1er décembre 2023

# RAPPORT DU PROJET

# JEU D'ASSEMBLAGE DE FORMES

Sélian Arsène
Matthieu Delille
Quentin Dumont
Marilou Preux

## L3 INFORMATIQUE
## UE Méthodes de conception TP4B

UNIVERSITÉ
CAEN
NORMANDIE

http://www.unicaen.fr

# Sommaire

# 1 INTRODUCTION

Le projet évalué de l'unité d'enseignement "Méthodes de conception" est un jeu d'assemblage de formes. Le concept est trivial : le joueur dispose de pièces aléatoirement générées et réparties sur un plateau. Il peut ensuite les déplacer et les faire tourner de sorte à minimiser la surface totale qu'elles prennent sur le plateau. Cette surface est plus précisément le plus petit rectangle englobant toutes les pièces. On peut ainsi attribuer un score au joueur, calculé en fonction de la grandeur de ce rectangle. Assez simple en apparence, ce projet requiert une certaine réflexion vis-à-vis de sa conception, afin d'optimiser sa complexité algorithmique et de conserver une certaine maintenabilité. Nous avons donc été amenés à utiliser différentes méthodes de conception, que nous détaillerons au cours de ce rapport. Nous avions pour consigne de nous baser sur l'architecture Modèle - Vue et Contrôleur ; notre rapport est construit selon ce grand principe. Nous vous souhaitons une bonne lecture.

# 2 MODÈLE

Dans un premier temps, il faut concevoir la base de l'application, autrement dit les objets et les méthodes nécessaires au bon déroulement d'une partie. Le modèle est lui-même divisé en deux parties (packages dans notre application). Il y a d'un côté les pièces à assembler, et de l'autre le plateau sur lequel ces pièces sont posées.

## 2.1 Pièces à assembler

### 2.1.1 Choix déterminants de conception –> Quentin

Plusieurs solutions sont possibles quand il s'agit de créer une pièce capable de tourner sur elle-même. J'ai choisi d'utiliser 4 grilles de booléens. Ces-dernières représentent la pièce dans ses 4 orientations possibles, une case de la grille valant `True` si elle est occupée par la pièce et `False` si elle est vide. Combinée à un entier indiquant l'orientation actuelle, cette représentation permet d'interroger en temps constant la pièce [1], en utilisant la grille correspondante à son orientation. Cette diminution de complexité en temps a néanmoins un coup en espace lors de la création de l'objet, puisqu'il faut tout de même stocker 4 grilles pour une seule pièce. Nous en reparlerons en 2.1.3.

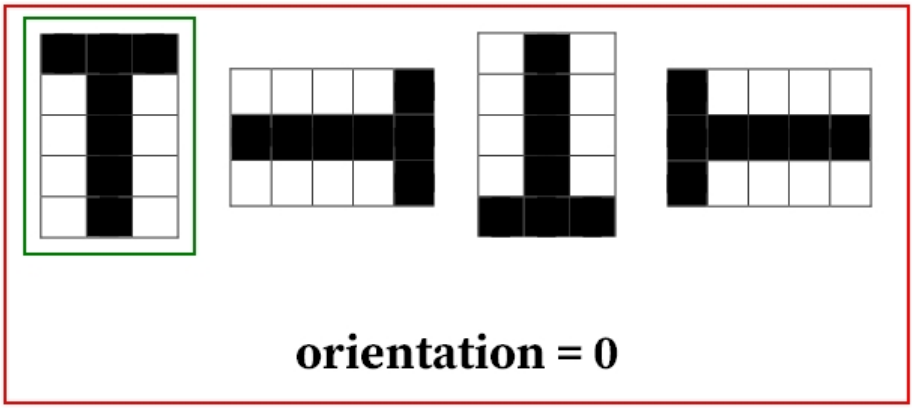


FIGURE 1 – Représentation d'une pièce en mémoire avec **State** –> Quentin

1. méthode estOccupee(i,j)

Pour ce qui est de la rotation, nous avons besoin de définir un point autour duquel la pièce va tourner. Il a été décidé que cet axe de rotation serait le centre de la grille correspondant à l'orientation actuelle de la pièce. Cela signifie que l'axe n'est pas fixe par rapport à la pièce elle-même. Nous perdons légèrement en ergonomie dans certains cas (la rotation peut ne pas être tout à fait intuitive), mais ce choix va grandement nous simplifier la tâche pour la suite, lorsqu'il faudra gérer les pièces sur le plateau.

