Projet Logiciel Transversal

Mohamed KHLIFI – Ihsen OUERGHI

Table des matières

1 Objectif

1.1 Présentation générale

Le jeu que nous souhaitons réaliser consiste en une opposition entre 2 joueurs dans la conquête d'une map. Notre jeu s'appuie ainsi sur les archétypes des jeux Total War et Civilization.

Au départ les deux joueurs se voient confier un nombre égal de ressources et une ville. Dès lors, les deux joueurs vont devoir explorer la map, par l'intermédiaire d'un type de joueur (explorateur ou armée), se développer et conquérir de nouveaux territoires. Des combats peuvent survenir pour la conquête d'une ville, ceux-ci seront simulés et donneront lieu à un vainqueur qui remportera la ville. Le jeu se finit lorsque l'un des joueurs conquière toute la map.

Notre jeu mêlera donc gestion/expansion de territoire, gestion de matières premières et gestion de ressources humaines.

Le jeu comporte une map principale découpée en plusieurs villes. Au départ celles-ci sont inhabitées et chaque joueur peut les coloniser. Concernant les personnages, nous commencerons avec deux types différents : les explorateurs et les soldats. Les explorateurs servent à coloniser des villes pour pouvoir les contrôler. Les soldats servent aux combats et déterminent en bonne partie l'issue de ceux-ci.

1.2 Règles du jeu

Une ville possède deux états : inhabitée ou colonisée.

Une ville peut être colonisée que par des explorateurs ou une armée.

Chaque ville possède deux notes : défense et fertilité. La note de défense est conditionnée par le nombre de soldats et par les fortifications construites.

Chaque joueur possède deux ressources : la nourriture et l'or.

Si une ville est déjà colonisée par l'adversaire, il faut l'attaquer pour la conquérir.

Les combats sont simulés et prennent en compte l'armée qui attaque d'un côté et la note de la ville et l'armée qui défend de l'autre.

Les bâtiments disponibles sont :
- Fermes et mines pour les ressources
- Caserne pour produire les soldats

1.3 Conception Logicielle

Pour les ressources nous avons trouvé pour démarrer un tileset minimaliste mais largement suffisant pour réaliser ce que nous avons en tête simplement. Il a été ajouté au dossier res du projet.

Figure 1 : Tileset pour map

Nous avons aussi besoin de sprites représentant les armées et les exportateurs, pour cela nous avons trouvé en ligne un éditeur de personnages que nous avons adapté pour l'IA et le joueur.

Figure 2 : Armée du joueur 1

Figure 3 : Explorateurs du joueur 1

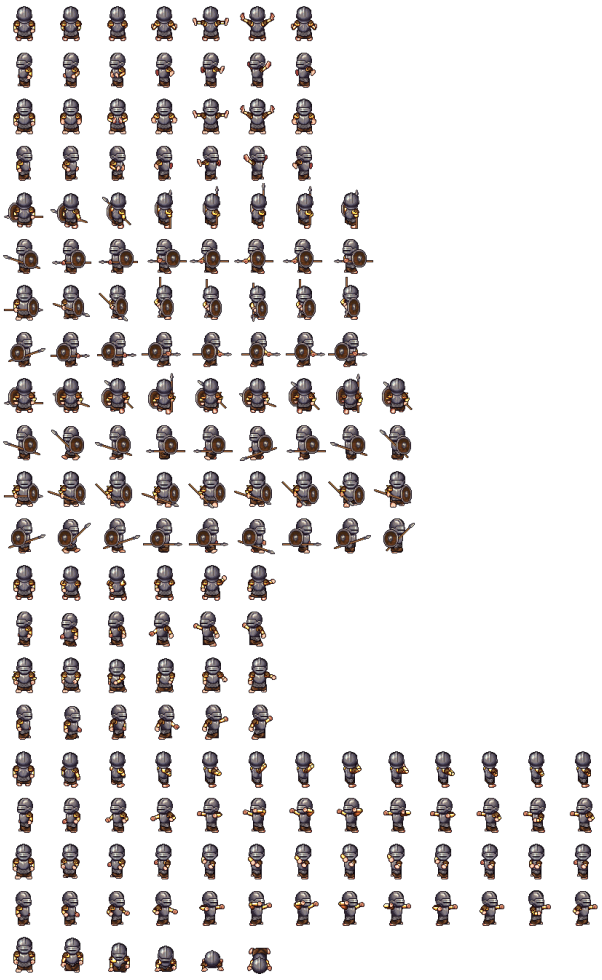


Figure 4 : Armée du joueur 2

Figure 5 : Explorateurs du joueur 2

Bien sur certaines animations du spritesheet ne seront pas utilisées.

2 Description et conception des états

2.1 Description des états

Notre jeu sera constitué d'une map principale sur laquelle se déroule tout le jeu. Au départ, nous prévoyons que le joueur ne connaisse qu'une partie de celle-ci, l'idée étant de s'aventurer sur la map pour pouvoir la découvrir. Ceci implique ainsi de ne connaître ni les positions des villes, ni la position initiale du joueur adversaire au départ.

Il y aura des éléments fixes et mobiles, ceux ci auront une position, et appartiendront à un des joueurs.

2.1.1 États à éléments fixes

Cases "City" : chaque ville possède un nombre de cases déterminé sur lesquelles elle va se construire. Nous avons aussi une classe Construction (pour bâtiment). Une City possède un tableau de constructions. Trois classes héritent de Construction : Mine (pour l'extraction de l'or), Farm (pour la production de nourriture) et Barrack (pour la création de soldats). Cette City peut être en état de construction ou non, elle peut être libre ou non.

