Projet Logiciel Transversal

Yoan AYROLES - Chloé PÉGUIN

Table des matières

1 Présentation générale

1.1 Archétype

L'objectif de ce projet est de réaliser un jeu semblable à **Fire Emblem 7** (soit **Fire Emblem : Rekka no Ken**), un jeu de rôle tactique au tour par tour de Intelligent Systems sorti en 2003 sur Game Boy Advance.

Figure 1 – Aperçu du jeu Fire Emblem 7 sur Game Boy Advance

1.2 Règles du jeu

Le joueur contrôle un groupe de personnages et affronte celui de son adversaire (contrôlé par l'ordinateur ou un autre joueur) dans une bataille tactique que le joueur remporte lorsqu'il atteint l'objectif défini en début de partie (capture d'une case, survie un certain nombre de tours, élimination d'un boss ou encore élimination totale des forces adverses).

Le déroulement d'un tour est le suivant : le joueur commence par déplacer une unité et lui donne des ordres (attendre, utiliser un objet ou encore attaquer). Une fois qu'il a fait cela pour chacune de ses unités, c'est au tour de l'adversaire de faire de même. Si toutes les unités d'un joueur sont tuées, il a perdu.

Les équipes de personnages sont constituées d'unités ayant des statistiques personnelles pour le combat (points de vie, attaque, défense, esquive…). Ces statistiques changent en fonction de la classe de l'unité (chevalier, archer, guerrier et brigand). À chaque classe correspond une arme précise (par exemple un brigand se bat à la hache, alors qu'un chevalier se bat à la lance). Une particularité des combats est le « triangle des armes » : les armes sont plus ou moins efficaces les unes contre les autres (l'épée est efficace contre la hache, qui est efficace contre la lance, elle-même efficace contre l'épée).

Chaque niveau se déroule sur une carte constituée de cases carrées sur lesquelles se déplacent les unités de chaque camp. Chaque case possède des caractéristiques qui influencent les combats en donnant des avantages aux combattants (par exemple, une unité sur une case « forêt » a plus de chance d'esquiver une attaque).

1.3 Ressources

Tout d'abord, nous utiliserons les textures de 16 pixels par 16 pixels du jeu original pour la création de cartes :

Figure 2 – Textures pour les cartes du jeu

Nous avons également les textures pour les personnages en gros plan de taille 64x64 :

Figure 3 – Textures pour les gros plans sur les personnages

Enfin, il y a les textures pour les personnages, de taille 16 pixels par 16 :

Figure 4 – Textures pour les personnages

2 Description et conception des états

2.1 Description des états

Un état de jeu est formé par un ensemble d'éléments fixes (les différents terrains qui composent la carte de jeu) et un ensemble d'éléments mobiles (les personnages). Tous les éléments possèdent les propriétés suivantes :

- Un nom
- Des coordonnées (x,y) dans la grille
- Des statistiques propres (PV, attaque, défense, esquive, critique)
- Un code de tuile

2.1.1 États éléments fixes

La carte de jeu est formée par une grille d'éléments de type Terrain. C'est une grille de taille 25 cases par 25 cases. Il existe différents types de terrains :

Terrains non praticables : ils sont inaccessibles aux éléments mobiles et n'ont donc aucune influence sur eux. Les terrains non praticables sont listés ci-dessous :

- Mur
- Rivière
- Montagne
- Falaise

Terrains praticables : ils sont accessibles aux éléments mobiles, ce sont les cases sur lesquelles les unités peuvent se déplacer. Chaque type de terrain praticable va influencer les statistiques des personnages en jeu. En effet les statistiques du terrain sur lequel se trouve un personnage s'ajoutent aux statistiques propres du personnage. On considère les types de terrains praticables suivants :

- Les plaines : elles ne changent pas les statistiques des personnages
- Les forêts : elles augmentent l'esquive des personnages situés dessus
- Les collines : elles augmentent la défense des personnages situé dessus
- Les ponts : ils ne changent pas les statistiques du personnage situé dessus mais permettent de traverser les rivières
- Les forteresses : à chaque tour elles redonnent plusieurs points de vie au personnage situé dessus sauf s'il a déjà tous ses points de vie.
- Les maisons : à chaque tour elles redonnent un point de vie au personnage situé dessus sauf s'il a déjà tous ses points de vie.

2.1.2 États éléments mobiles

Ces éléments sont dirigés par les joueurs.

2.1.2.1 Personnages

Les personnages possèdent une arme (identifiée par un code et un nom), un camp, un champ de mouvement, un champ d'attaque, un statut et un typeID.

Le statut représente l'état dans lequel se trouve un personnage à un instant donné :

- Sélectionné : dans le cas où le personnage a été sélectionné par le joueur pour effectuer une action.
- Disponible : dans le cas où le personnage peut-être sélectionné et n'a pas encore agi.
- Attente : dans le cas où le personnage ne peut-être sélectionné car :
  - c'est le tour du joueur adverse
  - le personnage a déjà effectué toutes les actions disponibles pour ce tour
- Mort : dans le cas où le nombre de points de vie du personnage vaut 0.

Le typeID représente le type auquel appartient un personnage :

- Archer
- Brigand
- Chevalier
- Guerrier

Les différents types de personnages sont différenciés par leur arme, leur champ de déplacement et d'attaque et leurs statistiques propres.

2.1.2.2 Curseur

Le curseur permet de sélectionner un personnage pour lui faire effectuer des actions.