La façon dont nous avons construit la classe `piecePuzzle` nous permet d'obtenir ses informations en fonction d'une variable d'état `orientation`. En effet, ses accesseurs `getMilieu`, `getHauteur`, `getLargeur` et sa méthode `estOccupee(i,j)` renvoient des informations différentes selon l'orientation de la pièce. Cette conception reprend le principe du **pattern State**.

### 2.1.2 Génération de plusieurs formes –> Sélian

Afin de définir les différentes formes utilisées pour générer une partie de jeu, j'ai choisi de créer 5 formes différentes, ayant toutes un nom basé sur leurs caractéristiques. Un "O", "S", "C", "T" ainsi qu'un "L". Pour ce qui est de leur création, j'ai opté pour une grille à deux dimensions remplie de booléens, avec comme valeurs "true" quand la case est pleine, "false" sinon. J'ai aussi dû réfléchir aux dimensions minimales de chaque pièce, allant de largeur 2, hauteur 2 cases pour les formes "L" et "O", 2 par 3 cases pour la forme "C" et 3 par 2 unités pour les formes "S" et "T".

### 2.1.3 Factorisation et optimisation –> Quentin

À ce stade, nous avons des classes représentant plusieurs formes de pièces. Ces classes sont quasiment identiques à un morceau de code près : celui qui définit les contraintes de génération nécessaires à l'obtention d'une certaine forme. Cet ensemble de pièces, tel qu'il est, n'est pas maintenable. Imaginez devoir changer une des méthodes partagées par toutes les pièces : ce serait le code d'autant de classes qu'il faudrait modifier. Afin de factoriser tout ce code identique, nous avons pensé à utiliser le **pattern Strategy**. En effet, on peut définir un groupe de contraintes de génération comme une stratégie utilisable par n'importe quelle pièce. On peut alors définir autant de stratégies qu'il y a de groupe de contraintes. De ce fait, il n'y a plus besoin de séparer les pièces en différentes classes, nous avons seulement à passer une stratégie au constructeur de `piecePuzzle`, qui utilisera la méthode `generepiece(hauteur,largeur)` de la stratégie pour générer une forme établie par son groupe de contraintes.

Nous avons réussi à factoriser le code, mais nous pouvons encore améliorer cette partie du projet. En effet, à chaque fois qu'une pièce est créée, 4 grilles de mêmes dimensions que la pièce sont calculées et stockées. Au lieu de ne servir qu'à une seule pièce, pourquoi ne pas les utiliser avec d'autres pièces compatibles ? Finalement c'est ce que nous avons fait, en réutilisant les grilles existantes si une pièce de mêmes dimensions et de même forme est créée. Les grilles existantes sont stockées dans un tableau associatif `(clés:dimensions -> valeurs:tableaux)` propre à chaque stratégie de génération de pièce. Ce tableau est donc un attribut `static` des classes héritant de `Strategypiece`. Dans `generepiece(hauteur,largeur)`, avant de créer de nouvelles grilles, on appelle une méthode `recyclage(cachepieces)` qui regarde si une pièce semblable a déjà été créée, et si oui, qui attribue les grilles de cette pièce à la nouvelle. Pour se rendre compte de l'utilité de cette fonctionnalité, j'ai laissé des `print` dans le terminal à chaque fois que 4 tableaux de pièce sont recyclés. Si on génère un grand nombre de pièces, on fait de belles économies en espace !