Cases "Landscape" : cases en dehors des villes dans lesquelles peuvent se déplacer les explorateurs et les soldats. Dans la classe correspondante, chaque Landscape peut être accessible ou non pour les éléments mobiles.

2.1.2 États à éléments mobiles

Élément "Army" : Unité de combat pour la défense d'une ville et pour la conquête armée de territoire. Elle peut attaquer une ville déjà conquise ou les éléments adversaires hors des villes :
- si elle rencontre un groupe de soldat : simulation du combat
- si elle rencontre des explorateurs, elle les tue.

Elles peuvent être en garnison dans une ville ou bien en déplacement sur la carte.

Élément "Settlers" : Unité d'exploration et d'expansion. Permet de découvrir la map, et des colonies dans les villes inhabitées. Cette unité précieuse dans l'expansion est complètement impuissante contre les soldats et meure si elle se fait attaquer.

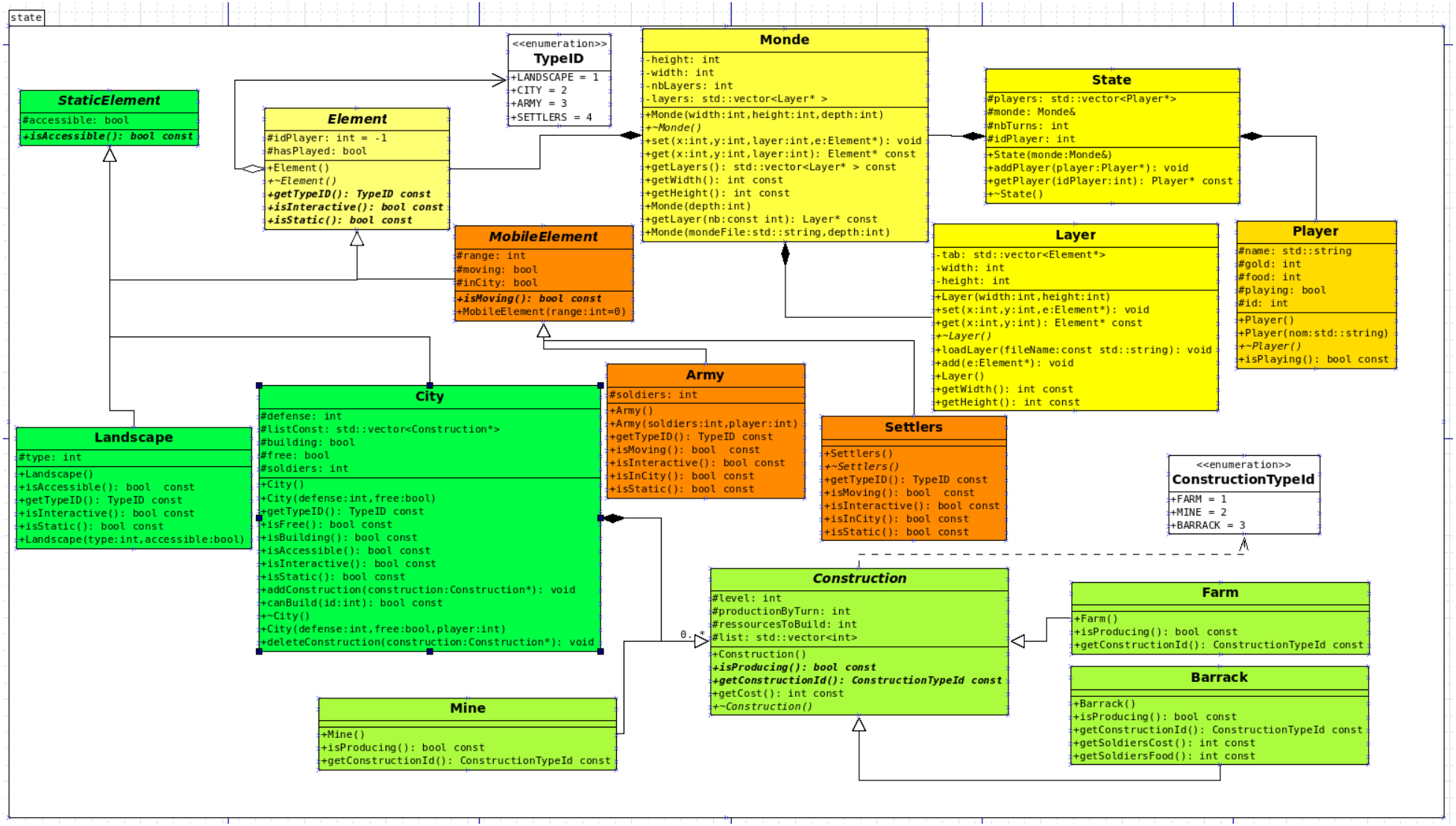
2.1.3 États généraux

L'état général est défini par les époques, ainsi que le joueur qui est en train de jouer.

2.2 Conception logicielle

Le diagrame UML est donné ds la prochaine figure, les principales classes sont Element et ses heritières.
Ensuite la classe Player, et enfin la classe monde, contenant une grille d'éléments. On a aussi rajouté une classe state décrivant l ‘état du jeu.

