LO21
Projet Hive

Barreddine Rayan, Rabasse Agathe
Theunissen Guillaume, Thomasson Noémie
Semestre A24

Compte rendu final

Table des matières

1 Introduction

Ce projet avait pour but de concevoir et développer une application permettant de jouer au jeu de société *Hive* créé par John Yianni, et édité par Gen Four Two Games. Il s'agit d'un jeu de 2 joueurs dans lequel il est possible de placer ou déplacer des insectes à tour de rôle sur une grille hexagonale pour "emprisonner" le pion Abeille de l'adversaire en l'entourant de pions.

2 Rappel des consignes

L'application développée doit permettre de jouer des parties du jeu, contre un autre joueur, contre une intelligence artificielle ou deux intelligences artificielles entre elles.

L'application doit permettre de jouer la partie, de proposer les actions possibles aux joueurs, de déterminer la fin de la partie et son gagnant.

L'architecture de l'application doit permettre d'intégrer de nouveaux composants sans remettre en cause le code existant.

L'application doit également disposer d'une interface graphique via le framework Qt.

2.1 Répartition de la responsabilité des livrables

- Rapport 1 : Guillaume THEUNISSEN
- Rapport 2 : Rayan BARREDDINE
- Rapport 3 : Noémie THOMASSON
- Rapport 4 : Agathe RABASSE

3 Synthèse de l'application

EXPLIQUER ICI DES TRUC SUR QUOI FAIT QUOI PERMET QUOI

3.1 Jouabilité (Joueur et IA)

L'utilisateur peut choisir entre une partie Joueur VS Joueur, Joueur VS IA, ou IA VS IA. Nous avons donc implémenté une IA qui se différencie dès qu'une entité (joueur ou IA) doit faire un choix dans le cycle de jeu, soit aux moments suivants : choisir entre placer un pion ou déplacer un pion, choisir quel pion placer ou déplacer, choisir la position d'arrivée du pion.

La différence entre un Joueur et l'IA se fait lors de la redéfinition des méthodes virtuelles : *selectPionFromInventory*, *selectPionFromPlateau*, *selectPionFromAll*, et *selectPlace*. Ces fonctions utilisent à un moment donné une fonction renvoyant un choix compris entre un min et max en entrée, *getUserChoice* pour un joueur et *getIAChoice* pour la StandardIA. Nous avons opté pour un random pour le choix de l'IA dans*getIAChoice*.

3.2 Extensions officielles

3.2.a Rappel des règles des extensions

Parmi les 3 extensions officielles : la coccinelle, le moustique et le cloporte, il est possible de jouer avec deux d'entre elles, à savoir la coccinelle et le moustique. Voici, rapidement, leurs règles de déplacements :

- **La coccinelle** se déplace de deux cases sur un autre pion puis d'une dernière case en redescendant.

- **Le moustique** adopte les règles de déplacements des pions directement adjacents à lui. Cependant, s'il est en contact avec seulement des moustiques alors il ne peut pas bouger. De plus, s'il recouvre un pion après avoir utilisé les règles du scarabée, il doit continuer d'adopter le déplacement du scarabée jusqu'à "redescendre" de la ruche. Grâce au développement de notre projet qui respecte le principe SOLID, nous avons pu ajouter facilement ces extensions.

3.3 Interface de paramétrage

Dans le Menu principal, il est possible d'accéder à un menu *Settings* qui permet de faire des réglages pour toutes les prochaines parties avant un nouveau changement des paramètres. Voici les possibilités : ajout d'une extension, modification du nombre de rollbacks.

3.3.a Modifier le nombre de rollback(s)

Il est possible de modifier le nombre de retours en arrière possible. Par défaut, le joueur peut en faire 3. Mais il peut modifier ce nombre en début de partie.

3.3.b Menu Extension

Dans la menue extension, l'utilisateur peut ajouter, retirer une extension ou consulter les règles (dans la console).
Grâce au vecteur *pionEnable* dans **Setting.h**, et à la *enum class PionName* dans **utils.h**, on a à la fois tous les pions possibles en général (*enum class PionName*) et tous les pions possibles pour une utilisation (*pionEnable*). Ainsi, lors de l'ajout de l'extension, on ajoute l'extension au **std : :vector pionEnable** puis on construit les inventaires à partir de ce vecteur. Avant cela, on construisait directement l'inventaire des joueurs à partir de l'enum class. De plus, on vérifie à chaque ajout que l'extension n'est pas déjà ajouté pour éviter les doublons.
Cela conserve la sécurité de l'enum class et l'adaptabilité, car il suffit presque de rajouter le nom de l'extension dans cette enum class (voir l'ajout d'extensions ci-dessous).

3.3.c Charger une partie

Il est possible à n'importe quel moment de sauvegarder la partie pour la continuer plus tard. Au départ, nous pensions qu'il n'était pas possible d'utiliser une librairie JSON. C'est pourquoi l'état du jeu est sauvegardé à la main dans un fichier texte. On est capable de

traduire une instance du jeu (les pions, le plateau, les joueurs...) en un document texte. On peut ensuite recharger ce document texte depuis le menu du jeu afin de reprendre la partie.

Cependant, entre les différentes extensions de pion et d'IA, il est difficile de garantir qu'une sauvegarde reste compatible due à l'ajout et ou la suppression d'extensions entre temps.

3.4 Cycle de jeu

Le cycle de jeu repose sur une alternance de tours entre deux entités (joueurs) : soit deux humains, soit un humain contre une IA, soit deux IA. Chaque entité peut soit placer un pion sur le plateau si cela est permis, déplacer un pion déjà placé sur le plateau (en respectant les règles spécifiques aux pions), soit passé son tour si aucune action légale n'est possible. Le jeu continue jusqu'à ce qu'une condition de victoire soit atteinte (l'encerclement de l'abeille d'un des joueurs). La première phase du cycle de jeu est l'initialisation, où le joueur peut choisir les paramètres de jeu : entités, nombre de rollbacks, activation des extensions.

3.5 Retour en arrière

Notre jeu, tout comme le jeu officiel, dispose d'un système de retour en arrière. À tout moment de la partie, l'utilisateur peut choisir d'annuler son dernier coup afin de le rejouer. Concrètement, c'est le design pattern *Memento* qui est utilisé. Son fonctionnement est expliqué dans *Structure de l'architecture/Module de rollback*

3.6 Mode console et GUI

L'utilisateur a la possibilité de choisir entre jouer en mode console ou avec l'Interface Graphique. Ce choix se fait dès le début dans le main. Si l'utilisateur choisit le mode console, on lance le menu principal qui comporte les Settings ou le début de partie. Si l'utilisateur choisit l'Interface graphique, on lance l'interface avec *app.exec()*.

À noter qu'en console, l'affichage du plateau marche mieux sur Linux, avec les couleurs des pions.

3.7 Ajout de nouvelles extensions

Nous avons fait en sorte de rendre le Projet le plus adaptable possible en vue de possibles ajouts d'extensions ou d'IA. Ainsi, si un développeur souhaite ajouter une extension, il peut suivre les étapes suivantes :

