Ecole Nationale
Supérieure
de l'Electronique
et de ses Applications

Projet Advance Wars

Pierre MARIET - Joanna XANH
3ème année, Option Informatique et Systèmes

18 janvier 2018

Table des matières

Chapitre 1

Présentation générale

1.1 Archétype

Le projet est de recréer le jeu Advance Wars™ développé par Intelligent Systems® pour la Game Boy Advance™ de Nintendo®.
Le jeu est en tour par tour, sur une carte découpée en cases. Le but du jeu est de capturer le Quartier Général adverse. Il est aussi possible de gagner par forfait. Les cartes pourront être crées à partir d'un fichier texte.

1.2 Règles du jeu

1.2.1 Description des unités

Sur une carte composée de différents terrains et séparée en cases s'affrontent 2 armées. Les terrains augmentent ou non la défense de l'unité qui se trouve dessus. Ces armées sont composées de différentes unités qui ont chacune des spécificités qui varient :

- Les unités aériennes peuvent se déplacer n'importe où sur la carte. Les unités de transport aérien ont une capacité de 1 et ne peuvent débarquer en mer ou en montagne.
- Les unités navales ne peuvent se déplacer qu'en mer, pas dans les rivières. Les unités de transport naval peuvent débarquer uniquement sur une plage ou dans un port.
- Les unités d'infanterie et bazookas peuvent se déplacer sur tous les sols et traverser les rivières.
- Les unités terrestres motorisées ne peuvent franchir les montagnes.

1.2.2 Les généraux et leurs pouvoirs

Chaque armée est dirigée par un général possédant un pouvoir qui donne un avantage à son armée ou un inconvénient à l'ennemi. Ce pouvoir dure un tour (sauf dans pour le Blizzard) et se recharge sur plusieurs tours. Le général donne en plus un avantage moindre, ou non, mais en rapport avec son pouvoir et qui dure toute la partie.

Réparation Redonne 2 points de vie à toutes les unités alliées. Aucune autre spécificité en dehors du pouvoir. Les statistiques de ce général représentent les statistiques moyennes.

Blizzard Provoque une tempête de neige qui dure deux tours et qui diminue l'amplitude de déplacement des unités adverses. Aucune autre spécificité en dehors du pouvoir.

Marche forcée L'infanterie peut se déplacer d'une case en plus et les bazookas ont une force de frappe augmentée de 50%. Par défaut, l'infanterie et les bazookas ont un bonus de force de frappe de 20% et de 10% de défense contre les unités d'infanterie adverse. Une unité qui capture un bâtiment a vitesse de capture de 150% de ses points de vie. La puissance de frappe des autres unités de tir direct est en revanche moins élevée que la moyenne. Les unités de transports peuvent se déplacer d'une case en plus par rapport à la moyenne.

Puissance Max Les chars d'assaut voient leur puissance de frappe augmenter de 40%, leur défense augmente de 1 point et leur amplitude de déplacement augmente de 1 case. Par défaut, les chars d'assaut ont une puissance de frappe 50% plus élevée que la moyenne.

Embuscade Les unités de tir indirect voient leur portée augmenter de 1 et leur force de frappe augmenter de 50%. Par défaut, la portée des unités de tir indirect est augmentée de 1 case. Les unités de tir direct ont une force de frappe diminuée de 20%.

Frappe éclair Les unités aériennes peuvent se déplacer une nouvelle fois pendant le tour. Par défaut, la force de frappe des unités aériennes est 15% plus élevée que la moyenne Les forces navales ont une force de frappe réduite de 20% par rapport à la moyenne.

FIGURE 1.1 – Sprites des généraux

1.3 Ressources

Les ressources graphiques utilisées sont celles du jeu d'origine. Néanmoins, en considérant le temps de développement des autres paramètres du jeu, seuls la carte, les unités sur celle ci ainsi que les animations de capture seront affichées.

