Projet Fire Emblem

Thanusan SATHIAKUMAR – Léo CHAZALLET

Table des matières

1 Objectif

1.1 Présentation générale

Le jeu va être un jeu de combat stratégique en tour par tour. Le principe se base sur la partie combat de la série de jeux Fire Emblem.

FIGURE 1 - Image tirée du jeu Fire Emblem Fates : Birthright

Le joueur possède une ou plusieurs unités qui sont placées sur un espace quadrillé. Chacune des unités aura ses propres caractéristiques (points de vie, point de mana, sorts, portée de déplacement, portée d'attaque, valeur d'attaque, valeur de défense). Des ennemis sont présents sur chaque carte/niveau.possédant eux aussi des caractéristiques spécifiques. Le but va donc d'éliminer tous les ennemis présent sur la carte pour obtenir une victoire. A chaque tour le joueur va pouvoir décider pour une unité une action (se déplacer, attaquer, utiliser un sort).

1.2 Règles du jeu

L'utilisateur jouera contre des unités ennemis contrôlées par une IA par défault. Il y aura cependant la possiblité de passer en mode 2 joueurs par la suite.

Voici une liste des règles principales du jeu :

- le/les unité/s du joueur apparaissent du côté opposé de la carte à celle des unités ennemies
- le joueur ne peut effectuer qu'une seule action par unité par tour
- une action peut être soit un déplacement, une attaque ou l'utilisation d'un sort/objet
- la partie se termine quand tous les ennemis sont éliminés ou lorsque le joueur n'a plus d'unité

Le jeu est constitué d'un seul niveau pour l'instant. Il y aura une unité alliée (un héros) et une unité ennemie (un monstre). Celles-ci apparaîtront à un emplacement prédéfini. Ces unités auront des caractéristiques basiques et communes. Chacune aura des coordonnées de position, son orientation, des points de vie, une capacité de déplacement, une portée d'attaque, une valeur d'attaque et une valeur de défense.
Au fur et à mesure des niveaux, le nombre d'unités ainsi que la difficulté va augmenter. D'autres unités seront aussi disponible comme des mages qui pourront avoir en plus différents sorts disponibles, auront des points de mana et qui pourront attaquer à distance.
Chaque niveau aura une carte dédiée qui aura une taille adaptée au nombre d'unités.

S'il reste du temps, voici les fonctionnalités que nous aimerions implémenter :

- présence sur le terrain d'éléments apportant des bonus (points de vie, attaque ou défense) répartis aléatoirement sur la grille du jeu. Le joueur doit choisir de manière stratégique les éléments dont il aura besoin pour vaincre l'ennemi
- présence sur le terrain d'éléments apportant des malus au joueur (par exemple des trous à éviter pour ne pas perdre de la vie...)
- présence de cases permettant la téléportation du joueur d'une case à une autre prédéfinie ou aléatoire (permettant d'offrir au jeu un aspect stratégique)

1.3 Conception Logiciel

FIGURE 2 – Carte de la zone de combat

La zone de combat est une image de 800 par 800 pixels décomposée en 25 tiles sur la longeur et la largeur (un tile est un carré de 32 par 32 pixels).

FIGURE 3 - Sprite d'un personnage

Pour tout ce qui est gestion du texte, nous utiliserions les fonctionnalités proposées par la bibliothèque SFML.
Nous n'avons pas pu mettre l'ensemble des ressources que nous allons utiliser pour le jeu pour éviter de surcharger le rapport. Il est possible de tous les trouver dans le dossier « rsc » récupérable sur le GitHub: https://github.com/LeoChazl/plt/tree/master/rsc .

2 Description et conception des états

2.1 Description des états

Un état du jeu est composé d'entités fixes qui vont constituer la carte (les obstacles et les espaces) sur laquelle des éléments mobiles vont évoluer (les personnages, le curseur). Ces derniers sont associés à un joueur et possèdent de nombreuses caractéristiques (vie, valeur d'attaque, armure, ...).

2.1.1 Etat général

Il faut être capable de savoir dans quelle situation et à quel avancement se trouve le jeu.
Un nombre permet d'identifier le tour de jeu actuel et un autre indique le niveau.
Un booléen informe le système si la partie est terminée.
L'état général a aussi accès à tout ce qui constitue le jeu :

- la liste des joueurs qui ont eux-mêmes une liste indiquant les unités qu'ils contrôlent
- la carte du niveau avec des informations sur ses dimensions
- le curseur permettant au joueur d'interagir avec le jeu

2.1.2 Etats éléments fixes

L'ensemble des éléments fixes du jeu sont contenus dans un tableau à deux dimensions. Le tableau a des dimensions différentes en fonction du niveau et donc de la carte (le premier niveau est de taille 25x25 cases).
Il n'y a que deux types de cases possible :

- les obstacles : les emplacements sur lesquels les unités ne pourront se déplacer
- les espaces : les emplacements sur lesquels les unités pourront se déplacer

Le type de l'emplacement est identifiable grâce à un booléen et la position de chaque case est accessible via leur emplacement dans le tableau.

2.1.3 Etats éléments mobiles

Les éléments mobiles sont toutes les entités sur lesquelles le joueur pourra agir. Ils partagent tous une caractéristique commune : une position (x et y).

Les unités

Toutes les unités partagent les caractéristiques suivantes :

- points de vie
- capacité de déplacement
- valeur d'attaque
- valeur d'armure

- identité
- statut
- direction

L'identité permet de savoir le type d'unité. Il y a pour l'instant 3 types d'unités :

- le troll : une unité ennemie qui attaque au corps à corps et n'est capable que d'une attaque basique
- le chevalier : une unité alliée similaire au troll en terme de capacités
- le mage : une unité alliée ou ennemie qui en plus de pouvoir attaquer au corps à corps sera capable de lancer un sort et donc attaquer d'une plus grande portée. Ses capacités au combat rapproché est cependant bien inférieure au 2 unités précédentes. Pour pouvoir gérer le nombre de sorts que celui-ci peut envoyer, cette unité a une caractéristique mana supplémentaire.

Le statut indique l'état et la disponibilité du personnage. Il existe 4 status possible :

- sélectionné : le curseur pointe sur le personnage
- disponible : une action peut encore être effectué par le personnage pendant le tour actuel
- attente : le personnage a déjà effectué son action pour le tour
- mort : les points de vie du personnage sont à 0

Enfin la direction nous donne l'information sur l'orientation de l'unité et permet d'afficher la sprite correspondante. Il y a donc 4 direction possible : haut, bas, droite et gauche.

Le curseur

Cette entité va permettre d'indiquer au joueur ce qu'il sélectionne pour lui permettre de faire ses actions ou pour lui donner des informations.

2.1.4 Etat du joueur

Le joueur aura un identifiant pour le différencier des autres ainsi qu'un nom que le joueur pourra choisir selon sa préférence. Cet état contiendra aussi la liste de toutes les unités à son commandement.

2.2 Conception logiciel

Le diagramme des différents états de notre jeu est illustré via la figure n°4. Ce diagramme permet de mettre en évidence les groupes de classes suivants.

2.2.1 Classes générales

La classe State est le conteneur principal à partir duquel on peut accéder à toutes les données de l'état. Ainsi l'appel à cette classe permet d'instancier l'ensemble de l'état du jeu à un moment donné. Cette classe est en relation de composition avec la classe EntityMap et la classe Player. Pour ce dernier c'est une liste de pointeurs vers ce type d'objet qui permet la composition. Grâce à cela il est possible d'échelonner l'aspect multijoueur de notre jeu à un nombre plus important de joueurs..

Player permet d'instancier pour chaque joueur prenant part à une partie les différentes unités qui lui sont associées via une liste de référence de pointeurs vers les unités mobiles. De cette manière, il sera possible de rajouter autant d'unité que nécessaire dans les cas où le joueur possèderait de nombreuses unités.

