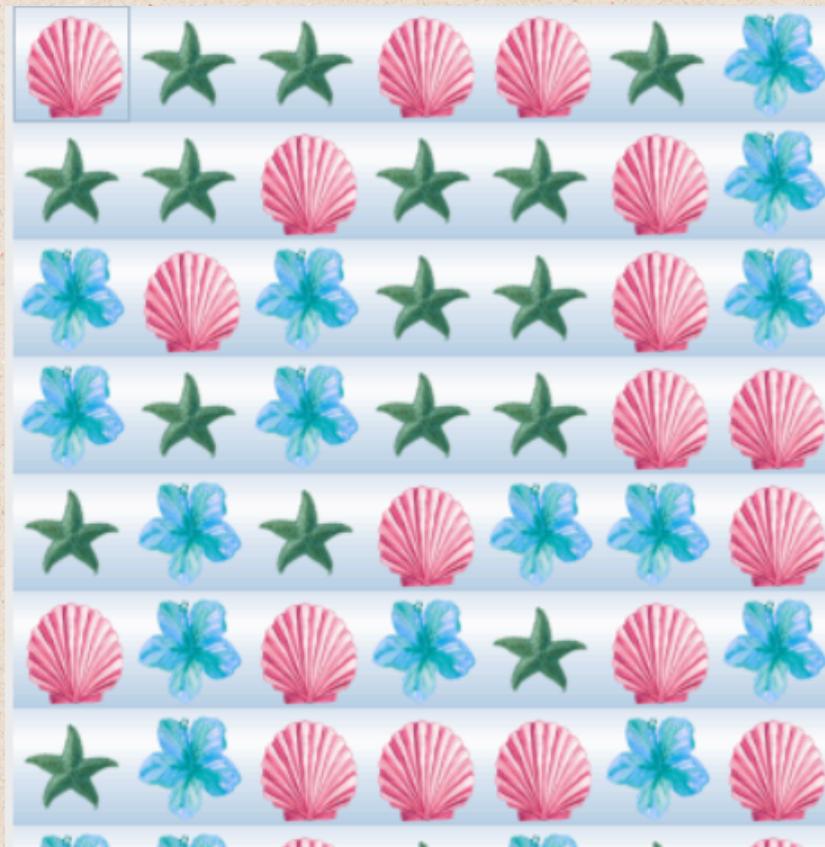




RAPPORT JEU DE SAMEGAME

A RENDRE POUR LE
20 AVRIL 2025



**EL MCHANTEF Sarah
PEIRO TOMAS Maylee**



SOMMAIRE

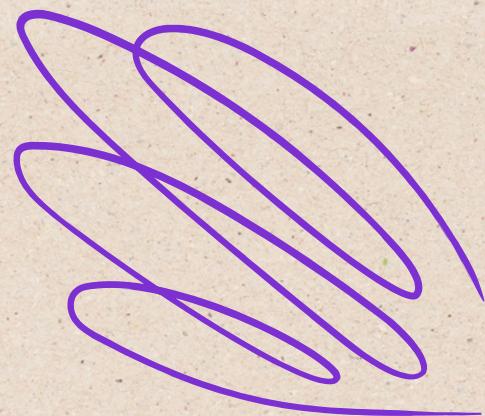
1 Introduction

2 Fonctionnalités du code

3 Structure et diagramme de classe

4 Algorithmes et groupes

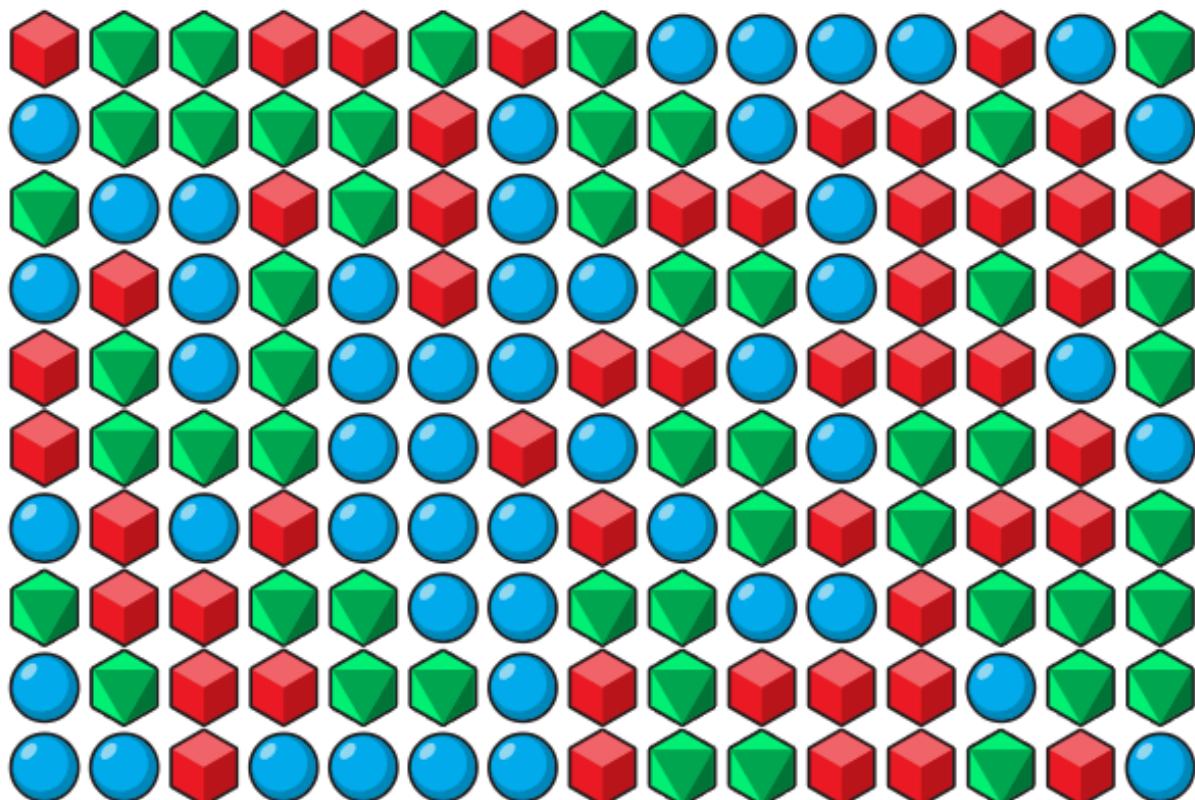
5 Conclusions personnelles



Introduction :

Rappel des règles du jeu :

Le principe du jeu de **SameGame** est de vider une grille constituée de 3 types de blocs de façon méthodique. En effet, les blocs de même type qui sont adjacents constituent un groupe qui peut se supprimer en cliquant dessus. Dès qu'un groupe est supprimé, les blocs des cases du dessus tombent et remplissent les cases vides de sorte à ce qu'il n'y ait pas de vide. En fonction de la taille du groupe supprimé, le score augmente différemment. Plus le groupe de blocs supprimé est grand, plus le score augmente rapidement. Evidemment, le but est d'obtenir le **score le plus élevé possible**.



Dans ce rapport nous décrivons les fonctionnalités, structures, algorithmes, etc de notre code en Java, dans le respect des consignes imposées.