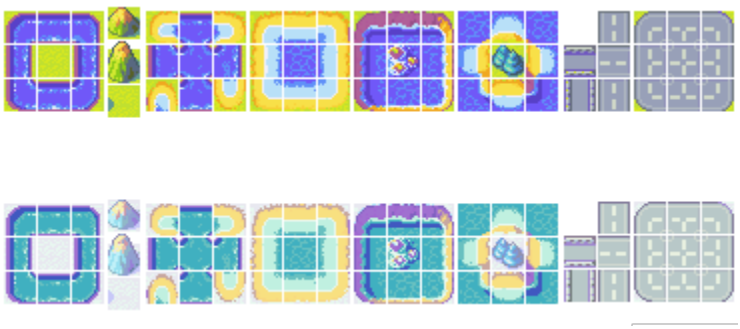

FIGURE 1.2 – Sprites pour les tuiles de la carte

FIGURE 1.3 – Sprites des différents batiments de la carte

FIGURE 1.4 – Sprites des unités ainsi que les informations de statut

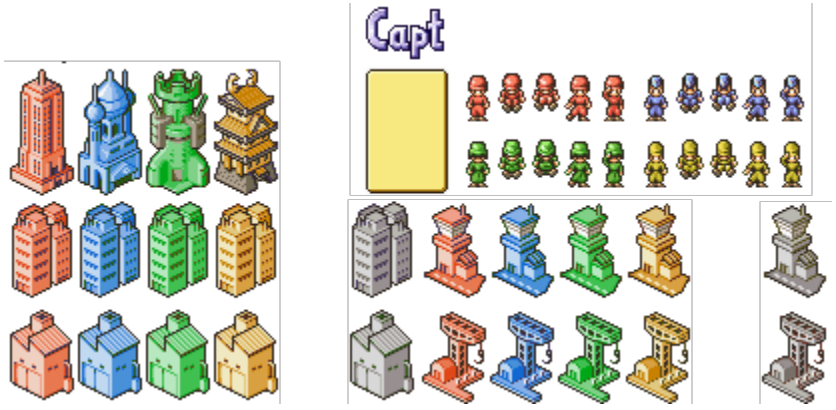

FIGURE 1.5 – Animations de capture

FIGURE 1.6 – Sprites des fenêtres d'attaque

Chapitre 2

Description et conception des états

2.1 Description des états

Un état du jeu est représenté par deux types d'éléments, ceux fixes qui sont les terrains et bâtiments et ceux mobiles qui sont les unités déployées. Ces éléments ont comme point commun des coordonnées et un identifiant pour savoir s'ils sont fixes ou statiques.

2.1.1 État éléments fixes

La carte est une grille d'éléments dont les dimensions sont fixées en début de partie. Toutes les cases ont des attributs communs comme le niveau de défense et le nombre de points de mouvements nécessaires pour franchir ce terrain. Il existe cependant deux types de cases :

Les cases terrain

Il en existe deux grand types : sol et eau. Pour les cases terrain sol, aucune unité navale ne peut se déplacer sur ces cases :

- Plain + Forest + Road + Bridge : Toutes les unités terrestres peuvent traverser. La particularité des cases Bridge est qu'aucune unité navale ne peut la traverser.
- Mountain + River : Seules les unités à pied peuvent traverser.
- Beach : Terrain particulier puisque les Lander peuvent se déplacer sur cette case pour charger et décharger. Toutes les unités terrestres peuvent traverser.

Pour les cases terrain eau, aucune unité terrestre ne peut se déplacer sur ces cases.

-

2.1.2 État éléments mobiles

Les éléments mobiles peuvent se déplacer sur la carte en fonction de leur condition et du nombre de points de mouvement qu'ils ont a disposition. Les unités peuvent passer par dessus d'autres unités alliées pour se rendre sur une autre case. Il ne peut y avoir qu'une unité par case. Deux types d'unités existent :

- **Les unités standard :** Elles peuvent attaquer, capturer, se déplacer selon leur capacité de mouvement et selon terrain sur lequel elles se déplacent.

— **Les unités de transport :** Ces unités peuvent transporter certains types d'unités

2.1.3 État général

Pour l'ensemble des éléments, nous avons :
— les jours : qui correspond aux nombres de tours de jeux.
—

2.2 Conception logiciel

Le diagramme des classes pour les états est présenté en Figure 2.1, les groupes de classes sont les suivants :

— **classe Element :** c'est la classe mère des éléments statiques et mobiles. C'est une classe abstraite puiqu'il ne peut y avoir d'instances de cette classe dans le jeu. Elle contient uniquement une variable permettant d'identifier si l'élément de la classe fille est statique ou non.
— **classe Commander :** cette classe est associée aux classes Building et Unit car elle peut agir sur leur état.
— **classe ElementTab :** c'est une classe qui regroupe plusieurs éléments dans un tableau 2D. Ce tableau est dynamique afin de pouvoir charger un monde à partir d'un fichier texte. Les coordonnées du tableau sont celles qui permettent d'identifier les positions des éléments contenus.
— **classe State :** cette classe regroupe deux tableaux de la classe ElementTab : un pour les éléments statiques, un pour les éléments mobiles. De plus, elle contient un compteur de jours (équivalent des tours) et un timer pour indiquer le temps restant pour jouer un tour.

Tableau des classes :

Classe	Rôle de la classe
Élément	Classe mère de tous les éléments d'états du jeu
Land	Créer les terrains du jeu
Unit	Créer les unités de l'armée
Building	Créer les bâtiments présents dans le niveau
Transport	Créer les unités de transport
Commander	Créer les officiers du jeu qui peuvent apporter des bonus à leur armée
State	Rassemble le tableau des éléments fixes et le tableau des éléments mobiles du jeu
ElementTab	Contient un tableau 2D d'éléments