Les modifications de l'état du jeu sont possibles en suivant un design pattern Observer et notamment grâce à l'interface IObserver. Cette interface en relation avec le rendu permet de prendre en compte l'ensemble des différentes modifications apportées par le joueur pour en modifier l'état.
Ainsi à l'arrivée d'un événement, la classe StateEvent en relation de contrainte avec IObserver détermine le type de changement à apporter sur l'état du jeu selon la liste d'énumération qui lui est attribué (changement sur le joueur, des unités mobiles, changement du tour de jeu ou du niveau).
La vitesse de traitement des événements pouvant varier, les événements reçus sont enregistrés dans une liste "obsevers" de type Obsevers. Cette liste permet de notifier l'ensemble des observer des événements à traiter.

2.2.2 Classes fixes

Les éléments statiques de la carte sont regroupés sous la classe EntityMap. Celle-ci contient des objets StaticEntity dont hérite les classes filles Obstacle et Space.

EntityMap permet d'instancier une grille de jeu sous la forme d'un tableau à deux dimensions contenant des pointeurs vers des objets Obstacle et Space. Cette grille appelée mapArray, est formée via la lecture d'un fichier texte constitué d'une suite de "1" (pour les espaces) et "0" (pour les obstacles). Ainsi à chaque niveau du jeu la génération de la map peut être rendu flexible selon le fichier texte ouvert par la fonction de génération du tableau mapArray.

Les classes Obstacle et Space possèdent une méthode qui permet de les différencier dans le tableau et ainsi connaître les emplacements sur lesquels les unités pourront se déplacer.

2.2.3 Classes mobiles

Les éléments mobiles de notre jeu sont regroupés via la classe abstraite MobileEntity. A partir de cette classe abstraite hérite plusieurs classes filles : Troll,Knight,Mage. Ces classes héritent des caractéristiques similaires issues de la classe mère (vie, armure, capacité de déplacement, valeur d'attaque, vie maximale possible, direction, statut et type d'unité). Les 3 dernières caractéristiques sont gérées avec des énumérations. Pour la direction c'est une énumération de 4 éléments (haut, bas, gauche et droite) alors que le statut en possède 4 (sélectionné, disponible, en attente et mort). Enfin, l'EntityId permet leur unicité en tant que classe avec 3 types possibles correspondant aux classes filles.
Les classes filles possèdent cependant des particularités permettant de les différencier notamment une méthode renvoyant l'élément de l'énumération d'EntityId correspondant ou encore via les différents types d'attaque possible pour un Mage amenant des méthodes et attributs particulière à cette classe.

Les coordonnées des différentes unités mobiles sur la carte sont instanciées via la classe Position dont hérite la classe fille MobileEntity.

De plus cette classe Position est la classe mère de la classe Cursor. Cette classe permet de déplacer les coordonnées du curseur sur une case particulière de la map. Ainsi elle peut être utilisé pour visualiser les caractéristiques de l'ennemi, pour choisir les coordonnées de déplacement d'une unité par le joueur ou pour viser une cible.

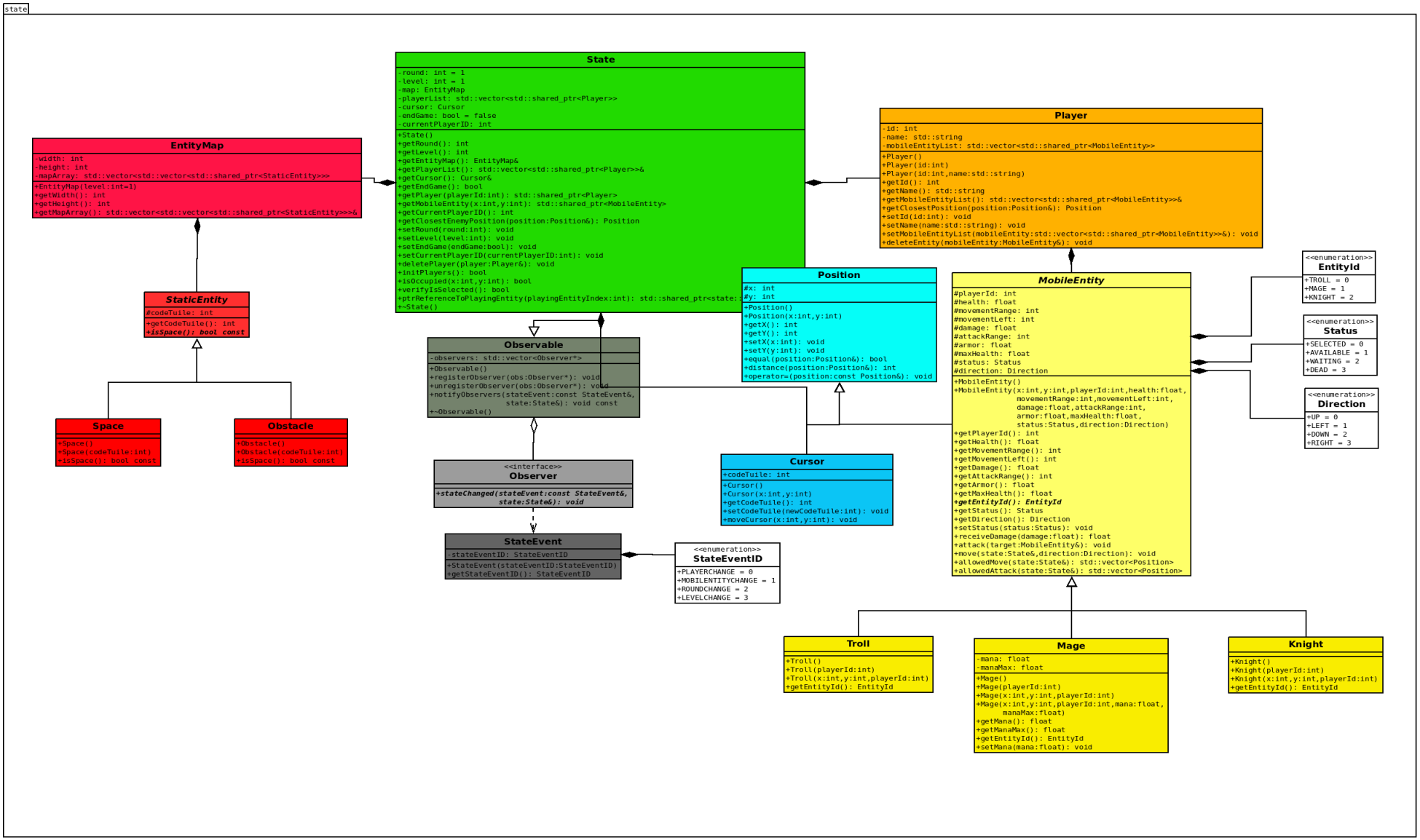

FIGURE 4 - Diagramme des classes d'état

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Dans le but de générer un rendu pour chaque état du jeu, nous avons choisi d'adopter un rendu par tuile à l'aide de la bibliothèque SFML.

Notre affichage va se diviser en deux parties. D'un côté, il y aura les ajouts graphiques faisable directement grâce à la bibliothèque SFML (image logo, rajout d'un fond de couleur). D'un autre côté, pour afficher l'état actuel, nous le découpons en trois couches : la couche pour la carte du terrain qui correspondra au mapArray (objet EntityMap), celle pour les unités qui récupéreront les informations sur les MobileEntity contenu dans les mobileEntityList des différents joueurs et enfin une ocuche pour le curseur qui correspond logiquement à notre objet Cursor. Chacune de ces couches à un tileset correspondant pour lequel il faudra déterminer les bonnes coordonnées pour récupérer le tile voulu. Pour réaliser l'affichage pour ces différentes couches, nous allons donc récupérer les informations nécessaires dans l'état et en fonction de celles-ci construire les objets SFML correspondants avec les bonnes positions sur les tilesets et sur l'écran pour avoir un affichage correspondant à l'état actuel.