- **Creer le .h et .cpp** : en remplaçant NAME_name par le nom de votre extension **name.h**

```
1 #ifndef LO21_PROJET_HIVE_NAME_H
2 #define LO21_PROJET_HIVE_NAME_H
```

```
3
4           #include"Pion.h"
5           #include"../utils.h"
6
7           class Name: public Pion {
8           private:
9               static std::map<std::string, int> rules;
10
11          public:
12              Name(PionName n, Color c, int id, int i, int j):
                    Pion(n, c, id, i, j){}
13
14              Coordonnees getLegals(GameBoard* gameBoard,
                    Inventory pionOnGameboard, Pion &pion)const
                    override;
15
16              int getRule(std::string label) const override {
                    return rules[label];}
17          };
18
19          #endif //LO21_PROJET_HIVE_NAME_H
```

name.cpp

```
1               #include "../../include/Pions/Name.h"
2           #include "../../include/Struct/GameBoard.h"
3           #include "../../include/Struct/Inventory.h"
4
5           std::map<std::string, int> Name::rules = {
6                   {"nbToCreate", 1}, // Nombre de pion cr e par
                        joueur
7                   {"flyEnable", 0},  // Autoriser ou non la
                        capacit  de voler
8                   {"coverEnable", 0}  // Autoriser ou non la
                        capacit  de recouvrir un pion
9           };
10
11          Coordonnees Abeille::getLegals(GameBoard* gameBoard,
                Inventory pionOnGameboard, Pion &pion) const{
12              // Ecrivez ici la fonction qui impl mente la
                    m canique du pion. Cette fonction dispose de l
                    ' tat  de lu plateau actuel (gameBoard),
13              // de la liste des pions du joueur actuel sur le
                    gameBoard (pionOnGameBoard), et du pion
                    selection .
14              // Cette fonction doit retourner un objet
                    Coordonnees, qui est enfaite une vecteur de
                    coordonn es.
15              // Vous pouvez vous inspirer des autres pions pour
```

```
                coder votre m thode.
16      };
```

- **Emplacement** : placez ces fichiers dans le même dossier que les autres pions. Le .h dans include/Pion et le .cpp dans src/Pion. Ne pas oublier d'ajouter les 2 fichiers crées dans le CMakeList.txt.

- **Autres Modifications** :
  - **utils.h** : Ajoutez le nom de votre extension dans la enum class PionName, et incrémenter l'indice du membre "Last".
  - **utils.cpp** :
    - ❑ Dans getPionNamefromstring rajouter un else if avec le cas de votre pion.
    - ❑ Dans la surcharge d'operateur «, rajoutez une case comme celui-ci :

```
1 case PionName:: votrePion:
2 os << "votrePion";
3 break;
```

  - **ConsoleView.cpp** : ajoutez les règles de votre extension dans la fonction printRulesExtensions().

  -**pionFactoryImp.h ((include/Pion/PionFactoryImp.h))** : ajouter un #include "votrePion".
  -**pionFactoryImp.cpp ((src/Pion/PionFactoryImp.cpp)) :** Dans create et reCreate, ajouter un case correspondant à votre pion comme celui-ci :

```
1 //dans create
2 case PionName::votrePion:
3 return new votrePion(name, c, incId(), i, j);
4
5 //dans recreate
6 case PionName::votrePion:
7 return new votrePion(name, c, uid, i, j);
```

3.8 Ajout de nouvelles IA

- Dans le cas d'une nouvelle IA, il suffit de créer un fichier .h et .cpp avec le nom de votre IA respectant le pattern de class suivant : (remplacer NAMEIA/NameIA par le nom de votre IA)
  dans le .h :

```
1     #ifndef LO21_PROJET_HIVE_NAMEIA_H
2     #define LO21_PROJET_HIVE_NAMEIA_H
```

```
#include "Entity.h"

class NAMEIA: public Entity {
private:

public:
   NAMEIA(EntityType ett, std::string& ps, Color c):
      Entity(ett, ps, c){}
   Pion *selectPionFromInventory() override;
   Pion *selectPionFromPlateau() override;
   Pion *selectPionFromAll() override;
   Coord selectPlace(Coordonnees &p) override;
};

#endif //LO21_PROJET_HIVE_NAMEIA_H
```

dans le .cpp :

```
#include "../../include/Entity/NAMEIA.h"

Pion* NAMEIA::selectPionFromInventory() {
   cout << "\nInventaire de l'IA :\n";
   cout<<playerInventory;
   //**Votre Algorithme  d IA  pour le choix du pion
      placer depuis  l  inventaire  **//
}

Pion * StandardIA::selectPionFromPlateau() {
   cout << "\nLes pions de L'IA sur le plateau :\n";
   cout<<pionOnGameboard;
//**Votre Algorithme  d IA  pour le choix du pion
   joueur sur le plateau**//
}

Pion * StandardIA::selectPionFromAll() {
   const string s = "\nL'IA a le choix entre: 1| Placer
      un nouveau pion, 2| d placer un pion sur le
      plateau :";
   //**Votre Algorithme  d IA  pour le choix entre
      d placer un pion ou placer un nouveau pion**//
};

Coord StandardIA::selectPlace(Coordonnees &p) {
   cout << "\nEmplacements disponibles :\n";
   cout<<p;

  //**Votre Algorithme  d IA  pour le choix du placement
     du pion parmis ceux disponibles**//
}
```

- Placer ensuite le fichier .h et .cpp respectivement dans include/Entity et dans src/Entity

- Dans utils.h : dans la enum class Entity, rajoutez le nom de votre IA

On remarque qu'on a décidé de passer les méthodes : *selectPionFromInventory , selectPionFromPlateau, selectPionFromAll, selectPlace* en virtuelles à redéfinir par la suite et pas seulement le choix (getIAchoice et getUserChoice). En effet, dans le cas où la seule IA serait une IA standard basée sur un random comme la nôtre, la différence dans le code se fait uniquement lors du choix. Ici notre StandardIA utilise la même stratégie de choix pour choisir un pion, choisir un placement ou choisir entre déplacer un pion ou en poser un ; or, dans le cas d'IA plus élaborée, les différents choix ne reposent pas sur la même stratégie.

4 Description de l'architecture

Comme énonce précédemment, le jeu se compose de différents modules. Chaque module est lié avec les autres de manière plus ou moins forte, et permet d'obtenir une structure assez modulaire afin de respecter le premier principe *SOLID* qui est le Single responsibility principle. C'est qu'une classe, une méthode, une fonction, doit avoir une seule et unique raison d'être modifiée. Pour ce faire, il faut donc découper le plus possible nos fonctionnalités.

4.1 Structures de données

Lors de la conception de l'architecture, nous n'avons pas décidé d'implémenter les structures de données utilisées dans des classes sous-jacentes. Cependant, il a été jugé très intéressant de le faire. En effet, externaliser les structures utilisées permet de nous affranchir des mécaniques propres à ces structures. Admettons qu'une personne tierce décide de remplacer la structure par une autre, il ne devra que modifier la classe qui implémente cette structure et non le code client. Par exemple, un *std : :vector* possède une méthode *push_back()* pour y ajouter un élément. On décide de changer et d'utiliser une *std : :stack*. On empile un élément avec la méthode *push()*. Il faut remplacer à chaque endroit l'utilisation de *push_back()* par *push()*, sans compter la modification des types de retour des méthodes et de leurs paramètres.

Au lieu de se retrouver à faire ce travail laborieux, il faut prévoir une structure ou une classe *maListe* par exemple, et implémenter une méthode *void ajouter(int)*. L'unique attribut de cette structure sera la réelle structure utilisée *std : :vector* ou *std : :stack*. Aussi, au lieu d'utiliser cette réelle structure directement dans le code client, on utilise un pointeur sur *maListe*. Il suffit alors lors d'un changement de modifier *maListe* et le contenu de ces méthodes sans avoir besoin de modifier le code de notre application.

Nous avons donc défini 3 structures qui sont regroupées dans le dossier *src/struct* et *include/struct*.

- *Coordonnees* : une liste de coordonnées implémentée par un *std : :vector*
- *Inventory* : une liste de pointeurs de pion implémenté par un *std : :vector*
- *Gameboard* : un plateau de jeu implémenté par un *std : :vector>*

À noter également que *Coord* représente une coordonnée qui se compose de deux entiers i et j. Et *Pion* qui sera défini dans la partie suivante.

Chacune de ces structures possède les méthodes qui nous sont nécessaires. Cela inclut les méthodes dites standards, c'est-à-dire l'ajout, la suppression, les surcharges d'opérateurs... Mais aussi les méthodes propres au jeu. En voici un exemple avec *Gameboard*.

A plusieurs endroits dans le jeu, il est nécessaire de calculer des positions de pions accessibles. Il faut dans la très grande majorité des cas obtenir les voisins d'un pion donné. Cette méthode-ci a pu être implémentée dans la classe **Gameboard** ainsi que plusieurs autres qui en découlent. Par exemple, obtenir les pions voisins de couleur identique ou différente à un pion donné, obtenir les listes des voisins qui sont vides (une liste de coordonnées)... L'ajout de la classe *Gameboard* permet aussi de réduire les méthodes de la classe *Jeu* et ainsi respecter un peu plus le principe 'S' des principes *SOLID*.