Illustration 1: Diagramme des classes d'état



3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Pour réaliser notre rendu nous sommes partis d'un tileset que nous avons édité grâce au logiciel Tiled. Celui-ci permet de dessiner des maps à partir d'un tileset importé. A partir de celui disponible en figure 1, nous l'avons découpé en pixels de 16x16 grâce à Tiled et avons dessiné plusieurs maps pour la suite du projet.

Dans un premier nous sommes partis sur trois maps :

- Une map "Obstacles"pour les test que nous réaliserons plus tard, assez basique pour l'instant
- Une map "Montagne" qui s'appuie principalement sur les éléments de roche du tileset
- Une map "Mer" qui s'appuie principalement sur les éléments marins du tileset.

Pour réaliser toutes ces maps, nous choisissons de découper chaque map en plusieurs couches ou "layers" qui regroupent des informations ou éléments différents. Ainsi, notre premier layer correspond à l'arrière plan et se compose exclusivement d'herbe verte, et est donc commun à toutes les maps. Le deuxième layer nous sert à définir la map : suivant la map à éditer on aura tel ou tel paysage, ce layer contient aussi les villes, la superposition des deux premiers layers donnera donc "naissance" à une map vierge. Le troisième layer nous servira pour la gestion des éléments mobiles, à savoir les explorateurs et les armées. Enfin nous prévoyons un quatrième layer pour l'interface utilisateur qui regroupera les informations du jeu, les interactions avec les objets du jeu, ... Ce dernier layer sera précisé et conçu par la suite.

Une fois ces cartes dessinés on exporte au format CSV chaque layer de la carte. Ainsi à chaque démarrage du jeu, on aura la possibilité d'initialiser les états pour les éléments fixes à partir des fichier CSV correspondant aux layer, et d'un autre fichier CSV contenant une matrice permettant de transformer directement un tile en état.

Ensuite durant le jeu, nous réaliserons le rendu directement à partir des états.

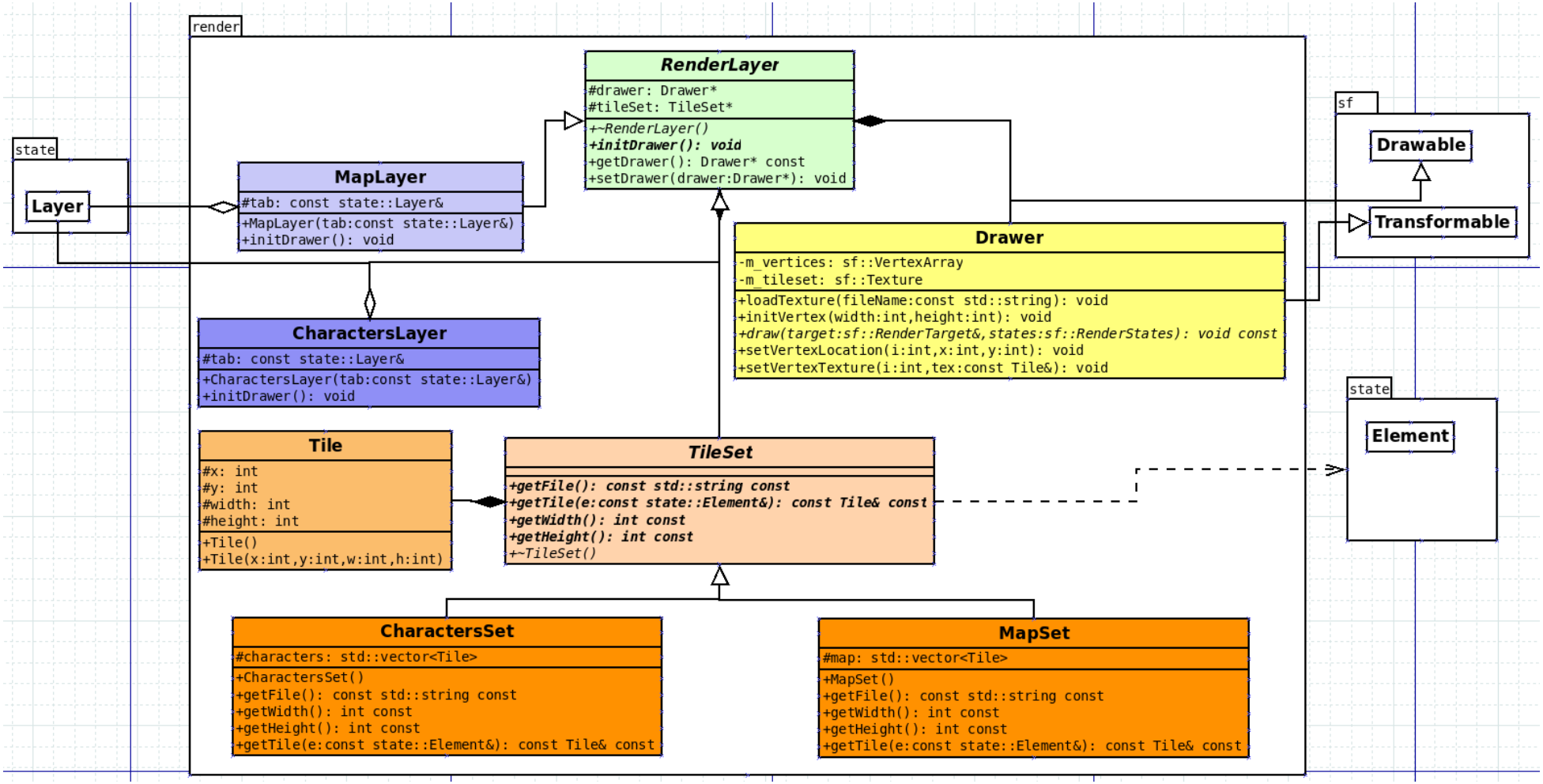
3.2 Conception logicielle

RenderLayer. Cette classe charnière va nous permettre de concevoir le rendu visuel de notre jeu et s'appuie notamment sur les classes MapLayer et CharacterLayer qui héritent de celle-ci MapLayer contient les deux premières couches d'éléments fixes composant le niveau, et CharactersLayers contient les éléments mobiles de la troisieme couche. Ces deux dernières classes initialisent les vecteurs 2D qui préparent le rendu à afficher. En effet ce sont ces classes qui sont reliés aux états, on peut ainsi créer le rendu à partir des états directement.

