Projet Advance Wars

Moothery Franck
Boutaleb Yousr

Livrable 4.1

Tables des matières

1. 1 Présentation générale

1.1 Archétype

L'objectif de ce projet est de réaliser le jeu Advance wars.

1.2 Règle du jeu

Le joueur gagne la partie lorsqu'il capture la base adverse. Le jeu se déroule en tour par tour. Il gagne par tour une somme d'argent définie, qu'il doit investir pour acheter des unités de combat. La possibilité de capturer des bâtiments est possible pour améliorer son armée, ses gains ou son avantage du terrain. Quatre personnes peuvent y jouer simultanément dans une partie.

1.3 Ressources

L'affichage repose sur quatre textures

Figure 1: Texture soldat

Figure 2: Texture bâtiment

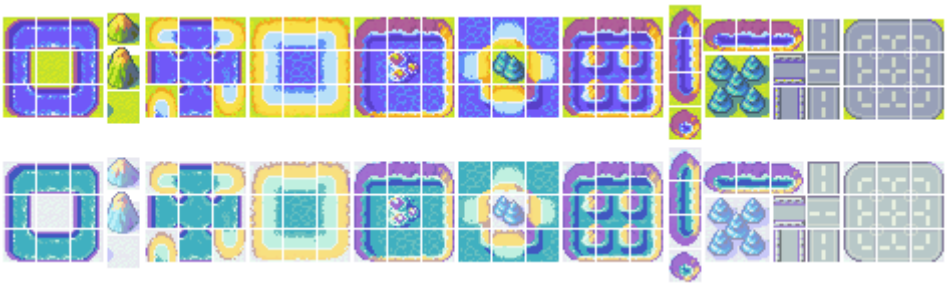

Figure 3: Texture carte

Figure 4: Texture icône

2. Description et conception des états

2.1 Description des états

Un état du jeu est formé par un ensemble d'éléments fixes (la carte) et un ensemble d'éléments mobiles (les soldats). Tous les éléments possèdent les propriétés suivantes :
— Coordonnées (x, y) dans la grille
— Identifiant de type d'élément : ce nombre indique la nature d'élément (ie classe)

2.1.1 Etat éléments fixes

La carte est formée par une grille d'éléments nommée « cases ». La taille de cette grille est fixée au démarrage de la partie. Les types de cases sont :
Cases « Eau ». Ces dernières sont des éléments infranchissables pour les éléments mobiles terrestres. Le choix de la texture est purement esthétique, et n'a pas d'influence sur l'évolution du jeu.

Cases « terrain ». Ce sont les éléments franchissables par tous les éléments mobiles. On considère les types de cases « terrain » suivants :
— Les espaces « herbe », qui contiennent de l'herbe
— Les espaces « route », qui contiennent une route
— Les espaces « sable/eau », qui contiennent du sable et de l'eau.

Cases « Rocher ». Ces dernières sont les éléments franchissables par certains éléments mobiles.
Cases « Bâtiment ». Ce sont les éléments franchissables par tous éléments mobiles. Elles permettent aussi d'utiliser certaines fonctionnalités comme celles qui servent à modifier des éléments du jeu (caserne, QG, bâtiment). On considère les types de cases « bâtiment » suivants :
— Les espaces « bâtiment »
— Les espaces « caserne », qui contiennent une caserne, qui permet de créer des unités
— Les espaces « QG », qui contiennent un QG. C'est le bâtiment qui doit être capturer pour gagner la partie.

Suivant l'élément fixe, nous avons un niveau de défense qui va s'appliquer à l'unité.

2.1.2 Etat éléments mobiles

Les éléments mobiles possèdent un mouvement, des points de vie, une défense, une puissance et une position. Un élément mobile doit être forcément sur une case. Il ne peut pas être entre deux cases. Ces éléments sont dirigés par le joueur, qui commande la propriété de direction.
Elément mobile « infanterie ». On utilise une propriété que l'on nommera « statut ». Elle peut prendre les valeurs suivantes :
— Statut « mouvement en attente » : dans le cas où, l'infanterie est en attente d'action
— Statut « mouvement fini » : dans le cas où, l'infanterie a fini de faire son mouvement
— Statut « capture » : dans cas où, l'infanterie capture un bâtiment.