4.2 Module Entity

Dans notre projet, la classe *Entity* est une classe abstraite qui joue un rôle central en regroupant les attributs et méthodes communs aux différentes entités qui peuvent participer au jeu, ici un Joueur ou une IA. En établissant cette structure, Entity permet de simplifier le code et de le rendre plus modulaire, car Joueur et IA partagent ainsi une interface commune qui garantit qu'ils possèdent tous les deux les fonctionnalités nécessaires pour participer au jeu.

Le module *Entity* représente les joueurs, qu'ils soient humains ou contrôlés par une intelligence artificielle. Il gère leur état (pions disponibles, pions placés, couleur) et leur interaction avec les autres modules.

4.2.a Gestion des pions et inventaire

Entity contient des fonctions pour gérer les pions dans son inventaire et ceux déjà placés sur le plateau. Ces fonctions incluent : *getInventory* et *getPionOnGameBoard* pour accéder respectivement aux pions non encore placés et à ceux sur le plateau. *addPionOnGameBoard* et *removePionOfInventory* pour mettre à jour l'état de l'entité après un déplacement ou une nouvelle pose.

4.2.b Choix des actions

L'Entity agit comme un intermédiaire entre le joueur et les mécaniques du jeu. Selon sa sous-classe (e.g., Joueur ou StandardIA), elle implémente des comportements spécifiques :

selectPionFromInventory, selectPionFromPlateau : pour choisir un pion à placer ou déplacer, selon les règles et l'état du jeu. selectPlace : pour déterminer la position sur le plateau, où jouer le pion choisi. La méthode peut s'appuyer sur une interface utilisateur (Joueur) ou sur une logique automatisée (StandardIA).

4.2.c Interactions avec le reste de l'application

Jeu : Entity collabore étroitement avec le module *Jeu*, qui appelle ses méthodes pour sélectionner les pions et les positions. *Jeu* orchestre la logique du tour en combinant les décisions prises par chaque entité et en les appliquant via le plateau (GameBoard).

Gameboard : Lorsqu'une entité choisit un pion ou une position, elle s'appuie sur les fonctionnalités du *GameBoard* pour valider ses choix. Par exemple, elle utilise les méthodes comme *getLegals* pour vérifier les déplacements possibles ou *isphyspossible* pour respecter les contraintes de déplacements du jeu (un déplacement doit être physiquement possible.).

Sous-classe Joueur : Cette sous-classe permet une interaction directe avec un utilisateur humain. Elle propose des choix à l'utilisateur et attend des réponses via une interface.

4.3 Module Pion

Dans le jeu Hive, tous les types d'insectes (Fourmis, Sauterelles, Scarabées, Abeilles, Araignées, Coccinelles, Moustiques) partagent des caractéristiques et comportements de base : ils sont tous des pièces de jeu avec des règles spécifiques de déplacement et d'interaction. Pour représenter cette structure en code, nous avons créé une classe mère appelée *Pion* dont héritent tous les insectes du jeu. Pion est en effet une classe abstraite qui définit les propriétés et le comportement de base de chaque pion dans le jeu.

Les méthodes de ce module ainsi que les fonctions de possibilités de déplacement des autres pions ont été développées dans les rapports intermédiaires.

4.3.a Possibilités de déplacement du Moustique (Moustique : :getLegals)

But de la fonction : Cette fonction calcule les déplacements légaux d'un pion "Moustique" sur le plateau de jeu. Le Moustique est une pièce spéciale qui peut imiter les déplacements des pièces adjacentes à lui sur le plateau, mais il y a 2 spécificités :

- Le Moustique ne peut pas bouger s'il est entouré exclusivement par d'autres Moustiques
- Si le Moustique est surélevé, il se comporte comme un Scarabée jusqu'à descendre de la ruche

Logique de l'algorithme :

On commence par récupérer les voisins non vides du moustique grâce à la fonction *getNotEmptyNeighbours* : ce sont eux dont le moustique imite les déplacements. Puis, on parcourt ces voisins et si le Moustique est entouré uniquement par d'autres Moustiques, il ne peut pas bouger, et une liste vide est retournée.

Si le Moustique est sur un autre pion, donc que son niveau, déterminé par *pion.getLevel()*, est différent de 0, il agit comme un Scarabée. Pour récupérer les déplacements légaux d'un Scarabée, la fonction cherche dans pionOnGameboard un pion de type Scarabée et utilise

ses déplacements en appelant la fonction *getlegals* de ce pion sur le Moustique. Puis on retourne ces déplacements.

Sinon, si le moustique est au sol, il imite les déplacements des pions adjacents : en effet, pour chaque pièce adjacente, on appelle sa méthode *getLegals* sur le Moustique pour récupérer ses déplacements possibles. Ces déplacements sont ajoutés à la liste *legalMoves*, tout en vérifiant qu'il n'y a pas de doublons. Enfin, on renvoie la liste *legalMoves*.

4.3.b Possibilités de déplacement de la Coccinelle (Coccinelle : :getLegals)

But de la fonction : Cette fonction calcule les déplacements légaux d'un pion "Coccinelle". La spécificité de la Coccinelle est qu'elle doit effectuer exactement trois déplacements consécutifs :

- Elle doit monter sur une pièce adjacente (premier déplacement vers une case occupée).
- Elle doit se déplacer sur une autre pièce adjacente à sa nouvelle position (deuxième déplacement sur une autre case occupée).
- Elle doit redescendre sur une case vide adjacente (troisième déplacement vers une case vide).

Logique de l'algorithme :
Les cases adjacentes occupées par des pièces (les voisins non vides) sont calculées à l'aide de *gameBoard->getNotEmptyNeighbours(currentPos)*, car chaque voisin non vide devient une position possible pour le premier déplacement de la coccinelle.

Depuis chaque position atteinte au premier déplacement *firstMove*, on calcule les cases adjacentes occupées à cette nouvelle position avec *gameBoard->getNotEmptyNeighbours(firstMove)*, mais pour éviter de revenir immédiatement à la position initiale, la case d'origine est retirée de cette liste avec la fonction *removeCoord(currentPos)*.

Enfin, depuis chaque position atteinte au deuxième déplacement, on calcule les cases adjacentes **vides** avec *gameBoard->getEmptyNeighbours(secondMove)*. Ces positions sont ajoutées à la liste *legalMoves*, en évitant les doublons, puis on la renvoie.

4.4 Module de jeu

Le module *Jeu* est au cœur de la mécanique du jeu. Il orchestre la boucle principale du jeu et gère les interactions entre les entités, le plateau (GameBoard) et les pions. Ce module contient plusieurs fonctions clés, comme *play*, qui permet de coordonner le tour de chaque joueur, *selectPionToPlay*, pour sélectionner un pion selon les règles en vigueur, et *appliedMove*, qui effectue les mouvements sur le plateau. En parallèle, *Jeu* interagit avec des modules comme *Entity* pour obtenir les choix des joueurs, et avec le *GameBoard* pour vérifier la validité des mouvements (par exemple la connexité via *isHiveConnected*) et pour mettre à jour l'état du plateau après chaque action.

4.5 Module de contrôle

Le module *Controleur* agit comme l'orchestrateur central de l'application. Il est responsable de la coordination entre les différents modules du jeu, en facilitant la communication entre la vue (*ConsoleView*), le gestionnaire de jeu (*Jeu*), et les entités (joueurs ou IA).

Le Controleur est implémenté en tant que singleton, garantissant qu'une seule instance de cette classe existe à tout moment. Cela est réalisé via la méthode getInstance et la gestion de l'instance unique.

4.6 Module de rollback

Le retour en arrière, ou aussi "rollback" est rendu possible grâce à l'utilisation de design pattern *Memento*. Ce design permet de faire des captures instantanées (snapshot) des objets voulus, et de les restaurer au moment voulu. Concrètement, ce module s'implémente avec 2 classes, *gameStateMemento* et *cateTaker*.

