Projet Ninja Saga

Moïse BEUGRE – Ouajih DADAOUA

w

Figure 1 - Exemple du jeu Ninja Saga

Table des matières

Objectif

1.1 Présentation générale

L'objectif de ce projet est la réalisation du jeu "Ninja Saga", avec les règles les plus simples. Un exemple est présenté en Figure 1. Des combats entre personnages pourront être faits en tenant compte des points de vie, des points de mouvement en possession selon le personnage choisi.

Les personnages auront des caractéristiques qui influencent le combat, par exemple plus un personnage sera fort grand et imposant moins il sera rapide mais ses attaques auront plus d'impact.

Modes de jeu :

-Multijoueur : Player vs Player, chaque joueur choisit un personnage et initie ensuite le combat.

-Solo : Player vs IA , le joueur choisit un personnage et le personnage contre qui il veut se battre et initie un combat contre l'ordinateur (possibilité de choisir la difficulté).

-Survie : Player vs IA, le joueur choisit un personnage et va se battre contre tous les personnages du jeu en gardant à chaque fois le nombre de points de vie avec lesquelles il a terminé le combat précédent sachant que la difficulté augmente crescendo.

Type de personnages :

-Thork -> Fort, grand et robuste il se caractérise par ses attaques dévastatrices bien aidé par son marteau ravageur. Cependant il accuse un déficit en terme de vitesse et de réactivité.

-Flint -> Connu pour sa vitesse éclair ainsi que ses pyro-attaques, il se démarque par ses réflexes supersoniques qui lui permettent d'avoir toujours un temps d'avance sur ses adversaires.

-Seku -> Grand maître des arts martiaux il est connu pour ses attaques corps à corps redoutables et précises.

-Kuro -> Célèbre Ninja reconnu pour ses attaques furtives, maître du Kung fu et du Nunchaku, très véloce mais aussi moins robuste que ses compères.

Type de coups:

- Coup de poing & Coup de pied : ce coup cause des dommages valant 20 points de vie à l'adversaire sinon 10 point s'il est en position de défense.
- Uppercut & Flash Kick : ce coup cause des dommages valant 30 points de vie à l'adversaire sinon 15 points s'il est en position de défense.

  L'uppercut est très puissant pour repousser l'adversaire il a un seul désavantage c'est que sa zone d'impact est très réduite. Quant au Flash Kick, il est rapide et efficace. Il couvre un large champ d'action et limite les interventions et parades de l'opposant.

1.2 Règles du jeu

Au début de la partie, chaque joueur choisit un combattant et les combattants ont le même nombre de points de vie. La partie se termine lorsqu'un joueur n'a plus de points de vie.

1.3 Conception Logiciel

Les textures de personnages sont de tailles 100x100 pixels tandis que les textures de terrains de taille 700x400.

Les textures de personnages ont été ajustées de sorte à avoir quatres différentes lignes de sprites:

- La première ligne représente la position de ‘’garde’’. C’est la position initiale du Fighter en début de partie.
- La deuxième ligne met en animation l’attaque ‘’coup de poing” d’un Fighter.
- La troisième ligne est le mode ‘’défense” qui permet de réduire les dégâts subis lors d’une attaque
- La dernière sera utilisé lorsque le Fighter n’a plus de points de vie. C’est le statut ‘’DEAD”.

Figure 2 - Textures pour les personnages

Figure 3 - Textures pour les terrains de combat

Figure 4 - Textures pour les tuiles de lettres et chiffres

2 Description et conception des états

2.1 Description des états

Un état du jeu est formé d'un ensemble d'éléments mobiles (les combattants) dont l'état varie au cours de la partie et d'un décor statique.

2.1.1 Etats des combattants

Un combattant est soit contrôlé par un joueur soit contrôlé par une Intelligence Artificielle (I.A), il possède un nombre de points de vie à l'état t, un nom et un ensemble de statistiques (Points de vie, Points de magie, Combo) qui lui sont attribués en début de partie. Lorsqu'un combattant effectue un nombre de coups réussi dépassant un seuil, il entre dans un statut *Spécial* qui lui donne droit à des attaques plus puissantes. Le combattant est considéré comme étant "mort" si son nombre de points de vie atteint 0.

2.1.1 Etats de l'arrière plan (BackgroundManager)

L'arrière plan représente un terrain attribué à chaque combattant. Les combattants sont donc favorisés lorsque le combat se déroule sur le terrain qui leur ait attribué. Les statistiques d'attaques et de défense sont améliorées.
Le passage d'un terrain à un autre peut représenter le passage à un niveau supérieur lors d'une partie.

2.2 Conception logiciel

Le package état peut se diviser en trois sous-partie:

- Une partie gérant les personnages, en rouge
- Une partie gérant l'environnement, bleu
- Une classe représentant l'état global du jeu, en orange

La classe Player contient l'ensemble des éléments permettant de caractériser l'état d'un joueur. Chaque joueur est lié à un combattant par une relation de composition : un combattant ne peut pas exister sans joueur. Dans le cas où le combattant est contrôlé par l'IA, l'IA est considérée comme un joueur et contrôle donc son combattant.

Pour notre implémentation du jeu, nous avons prévu d'utiliser les quatre combattants suivant :

- Thork

- Flint
- Seku
- Kuro

Le constructeur Fighter permet d'avoir les quatre types de combattants en utilisant les différents identifiants qu'il a à sa disposition.