Elément mobile « tank ». Il possède plus de points de vie que l'infanterie, il peut se déplacer sur un plus grand nombre de case et peut attaquer de loin, avec une portée maximale précise. La propriété « Statut » peut prendre les valeurs suivantes :
— Statut « mouvement en attente » : dans le cas où, l'infanterie est en attente d'action
— Statut « mouvement fini » : dans le cas où, l'infanterie a fini de faire son mouvement.

Elément mobile « hélicoptère ». Il est semblable au tank sauf, qu'il peut se déplacer au-dessus de tous les éléments fixes. La propriété « Statut » peut prendre les valeurs suivantes :

— Statut « mouvement en attente » : dans le cas où, l'infanterie est en attente d'action
— Statut « mouvement fini » : dans le cas où, l'infanterie a fini de faire son mouvement.

2.1.3 Etat général

Sur l'ensemble des éléments statiques et mobiles, nous rajoutons la propriété de tour pour chaque joueur. Cela va permettre de donner la main au joueur qui devrait jouer.
Le diagramme des classes pour les états est présenté en page 7, dont nous pouvons mettre en évidence les groupes de classes suivants :

Classes Élément. Toute la hiérarchie des classes filles d'élément permettent de représenter les différentes catégories et types d'éléments :

— Eléments mobiles
— Eléments fixes

Conteneurs d'élément. Les classes State (), ElementTab et ElementChar viennent ensuite. Elles permettent de contenir les éléments instanciés. ElementTab est un tableau en deux dimensions qui contiendrait les éléments statisques, et ElementChars contiendrait les éléments mobiles (ils contiennent une liste de pointeur qui va viser chaque objet instancié). Enfin, la classe State est le conteneur principal, à partir duquel on peut accéder à toutes les données de l'état (nous n'avons pas encore défini toutes ces méthodes).
Méthode du Conteneurs ElementTab :

— Deux constructeurs pour créer différentes tailles de liste
— SetElement : ajoute un élément en bout de liste
— chgList : ajoute un élément à l'endroit défini, si la case est NULL
— chgList2 : interverti deux éléments dans la liste
— createElementcsv : crée une liste d'unique ptr<Element> à partir d'un fichier csv. Le fichier csv est traduit en une liste qui, à partir de la valeur des éléments de la liste instancie un Element quelconque (CHAMPDEBATAILLE, HERBE, EAU, …)
— ElementTocarte : permet de créer une liste d'entier à partir de la liste d'unique ptr<Element>

Classes Joueur. Toute la hiérarchie des classes filles d'élément permettent de représenter les différentes catégories et types d'éléments :

