

# **Projet Logiciel Transversal**

Steven SENG – Francis KINAVUIDI

## Table des matières

1 Objectif	3
1.1 Présentation générale	3
1.2 Règles du jeu	3
1.3 Ressources	3
2 Description et conception des états	6
2.1 Description des états	6
2.2 Conception logiciel	6
2.3 Conception logiciel : extension pour le rendu	6
2.4 Conception logiciel : extension pour le moteur de jeu	6
2.5 Ressources	6
3 Rendu : Stratégie et Conception	8
3.1 Stratégie de rendu d'un état	8
3.2 Conception logiciel	8
3.3 Conception logiciel : extension pour les animations	8
3.4 Ressources	8
3.5 Exemple de rendu	8
4 Règles de changement d'états et moteur de jeu	10
4.1 Règles de changements d'état	10
4.2 Changements extérieurs	10
4.3 Changements autonomes	10
4.4 Conception logiciel	10
4.5 Sauvegarde des commandes	12
4.6 Conception logiciel : extension pour la parallélisation	10
5 Intelligence Artificielle	12
5.1 Stratégies	13
5.1.1 Intelligence minimale	13
5.1.2 Intelligence basée sur des heuristiques	13
5.1.3 Intelligence basée sur les arbres de recherche	13
5.2 Conception logiciel	13
5.3 Conception logiciel : extension pour l'IA composée	13
5.4 Conception logiciel : extension pour IA avancée	13
5.5 Conception logiciel : extension pour la parallélisation	13
6 Modularisation	13
6.1 Organisation des modules	14
6.1.1 Répartition sur différents threads	14
6.1.2 Répartition sur différentes machines	14
6.2 Conception logiciel	14
6.3 Conception logiciel : extension réseau	14
6.4 Conception logiciel : client Android	14

# 1 Objectif

## 1.1 Présentation générale

Ce jeu base ses mécaniques sur des jeux comme Fire emblem, un tactical RPG, et Dofus, un MMORPG. C'est un tactical RPG au tour par tour où l'on contrôle un personnage.

## 1.2 Règles du jeu

En début de partie, le joueur sélectionne une classe de personnage, possédant des caractéristiques propres (point de déplacement de base, type de déplacement, point de vie de base, attaque de base, initiative, etc.), et une liste 4 attaques dépendant de la classe choisie. Chaque attaque a des caractéristiques de base qui peuvent être modifiées par l'état du joueur ou le terrain : dégât, porté, précision et effet (sur le terrain, le lanceur ou la victime). Une fois le choix fait, le joueur est placé sur un terrain et a pour objectif d'éliminer son/ses adversaire(s).

La carte du jeu, ou le terrain, est une grille constituée de nombreuses cases, chacune possédant un type prédéfini (forêt, ville, plaine, etc.) et un état (normal, neige, brume, etc.). Ces cases influencent les statistiques, états et attaques des personnages.

Un tour de jeu se déroule donc de la manière suivante : en fonction de leur statistique d'initiative, chaque personnage encore en vie joue un par un leur tour de joueur. Une fois qu'ils ont tous fini, les événements aléatoires du terrain s'activent.

Un tour de joueur se divise en 2 parties, la partie active et l'application des effets. Lors de la partie active, le joueur peut se déplacer autant de fois qu'il le veut tant qu'il a encore des points de déplacement mais ne peut utiliser que 2 attaques.

Pour se déplacer vers une case adjacente, le personnage doit dépenser ses points de déplacement en fonction du type de case qu'il traverse et de son type de déplacement.

Pour attaquer, le joueur doit sélectionner une de ces 4 attaques puis la case sur laquelle il va l'exécuter (toutes les attaques ne nécessitent pas la présence d'un ennemi pour être lancées). L'attaque réussie si le test de réussite d'attaque est positif, son pourcentage de réussite dépend de la précision de l'attaque, du terrain ciblé et de l'état du lanceur. Les dégâts infligés à l'adversaire sont proportionnels à l'attaque du joueur et aux dégâts de l'attaque. Une fois lancée, une attaque n'est plus utilisable pour un nombre de tour égal à sa valeur dite de recharge.

Cette phase se finit si le joueur n'a plus d'action à effectuer ou s'il le décide. L'application des effets se fait automatiquement après la partie active. Les points de mouvement du personnage se régénèrent et les effets affectant ses points de vie et les changements d'état aléatoires du terrain s'appliquent.

Lorsque les points de vie d'un personnage tombent à 0 suite à une attaque ou à un effet, il meurt et disparaît du terrain.