La première classe *gameStateMemento* permet de stocker les états des objets que l'on veut sauvegarder. L'appel de son constructeur réalise pour chacun de ses objets en paramètre l'appel à la méthode *clone()* de cet objet. Dans notre cas, on souhaite sauvegarder les 2 listes de type *Inventory* contenues dans une *Entity*. Etant donné qu'il y a 2 *Entity* dans une partie, il y a donc 4 listes. De plus, une sauvegarde partielle de *Jeu* est faite. En effet, *Jeu* ne contient que le *gameBoard* qui n'est rien de plus q'un moyen de représenter les pions, et un entier pour stocker le nombre de tours. On a donc choisi de sauvegarder seulement 3 entiers. Le nombre de tours, et la taille $M * N$ du *gameBoard*. Pour ce qui est dans liste de type *Inventory*, on réalise une copie profonde de chacune des listes (et donc de tous les pions de la partie) afin que les pions restent modifiables dans la suite du jeu sans modifier les sauvegardes précédentes.

La seconde classe *careTaker* stocke les objets *gameStateMemento* dans un vecteur. Celui-ci est personnalisé par rapport au design pattern original. En effet, on restreint sa taille au paramètre qui définit le nombre de retours en arrière possibles, défini par l'utilisateur en début de partie. Cette classe a une méthode *save()* et *restaure()* qui permettent de sauvegarder l'état du jeu en créant un objet *gameStateMemento* et en le stockant dans le vecteur. Et aussi de dépiler la dernière sauvegarde ajoutée afin de la retourner.

Une sauvegarde est effectuée après chaque coup joué. Quand la taille limite du nombre de retours en arrière possibles a été atteinte, alors la plus ancienne sauvegarde est supprimée pour laisser la place à la nouvelle. Lors du chargement d'une sauvegarde, le jeu est capable de réinitialiser un *gameBoard* à la bonne taille. Aussi, chacune des 2 listes des 2 *Entity* se font remplacer de manière profonde par celle dans la sauvegarde.

Dans le cas où un joueur joue contre une intelligence artificielle, le retour en arrière annule également le coup de l'IA. Nous avons pris la décision d'autoriser l'IA par défaut d'utiliser le retour en arrière.

4.7 Système de coordonnées

Dans un hexagone, chaque cellule a 6 voisins, que nous avons représentés sous forme de quadrillage pour privilégier le nombre de calculs par rapport au stockage.

Ce système permettra de localiser chaque pion du jeu. Si nous avons besoin de chercher tous les voisins d'un pion, cela réduit le nombre de calculs dans la recherche grâce à la représentation dans un tableau.

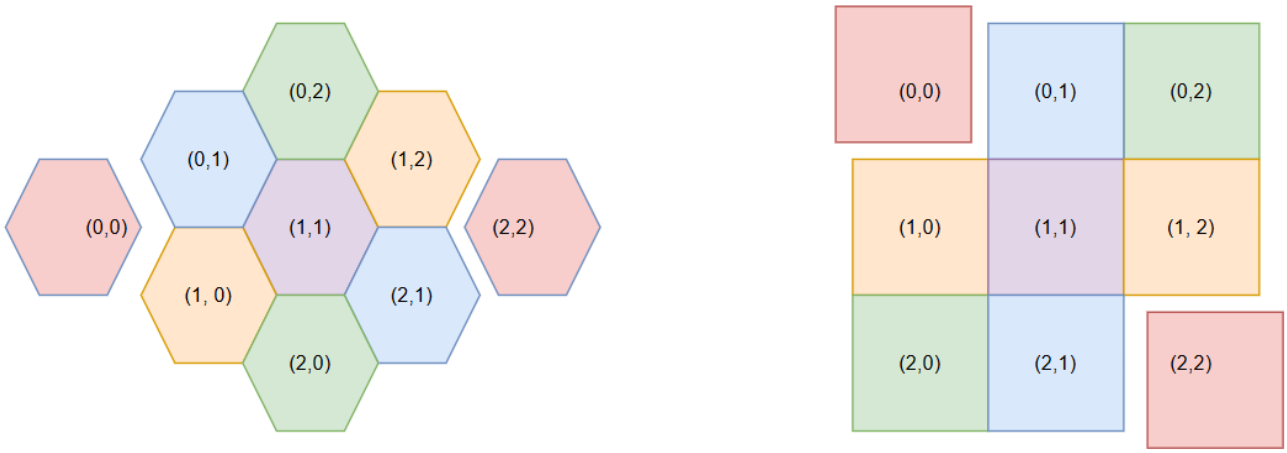


FIGURE 1 – Système de coordonnées utilisé

5 Description du GUI

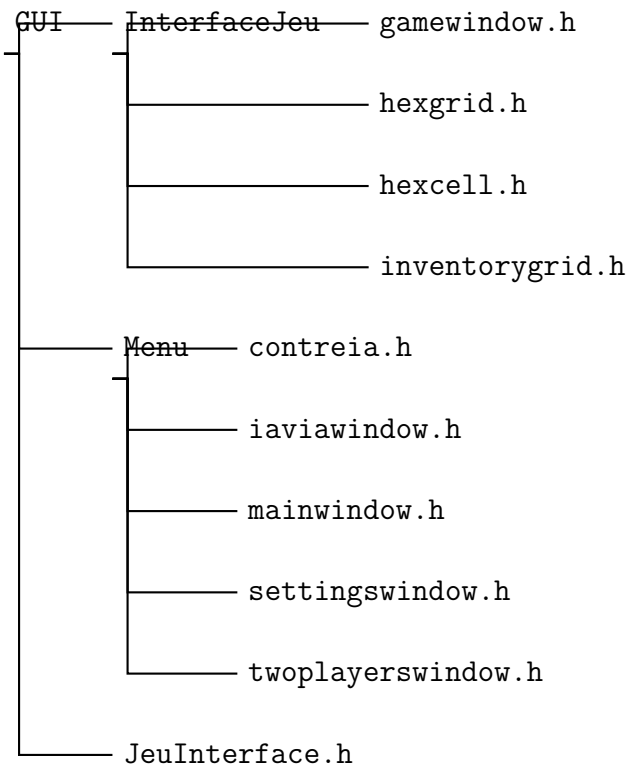
5.1 Structure de l'interface graphique

L'interface graphique est organisée en différents dossiers, chacun ayant un rôle bien défini. Le dossier **InterfaceJeu** contient les éléments liés au jeu lui-même. Par exemple, `gamewindow.h` gère la fenêtre principale du jeu, tandis que `hexgrid.h` et `hexcell.h` s'occupent de la gestion de la grille hexagonale et des cellules qui la composent. Ces fichiers permettent d'afficher la grille et de gérer les interactions avec les cellules. On trouve également `inventorygrid.h`, qui sert à afficher et organiser les inventaires des joueurs de manière visuelle.

Le dossier **Menu** regroupe les éléments liés à la navigation et aux options du jeu. Par exemple, `mainwindow.h` représente la fenêtre principale d'accueil, tandis que `settingswindow.h` permet de configurer des éléments du jeu, comme charger ou sauvegarder une partie. D'autres fichiers comme `iaviawindow.h` et `twoplayerswindow.h` gèrent les sous-menus pour configurer les modes de jeu, qu'il s'agisse de jouer contre un autre joueur, contre une IA, ou de lancer une partie IA contre IA. Ces fichiers travaillent ensemble pour garantir une interface fluide et facile à utiliser.

Enfin, `JeuInterface.h` joue un rôle essentiel en faisant le lien entre la logique du jeu (controleur) et l'interface graphique affichée à l'écran. Cette organisation permet de bien séparer les responsabilités, ce qui facilite les modifications et l'ajout de nouvelles fonctionnalités.