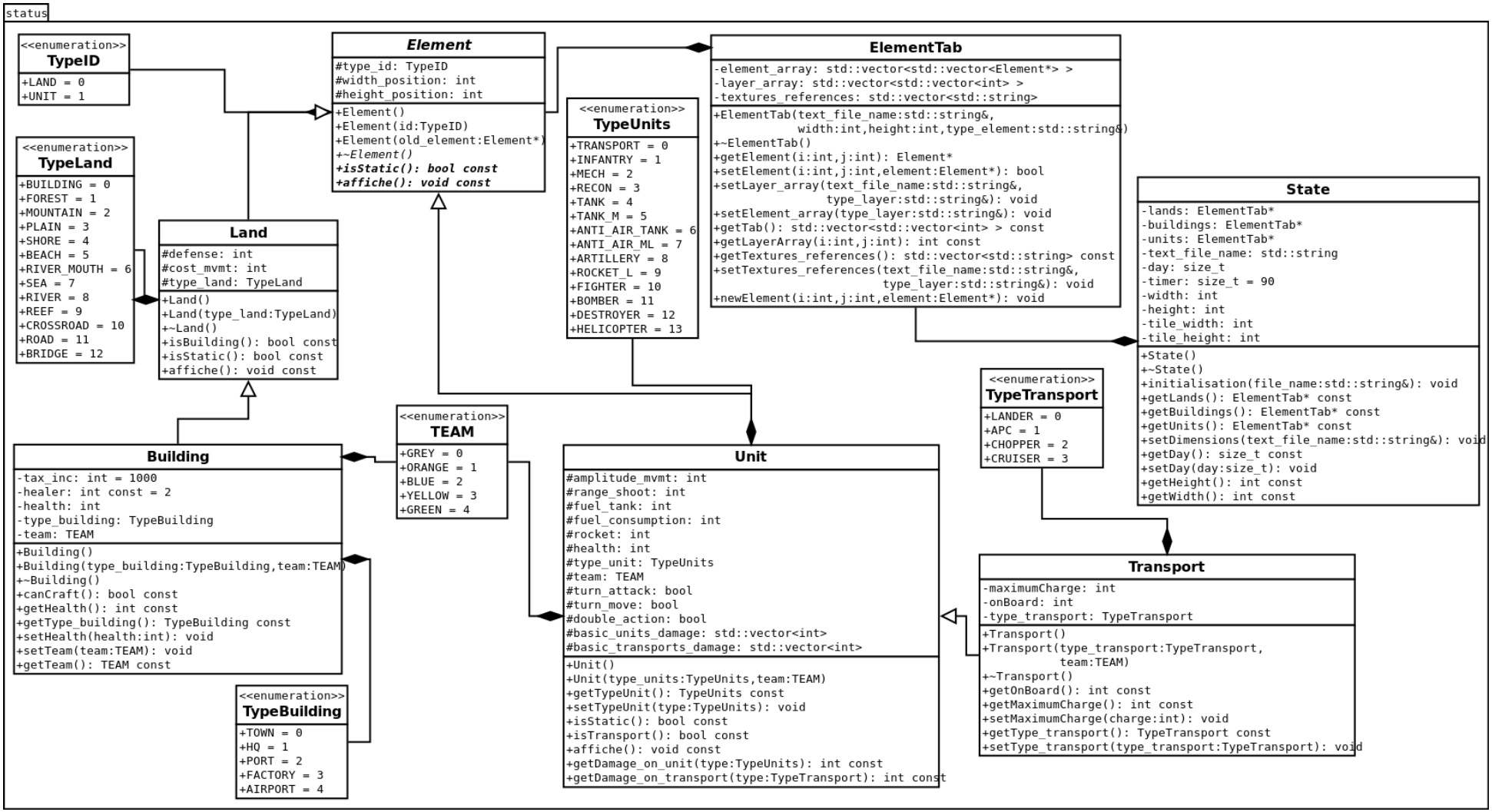

FIGURE 2.1 – Diagramme des classes d'état

Chapitre 3

Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

La stratégie que nous avons établi pour le rendu d'un état est une stratégie de bas niveau. Nos scènes sont coupées en plans : un plan pour le niveau qui contient différents paysages (plaine, montagne, forêt,etc.), un plan d'unités mobiles (infanterie, tank, navire,etc.), et un plan d'informations d'état du jeu (vie, jauge de pouvoir,etc.).. Chaque plan possède différentes informations : une unique texture contenant les tuiles, une unique matrice avec la position des éléments, un tableau de nos plans, le nom du fichier texte à partir du quel nous générons notre état, un tableau des différentes textures, les dimension de la fenêtre ainsi que les dimensions de tuiles.

L'organisation du rendu se fait à partir de l'état et de la détection d'un changement. Si le changement dans l'état donne lieu à un changement permanent dans le rendu, on met à jour le morceau de la matrice du plan correspondant. Pour les changements non permanent, comme les éléments mobiles, nous tiendrons à jour une liste d'éléments visuels à mettre à jour (=> modication de la matrice du plan) automatiquement à chaque rendu d'une nouvelle frame.

3.2 Conception logiciel

Le diagramme du rendu est présent en Figure 3.1.

Plans : Le principal objectif des instances de Layer est de donner les informations basiques pour former les éléments bas-niveau à transmettre à la carte graphique ; ces informations sont données à une instance de Surface, et la dénition des tuiles est contenu dans une instance de TileSet.

La méthode setSurface() fabrique une nouvelle surface à partir de notre fichier texte, grâce aux informations contenues dans ce fichier on peut lui demander d'initialiser la liste des tableaux de vertex. Pour afcher le niveau, elle demande les dimensions de la fenêtre avec initQuads().Puis, pour chaque cellule du niveau, elle xe leur position avec setQuadPosition() et leur tuile avec setQuadTextureCoordinates().

Surfaces : Chaque surface contient une texture du plan et une liste de paires de quadruplets de vecteurs 2D. Les éléments texCoords de chaque quadruplet contient les coordonnées des quatre coins de la tuile à sélectionner dans la texture. Les éléments position de chaque

quadruplet contient les coordonnées des quatre coins du carré où doit être dessiné la tuile à l'écran.

Tuiles : La classe TileSet contient un tableau de toutes les tuiles d'un même plan. Elle peut grâce à la méthode getTile(), récupérer la tuile correspondant à un indice de son tableau, et à partir de cette tuile on peut savoir ses dimensions ainsi que l'emplacement de sa texture dans nos dossiers. Pour obtenir ces informations, un client dispose des méthodes getWidth(), getHeight() et getAdressFile().