## 1.3 Ressources

Chaque personnage possède un ensemble de sprites représentant celui-ci dans différentes positions : immobile, se déplaçant, attaquant et lançant un sort, et ce dans les quatre directions. Les mouvements de déplacement, d'attaque et de sortilège se décomposent chacun en une animation de 3 sprites.



Nous avons ensuite divers affichages représentant les différents types de terrains ainsi que les effets qui s'y produisent.



## 2 Description et conception des états

### 2.1 Description des états

#### 2.1.1 Etat du Terrain

Le terrain est une grille d'éléments fixes nommées cases. Ses dimensions et son contenu dépendent de la carte choisie. Les cases ont 3 propriétés propres : leur position, leur type et leur statut. Elles influencent les actions faites durant la partie. Le terrain est ici

décrit par un tableau composé d'éléments qui décrivent le type et les statuts de chaque case.

Les types de cases permettent de définir le sprite utilisé pour les cases, qui sont donc discernables graphiquement par le joueur. Elles peuvent avoir des effets spécifiques sur les statistiques des joueurs et des attaques.

Ces types sont :

- Les plaines « plain » ;
- Les routes « road » ;
- La forêt « forest » ;
- La montagne « mountain » ;
- Le désert « sand » ;
- Les marécages « swamp » ;
- La ville « city » ;
- L'eau « water » : infranchissable par les personnages ;
- Les murs « wall » : infranchissable par les personnages et bloque les attaques à distance ;

Les statuts des cases sont décrits sous forme d'un tableau de 2 colonnes, l'une contenant l'Id du statut, et l'autre la durée respective du statut (en tour de joueur). Ces différents statuts sont séparables en 2 catégories, les cumulables et les statuts météo. Une case ne peut avoir qu'un statut météo. Les différents statuts sont :

- La pluie « rain », statut météo ;
- La neige « snow », statut météo, probabilité de diminuer les PM du joueur présent sur la case ;
- La brume « mist », statut météo, diminue la précision des attaques ;
- Le statut infranchissable « blockmove » ;
- Le statut blocage d'attaque « blockattack » ;
- Le statut incendie « burning », probabilité de brûler le joueur présent sur la case ;
- Le statut toxic « poison », probabilité d'empoisonner le joueur présent sur la case ;
- Le statut gravité « gravity » ;

### 2.1.2 Etat des Éléments mobiles

Les éléments mobiles ont comme propriété commune leur position sur la grille du terrain. Il en existe deux types: les "joueurs", les personnages sur le terrain associé à un utilisateur (humain ou IA) et le "curseur", l'élément directement contrôlé par l'utilisateur humain pour jouer.

#### ► Joueur

Les joueurs sont les éléments principaux du jeu, ce sont les personnages contrôlés par les utilisateurs. Ils possèdent une position sur la grille, une direction (haut, bas, gauche, droite), une classe, des statistiques, un statut et une liste d'attaque.

La classe et la liste des attaques du personnage sont choisies par l'utilisateur avant la partie. Le choix de la classe définit les statistiques de base du personnage, qui sont fixes contrairement aux statistiques du joueur qui sont amenées à évoluer au cours de la partie. Ce choix permet également de définir dans le code la liste des attaques disponibles pour le personnage.

### ○ Statut du joueur

Le statut d'un joueur représente une liste de statut dans lequel il peut être. Il est défini dans un tableau de 2 colonnes : l'id du statut et la durée du statut (en tour de joueur). Le statut "mort" n'en fait pas partie, il est déterminé simplement par un test du nombre de points de vie restants.

Ces statuts sont :

- Le statut empoisonné "poison", le joueur perd 5% de ces PV et son initiative baisse;
- Le statut brûlé "burning", le joueur perd 5% de ces PV;
- Le statut ébloui "dazzled", la précision des attaques du joueur baisse de ;
- Le statut apeuré "fear", le joueur ne joue pas et se déplace aléatoirement;

### ○ Classe de personnage

La classe de personnage du joueur définit les sprites du joueur ainsi que les statistiques de base du personnage :

- Le nombre de points de vie (PV);
- Le nombre de points de mouvement (PM);
- L'initiative, qui définit l'ordre de jeu des joueurs dans un tour;
- L'attaque;
- Le nombre d'attaques par tour (pour l'instant égal à 2 pour toutes les classes);
- Le type de déplacement, qui définit le nombre de PM consommé par le joueur pour chaque type de case:
  - normal : malus forêt & desert, fort malus marécage & montagne;
  - agile : malus montagne, fort malus marécage & desert;
  - nature : marécage, fort malus desert & ville
  - monture : malus montagne & forêt, fort malus marécage

On a donc les classes suivantes :

Nom	type de dpl	PM	PV	attaque	initiative
Knight	monture	12	elevé	moyen	elevé

Frog	nature	6	moyen	elevé	elevé
Archer	agile	10	faible	moyen	moyen
Dwarf	normal	6	moyen	moyen	faible

Il s'agit des différentes statistiques qu'aura le joueur.

