

Rapport du Projet Logiciel Transversal

Matthieu PILLOT – Estelle CHARRET - Louis PREYS - Souhaila BERDIJ

TABLE DES MATIÈRES

I. Présentation générale	3
1.1 - Objectif	3
1.2 - Règles du jeu	3
<i>i) But du jeu</i>	3
<i>ii) Les pièces</i>	3
<i>iii) Déroulement d'un tour</i>	4
<i>iv) Conditions de victoire</i>	4
1.3 - Conception Générale	4
II. Conception Logiciel	6
2.1 - State	6
<i>i) Game</i>	7
<i>ii) Player</i>	8
<i>iii) Pieces</i>	8
<i>iv) Board</i>	9
2.2 - Engine	10
2.3 - Client	11
<i>i) ScenarioManager : gestion des scénarios de jeu</i>	11
<i>ii) PlayerController : communication entre les joueurs et l'Engine</i>	12
2.4 - Render	13
2.5 - AI	16
<i>i) IA Random</i>	16
<i>ii) IA Heuristique</i>	17
<i>iii) IA Avancée</i>	17
2.6- Serveur	19
<i>i) Server</i>	20
<i>ii) Network Client</i>	21
III. Analyse & Rendu	23
3.1 - Tests réalisés	23
3.2 - Problèmes rencontrés et solutions	24
<i>i) Modularité</i>	24
<i>ii) L'IA avancée</i>	24
3.3 - Conclusion et points d'amélioration	25

I. Présentation générale

1.1 - Objectif

Le projet consiste à concevoir et développer une version numérique du jeu de société **Stratego**, en utilisant le langage C++. L'objectif principal est de recréer les mécanismes fondamentaux du jeu tout en proposant une expérience utilisateur fluide et fidèle au jeu d'origine. Ce projet met l'accent sur la conception orientée objet et le respect des principes de programmation modulaire.

Stratego est un jeu de stratégie où deux joueurs s'affrontent sur un plateau composé de cases. Chaque joueur dispose de 40 pièces ayant des rôles et des valeurs spécifiques, qu'il doit positionner de manière stratégique. Ces pièces sont placées secrètement, face cachée, afin que l'adversaire ne connaisse pas leur position ni leur type. Le but est de capturer le drapeau de l'adversaire tout en protégeant le sien. Les joueurs se déplacent à tour de rôle, en engageant des combats pour éliminer les pièces adverses, selon des règles de hiérarchie des forces.

1.2 - Règles du jeu

i) But du jeu

Le premier joueur à capturer le **drapeau (Flag)** de son adversaire remporte la partie. La partie peut également se terminer si un joueur ne peut plus effectuer de mouvements légaux, dans ce cas, l'autre joueur gagne.

ii) Les pièces

Chaque pièce possède :

- **Un rang** : Détermine sa puissance lors des combats. Une pièce de rang supérieur bat généralement une pièce de rang inférieur.
- **Des spécificités** : Certaines pièces ont des capacités ou des restrictions uniques.

Voici un tableau récapitulatif des pièces et leurs particularités :

Nom de la pièce (fr/en)	Rang	Quantité par joueur	Spécificités
Maréchal/Marshal	10	1	/
Général/General	9	1	/
Colonel/Colonel	8	2	/
Major/Major	7	3	/
Capitaine/Captain	6	4	/
Lieutenant/Lieutenant	5	4	/

Sergent/Sergeant	4	4	/
Démineur/Miner	3	5	Capture la bombe s'il en attaque une
Éclaireur/Scout	2	8	N'a pas de limites de portée lors de ses déplacements
Espion/Spy	1	1	Capture le Maréchal s'il l'attaque
Bombe/Bomb	/	6	Fixe et explose si on l'attaque (hors Démineur)
Drapeau/Flag	/	1	Fixe

iii) Déroulement d'un tour

1. **Placement initial des pièces**
 - Chaque joueur place ses **40 pièces** sur les **4 premières lignes** de son côté du plateau.
2. **Mouvement d'une pièce**
 - Lors de son tour, un joueur peut déplacer l'une de ses pièces d'une case dans l'une des **4 directions (haut, bas, gauche, droite)**.
 - Exception : les **Éclaireurs (Scouts)** peuvent se déplacer sur une distance illimitée en ligne droite, à condition qu'il n'y ait pas d'obstacle (pièces alliées ou ennemies).
3. **Attaque**
 - Si une pièce se déplace sur une case occupée par une pièce ennemie, un **combat** a lieu.
 - La pièce avec le rang le plus élevé gagne le combat, et la pièce perdante est retirée du plateau.
 - En cas d'égalité (même rang), les deux pièces sont éliminées.
 - Les seules exceptions sont l'attaque de **l'Espion (Spy)** sur le Maréchal (Marshal) qui capture ce dernier (mais l'inverse résulte en la défaite de l'Espion), et l'attaque de toute pièce mobile, hors Démineur, sur une **Bombe**, ce qui élimine la pièce et la bombe.
4. **Fin du tour**
 - Une fois une action effectuée (déplacement ou attaque), c'est au tour de l'autre joueur.

iv) Conditions de victoire

- **Capture du drapeau** : Si un joueur parvient à capturer le **drapeau (Flag)** de l'adversaire, il remporte immédiatement la partie.
- **Blocage total** : Si un joueur n'a plus de mouvements possibles (parce que toutes ses pièces mobiles ont été capturées ou bloquées), il perd la partie.
- **Égalité** : La partie peut être considérée comme nulle si aucune pièce ne peut capturer le drapeau adverse (rare).

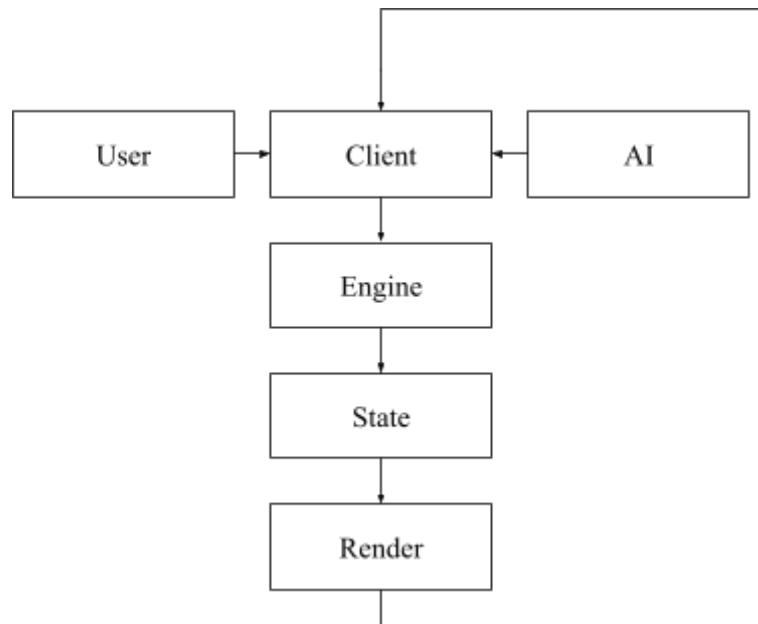
1.3 - Conception Générale

Nous avons réfléchi à de nombreuses façons de concevoir le jeu, et nous nous sommes décidés sur une solution organisée en plusieurs **modules logiciels** (ou **packages**) afin de garantir une **architecture**

modulaire, extensible, et maintenable. Chaque module joue un rôle bien défini dans le fonctionnement global du jeu. Ces modules sont :

- State : Représente l'état du jeu
- Engine : Implémente les règles du jeu
- Client : Orchestre les interactions entre les utilisateurs et l'Engine
- AI : Permet de calculer de manière automatisé des mouvements avant de les fournir à l'Engine lorsqu'un des joueurs est contrôlé par l'ordinateur.

Les modules interagissent entre eux selon cette manière :



Le **Client** ou l'**AI** décident d'une action, qui est envoyée sous forme de commande dans l'**Engine**. Cette dernière va vérifier que ce coup est possible, qu'il est en accord avec les règles du jeu, puis, une fois la validation effectuée, elle le transmet au **State** afin qu'il l'applique. Le **State** va obéir à la demande de l'Engine et modifier le plateau de jeu en conséquence, ce qui va s'afficher sur le **Render**. Grâce à cela, le **Client** peut observer la nouvelle configuration et réfléchir à ses prochaines décisions. L'**AI**, n'étant pas humaine, n'a pas besoin de voir le **Render** afin de comprendre l'évolution du jeu.

