Projet Logiciel Transversal

Anand Candassamy & Paul Estano

Table des matières

1 Présentation Générale

1.1 Archétype

Notre jeu s'inspirera principalement du jeu *Pokemon Donjon Mystère*. En effet, nous avons prévu de conserver le mécanisme des combats et de donjon de ce jeu.
Dans notre logiciel l'utilisateur incarnera un pokemon dans les salles d'un donjon qui contiennent chacune des pokemons qui peuvent l'"agresser". Le nombre de pokemons dans une salle évolue en fonction de l'avancement du joueur dans le jeu.
Pour simplifier le jeu nous abandonnons également d'évolution des pokemons.

1.2 Règles du jeu

Le donjon contient un nombre fini de salles et le joueur gagne lorsqu'il sort de la dernière salle du donjon. Un donjon a 10 salles, le pattern de chaque étage n'est pas tiré aléatoirement. Le joueur a la possibilité d'accéder à des étages bonus, dîtes arène de combat où il pourra se battre contre un autre joueur en ligne.
Le joueur est provoqué en duel automatiquement par les pokemons qui sont autour de lui. Il faut qu'il tue l'IA ou le joueur adverse pour activer la case de l'esaclier qui leur permet de passer à l'étage suivant.
Les combats fonctionnent en tour par tour. A chaque tour, l'utilisateur peut effectuer une action :

- attaquer avec une compétence
- soigner le pokemon
- se déplacer d'une case

1.3 Ressources

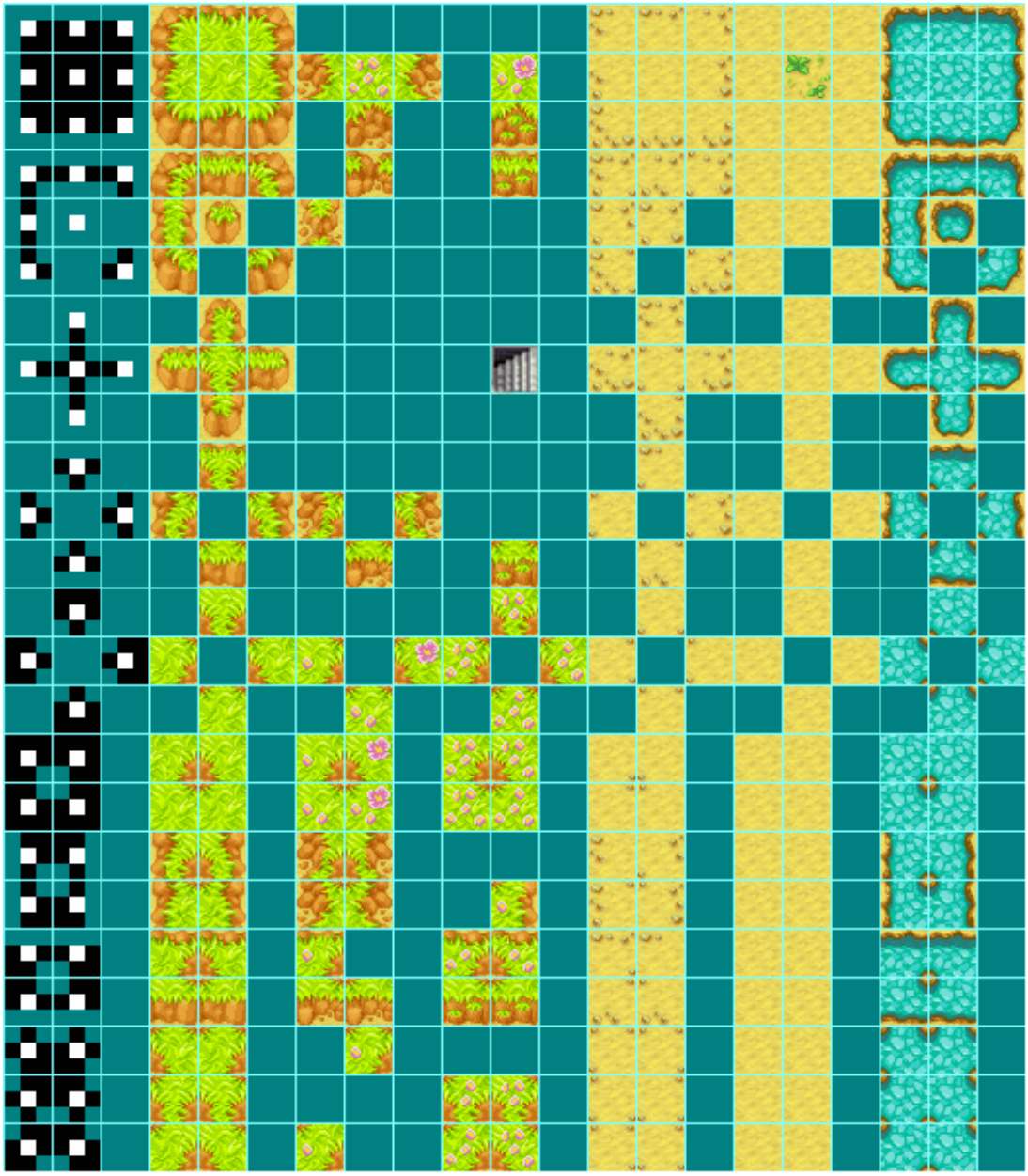

FIGURE 1 – Tileset utilisé pour construire un monde

FIGURE 2 – Tileset utilisé pour les pokémons

2 Description et conception des états

2.1 Description des états

Un état du jeu est formé d'une carte qui est statique au cours du temps et d'une liste de joueurs dont l'état varie au cours du jeu.

2.1.1 État de la carte

La carte est un élément fixe de l'état du jeu. Il se décompose en plusieurs cases.
Les cases se distinguent en 3 parties :
— les murs, qui ne sont pas franchissables par les éléments mobiles.
— les cases "conteneurs", elles peuvent être vide ou occupé par un seul élément mobile.
— les cases escaliers, il en existe un seul par étage, qui permet d'aller au niveau suivant du labyrinthe.

2.1.2 État des joueurs

Un pokemon est soit contrôlé par un joueur soit contrôlé par une Intelligence Artificielle (I.A), il possède une position, un nombre de points de vie à l'état t, un nom et un ensemble de statistiques qui lui sont attribués en début de partie. Le pokemon est considéré comme étant "mort" si son nombre de points de vie atteint 0.

2.1.3 État général

A l'ensemble des éléments statiques et mobiles, nous rajoutons les propriétés suivantes : — Époque : représente « l'heure » correspondant à l'état, ie c'est le nombre de « tour » passés globale depuis le depuis de la partie.

Ces statistiques correspondent aux attaques qu'il peut utiliser, à son nombre de point de vie en début de partie et son type (Salameche, Bulbizarre, Carapuce).