2.1.3 État général

A l'ensemble des éléments statiques et mobiles nous rajoutons la propriété suivante :

- tour : le nombre de tours qui ont été joués
- fin : un booléen qui indique si la partie est terminée

2.2 Conception Logicielle

Le diagramme des classes pour les états est présenté en Figure 5, dont nous pouvons mettre en évidence les groupes de classes suivants :

Classe Etat : Classe qui permet d'accéder à toutes les données qui forment un état. Elle contient un tableau à deux dimensions de pointeurs de Terrain formant la grille de jeu, un tableau à une dimension de pointeurs de Personnage se situant sur la grille et un pointeur de Curseur.

Classes Element : Toutes les classes filles issues de la classe mère Element (beige) permettent de représenter les différentes catégories d'éléments (mobiles et statiques).

Classe Position : Classe qui décrit la position de chaque élément sur la grille de jeu.

Classe Statistiques : Classe qui regroupe les statistiques de chaque élément.

Classe Correspondances : Classe qui regroupe des sets de doublets composés d'un code de tuile et d'un ID d'élément.

Classes Observers : La conception de ces classes suit le Design Pattern Observer. Le but de la classe Observable dont hérite Etat est d'enregistrer des observateurs puis de les notifier à chaque changement d'état. Par exemple la classe StateLayer (render.dia) est un observateur et met à jour les textures/sprites à chaque action d'un personnage.

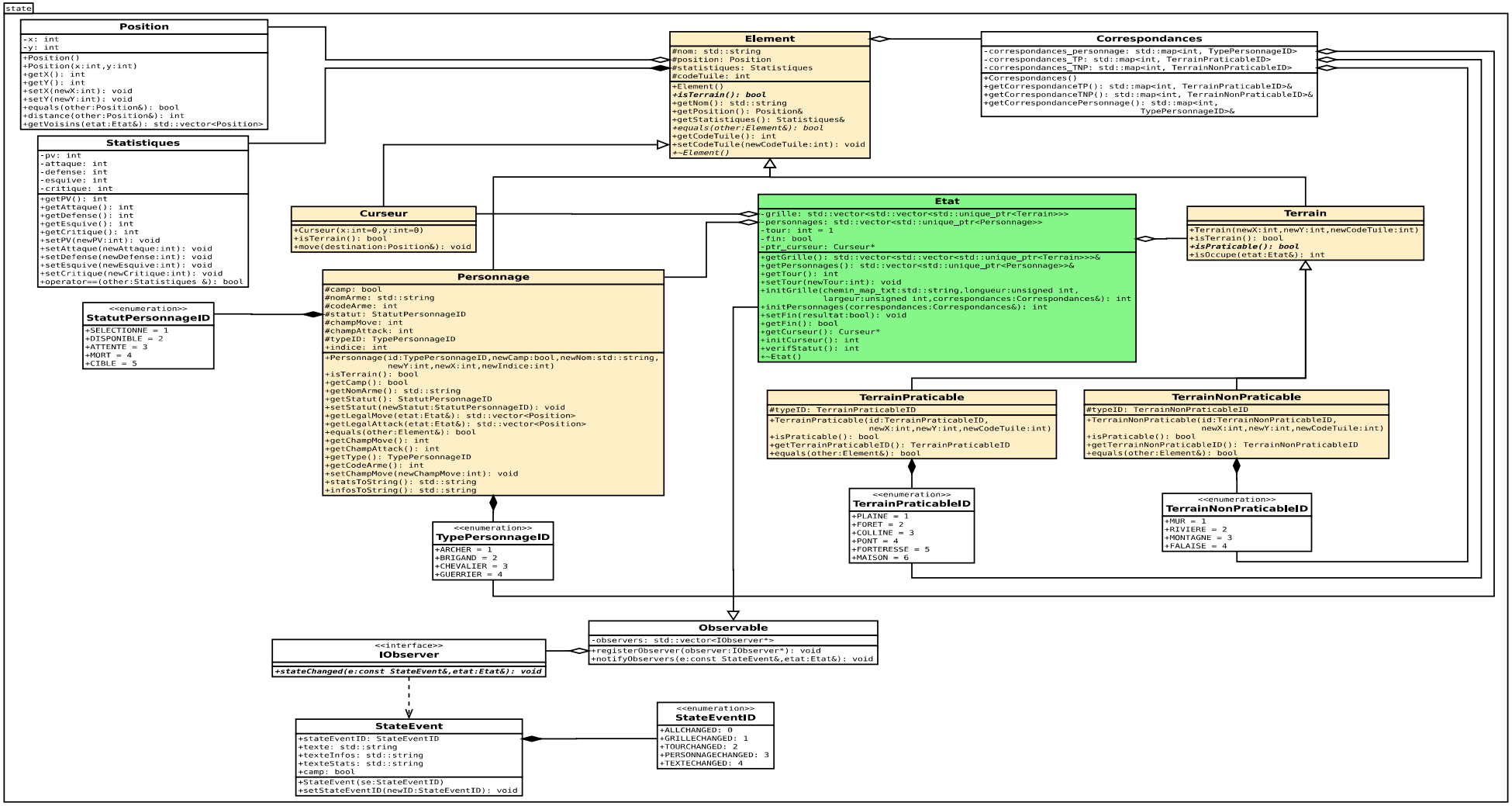

Figure 5 – Diagramme des classes d'état

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous avons fait le choix d'un rendu par tuile à l'aide de la bibliothèque SFML.

Nous divisons la scène à afficher en trois surfaces : une surface contenant les éléments statiques Terrain, une surface pour les éléments mobiles Personnage et une surface pour les informations de jeu (statistiques et nom d'un personnage, tour de jeu…).