— Eléments mobiles
— Eléments fixes

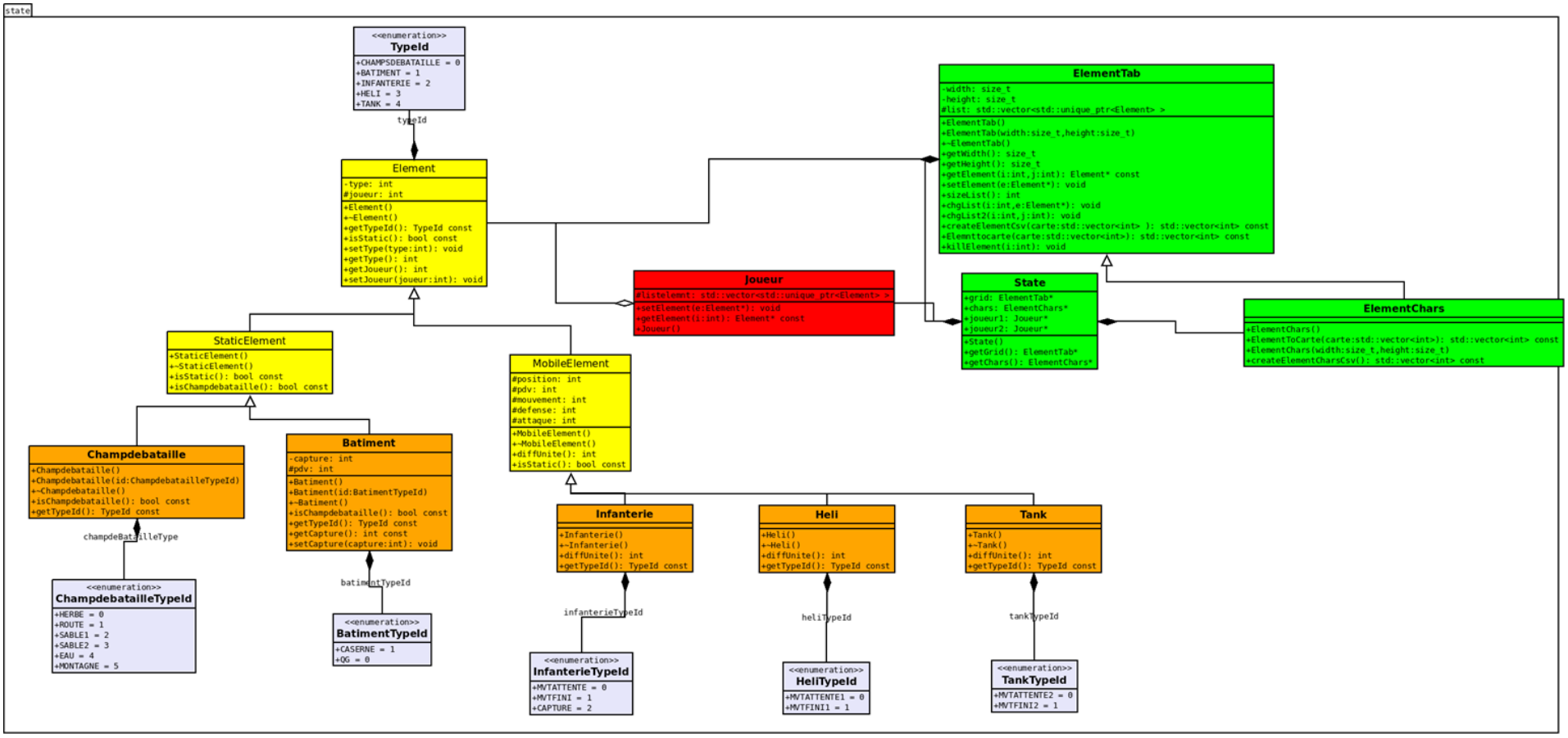

Figure 5 : Diagramme des classes d'état

3. Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, une stratégie de bas niveau a été réalisée. Le rendu d'un état est la superposition de deux plans. Un premier plan est utilisé pour afficher les éléments du décor et un deuxième plan est utilisé pour afficher les éléments mobiles. L'affichage du premier plan est réalisé par le chargement d'un fichier « .csv » qui contient des chiffres entre 0 et 7. Pour l'affichage du deuxième plan, la liste d'Element qui est remplie avec des éléments ou des éléments null est utilisée, pour créer une liste de chiffres entre 0 et 2. Les chiffres serviront à différencier tous les types d'Element mobile. De la même manière que le premier plan, le deuxième plan est géré par Surface. Une classe Tile est utilisée pour ranger toutes les informations des images.

3.2 Conception logiciel

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

Méthode de Surface :

— loadTexture : charge le fichier png dans la variable
— initQuads : initialise un tableau qui contiendra les informations à afficher
— setSpriteLocation : définie l'endroit des futures images
— setSpriteTexture : associe l'image aux places réservées par setSpriteLocation.

Méthode de GridTileSet/CharsTileSet :

— GridTileSet : remplit la liste des informations nécessaires
— getTile : permet de renvoyer la tuile correspondante à l'élément envoyé.
— getImageFile : renvoie le chemin d'accès à l'image.