2

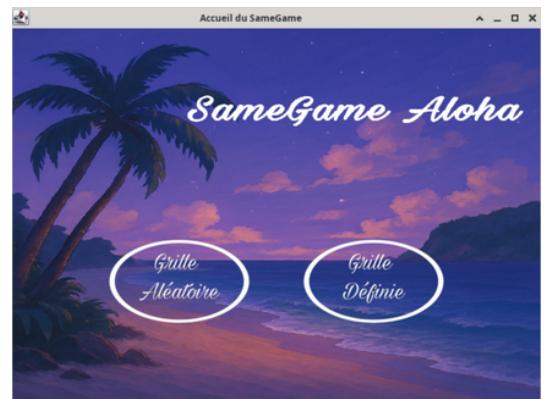
Fonctionnalités du code :

Le jeu de SameGame offre assez de règles pour y tirer de nombreuses fonctionnalités. Nous citons ici les fonctionnalités les plus intéressantes et indispensables à notre jeu de SameGame.

A) Affichage d'une page d'accueil :

La page d'accueil de notre jeu de SameGame permet au joueur de choisir le type de partie qu'il veut lancer : soit il choisit de jouer avec une grille aléatoire, soit il peut lui même choisir la disposition exacte des blocs dans la grille via un **fichier .txt**.

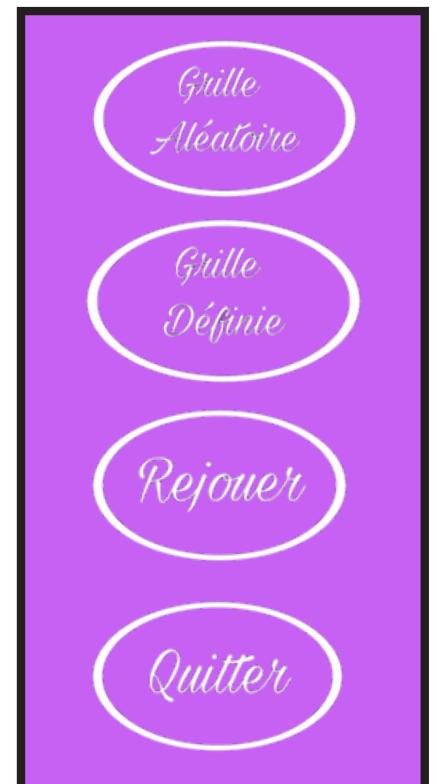
Elle est composée de la fenêtre principale **AccueilSameGame** et de deux boutons dans un JPanel secondaire. Les deux boutons utilisent un écouteur d'évènements pour déclencher l'action souhaitée.



B) Utilisation de boutons images et leurs fonctionnalités:

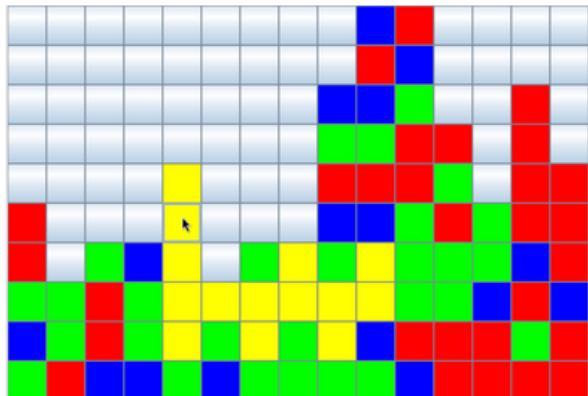
Nous avons créé 4 boutons images avec l'outil de montage Picsart. Ils se servent de l'écouteur d'évènement MouseListener pour rendre la page interactive :

- Le bouton **Grille Aléatoire** : Il sert à générer une grille aléatoirement. Lorsqu'il est cliqué, l'écouteur appelle la méthode **LancerJeuAleatoire**. Ainsi, une instance de **Grille** est créée et la grille est remplie de blocs de façon aléatoire grâce à la classe **GenererGrilleAleatoire**.
- Le bouton **Grille Définie** : l'écouteur d'évènement ouvre le JFileChooser et permet à l'utilisateur de choisir un fichier qui définira la disposition des blocs de sa grille. La méthode **LireGrilleDepuisFichier** est appelée et interprète le fichier texte constitué de R, V et B (respectivement rouge, vert et bleu) pour remplir la grille des images correspondantes grâce à la classe **SetBloc**.
- Le bouton **Rejouer** : Il est placé sur la fenêtre de fin de jeu et sert à revenir à l'écran d'accueil. Le score est remis à zéro et le jeu est réinitialisé. La méthode **AccueilSameGame** est appelée et initialisée par un **new AccueilSameGame()**;
- Le bouton **Quitter** : Il sert à fermer le jeu de SameGame. Il appelle simplement la méthode **System.exit(0)**;