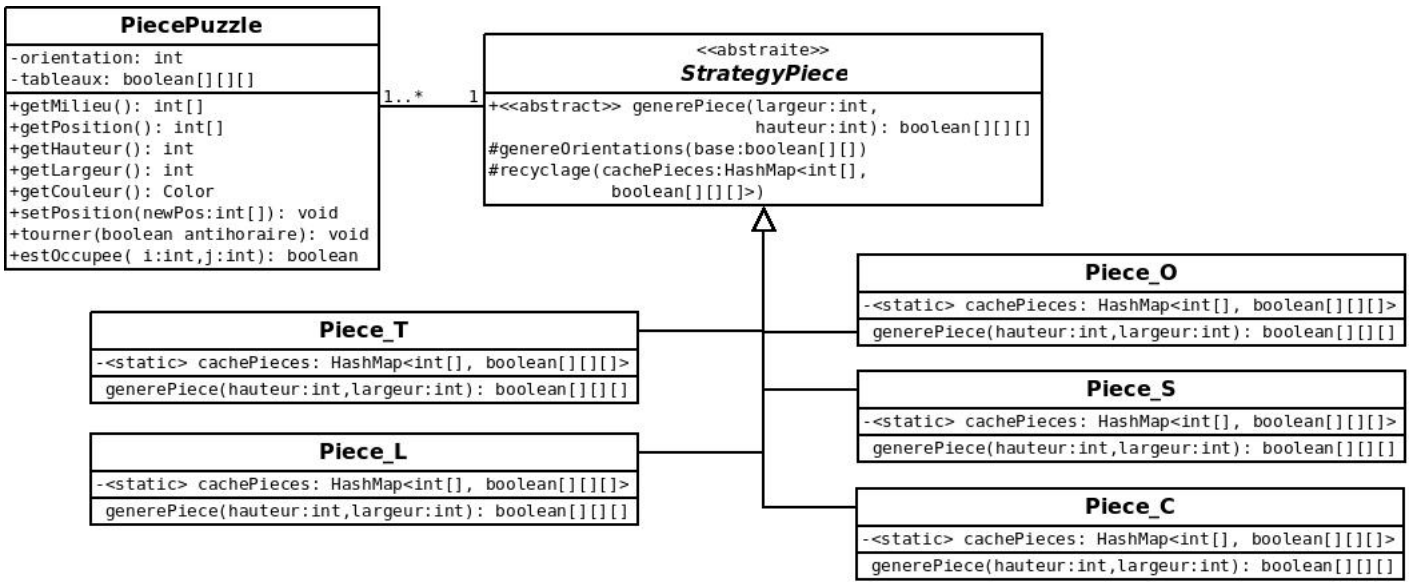


FIGURE 2 – Génération de plusieurs formes et recyclage avec **Strategy** –> Marilou et Quentin

## 2.2 Plateau d'assemblage

### 2.2.1 Représentation du plateau –> Quentin

Nous disposons maintenant de pièces que nous allons devoir intégrer à un plateau de jeu. Pour diminuer le coût du programme en espace, nous avons choisi de ne pas construire une grille de la taille du plateau, qui contiendrait à chaque case (i,j) une référence à une pièce. Par ailleurs, la mise-à-jour de ce type de représentation est souvent laborieuse. Nous avons alors opté pour l'idée suivante : en stockant la position du centre d'une pièce directement dans sa représentation, on peut savoir où toutes ses cases se situent dans le plateau de jeu. Il n'y a donc plus besoin d'une grande grille, et alors notre représentation du plateau nommée `PlateauPuzzle` ne garde en mémoire qu'une liste de pièces, une hauteur et une largeur.

### 2.2.2 Gestion des collisions –> Quentin

Nous y sommes, les pièces sont placées sur le plateau. Comment savoir si un déplacement ou une rotation d'une pièce est possible à tel ou tel endroit ? Pour le savoir, nous procédons de la façon suivante :

- Nous commençons par retirer la pièce concernée de la liste de pièces connues du plateau pour ne pas que la pièce s'empêche elle-même de bouger.
- Ensuite, nous listons toutes les pièces avec lesquelles la grille de notre pièce est en collision. Pour ce faire, nous créons un rectangle avec la classe `Rectangle` de `java.awt`, qui va représenter notre pièce dans sa nouvelle position/orientation. Ensuite nous parcourons notre liste de pièces, et nous ajoutons dans une autre liste ceux dont le `Rectangle` est en collision avec le `Rectangle` de notre pièce.
- Pour finir, il ne reste plus qu'à regarder, pour toutes les pièces en collision, si elles occupent une case occupée par notre pièce.