La grille de Terrain est créée à partir du fichier texte « ***map1.txt*** » composé de 25x25 codes de tuiles. La classe Correspondances (dans le package state) possède un tableau de correspondance pour les TerrainPraticables et un autre pour les TerrainsNonPraticables. Ces std::map associent à chaque code de tuile un ID de terrain spécifique. Par exemple, la première tuile du tileset pour la grille porte le numéro 0 et représente une MAISON, et cette association est donc répertoriée dans l'attribut correspondanceTP de la classe Correspondances. Pour modifier la grille il suffit donc de modifier les entiers dans le fichier *« **map1.txt** »* (qui se trouve dans res/)**.**

La méthode initGrille de la classe Etat (package state) nous permet de fabriquer pour chaque code tuile du fichier *« **map1.txt** »* le terrain correspondant, de créer un pointeur unique vers cet objet et de l'ajouter à la grille (tableau à deux dimensions contenant des pointeurs de Terrain).

Les personnages eux ne sont pas créés à partir d'un fichier, pour l'instant ils sont créés par la méthode initPersonnages de la classe Etat.

La méthode initCurseur permet d'initialiser un pointeur de Curseur.

3.2 Conception logicielle

Le diagramme des classes pour le rendu général est présenté en Figure 6.

Classe Surface : cette classe possède deux attributs : une texture et un tableau de Vertex (quads) contenant la position des éléments et leurs coordonnées dans la texture. Elle possède les méthodes loadGrille, loadPersonnage et loadCurseur lui permettant d'initialiser ses attributs à partir d'un tableau de Terrain, d'une liste de Personnage ou d'un Curseur et d'un fichier « *grille_tileset.png* », « *personnage_tileset.png* » ou « *curseur_tileset.png* ». La méthode Draw a pour but de dessiner une texture pour ensuite permettre son affichage dans une fenêtre.

Classe TileSet : Cette classe possède plusieurs attributs : un id de type TileSetID, des entiers cellWidth et cellHeight (qui représentent respectivement la largeur et la longueur en pixel d'une tuile) et une chaîne de caractères imageFile (chemin vers un « *fichier.png* »). L'ID peut prendre différentes valeurs :

- GRILLESET
- PERSONNAGETILESET
- CURSEURTILESET
- INFOSTILESET

Suivant l'ID passé en argument du constructeur de TileSet, les attributs cellWidth, cellHeight et imageFile sont initialisés différemment. De plus, les textures sont chargées depuis le fichier correspondant lors de l'appel au constructeur, et donc une seule fois par TileSet. Les méthodes getCellWidth, getCellHeight et getImageFile permettent de récupérer ces attributs.

Classe StateLayer : Cette classe possède un attribut etatLayer qui est une référence à une instance d'Etat. Elle possède également un tableau de pointeurs de TileSet et un autre de pointeurs de Surface. Le but de cette classe est de créer trois surfaces grâce à la méthode initSurface et d'initialiser leur texture. Cette classe est un observateur, elle implémente l'interface IObserver pour être avertie des changements d'état. Elle réagit ensuite en actualisant les textures.