Methode de Layer :

— Layer : initialise des variables pour l'utilisation de la surface
— initSurface : construit les rendus.

3.3 Description des fonctions tests

Suite à l'implémentation de chaque méthode, on utilise des fonctions tests qui vérifient le bon fonctionnement de nos méthodes, on retrouve :

Fonction touteslesfonctions :

— teststate : Elle permet de faire le test du livrable 1. Final
— testrender : Elle permet de faire le test du livrable 2.1
— testengine : Elle permet de faire le test du livrable 2.2
— testai : Elle permet de faire le test du livrable 2. Final.
— testheuristicAI : Elle permet de faire le test du livrable 31.
— testcommande : Elle permet de faire des commandes à partir de Engine

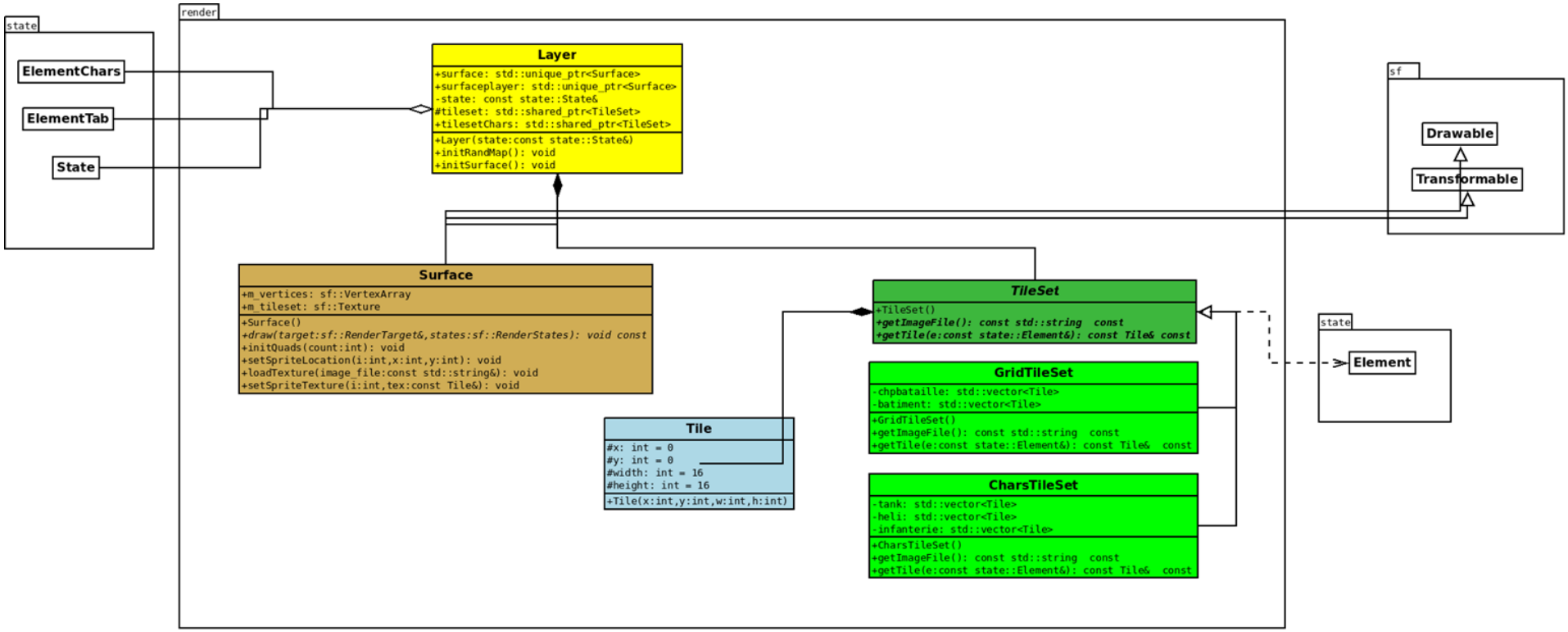

Figure 6 : Diagramme du render

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

4.1 Changements

Les changements extérieurs sont provoqués par des commandes extérieures :

