Projet Logiciel Transversal

MagiX

Maxime Marroufin, Quentin Chhean,

Abinaya Mathibala, Alban Benmouffek

Table des matières

1 Objectif

1.1 Présentation générale

Ce projet s'inscrit dans le cadre du programme de 3ème année de l'option « Informatique et Systèmes » de l'ENSEA. Il a pour but de concevoir un jeu de carte virtuel. Vous trouverez toute la documentation sur le lien suivant : https://github.com/kangulu/plt

1.2 Règles du jeu

Le plateau généré dispose de trois rivières ordonnées en trois niveaux, communs à tous les joueurs. Chaque niveau de rivière est composé de cinq cartes fixes. Chaque carte est caractérisée par un gain et un coût. La monnaie du jeu est représentée en quatre types de ressources : eau, bois, roche et points de victoire.
Au premier tour les joueurs n'ont accès qu'à la première rivière. Les cartes situées sur cette dernière ne requièrent pas de coût pour jouer. Chaque joueur peut récupérer une carte puis passer son tour. Chaque carte récupérée est remplacée par une nouvelle carte sur le plateau.

Aux tours suivants, les joueurs ont accès aux deux autres rivières. Les joueurs achètent un objet et obtiennent un gain (points de victoire ou ressources) en échange d'un coût précisé sur la carte. Ce gain peut être soit un gain unique, soit un revenu qui donne au joueur un certain nombre de ressources à chaque tour.

Voici, ci-dessous, un récapitulatif des cartes disponibles sur chaque niveau de rivières :

- Rivière 1 : Roche (Stone), Bois (Wood), Eau (Water), Points de victoire (VictoryPoint) ;
- Rivière 2 : Puits (Well), Mine, Fontaine (Fountain), Maison (House) ;
- Rivière 3 : Château (Castle), Bibliothèque (Library), Piscine (Pool), Fontaine (Fountain), Aqueduc (Aqueduct).

Le but du jeu est d'avoir le plus de points de victoire à la fin de la partie.

1.3 Réévaluation du cahier des charges

Suite aux difficultés d'avancement de la première version du jeu, nous avons réévaluer le cahier des charges et avons effectués plusieurs changement vis-à-vis des règles du jeu.

Ces changements ont permis une simplification de l'architecture du jeu afin de pouvoir avancer plus vite dessus.

Règles	**Version 1**	**Version 2**
Nombre de joueurs	2	2
Condition de victoire	- L'adversaire n'a plus de points de vie - L'adversaire n'a plus de carte à piocher - Effet de carte	- Avoir plus de points de victoire que l'adversaire
Caractéristiques du joueur	- Cartes dans sa main - Cartes de son côté du Battlefield - Cartes dans sa Library - Cartes dans son Graveyard - Mana	- Ressources (Water, Wood, Stone, Victory Points) - Revenu

Terrain de jeu	- Points de vie - Main des 2 joueurs - Libraries des 2 joueurs - Graveyard des 2 joueurs - Exile - Battlefield	- 3 rivières
Déroulement d’un tour	- Phase de début - Phase de pré-combat - Phase de combat - Phase de post-combat - Phase de fin	- Phase de choix de carte - Phase de gain de ressources
Actions possibles	- Piocher - Lancer un sort - Attaquer - Défendre	- Choisir une carte

1.4 Conception Logiciel

Présenter ici les packages de votre solution, ainsi que leurs dépendances.

Pour concevoir le jeu, nous allons créer plusieurs packages qui correspondent à plusieurs aspects du jeu.

- Le State, qui décrira à l’instant donné l’état du jeu
- Le Render, qui s’occupera de dessiner et d’afficher le jeu dans une fenêtre
- L’Engine, qui permet de passer d’un état du jeu à un autre et de faire avancer la partie
- L’IA qui jouera contre nous.

2 Description et conception des états

2.1 Description des états

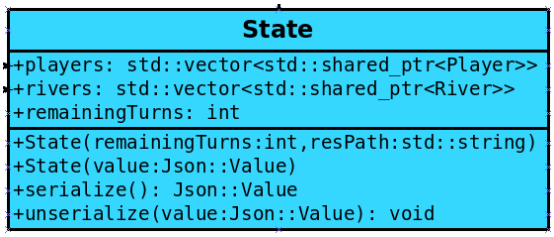
Un état du jeu est composé de plusieurs éléments :

- Un State, qui est composé de Players (2) et de Rivers (3)
- Un Player qui possède des Ressouces
- Une River qui est composée de Cards

Un état donné du jeu est donc décrit de la manière suivante : les cartes disponibles dans les différentes River, les états des différents Players (leurs ressources et leur income) et le nombre de tours restant.

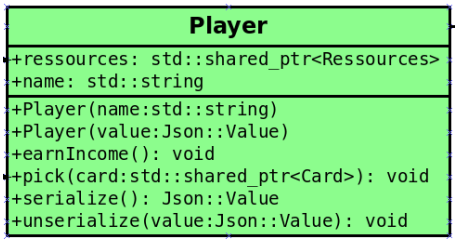
2.2 Conception logiciel

La description de chaque classe et de leurs dépendances :



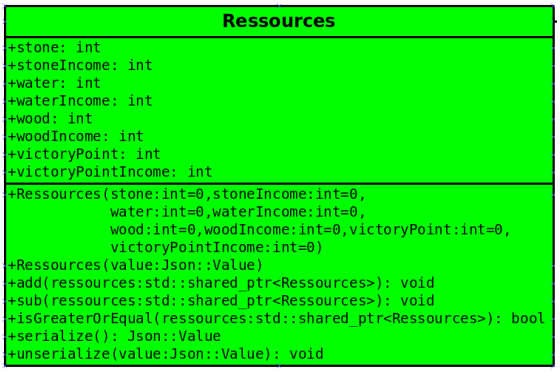
La classe State possède :

- Un attribut players qui est un vecteur de pointeurs partagés de Player, correspondant aux joueurs dans la partie
- Un attribut rivers qui est un vecteur de pointeurs partagés de River, correspondant aux rivières présentes dans le jeu
- Un attribut remainingTurns qui est un int correspondant au nombre de tours restants
- Un constructeur ainsi que les méthodes serialize et unserialize qui permettent de sauvegarder un état du jeu dans un fichier json.



