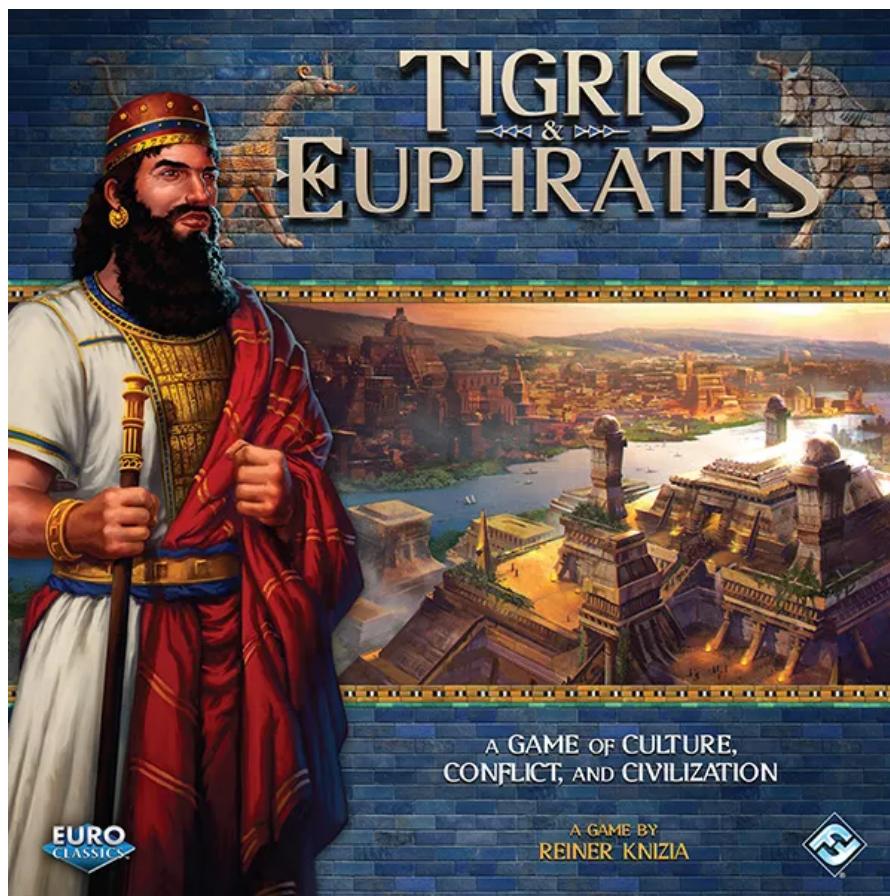




Ecole Nationale  
Supérieure  
de l'Électronique  
et de ses Applications

# Projet Logiciel Transversal

Option Informatique Et Systèmes  
3ème année



Bruno MACHADO - Rokhaya SOUMARÉ - Ying ZHOU

# Table des matières

<b>1 Présentation Générale</b>	<b>3</b>
1.1 Introduction	3
1.2 Règles du jeu	3
1.3 Ressources	8
<b>2 Description et conception des états</b>	<b>10</b>
2.1 Description des états	10
2.2 Conception Logiciel	12
<b>3 Rendu: Stratégie et Conception</b>	<b>14</b>
3.1 Stratégie de rendu d'un état	14
3.2 Conception logiciel	15
<b>4 Moteur de jeu</b>	<b>17</b>
4.1 Règles de changement d'états	17
4.2 Conception logiciel	17
<b>5 Intelligence Artificielle</b>	<b>20</b>
5.1 Stratégies	20
5.1.1 Intelligence aléatoire	20
5.1.2 Intelligence basée sur des heuristiques	20
5.1.3 Intelligence avancée	21
5.2 Conception logiciel	23
<b>6 Modularisation</b>	<b>25</b>
6.1 Organisation des modules	25
6.1.1 Répartition sur différents threads	25
6.1.2 Répartition sur différentes machines: rassemblement des joueurs	25
6.1.3 Répartition sur différentes machines: échange des commandes	27
6.2 Conception logiciel	29

# 1 Présentation Générale

## 1.1 Introduction

L'objectif de ce projet est d'implémenter le jeu de société Tigris & Euphrates, créé par Reiner Knizia et publié pour la première fois en 1997. Dans ce jeu, chaque joueur doit développer la civilisation mésopotamienne dans quatre domaines distincts (**établissements**, **temples**, **fermes** et **marchés**). Pour cela, les joueurs vont utiliser leurs leaders, créer et étendre des royaumes, bâtir des monuments et résoudre des conflits, gagnant ainsi des points de victoire dans les différents domaines. Le vainqueur sera celui qui contribue le plus, et de manière équilibrée, au développement de la civilisation.

## 1.2 Règles du jeu

Le jeu se joue à tour de rôle, à chaque tour le joueur actif peut faire 2 actions parmi:

- Placer, déplacer ou retirer un leader.
- Placer une tuile.
- Jouer une tuile catastrophe.
- Remplacer jusqu'à six tuiles de sa main.

Chaque action est détaillée ci-dessous:

### Placer, déplacer ou retirer un leader

Chaque joueur possède une dynastie de quatre leaders: un **roi**, un **prêtre**, **agriculteur** et **commerçant** (voir Figure 4). Un leader peut venir de la main du joueur ou être repositionné à partir d'un autre espace sur le plateau. En outre, un leader peut être retiré du plateau. Un leader doit toujours être adjacent à un **temple** (toutes les tuiles sont indiquées à la figure 2). Le placement du leader doit respecter les restrictions suivantes:

- ❖ Un leader doit être placé dans un espace vide.
- ❖ Un leader ne peut être placé qu'à côté d'un **temple**.
- ❖ Un leader ne peut pas être placé sur une rivière.
- ❖ Un leader ne peut être placé de manière à unir les royaumes.

Une région est toute zone sur le plateau de jeu couverte par une tuile seule ou deux ou plus tuiles adjacentes. Une région qui a au moins un leader adjacent est un royaume. Un royaume peut contenir plusieurs leaders de différentes couleurs. Les conflits surgissent quand il y a deux leaders de même couleur dans un royaume.

## Placer une tuile

Un joueur aura six tuiles dans sa main au début de son tour. Les joueurs utilisent ces tuiles pour étendre les régions, renforcer les dirigeants et gagner des points de victoire. Pour placer une tuile, un joueur prend une tuile de sa main et le met à bord selon les restrictions suivantes:

- ❖ Une tuile doit être placée sur un espace vide.
- ❖ Les **fermes** ne peuvent être placées que sur les rivières.
- ❖ Les **établissements**, les **temples** et les **marchés** ne peuvent pas être placés sur les rivières.
- ❖ Une tuile ne peut pas être placée là où elle unirait plus que deux royaumes.

Après qu'une tuile est placée, elle ne peut pas être déplacée. Si une tuile est placée dans un royaume, et que ce royaume contient un leader de cette même couleur, alors un point de victoire de cette couleur est gagné par le joueur qui contrôle ce leader. S'il n'y a pas de leader de la même couleur, mais qu'il y a un **roi** (leader noir) dans le royaume, alors un point de victoire de la couleur de la tuile est gagné par le joueur qui contrôle le roi. Si la tuile est placée de sorte qu'elle unit deux royaumes, même s'il n'y a pas de conflits, aucun point de victoire n'est gagné.

## Jouer une tuile catastrophe

En début de partie, chaque joueur reçoit deux tuiles catastrophe. Une fois joué, la tuile catastrophe reste sur le plateau pour le reste du jeu. Il crée un espace bloqué qui ne peut pas être utilisé à nouveau. Un joueur peut placer des tuiles catastrophe sur un espace vide ou sur une tuile existante. Une tuile catastrophe ne peut être jouée sur une tuile portant un trésor ou un monument ni sur un espace occupé par un leader.

## Remplacer jusqu'à six tuiles de sa main

Un joueur peut prendre jusqu'à six tuiles de sa main et les jeter. Ensuite, il tire le même nombre de tuiles et les ajoute à sa main.

Le tour d'un joueur se termine après ses deux actions. Si un ou plusieurs de ses dirigeants sont liés avec des monuments, il peut gagner des points de victoire supplémentaires. Enfin, tous les joueurs tirent jusqu'à ce que chacun ait 6 tuiles en main. Ensuite, c'est au tour du joueur suivant.

## Autres règles pertinentes:

### Trésors

Chaque trésor compte comme un point de victoire joker. À la fin du jeu, vous pouvez attribuer chaque trésor individuellement à n'importe quel type de point de victoire de votre choix.