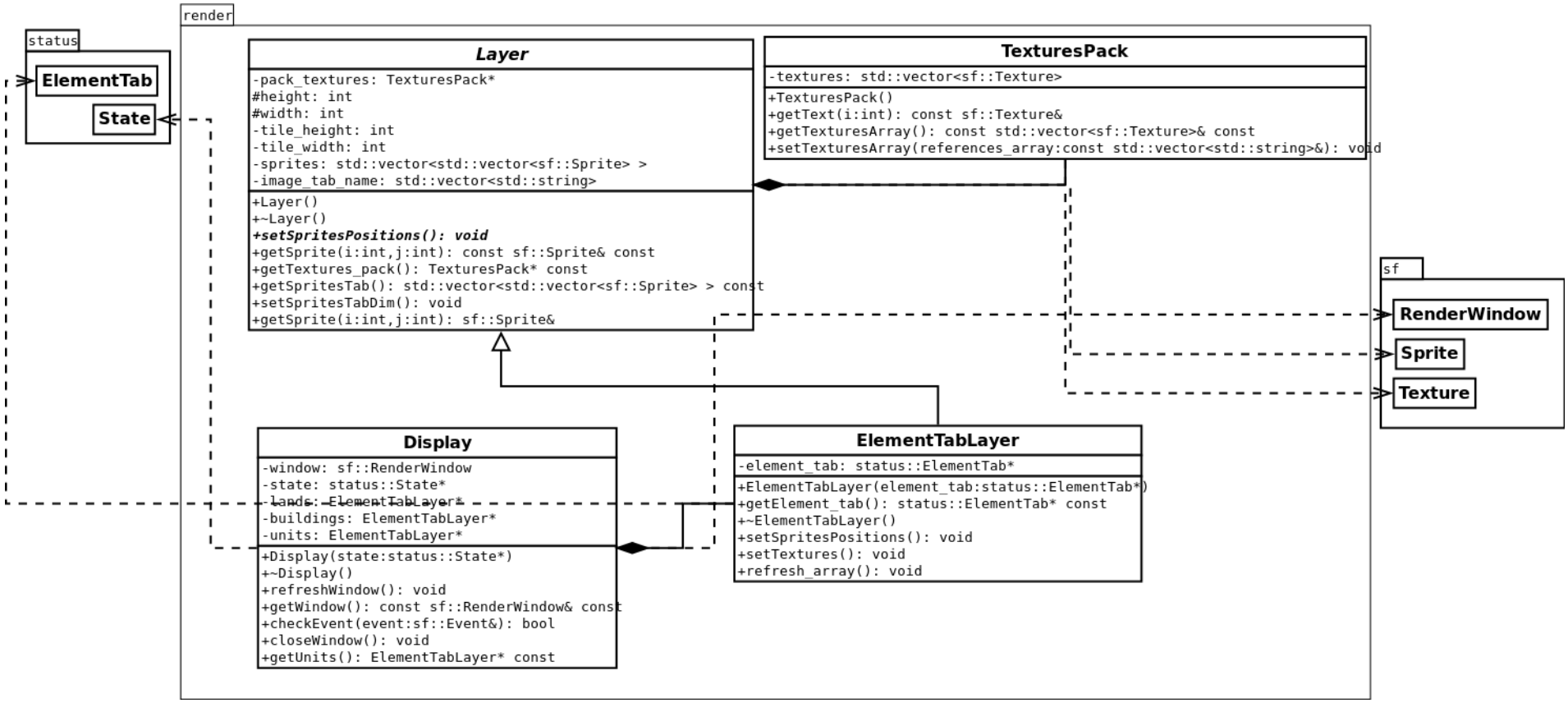

FIGURE 3.1 – Diagramme du rendu

Chapitre 4

Règles de changement d'état et moteur du jeu

4.1 Tours et horloges

Lors du tour d'un joueur, plusieurs actions peuvent être effectuées (il est tout à fait possible de passer sont tour). Une action entraîne un changement d'état. Un changement d'état est instantané mais l'animation d'un déplacement ou d'une attaque introduit un temps de pause pendant lequel le joueur ne peut pas effectuer une autre action. Afin de ne pas bloquer la partie, un temps total par tour doit être imposé. Le jeu introduira donc deux horloges : une pour le décompte du temps de tour, une autre pour afficher un changement d'état qui a déjà été effectué.

4.2 Changement extérieurs

Ce sont les changements demandés par une entrée externe (pression de touche ou réseau). Les différents changements extérieurs peuvent être un chargement de carte, une attaque, un déplacement, la création d'une nouvelle unité, la capture d'un bâtiment, la fusion de deux unités ou encore la fin d'un tour.

4.3 Changements autonomes

Les différents changements autonomes qui peuvent avoir lieux sont :

— la capture d'un bâtiment fait diminuer le nombre de points de capture restant avant qu'il soit acquis
— l'attaque d'une unité par une autre fait diminuer les points de vie de l'unité attaquée et les munitions de l'unité attaquante si les munitions sont spécifiques
— le déplacement d'une unité fait diminuer ses points de carburant/rations en plus de ceux obligatoires à chaque tour
— la fusion d'une unité avec une autre identique fait disparaître l'une des deux, additionne les points de vie et crée une rentrée d'argent si la somme des points de vie dépasse la limite maximale

— un changement de tour crée une rentrée d'argent proportionnelle au nombre de villes possédées

4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Figure 4.1. Le moteur se compose de différentes classes(Command et Action) qui permettent la mise en oeuvre différée de commandes extérieures sur l'état du jeu.

