

---

# LO21 - Schotten Totten

## Rapport 3

---

HAN Yushi

KNOCKAERT Myrtille : auteure responsable du rapport 3

LAMIC Svetlana

SAINT-MARC Arman

10 juin 2023



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Architecture</b>	<b>1</b>
2.1	Jeu . . . . .	1
2.2	Manche . . . . .	3
2.3	Carte . . . . .	4
2.4	Joueur . . . . .	6
2.5	Borne . . . . .	6
2.6	Pioche . . . . .	9
2.7	UML mis à jour . . . . .	9
2.8	Explication de la version dans la console . . . . .	11
<b>3</b>	<b>Liste des tâches mises à jour</b>	<b>12</b>
3.1	Affectation des tâches restantes . . . . .	12
3.2	Nouvelles tâches suite au rapport précédent . . . . .	12
<b>4</b>	<b>Etat d'avancement</b>	<b>13</b>
4.1	HAN Yushi . . . . .	13
4.1.1	Tâches effectuées et durée a posteriori . . . . .	13
4.1.2	Pourcentage d'avancement des tâches restantes . . . . .	13
4.2	KNOCKAERT Myrtille . . . . .	13
4.2.1	Tâches effectuées et durée a posteriori . . . . .	13
4.2.2	Pourcentage d'avancement des tâches restantes . . . . .	13
4.3	LAMIC Svetlana . . . . .	14
4.3.1	Tâches effectuées et durée a posteriori . . . . .	14
4.3.2	Pourcentage d'avancement des tâches restantes . . . . .	14
4.4	SAINT-MARC Arman . . . . .	14
4.4.1	Tâches effectuées et durée a posteriori . . . . .	14
4.4.2	Pourcentage d'avancement des tâches restantes . . . . .	14
4.5	Groupe . . . . .	14
<b>5</b>	<b>Bilan sur la cohésion de groupe</b>	<b>15</b>
<b>6</b>	<b>Conclusion</b>	<b>15</b>



# 1 Introduction

Durant cette troisième phase de travail sur le code du jeu Schotten Totten, nous avons terminé le code de manière à pouvoir jouer sur une console. Nous avons peaufiné notre UML sur les quelques points pour lesquels c'était nécessaire.

Par ailleurs, nous avons commencé sur l'interface graphique sur Qt.

Pour tester notre code, nous vous avons ajouté sur notre gitlab, vous pourrez trouver le code fonctionnel en mode console sur la branche "Rendu". Nous avons aussi fait une petite vidéo pour vous montrer le déroulé d'une partie. Le gitlab est disponible sur ce lien : <https://gitlab.utc.fr/mknoackae/projet-lo21>

# 2 Architecture

Nous avons créé dans notre code un fichier.h et un fichier .cpp pour chacune de nos classes, à savoir Jeu, Manche, Joueur, Carte, Pioche et Borne.

Notre architecture permet de choisir entre les modes de jeu Normal, Tactique ou Expert, que nous avons décidé d'implémenter en plus.

Nous ne feront pas la liste exhaustive des arguments pour chaque classe car ils sont tous disponibles dans l'UML. Dans cette partie, nous nous attacherons à détailler les modules qui nous permettent de jouer et leurs interactions entre eux.

## 2.1 Jeu

La classe Jeu est la classe qui gère les différentes manches et donc, le jeu entier selon le mode de jeu choisi. Elle compose la classe Joueur et la classe Manche.

Il est construit avec un design pattern Singleton. En effet, il est composé d'une entité unique Jeu, qui permet de garantir l'unicité de la classe à tout moment.

La classe jeu a également trois méthodes qui lui permettent de gérer la partie. Elle crée les joueurs et gère les manches selon le nombre de manches choisi par les joueurs.

Le constructeur Jeu de la classe Jeu prend en arguments :

- deux chaînes de caractères const qu'on appelle nomJ1 et nomJ2 pour les noms du joueur 1 et du joueur 2
- un booléen adversaireBot pour savoir si le joueur 2 est un humain ou une machine
- un élément modeDeJeu de type Mode (enumerate, voir plus loin) pour choisir la variante de jeu
- un entier positif ou nul (unsigned int) nbManchesGagnantes pour le nombre de manches gagnantes

Les méthodes const getNbManchesGagnantes, getJoueur1, getJoueur2 et getModeDeJeu sont des méthodes const, sans arguments et définies de manière inline, qui permettent d'accéder aux attributs de la classe Jeu.



La méthode `donnerInstance`, de type `Jeu&`, prend en argument les noms des joueurs 1 et 2 sous forme de string `nomJ1` et `nomJ2`, un booléen `adversaireBot` pour savoir si l'adversaire est un robot, un élément `modeDeJeu` de type `Mode` pour connaître la variante, et un entier positif ou nul (unsigned int) `nbManchesGagnantes` pour le nombre de manches gagnantes. Si le jeu n'a pas encore été créé (soit `_jeuUnique` est un pointeur NULL), la méthode affecte à `_jeuUnique` un jeu créé dynamiquement et initialisé avec les éléments passés en arguments (`nomJ1`, `adversaireBot`, `nomJ2`, `modeDeJeu`, `nbManchesGagnantes`). Elle renvoie ensuite un pointeur sur `_jeuUnique`.

La méthode `libererInstance` permet d'effacer le jeu créé avec l'instruction "delete `_jeuUnique`" et affecte à `jeuUnique` la valeur `nullptr`.

La méthode `getInstance` permet d'obtenir une instance du jeu unique et renvoie donc un pointeur sur `_jeuUnique`.

Enfin, on peut jouer une manche avec la méthode `jouerManche`, de type `void`. Celle-ci est sans arguments. Elle commence par réinitialiser le nombre de cartes des joueurs 1 et 2 avec les fonctions `resetCartes` de la classe `Joueur`. Le nombre de cartes tactiques jouées est également remis à 0 pour chaque joueur avec la méthode `resetTactiquesJouees`.

