

**- L021 Livrable Final -**



---

**Membres du groupe :**

- BENJOUAD Yasmine
- CHAHINE Alissa
- CHEMLI Cyrine
- GABA Auriane
- MEZEMATE Ahlem

**Responsable du livrable :** CHAHINE Alissa

**Semestre :** Automne 2024

# - Sommaire -

|  |           |
|--|-----------|
| <b>1. Introduction</b>                         | <b>1</b>  |
| <b>2. Application</b>                          | <b>1</b>  |
| Fonctionnalités générales                      | 1         |
| Fonctionnalités liées aux pièces et au plateau | 1         |
| Placements, déplacements et règles spécifiques | 2         |
| Affichage d'une partie                         | 2         |
| <b>3. Architecture</b>                         | <b>4</b>  |
| Architecture Finale                            | 4         |
| La classe Coordinate                           | 4         |
| La classe Token                                | 4         |
| La classe Board                                | 5         |
| La classe Match                                | 6         |
| La classe Game                                 | 7         |
| Les classes Insectes                           | 7         |
| La classe Player                               | 9         |
| La classe IA                                   | 9         |
| Architecture graphique                         | 11        |
| La classe MainMenu                             | 11        |
| La classe VueToken                             | 11        |
| La classe GameEnd                              | 14        |
| Justification de l'Architecture                | 14        |
| Organisation modulaire                         | 14        |
| Encapsulation et polymorphisme                 | 14        |
| Utilisation des Design Patterns                | 15        |
| Évolutivité du plateau de jeu                  | 19        |
| <b>4. Planning effectif</b>                    | <b>19</b> |
| <b>5. Contributions</b>                        | <b>20</b> |
| Livrable 1                                     | 20        |
| Livrable 2                                     | 20        |
| Livrable 3                                     | 21        |
| Livrable Final                                 | 22        |
| Projet total                                   | 25        |
| <b>6. Conclusion</b>                           | <b>25</b> |
| <b>7. Annexes</b>                              | <b>26</b> |
| 1.UML Complet en mode Console:                 | 26        |
| 2.UML Complet avec interface graphique         | 27        |
| 3.UML du Design Pattern Memento:               | 28        |
| 4.UML de Token et ses classes Filles:          | 29        |

# 1. Introduction

Dans le cadre de ce projet, nous avons développé une application interactive visant à simuler et gérer le jeu de société **Hive** créé en 2001 par John Yianni. Ce projet s'inscrit dans un contexte pédagogique où la conception, l'implémentation et l'organisation d'une application en suivant le paradigme de programmation orientée objet sont au cœur des objectifs.

L'application repose sur une architecture modulaire structurée en différentes couches, permettant ainsi une séparation claire des responsabilités fonctionnelles entre les différents modules. Chaque module offre un ensemble d'API (Application Programming Interface) à consommer par d'autres modules sans se soucier de leur implémentation. Cette approche favorise la maintenabilité de l'application, l'évolutivité d'un module indépendamment des autres et ainsi la lisibilité du code.

Le projet est divisé en plusieurs modules principaux, chacun implémenté par une ou plusieurs classes dédiées. Le recours aux Design Patterns a été favorisé durant l'implémentation afin d'aboutir à un code maîtrisable et générique au maximum. Dans ce rapport, nous détaillerons ces modules, ainsi que les relations entre les classes et leurs rôles respectifs. Nous mettrons également en avant le planning suivi tout au long de ce projet et la manière dont nous nous sommes organisées afin de garantir un développement progressif et maîtrisé de l'application.

# 2. Application

Notre application permet de jouer au jeu de société Hive en console ainsi que sur interface graphique. Voici un récapitulatif des opérations attendues :

## Fonctionnalités générales

### *Gestion des joueur.euse.s*

- Création de parties avec deux joueur.euse.s humain.e.s ou un joueur.euse humain.e contre une IA.
- Choix des paramètres des joueur.euse.s (nom, joueur.euse humain.e/IA, IA facile ou difficile).

### *Sauvegarde et reprise des parties*

- Pause de la partie en cours pour une reprise ultérieure.
- Possibilité de retour en arrière et de fixer le nombre de retours possibles.

### *Fin de partie*

- Détection de la fin de partie (Reine Abeille encerclée).

## Fonctionnalités liées aux pièces et au plateau

### *Jeu de base*

Gestion des 11 pièces par joueur.euse (Reine Abeille, Araignées, Scarabées, Fourmis, Sauterelles).

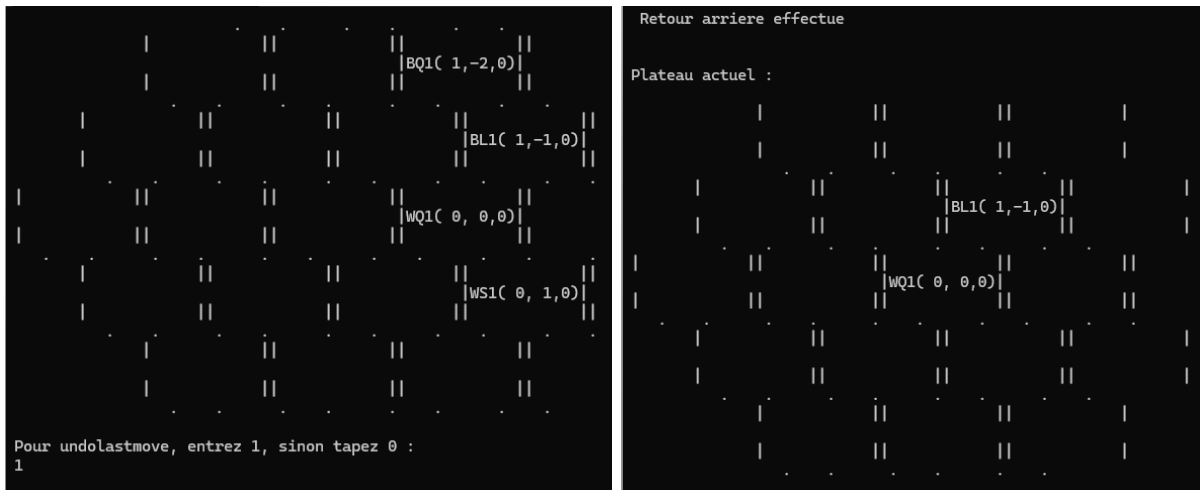
### *Gestion des extensions*

Intégration de deux extensions parmi les trois disponibles (Moustique et Coccinelle) et choix du nombre d'extensions au début de la partie (aucune, les deux, ou une seule).

### *Plateau dynamique*

- Création et affichage d'un plateau dynamique formé par l'agencement des pièces.
- Respect de la règle de la Ruche unique (les pièces doivent toujours former un bloc continu).





**Fin de la partie :** L'utilisateur a le choix entre recommencer une partie avec les mêmes paramètres ou quitter.

```
Le gagnant est le joueur IA Player
Voulez vous rejouer? 0 pour non, 1 pour oui :
1
Combien d'extensions souhaitez-vous ajouter ? 0/1/2 1
Pour choisir ladybug, tapez 0
Pour choisir mosquito, tapez 1 1
Quel est le nombre de retour en arriere autorises dans la partie ?
2
Le match va commencer !
Tour du joueur: Alissa
Tapez 0 pour placer, 1 pour deplacer :

Le gagnant est le joueur IA Player
Voulez vous rejouer? 0 pour non, 1 pour oui :
0
Merci d'avoir joue. Fermeture de l'application...
```

### Interface Qt

- Le menu permet d'initialiser le jeu avec les paramètres transmis par l'utilisateur
- L'interface du jeu permet de jouer de la même manière qu'en console :
  - On peut jouer à 2 joueurs ou 1 joueur contre une IA (niveau *easy* ou *hard*)
  - le pseudo de la personne qui doit jouer (`currentPlayer`) est affiché
  - lorsqu'on clique sur une pièce de la main ou du plateau, les cases sur lesquelles on peut placer ou déplacer la pièce sont affichées sur le plateau
  - les règles de placement et déplacements sont respectées (par exemple obligation de placer la Bee au 4ème tour si elle n'est pas déjà sur le plateau)
  - lorsque le match est gagné on arrive sur un menu de fin qui nous propose de soit rejouer avec les mêmes paramètres, soit recommencer le jeu avec de nouveaux paramètres, soit quitter l'application.
- L'interface du jeu donne aussi accès à un bouton pause qui permet de reprendre la partie en cours, recommencer un match, recommencer un jeu ou quitter l'application.

### Non implémenté sur Qt

- Nous n'avons pas implémenté le retour arrière (undo qui utilise le pattern Memento) sur l'interface Qt. Pour cela nous aurions pu reprendre la logique utilisée pour le jeu en console avec `savestate` et `undolastmove` en modifiant les `vueTokensMatch` puis en réinitialisant les scènes du board et des mains.

### 3. Architecture

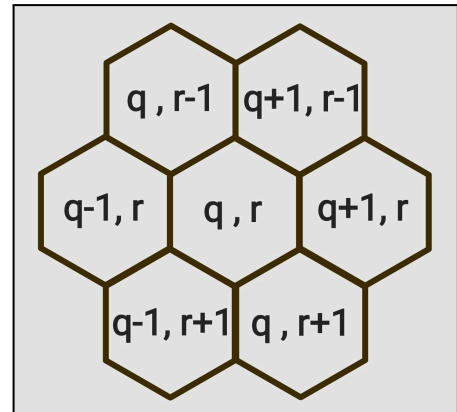
#### Architecture Finale

Les schémas UML des Annexes 1 et 2 représentent notre architecture finale respectivement en mode console ainsi qu'avec l'interface graphique.

##### *La classe Coordinate*

Pour la gestion des coordonnées des jetons, nous avons décidé d'utiliser un système de coordonnées hexagonal comme dans l'illustration ci-contre (où  $q=x$  et  $r=y$ ).