Classes Command : Le rôle de ces classes est de représenter une commande, quelque soit sa source. Pour chacune de ces classes, on a défini un type de commande avec CommandTypeId pour pouvoir identifier la classe d'une instance.

— *LoadMap* : Charge un niveau depuis un fichier ;
— *CreateUnit* : Crée une unité ;
— *MoveUnit* : Déplace une unité ;
— *AttackUnit* : Une unité attaque une unité adverse ;
— *CaptureBuilding* : Une unité capture un bâtiment (neutre ou adverse) ;
— *ResetUnits* : Détruit les unités sur le terrain ;

Classes Action : Chaque commande, lorsqu'elle est exécutée, donne lieu à certain nombre d'actions à réaliser, suivant les cas. Ces actions sont concrétisés par les classes lles de Action. Cela permet d'atomiser les opérations effectives, et donc de les inverser grâce à l'appel aux methodes undo(). Ces classes nous permettent donc d'annuler les conséquences de commandes, et nous servent à remonter dans l'arbre des états pour l'intelligence articielle avancée (algorithme minmax).

Engine : Cette classe correspond au moteur du jeu. Elle stocke les commandes dans une std : :map avec clé entière. Ce mode de stockage permet d'introduire une notion de priorité : on traite les commandes dans l'ordre de leur clés, de la plus petite à la plus grande. Lorsqu'une nouvelle époque démarre, c'est à dire quand on appelle la méthode update(), le moteur appelle la méthode execute() de chaque commande, incrémente l'époque, puis supprime toutes les commandes. Il en résulte une pile d'actions qui, si on les appliquent en ordre inverse, permettent d'annuler les commande précédemment invoquées. Cette classe peut être exécutée dans un thread séparé.