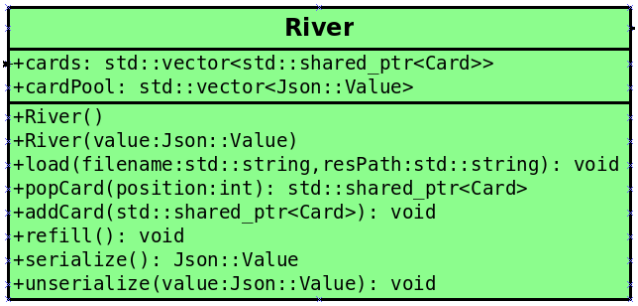
La classe Player possède :

- Un attribut ressources qui est un pointeur partagé vers une instance de la classe Ressources correspondant aux ressources actuelles du joueur en question
- Un attribut name qui correspond au nom du joueur
- Un constructeur
- Une méthode earnIncome qui permet de faire gagner au joueur les ressources qu'il devrait toucher par rapport à son revenu actuel
- Une méthode pick qui permet de choisir une carte à jouer dans les River
- Les méthodes serialize et unserialize qui sauvegardent un état du Player.



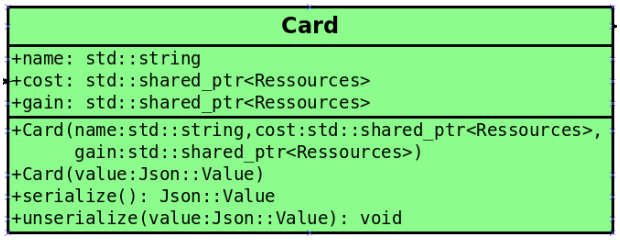
La classe Ressources possède :

- Les attributs stone, water, wood et victoryPoint correspondant aux nombre de ressources de chaque élément que le joueur possède
- Les attributs stoneIncome, waterIncome, woodIncome et victoryPointIncome correspondant au revenu de chaque élément que le joueur possède
- Un constructeur
- Les méthodes add et sub qui permettent d'ajouter et de retirer des ressources
- La méthode isGreaterOrEqual qui vérifie si le nombre de ressources d'un élément est suffisant pour acheter une carte.
- Les méthodes serialize et unserialize qui enregistent un état de Ressources.



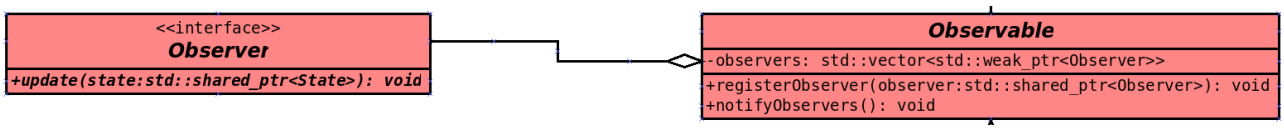
La classe River possède :

- Un attribut cards qui est un vecteur de pointeurs partagés de Cards qui correspond aux cartes présentes dans la River
- Un attribut cardPool qui est un vecteur de Json values correspondant aux cartes qui peuvent de présenter sur cette River et dans lequel nous allons piocher pour choisir une carte au hasard lorsqu'un faut la remplir.
- Un constructeur
- Une méthode load qui va charger la River de cartes.
- Les méthodes popCard et addCard qui vont ajouter et enlever une carte de la River
- La méthode refill qui va re-remplir la River jusqu'à qu'elle ait 5 cartes.
- Les méthodes serialize et unserialize qui permettent de sauvegarder un état de la River.



La classe Card possède :

- Un attribut name qui correspond au nom de la carte
- Les attributs cost et gain qui correspondent au coût de la carte et le gain qu'elle apporte
- Un constructeur
- Les méthodes serialize et unserialize qui permettent de sauvegarder un état de la Card



On utilise également l'Observable Pattern. Lorsque State subira un changement, Observable notifiera les Observers du changement.

2.3 Conception logiciel : extension pour le rendu

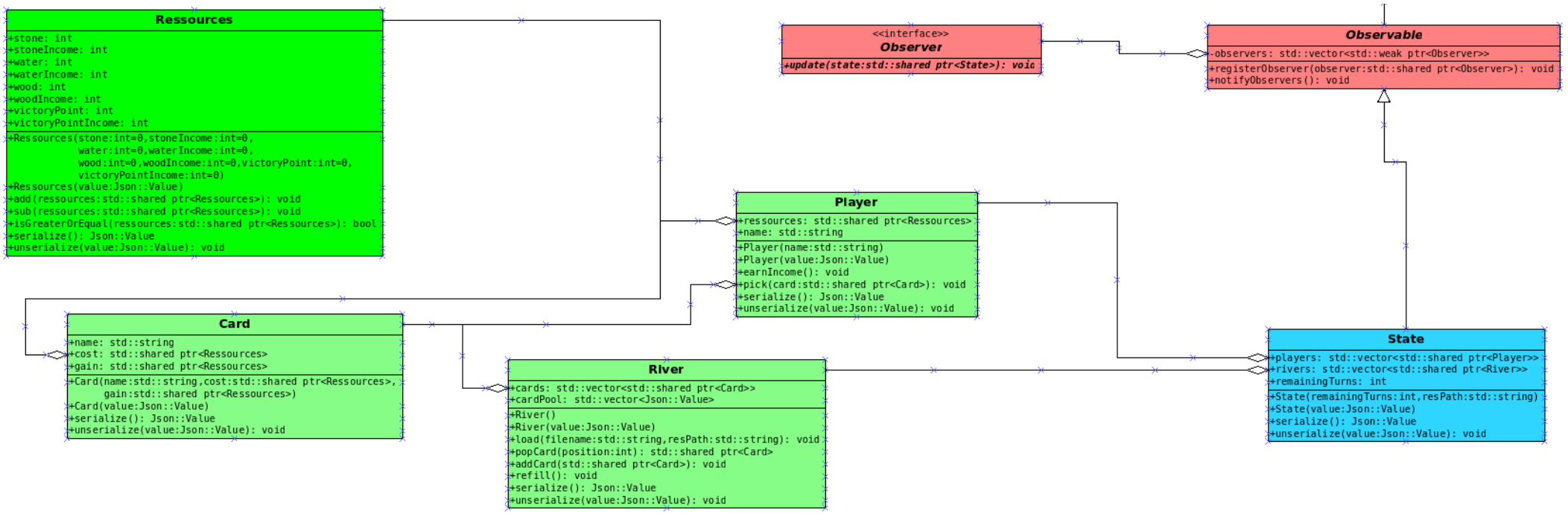
2.4 Conception logiciel : extension pour le moteur de jeu