2.2 Conception Logiciel

Le package état peut se diviser en trois sous-partie :
— Une partie gérant les personnages, en bleu
— Une partie gérant l'environnement, en rouge
— Une classe représentant l'état global du jeu, en vert

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

Pour notre implémentation du jeu, nous avons prévu d'utiliser que les trois pokémon suivant :
— Carapuce
— Salamèche
— Bulbizarre

Le constructeur respectif de chacun de ces trois pokemons héritent de la classe Pokemon. Nous avons mis en place un identifiant pour chaque pokemon, ce qui permet d'avoir plusieurs pokemon construit à partir de la même classe.

Chaque pokemon possède une orientation (*SOUTH, NORTH, WEST ou EST*), et un nombre de points de vie maximum qui dépend de son type (*SALAMECHE, CARAPUCE, BULBIZARRE*) .

La partie environnement est calqué sur le modèle proposé par les fichiers exportés par le logiciel *Tiled Map Editor* au format JSON. Les objets les plus consommateurs en mémoire tels les *data* dans la classe *Layer* sont par ailleurs stockés dans le tas pour éviter les copies entre les différentes classes.

L'état global contient un pointeur vers l'état de l'environnement et et une liste de pointeur de contenant l'état des différents joueurs de la partie. Un booléen permettant de détecter la fin du jeu a également été ajouté.

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 2 types d'*Event* :

— Les *TAB_EVENT* qui représente un changement d'état lié à la position des pokemons sur la carte (absent si mort, changement de position , changement d'orientation)

— Les *STATE_EVENT* qui correspondent à des changement liés à l'environnement (changement de niveau) ou à des changements liéés aux statistiques de la partie

ditemie Afin d'éviter des problèmes techniques dans notre implémentation, nous avons mis en place des fonctions afin de capturer les exceptions rencontrées lors de l'exécution, principalement dans la classe qui charge les données de la carte. De plus nous avons utilisé des entiers non signés afin que les attributs des pokemons ne prennent jamais de valeurs négatives.

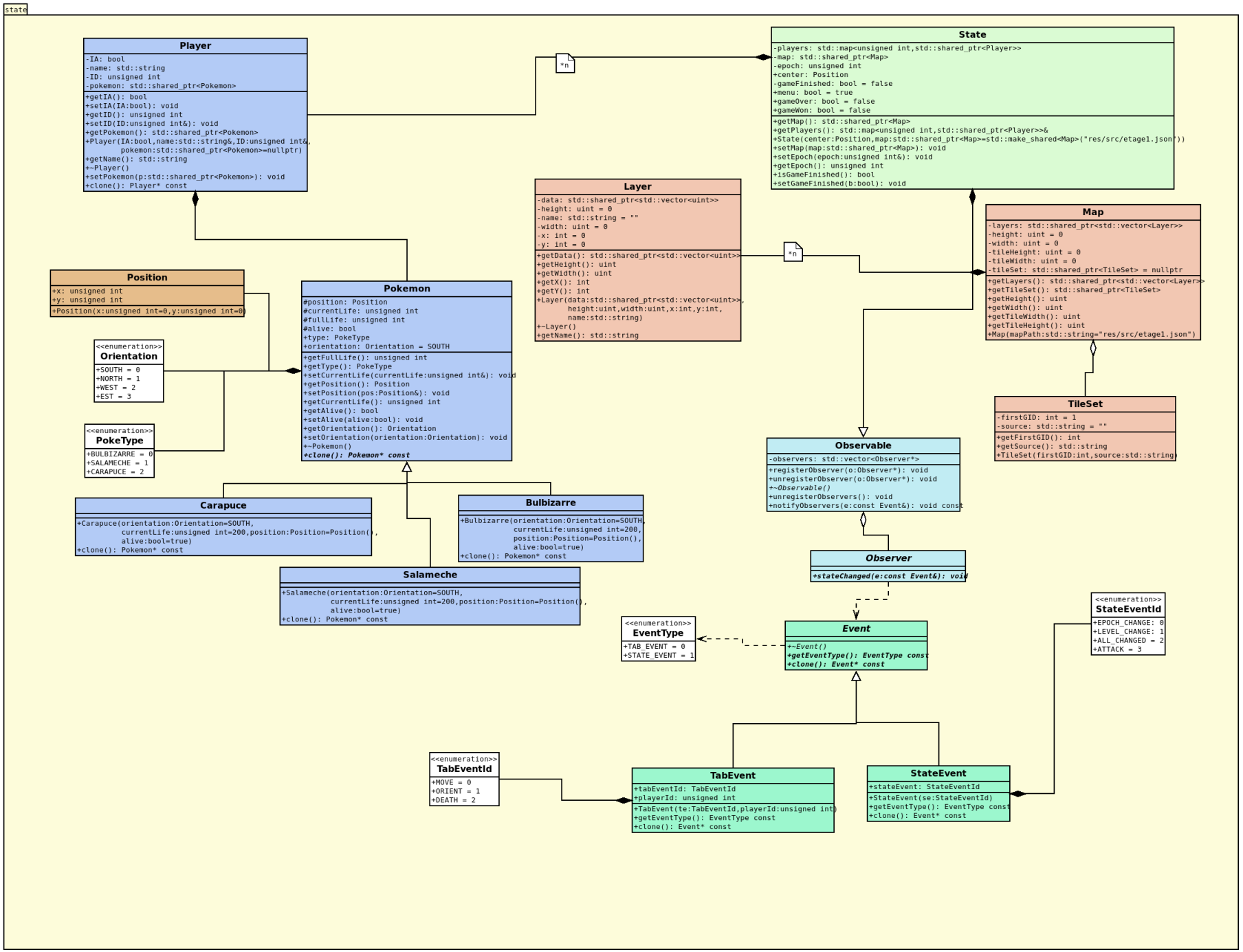

FIGURE 3 – Diagramme des classes d'état.

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, la carte avec les éléments décoratifs (mur, sol, escalier) et les éléments mobiles à savoir les pokemons (Carapuce, Salamèche, Bulbizarre) qui se superposent sur la couche précédente. On transmet l'état du jeu ainsi que les textures de toute la carte avec leurs coordonnées et la texture des pokemons avec leurs coordonnées et leur orientation à afficher.

La carte étant très grande, nous l'avons entièrement chargé dans la mémoire pour l'affichage et on a créé une vue (un zoom) centrée sur le pokemon du joueur qui évolue sur la carte, car le but du jeu reste avant tout d'évoluer dans un labyrinthe.

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 pokemons, seule le rendu des pokemons 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.

Nous avons rajouté sur la scène finale, certains attributs des pokemons (point de vie actuelle et id des pokemon) en haut à gauche.

Pour simplifier au mieux notre logiciel, nous n'avons pas inclus les animations de déplacements lorsque le pokemon évolue d'une case à l'autre ainsi que les animations d'attaques quand un pokemon lance une compétence sur un autre.

