Projet Logiciel Transversal

Federico MONTES DE OCA – Billy UM

Figure 1 - Exemple du jeu Dofus (source : dofus.com)

Table des matières

1. Présentation Générale

1.1 Archétype

L'objectif de ce projet est de réaliser un jeu de type Dofus. A l'origine, Dofus est un jeu de rôle en ligne massivement multijoueur (MMORPG) développé et édité par Ankama, dans lequel les joueurs incarnent un ou plusieurs types de personnages. Comme les autres MMORPG, le joueur peut faire évoluer son personnage en augmentant son niveau, en personnalisant son équipement ou bien encore en choisissant sa classe de personnage. Selon sa classe, le joueur aura alors un gameplay différent. Cependant, notre projet se focalisera en particulier à créer un jeu reposant sur son système de combat. En effet, il s'agira d'un jeu de combat entre personnages se déroulant tour par tour sur des terrains variés. Un exemple du système de combat est présenté Figure 1.

1.2 Règles du jeu

Deux équipes s'affrontent. Dans chacune des équipes, il y aura 3 personnages de type différent maximum. Le but du jeu est alors de vaincre l'ennemi adverse en éliminant tous ses personnages. Chaque tour, un joueur aura la possibilité de faire avancer un personnage et le faire attaquer. Le déplacement du personnage sera régi selon des cases prédéfinis sur un terrain. En effet, pour avancer, le personnage devra se déplacer selon des cases proposées. Quant au système de combat, il se déroulera tour par tour. Durant ce tour, le joueur choisira le personnage qu'il veut jouer. Il décidera à la fois de son déplacement et du coup porté à un personnage adverse.

Types de personnage

Le joueur disposera de trois types de personnage :

- Un personnage qui attaque à distance,
- Un mage qui utilise des sorts,
- Un personnage qui attaque au corps à corps.

Modes de jeu

Deux modes de jeu seront proposés :

- Seul, contre l'IA.
- Multiplayer, en ligne, une personne contre une autre personne.

1.3 Ressources

Des TileSets complets ont été trouvés sur le site : https://kenney.nl.

Avec les éléments de la Figure 3, on peut assembler des personnages à partir d'une base.

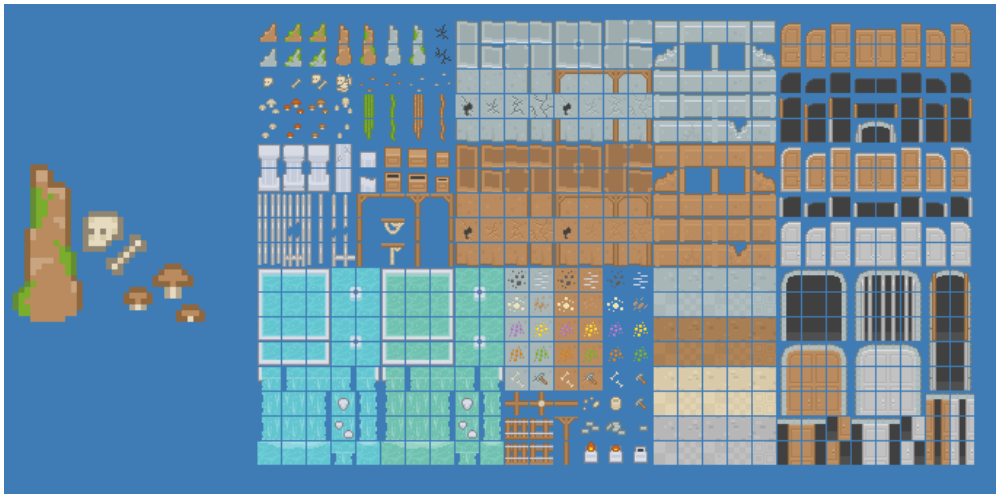

FIGURE 2 -Texture pour les terrains et les décors

FIGURE 3 - Texture pour les personnages

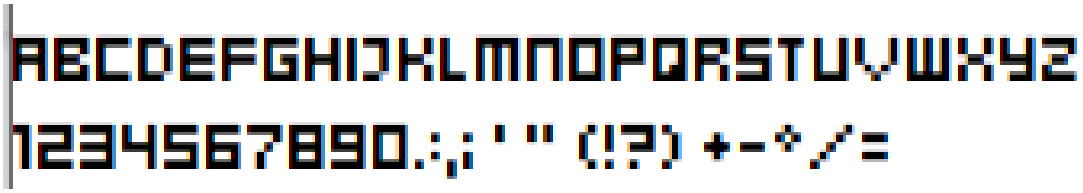

FIGURE 4 - Typographie (Kenney Mini Square v1.0)

2. Description et conception des états

2.1 Description des états

Un état du jeu est formé par un ensemble d'éléments fixes sur le terrain et un ensemble d'éléments mobiles. Tous les éléments possèdent les propriétés suivantes :

- Coordonnées (x, y) dans la grille
- Nom

2.1.1 Etat éléments fixes

Le terrain est formé par une grille d'éléments nommé « MapCell ». La taille de cette grille est fixée au démarrage du niveau. Les types de cases sont :

1. Cases « ObstacleMapCell ». Les cases « ObstacleMapCell » sont des éléments infranchissables pour les éléments mobiles. Le choix de la texture est purement esthétique, et n'a pas d'influence sur l'évolution du jeu. On considère les types de cases suivants :
    - Les obstacles « ROCK », pour les rochers.
    - Les obstacles « TREE », pour les arbres.
    - Les obstacles « WATER », pour l'eau.
    - Les obstacles « FIRE », pour le feu.

2. Cases « SpaceMapCell ». Les cases « SpaceMapCell » sont les éléments franchissables par les éléments mobiles. On considère les types de cases « SpaceMapCell » suivants :
    - Les espaces « GRASS » pour désigner l'herbe.
    - Les espaces « SAND », pour désigner le sable.
    - Les espaces « CONCRETE », pour désigner le béton.

    De plus, certaines cases « SpaceMapCell » ont la possibilité d'améliorer les statistiques des personnages. Ils vont par exemple pouvoir augmenter la santé des personnages ou bien leurs attaques.

2.1.2 Etat éléments mobiles

Deux types d'éléments mobiles peuvent être déplacés par le joueur :

1. Elément mobile « Character » qui représente un personnage. Cet élément est dirigé par le joueur. Chaque « Character » possède des statistiques propres à sa classe. On lui associe ainsi un niveau de santé "heath", un niveau d'attaque "attack", un niveau de mana "mana" et un niveau de défense "defense". Il se distingue également par son caractère "boosted" si le personnage est boosté, par son champ d'attaque et ses types de mouvements.

   À « Character » est également associée une propriété nommée « CharacterStatusID », qui peut prendre les valeurs suivantes :
   - Statut « SELECTED » : cas où le « Character » est sélectionné.
   - Statut « AVAILABLE » : cas où le « Character » est disponible.
   - Statut « WAITING » : cas où le « Character » est en attente.
   - Statut « DEATH » : cas où le « Character » est mort.
   - Statut « TARGET » : cas où le « Character » est visé.

   « Character » possède enfin une propriété nommée « CharacterTypeD » pour distinguer le type de personnage, et qui peut prendre les valeurs suivantes :
   - Type « STRENGHT » : cas où le « Character » est un personnage orienté corp à corp.
   - Type « DISTANCE » : cas où le « Character » est un personnage orienté attaque à distance.
   - Type « MAGICIAN » : cas où le « Character » est un personnage orienté magie.

2. Elément mobile " Cursor " qui représente le curseur pointé. Ce " Cursor" permet de sélectionner un personnage. "Cursor" dispose d'une position repérée par ses coordonnées (x, y).