Drawer. Cette classe va charger les textures, elle va aussi initialiser et configurer les vecteurs par l'intermédiaire des méthodes initVertex, setVertexLocation et setVertexTexture. Cette classe fait aussi le lien avec SFML pour afficher les textures à l'écran. On affiche le rendu grâce aux tableaux de vertex, ainsi cette classe comprend le tableau de vertex qui stocke les quads, ainsi que la texture.

Tileset. Cette classe et ses filles se chargent des tiles, on a créé une classes génériques appelé Tile, permettant de créer une représentation des éléments du tileset, on stocke notamment la taille des éléments du tileset la position dans le tileset. MapSet permet de stocker un vecteur de tile, mais aussi la taille du tileset et le nom du fichier, et CharactersSet fait de même pour les personnages. Mais surtout c'est dans ces classes que se trouve la fonction getTile, qui permet de transformer un Élément en entité graphique que l'on peut afficher.

Illustration 2: Diagramme de classes pour le rendu



4 Règles de changement d'états et moteur de jeu

4.1 Horloge globale

Notre jeu s'effectuant en tour par tour, il possède une chronologie conditionnée par le moment où le joueur réalise une commande. En effet, le joueur ne possède pas de limite de temps pour effectuer ses commandes, et les changements d'états ne peuvent survenir qu'après une action (ou commande) du joueur.

4.2 Changements extérieurs

Pour l'instant, nous prévoyons seulement des commandes avec la souris mais d'autres commandes (par exemple du clavier) pourront survenir plus tard.

Nos commandes peuvent se répartir suivant 3 catégories :

Avant quasiment toutes les commandes, on vérifie que l'on utilise un élément qui appartient au joueur qui est en train de jouer.

- Commandes de personnages (armée et explorateurs) :
  - Déplacement : on indique une case à laquelle ils doivent se rendre, on vérifie que celle ci est accessible et dans la portée de déplacement de l'unité
  - Attaquer armée(pour les armées) : rentrer en combat avec une autre armée ou une ville
  - Attaquer ville (pour les armées) : rentrer en combat avec une autre armée ou une ville
  - Coloniser (pour les explorateurs et armées) : prendre le contrôle reviens à attaquer
  - Entrer dans une ville (on augmente ainsi le nombre de soldats de la ville et donc sa capacité à se défendre).
  - Fusionner deux armées
  - Diviser une armée

- Commandes de gestion (villes)
  - Construire un bâtiment : ajoute un bâtiment dans la ville, ces bâtiments produisent des ressources, et on peut en avoir uniquement un seul de chaque type (coûte des ressources)
  - Augmenter le niveau : permet à une ville de passer au niveau supérieure, amélioration de la note de défense de la ville (coûte des ressources)
  - Créer des soldats si la ville possède une caserne (coûte des ressources)
  - Faire sortir des soldats de la ville, cela à pour effet de diminuer le nombre de soldats de la ville et de créer une nouvelle armée qui pourra se déplacer sur la carte

Enfin il faut incrémenter les ressources à la fin de chaque tour.

