

Projet

Attigui Youness - Quellier Louis

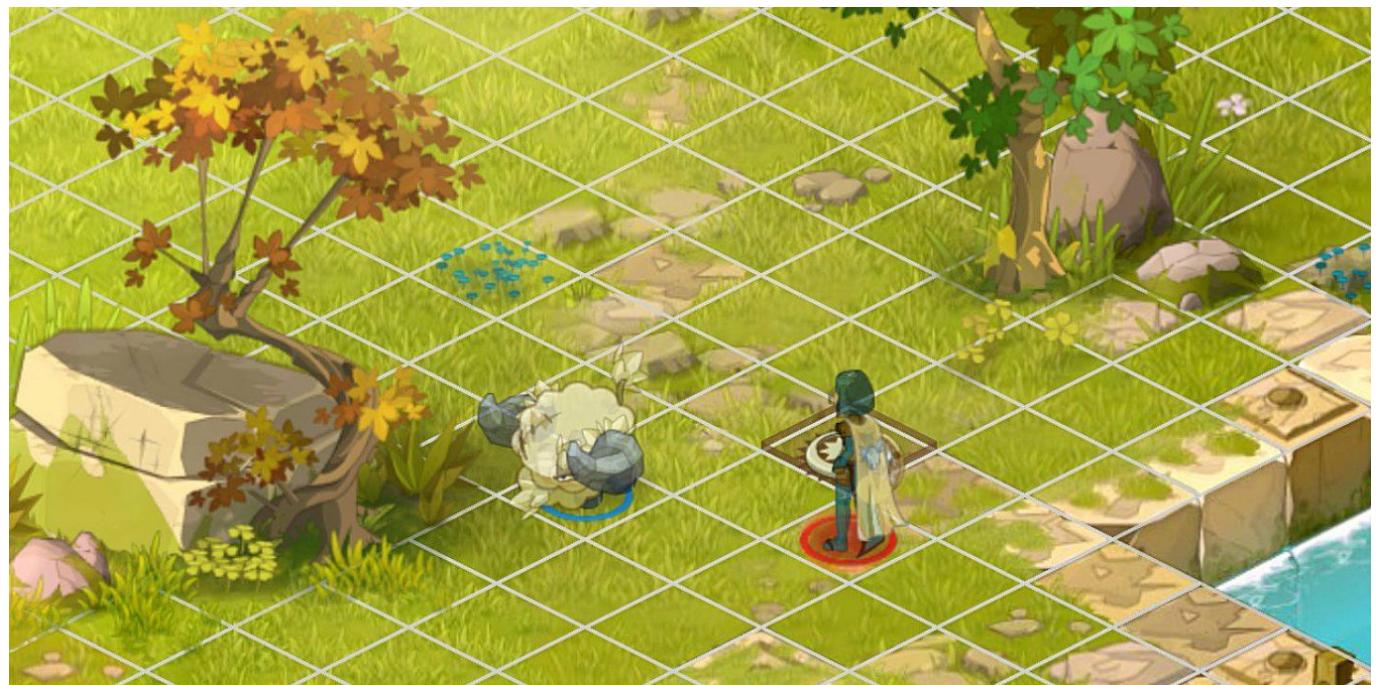


FIGURE 1 – Exemple du jeu

Table des matières

1 Présentation Générale	1
1.1 Archétype	1
1.2 Règles du jeu	1
1.3 Ressources	2
2 Description et conception des états	7
2.1 Description des états	7
2.1.1 State	7
2.1.2 Cell	7
2.1.3 Character	7
2.1.4 Team	7
2.1.5 Inventory	8
2.1.6 Weapon	8
2.1.7 Ability	8
2.1.8 Fight	8
2.1.9 MainQuest	8
2.2 Conception Logiciel	8
3 Rendu : Stratégie et Conception	11
3.1 Stratégie de rendu d'un état	11
3.2 Conception logiciel	11
4 Règles de changement d'états et moteur de jeu	15
4.1 Horloge globale	15
4.2 Changements extérieurs	15
4.3 Changements autonomes	15

4.4	Conception logiciel	15
5	Intelligence Artificielle	17
5.1	Stratégies	17
5.1.1	Intelligence aléatoire	17
5.1.2	Intelligence basée sur des heuristiques	17
5.1.3	Intelligence avancée	17
5.2	Conception logiciel	18
6	Modularisation	20
6.1	Organisation des modules	20
6.2	Conception logiciel	21

1 Présentation Générale

1.1 Archétype

Pour ce projet nous nous inspirons du gameplay du jeu "Dofus". Il s'agit d'un jeu de stratégie au tour par tour où il est possible de se déplacer de carte en carte pour combattre des ennemis et gagner de l'équipement et de l'expérience permettant d'améliorer son personnage et d'avancer plus loin dans le monde.

1.2 Règles du jeu

Vous incarnez un héros auto-proclamé qui vient de fonder sa guilde d'aventurier dans le but de libérer le royaume de l'emprise du roi démon.

L'aventure commence avec une arme basique au choix, combiné avec un élément de prédilection. Ces armes pourront être trouvées ou achetées et gagneront des niveaux selon le nombres d'ennemis vaincus, débloquant ainsi des bonus de statistiques ainsi que des nouvelles actions, pouvant également être améliorées. Ce niveau est inhérent à l'arme et sera alors conservé lors du transfert vers un autre personnage.