2.5 Ressources

Les ressources utilisés pour le projets sont conservé dans le dossier « /home/ensea/Desktop/MagiX/plt/res/». Il contient trois sous-dossier :

- « cardsData » contient la descriptions des cartes et des rivières sous format JSON,
- « fonts » contient la police de caractère utilisée,
- « textures » contient les textures par exemple pour les couleurs de fond, les contours, les logos...

Illustration 1: Diagramme des classes d'état



3 Rendu : Stratégie et Conception

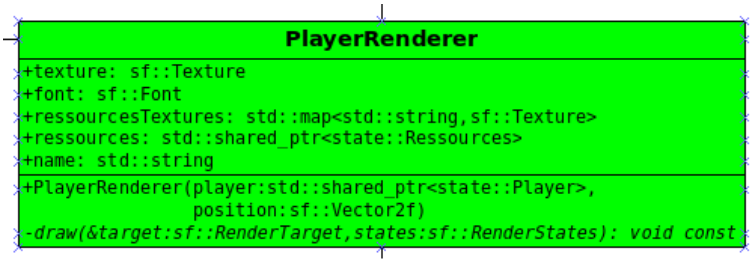
Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémente pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

3.1 Stratégie de rendu d'un état

Le render.dia gère l'aspect graphique du jeu. Dans cette partie, les informations sont récupérées du state.dia et transformées pour en tirer une fenêtre sur l'écran dans laquelle on observera l'état du jeu. Pour cela, nous utilisons la librairie SFML2.0.
Le paquetage render correspond à toutes les informations auxquelles le joueur aura accès. Dans notre cas, le joueur peut avoir accès à toutes les informations (l'état de l'adversaire et le sien). Ainsi, la fenêtre doit être composé de deux parties distinctes : les rivières avec les cartes et les informations du joueur.

3.2 Conception logiciel

Les informations des joueurs sont affichés à l'aide de la classe « render ::PlayerRender ».



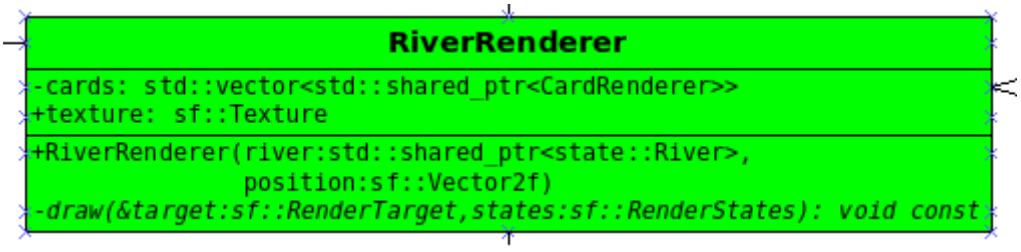
Cette classe possède comme attribut les informations à afficher pour un seul joueur. Les éléments à afficher (attributs de la classe) sont les suivants :

- texture : la texture de l'arrière-plan
- font : la police du texte
- ressourcesTextures : map contenant les logos des ressources
- ressources : issue de la classe Player du state, la quantité de ressources possédée par le joueur
- name : nom du joueur
-

Elle possède également deux méthodes :

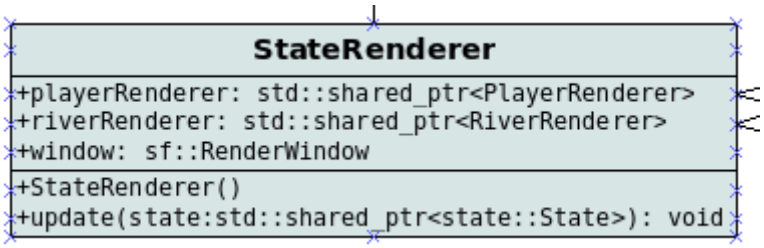
- un constructeur prenant en argument le pointeur du joueur concerné et la position dans la fenêtre,
- draw() est la fonction issue de sf ::Drawable qui permet de dessiner les éléments dans un rendu cible.

De la même façon, la classe « render ::RiverRenderer » permet de dessiner une rivière à partir des informations de la classe state ::River :



Elle est composée d’un vecteur de classes, d’une texture d’arrière-plan et des méthodes draw() et le constructeur.

La classe render::CardRenderer est lié par un lien d’agrégation à la classe render ::RiverRenderer. Elle permet d’afficher une carte composé d’une texture, d’une police, les ressources, prix, gain et nom grâce aux méthodes draw() et constructeurs, à partir des informations de la classe state ::Card.



Et enfin, la classe render::StateRenderer permet de gérer l’affichage selon l’état de jeu grâce à la méthode update(). En plus de la fenêtre, elle récupère comme attributs par liens d’agrégation de RiverRenderer et PlayerRenderer.