— Commandes principales : « Charger un niveau » : On fabrique un état initial à partir d'un fichier
— Commandes bouger un personnage : On déplace le personnage désiré sur une case désirée s'il n'y a pas un autre personnage dessus ou, s'il peut se déplacer sur ce type de terrain
— Commandes attaquer un personnage : On attaque le personnage désiré avec un autre personnage qui appartient à un autre joueur. Si un personnage à ses points de vie réduit à 0, il est détruit
— Commandes capture un bâtiment : On capture un bâtiment avec une infanterie, qui est placée sur celui-ci
— Commandes créer un personnage : On crée un personnage.

5. Conception logiciel

On utilise le patron de conception Command pour appliquer des commandes à notre état.

Classes Command :

— MoveCharCommand : Cette fonction permet de déplacer une unité
— LoadCommand : Cette fonction permet de créer nos éléments statiques
— AttaqueCharCommand : Cette fonction permet à une d'en attaquer une autre
— createCharCommand : Cette fonction permet de créer une unité.

Classes Engine. Engine nous permet de stocker les commandes à exécuter, puis à les utiliser avec la méthode update.
On ajoute une méthode updaterecord qui sérialise les commandes avant de les exécuter .
On a ajouté un attribut de type mutex qu'on utilise dans la méthode run pour permettre l accès au thread .

Classes Action : Action permet de faire les commandes et de les mémoriser puis par la suite de revenir à une version du jeu en faisant le contraire de chaque commande dans le bon ordre.