Tous les personnages du jeu possèdent en plus de leur arme un ensemble de caractéristiques propres à chacun :

- pv : point de vie (s'ils tombent à 0, le personnage meurt et lâche son matériel sur la case sur laquelle il se trouvait)
- pa : point d'action (permettent d'utiliser les actions associées aux armes)
- pm : point de mouvement (permettent de se déplacer)
- éléments : eau/terre-feu/air

Les différents éléments (l'eau, la terre, le feu et l'air) font partie intégrante de ce monde et sont au coeur des stratégies de combat. Des bonus et malus de type élémentaire sont appliqués selon l'attribut du joueur, de son arme et de la case du terrain sur lequel il est placé. C'est à vous d'utiliser le terrain et vos compétences à votre avantage.

Au cours de l'histoire, d'autres aventuriers rejoindront votre guilde afin de vous aider dans votre tâche et vous serez engagés pour effectuer des missions variées afin de gagner de l'or pour acheter de nouvelles armes ou recruter de nouveaux mercenaires.

En tant que guilde indépendante vous êtes libres de vous déplacer à votre guise dans le royaume pour accomplir les missions dans l'ordre que vous souhaitez. Néanmoins, dans ce monde il n'existe pas d'organisme régissant les missions, celles-ci vous sont confiées lorsque vous arrivez sur les lieux.

Par moments vous devrez défendre une ville contre une attaque adverse, parfois vous devrez protéger un voyageur surpris par une embuscade ennemie ou souhaitant une escorte, d'autres fois vous devrez détruire une base fortifiée mais le plus souvent vous devrez simplement éliminer tous les adversaires présents dans une zone.

Un combat se déroule sur un des terrains de la carte du monde. Chaque terrain est composé d'un nombre fixe de cases. Chaque case est associé à un élément (eau, terre, feu, air, neutre). Le déploiement des personnages s'effectue au début du combat sur une zones prédéterminée dépendant de la direction d'arrivée, la disposition à l'intérieur de cette zone étant libre.

Au cours de votre aventure votre guilde acquiert une certaine notoriété dans le royaume, celle-ci peut aussi bien être positive que négative, influant sur vos interactions avec les différents personnages. Au fil de vos rencontres, vous pourrez refuser certaines missions pour diverses raisons mais cela fera diminuer votre réputation. Le degré de réussite ou d'échec des missions acceptées modifiera également cette réputation.

Si votre guilde a trop mauvaise réputation, vous serez exécutés en tant que criminel et l'aventure s'arrête là. Au contraire une très bonne réputation vous donne le privilège d'accéder à l'armurerie de Vulc contenant des armes surpuissantes mais également excessivement chères...

L'aventure se termine après avoir vaincu l'un des 4 boss élémentaires, vous permettant débloquer son arme associée. Vous pouvez cependant perdre la partie si tous vos personnages meurent ou via le système de réputation. Les armes rencontrées au cours de l'aventure seront stockées dans votre armurerie personnelle, vous permettant de choisir une nouvelle arme de départ en recommençant une aventure. La progression d'expérience est perdue mais l'or est gardé entre chaque partie.

1.3 Ressources



FIGURE 2 – Tileset personnages



FIGURE 3 – Exemple tileset ennemis



FIGURE 4 – Exemple tileset terrain



FIGURE 5 – Arbres



FIGURE 6 – Rochers



1.0 ©COPYRIGHT 2013 ORYXDESIGNLAB
WWW.ORYXDESIGNLAB.COM

 **ORYX**
16-BIT FANTASY SPRITES

FIGURE 7 – Exemple animations attaques

2 Description et conception des états

2.1 Description des états

Un état de jeu est formé de la carte du monde possédant des cases d'un certain élément, repérées par ses coordonnées dans le monde, pouvant éventuellement posséder un obstacle infranchissable inamovible ou un personnage mobile. Ce personnage fait partie d'une équipe possédant un inventaire contenant des objets, ceux-ci étant principalement des armes pouvant être équipées individuellement et possédant diverses capacités et caractéristiques. L'équipe du joueur peut combattre et obtenir des quêtes fournissant des récompenses lors de leur accomplissement. Le détail des attributs et opérations des différentes classes est présenté ci-dessous.

2.1.1 State

- L'état possède une matrice de pointeurs de cellules qui contiennent toutes les informations liées au terrain et à la position des personnages
- Il possède aussi toutes les équipes de personnages, ainsi que les quêtes et le combat éventuel qui se déroule.
- L'heure correspondant au nombre de fronts d'horloge, ainsi que la vitesse à laquelle le jeu est actualisé permet d'associer un temps à chaque état du jeu
- Un booléen indique si l'inventaire est ouvert

2.1.2 Cell

- La case doit contenir un élément choisi parmi une énumération et peut contenir un obstacle infranchissable par les personnages

2.1.3 Character

- Les coordonnées du personnage et la direction dans laquelle il se tourne
- Le personnage a un nom et appartient à une race qui sera utile par la suite pour les quêtes notamment
- Il possède également des attributs utiles au combat lui permettant de se déplacer, d'attaquer et de pouvoir encaisser des attaques
- Chaque personnage possède enfin une arme lui permettant d'effectuer diverses actions

2.1.4 Team

- L'équipe a un nom
- Elle comprend l'ensemble des personnages aptes au combat. Seul un combattant représente l'équipe hors combat lors du déplacement sur la carte du monde, et seul un nombre limité de personnages peuvent combattre simultanément

- L'équipe stocke également l'inventaire des objets récupérés, pouvant être affectées aux différents personnages de l'équipe

2.1.5 Inventory

- L'inventaire comprend différents objets, entre autres les armes affectées aux combattants et celles en réserve

2.1.6 Weapon

- L'arme peut posséder un nom
- L'arme possède des capacités permettant d'attaquer ou de se défendre
- Elle possède également un élément de prédilection répercuté sur son porteur

