

# Rapport Final - LO21

TD 2 : Mardi 8h-12h

Groupe 3

Oscar LETULIER

Robin CIVELLI

Emma FALKIEWITZ

Evan MORIN

# Sommaire

<b>Sommaire.....</b>	<b>2</b>
<b>Introduction.....</b>	<b>3</b>
<b>Description de l'application.....</b>	<b>4</b>
Interface console.....	4
Interface graphique.....	8
Extensions implémentées.....	16
<b>Description de l'architecture.....</b>	<b>17</b>
<b>Évolutivité du jeu.....</b>	<b>20</b>
Structure orientée objet.....	20
Étapes pour ajouter un nouvel insecte (officiel ou inventé).....	20
Modularité.....	20
<b>Planning effectif du projet.....</b>	<b>23</b>
<b>Contributions personnelles.....</b>	<b>25</b>
<b>Conclusion.....</b>	<b>26</b>

# Introduction

L'objectif de ce projet était de concevoir et développer une application permettant de jouer au jeu de société **Hive**, un jeu stratégique pour deux joueurs dans lequel chaque joueur possède un ensemble de pièces représentant des insectes. Chaque insecte dispose de règles spécifiques de déplacement, et l'objectif est de capturer la Reine Abeille de l'adversaire en l'entourant complètement, tout en empêchant l'adversaire d'en faire de même. Le jeu peut également se terminer par une égalité si les deux joueurs se retrouvent bloqués ou si leurs actions mènent simultanément à la capture des deux Reines Abeilles.

Chaque joueur dispose de 11 pièces de base, comprenant des fourmis, des sauterelles, des scarabées, des araignées, et une Reine Abeille, avec la possibilité d'ajouter des extensions comme le moustique ou la coccinelle. Les joueurs alternent leurs tours, en posant ou en déplaçant des pièces sur un plateau virtuel. Les règles imposent des restrictions, comme l'obligation de poser la Reine Abeille dans les quatre premiers tours ou l'interdiction de rompre la continuité de la ruche en jeu.

L'objectif de ce projet était de développer une application permettant de jouer des parties de Hive dans plusieurs configurations : entre deux joueurs humains ou contre une IA simulant un joueur. Elle devait aussi intégrer des extensions officielles pour diversifier les parties. Ce rapport détaille les étapes de conception et d'implémentation de l'application, en couvrant les fonctionnalités principales, l'architecture sous-jacente, et les solutions aux défis techniques rencontrés. Enfin, il inclut une réflexion sur l'organisation du projet et la contribution de chaque membre de l'équipe, puis il se termine par un bilan du travail accompli par notre groupe.

# Description de l'application

Maintenant que nous avons établi précisément les règles du jeu, nous présenterons et expliquerons les différentes fonctionnalités de notre application. Nous commencerons par expliquer les fonctionnalités de base de l'application, c'est-à-dire l'obtention des différents déplacements des pièces et leur exécution. Nous présenterons ensuite les interactions possibles avec l'application, à savoir joueur vs joueur ou joueur vs IA (qui génère des placements et déplacements de manière aléatoire), puis nous détaillerons les différentes extensions implémentées pour finir par présenter les interfaces consoles et graphiques de l'application.

## Interface console

### Fonctionnalités principales

L'application permet à un utilisateur de lancer une partie de Hive en console ou interface graphique. Tout d'abord, l'application console propose de choisir entre 2 joueurs ou 1 joueur et 1 IA puis permet à l'utilisateur de nommer les 2 joueurs. L'utilisateur a ensuite le choix le nombre de retour en arrière autorisé durant la partie, puis d'utiliser 0, 1 ou 2 extensions qui sont le moustique et la coccinelle.

Le déroulement se fait comme suit par la suite :

- Un menu affiche chaque joueur avec leur couleur (Blanc ou noir), les pièces que possède le joueur dont c'est le tour puis les 3 actions possibles (Poser une pièce, Déplacer une pièce ou Revenir en arrière). Lors du premier tour, le joueur ne peut pas choisir de déplacer une pièce ou de revenir en arrière, il ne peut que choisir quelle pièce poser sur la case (0,0).
- Il est également proposé au joueur d'abandonner la partie, auquel cas le joueur adverse est déclaré gagnant. Enfin, il peut décider de quitter la partie, on lui propose alors de la sauvegarder s'il le souhaite pour pouvoir la recharger ensuite

```

***** CHOIX ACTION *****

*** PLATEAU ***

Q: 0, R: 0

  / \
 /   \
/     \
|  0, 0  |
| -R-   |
|     \  |
 \     /
  \   /
  \ /

C'est a emma (Noir) de jouer !

*** DECK ***

Araignee : 2
Fourmis : 3
Reine Abeille : 1
Scarabee : 2
Sauterelle : 3

Voulez-vous poser une piece ou en deplacer une ?
1: Poser une piece
2: Deplacer piece
3: Revenir en arriere
4: Abandonner
5: quitter

```

Interface utilisateur d'un tour classique

- Les tours suivants se déroulent sans cas particulier, à part si au 4ème tour le joueur n'a pas posé sa Reine Abeille, il est obligé de la placer sur une case disponible. Si le joueur choisit de placer une nouvelle pièce, l'ensemble des cases disponibles est affiché et s'il choisit de déplacer une pièce, les déplacements possibles pour la pièce choisie sont aussi affichés. S'il choisit de revenir en arrière, les 2 coups précédents sont annulés (celui de l'adversaire et le sien qu'il doit rejouer). Après chaque tour un affichage du plateau de jeu est visible pour permettre au joueur de visualiser l'apparence de celui-ci et la position de ces différentes pièces.

[illegible]

### Exemple d'affichage de la partie

C'est a Robin (Blanc) de jouer !