Le mapArray est créé grâce aux données du fichier csv « map.csv » composé de 50x25 codes de tuile. Ces codes de tuiles sont utilisés pour déterminer quelle tuile doit être utilisé pour cette position dans le tableau de StaticEntity (qui correspondant au mapArray). Il suffit donc de modifier les valeurs des codes de tuile dans ce fichier pour modifier la map. Ce fichier se trouve dans le dossier rsc/Images.

La position de personnages est aussi définie aléatoirement dans un certain rectangle ce qui change la disposition à chaque lancement du jeu. Ceux-ci sont initialisés directement dans le code dans une fonction d'initilisation d'un état pour ce niveau se trouvant dans l'état.

Le curseur et la mapArray sont initialisés dans le constructeur de l'état puisque pour l'instant nous travaillons que sur un seul niveau.

3.2 Conception logiciel

Le diagramme des classes pour le rendu du jeu est décrit sur la figure 10 ci-dessous. Ce diagramme est constitué des trois classes suivantes.

Classe TileSet: Cette classe a pour but de charger et d'initialiser l'ensemble des tuiles nécessaires à l'affichage du rendu des différents états du jeu. En effet, à l'appel du constructeur un fichier de tuiles est chargé dans un objet de la classe "Texture" issue de la bibliothèque SFML. Ce fichier dépend de l'identifiant passé en argument du constructeur.
Cet identifiant est une variable de type TileSetID issue d'une classe d'énumération contenant :

- UNITSTILESET ⟶ identifiant permettant de charger les tuiles des personnages du jeu.
- CURSORTILESET ⟶ identifiant permettant de charger les tuiles des curseurs du jeu.
- MAPTILESET⟶ identifiant permettant de charger les tuiles du terrain de jeu.

En fonction de l'identifiant, le chemin du fichier de tuile à charger dans l'objet TileSet est changé. Cett classe possède aussi la taille de chaque tuile du fichier en attribut avec "cellWidth" et "cellHeight".

Classe TextureArea: Cette classe a pour but de définir les différentes couches de Texture à afficher sur le rendu via trois méthodes "loadMap", "loadUnits" et "loadCursor". Ces méthodes prennent en argument principal un pointeur sur l'objet d'état du jeu ainsi qu'un pointeur sur l'objet de TileSet correspondant.
Dans chacune de ces méthodes, un tableau de vertex (qui va contenir des vertex de type quads permettant d'avoir des objects graphiques rectangulaires) permettant de définir la position des Tuiles sur le rendu ainsi que les coordonnées dans le fichier de texture (le tileset) ainsi qu'un attribut de type Texture sont initialisés à l'aide d'une matrice de terrain de jeu (créée à partir d'un fichier .csv dans la classe "EntityMap" de l'espace de travail "state"), d'une liste de personnage issue de la classe "Player" et de la position du curseur issue de la classe "Cursor".

Classe StateLayer: Cette classe contient l'ensemble des informations lié au rendu du jeu. Elle permet dans un premier temps d'instancier les différents TileSet par appel du constructeur. Dans un deuxième temps, elle permet d'instancier via une méthode "initTextureArea" les différentes couches de Texture (couche "map, "units" et "cursor") à afficher sur le rendu du jeu. Enfin, une méthode "draw" de la classe StateLayer permet de tracer ces différentes couches de texture grâce à des méthodes issue de la bibliothèque SFML ainsi que d'autres objets permettant de décorer ou sur des emplacements sur lesquels nous rajouterons des informations pour le joueur plus tard.

Ainsi en instanciant la classe StateLayer dans le fichier “main” et en faisant appel aux méthodes citées précédemment il est possible d’obtenir le rendu du jeu correspondant à l’état du jeu en cours.

3.3 Ressources

FIGURE 5 - Tileset pour la carte

FIGURE 6 - Tileset pour les personnages

FIGURE 7 - Tileset pour le curseur

FIGURE 8 – Logo Fire Emblem

3.4 Exemple de rendu

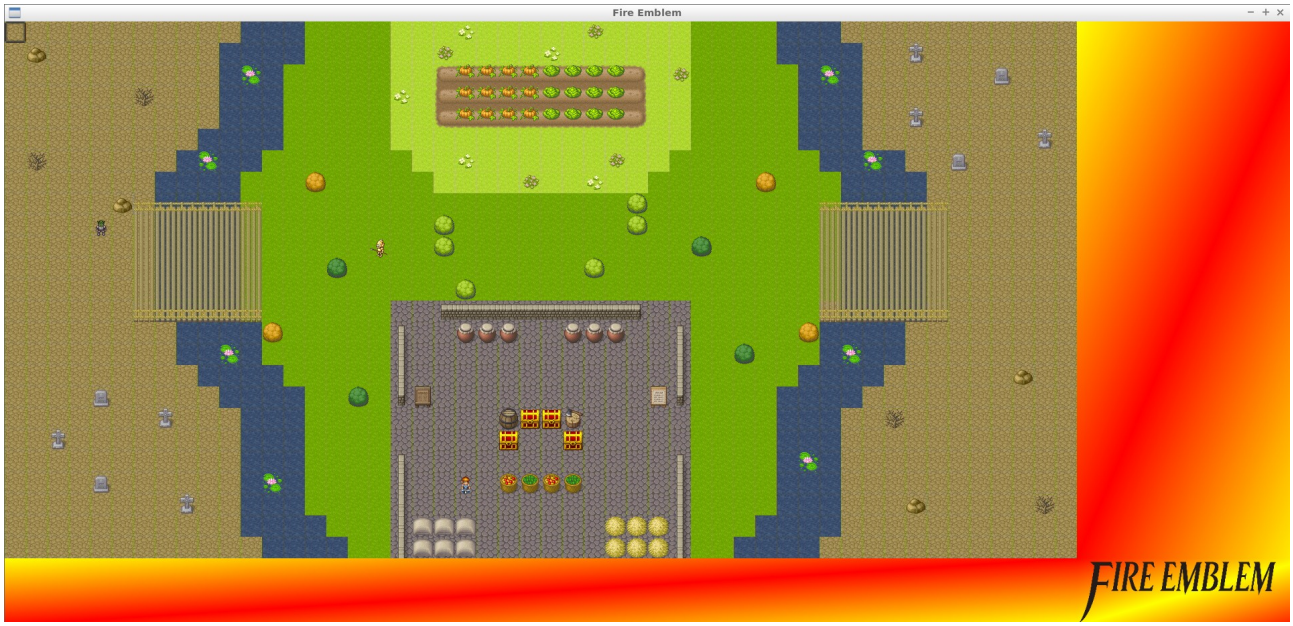

FIGURE 9 - Rendu d'un état

Les zones rectangulaires vides pour l'instant seront utilisés pour afficher des informations sur le personnage sélectionné par exemple.