2.1.7 Ability

- Les habilités peuvent avoir un nom, coûtent un certain nombre de points d'action, infligent un certain nombre de dégâts (ou de soin en cas de valeur négative) et possèdent éventuellement un temps de recharge de la capacité
- Elles possèdent un élément répercuté sur l'attaque en elle-même. Elle sera souvent du même élément que l'arme, mais pourront parfois différer
- L'attaque possède une portée effective, et une zone d'effet dépendant du type d'arme utilisé
- Une réduction de dégât peut être envisagée pour les attaques de zone afin de moins affecter les personnages éloignés du point d'impact

2.1.8 Fight

- Le combat enregistre le nombre de tours passés afin de définir la prochaine équipe à jouer
- Il possède les différentes équipes se battant afin que chaque membre de l'équipe puisse se déployer avant de commencer

2.1.9 MainQuest

- Les quêtes principales s'obtiennent avant d'entrer en phase de combat en effectuant certaines actions et peuvent se décliner en divers objectifs à remplir à travers plusieurs combats à travers ses classes filles

2.2 Conception Logiciel

Le diagramme des classes pour les états est présenté en Figure 8 :

- La classe en vert contient les informations globales sur le déroulement du jeu et permettent de déterminer l'état actuel du jeu
- Les classes en jaune caractérisent les personnages et le terrain dans lequel ils évoluent
- Les classes en bleu déterminent les actions possibles par les différents personnages
- Les classes en rouge permettent d'implémenter les phases de combat ainsi que les quêtes, permettant d'obtenir du matériel ou de nouveaux membres d'équipe.

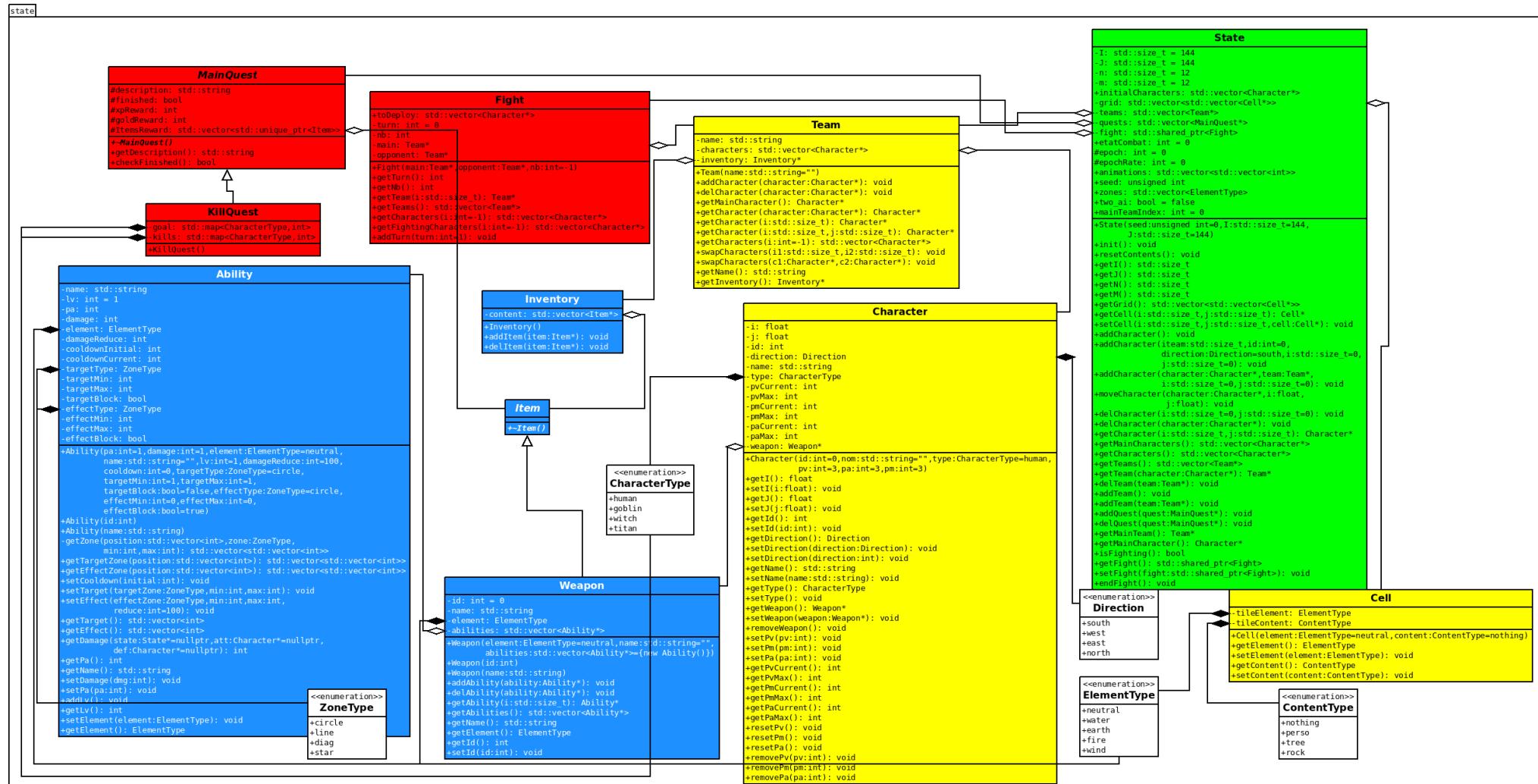


FIGURE 8 – Diagramme des classes d'état.

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Le rendu de l'état est possible grâce aux différentes informations récupérées de la classe World, c'est à dire les tuiles et les personnages. Il s'agit donc d'un rendu à 3 couches successives. Toutes les textures sont chargées et découpées afin de renvoyer le sprite correspondant à un personnage donné ou à une cellule du monde donnée (élément et contenu optionnel). Ce sprite est ensuite déplacé au lieu d'en générer un nouveau. Un système de vue centre la fenêtre sur une petite zone autour du personnage principal (encadré d'un carré blanc) et n'actualise l'affichage des sprites sur cette vue uniquement et pas sur le monde entier. Cela évite d'une part de charger des sprites inutilement et d'autre part de créer une carte du monde où les zones pas encore visitées ne seraient pas visibles (ce serait alors une vue du monde en entier).