Voici la structure des fichiers de l'interface graphique :



5.2 La partie Menu

Lorsque l'application est lancée, l'utilisateur accède à un menu principal. Ce menu est généré par les fichiers `mainwindow.cpp` et `mainwindow.h`. Il contient quatre boutons, dont trois permettent de configurer différents modes de jeu : *"Jouer avec un autre Joueur"*, *"Jouer contre l'IA"*, et *"IA contre IA"*.

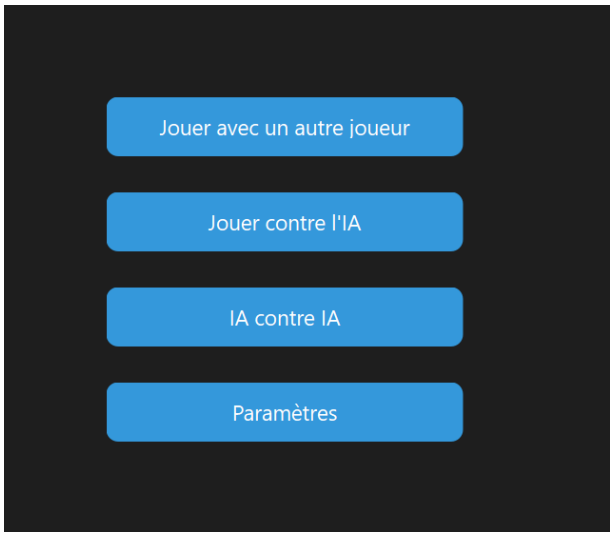


FIGURE 2 – Menu principal

Ces boutons mènent à des sous-menus spécifiques. Ces sous-menus sont générés par les

fichiers `twoplayerswindow.cpp (.h)`, `contreia.cpp (.h)`, et `iaviawindow.cpp (.h)`. Ils permettent d lancer une partie de HIVE avec l'interface graphique. Si l'utilisateur choisit de jouer contre l'IA, il ne pourra jouer qu'avec une IA effectuant des choix aléatoires.

Ces menus sont contruits simplement avec des objets des bibliothèques Qt comme QLabel, QPushButton, QComboBox et des QCheckBox. Tout ceci agencé dans des Frame et des Layout pour avec un rendu clair.

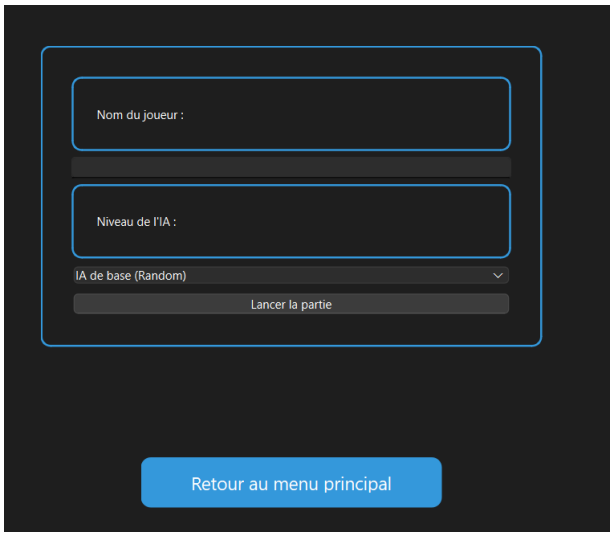


FIGURE 3 – Jouer contre l'IA

Settings

Pour le menu settings, nous devons configurer plusieurs choses : le rollback, le chargement de la partie et l'activations des extensions. Pour le rollback, nous avons naturellement utilisé un slider, pour le chargement de la partie nous avons utiliser une fenêtre de dialogue pour récupérer le nom du fichier de sauvegarde à charger et pour l'activation des extensions une ticking box.

Cependant, nous avons rencontré un problème pour transmettre les informations dans un objet Settings lorsque l'on lance la partie. Comme nous utilisons un singleton, nous avons donc choisi de créer un fichier TXT pour enregistrer les paramètres et lors du démarrage de la partie, nous chargerons les paramètres enregistrés dans settings.txt.

Attention, veillez à bien enregistrer les paramètres avant de lancer la partie.

5.3 La Partie Jeu

5.3.a GameWindow

La fenêtre de jeu est composée de plusieurs parties : le quadrillage hexagonal, les inventaires des joueurs et les boutons pour interagir avec le jeu. Tous ces éléments sont agencés de manière harmonieuse grâce à l'utilisation des *layouts* et des *frames*, permettant une organisation fluide et responsive de l'interface graphique.

Le quadrillage hexagonal, qui représente le plateau de jeu, est généré dynamiquement en fonction de la taille du `gameboard`. Cette génération repose sur la classe `HexGrid`

FIGURE 4 – Paramètres

pour organiser l'ensemble des cellules, et sur la classe `HexCell`, qui définit chaque cellule hexagonale. Chaque cellule est positionnée de manière précise et suit les dimensions du plateau, permettant ainsi une adaptation automatique de l'affichage à la taille du plateau spécifiée par le jeu.

Les inventaires des joueurs sont gérés par la classe `InventoryGrid`. Ces inventaires sont synchronisés en temps réel avec l'état du jeu, ce qui signifie que toute modification dans le jeu, comme le placement ou le déplacement d'un pion, se reflète instantanément dans l'interface. Chaque joueur dispose ainsi d'une vue claire et à jour de ses pions restants, rendant l'expérience utilisateur plus intuitive.

Enfin, des boutons d'interaction sont disponibles pour permettre au joueur d'effectuer des actions spécifiques, comme valider un mouvement, annuler une action ou quitter la partie. Ces boutons sont positionnés stratégiquement autour de la fenêtre de jeu pour une accessibilité optimale.

L'ensemble de ces composants s'intègre grâce à l'utilisation des *layouts*. Cette approche garantit que tous les éléments restent alignés et correctement dimensionnés, quelles que soient les dimensions de la fenêtre ou les éventuelles modifications de l'interface. Ainsi, la fenêtre de jeu offre une expérience utilisateur fluide et adaptée aux besoins du jeu HIVE.

JeuInterface

Comme la manière de gérer les cycles de jeu a changé, une nouvelle classe appelée `JeuInterface` a été implémentée. Son rôle principal est de servir d'intermédiaire entre l'interface graphique et le contrôleur du jeu. Cette classe joue un rôle essentiel dans la communication entre la logique du jeu et les éléments affichés à l'écran.

La classe `JeuInterface` est initialisée dans le fichier `gamewindow.cpp`. Elle permet de déclencher des signaux précis en réponse aux actions effectuées par l'utilisateur, comme cliquer sur une cellule hexagonale, sélectionner un pion, valider un coup ou mettre à jour l'état du jeu. Ces signaux sont connectés à différentes méthodes pour effectuer des actions spécifiques, telles que jouer un mouvement, mettre en surbrillance les coups possibles, ou encore synchroniser l'état du plateau avec les inventaires.

En pratique, chaque clic ou action sur l'interface graphique déclenche un signal dans `JeuInterface`, qui transmet les informations nécessaires au contrôleur pour appliquer les

changements dans le modèle de jeu. Par exemple, lorsque l'utilisateur sélectionne un pion ou clique sur une case hexagonale pour le déplacer, un signal est envoyé pour valider la sélection, vérifier la légalité du mouvement, et mettre à jour l'affichage en conséquence.

Cette approche modulaire garantit une séparation claire entre l'interface graphique et la logique du jeu, facilitant ainsi la maintenance et l'extensibilité du projet. Grâce à `JeuInterface`, les interactions entre l'utilisateur et le jeu sont fluides et intuitives, tout en restant cohérentes avec les règles et la logique sous-jacente.

La fonction `updateAll` est une méthode clé utilisée pour synchroniser l'interface graphique avec l'état actuel du jeu. Elle est appelée chaque fois qu'un changement majeur survient, comme un déplacement de pion, une modification de l'inventaire, ou un changement de tour. Son rôle est de garantir que tous les éléments de l'interface (grille hexagonale, inventaires des joueurs, boutons d'interaction, etc.) reflètent fidèlement l'état du jeu. En centralisant cette mise à jour, `updateAll` simplifie la gestion des interactions et minimise les risques d'incohérences visuelles ou logiques entre l'affichage et les données du jeu. Cette méthode est essentielle pour offrir une expérience utilisateur fluide et cohérente.