Une manche est ensuite créée dynamiquement et nommée `mancheActuelle`, et on crée un unsigned int gagnant, initialisé à 0, pour le numéro du joueur gagnant (1 ou 2).

Par la suite, on crée une boucle `while` pour faire jouer tour à tour les joueurs 1 et 2. On prend ici le cas du joueur 1, mais il en est de même pour le joueur 2. Lorsque c'est au tour du joueur 1 de jouer, on affiche le mot "Joueur 1" avec la fonction `afficherJoueur1()`. On écrit ensuite sur le flux de sortie les mots "Cartes de" suivi du pseudonyme du joueur avec la méthode `getPseudo` de la classe `Joueur`. On appelle alors la méthode `afficherBornes` de `mancheActuelle` afin d'afficher les bornes du jeu, puis on crée un unsigned int `choiceCarte` initialisé à 0 et un booléen `carteVerif` initialisé à `false`. Tant que la carte choisie est toujours 0 ou que `carteVerif` est toujours faux, on effectue les opérations suivantes.

On écrit sur le flux de sortie "1 - Sélectionnez une carte (n°) : " et on attend le choix de la carte du joueur, que l'on stocke dans la variable `choiceCarte`. On teste les conditions suivantes pour s'assurer que la carte sélectionnée est valide :

- le nombre de cartes du joueur, obtenu par la consigne `_joueur1->getCartes().size()` est supérieur ou égal à la variable `choiceCarte`, initialisée à 0, c'est à dire qu'on choisit une carte dont l'indice est inférieur ou égal au nombre de cartes.
- le choix de la carte est supérieur à 0 (on ne peut pas choisir la 0ème carte).

Si ces conditions sont réunies, alors le booléen `carteVerif` prend la valeur renvoyée par l'instruction `"_joueur1->getCartes()[choiceCarte-1]->jouer(mancheActuelle, *_joueur1, this);"`. Si les conditions ne sont pas réunies, la carte n'est pas vérifiée donc `carteVerif` prend la valeur `false` et on écrit sur le flux de sortie "Erreur - La carte n'existe pas!". Toujours dans la boucle `while`, on crée ensuite une boucle `for` (`Borne* borne : mancheActuelle ->getBornes()`) dans laquelle on parcourt les bornes de la manche actuelle. Si la borne actuelle est pleine du côté des deux joueur et qu'elle n'a pas encore été remportée(méthode



estPleine de la classe borne et borne->getGagnant() == nullptr), on vérifie si le gagnant est le joueur 1 ou le joueur 2 avec la méthode trouverGagnant de Borne. Si celle-ci renvoie 1, le joueur 1 remporte la borne avec la fonction setGagnant de la classe Borne. Sinon, c'est le joueur 1 qui gagne la borne. On sort ensuite du if else pour définir un booléen isGagnant, initialisé avec la méthode vérifGagnant de la classe manche, appliquée au joueur 1. Si isGagnant est égal à true, le joueur 1 est déclaré gagnant de la manche et on arrête de jouer. Si ce n'est pas le cas, isGagnant prend la valeur de verifGagnant appliquée au joueur 2. Si isGagnant est égal à true, le joueur 2 est déclaré gagnant de la manche, et on arrête de jouer. Si aucune de ces situations ne se produit, c'est au tour du joueur suivant.

Mode est un enumerate qui permet de choisir la variante, base ("normal") ou tactique.

Les fonctions afficherJoueur1 et afficherJoueur2, en dehors de la classe Joueur, permettent d'afficher sur cout les mots Joueur 1 ou Joueur 2, puis l'instruction "Appuyez sur entrer pour continuer". On efface ensuite ce qu'on vient d'écrire avec fflush. On utilise ensuite l'appel système "read" pour obtenir la réponse du joueur, puis "clear" pour vider l'écran d'application

## 2.2 Manche

La classe Manche va nous servir à gérer chacune des manches en créant à chaque fois les cartes, la pioche et les bornes. Cette classe compose donc les classes Carte, Pioche et Borne. Il y a aussi un constructeur qui est utilisé pour créer une nouvelle manche en fonction du jeu actuel. De même, il y a un destructeur pour nettoyer la mémoire allouée pour la pioche.

La classe Manche contient deux vecteurs. Le vecteur \_bornes contient des pointeurs de type Borne\*, qui pointe sur les bornes du jeu.

Le vecteur \_cartes contient des pointeurs de type Carte\* qui pointeront sur les cartes du jeu.

Enfin, un pointeur de type Pioche\* pointe sur la pioche du jeu.

Les méthodes const getBornes, getCartes et getPioche permettent d'accéder aux attributs de la classe, à savoir respectivement \_borne, \_cartes et \_pioche.

Le constructeur Manche prend en argument l'adresse jeu d'un Jeu. Elle commence par créer les bornes en initialisant un vecteur bornes de type Borne\*. Pour un size\_t i allant de 0 à BORNES (défini hors de la fonction comme étant égal à 9, puisque nous avons 9 bornes), on rajoute au vecteur bornes une nouvelle borne créée dynamiquement avec le numero i. \_bornes prend ensuite la valeur de bornes. Par la suite, les cartes clan et tactique sont créées. Un Mode modeDeJeu est créé avec la méthode getModeDeJeu du jeu passé en argument du constructeur. Un vecteur cartes de Cartes\* est également initialisé. On effectue ensuite deux boucles for l'une dans l'autre pour parcourir les listes Couleurs et Nombres, et on ajoute pour chaque itération la carte Clan correspondante (créée dynamiquement) au vecteur cartes. Si le mode de jeu est le mode tactique, on ajoute les cartes tactiques de la même manière, en utilisant cette fois une seule boucle for qui



parcourt `TypeTactique`(voir plus bas). `_cartes` prend alors la valeur `cartes`. On initialise ensuite `_pioche` avec une nouvelle pioche créé dynamiquement grâce au constructeur `Pioche` et au vecteur `cartes`. Enfin, on distribue les cartes au joueur en fonction du type. On parcourt les entiers de 0 à 7 avec une boucle `for`, si `j=6`, si le mode de jeu `modeDeJeu` est tactique, on pioche une septième carte `clan` pour chaque joueur dans la pioche `_pioche` avec la fonction `piocher`. Sinon, si `j` est inférieur à 6, on pioche des cartes `clan` de la même manière pour chaque joueur.