3.2 Conception logiciel

Le diagramme de classes du rendu est présenté en Figure 10 :

- La classe Sprites est le gestionnaire des différentes ressources. Elle charge les différentes textures du jeu lors de l'instanciation et génère les différents sprites de résolutions paramétrables, compatibles avec l'état du monde, dépendant alors de la tuile à afficher (élément + contenu) et du personnage à afficher. Le SpriteGeneratorById permet de générer les sprites pour les différentes fenêtres d'interface joueur.
- Le Render réutilise alors les sprites générés pour afficher la carte grâce au RenderWindow de sfml et gère la View correspondante via display().
- L'interface utilisateur se base sur la classe Element. Cette classe permet de créer des conteneurs pour les objets/éléments sfml (text, sprite et rectangle). La classe Elements applique le design pattern composite, ainsi un élément peut être un "parent" et contenir des éléments "enfants" qui vont automatiquement se positionner et se dimensionner en fonction de la position et taille du parent. Un élément parent peut avoir comme enfants d'autre élément parent, ainsi les éléments s'organisent selon une structure d'arbre. La classe Element permet également d'encapsuler la gestion des évènements sfml. Voici quelques Elements particuliers : Window (affiche une fenêtre déplaçable et fermable), CharacterStat (affiche les stats d'un perso), Sheets (affiche les informations d'une équipe, d'un personnage, d'une arme, d'une capacité)



FIGURE 9 – Preview du rendu

```

N6render9RectangleE 11 Mouse over
*this abs: ( 0, 0) (700, 700)

N6render6WindowE 12
parent abs: ( 0, 0) (700, 700)
*this rel: ( 50, 50) (200, 100)
*this abs: ( 50, 50) (200, 100)

N6render9RectangleE 13
parent abs: ( 50, 50) (200, 100)
*this rel: ( 0, 0) (100%, 30)
*this abs: ( 50, 50) (200, 30)

N6render4TextE 14
parent abs: ( 50, 50) (200, 30)
*this rel: ( 5, 5) ( 10, 10)
*this abs: ( 55, 55) ( 10, 10)

N6render9RectangleE 15
parent abs: ( 50, 50) (200, 30)
*this rel: ( -5, -5) ( 20, 20)
*this abs: (225, 55) ( 20, 20)

N6render9RectangleE 16
parent abs: ( 50, 50) (200, 100)
*this rel: ( 0, 30) (100%, -30)
*this abs: ( 50, 80) (200, 70)

N6render6WindowE 17
parent abs: ( 0, 0) (700, 700)
*this rel: (400, 50) (200, 100)
*this abs: (400, 50) (200, 100)

N6render9RectangleE 18
parent abs: (400, 50) (200, 100)
*this rel: ( 0, 0) (100%, 30)
*this abs: (400, 50) (200, 30)

N6render4TextE 19
parent abs: (400, 50) (200, 30)
*this rel: ( 5, 5) ( 10, 10)
*this abs: (405, 55) ( 10, 10)

N6render9RectangleE 20
parent abs: (400, 50) (200, 30)
*this rel: ( -5, -5) ( 20, 20)
*this abs: (575, 55) ( 20, 20)

N6render9RectangleE 21
parent abs: (400, 50) (200, 100)
*this rel: ( 0, 30) (100%, -30)
*this abs: (400, 80) (200, 70)