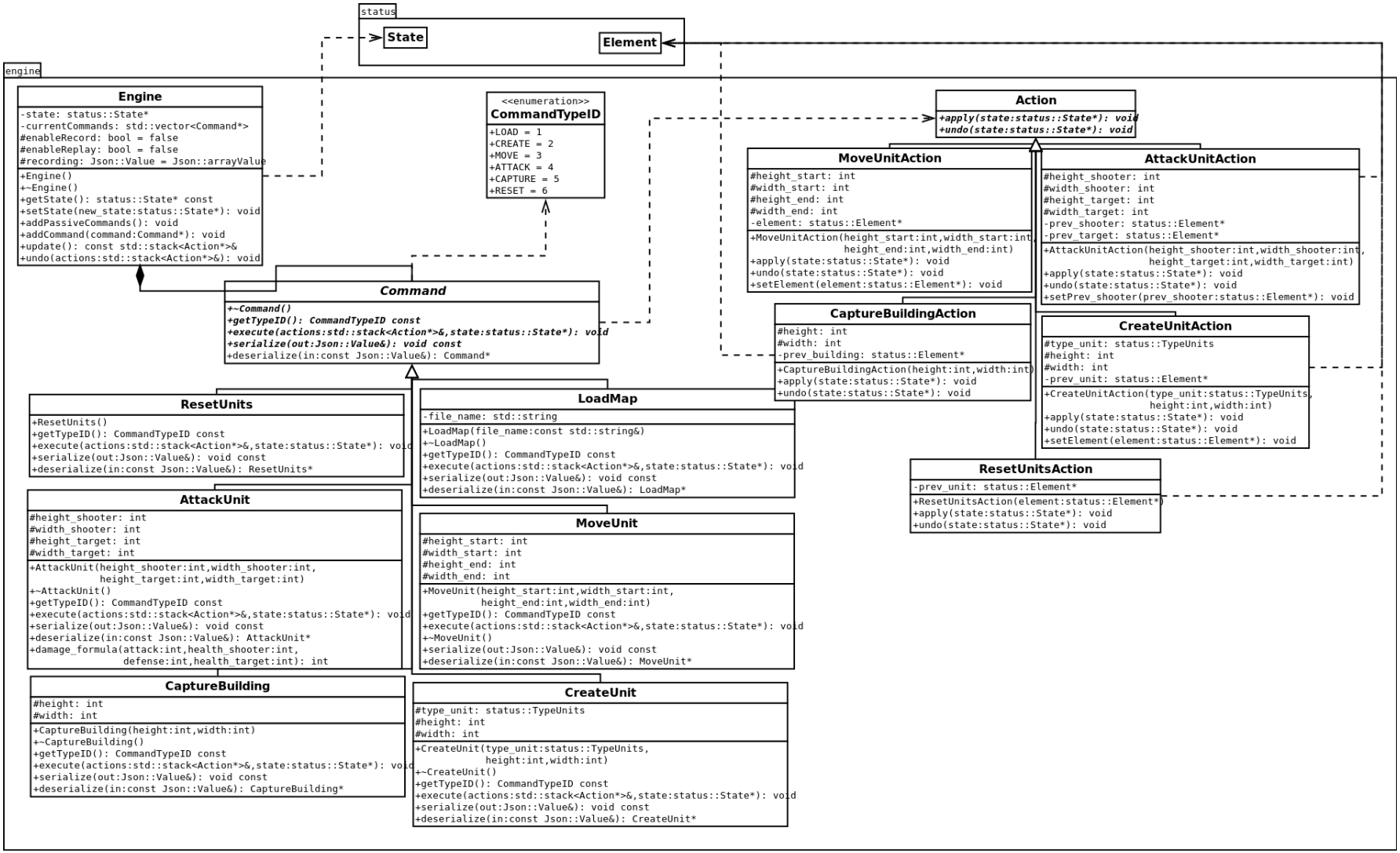

FIGURE 4.1 – Diagramme de l'engine

Chapitre 5

Intelligence Artificielle

5.1 Stratégie

5.1.1 Intelligence artificielle bas-niveau

Le principe consiste à donner une liste de commandes possibles aux unités et de parcourir le tableau des unités afin qu'elles effectuent une action choisie aléatoirement dans la liste de commandes passées en argument. Pour respecter le concept de carte vide au premier tour, un passage dans le tableau des bâtiments sera effectué afin de générer des unités (aléatoirement si possible) et ensuite de parcourir le tableau des unités créées pour effectuer des commandes aléatoires.

5.1.2 Intelligence basée sur des heuristiques

Pour obtenir un comportement meilleur que le hasard, nous voulons permettre à l'IA de rechercher l'ennemi le plus proche pour l'attaquer ou de chercher le bâtiment le plus proche pour le capturer ou soigner l'unité en déplacement.Pour cela nous allons fabriquer des cartes de distances. Pour calculer ces cartes, nous utilisons l'algorithme de Dijkstra.

5.1.3 Intelligence avancée

Nous proposons une intelligence plus avancée en suivant les méthodes de résolution de problèmes à états nis. Dans cette conguration, un état est un état du jeu à une époque donnée, tel que déni dans la section état. Les arcs entre les sommets du graphe d'état sont les changements d'états.. Passer d'un sommet du graphe d'état à un autre reviens à passer d'une époque à une autre du jeu, fonction de l'ensemble des commandes reçues (clavier, réseau, IA,...). L'évaluation/score d'un sommet/état du jeu n'a pas encore pu être déterminée.
Pour que notre AI prenne les meilleures décisions, nous suivons des méthodes basées sur les arbres de recherche, avec une propriété importante. En effet, nous n'envisageons pas de dupliquer l'état du jeu à chaque sommet du graphe d'état. Nous n'allons considérer qu'un seul état que nous modions suivant la direction choisie par la recherche. Si le sommet suivant est à une époque suivante, i.e. on descend dans l'arbre de recherche, on applique les commandes associées, et notre état gagne une époque. Si le sommet

suivant est à une époque précédente, i.e. on remonte dans l'arbre de recherche, on annule les commandes associées, et notre état retrouve sa forme passée. Ces pour cela que plus tôt nous créons les classes Action.

5.2 Conception logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 5.1.
Classes AI : Les classes filles de la classe AI implante différentes stratégies d'IA :
— RandomAI : Intelligence aléatoire
— HeuristicAI : Intelligence heuristique
— DeepAI : Intelligence avancée
PathMap : La classe PathMap permet de calculer une carte des distances à plusieurs unités. Pour chaque case « espace » du niveau, on peut demander un poids qui représente la distance à ces objectifs. Pour s'approcher d'un objectif lorsqu'on est sur une case, il suffit de choisir la case adjacente qui a un plus petit poids. Deux cartes de distantes sont crées en début de tour :
— adv_units_map : Objectif cases unités adverses, pour pouvoir attaquer l'unité la plus proche
— building_map : Objectif building, pour s'emparer des building ennemis ou se soigner si le building nous appartient
DeepAI : Nous proposons ici une implantation pour une IA basée sur la résolution de problèmes à état ni, tel que déni dans la section précédente. L'algorithme utilisé est une version du minimax pour deux joueurs.
(Le critère d'évaluation n'a pas encore été trouvé, car nous voulons évaluer l'avancée d'un joueur sur la distance de ces unités au QG adverse (car le but est de le capturer) mais aussi au nombre d'unité encore présentes sur le terrain ainsi que le nombre de bâtiments possédés, car ce sont des avantages déterminants pour la réussite du joueur).