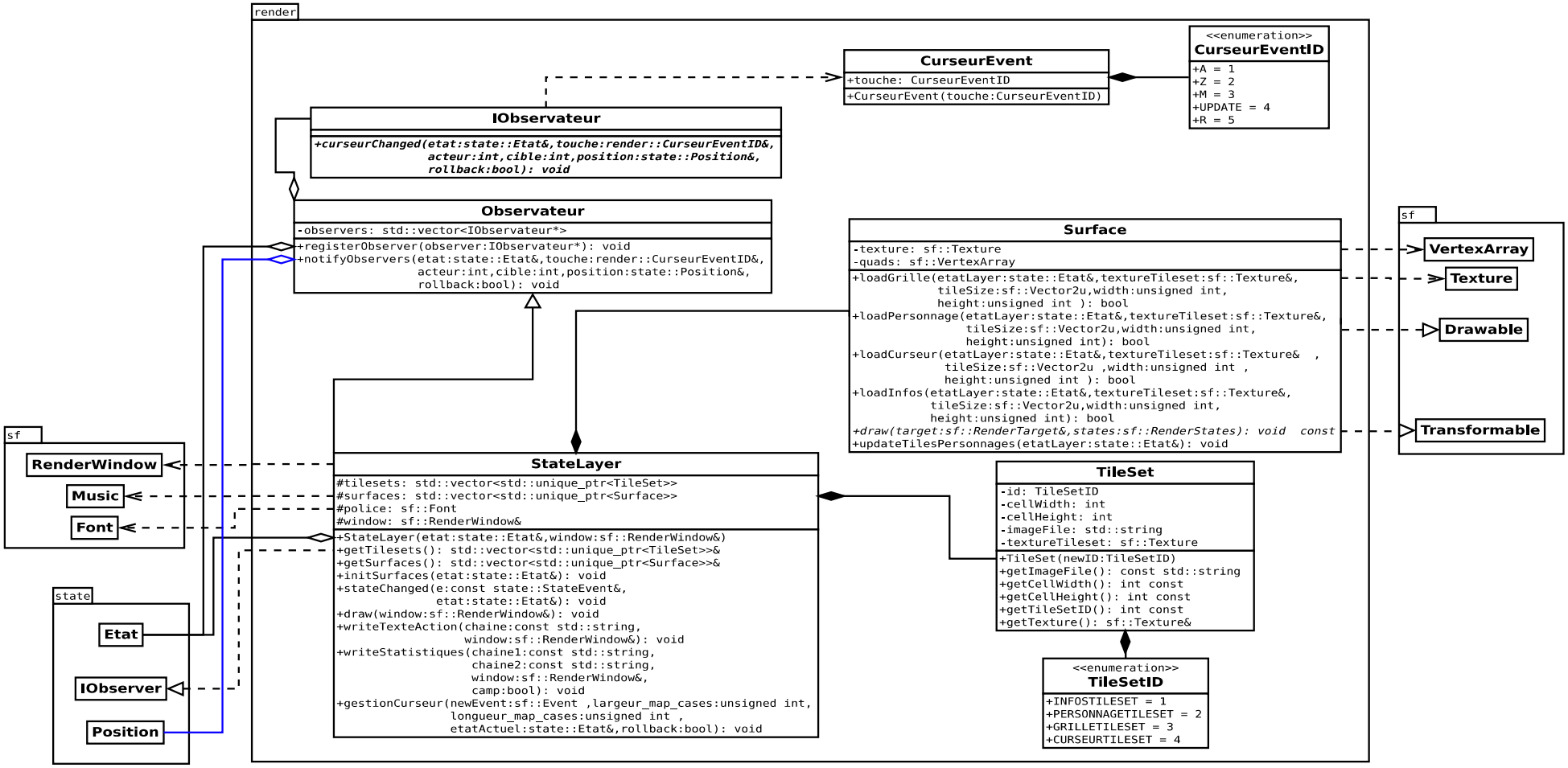

Figure 6 – Diagramme des classes de rendu

4 Règles de changement d’états et moteur de jeu

4.1 Règles de changement d’états

En début de tour, tous les personnages d’un joueur possèdent le statut DISPONIBLE. Ils peuvent donc tous être SÉLECTIONNÉS. Lorsqu’un personnage est SÉLECTIONNÉ par un joueur il peut effectuer selon les cas au moins une des 3 actions listées ci-dessous :

- Se déplacer
- Attaquer
- Terminer ses actions (son tour)

Un personnage ne peut se déplacer que d’une case par une case (sur des terrains praticables et inoccupés) . Il possède des points de mouvements qui correspondent au nombre de déplacement maximum qu’il lui est possible de faire. Lorsque tous ses points de mouvement ont été utilisés, il lui est impossible de continuer son déplacement.

Un personnage ne peut attaquer un autre personnage que lorsque celui ci se trouve dans son champ d’attaque et appartient au camp adverse (les attaques alliées ne sont pas autorisées).

Lorsqu’un personnage termine ses actions, son statut devient ATTENTE. Cela signifie qu’il ne pourra plus être SÉLECTIONNÉ (et ne pourra donc plus effectuer d’actions) avant le prochain tour du joueur.

L’action « Terminer son tour d’action » doit obligatoirement être effectuée par chaque personnage encore actif à chaque tour du joueur (même lors du dernier tour lorsque tous les adversaires ennemis sont morts, c’est ce qui déclenchera alors la victoire). Il existe 4 enchaînements d’actions possibles :

- Attaquer directement (lorsque cela est possible) ce qui termine automatiquement le tour d’un personnage
- Effectuer un ou plusieurs déplacements , attaquer puis terminer son tour automatiquement
- Effectuer un ou plusieurs déplacements puis terminer son tour manuellement
- Terminer son tour directement sans avoir effectuer aucune autre action

Le tour de jeu d’un joueur est terminé lorsque tous ses personnages sont en ATTENTE. C’est alors le tour du joueur adverse.

Lorsqu’un personnage est attaqué par un ennemi, il tente toujours une contre-attaque (à moins d’avoir été achevé durant la première attaque).

Si un personnage perd tous ses points de vie, son statut évolue et prend la valeur MORT.

Si tous les personnages d'un joueur meurent, la partie est terminée à la fin du tour adverse et le joueur adverse gagne.

Chaque action effectuée modifie l'état. Le choix du personnage sélectionné et des actions effectuées est défini par des commandes.

Les commandes du joueur sont provoquées par une pression sur une touche.

- Déplacer le curseur : flèches « haut/bas/gauche/droite ».
- Sélectionner un personnage : « Enter » (lorsque le curseur encadre le personnage).
    - Déplacer un personnage : flèche « haut/bas/droite/gauche » (le personnage doit avoir été sélectionné au préalable).
    - Attaquer : « A » puis déplacer le curseur avec les flèches sur la case d'un adversaire puis « Enter » pour confirmer l'attaque (le personnage doit avoir été sélectionné au préalable).
        - Annuler une attaque : Appuyer sur « N ».
    - Terminer le tour d'actions d'un personnage : « Z » (le personnage doit avoir été sélectionné au préalable).

4.2 Conception logicielle

Le diagramme des classes pour le moteur de jeu (« ***engine.dia*** ») est présenté en Figure 7. Le moteur de jeu repose sur le Design Pattern Command.

Classe Moteur : C'est le cœur du moteur. Elle permet de stocker les commandes dans une std::map avec clef entière (avec « addCommande »). Ce mode de stockage permet d'introduire une notion de priorité : on traite les commandes dans l'ordre de leur clef (de la plus petite à la plus grande). Lorsque la méthode « update » est appelée, le moteur appelle la méthode « execute » de chaque commande puis supprime toutes les commandes une fois executées. La méthode « verificationDebutTour » change le joueur actif, réinitialise les points de mouvement de tous les personnages du joueur et procède à la récupération des PV des personnages se trouvant sur des cases MAISON ou FORTERESSE. La méthode « verificationFinTour » incrémente le nombre de tours si tous les personnages restants du joueur actif sont en ATTENTE et déclare la partie terminée si tous les personnages du joueur adverse sont MORTS.

La classe Moteur a été étendue pour permettre une exécution dans un thread séparé.

Classes Commandes : Les classes Attaque, Deplacement et FinAction, héritant de la classe Commande, possèdent chacune une méthode « execute » qui fait effectuer à un personnage l'action correspondante.

Classes Actions : Les classes Attaque_Action, Dep_Action et FinActions_Action héritant de la classe Action, possèdent chacune une méthode « apply » qui change l'état de la même manière que le ferait une Commande, mais possèdent également un méthode « undo » qui remodifie l'état mais en annulant les changements opérés par l'action en question, grâce aux attributs de chaque Action qui correspondent aux changements que provoque l'action.