2.3 Conception logiciel : extension pour le rendu et moteur de jeu

On utilise le pattern Observer pour notifier les éléments dépendant de l'état lorsque celui-ci change. La classe State hérite de la classe Observable qui contient une liste d'objets héritant de la classe abstraite Observer.
Lorsque la classe State subit un changement, l'utilisateur utilise la méthode notifyObservers() pour notifier ses observers avec l'évènement correspondant au changement appliqué.
Il existe différents types d'Events:

- Les FighterEvent qui représente un changement d'état lié au statut des combattants (En vie, mort, en mode super) .
- Les environnement Event qui correspondent à des changement liés à l'environnement (changement de niveau) ou à des changements liées aux statistiques de la partie.
- Dans le cas où les fighters et le terrain change l'évènement est appelé : ALLCHANGED

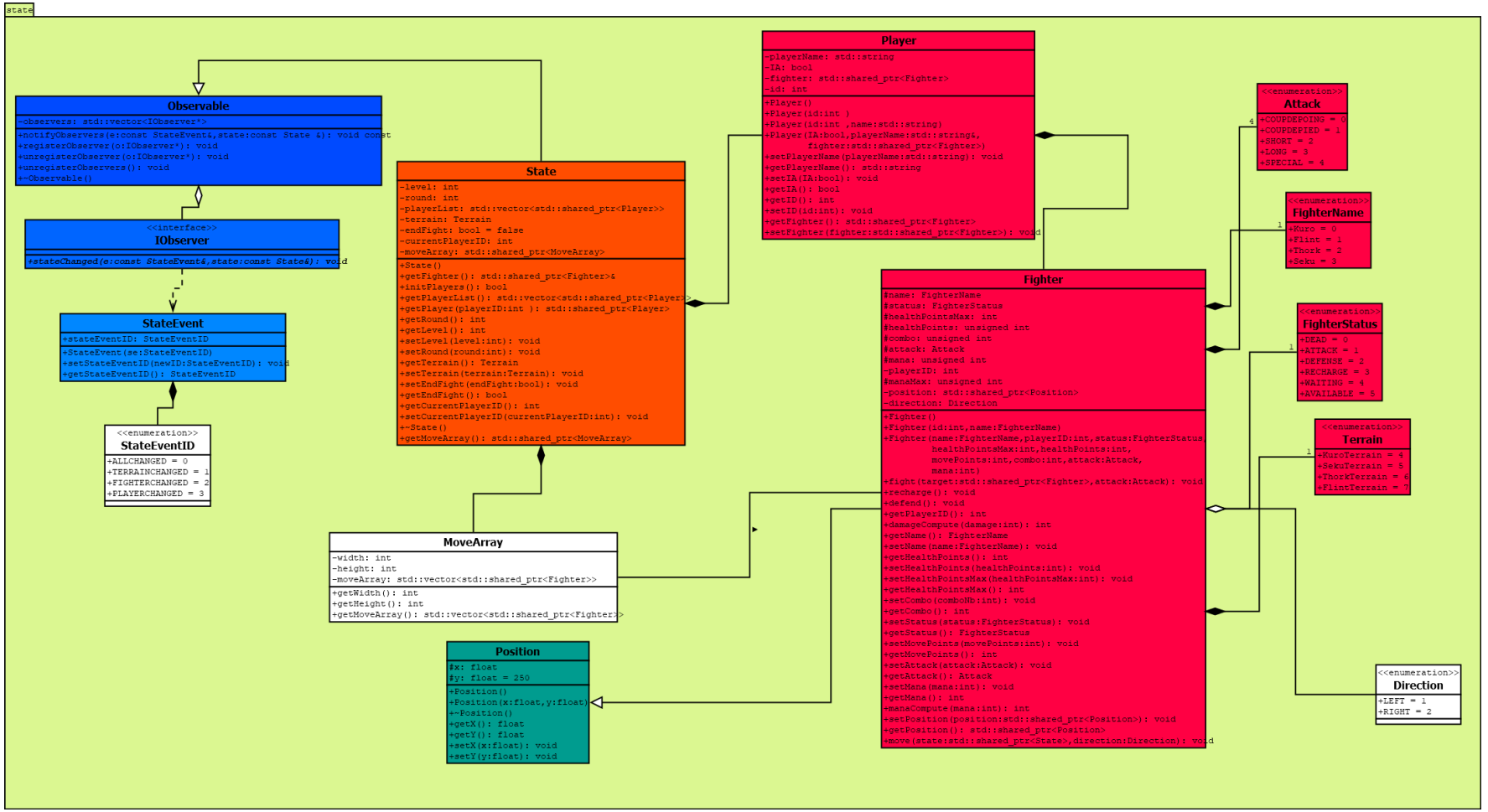

Figure 5 -Diagramme des classes d'état

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Notre stratégie de rendu d'un état se base sur l'utilisation de l'interface SFML qui s'appuie sur OpenGL afin de générer un rendu en 2D. Nous avons utiliser les fonctions de SFML afin de charger dans le processeur, la liste des éléments à afficher par le processeur graphique.

Nous avons découpé notre affichage graphique sur deux niveaux, l'arrière plan qui représente le terrain de combat et les éléments mobiles à savoir les combattants (Thork, Flint, Seku, Kuro) qui se superposent sur la couche précédente. On transmet l'état du jeu ainsi que les textures des différents terrains et la texture des combattants à afficher.