La distribution de trésor peut être initiée quand un joueur positionne un leader ou place une tuile. Au début du jeu, dix jetons trésor sont placés sur les dix premiers temples du plateau. Les trésors sont gagnés de la manière suivante :

- ❖ Si un royaume contient plus d'un trésor à la fin de l'action d'un joueur, le **commerçant** dans ce royaume doit gagner tous les trésors sauf un. S'il y a des trésors sur un espace spécial de frontière, le joueur doit prendre des trésors de ces espaces d'abord; après cela, il peut choisir de quel espace retirer ses trésors restants.
- ❖ S'il n'y a pas de **commerçant** dans le royaume, les trésors y restent jusqu'à ce que le royaume ait un commerçant.

### Monuments

La construction d'un monument peut se produire lorsque quelqu'un place une tuile. Il y a six monuments dans le jeu. Chaque monument a deux couleurs différentes. Si le joueur actif place une tuile de telle manière qu'il crée un carré 2x2 de quatre tuiles de la même couleur, il peut placer un monument. Une couleur du monument doit correspondre aux tuiles. Si cette couleur n'est plus disponible, le monument ne peut pas être construit. Un monument ne peut pas être détruit. Lorsque quatre temples sont utilisés pour construire un monument, ce qui suit s'applique:

- ❖ Si un trésor se trouve sur un temple, il reste sur cette tuile.
- ❖ Si un leader n'est plus adjacent à un temple, retirez-le et remettez-le au joueur.

Un monument fournit régulièrement des points de victoire. A la fin du tour du joueur actif, il détermine si un ou plusieurs de ses dirigeants sont dans le même royaume qu'un monument de couleur semblable. Pour chacun de ces leaders, il gagne un point de victoire de la couleur du leader pour chaque monument de même couleur dans le même royaume. Un **roi** ne gagne un point de victoire que lorsqu'il est relié à un monument noir.

## Conflits

Des conflits peuvent survenir lorsque quelqu'un positionne un leader ou place une tuile. Lorsqu'un conflit survient, il est résolu avant la reprise du jeu. Il y a 2 types de conflits dans le jeu :

➤ **Révolte:** Une révolte se produit quand un leader est positionné dans un royaume qui contient déjà un leader de la même couleur. Lors d'une révolte, le joueur qui positionne le nouveau leader est l'attaquant. Le contrôleur du leader existant de la même couleur dans ce royaume est le défenseur. L'attaquant et le défenseur tirent leur force des **temples**. Résolvez les révoltes comme suit:

- ❖ Comptez les **temples** adjacents aux leaders: L'attaquant et le défenseur tirent leur force en comptant le nombre de **temples** adjacents à leurs leaders en conflit. Un **temple** peut compter pour les deux leaders.
- ❖ Les joueurs engagent des **temples** de leurs mains: D'abord l'attaquant, puis le défenseur, peuvent augmenter leur force en engageant au conflit des **temples** supplémentaires de leurs mains. Chaque joueur peut engager des **temples** supplémentaires seulement une fois. Celui qui a la force totale supérieure (nombre total de **temples**) gagne le conflit. Le défenseur gagne en cas d'égalité.

Les conséquences d'une révolte sont les suivantes:

- ❖ Le perdant doit retirer son leader du plateau et le remettre dans sa main.
- ❖ Le vainqueur gagne un point de victoire rouge.
- ❖ Les deux joueurs retirent tous les **temples** engagés du jeu.

➤ **Guerres:** les guerres se produisent lorsque deux royaumes sont unis et que le nouveau royaume agrandi contient des leaders de la même couleur. Aucun point de victoire n'est gagné quand une tuile est placée de sorte à unir deux royaumes. Au lieu de cela, cette tuile est couverte par la tuile d'unification.

Tous les conflits entre deux leaders de même couleur sont résolus individuellement. Chaque fois qu'il y a des guerres entre leaders dans plus d'une couleur, le joueur actif décide quelle guerre à résoudre en premier. Pour chaque guerre après la première, les joueurs résolvent la guerre en utilisant la configuration actuelle du royaume, car les guerres récentes ont pu perturber le lien entre les leaders et les tuiles. Une guerre est résolue comme suit:

- ❖ Déterminez qui est l'attaquant : Si le joueur actif choisit de résoudre un conflit qui implique l'un de ses dirigeants, il devient l'attaquant. Sinon, le joueur suivant (dans le sens horaire) avec un leader impliqué devient l'attaquant. L'autre joueur impliqué dans le conflit est le défenseur.
- ❖ Comptez les supporters: L'attaquant et le défenseur tirent la force en comptant les partisans de leur leader, qui sont des tuiles de couleur semblable sur le tableau qui sont encore connectés au leader. Ces tuiles ne doivent pas être adjacentes au leader respectif. Elles doivent seulement être dans cette partie du royaume sur son segment de la tuile d'unification.
- ❖ Les joueurs engagent des tuiles de leurs mains: D'abord l'attaquant, puis le défenseur, augmentent la force de leurs partisans. Ils le font en engageant au conflit des tuiles supplémentaires de la couleur correspondante de leurs mains. Chaque joueur peut engager des tuiles seulement une fois.

Celui qui a la force totale la plus élevée gagne le conflit. Le défenseur gagne en cas d'égalité. Les conséquences d'une guerre sont les suivantes :

- ❖ Le perdant doit retirer son leader et retirer tous les partisans du royaume.
- ❖ Le vainqueur reçoit un point de victoire qui correspond à la couleur du leader retiré et un (également de couleur correspondante) pour chacun des supporters qui sont retirés du tableau. Les tuiles engagées de la main d'un joueur ne comptent pas pour les points de victoire.
- ❖ Le leader défait retourne dans la main du joueur perdant.
- ❖ Les partisans du meneur défait et toute tuile engagée ajoutée par les deux joueurs sont retirés du jeu.
- ❖ Si deux **prêtres** causent un conflit, alors il y a une exception: les **temples** qui portent un trésor, ou ont un autre leader adjacent à eux, ne sont pas enlevés.

## Fin de jeu

Le jeu se termine lorsque seulement un ou deux trésors restent sur le plateau au fin du tour d'un joueur, ou lorsqu'un joueur souhaite poser une tuile mais ne peut pas le faire car la pioche de tuile est épuisée. Chaque joueur détermine dans quelle sphère ils possèdent le moins de points de victoire, chaque joueur allouant librement ses trésors à n'importe quelle sphère. Le joueur dont la sphère la plus basse a le plus grand nombre de points de victoire est le gagnant. Dans le cas d'une égalité, les joueurs à égalité comparent leur deuxième plus basse sphère, et ainsi de suite.

## 1.3 Ressources

L'image du plateau est illustrée à la figure 1. La figure 2 montre tous les différents types de tuiles et figure 3 les différents types de points de victoire. Les leaders de 4 joueurs différents (un symbole différent pour chaque joueur) sont montrés à la figure 4 et la figure 5 montre tous les monuments possibles.