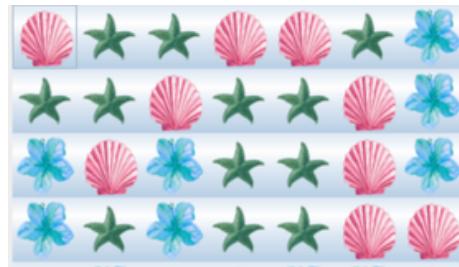


C) Faire apparaître les images dans la grille :

Notre code nous permet de remplir les cases de notre grille de blocs de trois types différents : rouge, vert ou bleu. Initialement ces blocs étaient représentés en coloriant les cases grâce à la méthode **SetBackground**.

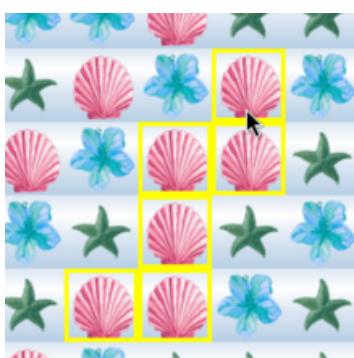


Or, cette façon de remplir la grille n'était que temporaire avant de pouvoir placer des images dans notre grille grâce à la méthode **SetIcon** qui est appelée dans la classe **RaffraichirGrille**.



Notre grille est un **tableau d'entiers** : il est rempli par trois valeurs possibles (1,2 et 3 pour rouge, vert et bleu). Selon l'entier contenu dans le tableau, une image y est placée. Pour rouge, un coquillage rose, pour bleu, une fleur bleue et pour vert, une étoile de mer verte. Pour une valeur différente de 1,2 ou 3, la case n'affiche aucune image. Ces correspondances sont gérées dans la classe **GetCouleur** (la classe gère aussi la taille des images à 50 pixels sur 50 pour qu'elles s'affichent bien). Ainsi, lorsque la grille est mise à jour, on appelle la méthode **SetIcon** pour afficher les images dans la grille.

D) Surligner les groupes :



Comme demandé dans la consigne, il est possible de mettre en valeur les groupes de blocs adjacents de même couleur. Pour cela, il faut d'abord identifier les groupes.

L'écouteur d'événements **MouseMotionListener** détecte le survol de la souris : chaque mouvement de la souris déclenche **MouseMoved()** et la méthode **survolerBloc(x,y)** de la classe **SurvolerBloc** est appelée.

Si la case contient un bloc, elle va regarder si les blocs adjacents contiennent la même valeur (1,2 ou 3). Puis elle fait de même pour la case voisine de même valeur.

La méthode **SurvolerGroupe.survolerGroupe(int ligne, int colonne, int couleur, boolean[] visite)** effectue une recherche récursive : à partir du bloc que la souris survole, elle regarde la case du haut, du bas, de gauche et de droite. Si la case analysée est de même couleur alors elle part de cette case là et effectue la même recherche et ainsi de suite. Il s'agit là d'un **parcours en profondeur**. Le tableau booléen '**visite**' est utilisé pour éviter de repasser sur une case déjà visitée et évite donc les boucles infinies.

Ainsi, la méthode **Raffraichirgrille.raffraichir()** est appelée. Elle parcourt toute la grille et lorsqu'elle détecte un groupe, elle applique une bordure jaune avec **BorderFactory.createLineBorder(Color.YELLOW, 3)**.