Lorsque qu'un changement d'état se produit, la vue est modifiée en fonction du changement appliqué à l'état. Si le changement modifie uniquement la position des combattants, seule le rendu des combattants est mis à jour, si ce changement s'applique à l'environnement l'ensemble du rendu de la carte est mis à jour. Lorsque l'ensemble de l'état est modifié le rendu de l'état est entièrement mis à jour. Dans le cas où le jeu est fini la fenêtre est automatiquement fermée.

3.2 Conception logiciel

Pour afficher un état, on utilise les cinq classes suivantes: TileSet, TextureManager,BackgroundManager, FighterRender et StateLayer.

La classe **TextureManager** est une classe singulière qui va nous permettre de récupérer une instance du Tile que l'on va dessiner (soit un Tile de terrain ou de fighter) à partir de la classe TileSet qui charge les textures. Cette instance est ainsi appelée dans les classes **BackgroundManager** et **FighterRender** qui nous permettent de dessiner les bonnes textures. Enfin, la classe **StateLayer** effectue un travail de synthèse en positionnant et dessinant les différentes textures récupérées dans **BackgroundManager** et **FighterRender** constituant ainsi le rendu final.

3.3 Exemple de rendu

Figure 6 - Exemple de rendu

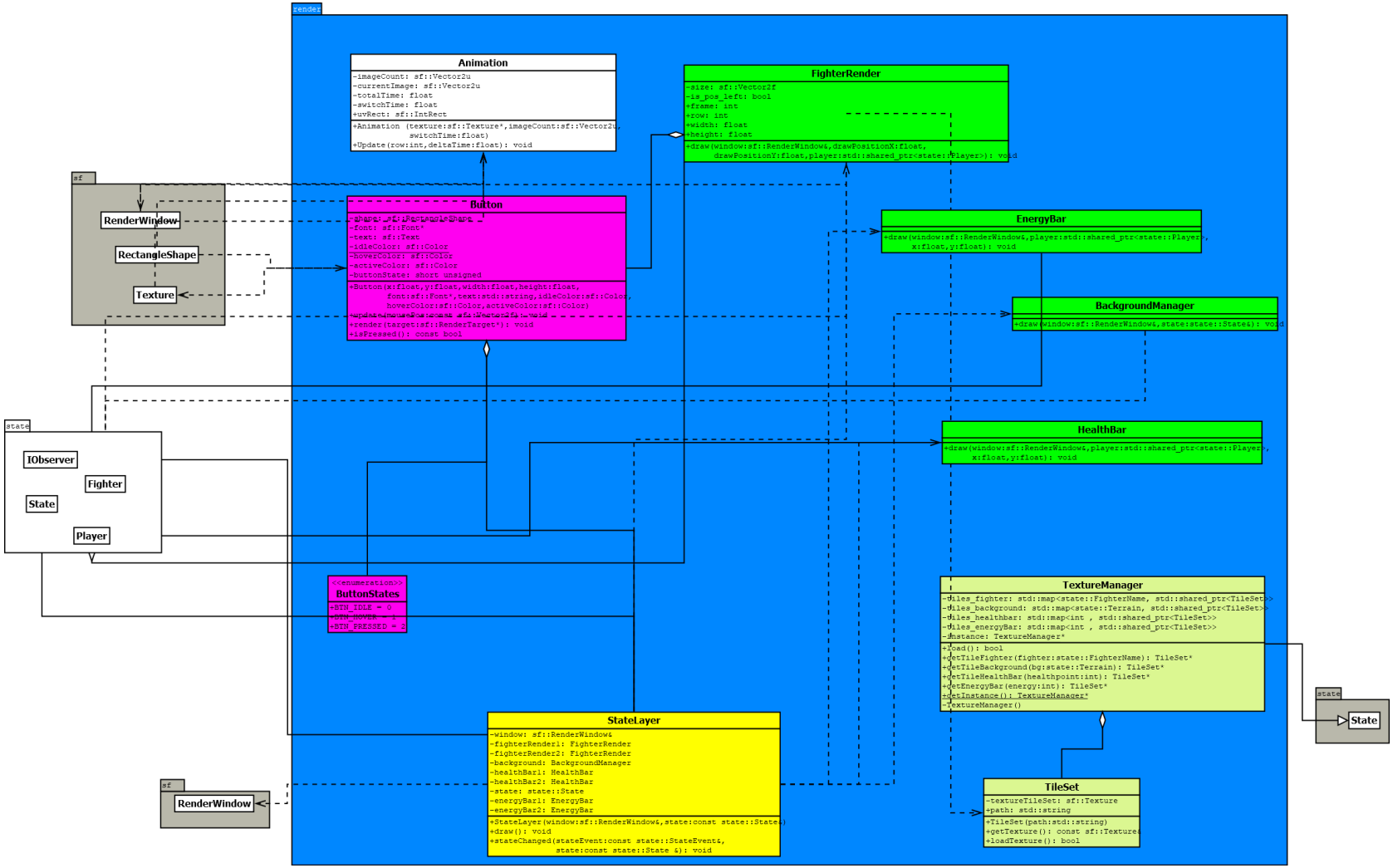

Figure 7 -*Diagramme de classes pour le rendu*

4 Règles de changement d'états et moteur de jeu

4.1 Stratégie de conception du moteur de jeu et règles de changement d'états

En début de partie, tous les fighters ont le statut "AVAILABLE". Ils peuvent donc attaquer, défendre ou recharger en cas de points d'énergie (mana) insuffisant. Une fois qu'un fighter a fait une action, c'est la fin de son tour et c'est le tour du fighter adverse.