3.2 Conception logiciel

Pour afficher un état on crée une scène qui génère un ensemble de pointeurs sur des objets graphiques de types *LayerRender* pour la carte. Pour les Pokemon, on instancie un *Sprite* par Pokemon. Ce *Sprite* contient à la fois les coordonnées du Pokemon sur la fenêtre et sa position sur le Tileset utilisé. Puis, on utilise la méthode *Scene : :draw()* pour déclencher l'ouverture de la fenêtre et l'affichage de l'état. Il est à noter que l'état contient d'ores et déjà toutes les informations nécessaires à l'affichage de la carte. En effet, à la création de l'état celui-ci parse un fichier JSON (ici *map.json*)contenant toutes les informations nécessaires à l'affichage de la carte.

classe *Scene* est un des *Observer* de *State*. Ainsi, lorsque l'état subit un changement, *Scene* est notifié et met à jour le rendu en fonction du type d'évènement qui lui est transmis :

- Si un *TAB_EVENT* (correspondant à changement d'état pour l'un des joueurs) la méthode *updatePlayers()* responsable de la mise à jour du rendu des pokemon est appelée.
- Si un *STATE_EVENT* (correspondant changement d'état lié à l'environnement ou à la partie) les méthodes *updatePlayers()* et *updateMap()*.

Par ailleurs, les points de vie de chacun des pokemon sont, pour le moment, affichés et mis à jour en temps réel à l'intérieur de la méthode *draw()* de la classe *Scene*.

Chaque objet *LayerRender* parcourt l'ensemble des tuiles d'un étage de la carte, détecte la position de la tuile sur la ressource graphique (i.e : l'image du tileset) et calcule sa position sur la fenêtre. Pour optimiser les performances seule l'image du tileset est chargée dans une *Texture*, l'objet *LayerRender* conserve seulement la position de chaque tuile de la carte sur cette texture. Les pokemons sont chacun rendus à l'intérieur d'un *Sprite*. Lorsqu'un mouvement de l'un des pokemons est détecté le la classe scène met à jour l'affichage de l'ensemble des pokemons.

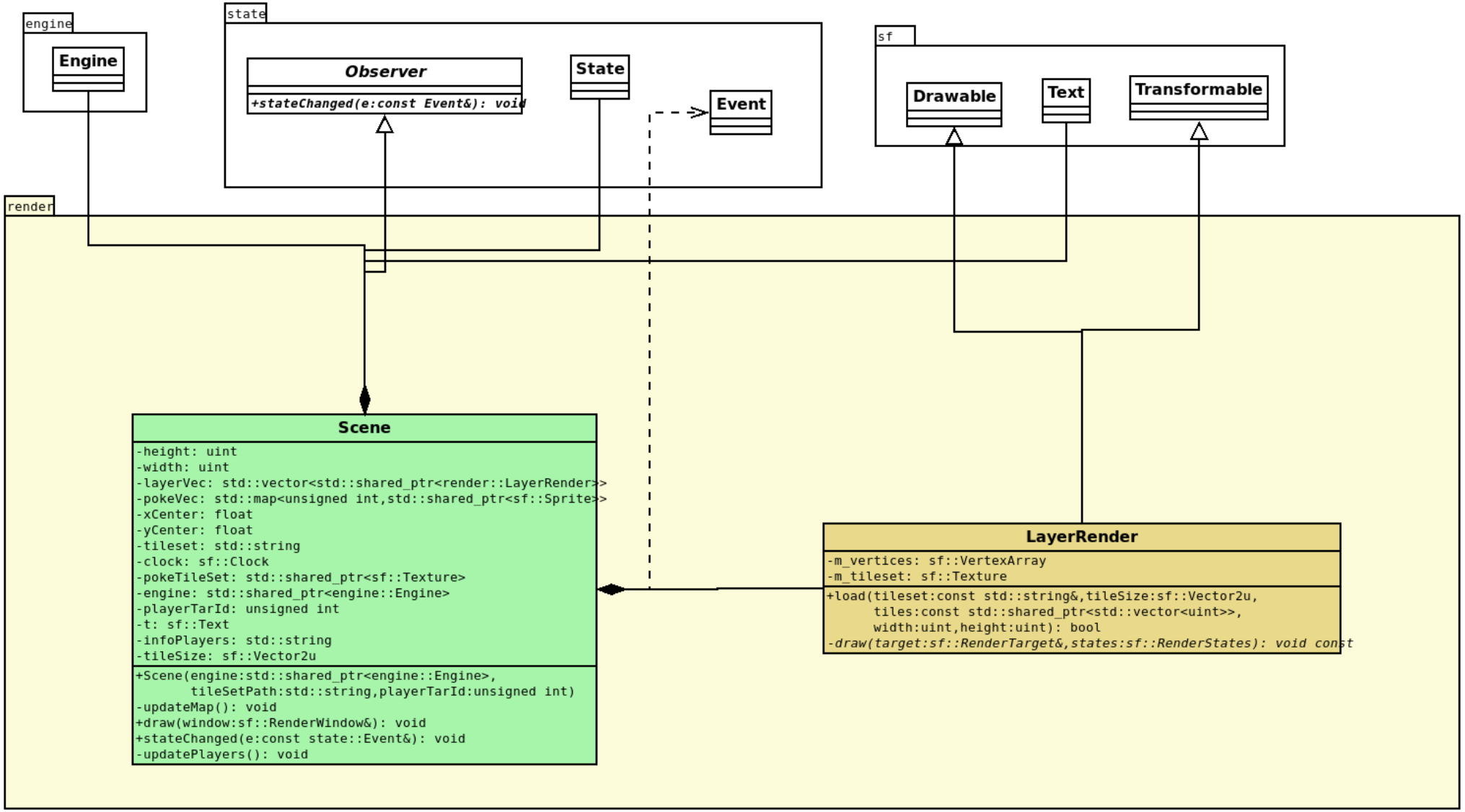

FIGURE 4 – Diagramme des classes de rendu.

4 Règles de changement d'états et moteur de jeu

4.1 Règles

Les changements d'états sont liés aux commandes exécutées par le moteur. Les commandes extérieures sont :

— Commande de déplacement
— Commande d'attaque
— Commande de soin

Une fois qu'une commande extérieure de la part du joueur et de l'IA ont été exécutés (joueur puis IA), on passe au tour suivant.

4.2 Changements autonomes

Les changements autonomes interviennent à la fin de chaque changement d'état lié à une commande extérieure et s'exécutent dans l'ordre suivant :

1. Si le joueur est mort, on affiche "GAME OVER"
2. Si l'IA est morte, on active la case escalier pour passer à l'étage suivante
3. On met à jour les statistiques (point de vie) de l'IA et du joueur en fonction des règles d'attaque et de soin.
4. On applique les règles de déplacement du joueur et de l'IA.
5. Si le joueur est sur la case d'escalier, on vérifie s'il est autoriser à passer au niveau suivant.
6. Si le joueur a gagné on affiche "GAME WON"

4.3 Conception logiciel

On utilise ici le pattern *Command* : un ensemble d'utilisateur peut ajouter des "commandes" à l'objet *Engine* puis lui demander de les exécuter à l'aide de la méthode *runCommands()*.