2.1.2 Etat général

A l'ensemble des éléments statiques et mobiles, nous rajoutons les propriétés suivantes :

- "turn" qui indique le nombre de tours.
- "end" qui indique la fin du jeu.
- "win" qui indique que le joueur a gagné.
- "loose" qui indique que le joueur a perdu.

2.2 Conception logicielle

Le diagramme des classes pour les états est présenté en Figure 11. Son architecture est fondée sur le Polymorphisme par sous-typage dont la classe "Element" est la classe mère. Toute la hiérarchie des classes filles "Element" permettent de représenter les différentes catégories et types d'élément.

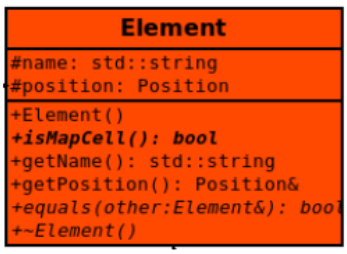

FIGURE 5 - Classe "Element"

On peut distinguer les classes filles qui héritent directement de la classe "Element" :

- La classe "Character" est la classe qui va permettre la création de nos trois personnages. Elle contient toutes les informations nécessaires à leur implémentation. À chaque personnage est associé des statistiques propres à lui par une relation de composition entre la classe "Character" et la classe "Stats". Dans notre cas, on considère que des statistiques ne peuvent exister sans personnage. On associe également par relation de composition une énumération "CharacterStatusID" décrivant le statut du personnage, ainsi qu'une énumération "CharacterTypeID" exposant sa classe de personnage.

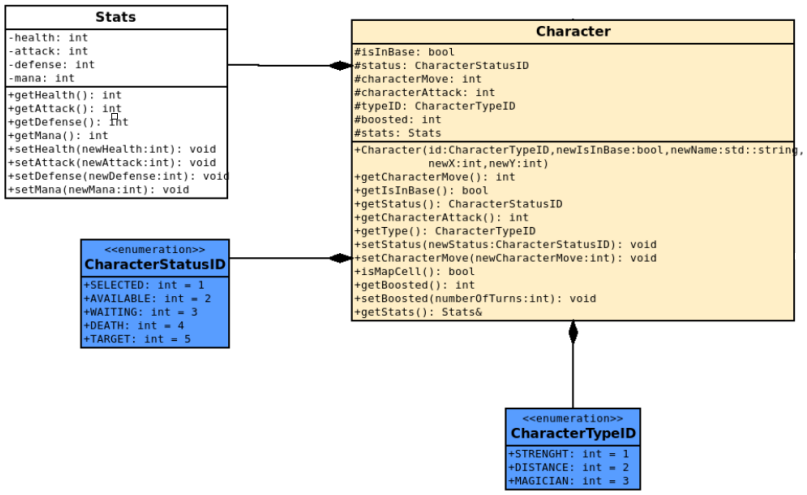

FIGURE 6 - Bloc “Character”

- La classe “Cursor” est la classe qui va permettre de représenter le curseur pointé et va permettre de sélectionner les personnages .

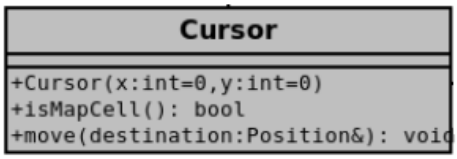

FIGURE 7 - Classe “Cursor”

- La classe “MapCell” établit également une relation d’héritage. Ses deux sous-classes “SpaceMapCell” et “ObstacleMapCell” sont des spécialisations de la classe “MapCell”. Leur rôle est la création du terrain :

    - “SpaceMapCell” représente les cases où l’on peut se déplacer. Une énumération “SpaceMapCellID” par composition lui est associée pour représenter les différents types d’espace.

    - “ObstacleMapCell” représente les obstacles. Une énumération “ObstacleMapCellID” par composition lui est associée pour représenter les différents types d’obstacle.

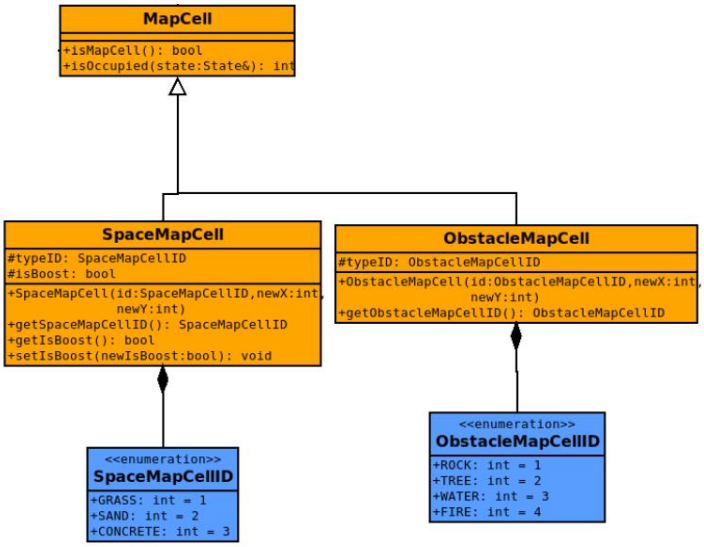

FIGURE 8 - Bloc “MapCell”

Ensuite, pour décrire tous les éléments de l’état de notre jeu, la classe “State” est le conteneur de nos ensembles d’éléments. Elle contient un vecteur “map” à deux dimensions de pointeurs d’éléments de “MapCell” qui forme le terrain de notre jeu, ainsi qu’un vecteur “characters” à une seule dimension de pointeurs des personnages de “Character”. Elle possède également des booléens pour déterminer la fin du jeu, la victoire et la défaite.

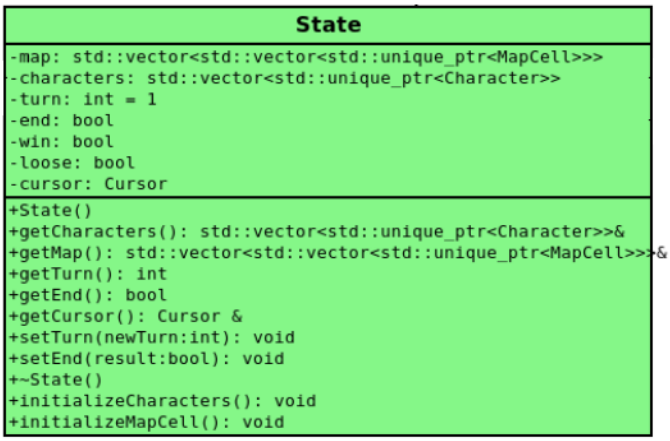

FIGURE 9 - Class “State”

On utilise le design pattern Observable pour signaler les changements d'états. La classe "State" hérite de la classe Observable. Lorsque la classe "State" subit un changement, l'observateur "IObserver" notifie l'évènement correspondant au changement appliqué.