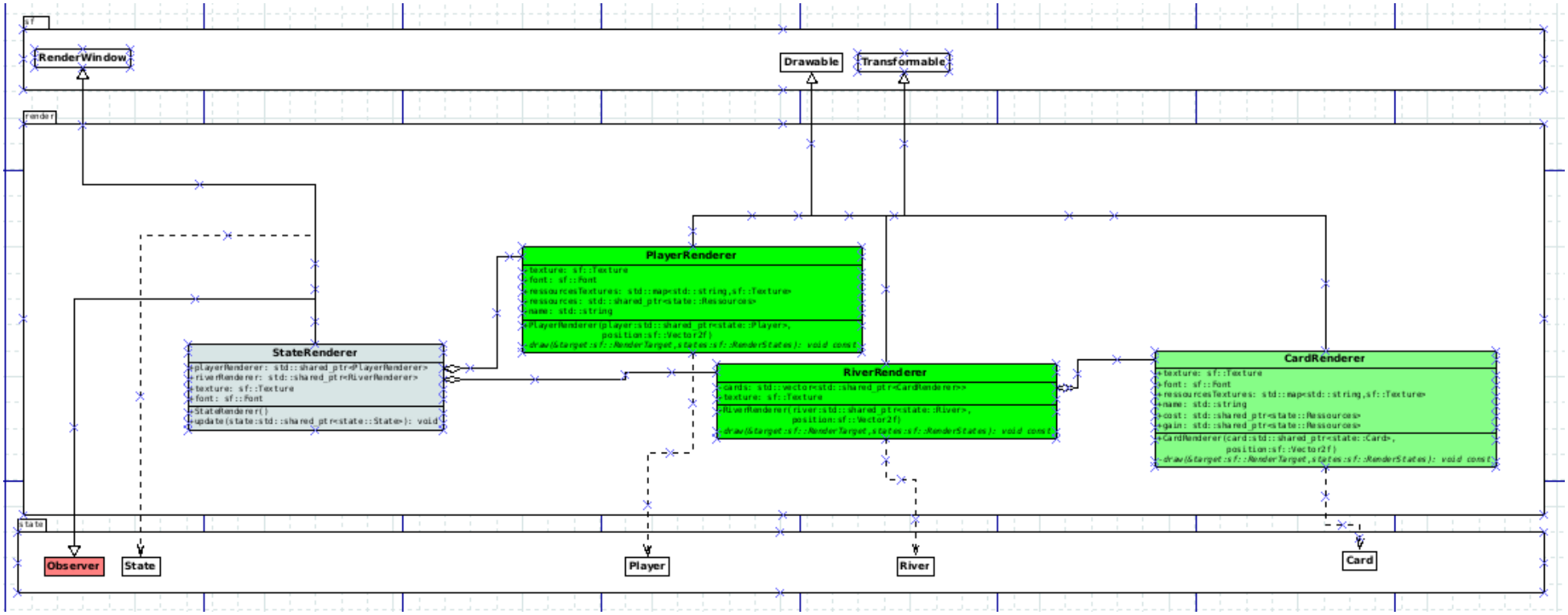
3.3 Conception logiciel : extension pour les animations

3.4 Ressources

La conception des textures et fenêtres a été faite par nous-même à l’aide du logiciel gratuit paint.NET
Pour ce qui est des images des cartes, il s’agit d’images libres de droit fournies par le site https://publicdomainvectors.org/

3.5 Exemple de rendu

Illustration 2: Diagramme de classes pour le rendu



4 Règles de changement d'états et moteur de jeu

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects lié au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

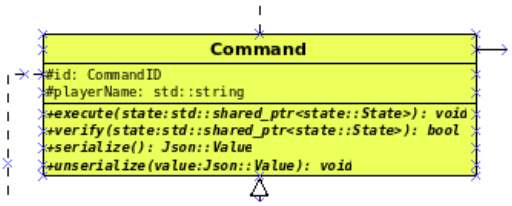
4.1 Horloge globale

L'avancement dans le temps est géré par tour de jeu.

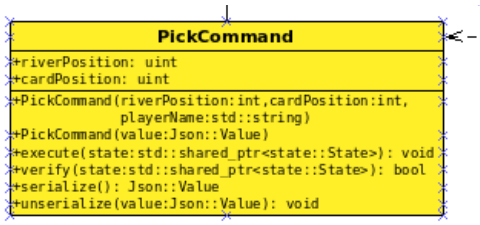
4.2 Conception logiciel

Le moteur de jeu ou engine correspond à la partie du jeu qui va récupérer les commandes entrées par le joueur et vérifier qu'elles sont autorisées avant de les exécuter.

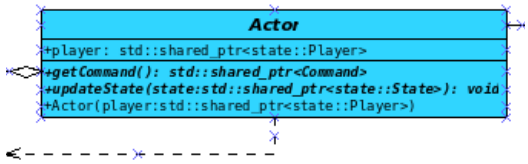
On a donc tout d'abord une classe engine ::Commande qui permet de vérifier et d'exécuter les commandes. Elle possède aussi deux méthodes serialize() et deserialize() qui permettent de conserver les données d'une commande sous format JSON. Cette classe contient deux attributs permettant d'identifier la commande joueur : « id », l'identifiant de la commande et « playerName », le joueur émetteur de la commande.



La classe engine ::Commande a une classe fille engine::PickCommand qui permet de récupérer et exécuter la commande « Récupération de la carte ».



La classe engine::Command est relié par lien d'agrégation à la classe engine::Actor. Elle permet de récupérer les commandes des joueurs et mettre à jour le state pour l'acteur selon les commande exécutées par l'engine.



Et enfin la classe engine::Engine permet de contenir l'état du jeu au complet. Elle est aussi responsable de l'avancement du jeu grâce à la méthode step(). Envoie au acteur un nouvel état du state selon besoin.