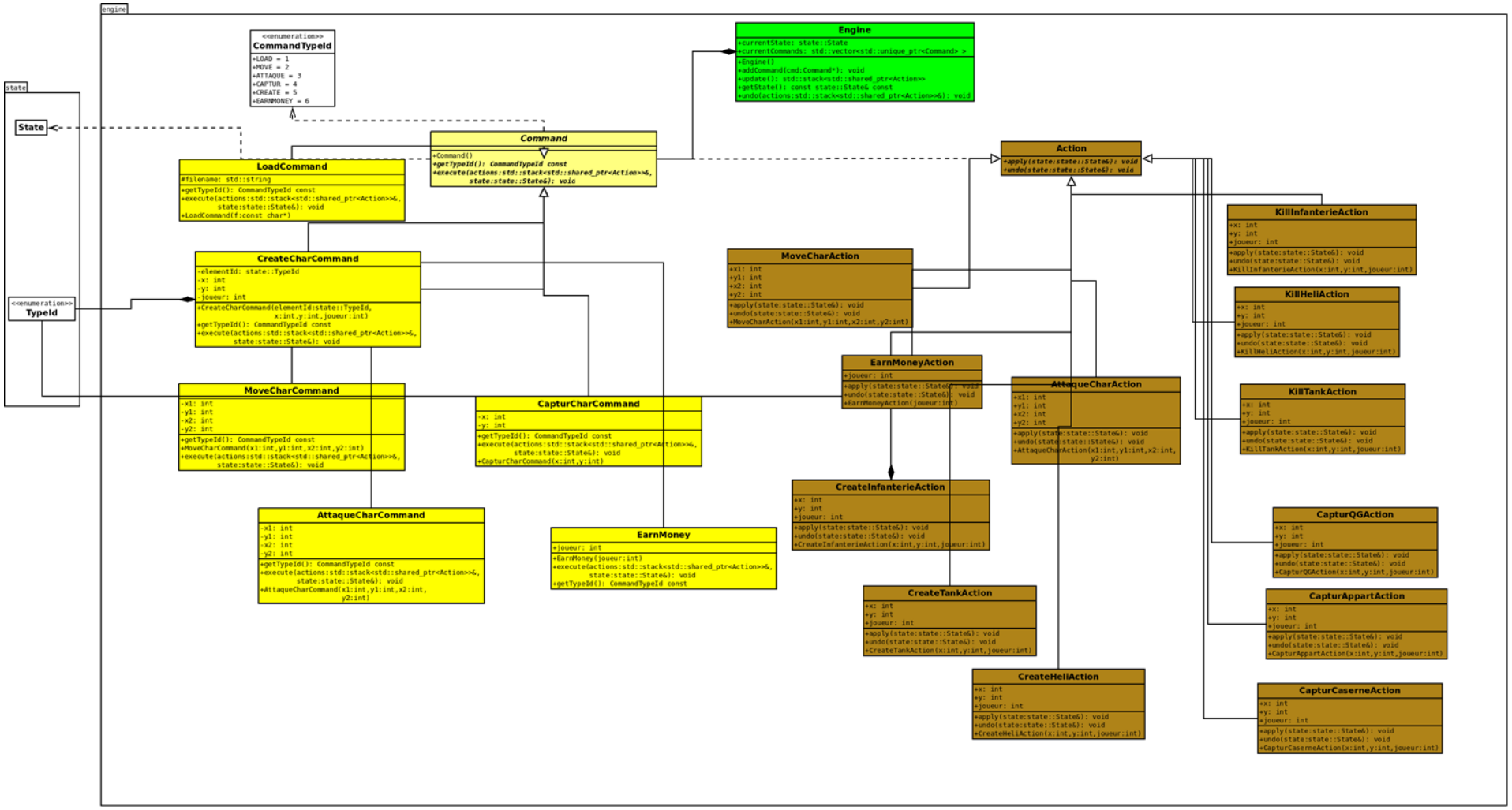

Figure 7 : Diagramme du engine

6. Intelligence Artificielle

6.1 Stratégies

6.1.1 Intelligence aléatoire

Cette stratégie est développée dans la fonction testai(). Elle permet de faire jouer deux joueurs qui exécuterons des commandes aléatoires.

6.1.2 Intelligence basée sur des heuristiques

Cette stratégie est développée dans la fonction testheuristicAI(). Elle permet de faire jouer deux joueurs qui exécuterons des commandes avec un ordre de priorité selon l'élément.
L'infanterie suit les règles suivantes :
—si une infanterie est sur un bâtiment ennemi il le capture,
—Sinon il regarde si une unité ennemie est à sa portée puis il l'attaque ou il se dirige vers un bâtiment ennemi le plus près

L'hélicoptère et le tank suivent les règles suivantes :
—Si une unité ennemie est à sa portée et il l'attaque
—Sinon il se dirige vers l'unité ennemie la plus poche

Ajout de rollback dans heuristiques : L'appuie sur une touche permet de faire un rollack. Un deuxième appuie permet de continuer le jeu avec une intelligence basée sur des heuristiques.
Ajout d'un gagnant si on capture le QG adverse.

6.1.3 Intelligence avancé

Cette stratégie est développée dans la fonction testAI. Elle permet de simuler toutes les possibilités puis cherche la possibilité qui maximise le gain.

6.2 Conception logiciel

6.2.1 Intelligence aléatoire

Classes AI :
Dans cette classe les méthodes permettent de créer une liste de pointeur sur command de toutes les commandes possibles au dépend de la position de l'élément.

Classes RandomAi :

— Run : Applique une commande aléatoire adapté pour chaque objet. Les éléments mobiles peuvent se déplacer et attaquer tandis que les éléments statiques créer des nouveaux éléments en fonction de l'argent disponible.