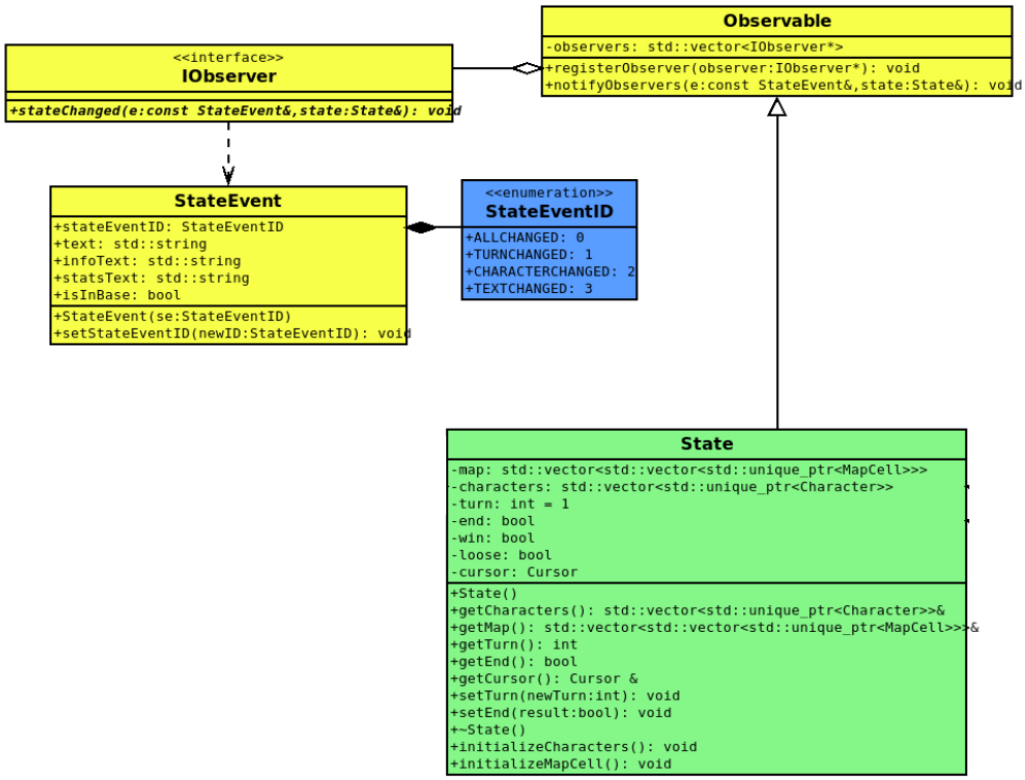

FIGURE 10 - Pattern Observable

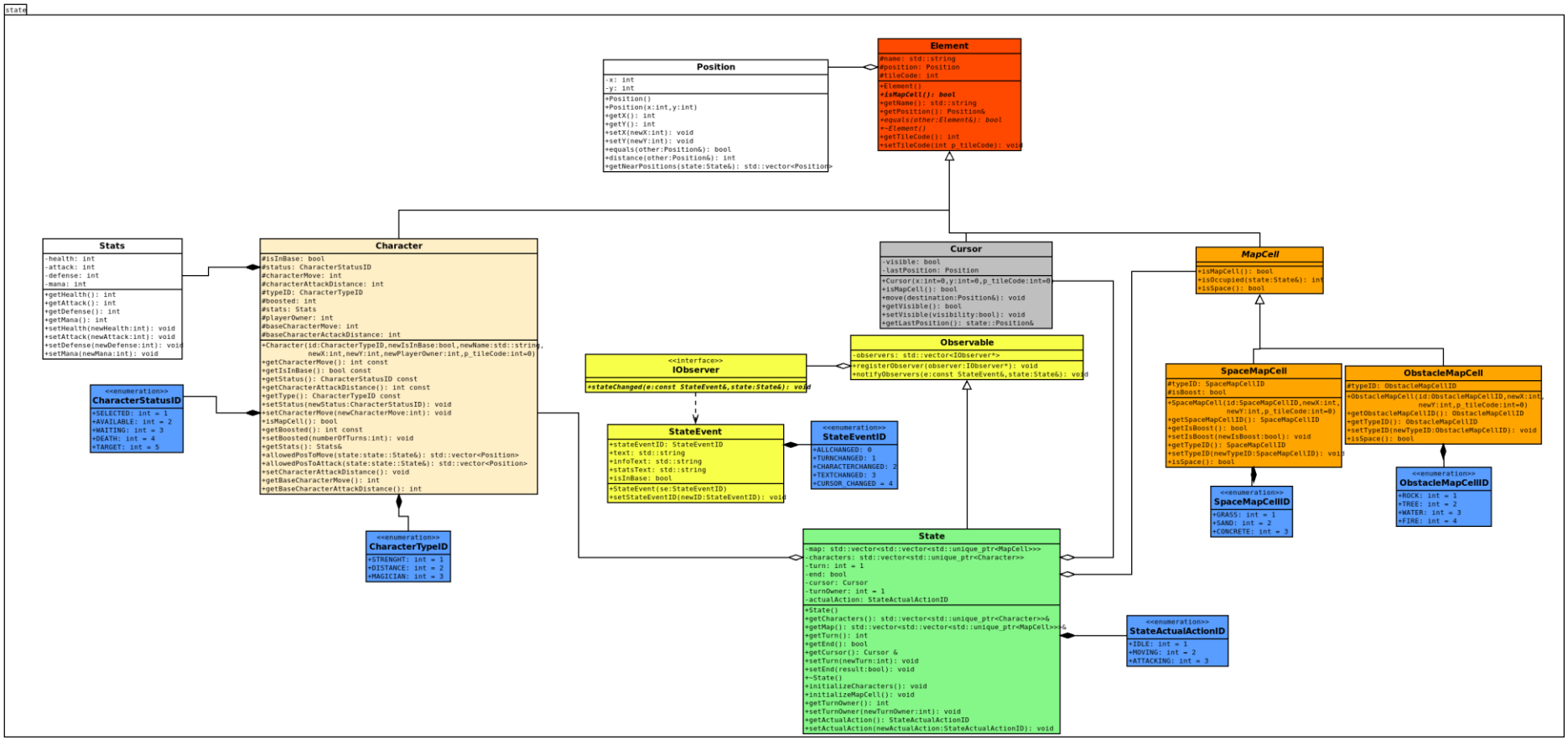

FIGURE 11 – State diagram

3. Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Un état est fixé par 3 éléments : un terrain, des personnages, et des informations autour du jeu. Pour le rendu de cet état, nous avons fait le choix d'un rendu par tuile à l'aide de la bibliothèque SFML. Par conséquent, notre affichage est composé de 3 surfaces :

- Une surface pour les éléments statiques SpaceMapCell et ObstacleMapCell.
- Une surface pour les éléments mobiles Characters et Cursor.
- Une surface pour les informations du jeu

La map est construite à partir d'un fichier texte « map_v0.txt » où est contenu chaque ID de chaque tuile constituant la map. Si on souhaite changer la map, il faut alors changer l'ID associé aux tuiles dans ce même fichier.

Deux méthodes nous permettent d'initialiser l'état du jeu. La méthode initializeMapCell de la classe State (package state) nous permet de fabriquer pour chaque code tuile du fichier « map1.txt » l'objet correspondant qui peut être un espace ou un obstacle. À chaque objet est alors associé un pointeur unique. Cela va permettre de remplir un tableau à 2 dimensions, une dimension pour les pointeurs des objets espaces, et une deuxième pour les pointeurs des obstacles du terrain. La méthode initializeCharacters suit le même fonctionnement. Elle charge le personnage en fonction de l'ID de la tuile et en fonction des coordonnées qui lui sont données.

Lorsque qu'un évènement se produit, un changement d'état est effectué. L'affichage doit alors être modifié en conséquence et être mis à jour. Pour l'instant, nous avons définis 4 évènements potentiels susceptibles de modifier un état.

3.2 Conception logicielle

Le diagramme des classes pour les états est présenté en Figure 16.

3.2.1 StateLayer

Cette classe agit en tant qu'intermédiaire avec le package state et avec l'observateur de la classe State. StateLayer a une collection de TileSet, de Font (typographie), de Window pour l'affichage graphique et une collection d'objets Surface. Dans le constructeur, on charge en mémoire les fonts et les TileSets correspondants. La fonction initSurfaces(State) initialise les objets Surfaces. La fonction draw(RenderWindow) est responsable de dessiner les surfaces dans la fenêtre Window. La fonction stateChanged(...) est responsable d'écouter les évènements et d'agir en conséquence.