Le destructeur `Manche` supprime la pioche `_pioche` avec `delete`.

La méthode `const afficherBornes` parcourt le vecteur `_bornes` et affiche les bornes avec la méthode `afficher`.

La méthode `const verifGagnant` prend en argument l'adresse d'un joueur `jou`. Elle crée trois unsigned int, `bornesWin`, `bornesMaxSuiteWin` et `bornesSuiteWin`, tous trois initialisés à 0. Un quatrième unsigned int, `idLastWin`, prend la valeur de l'id du joueur `jou` avec la méthode `getId`. On parcourt ensuite le vecteur `_bornes`. Dans cette boucle `for`, plusieurs `if` sont les uns dans les autres. Si (1) le gagnant n'est pas null, si (2) l'id du gagnant est la même que celle de `jou`, on augmente de 1 `bornesWin`. Si (3) `idLastWin` est égale à celle de `jou`, on incrémente de 1 `bornesSuiteWin`. Si (4) `bornesSuiteWin` est supérieur à `bornesMaxSuiteWin`, alors `bornesMaxSuiteWin` prend la valeur de `bornesSuiteWin`. Sinon (3), `bornesSuiteWin` prend la valeur 1. `idLastWin` prend ensuite l'id de `jou` (1). On sort de la boucle `for`. Si `bornesMaxSuiteWin` est supérieur ou égal à 3 ou `bornesWin` supérieur ou égal à 5 (si les bornes adjacentes remportées par le joueur sont au nombre de 3 ou plus ou si le joueur a remporté 5 bornes ou plus), on retourne `true` (le joueur est déclaré gagnant). Sinon, on retourne `false`.

Les fichiers `manche` contiennent deux listes, à savoir `Couleurs` et `Nombres`, contenant respectivement les couleurs et les numéros des cartes. Le fichier `Manche.cpp` contient une `unordered_map TypeTactique` contenant le type de la carte tactique, la méthode correspondante et son effet.

## 2.3 Carte

La classe `Carte` permettra de créer chacune des cartes de chacune des méthodes. C'est une classe abstraite qui représente une carte dans le jeu. Les classes `Clan` et `Tactique` sont des classes dérivées de `Carte` qui représentent respectivement une carte de clan et une carte tactique. Ces classes ont des fonctions membres pour jouer la carte, afficher ses informations et obtenir le type, le nom, le nombre et la couleur de la carte. Nous avons utilisé des méthodes `virtual` pour pouvoir redéfinir les méthodes dans les classes héritant de `Carte`.

Pour gérer les informations des cartes, nous avons utilisé des énumérateurs `Couleur`, `Nombre` et `TypeTactique` qui contiennent respectivement les couleurs des cartes, les numéros des cartes et les différents types des cartes tactiques.

Nous avons aussi utilisé plusieurs fonctions pour gérer l'affichage :



La fonction `toString` prend en argument une `Couleur`, un `Nombre` ou un `TypeTactique` et renvoie la chaîne de caractères correspondante.

On définit également l’affichage d’une caractéristique (`Couleur` ou `Nombre`) sur un flux `ostream`.

La fonction `printTextInColor` permet d’afficher le texte en couleur selon la `Couleur` `color` passée en argument.

La fonction `printTactique` permet d’afficher un `TypeTactique`.

- Carte

La classe `Carte` est une classe abstraite. Elle possède six méthodes virtuelles pures : `jouer`, `effet`, `afficher`, `getType`, `getNom`, `getNombre` et `getCouleur`.

- Clan

La classe `Clan` possède un attribut `__numero` de type `Nombre`, et un attribut `__couleur` de type `Couleur`.

La classe `Clan` est une classe fille de `Carte` et possède un constructeur `Clan` qui initialise le `numero` et la `couleur`.

La méthode `jouer` prend en argument une manche `Manche*`, l’adresse d’un joueur `j` et un `Jeu*` `jeu`. Elle demande de sélectionner une borne. Si la borne fait partie des bornes présentes dans le jeu, on choisit la carte à poser (sinon on renvoie un message d’erreur). Si on est dans la variante de base, on pioche une carte clan. Si on est dans la variante tactique, on a le choix entre la pioche de cartes clan et la pioche de cartes tactique.

La méthode `getType` nous est très utile car elle nous permet de savoir si une carte est une carte clan ou une carte tactique et cela nous permet de gérer le choix de pioche du joueur, de savoir quelle carte considérée lors de la revendication, etc. Les méthodes `getNombre`, `getCouleur` et `getNom` nous sont aussi très utiles car elles renvoient respectivement le nombre, la couleur et le nom de la carte pour les cartes tactiques. Ces getters nous permettent d’identifier les cartes.

La méthode `afficher` écrit en couleur le numéro `__numero` et la couleur `__couleur` de la carte.

- Tactique

La carte tactique à la spécificité d’avoir des méthodes `static jouerX` et `effetX` (remplacer `X` par le nom de la carte tactique) permettent d’implémenter les effets des cartes, et ce qui se produit lorsqu’on les place. A titre d’exemple, le `Traître` n’a pas d’effet, mais permet de déplacer une carte lorsqu’il est joué. En revanche, le `Porte-Bouclier` a pour effet de prendre une couleur au choix.