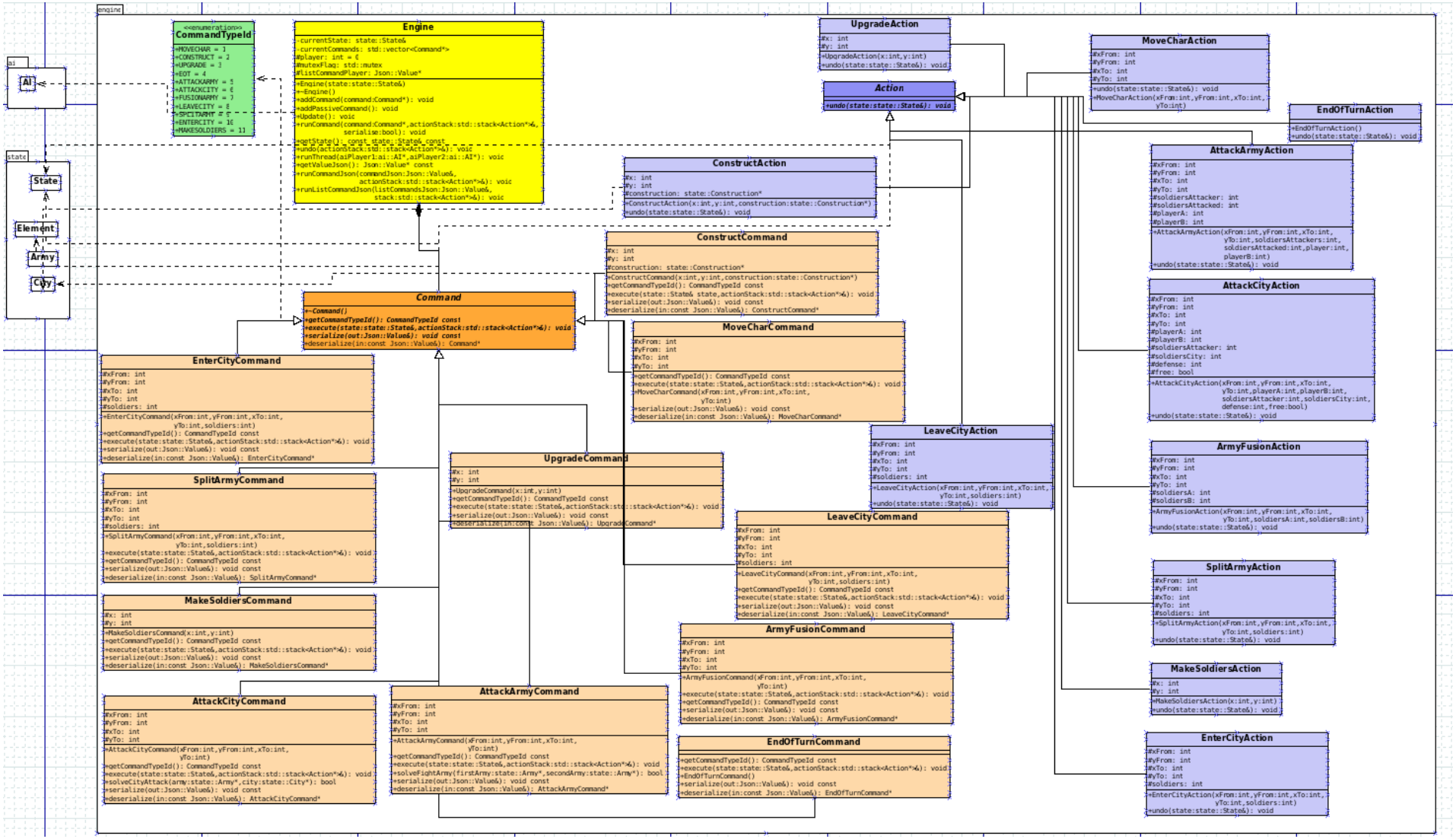
4.3 Conception logicielle

Le diagramme des des classes pour le moteur du jeu est présenté en figure 3. L'ensemble du moteur du jeu s'inspire du patron de conception de type Command. Il a pour but principal la mise en œuvre des commandes extérieures évoquées précédemment.

Class Command. Classe parente des différentes classes filles, qui possède un TypeId qui détermine le type de la commande utilisée parmi l'énumération "CommandTypeId".
Class Engine. Classe qui représente le moteur du jeu, génère notamment des commandes, génère des commandes (méthode runCommand) étant donné que nous faisons un jeu au tour par tour, nous lanceront les commandes avec runCommand a chaque fois qu'une commande sera lancé. Nous avons prévu un vector pour stocker des commandes et toutes les lancer avec update, mais il n'est pour l'instant pas nécessaire donc inutilisé.
Classe UpgradeCommand. Classe qui permet à une ville d'accéder au niveau supérieur.
Class MoveCharCommand. Classe qui permet de déplacer les personnages d'une position à une autre. En vérifiant que cela est possible.
Class ConstructCommand. Classe qui permet la construction d'un bâtiment à partir d'une ville.
Class EndOfTurnCommand. Classe qui à la fin du tour met à jour les ressources selon les villes que possède le joueur et les bâtiments de ces villes.
Class AttackArmyCommand. Classe qui gère les combats, l'armée gagnante est sélectionné en fonction de son nombre de soldats, plus l'écart entre les soldats est grand, plus la probabilité de gagner est élevée. Pour que cela soit réaliste, nous faisons passer la différence de soldats entre les armées (qu'on divise par le nombre total de combattant) par une fonction sigmoïde de paramètre 4. Ainsi il est peu probable qu'une armée ayant deux fois moins de soldats l'emporte.
Class AttackCityCommand. Classe similaire à la précédente, la force de l'armée est son nombre de soldats, celle d'une ville est sa note de défense et ses soldats (nbSoldats + defense*100), on procède ensuite comme pour les combats entre armées.
Class ArmyFusionCommand. Classe qui rassemble 2 armées.
Class SplitArmyCommand. Classe qui divise en deux une armée.
Class LeaveCityCommand. Classe qui crée une armée à partir des soldats que possède une ville, On crée à l'issue de cette commande une nouvelle armée sur la carte.
Class EnterCityCommand. C'est l'inverse de la précédente.
Class MakeSoldiersCommand. Permet de créer des soldats en échange de ressources, cette commande va directement incrémenter l'attribut soldat de la classe City. Ces soldats pourront ensuite former une armée.

Classes Action. Permettent de pouvoir faire le rollback, à chaque fois qu'une action est exécuté, on ajoute l'action correspondante dans une pile.

Illustration 3: Diagrammes des classes pour le moteur de jeu



5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence artificielle aléatoire

Dans un premier temps, nous allons générer des commandes de manière aléatoire. Pour cela, à chaque fois qu'un joueur joue, nous parcourons tous les éléments du layer 1 à la recherche des villes que possède le joueur qui est en train de jouer. Pour chacune des villes du joueur, on liste des commandes et on en choisi aléatoirement une pour l'exécuter. On fait la même chose en parcourant le layer 2 à la recherche des personnages, et on liste des commandes pour en exécuter une.