\*\*\* DECK \*\*\*

Araignee : 1  
Fourmis : 2  
Moustique : 1  
Reine Abeille : 1  
Scarabee : 1  
Sauterelle : 3  
Coccinelle : 1

Votre etes obligé de poser votre reine

1. Reine Abeille (Blanc)  $\rightarrow (-2, 0)$
2. Reine Abeille (Blanc)  $\rightarrow (-2, 1)$
3. Reine Abeille (Blanc)  $\rightarrow (-2, 2)$
4. Reine Abeille (Blanc)  $\rightarrow (-1, -1)$
5. Reine Abeille (Blanc)  $\rightarrow (-1, 2)$
6. Reine Abeille (Blanc)  $\rightarrow (0, -1)$

### Tour 4. Reine pas encore posée

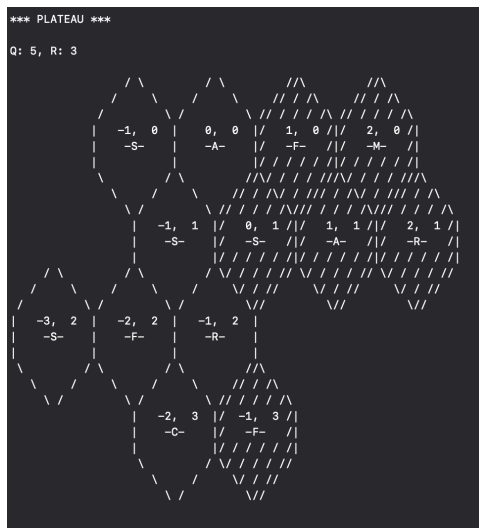
[illegible]

### Tour 5. État du plateau après avoir posé les reines

[illegible]

## Tour 6. Après être revenu en arrière

- La partie se déroule jusqu'à ce que la reine abeille soit encerclée, qu'un joueur abandonne ou qu'un joueur quitte le jeu.



\*\*\*\*\* FIN DE LA PARTIE \*\*\*\*\*

Partie termin\351e !  
Victoire de Oscar (Noir)

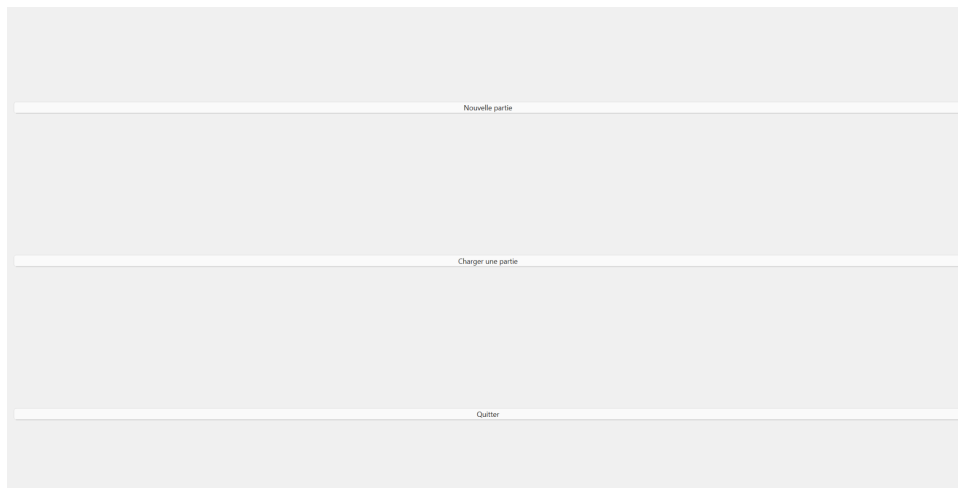
Etat de la partie à l'avant dernier tour  
(avant de déplacer la sauterelle)  
(-3,2) -> (0,2)

En conclusion, notre Jeu implémente l'ensemble des opérations attendues (2 extensions officielles, une IA, la possibilité d'effectuer des retours en arrière...), en plus d'une sauvegarde permettant de reprendre une partie qui avait été quitté avant d'être finie. Pour l'affichage des cases, nous avons porté un projet initialement fait en Kotlin et l'avons converti en c++ en corrigeant quelques mineures erreurs faites dans le programme, le rendu donne un bon affichage du en mode console.

## Interface graphique

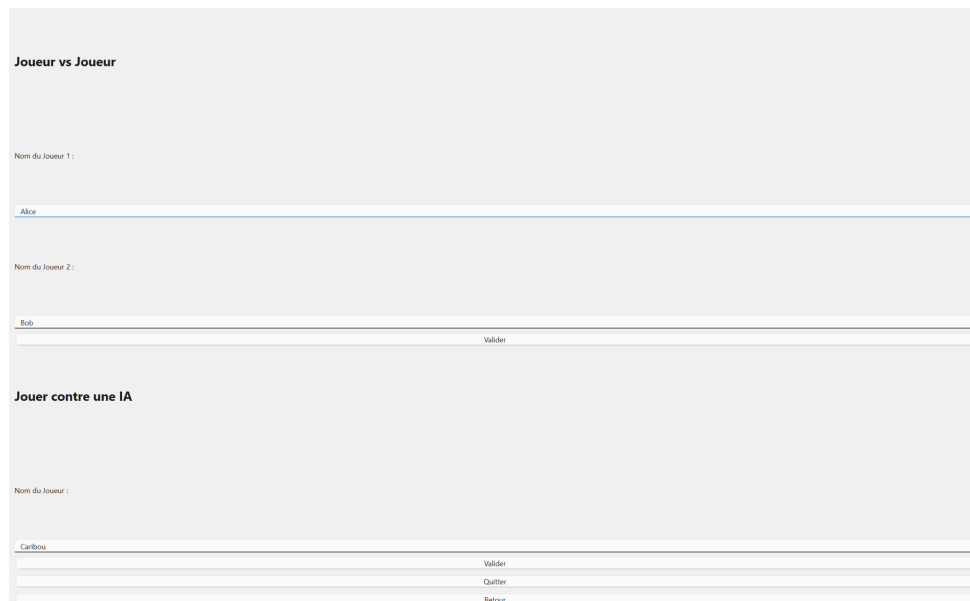
### Menus de configuration

À son arrivée sur l'interface graphique Qt, l'utilisateur peut, comme dans l'interface console, démarrer une nouvelle partie, en charger une ou quitter via le Menu principal. En cliquant sur « Charger une partie », un menu de sélection s'affiche, offrant la possibilité de continuer la partie en cours, de l'abandonner, de retourner au menu principal ou de quitter. Si « Nouvelle partie » est sélectionné, le menu de configuration des joueurs apparaît, permettant à l'utilisatrice d'entrer les noms des deux joueurs si elle souhaite jouer à deux, ou d'indiquer son nom dans la section "Joueur contre IA".



The screenshot shows a light gray rectangular window titled "Accueil du jeu". Inside, there are three horizontal white bars, each containing a text label: "Nouvelle partie", "Charger une partie", and "Quitter".

*Accueil du jeu*



The screenshot shows a light gray rectangular window titled "Configuration de la partie". It is divided into two main sections. The top section is titled "Joueur vs Joueur" and contains two input fields for player names. The first field is labeled "Nom du Joueur 1:" and contains the text "Alice". The second field is labeled "Nom du Joueur 2:" and contains the text "Bob". Below these fields is a "Valider" button. The bottom section is titled "Jouer contre une IA" and contains a single input field for the player's name, labeled "Nom du Joueur:", which contains the text "Caribou". Below this field are three buttons: "Valider", "Quitter", and "Retour".