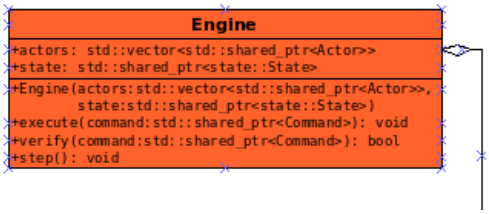
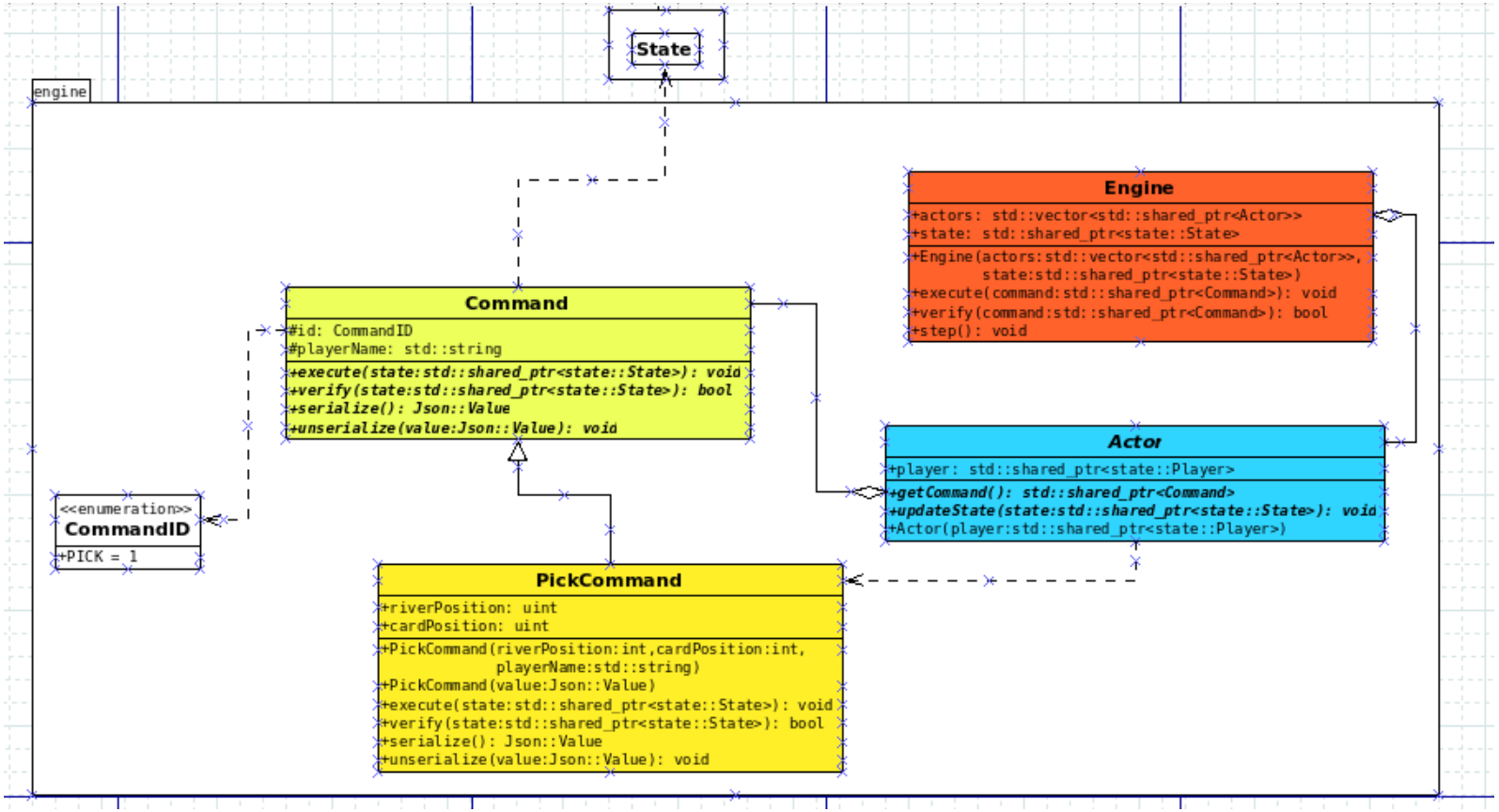

Engine
+actors: std::vector<std::shared_ptr<Actor>>
+state: std::shared_ptr<state::State>
+Engine(actors:std::vector<std::shared_ptr<Actor>>,
state:std::shared_ptr<state::State>)
+execute(command:std::shared_ptr<Command>): void
+verify(command:std::shared_ptr<Command>): bool
+step(): void

Illustration 3: Diagrammes des classes pour le moteur de jeu



5 Intelligence Artificielle

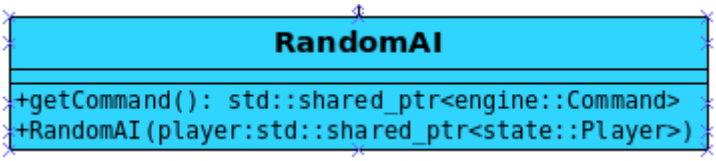
Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.

5.1 Stratégies

L'idée est de simuler le comportement d'un joueur. Plusieurs solutions plus ou moins évoluées sont possibles, elles sont présentées ci-dessous.

5.1.1 Intelligence minimale

La stratégie de l'IA aléatoire est très simple : à chaque tour, l'IA récupère les commandes possibles comme le ferait un vrai joueur. Ensuite, l'IA choisit une commande aléatoirement et la soumet à l'engine.

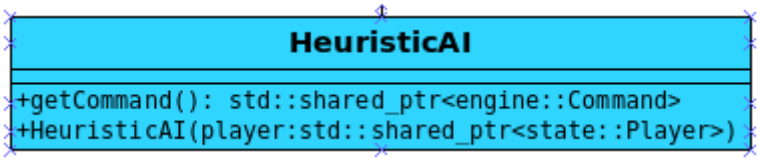


5.1.2 Intelligence basée sur des heuristiques

L'IA heuristique fait des coups afin d'optimiser les points de victoire. Le choix du coup joué est déterminé par des conditions relativement simples, ne faisant pas intervenir de structure de donnée particulière.