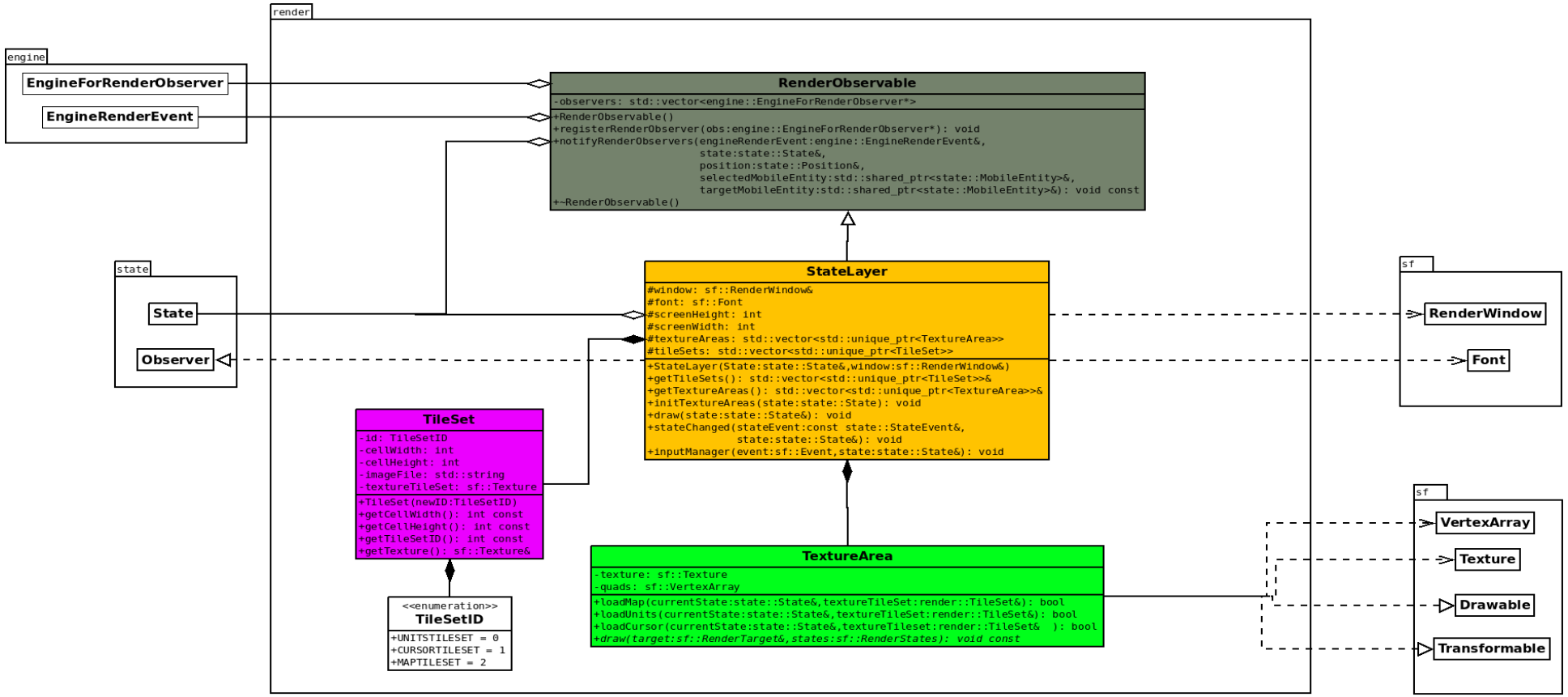

FIGURE 10 - Diagramme des classes de rendu

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

Chaque unité peut effectuer 3 types d'action lorsque son état est AVAILABLE c'est-à-dire disponible. Chacune présente des conditions pour pouvoir être menée à bout :

- faire un déplacement : une unité peut se déplacer d'une case. Pour pouvoir se déplacer dans cette nouvelle case, il doit d'abord vérifier que celle-ci est à sa portée. Ensuite, il faut qu'il reste des points de mouvement à l'unité. Enfin, cette case ne doit pas être un obstacle et ne doit pas être déjà occupée par une autre unité.
- attaquer : l'unité peut infliger des dégâts à une unité ennemie. Pour cela, il doit d'abord vérifier que la cible est dans son champ d'atteinte (dépendant de sa portée d'attaque) et bien sûr que cette unité appartient à un joueur adverse. Une attaque ne peut être suivie d'un déplacement. L'unité passera donc automatiquement dans un état WAITING (pour indiquer que l'unité est maintenant en attente du prochain tour). Si l'unité qui est la cible de l'attaque à ses points de vie à 0 après l'attaque celle-ci passe dans un état DEAD.
- conclure son tour d'action : l'unité peut décider de mettre fin à son tour cela avant même d'avoir effectué tous ses déplacements ou même avant d'avoir fait quelconque autre action. Cette action ne peut être effectuée et est utile seulement dans le cas où une unité n'effectue que des déplacements. L'unité passera dans un été WAITING après cette action.

Pour pouvoir choisir sur quelle unité l'action va être appliquée, il faudra d'abord la sélectionner. L'unité doit donc d'abord être dans un état AVAILABLE puis SELECTED. C'est dans ce dernier état que l'unité pourra vraiment effectuer une action.
Toutes les unités sont initialisées dans un état AVAILABLE au début de chaque tour sauf si elle est morte.

Pour savoir si un joueur a terminé son tour, on vérifie que toutes ses unités sont en WAITING ou DEAD. Le prochain joueur est sélectionné en incrémentant une variable entière permettant de savoir qui est le joueur actif.
Lorsque toutes les unités vivantes quelque soit le joueur auquel ils appartiennent sont en WAITING ou DEAD alors c'est la fin d'un tour de jeu général. Le round est incrémenté et le premier joueur redevient le joueur actif.
Lorsqu'il ne reste des unités vivantes que pour un seul joueur alors la partie est terminée.

Différentes commandes ont été implémentées pour gérer les actions via les touches via SFML. Celles-ci sont décrites en bas à droite de la fenêtre de jeu.

4.2 Conception logiciel

Le diagramme des classes pour le moteur du jeu est décrit sur la figure 11 ci-dessous. Ce diagramme est constitué de deux classes principales et de trois qui héritent de l'une d'entre elles.

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 eniter est associé à chaque action pour pourvoir indiquer son niveau de priorité. Plus le chiffre est bas (au minimum 0) plus la commande est prioritaire.
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 appellée. 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.
Le ''checkRoundStart'' permet de préparer chaque tour. Il remet la valeur du joueur actif pour que celui-ci indique le joueur 1, remet les ''movementLeft'' de chaque unité à sa valeur de base et leur état en AVAILABLE.
''checkRoundEnd'' de son côté va vérifier si le tour d'un joueur est terminé en regardant l'état de toutes ses unités et incrémente le ''' et que la partie est terminée si tous les joueurs sauf un ont toutes leurs unités en état DEAD.

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 Attack, Move et EndEntityRound 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.

La partie Observer de ce diagramme va fonctionner comme pour dans State. Seulement celui-ci permet de faire la communication entre le Render et l'Engine.