Cette classe est constituée de trois attributs :  $x$ ,  $y$ , et  $z$ , des entiers initialisés à 0. L'attribut  $z$  correspond à la hauteur du jeton (0 s'il est sur le plateau et 1 s'il est sur un autre jeton). La classe possède des accesseurs en lecture et en écriture pour les trois coordonnées. Nous avons surchargé les opérateurs **operator==**, **operator!=**, **operator<**, **operator+** au fur et à mesure de l'implémentation des différentes classes d'insectes lorsque cela a été nécessaire. Nous avons également implémenté la méthode **getAdjacentCoordinates** qui retourne un vecteur de toutes les coordonnées adjacentes à l'objet Coordinate. La méthode statique **hexDistance** calcule quant à elle la distance entre deux coordonnées. Cette dernière est utilisée dans notre classe IA.



##### *La classe Token*

La classe Token (jeton, pièce du jeu) est la classe mère des insectes.

Cette classe contient des attributs qui définissent pour chaque Token : sa couleur, son type d'insecte, son unique-pointeur vers une coordonnée (nullptr par défaut), son état (par défaut inactive car pas encore placée), son id (pour différencier lorsqu'il y a plusieurs pièce du même insecte), et si la pièce peut uniquement glisser ou non.

La classe token contient des getter et des setter pour ses attributs et les méthodes suivantes :

- **virtual bool canMove() const;**

Cette méthode virtuelle vérifie si le token peut être déplacé ou non. Pour cela, des test sont effectués sur chaque condition de déplacement défini par les règles de Hive. Si le token est inactif c'est à dire qu'il n'est pas sur le plateau on ne peut pas le déplacer, seulement le placer. La méthode vérifie aussi que les déplacements ne sont permis qu'une fois que la Queen Bee (de la couleur du Token) est placée.

- **vector<Coordinate> getPossiblesPlacements() const;**

Cette méthode calcule les placements possibles et les retourne sous forme de vecteur de coordonnées. Pour cela, la méthode vérifie d'abord s'il s'agit du premier ou du deuxième placement de la partie puisqu'ils fonctionnent différemment. Le seul premier placement d'une partie est la coordonnée (0,0,0), et le deuxième placement de la partie peut être sur n'importe quelle coordonnée adjacente à (0,0). Pour la suite du match afin de respecter les règles, la méthode renvoie comme coordonnées de placements possibles uniquement celles où la pièce ne serait à côté d'aucune pièce de couleur opposée. Aussi, si il existe une pièce à une hauteur supérieure (exemple : Beetle à la hauteur 1), c'est celle-ci qui sera considérée pour calculer les placements possibles (ie : si une pièce blanche est sur une pièce noire, la

case sera considérée comme était occupée par une pièce blanche à côté de laquelle on ne peut pas placer de pièce noire).

- **virtual vector<Coordinate> getPossibleMoves() const = 0;**

Cette méthode virtuelle pure calcule les placements possibles de this Token.

- **virtual vector<Coordinate> getPossibleMovesByCoordinate(const Coordinate& c) const = 0;**

Cette méthode virtuelle pure calcule, à partir d'une coordonnée passée en paramètres, les coordonnées possibles pour le déplacement de l'insecte en particulier, et retourne un vecteur de coordonnées.

- **bool canSlideWithData(const Coordinate& coord, const Coordinate& dest, map<Coordinate, Token\*>& board) const;**

Cette méthode vérifie si le Token peut glisser d'une coordonnée de départ vers une destination d'arrivée en se basant sur la map passée en paramètres.

- **bool canSlide(const Coordinate &coord, const Coordinate &dest) const;**

Cette méthode vérifie si le Token peut glisser d'une coordonnée de départ vers une destination en se basant sur la map de l'instance de Board.

- **Token\* deepCopy() const=0:**

Cette méthode appelle le constructeur par recopie pour cloner les insectes et renvoyer un Token\*.

### *La classe Board*

Le plateau est modélisé à l'aide d'une map (**std::map<Coordinate, Token\*>**), où chaque clé (**Coordinate**) représente une position sur le plateau, et chaque valeur (**Token\***) est un pointeur vers un jeton (représenté par la classe abstraite Token).

- **getAdjacentTokens(const Coordinate& coord)**

Cette méthode retourne les coordonnées des cases adjacentes contenant des jetons autour d'une position donnée.

- **getDataAdjacentTokens(const Coordinate& current, map<Coordinate, Token\*> board)**

Cette méthode a un comportement similaire, mais avec une version spécifique d'un plateau passé en paramètre. Elle est utilisée dans **beehiveIsCompact**.

- **beehiveIsCompact(map<Coordinate, Token\*> data)**

Cette méthode vérifie si tous les jetons du plateau forment une ruche unique et compacte en utilisant un algorithme de propagation. Elle commence par extraire les coordonnées des jetons situés au niveau  $z=0$  dans une liste **keys**, puis sélectionne un jeton initial pour démarrer l'exploration, le plaçant dans une liste temporaire **not\_tested**. Tant que **not\_tested** contient des jetons à explorer, elle examine leurs voisins adjacents et, si ceux-ci sont présents dans **keys**, les ajoute à **not\_tested** et les supprime de **keys**. À la fin, si tous les jetons ont été visités (**keys** est vide), la ruche est compacte, sinon, il existe des jetons isolés, et la ruche n'est pas compacte.

- **void showBoard() const;**

Cette méthode affiche la map du plateau de manière plus compréhensible en console. Pour cela, l'affichage suit le fonctionnement des nos coordonnées pour afficher dans l'ordre, ligne par ligne "WQ1 (0,0,0)" (White Queen Bee id=1, x=0, y=0, z=0) au centre d'un hexagone si une pièce est présente sur le plateau, et un hexagone vide sinon. La taille du plateau augmente en fonction de la coordonnée ayant la plus grande valeur.

- **void showPlacements(Token\* token) const;**

Cette méthode affiche les placements possibles pour un token donné sur le plateau.

- **void clear() { for (int i = 0; i < data.size(); i++) data.erase(std::prev(data.end())); }**

Cette méthode supprime tous les tokens du plateau.

- **void reset()** { data.clear();}:

Cette méthode de Board réinitialise l'attribut data.

- **void setData(std::map<Coordinate, Token\*> mapVersionToRestore);**

Cette méthode restaure l'état du plateau à une version précédente. Elle restaure l'attribut data à partir d'une map passée en paramètres.

### *La classe Match*

La classe Match permet de modéliser une partie dans le jeu. Son cycle de vie est contrôlé par la classe Game. Match crée les tokens de la partie, elle doit alors gérer le cycle de vie de tous les objets Token de la partie. Elle contient les méthodes permettant de vérifier les différentes contraintes à respecter lors de la partie telles que le placement de l'abeille avant le quatrième tour.

De plus, dans la classe Match est définie la méthode **placeonBoard** permettant de placer un Token sur le Board. La classe Match est également munie d'une méthode **isGameOver** permettant de vérifier si la partie est finie. Nous avons choisi de définir ces méthodes dans la classe Match car elles sont spécifiques au déroulement de la partie. De plus, Match permet d'accéder directement aux Tokens de la partie de par ses attributs whitetokens et blacktokens, qui sont des vecteurs modélisant les Tokens respectifs du/de la joueur.euse blanc.he et noir.e. La classe Match possède des getters permettant de renvoyer les Tokens pouvant être encore placés ainsi que ceux qui sont sur le plateau. Ces Tokens seront notamment utilisés dans la classe Game et dans notre interface QT.

Afin de faciliter les méthodes de Game modélisant le déroulement de la partie, nous avons défini dans Match des méthodes d'affichage en console. Nous avons également implémenté un itérateur permettant à l'utilisateur.ice de parcourir les deux vecteurs de Token en une seule fois.

**CheckGameOver()** permet de déterminer si une des Bee a été encerclée, ce qui impose la fin de la partie. Elle vérifie que les coordonnées voisines de celle de la Bee sont bien occupées par des insectes. Les coordonnées voisines sont calculées à partir de celles de la Bee.

**placeOnBoard(Token \*token, const Hive::Coordinate &coord)** est appelée lorsque l'utilisateur.ice souhaite déplacer ou placer un insecte. Cette méthode prend en paramètre l'objet Token à déplacer ainsi que un objet Coordinate modélisant l'emplacement souhaité. Elle utilise les méthodes **verifyPlacement(Token \*token, const Coordinate &coord)** et **verifyMoves(Token \*token, const Coordinate &coord)** qui vérifient que le placement ou le déplacement du token à la coordonnée coord sont bien valide. Elle distingue ainsi les deux cas et utilise les méthodes de la classe Token **getpossiblemoves** et **getpossibleplacements**. De plus, **placeonBoard** prend en compte le placement de la Bee avant le quatrième tour, puisque la classe Match est la seule qui tient compte des tours des joueur.euse.s. Le placement ou déplacement est modélisé par l'insertion de la paire **<Coordinate,Token\*>** dans la map de la classe Board modélisant le plateau. L'attribut isActive est mis à true dès que le token est placé.

Le constructeur de la classe Match s'occupe d'allouer de la mémoire des objets Token de la partie. Ces pointeurs sont stockés dans un vecteur. Le choix du vecteur par rapport à celui d'un tableau permet de garantir l'extensibilité du code, permettant de faciliter la modification du conteneur. De plus, un vecteur permet de prévenir un possible dépassement de mémoire. Le constructeur permet de créer un nombre précis de Tokens pour chaque type d'insecte. On peut ainsi très facilement modifier le nombre d'objets Token créés pour chaque type d'insecte. Comme la classe Token est abstraite, il n'est pas possible de



créer un objet de la classe mère. Nous avons choisi de passer par une méthode **createToken(PlayerColor color, TypeInsect type, int j)** qui crée un nombre *j* de d'objets Token pour le type d'insecte et la couleur passée en paramètres.