Le listage des pièces en collision et la vérification pour ces pièces sont des méthodes utilisées au sein d'une méthode englobante `estPositionnable(PiecePuzzle piece, int i, int j)`.

### 2.2.3 Génération d'une partie –> Marilou

Pour générer les pièces, nous avons choisi de faire un générateur de pièces essentiellement basé sur de l'aléatoire. Dans un premier temps, on vient choisir aléatoirement un chiffre entre 0 et 4, afin de sélectionner une pièce entre les 5 possibles : L,O,C,T et S. Ces pièces vont être créées avec une certaine hauteur et largeur prises aléatoirement entre la taille minimale qui est différente selon chaque pièce et la moitié du la taille minimum entre la hauteur et largeur de la grille.

Dans un second temps il a fallu placer les dans la grille. Pour chaque pièce, on tire aléatoirement des coordonnées dans la grille, et si ses coordonnées sont déjà occupées, on retire des nombres aléatoires jusqu'à ce que toutes les pièces soient placées.

## 2.3 Robot –> Matthieu

Afin d'avoir le plus petit rectangle, notre intuition a été de rajouter au fur et à mesure des pièces dans une nouvelle grille en cherchant la position qui donne le plus petit rectangle.

On commence par trier les pièces en fonction de la taille décroissante, la taille étant défini par la largeur fois la hauteur de la pièce. On place ensuite, dans une grille vide, la plus grande pièce au centre de celle-ci. En itérant sur le reste des pièces, on test toutes les positions, avec toutes les rotations possibles, qui collent le plus petit rectangle de l'itération d'avant. On choisit celle qui minimise ce rectangle. On l'ajoute à la grille initialement vide. Cette méthode marche car notre générateur ne peut pas générer un jeu très dense. Un humain peut tout à fait mettre des pièces dans leur bonne orientation et ensuite les mettre à leur place.

## 2.4 Calcul de score –> Quentin

Pour que le projet devienne un véritable jeu, il faut pouvoir évaluer une partie jouée avec un système de score. Avant que le robot créé par Matthieu ne soit opérationnel, j'avais décidé de calculer le score en comparant le rectangle minimal obtenu par le joueur et la somme des cases occupées par les pièces. Ce n'est pas optimal, car dans un système comme celui-ci on ne peut quasiment jamais atteindre le score maximal. Les pièces en forme de "O" par exemple posent problème puisqu'elles laissent un creux parfois impossible à combler. Heureusement, le robot a fini par voir le jour, et j'ai pu revoir le calcul du score pour qu'il soit plus juste. On compare désormais le rectangle minimal obtenu par le robot à celui obtenu par le joueur. Le robot ne trouvant pas la solution optimale à chaque fois (mais souvent quand même), il est possible d'obtenir un score supérieur à 100%. A la fin du projet, j'ai décidé de mettre en valeur ce robot en affichant sa solution proposée à l'écran, en terminal ou en interface selon le mode initialement choisi par l'utilisateur. J'ai cependant manqué de temps pour régler un problème lors de l'affichage de la solution en **terminal** : **il faut demander à terminer la partie une deuxième fois (entrer "2") avant de pouvoir accéder au score**.

# 3 VUE-CONTRÔLEUR

## 3.1 Vue –> Matthieu

Tous les dessins de notre vue sont faits dans la classe PlateauPuzzleVue. A la construction de notre vue, on doit faire le choix de la taille des cases dans notre grille. Chaque case doit être un carré et toutes les grilles doivent prendre le même espace (défini en brut). Par des calculs de rapport entre la hauteur de notre grille dans le jeu et la hauteur du dessins de notre grille (resp. pour la largeur), on obtient une taille de cases.