Les commandes hérite de la classe abstraite *Command* contient une méthode *execute()* virtual pure qui est appelée par le moteur lorsqu'un utilisateur appelle la méthode *runCommands()*. Cette méthode applique en fait directement des changements à l'objet *State* conservé par la classe *Engine*.

On a implémenté, pour le moment, 3 types de commande :

— *MoveCommand* qui correspond à un changement de position d'un des pokemon

— *AttackCommand* qui correspond à l'attaque d'un pokemon

— *HealCommand* qui est déclenché lorsqu'un joueur veut soigner son pokemon

Les trois commandes *MoveCommand*, *AttackCommand* et *HeakCommand* sont déclenchés sur un évènement de lecture de clavier, plus précisement sur un évènement d'appui sur une touche de clavier pour plus de fluidité lors des déplacements dans le parcours du labyrinthe.

La commande *MoveCommand* met à jour la position ou l'orientation du pokemon sur la map.

Les commandes *AttackCommand* et *HeakCommand* agissent sur l'état des pokemon et sur le rendu de la scène en gardant à jour les points de vie actuels affichés en haut à gauche de la scène.

Par ailleurs, le changement de niveau étant provoqué par un déplacement sur la case escalier après qu'un joueur (non IA) ait éliminé tous ses ennemis ; cet évènement est entièrement géré et intégré à la commande *MoveCommand*.

Lors de l'exécution de chaque commande, on crée un objet associé à la commande qui permet de sauvegarder des données importante de l'état du jeu comme la vie, la position et l'orientation d'un pokémon. Ces données sont stockées dans des objets de type *PreviousState* et eux même stockés dans une *stack*, ce qui permet d'annuler les commandes exécutées précédemment en dépilant un à un les éléments contenus dans cette *stack* grâce à la méthode *undoCommands()*.

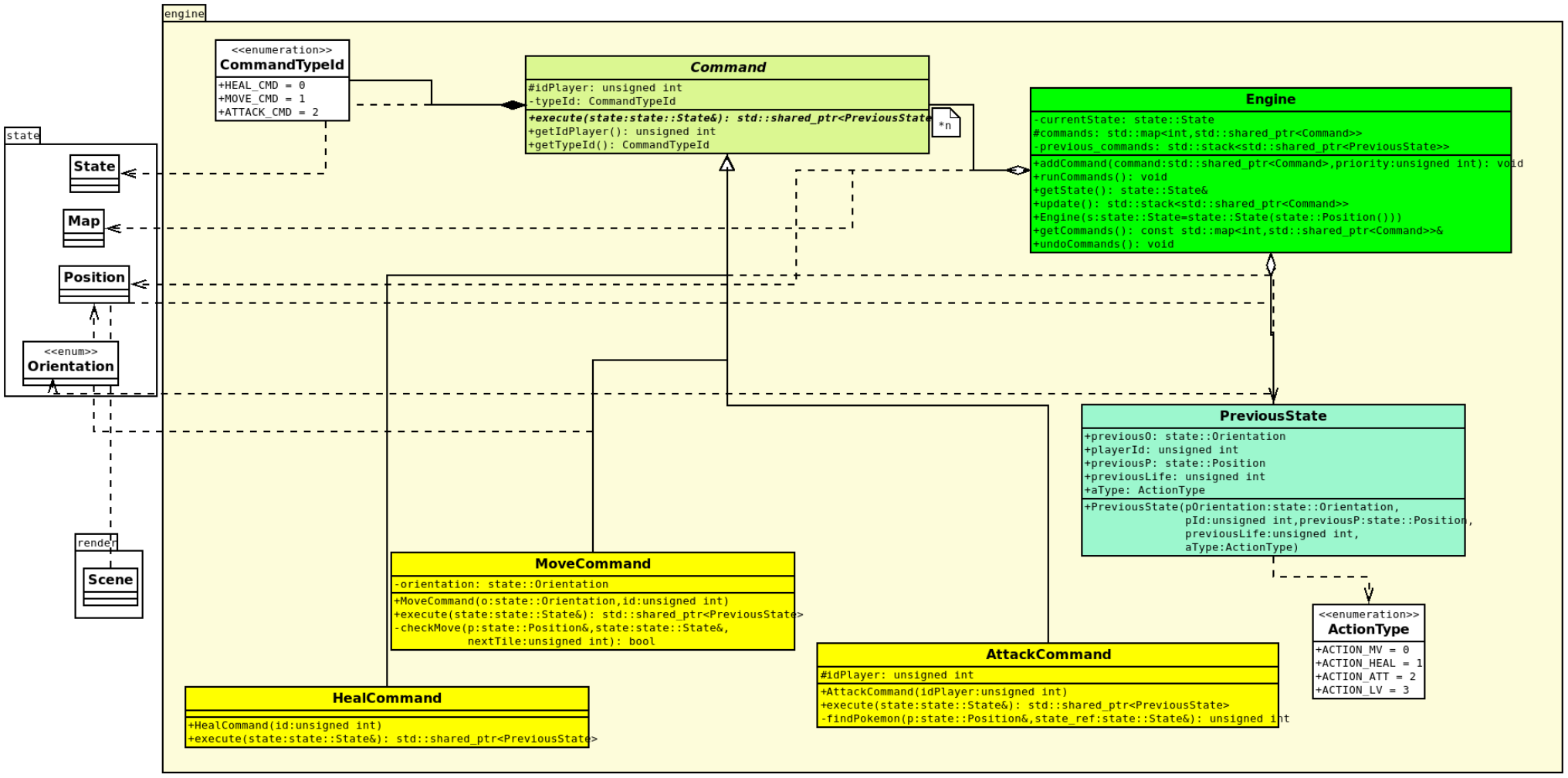

FIGURE 5 – Diagramme des classes de moteur de jeu.

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

Le stratégie de l'AI aléatoire est de se déplacer de manière aléatoire sur la map, lors qu'il rencontre un ennemi, il peut alors combattre et lorsque ses pv tombent bas, il se soigne.

5.1.2 Intelligence basé sur les heuristiques

Nous avons ajouté des heuristiques aux pokémons ennemies afin qu'il puisse battre le pokémon du joueur :

1. Si le pokémon a un nombre de points de vie supérieur à celui de son adversaire, il l'attaque si il est en face de lui.
2. Si le pokemon a un nombre de points de vie supérieure ou égale à celui de son adversaire et qu'il est à distance de celui-ci, il se rapproche.
3. Si le pokémon a un nombre de points de vie trop faible pour gagner le combat, il tentera de fuir son adversaire si celui-ci peut l'attaquer.
4. Si le pokémon ne risque pas de perdre des points de vie suite à une attaque ennemie et qu'il lui en manque, il tentera de se soigner.