## 2.4 Joueur

La classe Joueur a pour objectif de gérer la main de chaque joueur. Elle est composée d'un id, d'un vector<Carte\*> pour gérer la main du joueur, un pseudonyme, un booléen pour savoir si le joueur est une IA ou non, le nombre de cartes tactiques jouées, et le nombre de victoires. Des getters ont été créés afin de pouvoir récupérer ces différents composants. Plusieurs setters ont aussi été codés. La classe est aussi composée d'un constructeur et d'un destructeur.

Les méthodes const getId, getCartes, getPseudo, getIsBot, getNbTactiquesJouees et getVictoire permettent respectivement d'obtenir les valeurs des attributs \_id, \_cartes, \_pseudo, \_bot, \_nbTactiquesJouees et \_victoires.

Les méthodes addPoint et addTactiquesJouees permettent respectivement d'incrémenter de 1 les attributs \_victoires et \_nbTactiquesJouees lorsque le joueur remporte une manche ou place une carte tactique.

La méthode addCarte prend en argument un pointeur carte de type Carte\*. Elle ajoute carte au vecteur \_cartes.

La méthode supprimerCarte prend en argument un pointeur carte de type Carte\*. Elle recherche la position de carte dans le vecteur \_cartes et efface de ce dernier la carte située à cette position.

La méthode resetTactiquesJouees remet \_nbTactiquesJouees à 0, et la methode resetCartes efface le vecteur \_cartes avec clear.

Le constructeur Joueur de la classe Joueur prend en arguments un unsigned int, un booléen et un string pour initialiser \_id, \_bot et \_pseudo. Il initialise également \_nbTactiquesJouees à 0 et \_victoires à 0.

Le destructeur de la classe Joueur est le destructeur par défaut.

Enfin, la méthode const afficherCartes parcourt les cartes de \_cartes et les affiche avec la fonction afficher de la classe Carte.

## 2.5 Borne

La classe Borne va permettre de gérer chacune des bornes du jeu en prenant en compte les cartes posées au fur et à mesure par les joueurs. Elle permettra aussi de gérer la revendication dans les cas suivants :

- Il y a trois cartes de chaque côté, ou quatre si la carte Combat de Boue a été utilisée.
- Le joueur pense que dans n'importe quelle situation future, aucune combinaison ne pourra battre la sienne.





La classe Borne est en agrégation avec la classe Carte car les bornes ne gèrent pas le cycle de vie des cartes.

Ainsi, chaque borne possède un numéro permettant de l'identifier, un nombre de cartes maximal (3 habituellement, 4 lors de l'utilisation de la carte tactique Combat de Boue), un vecteur de type `<Carte*>` respectif pour stocker les cartes que les deux joueurs ont posées.

Elle possède également deux vecteurs de `Carte*` qui contiennent les cartes posées par chaque joueur de chaque côté de la frontière.

Enfin, elle a pour attribut un pointeur sur le joueur ayant remporté la borne.

Les méthodes const `getCartesJoueur1`, `getCartesJoueur2` et `getGagnant` permettent respectivement de récupérer les cartes du joueur 1, du joueur 2 et l'id du gagnant.

La méthode `setGagnant` prend en argument un `Joueur*` et l'affecte son id au gagnant.

La méthode `setNbCartesMax` prend en argument un `unsigned int` et l'affecte à `_nbCartesMax`.

Le constructeur Borne prend en argument un `size_t` `numero` et initialise `_gagnant` à `nullptr` (aucun gagnant lorsque la borne est créée), `_nbCartesMax` à 3 (aucune carte tactique n'est alors utilisée) et `numero` à `_numero`.

Le destructeur Borne efface `_gagnant`.

La méthode `poserCarte` prend en arguments l'adresse du joueur qui pose la carte et un pointeur sur la carte en question. Elle vérifie l'id du joueur pour savoir sur quel joueur effectuer la suite des opérations. Si le nombre de cartes sur la borne est inférieur au nombre maximal de cartes possible, elle ajoute la carte au vecteur `Carte*` correspondant au joueur et elle supprime la carte du jeu du joueur avec la méthode `supprimerCarte` de la classe `joueur`. Si on atteint le nombre de cartes maximal, elle utilise la fonction `revendiquerBorne`. Si la borne sur laquelle on a voulu poser la carte est pleine, elle renvoie "Attention : nombre max atteint, vous ne pouvez pas ajouter une carte."

La méthode `changerCarte` prend en arguments l'adresse du joueur concerné et un pointeur sur la carte en question. Elle vérifie l'id du joueur pour savoir sur lequel appliquer les opérations suivantes. Si le nombre de cartes du côté du joueur est inférieur au nombre de cartes maximal, elle rajoute la carte à déplacer. Sinon, elle renvoie le même message que pour la fonction `poserCarte`.

La méthode `retirerCarte` prend en arguments l'adresse du joueur concerné et un pointeur sur la carte en question. Elle vérifie l'id du joueur pour savoir sur lequel appliquer les opérations suivantes. Elle recherche la carte parmi celles du côté du joueur. Si elle la trouve, elle la supprime de `_cartesJoueur1`. Sinon, elle renvoie "Cette carte n'est pas sur cette borne".

La méthode `afficherCartesJ1` (respectivement `afficherCartesJ2`) parcourt les cartes du vecteur des deux joueurs et les affiche avec la méthode `afficher` de la classe `Carte`.



La méthode `const afficher` de la classe `borne` affiche "J1" puis les cartes du joueur 1 avec la méthode `afficherCartesJ1`. Si la borne a été gagnée, elle affiche le numéro de la borne et le pseudonyme du gagnant. Sinon, elle affiche juste le numéro de la borne. Enfin, elle affiche les cartes du joueur 2 puis "J2".