### ○ Classe d'attaque

Les attaques sont des actions effectuées par un joueur affectant les autres joueurs, le terrain ou lui-même. Elles sont définies par différentes propriétés :

- Les dégâts « damage », le nombre de points de vie de base enlevé à la victime ;
- La précision « precision », le pourcentage de chance que l'attaque réussisse ;
- La portée « range », définissant la portée minimale et maximale de l'attaque ;
- La forme de la zone d'action « area » ;
- Le type de l'attaque « special», si l'attaque est une attaque magique/spécial ou physique ;
- Le nombre de tours de charge « cooldown », le nombre de tours pendant lequel l'attaque ne peut pas être utilisé après avoir été utilisée, il est négatif pour les attaques utilisable plusieurs fois en 1 tour;
- Les effets “effect”, une liste de trois éléments: l'id de l'effet, le nombre de tour de jeu “n” où l'effet est appliqué et la probabilité que l'effet se fasse.

Ces effets sont:

- “poison\_char” : change le compteur du statut POISON de la cible à n tours;
- “burn\_char” : change le compteur du statut BURNING de la cible à n tours;
- “burn\_field” : change le compteur du statut BURNING du terrain à n tours;
- “dazzle” : change le compteur du statut DAZZLED de la cible à n tours;
- “fear” : change le compteur du statut FEAR de la cible à n tour;
- “heal\_low” : soigne 5% des PV de la cible;
- “heal\_medium” : soigne 10% des PV de la cible;
- “heal\_high” : soigne 20% des PV de la cible;
- “move\_user” : déplace l'utilisateur vers la position ciblé par l'attaque;
- “move\_foe” : déplace la victime d'une case aléatoirement;
- “boost\_att\_rain” : augmente les dégâts de l'attaque si une case entre la cible et le lanceur à un statut RAIN;
- “heal\_state” : change le compteur des statuts négatif à 0.

### ► Curseur

Le curseur est un élément mobile présent sur la carte lors du tour du joueur. Il lui permet d'effectuer des actions avec son personnage et obtenir des informations sur l'état du jeu.

### 2.1.3 Etat du jeu

En plus des éléments présentés précédemment, l'état du jeu est également décrit par le nombre de tours, la liste de l'ordre de jeu, dépendant de l'initiative des joueurs encore en vie, et l'id du joueur en train de jouer.

## 2.2 Conception logiciel

Le diagramme des classes pour les états est présenté dans la figure suivante, on y retrouve les classes suivantes:

**Classe State:** Classe qui permet d'accéder à toutes les données qui forment un état du jeu. Elle contient un tableau d'éléments de 2 dimensions contenant des éléments de classe Field, une liste des joueurs en jeu, le joueur en train de jouer, et le nombre de tours.