*Configuration de la partie*



Une fois la configuration du jeu et des joueurs effectuée, le menu des paramètres du jeu s'affiche. Ce menu permet de choisir le nombre de retours possibles dans la partie ainsi que les extensions à activer. Il est toujours possible de revenir à la page précédente ou de quitter. Une fois que le bouton « Valider » est cliqué, l'interface du jeu s'affiche, avec le plateau et les pièces des deux joueurs.

*Configuration du nombre de retours et des extensions*



*Interface du jeu*

Sur l'affichage du jeu, les mains des joueurs se situent sur les côtés de l'écran, et à chaque tour, le cadre des pièces de l'une des deux mains se colore pour indiquer à qui revient le tour. Le bouton « Quitter », situé en bas de l'écran, permet d'accéder au menu de Sauvegarde, puis à un autre menu (Menu Post Sauvegarde) où il est possible de revenir au menu principal, de quitter ou de retourner aux pages précédentes. La sauvegarde n'a cependant pas été implémentée sur la partie graphique.

Sauvegarder?

Oui

Non

Retour

Quitter

Menu de sauvegarde

Menu post sauvegarde

Menu

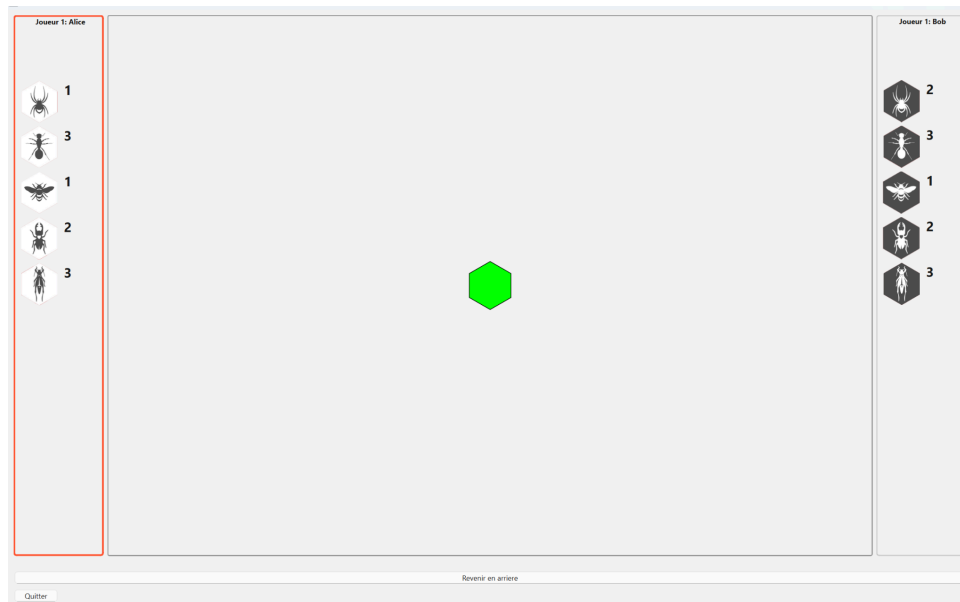
Retour

Quitter

Menu post sauvegarde

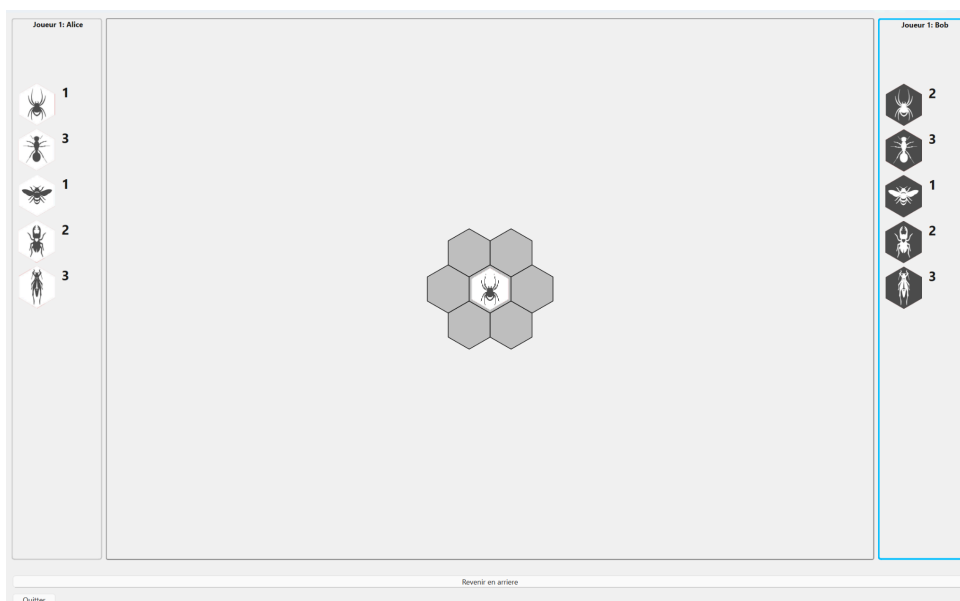
## Interface du jeu

L'interface du jeu permet de jouer la partie. Comme dit au dessus, les deux joueurs et leurs pièces sont situés de part et d'autre du plateau. En cliquant sur une pièce de la main du joueur qui joue, les cases où elles peuvent être placées sont coloriées en vert.