II. Conception Logiciel

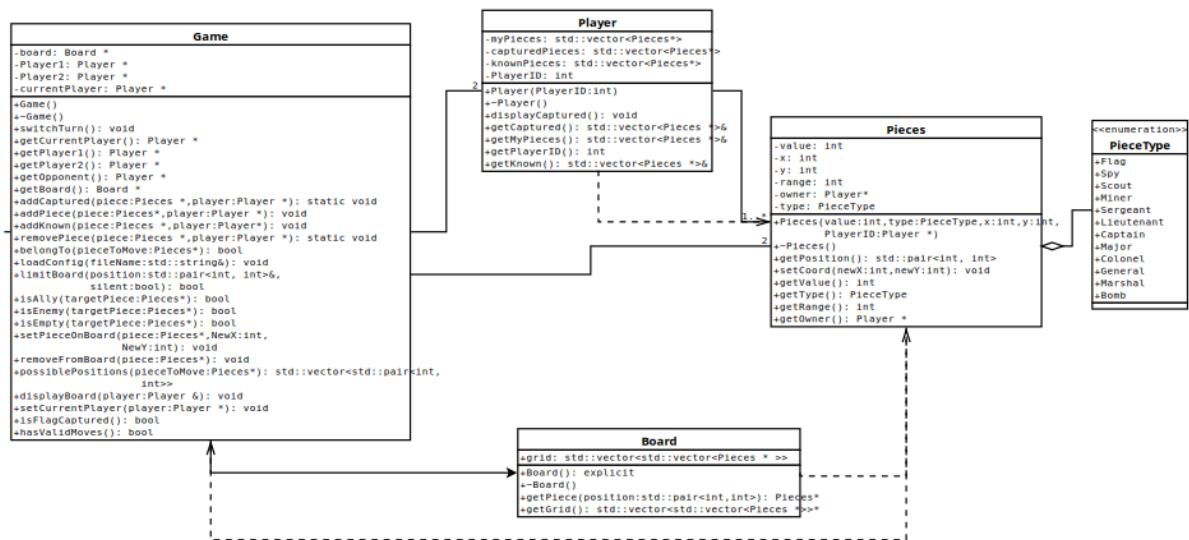
Dans cette partie, nous allons pénétrer au cœur de nos modules afin d'expliquer le fonctionnement du jeu.

2.1 - State

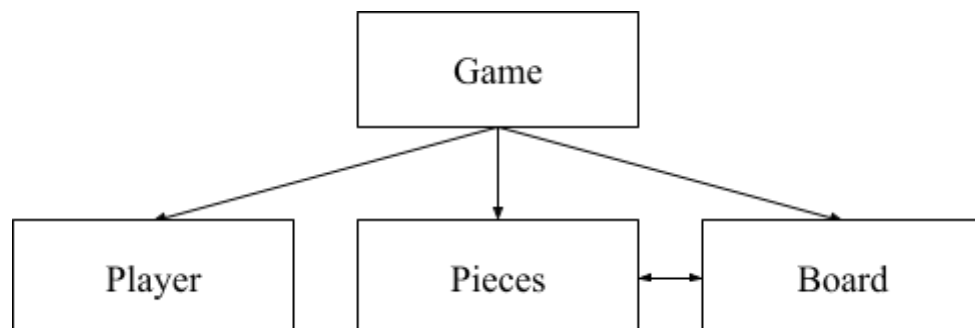
Ce module est celui qui contrôle et agit sur le plateau de jeu. Nous retrouvons donc en son sein toutes les classes liées au jeu réel :

- Game
- Pièces
- Player
- Board

Voici le diagramme UML de ce module :



Ces classes sont liées entre elles selon le schéma suivant :



Nous allons détailler le rôle de chaque classe ainsi que leurs attributs et méthodes principales, ce qui signifie que nous n'explicitons ni les constructeurs/destructeurs ni les getters d'attributs.

i) Game

Cette classe est chargée d'appliquer les changements ordonnés par l'Engine, elle possède les autres classes de ce module et a un accès total à chaque variable, comme montré sur le schéma ci-dessus.

Nous allons nous intéresser aux méthodes de la classe et définir leur utilité :

- ***void switchTurn()*** : Cette fonction permet d'alterner entre le joueur 1 et le joueur 2, donc de changer le currentPlayer.
- ***static void addCaptured(Pieces * piece, Player * player)*** : Cette fonction permet d'ajouter une pièce ennemie dans un vecteur de pièces capturées et donc de la sortir du plateau de jeu. Elle prend en argument la pièce capturée et le joueur qui va la recevoir.
- ***static void removePiece(Pieces * piece, Player * player)*** : Cette fonction est semblable à la précédente sauf qu'elle agit sur sa pièce. En effet, lorsque notre pièce se fait éliminer (soit par suicide, soit par attaque ennemie), nous l'enlevons du plateau de jeu ainsi que de notre vecteur de pièces MyPieces.
- ***bool belongTo(Pieces * piece)*** : Cette fonction permet de vérifier qui est le propriétaire de la pièce que l'on passe en argument.
- ***void addPiece(Pieces * piece, Player * player)*** : Cette fonction permet d'ajouter une pièce à son vecteur MyPieces. Elle est utilisée lors de l'importation et de l'implémentation des pièces dans le jeu.
- ***void loadConfig(std::string& fileName)*** : Cette fonction charge et place les pièces de chaque joueur sur le plateau grâce à la lecture de fichiers csv (actuellement, nous en avons trois : une configuration offensive, défensive et équilibrée).
- ***bool limitBoard(std::pair<int, int>& position, bool silent)*** : Cette fonction vérifie que la position passé en argument se trouve dans les limites du jeu, c'est-à-dire, pas au-delà de 10 cases en longueur et en largeur mais également pas dans les lacs qui se trouvent au milieu du plateau.
- ***bool isAlly(Pieces* targetPiece)*** : Cette fonction vérifie que la pièce en argument est dans la même équipe que celle du joueur actuel.
- ***bool isEnemy(Pieces* targetPiece)*** : Cette fonction vérifie que la pièce en argument est dans l'équipe adverse que celle du joueur actuel.
- ***bool isEmpty(Pieces* targetPiece)*** : Cette fonction vérifie que la pièce en argument correspond à une case vide.
- ***void setPieceOnBoard(Pieces* piece, int NewX, int NewY)*** : Cette fonction place la pièce en argument sur le plateau à la position (NewX, NewY).
- ***void removeFromBoard(Pieces* piece)*** : Cette fonction enlève la pièce en argument du plateau du jeu. Elle est appelée lors de la capture d'une pièce.
- ***std::vector<std::pair<int, int>> possiblePositions(Pieces* pieceToMove)*** : Cette fonction est très utile puisqu'elle permet de renvoyer la liste des positions possibles d'une pièce donnée, en utilisant les fonctions isAlly, isEnemy et isEmpty. Le module AI s'en sert énormément (Elle est aussi très utile pour le Render afin d'afficher les coups possibles).
- ***void displayBoard(Player& player)*** : Comme son nom l'indique, cette fonction permet d'afficher le plateau de jeu, par rapport au joueur concerné. Puisque les deux joueurs ne voient que leurs pièces, il faut un affichage du plateau où les pièces du joueur 1 sont cachées et celles du joueur 2 visibles, et un où la visibilité est inversée.
- ***void setCurrentPlayer(Player* player)*** : Cette fonction permet de désigner un des deux joueurs comme étant le joueur actuel.