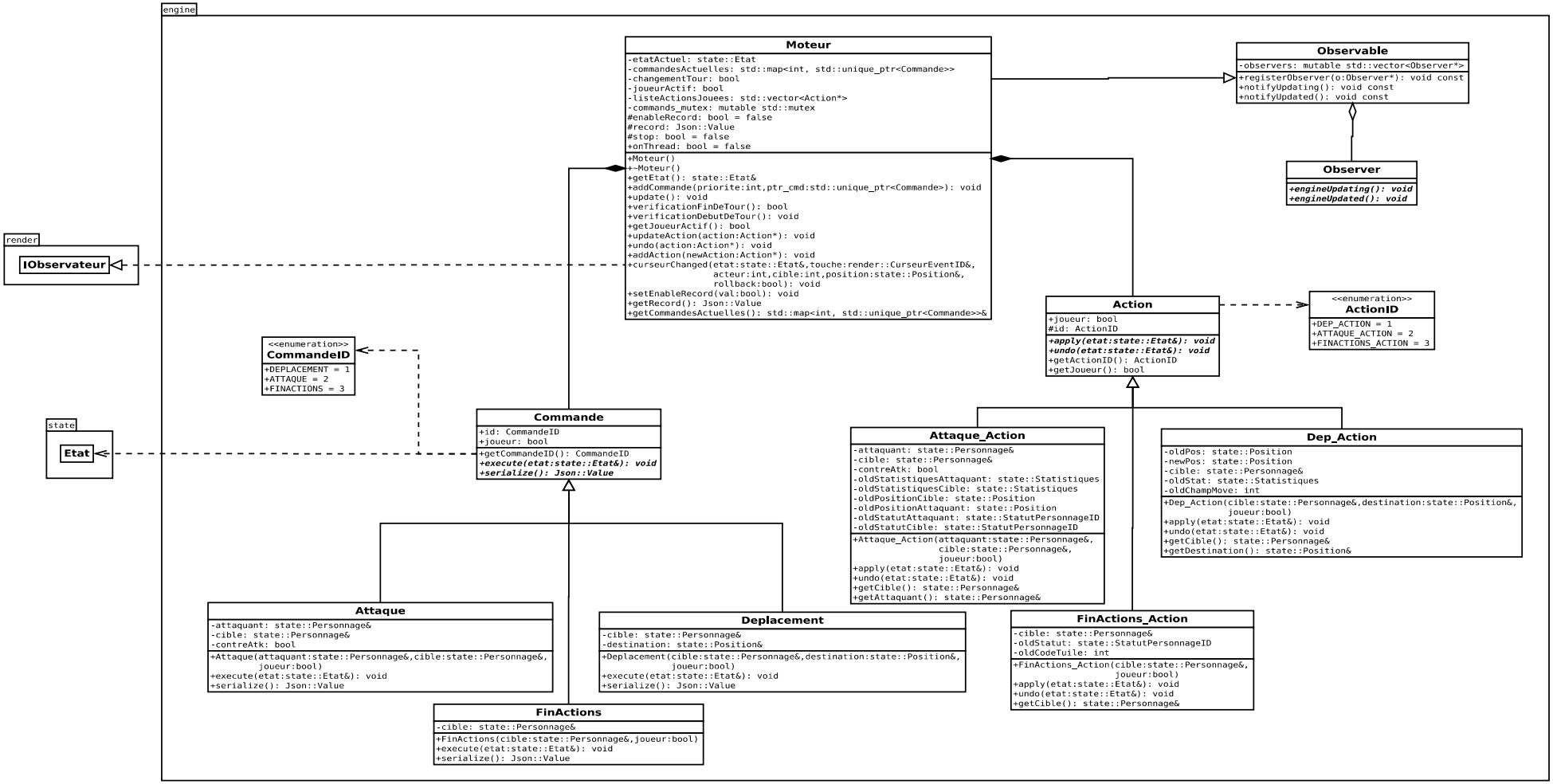

Figure 7 – Diagramme des classes de moteur

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

L' IA contrôle une liste de personnages qu'elle va sélectionner les uns après les autres lors de son tour de jeu. Une fois un personnage sélectionné, un entier correspondant à une action précise est choisi au hasard parmi 3 : 0 (se déplacer), 1 (attaquer), 2 (terminer son tour d'actions).

Si « se déplacer » est choisi et que le personnage sélectionné possède encore des points de mouvement, une destination va être choisie aléatoirement parmi une liste de destinations possibles. Si la liste n'est pas vide, le personnage se déplacera. Sinon il ne pourra pas se déplacer. Dans les deux cas, il restera SÉLECTIONNÉ et l'IA pourra donc continuer à le manipuler (une nouvelle action aléatoire sera tentée après celle-ci).

Si « attaquer » est choisi, une cible va être déterminée aléatoirement parmi une liste de cible possibles (qui diffèrent en fonction de la position du personnage et de son champ d'attaque). Si la liste n'est pas vide, le personnage va attaquer puis terminer directement son tour, comme défini dans les règles du jeu. Il passera donc en ATTENTE ou MORT (s'il est décédé des suites de son attaque), l'IA ne pourra plus le manipuler et sélectionnera le personnage suivant. Si la liste d'attaque est vide, l'attaque n'aura pas lieu, le personnage restera SÉLECTIONNÉ et l'IA pourra donc continuer à le manipuler (une nouvelle action aléatoire sera tentée après celle-ci).

Si « terminer son tour d'actions » est choisi, le personnage va terminer son tour d'actions directement. Il passera donc en ATTENTE. L'IA ne pourra plus manipuler ce personnage et sélectionnera le personnage suivant.