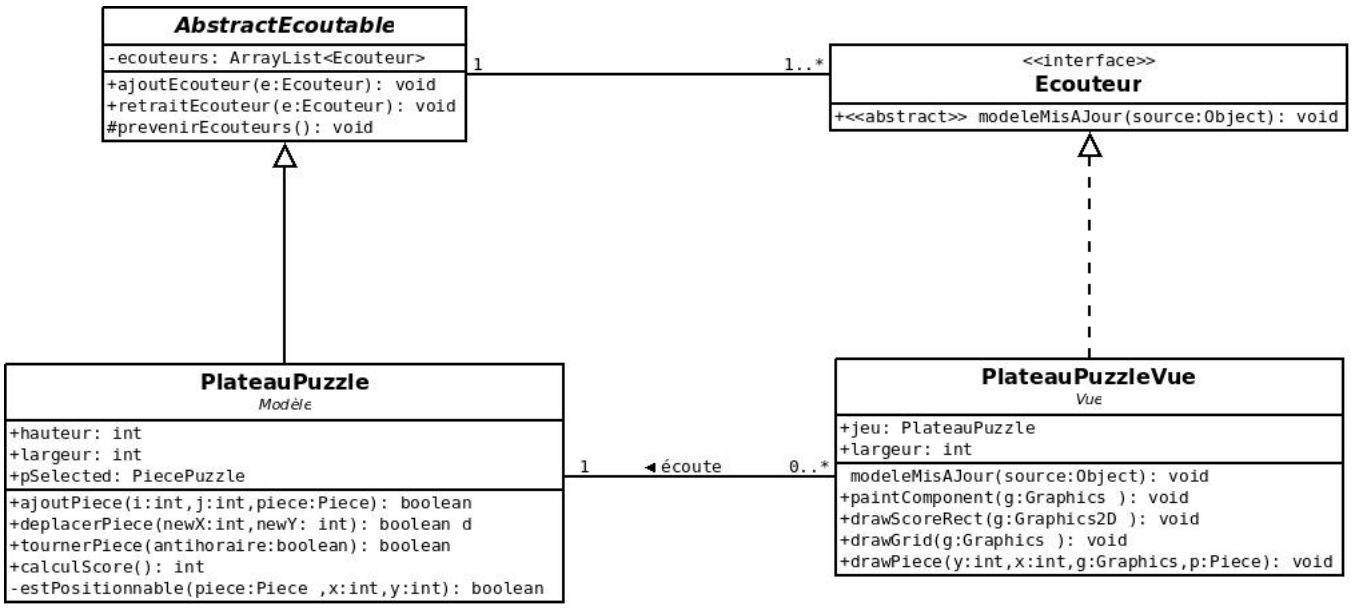


FIGURE 3 – Séparer vue et contrôleur du modèle avec **Observer** –> Marilou et Quentin

Notre vue possède deux états : en jeu ou le jeu est fini. Ces états permettent d'encadrer les actions possibles du contrôleur et de gérer l'affichage.

3 étapes sont nécessaires pour faire apparaître le jeu.

1. Dans un premier temps, on dessine les pièces. Chaque case d'une pièce est soit pleine, dans ce cas on dessine un carré de la couleur de la pièce, soit vide et l'on ne fait rien.

2. On dessine ensuite la pièce en drag (si elle existe). Si une pièce a été sélectionnée et que le clic est enfoncé, on dessine en double cette pièce de la même couleur avec une opacité plus faible. La position de dessin de cette pièce est déterminée dans le contrôleur.

3. On finit par le dessin de la grille en elle-même. Une grille est un ensemble de carrés vides dessinés côte à côte.

L'ajout de bouton permet de donner du choix à l'utilisateur (finir la partie, sauvegarder la configuration, ...). Un bouton est un rectangle dans lequel on a mis du texte. Le contrôleur permet rajouter un effet hover aux boutons.