Chaque fighter peut effectuer 3 types d'action lorsque son état est AVAILABLE c'est-à-dire disponible :

- attaquer : le fighter peut infliger des dégâts à un fighter. Le fighter aura le statut ATTACK. La touche "A" permet d'exécuter cette commande.

- défendre : le fighter peut choisir de se mettre en position de défense pour anticiper une potentielle attaque de l'adversaire. Ce statut va lui permettre de sauver des points de points de vie. Son statut à l'issue de cette action est DEFENSE. La touche "D" permet d'exécuter cette commande.

- recharger : le fighter peut choisir de recharger ses points d'énergie (mana) pour effectuer des attaques aux tours suivants. Son statut est RECHARGE. La touche "R" permet d'exécuter cette commande.

- terminer son tour: cette action permet au joueur adverse de jouer. La touche "T" permet d'exécuter cette commande.

4.2 Conception logiciel

Le diagramme des classes pour le moteur du jeu est disponible à la page suivante. Ce diagramme est constitué de deux classes principales Engine et Command.

Classe Engine: C'est le moteur du jeu en lui-même. Cette classe contient l'état du jeu et la liste des commandes qui devront être exécutées. Cette liste est un

tableau associatif std::map car un nombre entier est associé à chaque action pour pouvoir indiquer son niveau de priorité.
Une fonction ‘’addCommand’’ permet d’enregistrer chaque nouvelle commande avec sa priorité. La méthode ‘’update’’ va permettre lorsqu’un tour complet est terminé d’exécuter les commandes enregistrées dans le tableau associatif. Il va aussi prévenir le ‘’render’’ qu’il y a eu des changements sur l’état en appelant le ‘’notifyObserver’’ de state.

Cette méthode fait appel à stateChanged, fonction abstraite d’Observer dont le fils est stateLayer de render. Ainsi dans stateLayer la fonction stateChanged est appelée. Cette fonction lance des fonctions de render pour modifier l’affichage selon le state passé en argument. ‘’update’’ va aussi vider le tableau associatif de toutes ses données pour que celui-ci soit prêt à accueillir un nouveau tour d’actions.
Classe Command: Cette classe va permettre de définir un type commun pour pouvoir enregistrer toutes les actions dans une même std::Map.
Les actions héritent donc toutes de cette classe et ont un format très similaires. Chacune des classes **AttackCommand**, **RechargeCommand**, **ChangeCommand** et **DefenseCommand** ont une méthode ‘’execute’’ qui va modifier l’état actuel. Chacune de ces classes va enregistrer les informations dont il a besoin dans des attributs sur lesquels ‘’execute’’ pourra agir.