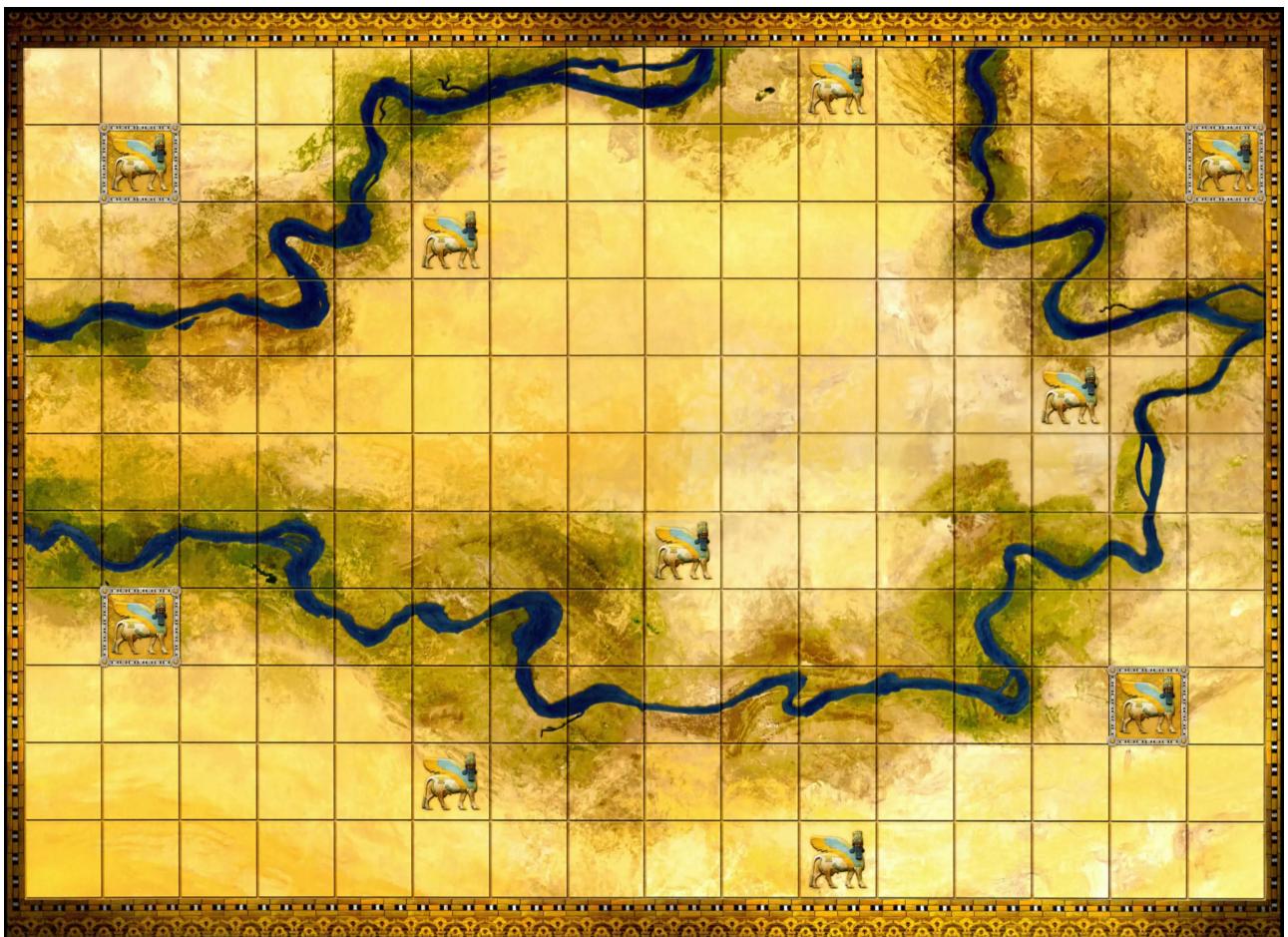


Figure 1. Image du plateau.



Figure 2. Image des différents types de tuiles.



Figure 3. Image des différents types de points de victoire.



Figure 4. Image des différents types de leaders pour 4 joueurs.



Figure 5. Image de tous les monuments.

## 2 Description et conception des états

### 2.1 Description des états

Un état de jeu se compose d'un plateau et des joueurs. Un joueur peut contenir des leaders et des tuiles tandis que le plateau contient des régions, qui à leur tour peuvent contenir des leaders, des tuiles, des monuments et des trésors, comme détaillé ci-dessous:

- ❖ Chaque **joueur** possède quatre leaders, qui peuvent être dans ses mains ou sur le plateau. Chaque joueur peut avoir aussi de 0 à 6 tuiles dans sa main. Un joueur a à sa disposition deux jetons catastrophe qui peuvent être joués tout au long du match. Tout au long du match, un joueur peut recevoir des points de victoire de quatre types différents. Il peut aussi gagner des trésors.
- ❖ Le **plateau** est composé d'une grille 16 x 11. Il y a des espaces de type terrain et rivière. Il y a aussi des espaces de terrain délimités pour commencer avec un **temple** sur eux. Le plateau est divisé en régions. Quand une catastrophe est jouée, cet espace du plateau est détruit et rien ne peut être joué à cette position.
- ❖ Une **région** est générée lorsqu'une tuile est posée sur le plateau dans une position où il n'y a pas d'autre tuile adjacente. Lorsqu'une tuile, un leader ou un monument entrent en jeu à côté d'une région, il pénètre dans cette région. Toutes les tuiles, leaders et monuments sur le plateau appartiennent à une région. Le jeu commence avec dix régions distinctes, une pour chacune des dix **temples** initiaux. Chacune de ces régions commence avec un trésor. Quand une région contient un leader, elle est un royaume. Lorsqu'une région contient plus d'un leader du même type, elle est en conflit et peut être une guerre ou une révolte.
- ❖ Un **leader** peut être dans la main d'un joueur ou dans une région. Peu importe où il se trouve, un leader appartiendra toujours au même joueur. Un leader aura un type (**roi**, **prêtre**, **agriculteur** ou **commerçant**) et aura une position (quand il est sur le plateau). Un leader a également une force, correspondant au nombre de tuiles de la même couleur dans la région dans laquelle il se trouve.
- ❖ Une **tuile** est définie par un type (**établissement**, **temple**, **ferme** ou **marché**) et une position (quand elle est sur le plateau).
- ❖ Un **trésor** aura une position sur la carte. Il peut ou non être spécial en fonction de votre position de départ.
- ❖ Un **monument** aura deux couleurs et une position (quand il est sur le plateau).

- ❖ Une **position** est définie par une valeur i (de 0 à 10) et une valeur j (de 0 à 15).

En plus des joueurs et du plateau de jeu, un **état** de jeu est composé d'un marqueur de tour, d'un marqueur du joueur actif, d'un marqueur pour le nombre d'actions effectuées par le joueur actif (chaque joueur peut effectuer deux actions sur son tour). Un état aura également une liste contenant le nombre de tuiles restantes (qui n'ont pas encore été piochées par un joueur) et un marqueur de combien de trésors il reste sur le plateau.

## 2.2 Conception Logiciel

Pour chacun des éléments décrits au chapitre 2.1, nous créerons une classe, à l'exception de la position, qui sera une structure. Tous les attributs et les méthodes de chaque classe sont présentés dans le diagramme de classes, illustré à la figure 6, mais certains détails des classes sont expliqués ci-dessous:

- ❖ **Classe Player:** L'attribut *id* est utilisé pour distinguer les différents joueurs. Le nombre de trésors qu'un joueur possède est stocké dans un *unordered\_map* avec les points de victoire. Puisqu'une fois qu'un joueur reçoit un trésor, il ne peut plus être perdu et il n'est plus affiché sur l'écran, nous supprimons l'objet et incrémentant le nombre de trésors sur le *unordered\_map*.
- ❖ **Classe Board:** Le plateau est composé de plusieurs régions, stockées dans un *unordered\_map*. Nous utilisons trois cartes (taille 11 x 16) pour suivre l'état du plateau. Le *regionMap* contient -1 quand il n'y a pas de tuile dans cette position et l'ID de la région correspondante autrement. Le *terrainMap* indique si cette position contient une rivière ou non. Le *boardStateMap* commence comme une copie de *terrainMap*, et garde des traces de tuiles, chefs, monuments et catastrophes quand ils sont ajoutés au plateau. Dans la liste *monuments*, les six monuments possibles sont générés et ajoutés avec des positions négatives (pour indiquer qu'ils ne sont pas construits). Cela se fait parce que ces monuments seront affichés sur l'écran, à la droite du plateau. Lorsqu'un monument est construit, il recevra une position valide et sera ajouté à la région correspondante, en laissant la liste des monuments du plateau.
- ❖ **Classe Region:** L'attribut *regionID* est utilisé pour distinguer les différentes régions. Il y a quatre vecteurs pour stocker les différents objets, un attribut pour marquer la présence d'un leader dans la région et deux autres attributs pour marquer la présence d'un conflit dans la région. Lorsqu'une guerre est déclenchée, la position de la tuile qui a commencé la guerre est stockée dans l'attribut *unificationTilePosition*, car elle sera nécessaire pour la résolution du conflit.
- ❖ **Classe Leader:** L'attribut *playerID* est utilisé pour savoir à quel joueur le leader appartient.
- ❖ **Struct Position:** Quand un objet n'est pas sur le plateau, nous lui attribuons une position négative.
- ❖ **Classe State:** Notez que cette classe garde également une trace des deux conditions de fin de partie (nombre de trésors restant sur le plateau et nombre de tuiles disponibles pour le tirage).