- ***bool isFlagCaptured()*** : Cette fonction renvoie un booléen qui indique si le drapeau a été capturé. Elle est utilisée pour déclarer la fin de la partie.
- ***bool hasValidMoves()*** : Cette fonction vérifie si un joueur a encore des coups valides, c'est-à-dire qu'au moins une de ses pièces peut bouger. Si ce n'est pas le cas, elle perd la partie.
- ***void addKnown(Pieces* piece, Player* player)*** : Cette fonction ajoute la pièce en argument dans le vecteur des pièces connus correspondant au bon joueur (par exemple, quand une de nos pièces attaque une pièce ennemie, elle se révèle pour nous affronter et à ce moment-là, elle est ajoutée à notre vecteur). Elle est utilisée dans le module AI.

ii) Player

Cette classe est utilisée afin de définir un joueur et ce qu'il peut faire, c'est-à-dire, avoir un certain nombre de pièces à jouer qui évolue tout au long de la partie, mais également conserver les pièces ennemies capturées, et dans le cas d'un joueur IA, stocker la valeur des pièces connues.

Nous allons maintenant détailler les différents attributs et méthodes de cette classe :

- ***std::vector<Pieces*> myPieces*** : Cet attribut définit le vecteur des pièces utilisées par un joueur.
- ***std::vector<Pieces*> capturedPieces*** : Cet attribut définit le vecteur des pièces capturées par un joueur.
- ***std::vector<Pieces*> knownPieces*** : Cet attribut définit le vecteur des pièces connues par un joueur.
- ***int PlayerID*** : Cet attribut définit l'identité du joueur.
- ***void displayCaptured()*** : Cette méthode affiche les pièces capturées.

iii) Pieces

Cette classe définit les différentes caractéristiques d'une pièce, on va donc y retrouver sa valeur, son type, sa portée etc.

Au départ, nous pensions utiliser le nom des pièces mais nous avons finalement décidé de définir une énumération **PieceType** afin de pouvoir y faire appel lors de fonctions de manière plus sûre :

enum PieceType {Flag, Spy, Scout, Miner, Sergeant, Lieutenant, Captain, Major, Colonel, General, Marshal, Bomb};

Nous allons maintenant présenter les différents attributs et méthodes de cette classe :

- ***int value*** : Cet attribut définit la valeur de la pièce.
- ***int x*** : Cet attribut définit la position, selon l'axe des abscisses, de la pièce.
- ***int y*** : Cet attribut définit la position, selon l'axe des ordonnées, de la pièce.
- ***int range*** : Cet attribut définit la portée de la pièce. Il n'existe que trois cas de figures possibles : 0 pour la Bombe et le Drapeau, 10 pour l'Eclaireur et 1 pour toutes les autres pièces.
- ***Player* owner*** : Cet attribut définit le joueur qui possède la pièce. Cela permet de lier une pièce à un joueur.
- ***PieceType type*** : Cet attribut définit le type de la pièce.
- ***void setCoord (int newX, int newY)*** : Cette fonction change les coordonnées actuelles de la pièce et les remplace par ceux en argument.

Nous avons envisagé de combiner les attributs `int x` et `int y` en un seul attribut `std::pair<int, int>` position mais nous avons finalement décidé de les garder tels quels car cela était plus pratique dans certaines fonctions comme `possiblePositions` par exemple.

iv) Board

Cette classe est utilisée afin de créer le plateau de jeu sur lequel les pièces se déplacent.

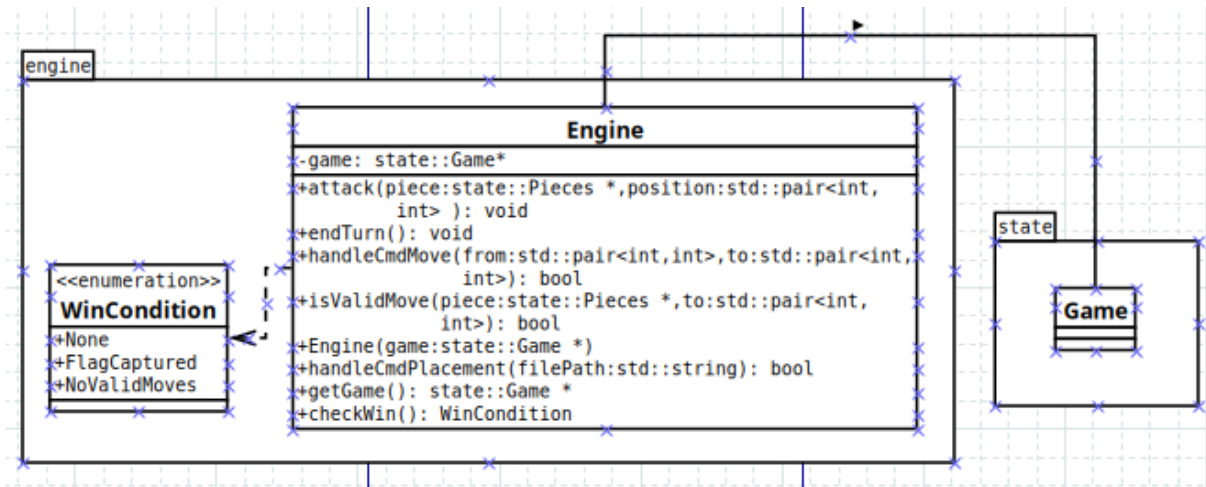
Voici ses attributs et méthodes principales :

- **`std::vector<std::vector<Pieces*>> grid`** : Cet attribut permet de créer la grille qui définit les différentes cases du jeu.
- **`Pieces* getPiece(std::pair<int, int> position)`** : Cette méthode permet d'obtenir la pièce associée à la position sur le plateau.
- **`vector<vector<Pieces*>>* getGrid()`** : Permet d'accéder directement à l'ensemble des cases du plateau, par exemple pour effectuer des vérifications globales ou des mises à jour.

Dans le schéma des liens entre les différentes classes, nous pouvons voir que le `Game` a accès à toutes les autres mais également que `Pieces` et `Board` ont accès l'un à l'autre. Cela se traduit par le fait que les pièces "savent" où elles se trouvent sur le plateau, et ce dernier "connaît" l'identité de chaque pièce sur ses cases.

2.2 - Engine

Ce module existe afin de faire respecter les règles du jeu et transmettre les commandes du **Client** au **State**. Voici son diagramme UML :



Nous retrouvons donc une seule classe qui possède plusieurs fonctions utiles pour gérer les actions principales du jeu, telles que le mouvement des pièces, les attaques, et les vérifications des conditions de victoire. Pour cette dernière action, nous avons d'ailleurs créé une énumération appelée WinCondition afin de vérifier plus facilement la victoire d'un joueur. La voici :

enum WinCondition {None, FlagCaptured, NoValidMoves};