Le tour de l'IA est terminé lorsque tous ses personnages ont été sélectionnés et sont en ATTENTE ou MORT. L'IA pourra de nouveau sélectionner ses personnages (à condition qu'ils ne soient pas morts) au tour suivant et ce schéma se répétera jusqu'à la fin de la partie.

5.1.2 Intelligence basée sur des heuristiques

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

On privilégie presque toujours l'attaque au déplacement et le déplacement à la fin du tour d'actions.

Une fois un des personnages sélectionné, on calcule un score pour chacun des personnages adverses à partir des statistiques du personnage sélectionné (son attaque, son taux de coup critique), de celles de l'adversaire (sa défense, son taux d'esquive, ainsi que le bonus d'arme que peut obtenir le personnage sélectionné contre cet ennemi) ainsi que de la distance entre ces deux personnages. Ainsi, plus une attaque peut être efficace contre un des personnages adverse plus le score sera élevé. Le personnage sélectionné prend donc ensuite pour cible l'adversaire pour lequel ce score est le plus élevé pour ce tour de jeu.

Si le personnage sélectionné a la possibilité d'attaquer sa cible c'est-à-dire si le personnage adverse ciblé se trouve dans son champ d'attaque, il le fera en priorité. Si plusieurs personnages adverses se trouvent dans son champ d'attaque il attaquera celui contre lequel il a le plus grand score calculé.

Si le personnage peut attaquer mais que son nombre de PV est trop faible (inférieur ou égal à 5 PV), il va tenter de se déplacer vers le refuge le plus proche (MAISON ou FORTERESSE) pour récupérer des PV. S'il n'y a aucun refuge sur la map ou si tous sont indisponibles, le personnage prendra le risque d'attaquer ou terminera son tour.

Si le personnage ne peut pas attaquer immédiatement sa cible, il va tenter de s'en approcher avec des déplacements définis après un calcul de chemin réalisé grâce à un algorithme A* (algorithme de recherche d'un chemin de coût minimal). Si ces déplacements ne sont pas possibles, il attaquera les autres ennemis à sa portée qui le bloquent.

Si le personnage ne peut ni attaquer ni se déplacer vers son objectif (s'il est bloqué par des alliés ou des murs par exemple), il va rechercher un nouvel objectif et refaire tous les tests précédents pour s'en approcher.

Si aucun des objectifs n'est atteignable, il terminera on tour.

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 minimax pour que l'IA détermine quels sont les choix de commandes optimaux en prenant en compte ce que va faire le joueur au prochain tour.

La stratégie est la suivante : on va simuler toutes les suites de coups possibles de l'IA, puis pour chacune de ces suites d'actions, toutes les suites de coups possibles du joueur. Une fois la simulation du tour des deux adversaires terminée, on appelle une fonction d'évaluation qui attribue un score à cette suite d'actions en fonction de l'état du jeu après simulation. Plus ce score est élevé, plus le scénario simulé (suite de coups de l'IA puis réponse du joueur) est intéressant à jouer pour l'IA.

On simule alors un graphe virtuel où chaque nœud correspond à une action simulée et dont les nœuds suivants sont les actions possibles après cette action. Une branche est donc une succession d'actions de l'IA puis du joueur, soit la simulation d'un tour complet de l'IA puis d'un tour complet du joueur.

A chaque arrivée en fin de branche, on compare son score à celui qui est pour l'instant considéré comme optimal et si le nouveau score est meilleur, on garde en mémoire ce nouveau score et la suite d'actions qui a conduit à cet état optimal pour l'IA.

Pour passer d'une branche à une autre, une fois qu'une branche est simulée en entier et que son score est calculé, on annule la dernière action et on revient donc à l'état précédent, un nœud plus haut dans le graphe, de manière récursive.

En effet, lors de ces simulations, ce sont des Actions qui sont effectuées et non des Commandes. Les Actions ont la particularité de pouvoir être annulées car chaque action garde en mémoire ce qu'elle a modifié dans l'état.

5.2 Conception Logicielle

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 8.

Classe IA : Les classes filles de la classe IA implémentent différentes stratégies d'IA. Elles possèdent un camp et une fonction « run ».

Classe RandomIA : Classe qui implémente l'IA aléatoire.

Classe HeuriticIA : Classe qui implémente l'IA heuristique. La fonction findObjectif définit quel sera l'objectif du personnage actif, que ce soit la cible de sa prochaine attaque ou la position du refuge le plus proche si ses PV sont trop faibles. L'algorithme A* permettra dans les deux cas de trouver rapidement le meilleur chemin vers l'objectif.

Classe Sommet : Un sommet est une Position qui possède un prédécesseur. Des instances de Sommet sont utilisées dans la recherche d'un chemin.

Classe DeepIA : Classe qui implémente l'IA avancée. Les déplacements sont définis comme pour l'IA Heuristique. La fonction findActionsPossibles liste toutes les actions possibles pour l'IA ou le joueur à un état donné et va simuler toutes les successions de coups possibles de manière récursive pour trouver quelle est la suite d'actions optimale pour l'IA avec un algorithme minimax.