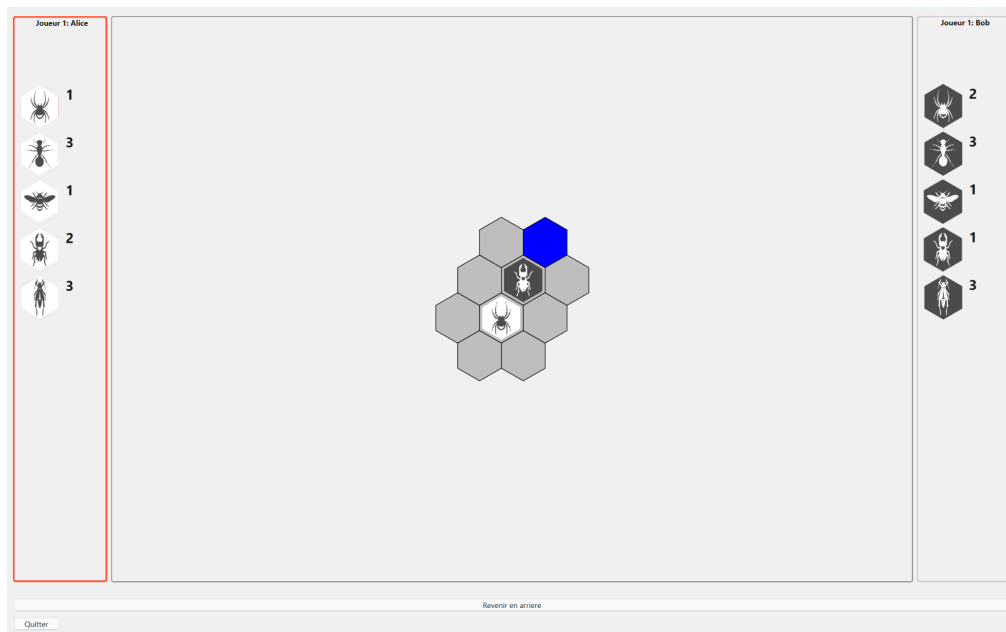
*Placement possible pour l'araignée*

Si l'on clique à nouveau sur la pièce de sa main, la pièce se désélectionne et est replacée dans la main, et la couleur verte disparaît. Si l'on clique sur la case, la pièce se pose comme prévue et le tour passe au jour suivant. Le plateau s'actualise avec les nouvelles pièces.



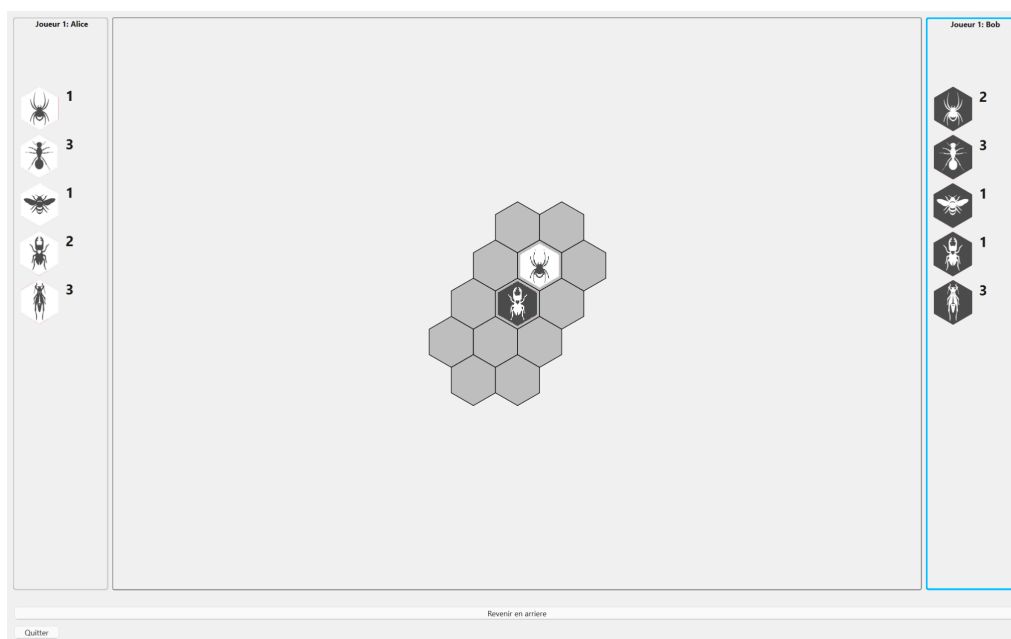
*Araignée sur le plateau*

Si l'on clique sur une pièce alliée, ses déplacements possibles apparaissent en bleu.



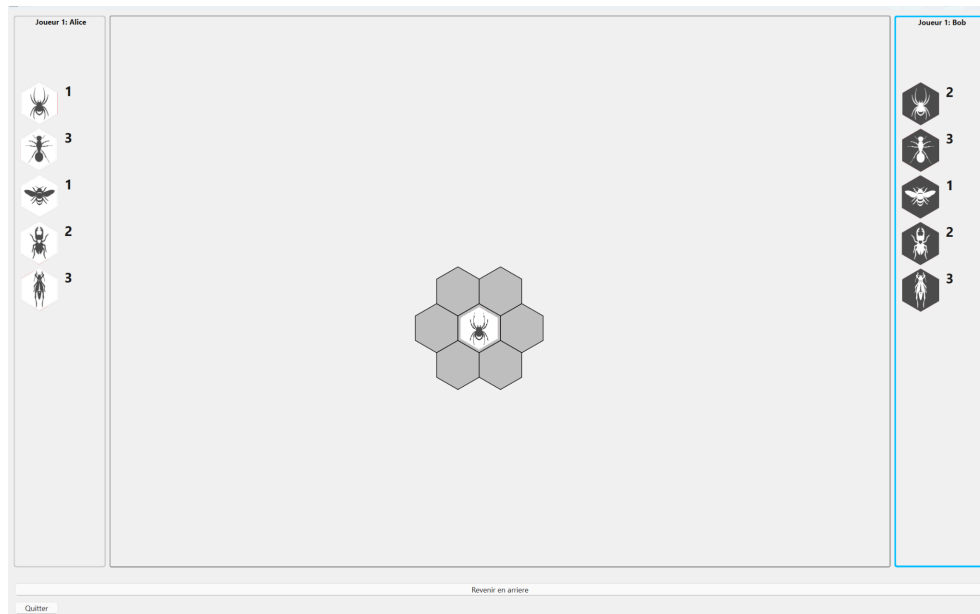
*Affichage des déplacements possibles*

Si on clique sur cette case bleue, le déplacement est effectué.



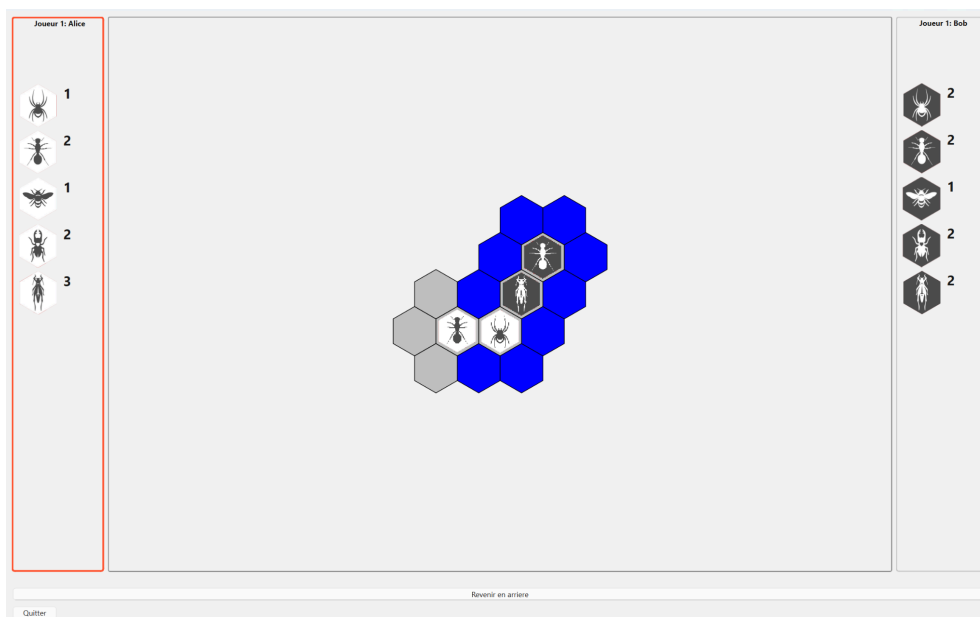
*Positions des pièces après déplacement*