Puisque Match alloue de la mémoire lors de la création des pointeurs sur les objets Token, il est impératif de la désallouer dans le destructeur.

### *La classe Game*

La classe **Game** dans laquelle nous avons utilisé le design pattern singleton permet de jouer à HIVE en faisant le lien entre les choix de l'utilisateur sur la console et les différentes méthodes de **Match**. Dans le fichier main, on appelle la méthode **play()** sur l'instance de Game pour commencer à jouer. Cette méthode **play()** va réaliser les actions suivantes :

- D'abord, elle appelle la méthode **setExtension()** pour demander à l'utilisateur.rice le nombre d'extension qu'il/elle veut ajouter et stocker leur type (ladybug, mosquito ou les deux).
- Ensuite, elle appelle **setPlayerIA()** qui demande si l'adversaire est humain ou une IA.
- La méthode **setPlayerNames()** permet d'entrer le nom de la ou des personnes qui vont jouer.
- Puis, **startMatch()** appelle le constructeur de la classe **Match** en fonction des paramètres de la partie : IA ou non, nombre de pièces total (en fonction des extensions) et les types d'extensions à ajouter.
- Elle crée un objet de type **Caretaker** pour gérer la sauvegarde et les retours en arrière dans la partie (design pattern **Memento**).

Une fois le match défini, **play()** permet de faire jouer à tour de rôle l'utilisateur.rice et l'IA ou la deuxième personne. Grâce à une boucle while la partie s'arrête lorsqu'il y a GameOver. Dans la boucle, on capture l'état actuel de la partie avec **saveState(Caretaker &c)**, on vérifie si le currentPlayer de match est une IA ou non et on appelle soit **playerTurn()** soit **IAturn()**. Pour **playerTurn()**, il sera demandé si l'utilisateur.rice souhaite placer ou déplacer une pièce, puis lui seront proposées les pièces qui peuvent être placées (la "main") en utilisant **getHand()**, ou déplacées en s'appuyant sur **canMove()** (de Token) **getActiveTokens()**, **getTokensMovable()**. Les positions possibles de placement ou déplacement pour une pièce particulière sont obtenus grâce aux méthodes correspondantes de chaque insecte (**getPossiblePlacements()**, **getPossibleMoves()**). Si la partie est entre deux joueurs, une fois que le joueur joue, on demande s'il souhaite annuler son coup, et dans ce cas la méthode **undoLastMove(Caretaker&c)** est appelée. Si c'est un match contre une IA, on demande à l'utilisateur.rice après qu'elle ait joué et que l'IA ait joué s'il/elle souhaite annuler son coup, et s'il/elle le souhaite, alors le coup de l'IA et le sien sont annulés et il/elle reprend son tour. Après chaque tour, l'état actuel du plateau est affiché avec **showBoard()**.

Enfin, une fois que la partie est terminée (GameOver), **play()** appelle la méthode **finishMatch()** qui affiche le gagnant.

Par la suite, une fois que la partie est finie on proposera de commencer une nouvelle partie.

### *Les classes Insectes*

Les classes d'insectes héritent de la classe abstraite Token. Ainsi, leur rôle est d'implémenter les méthodes virtuelles de Token qui concernent le déplacement des Jetons. Afin de faciliter l'implémentation de la classe Mosquito, nous avons implémenté dans chaque classe d'Insectes la classe

**getPossibleMovesByCoordinate** qui calcule les déplacements possibles de l'insecte mais qui les applique sur une coordonnée passée en paramètre. Dans **getPossibleMoves**, il nous suffit donc d'appeler cette méthode sur les coordonnées du Token.

```
//on appelle with coordinate mais avec ses propres coordinate
vector<Coordinate> Bee::getPossibleMoves() const {
    return getPossibleMovesByCoordinate(getTokenCoordinate());
}

vector<Coordinate> Bee::getPossibleMovesByCoordinate(const Coordinate& c) const {
    //on cherche les moves possibles pour une abeille
}
```

### Bee

Un objet Bee ne peut se déplacer que d'une case à chaque tour, de plus cette case doit être une case adjacente vers laquelle elle peut glisser. La méthode **getPossibleMovesByCoordinates** récupère les cases adjacentes à la case de départ de Bee et renvoie celles vides et vers lesquelles on peut glisser sans casser la ruche.

### Beetle

Tout comme l'abeille, l'objet de type Beetle ne peut se déplacer que d'une case à la fois sauf qu'il a la particularité de pouvoir se mettre au-dessus des pièces adjacentes à elle. De ce fait, ses méthodes diffèrent légèrement du reste des insectes étant donné qu'on prend en compte cette fois le niveau. On distingue donc deux cas :

- **Déplacement vers une case vide** : Si elle est adjacente à la position initiale et respecte les règles de glissement (on peut aussi "descendre" vers une case vide)
- **Monter sur une pièce adjacente** : Si la case est occupée, mais pas empilée à un niveau supérieur.

### Spider

L'objet Spider possède une capacité de déplacement particulière : il doit se déplacer exactement trois fois en glissant d'une case à une autre. Les déplacements sont soumis aux règles de la ruche compacte. La méthode **getPossibleNextSlides** vérifie quelles cases adjacentes sont accessibles par glissement à partir d'une position donnée. Par la suite la méthode **getPossibleMovesByCoordinate** utilise la première méthode pour effectuer une simulation complète des déplacements de l'araignée c'est-à-dire qu'elle calcule les trajectoires possibles en combinant trois glissements successifs.

### Ant

Un objet Ant peut se déplacer et se mettre autour de n'importe quelle pièce du plateau, du moment que la trajectoire qu'il effectue ne casse à aucun moment la ruche. La méthode **getPossibleMovesByCoordinates** dans ce cas effectue une recherche en parcourant en largeur à partir de la position initiale de l'objet Ant :

- On récupère d'abord tous les emplacements possibles, c'est-à-dire tous les voisins vides des pièces posées sur le plateau.
- Pour chaque destination éventuelle, on simule une trajectoire grâce au parcours en largeur mentionné plus haut :
  - On place le nœud de départ dans la file.
  - Tant que la file n'est pas vide :
  - On retire un nœud de la file.

- On visite ce nœud , il est validé si il est vide, la ruche reste compacte après le déplacement et la condition canSlide est validé.
- On ajoute tous ses voisins non visités à la file
- On itère jusqu'à être arrivé à la coordonnée cible ou jusqu'au moment où on a plus de voisins

### Grasshopper

L'objet de type GrassHopper est le seul qui n'effectue aucun glissement. Il saute en ligne droite au-dessus des pièces adjacentes et ne s'arrête qu'à la première case vide rencontrée dans cette direction. Il ne peut donc pas briser la continuité de la ruche et cette particularité fait qu'on ne vérifie qu'au début que la ruche ne se casse pas. La méthode **getPossibleMovesByCoordinates** définit dans un premier lieu les six directions hexagonales vers lesquelles la pièce pourrait se diriger et itère à chaque fois pour chercher les voisins de cette direction. Elle s'arrête à la première case vide.

### Ladybug

Un objet Ladybug peut effectuer des mouvements en trois étapes : deux sauts successifs sur des cases occupées, suivis d'un glissement sur une case vide. La méthode **getNextPossibleJump** identifie les cases adjacentes contenant une pièce pour les deux premiers déplacements. Pour le dernier déplacement, la coccinelle glisse sur une case adjacente vide, ce qui est implémenté par **getNextPossibleSlide**. La méthode **getPossibleMovesByCoordinate** combine ces capacités pour calculer tous les mouvements possibles. Elle effectue une recherche sur trois étapes :

- Elle identifie les premiers sauts possibles.
- Elle explore les sauts suivants possibles en excluant la case d'origine.
- Enfin, elle ajoute les chemins possibles au vecteurs de déplacements seulement si la dernière glissade est possible. Sinon, elle abandonne le chemin.

### Mosquito

Un objet Mosquito utilise les méthodes **getPossibleMovesByCoordinates** de chaque Jeton adjacent au moustique. Pour chaque jeton adjacent, la méthode récupère le type de déplacement du jeton mais l'applique à la coordonnée du moustique. Chaque déplacement est ajouté à au vecteur des déplacements possibles du moustique après avoir vérifié qu'il n'a pas déjà été ajouté.

### La classe Player

La classe Player représente un·e joueur·euse . Elle encapsule les informations propres à ces derniers (Nom, score , IA ou non). Elle est nécessaire lors du lancement du jeu et permet notamment de différencier entre une partie normale ( entre deux joueur·euse·s) et un partie contre IA .

### La classe IA

La classe IA joue contre un humain en prenant la place du/de la joueur.euse noire.

Le rôle de l'IA est d'effectuer une action aléatoire entre choisir un token parmi les tokens actifs (**choseTokenFromBoard()**) et les tokens inactifs( **choseTokenFromHand()**). Le choix du token parmi les actifs ou inactifs est également aléatoire. Pour gérer cela, nous avons implémenté la méthode **IA\_whatToMove()**.

Lors des 4 premiers tours, l'IA ne fait que placer des Tokens choisis au hasard de sa main sur le plateau, **IA\_whatToMove()** utilise alors uniquement la méthode **choseTokenFromHand()**. Si au quatrième tour sa Bee n'est pas placée, elle est placée automatiquement.

```
Tour du joueur: IA Player
quatrieme tour : l'IA est obligee de poser son abeille
choix token : black Bee
l'IA a choisi de placer un token
choix de placement de l'IA: (2,-3,0)

Plateau actuel :
```



*Au quatrième tour, l'IA place forcément la Black Bee*


Après ce quatrième tour, l'IA effectue une action au hasard entre placer et déplacer un Token. La méthode **IA\_whereToMove** prend le token choisi en paramètre. Si ce dernier est actif (donc déjà placé sur le Board), elle appelle la méthode **choseMoveOrPlacement** sur le vecteur des placements possibles du token (méthode **getPossiblePlacements** de Token), sinon elle appelle cette méthode sur ses déplacements possibles (méthode **getPossibleMoves** de Token).