```

Listing 1 – Structure d’abre de Element

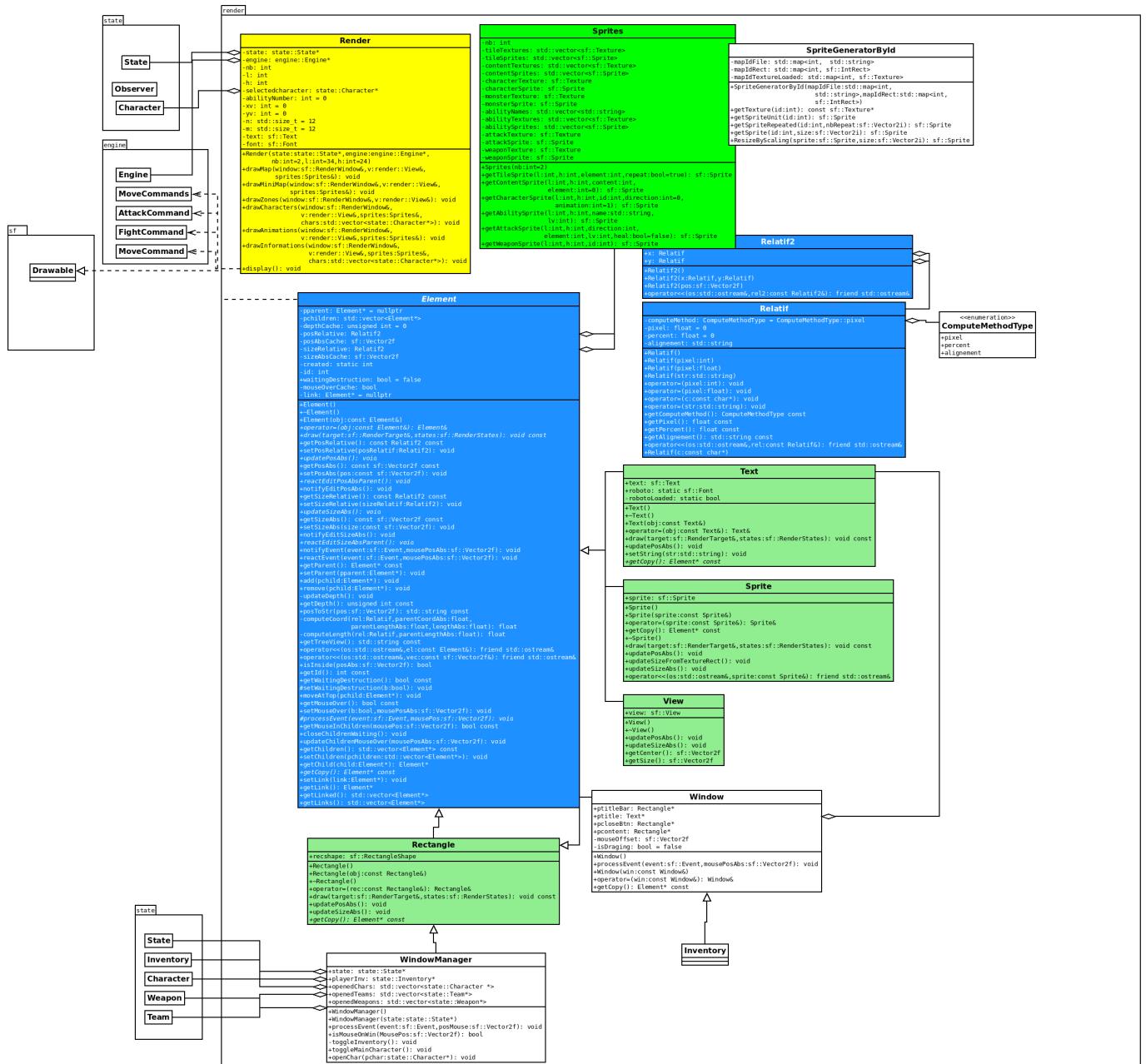


FIGURE 10 – Diagramme des classes de rendu.

4 Règles de changement d'états et moteur de jeu

4.1 Horloge globale

Les commandes sont exécutées à une fréquence de 30 Hz et chaque animation de déplacement est constitué d'une étape de changement de direction ainsi que de 12 étapes de déplacement pur séparés en 4 sprites différents.

4.2 Changements extérieurs

Les changements extérieurs sont le déplacement du personnage, que ce soit en combat ou non, ainsi que les phases d'attaque en combat.

4.3 Changements autonomes

Les changements autonomes sont le changement de direction du personnage lors de son déplacement, l'affichage des sprites fx lors de l'attaque ainsi que le démarrage de la phase de combat permettant de déployer tous les personnages de chaque équipe si un ennemi est rencontré lors de la phase d'exploration.

4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Figure 11. Le moteur de jeu repose sur le pattern Command et permet la mise en oeuvre différée des commandes modifiant l'état du jeu.

Classes Command Ces classes représentent les commandes exécutées par le moteur.

- MoveCommands permet de générer l'ensemble des commandes individuelles MoveCommand afin de créer une animation de déplacement en utilisant un pathfinding A* (rapide mais ne générant pas nécessairement le chemin le plus optimal) et FightCommand en cas de rencontre avec un ennemi, ce qui lance la phase de combat.
- AttackCommand permet aux différents personnages d'attaquer avec leurs capacités et ainsi d'infliger des dégâts à différentes cibles et génère des commandes DamageCommand pour générer les dégâts ainsi que les animations d'attaque.

Engine Le moteur exécute les commandes dans l'ordre d'apparition. Il enregistre également les commandes dans un rollback pour pouvoir les inverser pour l'IA avancée, ou encore pour proposer le choix d'annuler un tour au joueur.