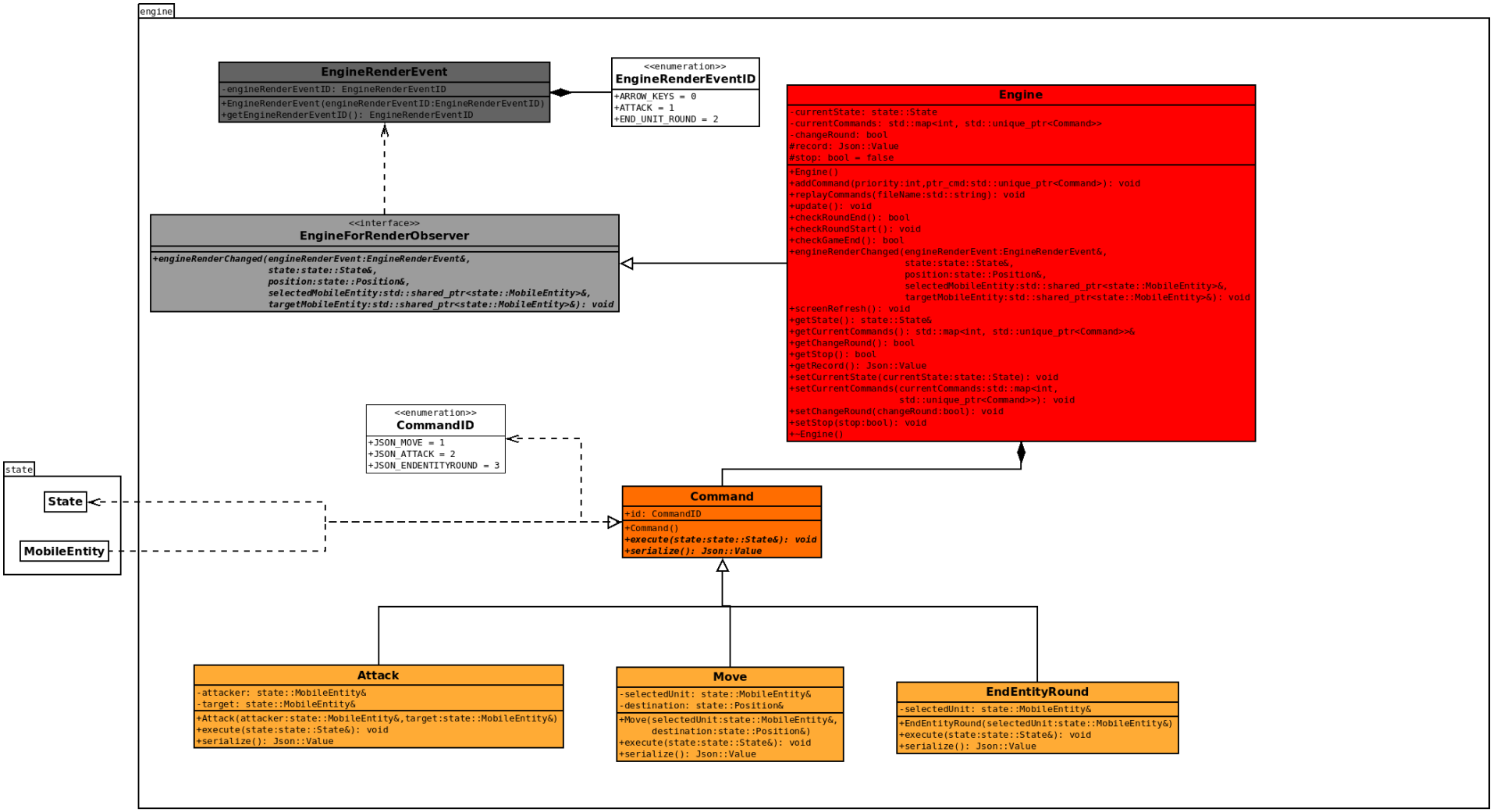

FIGURE 11 - Diagramme des classes du moteur de jeu

5 Intelligence Artificielle

5.1 Stratégies

L'IA sera vu comme un autre joueur auquel les actions seront défines soit aléatoirement soit par l'étude de l'état du jeu selon la complexité de l'intelligence. L'IA se verra donc attribuer un ID qui correspondra à son playerID dans la liste des joueurs.

5.1.1 Intelligence minimale

Dans le cas de l'intelligence minimale les actions seront choisies aléatoirement selon quelques conditions. L'IA va passer par chacune de ses unités encore vivantes et va contrôler l'unité active jusqu'à ce qu'elle est finie son tour (c'est-à-dire qu'elle se trouve à l'état WAITING).
L'IA va d'abord vérifier avec les méthodes à sa disposition si l'unité peut se déplacer ou attaquer et va effectuer un choix aléatoire en fonction d'un nombre tiré aléatoirement selon les différents cas :

- si l'unité ne peut ni bouger, ni attaquer alors il n'a d'autre choix que de terminer son tour
- si l'unité peut encore se déplacer mais ne peut attaquer alors l'IA va décider de soit effectuer un déplacement soit de terminer le tour.
- si l'unité ne peut se déplacer mais peut attaquer alors il fera le choix entre lancer une attaque ou terminer son tour
- sinon toutes les actions sont possibles pour l'unité.

Pour le choix de l'action, un entier va prendre une valeur aléatoire selon les cas avec 0 correspondant au déplacement, 1 à l'attaque et 2 à la fin de tour.

Pour le déplacement, la destination est aussi tirée aléatoirement en fonction du nombre de cases où l'unité peut se déplacer. L'IA sélectionne en fonction de ce chiffre un emplacement dans la liste des cases de destination potentielle.

Le choix de la cible pour l'attaque se fait de la même manière avec un chiffre tirée aléatoirement selon la taille de la liste des cibles possible et selon le résultat la victime de l'attaque est choisie.

5.1.2 Intelligence basée sur des heuristiques

Afin de donner une chance à l'IA de remporter la partie, nous proposons un ensemble d' heuristiques pour offrir un comportement meilleur, aux unités contrôlées par l'IA comparé à un comportement basé sur le hasard.

Pour cela on définit dans un premier temps un ordre de priorité des actions. En effet, l'attaque sera prédominante sur le déplacement et le déplacement sera lui-même prédominant à la fin du tour de l'unité contrôlée.

Au début de chaque tour de chacune des unités contrôlé par l'IA, on vérifie dans un premier temps s'il ya toujours des unités en vie autre que ceux de l'IA. Si ce n'est pas le cas alors le jeu est terminé.

S'il y a des unités ennemies encore en vie alors le jeu peut continuer.

Une fois l'unité sélectionnée, si celle-ci a moins de 10 points de vie alors on lui demande de fuir. Si l'unité a assez de points de vie, on récupère dans un premier temps l'ensemble des positions d'attaque possible pour l'unité actuelle. Dans un second temps, on vérifie s'il y a des unités ennemies dans la zone d'attaque de l'unité contôlé par l'IA. S'il y a plusieurs unité dans la zone d'attaque, on procède à un système de scoring permttant de déterminer l'unité ennemie à attaquer en estimant de manière logique les chances plus ou moins grandes d'une attaque efficace sur celle-ci,
Le système de « scoring » est basé sur les points de vie restants, les capacités défensive et offensive de l'unité contrôlée par l'IA et des unités ennemies se situant dans la zone d'attaque.

Dans le cas où l'unité sélectionnée par l'IA n'a pas de position d'attaque possible ou s'il n'y a pas d'unité ennemie dans la zone d'attaque alors l'IA va se déplacer en suivant l'algorithme A* (algorithme de recherche d'un chemin de coût minimal) vers une cible ennemie (la plus proche). L'algorithme A* va utiliser un système de nœuds et va parcourir les différents cases de la carte jusqu'à ce qu'il atteigne l'objectif (celui-ci étant l'unité ennemie). Pour cela, les nœuds sauvegardent le nœud qui le précède (ce qui permettera de reconstruire le chemin) ainsi que deux valeurs : la distance depuis le départ et une distance estimée par rapport à l'arrivée. En utilisant ces différentes informations, l'algorithme va réussir à déterminer efficacement et surtout rapidement (comparé à un algorithme de Djikstra) un court chemin vers la cible.

Si l'unité n'a plus de points de mouvement ni d'ennemi dans la zone d'attaque alors l'unité termine son tour,

5.1.3 Intelligence basée sur les arbres de recherche

L'intelligence artificielle avancée permet grâce à des méthodes de résolution de problèmes à états finis d'obtenir des commandes optimales par rapport à la méthode heuristique présentée précédemment.

Dans notre cas, l'algorithme min-max nous a paru être une solution abordable permettant d'anticiper les coups possibles de l'intellligence artificielle ainsi que celui du joueur adverse.
L'ensemble des coups possible nous ramènent à différents types de situations ou de « states ». Cependant ces différentes situations n'offrent pas les même chances de gagner pour l'intelligence artificielle.