L'IA a deux niveaux de difficulté pour les choix de placement et de déplacement des Tokens. Ces difficultés sont implémentées dans la méthode **choseMoveOrPlacement** :

- Niveau 1 : l'IA choisit un move au hasard parmi les placements/déplacements possibles
- Niveau 2 : l'IA choisit le move le plus proche de la White Bee (si elle est placée) parmi les placements/déplacements possibles en utilisant la méthode **calculateDistanceFromWhiteBee()**. Pour le placement et le déplacement de la Black Bee, l'IA effectue une action aléatoire et ne tente pas de la rapprocher de la White Bee.

```
Tour du joueur: IA Player
choix token : black Ant
l'IA a choisi de placer un token
distance entre (0,-2,0) et l'abeille blanche : 2
distance entre (1,-3,0) et l'abeille blanche : 3
distance entre (3,-4,0) et l'abeille blanche : 4
distance entre (2,-4,0) et l'abeille blanche : 4
choix de placement de l'IA: (0,-2,0)

Plateau actuel :
```



*Exemple de l'utilisation console de l'IA de niveau 2*

Une fois qu'un Black Token est le voisin de la White Bee, il n'est plus déplacé par l'IA de niveau 2 (utilisation de la méthode **whiteBeesIsNeighbour**).

L'utilisateur a le choix entre ces deux niveaux au début de la partie. L'IA joue contre le White Player par la méthode **IATurn** implémentée dans la classe **Game**.

## Architecture graphique

### La classe *MainMenu*



La classe **MainMenu** représente le menu principal de du jeu Hive. Elle permet aux utilisateurs de configurer les paramètres de la partie, comme le type de joueurs, le niveau de difficulté, et les extensions du jeu. Cette classe interagit avec d'autres composants du programme pour démarrer la partie en fonction des choix de l'utilisateur.

La classe **MainMenu** est dérivée de **QMainWindow** et utilise les widgets Qt pour créer une interface utilisateur interactive. Elle centralise les interactions initiales des utilisateurs avant le lancement du jeu. Les paramètres configurés par l'utilisateur dans cette interface sont ensuite transmis à d'autres classes,

notamment à la classe **Game**, qui gère la logique du jeu, et à la classe **VuePartie**, qui crée l'interface graphique de la partie en cours.

Le cœur de cette classe est la méthode **startGame** qui va nous permettre non seulement d'instancier une instance de **Match** mais aussi de gérer la création de la page qui affichera le board. Elle fait par la suite la transition entre elle-même et la page suivante grâce à une pile de navigation **StackedWidget**.

### La classe *VueToken*

La classe **VueToken** a été implémentée sur **Qt** afin de modéliser le graphisme d'un **Token**. Elle hérite de la classe **QObject** et de la classe **QGraphicsPolygonItem** afin de représenter l'hexagone. Elle possède comme attributs privés : un pointeur sur un **Token**, un pointeur sur une **Coordinate**, ainsi qu'un booléen **isClickable** qui indique si la **VueToken** est cliquable ou non.

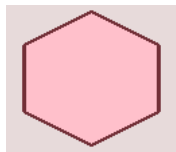
Elle est constituée de plusieurs accesseurs en lecture et en écriture pour ses attributs, permettant notamment de rendre la **VueToken** cliquable ou non, ou de mettre à jour ses coordonnées.

Le constructeur **VueToken(Hive::Token &t, QGraphicsItem \*parent = nullptr)** permet de créer un objet **VueToken** auquel un **Token** est attribué. Il correspond à une **VueToken** de la main du joueur ou à une **VueToken** placée sur le plateau. Si la **VueToken** n'est pas placée, son attribut coord (un pointeur sur **Coordinate**) pointerait sur **nullptr**.



*VueToken construit avec un pointeur sur Token*

Le constructeur est surchargé afin de permettre la création de **VueToken** modélisant les placements et déplacements possibles du **Token**. Ce constructeur ne prend en paramètre que la coordonnée où sera placée la **VueToken**, et le pointeur **token** pointera sur **nullptr**.



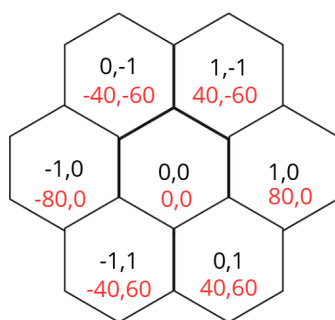
*VueToken construite avec un pointeur sur Coordinate*

**VueToken** possède une méthode protégée **mousePressEvent(QGraphicsSceneMouseEvent \*event)** qui appartient à la classe **QWidget** dans **Qt**. Lorsque l'utilisateur clique avec la souris, l'événement est transmis à cette méthode, qui émet un signal **tokenClicked(VueToken\*)**. Ce signal est perçu par la classe **VuePartie**, qui gérera le comportement après le clic.

Pour l'affichage d'une **VueToken** sur le plateau, la classe possède la méthode **affichageToken()** qui permet de placer la vue (**setPos**) à la bonne position. Pour cela, en prenant en compte la manière dont nos coordonnées s'agencent (cf. figure de la classe **Coordinate**) et la taille des hexagones sur **Qt** le calcul suivant a pu être déterminé pour la coordonnée **Qt** de l'hexagone :

$$X \text{ de l'hexagon Qt} = (\text{coord X du Token}) * 80 + (\text{coord Y du Token}) * 40$$

$$Y = (\text{coord Y du Token}) * 60$$



*Figure ci-contre : coordonnées (X,Y) du Token (noir) et (X,Y) de l'Hexagone Qt en rouge*

En effet, pour que deux hexagones soient collés verticalement, il faut décaler l'un de 60 par rapport à l'autre, pour cela seulement le **Y** du **Token** est pris en compte.

Horizontalement, sur la même ligne les **Tokens** ont la même coordonnée **Y**, il faut donc que les hexagones soient décalés de 80. Et lorsque le **Y** augmente d'un niveau (-1 ou +1), il faut en plus décaler le **Token** de 40.

### *La classe VuePartie*

La classe **VuePartie** est responsable du cycle de vie des objets **VueToken**, ce qui établit une relation de composition entre ces deux classes. Elle stocke les **VueToken** dans un vecteur nommé **vueTokensMatch**. Ces objets sont créés dans le constructeur de **VuePartie**.

La classe **VuePartie** possède un slot privé nommé **handleTokenClick(VueToken \*clickedToken)**. Lors de la création d'une **VueToken**, la méthode void **ajouterToken(VueToken \*token)** est appelée. Cette méthode connecte le signal de chaque **VueToken** au slot de la classe **VuePartie**. Ainsi, chaque fois qu'un objet **VueToken** est cliqué, le signal est reçu par la méthode membre **handleTokenClick**.

Cette méthode gère trois types de clics :

1. Clic sur une VueToken non placée sur le plateau :  
Affiche les emplacements possibles pour cette **VueToken**.

2. Clic sur une VueToken active (placée sur le plateau) :  
Permet d'afficher les déplacements possibles pour cette **VueToken**.
3. Clic sur une VueToken dont le pointeur token est à nullptr :  
Valide le placement ou le déplacement proposé en plaçant la **VueToken** aux coordonnées choisies. Les autres possibilités rejetées sont supprimées.

La classe **VuePartie** accède à l'objet **Match** via le singleton **Game**. Elle utilise la méthode **Match::placeOnBoard(Token \*token, const Hive::Coordinate &coord)** pour placer le **Token** (attribut de **VueToken**) aux coordonnées choisies (autre attribut de **VueToken**).

Un attribut supplémentaire, **Token \*toBePlaced**, a été ajouté à la classe **VuePartie** pour stocker le **Token** qui sera affecté à l'attribut de la **VueToken** modélisant le déplacement ou le placement choisi.

De plus, un attribut **VueToken \*mainstopp** a été ajouté pour gérer la mémoire des **VueToken**. Cela permet de supprimer l'ancienne **VueToken** de la scène et de désallouer sa mémoire. La désallocation est effectuée au sein de la méthode **handleTokenClick**.

L'implémentation du rôle de la méthode **play()** que nous avons dans la classe **Game** pour jouer en console a été divisée entre **handleTokenClick** et la méthode **nextTurn()**. Dans la fonction **handleTokenClick** lorsqu'une pièce a été placée ou déplacée, on appelle la fonction **nextTurn()** qui vérifie avant tout si le match est terminé, si oui cela envoie un message et appelle **finishMatch()** et sinon la fonction s'occupe de changer le **currentPlayer** et l'affiche. Si l'adversaire est une IA, la fonction **nextTurn()** appelle **IAturn()** qui fait jouer une IA, puis vérifie si le game est Over et change le **currentPlayer**. Enfin, dans **handleTokenClick** on s'occupe de rendre cliquable ou non les vues tokens en fonction du **currentPlayer**. Il y a aussi une condition qui vérifie que si la Bee du **currentPlayer** n'a pas encore été placée au 4eme tour, alors on appelle **Avertissement()** qui affiche un message informatique et rend clickable uniquement la Bee.

La classe **VuePartie** possède une méthode **finishMatch()** qui est appelée lorsqu'un des objets **Bee** est complètement encerclé. Cette méthode :

1. Affiche le gagnant.
2. Passe au menu de fin en créant un nouvel objet **QWidget**, lequel est ajouté à la pile de navigation.

La classe **VuePartie** possède également un menu Pause qui permet à l'utilisateur de suspendre temporairement la partie en cours en cliquant sur le bouton "Pause" situé à gauche du plateau. Lorsqu'une pause est activée, une fenêtre s'affiche avec quatre boutons : **Reprendre** qui ferme la fenêtre de pause et reprend le jeu là où il s'était arrêté, **recommencer le match** qui redémarre la partie depuis le début, **menu principal** qui permet de quitter la partie en cours et revenir au menu principal du jeu, **quitter** qui ferme l'application complètement.