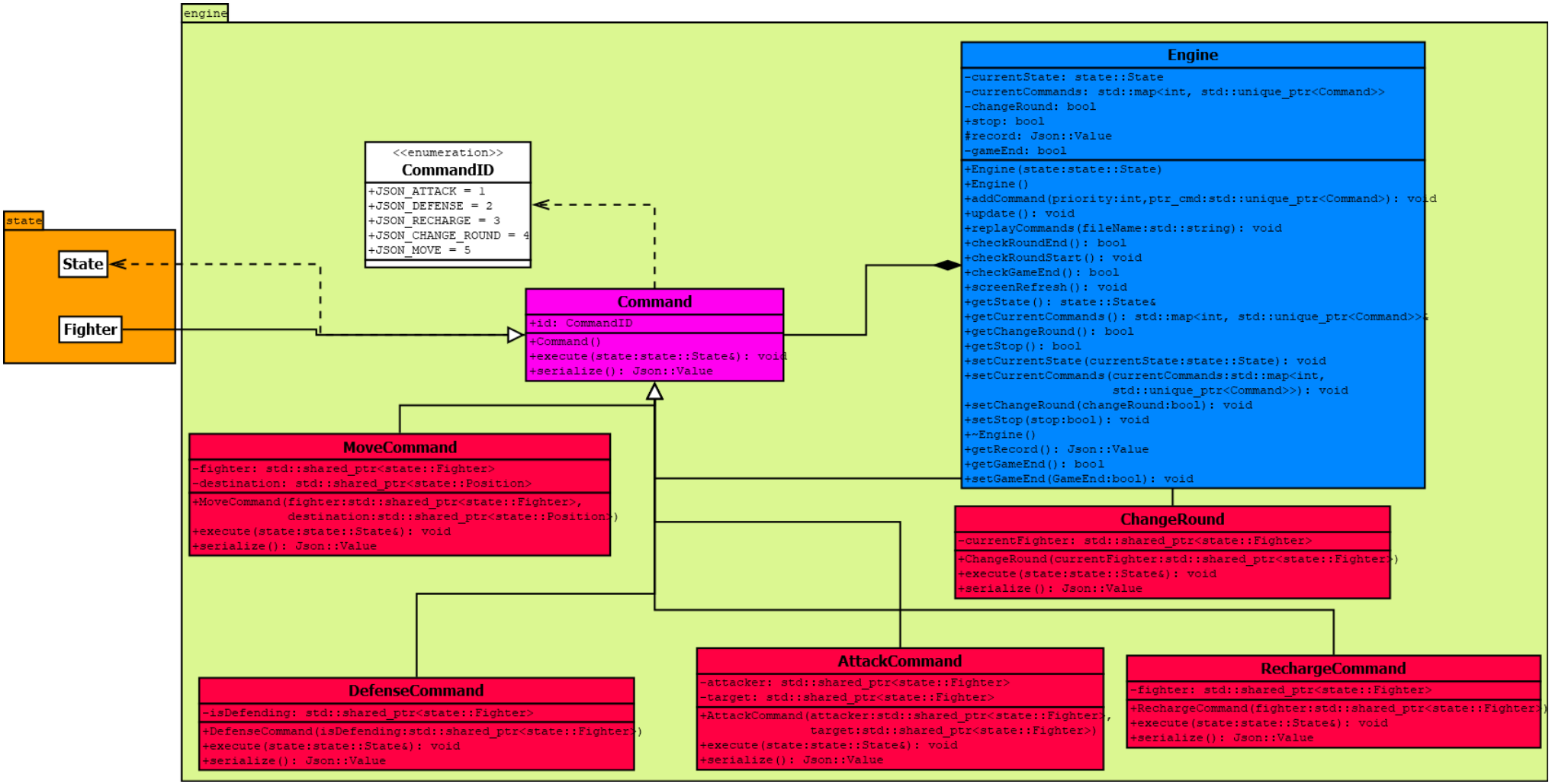

Figure 8 - Diagramme de classes pour le moteur

5 Intelligence Artificielle

L'IA est considérée comme un joueur avec une liste de commandes à exécuter. La commande à exécuter est choisie selon l'algorithme de l'intelligence Artificielle appliquée.

5.1 Stratégies

5.1.1 Intelligence minimale

Cette stratégie est la même pour tous les personnages. De manière aléatoire, on choisit une action parmi les trois possibles et l'IA l'exécute. Cette IA sera comparée par la suite à une IA basée sur des heuristiques puis à une autre basée sur une autre basée sur des arbres.

5.1.2 Intelligence basée sur des heuristiques

Nous proposons ensuite un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard et donner une chance à l'IA de remporter la partie.

On privilégie presque toujours l'attaque à la recharge d'énergie. Et ce dans la mesure où le « fighter » a assez de points d'énergie pour attaquer.

L'IA défend lorsque que ses points de vie sont inférieurs au quart des points de vie initiale.

L'IA recharge lorsque ses points d'énergie sont insuffisants pour lui permettre de faire une attaque.

L'IA attaque si :

- elle n'est pas en danger
- si elle est en danger mais elle dispose d'assez de points de vie pour faire une attaque qui peut lui permettre de remporter la partie.

Si l'IA peut attaquer mais que ses points de vie sont trop faibles, elle va prioriser la défense. Dans le cas où, l'IA ne peut pas attaquer, la défense est prioritaire.

5.1.3 Intelligence basée sur les arbres de recherche

Cette IA est basée sur la théorie des arbres de recherche. Dans notre cas, l'algorithme minimax est plus adapté pour établir une IA plus efficace que l'IA heuristique précédente. Un système de « scoring » est mis en place pour évaluer l'impact de chaque action sur le jeu. Le score est calculé en fonction des points de vie et des points d'énergie. Plus le score est important, plus cette action sera prioritaire.

5.2 Conception logiciel

L'IA va d'abord vérifier les actions possibles du fighter en tenant compte de ses points de mana et de ses points de vie.

Classe Random_ai
C'est la classe qui implémente l'IA Random.
Pour le choix de l'action, un entier va prendre une valeur aléatoire entre 0 et 2.
Selon le résultat, l'IA effectuera une action:

- 0 pour l'attaque,
- 1 pour la défense
- 2 pour la recharge.

Classe Heuristic_ai
C'est la classe qui implémente l'IA Heuristic.
Le choix des actions de l'IA tient compte des points de vie de l'IA et de ses points d'énergie(mana). Les points de vie et les points d'énergie du fighter adverse sont aussi pris en compte.

Classe Deep_ai
C'est la classe qui permet d'évaluer les différentes actions possibles de l'IA en calculant un score. L'algorithme minimax est appliqué en utilisant les méthodes maximiseScore and minimiseScore.
La fonction GetCommand permet de récupérer la commande choisise par l'utilisateur et de l'évaluer.

Classe NodeDeep_ai

Cette classe se charge de construire l'arbre en ajoutant les différents nœuds et els différents niveau de celui-ci. Une copie de l'état est faite afin d'évaluer les différents score à chaque nœud.