Et finalement pour que le pokémon ennemi se rapproche de celui du joueur, nous avons utilisé l'algorithme A* qui est un algorithme de recherche de chemin dans une graphe (ici notre carte). Nous avons notamment utilisé le code disponible ici : https ://github.com/daancode/a-star

5.1.3 Intelligence avancée

Nous proposons en dernier lieu une intelligence avancée basée sur l'algorithme minmax à deux joueurs. Dans notre implémentation, l'IA avancée va explorer toutes les issues possibles et va tenter de maximiser ses chances de victoire face au joueur. Pour cela, nous avons implémenté une fonction de coût :

$$cost = Life_{AI} - D_{AI/Player} - Life_{Player} \tag{1}$$

Cette fonction exprime trois comportements de l'IA avancée :

1. le fait de se déplacer quand il ne voit pas d'ennemi autour, caractérisé par la distance entre l'IA et le Joueur.
2. le fait d'attaquer l'ennemi lorsqu'il est en position de force, qui résulte de la différence de points de vie entre l'IA et le Joueur.
3. le fait de s'enfuir et de se soigner lorsqu'il est en position de faiblesse qui résulte lui aussi de la différence de points de vie entre l'IA et le Joueur.

Ainsi à chaque tour, l'IA avancée va choisir le coup optimal parmi l'ensemble des actions qu'il peut effectuer et qui va maximiser ses chances de gagner au tour suivant grâce à cette fonction de coût où l'un des trois comportements va être mis en avant par rapport aux deux autres. Pour cela, l'algorithme va parcourir l'ensemble des issues possibles à chaque tour. Pour effectuer cette tâche, on a choisi de dupliquer la partie de l'état nécessaire au bon fonctionnement de l'algorithme (état des joueurs et des pokemons) à chaque noeud de l'arbre de recherche.
Pour ce qui est du déplacement sur la carte, nous avons gardé le même algorithme que celui de l'heuristique, à savoir le A *, afin que l'IA puisse se diriger et s'orienter vers le joueur sur la carte de façon optimum.

5.2 Conception logiciel

Le diagramme des classes d'AI est affiché ci-dessous
Classes AI. Les classes filles de la classe AI implémente 3 stratégie d'IA suivi par le pokemon adverse :

1. RandomAI : Intelligence Artificielle aléatoire
2. HeuristicAI : Intelligence Artificielle Heuristique
3. DeeoAI : Intelligence Artificielle avancée

AStar Generator :
Malheureusement, le code provenant d'un projet extérieur, nous n'avons pas pu intégrer les classes du AStar dans notre fichier *ai.dia*. Pour utiliser le générateur on doit tout d'abord les fournir une version "alléger" de l'environnement (i.e. de la classe *Map*) contenant seulement les obstacles de la carte.
AIUtils :
Nous avons également séparé plusieurs méthodes utilitaires consommées par HeuristicAI (et à l'avenir par DeepAI) dans une classe *AIUtils*. Ses méthodes étant indépendantes, elles sont déclarées comme *static* ce qui permet de les utiliser sans instancier d'objet superflus. La méthode flee permet notamment de calculer le meilleurs déplacement possible pour fuir un adversaire.
DeepAI :
C'est une IA basée sur la résolution de problème à état finis. On utilise l'algorithme de parcours d'abres minmax. Ici on a un pokemon contre un autre. Les critères de "jugement" de l'IA sont :

- Sa vie qu'elle veut maximiser
- La vide de l'ennemi qu'elle veut minimiser
- Sa distance à l'ennemi qu'elle veut minimiser

Malheureusement, l'environnement étant relativement grand il est impossible pour l'IA de parcourir l'arbre de recherche en profondeur à chaque coup. En effet chaque noeud de l'arbre possède 4 branches (fuir, se soigner, attaquer, se rapprocher) et la carte, bien que possédant des murs et des obstacles, contient 25x25 = 625 cases. Ainsi, on comprend qu'il est impossible de parcourir l'ensemble de l'arbre dans un temps acceptable. Dans la pratique, l'algorithme parcourt au maximum 4 niveaux de l'arbre. Malheureusement, cela ne semble pas suffisant pour obtenir une intelligence plus performante que l'IA Heuristique. Pour améliorer la performance de l'algorithme nous pourrions mettre en place des conditions pour le parcours des noeuds de l'arbre de recherche pour ainsi éviter le parcours de noeuds inutiles (algorithme alpha bêta).

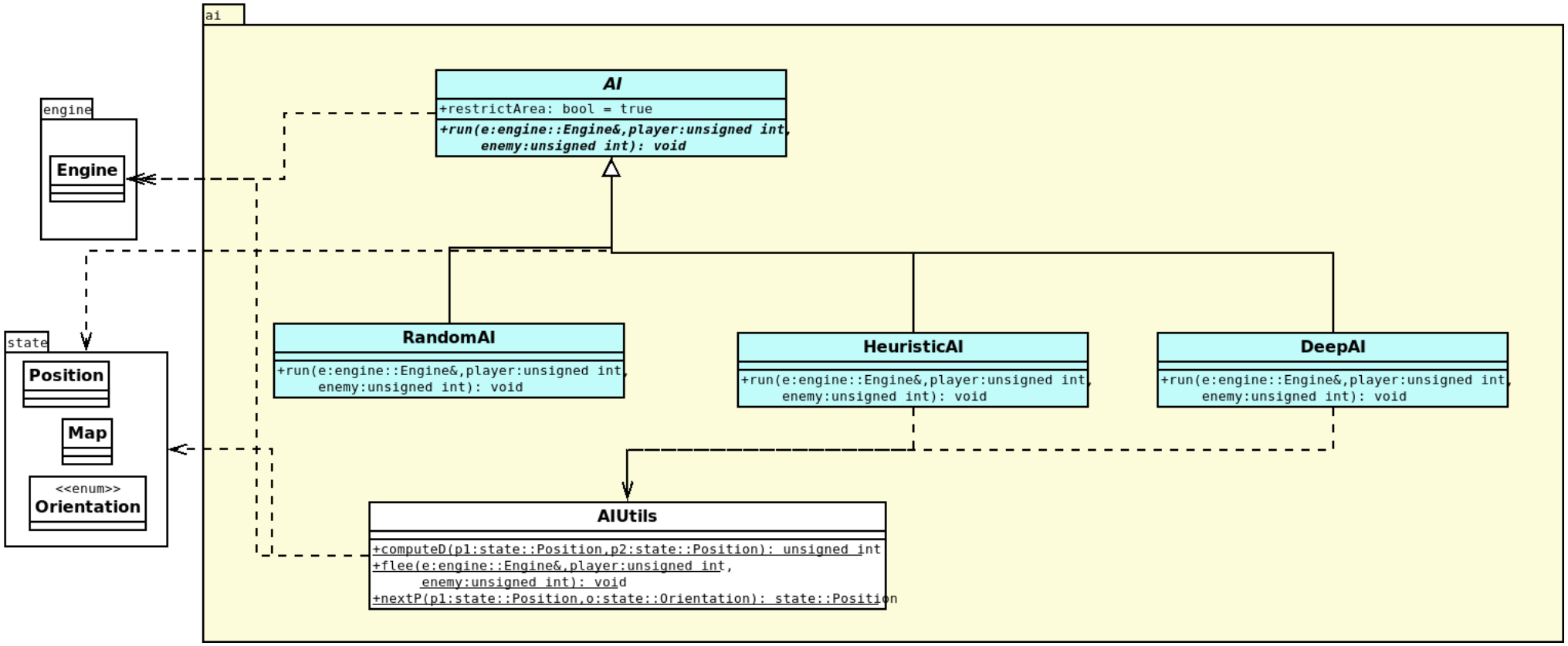

FIGURE 6 – Diagramme des classes d'intelligence artificielle.