**Classe Field:** Classe qui décrit le type et statut d'une case.

**Classe Player:** Classe qui décrit l'état du personnage d'un joueur dans le jeu.

**Classe Skill:** Classe qui décrit les statistiques et effets d'une attaque.

**Classe Curseur:** Classe qui décrit la position du curseur.

**Classe Character:** Classe qui regroupe les statistiques de base du personnage d'un joueur.

**Fabrique d'élément CharacterFactory:** Fabrique un personnage prédéfini à partir de son id.

**Fabrique d'élément SkillFactory:** Fabrique une attaque prédéfini à partir de son id.

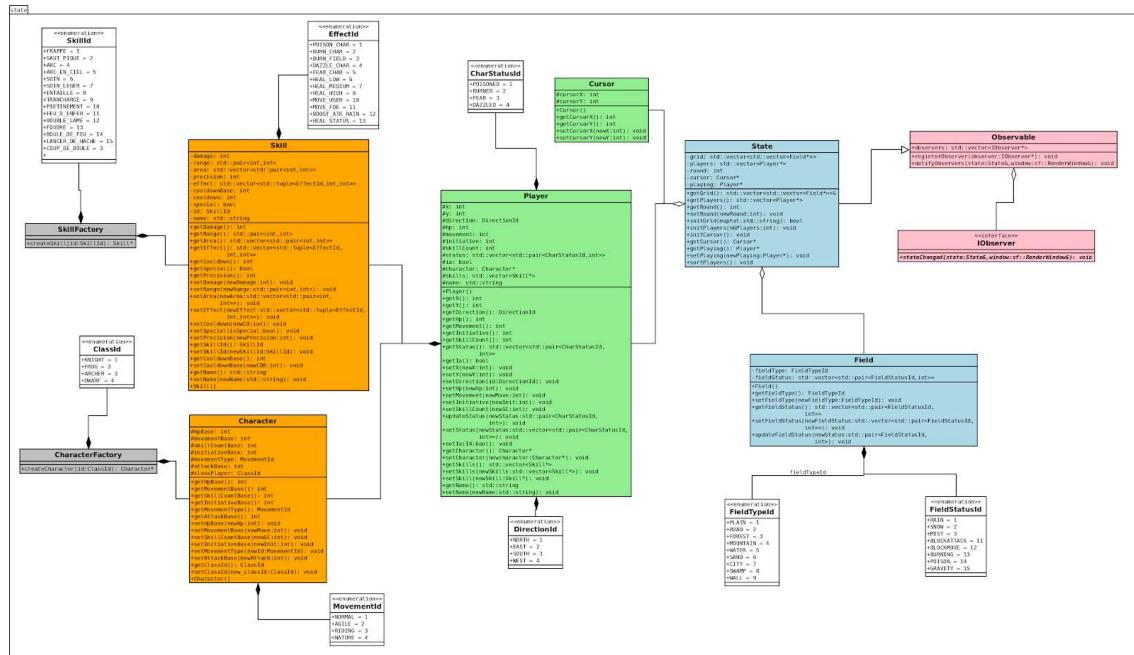


Illustration 1: Diagramme des classes d'état

### 3 Rendu : Stratégie et Conception

#### 3.1 Stratégie de rendu d'un état

Nous avons utilisé pour notre rendu d'état un rendu par tuile avec la bibliothèque SFML. Notre rendu se divise en plusieurs couches, une par type d'éléments: le terrain, les personnages, le curseur, le statut du terrain et le statut des joueurs.

Le terrain est créé à partir d'un fichier texte. Il contient les IDs des cases à associer à chaque tuile. Ses dimensions sont définies par le nombre d'IDs par ligne et le nombre de ligne. Les personnages et le curseur sont, eux, créés par les méthodes initPlayers et initCursor de State.

#### 3.2 Conception logiciel

Le diagramme des classes pour le rendu est présenté dans la figure suivante, on y retrouve les classes suivantes:

**Classe TileSet:** Classe qui charge les images utilisées pour les sprites des différents éléments du terrain dans les tuiles. Elle contient la longueur et la largeur en pixel d'un sprite dans l'image source, le chemin vers cette image source et la texture générée à partir d'elle. Son constructeur TileSet initialise ces attributs à partir de l'id de la couche (LayerId) à afficher.

**Classe Layer:** Classe qui construit chaque couche avec les textures correspondantes sous forme d'un tableau de quads. Elle permet aussi de dessiner le rendu de ces couches pour les préparer à l'affichage.

**Classe StateLayer:** Classe qui crée et gère les différentes couches du rendu. Elle réagit aux changement d'état en récupérant la classe State et permet éventuellement la mise à jour des textures.