Si l'on appuie sur le bouton revenir en arrière en tant que joueur de côté Noir, cela annule les deux derniers mouvements effectués. Le retour en arrière n'a pas vraiment de sens en joueur contre joueur, mais nous avons décidé de le laisser pour éviter d'augmenter la complexité.



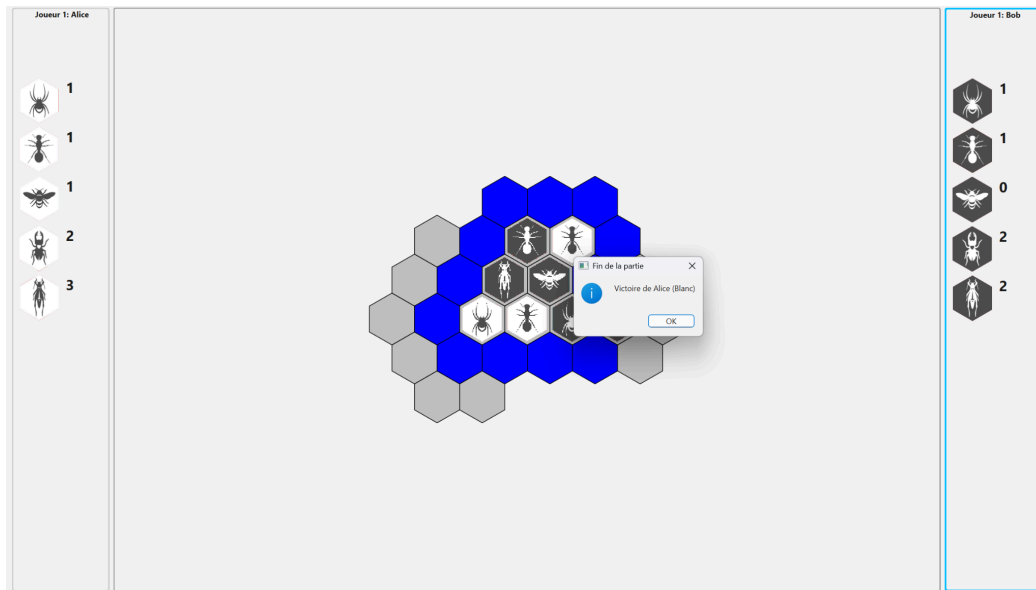
*Retour avant le déplacement de l'araignée et placement du scarabée*

Tous les déplacements ont bien été implémentés pour les insectes



*Déplacements de la fourmi blanche*

Le système de victoire est aussi implémenté, une fois qu'une reine est entourée, la partie se termine. Un message apparaît à l'écran avec le vainqueur.



*Fin de partie et victoire des blancs*

Si l'on ferme le message, un menu de fin de partie apparaît.



*Menu de fin de partie*

## Extensions implémentées

Nous avons choisi pour notre application de proposer au joueur d'utiliser le moustique et/ou la coccinelle au début de la partie.

Pour le moustique les règles de son déplacement sont les suivantes : Il se place comme tout autre pièce. Pour se déplacer il récupère les caractéristiques de déplacements de chaque insecte qu'il touche (que la pièce soit alliée ou adverse). Il peut ainsi changer de déplacement à chaque tour. S'il est placé à côté d'un scarabée empilé, il peut copier le scarabée mais pas la pièce en dessous.

Pour la coccinelle, ses règles de déplacements sont les suivantes : elle se déplace de trois cases exactement, les deux premiers déplacements se font sur des cases occupées puis elle descend d'une case vers le bas pour le troisième déplacement. Elle ne peut pas se déplacer autour de la ruche et finir son déplacement sur une autre pièce. Elle peut se déplacer vers des cases encerclées ou sortir de ces dernières.

Nous avons aussi implémenté une sauvegarde pour permettre, au démarrage du jeu, au joueur de choisir s'il veut reprendre une partie sauvegardée lors de la dernière utilisation, ou commencer une nouvelle partie à partir du début. Il a aussi la possibilité d'abandonner, ou quitter (avec ou sans sauvegarde) une partie en cours s'il le souhaite. Pour cela, nous avons implémenter des méthodes **save** et **load** dans les classes Deck, Joueur, Plateau et Memento pour permettre d'enregistrer et de renvoyer les informations essentielles de chaque classe à la restauration de la partie dans un fichier texte. Puis les méthodes **saveToFile** et **loadFromFile** de la classe Partie permettent l'enregistrement de la partie en appelant les méthode précédentes (via la classe Memento), puis de charger la partie à partir du fichier texte pour la reprendre.

# Description de l'architecture

De manière générale notre architecture repose sur 4 structures :

- Le “moteur de jeu” : Ce sont les classes comme Partie, Plateau, Case, Position
- Les entités du jeu : Les pièces et ses classes héritées
- L'interface utilisateur : Interface, InterfaceConsole, InterfaceGraphique
- Les joueurs : la classe Joueur et les héritages Humain et IA

## Composantes Principales

La classe **Partie** : elle représente une instance de partie Hive et gère la logique du jeu (initialisation, déroulement des tours, gestion des joueurs et les conditions de victoire).

Cette classe gère le cycle de vie des joueurs, du plateau, des pièces, pour les créer au démarrage de la partie et les supprimer à la fin, et elle gère la sélection des mouvements par l'utilisateur. La séparation de la logique de jeu dans partie permet de séparer les règles “globale” du plateau, cases...

La classe **Plateau** : elle représente comme son nom l'indique le plateau. Il est composé de la classe Case et gère le cycle de vie des cases le composant.

La classe a comme attributs une structure map<Position, case> pour stocker les cases par leur position, cela permet un accès rapide grâce aux positions des cases ce qui est pratique pour vérifier des conditions telles que le glissement, la liaison de la ruche...