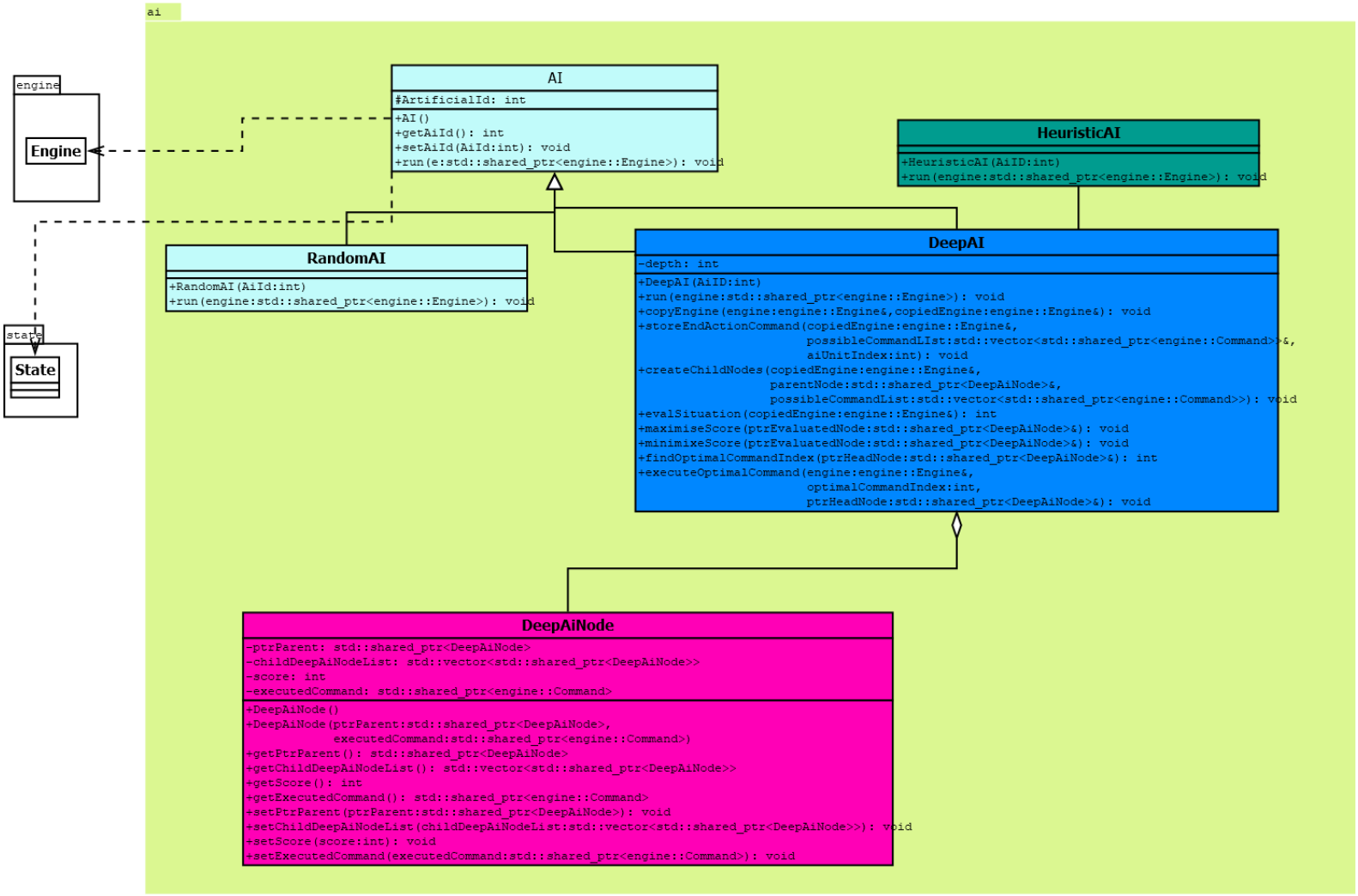

Figure 9 - Diagramme de classes pour l'Intelligence Artificielle

6 Modularisation

6.1 Organisation des modules

Afin d'anticiper la mise en réseau du jeu et la possibilité de jouer en mode multi-joueurs, le client et le serveur sont séparés dans des threads différents. Le jeu est décomposé en modules qui vont être mis d'un côté sur un serveur central et de l'autre du côté dans chaque client.
Les informations sur les commandes effectuées par un joueur qui sont récupérées dans le render du client sont envoyées via des fichiers au format Json. Chacune des commandes peut être sérialisée en format Json qui va contenir les informations nécessaires pour exécuter la commande dans le moteur du serveur. Ensuite, ce fichier sera lu du côté du moteur pour faire les modifications nécessaires sur l'état du jeu en lisant les commandes dans l'ordre. Le serveur pourra ensuite envoyer les modifications à faire à tous les clients pour que tous les joueurs possèdent le même état du jeu.

6.1.1 Répartition sur différents threads

Le « enginethread » permet de simuler le serveur en initialisant l'ensemble des éléments nécessaires pour le bon fonctionnement du moteur de jeu.
Le « clientthread » permet de mettre en place l'ensemble de l'affichage graphique en fonctions de l'état du jeu. Les éléments communiquants entre les deux threads sont les commandes. Les commandes saisies par l'utilisateur sont exécutées sur le moteur initialisé par enginethread. Après exécution, le moteur de jeu renvoie les commandes exécutables à l'état actuel du jeu. Ces commandes reçues du moteur sont alors exécutées pour mettre à jour le « state ».
On fait jouer un joueur contre l'IA heuristic.

6.1.2 Répartition sur différentes machines : rassemblement des joueurs

Le but de cette partie de créer une interface pour pouvoir permettre à plusieurs de se connecter et d'obtenir une liste des joueurs. Plus tard, il s'agira de connecter le jeu à cette interface pour qu'on puisse y jouer à plusieurs.
On implémente une API REST (Application Programming Interface Representational State Transfer). Un travail préalable de documentation a été fait en utilisant openclassroom : https://openclassrooms.com/fr/courses/3449001-utilisez-des-api-rest-dans-vos-projets-web/3449008-quest-ce-quune-api .

Nous utilisons donc les services CRUD (Create Read Update Delete) proposés par cette architecture de communication client-serveur. Les différentes requêtes utilisées sont :

- **GET /player/<id>** : récupére les données associées à un joueur déjà enregistré.
  {''name'' : ''nom_du_joueur'', ''free'' :true\false}
- **PUT/player** : ajoute un nouveau joueur à la liste des joueurs de la partie à condition que le nombre maximal de joueurs ne soit pas atteint : 2 dans notre cas.
  Dans le cas où la requête est valide, la réponse obtnenue est :
  {''id'' :numero}
- **POST /player/<id>** : modifie une ou plusieurs caractéristiques d'un joueur existant. Si l'Id ne correspond à aucun joueur, le message d'erreur « Invaalid Player id » s'affiche.
  La réponse à cette requête n'a pas de corps. Une requête GET permet de vérifier les nouvelles caractéristiques du joueur.
- **DELETE/player/<id>** : supprime un joueur de la liste des joueurs à condition que ce joueur fasse partie de la liste des joueurs.
  La réponse à cette requête n'a pas de corps.