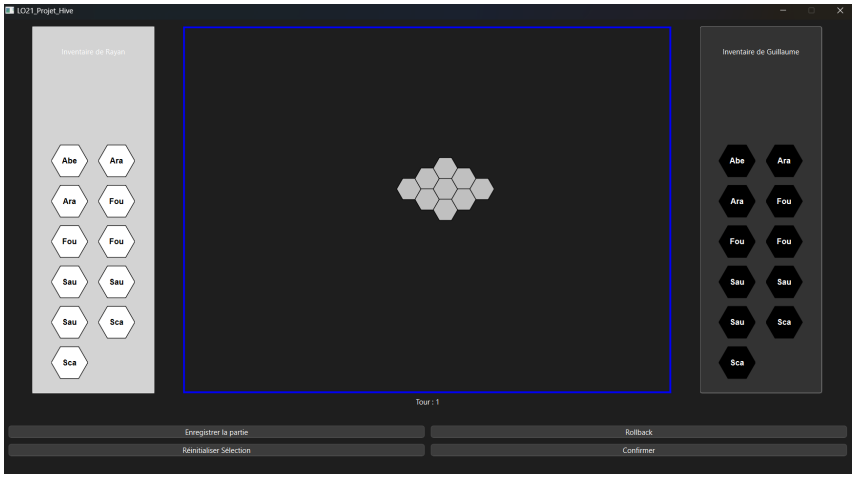


FIGURE 5 – Fenêtre de jeu

5.4 Les fonctionnalités

L'ensemble des règles du jeu, comme la connexité, la validation des coups possibles (`isPhyPossible`), et d'autres contraintes spécifiques, sont correctement intégrées dans le programme. Ces règles assurent que chaque action effectuée par le joueur est conforme aux mécaniques du jeu. En cas de tentative de coup illégal, des messages d'avertissement sont levés, indiquant clairement à l'utilisateur la raison pour laquelle l'action est refusée. Cela garantit une expérience utilisateur intuitive tout en respectant les règles strictes du jeu HIVE.

Une autre fonctionnalité importante intégrée est le **rollback**, qui permet aux joueurs de revenir en arrière dans le déroulement du jeu. Cette fonctionnalité repose sur des appels aux fonctions du contrôleur pour restaurer un état précédent, en utilisant un historique des mouvements. Cela permet une flexibilité dans les parties, notamment pour corriger des erreurs ou tester différentes stratégies.

De plus, le programme offre la possibilité de sauvegarder (`save`) et de charger (`load`) une partie à tout moment. Les fichiers de sauvegarde sont enregistrés dans le répertoire relatif `../saves/`, et chaque sauvegarde est associée à un fichier horodaté unique. Lorsqu'une partie est chargée, l'état du jeu est restauré de manière complète, incluant le plateau, les inventaires, et le tour en cours. Ces fonctionnalités rendent le programme robuste et adapté à des sessions de jeu prolongées ou interrompues.

Le programme intègre également un système d'extensions qui permet d'ajouter de nouvelles fonctionnalités et mécaniques au jeu de manière flexible. Elles sont gérées via des paramètres activables dans le contrôleur. Par exemple, les inventaires des joueurs sont adaptés dynamiquement en fonction des extensions activées, et les nouvelles règles sont intégrées de manière transparente dans les vérifications des coups valides. Grâce à cette approche, le programme reste évolutif et capable de s'adapter à de futures améliorations ou personnalisations.

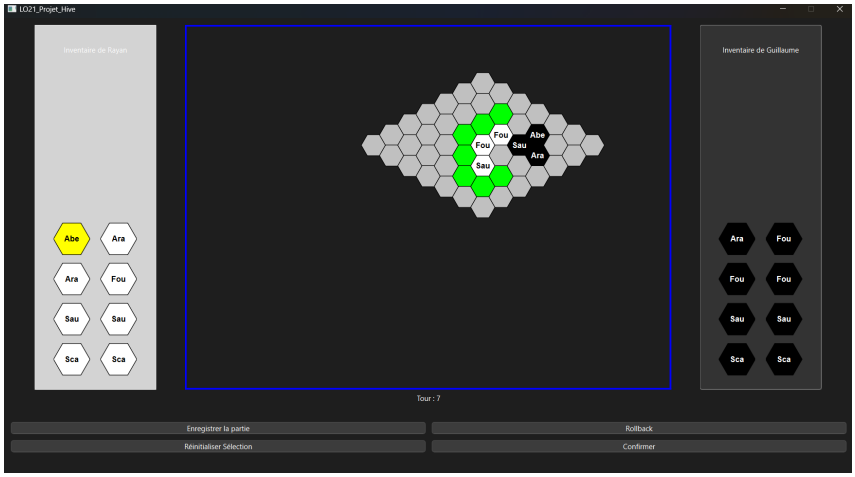


FIGURE 6 – Fenêtre de jeu

5.5 Expérience utilisateur

L'expérience utilisateur a été au centre du développement de l'interface graphique, avec une attention particulière portée aux interactions intuitives et fluides. Lorsqu'un joueur clique sur un pion, ce dernier est automatiquement sélectionné, et la grille met en évidence (*highlight*) les coups légaux possibles grâce à un système de surbrillance visuelle. Cela aide le joueur à mieux comprendre ses options et à éviter les erreurs. Si le joueur souhaite annuler une action, il peut réinitialiser sa sélection à tout moment grâce à la fonctionnalité de `reset`, qui efface la sélection en cours et supprime toutes les surbrillances affichées.

Ces interactions sont conçues pour être dynamiques et synchronisées avec l'état du jeu. Par exemple, si un mouvement est validé ou annulé, l'interface est immédiatement mise à jour pour refléter les changements, que ce soit sur le plateau ou dans les inventaires des joueurs. De plus, en cas de coup illégal, un message d'avertissement est affiché pour expliquer la nature du problème, renforçant ainsi la transparence des règles et rendant le jeu plus accessible aux nouveaux joueurs. Ce système assure une expérience utilisateur claire, réactive et alignée sur les mécaniques du jeu.

6 Ouverture à l'extension

Nous avons développé notre application dans l'optique de la rendre ouverte à l'extension, mais fermée à la modification. C'est exactement l'énoncé du second principe *SOLID* : *O : Open closed principle.*

Dans un premier temps, nous avons encapsulé les structures de données complexes utilisées afin de permettre à de futurs développeurs de les modifier plus simplement (voir Description de l'architecture).

Dans un second temps, l'ouverture à l'extension se fait aussi par la capacité du code à accepter de nouvelles fonctionnalités. C'est pourquoi l'ajout d'un pion et d'une IA est très rapide et simple pour un développeur, voire un joueur.

Pour ce faire, nous avons employé la même méthode pour le module *Pion* et le module *Entity*. Il existe deux interfaces dans notre jeu qui sont donc *Pion.h* et *Entity.h*. Ce sont 2 classes abstraites, qui possèdent autant de classe filles que de types de pion, respectivement de type d'IA (un joueur est considéré de la même manière qu'une IA). Ces interfaces possèdent les méthodes utilisées par chaque pion, respectivement entité. Certaines, peu nombreuses sont des méthodes virtuelles pures. Ainsi, il est nécessaire pour chaque type de pion ou d'entité de les redéfinir. C'est le cas notamment pour la méthode *getLegals* qui permet d'obtenir les coups légaux pour un pion sur le plateau. Étant donné que chaque pion a une mécanique particulière, chaque classe fille de la classe *Pion* doit redéfinir cette méthode.

Pour les IA, le principe est le même. Les méthodes virtuelles pures sont celles qui doivent retourner un pion choisi par exemple, ou un emplacement pour jouer. Dans le cas de la classe *Joueur*, ces méthodes se basent sur des entrées clavier ou des choix via l'interface graphique. Tandis que pour les autres classes (IA), le code dans ces méthodes implémente les algorithmes d'IA qui peuvent différer.