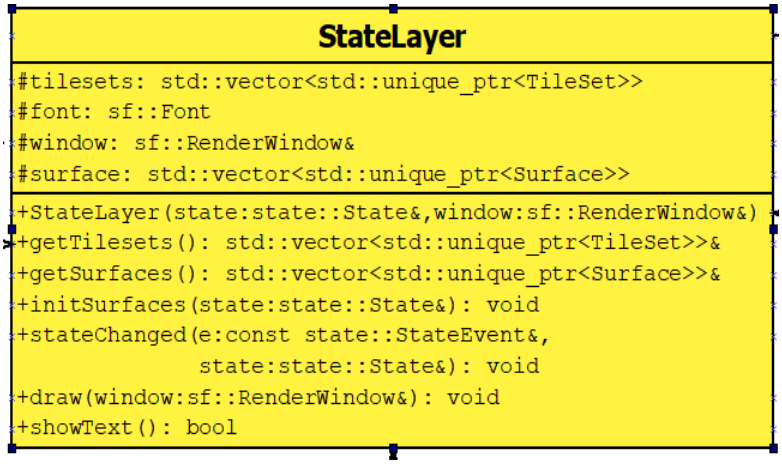

FIGURE 12 - StateLayer Class

3.2.2 Surface

Cette classe est le principal intermédiaire avec la librairie SFML. Chaque surface a une texture et une VertexArray (quads) qui servent à dessiner les tiles pour former une grille. Les fonctions load et draw sont inspirés par cette article de la documentation official SFML. En résumé, ces fonctions sont responsables de la localisation (sur un TileSet) des textures correspondant à chaque Tile.

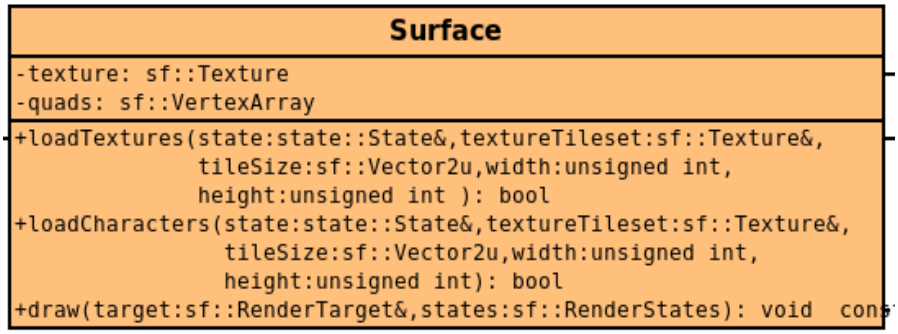

FIGURE 14 - Surface Class

3.2.3 TileSet

Cette classe permet d'identifier les ressources TileSet correspondant pour chaque TileSetID (chaque type de TileSet). CellWidth et cellHeight définissent les dimensions de chaque Tile. Dans notre cas, on préfère 32x32 pixels pour chaque tuile mais cela peut être amener à changer dynamiquement. Dans le constructeur, on passe un TileSetID pour choisir la ressource correspondante et la charger en mémoire.

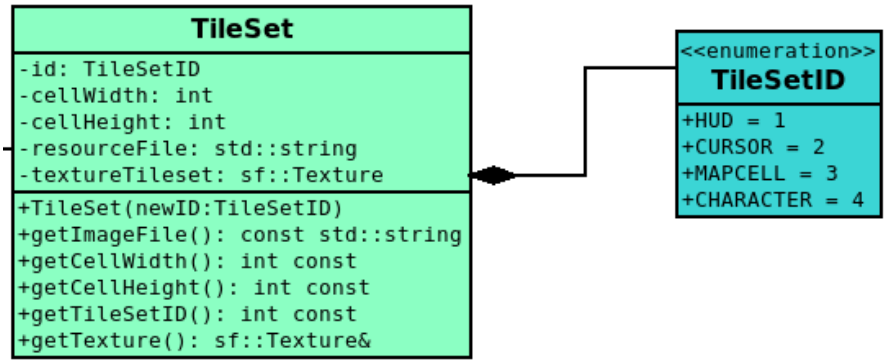

FIGURE 15 - TileSet Class and enumeration

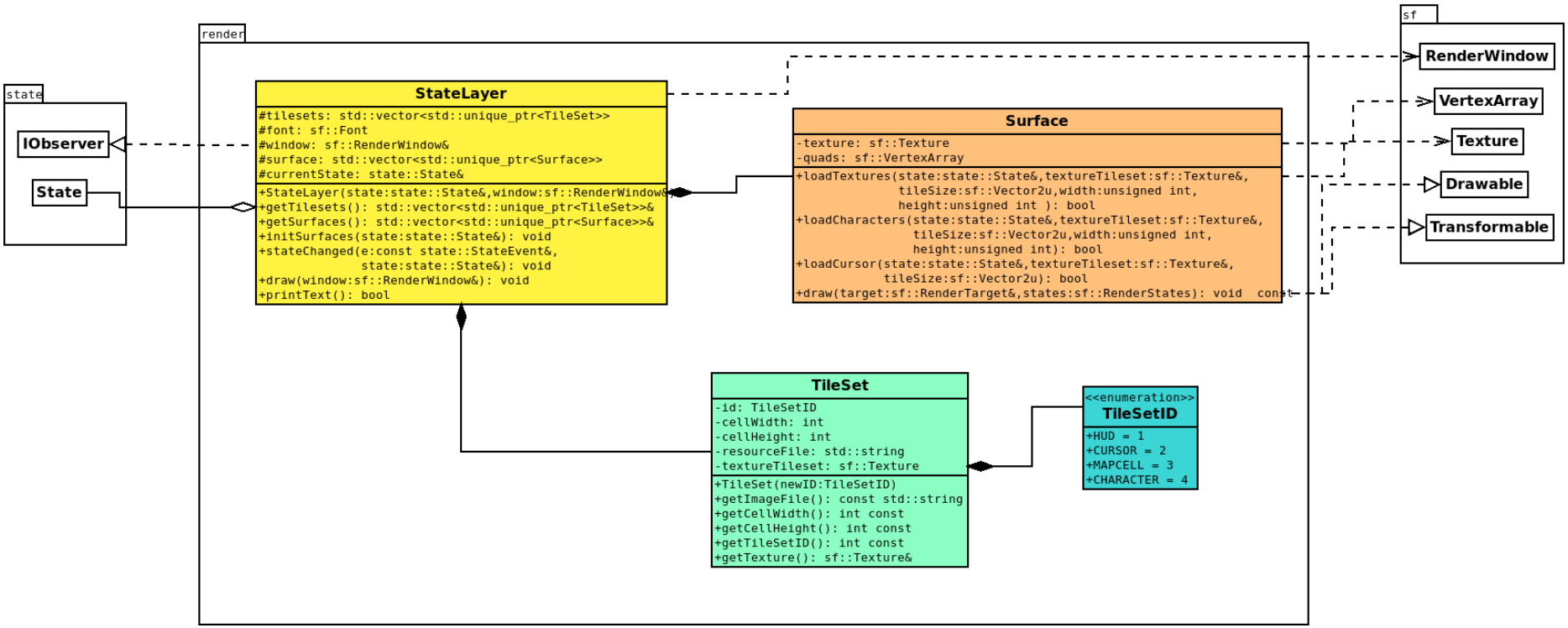

FIGURE 16 – Render diagram

4. Règles de changements d'états et moteur de jeu

4.1 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieures, qui seront une pression sur les touches du clavier. On distingue 3 types de commandes extérieures :

- Commande de sélection
- Commandes de déplacement
- Commandes d'attaque

4.1.1 Commandes de sélection

Au début de chaque tour, tous les personnages d'un joueur possèdent le statut AVAILABLE. Le joueur doit alors sélectionner un personnage avec le curseur qu'on pourra déplacer avec les touches du clavier. Le personnage aura alors le statut SELECTED.