Une simulation des différentes solutions de commandes possible pour l'IA est faite sur une copie de l'état actuel du moteur sans en modifier le contenu. Ceci a permi de modifier le « state » afin d'évaluer la situation dans le cas où le coup aurait été joué par l'IA.
Ainsi un poids ou un score est associé à chacune de ces situations possible.

Ces poids sont fixés par une fonction d'évaluation de seond dégré qui tient compte des points de vie alliées et adverses ainsi que des unitées restantes chez IA et chez l'adversaire. Cette fonction d'évaluation retourne une liste de valeurs caractérisant le caractère avantageux ou désavantageux de certaines situations pour l'intelligence artificielle.

Ainsi il suffit d'appliquer sur cette liste l'algorithme min-max pour trouver le coup optimal permettant d'augmenter les chances de gagner une partie pour l'IA par prévsion de l'action optimale joué par le joueur,

5.2 Conception logiciel

Le diagramme des classes pour l'intelligence artificielle est décrit sur la figure 12 ci-dessous. Ce diagramme est constitué de deux classes.

Classe AI: Cette classe va être le père de toutes les différentes intelligences artificielles. Elle possède un attribut artificialIntelligenceID qui va correspondre à son playerID et donc son numéro de joueur dans la partie. Elle possède aussi la méthode « run » qui va contenir tout le comportement de chaque AI et qui sera redéfinie dans chacun des fils.

Classe RandomAI: Implémente le comportement décrite dans la partie intelligence minimale en utilsiant les informations de « engine » le moteur de jeu et donc de l'état actuel du jeu notamment graĉe aux méthodes allowedMove et allowedAttack de MobileEntity permettant de savoir si l'unité en question peut se déplacer ou attaquer. Pour réaliser ses actions, cette classe utilise les méthodes et classes définies dans le moteur est les utilisent comme un joueur (Move, Attack et EndEntityRound).

Classe HeuristicAI : Implémenter l'ensemble des comportements heuristiques décrits dans la partie « Intelligence basée sur les heuristiques ». Cette implémentation est possible grâce l'appel à la méthode «run « ». La méthode « attackSuccessScoring » permet de choisir l'unité à attaquer pour s'offrir les meilleurs chances de victoire quand il y a plusieurs unités dans la zone d'attaque de l'unité contrôlée par l'IA. La méthode « algorithmAStar » permet d'implémenter l'algorithme de calcul du plus court chemin (A*) qui est utilisé pour les déplacements de l'unité de l'AI. Les méthodes « nodeInStack » et « minIndex » sont utilisées par l'algorithme A*. Ils permettent respectivement de tester si un objet Node est déjà dans la pile de Node qui doivent être parcourus et de déterminer l'indice du Node avec le distance heuristique la plus faible dans la pile de Node qui doivent être parcourus.

Classe Node: La classe Node est une classe Position évoluée et adaptée pour l'algorithme A*. Celle-ci va contenir l'information sur le nœud précédent par lequel on est passé. De plus, il enregistre la distance qui le sépare avec le nœud de départ (cela en rajoutant 1 à la distance du nœud précédant car dans notre cas tous les déplacements valent la même valeur quelque soit l'emplacement de la grille) ainsi que la distance estimée qui le sépare de l'arrivée.
Ces deux dernières valeurs vont permettre à l'algorithme A* de déterminer quel nœud il va visiter en priorité pour atteindre son objectif et de lui donner son comportement heuristique.

Classe DeepAI : Cette classe va permettre d'implémenter l'IA avancée. La méthode « copyEngine » permet de faire une copie de l'ensemble des attributs de engine à l'intant présent, Cela permet de tester les différentes situations attaingnable par l'IA sans modifier l'état présent. La méthode « optimalMoveCood » utilise l'algorithme « A Star » pour renvoyer les positions de déplacements optimales pour l'IA et « storeMoveCommands » permet d'en fabriquer les commades associées. Les méthodes « atackableEnnemies » et « storeAttackCommands » permettent de fabriquer les commandes d'attaque s'il y a un ennemie dans la zone attaquable de l'IA. La méthode « storeEndActionCommand » permet de fabrique la commande de fin d'action.
La méthode « createChildNode » permet d'intancier un nœuds enfant pour chacune des commandes possibles relié au nœud parent correspondant à l'état présent. On a fisé une profondeur de 2 pour l'arbre utilisé dans la fonction min-max,
La méthode « evalSituation » permet de donner un score aux différentes situations se trouvants sur les feuilles de l'arbre. Les méthode « maximixeScore » et « minimixeScore » permettent d'appliquer l'algorithme min-max sur l'arbre construitprécédamment.
Enfin les méthodes « findOptimalCommandIndex » et « ExecuteOptimalCommand » permettent d'exécuter la meilleur commandes résultant de l'algorithme min-max.

Classe DeepAiNode : Cette classe permet d'instancier les nœuds d'un arbre. Chaque nœud représente une situation possible du state selon les différents commandes pouvant être exécutées par l'Ai au moment présent,