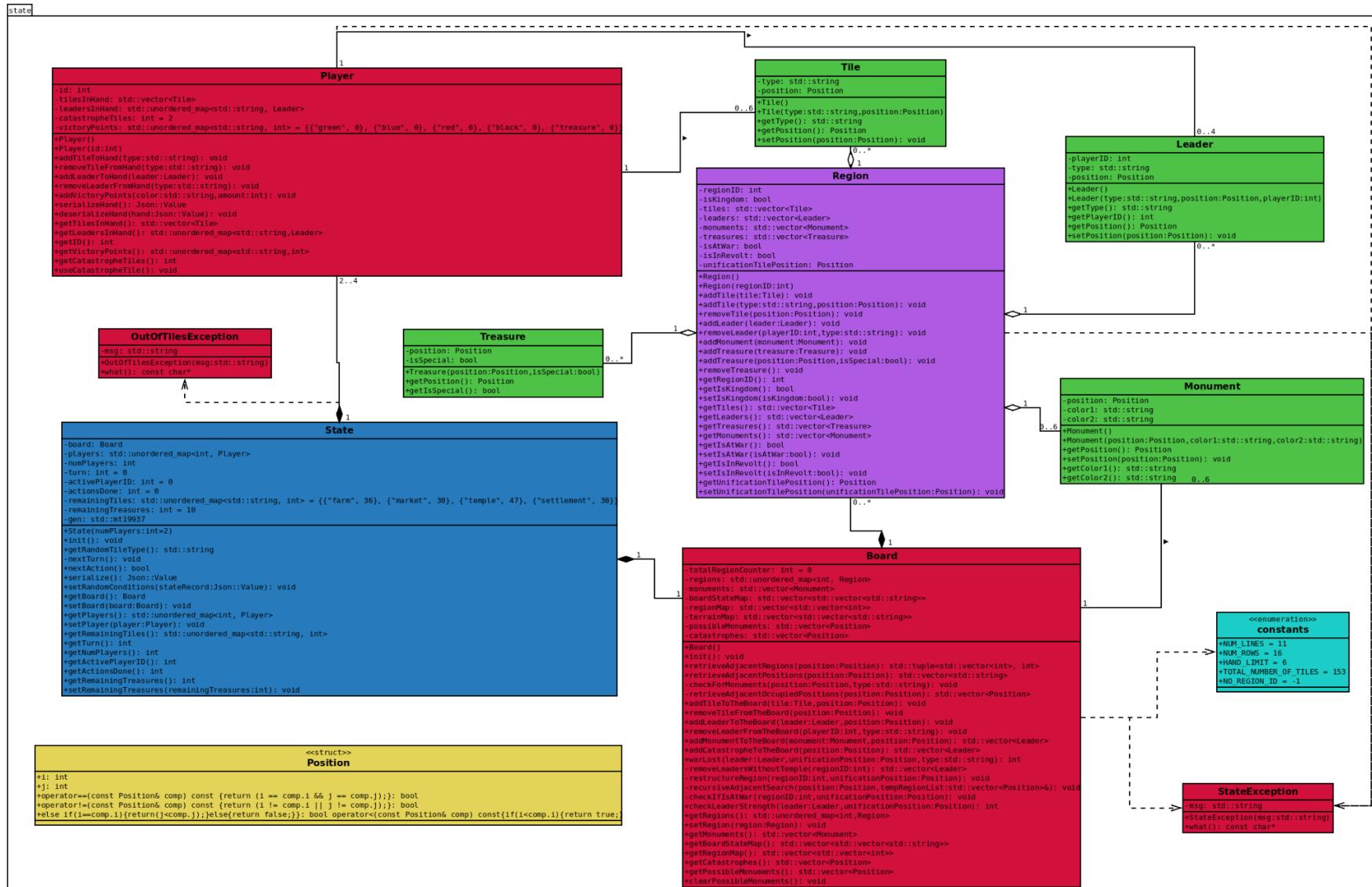


Figure 6. Diagramme des classes d'état.

## 3 Rendu: Stratégie et Conception

### 3.1 Stratégie de rendu d'un état

Comme notre jeu est au tour par tour, nous avons une stratégie de rendu simple. Nous pouvons afficher toutes les informations dont nous avons besoin en regardant l'état du jeu. L'état du jeu n'est modifié que lorsqu'un joueur exécute une action, ce qui signifie que nous n'avons aucun problème de synchronisation.

Lorsqu'un joueur exécute une action, l'état du jeu change et on affiche le nouvel état du jeu. Certains développements peuvent se produire au cours de l'action d'un joueur (e.g. un joueur met une tuile qui initie une guerre). Tous ces développements seront détaillés dans la section moteur du jeu, mais dans tous les cas, l'état du jeu ne changera qu'après que le joueur ait pris une décision. Donc, que ce soit une des actions de tour (e.g. jouer une tuile) ou une de ces décisions de jeu (e.g. combien de supporters un joueur envoie à une guerre), l'état du jeu ne change qu'après la commande d'un joueur.

Par conséquent, nous allons afficher un état de jeu, attendre que le joueur actif entre une commande (que ce soit un joueur humain ou un I.A.) et puis afficher le nouvel état de jeu, modifié par cette commande. Ensuite, on redémarre la boucle en attendant la commande du joueur actif.

Sur la figure 7, on peut voir un exemple de rendu d'un état de jeu sur lequel tous les éléments possibles qui peuvent apparaître pendant le jeu sont affichés.



Figure 7. Exemple de rendu d'état du jeu.

## 3.2 Conception logiciel

Nous avons créé 4 classes pour effectuer le rendu du jeu:

- ❖ **Classe Scene:** C'est la classe de rendu principal. Elle est responsable du rendu de tous les éléments qui seront affichés pour un écran d'un joueur. La méthode principale de cette classe est la *display()*, qui est responsable de rendre l'état du jeu passé en argument. Nous avons également créé la méthode *displayDemo()* afin de tester l'affichage des différents éléments. Lorsque l'utilisateur clique sur l'écran avec le bouton gauche de la souris, la méthode affiche un état différent de jeu. Pour mieux organiser et structurer notre code, nous avons créé les 3 autres classes auxiliaires décrites ci-dessous.
- ❖ **Classe Draw:** Il s'agit d'une classe abstraite à partir de laquelle les 2 autres classes auxiliaires seront héritées. Nous choisissons de créer cette classe parce que les 2 autres classes partagent certains attributs et méthodes.
- ❖ **Classe PlayerDraw:** Cette classe est responsable du rendu de tous les éléments qui sont exclusifs pour un joueur. Cela inclut les tuiles et les leaders que le joueur a en main, les catastrophes qu'il a de disponibles et ses points de victoire. Par conséquent, cette classe n'a besoin que d'un objet *state::Player* pour effectuer sa tâche car cette classe ne nécessite pas d'autres informations sur l'état du jeu. Le contenu d'un objet *state::Player* n'est connu que par le joueur auquel il est associé, même si les autres joueurs peuvent déduire quels leaders il a en main et combien de catastrophes il lui reste.
- ❖ **Class GameDraw:** Cette classe est responsable du rendu de tout ce qui est commun à tous les joueurs. Cela inclut tous les éléments présents dans le plateau (tuiles, leaders, trésors, monuments, catastrophes et tuiles d'unification (pendant une guerre)), monuments non encore construits et informations générales de jeu (tour actuel, joueur actif, etc). Cette classe est également chargée d'afficher l'image du plateau.

Lorsqu'une *Scene* est construite, elle construit un objet de *GameDraw* et un objet de *PlayerDraw* et les enregistre comme attributs de classe. Nous rappelons à nouveau que cette division de classe a été faite avec des fins d'organisation et de lisibilité de code. Le diagramme des classes pour le rendu est présenté en Figure 8.