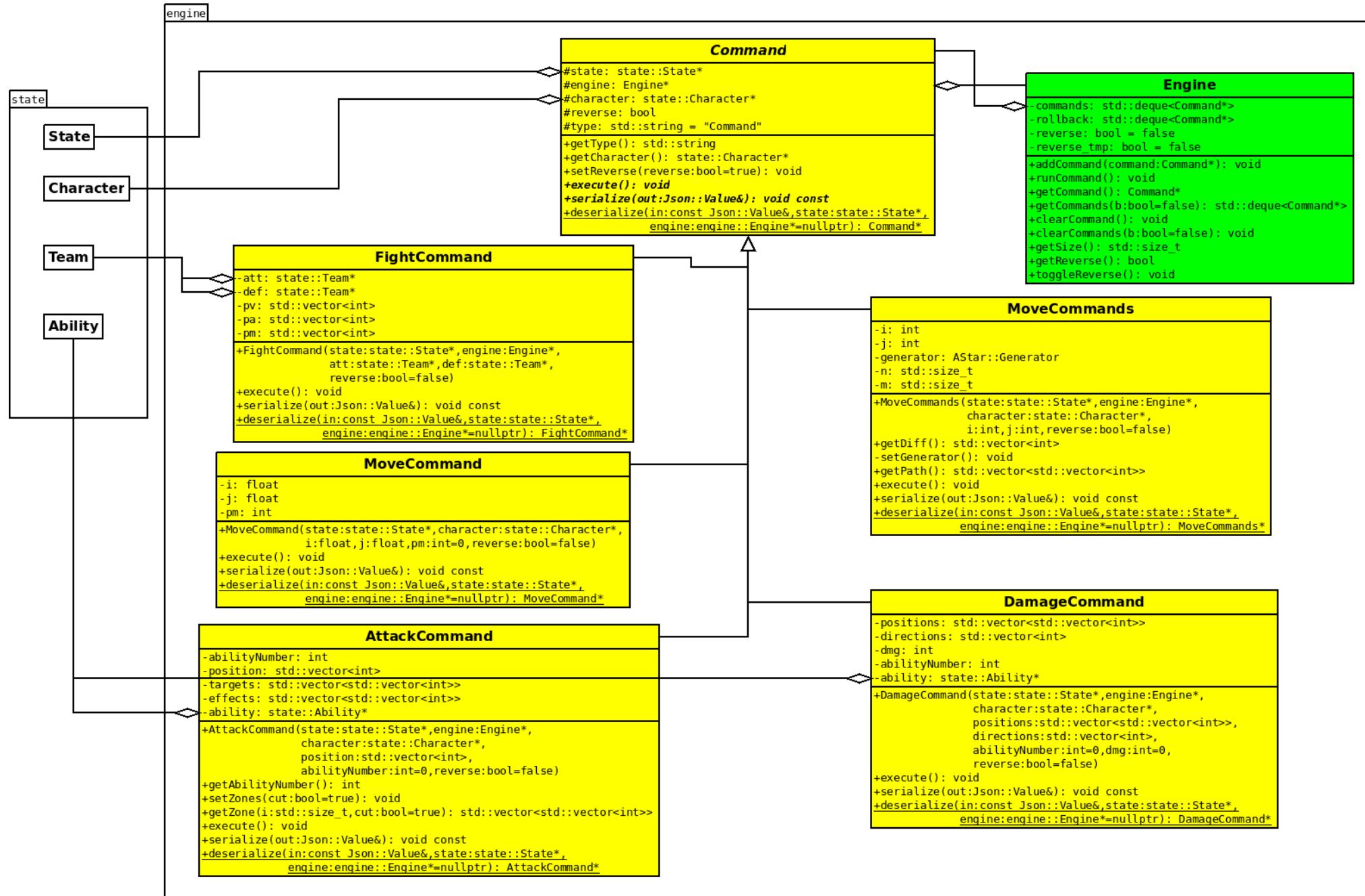


FIGURE 11 – Diagramme des classes du moteur.

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

L'IA execute des commandes aléatoires parmi le mouvement et l'attaque, s'il lui reste assez de pa/pm.

5.1.2 Intelligence basée sur des heuristiques

L'IA balaye toutes les positions possibles du personnage et pour chaque position va affecter un Score à chaque attaque possible à partir de cette position : pour chaque dégât réellement infligé à un ennemi (respectivement un allié) le score augmente (respectivement diminue) de ce montant (un soin est considéré comme des dégâts négatifs). Si un personnage meurt suite à cette attaque, le score est majoré (respectivement minoré), mais si les dégâts sont beaucoup plus élevés que la vie du personnage, un malus est appliqué pour éviter le gaspillage des ressources. Dans une plus faible proportion, plus l'action coûte de ressources en pa, plus le score diminue, mais plus l'IA se rapproche d'un ennemi, plus le score augmente. Ces différences de score sont minimes devant le score de dégât afin de différencier de manière plus précise des actions de score similaire, et permettent de se rapprocher des ennemis si aucune action n'est possible. Une liste de toutes les commandes de même score maximal est établie et une commande aléatoire parmi ces meilleures commandes (déplacement puis attaque) est envoyée au moteur jusqu'à épuisement des pa et pm de chaque joueur (ou la fin du combat).

5.1.3 Intelligence avancée

L'IA balaye plusieurs options parmi ses meilleurs coups (actions avec un score au dessus d'un certain seuil fixé) grâce à un arbre de profondeur fixé, représenté par la classe `TreeNode`, mais cette fois elle peut anticiper les mouvements adverses (qui soustraient leur score au noeud de l'arbre) si la profondeur de l'arbre est assez grande. Le système de score reste identique à celui de l'intelligence heuristique. Un système de rollback permet d'annuler les actions exécutées de proche en proche, modifiant directement l'état, au lieu de stocker plusieurs copies de l'état actuel.

5.2 Conception logiciel

Le diagramme de classes pour l'intelligence artificielle est présentée en figure 12

Classes AI Les classes filles s'organisent selon un pattern strategy :

- RandomAI : IA aléatoire
- HeuristicAI : IA heuristique
- HeuristicAI : IA avancée

Score Cette classe calcule et stocke le score d'une action avec les critères expliqués dans la section de l'intelligence heuristique

TreeNode Chaque noeud possède un parent (sauf la racine de l'arbre), une liste d'enfants, un score et les commandes exécutées pour arriver à ce noeud dans l'arbre. Il est donc associé à un état virtuel du jeu.