Les méthodes principales sont **deplacerPiece** et **peutBouger** qui permettent respectivement de gérer les déplacements des pièces sur le plateau et vérifier si une pièce peut bouger. Le fait que le plateau gère le cycle de vie des pièces rend aussi la création de cases plus simple dès qu'on ajoute des pièces.

La classe **Case** : une instance de cette classe représente une case du plateau et gère les pièces empilées sur celle-ci. Elle possède comme attributs une position et un tableau (vector) des pièces présentes sur la case pour gérer facilement le cas de l'empilement avec le scarabée.

Elle possède également des méthodes **getVoisins** et **estOccupee**, essentielles pour vérifier des contraintes sur les pièces.

La classe **Piece** : cette classe est la classe de base pour chaque pièce du jeu. Chaque insecte (ReineAbeille, Araignee, Sauterelle...) hérite de cette classe et implémente les règles spécifiques à son déplacement. Cela permet d'ajouter de nouveaux insectes sans modification du code. La classe Piece possède un attribut type permettant d'identifier l'insecte et l'appel du constructeur de la classe dans les classes héritées. La méthode **getDeplacements** est virtuelle pure et implémentée par chaque classe pour définir ses déplacements. Cela permet d'ajouter de nouveaux insectes sans modification du code.

La classe **Joueur** : Cette classe représente un joueur (**Humain** ou **IA**) et gère la logique de décision comme choix de la pièce. La classe IA hérite de cette classe, cette “centralisation” des données de base de chaque joueur permet une gestion des joueurs plus simple pour la partie, et rend plus simple l'implémentation d'une nouvelle IA.



La classe **Interface** : cette classe fournit les interactions entre le jeu et l'utilisateur. Elle est la base des classes **InterfaceConsole**, qui affiche en mode texte et permet d'interagir en commande, et **InterfaceGraphique**, qui utilise Qt pour une interface graphique.

La classe **Mouvement** : cette classe est la base des classes **Deplacement** et **Placement** qui permettent l'exécution d'un mouvement (déplacement ou placement), nous avons choisi un héritage par référence pour permettre de séparer le placement d'une pièce sur le plateau et le déplacement lorsque celle-ci est déjà présente sur le plateau. Chaque classe fille possède des méthode **executer** et **annuler** permettant les retours en arrière depuis la classe **Partie**.

## Composantes secondaires

Design Pattern **Memento** : nous avons choisi d'implémenter un design Pattern **Memento** sur **Partie** pour faciliter la gestion de la sauvegarde de la partie, utile pour le retour en arrière. Nous réalisons un mouvement inversé pour retourner en arrière puis grâce au Memento on remet le plateau comme il était au moment de la sauvegarde, c'est-à-dire avant les mouvements annulés.

Design Pattern Iterator : nous avons implémenté plusieurs designs pattern Iterator, pour des questions de place nous n'avons représenté sur l'UML que celui de **Deck** mais nous en avons implémenté un dans **Case** et un dans **Plateau**, pour faciliter le parcours des pièces dans Deck et Case, et le parcours des cases dans Plateau.

Classe **Position** et **Directions** : nous avons choisi de séparer la classe **Position** de **Piece**, cela permet principalement de gagner en clarté et pour la gestion des coordonnées hexagonales, parfois complexes. Une pièce ne possède pas de position ce qui permet de dissocier la pièce de sa position pour faciliter la gestion des mouvements... En effet, les règles comme le glissement ou l'adjacence s'appuient sur la classe Position, permettant d'éviter la duplication de calculs dans d'autres classes. La classe **Direction** permet d'obtenir plus facilement les voisins d'une pièce, ici aussi l'objectif était de gagner en clarté et en efficacité.



## Évolutivité du jeu

Notre architecture a été conçue pour rendre l'ajout d'extensions (nouvelles pièces ou insectes) simples et modulaires. Voici un résumé clair des étapes nécessaires et des principes qui rendent cela possible

### Structure orientée objet

Chaque insecte du jeu est représenté par une classe héritant de la classe **Piece**, qui contient les fonctionnalités de base communes à toutes les pièces. Cette structure permet de définir des comportements spécifiques à chaque insecte tout en réutilisant le code commun.

### Étapes pour ajouter un nouvel insecte (officiel ou inventé)

- Mise à jour de l'énumération `TypePiece` : le développeur doit tout d'abord ajouter le nouvel insecte à l'énumération `TypePiece` présente dans le fichier `TypeInsecte.h`. Cela permettra l'identification du type de la nouvelle pièce dans la classe `Piece`.
- Création d'une classe pour le nouvel insecte : le développeur doit créer une nouvelle classe (par exemple "Cloporte") qui hérite de la classe `Piece`. Cette classe possède la même architecture que les autres insectes. Elle doit implémenter un constructeur appelant celui de pièce avec comme paramètre le type du nouvel insecte.
- Implémentation de la méthode `getMouvements` : dans la classe du nouvel insecte la méthode `getMouvements` doit être implémentée. Elle décrit le comportement et les caractéristiques spécifiques du nouvel insecte et permet de récupérer les différents déplacements possibles lorsque celle-ci est posée. Cette méthode prend en paramètre la position de la pièce avant le potentiel déplacement et le plateau (représentant l'état actuel du jeu).
- Utilisation et implémentation optionnelle de `DeplacementValide` : pour certaines pièces nécessitant des règles de déplacement complexe nous avons décidé d'implémenter une méthode `DeplacementValide` appelée par la méthode `getMouvements`. Cette méthode prend en paramètre le plateau et le déplacement envisagé pour permettre de vérifier des contraintes supplémentaires sur les déplacements. Si les règles sont suffisamment simples pour être directement intégrées dans `getMouvement`, cette méthode n'a pas à être implémentée et n'influe pas sur le reste du code.