La méthode `const estPleine` prend en argument l'adresse d'un joueur. Elle initialise un `size_t` `taille` à 0. Elle vérifie l'id du joueur pour savoir sur lequel appliquer les opérations suivantes. Elle parcourt ensuite les cartes du côté du joueur en question. S'il s'agit de cartes `clan` ou d'un `Joker`, d'un `Espion` ou d'un `Porte-Bouclier` (cartes qui peuvent se placer sur une borne), on incrémente `taille` de 1. Si le nombre de cartes maximum est strictement supérieur à `taille` (nombre de cartes posées), la borne n'est pas pleine, donc on renvoie `false`. Sinon, on renvoie `true`.

La méthode `trouverGagnant` prend en arguments un `unsigned int` `idPremier`, deux adresses de `Joueur` `J1` et `J2` et un pointeur `manche` sur une `Manche`. Elle crée un booléen `estSomme`, initialisé à `false`. Pour chaque joueur, on vérifie s'il a posé un `Colin-Maillard`. Si c'est le cas, `somme` est égal à `true` et on utilise la méthode `effet` de la classe `Carte`. On crée deux `unsigned int` `pointsJ1` et `pointsJ2` dont la valeur est initialisée avec la méthode `calculerPoints`. On calcule ainsi les points de chaque joueur et on vérifie si `estSomme` est égal à `true`. Si c'est le cas, `pointsJ1` prend la valeur `pointsJ1%50`. Il en va de même pour `pointsJ2`. On renvoie ensuite l'id du joueur ayant le plus de points ou, en cas d'égalité, l'identifiant `idPremier` du premier joueur ayant complété la borne.

La fonction `calculerPoints` prend en argument un vecteur de `Carte*` `_cartesJoueur` et l'adresse `j` d'un joueur. Elle crée un booléen `estSuiteCouleur` initialisé à `false`, et trois booléens initialisés à `true`, à savoir `estBrelan`, `estCouleur` et `estSuite`. Elle crée également un `size_t` `taille`, dont la valeur est le nombre de cartes placées du côté de `_cartesJoueur`, et deux tableaux de longueur `taille` : `suiteNombre`, une liste d'entiers, et `suiteCouleur`, une liste d'éléments de type `Couleur`. Enfin, elle crée un `size_t` `i` égal à 0. Pour traiter toutes les cartes du joueur, elle ajoute, dans une boucle `for`, le nombre de la carte correspondante à `suiteNombre[i]` et la couleur de la carte à `suiteCouleur[i]`. Elle met ensuite en ordre les cartes avec une boucle `for`. Enfin, elle vérifie tous les types de combinaison possibles (suite, brelan, couleur, etc.) et modifie les booléens en fonction. Elle effectue ensuite la somme du tableau en créant un `unsigned int` `somme`, égal à 0. Pour `i` allant de 0 à `taille`, `somme` prend la valeur `suiteNombre[i]`. Enfin, elle teste les booléens `estSuiteCouleur`, `estBrelan`, `estCouleur`, `estSuite` pour renvoyer le résultat correspondant. Nous avons utilisé un système de multiplication selon la combinaison en prenant la somme des cartes puis  $50 * 1/2/3/4/5$  selon si l'on a une somme, une suite, une couleur, un brelan ou une suite couleur.

La fonction `revendiquer` utilise la méthode `trouverGagnant` et la fonction `creerPossibites`. Il y a trois possibilités pour la revendication :

1. Le cas où les deux joueurs ont rempli leur côté de la borne. Dans ce cas, nous appelons la méthode `trouverGagnant`.
2. Le cas où c'est le joueur 1 qui revendique la borne car il a rempli son côté et qu'il



pense pouvoir gagner peut importe ce que le joueur posera en face. Pour gérer ce cas, nous avons utiliser `creerPossibilites` qui crée un vecteur de `vector Carte*` avec tous les vector de combinaisons possibles entre les cartes déjà posées par l'autre joueur et les cartes qui n'ont pas encore été posées. Puis on appelle la fonction `trouverGagnant` et si elle trouve une seule combinaison meilleure, la borne ne peut pas être revendiquée. Sinon, la borne est gagnée par le joueur qui a revendiqué la borne.

Enfin, la fonction `toInt` prend en argument un `Nombre n` et retourne l'entier correspondant à sa valeur avec un `switch`.

## 2.6 Pioche

La classe `Pioche` a pour but de créer chacune des cartes composant la pioche, tactique ou normal. Avec celle-ci, le joueur aura la possibilité de piocher une carte lorsque c'est à son tour de jouer du type qu'il souhaite. On retrouve une agrégation entre `Pioche` et `Carte`, car la pioche ne gère pas le cycle de vie des cartes (c'est le `Jeu` qui s'en occupe).

La classe `pioche` possède un vecteur `_cartes` de pointeurs sur des éléments de type `Cartes`. Il s'agit des cartes n'ayant pas été piochées.

La classe `Pioche` possède un constructeur du même nom, qui prend en argument un vecteur de `Cartes*` et qui initialise `_cartes` avec ce vecteur.