5.1.2 Intelligence artificielle basée sur des heuristiques

Pour cette intelligence artificielle, nous choisissons de faire un compromis entre la distance et le poids de chaque commande à effectuer. Nous calculons à chaque tour le poids de chaque case adjacente, avant de prendre une décision et se déplacer vers l'une des cases disponibles.

Le poids de chaque case sera un compromis entre la distance d'une action et son poids. Pour mieux comprendre le compromis évoqué, prenons un exemple : si une armée décide de se déplacer vers une ville et rencontre sur son chemin une armée adverse de niveau faible, il y a une décision à prendre : continuer son chemin vers la ville à conquérir ou attaquer l'armée adverse. Dans notre jeu, le poids de la commande attaquer une ville est plus important car il représente le cœur de la stratégie du jeu alors que la commande attaquer une armée possède un poids plus faible.

5.1.3 Intelligence avancée

Pour réaliser cette intelligence artificielle, nous raisonnons avec des arbres de recherche. Le facteur de branchement étant énorme dans notre jeu, nous choisissons de ne pas ajouter d'armées dans le jeu, on commence ainsi avec deux armées pour chaque camps. Le facteur de branchement est ainsi réduit à 20 environ. La profondeur sera variable, l'algorithme tourne en un temps raisonnable pour une profondeur de 4 et de façon fluide avec pour profondeur 3. On va donc créer un arbre à partir d'un état et lui appliquer l'algorithme MinMax. L'heuristique pour évaluer un état est le nombre de villes que l'on possède et la différence de soldats avec l'ennemi, ces deux éléments sont pondéré différemment.

5.2 Conception logicielle

Classe AI. Les différentes classes qui héritent de cette classe mère visent à générer des commandes pour l'IA, elle possède des références vers les classes State et Engine.
Class RandomAI. Cette classe permet d'avoir des commandes qui sont sélectionnées aléatoirement, et créer ainsi un scénario aléatoire. Chaque fois qu'un joueur joue on passe par la fonction run.
Classe HeuristicAI. Cette classe intervient pour avoir un comportement de l'IA basé sur des heuristiques.
Classe DistanceMap. Cette classe hérite de la classe HeuristicAI et va permettre de générer une carte avec les poids sur chaque case, calculé à chaque tour.
Classe PointCompareWeight. Cette classe va comparer les poids de chaque point pour choisir le poids le plus faible ou le plus important suivant la stratégie que nous voulons adopter.
Classe CommandsWeight. Cette classe va nous permettre de calculer les poids de nos commandes afin de déterminer la meilleure stratégie.
Classe Point. Cette classe va attribuer un poids à chaque point de la map.
Classe DeepAI. Cette classe va nous permettre de simuler notre IA avancée, par l'intermédiaire des méthodes "createTree" (pour créer l'arbre) "findCommands" (pour trouver les commandes possible étant donné un état) deux autres méthodes de recherche MinMax récursives.
Class Node. Cette classe va permettre de paramétrer les nœuds de l'arbre avec des méthodes pour créer/récupérer un fils, la taille et les commandes.

(L'intelligence Avancé ne marche pas, et nous n'avons pas eu le temps de corriger l'erreur. Il aurai surement été plus judicieux de ne pas créer l'arbre avant d'appliquer le MinMax, car cela nous a pris beaucoup de temps alors que créer les états pendant l'algorithme aurait été plus pratique. Aussi l'heuristique choisie n'est pas assez précise).