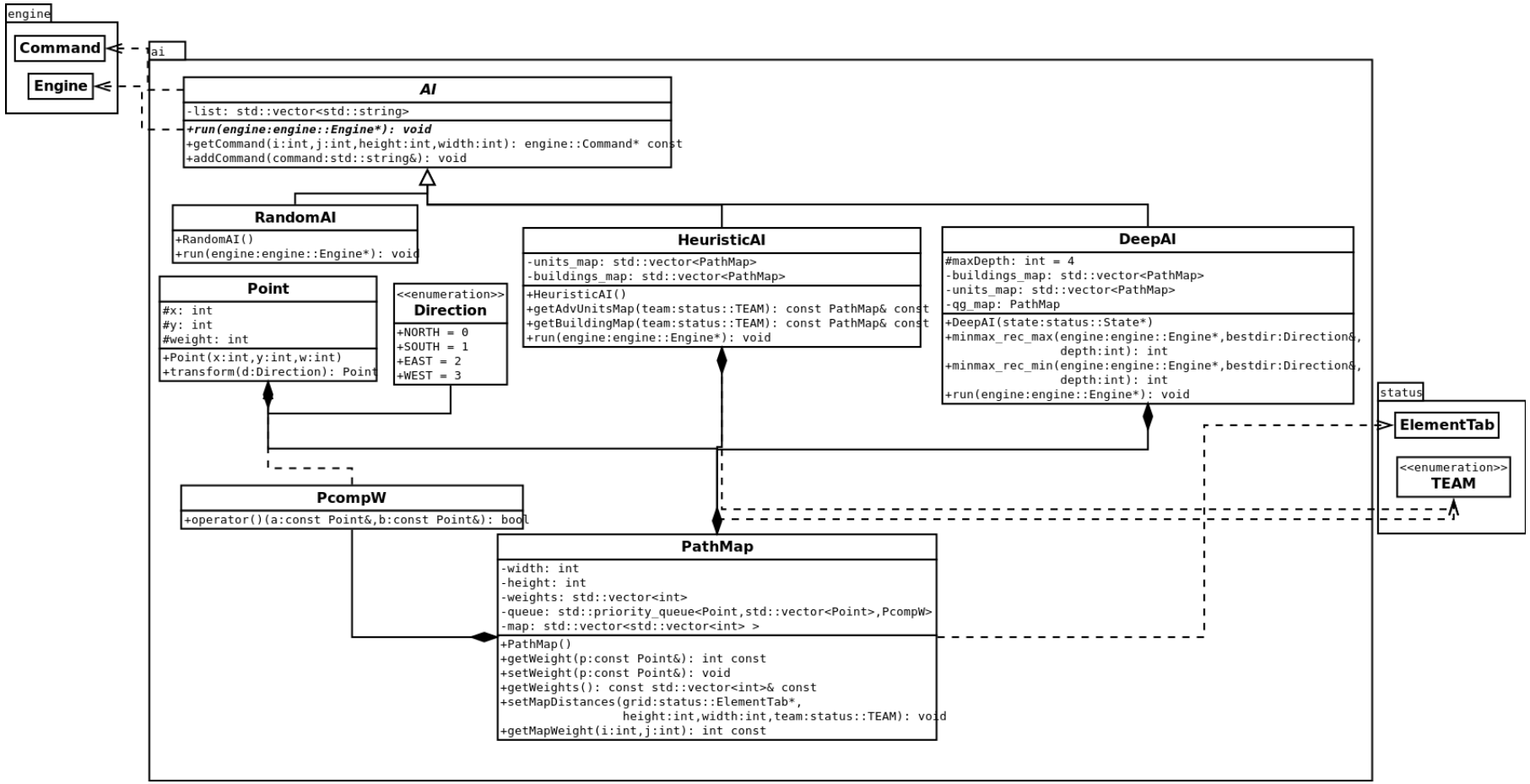

FIGURE 5.1 – Diagramme de l'intelligence artificielle bas-niveau

Chapitre 6

Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

Pour notre jeu nous avons choisi de placer notre moteur de rendu sur un thread principal et le moteur de jeu sur un thread secondaire. Pour cela nous utilisons deux types d'information : les commandes et les notifications de rendu.

Commandes : Nos commandes peuvent arriver n'importe quand même à la mise à jour d'un état du jeu. Nous utilisons donc un double tampon de commandes. L'un contenant les commandes actuellement traitées lors d'une mise à jour de l'état ; l'autre contenant les nouvelles commandes. A chaque nouvelle mise à jour de l'état du jeu, on copie les commandes d'un tampon à l'autre.

Notifications : (pas encore mis en place)

6.1.2 Répartition sur différentes machines : rassemblement des joueurs

Pour jouer en réseau nous devons créer une liste de client pour le serveur. Nous créons une API Web REST :

- Requête GET/player/<id> : Pas de données en entrée
  - Si <id> joueur existe : Statut 200 OK
    Données de sortie :

```
type : "object",
properties : {
    "name" : {type : string},
    "teamcolor" : {type : status::TEAM},
},
required : ["name"]
```

  - Si <id> joueur n'existe pas : Statut 404 NOT_FOUND
    Pas de données de sortie

- Requête PUT/player :
  Données en entrée :
  ```
  type : "object",
      properties : {
          "name" : {type : string},
          "teamcolor" : {type : status::TEAM},
      },
      required : ["name"]
  ```
  — S'il reste une place libre : Statut 201 CREATED
  Données de sortie :
  ```
  type : "object",
  properties : {
      "id" : {type : number, minimum : 0, maximum : 2},
  },
  required : ["id"]
  ```
  — S'il n'y a plus de place : Statut 503 OUT_OF_RESOURCES
  Pas de données de sortie

- Requête POST/player/<id> :
  Données en entrée :
  ```
  type : "object",
      properties : {
          "name" : {type : string},
          "teamcolor" : {type : status::TEAM},
      },
      required : ["name"]
  ```
  — Si <id> joueur existe : Statut 204 NO_CONTENT
  Pas de données de sortie
  — Si <id> joueur n'existe pas : Statut 404 NOT_FOUND
  Pas de données de sortie

- Requête DELETE/player/<id> : Pas de données en entrée
  — Si <id> joueur existe : Statut 204 NO_CONTENT
  Pas de données de sortie
  — Si <id> joueur n'existe pas : Statut 404 NOT_FOUND
  Pas de données de sortie

6.1.3 Répartition sur différentes machines : échange des commandes

Pour gérer les commandes, tous les clients envoient leurs commandes moteur au serveur. Ils doivent demander de manières régulière au serveur s'il y a de nouvelles commandes pour le prochain tour. S'il y en a, les clients exécutent ces commandes, puis attendent les prochaines. Pour cela nous créons un service web CR sur la donnée "commandes par tour" : on peut ajouter/créer un lot de commandes par tour :

