

# **Projet Logiciel Transversal**

Valentin LAURENT – Vincent AYME

# Table des matières

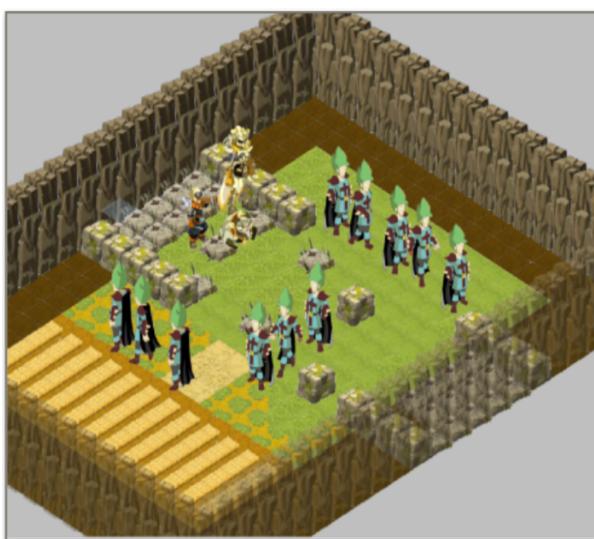
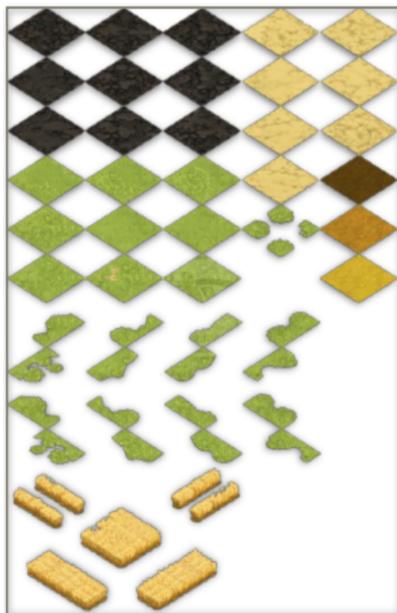
1 Objectif.....	3
1.1 Présentation générale.....	3
1.2 Règles du jeu.....	3
1.3 Conception Logiciel.....	3
2 Description et conception des états.....	4
2.1 Description des états.....	4
2.2 Conception logiciel.....	4
2.3 Conception logiciel : extension pour le rendu.....	4
2.4 Conception logiciel : extension pour le moteur de jeu.....	4
2.5 Ressources.....	4
3 Rendu : Stratégie et Conception.....	6
3.1 Stratégie de rendu d'un état.....	6
3.2 Conception logiciel.....	6
3.3 Conception logiciel : extension pour les animations.....	6
3.4 Ressources.....	6
3.5 Exemple de rendu.....	6
4 Règles de changement d'états et moteur de jeu.....	8
4.1 Horloge globale.....	8
4.2 Changements extérieurs.....	8
4.3 Changements autonomes.....	8
4.4 Conception logiciel.....	8
4.5 Conception logiciel : extension pour l'IA.....	8
4.6 Conception logiciel : extension pour la parallélisation.....	8
5 Intelligence Artificielle.....	10
5.1 Stratégies.....	10
5.1.1 Intelligence minimale.....	10
5.1.2 Intelligence basée sur des heuristiques.....	10
5.1.3 Intelligence basée sur les arbres de recherche.....	10
5.2 Conception logiciel.....	10
5.3 Conception logiciel : extension pour l'IA composée.....	10
5.4 Conception logiciel : extension pour IA avancée.....	10
5.5 Conception logiciel : extension pour la parallélisation.....	10
6 Modularisation.....	11
6.1 Organisation des modules.....	11
6.1.1 Répartition sur différents threads.....	11
6.1.2 Répartition sur différentes machines.....	11
6.2 Conception logiciel.....	11
6.3 Conception logiciel : extension réseau.....	11
6.4 Conception logiciel : client Android.....	11

# 1 Objectif

## 1.1 Présentation générale

Notre projet se base sur le système de combat tour par tour du jeu vidéo Dofus. Le joueur aura la possibilité de choisir entre plusieurs classes de personnages avec des aptitudes différentes telles que Iop, Sadida ou Sram. Le jeu débutera sur une carte proposant différents donjons. Chaque donjon entraînera une série de combats (le joueur versus une IA). Durant chaque affrontement, le joueur possèdera, à chaque tour, des points de déplacements ainsi que des points d'action lui permettant de se déplacer dans l'arène et d'invoquer ses différentes attaques.