4.1.2 Commandes de déplacement

Le personnage pourra alors ensuite se déplacer selon le nombre de cases que sa classe lui permet. Le joueur choisira les déplacements suivants :

- Un personnage MAGICIAN peut se déplacer de 1 case maximum
- Un personnage STRENGHT peut se déplacer de 2 cases maximum
- Un personnage DISTANCE peut se déplacer de 3 cases maximum

A la fin de son choix de déplacement, le personnage a toujours le statut SELECTED car le personnage peut encore faire une attaque. Cependant, le joueur peut décider de mettre fin à son tour, le personnage ne sera plus dans le statut SELECTED.

Une fois qu'une commande extérieure est effectuée par le joueur, on passe au tour du joueur suivant.

4.1.3 Commandes d’attaque

Une fois sélectionné, un personnage peut attaquer un ennemi adverse. Le joueur déplace le curseur sur la case de l’ennemi à attaquer avec les flèches du clavier. Il sélectionne ensuite cette case. Le personnage adverse sur cette case passe alors au statut TARGET. Le joueur décide de l’attaque à effectuer.

4.2 Changements autonomes

Les changements autonomes sont appliqués à chaque création ou mise à jour d’un état, après les changements extérieurs. Ils sont exécutés dans l’ordre suivant :

1. Si le joueur est mort, on affiche "GAME OVER".
2. On met à jour les statistiques des personnages des joueurs en fonction des règles du jeu.
3. On applique les règles de déplacement du joueur et de l’IA.
4. Si le joueur a gagné, on affiche "GAME WON".

4.3 Conception logicielle

Le diagramme des classes pour le moteur du jeu est présenté en Figure 17. L’ensemble du moteur de jeu repose sur un patron de conception de type Command.

- Le rôle de classe Command est de représenter une commande. A cette classe est associée un CommandId pour identifier le type de commande.

- La classe SelectedCharacterCommand permet de savoir quel personnage est selectionné. Elle hérite de Command

- La classe AttackCommand correspond à l’attaque d’un personnage et la classe MoveCommand représente le déplacement d’un personnage. Ces deux classes héritent de Command.

- La classe UpdateHudCommand permettra de mettre à jour les informations du jeu en dehors du terrain.

- La classe CheckWinnerCommand permet de savoir le joueur gagnant et la classe FinishTurnCommand permet la fin d'un tour.

- La classe Engine est le coeur du moteur. Elle stocke les commandes dans une std::map avec une clé entière. Ce mode de stockage permet d'introduire une notion de priorité : on traite les commandes dans l'ordre de leurs clés, de la plus petite à la plus grande. Lorsqu'un nouveau tour démarre, ie lorsqu'on a appelé la méthode update() après un temps suffisant, le moteur appelle la méthode execute() de chaque commande, incrémente le nombre de tours, met à jour les statistiques des personnages puis supprime toutes les commandes.

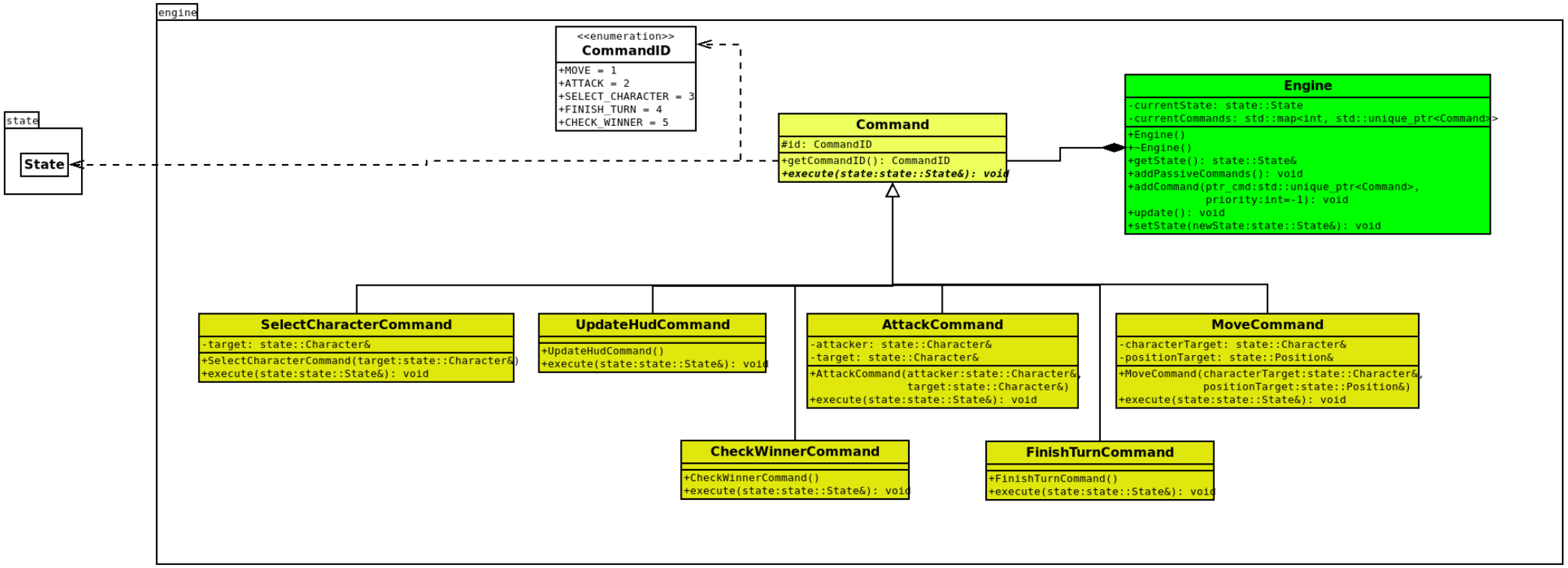

FIGURE 17 – Engine diagram

5. Intelligence Artificielle

5.1 Stratégie

5.1.1 Intelligence Aléatoire

La stratégie de notre intelligence aléatoire est fondée sur le diagramme suivant :

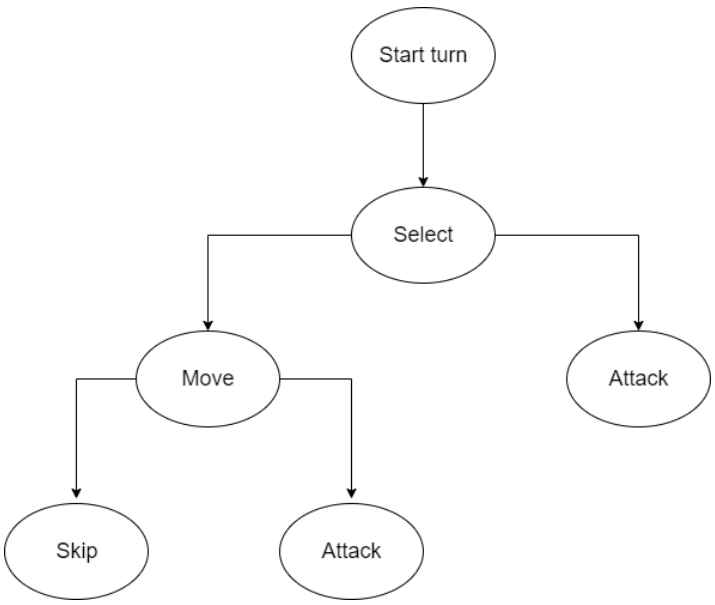

FIGURE 18 – Diagramme des actions possibles

Au début de chaque tour, notre IA sélectionne aléatoirement un personnage parmi les 3 personnages disponibles sur le terrain. Après avoir sélectionné un personnage, l'IA choisit aléatoirement une action : se déplacer ou attaquer. Si l'IA décide d'attaquer une cible et que son attaque est valable, alors après son attaque il ne pourra plus effectuer d'actions. Son tour est fini.