La fenêtre de pause est modale, ce qui signifie que l'utilisateur doit interagir avec elle avant de pouvoir revenir à la fenêtre principale. Les boutons sont reliés à des actions spécifiques pour gérer le jeu, comme reprendre, recommencer ou quitter.

### *La classe GameEnd*

La classe **GameEnd** affiche l'écran de fin de partie dans l'application de jeu Hive. Elle est intégrée dans une structure **QStackedWidget** pour faciliter la navigation entre les différentes pages de l'application.



Elle donne le choix à l'utilisateurice entre trois actions possibles : rejouer le match, relancer le jeu ou quitter le jeu. Le choix entre une de ces actions trigger une des trois méthodes de la classe :

- **RestartGame()** : Réinitialise complètement l'application en supprimant tous les widgets du stackedWidget ainsi que l'instance de Match. Elle recrée ensuite menu principal.
- **ReplayMatch()** : Supprime l'état actuel de la partie et nettoie le plateau puis refait appel au constructeur de Match pour en recréer une instance avec les mêmes configurations du début. Elle vide la pile de navigation ( sauf pour le premier élément) puis crée une nouvelle instance de VuePartie qu'elle remet dedans.
- **QuitGame()** : Supprime les widgets inutilisés, puis ferme l'application.

## Justification de l'Architecture

Nous avons utilisé des concepts tels que les Design Pattern, et une organisation en couches claires pour garantir que notre application puisse être étendue sans remettre en cause le code existant.

### Organisation modulaire

Nous avons structuré notre application en quatre couches principales :

- 1) **Couche du jeu** : Gère la logique de haut niveau du jeu et coordonne les autres composants (classe Game).
- 2) **Couche de la partie** : Responsable de la gestion d'une partie spécifique, y compris le déroulement des tours et les règles de victoire (classe Match).
- 3) **Couche des éléments du plateau** : Modélisation des composants spécifiques au jeu Hive (le plateau, et les pièces du jeu). Ces composants modélisent le terrain de jeu et les interactions entre les pièces (classes Board, Token, et leurs dérivées).
- 4) **Couche utilitaires** : Gestion des fonctionnalités auxiliaires (les classes d'exceptions, les types...).
- 5) **Couche présentation** : Composée des différentes classes permettant de gérer non seulement l'affichage mais aussi les actions liées aux différents clics et signaux lancés.

Cette séparation des responsabilités permet d'isoler les changements dans une seule couche sans affecter les autres. Par exemple :

- Si nous ajoutons une nouvelle règle de jeu, celle-ci sera encapsulée dans la couche de la partie (classe Match), sans nécessiter de modification dans les couches de jeu (classe Game) ou éléments du plateau.
- Une modification dans la manière d'afficher le plateau affecterait uniquement la classe Board dans la couche éléments du plateau.

### Encapsulation et polymorphisme

#### Classe abstraite Token (voir UML en Annexe 4)

La classe Token est abstraite et sert de base pour tous les types d'insectes. Chaque sous-classe implémente ses propres règles de déplacement via la méthode **getPossibleMoves**.

- Pour ajouter un nouvel insecte, il suffit de créer une nouvelle sous-classe de Token et de définir ses comportements spécifiques.
- Les interactions avec le plateau ou les joueur.euse.s restent les mêmes, car elles utilisent des pointeurs sur des Token et non des objets des classes d'insectes dérivées.

### Polymorphisme dans les méthodes des classes



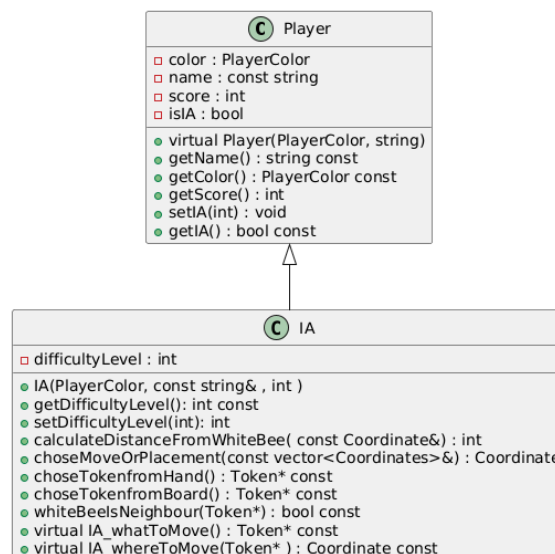
Les classes Match et Game reposent fortement sur le polymorphisme pour interagir avec les Tokens. Par exemple, lors d'un tour, la méthode **getPossibleMoves** est appelée sans se soucier du type précis d'insecte. Cela garantit que même en ajoutant de nouveaux types d'insectes, le code existant (affichage des mouvements, gestion des règles) reste fonctionnel.

Il nous est également possible d'implémenter l'extension officielle Cloporte qui déplace les places adjacentes à lui. En effet, nous pouvons utiliser la méthode **getPossibleMovesByCoordinate** qui applique le type de mouvement de l'insecte mais sur une coordonnée choisie. Nous pouvons donc redéfinir cette méthode dans la classe Cloporte, et appliquer le mode de déplacement du Cloporte (c'est à dire se déplacer d'une case, identique à la Bee) sur une pièce voisine du Cloporte et ensuite la bloquer le tour suivant en redéfinissant la méthode virtuelle canMove dans Token.

Nous avons également utilisé le polymorphisme pour la méthode virtuelle pure **deepCopy()** dans la classe Token. Cette dernière crée une copie indépendante d'un token, elle sera détaillée plus tard (Utilisation des Design Pattern: Memento).

### Implémentation de nouvelles IA

IA est une classe fille de Player, donc toutes les IA sont vues comme des Player. Cela permet à la classe Game de manipuler des objets Player, qu'ils soient des joueur.euse.s humains ou des IA. La méthode setPlayerIA de la classe Game est utilisée pour remplacer les joueur.euse.s en tant qu'IA. Ainsi, si on veut utiliser une autre IA, il suffit de faire hériter sa classe de la classe IA et d'y implémenter les méthodes virtuelles **IA\_whatToMove** et **IA\_whereToMove** qui sont utilisées dans la classe Match (**IATurn**).



### Utilisation des Design Patterns

#### Singleton

Les classes Game et Board utilisent le pattern Singleton pour garantir le fait qu'une seule instance du jeu et du plateau soit créée. Cette unique instance est alors partagée entre les différentes classes, qui n'ont alors plus besoin de la créer ou de la transmettre.

Ainsi, si on envisage une IA plus complexe, par exemple une IA qui utilise l'algorithme de MinMax, elle aura toutes les données nécessaires (état du board et main des joueur.euse.s).

#### Factory et la justification de la classe abstraite Token

La création des objets Token depuis la classe Match se base sur le Design Pattern Factory. Celui-ci permet de centraliser la création des objets, ce qui facilite la gestion des dépendances et la flexibilité dans les choix de classes instanciées.

Les composantes de ce Design Pattern sont :

- La factory : la méthode **createToken** de la classe Match pour créer des objets.
- Les produits : Les classes concrètes d'insectes à créer.
- La classe abstraite : La classe Token définissant un type commun pour les objets créés.

L'utilisation d'une classe abstraite Token permet de réduire la dépendance entre l'utilisateur.rice de cette classe et les classes concrètes d'insectes. Ainsi, il existe une réduction du couplage de la classe Match et des classes d'insectes. La modification d'un des objets insectes ne va pas affecter directement le constructeur de Match. De plus, une classe abstraite permet de rassembler les attributs et méthodes communes des insectes.

On utilise une factory modélisée par une méthode **createToken(PlayerColor color, TypeInsect type, int j)** qui permet de créer des objets Token. La classe Token étant abstraite, il n'est pas possible de créer directement des instances de cette classe. La factory doit ainsi prendre en compte le type de l'insecte à créer, le nombre d'objets à créer et leur couleur, envoyés en paramètre. Nous avons choisi de définir la Factory dans la partie privée de Match car celle-ci constitue l'unique classe à pouvoir créer des Tokens qui seront utilisés comme jetons du jeu.

### Iterator

Nous avons décidé d'implémenter le design pattern Iterator. Celui-ci doit permettre à l'utilisateur de pouvoir parcourir tous les objets Token de la partie. L'iterator permet d'étirer sur les vecteurs **blacktokens** et **whitetokens**, membres privés de la classe Match, qui contiennent l'ensemble des objets Token de la partie.

Les composantes de ce design pattern sont :

- La collection : La classe Match qui stocke un ensemble d'instances de la classe Token
- L'itérateur : La classe iterator qui regroupe les méthodes permettant de parcourir les objets contenus dans la collection

Ici, il n'était pas possible de redéfinir simplement le type de l'itérateur car plusieurs méthodes diffèrent de celles de la classe standard.

Ainsi, il faut principalement définir les méthodes :

- **operator\*()** : Cette méthode permet de renvoyer une référence sur l'objet Token
- **operator++()** et **operator--()** : Ces méthodes doivent prendre en compte le basculement d'un vecteur à un autre lorsqu'on arrive à une de leurs extrémités.

L'utilisation du pattern Iterator permet d'itérer sur une collection sans révéler ses détails internes favorisant son encapsulation.

De plus, les responsabilités sont davantage séparées lorsque la classe Match gère le stockage des éléments, tandis que l'itérateur gère leur parcours.

Enfin, si nous décidons de modifier le type de la collection, l'utilisateur.rice n'aura pas besoin de modifier l'ensemble de ses méthodes. Le code devient alors davantage extensible.