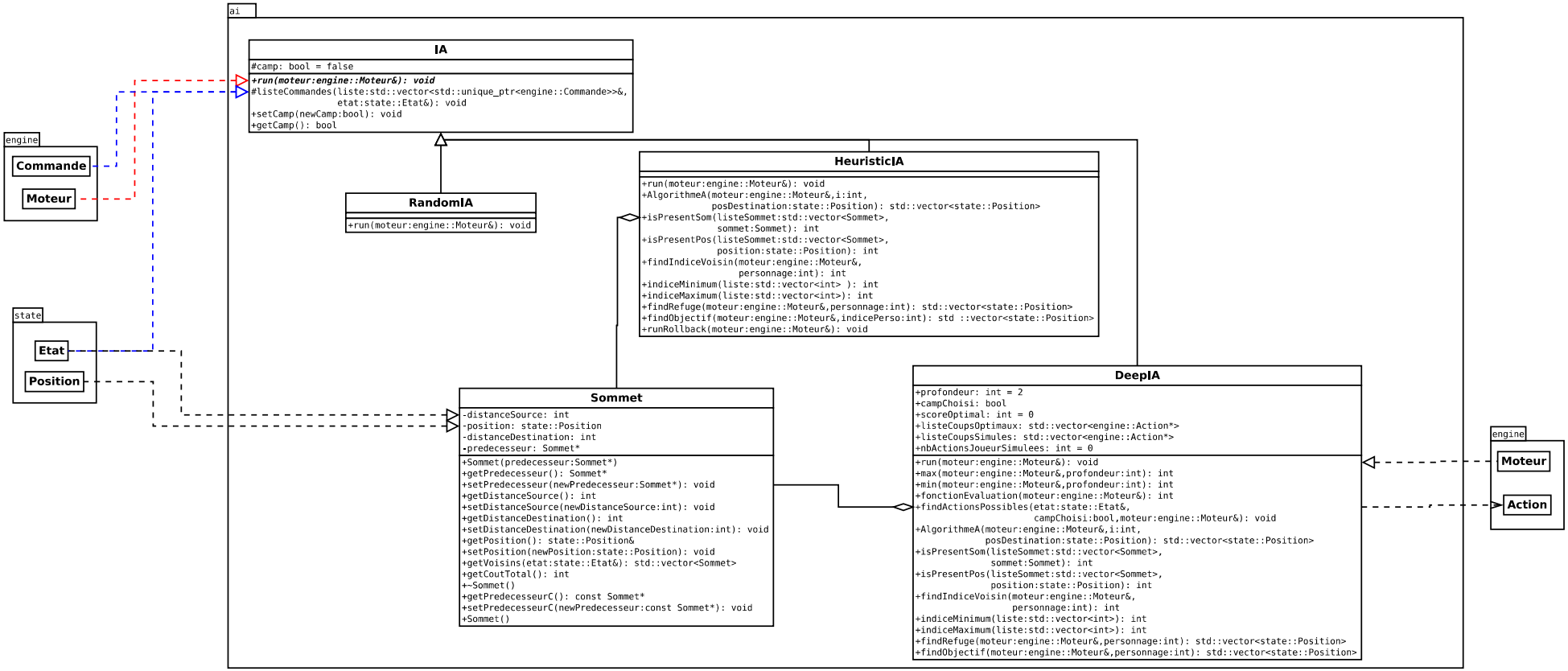

Figure 8 – Diagramme des classes d'intelligence artificielle

6 Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

Notre objectif est ici 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. Le moteur de jeu est le plus souvent passif, il ne devient actif que lorsqu'il lui faut exécuter la liste de commandes remplie par chaque IA. La liste de commandes étant une ressource critique, à ce moment là il ne doit plus être possible d'ajouter des commandes à la liste de commande.

Nous avons deux types d'informations qui transitent d'un module à l'autre : les commandes et les notifications de rendu.

6.1.2 Répartition sur différentes machines : rassemblement des joueurs

On propose une API Web REST pour permettre de 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 clients pour le serveur. Pour ce faire, nous formons des services CRUD sur la donnée "joueur" via une API Web REST :

- **Requête GET/player/<id>** : Si l'id n'est pas attribué à un joueur, cela renvoie un message d'erreur «Invalid player id ». Si l'id est bien attribué, cela renvoie les caractéristiques du joueur correspondant au format Json :

```
{
“name” : “nom_du_joueur ”,
“free” : true/false
}
```

- **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 . Dans le cas de notre jeu le nombre maximal de joueur est 2 et si on essaye d'ajouter un troisième joueur le message suivant s'affiche : «Aucune place libre » . Dans le cas où la requête est valide et le nombre de joueurs présents inférieur à 2, l'id attribué au nouveau joueur est renvoyé au format Json :

```
{
“id”: numero
}
```

Dans ce cas une requête GET est nécessaire pour afficher les caractéristiques du joueur ajouté.

- **Requête POST/player/<id>** : Modifie une ou plusieurs caractéristiques d'un joueur existant. Si l'id ne correspond à aucun joueur, le message d'erreur « Invalid player id » va s'afficher. Sinon il n'y a pas de valeur de retour et une requête GET est nécessaire pour afficher les nouvelles caractéristiques du joueur.
- **Requête DELETE/player/<id>** : Supprime un joueur de la liste des joueurs à condition que ce joueur fasse partie de la liste des joueurs. Si tout s'est bien passé il n'y a pas de valeur de retour.

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. Le client dont ce n'est pas le tour va envoyer régulièrement au moteur du serveur des requêtes pour savoir si son adversaire a joué de nouvelles commandes afin de mettre à jour son propre moteur et son état de jeu. Le serveur n'est alors qu'un relais de commandes.

- **Requête GET/command/<id>** : Récupère les commandes que le joueur qui a pour id « id » a joué et donc envoyé au moteur du serveur.
- **Requête POST/command** : Permet d'envoyer une commande d'un joueur au serveur.

6.2 Conception logicielle

Le diagramme des classes pour la modularisation est présenté en Figure 9.

Classe Client : 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.