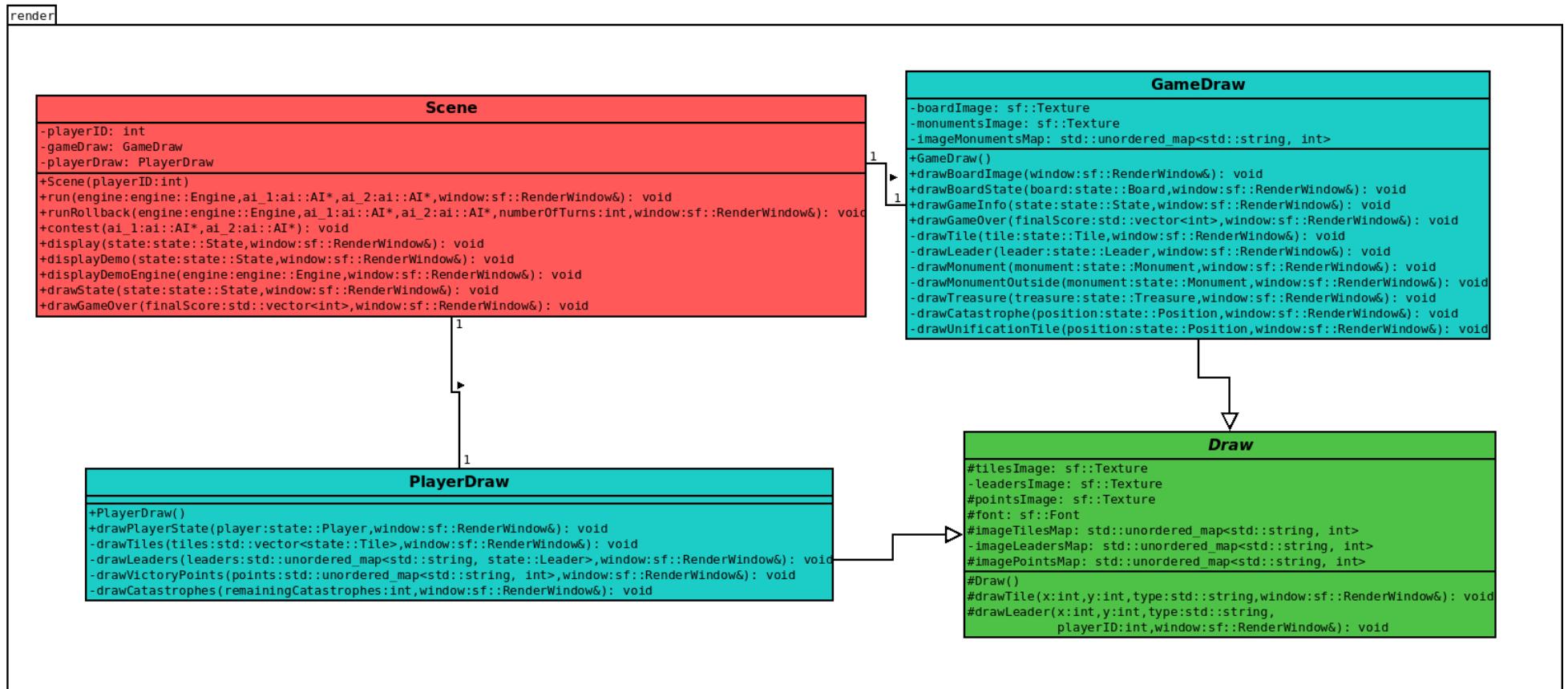


Figure 8. Diagramme des classes de rendu.

## 4 Moteur de jeu

### 4.1 Règles de changement d'états

Étant un jeu tour par tour, tout changement dans l'état du jeu est toujours déclenché par une action de l'un des joueurs. Comme décrit précédemment, à son tour, un joueur peut jouer une tuile, jouer ou déplacer un leader, jouer une tuile catastrophe ou changer certaines ou toutes les tuiles dans sa main pour de nouvelles. Au-delà de ces actions principales qu'un joueur peut choisir de faire à son tour, il y a aussi d'autres actions qui peuvent avoir un impact sur l'état du jeu.

Quand un joueur met la 4ème tuile adjacente, formant un 2 par 2 carré de tuiles du même type, il peut choisir de construire ou non un monument. Quand un joueur ajoute un leader dans une région où un autre leader du même type est déjà présent (déclenchant une révolte) ou ajoute une tuile unissant 2 régions sur lesquelles 2 leaders du même type seraient coexistants (déclenchant une guerre), le joueur doit choisir combien de tuiles il veut écarter de sa main pour augmenter sa force sur le conflit. De la même manière, le joueur attaqué peut également choisir le nombre de tuiles à mettre en jeu.

### 4.2 Conception logiciel

Compte tenu des 8 actions possibles, décrites ci-dessus, qu'un joueur pourrait effectuer pour changer l'état du jeu, nous créons les classes suivantes pour construire notre moteur de jeu:

- ❖ **Classe Engine:** C'est la classe principale qui fera tourner le jeu. Cette classe recevra une action, l'exécutera et appliquera les conséquences possibles de cette action. Cette classe gardera également une trace de tout ce qui s'est passé dans un match. Cette classe sera également responsable d'identifier si l'une des conditions de fin de match a été déclenchée et de définir le gagnant.
- ❖ **Classe Action:** C'est une classe abstraite, à partir de laquelle toutes les actions possibles seront héritées.
- ❖ **Classe PlayTile:** Cette action sera créée quand un joueur veut jouer une tuile sur le plateau.
- ❖ **Classe PlayLeader:** Cette action sera créée quand un joueur veut jouer un leader de sa main dans le tableau.

- ❖ **Classe PlayMoveLeader:** Cette action sera créée quand un joueur veut déplacer un leader qui est déjà sur le tableau à une position différente ou de retour à sa main.
- ❖ **Classe PlayCatastrophe:** Cette action sera créée lorsqu'un joueur veut jouer une tuile catastrophe.
- ❖ **Classe PlayDrawTiles:** Cette action sera créée quand un joueur veut défausser un certain nombre de tuiles de sa main pour en dessiner de nouvelles.
- ❖ **Classe PlayBuildMonument:** Cette action sera créée lorsqu'un joueur a la possibilité de construire un monument. Cette action est utilisée pour que le joueur puisse décider s'il veut construire le monument ou non et s'il le veut, quel monument il souhaite construire.
- ❖ **Classe PlayAttack:** Cette action sera créée juste après qu'un joueur déclenche un conflit. Elle sera utilisée pour définir combien de supporters il veut utiliser.
- ❖ **Classe PlayDefense:** Cette action sera créée pour répondre à une action PlayAttack. Elle sera utilisée pour définir combien de supporters le défenseur veut utiliser et pour résoudre le conflit.

Le rôle de l'IA sera de choisir l'une de ces actions en fonction de l'état du jeu au moment donné. Ils pourront alors créer l'action et la passer au moteur de jeu. Le diagramme de classe pour le moteur de jeu est montré sur la figure 9.

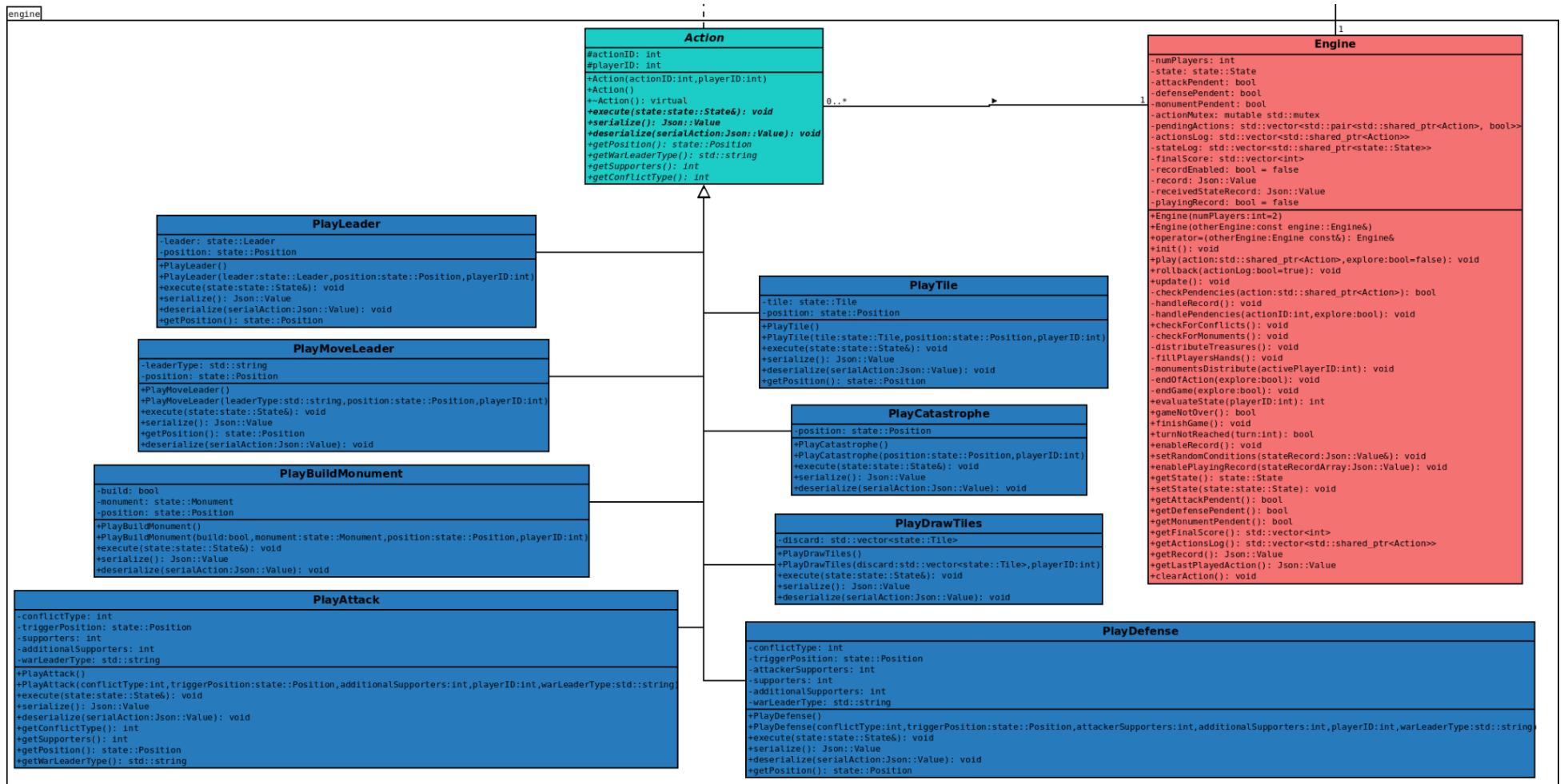


Figure 9. Diagramme des classes de moteur de jeu.