### Memento

L'utilisation du pattern Memento dans la classe Match permet de sauvegarder et de restaurer l'état du jeu sans impacter les autres classes. Notre objectif était de créer une méthode **undoLastMove()**

dans Match afin de permettre à un.e utilisateur.rice de faire un retour en arrière. Pour les parties à deux joueur.euse.s, lorsqu'un.e joueur.euse place une pièce, il/elle a le choix de changer d'avis, annuler son coup et placer ou déplacer une autre (ou la même) pièce.

Dans le cadre d'une partie contre une IA, le/la joueur.euse joue, l'IA joue, puis le/la joueur.euse peut décider de faire un retour en arrière et retourner au stade de la partie ou l'IA n'a pas encore joué, et il/elle peut changer son coup. Notre solution permet de faire des retours en arrière jusqu'au début de la partie, mais nous avons décidé de fixer un nombre limité de fois où le/la joueur.euse peut faire un retour en arrière, et ce nombre est fixé en début de partie en demandant à l'utilisateur.trice.

Nous nous sommes alors questionnées sur le meilleur moyen de sauvegarder les différents états d'une partie et de permettre des retours en arrière à des états précédents de la partie, tout en respectant l'encapsulation et une gestion sécurisée des accès aux différents attributs des classes et en préservant un code maintenable et propre. Nous avons opté pour l'implémentation du design pattern Memento.

Le design pattern Memento permet de capturer l'état de la partie en créant un objet de type **Memento**, qui sera empilé dans un objet de type **Caretaker**, puis dépilé pour restauration selon l'UML disponible en [annexe 3](#).

Les informations sur la partie (attributs de Match) à sauvegarder à chaque fois sont: le/la joueur.euse courant.e (**currentPlayer**), les tokens du/de la joueur.euse blanc.he (**whitetokens**), les tokens du/de la joueur.euse noire (**blacktokens**), le nombre de coups des joueur.euse.s (**WhiteMoveCount**, **BlackMoveCount**), et l'information sur le placement de la reine abeille (**WhiteBeePlaced**, **BlackBeePlaced**). Il faut aussi prendre une capture de l'état du plateau (l'attribut **data** de board), qui est un attribut statique.

Ce design pattern garantit une séparation des responsabilités de chacune des classes. Le **Originator**, ici **Match**, est responsable de la création d'un objet **Memento** (relation de **composition**), avec la méthode **saveMatchToMemento()** dans Match qui crée un pointeur sur un objet Memento à partir des attributs de l'objet Match à sauvegarder et le retourne.

La classe **Memento** encapsule cet état de manière immuable et protégée, car l'utilisateur.rice n'accède jamais directement à Memento. Il/Elle accède aux méthodes de Match qui utilisent Caretaker comme intermédiaire pour utiliser les objets Memento.

La classe **Caretaker** (gestionnaire de l'application), gère le stockage des objets Memento qui lui sont transmis à travers une **pile**. Caretaker n'a pas besoin de connaître les détails de la classe Match. Avec cette implémentation, on respecte le **principe d'encapsulation**. Les attributs de Match restent protégés.

Memento est aussi étendable, on peut choisir les attributs à sauvegarder. Il est aussi facile de changer les attributs de Match sans devoir tout modifier. On peut aussi fixer le nombre de retours en arrière qu'on veut.

**Gestion de la recopie indépendante des instances de token :** Afin de sauvegarder un état de match, il est nécessaire d'assurer une recopie indépendante des token afin de pouvoir recopier un état des vecteurs de Token\* **whitetokens** et **blacktokens**. Pour cela nous avons créé la méthode **deepCopy()** virtuelle pure dans la classe Token, redéfinie dans les classes filles, et mis en place un constructeur par recopie.

La classe Memento est interne à Match et ses attributs doivent stocker les attributs de Match que l'on souhaite sauvegarder. **DeepCopy()** a été utile pour le constructeur de Match afin d'initialiser les attributs de whitetokensState et blacktokensState avec des copies indépendantes des vecteurs de Match, ainsi que pour les getters de Memento qui seront utilisés pour restaurer les attributs de Match à partir de ceux d'un objet Memento (dans **RestoreMatchFromMemento()**).

Ensuite nous repassons aux méthodes de Match. **saveMatchToMemento()** crée et retourne un pointeur sur objet Memento à partir des attributs actuels de Match et **RestoreMatchStateFromMemento()** prend en paramètre un shared\_ptr sur un objet Memento et remplace les attributs actuels de Match par ceux de Memento.

Enfin, nous implémentons la classe Caretaker, qui va gérer l'historique de la partie à l'aide d'une pile d'objets shared\_ptr sur Memento. Les pointeurs permettent de pouvoir gérer la pile vide en retournant un nullptr.

Finalement, nous pouvons à présent implémenter les deux seules méthodes de Match appelées lors de la partie: **void saveState(Caretaker &c)** et **void undoLastMove(Caretaker &c)**. **saveState()** construit un shared\_ptr sur un objet Memento créé à partir des attributs actuels de Match, et l'empile dans la pile history de l'objet Caretaker passé en paramètres.

```
void Hive::Match::saveState(Caretaker& c) {
    std::shared_ptr<Hive::Match::Memento> mementoMatch = Match::saveMatchToMemento();
    //cout << "match saved to memento \n";
    c.addMemento(mementoMatch);
    //cout << "match stacked to history";
}
```

**undoLastMove()** vérifie que le/la joueur.euse n'a pas dépassé le nombre de fois possible de retour en arrière dans la partie (fixé dans Game.cpp) et restaure l'état du match à partir du sommet de la pile de l'objet Caretaker passé en paramètres.

```
void Hive::Match::undoLastMove(Caretaker& c) {
    if (currentplayer == Hive::PlayerColor::White && whiteUndoCount < 1) {
        //cout << "Vous n'avez plus de retour arriere possible :( \n";
        return;
    }
    else if (currentplayer == Hive::PlayerColor::Black && blackUndoCount < 1) {
        //cout << "Vous n'avez plus de retour arriere possible :( \n";
        return;
    }
    else {
        std::shared_ptr<Hive::Match::Memento> m = c.getLastState(); //on depile
        if (m != nullptr) {
            RestoreMatchStateFromMemento(m);
            //cout << "match state restored \n" << endl;
        }

        if (currentplayer == Hive::PlayerColor::White) {
            whiteMoveCount--;
        }
        if (currentplayer == Hive::PlayerColor::Black) {
            blackMoveCount--;
        }
    }
}
```

## Évolutivité du plateau de jeu

La modélisation du plateau comme un dictionnaire (map<Coordinate, Token\*>) rend sa structure flexible : Si nous décidons de modifier la disposition des cases (par exemple, passer à une grille non hexagonale), cela n'affectera que la classe Coordinate et les méthodes associées dans Board.

## 4. Planning effectif

| Phase  | Durée                                       | Tâches réalisées  |
|--|---|---|
| <b>Phase 1</b> :<br>Initialisation                     | 4 semaines<br>(Septembre)                   | → Apprendre à jouer.<br>→ Identifier les fonctionnalités essentielles au fonctionnement correct du jeu et les fonctionnalités supplémentaires obligatoires.<br>→ Définir l'organisation du projet, la gestion du travail de groupe, et les outils.<br>→ Rechercher les design patterns utiles au projet.<br>→ <i>Livrable 1</i> (semaine du 30 Septembre)   |
| <b>Phase 2</b> :<br>Planification                      | 2 semaines<br>(début Octobre)               | → Élaborer un plan de projet ( tâches, difficulté)<br>→ Estimer la durée de chaque tâche.<br>→ Répartir les responsabilités au sein de l'équipe du projet.  |
| <b>Phase 3</b> :<br>Développement de base              | 3 semaines<br>(fin Octobre, début Novembre) | → Premières réflexions sur l'architecture et premiers essais d'implémentation.<br>→ Implémentation des règles de bases. <ul style="list-style-type: none"> <li>• Logique des coordonnées pour le plateau Hexagonal</li> <li>• Logique du Board et de la map</li> <li>• Placement des Tokens sur le board en respectant la contrainte de couleur</li> <li>• Placement obligatoire de la Bee au quatrième tour</li> <li>• Début d'implémentation des insectes</li> <li>• Début d'implémentation de la règle de la ruche unique</li> <li>• Architecture de la classe Game</li> </ul> → <i>Livrable 2</i> (semaine du 5 Novembre) |
| <b>Phase 4</b> :<br>Développement de la logique du jeu | 4 semaines<br>(Novembre)                    | → Implémentation des Insectes et de leur méthode pour obtenir les déplacements possibles.<br>→ Implémentation des règles du jeu avancées. <ul style="list-style-type: none"> <li>• Obtenir les placements possibles</li> <li>• Commencer un match</li> <li>• Choix des extensions</li> <li>• Choix de player IA ou non</li> <li>• Obtenir les Tokens dans la main</li> <li>• Obtenir les Tokens sur le plateau</li> <li>• Gestion de la fin de partie</li> </ul>  |
| <b>Phase 5</b> :<br>Tests et Interface                 | 2 semaines<br>(début Décembre)              | → Implémentation d'une IA à deux niveaux de difficulté.<br>→ Début d'implémentation du design pattern Memento pour annuler des actions.   |

|  |                                 |  |
|--|---------------------------------|--|
| Console  |                                 | →affichage de Board en mode console.<br>→ <i>Livrable 3</i> (semaine du 2 Décembre)  |
| <b>Phase 6</b> :<br>Tests et<br>Interface<br>Graphique | 1 semaines<br>(fin<br>Décembre) | →Sur Console: test et modification des méthodes de jeu avec le retour en arrière.<br>→Interface QT. <ul style="list-style-type: none"> <li>• Page graphique du Jeu</li> <li>• Affichage graphique d'un Token</li> <li>• Affichage de la main des joueur.euse.s</li> <li>• Affichage du Board</li> <li>• Lien entre différentes pages</li> <li>• Gestion des clics sur Tokens</li> <li>• Gestion des Tokens cliquables ou non</li> <li>• Affichage des déplacements possibles</li> <li>• Gestion d'une partie (test game Over, nextTurn)</li> <li>• Gestion du tour de l'IA</li> <li>• Menu "Pause"</li> <li>• Menu "Nouveau Match"</li> <li>• Menu "Fin de Match"</li> </ul> → <i>Livrable Final</i> (22 décembre) |