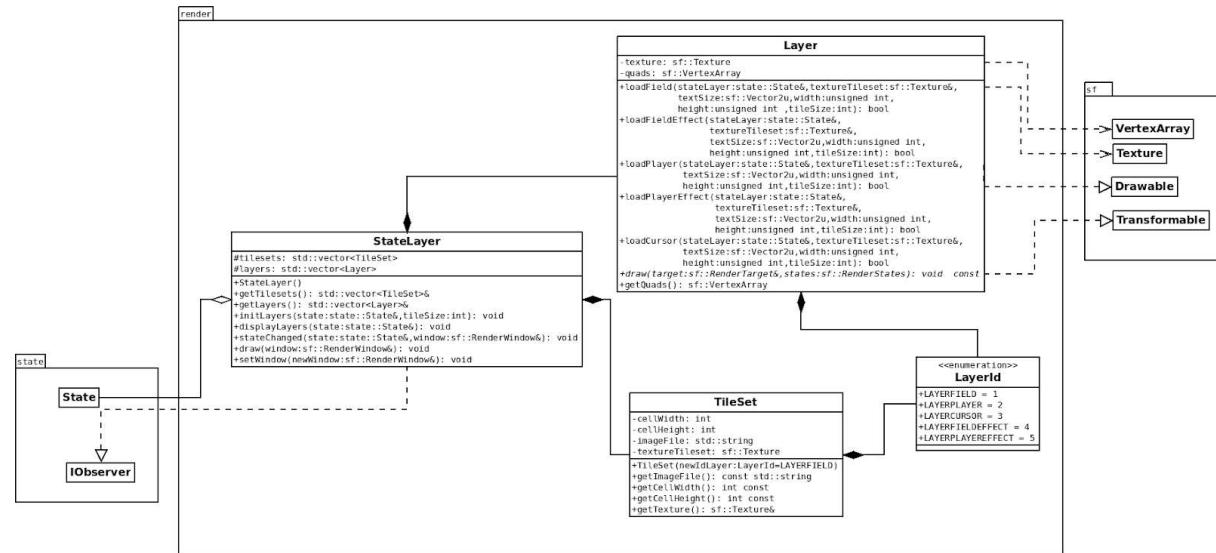


Illustration 2: Diagramme de classes pour le rendu

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

### 4.1 Règles de changement d'états

En début de tour de jeu, le moteur du jeu arrange la liste de l'ordre d'action des joueurs en fonction de leur statistique d'initiative, en prenant en compte les bonus/malus du terrain et de l'état des joueurs. En cas d'initiative égale, l'ordre est choisi aléatoirement.

Le tour d'un joueur commence quand le pointeur "playing" de la classe State le désigne, lui permettant effectuer des actions. Chaque action est provoquée par une commande et modifie l'état du jeu. Lors d'un tour joueur, le joueur peut effectuer, selon les cas, les trois actions listées ci-dessous :

- **Se déplacer :**

Le joueur déplace son personnage sur les cases du terrain. Un déplacement peut se faire sur une ou plusieurs cases, et le joueur peut faire autant de déplacement qu'il le souhaite tant qu'il lui reste des points de mouvement durant le tour. Le nombre de points de mouvement consommés par chaque déplacement dépend du type de déplacement du personnage et du type de terrain traversé. Certains statuts de terrain ou du personnage peuvent modifier cette valeur.

Enfin, un personnage ne peut pas se déplacer dans une case dans 3 cas : si le nombre de points de mouvement requis pour le déplacement est supérieur aux points de mouvement restant du personnage, si le terrain a un statut BLOCKMOVE qui le rend infranchissable (un mur par exemple), ou s'il y a déjà un autre joueur sur la case.

Lorsque le personnage arrive sur une case, les tests des effets du terrain (poison, brûlure...) se lancent pour décider s'il subit l'effet ou non. Si un test réussit, le personnage s'arrête sur la case et il subit l'effet associé au test. Dans ce cas, le joueur consomme uniquement les points de mouvement requis pour arriver sur la case où il s'est arrêté, et il peut continuer son tour normalement.

- **Attaquer :**

Pour être lancé, une attaque doit respecter plusieurs conditions: il doit avoir assez de point d'action, son compteur de tours de recharge Cooldown doit être nul, aucune case située entre la case visée et le lanceur ne doit avoir le statut BLOCKATTACK (case ciblée comprise), et la case visée doit être sur la même ligne ou colonne que l'attaquant à une distance comprise entre sa portée (range) minimale et maximale. Une attaque n'a donc pas nécessairement besoin de viser la position d'un autre joueur. Si ces conditions sont respectées, le joueur peut lancer une de ses attaques et sélectionner la case qu'il souhaite cibler parmi les cases disponibles.

Lorsqu'une attaque est lancée, le moteur effectue d'abord un test pour vérifier si l'attaque aboutit. Ce test se fait en fonction de la précision de l'attaque, du statut du personnage (par exemple, l'état DAZZLED diminue la précision), du type et statut de la case ciblée, et du statut des cases intermédiaires. Si le test rate, l'attaque échoue, sinon le moteur vérifie quelles cases sont

affectées par l'attaque, en particulier pour les attaques de zone. Si une case possède le statut BLOCKATTACK, cette case et les cases derrière elle ne seront pas affectées par l'attaque.

Après une attaque effectuée avec succès, on lance un test pour vérifier si les effets liés aux terrains s'appliquent. Pour cela, un seul test est effectué pour l'ensemble des cases de la zone d'attaque. S'il réussit, le statut du terrain est modifié selon l'effet de l'attaque, et l'effet dure pendant un nombre de tour de jeu donné. Ensuite, pour chaque joueur, on effectue un test pour vérifier s'ils sont touchés, à l'exception du joueur présent sur la case ciblé qui est obligatoirement touché. Enfin, on applique le calcul des dommages puis on lance, pour chaque joueur touché, un dernier test pour vérifier si les effets liés aux personnages sont appliqués. Enfin, on remet le compteur de recharge CoolDown à la valeur de recharge de l'attaque, CoolDownBase, et on diminue le nombre de point d'action du joueur de 1.