6.2.2 Intelligence basée sur des heuristiques

Classes HeuristicAI :

— HeuristicAI: initialise le vecteur direction
— setInfmap : calcule la carte des poids pour les infanteries d'un joueur
— setTankmap : calcule la carte des poids pour les hélicoptères d'un joueur
— setHelimap : calcule la carte des poids pour les tanks d'un joueur
— getInfmap : retourne la carte des infanteries
— getTankmap : retourne la carte des tanks
— getHelimap : retourne la carte des hélicoptères
— Run : Applique des commandes en fonction d'une priorité logique

Classes Pathmap:

— addsink : ajoute un objectif sur la carte
— isWall: détermine si le point donné est un mur ou une sortie de carte
— chgWeights : change le poids de la liste par le poids du point
— getWeights: retourne le poids de la liste par rapport à un point
— getPoidlist: retourne le poids de la liste par rapport à un entier
— update : Applique des commandes en fonction d'une priorité logique

Classes Pathinf:
— init : initialise la carte en fonction des infanteries

Classes Pathtank:
— init : initialise la carte en fonction des tanks

Classes Pathheli:
— init : initialise la carte en fonction des hélicoptères

Classes PointCompareWeight:
— operator : permet de ranger la liste priority_queue

Classes Point :
— Point : initialise un point
— transform : renvoi un point en fonction de la direction

6.2.3 Intelligence avancée

Classes Deep :

— run : cette fonction parcourt la liste des éléments mobiles et applique la fonction min sur chaque élément.

— min : une fonction récursive qui simule le jeu et puis applique les meilleures commandes pour chaque élément mobile.

