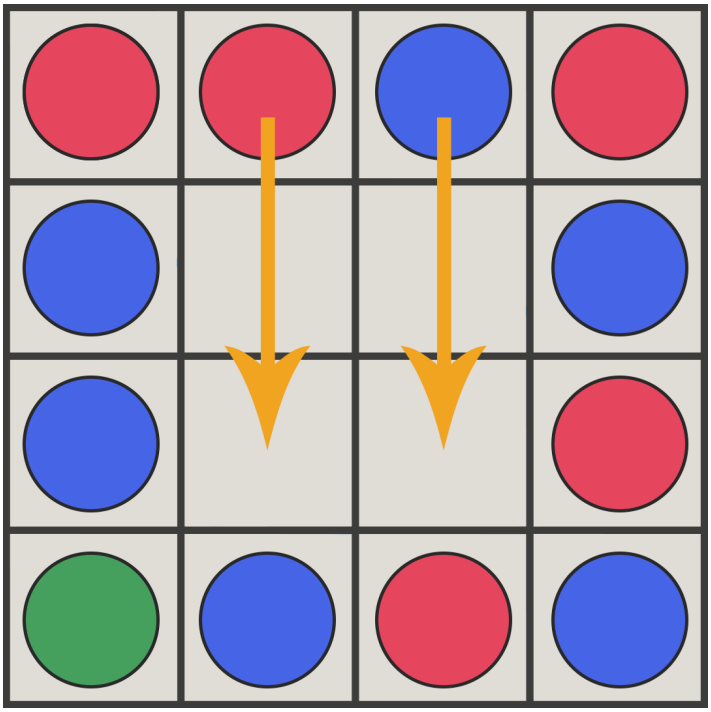


Figure 8: Descente des blocs.

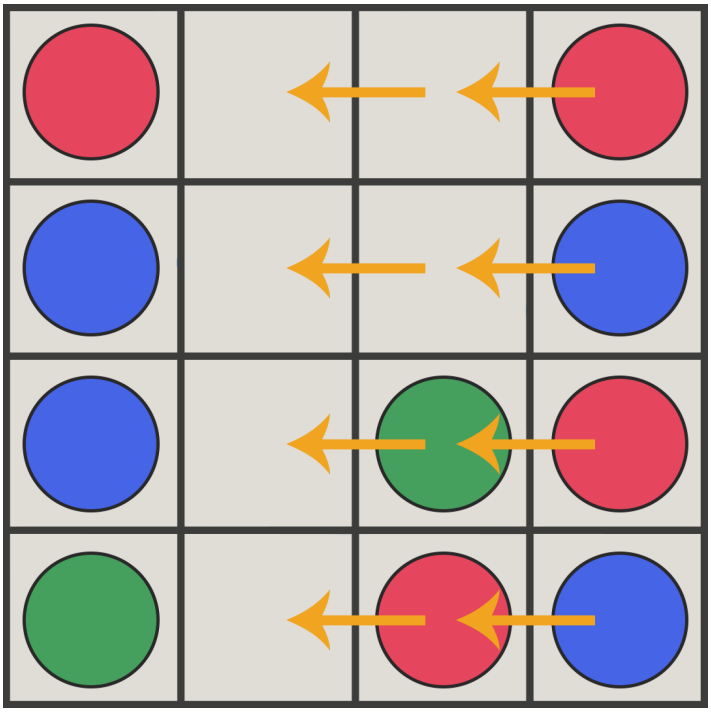


Figure 9: Décalage des colonnes à gauche.

C'est la méthode '**deleteGroup()**' de la classe 'Grid' qui s'occupe de tous les séquencements de décalage de blocs en descente, dans un premier temps et, à gauche dans un second temps. Cette méthode est utilisée lorsque l'utilisateur clique sur un groupe de blocs à supprimer.

4.3.1 Décalage du haut vers le bas

Pour s'occuper de la descente d'un bloc, on récupère dans un premier temps, son index courant. Puis, on vérifie avec la méthode récursive '**recurciveDrops()**' de la classe 'Grid' pour déplacer le bloc tant qu'il n'y a pas de bloc en-dessous.

4.3.2 Décalage de droite à gauche

Pour s'occuper du déplacement de colonnes, et au lieu de vérifier si l'entièreté d'une colonne est vide, on regarde directement la dernière ligne de la grille. Si on y trouve un bloc vide, cela veut dire que toute la colonne est vidée, ceci est vrai car le programme s'occupe en premier de descendre tous les blocs avant de chercher les colonnes vides.

Au niveau de l'algorithme, tout ce passe dans la classe '**Grid**'. Dans un premier temps, la méthode '**getEmptyCols()**' permet de connaître toutes les colonnes vides.

Pour toutes les colonnes vides précédemment trouvées, le programme va utiliser la méthode '**shiftColsLeft()**' qui permet de décaler une colonne à gauche.

Une dernière méthode '**emptyCol()**' permet de vider la colonne précédemment décalée.

5 Conclusion

5.1 Théo Chollet

Le développement de ce jeu, m'a permis d'exercer et de comprendre plus en profondeur la programmation orientée objet, ainsi que découvrir l'étendue des possibilités qu'offre le langage. Pour répartir le travail, l'idée de scinder et structurer les classes selon le modèle, la vue et le contrôleur a été efficace, on a pu facilement se répartir le travail de façon logique. Cette méthode s'est révélée viable durant toute le développement. Au niveau de la répartition du travail je me suis penché plus sur la partie modèle dans un premier temps, puis j'ai pu toucher à la partie vue, et au contrôleur par la suite notamment pour faire des liens entre les classes. Cependant on a pas omis de garder une bonne communication pendant la conception de ce projet pour ne pas reproduire des erreurs passés sur le git. La partie de l'algorithmie sur la détection de groupe n'a pas été si rude, dû à notre prise de recul sur le sujet, on a avant tout réfléchis à comment le résoudre à l'aide de schémas.

5.2 Tomy Da Rocha

Ce projet m'a fait me rendre compte des avantages de la programmation orientée objet, des différences notables par rapport au C. Le travail en groupe a été simplifié, structuré, et beaucoup plus facile à mettre en place par rapport au précédent projet tutoré. J'ai pu utiliser les Travaux Dirigés afin de mettre en place certaines fonctionnalités, tout en étant inventif, comme avec les fenêtres pop-up pour montrer que la sauvegarde a bien été effectuée. Grâce aux forums, et à la documentation, le JAVA offre beaucoup de possibilités et est un langage très vaste. La documentation du JAVA est extrêmement appréciable quand on cherche à faire quelque chose, et c'est tout aussi appréciable de pouvoir à son tour générer de la documentation pour pouvoir partager son code.