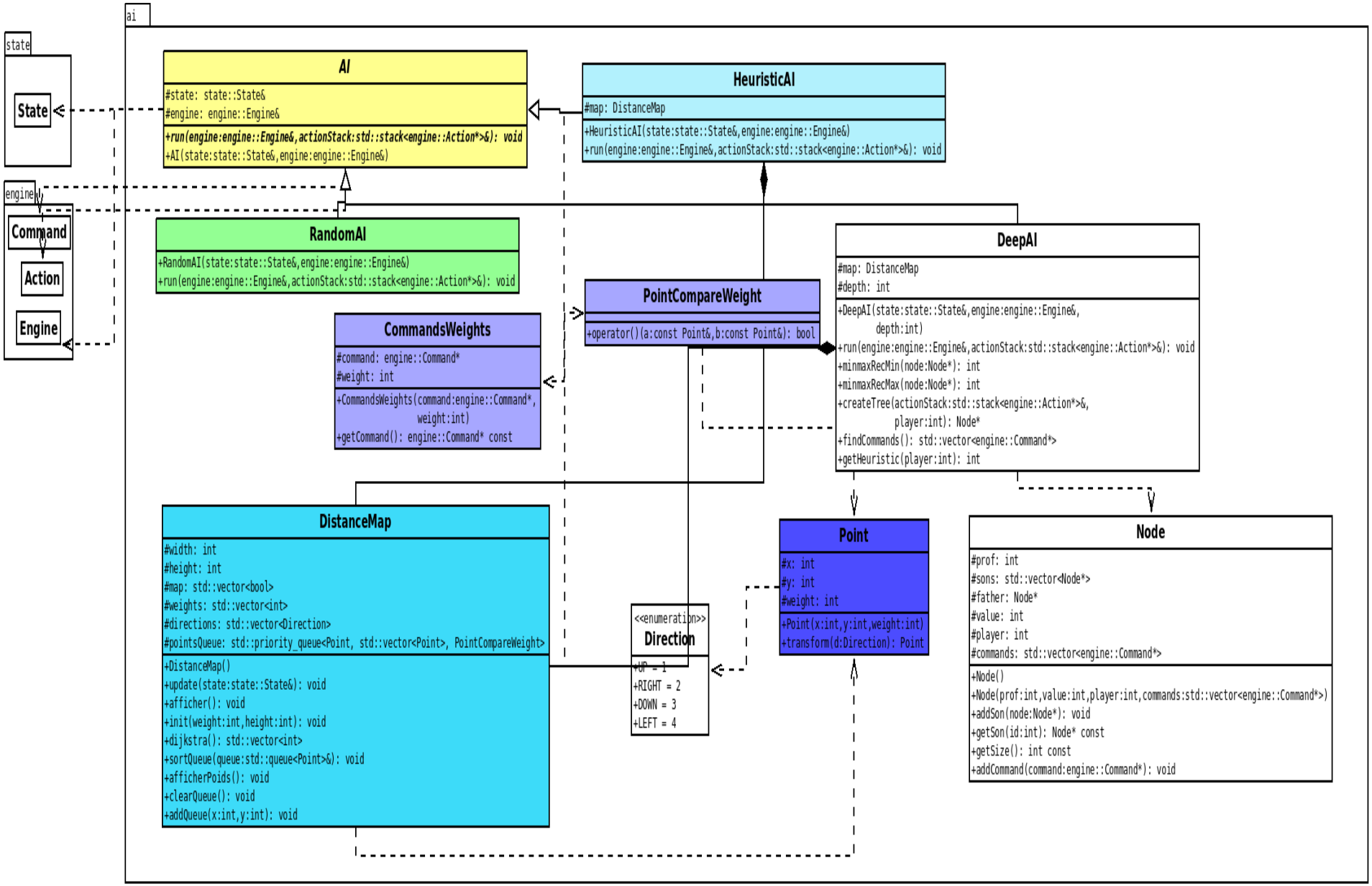


Illustration 4: Diagramme de classes pour l'intelligence artificielle

6 Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

Nous voulons faire fonctionner notre jeu avec différents threads, ainsi il sera possible de paralléliser les opérations et être plus performant. Nous aurons deux threads, un thread principal, et un tread secondaire, le thread principal nous permet de préparer et afficher le rendu graphique, le secondaire se focalise sur le moteur de jeu. Les commandes étant executés directement par l'IA, il est nécessaire de lancer l'IA dans le thread directement. Nous allons donc faire un thread qui va boucler, et une variable indiquera ce que doit faire ce thread.

6.1.2 Répartition sur différentes machines : rassemblement des joueurs

Nous allons tout d'abord permettre aux joueurs, les clients, de se réunir dans une salle d'attente avant de démarrer une partie. Nous avons un nombre maximum de joueurs pouvant être dans la salle, logiquement pour notre jeu, ce nombre maximal est 2.

Différentes requêtes peuvent être faites par les clients au serveur, voici leur liste, contenu et réponses :

→ Requête GET/player/<id>:

Pas de donnée en entrée

<id> existe	Statut **OK** *En sortie :* type: "object", properties: { "name": {type: string}, }, required : ["name' "]

<id> négatif	Statut **OK**
	En sortie : type: "array",
	items: {
	type: "object",
	properties: {
	"name": {type: string},
	},
	},
	required : ["name' "]

<id> n'existe pas	Statut **NOT_FOUND** *Pas de données en sortie*

→ Requête PUT/player:

En entrée : type: “object”, properties: { “name”: {type: string}, }, required : [“name’ ’]

Il reste une place libre	Statut **CREATED** *En sortie :* type: “object”, properties: { “id”: {type: number,minimum: (),maximum : 2}, }, required : [“id’ ’]
Il n’y a pas plus de place libre	Statut **OUT_OF_RESSOURCES** *Pas de données en sortie*

→ Requête POST/player/<id>:

En entrée : type: “object”, properties: { “name”: {type: string}, }, required : [“name”]

<id> existe	Statut **NO_CONTENT** *Pas de données en sortie*
<id> n’existe pas	Statut **NOT_FOUND** *Pas de données en sortie*

→ Requête DELETE/player/<id>:

Pas de données en entrée

<id> existe	Statut **NO_CONTENT** *Pas de données en sortie*
<id> n’existe pas	Statut **NOT_FOUND** *Pas de données en sortie*

6.1.3 Répartition sur différentes machines : échange de commandes

Pour réaliser cette répartition, nous rajoutons deux nouvelles classes qui vont permettre d'avoir des listes de commandes pour chaque joueur, qui seront envoyées au serveur. Nous nous appuyons sur un service web CommandsService pour récupérer les listes de commandes et en ajouter des nouvelles. Le comportement des différentes requêtes est résumé dans les tableaux ci-dessous. De plus, nous avons une variable "player" dans la classe Game qui nous permet de savoir le joueur qui doit jouer. Cette information sera récupérée via une requête Get du nouveau service GameService.