## 3.2 Contrôleur en interface –> Matthieu

Dans tout ce sous-chapitre, tous les événements sont natifs de swing et awt.

### 3.2.1 Événement de la souris

*On suppose dans cette partie que notre vue est dans l'état de jeu.*

Afin de bouger une pièce, il faut d'abord la sélectionner puis la glisser jusqu'à l'emplacement souhaité en finissant par la déposer.

Lorsque l'on clique dans notre vue, les coordonnées du clic sont transmises à une fonction : mousePressed. Elle vérifie d'abord si les coordonnées sont à l'intérieur de notre grille. Si c'est le cas, on calcule la case à laquelle le clic se situe pour l'envoyer au modèle. Dans le cas où une pièce est sélectionnée, on stocke deux informations : les coordonnées du clic initial et les coordonnées du clic initial par rapport à la grille. La première information nous permettra de calculer le décalage en x et en y de la nouvelle position quand le clic sera relâché. La deuxième sert pour le dessin du drag de la pièce.

Grâce à l'événement mouseDragged de MouseAdapter, on met à jour la position de la pièce sélectionnée.

Enfin, lorsque le clic est relâché, on vérifie si la position du relâchement est dans la grille. Dans ce cas, on calcule le delta en x et en y entre la position initiale du clic et la position du relâchement. On appelle enfin la fonction deplacerpiece du modèle avec les coordonnées initiales de la pièce auxquelles on rajoute le delta.

## 3.3 Contrôleur en terminal –> Quentin

Le contrôleur de la boucle de jeu en terminal est nettement plus simple. Pas d'événements souris ni d'événements clavier à gérer, seulement des entrées d'entier. C'est plutôt du côté de l'ergonomie qu'il y a du travail. En effet il est nécessaire de décomposer les étapes pour déplacer ou tourner une pièce, de façon à ne pas surcharger la sortie du terminal. J'ai donc écrit la boucle en faisant des menus s'appelant les uns les autres en fonction de l'entrée utilisateur.

# 4 TESTS UNITAIRES

## 4.1 Implémentation de tests unitaires avec JUnit –> Sélian avec l'aide de Marilou

Pour tester nos classes, on a utilisé JUnit avec l'éditeur de code Visual Studio Code et ses extensions implémentant JUnit. J'ai créé un package test, situé au même niveau que le package src, et suivant la même arborescence que ce dernier. N'ayant pas remarqué assez tôt cette partie du sujet, nous nous sommes pris très tardivement à l'écriture des tests unitaires. Par manque de temps, nous n'avons donc pas ajouté dans le build les commandes Ant permettant de lancer rapidement ces tests, nous permettant d'écrire puis d'exécuter plusieurs tests, notamment sur des méthodes de la classe `PiecePuzzle`.

# 5 COMPLÉMENT

## 5.1 Menu –> Sélian

Afin de rendre l'expérience du joueur plus agréable, j'ai décidé de créer un menu (`SetupPanel`) regroupant tout ce qui est indispensable afin de créer ou de charger une partie de jeu, dans un terminal ou bien sur une interface. Pour ce qui est de l'identité visuelle, j'ai pour le fond recréé une grille de Jeu Assemblage avec toutes les formes que l'on peut retrouver dans le jeu. J'ai aussi ajouté un logo, représentant aussi des formes du jeu, dont on peut retrouver le pattern sur le fond d'écran du menu. J'ai décidé de fixer la taille de la fenêtre du menu à 1080 pixels de largeur et 720 pixels de hauteur pour que tous les éléments soient bien espacés.

Afin de faciliter la personnalisation des textes sur le menu, j'ai crée une classe implémentant la classe `JLabel`, que j'ai appelé `TextLabel`, afin d'ajouter des méthodes simplifiant la personnalisation des JLabel, telle qu'une méthode permettant de facilement changer le "poids", ou bien pour changer la taille du texte. Cela m'a permis d'éviter de surcharger la classe `SetupPanel`, en regroupant les méthodes nécessaires afin de réduire, changer la taille et le poids du texte.