Voici un aperçu des textures graphiques que nous allons utiliser:



## 1.2 Règles du jeu

- Le joueur, au début du jeu, choisit sa classe qui lui confèrera certains avantages en fonction des éléments de chaque donjon.
- Le joueur doit triompher de l'ensemble des donjons afin de terminer le jeu.
- Le joueur doit finir un donjon pour débloquer le suivant.
- A chaque arène terminée, le joueur recevra de l'expérience qui lui permettra de monter différents niveaux tout au long du jeu.
- Une fois qu'une arène est terminée, le joueur aura une probabilité de recevoir de l'équipement additionnel.
- Le joueur dispose à chaque tour de points de déplacement et des points d'action qui sont réinitialisés à chaque début de tour (sauf malus) lui permettant de se déplacer dans l'arène et d'utiliser des compétences.
- Si le joueur est vaincu lors d'une arène d'un donjon, celui-ci devra recommencer le donjon depuis la première arène pour le terminer.
- (Optionnel) Possibilité de terminer un donjon à plusieurs joueurs, en jouant en réseau.
- (Optionnel) Possibilité d'affronter un autre joueur en réseau.

## 1.3 Conception Logiciel

*Présenter ici les packages de votre solution, ainsi que leurs dépendances.*

## 2 Description et conception des états

*L'objectif de cette section est une description très fine des états dans le projet. Plusieurs niveaux de descriptions sont attendus. Le premier doit être général, afin que le lecteur puisse comprendre les éléments et principes en jeux. Le niveau suivant est celui de la conception logiciel. Pour ce faire, on présente à la fois un diagramme des classes, ainsi qu'un commentaire détaillé de ce diagramme. Indiquer l'utilisation de patron de conception sera très apprécié. Notez bien que les règles de changement d'état ne sont pas attendues dans cette section, même s'il n'est pas interdit d'illustrer de temps à autre des états par leur possibles changements.*

### 2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (arène) et un ensemble d'éléments mobiles (personnage joueur et adversaire). Tous les éléments possèdent les propriétés suivantes:

- Coordonnées (x,y) sur l'arène
- Statistiques (Liste[PV (Points de Vie), PA (Points d'Action), PM (Points de Mouvement)])
- Compétences (propres à la classe du personnage): ensemble des actions que peut faire
- l'élément
- Equipement: Ensemble des objets dont le personnage est équipé
- Identifiant de type d'élément (nombre indiquant la nature de l'élément: IA ou joueur)

#### 2.1.1 Etat éléments fixes

Le labyrinthe est formé par une grille d'éléments nommé « cases ». La taille de cette grille est fixée au démarrage du niveau. Les types de cases sont :

**Cases “Obstacle”** Les cases “obstacle” sont des éléments infranchissables pour les éléments mobiles, elles correspondent à la fois aux frontières d'une arène mais également à certaines cases au sein de celle ci. Le choix de la texture aura une influence sur l'évolution du jeu car certaines peuvent être traversées par des compétences lors d'un combat. On considère les types de cases “obstacle“ suivants:

-Les obstacles “mur“, qui définissent les contours d'une arène ou un obstacle qui ne peut être traversés par une attaque.

-Les obstacles “décors“, qui peuvent être traversés par une attaque d'un joueur. **Cases “Espace”** Les cases “espace“ sont les éléments franchissables par les éléments mobiles.

On considère les types de cases “espace“ suivants: -Les espaces “vides“

-Les espaces “départ“, qui définissent les positions initiales pour le joueur et l'adversaire. -Les espaces “piège“, qui déclenchent une action lorsqu'un personnage marche dessus.

### 2.1.2 Etat éléments mobiles

Les éléments possèdent une direction (gauche, droite, haut ou bas) et une position.

**Element mobile “Personnage”** Cet élément est soit dirigé par le joueur soit par l’IA qui, en fonction du nombre de points de déplacement, commande la propriété de direction. On utilise une propriété que l’on nommera “statut”, et qui peut prendre les valeurs suivantes:

- - Statut “normal”: cas le plus courant, le personnage peut se déplacer lors de son tour de jeu.
- - Statut “empoisonné”: cas où le personnage a subi une certaine attaque. Ses points de vie diminueront légèrement à chaque début de tour pendant un nombre de tours définis avant de repasser au statut “normal”.
- - Statut “mort”: Les points de vie du personnage sont inférieurs ou égale à 0, le personnage ne peut plus bouger ni réaliser aucune action.

### 2.1.3 Etat général

A l’ensemble des éléments statiques et mobiles, nous rajoutons les propriétés suivantes:

-**Tour** : Permet à un personnage d’effectuer ses actions et déplacements. -**Compteur d’ennemis en vie** : le nombre d’ennemis qu’il reste à vaincre avant de terminer une arène.  
-**Timer (Optionnel)**: Délimite le temps passé pour un tour de jeu lorsque deux joueurs humains s’affrontent.

## 2.2 Conception logiciel

Le diagramme des classes pour les états est présenté sur la figure suivante, nous pouvons mettre en évidence les groupes de classes suivants:

**Classe Elements** Toute la hiérarchie des classes filles de “Element” (en jaune), permet de représenter aussi bien les personnages (joueur ou IA) que les éléments présents sur une map, tels que les cases où un personnage peut se déplacer, où il commence le combat ou les cases occupées par un décors.

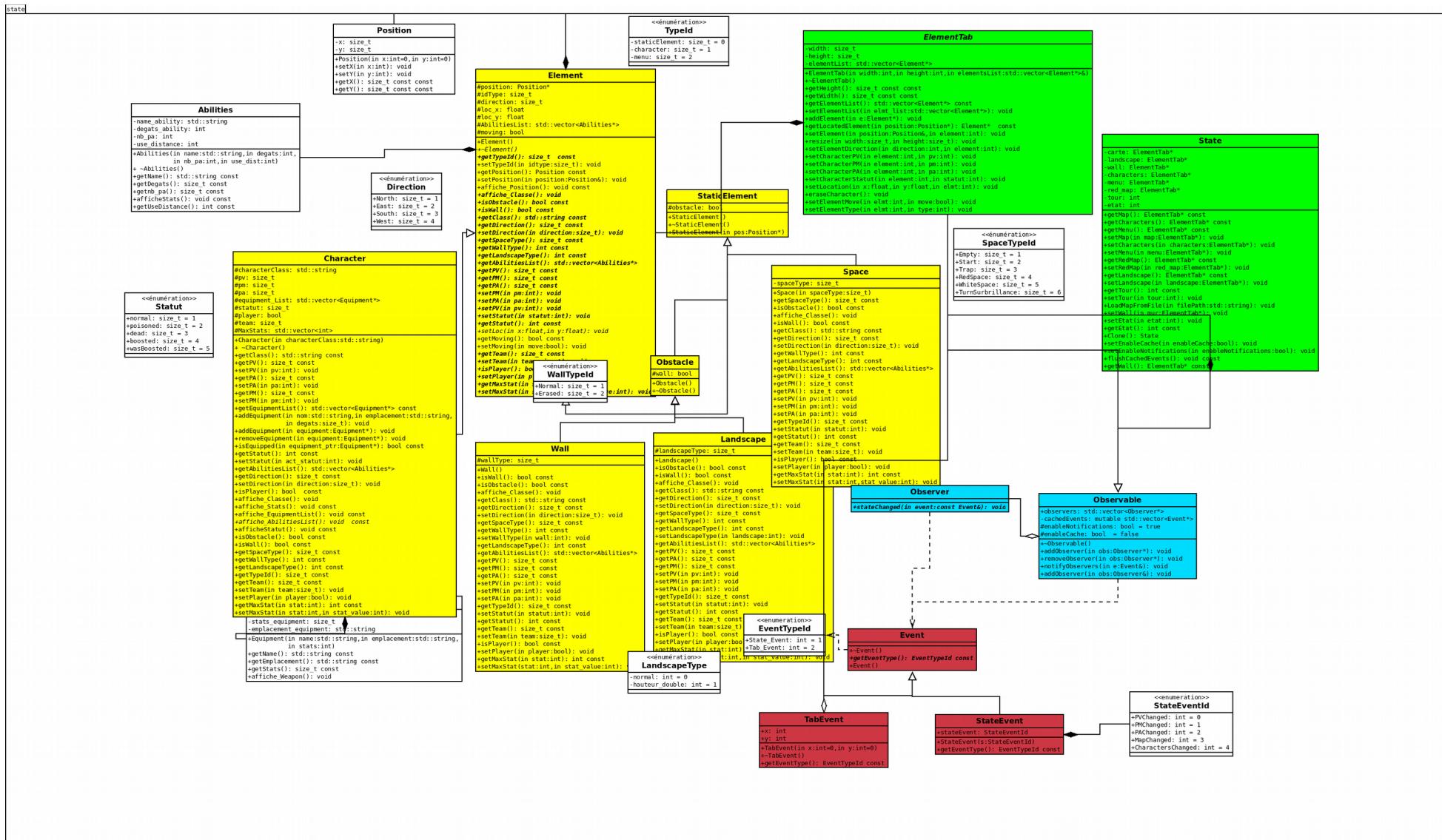
**Conteneurs d’Elements** On peut s’apercevoir que les classes “State“ ainsi que “ElementTab“ viennent contenir l’ensemble des éléments sur une carte donnée. Une instance du type “ElementTab“ va donc être une liste de l’ensemble des objets présents sur la map tels que les éléments de décors et les personnages.

## **2.3 Conception logiciel : extension pour le rendu**

## **2.4 Conception logiciel : extension pour le moteur de jeu**

## **2.5 Ressources**

Illustration 1: Diagramme des classes d'état



## 3 Rendu : Stratégie et Conception

Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémentent pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

### 3.1 Stratégie de rendu d'un état

Pour le rendu nous avons décidé de le faire sous forme de couches, en fonction de l'importance des éléments. Nous trouverons donc en couche basse les éléments de type Space, puis la couche des éléments de type Obstacle, la couche Character avec les différents personnages en jeu et enfin la couche qui affiche l'état du Character (points de vie restants, points d'action restants, points de mouvement restants). Chaque couche sera composée de deux éléments bas-niveau : un vecteur d'éléments de type Element et une texture.

Lorsqu'un changement a lieu (changement de position d'un personnage), l'état change et le rendu affiche le nouvel état. Il est important que le rendu se rafraîchisse plus rapidement que ne change l'état pour être certain d'afficher tous les états.

### 3.2 Conception logiciel

Le diagramme des classes pour les états est présenté sur la figure suivante.

**Couches.** Le cœur du rendu réside dans le groupe (en jaune) autour de la classe Layer. 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 définition des tuiles est contenu dans une instance de TileSet. Les classes filles de la classe Layer se spécialisent pour l'un des plans à afficher. Par exemple, la classe StateLayer va afficher le nombre de points de vie, points d'action et points de mouvement restants, et la classe ElementTabLayer peut afficher le niveau ou les personnages. La méthode initSurface() fabrique une nouvelle surface, lui demande de charger la texture, puis initialise la liste des sprites. Par exemple, pour afficher le niveau, elle demande un nombre de quads/sprites égal aux nombre de cellules dans la grille avec initQuads(). Puis, pour chaque cellule du niveau, elle fixe leur position avec setSpriteLocation() et leur tuile avec setSpriteTexture().

**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.

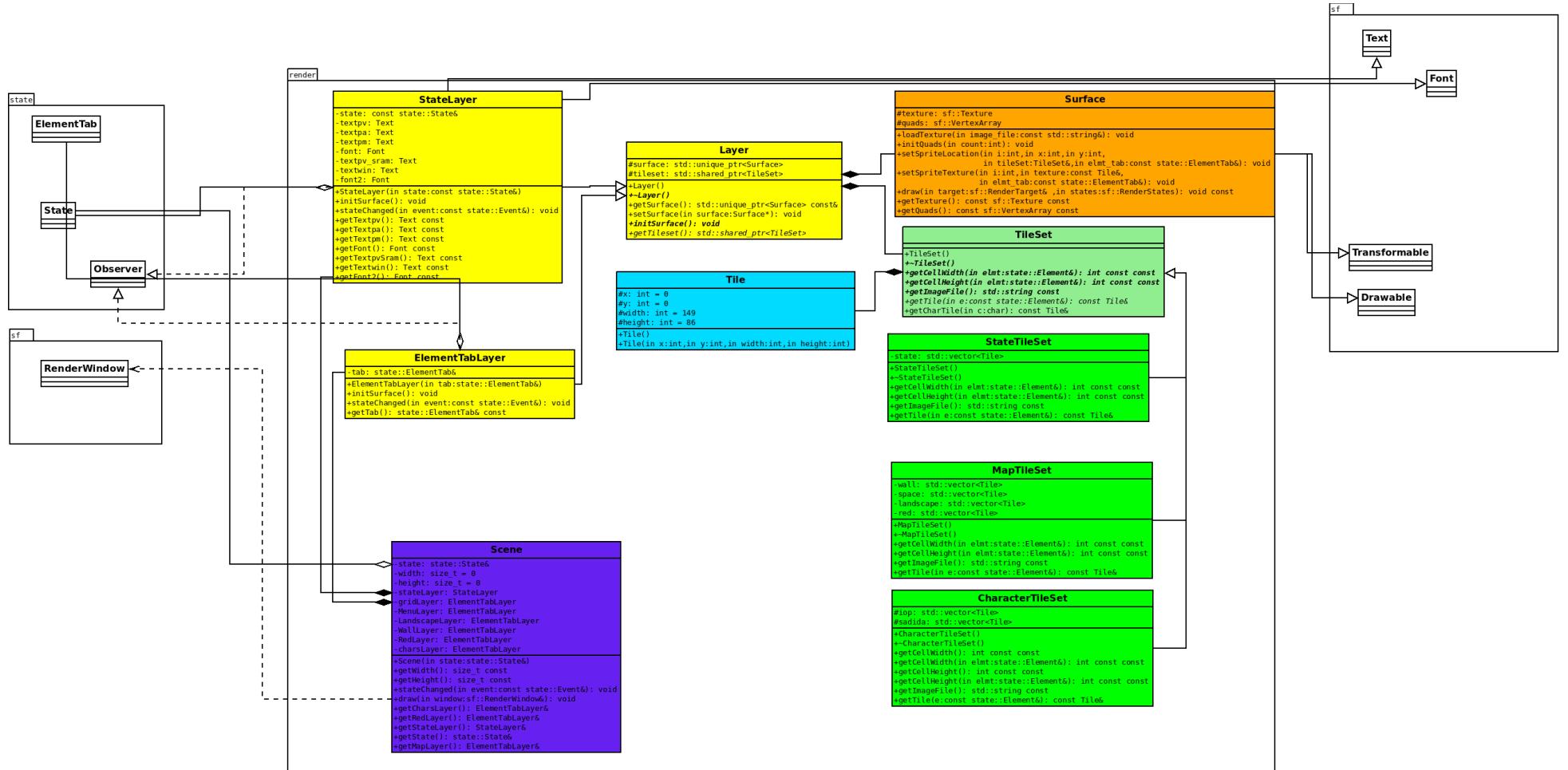
**Tuiles.** Les classes filles de TileSet regroupent toutes les définitions des tuiles d'un même plan. Par exemple, elle sait que les coordonnées de la tuile avec un sol volcanique est (0,0) et de taille (149,86). Pour obtenir une telle information, un client de ces classes utilise la méthode getTile(). Par exemple si on passe à cette méthode une instance de la classe Space, alors elle va renvoyer une instance de Tile qui correspond à cet élément.

### **3.3 Conception logiciel : extension pour les animations**

### **3.4 Ressources**

### **3.5 Exemple de rendu**

Illustration 2: Diagramme de classes pour le rendu



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

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects liés au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

### 4.1 Horloge globale

Les changements d'états sont effectués dès lors qu'une commande est activée. En effet lorsqu'on effectue une commande (déplacement, attaque) nos observables notifient nos observateurs qui réinitialisent leur surface, ce qui permet d'avoir au prochain affichage la version actuelle de l'état du jeu.

### 4.2 Changements extérieurs

Les changements extérieurs sont provoqués par la position et les cliques d'une souris permettant :

- D'avoir une surbrillance sur la case de la carte indiquée par la souris.
- D'avoir une surbrillance de la case de la barre menu en bas de l'écran si la souris la pointe.
- De déplacer notre personnage si on clique sur une case de la carte.
- D'attaquer un autre personnage si on clique sur celui-ci.

### 4.3 Changements autonomes

Les changements autonomes sont appliqués à chaque création ou mise à jour d'un état, après les changements extérieurs.

- Si on déplace son personnage, les points de déplacement vont diminuer en fonction du déplacement effectué.
- Si notre personnage attaque les points d'action diminuent.
- Si le personnage subit une attaque ses points de vie diminuent du montant des dégâts de l'attaque.
- Si un personnage n'a plus de points de vie, il n'est plus représenté sur la carte.

### 4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté sur la figure suivante. L'ensemble du moteur de jeu repose sur un patron de conception de type Command, et a pour but la mise en œuvre 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 (automatique, clavier, réseau, ...). Notons bien que ces classes ne gèrent absolument pas l'origine des commandes, ce sont d'autres éléments en dehors du moteur de jeu qui fabriqueront les instances de ces classes.

A ces classes, on a défini un type de commande avec CommandTypeId pour identifier précisément la classe d'une instance.

- **LoadCommand :** Charge un niveau depuis un fichier
- **MoveCharacterCommand :** Déplace un personnage à la position donnée en fonction de ses PA .
- **HandleCollisionsCommand :** Regarde si un obstacle ou un personnage n'est pas déjà sur la

case où l'on veut se déplacer.

- **AttackCommand** : Attaque un personnage.
- **SurbrillanceCommand** : Met en surbrillance la case pointée par la souris.
- **WhiteSurbrillanceCommand** : Permet de mettre en surbrillance l'attaque que l'on souhaite effectuer.

**Engine** : C'est le cœur du moteur. 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 lorsqu'on a appelé la méthode update() après un temps suffisant, le moteur appelle la méthode execute() de chaque commande, puis supprime toutes les commandes.

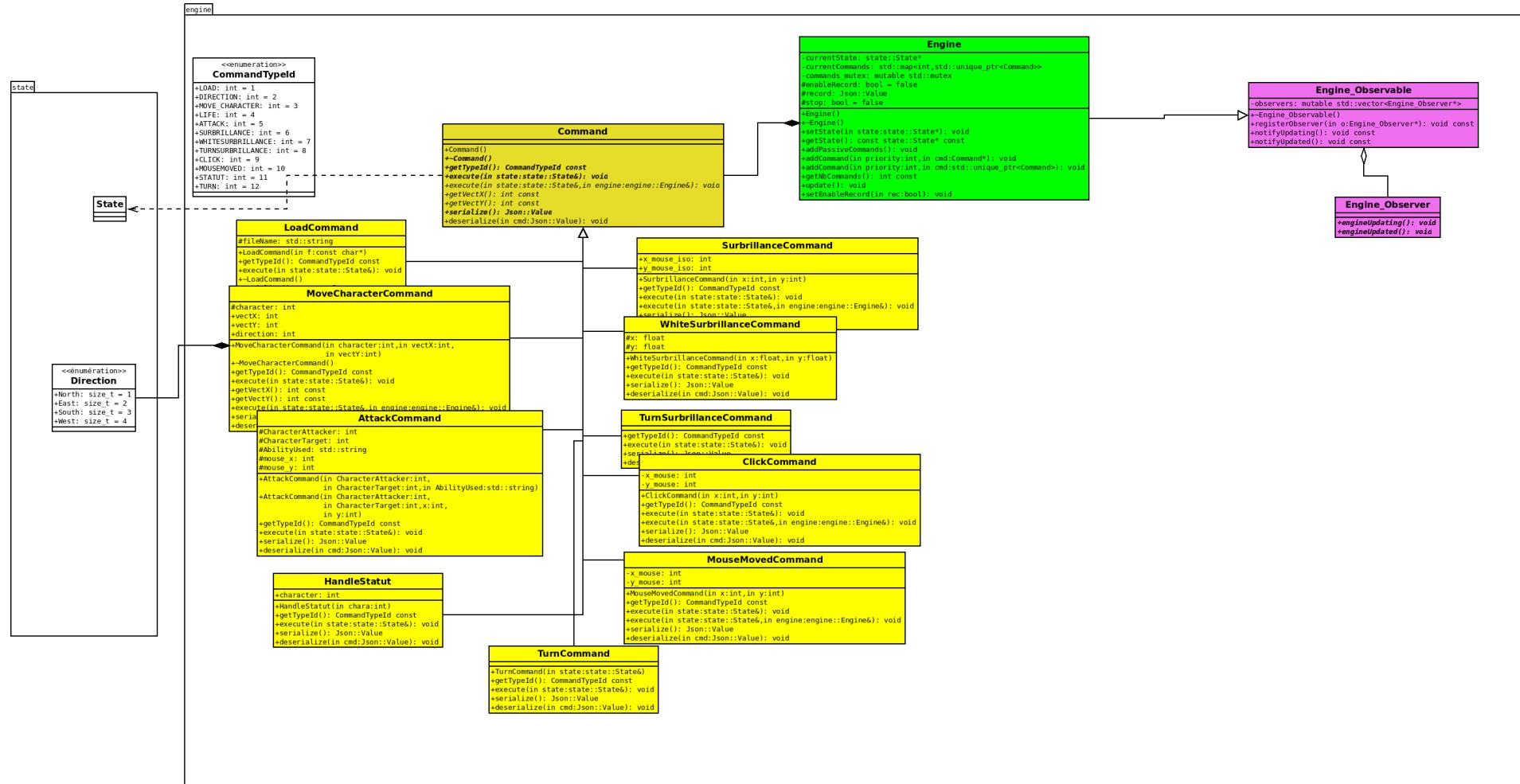
Le temps suffisant a été fixé, pour le moment à 10Hz.

On a ajouté à chacune de ces commandes une méthode serialize et deserialize qui permet d'enregistrer la commande en Json dans un fichier Json pour la première et de faire la traduction inverse pour la deuxième. Cela nous permet d'enregistrer les coups de la partie et de la rejouer.

## 4.5 Conception logiciel : extension pour l'IA

## 4.6 Conception logiciel : extension pour la parallélisation

Illustration 3: Diagrammes des classes pour le moteur de jeu



# 5 Intelligence Artificielle

*Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.*

## 5.1 Stratégies

### 5.1.1 Intelligence minimale

### 5.1.2 Intelligence basée sur des heuristiques

### 5.1.3 Intelligence basée sur les arbres de recherche

#### 5.1.1 Intelligence minimale

Cette stratégie est purement aléatoire. A chaque tour, le personnage va choisir aléatoirement un déplacement à effectuer, puis va vérifier si il peut attaquer l'adversaire. Dans le cas où il est à une distance suffisante pour attaquer, il choisira d'effectuer une attaque aléatoire parmi les différentes attaques qu'il peut effectuer.

#### 5.1.2 Intelligence basée sur les heuristiques

Nous améliorons l'IA afin qu'elle n'effectue plus des actions aléatoires. Notre IA heuristique est agressive, c'est à dire qu'elle va toujours vouloir effectuer son attaque qui fait le plus de dégâts à l'ennemi. Si cela lui est impossible, elle se rapproche au maximum possible de sa cible, et effectue l'attaque avec le plus de dégâts possibles depuis cette distance. Lorsque l'IA peut effectuer sa meilleure attaque sans se déplacer elle ne se déplace pas.

### 5.1.3 Intelligence basée sur les arbres de recherche

## 5.2 Conception logiciel

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

**Classes AI :** Les classes filles de la classe AI implante différentes stratégies d'IA, que l'on peut appliquer pour un personnage :

- RandomAI : Intelligence aléatoire
- HeuristicAI : intelligence heuristique
- DeepAI : Intelligence avancée

**PathMap.** La classe PathMap permet de calculer la distance de l'ennemi le plus proche. Nous avons utilisé la classe Point qui correspond à une case avec un poids. Le poids est égal à la distance entre la case et l'ennemi le plus proche du personnage géré par l'IA. On rentre tous les points où le personnage peut se rendre (en fonction de ses PM) dans une file prioritaire avec en tête le point avec le poids le plus faible.

**DeepAI.** Cette IA se base sur la méthode MiniMax. L'IA se projette dans un état futur évalue la situation revient à l'état présent et réitère l'opération pour tous ses choix de déplacement possibles lors de son tour. L'IA prend alors le score maximal de l'évaluation comme choix de tour. En profondeur 2, on fait l'étape de la profondeur 1 mais pour chaque état futur, on évalue les déplacements possibles de l'adversaire suite à notre déplacement fictif. Comme c'est le tour du joueur adverse, le score pris est le score minimum à cette étape et on prend ensuite le score maximum des scores minimum obtenus. Pour l'évaluation, le joueur prend en compte sa meilleure attaque depuis sa position et celle de l'ennemi. Il prend aussi la distance qui le sépare du joueur ennemi, ses pv et ceux de l'ennemi.