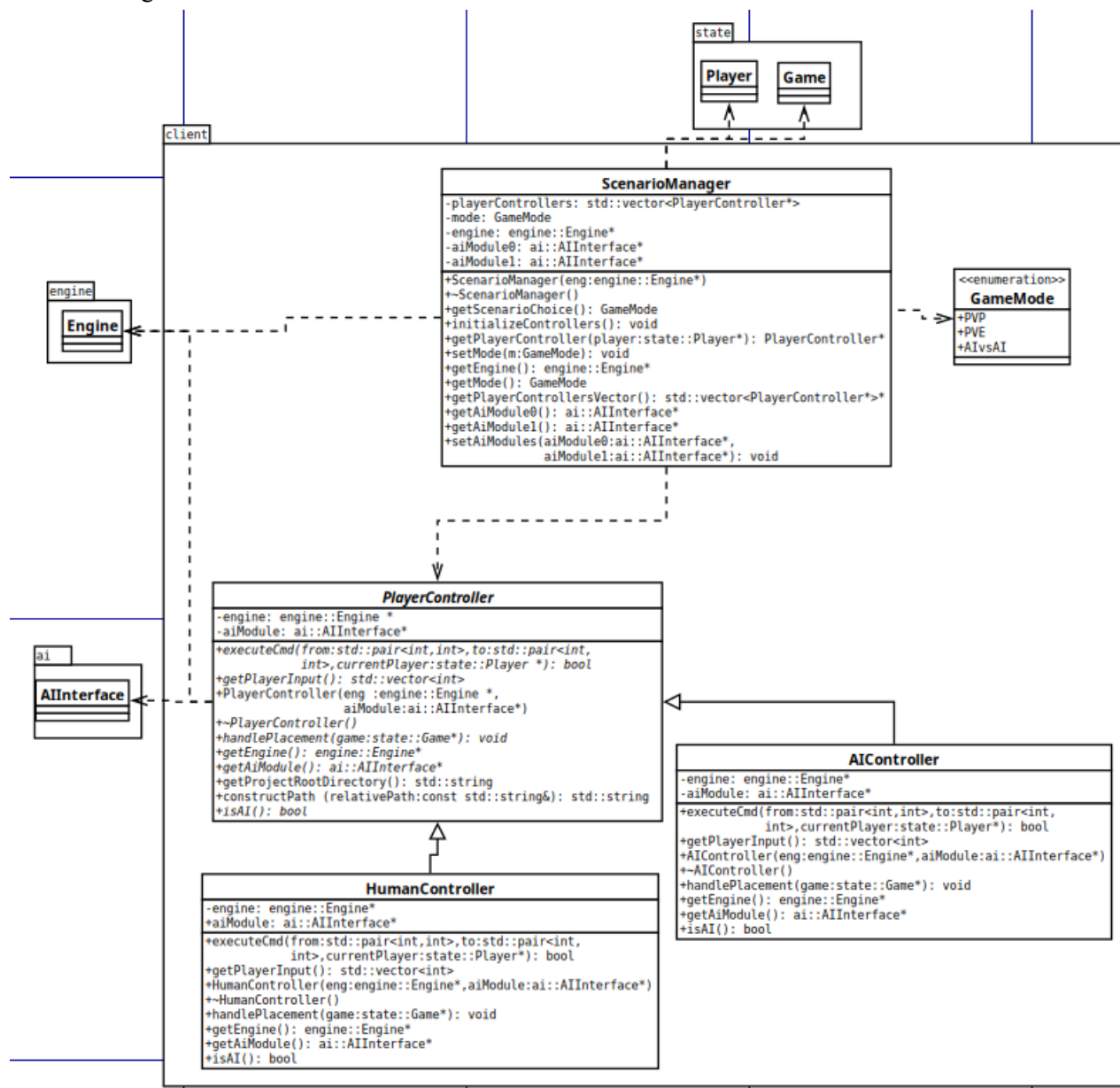
Nous expliquons maintenant les différents attributs et méthodes :

- **state::Game* game** : Cet attribut pointe vers l'état actuel du jeu, permettant à Engine de modifier ou d'interroger les informations du jeu en cours.
- **void attack (state::Pieces* piece, std::pair<int, int> position)** : Cette méthode gère la logique des attaques. Elle vérifie les interactions entre la pièce attaquante et la pièce cible, et met à jour les captures et le plateau selon le résultat de l'attaque. Les règles spécifiques, comme l'interaction entre une Bombe et un Mineur ou entre un Espion et un Maréchal, sont implémentées ici.
- **void endTurn()** : Cette méthode termine le tour en cours et passe le contrôle au joueur suivant. Elle fait appel à la fonction switchTurn().
- **bool handleCmdMove(std::pair<int, int> from, std::pair<int, int> to)** : Cette méthode gère une commande de déplacement. Elle vérifie que la pièce appartient au joueur actuel et que la destination est valide, puis gère les interactions sur la case cible (déplacement simple ou attaque) et termine le tour après le déplacement.
- **bool isValidMove(state::Pieces* piece, std::pair<int, int> to)** : Cette méthode vérifie si un déplacement vers une position cible est autorisé pour une pièce donnée.
- **bool handleCmdPlacement(std::string filePath)** : Cette méthode utilise loadConfig afin de charger une configuration sur le terrain puis termine le tour.
- **WinCondition checkWin()** : Cette méthode vérifie si une condition de victoire a été atteinte.

2.3 - Client

Ce module sert d'interface entre les joueurs (humains et IA) et l'Engine. Il récupère les instructions fournies par les utilisateurs et les transmet à l'Engine pour qu'elles soient validées et appliquées à l'état du jeu.

Voici son diagramme UML :



Ce module dispose donc de 2 classes principales :

i) ScenarioManager : gestion des scénarios de jeu

La classe **ScenarioManager** est responsable de l'initialisation du jeu. Elle permet de sélectionner le scénario souhaité parmi les options disponibles (Humain vs Humain, Humain vs IA ou IA vs IA) ainsi que de définir le niveau des joueurs IA lorsque cela est nécessaire.

Les modes de jeu sont définis dans une énumération :

enum Gamemode {PVP, PVE, AIvsAI};

Son fonctionnement repose sur deux méthodes ***getScenarioChoice()***, qui demande au joueur de choisir le mode de jeu souhaité, et ***initializeControllers()*** qui initialise les contrôleurs correspondant au mode choisi. Par exemple, pour le mode PVP (joueur contre joueur), deux instances de **HumanController** sont créées pour représenter les deux joueurs.

De plus, elle dispose de 3 attributs :

- ***std::vector<PlayerController*> playerControllers*** : Cet attribut contient les contrôleurs des deux joueurs participant au jeu, qu'ils soient humains ou IA.
- ***Gamemode mode*** : Indique le mode de jeu choisi par le joueur parmi les options disponibles (PVP, PVE, AIvsAI).
- ***ai::AIInterface* aiModule0/aiModule1*** : Représente le module d'intelligence artificielle utilisé par le premier joueur IA (si applicable). Ce module contient les algorithmes décisionnels nécessaires à l'exécution des actions par le joueur IA.

ii) PlayerController : communication entre les joueurs et l'Engine

La classe **PlayerController** assure la communication entre l'**Engine** et les joueurs. Pour gérer la dualité des types de joueurs (humains ou IA), cette classe est abstraite. Cela permet d'exploiter le polymorphisme et de traiter les contrôleurs des joueurs de manière uniforme, quel que soit leur type.

Les classes dérivées incluent :

- **HumanController** : Gère les interactions basées sur les saisies des utilisateurs en console.
- **AIController** : Repose sur le module IA du projet pour automatiser les actions du joueur IA.

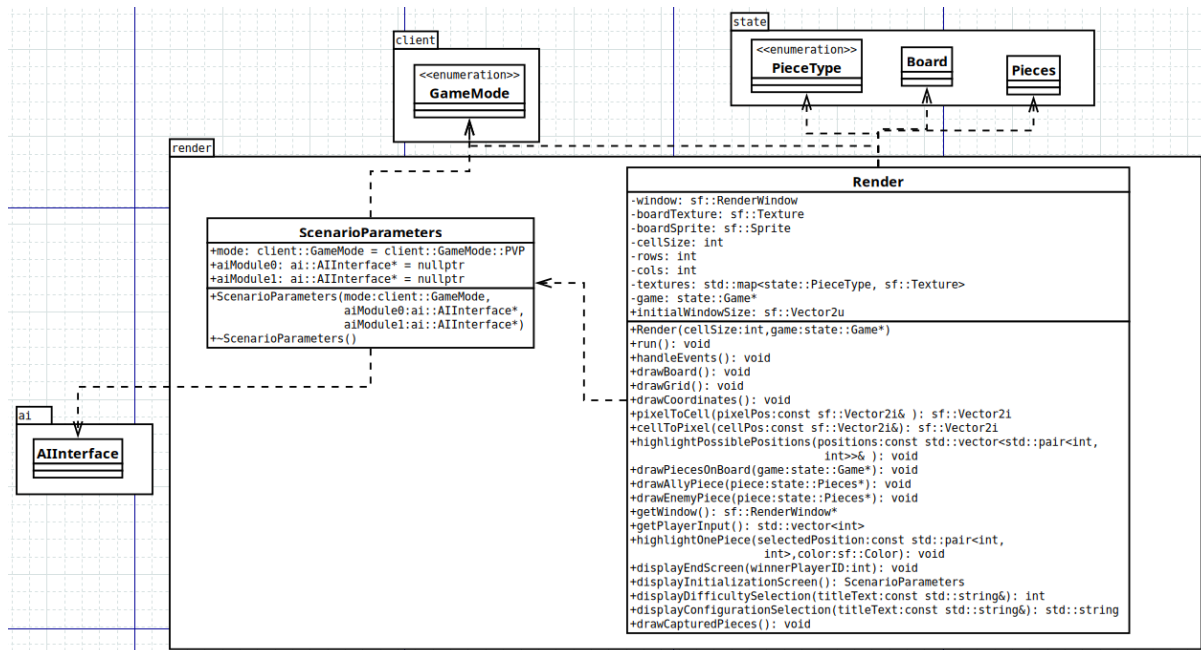
Les méthodes essentielles de cette classe sont :

- ***void handlePlacement(Game* game)*** : Permet au joueur de positionner ses pièces sur le plateau (Board) en début de partie. Cette phase utilise des configurations prédéfinies pour simplifier le placement.
- ***std::vector<int> getPlayerInput()*** : Permet au joueur de sélectionner la pièce à déplacer et sa destination en donnant les coordonnées de départ et de destination sur la grille.
- ***bool executeCmd(std::pair<int, int> from, std::pair<int, int> to, Player* currentPlayer)*** : Transmet les choix du joueur à l'Engine pour validation et exécution. L'Engine s'assure de la validité des commandes avant de les appliquer.