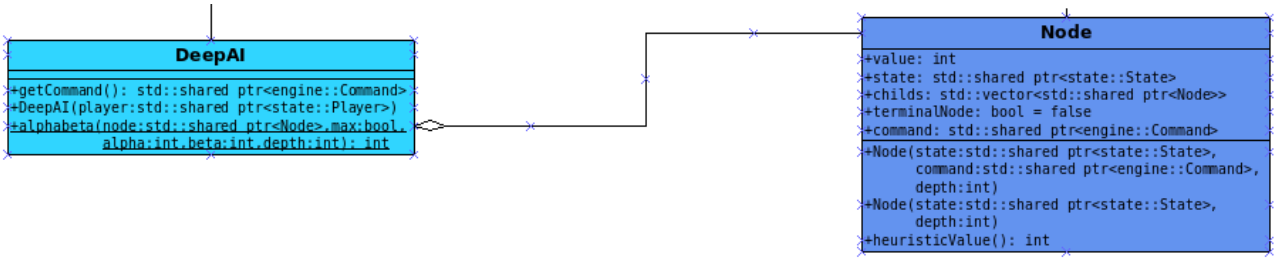
L'idée est de d'optimiser les points de victoire en choisissant les meilleurs cartes abordables. Les cartes sont triées en fonction de leur rivière : plus la rivière est élevée, plus la carte est bien.

Au début de la partie, on ne pourra pas choisir les meilleures cartes, par manque de ressources. Ce n'est pas grave car les cartes que l'on choisira par défaut donneront accès à plus de ressources.



5.1.3 Intelligence basée sur les arbres de recherche

Le but de cette IA est de choisir le coup joué afin d'avoir un état futur optimal. Le meilleur choix est déterminé grâce à un arbre Alpha Beta contenant les états futurs possibles.



5.2 Conception logiciel

L'intelligence artificielle est représentée par une classe héritée de Actor car l'IA est une entité qui interagit avec le jeu (comme un joueur par exemple).

La classe AI qui hérite directement de Actor est une classe abstraite qui définit les méthodes dont une IA quelconque a besoin pour fonctionner.
Lors de l'implémentation d'une IA, il faut surtout redéfinir la méthode getCommand : c'est elle qui décide quelle comande choisir.

5.3 Conception logiciel : extension pour l'IA minimale

RandomAI hérite de AI, il s'agit d'une IA qui récupère les commandes autorisées, puis en choisit une aléatoirement. La méthode getCommand est redéfinie afin d'avoir un comportement aléatoire.

5.4 Conception logiciel : extension pour IA heuristique

Cette IA est représentée par la classe HeuristicAI, qui hérite de AI. Pas de structure particulière utilisée.

5.5 Conception logiciel : extension pour IA basée sur les arbres de recherches

Cette IA est représentée par la classe DeepAI, qui hérite de AI.

On implémente un arbre qui contient des états futurs possibles.

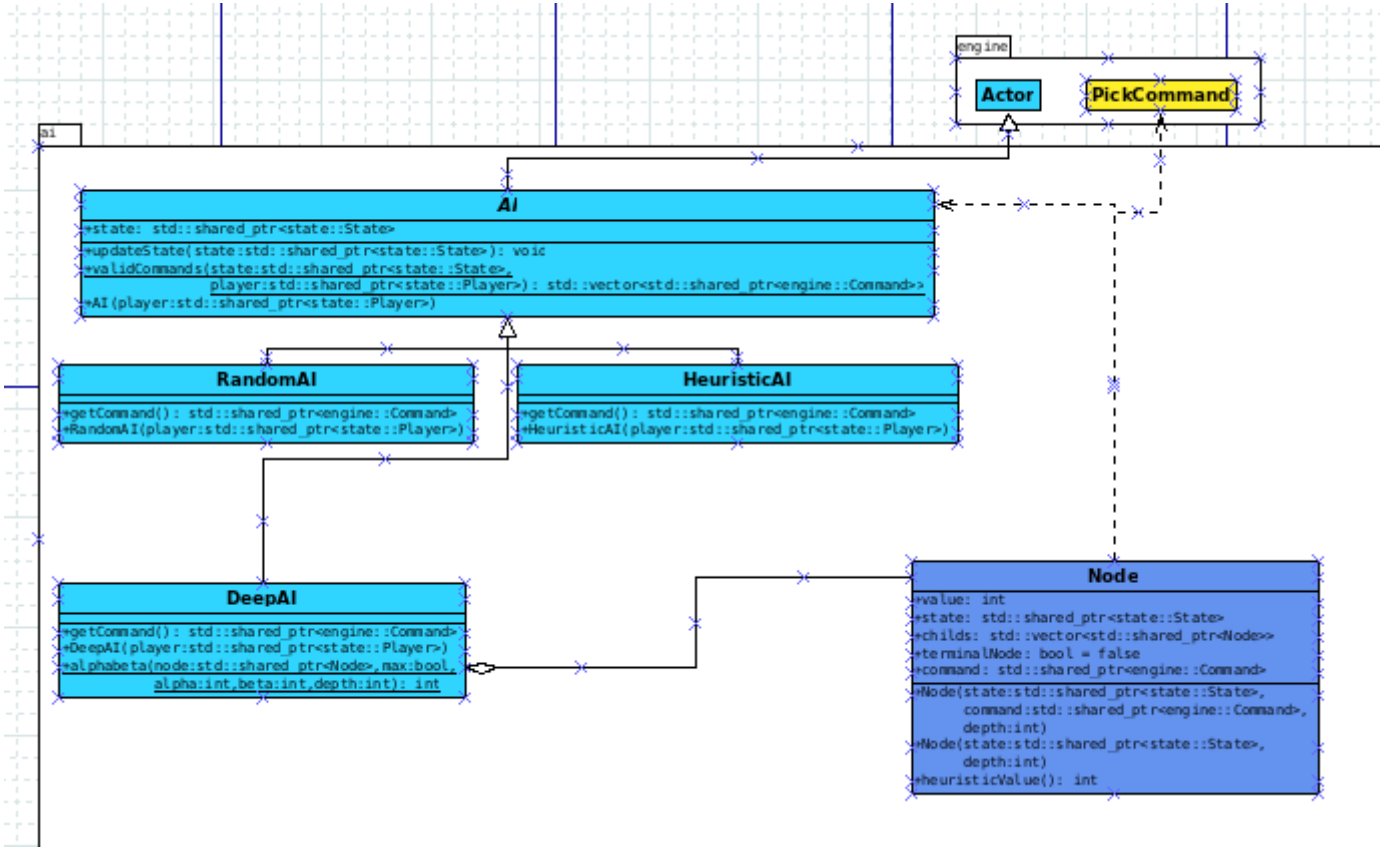
Ensuite, on applique la méthode de l'élagage alpha-bêta, une version amélioré de l'algorithme MinMax (jeu tour à tour deux joueurs). L'idée est d'identifier les chemins dans l'arbre qui sont explorés inutilement. , afin de les éviter et réduire la quantité de calculs dans la recherche de solution.