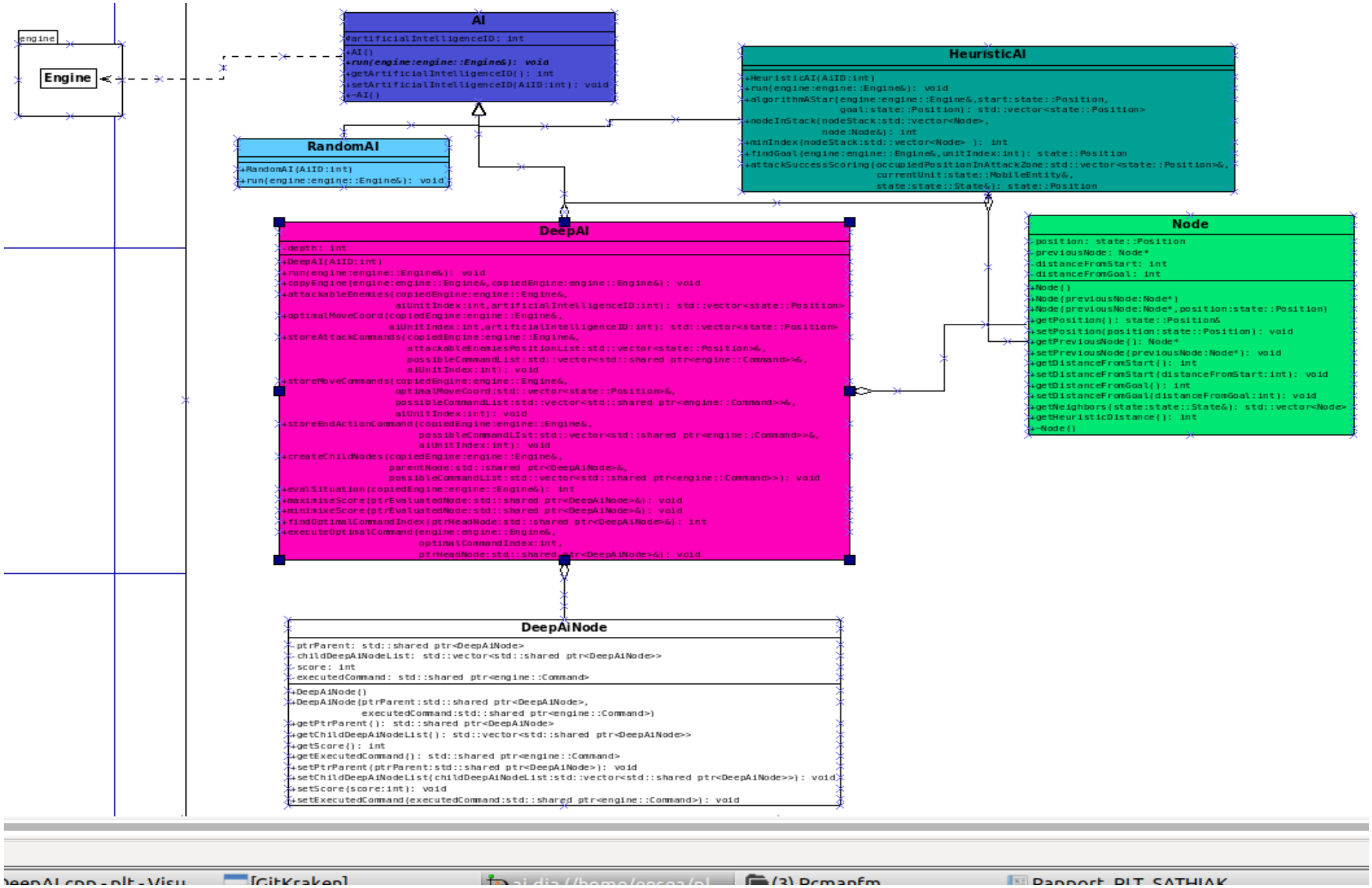

FIGURE 12 - Diagramme des classes de l'IA

6 Modularisation

6.1 Organisation des modules

La client et le serveur sont maintenant séparés. 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 (voir Figure 13).
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 (c'est-à-dire traduite) 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.

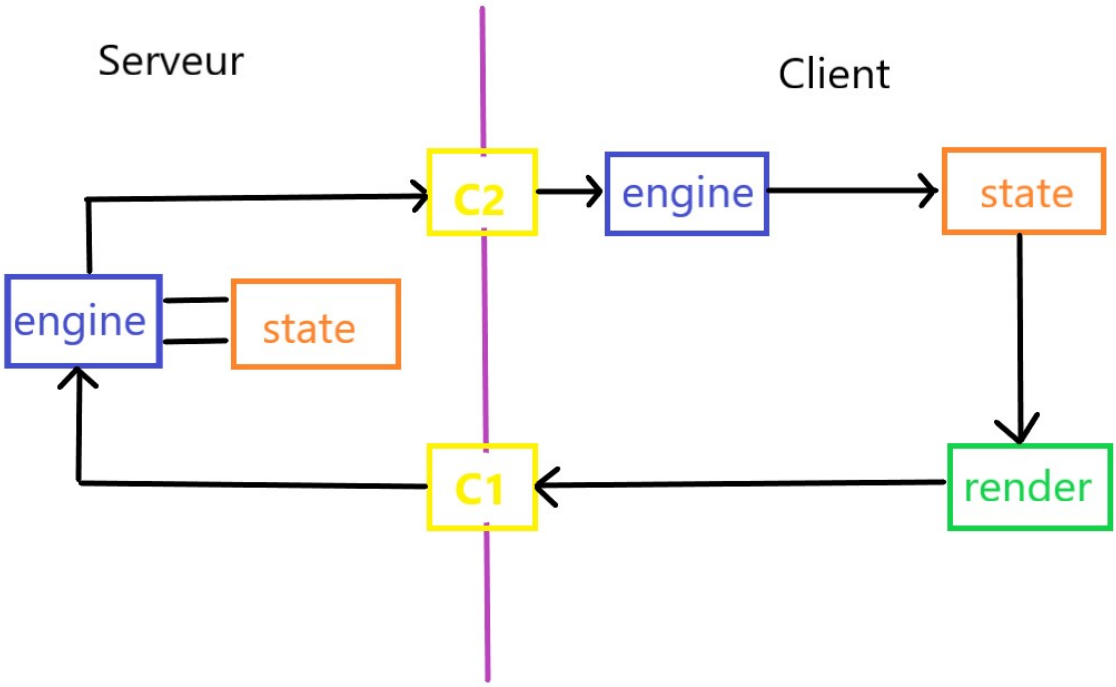

FIGURE 13 – Schéma de l'organisation des modules entre le serveur et le client

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 ». La mise à jour de l'affichage graphique s'opère en fonction des modifications qui ont eu lieu dans state.

6.1.2 Implémentation de l'API et lobby

L'implémentation de l'API se fait du côté du serveur. Nous allons créer pour cela une interface capable de gérer les requêtes classiques en HTTP(GET, POST et PUT). Cette interface sera aussi capable de gérer les erreurs au niveau des requêtes et d'informer le client du mieux possible de la raison pour laquelle celle-ci n'a pas pu obtenir une réponse satisfaisante.

De plus, nous modélisons le lobby par une liste de joueur. Celle-ci sera mis à jour en fonction des requêtes reçues par l'API. Le lobby est limité à un maximum de 2 joueurs puisque pour l'instant notre jeu n'est pas capable d'en accueillir plus. Pour modéliser ce lobby nous aurons une instance de jeu qui contiendra une liste joueurs. Chaque joueur pourra avoir un nom et sera attribué un identifiant selon l'ordre d'arrivée en connexion au serveur.

6.2 Conception logiciel

6.2.1 De la modularisation

La fonction « thread() » lance un client et un serveur séparément. La fonction « runRecord() » permet de tester l'enregistrement des commandes sur une partie entre deux IA heuristiques. 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 « runPlay() » qui va permettre cette fois de tester l'exécution des commandes à partir des informations enregistrées au format Json.

Classe ModularisationTest : 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 « runRecord() » 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 « runPlay() » permettant ainsi de reproduire à l'identique les actions enregistrées.

6.2.2 API et lobby

Pour cette partie, nous avons d'une partie le client qui est modélisé par la classe ServerTest dans la figure 14 Dans une autre partie, nous avons l'API et le lobby du côté serveur qui est décomposée en plusieurs classes dans la figure 15.

Classe ServerTest: Cette classe va permettre de modéliser un client simple pour l'instant. Sa fonction run va d'abord demander à l'utilsateur d'entrer son nom de joueur. Lorsque le nom a été entré, une requête sera envoyé au serveur pour pouvoir enregistrer ce nouveau joueur. La liste des joueurs apparaît après qu'il y ait eu une réponse du serveur. Le joueur peut ensuite se déconnecter en appuyant sur le bouton 'd' puis entrer. Une nouvelle requête sera envoyée au serveur et lorsque la réponse de cette deuxième requête sera obtenue, la nouvelle liste des joueurs sera affichée.

Classe Game: Game représente le lobby qui va accueillir les joueurs. Celui-ci possède une liste pour stocker les joueurs ainsi qu'un identifiant pour savoir le nombre de joueurs qui s'est connecté. Ses méthodes lui permettent de gérer la liste des joueurs en fonction de ce qui a été reçu par l'API (« getPlayer() », « setPlayer() », « addPlayer » ou encore « removePlayer »).

Classe Player: Modélise tout simplement les joueurs avec leur nom ainsi qu'un booléen permettant de connaître leur disponibilité.

Classe AbstractService: C'est le format générique de l'API qui pourra être déclinée selon les besoins d'utilisation. On pourra définir différents services à partir de cette classe. Elle met en place tous les prototypes pour les requêtes HTTP classiques (GET, POST, PUT).

Classe PlayerService: L'API qui est adaptée à la gestion des joueurs pour la connection au lobby. C'est celle-ci qui va faire appel aux fonctions de « Game » pour pouvoir modifier la liste des joueurs et changer l'identifiant indiquant le nombre de joueurs qui se sont connectés à ce serveur.

Classe ServicesManager: Cette classe contient tous les services de l'API qui ont été mis en place et qui font le lien entre le serveur et le client.

Classe ServiceException: Permet de lancer différentes erreurs au niveau des requêtes HTTP pour mieux informer l'utilisateur sur l'origine du problème. L'énumération « HttpStatus » fait la liste de toutes ces possiblités.