2.4 - Render

Le module Render gère l'affichage graphique du jeu Stratego à l'aide de la bibliothèque SFML. Il affiche le plateau, les pièces et leurs interactions en temps réel, tout en s'adaptant au redimensionnement de la fenêtre. Il gère les actions des joueurs, comme les clics, et met à jour dynamiquement l'affichage, en distinguant clairement les pièces alliées et ennemies.

Voici le diagramme UML :



Structure de la Classe Render :

Attributs principaux :

- ***sf::RenderWindow window*** : Fenêtre graphique dans laquelle tous les éléments du jeu sont affichés.
- ***sf::Texture boardTexture*** et ***sf::Sprite boardSprite*** : Texture et sprite pour représenter visuellement le plateau de jeu.
- ***std::map<state::PieceType, sf::Texture> textures*** : Conteneur associant chaque type de pièce (colonel, capitaine, etc.) à son image respective.
- ***sf::Vector2u initialWindowSize*** : Taille initiale de la fenêtre, utilisée pour maintenir des éléments graphiques stables même après redimensionnement.
- ***int cellSize, rows, cols*** : Dimensions du plateau, où chaque case est une cellule carrée.
- ***state::Game* game*** : Pointeur vers l'état actuel du jeu, utilisé pour accéder aux informations sur les pièces, les joueurs et le plateau.

Méthodes principales :

- ***void run()*** :
Boucle de test utilisée pendant la phase de développement afin de vérifier la gestion des événements utilisateur, le dessin des éléments graphiques.

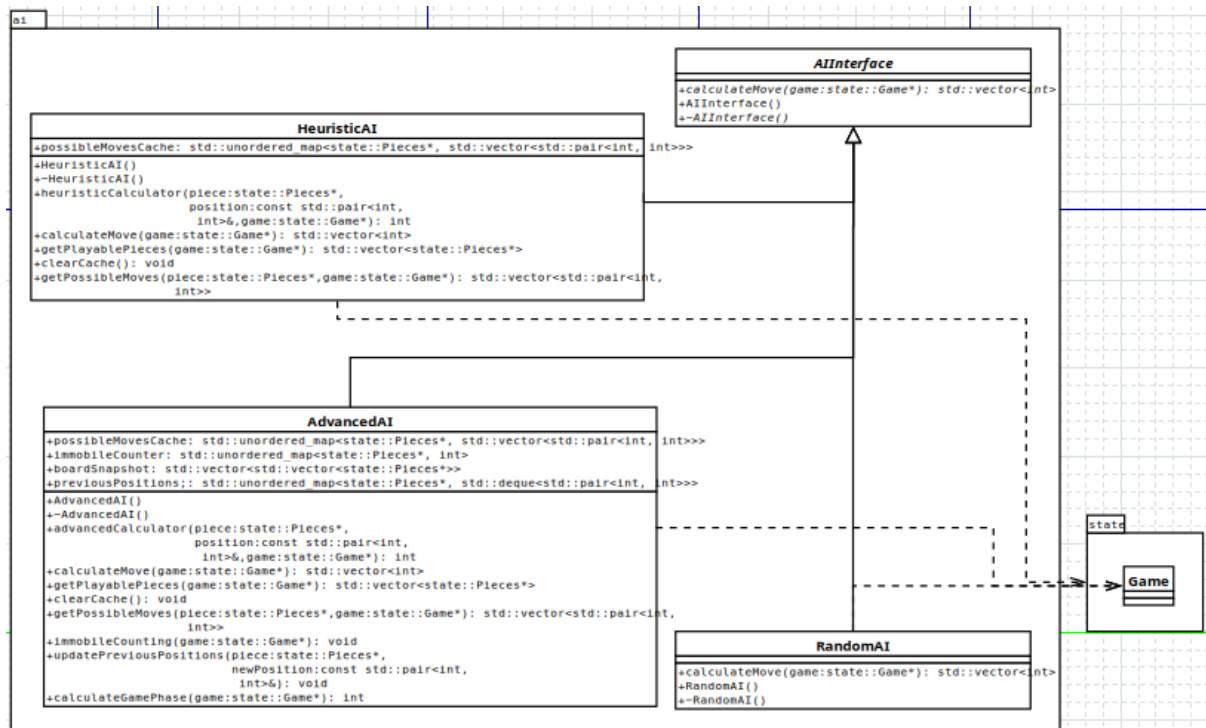
- ***void drawBoard()*** :
Affiche le sprite du plateau en ajustant automatiquement sa taille à celle du plateau logique.
- ***void drawGrid()*** :
Dessine les lignes de la grille du plateau pour séparer les cellules.
- ***void drawCoordinates()*** :
Affiche les indices des lignes et colonnes pour une meilleure compréhension des coordonnées des cellules.
- ***void drawAllyPiece()*** et ***void drawEnemyPiece()*** :
Affichent respectivement les pièces alliées et ennemies :
Les alliées sont affichées avec leurs textures respectives.
Les ennemies non révélées utilisent une texture de point d'interrogation.
- ***void drawCapturedPieces()*** :
Gère l'affichage des pièces capturées par les deux joueurs :
Les pièces capturées du joueur 1 sont affichées en bas à droite du plateau.
Les pièces capturées du joueur 2 sont affichées en haut à droite du plateau.
- ***void drawPiecesOnBoard(state::Game* game)*** :
Affiche toutes les pièces sur le plateau en fonction de leur position logique.
Distingue les pièces alliées (affichées avec leurs textures respectives) et ennemies (non révélées avec une texture générique) grâce aux 2 fonctions définies plus haut.
- ***void handleEvents()*** :
Capture et traite les interactions de l'utilisateur, notamment les clics de souris sur les cellules du plateau et les redimensionnements de la fenêtre.
- ***void highlightPossiblePositions()*** et ***void highlightOnePiece()*** :
Dessinent des effets visuels (couleurs semi-transparentes) pour indiquer les cases sélectionnées ou les mouvements possibles.
- ***sf::Vector2i pixelToCell(const sf::Vector2i& pixelPos)*** :
Convertit une position en pixels (coordonnées graphiques) en coordonnées logiques sur le plateau.
- ***sf::Vector2i cellToPixel(const sf::Vector2i& cellPos)*** :
Convertit des coordonnées logiques du plateau en une position en pixels pour l'affichage graphique.
- ***ScenarioParameters displayInitializationScreen()*** :
Affiche l'écran d'initialisation permettant de sélectionner le mode de jeu (Joueur contre Joueur, Joueur contre IA, IA contre IA).
- ***std::string displayConfigurationSelection(const std::string& titleText)*** :
Permet au joueur de choisir une configuration stratégique parmi plusieurs options (défensive, offensive, équilibrée).

- ***int displayDifficultySelection(const std::string& titleText) :***
Affiche une interface de sélection de difficulté pour le joueur ou l'IA.
Le titre et les boutons sont centrés et restent stables, quelle que soit la taille de la fenêtre.
- ***void displayEndScreen(int winnerPlayerID) :***
Affiche un écran de fin de jeu indiquant le gagnant avec un fond semi-transparent.

2.5 - AI

Ce module gère la logique de nos trois IA: la première joue de façon aléatoire, la deuxième joue de façon heuristique et la dernière joue de façon avancée. Ce module communique avec le **Client**.