E) Afficher un score actuel et final :

Au cours de la partie, le score (initialisé à 0 au début dans la classe **BaseScorePanel**) augmente en fonction des suppressions des groupes de la grille par le joueur. La variable **score** dans la classe **BaseScorePanel** est un entier qui garde en mémoire le score du joueur. Il augmente plus ou moins rapidement selon la taille des groupes que le joueur supprime. Le score est mis à jour grâce à la méthode **ScorePanel.addPoints** qui additionne au fur et à mesure les points accumulés pour former le score.

L'affichage du score est donc mis à jour avec la méthode **updateScore()**. C'est grâce à la classe **ScorePanel** que le score est affiché en haut de la grille. Elle contient un champs **score** pour stocker la valeur et un composant Swing **JLabel** pour l'afficher dans la zone NORTH de la fenêtre. Enfin, le score est contenu dans un panneau **JPanel**.



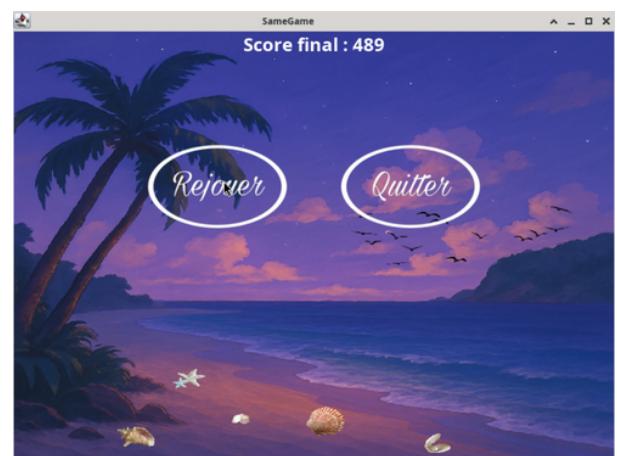
Sur la page de fin, le score final est affiché avec la classe **SuperpositionFinDePartie** à l'aide d'un label. La variable **scoreFinal** récupère le score en fin de partie et permet de l'afficher.

F) Affichage d'une page de fin :

Dans la classe **DetecterEtSupprimerGroupe**, la méthode **verifierSiFinDePartie()** parcourt le tableau pour voir si il reste des groupes supprimables ou non. Si il ne reste aucun groupe, la méthode retourne **true**.

Dès que la méthode retourne **true**, la partie est terminée et la page de fin s'affiche.

Cela se fait avec un **new SuperPositionFinDePartie** dans la classe **DetecterEtSupprimerGroupe**. Il s'agit d'une superposition (**glassPane**) : le fond image recouvre la grille et le panneau de score s'affiche (version fin de partie).



Structure et diagramme de classe

A) Structure et héritage :

Notre code est organisé de la façon la plus claire possible : une classe par fichier. Cela fait donc un total de **38 fichiers .java** avec un **makefile** :

- 1.TailleGroupe.java
 - 2.SurvolerGroupe.java
 - 3.SurvolerBloc.java
 - 4.SupprimerGroupe.java
 - 5.SupprimerBloc.java
 - 6.SuperpositionFinDePartie.java
 - 7.SetBloc.java
 - 8.ScorePanel.java
 - 9.SameGame.java
 - 10.RaffraichirGrille.java
 - 11.PanneauBoutonsFinDePartie.java
 - 12.Main.java
 - 13.LireGrilleDepuisFichier.java
 - 14.LancerJeuAleatoire.java
 - 15.InitGrille.java
 - 16.InitGame.java
 - 17.Grille.java
 - 18.GetCouleur.java
 - 19.GenererGrilleAleatoire.java
 - 20.FinDePartie.java
 - 21.FenetreFinDePartie.java
 - 22.FenetreDeBase.java
 - 23.FaireTomberLesBlocs.java
 - 24.EstGroupeValide.java
 - 25.DetecterEtSupprimerGroupe.java
 - 26.DeplacerBloc.java
 - 27.DecalerColonnes.java
 - 28.ComposantFond.java
 - 29.ChargerGrilleDepuisFichier.java
 - 30.AccueilSameGame.java
 - 31.BoutonSamegame.java
 - 32.BaseScorePanel.java
 - 33.AfficherFinDePartie.java
 - 34.BoutonAleatoire.java
 - 35.BoutonChoix.java
 - 36.BoutonQuitterFin.java
 - 37.BoutonRejouerFin.java
 - 38.GrilleBoutonController.java
- + un makefile