6 Modularisation

6.1 Organisation des modules

On veut ici lancer le moteur et le rendu dans deux threads différents. Le thread principal sera celui du rendu (contrainte matérielle) et le secondaire celui du moteur. Deux types de données transitent entre le moteur et le rendu :

— Les commandes, qui transistent du rendu vers le moteur
— Les évènements notifiant le rendu d'un changement d'état

6.1.1 Commandes

Les commandes peuvent arriver à tout moment et l'ajout d'une commande à la map contenant les commandes lors d'une mise à jour de l'état de jeu pourrait engendrer des conflits. Pour palier à ce problème, on utilise un double buffer : lorsqu'une mise à jour de l'état a lieu, on vide la map dans un buffer, que l'on va ensuite utiliser pour exécuter l'ensemble des commandes. Pour empêcher l'ajout d'une commande dans la map lors de la copie dans le buffer, on utilise un mutex (*commands_mutex*) qui bloque l'exécution de la méthode *addCommand* pendant toute la durée de la copie. Le temps de copie restant négligeable, l'utilisateur peut ajouter des commandes aussi rapidement qu'il le souhaite presque sans aucune latence.

6.1.2 Notifications de rendu

Lors d'une mise à jour, l'état notifie le rendu par le biais d'*events*. Pour le moment, le traitement de ces notifications bloque le rendu des Pokemons et de l'environnement et inversement, le rendu des pokemons et de l'environnement bloque le traitement des notifications. Ce mécanisme de verrou est implémenté à l'aide d'un mutex et est coûteux en termes de performances. Pour remédier à ce problème on pourra à l'avenir utiliser une liste stockant l'ensemble des notifications ainsi qu'un signal permettant de signaler la présence d'un évènement qui doit être traité au rendu. Ainsi, lorsque le rendu a fini sa dernière tâche et que le signal est actif, il vide la liste, traite les notifications et réarme le signal. Ce mécanisme devrait permettre d'améliorer notablement les performances.

6.2 Répartition sur différentes machines : mode joueur contre joueur

Nous avons crée une base de données d'utilisateurs sur notre serveur. Pour gérer les données qui transistent entre les clients et notre serveur, nous formons des services CRUD sur la donnée "user" via une API Web REST :