- **Terminer son tour:**

La fin de tour n'est pas automatique, c'est une commande faite par le joueur uniquement. Lorsqu'un personnage termine ses actions, on :

- rétablit ses points d'action et de mouvement en fonction de leurs valeurs de bases et des effets en cours,
- applique les dégâts dus au statut du personnage et du terrain,
- applique les changements de statut du joueur dus au terrain,
- décrémente tous les compteurs de 1 (cooldown d'attaque, statut de joueur, statut de terrain),
- on passe le pointeur "playing" au joueur suivant.

Si il n'y pas d'autre joueur lorsque l'on passe au suivant, alors c'est la fin du tour de jeu. On applique alors un changement de statut aléatoire (à faible probabilité) sur l'ensemble du terrain. On retourne ensuite en début de tour de jeu.

Si, après une attaque ou l'application d'un effet, un personnage perd tous ses points de vie, il est retiré du terrain et de la liste des joueurs. S'il ne reste qu'un joueur, la partie se termine.

Toutes les commandes peuvent être enregistré dans un fichier .txt et rejoué en lisant ce fichier. Le moteur a donc une variable "record" contenant la liste des commandes effectuées et le nombre de commande total, elle est initialement vide. A chaque commandes effectuées, le nombre de commande augmente et on ajoute le descriptif de la commande à la liste. Le contenu du descriptif est différent pour chaque types de commande:

- **Se déplacer:**

- id : id de la commande
- joueur : joueur lançant la commande
- xDestination : coordonnée horizontale de la case visée
- yDestination : coordonnée verticale de la case visée

- **Attaquer:**

- id : id de la commande
- joueur : joueur lançant la commande

- xDestination : coordonnée horizontale de la case visée
- yDestination : coordonnée verticale de la case visée
- attaque : numéro du skill dans la liste de skill du joueur

•Terminer son tour:

- id : id de la commande
- joueur : joueur lançant la commande

## 4.2 Conception logiciel

Le diagramme des classes pour l'engine est présenté dans la figure suivante, on y retrouve les classes suivantes:

**Classe Command:** Classe dont hérite toutes les classes décrivant une commande du joueur.

**Classe Attack:** Classe héritée de Command. Elle gère le lancement d'une attaque sur une position de la grille.

**Classe Move:** Classe héritée de Command. Elle gère le déplacement d'un personnage d'une case à une autre.

**Classe EndActions:** Classe héritée de Command. Elle gère la fin de tour d'un joueur et du tour de jeu.

**Classe Engine:** Classe qui gère le lancement des commandes et effectue donc les changements d'état.

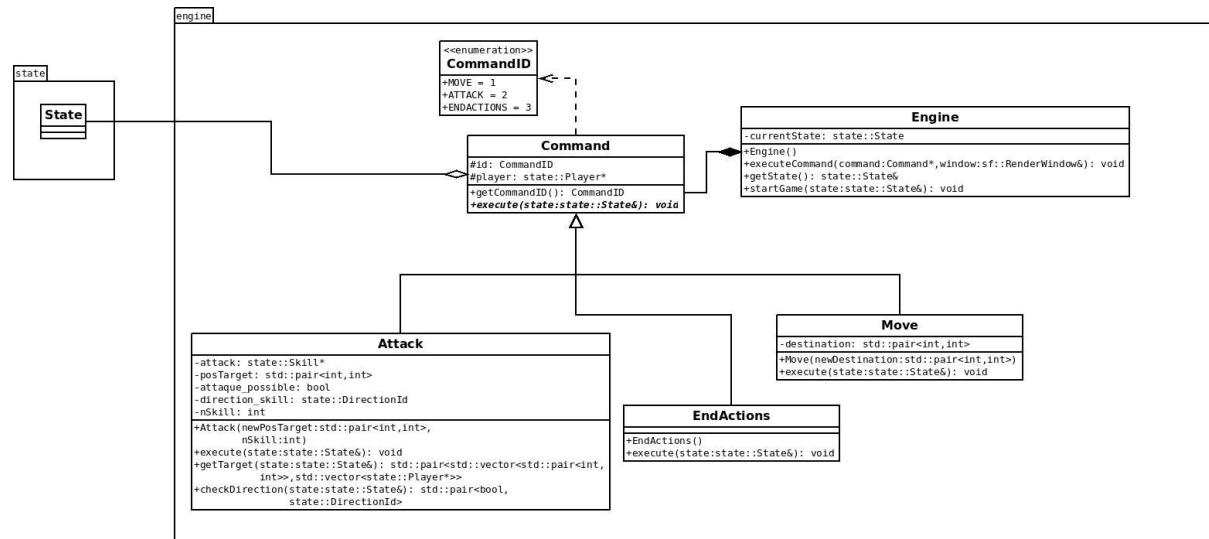


Illustration 3: Diagrammes des classes pour le moteur de jeu

## 5 Intelligence Artificielle

### 5.1 Stratégie

#### 5.1.1 Intelligence minimale

L'IA contrôle l'ensemble des joueurs non-humains lors de leurs tours de jeu. Une fois qu'un de ces joueurs est actif, une action précise est choisie au hasard parmi 3 : se déplacer, attaquer et terminer son tour d'action. Tant que l'action "terminer son tour" n'a pas été réalisée, on relance le choix aléatoire de l'action et on l'exécute. Chaque action ne peut être sélectionnée que sous certaines conditions : pour "se déplacer", il faut que le joueur ait encore des PM et pour "attaquer", il faut qu'il lui reste des PA et que ses attaques aient un cooldown nul.

Si « se déplacer » est choisi, une destination va être choisie aléatoirement parmi les quatre directions possibles. Si la case ciblée n'est pas accessible ou que le déplacement est impossible, l'action est annulée et on passe à la prochaine action aléatoire, sinon, on exécute le déplacement.

Si « attaquer » est choisi, une attaque est choisie aléatoirement parmi les attaques disponibles du personnage (c'est-à-dire avec un cooldown nul). Une direction d'attaque est choisie aléatoirement ainsi qu'une distance comprise entre la portée minimale et maximale de l'attaque pour déterminer la case visée. On exécute alors l'attaque sur cette case.

Si « terminer son tour d'action » est choisi, le joueur va terminer son tour directement. Le tour de l'IA est terminé lorsque l'action "terminer son tour" est choisie ou que le personnage est mort durant son tour.

#### 5.1.2 Intelligence basée sur des heuristiques

Nous améliorons maintenant le comportement de l'IA en proposant un ensemble d'heuristiques.

L'IA va principalement chercher à attaquer l'ennemi le plus proche.

En cas de points de vie faibles, le personnage tentera prioritairement de se soigner si possible.

Ensuite, si le personnage peut attaquer, c'est-à-dire qu'il lui reste des points d'action et qu'au moins une de ses compétences offensives n'est pas en recharge, il tentera d'attaquer l'ennemi le plus proche. Pour cela, il se rapprochera si nécessaire de l'ennemi afin qu'il soit à portée de son attaque disponible la plus puissante et lancera l'attaque.

Si le personnage ne peut pas attaquer, il cherchera d'abord à sortir du terrain dangereux (brûlé, poison) sur lequel il se trouve si c'est possible. Puis, il décidera de se rapprocher de l'ennemi le plus proche en fonction de s'il est désavantage ou avantagé ou de se soigner si il a reçu des dommages, même léger.

Enfin, le personnage passera son tour d'action si le personnage n'a plus d'actions possibles.

### 5.1.3 Intelligence basée sur les arbres de recherche

Nous améliorons encore l'intelligence artificielle en lui faisant choisir une action à effectuer à partir d'un arbre de recherche.

Tout d'abord, la liste des actions que peut effectuer le joueur actuel est créée. Pour diminuer la complexité, on ne va pas s'intéresser directement à une suite d'actions (ex : trajectoire de déplacement suivie d'une attaque), mais on va se limiter à une seule action à la fois (ex : se déplacer d'une case, attaquer, finir son tour). Ensuite, il s'agit de simuler une à une ces actions. Pour chacune des actions simulée, on va simuler toutes les actions possibles par chaque ennemi sur le terrain.

On va alors attribuer un score à chaque état qui résulte de ces actions (l'action du joueur suivie de l'action de l'ennemi). On obtient alors, pour chaque action possible du joueur, une liste de scores associée.

Le recherche de l'action optimale s'effectue en utilisant l'algorithme MinMax. On cherche d'abord, pour chacune des actions possibles du joueur, le score le plus faible issu de l'évaluation (Min). Puis, parmi la liste des faibles scores, on choisit le score plus élevé (Max). L'action optimale est alors l'action associée à ce dernier score.

Pour le calcul du score, la fonction d'évaluation prend en compte les points de vie du joueur et des adversaires. Le nombre de points de vie du joueur fait augmenter le score tandis que celui des ennemis le fait diminuer. Un autre paramètre est aussi pris en compte : la vie et la distance par rapport aux ennemis. Etre proche d'un ennemi plus faible fait augmenter le score alors qu'il diminue c'est le joueur qui est le plus faible.

## 5.2 Conception logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 4.

**Classe IA :** Les classes filles de la classe IA implémentent différentes stratégies d'IA. Elles possèdent une fonction « run ».