Toutes les classes ont un rôle important comme la **gestion de la grille et de la logique du jeu** (**Grille**, **TailleGroupe**, **SurvolerGroupe**, **DetecterEtSupprimerGroupe**...), l'**aspect graphique** (affichage avec **FenetreDeBase**, **ComposantFond**, **BaseScorePanel**...), l'**interactivité du jeu avec les boutons** par exemple (**BoutonRejouer**, **BoutonAleatoire**, etc) et enfin la fonctionnalité de **lancement du jeu** avec les classes **Main**, **LancerJeuAleatoire** et **ChargerGrilleDepuisFichier**.

Les 4 boutons “**Grille Aléatoire**”, “**Grille Définie**”, “**Rejouer**” et “**Quitter**” sont liés à des contrôleur qui permettent de déclencher différentes actions. Ils sont implémentés via **MouseListener**. Pour les boutons, seulement **MouseClicked()** est utilisé et chaque contrôleur s’occupe de la méthode qui va être appelée ensuite selon le bouton (**LancerJeuAleatoire**, **ChargerGrilleDepuisFichier**, etc).

De plus, chaque classe est accompagnée d'une documentation **Javadoc** inspirée de l'exemple de la consigne. Cela permet au correcteur, mais à nous aussi, de comprendre l'utilité de chacune de nos classes et de ses paramètres.

Enfin, nous avons deux cas de figure pour lesquels nous avons trouvé intéressant de faire de l'**héritage** entre nos classes. En effet, l'héritage permet de rassembler du code commun dans une classe spéciale qui va permettre de ne pas se répéter dans les classes qui en héritent :

-Les classes **FenetreFinDePartie**, **InitGame** et **AccueilSameGame** héritent de la classe **FenetreDeBase**, cela permet d'avoir la même disposition de la fenêtre de jeu. Les trois fenêtres ont donc une apparence cohérente et similaire grâce à **FenetreDeBase**, ce qui rend le jeu uniforme.