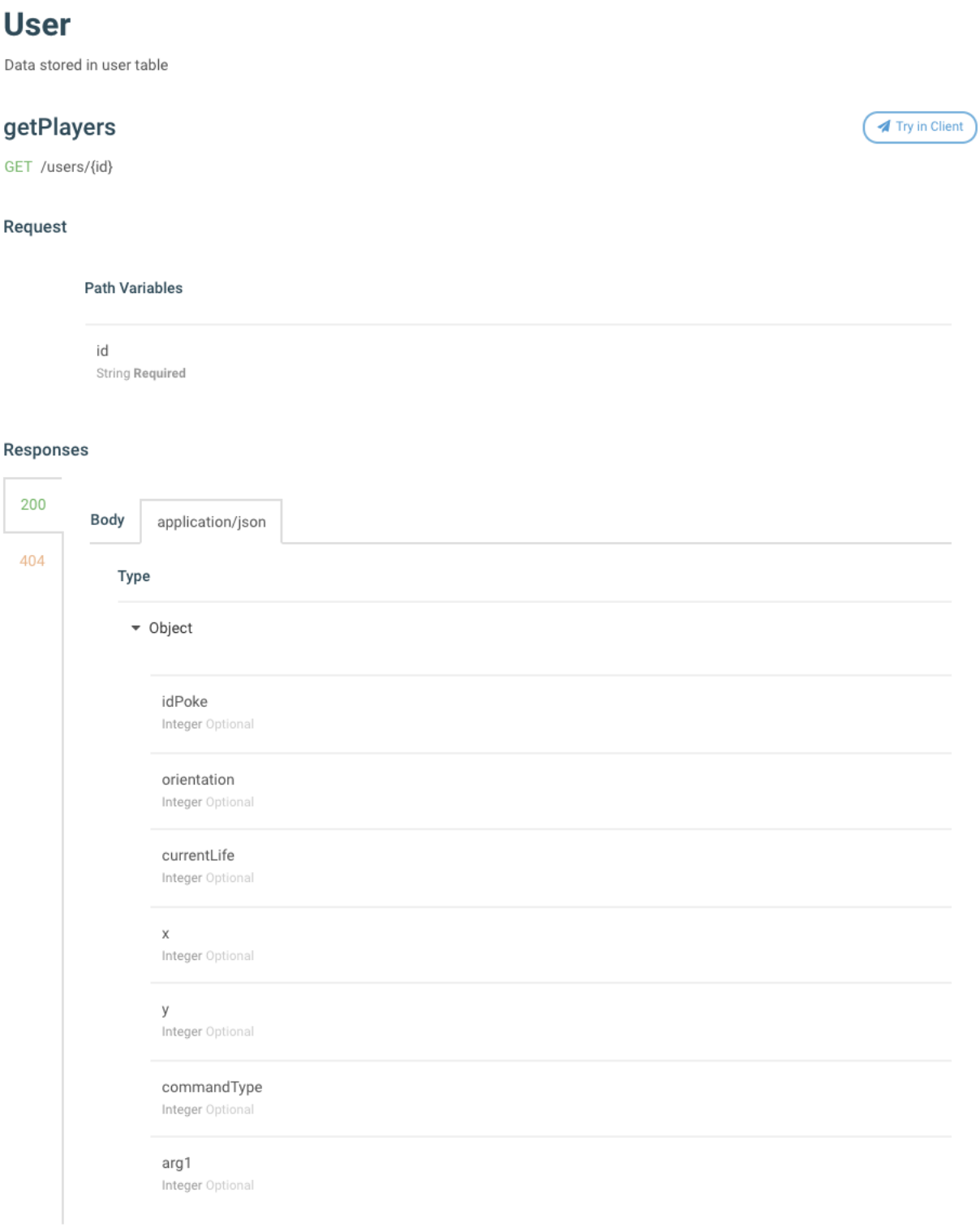

FIGURE 7 – Méthode GET

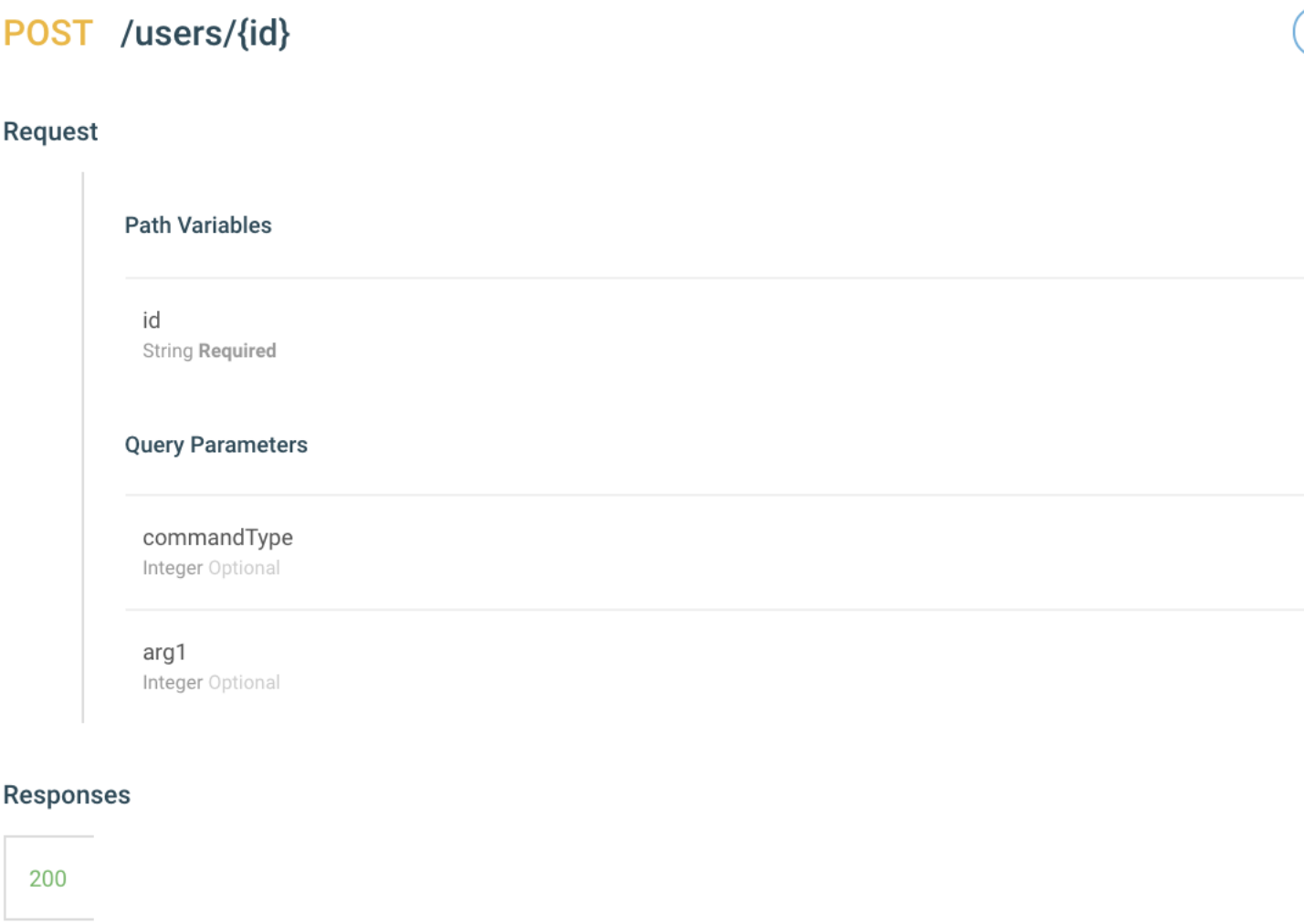

FIGURE 8 – Méthode POST

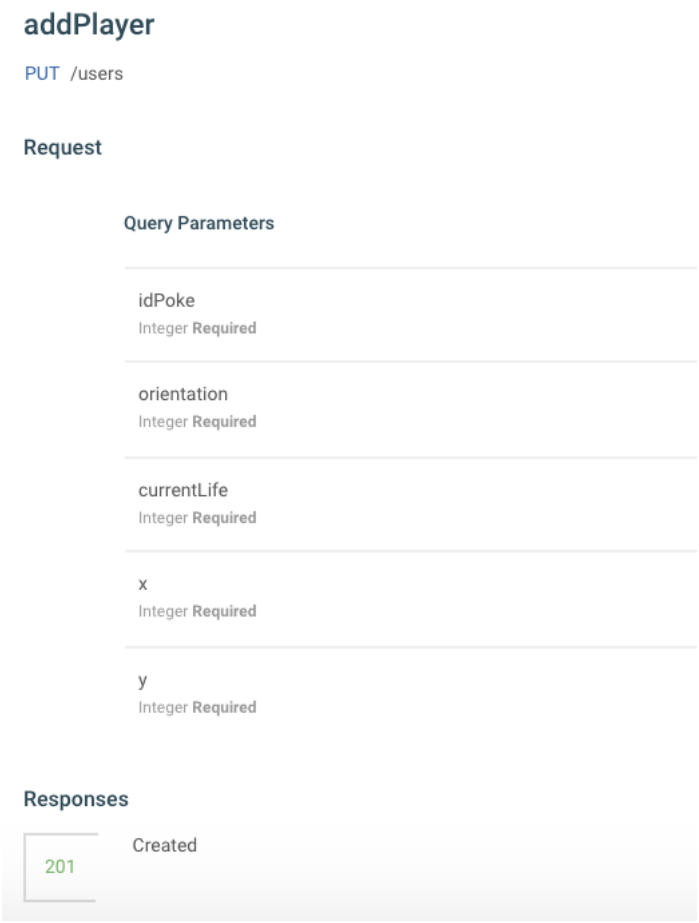

FIGURE 9 – Méthode PUT

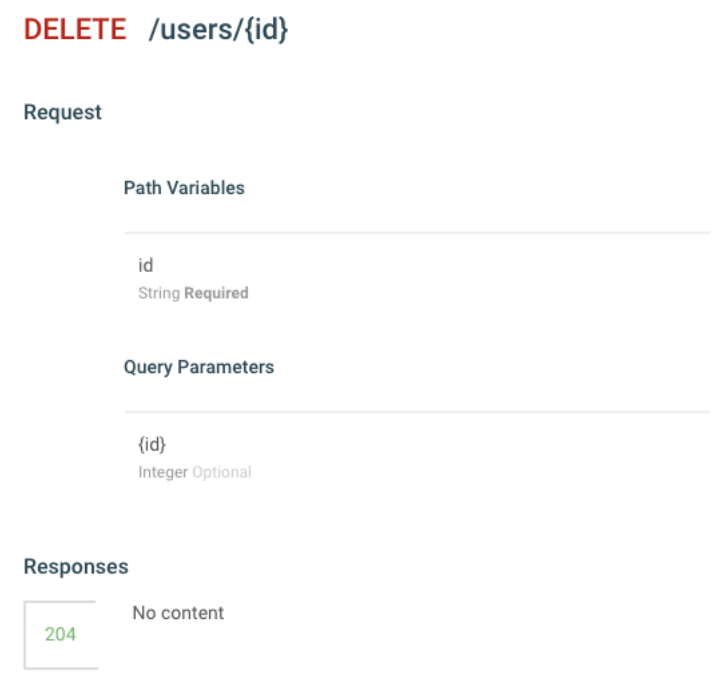

FIGURE 10 – Méthode DELETE

6.3 Conception logiciel

Le diagramme des classes pour la modularisation est présenté en Figure 7. *Client* La classe Client contient le moteur de jeu, les intelligences artificielles et les rendus. Elle utilise ces différents éléments dans deux threads distincts. Dans le thread principal gérant l'affichage on procède selon l'ordre suivant :