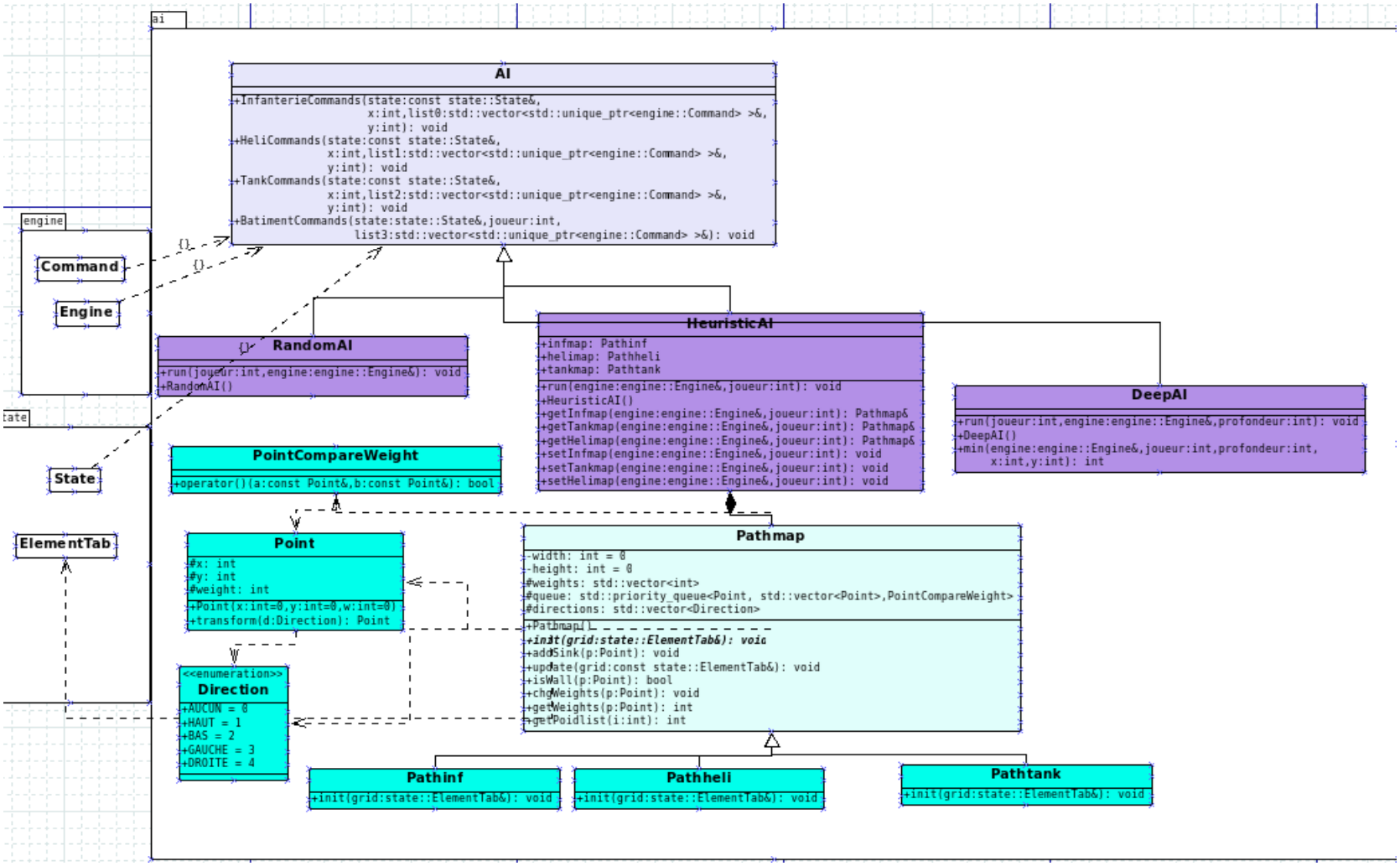

Figure 8 : diagramme ai

7. 7 Modularisation

7.1 Organisation des modules

7.1.1 Répartition sur plusieurs threads

Dans cette partie nous exécutons les commandes par un thread qui va boucler dans une méthode de la classe Engine.

7.1.2 Sauvegarde d'une partie

Dans cette partie nous faisons jouer une intelligence heurasticAI et nous sauvegardons les commandes grâce à sérialiser.

7.1.3 Chargeement d'une partie

Dans cette partie nous lisons un fichier qui a été créé précédemment par la commande « record » puis nous jouons la même partie.

7.1.4 Répartition sur différentes machines : rassemblement des joueurs

Nous formons des services CRUD sur la donnée « joueur » via une API Web REST pour ainsi créer une liste de client pour le serveur

1) Requete GET /player/<id>

Pas de données en entrée.

Si <id> existe Statut OK

Données sortie :

type: "object",

properties: {

"name": { type:string },

},

required: ["name"]

Si <id> est négatif Statut OK

Données sortie :

type: "array",

items: {

type: "object",

properties: {

"name": { type:string },

},

},

required: ["name"]

si <id> n'existe pas Statut NOT_FOUND

Pas de données de sortie

2)Requete PUT/player

Données en entrée : type: "object",

properties: {

"name": { type:string },

},

required: ["name"]

Si il reste une place Statut CREATED

Données sortie :

type: "object",

properties: {

"id": { type:number,minimum:0,maximum:2 },

},

required: ["id"]

Si plus de place libre Statut OUT_OF_RESOURCES

Pas de données de sortie

3)Requête POST /player/<id>

Données en entrée : type: "object",

properties: {

"name": { type:string },

},

required: ["name"]

Si joueur <id> existe Statut NO_CONTENT

Pas de données de sortie

Si joueur <id> n'existe pas Statut NOT_FOUND

Pas de données de sortie

4)Requête DELETE /player/<id>

Pas de données en entrée

Si joueur <id> existe Statut NO_CONTENT

Pas de données de sortie

Si joueur <id> n'existe pas Statut NOT_FOUND

Pas de données de sortie

7.2 Conception logiciel

7.2.1 Répartition sur plusieurs threads

Classes Engine:

— Run : execute les commandes à l'infini

7.2.2 Sauvegarde d'une partie

Classes Engine :

— UpdateRecord : Enregistre les commandes dans un fichier

7.2.3 Chargement d'une partie

Classes Engine :

— UpdatePlay : execute les commandes à partir d'un fichier

7.2.4 Serveur

Classe Client:Cette classe contient toutes les informations permettant de faire fonctionner le jeu ,c'est un observateur du jeu.

Classe Game : dans cette classe on trouve la liste des joueurs .

Classe Service : Les services REST sont implantés via les classes filles de AbstractService, et gérés par la classe ServiceManager :

— 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.