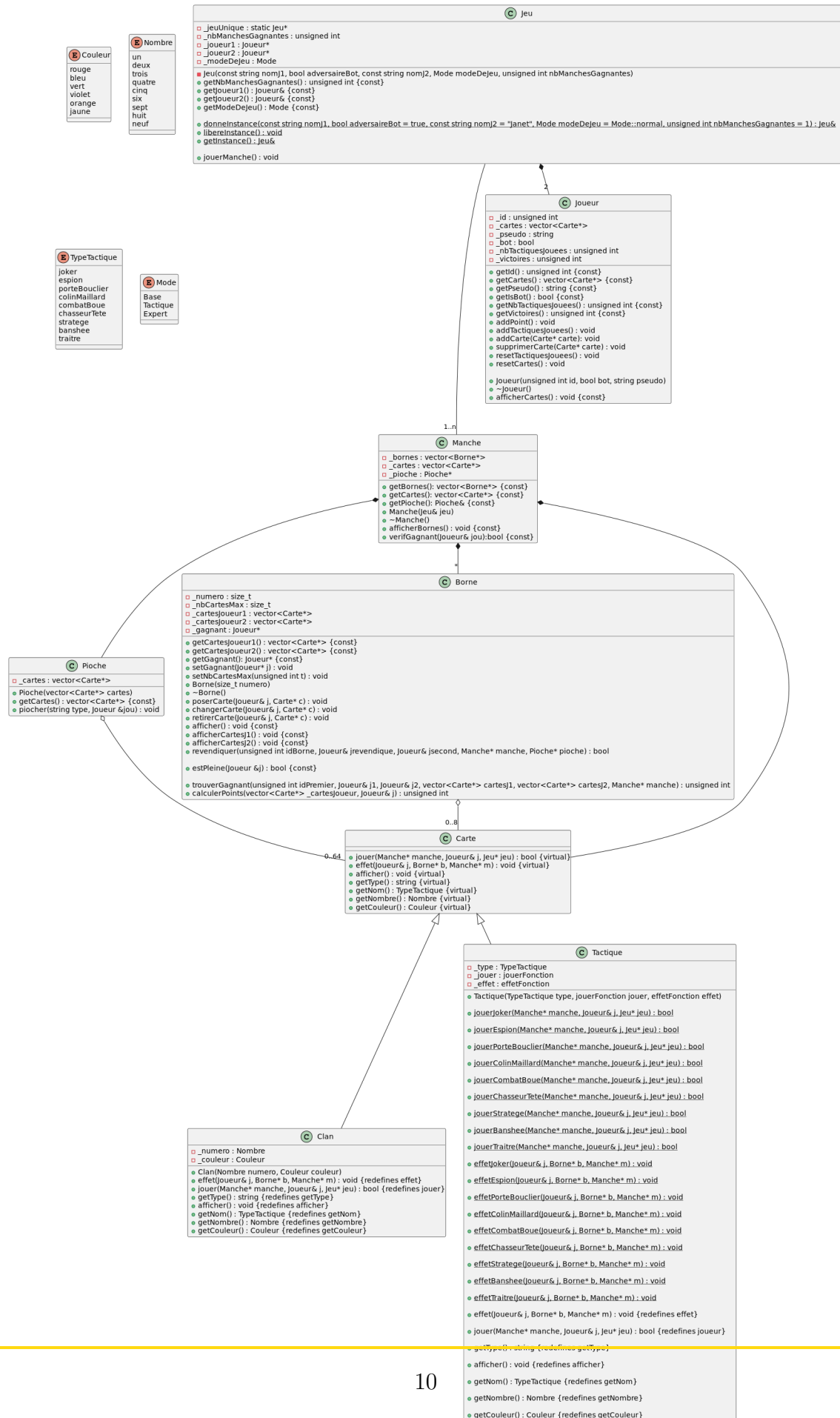
La méthode `piocher` prend en arguments une chaîne de caractères `type` et l'adresse d'un `Joueur jou`. Elle crée un vecteur `cartesType` d'éléments de type `Carte*`. Avec une boucle `for`, elle parcourt les cartes de la pioche et, pour chacune, teste si son type est égal à `type`, avec la fonction `getType` de la classe `Carte`. Si c'est le cas, elle ajoute la carte au vecteur `cartesType`. En sortie de la boucle `for`, la fonction initialise un générateur de nombres aléatoires, puis crée un élément `size_t random` dont la valeur est parmi les nombres inférieurs ou égaux à la taille de `cartesTypes`. Elle crée alors un pointeur sur une carte nommé `carte`, qui prend la valeur `cartesTypes[random]`. Par la suite, elle initialise un `size_t j` à 0. Pour `j` allant de 0 à la taille du vecteur `_cartes`, si `_cartes[j]` est égal à la carte sélectionnée `carte`, on sort de la boucle `for`. En sortant de la boucle `for`, on efface `carte` de la pioche avec `erase` et on ajoute la carte au jeu du joueur avec la fonction `addCarte` de la classe `Joueur`.

## 2.7 UML mis à jour

Nous avons représenté notre UML sur CodiMD. La légende est la suivante :

+ : rond vert, - : carré rouge, `override` : redefines x, `virtual` : soulignement

Pour la visibilité, nous avons aussi mis l'uml en version svg sur gitlab.







Finalement, à chaque tour, on propose au joueur de renvidiquer au maximum une borne. Un texte va s'afficher à la ligne de la borne de côté gagnant. Si une borne n'est pas pleine, mais une combinaison correspond à une situation dans notre fonction revendication, cela veut dire que cette combinaison sera plus forte que celle que soit la combinaison de son adversaire. Dans ce cas, le joueur gagne la borne.