On fait jouer deux IA heuristiques l'une contre l'autre. Lorsque l'une des IA a ajouté des commandes à la liste des commandes du moteur et qu'elle souhaite les voir s'exécuter, la méthode engineUpdating est appelée pour que le moteur alors en attente dans le thread secondaire devienne actif et exécute toutes les commandes de sa liste de commandes avant de se remettre en attente.

Lorsque les commandes sont exécutées par le moteur, ce dernier envoie une notification au rendu du thread principal afin de mettre à jour l'affichage.

Classe NetworkClient : 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. A chaque frame rendue, elle demande de manière périodique au serveur si des commandes sont disponibles. Si c'est le cas, elle les exécute.

Classe Game : 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 différentes méthodes parlent d'elles-mêmes, et correspondent aux quatre opérations CRUD usuelles.

Une fois la partie démarrée, le moteur de jeu tourne en boucle, et ne fait aucune notification : c'est aux clients de demander régulièrement. Pour être capable de donner les

commandes, nous avons activé le système d'enregistrement qui mémorise les commandes en JSON. Ainsi, elles peuvent très facilement être délivrées : il suffit de prendre l'élément dans le tableau.

Classe Player : La classe Player contient les caractéristiques d'un joueur : son nom et sa présence ou non dans le jeu (attribut Player : :free).

Classe 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, modifier, consulter et supprimer des joueurs.

— CommandService : Permet d'ajouter, et consulter les lots de commandes.

— GameService : fournit un unique service de consultation de l'état du jeu.

Classe ServiceException : cette classe permet de jeter une exception à tout moment pour interrompre l'exécution du service, de la manière suivante :

throw new ServiceException(<code status HTTP>, <message>);

Le code status HTTP est une des valeurs de l'énumération HttpStatus, et le message une chaîne de caractère explicative.

Par exemple : throw ServiceException (HttpStatus : :NOT FOUND, " Invalid player id ") ;

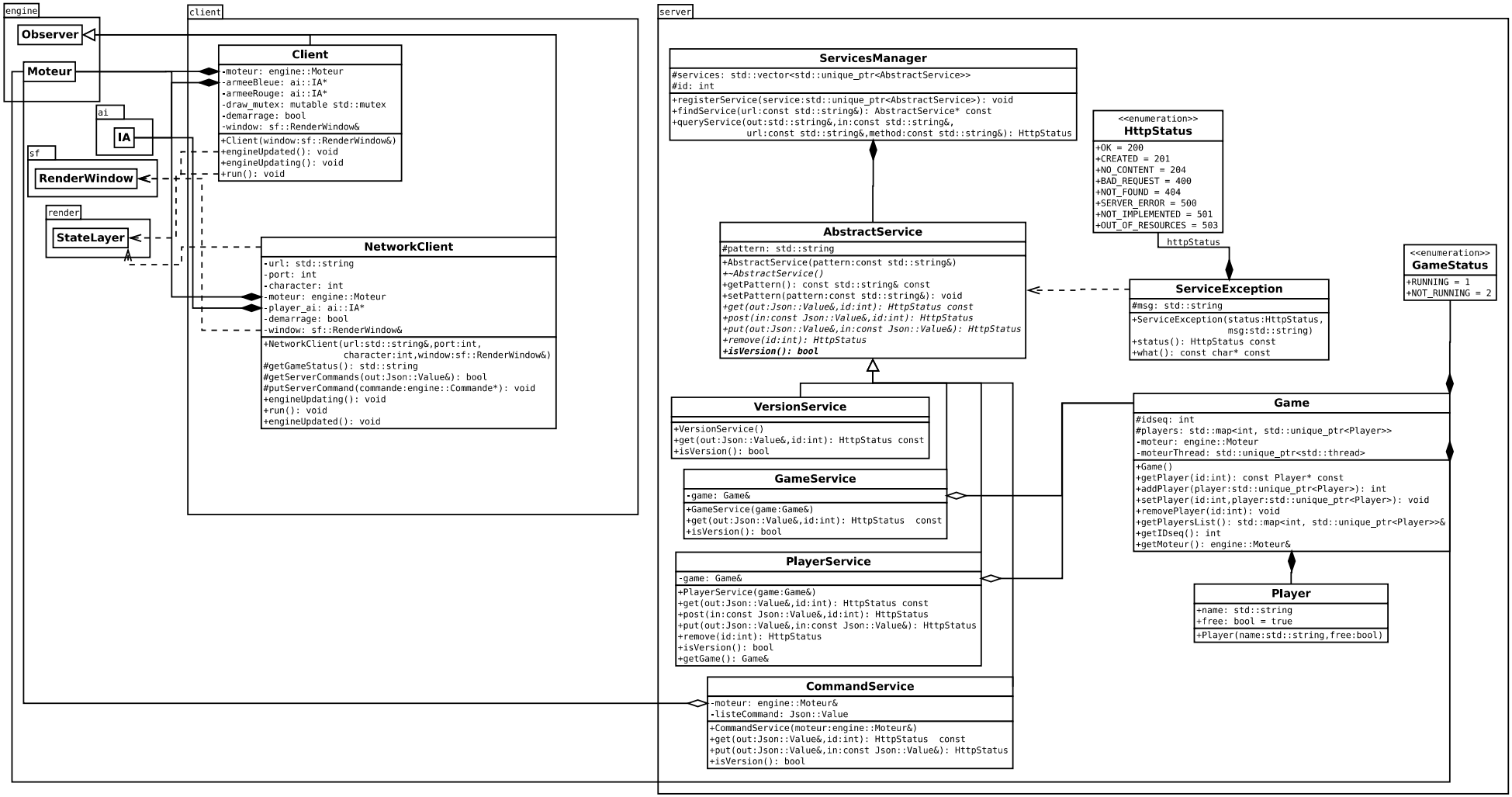

Figure 9 – Diagramme des classes pour la modularisation