Ainsi, pour ajouter un nouveau type de pion ou d'entité, il suffit d'utiliser un pattern de classe déjà défini, de le modifier afin de personnaliser son IA ou la mécanique de son pion. Puis, il suffit de la placer dans le répertoire du projet correspondant, d'ajouter les fichiers *.h/.cpp* au *CMake*, et enfin d'ajouter le nom du pion, ou de l'IA, dans la bonne énumération dans le fichier *utils.h*.

6.1 Designs patterns

Afin de respecter les principes de programmation *SOLID*, il a été nécessaire d'inclure plusieurs design patterns dans notre modèle.

6.1.a Factory

Un des design patterns le plus important de notre projet est le design *abstract factory*. Son but ? Gérer la construction des objets *Pion*.

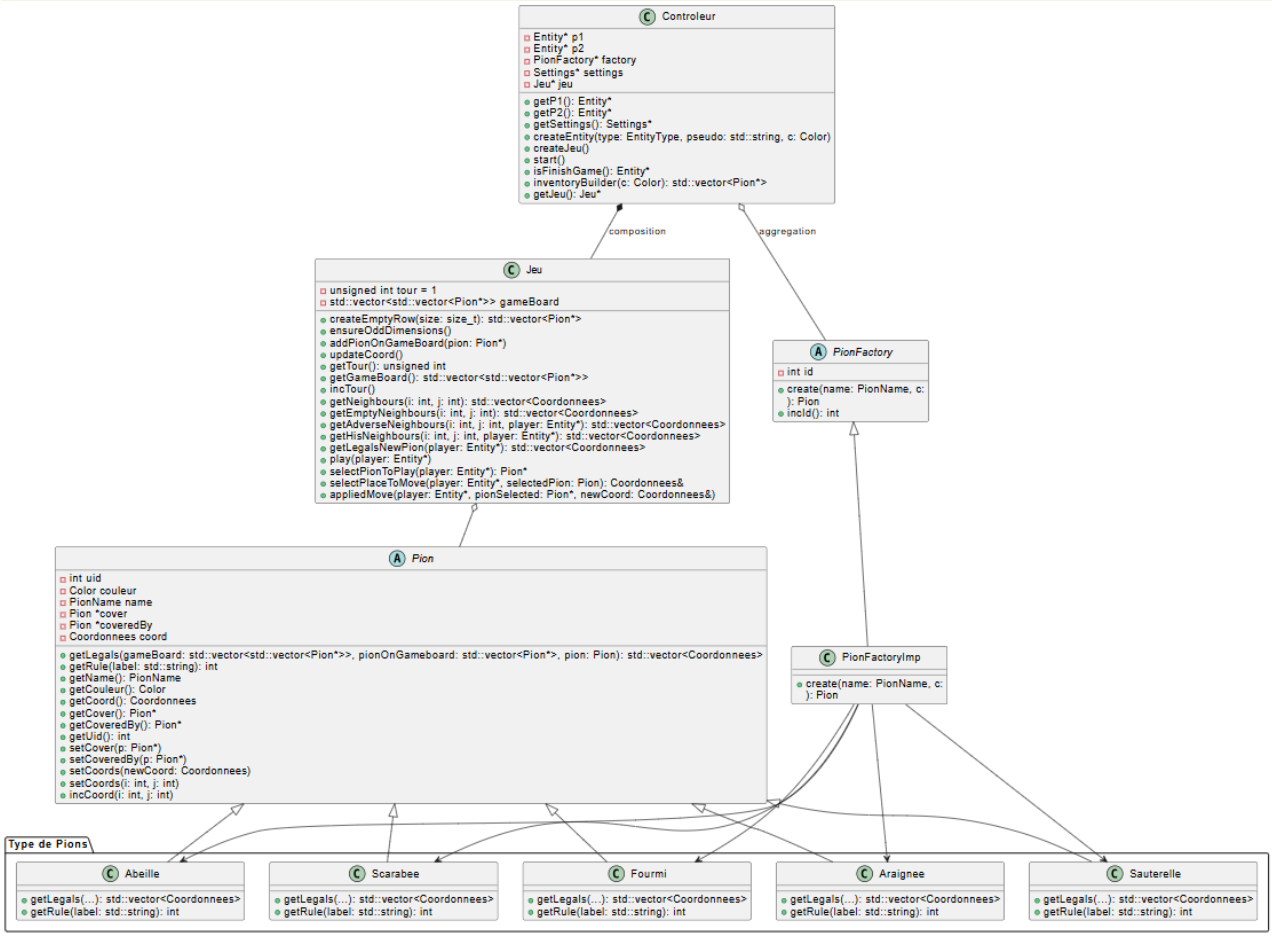


FIGURE 7 – UML détaillé de factory

L'utilisation de ce design est nécessaire afin de respecter les principes *SOLID*, notamment le principe *O : Open/close principle*, et le principe *D : Dependency inversion principle*. En effet, afin de laisser notre programme ouvert à l'extension et fermé à la modification, il est nécessaire que les objets provenant des classes filles de la classe *Pion* soient instanciés au même endroit, dans un switch par exemple. Admettons qu'un utilisateur souhaite créer un nouveau type de pion, il aura juste à ajouter un *case* dans ce switch sans toucher au code "client". C'est-à-dire toute la logique qui se base sur les règles du pion et définit quand créer le pion, combien, etc. Cependant, pour implémenter cette mécanique, il faut aussi que le code client ne dépende pas des détails d'implémentation. C'est le principe *D*. Pour ce faire, le pattern s'implémente avec une interface (classe abstraite) *Factory*, et une classe fille *FactoryImplementation*. Le code de la création des pions avec un *switch* est défini dans une méthode de cette implémentation, ce qui la rend dépendante des détails (les classes filles à *Pion*. Mais c'est bien notre classe *Controleur* qui est dépendante de la factory. Ce lien d'héritage nous permet de ne pas faire remonter la dépendance au code client. Il suffit alors de réaliser une injection de dépendance dans le *controleur* lors de sa création dans le main.

Il sera aussi nécessaire d'implémenter une *abstract factory* pour gérer la création des différentes IA. Aussi, une *stratégie* serait potentiellement intéressante à implémenter si le *switch* présent dans l'implémentation de la factory venait à se répéter.

6.1.b Singleton

Nous utilisons le design pattern *Singleton* pour les classes *Controleur* et *Settings*.

Le Singleton garantit qu'une seule instance d'une classe existe durant l'exécution du programme, et fournit un accès global à cette instance. Une instance unique est accessible via une méthode statique (getInstance()), et les instances sont nettoyées via un Handler.

Dans le cas de la classe *Controleur*, il est crucial de maintenir une cohérence dans la gestion des règles, de l'état de la partie, et de la communication entre les différents composants du système (par exemple, les joueurs et le plateau de jeu). La classe Controleur est en effet le gestionnaire central du jeu. Elle orchestre l'ensemble des opérations principales, en créant et en manipulant les objets Jeu, Settings, et les entités du jeu (joueurs et IA). En centralisant cette gestion dans une unique instance *Controleur*, nous évitons les conflits les erreurs logiques qui pourraient survenir si plusieurs instances tentaient de gérer le jeu simultanément, et nous assurons une synchronisation parfaite des actions et des événements de jeu.

De même pour la classe *Settings* : elle représente la configuration globale du jeu. Elle contient différents paramètres comme le nombre de retours en arrière autorisés (nbOfRollBack), qui affectent l'ensemble du système. Le choix d'implémenter Settings en tant que Singleton présente plusieurs avantages. Tout d'abord, cela permet une cohérence des paramètres globaux car le fait qu'il n'y ait qu'une seule instance de Settings garantit que tous les composants du projet (par exemple Controleur ou Jeu) accèdent aux mêmes paramètres et configurations. Cela évite des incohérences qui pourraient survenir si des composants utilisaient des instances différentes avec des paramètres différents. De plus, cela permet un accès centralisé : il y a un accès global à la configuration du jeu sans avoir à transmettre l'instance de Settings à chaque classe qui en a besoin. Toutes les classes qui nécessitent des paramètres peuvent simplement accéder à l'instance unique via Settings : :getInstance(). Cela simplifie le code et évite des dépendances circulaires. Enfin, l'implémentation du design pattern Singleton pour la classe Settings permet de gérer les changements de configuration de manière centralisée : si les paramètres du jeu doivent être modifiés au cours de l'exécution (par exemple, via notre option de menu), ces modifications n'affecteront qu'une seule instance de Settings, qui est ensuite utilisée par tout le programme.