Pour cela l'algorithme repose sur l'idée de la génération de l'arbre selon un processus dit en « profondeur d'abord » où, avant de développer un nœud frère (issue du même parent), les fils sont développés. Ainsi, on navigue dans l'information en la remontant des feuilles et en la redescendant vers d'autres feuilles. .

Le principe de alpha-bêta est de tenir à jour à chaque deux variables α et β qui contiennent respectivement à chaque moment du développement de l'arbre un intervalle de confiance sur la valeur minimale que le joueur peut espérer obtenir pour le coup à jouer étant donné la position où il se trouve et la valeur maximale.

Au début de l'algorithme, on initialise cet intervalle de]- inf;+inf[. On identifie ensuite les nœuds lors de l'exploration qui sont en contradiction avec cet intervalle et on met à jour.

Illustration 4: Diagramme de classes pour l'intelligence artificielle



6 Modularisation / Réseau

6.1 Organisation des modules

Pour avoir un affichage fluide et une logique de jeu réactive nous sépareront le client en deux modules, une responsable de la communication avec le serveur et une responsable de l'interface.

Pour que plusieurs parties puissent être joués en parallèle chaque partie sera sur un module de plus le lobby qui accepte les clients et les gèrent en attendant qu'il entre dans une partie est également un module.

6.1.1 Répartition sur différents threads

Côté client :
Le module de communication avec le serveur tourne sur le thread principal et le module de l'interface est sur un second thread détaché.

Côté serveur :

Sur le thread principal nous lisons le terminal pour arrêter le serveur si l'utilisateur écrit « stop » dans le terminal.

Le serveur est sur un thread détaché qui va attendre que les clients se connecte à celui-ci , lorsque c'est le cas l'identification du joueur va se faire sur un nouveau thread qui se terminera une fois que le joueur rejoindra le lobby.

Lorsqu'une partie commence, celle-ci va tourner sur un thread détaché.

Avec cette répartition le serveur peut simultanément recevoir des clients et faire jouer plusieurs parties à différents clients en parallèle.

6.2 Conception logiciel

6.2.1 Classes partagées

Pour pouvoir communiquer proprement, des messages sérialisables sont crées, les messages peuvent être de différentes natures :

- Simple accusé de récépton
- Requete de commande émise par le server pour que le client joue son tour
- Commande émise par un client pour le serveur
- Nom d’utilisateur du client (utilisé pour s’identifier aupres du serveur)
- Nouveau state envoyé par le serveur pour que le client se synchronise

Tous ces messages héritent de la classe Message , classe qui possede une méthode statique qui

permet de deserializer un message sans connaître sa nature au préalable, tous les message peuvent être serialisé pour être envoyé sur le réseau.

6.2.2 Classes serveur

L'architecture utilisé ici est grandement inspiré de celle créée par Dmytro Radchuk dans son livre Boost.Asio C++ Network programming cookbook.

Le serveur va utiliser l'accepteur pour recevoir les connection et une fois qu'une connection est reçu le connectionService prend le relais sur un thread séparé pour gérer cette connexion entrante.

Notre ConnectionService va attendre que le client envoie son nom d'utilisateur puis lui envoyé un accusé de récéption puis va créer un objet client et l'ajotuer au lobby.

Lorsque qu'une partie est crée les deux joueurs sont sortis du lobby pour aller dans la partie, a partir de ce moment la c'est la partie qui va communiquer avec les clients.

La classe Human hérite de Actor (de l'engine) ainsi cet acteur va lorsqu'on lui actualise son état du jeu envoyer au client distant l'état du jeu (StateMessage) et lorsque qu'on va faire un getCommand envoyer au client une requete de commande (RequestCommandMessage) et recevoir une commande (CommandMessage) du client.

La classe Game représente une partie et possede ainsi un engine , c'est celle ci qui va dire a l'engine d'avancer tant que la partie n'est pas fini, un thread lui est dédié.

6.2.3 Classes client

Le client est plutot simple , on a une classe Client qui va se connecter au server et entretenir cette connexion, lorsqu'il reçoit le premier état du jeu , il créé l'interface pour l'utilisateur.

L'inteface est sur un thread séparé et a pour fonction d'afficher le render et de recevoir les input client, elle stocke une commande lorsque qu'un utlisateur clique sur une carte.