## 5 Intelligence Artificielle

### 5.1 Stratégies

#### 5.1.1 Intelligence aléatoire

A son tour, le joueur actif peut effectuer l'une des 4 actions principales possibles (jouer une tuile, jouer/déplacer un leader, jouer une catastrophe ou piocher des tuiles). Pour mettre en œuvre cette stratégie, nous avons simplement attribué une probabilité à chaque action. Chaque fois que l'IA joue, elle choisira l'une des 4 actions au hasard, selon la probabilité associée.

Pour jouer une tuile, un leader ou une catastrophe, nous balayons le tableau pour vérifier toutes les positions disponibles pour l'action donnée et on en prend une aléatoirement. Nous choisissons également quelle tuile / leader à jouer, parmi ceux disponibles pour le joueur, au hasard. Pour l'action de piocher des tuiles, nous sélectionnons d'abord au hasard combien de tuiles seront changées et ensuite lesquelles.

Les 3 autres actions (construire un monument, jouer une attaque ou jouer une défense) doivent être jouées sur demande. Lorsque l'une de ces actions est demandée, nous choisissons le nombre de supporters (pour l'attaque/défense) ou le monument à construire au hasard.

#### 5.1.2 Intelligence basée sur des heuristiques

Pour mettre en œuvre une stratégie qui fonctionnera mieux que la stratégie aléatoire, nous mettons en œuvre une intelligence basée sur l'heuristique, un ensemble de règles qui orienteront les choix de l'IA. Afin d'obtenir un score élevé, un joueur doit marquer dans tous les 4 domaines, ce qui signifie avoir des points distribués dans les 4 couleurs.

Pour y parvenir, nous commençons par sélectionner la couleur où le joueur a le moins de points. Si le leader de cette couleur est sur la main du joueur, l'IA doit alors jouer ce leader. La position du leader repose sur la force qu'il aura dans cette position. Par conséquent, l'IA jouera le leader dans une région sur laquelle il y a la plus grande quantité de tuiles de la couleur du leader, mais en évitant les conflits avec d'autres leaders de la même couleur. Dans le cas de **l'agriculteur**, il sera joué à côté d'une rivière, car les **fermes** ne peuvent être jouées que sur les rivières. Si plusieurs régions présentent les mêmes conditions, l'une d'elles est choisie au hasard.

Si le leader est déjà sur le plateau, si l'IA peut, elle jouera une tuile de la même couleur sur la région sur laquelle le leader est. S'il y a plus d'un poste possible dans la région de leader, l'un d'eux sera choisi au hasard.

Dans le cas où le joueur n'a pas une tuile de la couleur correspondante dans sa main pour jouer, tout le processus de décision est répété pour la deuxième couleur où le joueur possède le moins de points. Si ce n'est pas possible de jouer un leader ou tuile de cette couleur non plus, l'IA va alors piocher une nouvelle main.

Comme les catastrophes sont vraiment spécifiques (un joueur ne peut jouer que 2 fois dans un match) et n'ont pas de contexte clair sur lequel elles devraient être jouées, cette IA ne les jouera jamais.

Pour l'action de construction de monument et d'attaque, la même stratégie a été gardée de l'IA aléatoire. Le monument sera toujours construit si possible et l'attaque ajoutera le nombre maximum de supporters disponibles. Par contre, pour la défense, l'IA vérifiera le nombre de supporters dont elle a besoin pour gagner le conflit. Si le joueur a ce nombre de tuiles à défausser, il le fera, gagnant ainsi le conflit. Si la victoire n'est pas possible, l'IA ne se défaussera pas d'une seule tuile. Pour valider notre IA, nous avons joué 100 parties entre une IA heuristique contre une IA aléatoire. L'heuristique a remporté toutes les parties.

### 5.1.3 Intelligence avancée

Pour implémenter notre intelligence avancée, nous avons commencé par implémenter une version adaptée de minimax. Le premier changement que nous avons mis en œuvre était dû au fait qu'un joueur joue 2 fois à son tour. Cela signifie que l'algorithme va maximiser/minimiser 2 fois de suite (max, max, min, min, max ...). Afin d'optimiser l'algorithme, nous avons implémenté la version de minimax avec coupure alpha/beta.

Afin d'effectuer l'exploration de l'arbre, nous avons également implémenté une méthode "rollback" dans le moteur de jeu, nous permettant d'annuler la dernière action, en revenant à l'état de jeu précédent. Cela nous permet de limiter la mémoire nécessaire pour exécuter l'algorithme.

La première limite que nous avons rencontrée pour cette stratégie est le long temps d'exécution pour trouver chaque action. Ce problème découle de la complexité du jeu. Au début du jeu, il y a environ 700 actions différentes possibles pour chaque joueur à chaque tour. Compte tenu de la complexité de l'algorithme ( $O(b^{d/2})$ ) même pour explorer dans une petite profondeur, le facteur de ramifications est beaucoup trop grand.

Nous avons essayé de mettre en œuvre seulement des actions qui "font sens" (les tuiles ne seront jouées que dans les régions, etc), en fonction de la façon dont les joueurs jouent habituellement. Nous pourrions réduire le nombre d'actions possibles à environ 200. Même si, le facteur de ramifications encore trop grand pour être viable.

Par conséquent, nous avons dû trouver une autre solution pour mettre en œuvre notre intelligence avancée. Notre solution était de mélanger l'intelligence heuristique avec l'algorithme minimax.

L'intelligence artificielle fonctionnera comme heuristique, mais en utilisant minimax pour choisir parmi toutes les positions possibles dans la région du leader lors du jeu d'une tuile. Cette stratégie est limitée, en ce sens qu'elle ne regarde pas toutes les actions possibles, mais elle augmentera la performance par rapport à l'intelligence heuristique, considérant que plusieurs tours à venir permettront à l'IA de choisir une position pour la tuile qui favorise la construction de monuments, l'acquisition de trésors et s'engager dans des guerres sur lesquelles il sortira victorieux.

Pour valider le fonctionnement de notre IA, nous avons effectué 100 parties entre une IA avancée et une IA heuristique. L'IA avancée a remporté 57 fois. Même si la performance de notre IA avancée n'est pas vraiment supérieure à l'heuristique, nous étions satisfaits du résultat, compte tenu de la complexité du jeu.

Pour comparaison, le jeu de Go a un facteur de ramifications moyen de 250. Il est bien connu que pour implémenter une IA performante pour le jeu de Go (comme AlphaGo) il est nécessaire d'utiliser des techniques d'apprentissage profond. Par conséquent, nous croyons que pour obtenir une IA vraiment performante sur notre projet, nous devrions utiliser des stratégies similaires (e.g. former un réseau neuronal pour évaluer un état de jeu et utiliser cette évaluation pour mettre en œuvre minimax).

## 5.2 Conception logiciel

Nous avons créé les classes suivantes pour implémenter l'intelligence artificielle:

- ❖ **Classe AI:** C'est une classe abstraite dont toutes les classes d'intelligence artificielle seront héritées. Chaque AI sera associée à un joueur, par un player ID.
- ❖ **Classe RandomAI:** Implémente la stratégie aléatoire, décrite dans la session 5.1.1. Les actions sont prises au hasard, avec des probabilités différentes selon le type d'action.
- ❖ **Classe HeuristicAI:** Implémente la stratégie basée sur des heuristiques, décrite dans la session 5.1.2. Les leaders et les tuiles sont placés sur le plateau selon la couleur sur laquelle le joueur a le moins de points, obtenant un score final équilibré.
- ❖ **Classe DeepAI:** Implémente la stratégie avancée, décrite dans la session 5.1.3. Combinez la stratégie heuristique avec l'algorithme minimax (en utilisant la coupure alpha/beta) pour placer les tuiles dans des positions plus avantageuses.

Le diagramme de classes pour les intelligences artificielles est montré sur la figure 10.