- Requête GET/commands/<tour> : Pas de données en entrée
  — Si <tour> existe : Statut 200 OK
  Données de sortie : liste des commandes sérialisées en JSON
  — Si <tour> n'existe pas : Statut 404 NOT_FOUND
  Pas de données de sortie

- Requête PUT/commands :
  Données en entrée :

```
type : "object",
    properties : {
        "command" : commande sérialisée
    },
    required : ["command"]
```

  Tous les cas : Statut 201 CREATED
  Données de sortie :

```
type : "object",
    properties : {
        "tour" : {type : number},
    },
    required : ["tour"]
```

6.1.4 Répartition sur différentes machines : statut de la partie

Un des services web nous permet d'obtenir le statut de la partie : CREATING(en cours de création) ou RUNNING (partie en cours). Cela permet au client de savoir si une partie est en cours ou si le serveur est toujours actif.

6.2 Conception logiciel

Le diagramme de la classe Serveur est présenté en Figure 6.1.
Game Contient la liste des joueurs connectés ou non.
PlayerService Répond aux requêtes envoyées par les clients connectés : ajout, modification, consultation et suppression de profil.
VersionService Renvoie la version actuelle de l'API.
GameService Permet de consulter l'état du jeu.
CommandsService Fournit les services CR pour la ressource "commandes par tour". Permet d'ajouter et consulter ces lots de commandes.

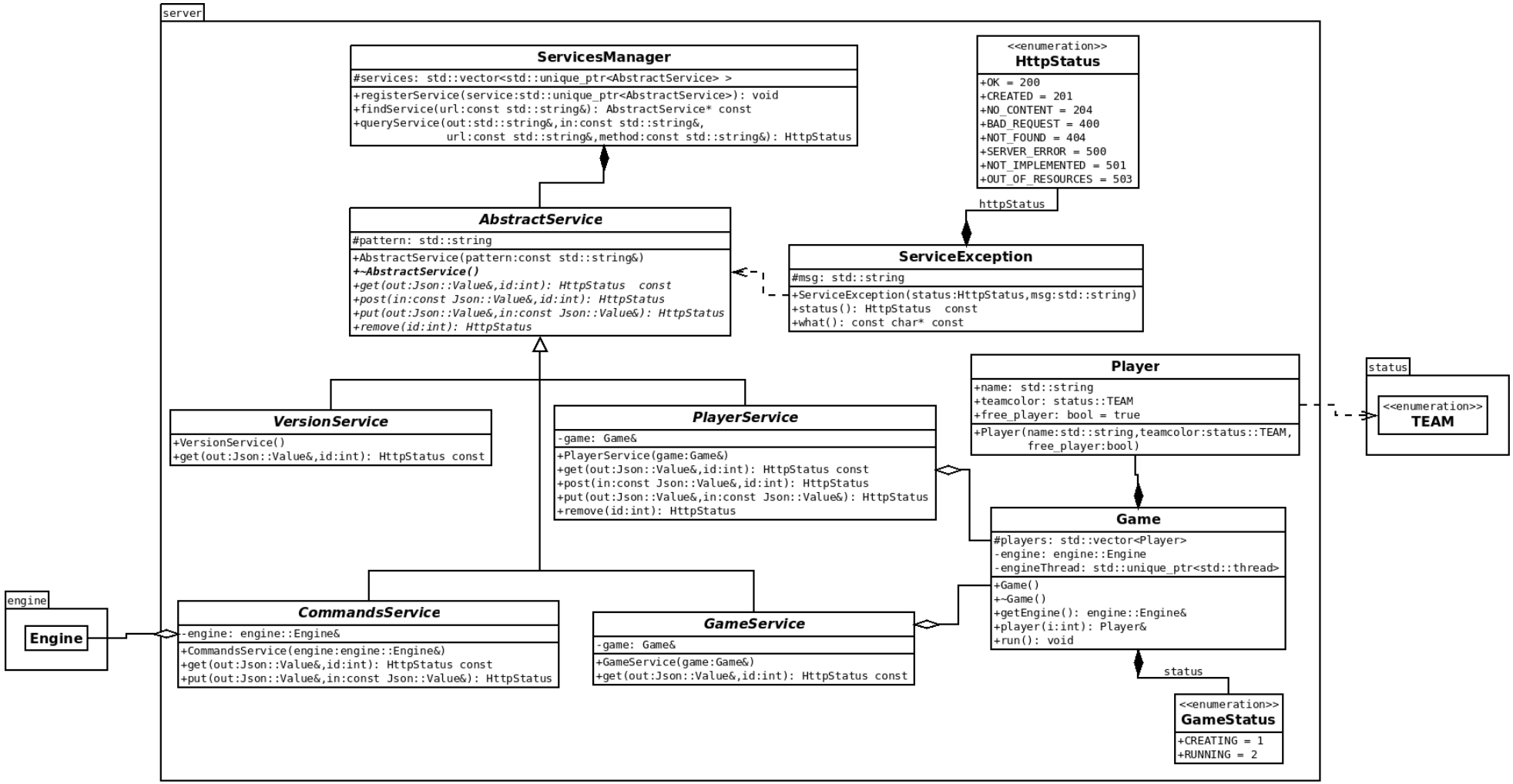

FIGURE 6.1 – Diagramme du serveur