## 5. Contributions

### *Livrable 1*

| Membre                            | Temps estimé                                   | Tâches contribuée  |
|-----------------------------------|--|--|
| Yasmine<br>(responsable livrable) | 5h + 4h de<br>réunion/déco<br>ouverte projet   | →Réflexion sur les fonctionnalités de base du jeu<br>→Recherche sur les design patterns<br>→Rédaction de la partie "Fonction de base du jeu" du livrable 1 |
| Alissa                            | 5h + 4h de<br>réunion/déco<br>ouverte projet   | →Rédaction des parties "Organisation", "Difficultés et Pistes",<br>"Répartition des tâches estimées" du Livrable 1<br>→Recherche sur les design patterns   |
| Cyrine                            | 6h30 + 4h de<br>réunion/déco<br>ouverte projet | →Rédaction des parties difficultés, fonctions supplémentaires et<br>UML<br>→Recherche sur les design patterns  |
| Auriane                           | 5h + 4h de<br>réunion/déco<br>ouverte projet   | →Rédaction des parties Fonctions de base du jeu, Prototype Figma du<br>jeu<br>→Recherche sur les design patterns   |
| Ahlem                             | 6h30 + 4h de<br>réunion/déco<br>ouverte projet | →Recherche sur les design patterns<br>→Rédaction des parties Phases du projet , organisation (tâches),<br>Modèle conceptuel                                |

### *Livrable 2*

| Membre | Temps<br>estimé | Tâches contribuée |
|--------|-----------------|-------------------|
|--------|-----------------|-------------------|

|                                |                       |  |
|--------------------------------|-----------------------|--|
| Yasmine                        | 20h30 +9h de réunions | <ul style="list-style-type: none"> <li>→Premiers essais d'implémentation de l'architecture du projet</li> <li>→Classe Beetle, implémentation de la méthode de déplacements possibles</li> <li>→Réflexion sur la mise en place du design pattern memento pour pouvoir mettre en place la méthode undoLastMove() et livrable</li> </ul>  |
| Alissa                         | 17h30+9h de réunions  | <ul style="list-style-type: none"> <li>→Création de la classe Coordinate et de ses méthodes</li> <li>→Création de la première version des classes Token et Insect</li> <li>→Création de la première version de la classe Board et de ses méthodes pour ajouter un Token, obtenir ses voisins, vérifier la contrainte de couleur lors du placement, du placement de la reine au 4eme tour et de la règle de la ruche unique</li> <li>→Création de la classe Bee qui hérite de Token, et début d'implémentation de sa méthode pour calculer les déplacements possibles</li> <li>→Création de guides pour l'utilisation de map, vector, et les commandes principales GIT</li> </ul> |
| Cyrine                         | 15h30 +9h de réunions | <ul style="list-style-type: none"> <li>→Premiers essais d'implémentation de l'architecture du projet</li> <li>→Réflexion et création de la première version de la classe Game : architecture de la classe et définition des méthodes de base (constructeur, destructeur, getter..)</li> <li>→Création de la première version de la classe Match : réflexion sur l'architecture de la classe, définition de toutes les méthodes (apart undoLastMove)</li> <li>→Versions UML de l'architecture actuelle</li> <li>→Rédaction Livrable 2 : justification de l'architecture de Game et Match</li> </ul>   |
| Auriane (responsable livrable) | 19h+9h de réunions    | <ul style="list-style-type: none"> <li>→Premiers essais d'implémentation de l'architecture du projet</li> <li>→Évolution de la définition de la classe Token et des sous-classes d'insectes, classe Player</li> <li>→Début d'implémentation des méthodes de Token et de la classe Spider</li> <li>→Mise au point de l'algorithme pour calculer les déplacements possibles de Spider</li> <li>→Versions UML de l'architecture actuelle et rédaction Livrable 2</li> </ul>   |
| Ahlem                          | 15h30+9h de réunions  | <ul style="list-style-type: none"> <li>→Premiers essais d'implémentation de l'architecture du projet</li> <li>→Autre Méthode qui vérifie la condition de la ruche unique</li> <li>→Création de la classe Ant qui hérite de Token, et de sa méthode pour calculer les déplacements possibles</li> <li>→Versions UML de l'architecture actuelle</li> <li>→Rédaction Livrable</li> </ul>  |

### Livrable 3

| Membre  | Temps estimé         | Tâches contribuée   |
|---------|----------------------|---|
| Yasmine | 33h + 7h de réunions | <ul style="list-style-type: none"> <li>→Implémentation des méthodes de la classe Game: startMatch, setExtension, setPlayerIA</li> <li>→GameException</li> </ul> |



|                              |                        |   |
|------------------------------|------------------------|---|
|                              |                        | →Modification et débogage de Beetle.cpp<br>→Création de la méthode virtuelle deepCopy pour tous les insectes, surcharge du constructeur par recopie pour token<br>→Documentation et implémentation et modification/debogage du design pattern Memento dans match (classe MatchMemento), implémentation des méthodes saveState et undoLastMove<br>→Débogage sous Visual Studio Code, résolution d'erreurs de linkage et de compilation sous linux, et débogage et test des méthodes à travers un main<br>→Livable  |
| Alissa                       | 31h + 7h de réunions   | →Fin d'implémentation et tests de beehivelsCompact<br>→Fin d'implémentation de la classe Bee<br>→Implémenté la classe Ladybug<br>→Implémenté la classe Mosquito<br>→Implémenté la classe IA<br>→Rédaction de la partie "La classe IA" dans le Livable 3   |
| Cyrine                       | 38h15 + 7h de réunions | →Finir la classe Match<br>→Main pour tester Match<br>→Débogage Visual<br>→Implémentation de MatchException<br>→ Modification des méthodes de la classe Game<br>→modification de l'UML et Livable classe Match   |
| Auriane                      | 33h+ 7h de réunions    | →Méthodes de Token et de la classe Spider (getNextPossibleSlides, getPossibleMoves)<br>→Mise à jour de l'architecture de Token et les insectes, fonctions canMove de Token, getPossiblePlacement<br>→Modification de l'architecture des classes Tokens, Board, Coordinate, Spider<br>→Réflexion et implémentation de l'affichage du plateau hexagonal en console (méthode showBoard)<br>→Mise à jour de Match : getHand, getTokensOnBoard<br>→Fonction play de Game afin de pouvoir jouer (placer, déplacer, choix de position) contre un joueur humain avec l'affichage en console, méthode pour demander nom joueurs<br>→Modifications de Game, Match, Player pour adapter la présence d'une IA<br>→Tests et débogage, Mise à jour de Game, insectes, match<br>→Livable rédaction de la classe Game |
| Ahlem (responsable livrable) | 15h+ 7h de réunions    | →Implémenter classe Ant<br>→Implémenter classe Grasshopper<br>→Installation et Documentation QT + début correction du code pour l'adapter à l'interface graphique<br>→Livable   |

### Livrable Final

| Membre | Temps | Tâches contribué |
|--------|-------|------------------|
|--------|-------|------------------|



|                               | estimé               |  |
|-------------------------------|----------------------|--|
| Yasmine                       | 60h + 5h de réunions | <ul style="list-style-type: none"> <li>→ Implémentation du design pattern Memento: (implémentation + modification + gestion de problèmes de mémoire + débogage et tests )</li> <li>→ Finalisation de son ajout dans les fonctions du jeu</li> <li>→ Rédaction du rapport partie Memento, Game, Token, conclusion</li> <li>→ Documentation Doxygen</li> <li>→ Montage de la vidéo</li> </ul>  |
| Alissa (responsable livrable) | 62h + 5h de réunions | <ul style="list-style-type: none"> <li>→ Amélioration de la gestion de la mémoire du code en mode console avec ajout de unique pointers</li> <li>→ Création sur QT du visuel des tokens et des tokens dans la main</li> <li>→ Création sur QT du visuel d'une partie avec le plateau et la main des joueur.euse.s</li> <li>→ Amélioration du mode console avec la possibilité de recommencer une partie</li> <li>→ Création du bouton et du mode pause sur QT</li> <li>→ Tests du jeu en mode console et résolution d'erreurs de fonctionnement</li> <li>→ Nettoyé l'entièreté du code en mode console, ajout de commentaire, suppression commentaire/code inutile</li> <li>→ Merge des versions QT et mode console pour obtenir le code final</li> <li>→ Rédaction des parties "Application", "Justification de l'Architecture", "La classe IA", "Board", "Coordinate", "Planning Effectif" du Livrable Final</li> </ul>  |
| Cyrine (responsable vidéo)    | 62 + 5h de réunions  | <ul style="list-style-type: none"> <li>→ Ajout possibilité de rajouter une partie</li> <li>→ Ajout de la classe Iterator dans la classe Match</li> <li>→ Mise à jour des classes Game et Match pour tenir compte des erreurs rencontrées</li> <li>→ première réflexion sur l'architecture à modifier sur QT et essai création de Token</li> <li>→ Implémentation de la classe VueToken sur QT</li> <li>→ méthode handleClick afin de gérer le clic d'une VueToken</li> <li>→ méthode FinishMatch pour passer à la fenêtre suivante</li> <li>→ résolution des erreurs sur QT de handleClick</li> <li>→ debug des méthodes de la classe GameEnd et Menu</li> <li>→ résolution problème de mémoire</li> <li>→ Rédaction du rapport : architecture de Match, VuePartie, VueToken. Justification du choix d'implémentation des design patterns Iterator et Factory</li> <li>→ Vidéo(responsable) : screen record du déroulement de 2 parties avec explication des connexions entre signaux et slots et Concevoir la structure principale de la vidéo</li> </ul> |
| Auriane                       | 62h + 5h de réunions | <ul style="list-style-type: none"> <li>→ Mise à jour de Spider</li> <li>→ ajouts de tests sur les cin pour faciliter le jeu en console</li> <li>→ Débogage et modification de beehive is compact pour être adapté à des pièces de hauteur 1</li> <li>→ MAJ de Token pour la règle de déplacement après bee placed et prise en compte des pieces de hauteur 1 pour le placement à côté d'une même couleur, échanges</li> <li>→ Élaboration des tâches suivantes</li> <li>→ <b>Qt :</b></li> </ul>   |