Cependant, si l'IA décide de se déplacer dans les cases possibles. A la fin de son déplacement, l'IA pourra choisir aléatoirement une action suivante : mettre fin au tour ou attaquer. De même, s'il choisit d'attaquer, cela signifiera la fin de son tour.

5.1.2 Intelligence Heuristique

Nous proposons ensuite un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard et permettre à l'IA d'être plus « intelligente ».

Nous avons une première heuristique qui permet à l'IA de savoir quel personnage choisir. L'IA choisit le personnage dont la distance à l'ennemi est la moins faible.

Nous avons une seconde heuristique qui va de pair avec la première, et permet à l'IA de cibler l'ennemi le plus proche. L'IA choisit donc le personnage dont la distance à l'ennemi est la moins faible (algorithme similaire au précédent).

Nous avons une troisième heuristique basée sur BFS (Algorithme de parcours en largeur) qui permet à l'IA de trouver un chemin vers l'ennemi adverse.

5.1.3 Intelligence Avancée

Nous proposons une intelligence plus avancée en suivant les méthodes de résolution de problèmes à états finis. On se base sur un algorithme basé sur un arbre de recherche pour que l'IA détermine quels sont les choix de commandes optimaux en prenant en compte les statistiques des personnages sur le terrain.

Pour faire fonctionner l'algorithme, l'IA a besoin d'une fonction d'évaluation qui attribue une valeur à chaque action possible en fonction de l'état du jeu. Selon la valeur donnée, l'IA choisira l'action à effectuer.

Nous décomposons notre intelligence avancée en 3 phases :

1. L'IA choisit quel personnage il va jouer
2. L'IA choisit la cible la plus proche
3. L'IA se déplace vers cette cible.

Nous avons choisi d'améliorer notre IA sur la phase 1 et 2. Pour la phase 3, on reprend l'algorithme BFS.

Notre algorithme suit le graphe suivant pour la phase 1 et 2 :

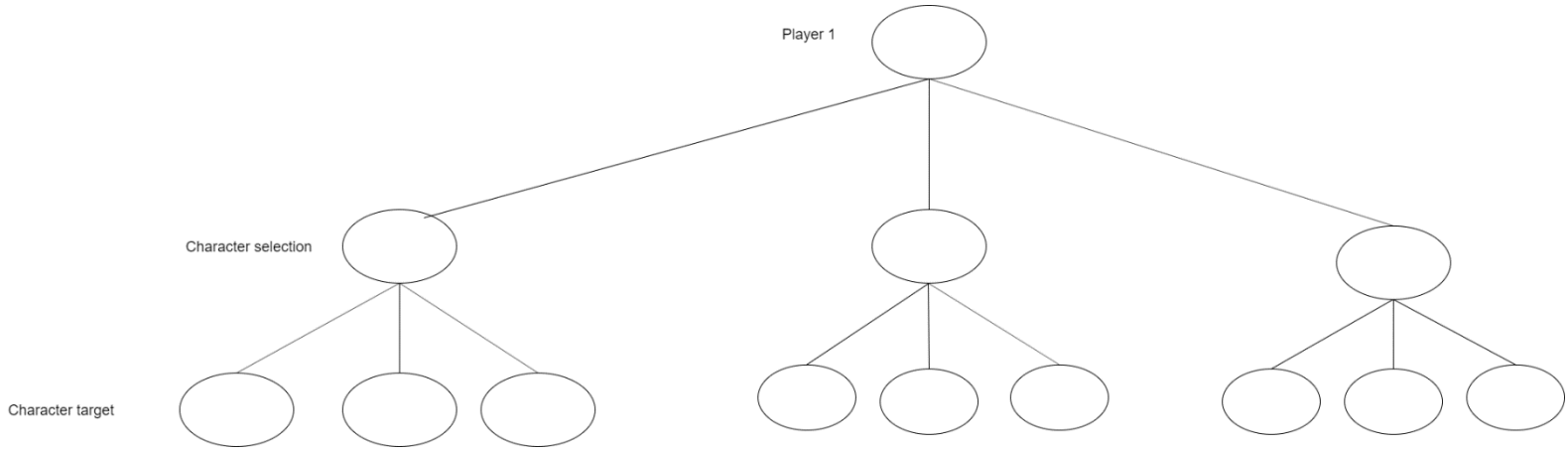

FIGURE 19 – Deep AI diagram

Player1 est notre IA.

Phase 1 : Character Selection

Dans un premier temps, l'IA doit choisir un personnage. Nous avons implémenté une fonction qui permet de déterminer le personnage avec le plus de points de vie et de défense possible. Ensuite, nous avons soustrait cette valeur à la distance la plus proche à un ennemi. Tout cela forme notre fonction d'évaluation. Elle parcoure donc un arbre parmi les personnages disponibles sur le terrain.

Phase 2 : Character target

Dans un second temps, l'IA doit choisir une cible adverse. Nous avons implémenté une seconde fonction d'évaluation. Elle évalue à la fois la santé de ses propres personnages puis la défense et la santé des ennemis adverses. L'IA choisira alors l'ennemi possédant le moins de défense et de santé parmi ses ennemis adverses. Elle parcoure également un arbre parmi les personnages ennemis disponibles sur le terrain.

5.2 Conception Logicielle

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 18.

Classe AI : Les classes filles de la classe AI implantent différentes stratégies d'IA, que l'on peut appliquer pour notre jeu.

RandomAI : Intelligence aléatoire.

HeuristicAI : Classe qui implémente l'IA heuristique. L'algorithme parcours en largeur BFS permettra de trouver un chemin vers les cibles.

MapNode : Permet d'avoir la liste des nœuds nécessaire au fonctionnement de BFS.

DeepAI : Classe qui implémente l'IA avancée. Elle contient les fonctions d'évaluations nécessaires au fonctionnement du DeepAI.

Node : Classe qui permet d'avoir la liste des index des personnages disponibles sur le terrain.