Voici son diagramme UML :



Comme nous pouvons le voir, nos trois ia héritent d'une classe **AIInterface** qui communique avec le Client grâce à la fonction commune `std::vector<int> calculateMove(Game* game)`, qui détermine et lui renvoie les coordonnées de la pièce choisie puis celles de la nouvelle position.

Pour l'ia random, elle choisit simplement un pièce au hasard et une position prise aléatoirement dans la liste des positions possibles.

Pour l'ia heuristique, elle applique à chaque pièce associée à une position jouable un poids, elle choisit ensuite paire avec le poids le plus élevé.

Pour l'ia avancée, elle agit similairement à l'ia heuristique sauf que ses poids sont plus complexes et nombreux. De plus, idéalement, elle regarde les possibles états futurs du plateau avant de faire son choix final.

i) IA Random

Cette classe implémente une IA simple qui effectue des mouvements aléatoires basés sur les pièces jouables d'un joueur. Elle utilise un générateur de nombres aléatoires pour choisir une pièce et une position de destination valide.

Cette classe ne possède qu'une seule méthode :

- ***std::vector<int> calculateMove(state::Game* game)*** :
 - Cette méthode calcule un mouvement aléatoire pour une pièce appartenant au joueur courant.
 - Elle suit les étapes suivantes :
 1. Filtre les pièces du joueur pour ne conserver que celles ayant des mouvements possibles.
 2. Sélectionne aléatoirement une pièce parmi celles pouvant se déplacer.
 3. Détermine une position de destination valide aléatoire pour cette pièce.
 4. Retourne les coordonnées de départ et d'arrivée sous la forme d'un vecteur d'entiers : [startX, startY, endX, endY].
 - Si aucune pièce ne peut se déplacer, un message d'erreur est affiché, et un vecteur vide est renvoyé.

Cette IA est utilisée pour des tests ou des parties sans stratégie complexe. Sa simplicité en fait un bon point de départ pour expérimenter avec des mécaniques de jeu ou des comparaisons avec des IA plus sophistiquées.

ii) IA Heuristique

Cette classe implémente une IA heuristique pour le jeu de Stratego. Elle utilise des poids afin de favoriser certains mouvements plus que d'autres comme avancer par exemple.

Dans cette classe, nous retrouvons les attributs et méthodes suivantes :

Attributs :

- ***std::unordered_map<state::Pieces*, vector<pair<int,int>>> possibleMoveCache*** : Cet attribut renvoie une map des pièces associées aux positions qu'elles ont le droit de prendre. Cela évite d'avoir à recalculer toutes les positions possibles des différentes pièces à chaque besoin.

Méthodes :

- ***void clearCache()*** : Réinitialise le cache des mouvements possibles.
- ***std::vector<std::pair<int, int>> getPossibleMoves(state::Pieces piece, state::Game game)*** : Retourne tous les mouvements possibles pour une pièce donnée. Les mouvements sont calculés si non déjà présents dans le cache.
- ***int heuristicCalculator(state::Pieces* piece, pair<int,int>& position, Game* game)*** : Cette méthode permet de calculer le poids du mouvement de la pièce à cette position.
- ***std::vector<state::Pieces*> getPlayablePieces(state::Game game)*** : Retourne une liste des pièces du joueur actuel qui peuvent effectuer des mouvements valides.
- ***std::vector<int> calculateMove(state::Game game)*** : Identifie et retourne le meilleur mouvement possible (coordonnées de départ et d'arrivée) en fonction des évaluations heuristiques.

iii) IA Avancée

Cette classe implémente une IA avancée pour le jeu de Stratego. Elle utilise des méthodes avancées pour analyser les mouvements possibles, suivre l'historique des pièces, et déterminer les meilleures actions à entreprendre en fonction de l'état du jeu.

Voici une description simple et concise des attributs et méthodes de la classe AdvancedAI, la plupart sont identiques à celle de l'ia heuristique :

Attributs :

- ***std::unordered_map<state::Pieces*,vector<pair<int,int>>> possibleMoveCache*** : Idem que pour l'ia heuristique.
- ***std::unordered_map<state::Pieces*,int> immobileCounter*** : Cet attribut renvoie une map des pièces adverses associées à leur nombre de tours passées immobiles. Cela permet de déterminer si une pièce inconnue à un risque d'être une bombe ou le drapeau.
- ***std::vector<std::vector<state::Pieces*>> boardSnapshot*** : Cet attribut renvoie une copie du grid du tour précédent. Il permet de stocker une capture d'instantané de la disposition actuelle du plateau pour comparer les positions des pièces au fil des tours.
- ***std::unordered_map<state::Pieces*, std::deque<std::pair<int, int>>> previousPositions*** : Cet attribut permet de stocker l'historique des positions précédentes de chaque pièce pour éviter les mouvements répétitifs et analyser les comportements.

Méthodes :

- ***void clearCache()*** : Réinitialise le cache des mouvements possibles.
- ***std::vector<std::pair<int, int>> getPossibleMoves(state::Pieces piece, state::Game game)*** : Retourne tous les mouvements possibles pour une pièce donnée. Les mouvements sont calculés si non déjà présents dans le cache.
- ***int calculateGamePhase(state::Game game)*** : Détermine la phase actuelle du jeu (début, milieu, fin) en fonction du nombre de pièces restantes pour les deux joueurs.
- ***void updatePreviousPositions(state::Pieces piece, const std::pair<int, int>& newPosition)*** : Met à jour l'historique des positions d'une pièce en ajoutant la nouvelle position tout en limitant l'historique à 5 entrées.
- ***void immobileCounting(Game* game)*** : Cette méthode compte le nombre de tours d'immobilité pour chaque pièce. Plus précisément, elle met à jour les compteurs des pièces immobiles en comparant l'état actuel du plateau avec le dernier instantané.
- ***std::vector<state::Pieces*> getPlayablePieces(state::Game game)*** : Retourne une liste des pièces du joueur actuel qui peuvent effectuer des mouvements valides.
- ***int advancedCalculator(state::Pieces piece, const std::pair<int, int>& position, state::Game game)*** : Calcule une valeur de poids pour un mouvement spécifique en fonction de plusieurs critères comme la phase du jeu, la position de l'ennemi, et les règles stratégiques.
- ***std::vector<int> calculateMove(state::Game game)*** : Identifie et retourne le meilleur mouvement possible (coordonnées de départ et d'arrivée) en fonction des évaluations avancées.

Finalement, notre implémentation de l'IA avancée s'est soldée par un échec du point de vue technique même si cela nous a beaucoup appris sur la conception d'une heuristique mais aussi cela nous a permis de développer notre logique et réflexion.

2.6- Serveur

Une étude initiale a permis d'identifier les besoins spécifiques pour la mise en réseau du jeu :

- **Communication fiable** : Assurer une transmission correcte et cohérente des messages entre les joueurs via le serveur.
- **Synchronisation de l'état du jeu** : Maintenir une cohérence parfaite entre les versions locales des clients connectés et le serveur.
- **Flexibilité et évolutivité** : Permettre l'ajout futur de nouveaux scénarios ou fonctionnalités sans modifications majeures de l'architecture réseau.