→ Requête GET/commands/<id>:

int id (l'identifiant du client)

Vecteur commands vide et IdPlayer!= id (L'adversaire à joué)	Statut **OK**
	En sortie :
	type: "array",
	items: {
	type: "object",
	properties: {
	"CommandID": {type: string},
	},
	},
	required : ["CommandID' ']

Tous les autres cas	Statut **NO_CONTENT** *Pas de données en sortie*

→ Requête PUT/commands:

En entrée : La liste de commande à ajouter dans la variable currentCommands de Game
type: "array",
items: {
type: "object",
properties: {
"CommandID": {type: string},
},
},
required : ["CommandID' ']

Vecteur commands vide	Statut **CREATED** *En sortie :* type: "object", properties: { "idP": {type: number,minimum: (),maximum : 2}, }, required : ["idP' ']

Tous les autres cas	Statut **OUT_OF_RESSOURCES** *Pas de données en sortie*

6.2 Conception logicielle

Nous allons créer une fonction runThread dans la classe Engine, c'est cette fonction que l'on lancera lorsque l'on créera le thread. Une fois lancé, cette fonction réalisera une boucle infinie, et la variable "player" permettra de savoir ce que nous devons faire. La valeur de player indique le joueur qui doit jouer, après avoir lancé l'IA, on met cette variable à 0, quand cette variable est à 0, on ne fait rien, et quand elle est à -1, on quitte la boucle du thread.

Le diagramme des classes pour le serveur est donné ci-dessous.

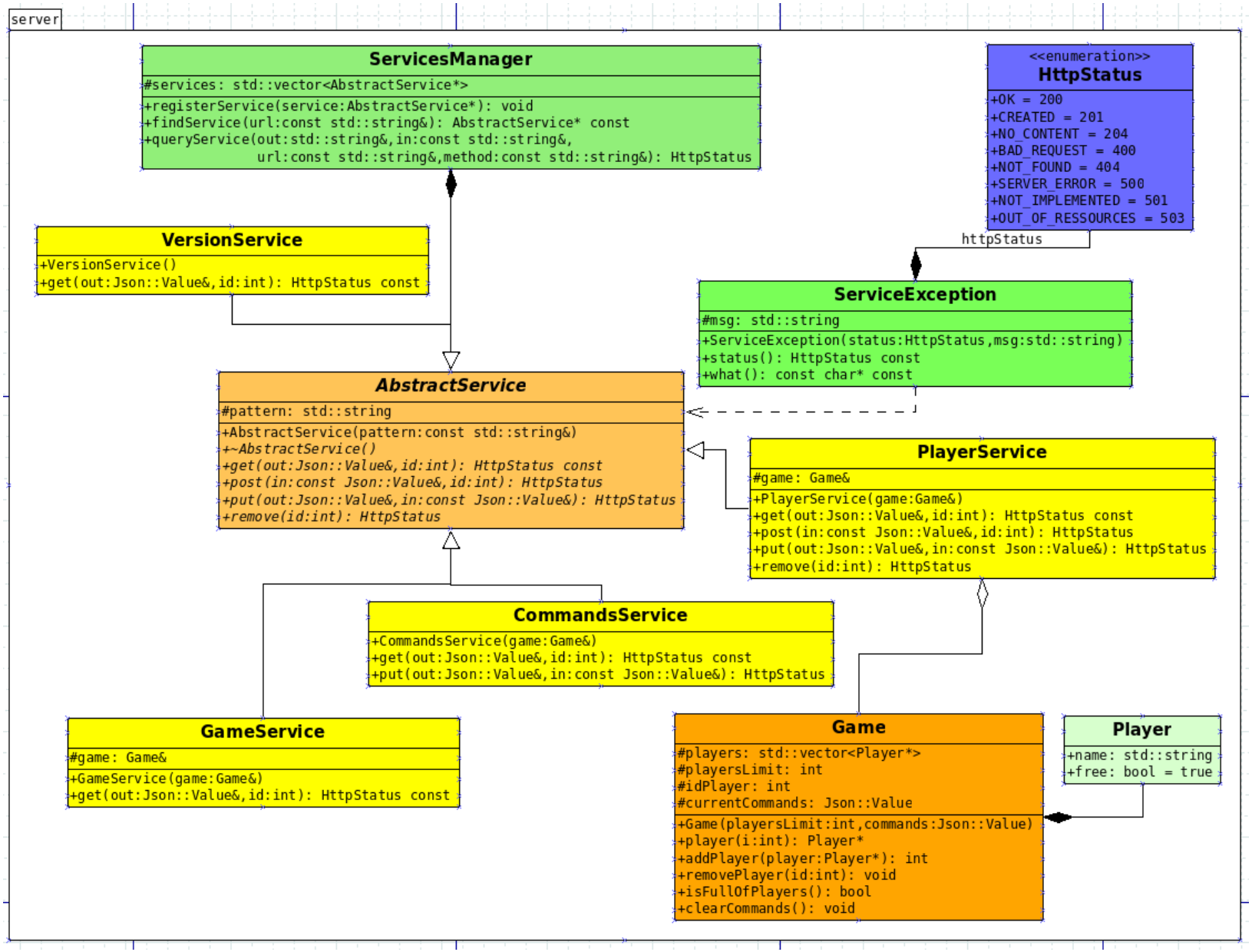


Illustration 6: Diagramme de classes pour la modularisation

Game. La classe game et les classes associées représentent les éléments d'une partie. Dans cette classe on trouve une liste de joueurs, qui va nous permettre de stocker les joueurs connectés et leurs noms via les attributs de la classe **Player.**
Cette classe possède aussi un attribut donnant le nombre limite de joueurs pouvant se connecter pour une partie.
Services. Les services sont lancé grâce à **ServicesManager** qui permet de nous rediriger vers les bons services et les bonnes requêtes après analyse de l'URI. Et il y a la classe **AbstractServices** déclarant les requêtes utilisables. De cette classes héritent quatre autres:
- 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.
- GameService : permet la consultation de l'état du jeu, ici on fait un get avec ce service pour connaître l'identifiant qui doit jouer.
- CommandsService : Permet d'ajouter et de récupérer la liste des commandes.