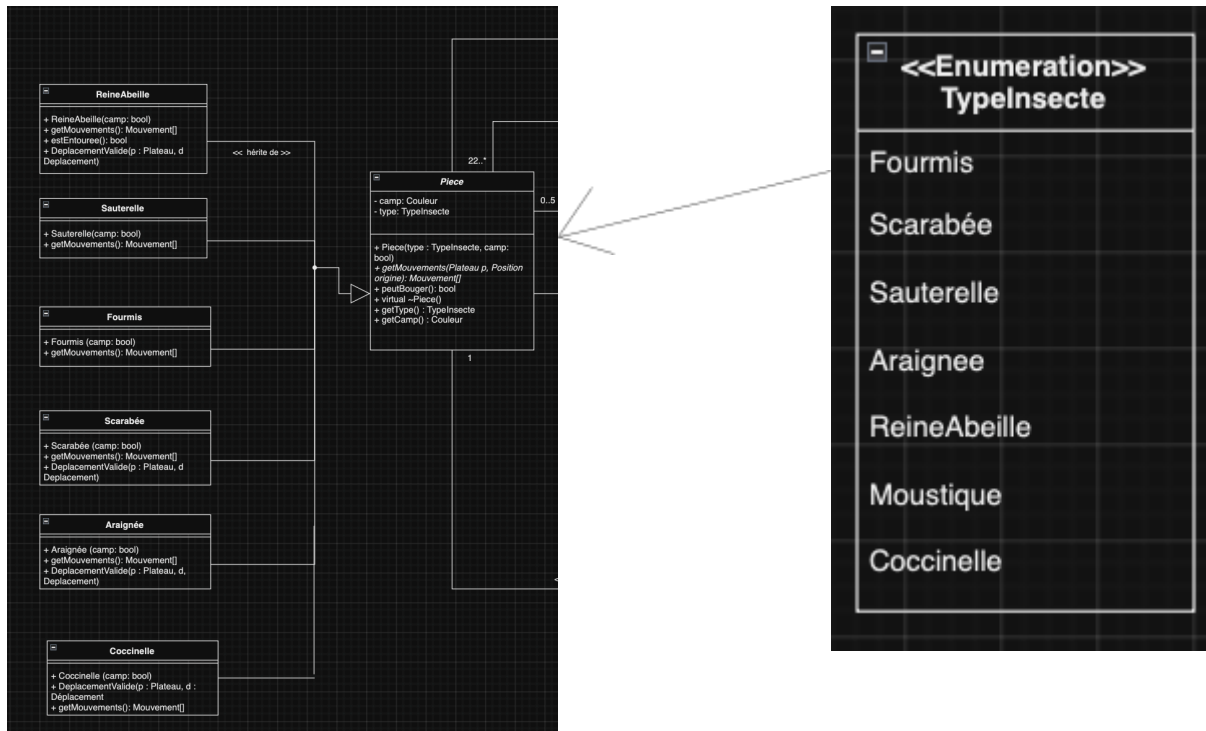
### Modularité

L'ajout d'un nouvel insecte n'impact pas les autres parties du code. En effet, la méthode `getMouvements` de chaque insecte est indépendante et son implémentation ne nécessite pas de modifications d'autres classes.

Les sous-parties du diagramme UML ci-dessous montrent que la classe `Piece` agit comme classe Mère, c'est-à-dire comme base pour tous les insectes. Ensuite, chaque classe fille (`ReineAbeille`,

Araignée, Sauterelle...) implémente ses propres règles de déplacement dans *getMouvements*, et la gestion des différents insectes est rendue simple grâce à l'énumération des différents types.

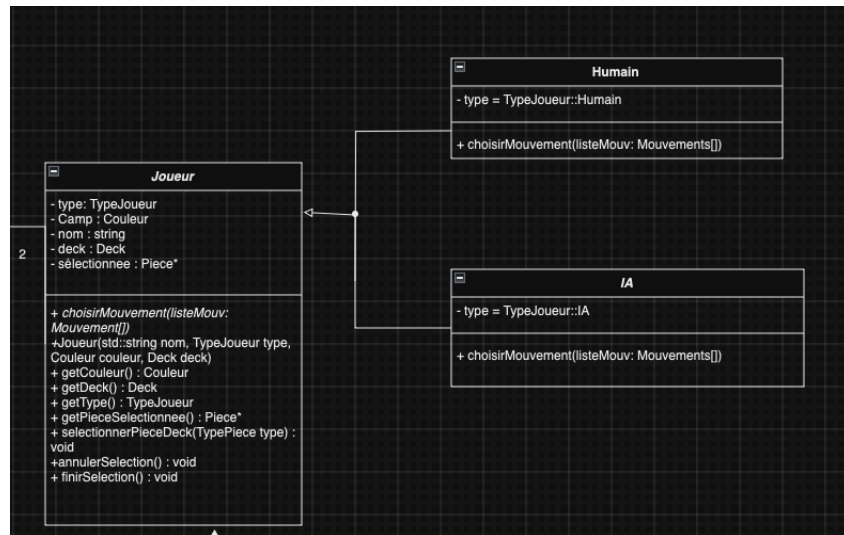
En conclusion, notre architecture permet d'intégrer de nouvelles extensions rapidement, pour lesquelles l'implémentation se limite à la définition des caractéristiques de déplacement du nouvel insecte, sans modification du reste du code.



Notre UML souligne bien le propos précédent puisqu'on voit que chaque pièce est une classe fille, et en supprimer ou ajouter une ne nécessiterait pas de changer la structure du code (hors ReineAbeille).

L'ajout d'une IA plus poussée ne nécessiterait pas de modifications complexes dans l'architecture existante du code. En effet, les méthodes utilisées sont suffisamment flexibles pour intégrer de nouveaux comportements plus poussés.

Pour ajouter une IA plus avancée, il suffirait de créer une nouvelle classe qui hérite de l'IA existante et de redéfinir la méthode qui choisit le mouvement. Par exemple, au lieu de choisir un mouvement aléatoire, l'IA avancée pourrait utiliser une logique de recherche pour évaluer les meilleures options, sans pour autant être très poussée. Cette approche permettrait d'ajouter de nouvelles fonctionnalités pour l'IA sans toucher au reste du code, rendant l'extension de l'IA simple à mettre en place.



Architecture de la sous-partie sur la structure des joueurs.

# Planning effectif du projet

Initialement notre planning était le suivant :

- Rendu 1 : Compréhension des règles et contraintes du jeu et conception de l'architecture.
- Rendu 2 : Finalisation de l'architecture et implémentation des classes de bases.
- Rendu 3 : Implémentation des classes restantes et ajout des extensions pour permettre au jeu un fonctionnement en console.
- Rendu Final : Développement de l'interface graphique.

Nous avons cependant fait face à plusieurs défis et nous avons dû revoir notre planning au fur et à mesure de l'avancée du projet.