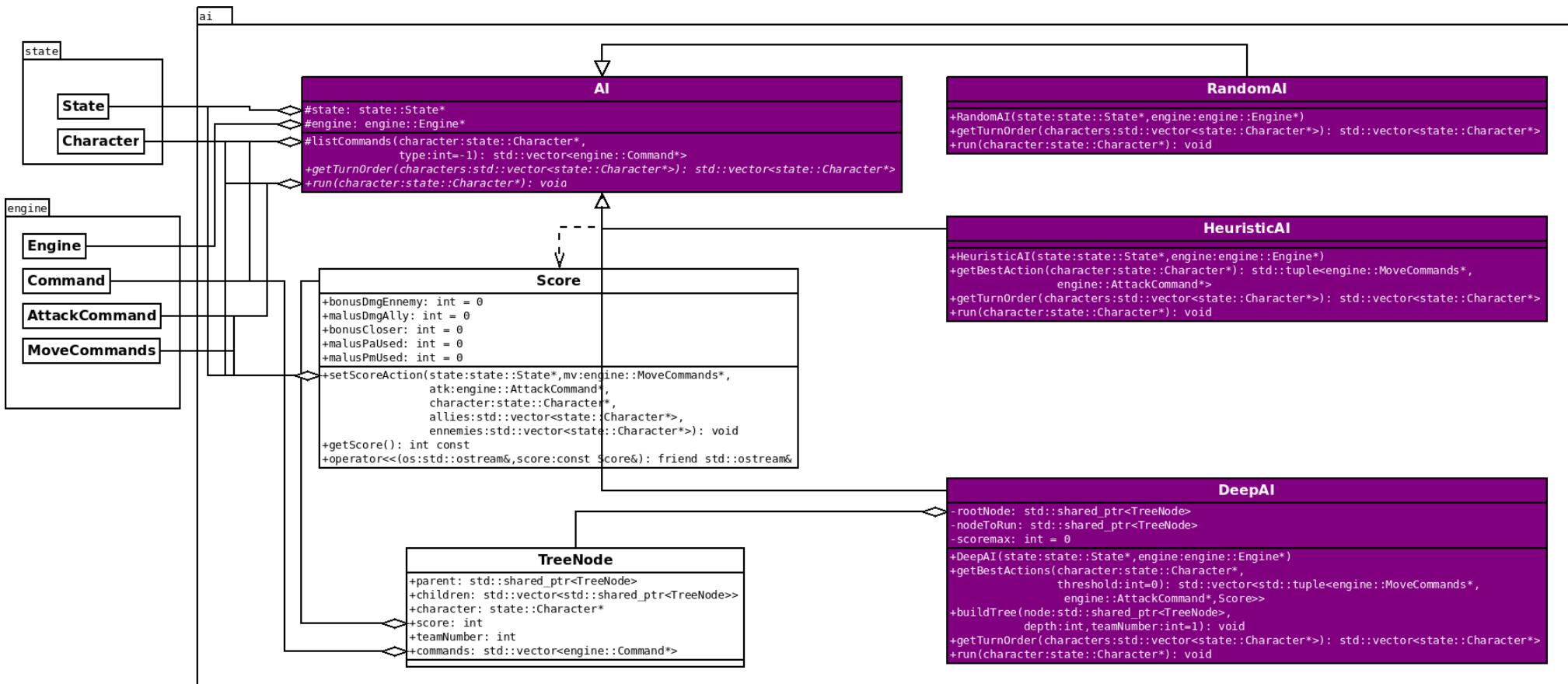


FIGURE 12 – Diagramme des classes d'intelligence artificielle.

6 Modularisation

On décide ici de proposer au joueur le choix de faire affronter deux intelligences artificielles afin d'observer les choix pris par ceux-ci. Pour cela, un booléen indique si ce choix a été pris, et le thread de l'intelligence artificielle a été modifié pour inclure une recherche de l'ennemi le plus proche, un déploiement aléatoire en combat, puis le même algorithme de combat est appliqué, et ceci en boucle jusqu'à la victoire/défaite du joueur principal.

La seconde fonctionnalité implémentée est un enregistrement et une vue d'une sauvegarde du jeu. La sauvegarde s'effectue via un fichier texte replay.txt écrit au format Json qui reprend le principe du rollback qui sauvegarde toutes les commandes effectuées. Il s'agit donc ici de défiler le rollback et de sérialiser chaque commande au lieu de le dépiler. Les commandes parasites d'annulation et de calcul ne sont alors pas prises en compte. La lecture de la sauvegarde se fait alors en déserialisant chaque commande en les envoyant dans la file de commandes du moteur qui les exécute alors soit normalement pour jouer la sauvegarde, ou alors à toute vitesse pour juste charger une partie. En plus des commandes, la random seed est aussi stockée en Json afin de pouvoir régénérer le state à partir des mêmes commandes aléatoires sans avoir à sauvegarder l'état en entier en sérialisant chacune des classes qui peuvent sensiblement changer assez régulièrement.

Cette sauvegarde est effectuée à chaque fin de combat (car cette fin supprime des classes et est donc irréversible, ce qui annule le rollback. Avant d'effacer le rollback, toutes les commandes en cours sont écrites et les suivantes seront ajoutées après) et à la fermeture de la fenêtre sfml mais pas en cas de fermeture brutale (Ctrl-C).

6.1 Organisation des modules

L'objectif est de créer un serveur qui écoute les requêtes des différents clients, gérant la connexion et la déconnexion des joueurs, l'envoi et à la réception de commandes afin que chaque client soit synchronisé avec l'état du serveur. Dans ce cas, un nouveau client notifie son envie de se connecter au serveur, qui va alors créer une nouvelle équipe pour tous les clients déjà connectés et va envoyer à ce nouveau client les informations nécessaires pour recréer l'état initial du serveur (avant qu'aucune connexion ne se fasse), ainsi que toutes les commandes qui lui ont permis de se trouver dans son état actuel (l'ensemble des commandes des autres clients dans l'ordre chronologique). Chaque client demande également périodiquement l'ensemble des commandes qui lui manquent pour retrouver l'état du serveur. Ces demandes ne sont pas synchronisées entre chaque client donc les commandes du moteur du serveur ne sont pas supprimées après l'exécution mais sauvegardées pour pouvoir renvoyer des commandes différentes pour chaque client selon son retard sur l'état actuel, ainsi que pour reconstituer l'état pour un nouveau joueur (en envoyant alors toutes les commandes disponibles).

Le rendu de chaque client reste toujours désynchronisé du moteur.

6.2 Conception logiciel

Le diagramme de classes pour la modularisation est présentée en figure 13.

NetworkClient Cette classe permet de se connecter au serveur grâce à l'api de connexion, puis lance une partie comme en local avec certaines restrictions liées au jeu en ligne (pas de rollback ou de replay, les commandes ne sont pas exécutées directement localement mais sont envoyées sur le serveur (putServerCommand) dès que possible puis reçues (getServerCommands) périodiquement). Il contient donc un état, un moteur, un rendu et une intelligence artificielle. Il contient également de quoi se connecter en ligne, ainsi que l'identifiant de la dernière commande du serveur exécutée afin de récupérer les bonnes commandes lors de la demande au serveur.