7 Répartition des tâches

Tâche	**Personne(s) Assignée(s)**	Durée de réalisation
Décider d'un système de sauvegarde / rollback	Guillaume	3h
Système d'import-export d'une partie	Guillaume	10h
Amélioration de l'affichage console	Guillaume	5h
Implémentation des nouvelles structures de données	Guillaume	8h
Implémentation de la StandardIA et Menu IA	Noémie	4h

Tâche	**Personne(s) Assignée(s)**	Durée de réalisation
Ajout des fichiers extensions et Menu IA	Noémie	5h
Fonctions de positions possibles pour l'Araignée, l'Abeille, le Scarabée	Noémie	14h
Fonction de vérification de l'encadrement lors d'un déplacement de pion	Noémie	9h
Ecriture du Readme	Noémie	3h
Fonctions de positions possibles pour la Sauterelle et la Fourmi	Agathe	9h
Création de jeux de test pour les positions possibles et tests et debug	Agathe	6h
Fonctions de vérification de la connexité	Agathe	5h
Fonctions de positions possibles pour les extensions Moustique et Coccinelle	Agathe	8h
Gestion des cas déplacement / placement des pions si pas possible	Agathe	6h
Programmation du Menu et de l'interface graphique du jeu avec Qt	Rayan	42h (cumulé)

Cette répartition n'inclut pas les livrables (rapports intermédiaires, rapport final, vidéo).

7.0.a Part de contribution de chaque membre sur l'ensemble du projet

Bien que difficile à estimer et que nous ayons tous contribué à ce projet, nous estimons nos parts de contribution à : 40% pour Guillaume THEUNISSEN, 21% pour Agathe RABASSE, 21% pour Noémie THOMASSON et 18% pour Rayan BARREDDINE.

8 Contribution personnelle et retour d'expérience

8.1 Barreddine Rayan

Ce projet a été une expérience extrêmement enrichissante et formatrice, notamment sur le plan de la conception et de l'implémentation de l'interface graphique. Étant donné que je découvrais les bibliothèques utilisées, cela a représenté un véritable défi. L'interface graphique, en particulier, nécessitait une attention particulière pour s'assurer qu'elle soit intuitive, fonctionnelle et qu'elle reflète les mécaniques du jeu de manière claire et efficace. J'ai dû repenser et réadapter une grande partie du fonctionnement du jeu pour qu'il s'aligne avec cette nouvelle interface. Cela a impliqué une approche différente du gameplay, avec une refonte de certains éléments pour garantir une interaction fluide entre les utilisateurs et les fonctionnalités du programme. En plus de relever le défi technique de la programmation de l'interface, j'ai également dû m'assurer que la logique du jeu soit

parfaitement intégrée, en prenant en compte des cas particuliers et en rendant le tout robuste face aux éventuelles erreurs ou incohérences. Ce travail m'a permis de développer des compétences en programmation orientée objet, en gestion des événements utilisateurs et en optimisation de l'expérience utilisateur, tout en renforçant ma capacité à m'adapter à de nouvelles méthodes et à les intégrer dans un projet complexe.

8.2 Rabasse Agathe

J'ai surtout travaillé sur des parties algorithmiques comme les calculs de déplacements légaux de plusieurs pions, par les fonctions *getlegals* pour la Sauterelle, la Fourmi, le Moustique et la Coccinelle. De plus, je me suis occupée des vérifications de la connexité pour qu'on ne puisse pas déplacer un pion si cela déconnecterait la ruche, et du debugage de la fonction pour vérifier si le déplacement est physiquement possible. J'ai également travaillé sur les placements de pions et les gestions lorsque aucun pion ne peut être placé, déplacé, etc. et sur les gestions de voisins. Par ailleurs, j'ai réalisé des fonctions de tests pour les différents pions avec des configurations qui ont permis un gain de temps pour identifier des problèmes. J'ai compris que le plus long était de trouver la source des erreurs et j'ai compris l'utilité du debugger.

De plus, ce projet m'a beaucoup appris sur le travail collaboratif de projets en informatique, en particulier sur Git : bien que je l'avais déjà utilisé auparavant, c'était surtout sur des projets moins en simultanée. J'ai pu en apprendre plus sur les avantages des branches et la gestion des conflits pour merge plusieurs branches.

8.3 Theunissen Guillaume

Au cour de ce projet, j'ai pu travailler sur les parties suivante : définition de l'architecture, Conception de l'UML, création de la structure du projet, définition et création de la partie console. Mais aussi concernant les choix techniques, notamment dans le but de suivre les principes de programmation *SOLID* : l'implémentation des conteneurs encapsulés dans d'autres classes, l'implémentation de 90% du code du cycle de jeu, l'implémentation des différents design patterns (factory, singleton, memento).

grâce à cela, j'ai pu acquérir de nouvelles compétences notamment en programmation orienté objet, mais aussi de manière plus général en gestion de projet de développement. Aussi, la découverte des principes *SOLID* et des design pattern ont pu faire évoluer ma manière de penser et d'implémenter un projet.

8.4 Thomasson Noémie

Durant ce projet j'ai pu participer à l'implémentation de plusieurs modules. J'ai commencé par m'occuper des méthodes obtenant les positions de déplacement possibles d'un pion (*getLegals*), pour l'Araignée, l'Abeille et le Scarabee. Pour cela j'ai dû créer d'autres méthodes intermédiaires comme *getUniversallegals1* ou *isphyspossible*. Une majeure part du développement de ces fonctions étaient les tests et debugs.

Je me suis ensuite tournée vers l'implémentation de la StandardIA et de l'adaptation du code pour l'ajouter : utilisation de la classe abstraite **Entity** et passage des méthodes non virtuelles en méthodes virtuelles ainsi que leurs redéfinition dans les classes dérivées.

J'ai donc aussi implémenté le Menu IA dans les Settings qui permet de choisir le type de joueur (IA à préciser si plusieurs ou Joueur).

J'ai finalement participé à l'ajout des extensions en créant les fichiers et en implémentant le Menu Extension permettant d'ajouter, retirer ou consulter les règles d'une extension.

J'ai aussi rédigé le Readme permettant à chaque utilisateur de notre projet de comprendre sa logique et comment il peut y contribuer.

Le fait de travailler sur des parties liées à l'architecture et la logique générale du projet (dans les Menus) m'a poussé à conserver une méthode de code clair, fonctionnelle et adaptée, c'est-à-dire nécessitant le moins de changements de la structure actuelle.

9 Améliorations à apporter

Il aurait été intéressant d'implémenter le design pattern stratégie afin de rendre encore plus simple l'ajout d'extension dans notre jeu.

Nous pouvons aussi faire en sorte d'améliorer l'IA mise en place en faisant en sorte qu'elle prenne en compte les coups joués ou bien qu'elle teste dix prochains coups et joue le coup actuel en conséquence.

Actuellement, les parties entre IA et l'importation de partie marchent mieux en console que sur l'interface graphique.

10 Conclusion

Nous avons fait en sorte de rendre une application la plus fonctionnelle, facile d'utilisation mais surtout la plus adaptable. Tout au long du projet nous avons tenu à respecter le principe SOLID mais aussi au sein de notre code, nous avons conservé une méthode logique en se projetant sur la future utilisation des structures quitte à les modifier plusieurs fois durant le développement du jeu.

Même en partant chacun sur une base de compétences en C++ et autres langages différente, nous avons essayé de contribuer au mieux au projet tout en assimilant le C++.

Pour certains nous avons aussi découvert Git et la dynamique et organisation de projet de programmation en groupe et avons pris en compte les responsabilités d'un projet si conséquent.