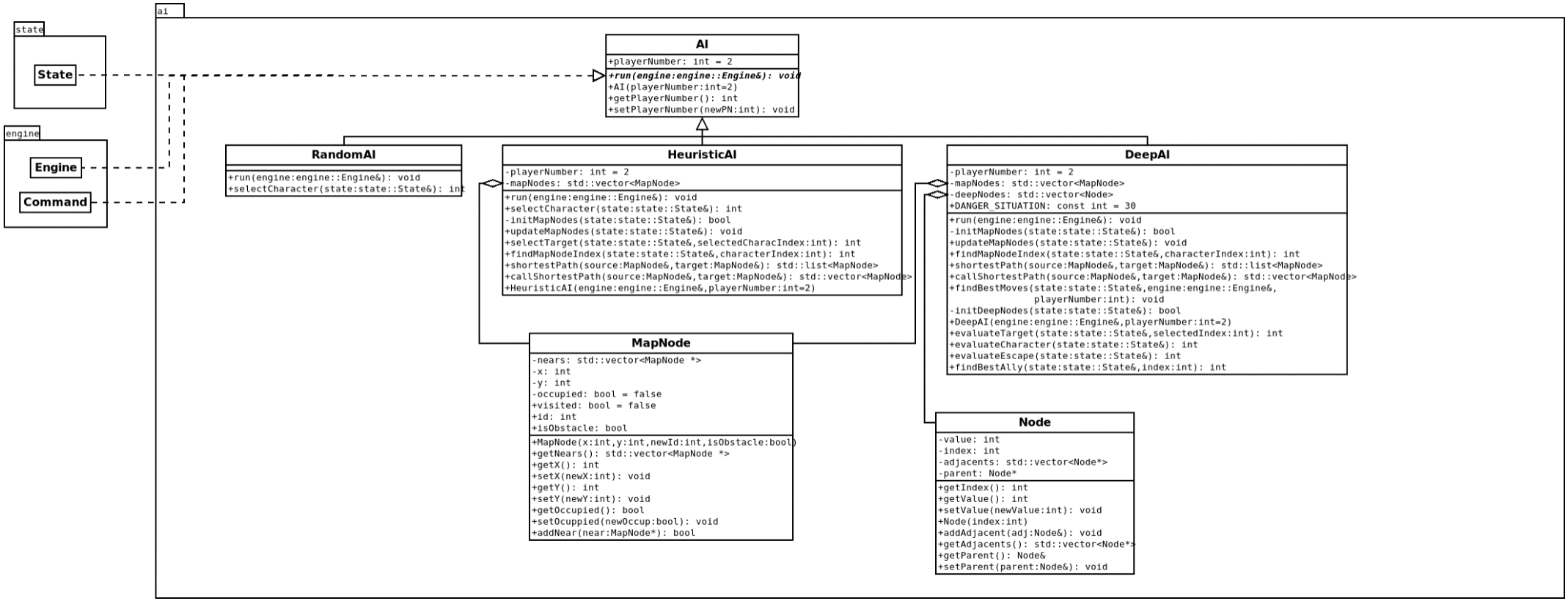

FIGURE 20 – AI diagram

6. Modularisation

6.1 Organisation des modules

Notre objectif ici est de placer le moteur de jeu sur un thread, puis le moteur de rendu sur un autre thread. Le moteur de rendu est nécessairement sur le thread principal et le moteur du jeu est sur un thread secondaire. Les commandes transitent d'un module à l'autre.

6.1.1 Commandes

Les commandes peuvent arriver à l'importe quel moment, y compris lorsque l'état du jeu est mis à jour. Pour résoudre ce problème, nous proposons d'utiliser une fonction "threadEngine" qui va faire marcher le moteur de jeu en parallèle seulement si la variable "canRunEngine" est "true". La variable "canRunEngine" devient « true » quand "engineUpdating" est appelé et false quand la fonction "threadEngine" est appelé.

6.1.2 Répartition sur différentes machines : rassemblement des joueurs

On propose une API Web REST pour réunir différents joueurs sur différentes machines. Cette opération sera effectuée avant le début d'une partie. Une fois tous les joueurs réunis, la partie pourra démarrer et il ne sera plus possible de changer les joueurs.

La première étape pour pouvoir jouer en réseau est la création d'une liste de client pour le serveur. Pour ce faire, nous formons des services CRUD sur la donnée "joueur" via une API Web REST.

Requête GET/player/<id>

Pas de données en entrée

Cas joueur <id> existe	Status OK Données sorties : type : « object », properties : { « name » : { type : string }, }, required : [« name »]
Cas joueur <id> n'existe pas	Status NOT_FOUND Pas de données de sortie

La requête **GET/player/<id>** permet d'obtenir des informations sur le joueur.

Requête PUT/player

Données en entrée :

type : « object »,
properties : {
« name » : { type : string },
},
required : [« name »]

Cas il reste une place libre	Status CREATED Données sorties : type : « Object », properties : { « id » : { type : number, minimum : 0, maximum : 2 }, }, required : [« id »]
Cas plus de place libre	Status OUT_OF_RESOURCES Pas de données de sortie

La requête **PUT/player** ajoute un nouveau joueur à la liste des joueurs de la partie à condition que le nombre maximal de joueur ne soit pas déjà atteint.

Requête POST/player

Données en entrée :

type : « object »,
properties : {
« name » : { type : string },
},
required : [« name »]

Cas joueur <id> existe	Status OK Pas de données de sortie
Cas joueur <id> n'existe pas	Status NOT_FOUND Pas de données de sortie

La requête **POST/player** modifie une ou plusieurs informations d'un joueur existant.

Requête DELETE/player/<id>

Pas de données en entrée

Cas joueur <id> existe	Status OK Pas de données de sortie
Cas joueur <id> n'existe pas	Status NOT_FOUND Pas de données de sortie

La requête **DELETE/player/<id>** supprime un joueur de la liste des joueurs à condition que ce joueur fasse partie de la liste des joueurs**.**

6.1.3 Répartition sur différentes machines : échange des commandes

Pour la gestion des commandes, tous les clients envoient leurs commandes moteur au serveur lorsque c'est leur tour de jouer. Chaque client exécute ses commandes dans son propre moteur et les envoie également au moteur du serveur.

Les clients envoient des requêtes GET et POST au serveur. Le premier pour obtenir les dernières commandes, c'est-à-dire qu'à chaque fois qu'un GET est reçu, le serveur réinitialise la liste des commandes en attendant une requête POST. Quand une requête POST arrive, le serveur ajoute la commande dans la liste dynamique (pour les GET) où on voit toutes les actions passées de la partie.

Lorsque le client est en attente, il fait un polling chaque seconde (GET /command) pour obtenir les commandes de l'adversaire. Lorsque c'est son tour, il va jouer en appelant l'IA. Et en profitant du pattern Observateur, le client au lieu de faire juste un engine.apply(), il fera appel à notifyObserver pour publier chaque commande utilisée avec POST avec serialize().

- Requête GET/command/<id> : Récupère les commandes que le joueur avec l'id correspondant a joué et a envoyé au serveur.
- Requête POST/command : Permet d'envoyer une commande d'un joueur au serveur

6.2 Conception logicielle

La classe **Client** contient toutes les informations pour faire fonctionner le jeu : Moteur de jeu (avec état intégré), intelligences artificielles, et rendu. Cette classe est un observateur du moteur de jeu :

- Lorsque le moteur est sur le point d'exécuter ses commandes (méthode engineUpdating()), il appelle les IA pour ajouter les commandes
- Lorsque le moteur a terminé d'appliquer les commandes (méthode engineUpdated()), il vide le cache des événements émis vers le moteur de rendu.

La classe **KeyboardListener** permet d'enregistrer les commandes effectuées par le joueur.

La classe **NetworkClient** contient les informations pour faire fonctionner le jeu : Moteur de jeu (avec état intégré) et rendu. Contrairement à la classe Client, elle ne contient une IA, que pour le joueur qu'elle dirige.

La classe **Game** et les classes associées représentent les éléments d'une partie. En premier lieu, nous trouvons la liste des joueurs, présents ou non dans la partie. Les méthodes CRUD sont implémentées.

Les classes **Services**. Les services REST sont implantés via les classes filles de AbstractService, et gérés par la classe ServiceManager qui sélectionne le bon service et la bonne opération à exécuter en fonction de l'URL et de la méthode HTTP :

- VersionService : le traditionnel service qui renvoie la version actuelle de l'API. Indispensable dans toute API pour prévenir les conflits de version.
- PlayerService : fournit les services CRUD pour la ressource joueur. Permet d'ajouter, de modifier, de consulter et de supprimer des joueurs.
- CommandService : permet d'ajouter, et consulter les lots de commandes.
- GameService : fournit un service de consultation de l'état du jeu.