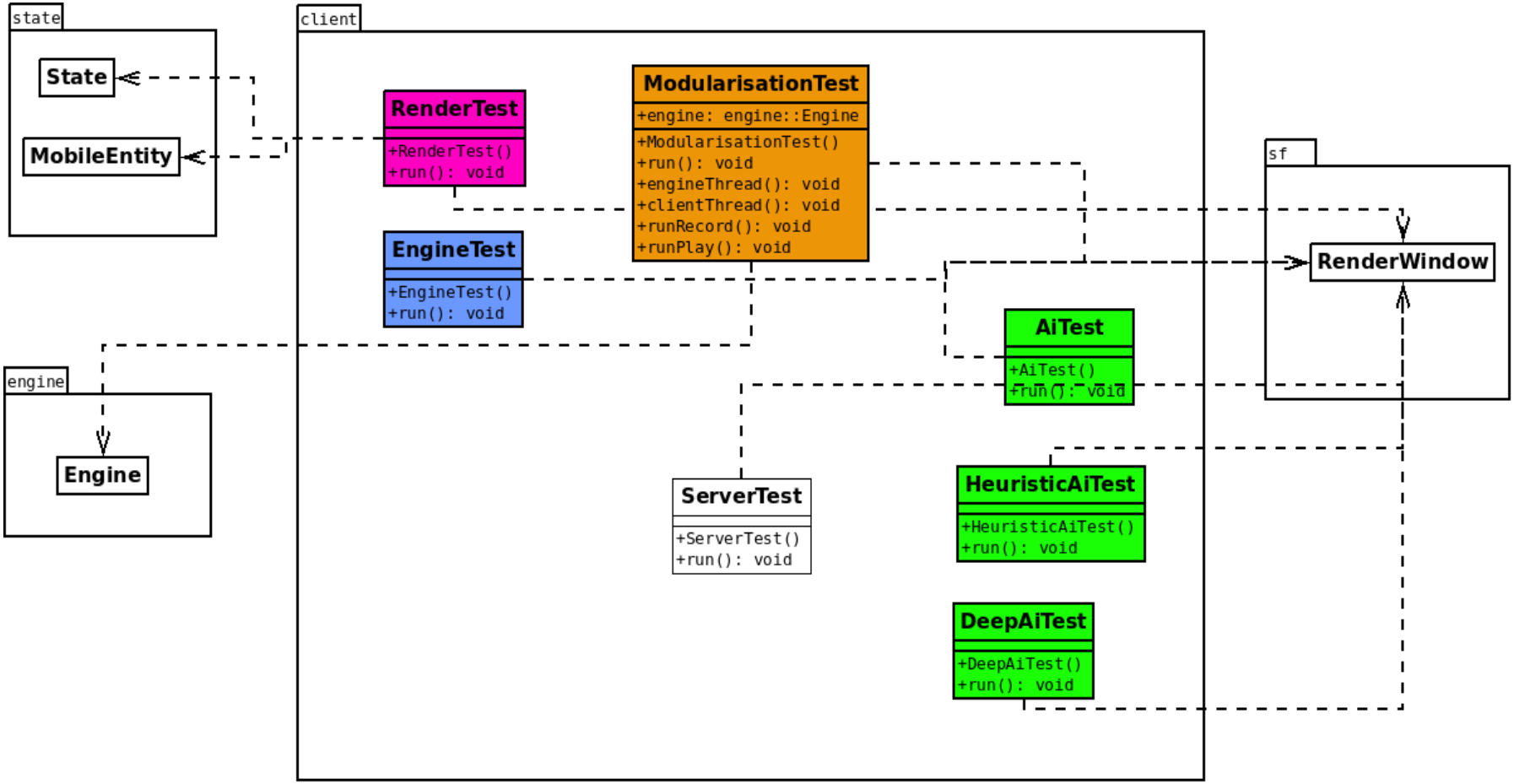

FIGURE 14 - Diagramme de classes pour la modularisation

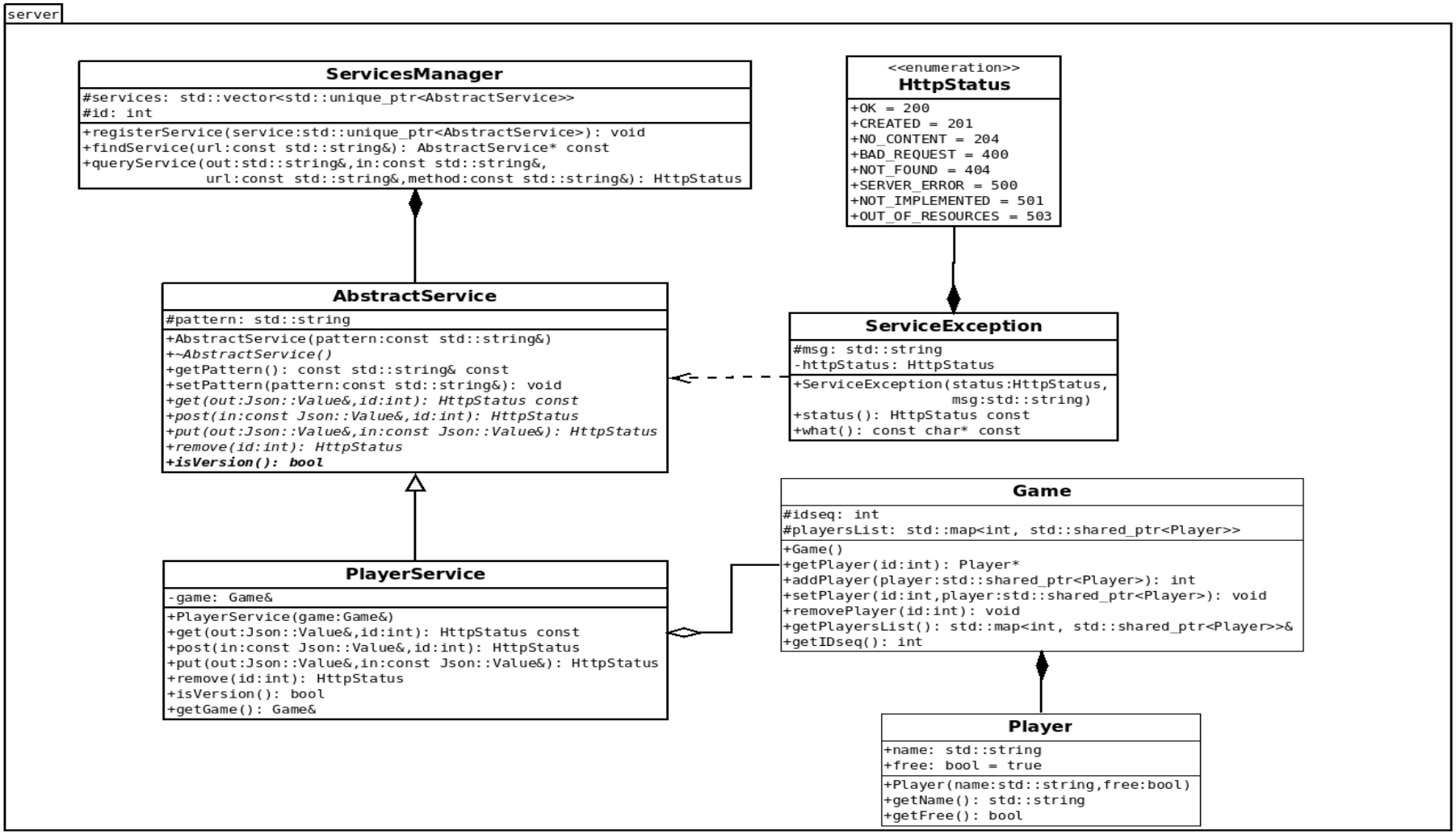

FIGURE 15 - Diagramme de classes pour le serveur