6.2 Conception logiciel

6.2.1 Pour la modularisation

La fonction « thread() » lance un client et un serveur séparément.
La fonction « record() » permet de tester l'enregistrement des commandes sur une partie entre une IA heuristique et un joueur. Toutes les informations de cette partie sont écrites sous format Json dans le fichier « record.txt ».
C'est celui-ci qui sera ouvert par la fonction « play() » qui va permettre cette fois de tester l'exécution des commandes à partir des informations enregistrées au format Json.
En format Json, on enregistre les commandes exécutés par chaque. L'Id du Player *IdPlayer* et la commande exécutée *CommandTypeId* qui peut être une action d'attaque, de défense, recharge, de passage de tour ou de mouvement.

Classe Modularisation:
Cette classe permet de mettre en place les bases permettant la séparation des modules client et serveur. Elle contient une fonction « engineThread() » permettant de lancer un thread engine et une fonction « clientThread() »

permettant de lancer une pour le visuel et la récupération des commandes saisie par l'utilisateur du côté client. La fonction « record() » permet d'enregistrer les commandes sérialisées en format Json et stockées dans l'attribut « record » de engine puis dans un fichier « record.txt » situé à la racine du projet. Enfin, ce fichier est lu et exécuté dans la fonction « play() » permettant ainsi de reproduire à l'identique les actions enregistrées.

Classe EngineClient :
Cette classe teste le moteur de jeu « engine »qui jusqu'à maintenant était testé dans le fichier main.cpp. Ceci donne une meilleure lisibilité au code.

Classe RenderClient :
Cette classe teste le rendu du jeu « render » qui jusqu'à maintenant était testé dans le fichier main.cpp. Ceci donne une meilleure lisibilité au code.

6.2.2 Pour le serveur

Pour réaliser l'API proposé, on utilise deux catégories de classes : l'une pour gérer les services de requêtes et l'autre pour gérer les informations du jeu.

AbstractService permet de définir toutes les méthodes des requêtes à savoir get, post, put et remove. Ensuite **ServiceException** permet de renvoyer le statut des requêtes en utilisant les codes http définis dans l'énumération *HttpStatus*.
La classe **PlayerService** sert d'intermédiaire entre les classes de gestion de requêtes et les classes de gestions de jeu. Elle redéfinit les méthodes de requêtes et les appliquent à une partie **Game** avec des joueurs **Player.**
L'ensemble des classes de service est géré par la classe **ServicesManager**. Elle se charge d'enregistrer la requête puis de la délivrer au service adéquat**.**