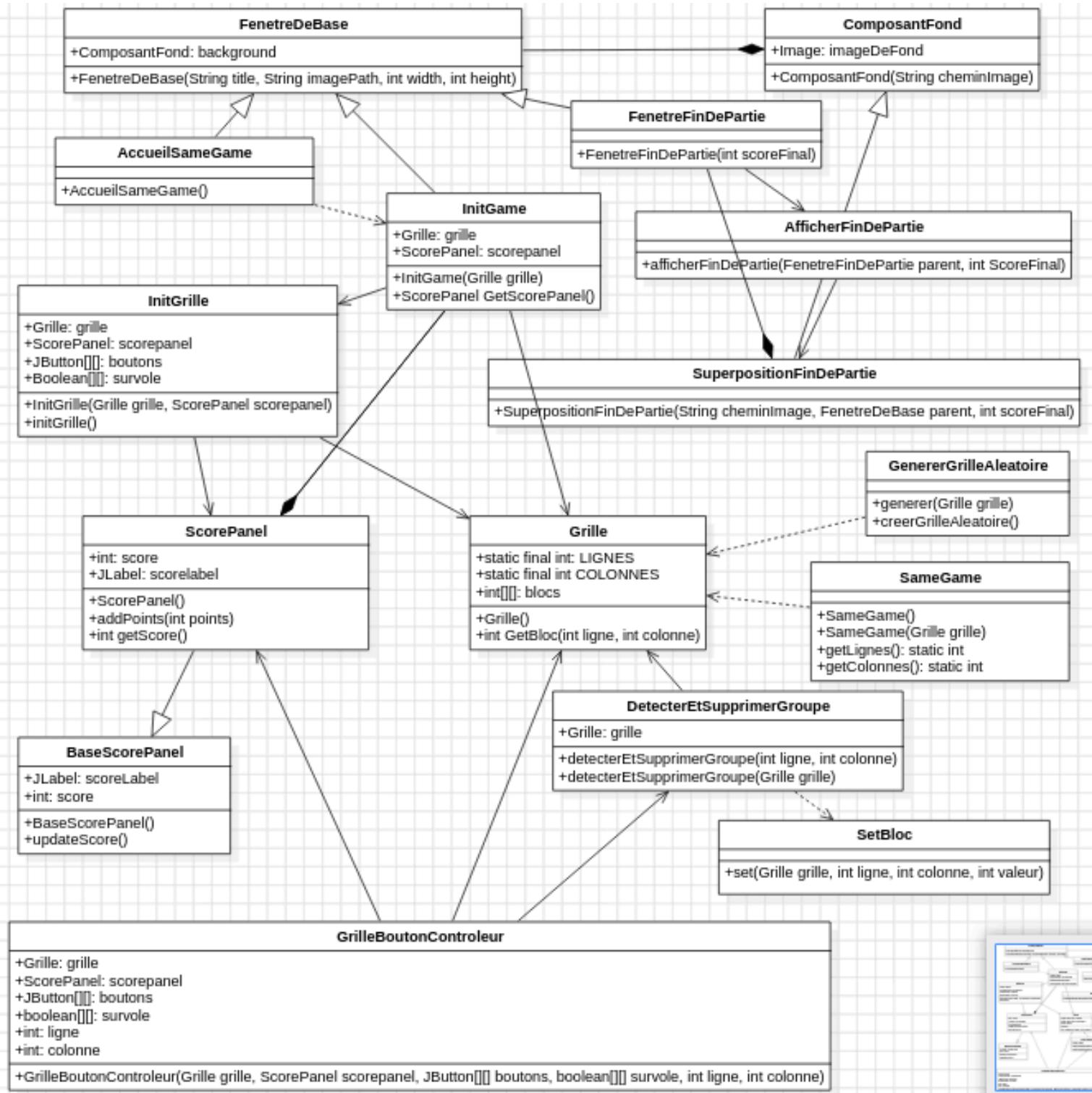
-La classe **ScorePanel** qui hérite de **BaseScorePanel**, ceci permet à ScorePanel d'hériter du style d'affichage du panneau de score de **BaseScorePanel**. Initialement nous avions une classe **ScorePanelFin** qui affichait le score final sur la page de fin en héritant aussi de **BaseScorePanel**, or, nous nous sommes rendues compte que le panneau pour le score de fin ou de début pouvait être contenu dans une même classe.

Nous avons tout de même décidé de garder cette notion d'héritage avec **BaseScorePanel** car il pourrait être intéressant dans le futur. Par exemple, si nous décidons d'afficher un message du style “Reccord à battre : 1450 !” sur la page d'accueil pour motiver l'utilisateur à jouer, nous aurions seulement à ajouter une classe que nous appellerions par exemple **MeilleurScore** qui hériterait de **BaseScorePanel** pour garder le style d'affichage du jeu.

L'héritage nous permettrait aussi d'effectuer des modifications plus rapidement en modifiant seulement la classe qui fait hériter les autres sans avoir à changer plusieurs fichiers. Par exemple, si nous souhaitons modifier la **police d'écriture** du score, il nous suffit de modifier **BaseScorePanel** et non pas toutes les classes affichant un score sur chaque fenêtre.