La classe **ServiceException** permet de réaliser une exception à tout moment pour interrompre l'exécution du service.

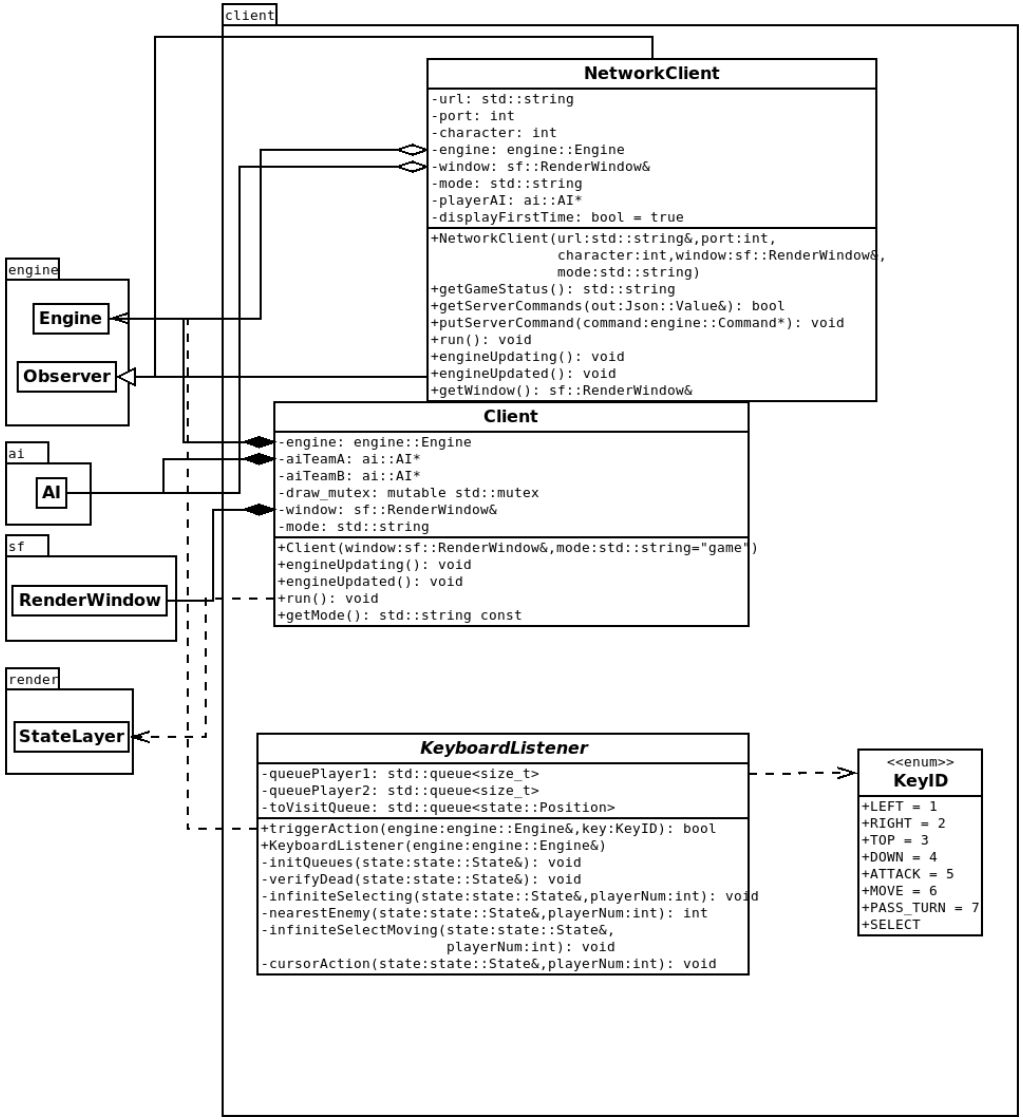

FIGURE 21– Client diagram

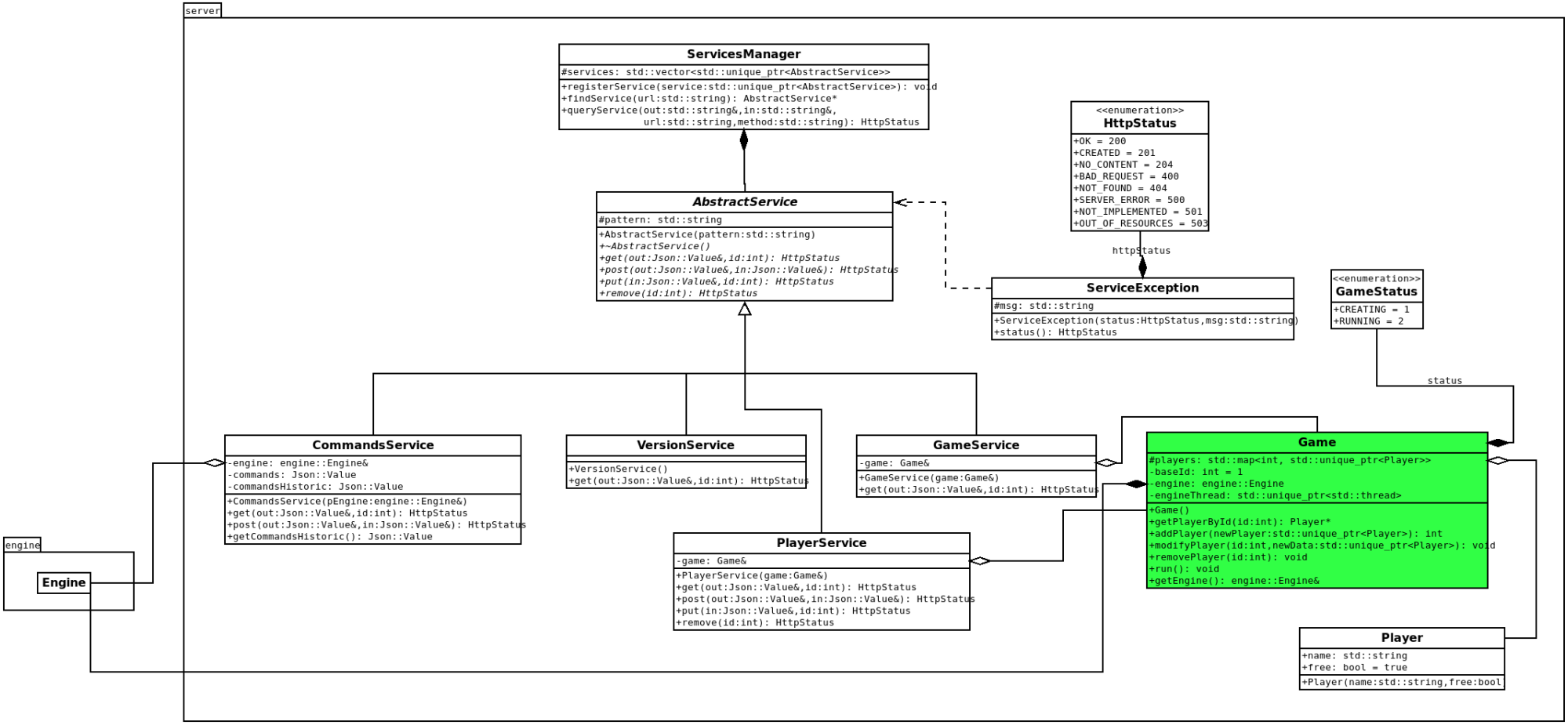

FIGURE 22– Server diagram