Illustration 5: Diagramme de classes pour les messages

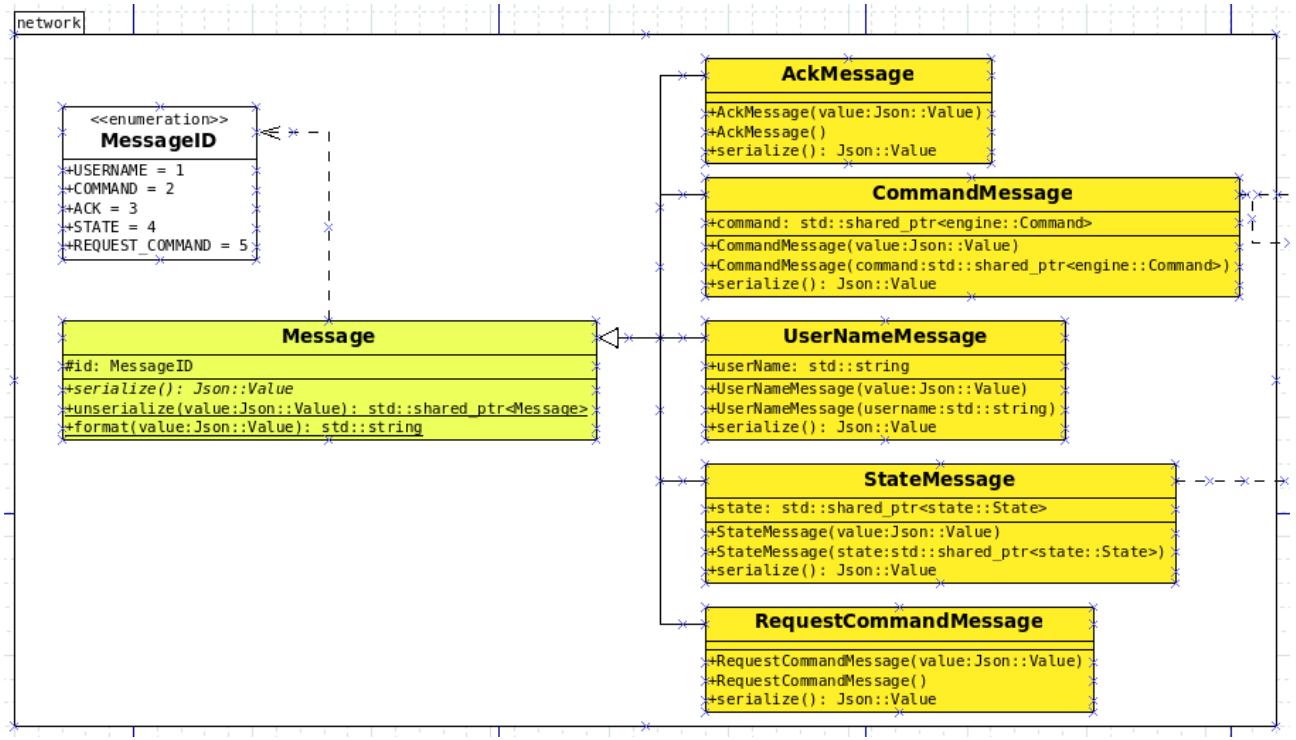


Illustration 6: Diagramme de classes pour le serveur

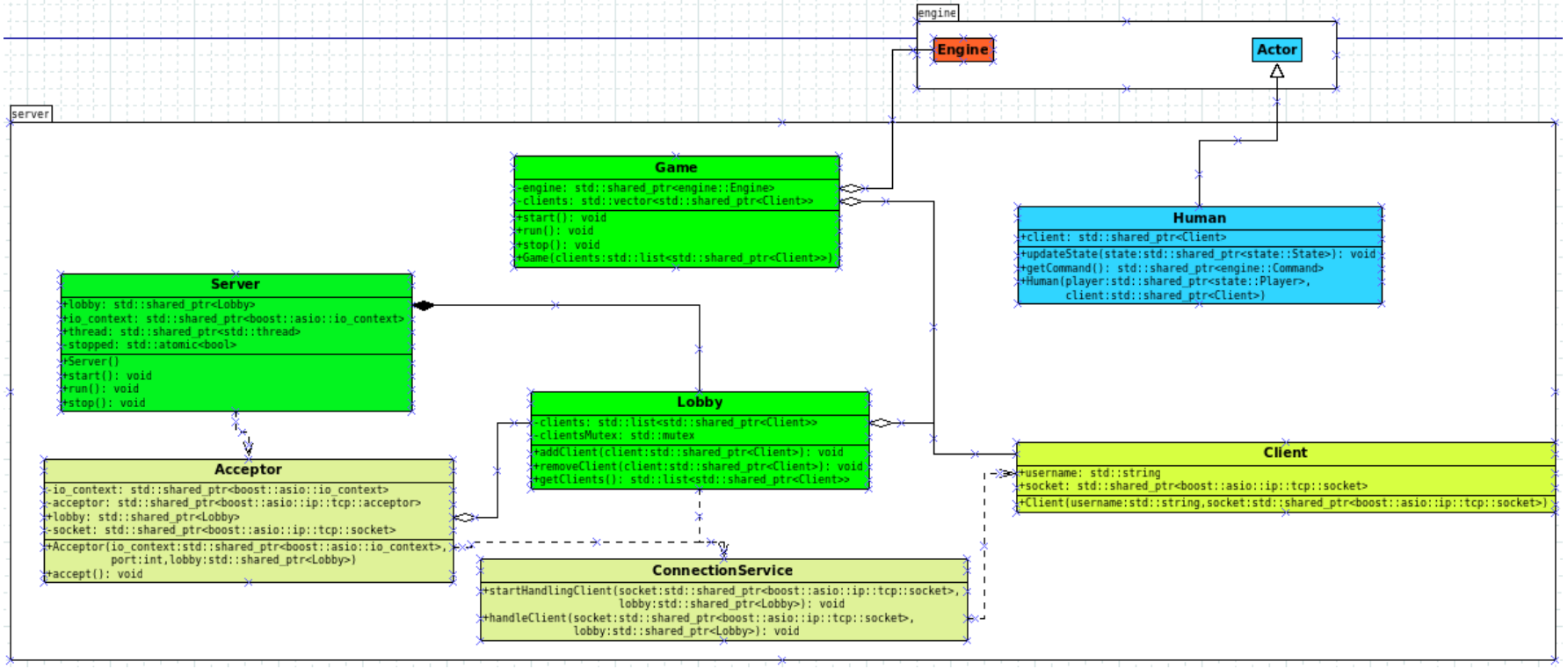


Illustration 7: Diagramme de classes pour le client