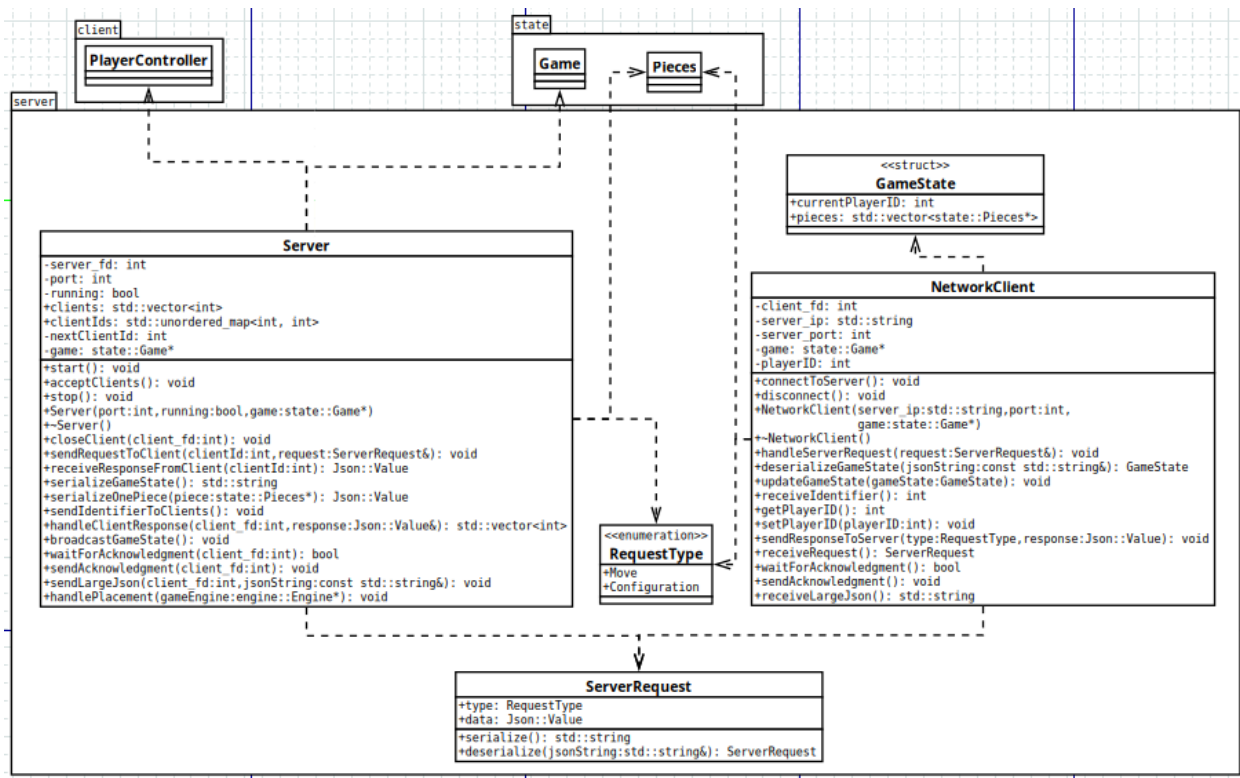
L'architecture choisie repose sur un modèle **client-serveur**, garantissant une gestion centralisée des échanges et de l'état du jeu :

- **Serveur** : Responsable de la gestion des connexions, de la validation des actions des joueurs, et de la synchronisation des états du jeu.
- **Client** : Chargé d'envoyer les actions du joueur et de mettre à jour l'affichage local en fonction des messages reçus du serveur.

Pour mettre en œuvre cette communication, nous avons utilisé des **sockets TCP**, assurant une transmission fiable et ordonnée des données. Afin de simplifier et d'optimiser la transmission, les données ont été sérialisées avant envoi. Nous avons fait le choix de ne transmettre que les informations relatives aux pièces, laissant au client la responsabilité de reconstruire l'état global du jeu à partir des attributs de ces pièces.

De plus, pour garantir le bon déroulement de la partie, un système d'**acknowledgment** (acquittement mutuel) a été mis en place. Chaque échange entre un client et le serveur est validé afin d'assurer une synchronisation précise et de prévenir tout désalignement entre les différents participants.

Nous avons donc implémenté les classes ainsi :



i) Server

Attributs

La classe maintient une gestion structurée des clients connectés et de la configuration réseau :

- ***std::vector<int> clients*** : Liste des descripteurs de fichier des clients connectés.
- ***std::unordered_map<int, int> clientIds*** : Association entre les descripteurs des clients et leurs identifiants uniques.
- ***int server_fd*** : Descripteur du socket serveur.
- ***int port*** : Port d'écoute du serveur.
- ***bool running*** : Indicateur de l'état du serveur (actif ou non).
- ***int nextClientId*** : Identifiant unique à attribuer au prochain client.

Elle maintient également une référence directe à l'état du jeu pour assurer la synchronisation.

Méthodes

Pour assurer une communication efficace entre le serveur et les clients, la classe propose des fonctionnalités suivantes :

- **Gestion des connexions et des ressources** :
 - ***void start()*** : Lance le serveur et commence à écouter les connexions.
 - ***void acceptClients()*** : Accepte les connexions entrantes et ajoute les clients à la liste.
 - ***void stop()*** : Arrête le serveur et libère les ressources.
- **Communication et synchronisation** :

- ***void sendRequestToClient(int clientId, ServerRequest& request)*** : Envoie une requête à un client spécifique. Les requêtes peuvent être de 2 types : ***Move*** et ***Configuration***. Ces types sont définis sous forme d'énumérés.
- ***Json::Value receiveResponseFromClient(int clientId)*** : Reçoit une réponse d'un client.
- ***void broadcastGameState()*** : Envoie l'état actuel du jeu à tous les clients.
- ***bool waitForAcknowledgment(int client_fd)*** : Attend l'acquittement d'un client pour valider une action.
- ***void sendAcknowledgment(int client_fd)*** : Envoie un acquittement au client.
- ***void sendLargeJson(int client_fd, const std::string& jsonString)*** : Transmet des données volumineuses au format JSON à un client. Les données sont découpées en chunk de 4096 octets.
- **Sérialisation des données :**
 - ***std::string serializeGameState()*** : Sérialise l'ensemble des pièces ainsi que le joueur courant.
 - ***Json::Value serializeOnePiece(state::Pieces* piece)*** : Sérialise les informations d'une pièce spécifique.
- **Gestion des interactions :**
 - ***std::vector<int> handleClientResponse(int client_fd, Json::Value& response)*** : Traite les réponses des clients en fonction du type de réponse et transmet les informations au moteur de jeu.
 - ***void handlePlacement(engine::Engine* gameEngine)*** : Gère le placement initial des pièces sur le plateau.

ii) Network Client

Attributs

Les attributs de la classe définissent les paramètres de connexion et les données spécifiques au client :

- ***int client_fd*** : Descripteur du socket utilisé pour communiquer avec le serveur.
- ***std::string server_ip*** : Adresse IP du serveur.
- ***int server_port*** : Port utilisé pour établir la connexion avec le serveur.
- ***state::Game* game*** : Référence au jeu local pour permettre la mise à jour de son état en fonction des données reçues.
- ***int playerId*** : Identifiant unique du joueur, attribué par le serveur.

Méthodes

Les méthodes principales de la classe sont regroupées en plusieurs catégories fonctionnelles :

- **Gestion de la Connexion :**
 - ***void connectToServer()*** : Établit la connexion avec le serveur en utilisant les paramètres *server_ip* et *server_port*.
 - ***void disconnect()*** : Ferme la connexion avec le serveur et libère les ressources réseau.
- **Communication avec le Serveur :**
 - ***void sendResponseToServer(RequestType type, Json::Value response)*** : Envoie une réponse au serveur contenant le type de requête.
 - ***ServerRequest receiveRequest()*** : Reçoit une requête du serveur.

- ***void sendAcknowledgment()*** : Envoie un acquittement au serveur pour valider une requête reçue.
- ***bool waitForAcknowledgment()*** : Attend un acquittement du serveur pour confirmer la réception d'une réponse du client.
- ***std::string receiveLargeJson()*** : Reçoit des données volumineuses au format JSON provenant du serveur. Les données sont reçues par chunk de 4kB.
- **Gestion des Données du Jeu :**
 - ***GameState deserializeGameState(const std::string& jsonString)*** : Convertit une chaîne JSON reçue en un état du jeu.
 - ***void updateGameState(GameState gameState)*** : Met à jour l'état local du jeu en fonction des données reçues du serveur. Cette fonction reconstruit l'état du jeu à partir des attributs des pièces.
 - ***void handleServerRequest(ServerRequest& request)*** : Traite une requête reçue du serveur et demande à l'utilisateur de saisir les données correspondantes.
- **Gestion de l'Identifiant Joueur :**
 - ***int receiveIdentifier()*** : Reçoit et retourne l'identifiant unique attribué au joueur par le serveur.

Seul le jeu dans sa version console a été implémenté en réseau.