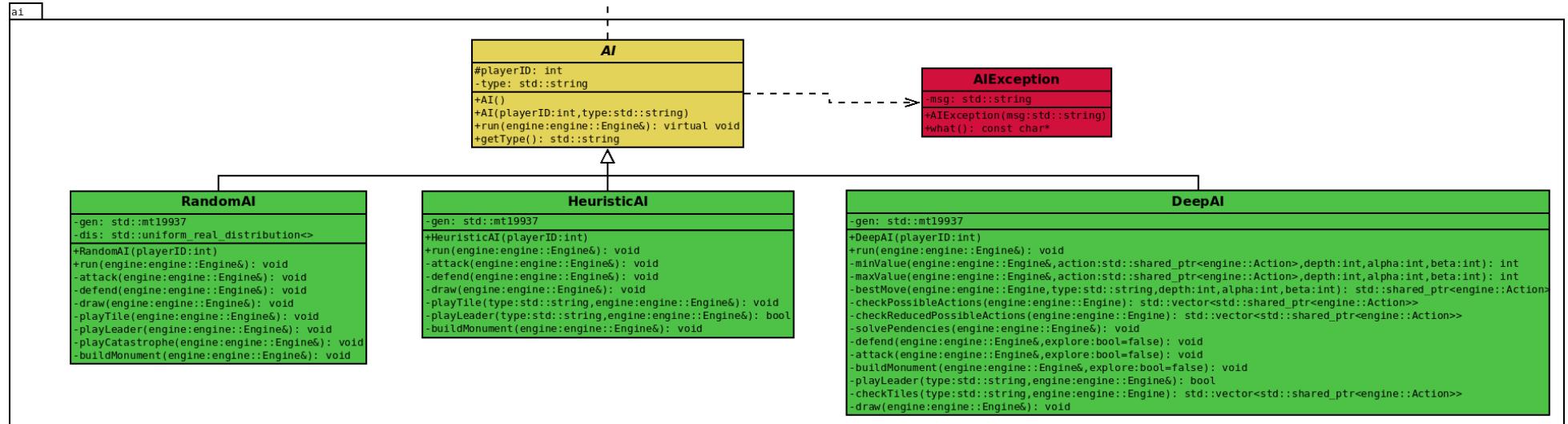


Figure 10. Diagramme des classes d'intelligence artificielle.

## 6 Modularisation

### 6.1 Organisation des modules

#### 6.1.1 Répartition sur différents threads

Nous avons commencé la modularisation en implémentant le multithreading. Pour ce faire, nous avons séparé le moteur de jeu du rendu. Le rendu continue à tourner sur le thread principal, tandis que nous créons un thread pour faire tourner le moteur de jeu.

La thread du moteur de jeu se constitue d'une boucle *while* jusqu'à la fin du jeu, sur lequel le moteur vérifie s'il y a une action à jouer dans la liste des actions en attente. Si c'est le cas, le moteur exécute l'action, sinon, le moteur vérifie à nouveau la liste.

Les actions sont mises sur la liste en attente soit par une IA jouant localement, soit en lisant le fichier JSON de reprise ou soit en lisant un JSON reçu du serveur. Cela signifie que les commandes sont utilisées par le moteur de la même manière, indépendamment de son origine, ce qui facilite la mise en œuvre de la reprise et du réseau.

#### 6.1.2 Répartition sur différentes machines: rassemblement des joueurs

Pour rassembler les joueurs sur le serveur, nous avons implémenté un lobby de jeu utilisant les services CRUD via une API web REST. Nous avons commencé par mettre en œuvre le service de version. Ce service implémente uniquement la requête GET, qui retourne la version de l'API.

Ensuite, nous avons mis en œuvre le service de joueur, qui permet aux clients de se connecter au serveur. La requête GET permet de récupérer le nom des joueurs connectés. La requête PUT nous permet d'ajouter des clients à la liste des joueurs du serveur. La requête POST peut être utilisée pour renommer un joueur connecté. La requête DELETE nous permet de supprimer les joueurs de la liste des joueurs connectés.

Le tableau 1 présente les entrées, les sorties et le statut de chaque demande dans différents scénarios pour ces deux services.

Tableau 1. Web REST API - Services de joueur et de version.

Requête	Données En Entrée	Données En Sortie	Status
GET /version	-	type: "object", properties: { "version":string }	Ok
GET /player/<id> (Cas joueur <id> existe)	-	type: "object", properties: { "name":string }	Ok
GET /player/<id> (Cas <id> négatif)	-	type: "array", items: { type: "object", properties: { "name":string } }	Ok
GET /player/<id> (Cas joueur <id> n'existe pas)	-	-	Not_Found
PUT /player (Cas il reste une place libre)	type: "object", properties: { "name":string }	type: "object", properties: { "id":int [0, 1] }	Created
PUT /player (Cas plus de place libre)	type: "object", properties: { "name":string }	-	Out_Of_Resources
POST /player/<id> (Cas joueur <id> existe)	type: "object", properties: { "name":string }	-	No_Content
POST /player/<id> (Cas joueur <id> n'existe pas)	type: "object", properties: { "name":string }	-	Not_Found
DELETE /player/<id> (Cas joueur <id> existe)	-	-	No_Content
DELETE /player/<id> (Cas joueur <id> n'existe pas)	-	-	Not_Found

### 6.1.3 Répartition sur différentes machines: échange des commandes

Pour que le jeu fonctionne en réseau, nous avons mis en œuvre deux autres services. D'abord, nous avons mis en place un service de jeu, qui ne répond qu'à la requête GET, en lui renvoyant les informations si le jeu a commencé ou si le serveur attend que les autres joueurs se connectent.

Ensuite, nous avons mis en place un service d'action, qui permet l'échange d'actions entre le serveur et le client. La requête GET permet au client de récupérer une action d'une époque spécifique, tandis que la requête PUT permet au client d'envoyer des actions au serveur. Une "époque" contient une action unique réalisée par l'un des joueurs.

La routine serveur-client se déroule comme suit. Les joueurs se connectent au serveur, recevant du serveur un ID de joueur, qu'ils utiliseront comme l'ID de joueur dans l'état du jeu. Ils reçoivent également du serveur les conditions initiales du jeu. Puisque le jeu a des conditions initiales aléatoires, le serveur doit générer ces conditions et les envoyer à tous les clients. De cette façon, tous les joueurs auront le même état de jeu pour commencer le jeu. Remarquez que nous ne sérialisons que les conditions qui sont aléatoires (tuiles initiales du joueur en main et les tuiles restantes sur le sac), pas tout l'état du jeu.

Après la connexion, un client vérifiera si le jeu a commencé en utilisant le service de jeu. Si le jeu n'a pas encore commencé, le client attendra et vérifiera à nouveau, jusqu'à ce que le jeu commence.

Une fois le jeu commencé, le client va vérifier s'il est le joueur actif. Si oui, il va générer une action en utilisant son IA, sérialiser cette action et l'envoyer au serveur. Notez qu'il ne va pas exécuter l'action. Ensuite, il demandera au serveur l'action de la première époque.

Une fois que le serveur reçoit une commande de joueur, il la déserialise et l'ajoute à la liste des actions en attente, qui sera exécutée une fois que le serveur met à jour le moteur de jeu. Après que l'action soit exécutée par le serveur, le serveur peut répondre à une requête GET pour cette "époque".

Le client demandera l'action de son époque actuelle en boucle, jusqu'à ce qu'il reçoive une réponse. Une fois qu'il recevra l'action, il la déserialise, l'exécute et augmente son compteur d'époque. Ensuite, le client retourne vérifier s'il est le joueur actif pour décider s'il doit envoyer une commande ou non.

Avec cette approche, nous avons pu synchroniser le jeu entre différentes machines simplement en gardant un compte du nombre d'actions exécutées au total. Nous remarquons que le serveur envoie toujours les conditions aléatoires du jeu après que l'action a été exécutée avec l'action sérialisée. De cette façon, les clients exécutent la commande reçue et ensuite ils définissent les conditions aléatoires pour correspondre à celles reçues par le serveur.

Le tableau 2 détaille les différentes requêtes mises en œuvre pour les services de jeu et d'action.

Tableau 2. Web REST API - Services de jeu et d'action.

Requête	Données En Entrée	Données En Sortie	Status
GET /game	-	<pre>type: "object", properties: {   "status":int [1,2] }</pre>	Ok
GET /action/<id>  (Cas où l'action de l'époque <id> était déjà exécutée par le serveur)	-	Action sérialisée et conditions aléatoires après exécution de l'action.	Ok
GET /action/<id>  (Cas où l'action de l'époque <id> n'a pas été exécutée par le serveur)	-	-	Not_Found
PUT /action	Action sérialisée	<pre>type: "object", properties: {   "epoch":int }</pre>	Created

## 6.2 Conception logiciel

Cette partie est divisée entre serveur et client. Côté client, nous avons implémenté les classes suivantes:

- ❖ **Classe Client:** Cette classe est responsable de l'exécution du jeu localement. Il contient un moteur de jeu et deux IA. Il permet également d'exécuter un match à partir d'un fichier JSON, implémentant la fonctionnalité de reprise. Il exécute le jeu en utilisant le multithreading, comme décrit dans la section 6.1.1.
- ❖ **Classe NetworkClient:** Cette classe permet d'exécuter le jeu sur le réseau. Elle implémente la partie client du processus décrit dans la section 6.1.3. Elle contient une seule IA qui générera les actions qui seront envoyées au serveur. La classe contient également un moteur de jeu, pour exécuter les commandes reçues par le serveur.

Le diagramme de classes pour le client est montré sur la figure 11.