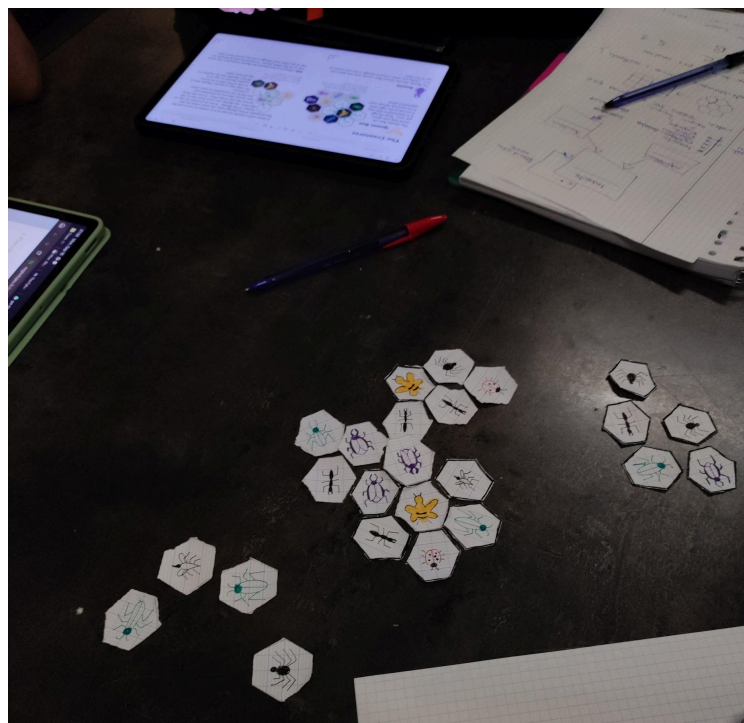
|       |                      |   |
|-------|----------------------|---|
|       |                      | <ul style="list-style-type: none"> <li>• Réflexion et implémentation de l’affichage du plateau (les positions des VueToken)</li> <li>• Tests et finalisation du visuel</li> <li>• Adaptation de Qt pour vuePartie</li> <li>• Partie de VuePartie qui permet d’afficher l’état initial du jeu depuis les données de Game et Match, avec nouvelle méthode dans Game pour tester en attendant le menu</li> <li>• Interactions entre les pages, comment passer d’une page à l’autre</li> <li>• VuePartie : adaptation de la fonction play en console dans handleTokenClick (game Over, nextTurn) pour 2 joueurs</li> <li>• Implémentation de IA turn pour jouer contre une IA sur Qt</li> <li>• Echanges, debug et tests : <ul style="list-style-type: none"> <li>○ handleTokenClick, menu</li> <li>○ fonctions pour quitter le jeu et rejouer partie en console et Qt</li> <li>○ placement/déplacement de Beetle, finalisation de jeu gagné Qt et IA</li> </ul> </li> </ul> <p>→Réflexion et debug de Memento, implémentation du choix du nombre de retour arrière en console dans Game</p> <p>→Documentation du projet et nettoyage code</p> <p>→Vidéo : partie placements/moves</p> <p>→Montage vidéo</p> <p>→Livrable final rédaction parties Game, affichage VueToken, fonctionnement d’une partie dans VueToken</p> |
| Ahlem | 35h + 5h de réunions | <p>→Création du menu principal</p> <p>→Link menu principal avec les classes nécessaires et →gestion navigation</p> <p>→Débogage</p> <p>→Création de la page de fin</p> <p>→Rédaction du rapport “introduction” , “classes insectes” ,</p> <p>→“MainMenu” et “Game End”</p> <p>→Mise à jour des UML</p> <p>→Screenrecord vidéo avec explication du code</p>  |

## Projet total

| Membre  | Pourcentage de contribution | Nombre d'heures totales |
|---------|-----------------------------|-------------------------|
| Yasmine | 22%                         | 118h30 + 25h de réunion |
| Alissa  | 22%                         | 115h30 + 25h de réunion |
| Cyrine  | 22%                         | 122h + 25h de réunion   |
| Auriane | 22%                         | 119h + 25h de réunion   |
| Ahlem   | 12%                         | 72h + 25h de réunion    |

## 6. Conclusion

Ce projet de développement d'une application interactive pour le jeu de société Hive a permis d'appliquer de manière concrète les principes de la programmation orientée objet, tout en explorant des concepts avancés tels que la modularité, la gestion des interfaces, ainsi que l'implémentation de Design Patterns. La structuration de l'application en modules distincts a facilité la gestion des différentes fonctionnalités, offrant ainsi une architecture évolutive, maintenable et facilement compréhensible. Cette expérience était également pour nous l'occasion d'apprendre à gérer, organiser et mettre en œuvre un projet informatique du début à la fin, à travailler en autonomie et nous avons réussi à avoir une bonne dynamique et cohésion de groupe tout au long des défis et difficultés rencontrées. A travers les réunions hebdomadaires auxquelles se sont ajoutées des sessions de travail en groupe quotidiennes vers la fin, notre communication et notre cohésion ont été un facteur clé qui nous a permis de concrétiser ce projet.



*Première game night - 27 Septembre*

## 7. Annexes

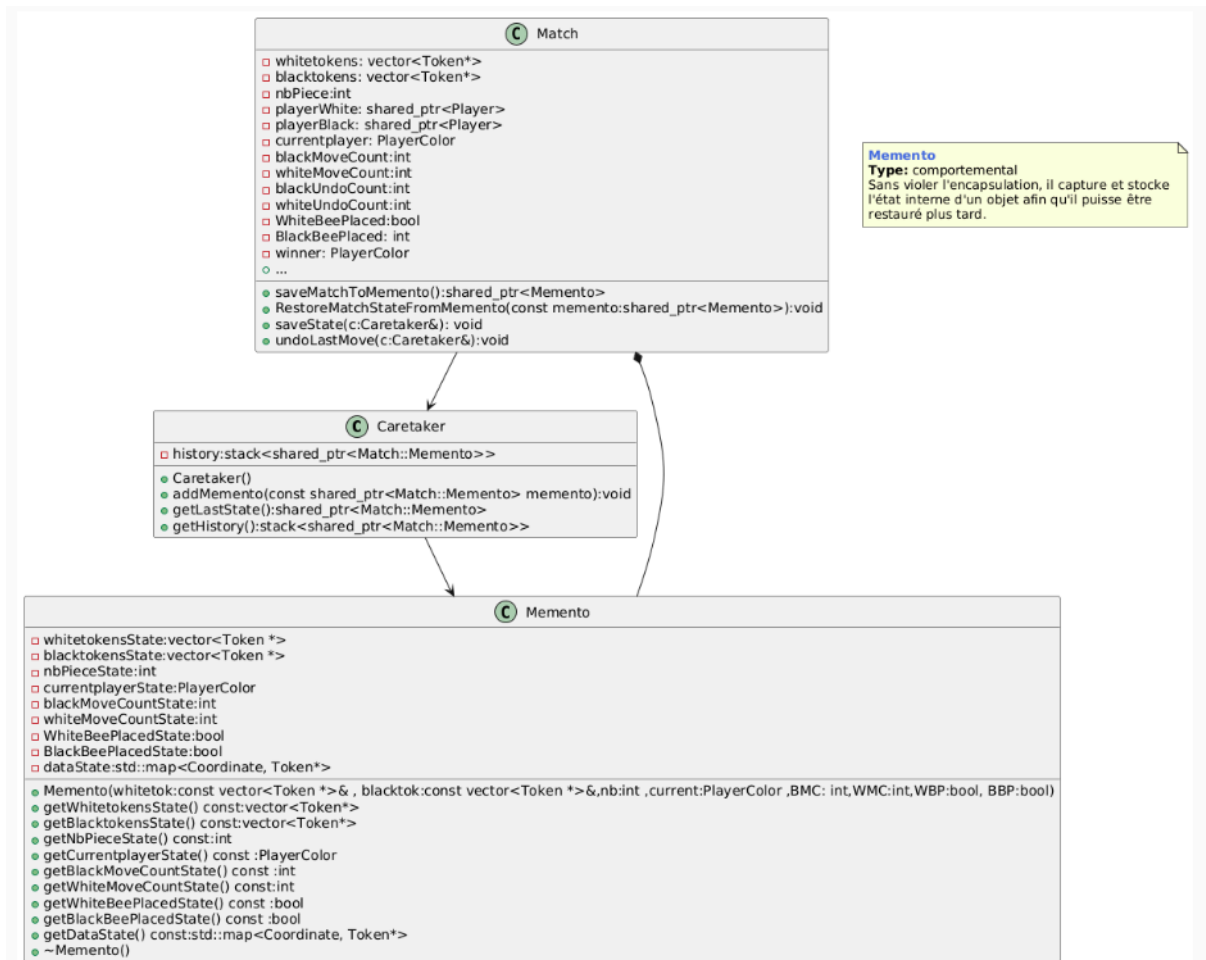
## 1.UML Complet en mode Console:



## 2.UML Complet avec interface graphique



### 3.UML du Design Pattern Memento:



#### 4.UML de Token et ses classes Filles:

