FAUJEAU François
BALACHANDRAN Chirojean
3A IS

PROJET CIVILIZATION

Rapport du projet logiciel transversal

Année scolaire : 2018-2019

Table des matières

1 Présentation générale

1.1 Archétype

Le but de notre projet logiciel transversal est de créer une version très simplifiée du jeu civilization. Ce jeu est un jeu de stratégie tour par tour où l'objectif d'un joueur est de développer un empire en choisissant une civilisation au choix parmi plusieurs.

FIGURE 1 – Image d'un gameplay de civilization 3

1.2 Règles du jeu

Le but pour chaque joueur est de développer un empire avec la civilisation romaine. Pour cela, le joueur doit améliorer et gérer ses ressources (mines d'or, terres agricoles) ainsi que ses bâtiments (caserne et palais).
Nous avons choisi que chaque joueur pourrait seulement utiliser la civilisation romaine. Chaque empire permet de créer différents bâtiments :

1. Le palais : bâtiment principal de l'empire. Son niveau correspond au niveau de l'empire. Plus sont niveau est élevé et plus il est possible d'améliorer les autres bâtiments et unités.
2. La caserne : permet de former des soldats. Quand le niveau de la caserne augmente, le niveau des troupes formées et le nombre des troupes qui peuvent être formées augmentent. Il y aura quatre types d'unités de combat : un cavalier, un épéiste, un archer et une catapulte. A la création d'une unité, le joueur choisira si elle est créée pour défendre l'empire ou si elle peut se déplacer sur la carte de jeu.
3. Ressources : le joueur possèdera trois types de ressources à savoir les mines d'or, la réserve de bois et les terres agricoles. Quand le niveau d'une ressource

augmente, la capacitée de stockage de cette ressource et la quantité produite sont augmentées.

Pour améliorer les bâtiments, il faudra que le joueur dépense une certaine quantité de bois et d'or et pour la formation de soldats une certaine quantité de récoltes agricoles et d'or.
Pour développer son empire plus rapidement il est possible d'en combatre un autre (détenu soit par l'IA, soit par un joueur) ou de s'en défendre. En effet, en cas de victoire l'empire victorieux reçoit une partie des ressources de l'empire adverse et un bonus de victoire. L'empire qui a perdu recupère, lui, un faible bonus de défaite.
Le système de combat est le suivant. Chacun leur tour, les joueurs pourront décider de :

— se déplacer pour se rapprocher du combat.
— choisir qu'un soldat en attaque un autre s'il est dans sa zone d'action.

Un combat se termine lorsque tous les soldats d'un joueur sont éliminés.
Dans notre jeu, un tour correspond à une action : une amélioration de bâtiment, la formation d'une unité, son déplacement.

1.3 Ressources

Voici les différentes ressources (images, textures, sprites) utilisées pour le développement du jeu. Pour créer les cartes de jeu nous les génèrerons de manière aléatoire en mode isométrique. Ces cartes sont des cartes de jeu 25 x 25.

FIGURE 2 – Bâtiments de la civilisation romaine en fonction des niveaux

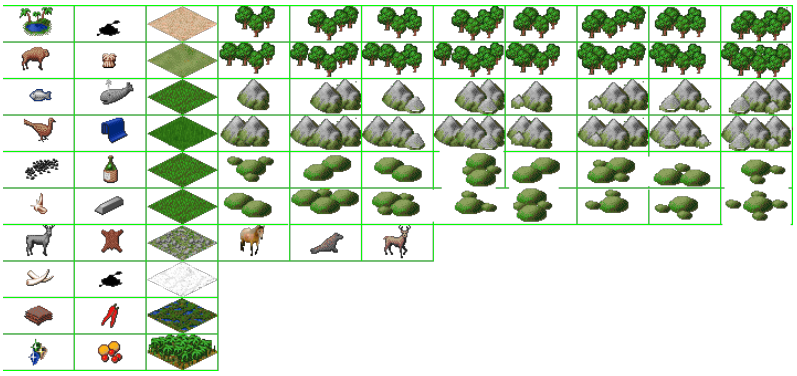

FIGURE 3 – textures pour la création de la map

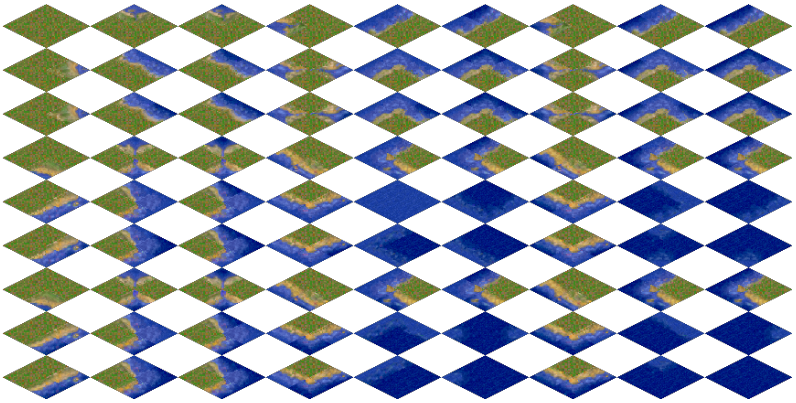

FIGURE 4 – textures pour la mer sur la map

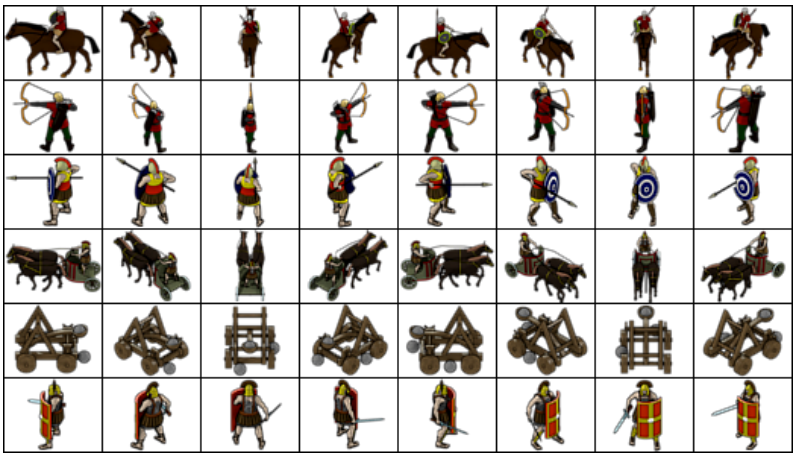

FIGURE 5 – images des unités de combats

2 Description et conception des états

2.1 Description des états

Un état de jeu est constitué d'une carte sur lequel se situent différents empires avec leurs différents bâtiments (éléments fixes), les éléments de décor (éléments fixes) et des unités de combats (éléments mobiles).

2.1.1 Description de la carte

La carte est représentée avec une grille composée de cases contenant des éléments sur chacunes d'elles avec une certaine position. Elle est composée d'empires, qui sont eux-mêmes constitués de bâtiments, et d'éléments comme les bâtiments, les unités de combats et les éléments de décor. Certains éléments sont franchissables par les unités et d'autres non. La taille de la carte est fixée.

2.1.2 Description des états éléments fixes

Les éléments fixes sont les décors et les bâtiments et les empires qui sont abstraits. Tous les éléments fixes possèdent une position (x,y) entre 0 et 24.

Décors

Les décors sont de différents types : eau, herbe, dunes, montagnes, arbre, animaux... Ils ont chaucun un identifiant. Les décors possèdent un attribut passable qui permet de savoir si l'élément est franchissable ou non par les unités de soldats.

Empire

Un empire a plusieurs attributs :
— un identifiant pour le différencier des autres empires (chiffre)
— un nom
— un niveau compris entre 1 et 4
— des points de vie qui sont égaux à ceux du palais
— des ressources (or, bois, nourriture) pour se développer et former des troupes
— une liste de position permettant d'indiquer où se positionne l'empire sur la carte

Enfin, il est composée d'une caserne, d'un bâtiment ressource et enfin d'un palais.

Bâtiments

Les bâtiments ont tous ces attributs :
— un identifiant pour le différencier des autres (chiffre)
— un niveau qui va de 1 à 4
— un coût en bois et en or pour les construire
— un identifiant pour leur texture graphique

Il y a trois types de bâtiments : la caserne, les ressources et le palais.

La caserne a une capacité maximale de formation de troupes et possède comme attribut le nombre de troupes formées. Elle permet de former les différentes troupes.

Les ressources ont un niveau de production identique pour l'or, le bois ainsi que la nourriture. Ils vont permettre d'augmenter la richesse d'un joueur à chaque tour.

Le palais quant à lui possède des points de vie. C'est lui qui sera attaqué par un joueur et sa destruction mènera à la fin d'une partie.

Une carte de jeux typique sera composée de 25 x 25 cases disposées de manière isométrique. Ces cartes seront générées de manière aléatiores.

2.1.3 Description des états éléments mobiles

Tous les éléments mobiles possèdent une position (x,y). Ils possèdent tous :
- un identifiant pour les différencier
- un niveau de vie
- un niveau de dommages en attaque
- une portée pour attaquer nombre de déplacements de cases limité
- un coût en or et en nourriture pour la former
- un identifiant de texture graphique.

Les unités de soldats dans le jeu sont les unités de combats : décurion, cavalier, catapulte et archer.

2.2 Conception des états

Dans cette partie nous allons décrire le diagramme UML d'état du jeu. (Voir figure 6)

Class "Element" : Cette classe est la classe mère de tous nos éléments, mobiles ou statiques. Nous avons choisi pour cette classe qu'elle contienne une position, la position de notre élément sur la map, et une méthode isPassable(), qui nous permet de savoir si l'élément peut être traversé par un autre.

Class "Units" : Cette classe est la classe mère de toutes nos unités mobiles sur la carte. Elle hérite d'"Element". Elle contient tous les paramètres nécessaires à la création d'une unité (vie, attaque, niveau, ...). Cette classe contitent de plus une méthode isPassable pour forcer le fait qu'on ne peut pas passer à travers une unité.

Class "Arrow" : Cette classe est la classe fille de Units. Elle hérite d'"Element". Elle permet de construire des unitées de type archer avec leurs attributs initialisés en fontion de leur niveau. Ce sera la façon la plus utilisée de créer un archer. Il existe de même des classes pour Decurion, Cavalier et Catapult.

Class "Buildings" : Cette classe est la classe mère de toutes nos unités statiques sur la carte. Elle hérite d'"Element". Elle contient tous les paramètres nécessaires à

la création d'un bâtiment (niveau, coût, ...). Cette classe contitent de plus une méthode isPassable pour forcer le fait qu'on ne peut pas passer à travers un bâtiment.

Class "Barrack" : Une caserne est un bâtiment et hérite donc de "Buildings", elle ajoute en plus quelques fonctionnalitées telles que la création d'unités. De même pour les bâtiments de type ressource et palais.

Concernant les coûts, nous avons créé deux classes qui permettent de créer le coût des unités et des bâtiments, et de pouvoir les modifier.

Nous avons aussi choisi d'attribuer à chaque élément un id qui permet de savoir instantanément lequel il est.

Class "Decor" : Cette classe hérite d'élément, elle permet de créer les différents décors qui seront présents sur la carte de jeu.

Class "Empire" : Dans notre jeu nous avons besoin à tout instant d'avoir une classe qui stocke les éléments importants d'un empire. Par exemple un empire à un nom, une vie, de l'or, du bois, de la nourriture, ... Ainsi, cette classe va agir comme un conteneur de tous les élements clés de notre empire.

Class "Map" : cette classe contient les éléments sur la carte de jeu. Elle permet d'ajouter des éléments, de les récupérer ou d'en supprimer.
Nous avons procédé de la manière suivante :

- basicMap : ce tableau de "unique_ptr" contient seulement des pointeurs de "Decor" de type herbe. Il s'agit de la carte de base sur laquelle va se supperposer le reste des textures.
- decorMap : ce tableau de "unique_ptr" va contenir tous les autres décors que l'herbe à savoir les forêts, les montagnes, les oceans et les animaux.
- unitsMap : ce tableau de "unique_ptr" contiendra les unités lorsqu'elles seront crées par nos observeurs.
- buildingsMap : ce tableau de "unique_ptr" contient tous les batiments des différents empires qui jouent.
- selectedMap : ce tableau de "unique_ptr" contient a chaque état du jeu, la ou les cases qui doivent être affichées en surbrillance pour aider le joueur.
- statsMap : ce tableau de "unique_ptr" contient les stats associées à chaques bâtiments, unités.
- mapMatrix : ce tableau de "int" contient l'id de chaque type de bâtiments. Il nous permet de créer les autres tableaux pour éviter que les décors, les bâtiments, les unités se chevauchent.

Il y a ensuite un getter pour chacun de ces tableaux.
Pour créer la map nous avons plusieurs possibilités : si on est en "record" on doit créer mapMatrix comme d'habitude mais l'enregistrer dans notre fichier JSON, si on est en "play" on remplit "mapMatrix" avec ce qui est enregistrée dans le JSON et, dans l'autre cas on créé la map normalement.

Class "Observable" : cette classe permet de réagir au des actions gérées par le moteur de jeu. Par exemple, lorsqu'un joueur souhaite améliorer un bâtiment ou créer une unité sur la carte de jeu, il faut que les cartes contenues dans notre classe Map soient modifées en conséquence.
Elle contient une méthode appelée "notifyObserveur" qui va notifier ses observeurs et une méthode "notifyObserveurPrev" pour notifier ses observeurs lors du rollback. Cet observable est composé de deux observeurs : UnitsObserver et BuildingsObserver. NotifyObservers appelle ensuite le bon observeur en fonction des paramètre qu'on lui donne. La classe "Observable" contient de plus une méthode "getAllMaps" qui va permettre de récupérer les maps générées par le constructeur de Map. Par la suite, pour créer nos cartes de jeu initiales on créera une instance de "Obsevable". Elle contient aussi en attributs deux listes qui permettent de stocker les unités détruites (ressurectionUnits) ainsi que les empires détruits (ressurectionPalace) qui permettront de remettre ces éléments lors d'un rollback. A l'instanciation de cette classe nous allons pouvoir choisir si on souhaite enregistrer la partie (Record), rejouer la partie enregistrée (Play) ou jouer une nouvelle partie sans enregistrer. Dans les cas où nous souhaitons rejouer une partie ou en enregitrer une, l'observable appelle les méthodes "beginRecord" ou "beginReplay". Ces méthodes vont ensuite communiquer avec la class "Map" et lancer la construction de la carte de jeux en fonction de ce qu'on veut.

Class "UnitsObserver" : Cette classe est un Observer sur les Units. Elle va permettre de réaliser tous les changements nécessaires sur le state des units demandés par le moteur.

Class "BuildingsObserver" : Cette classe est un Observer sur les Buildings. Elle va permettre de réaliser tous les changements nécessaires sur le state des buildings demandés par le moteur.

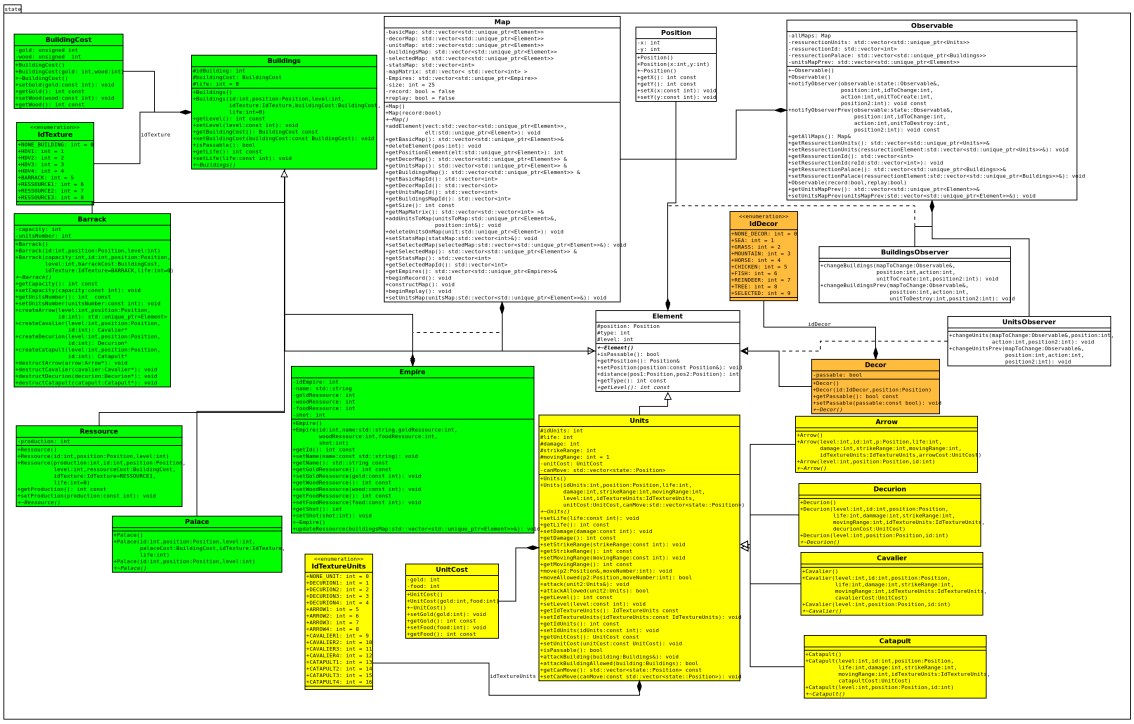

FIGURE 6 – Diagramme de classes d'état

3 Rendu : Stratégie et conception des états

3.1 Stratégie de rendu d'un état

Pour le rendu des états, nous avons décidé de faire une carte en 2D avec des tuiles et en isométrie. Cette carte sera à chaque fois générée aléatoirement et donc à chaque lancement du programme elle sera différente. Nous utilisons la librairie SFML. Nous avons trois tilesets qui vont nous servir à créer les textures de la carte :

— un pour les textures du terrain (eau, herbe, animaux ...)
— un pour les bâtiments (caserne, ressource, palais)
— un pour les unités (archer, cavalier, decurion, catapulte)

Sur le côté gauche, un petit menu permet de donner le niveau, la vie et les dommages s'il y en a des éléments.

Nous aurons une carte divisée en quatre plans :

— un pour l'herbe et l'eau qui sont nos éléments visuels de base
— un pour les éléments de décor comme les montagnes, les arbres ou encore les animaux
— un pour les bâtiments
— un pour les unités mobiles

Les trois premiers plans seront initialisés au départ du jeu grâce aux textures et à des tableaux constitués des numéros de texture d'éléments, selon une position, générés aléatoirement, selon certaines règles de création de map :

— Toute la carte est préalablement constitué d'herbe seulement
— Ensuite, trois zones d'eau sont tirées au sort aléatoirement

— Puis, trois zones d'arbres et de montagnes sont tirées aléatoirement et quelques animaux sont placés
— Enfin, trois positions pour les empires sont tirées au sort et les trois bâtiments sont placés côte à côte

Tous les plans seront ensuite superposés pour former la carte. Les troix premiers plans seront fixes et seuls les plans des unités et des sélections des cases changeront. Ce dernier sera en effet modifié à chaque tour lorsque l'utilisateur fera déplacer ses unités mobiles. La mise à jour de l'affichage et du changement d'états se fera à une fréquence de quelques hertz.

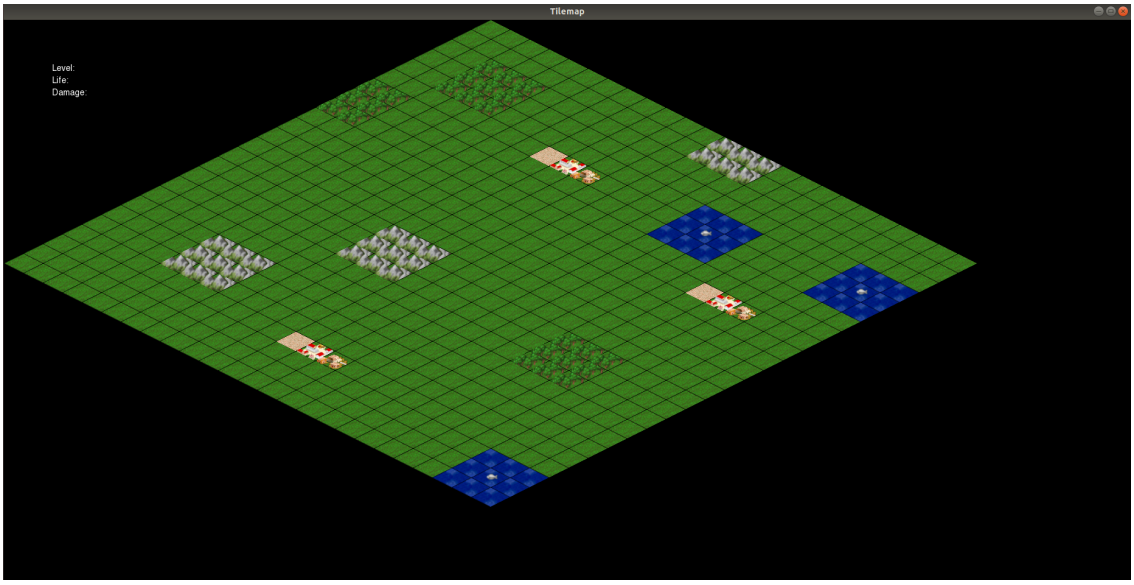

FIGURE 7 – Exemple d'un état généré aléatoirement

3.2 Conception de rendu d'un état

Dans cette partie nous allons décrire le diagramme UML de rendu d'un état du jeu.

Class "Tiles" : C'est un élément de base des tilesets. Ils ont une longueur et une largeur comme attributs ainsi qu'une position dans le tileset.

Class "TileSet" : Cette classe est la classe mère de tous les tilesets (tableaux de tuiles associés aux éléments). Elle est composée de tuiles. Nous avons choisi pour cette classe qu'elle contienne une méthode pour connaître la largeur d'une tuile et une autre qui permet de renvoyer la hauteur d'une tuile.

Class "BuildingTileSet" : Elle hérite de "TileSet". Elle contient trois tableaux de tuiles comme attributs (un pour les casernes, un pour les palais et un pour les ressources) permettant de donner la texture associée à chaque élément. Elle possède une méthode permettant de renvoyer le chemin du fichier png contenant le tileset et une autre pour obtenir la tuile d'un bâtiment donné.

Class "DecorTileSet" : Elle hérite de "TileSet". Elle contient un tableau de tuiles pour les décors comme attributs permettant de leur donner la texture associée. Elle possède une méthode permettant de renvoyer le chemin du fichier png contenant le tileset et une autre pour obtenir la tuile d'un décor donné.

Class "UnitsTileSet" : Elle hérite de "TileSet". Elle contient quatre tableaux de tuiles comme attributs (un pour les archers, un pour les cavaliers, un pour les décurions et un pour les catapultes) permettant de donner la texture associée à chaque unité. Elle possède une méthode permettant de renvoyer le chemin du fichier png

contenant le tileset et une autre pour obtenir la tuile d'une unité donnée.

Class "RenderMap" : Cette classe permet de créer une carte avec les textures d'un tileset. Elle constitue les plans d'une carte On empile ici les différents layers pour les différentes maps dans leur ordre de priorité. Cette classe contient des textures et un tableau en attributs. Elle permet d'ajouter une image de fond d'écran pour le jeu, de mettre à jour tous les layers et de les réaffficher à l'écran.
Cette classe contient aussi la méthode "handle" qui permet de gérer l'intéraction du joueur avec le rendu et d'envoyer les commandes correspondantes au moteur. Cette méthode utilise une méthode de récupération de click pour savoir où le joueur a cliqué sur l'écran et réagir en fonction.

Class "Layer" : Cette classe permet de créer des plans de carte. Elle est composée de RenderMap. Elle a comme attributs les tableaux de textures, selon leurs position, des éléments de base (herbe, eau), des décors et des bâtiments. Elle fait le lien entre la map du jeu contenue dans le diagramme d'état et le rendu. Elle a des méthodes qui permettent de charger un fichier de texture, de positionner un sprite sur une carte et lui donner sa texture et enfin d'afficher cette carte. Elle permet de plus d'initialiser le layer des stats qui affichent la vie, les ressources, les niveaux ainsi que le layer qui affiche les boutons lorsque le joueur a sélectionné une caserne pour créer une unité ou faire une amélioration de bâtiment. Cette classe nous permet aussi d'afficher du texte à l'écran.

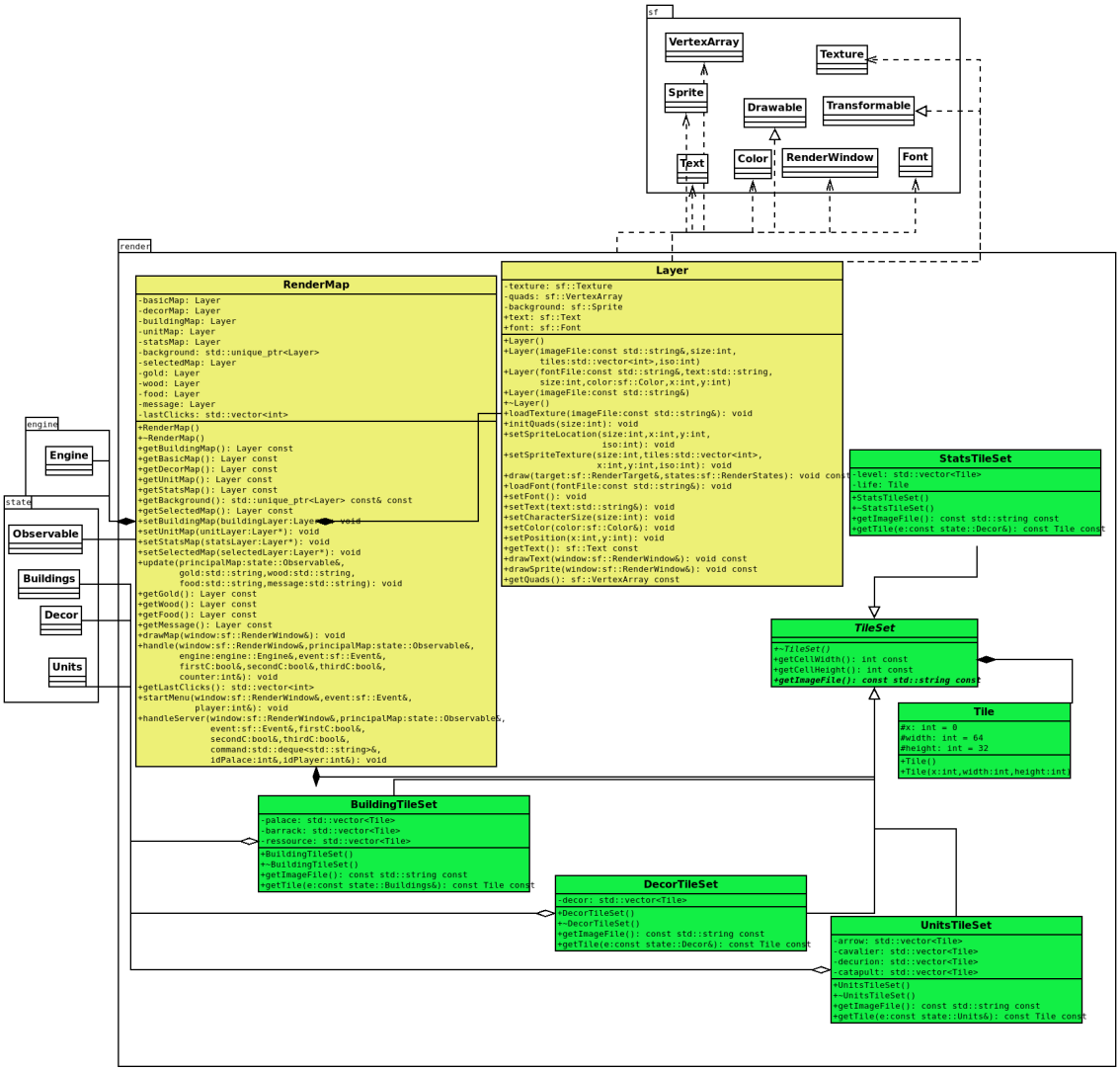

FIGURE 8 – Diagramme de classes du rendu

,

4 Règles de changements d'états et moteur de jeu

4.1 Horloge globale

Tous les changements dans le jeu suivent une horloge globale pour le passage d'un état à un autre.
Ces changements sont réalisés en fonction du temps de passage d'un état à un autre. On attend la fin du changement d'état pour en réaliser un nouveau.

4.2 Changements extérieurs

Les changements extérieurs sont générés par des cliques du joueur. Les commandes sont les suivantes :

- Cliquer sur n'importe quelle texture et consulter les actions qui lui sont associées ainsi que ses attributs (niveau de vie avec les coeurs, niveau, capacité ...).
- Augmenter le niveau des bâtiments.
- Créer des unités en séléctionnant leurs positions initiales avec un clique.
- Déplacer des unités en choisissant les cases parmi celles possibles.
- Faire attaquer des unités entre elles.
- Attaquer un hôtel de ville.

4.3 Changements autonomes

Ces changements sont ceux qui sont gérés par le moteur sans être provoqués par des actions utilisateurs

- Augmentation des ressources à chaque tour.
- Disparition des unités qui n'ont plus de vie ainsi que de l'empire du palais n'ayant plus de vie.
- Règles de mouvement et d'attaque.
- Affichage des statistiques des bâtiments et des unités.

4.4 Conception logiciel

Le diagramme des classes du moteur est en figure 9 et diponible dans le dossier src du projet sous le nom de engine. Dans cette étape de conception logiciel nous avons utilisé un pattern command. A son exécution, le moteur appelle des commandes en fonction des actions utilisateur.

Classes de commandes : Elles héritent toutes de la classe Command. Les commandes suivantes sont disponibles :

- Engine : cette classe contient quatre contener :
  - CommandList : Ce conteneur contiendra la liste des commandes à exécuter. On met en queue toute les commandes qui doivent être exécutées lorsqu'un joueur (physique ou ia) veut réaliser une action.
  - CommandListId : qui contient l'id de chaque commande mise en queue pour pouvoir caster ses comandes sur leurs bon types et les lancer. On

empile et dépile en même temps dans ces deux conteneurs. On empile et dépile en même temps dans ces deux conteneurs.

- CommandListPrev : Ce conteneur contiendra la liste des commandes à exécuter. On met en deque toute les commandes qui doivent être exécutées lorsqu'un joueur (physique ou ia) veut réaliser un rollback.
- CommandListIdPrev : qui contient l'id de chaque commande mise en deque pour pouvoir caster ses comandes sur leurs bon types et les lancer. De plus, le moteur a un attribut text qui contient le numéro du joueur actuel ainsi que son action en cours. Cet attribut sera affiché lors du rendu.

Engine a une méthode run qui permet de connaître le prochain joueur à jouer ainsi que la mise à jour des ressources à chaque fin de tour et savoir si la partie est terminée. Engine contient de plus une méthode "execute" qui, tant que la queue n'est pas vide, exécute les actions et les dépiles. Dans le cas ou on a décidé d'enregistrer les commandes, elles sont ajoutées à une chaine de caractères et enregistrées au format JSON à l'exécution du destructeur. Le moteur a une méthode rollback qui permet de revenir à un état précédent. Méthode "replay" : cette méthode permet de lire toutes les commandes au format json et les ajouter à la FIFO de commandes. Ainsi, la méthode "execReplay" permet d'exécuter toutes les commandes empilées.

Les différentes commandes sont les suivantes :

- CaseIdentifier : Permet d'identifier la case sur laquelle on se trouve.
- Possibilities : Permet de déterminer quelles sont les actions possibles à partir de cette case.
- PrintStats : Affiche les stats (vie, niveau, bouttons) des joueurs et des bâtiments.
- LevelUp : Augmente le niveau d'un bâtiment.
- CreateUnit : Créer une unité à partir d'un premier clique sur une caserne. Le second clique permet de positionner l'unité crée sur la map. Des boutons seront disponibles pour choisir le type d'unités à créer.
- Move : Permet de déplacer une unité
- Attack : Permet d'attaquer une unité ou un hôtel de ville.

Chaque commande a une méthode pour éxecuter et une méthode pour faire la commande inverse correspondante.

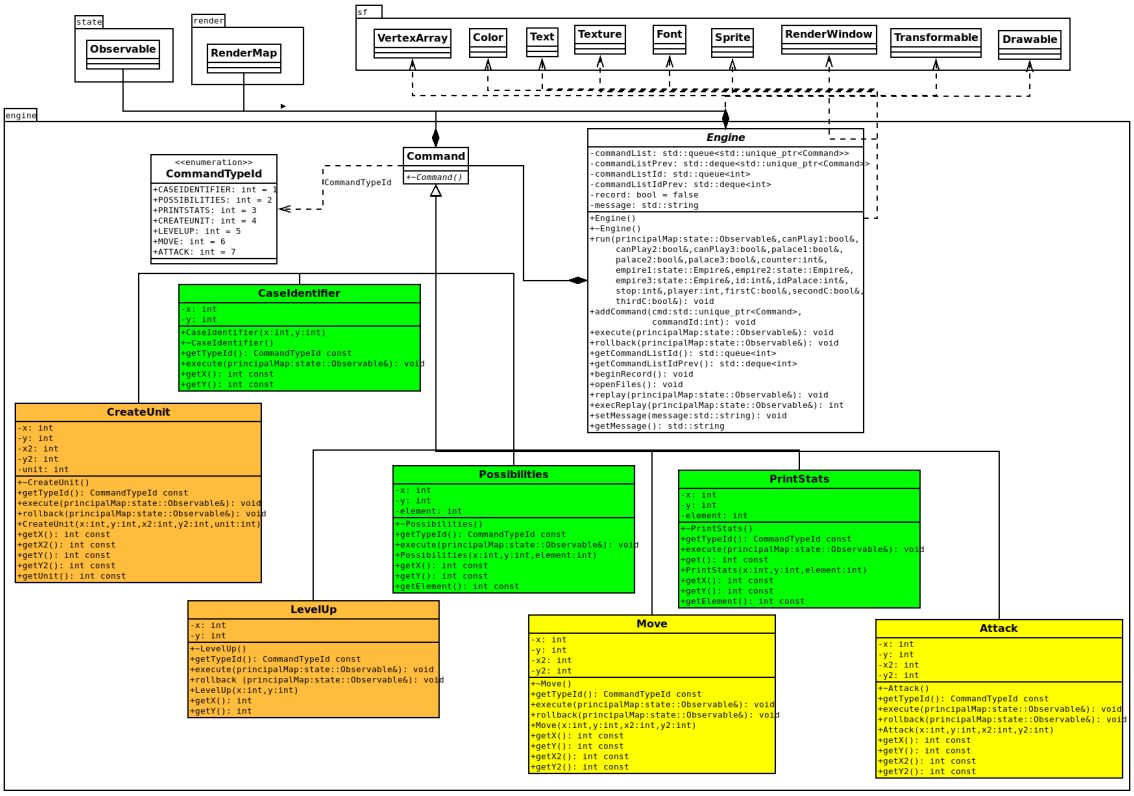

FIGURE 9 – Diagramme de classes du moteur

Voici une description de comment est implémenté le jeu grâce au moteur :

— On instancie un Observable qui instancie toutes les cartes de jeu.
— On instancie un objet pour actualiser le rendu.
— On instancie le moteur pour pouvoir l'utiliser par la suite.
— On réalise le premier affichage de la map.
— On ouvre une boucle qui tourne tant que la fenêtre est ouverte. C'est dans cette boucle que le jeu est réelement.
— On met en place une logique de gestion de tours
— On appelle une méthode "handle" qui permet d'ajouter les actions du joueur en tant que commandes dans les listes de commandes du moteur.
— On appelle la méthode execute du moteur qui exécute toutes les commandes contenues dans sa liste de commandes.
— On met à jour l'affichage.

5 Intelligence artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

Cette stratégie est la même pour tous les empires. A chaque tour, chaque empire peut effectuer 3 actions : créer une unité, améliorer un bâtiment, déplacer une unité, attaquer un bâtiment ou une unité ennemie avec un de ses soldats. On choisit donc trois actions au hasard parmis celles-ci et on les effectue.

5.1.2 Intelligence basée sur des heuristiques

Cette intelligence artificielle est basée sur des heuristiques que nous avons définies et qui, selon nos critères, doivent permettre de challenger un joueur humain. Nous pensons en effet qu'un empire a, par exemple, intérêt à augmenter le niveau de ses bâtiments en fonction de ses moyens dès qu'il peut. Ensuite, l'empire devra créer des unités en fonction de ses capacités (nombre d'unités pouvant être formées pour chaque empire limité). Enfin, l'empire pourra déplacer ses unités pour les mener à la conquête ou au combat.
Pour cela notre système intelligent va procéder comme suit en fonction de ses ressources et capacités de bâtiments :

1. Si l'empire a assez de ressources, l'AI doit améliorer son palace en premier (seul bâtiment pouvant se faire attaque).
2. Améliorer la caserne qui permet d'améliorer les unités qui peuvent être formées.
3. Améliorer les ressources pour que la production à chaque tour soit augmentée.
4. Créer une unité, le choix de l'unité à créer étant aléatoire.

Une fois que ceci est fait nous gérons la partie mobile de notre jeu.
Voici comment se décompose le système de déplacement et d'attaque de nos unités.

1. Si une unité ennemie se situe à coté d'une unité de l'empire, on choisit cette unité et l'unité ennemie est attaquée.
2. Si une unité de l'empire se situe à côté du Palace d'un empire ennemi, on choisit cette unité et, si elle est aussi à coté d'une unité ennemie elle attaque l'unité en priorité, sinon elle attaque le palace ennemi. Sinon :
3. On choisit une unité aléatoirement de l'empire courant.
4. Si elle se situe à moins de trois cases d'une unité ennemie, on s'en rapproche pour aller au combat.
5. Sinon elle se rapproche de l'empire ennemi le plus proche.

Dans notre système, l'amélioration des bâtiments est privilégiée par rapport au déplacement ou attaques de nos unités. Si une unité d'un empire tombe au combat, cet empire pourra en créer une nouvelle en fonction de ses moyens au tour suivant. Nous pouvons de plus remarquer qu'il y a parfois une unité qui ne va pas vers la

bonne direction pendant au maximum 3 coups. Cela est dû à notre heuristique de déplacement qui dans un seul cas peut poser ce problème. Cependant cela arrive que très rarement et n'est pas pénalisant pour notre IA heuristique.

5.1.3 Intelligence avancée

Cette intelligence artificielle est basée sur des méthodes de résolution de problèmes à états finis. Pour cette intelligence artificielle nous avons choisi d'utiliser un algorithme de type min-max pour déterminer la meilleure action à réaliser. Ensuite, l'action déterminée par l'algorithme est réalisée avec l'heuristique de chaque action. Si l'action renvoyée par le min-max est un move, on appelle l'heuristique de move pour le réaliser...
Pour réaliser le min max nous avons dû limiter le nombre d'actions qu'il prend. En effet, nous avons 3 empires. Chaque empire peut réaliser 1 coup parmi 55 et ceux trois fois de suite. Ainsi, si on utilise toutes ces actions nous avons un nombre trop important de coups possibles et le temps de calcul pour optimiser une action serait très important.
Le min-max fonctionne de la façon suivante :

1. On fait réaliser à l'empire en cours une de ses actions possibles.
2. On fait réaliser à l'empire suivant une de ses actions possibles.
3. On fait réaliser au dernier empire une de ses actions possibles.
4. On actualise les poids
5. A la fin de chaque action et actualisation des poids on effectue un rollback de l'action précédente pour ne pas que l'état soit modifié indéfiniemment.

On itère ce processus pour toutes les actions possibles de chaque empire. Une fois que toutes les branches du dernier empire sont explorées, on fait remonter les poids en les minimisant au second empire à traiter. Une fois que toutes les branches du second empire sont explorées, on fait remonter les poids en les minimisant au premier empire à traiter. A la dernière étape, on maximise les poids pour maximiser l'influence du coup de l'empire 1 sur les deux autres.
Lors de notre exploration des branches, celles qui ne sont pas réalisables ne sont pas explorées. Par exemple, lorsqu'il n'y a pas d'unités sur la map on n'essaie pas de faire de actions de type "move" ou "attack".
Cette intelligence avancée pourrait être très efficace si elle pouvait prévoir la meilleure chose à faire en fonction de toutes les actions de chaque tour de chaque empire possibles. Seulement, ceci serait bien trop long. Ainsi, le modèle proposé réalise une recherche sur un coup de chaque empire et ne prend pas en compte le fait qu'un empire peut réaliser 3 actions à chaque tour.

Pour cette IA, un rollback, qui permet de revenir à un état précédent, a été implémenté afin de faire le min-max. Nous avons décidé de mémoriser les commandes dans un deque puis de dépiler la dernière lorsque le rollback est utilisé. Chaque commande a une méthode execute qui permet d'exécuter la commande et une méthode rollback qui permet de faire la commande inverse et ainsi revenir à l'état précédent.

5.2 Conception logiciel

— Classe AI : qui sera la mère des trois différentes intelligences artificielles.
— Classe RandomAI : contient une méthode run qui permet à l'intelligence artificielle naïve de lancer ses coups.
— Classe HeuristicAI : contient une méthode run qui permet à l'intelligence artificielle heuristique de lancer ses coups. Elle utilise les heuristiques définies dans la sous section précédente. Pour réaliser le déplacement des unités nous avons besoin d'une "mémoire". En effet, notre carte contient des obstacles. Ainsi, pour les contourner nous avons besoin à chaque mouvement de rechercher la case la plus proche de la cible mais qui n'implique ni un retour en arrière pénalisant ni d'être bloqué par un obstacle. Nous avons donc ajouté un attribut à la classe Units qui s'appelle "canMove" et qui stocke les 3 dernières cases utilisées par l'unité.
— Classe DeepAI : Contient une méthode "run" qui fait appel à une fonction minmax. Cette fonction minmax fait appel à une fonction min et une fonction max qui permettent de trouver le min ou le max d'une portion de tableau. La fonction min max réalise de plus des actions avec la méthodes preformActions puis elle les rollback et renvoie le meilleur coup à jouer à la méthode "run". Le poids de chaque action lorsqu'on est en bas de l'arbre est calculé par la fonction weightUpdate. Cette fonction permet de vérifier ce que peut faire le joueur et de renvoyer le poids de chaque action possible ou -1 dans le cas ou l'action ne peut pas être faite. Chaque action renvoie à une heuristique qui vérifie sa faisabilité et l'exécute ensuite. Elle a de plus une méthode runServer qui lance une IA heuristique correspondant aux attentes d'un jeu en réseau.
— Classe AStar : Cette classe permet d'utiliser l'algorithme AStar qui est un algorithme de recherche du plus court chemin entre deux points. Pour appliquer cette algorithme nous avons empêché que l'algorithme cherche à passer par des obstacles non franchissables.

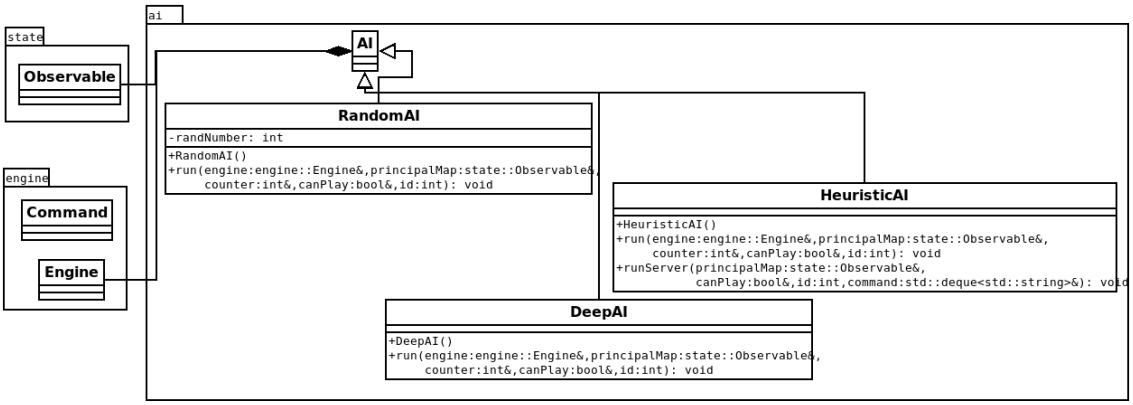

FIGURE 10 – Diagramme de classes de l'ia

6 Modularisation

6.1 Organisation des modules

6.1.1 Serialisation des commandes

L'objectif de cette partie est d'être capable d'enregistrer, si l'utilisateur le souhaite, l'ensemble des commandes et des statistiques de jeu de la partie en cours pour pouvoir la rejouer entièrement. Ceci n'est qu'une petite application de la sérialisation de commandes. En effet, son objectif principal est de pouvoir être utilisable par le serveur. La partie serveur de notre application est celle qui contiendra l'intégralité des calculs à savoir le moteur de jeu et les intelligences artificielles qui calculent les coups suivants. Ainsi, pour réaliser les commandes d'un joueur, il faut les envoyer au serveur. Le format utilisé est le format JSON. Il a été choisi pour sa simplicité d'écriture et de lecture.
Pour la partie "record" et "replay" nous devons :

- Enregistrer la map aléatoire pour pouvoir la regénérer dans le "replay".
- Enregistrer toutes les commandes effectuées par les intelligences artificielles pour pouvoir les rejouer totalement.

Pour l'enregistrement général de notre partie, nous avons décidé d'utiliser un booléen à l'instanciation de notre observable qui dit si nous souhaitons faire un "record", un "replay" ou aucun des deux.

Partie "Record" :

Dans cette partie, nous enregistrons tout d'abord la map créée dans la classe "Map" de "State". Puis l'intégralité des commandes dans la classe "Engine" de "Engine". Pour cela, nous créeons une chaîne de caractère contenant les différentes instructions puis nous l'ajoutons au fichier JSON. L'écriture dans le fichier JSON de l'ensemble des commandes est fait dans le destructeur. Ainsi, il faut stopper l'exécution du code en fermant la fenêtre sfml et non en effectuant un "ctrl+c" dans le terminal. Pour réaliser le record il faut taper la commande : ./bin/client record.

Partie "Play" :

Dans cette partie, nous lisons simplement le fichier au format JSON. Tout d'abord, dans le constructeur de la classe "Map" pour reconstituer la carte enregistrée. Puis, dans engine pour exécuter les commandes enregistrées une par une. Pour ceci, nous utilisons la bibliothèque "jsoncpp" qui permet de parser et de lire très facilement des données au format JSON. Cette partie se fait dans le moteur.

6.1.2 Répartition sur différents threads

L'objectif est de placer le rendu dans un thread principal et le moteur dans un thread séparé.

Nous avons en tout quatre threads :

- thread principal : il permet d'initialiser au début les différentes variables, la fenêtre puis il gère l'affichage. Ce dernier est mis à jour dès que le moteur exécute une commande. Pour cela, le thread moteur envoie un signal pour

informer au thread principal de rafraîchir le rendu.
— thread moteur : il exécute en permanence les commandes dès qu'il peut et émet un signal pour informer le thread principal de la mise à jour du rendu.
— thread IA : il ajoute des commandes pour le joueur en cours. Il est exécuté après le thread secondaire.
— thread secondaire : il est exécuté avant le thread IA. Il permet de connaître le numéro du joueur à jouer et de faire les mises à jour des ressources.
— quatrième thread : Il permet d'envoyer les commandes du client lorsqu'il y en a en réseau
— cinquième thread : Il permet de demander les nouvelles commandes au serveur et les ajoute au moteur du client en réseau

Ces deux derniers threads ont un mutex en commun pour éviter l'utilisation de ressources critiques en même temps. Nous devons mettre un certain délai entre l'éxecution des différentes commandes pour avoir un affichage et une exécution des commandes synchronisés.

6.1.3 Répartition sur différentes machines : rassemblement des joueurs

Comme expliqué précedemment, l'objectif global de la partie modularisation est de connecter différents joueurs à un réseau pour qu'ils puissent jouer ensemble. La première étape était de créer et de faire fonctionner la partie client de l'application de manière indépendante en multithread et de sérialiser les commandes.
Dans cette partie, l'objectif est de créer la partie serveur de l'application et de s'y connecter par l'intermédiaire du client en exécutant des requêtes sur le serveur. Sur le serveur, nous allons écrire des APIs Web de type REST. Les services sont de type CRUD, ils doivent être capables de réaliser des ajouts, lectures, mises à jour et suppression dans des objets qui stockent temporairement les données de la partie en cours.
Dans un premier temps, l'idée est de réaliser ces différentes requêtes sur un service qui gère les joueurs.

Nous détaillerons par la suite le corps de ces requêtes :

Requête GET /player/	
Cas pas de données en entrée	

Cas où il y a au moins un player	Statut OK
	status : 200
	type : "array"
	body : {
	"players" : [
	{
	"name" : {type :string},
	"type" : {type :number,ai :0,human :1}
	},
	{
	"name" : {type :string},
	"type" : {type :number,ai :0,human :1}
	},

	{
	"name" : {type :string},
	"type" : {type :number,ai :0,human :1}
	}
	]
	}

Cas où il n'y a pas de player	Statut OK
	status : 200
	type : "object"
	body : {
	"players" : null
	}

Requête GET /player/<id>

Cas pas de données en entrée	

Cas où il y a au moins un player	Statut OK
	status : 200
	type : "object"
	body : {
	{
	"name" : {type :string},
	"type" : {type :number,ai :0,human :1}
	}
	}

Cas où il n'y a pas de player	Statut OK
	status : 200
	body : {
	null
	}

Requête PUT /player

Données en entrée	

type : "object"
{
"name" : {type :string},
"type" : {type :number}
}	

Si place libre :	Status CREATED
	Données sortie :
	type : "object"
	body : {
	"id" : {type :number,minimum :1,maximum :3},
	}

Si plus de place libre :	Status NOT_CREATED Données sortie : Pas de données de sortie

Requête POST /player/<id>

Données en entrée	

type : "object"
{
"name" : {type :string},
"type" : {type :number}
}	

Cas joueur <id> existe :	Status NO_CONTENT Pas de données en sortie
Cas joueur <id> n'existe pas :	Status NOT_FOUND Pas de données en sortie

Requête DELETE /player/<id>

Pas de données en entrée	
Cas joueur <id> existe :	Status NO_CONTENT Pas de données en sortie
Cas joueur <id> n'existe pas :	Status NOT_FOUND Pas de données en sortie

6.1.4 Répartition sur différentes machines : partie en réseau et échanges de commandes

Au lancement du jeu, nous avons le choix entre une partie locale (1 joueur, 2 joueurs et une démonstration faite par les IA) et une partie en multijoueur. Lorsque le joueur choisit une partie en multijoueur, il doit attendre 10 secondes pendant lesquelles un autre joueur peut se connecter. Si le joueur est seul sur le serveur, il fera une partie en locale contre deux IA. Sinon, s'il y a un deuxième joueur connecté, la partie est lancée entre les deux joueurs avec une IA. Nous avons décidé pour ce livrable d'avoir une map fixe et non aléatoire pour éviter l'envoi de celle-ci. Si un troisième joueur essaie de se connecter, cela ne sera pas possible.

Échanges de commandes :

Nous avons décidé que le serveur sert que de relais pour les commandes, il reçoit les commandes de tous les clients et permet la redistribution à tous les clients. Ainsi chaque client à son propre rendu, calcul du joueur à jouer, moteur ... Ainsi le chemin de l'exécution d'une commande est le suivant : un client envoie une commande au serveur, tous les clients récupèrent la commande, tous les clients l'exécute.
L'API pour les commandes suit complètement le fonctionnement de l'API précédente. On peut réaliser toutes les requêtes définies précédemment avec "command"

à la place de "player". Les réponses à ces requêtes sont les mêmes. Nous ne recopierons donc pas la doc de l'API précédente en changeant "player" par "command". Ainsi lorsqu'un joueur clique sur la map et fait une action ou l'IA fait une action, les commandes associées à cette action sont stockées temporairement dans le client puis envoyées au serveur. Celui-ci numérote les commandes au fur et à mesure. Le client récupère en permanence les nouvelles commandes ajoutées sur le serveur et les ajoute dans son moteur qui les exécutera. Lorsqu'un joueur a fini son tour, il ne peut plus cliquer sur la map. Le joueur 1 commence bien évidemment la partie et c'est son IA qui joue à la place du troisième joueur.

6.2 Conception logiciel

Partie client :

La classe client a les attributs suivants :

- un moteur
- une observable contenant les états du jeu
- le rendu
- une intelligence artificielle heuristique
- un mutex
- un deque qui stocke les commandes effectuées par le client (IA ou joueur)

La création d'un client permet donc d'avoir tous les éléments nécessaires pour faire tourner le jeu. La classe client a les méthodes suivantes :

- le constructeur
- une méthode run : elle initialise les différentes variables, la fenêtre d'affichage et le thread principal. Elle lance aussi les différents threads.
- une méthode aiUpdating : elle permet de faire tourner l'IA
- une méthode aiUpdatingServer : elle permet de faire tourner l'IA qui envoie les commandes dans le deque du client
- une méthode engineUpdating : elle permet d'exécuter les commandes et d'envoyer un signal pour mettre à jour le rendu
- une méthode playerUpdating qui met à jour le numéro du joueur à jouer ainsi que les ressources à chaque fin de tour
- une méthode playerUpdating qui met à jour le numéro du joueur à jouer ainsi que les ressources à chaque fin de tour lorque nous sommes en réseau
- une méthode connect : elle permet de tester la connexion au serveur. Ajouter un joueur à une partie, visualiser des joueurs, le supprimer. Cette méthode s'appuie sur les fonctions "sendPut", "sendDelete" et "sendGet" écrites pour éviter de dupliquer du code.
- une méthode commandSend qui envoie les commandes du client dès qu'il y en a
- une méthode commandRequest qui demande s'il y a de nouvelles commandes

Partie serveur :

Pour cette partie nous nous somme appuyés sur le td de moodle qui réalise un serveur REST très simple avec "microHTTPd". Nous avons fait le td et l'avons adapté à notre projet.

Notre conception contient les éléments suivants :

- Méthode "main" : créée et lance le serveur microHTTPd.
- Classe "ServiceManager" : elle permet d'enregistrer de nouveaux services pour communiquer avec le serveur, de trouver les services correspondant aux routes appelées par les clients, et d'interroger ces services en fonction de la méthode qu'ils utilisent (GET, POST, PUT, DELETE).
- Classe Abstract Service : c'est cette classe qui permet au ServiceManager d'instancier n'importe quel service. Cette classe est une classe abstraite qui est héritée par tous les services de notre projet. Elle contient toutes les méthodes (CRUD) qui seront nécessaires à nos différents services et qui seront ré-implémentées grâce au principe du polymorphisme.
- Classe "PlayerService" : Permet de réaliser les opérations de type CRUD sur la classe PlayerDB qui stocke dans un objet de type "map" les joueurs de la partie en cours. Un joueur est créé en instanciant un nouveau "Player" de la classe "Player". On peut ainsi ajouter, récupérer, modifier ou supprimer très facilement tous les joueurs contenues dans la "map" base de données.
- Classe "Command" : Cette classe permet de créer une commande qui sera enregistrée dans la base de donnée du serveur.
- Classe "CommandDB" : Cette classe contient la liste des commandes et permet de réaliser toutes les méthodes que l'API REST a défini (GET, POST, PUT, DELETE).
- Classe "CommandService" : Interroge la base de données "CommandDB" pour réaliser les requêtes HTML.
- Classe "VersionService" : permet de connaître la version de l'API en cours. Ceci permet d'éviter des conflits entre versions.
- Classe "ServiceException" : permet de déterminer et de renvoyer le bon statut à l'issue de chaque requête sur le serveur contenu dans l'énumeration "HttpStatus" ("OK, "BAD_REQUEST", "CREATED"...).

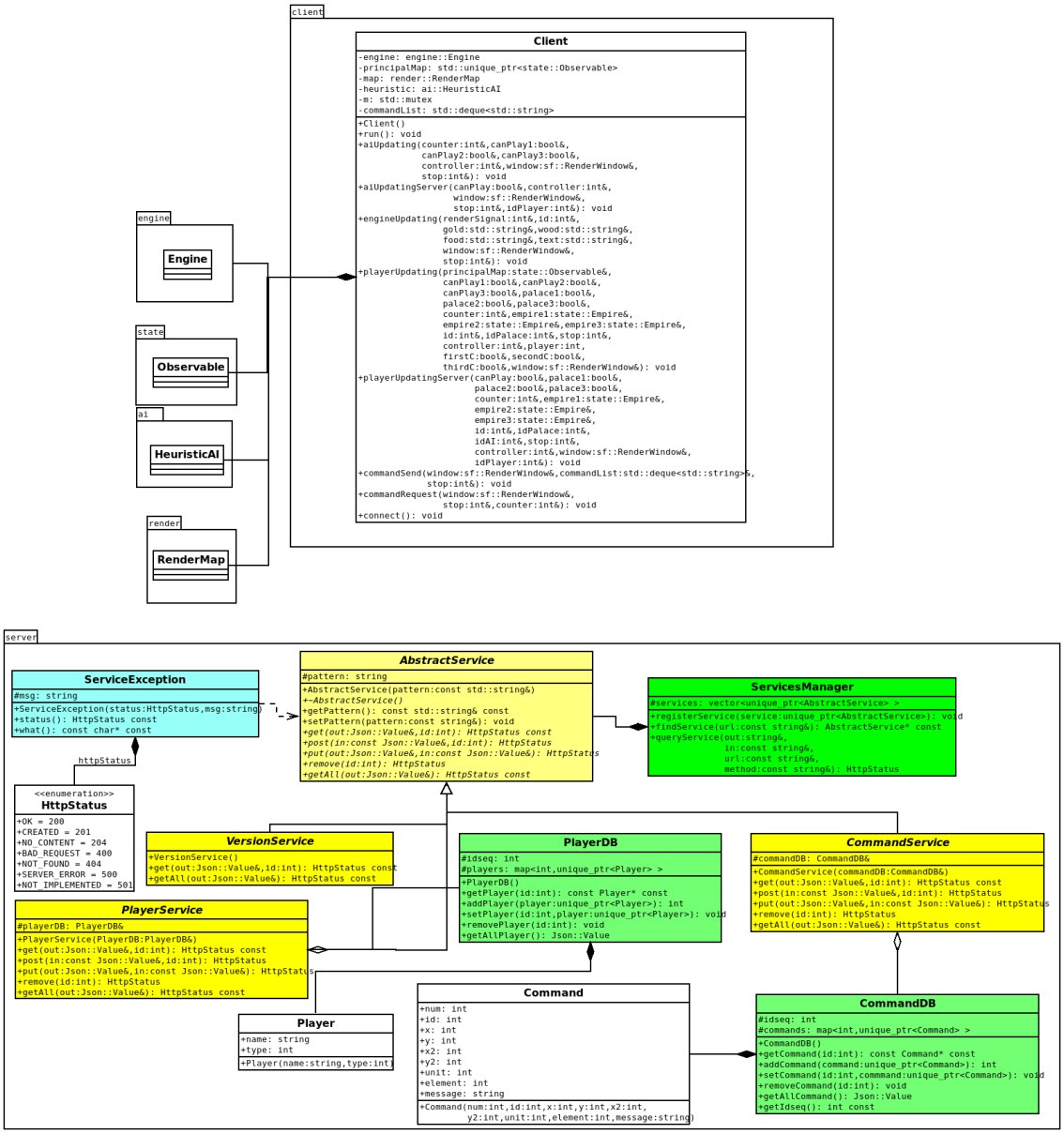

FIGURE 11 – Diagramme de classes du client

7 Conclusion générale du projet

Pour conclure, nous avons réussi à rendre chaque livrable terminé à l'heure. Pour chaque livrable nous avons globalement répondu à ce qui était attendu. Plus précisement, pour le livrable 4.final nous avons testé de mettre le jeu en ligne sur le serveur Kerosen.
Il y a encore beaucoup d'améliorations possibles pour le projet. Le temps étant très court entre chaque livrable nous avons du rester sur les bases de ce qui est possible pour chaque rendu. Nous pourrions en particulier optimiser et modulariser encore plus le code, améliorer énormément la partie graphique des gameplays et procurer un mode réseau au jeu plus solide pour pouvoir résister à la triche. Nous aurons, sans aucun doute progressé énormément en conception logicielle et en programmation.
Nous avons de plus pris du plaisir à réaliser ce projet et nous sommes fiers d'avoir pu arriver au bout.

8 Remerciements

Nous remercions David Picard, Christophe Barès, Luvizon Diogo ainsi que Veronica Belmega pour nous avoir aidés et guidés au cours du projet logiciel transversal.