## 5.2 Sauvegarde

### 5.2.1 Enregistrement et chargement –> Quentin

En Java nous pouvons "écrire" et "lire" des objets dans un fichier grâce aux classes `Object OutputStream` et `ObjectInputStream`. Par souci de concision, j'ai préféré créer une classe regroupant uniquement les informations nécessaires pour sauvegarder une partie, afin de manipuler un objet qui soit le plus léger possible. Pour restaurer une partie (un `PlateauPuzzle`), nous avons besoin de sa liste de pièces telle qu'elle est au début de la partie (avec les positions de base des pièces), de sa hauteur et de sa largeur. J'ai donc créé une classe `Config` qui ne stocke que ces informations essentielles. Ensuite, j'ai créé une classe `Enregistreur` dont les méthodes `enregistrer(String nom, Config partie)` et `restaurer(String nom)` permettent de stocker et lire un objet de classe `Config` par fichier de sauvegarde. Ainsi à la fin d'une partie, nous avons la possibilité de sauvegarder le jeu tel qu'il était au départ, si par exemple nous voulons tenter d'améliorer notre score, mais pas maintenant, une prochaine fois, parce que la vie est trop courte et qu'on a plein de choses à faire, enfin une situation dans ce genre-là quoi. J'ai manqué de temps pour gérer la mise-à-jour automatique d'une sauvegarde avec une mémorisation du meilleur score obtenu.

### 5.2.2 Adaptation du menu –> Sélian

Pour séparer la création et le chargement d'une partie de jeu, j'ai décidé de faire une séparation verticale. Sur la gauche du menu, j'ai mis 3 champs de texte, avec des indications pour les remplir. Le premier champ est la hauteur de la grille de jeu, le second la largeur de la grille, puis le dernier est le nombre de pièces que le joueur veut générer. Sur la droite du menu, j'ai mis en place un bouton ouvrant un sélecteur de fichiers de sauvegarde, permettant au joueur de charger une partie préalablement sauvegardée. Une fois le sélecteur de fichiers ouvert, j'ai choisi de bloquer, puis nous avons directement décidé de ne plus afficher les champs de texte indispensables lors de la création d'une nouvelle partie, évitant ainsi la confusion chez le joueur.

Ensuite, si un fichier de sauvegarde a été sélectionné, le nom de la sauvegarde est affiché en dessous du bouton. Sinon, si le joueur décide d'annuler la sélection d'une sauvegarde, je regarde si un fichier est déjà chargé. Si c'est le cas, il est impossible pour le joueur de créer une nouvelle partie, mais si aucune sauvegarde est chargée, les champs de texte redeviennent

visibles, ce qui permet au joueur de créer sa partie. Si le joueur a chargé une sauvegarde, mais souhaite finalement créer une nouvelle partie, il suffit qu'il appuie sur le bouton situé à droite du nom de la sauvegarde chargée afin de décharger cette dernière et de faire réapparaître les champs de création de partie.

A droite du nom de fichier, j'ai inséré un bouton permettant d'annuler le chargement de la sauvegarde, effaçant le nom du fichier sauvegardé et réactivant les champs de texte permettant de créer une nouvelle partie. J'ai ensuite décidé de garder les deux boutons permettant soit de jouer sur un terminal, soit sur une interface. En implémentant toutes ces fonctionnalités, le menu est devenu un élément important de notre jeu, permettant au joueur de savoir ce qui est possible de faire dès le début et en facilitant la personnalisation d'une nouvelle partie.

# 6 CONCLUSION

Au fil de ce projet, nous avons été amenés à essayer des choses, rebrousser chemin puis tenter d'autres choses, jusqu'à trouver des solutions souvent plus concises et maintenables que nos premiers essais. Ces solutions se trouvent parfois être des Design Pattern bien connus comme Strategy ou State par exemple, ce qui nous a démontré l'utilité de ces méthodes de conception. Nous vous remercions pour le temps passé à la lecture de ce rapport, en espérant que celle-ci fut agréable.