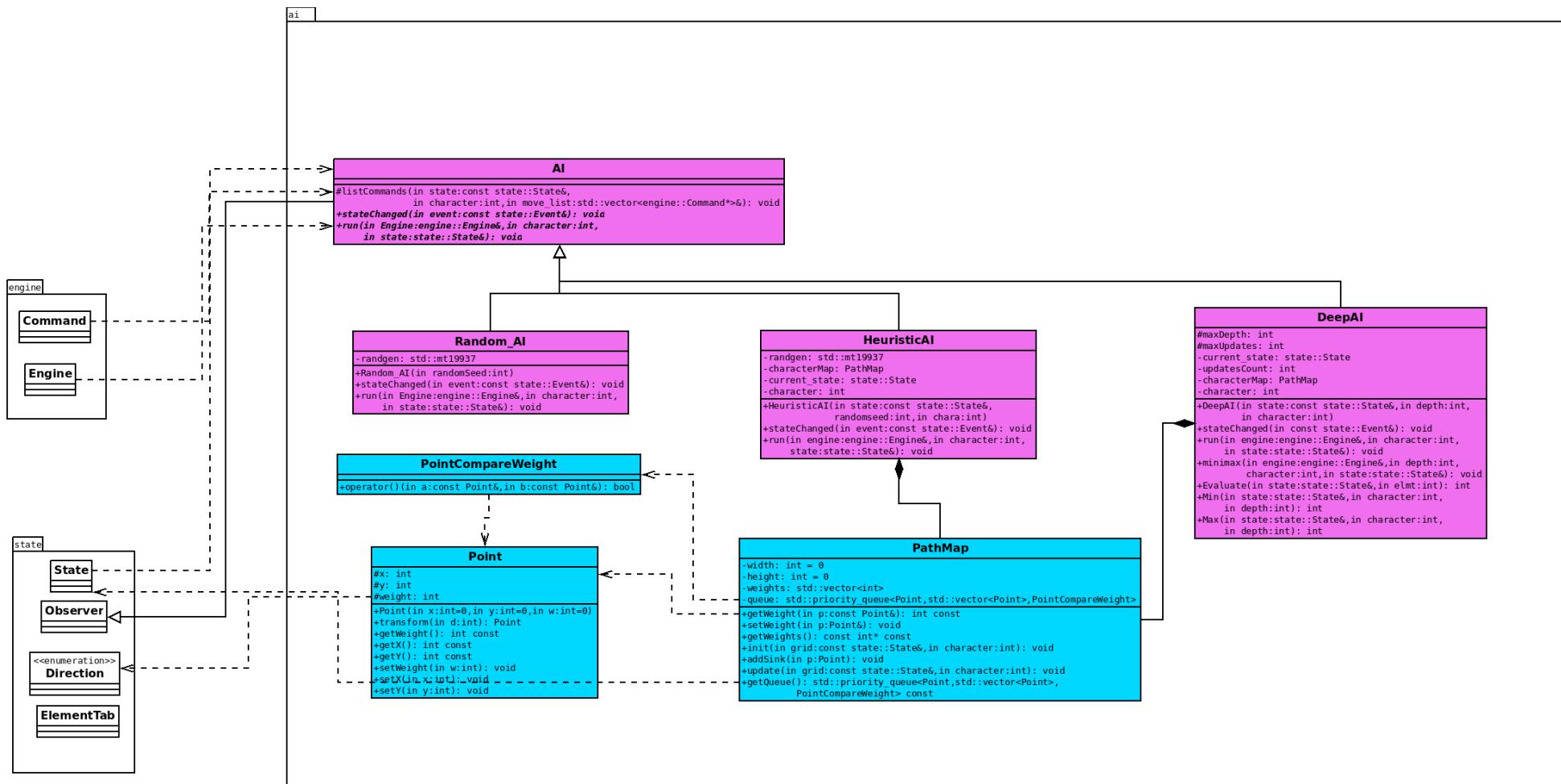


FIGURE 8 – Diagramme des classes d'intelligence artificielle.



### **5.3 Conception logiciel : extension pour l'IA composée**

### **5.4 Conception logiciel : extension pour IA avancée**

### **5.5 Conception logiciel : extension pour la parallélisation**

# 6 Modularisation

Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est un parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.

## 6.1 Organisation des modules

### 6.1.1 Répartition sur différents threads

Nous allons à présent utiliser différents threads pour l'exécution de notre programme. Dans notre cas nous avons 4 threads, un pour le moteur, un autre pour le rendu et un pour chacune de nos deux IA. L'exécution des IA est régulée grâce à une variable qui correspond au numéro du tour ce qui fait qu'elles ne pourront pas jouer en même temps. On utilise alors un mutex qui permet l'accès aux ressources critiques de chacun des threads : dans le thread moteur il s'agit de l'exécution des différentes commandes tandis que dans le thread rendu il s'agit du rafraîchissement de l'affichage écran. De cette façon, le rendu devra attendre que le moteur ait fini d'exécuter les commandes d'un tour avant de rafraîchir l'état du jeu et le moteur devra attendre la fin du rafraîchissement avant d'exécuter les commandes d'un nouveau tour. Les deux autres threads dédiés aux intelligences artificielles, fonctionnent exactement de la même manière.

Vous trouverez le fichier client.dia correspondant au client. Nous avons seulement fait sa description schématique, mais celui-ci n'est actuellement pas utilisé au sein de notre projet, nous l'utiliserons lorsque nous travaillerons sur la partie réseau de ce dernier.

### 6.1.2 Répartition sur différentes machines

Le but de notre mise en réseau du jeu est de pouvoir jouer depuis différentes machines à la même partie en incarnant donc deux personnages présents dans la partie mais il ne faut pas que ce soit le même.

Les requêtes GET : Pour l'instant nous avons limiter les id à 1 et 2. Il faut donc un de ces deux nombres pour que la requête fonctionne.

Les requêtes PUT : Dans la requête PUT on vérifie que la donnée comporte au moins le champ « name ». Dans le cas contraire il n'y a pas d'ajout de joueurs. Si le champ « free » n'est pas présent, on le met à true de base. La liste des joueurs s'affiche ensuite.

Les requêtes POST : On vérifie qu'il y a au moins un des champs de *Player* présent dans la donnée et si c'est le cas on la/les modifie. Sinon on renvoie une exception « Bad POST Request ! ».

Les requêtes DELETE : On supprime un joueur correspondant à un id si celui-ci est valide. Sinon on revoie une exception « Bad Player ID ! ». La liste des joueurs, mise à jour, s'affiche ensuite.

## 6.2 Conception logiciel

**Client.** La classe Client contient toutes les informations pour faire fonctionner le jeu : Moteur de jeu (avec état intégré), intelligences artificielles, et rendu. N'ayant pas encore réalisé le travail pour le rendu 4.final, nous n'avons pas encore utilisé au sein de notre code cette classe.

**Services :** La classe *ServiceManager* contient la liste des différents service. Il permet de déterminer depuis une requête quel service est utilisé et quel type de requête est appelé.

On a pour l'instant 2 différents service : *VersionService* et *PlayerService*.

**Game :** La classe *Game* contient la liste des joueurs, que l'on peut modifier.

## 6.3 Conception logiciel : extension réseau

## 6.4 Conception logiciel : client Android

Illustration 4: Diagramme de classes pour la modularisation