## 3 Liste des tâches mises à jour

### 3.1 Affectation des tâches restantes

SHOTTEN TOTTON								
Catégorie	Quoi ?	Qui ?	Temps estimé	Avancement	Priorité	Difficulté	Deadline	Temps final réalisation
Phase 1								
Réflexion	Lire et noter toutes les fonctionnalités attendues	Groupe	1h	Fait	Indispensable	3	10 mars	30m
Réflexion	Lire les règles du schotten totton et noter les points clés	Groupe	1h	Fait	Indispensable	3	10 mars	30m
Jeu	Jouer au jeu pour s'imprégner des règles et spécificités	Groupe	3h	Fait	Utile	3	10 mars	2h30
Réflexion	Brainstorming de la structure du projet et vision des choses	Groupe	45m	Fait	Importante	2	10 mars	30m
Réflexion	Recherche des design patterns à intégrer	Groupe	1h30	Fait	Indispensable	2	10 mars	2h
UML	Résumé dans une v0 de l'UML	Groupe	1h	Fait	Importante	2	10 mars	1h30
UML	Création d'un scénario pour trouver les méthodes basiques	Groupe	1h30	Fait	Utile	3	10 mars	2h
Phase 2								
Réflexion	Finaliser les cahier des charges pour le code	Groupe	2h	Fait	Utile	3	5 mai	1h30
Code	Code du fichier .h de la classe Jeu	Arman	1h	Fait	Indispensable	2	5 mai	2h
Code	Code du fichier .h de la classe Manche	Arman	1h	Fait	Indispensable	1	5 mai	1h
Code	Code du fichier .h de la classe Carte	Myrtille	30m	Fait	Indispensable	2	5 mai	1h
Code	Code du fichier .h de la classe Clan	Myrtille	30m	Fait	Indispensable	3	5 mai	1h
Code	Code du fichier .h de la classe Tactique	Myrtille	1h	Fait	Indispensable	1	5 mai	2h
Code	Code du fichier .h de la classe Borne	Yushi	1h	Fait	Indispensable	2	5 mai	1h
Code	Code du fichier .h de la classe Pioche	Svetlana	1h	Fait	Indispensable	2	5 mai	1h
Code	Code du fichier .h de la classe Joueur	Svetlana	1h	Fait	Indispensable	2	5 mai	1h
Code	Code du fichier .cpp de la classe Jeu (méthodes basiques)	Arman	2h	Fait	Indispensable	3	5 mai	2h
Code	Code du fichier .cpp de la classe Manche (méthodes basiques)	Arman	4h	Fait	Indispensable	1	5 mai	4h
Code	Code du fichier .cpp de la classe Carte (méthodes basiques)	Myrtille	1h	Fait	Indispensable	2	5 mai	4h
Code	Code du fichier .cpp de la classe Clan (méthodes basiques)	Myrtille	1h	Fait	Indispensable	2	5 mai	3h30
Code	Code du fichier .cpp de la classe Tactique (méthodes basiques)	Myrtille/Arman	3h	Fait	Indispensable	1	5 mai	4h
Code	Code du fichier .cpp de la classe Borne (méthodes basiques)	Yushi	2h	Fait	Indispensable	3	5 mai	6h
Code	Code du fichier .cpp de la classe Pioche (méthodes basiques)	Svetlana	2h	Fait	Indispensable	2	5 mai	6h
Code	Code du fichier .cpp de la classe Joueur (méthodes basiques)	Svetlana	3h	Fait	Indispensable	2	5 mai	6h
UML	Mise à jour de UML avec classes et méthodes mises à jour	Svetlana	30m	Fait	Importante	3	5 mai	45m
Phase 3								
Réflexion	Finir de réfléchir à toutes les méthodes nécessaires	Groupe	2h	Fait	Importante	1	25 mai	1h30
UML	Finir l'UML complet avec les dernières méthodes et design patterns	Groupe	2h	Fait	Indispensable	3	25 mai	2h
Code	Code de la carte tactique Joker	Myrtille	2h	Fait	Indispensable	2	25 mai	2h30
Code	Code de la carte tactique Espion	Yushi	2h	Fait	Indispensable	2	25 mai	4h
Code	Code de la carte tactique Porte-Boudier	Svetlana	2h	Fait	Indispensable	2	25 mai	3h30
Code	Code de la carte tactique Colin-Mallard	Arman	2h	Fait	Indispensable	2	25 mai	1h
Code	Code de la carte tactique Combat de Boue	Myrtille	2h	Fait	Indispensable	2	25 mai	1h30
Code	Code de la carte tactique Chasseur de Tête	Yushi	2h	Fait	Indispensable	2	25 mai	2h
Code	Code de la carte tactique Stratège	Svetlana	2h	Fait	Indispensable	2	25 mai	2h30
Code	Code de la carte tactique Banshee	Arman	2h	Fait	Indispensable	2	25 mai	1h30
Code	Code de la carte tactique Traître	Myrtille	2h	Fait	Indispensable	2	25 mai	1h
Code	Code de la carte tactique permettant de créer et jouer une manche	Svetlana	2h	Fait	Indispensable	3	25 mai	4h
Code	Code des méthodes d'affichage (borne, carte, joueur, manche, etc.)	Yushi	2h	Fait	Indispensable	3	25 mai	1h30
Code	Code des méthodes de décompte des points (borne, manche)	Arman	4h	Fait	Indispensable	1	25 mai	3h
Code	Code des méthodes de pioche et gestion des cartes (supprimer, ajouter, poser)	Myrtille	4h	Fait	Indispensable	1	25 mai	3h
Code	Code de la fonction random permettant à l'IA de jouer	Yushi/Svetlana	2h	En cours	Indispensable	2	2 juin	3h
Code	Code de la fonction pour revendiquer une borne	Myrtille/Arman	2h	En cours	Indispensable	1	2 juin	5h
Debug	Debugger la partie mode Normal	Groupe	4h	Fait	Indispensable	2	2 juin	x
Debug	Debugger la partie mode Tactique	Groupe	6h	Fait	Indispensable	1	2 juin	x
Jeu	Jouer à des parties pour trouver le plus de bugs	Groupe	6h	En cours	Importante	3	2 juin	x
Design	Début du design UI/UX pour le jeu	Myrtille	1h	Fait	Bonus	3	2 juin	1h
Phase 4								
Réflexion	Réflexion implémentation Qt dans architecture	Groupe	2h	A faire	Importante	2	10 juin	
Code	Code de l'interface de paramètres	Svetlana/Yushi	4h	A faire	Indispensable	2	10 juin	
Code	Code de l'interface de jeu	Myrtille/Arman	4h	A faire	Indispensable	1	10 juin	
Code	Code des méthodes onClick sur carte	Svetlana/Yushi	1h	A faire	Indispensable	1	10 juin	
Code	Code des méthodes onClick sur borne	Myrtille/Arman	1h	A faire	Indispensable	1	10 juin	
Debug	Debugger la partie mode Normal	Groupe	4h	A faire	Indispensable	2	10 juin	
Debug	Debugger la partie mode Tactique	Groupe	6h	A faire	Indispensable	1	10 juin	
Jeu	Jouer à des parties pour trouver le plus de bugs	Groupe	6h	A faire	Importante	3	10 juin	

### 3.2 Nouvelles tâches suite au rapport précédent

Maintenant que le jeu fonctionne en mode console, nous allons effectuer le code sur Qt afin de créer une meilleure interface entre l'utilisateur et la machine.