1. La méthode *handleInputs* gère l'ajout des commandes au moteur lorsqu'un utilisateur appuie sur un touche.
2. On affiche le rendu à l'aide de la méthode *Scene : :draw*

Dans le thread secondaire on gère principalement les IA et le moteur. En particulier, on utilise l'IA pour ajouter les commandes automatisé au moteur avant d'éxecuter la méthode *Engine : :runCommands* qui applique les changements d'état. La méthode *Client : :run* permet d'exécuter le client. Les services REST implémentés dans le serveur héritent tous de la classe abstraite *AbstractService*. La classe *ServiceManager* permet de gérer l'ensemble des services disponibles sur le server :

— elle permet d'ajouter de nouveaux services REST grace à la méthode *registerService*
— d'accéder à la méthode GET du service spécifié (i.e. en fonction de son url) en paramètre de la méthode *findService*
— d'accéder à une méthode d'un des services en fonction des paramètres du requête reçue par le serveur grâce à la méthode *queryService*

Pour l'instant nous utilisons deux services :

— Un service *UserService* qui permet de gérer les utilisateurs du jeu
— Un service *VersionService* qui renvoie la version du serveur.

Plusieurs méthodes ont été rajoutés à *Client* afin de pouvoir envoyer des requêtes à notre API :

— *addPlayer*, qui envoie une requête PUT sur le serveur pour ajouter un utilisateur, et qui renvoie l'idPlayer que le serveur lui a attribué.
— *getPlayer*, qui envoie une requête GET sur le serveur afin d'afficher les données de l'utilisateur dont l'id a été passé en paramètre de la fonction.
— *addCommand*, qui envoie une requête POST sur le serveur pour envoyer la commande actuelle joué par le joueur.
— *getCommand*, qui envoie une requête GET sur le serveur pour demander la dernière commande joué par un joueur
— *getTour*, qui envoie une requête GET sur le serveur pour demander le numéro du tour de jeu actuelle.
— *deletePlayer*, qui envoie une requête DELETE sur le serveur pour retirer le joueur dont l'id a été passé en paramètre de la liste des joueurs connectés à notre serveur.

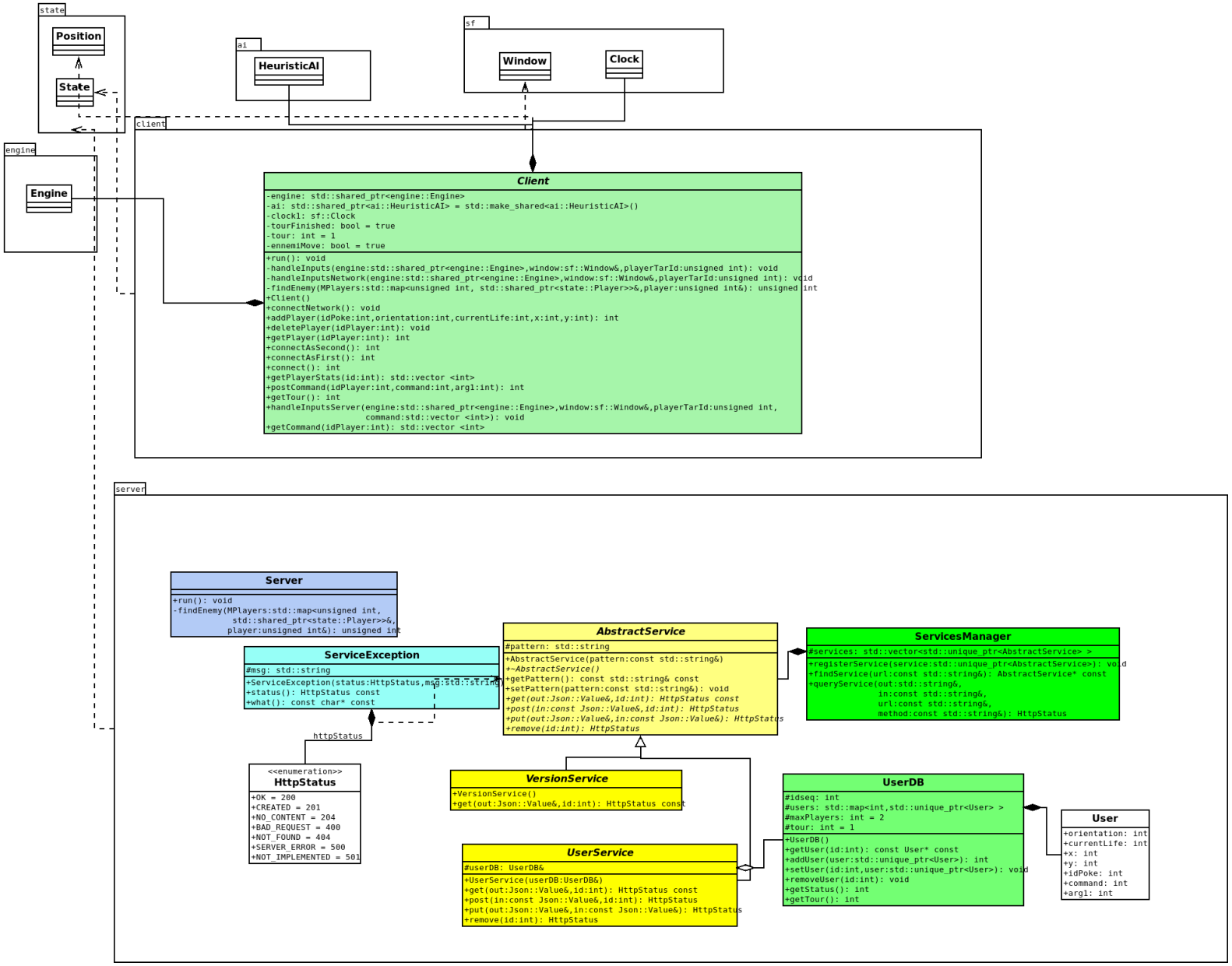

FIGURE 11 – Diagramme des classes pour la modularisation.

7 Annexe

7.1 Bibliographie