Côté serveur, nous avons implémenté les classes suivantes:

- ❖ **Classe Game:** Cette classe exécute le jeu côté serveur. Elle contient une liste de joueurs connectés et un moteur de jeu pour exécuter les commandes reçues. Cette classe implémente également toutes les méthodes utilisées par les services pour apporter des modifications et récupérer des informations sur le jeu.
- ❖ **Classe ServiceManager:** Cette classe est utilisée pour gérer tous les services, servant d'intermédiaire entre la bibliothèque *microhttpd* et chaque service.
- ❖ **Classe Service:** C'est une classe abstraite à partir de laquelle tous les services sont hérités.
- ❖ **Classe VersionService:** Implémente le service de version comme décrit dans la section 6.1.2. Il permet de récupérer la version actuelle de l'API.
- ❖ **Classe GameService:** Implémente le service de jeu comme décrit dans la section 6.1.3. Il permet de récupérer le statut du serveur (attente des joueurs ou course).

- ❖ **Classe PlayerService:** Implémente le service de joueur comme décrit dans la section 6.1.2. Il permet d'ajouter et de supprimer des joueurs du serveur, permettant aux clients de rejoindre le match. Il permet également de renommer les joueurs et de récupérer tous les noms de joueurs connectés.
- ❖ **Classe ActionService:** Implémente le service d'action comme décrit dans la section 6.1.3. Il permet d'échanger des actions entre les clients et le serveur.

Le diagramme de classes pour le serveur est montré sur la figure 12.

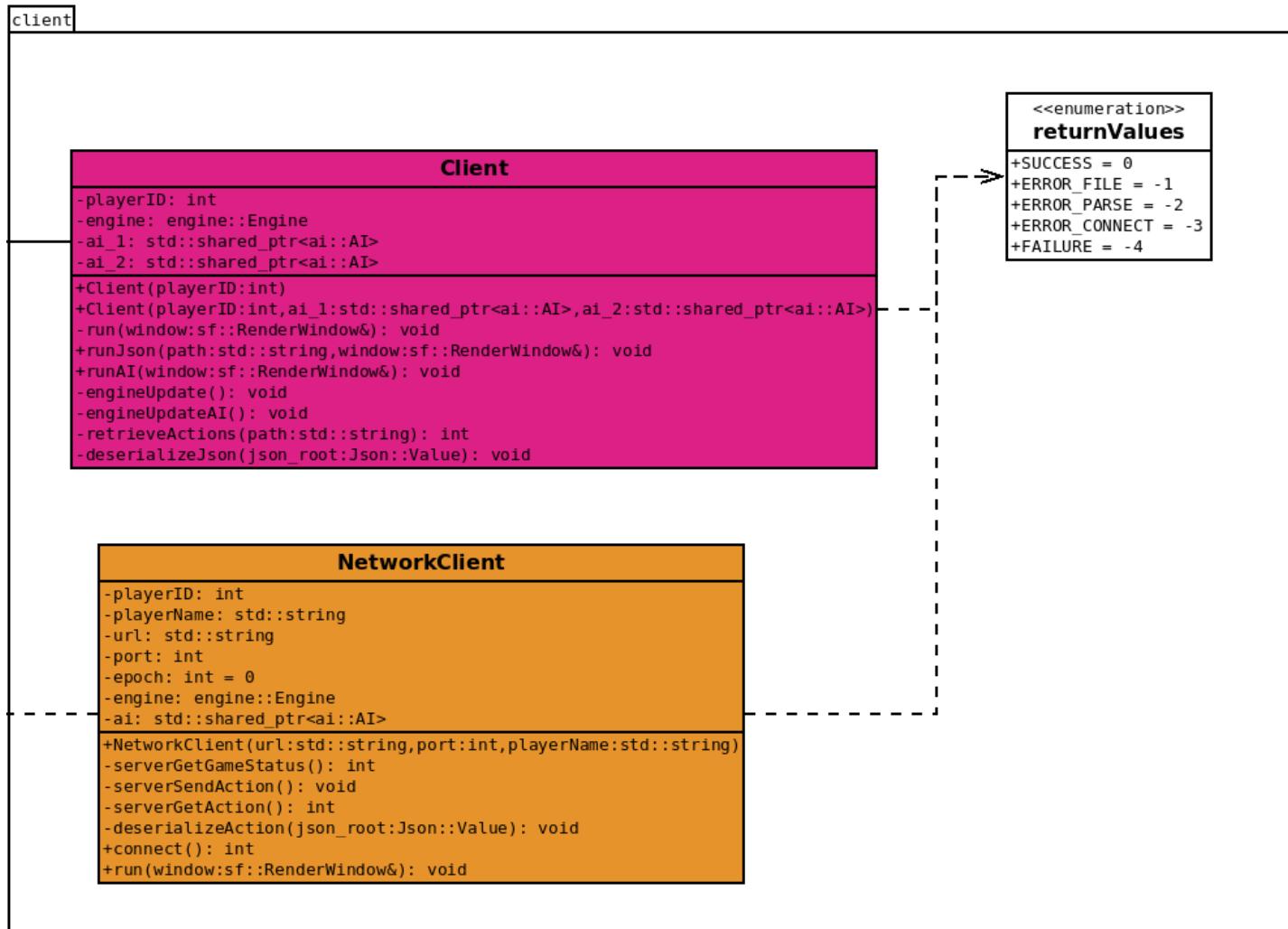


Figure 11. Diagramme des classes de client.

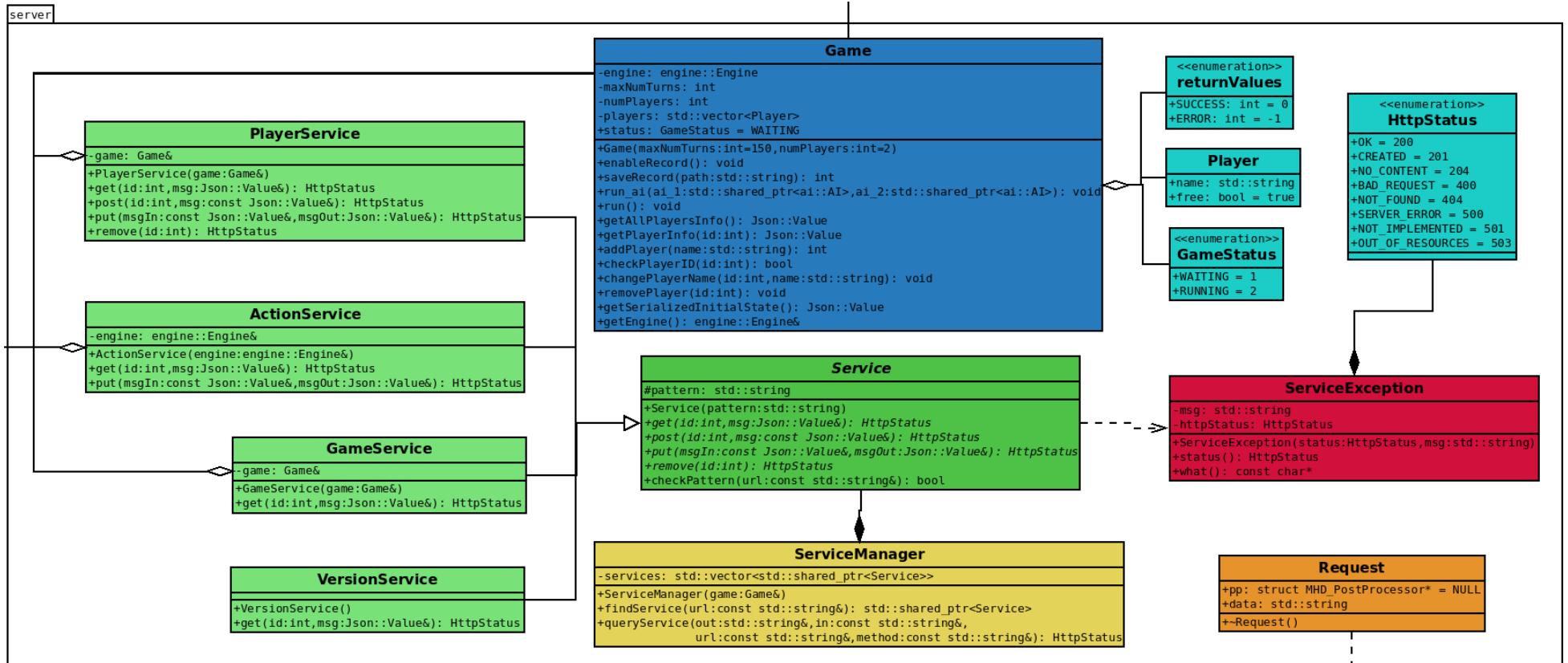


Figure 12. Diagramme des classes de serveur.