III. Analyse & Rendu

3.1 - Tests réalisés

Afin d'assurer la qualité et la fiabilité des fonctionnalités du jeu tout en détectant rapidement les éventuels problèmes pendant le développement, nous avons utilisé la bibliothèque **Boost Unit Test Framework**.

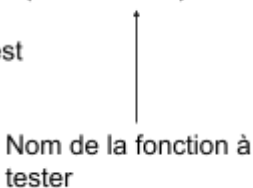
L'objectif principal de ces tests est double :

1. Vérifier que les fonctions se comportent conformément aux attentes.
2. S'assurer que les cas d'erreurs sont correctement gérés.

Pour atteindre cet objectif, nous avons adopté une approche structurée en créant un fichier de test dédié pour chaque classe. Ces fichiers sont organisés sous forme de suites de tests, où chaque fonction d'une classe est testée individuellement. Une exception est faite pour les constructeurs, qui sont généralement testés conjointement avec les getters, ce qui simplifie la détection et la correction des erreurs potentielles.

Pour cela, nous avons utilisé la macro `BOOST_AUTO_TEST_CASE` qui permet de délimiter les différents tests. Elle se présente sous cette forme :

```
BOOST_AUTO_TEST_CASE(testFonction)
{
    //implémentation du test
}
```



Nom de la fonction à tester

Le nom de la fonction permet de trouver facilement les tests qui ont échoué lors de leur exécution.

Lors de l'implémentation des tests, nous avons identifié deux types principaux d'actions réalisées par nos fonctions :

1. **Gestion des éléments du jeu** : Les fonctions manipulent les différents composants du jeu en effectuant des opérations comme copier, déplacer ou fournir des informations sur ces éléments.
2. **Affichage de messages** : Elles affichent des messages pour signaler des erreurs ou fournir des informations lors de certains coups spécifiques.

Pour garantir une validation optimale de ces actions, nous avons opté pour des tests unitaires. Ces derniers permettent de vérifier chaque fonction de manière isolée afin d'assurer leur bon fonctionnement.

Pour mettre en œuvre ces tests, nous avons utilisé la macro `BOOST_CHECK`, qui évalue des conditions sous forme de booléens. Cela nous permet de comparer les résultats attendus avec les résultats obtenus, assurant ainsi une détection rapide et précise des éventuelles anomalies.

Lors de la conception des tests, notre priorité a également été d'assurer une couverture maximale du code. L'objectif était de tester chaque ligne afin de garantir qu'aucune partie du code ne reste non vérifiée. Voici le rapport de code coverage de notre projet.

LCOV - code coverage report

Current view: [top level](#)

Test: [code-coverage.info.cleaned](#)

Test Date: 2025-01-10 00:08:28

Lines:

Functions:

Coverage

36.8 %

57.9 %

Total

1739

164

Hit

640

95

Directory	Line Coverage ↕			Function Coverage ↕		
	Rate	Total	Hit	Rate	Total	Hit
client	<div><div></div></div> 0.0 %	96		<div><div></div></div> 0.0 %	3	
client/client	<div><div></div></div> 94.5 %	218	206	<div><div></div></div> 94.3 %	35	33
client/render	<div><div></div></div> 0.0 %	506		<div><div></div></div> 0.0 %	24	
server	<div><div></div></div> 0.0 %	67		<div><div></div></div> 0.0 %	3	
server/server	<div><div></div></div> 0.0 %	398		<div><div></div></div> 0.0 %	35	
shared/ai	<div><div></div></div> 100.0 %	38	38	<div><div></div></div> 85.7 %	7	6
shared/engine	<div><div></div></div> 98.0 %	98	96	<div><div></div></div> 100.0 %	8	8
shared/state	<div><div></div></div> 94.3 %	318	300	<div><div></div></div> 98.0 %	49	48

Generated by: [LCOV version 2.0-1](#)

Les quatre modules principaux constituant le cœur du jeu — **State**, **AI**, **Engine**, et **Client** — ont été testés avec une couverture d'au moins **90 %**.

Cependant, nous n'avons pas pu atteindre une couverture de **100 %** dans certains cas, notamment dans les classes abstraites comme AIInterface dans le module **AI**. Ces limitations sont dues à la nature même des classes abstraites, qui ne contiennent pas d'implémentations concrètes à tester directement.

3.2 - Problèmes rencontrés et solutions

Nous avons rencontré bon nombre d'obstacles sur notre chemin pour arriver là où nous en sommes actuellement. Dans cette partie, nous allons les détailler dans l'ordre chronologique tout en les accompagnant de la solution trouvée pour le résoudre.

i) Modularité

Nous avons perdu beaucoup de temps sur la définition de la structure du projet. Dans un premier temps,nous nous sommes tournés vers ce que nous avions de plus proche et similaire : notre exercice de jeu Pokémon codé en début d'année en C++. Dans ce jeu, nous avançons dans la partie à l'aide d'une machine à états, cela nous a alors paru logique de faire la même chose ici, c'est pourquoi nous avons réfléchi à différents états qui pourraient correspondre au jeu Stratego tels que PlacementState, TurnState ou WinState par exemple. Une fois le jeu fonctionnel sous cette forme, il nous a été signifié lors d'une revue de code, que la structure était à revoir pour arriver à la structure complètement modulaire. Nous avons donc repensé notre projet en 5 modules (défini précédemment). Cela ne signifie pas reprendre de zéro mais plutôt déplacer certaines fonctions vers d'autres classes/modules ou bien les transformer légèrement. De plus, nos tests unitaires étaient correctement implémentés à ce stade du projet ce qui a grandement facilité cette réorganisation.

ii) L'IA avancée

L'implémentation de cette IA a été très compliquée, non pas par l'écriture du code en lui-même, mais plutôt dû aux différents poids nécessaires pour réussir à avancer dans le jeu. En effet, il fallait mettre des poids élevés aux déplacements vers l'avant, i.e vers l'adversaire, mais également faire en sorte que notre drapeau soit protégé. Nous avons pensé aussi à mettre un poids plus élevé si l'ia jouait des

Éclaireurs au début de la partie et les Démineurs vers la fin. Malgré tous nos efforts, cette IA n'a jamais voulu fonctionner correctement, nous avons identifié deux problèmes majeurs :

- **les boucles** : parfois, l'IA décidait de jouer une pièce (un Éclaireur par exemple ou un Démineur) et de la déplacer d'une case, puis, au tour suivant, de venir la remettre à sa position initiale, ce qui l'empêchait d'avancer dans la partie. Nous avons tenté de résoudre ce problème en limitant le retour sur les anciennes positions en diminuant énormément le poids, mais cela n'a pas suffi
- **les alliés** : ce problème est quelque peu lié au précédent puisque les alliés "ordinaires" n'avaient pas de condition, donc moins de poids que les autres pièces dites "spéciales" donc l'Éclaireur et le Démineur qui nécessitaient tous deux des bonus, comme l'avancée des Éclaireurs ou la conservation des Démineurs. Cela a entraîné un véritable blocage au sein du jeu puisque toutes les pièces ordinaires ne bougeaient pas, ce qui ne permettait pas de libérer le jeu et de faire apparaître d'autres conditions, notamment celle sur l'attaque.

Bien que le développement de cette IA avancée fut un échec technique, nous en ressortons avec une nouvelle expérience et une connaissance accrue du sujet, ce qui est toujours bon à prendre.

3.3 - Conclusion et points d'amélioration

Ce projet nous a beaucoup appris, notamment sur la communication intra-équipe, la répartition du travail au sein d'un groupe, l'importance de la modularité d'un projet, l'utilisation d'un outils de gestion de version logiciel comme Git mais également d'un point de vue technique où nous nous sommes renseignés sur plein de sujets divers, nous pensons par exemple au SFML pour le Render ou à la création de la partie Serveur par exemple.

Un axe d'amélioration possible aurait été de résoudre les problèmes liés à l'IA avancée et même de la développer encore plus en lui ajoutant des fonctionnalités comme, par exemple, le calcul de coups en avance en utilisant des copies de board et de game à part, où elle pourrait tester des probabilités de futurs coups de son adversaire.

Un autre axe d'amélioration serait la mise en place de l'utilisation du render lors d'un fonctionnement en réseau.