Planning effectif :

- Rendu 1 : Pour le premier rendu, notre objectif était de comprendre le déroulement du jeu et de proposer une première architecture globale de notre projet et d'estimer le temps qu'il nous faudrait pour implémenter les premières classes "principales" (Piece, Insecte, Joueur, Plateau...). Nous avons donc à ce stade une bonne compréhension des différentes règles et contraintes du jeu, ainsi qu'une première architecture.
- Rendu 2 : L'objectif que nous nous étions fixé était d'avoir implémenté les classes principales pour le 2ème rendu, cependant nous avons fait face à plusieurs problèmes liés à notre architecture qui n'était pas assez complète pour nous permettre d'avancer sur des modules liés à l'implémentation. Nous avons donc choisi de nous concentrer à nouveau sur la conception de l'architecture et de la repenser en fonction des contraintes auxquelles nous avons pu faire face. Nous avons rajouté des classes pour alléger les classes que nous avions préalablement établies et listé l'ensemble des méthodes et attributs dont nous pensions avoir besoin. A partir de cette nouvelle architecture plus développée, nous avons pu commencer à implémenter les premières classes.
- Rendu 3 : L'objectif était que l'application fonctionne sur console pour le rendu 3, nous nous attendions donc à avoir fini d'implémenter les classes permettant le déroulement du jeu. Cependant, nous avons fait face à plusieurs complications liées notamment à la cohérence entre les différentes implémentations de classe, développées chacune en local, qui ont provoqué des conflits de branche lors de la fusion sur la branche master.  
Notre architecture impliquait de nombreuses dépendances pouvant être évitées (notamment entre la classe Case et le plateau par exemple, où la case n'a pas besoin d'avoir accès aux pièces autres que celle sur sa case). Nous avons donc choisi de modifier notre architecture à nouveau pour aligner nos structures de code avant d'aller plus loin, quitte à prendre du retard sur le planning prévu mais nous permettant par la suite d'avancer plus facilement.  
A ce stade, nous ne possédions pas de méthode permettant de lier nos classes à une interface console et nous n'avions pas commencé l'interface graphique. Ainsi pour le 3ème rendu, notre jeu ne pouvait pas encore fonctionner en console car nous avons fait face à plusieurs erreurs que nous n'avons pas eu le temps de corriger avant le rendu.
- Rendu final : Pour ce rendu, nous avons donc choisis de diviser les tâches par binôme. L'un a continué de développer le backend tandis que l'autre a travaillé sur le frontend, c'est-à-dire

l'interface graphique. Il a en plus de ça fallu modifier à nouveaux des méthodes pour lesquelles nous nous sommes rendu compte qu'il y avait des erreurs au moment de jouer (des méthodes de déplacements, la sauvegarde qui omettait certains attributs...). Nous avons finalement pu finaliser l'interface graphique et la sauvegarde pour le 24 décembre.

En conclusion, nous avons pris du retard sur le planning initialement prévu avant le 3ème rendu car notre architecture n'était pas optimale. Cependant, nous avons revu notre architecture, réorganisé la répartition des tâches et augmenté nos réunions pour être plus efficace et rattraper le retard accumulé.

Phase	date début prévue	date fin prévue	date début réelle	date fin réelle	statu
Définition de toutes les contraintes et règles du jeu à implémenter	début de semestre	30 septembre	début de semestre	30 septembre	terminé
Conception de l'architecture	début de semestre	30 septembre	début de semestre	mi-novembre / fin-novembre	terminé
Implémentation des classes "principales"	30 septembre	5 novembres	30 septembre	fin novembre	terminé
Implémentation des classes permettant d'alléger l'architecture	20 novembre	27 novembre	20 novembre	27 novembre (nous en avons rajouté encore plus tard mais l'essentiel avait déjà été implémenté à cette date)	terminé
implémentation et développement des classes plus avancées (gestion de sauvegarde, extensions, IA...)	5 novembre	2 décembre	2 décembre	19 décembre	terminé
Jeu opérationnel en console		2 décembre		14 décembre (dans l'ensemble, sans les extensions, IA et sauvegarde) 19 décembre avec ajout des extensions...	terminé
Jeu opérationnel en interface graphique		22 décembre	15 décembre	22 décembre	terminé

## Contributions personnelles

Sur chacun des différents livrables, nous avons tous participé à la rédaction. Le livrable n°1 a été pris en main par Oscar, le 2ème par Emma, le 3ème par Evan et enfin le rendu final par Robin. Chacun a évidemment pris plus de temps et écrit davantage sur chacun de ses livrables surtout au niveau des explications. Ensuite la complétion s'est faite par tout le monde. Au niveau du rendu final c'est davantage Robin qui l'a écrit avec l'aide de Emma le pendant que Oscar et Evan continuaient le code.

Au niveau de la part de contribution de chacun d'entre nous, c'est évidemment délicat à dire puisque nous avons tous testé en local pas mal de choses avant que l'un d'entre nous y arrive vraiment et que l'on prenne sa solution. Nous pouvons quand même dire qu' Oscar (ce n'est pas lui qui écrit présentement) a beaucoup contribué. Ensuite Emma et Robin et enfin Evan. Cela pourrait donner Oscar 35%, Emma 25%, Robin 25% et Evan 15%.

Pour le nombre d'heures de travail consacré au projet:

- Emma : environ 45h
- Robin : Environ 55-60h
- Oscar : Environ 80h
- Evan : Environ 45h



## Conclusion

Ce projet, qui consistait à implémenter un jeu en version console et graphique sous Qt, a été à la fois formateur et exigeant. Il a demandé beaucoup de temps et d'efforts, surtout que notre maîtrise du code n'était pas encore optimale. Le travail a été intense, mais l'ambiance au sein du groupe est restée très bonne, ce qui a facilité la gestion de la charge de travail.

La difficulté résidait principalement dans l'organisation et la coordination que nécessite un projet comme celui-ci. En effet, il fallait faire en sorte que chaque membre du groupe puisse avancer sur sa partie. Ainsi, il fallait avoir une base cohérente, avancer de manière progressive et bien communiquer pour éviter des incompatibilités dans les différentes parties du projet.

Cette coordination impliquait une gestion claire des priorités, des objectifs communs à chaque étape, ainsi qu'une répartition des tâches en fonction des compétences et disponibilités de chacun. Les réunions régulières et l'utilisation d'outils collaboratifs, comme un gestionnaire de version (Git), nous ont permis d'intégrer nos contributions et de résoudre efficacement les conflits qui pouvaient survenir.

Malgré les défis techniques et organisationnels, ce projet nous a permis de renforcer nos compétences en développement logiciel, en gestion de projet et en travail d'équipe. Nous avons également appris à surmonter les imprévus et à ajuster nos plans pour atteindre nos objectifs. Cette expérience constitue une base solide pour aborder des projets encore plus ambitieux à l'avenir.

En point d'amélioration, nous aurions dû commencer l'interface graphique plus tôt, au moins une semaine avant ce que nous avons fait, même si le jeu n'était pas totalement opérationnel en console, pour être dans les temps. Finalement, nous avons réussi à rendre le projet complet dans le délai imparti, qui a été décalé au 24 décembre.