Game Le serveur possède sa propre version de l'état et du moteur, ainsi qu'une base de données des joueurs connectés. Il exécute toutes les commandes qu'il reçoit de chaque client puis les sauvegarde pour ensuite les restituer lors de la demande de chaque client qui peut demander des commandes différentes selon leur état d'avancement de leur état.

PlayerDB stocke tous les joueurs grâce à un identifiant unique auto-incrémenté. Chaque joueur possède un pseudo servant à s'inscrire, un booléen indiquant s'il est connecté ou pas et le numéro de l'équipe que le client associé contrôle.

AbstractService Les services REST sont implémentés via cette classe en coordination avec le ServicesManager pour fournir les différents services attendus :

- VersionService renvoie la version de l'API afin d'éviter les conflits de version
- GameService renvoie les éléments nécessaires aux clients pour reconstituer l'état du serveur lors de l'inscription et de la connexion
- CommandsService permet d'échanger les commandes entre client et serveur afin d'assurer que chaque client exécute toutes les commandes, incluant celles des autres joueurs
- PlayerService gère l'inscription, la connexion et la déconnexion des différents clients. Lors de l'inscription, si le pseudo choisi n'existe pas il est créé et le joueur se connecte alors. Sinon, s'il existe et que le champ isLoggedIn est libre, il devient pris et le client se reconnecte avec son personnage déjà créé sur le serveur lors d'une précédente connexion puis déconnexion. Enfin, si ce champ est déjà pris, un pseudo est redemandé pour reproduire les étapes précédentes. Au bout de 10 tentatives infructueuses, la connexion s'arrête.

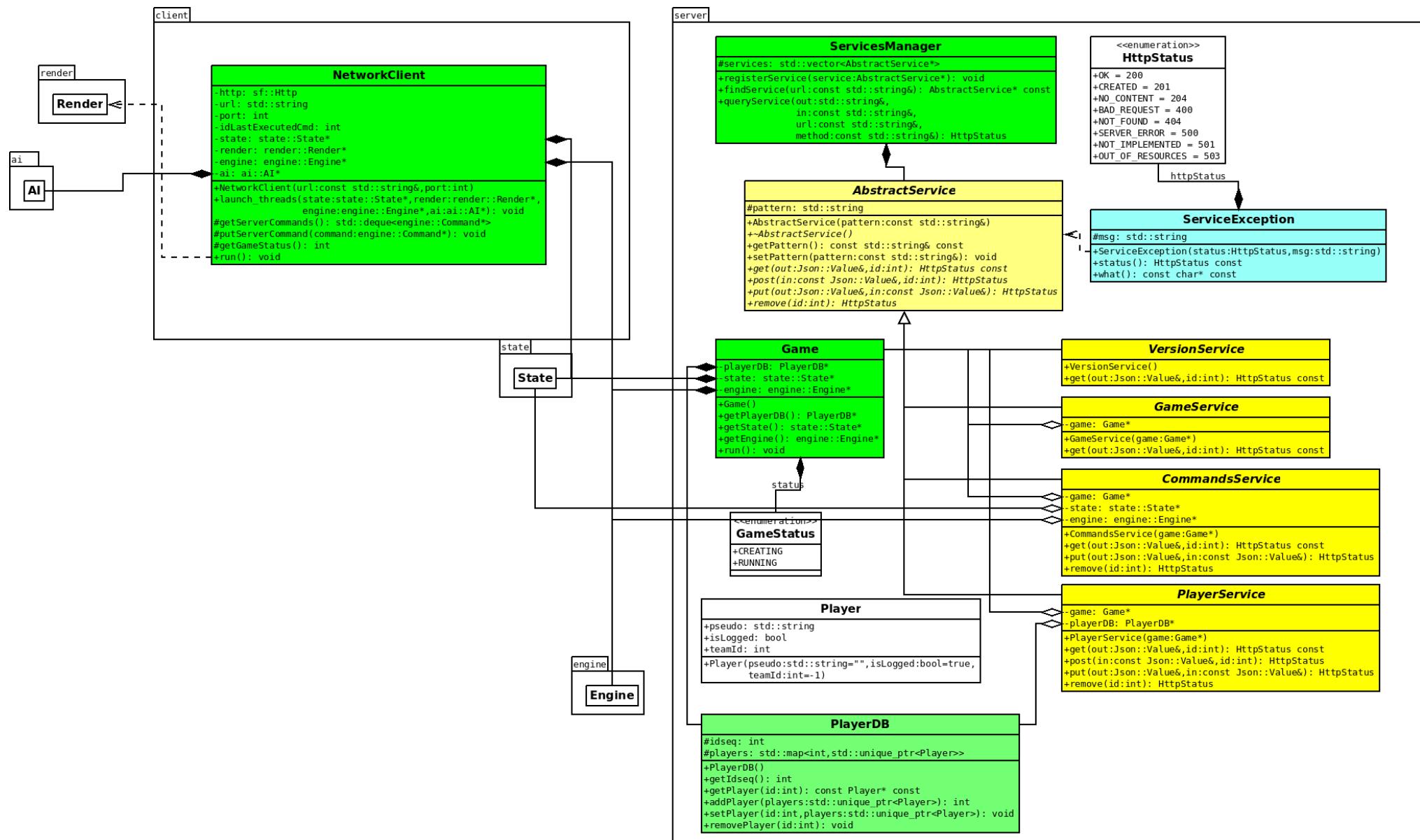


FIGURE 13 – Diagramme des classes pour la modularisation.