Pour voir le détail des tâches et leur répartition, il faut regarder sur le tableau.

## 4 Etat d'avancement

Après le rendu du deuxième rapport, nous avons commencé le codage selon la répartition des tâches. Le travail effectué a été réparti de la manière suivante :

### 4.1 HAN Yushi

#### 4.1.1 Tâches effectuées et durée a posteriori

Yushi s'est occupée de mettre à jour l'UML avec Svetlana, en ajoutant les nouvelles fonctions tout en respectant les conventions de l'UML. Ce travail lui a pris 3h.

Elle a aussi travaillé sur le test global du code et le debug. Elle a réfléchi sur la fonction revendication et elle a commencé à la coder. Ce travail lui a pris 7 heures.

Enfin, elle a travaillé sur le test dans la console et l'explication du jeu dans la console, ce qui lui a pris 3h. Depuis le début, elle aura passé 39h sur le projet.

#### 4.1.2 Pourcentage d'avancement des tâches restantes

1. Ajout de la méthode revendication d'une borne : 20%

### 4.2 KNOCKAERT Myrtille

#### 4.2.1 Tâches effectuées et durée a posteriori

Myrtille a codé la classe Carte. Elle a aussi fait fonctionner toutes les fonctions liées aux effets des cartes tactiques car le travail du groupe comportant pas mal d'erreur. Ce travail lui a pris 11h.

Elle a aussi codé la classe Pioche et ses méthodes avec Arman, qui l'a aidée à déboguer le code. Ce travail lui a pris 4h.

Enfin, elle a codé la classe Borne avec la méthode poserCarte(), retirerCarte() et estPleine(). Ce travail lui a pris 3h.

Elle a aussi codé la classe Joueur, en implémentant les méthodes contientCombatDeBoue(), addCarte() et supprimerCarte(). Ce travail lui a pris 2h.

Elle a aussi beaucoup travaillé sur la méthode revendiquer car il a fallu beaucoup déboguer le code car il y a avait beaucoup de subtilités et d'exceptions à gérer. Ce travail lui a pris 14h.

Elle a aussi commencé à faire la partie sur Qt en créant la première fenêtre pour récupérer les informations nécessaires au lancement d'une partie. Ce travail lui a pris 2h. Depuis le début, elle a travaillé 55h sur le projet.

#### 4.2.2 Pourcentage d'avancement des tâches restantes

1. Code en mode console : 100%
2. Débug du mode console : 95
3. Création des événements de clique sur les cartes avec Qt : 5





4. Débug du mode graphique : 5

## 4.3 LAMIC Svetlana

### 4.3.1 Tâches effectuées et durée a posteriori

Svetlana s'est occupée de mettre à jour l'UML avec Yushi, en ajoutant les nouvelles fonctions tout en respectant les conventions de l'UML. Ce travail lui a pris 3h.

Elle a également travaillé sur l'explicitation du code afin d'en faciliter la compréhension dans le rapport. Ce travail lui a pris 6 heures.

Enfin, elle a codé plusieurs méthodes, et a notamment commencé à travailler sur la revendication et sur le codage de l'interface sur Qt. Ce travail lui a pris 8h.

Depuis le début, elle aura passé 39h sur le projet.

### 4.3.2 Pourcentage d'avancement des tâches restantes

1. Réflexion implémentation Qt dans architecture 30
2. Code de l'interface de paramètres : 15%
3. Code des méthodes onClick sur carte : 5%

## 4.4 SAINT-MARC Arman

### 4.4.1 Tâches effectuées et durée a posteriori

Arman a codé les méthodes de calcul des points afin de trouver les gagnants d'une borne. Cela a pris 3h.

Il a aidé coder certaines cartes tactiques notamment en codant leurs effets et la manière de les jouer. Cela aura pris plus de 4h.

La plus grosse partie de son travail aura été de débiter le code des autres membres du groupe et de créer des fonctions facilement réutilisables appelées "utils" pour les autres membres du groupe. Cela aura pris 25h. (4 journées de 6h + quelques heures en plus)

Depuis le début, il a travaillé 45h sur le projet.

### 4.4.2 Pourcentage d'avancement des tâches restantes

1. Code en mode console : 100%
2. Débug du mode console : 95
3. Création des événements de clique sur les cartes avec Qt : 5
4. Débug du mode graphique : 5

## 4.5 Groupe

Nous avons effectué des séances de travail par binôme qui ont duré plusieurs heures.





## **5 Bilan sur la cohésion de groupe**

Le bilan de la cohésion de groupe pour notre projet de développement du jeu Shotten Totten en C++ est très positif. Nous avons réussi à maintenir une excellente harmonie et une coopération efficace tout au long du codage du jeu. La communication a été fluide et régulière et nous avons fréquemment pu organiser des sessions de travail communes ou des réunions.

Chaque membre du groupe a contribué de manière active.

Par ailleurs, la répartition des tâches a été équitable (en nombre d'heures), en tenant compte des compétences et des préférences de chaque membre. Nous avons veillé à ce que personne ne soit surchargé de travail et que chacun puisse contribuer de manière équilibrée.

## **6 Conclusion**

Ainsi, plus de la moitié du travail sur le projet est effectuée. La partie code est fonctionnelle et il est possible de jouer avec la console. Il reste à finaliser l'ensemble de l'interface graphique.



★ ★ ★