B) Diagramme de classes :

Voici le diagramme de classe de notre code principal. Etant donné que nous avions énormément de classes, créer un diagramme avec absolument toutes les classes n'aurait pas été intéressant (aussi et surtout illisible). Nous avons donc décidé de le simplifier comme ceci de sorte à pouvoir montrer les relations d'association, de généralisation (héritage) et de composition.



4

Algorithmes et groupes

Dans notre code, notre algorithme sert à **déetecter le nombre de blocs reliés**, vérifier que le groupe contient au moins deux blocs de même type et supprimer le groupe une fois cliqué. Comme dit précédemment, nous nous basons sur un **parcours en profondeur** de la grille. Le joueur survole une case de son choix avec la souris (sans pour autant cliquer) et l'algorithme vérifie si la case n'a pas déjà été visitée par l'algorithme (même si la souris est posée dessus), mais il vérifie aussi si la case est bien contenu dans la grille de départ.

L'algorithme analyse la valeur (entier : 1,2 ou 3) contenue dans le tableau qu'est la grille puis il visite les cases voisines. Commençant par la case du haut, si elle contient la même valeur que la case déjà visitée, alors la case est marquée comme **visitée** et est comptée dans le groupe. On fait la même chose pour cette nouvelle case : on visite ces voisins, si la première case visitée contient la même valeur, alors on continue la recherche à partir de celle-ci. Si elle n'a pas de voisin de la même valeur, alors on retourne en arrière pour trouver des voisins aux cases visitées précédemment, et ainsi de suite.

C'est la méthode **calculerTaille** dans **TailleGroupe** qui se charge de cette recherche en suivant cet algorithme. Cette méthode compte chaque case reliée à celle visitée en premier et fait en sorte de ne pas repasser sur une cellule déjà visitée grâce au tableau **visite**.

Une fois que **calculerTaille** a compté le nombre de blocs dans un groupe, la méthode **EstGroupeValide** vérifie que le groupe contient au moins deux blocs.

Une fois validé, la méthode **supprimerGroupe** utilise **SetBloc** pour mettre la valeur de la case à zéro avec comme argument sa grille, sa ligne, sa colonne et sa valeur (0).



5

Conclusions personnelles

Maylee

Encore une fois, la SAé me permet de progresser beaucoup plus vite que les TP car ils me demandent beaucoup de concentration en cours, ce qui est plus difficile que de travailler chez moi. Durant la réalisation du projet de SameGame, je me suis très rapidement familiarisée avec l'API Java et j'ai intégré de meilleures façons de raisonner comme avec l'algorithme en profondeur. De plus, contrairement au projet Blocus, nous nous y sommes mises dès le départ, ce qui nous a permis de soigner au maximum notre code. Nos problèmes principaux ont été la division de notre code car comme demandé, il fallait diviser un maximum le code source. Cela a pris un peu de temps, Ozvann nous a aidé de bon coeur d'ailleurs. En fait, notre code était relativement divisé puis nous avons entendu le professeur dire que plus les fichiers étaient courts et simples, mieux c'était. Alors on a fait de notre mieux pour avoir des classes concises. Finalement, le projet Java nous aura confronté à des difficultés inattendues de débogage.

Sarah

En réalisant ce projet, j'ai vraiment eu l'occasion de mettre en pratique tout ce qu'on a vu en TP et en TD, mais de façon beaucoup plus poussée et surtout avec plus de liberté. J'ai trouvé que c'étais super intéressant de pouvoir construire quelque chose de complet à partir de tout ce qu'on a appris petit à petit pendant le semestre.

J'ai retrouvé les notions de listeners, de mise en page avec les JPanel, et même les tableaux et les parcours qu'on avait évoqué en mathématiques, mais ici c'était appliqué à un vrai jeu !

Ce que j'ai particulièrement apprécié, c'est que je n'étais pas limitée par un énoncé strict : je pouvais vraiment organiser mon code comme je le voulais, choisir les images, réfléchir à l'interface, etc.



EL MCHANTEF Sarah
PEIRO TOMAS Maylee