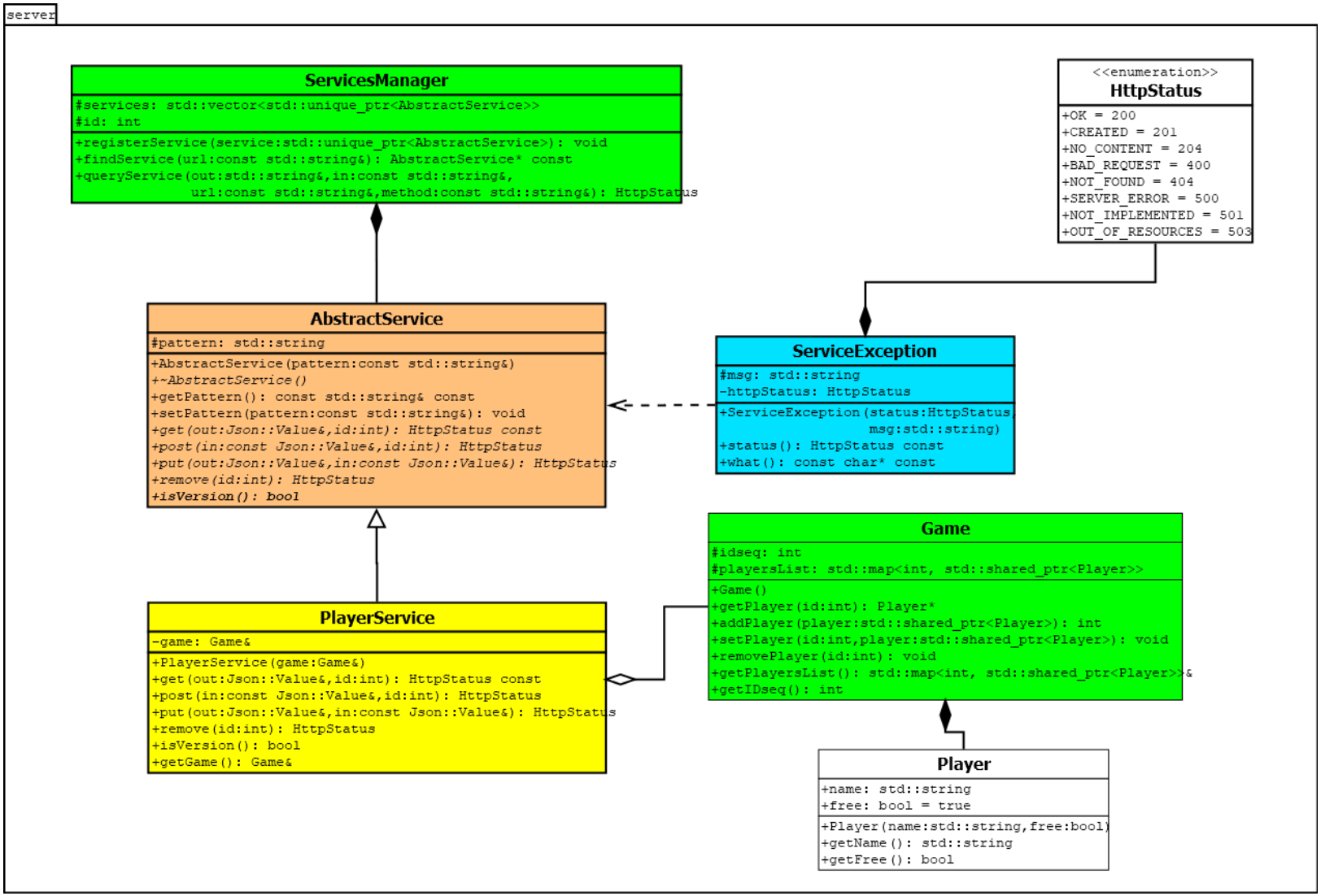

Figure 10 : Diagramme de classe pour le serveur

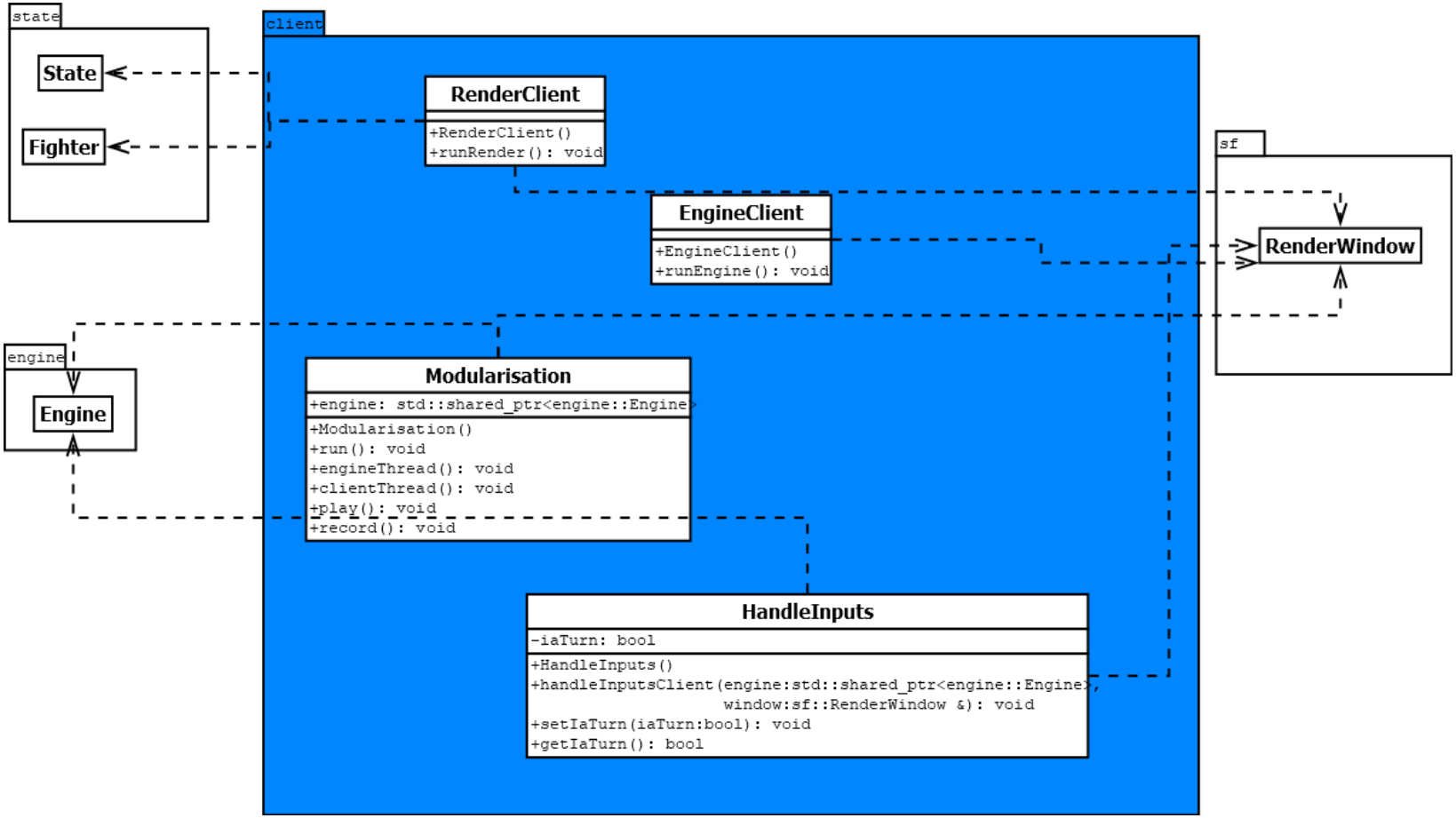

Figure 10 : Diagramme de classe pour la modularisation