**Classe RandomIA :** Classe qui implémente l'IA aléatoire.

**Classe HeuristicIA :** Classe qui implémente l'IA heuristic.

**Classe DeepIA :** Classe qui implémente l'IA avancé.

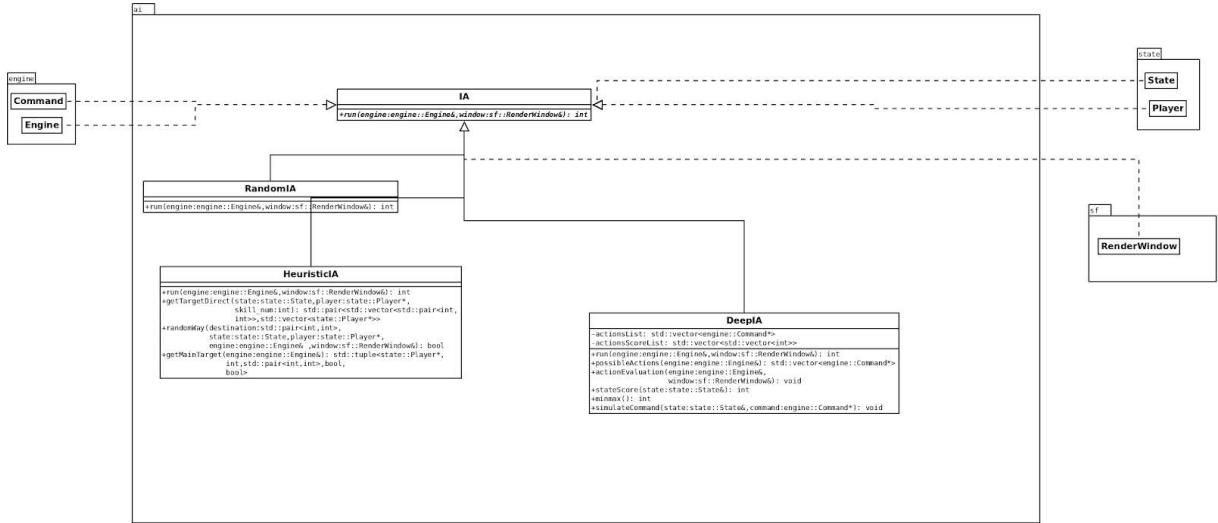


Illustration 4: Diagrammes des classes pour l'intelligence artificielle

## 6 Modularisation

### 6.1 Organisation des modules

#### 6.1.1 Répartition sur différents threads

L'objectif est de faire en sorte que le moteur de jeu et le moteur de rendu soient sur des threads différents. On a alors le moteur de rendu sur le thread principal et le moteur de jeu sur un thread secondaire.

Le moteur du jeu reste généralement passif, il ne devient actif lorsqu'il est nécessaire d'exécuter une commande et de changer l'état de jeu actuel, par exemple lorsque l'IA envoie une commande à exécuter ou quand l'utilisateur appuie sur une touche pendant son tour de jeu. Les commandes à exécuter sont stockées dans les variables `nextCommand` (par l'IA) et `nextKeyCommand` (touche utilisateur). Une variable située entre les deux threads est nécessaire afin d'informer le moteur de jeu qu'une nouvelle commande est stockée puis, après exécution, notifier la mise à jour du rendu.

#### 6.1.2 Répartition sur différentes machines

On utilise une API Web REST pour rassembler les joueurs sur les différentes machines (ou processus) avant le début de la partie. Une fois cela fait, il sera possible de démarrer la partie. Aucun changement de joueur ne peut être effectué après le début de la partie.

Le serveur possède une liste de clients et peut recevoir différentes requêtes basées sur la donnée "joueur" grâce à l'API Web REST :

- **Requête GET/player/<id>** : Renvoie les caractéristiques (son nom et sa présence) du joueur correspondant à l'id au format Json.
- **Requête PUT/player** : Ajoute un nouveau joueur à la liste des joueurs. Si le nombre maximal de joueur n'a pas été dépassé, le nouvel id attribué à ce joueur est renvoyé au format Json. Une requête GET est nécessaire si on veut consulter les caractéristiques du joueur ajouté.
- **Requête POST/player/<id>** : Modifie les caractéristiques du joueur correspondant à l'id si celui-ci est valide. Une requête GET est nécessaire si on désire consulter ces nouvelles caractéristiques.
- **Requête DELETE/player/<id>** : Supprime le joueur correspondant à l'id de la liste des joueurs s'il existe.

## 6.2 Conception logiciel

**Classe Client** : La classe Client contient tous les éléments permettant de faire fonctionner le jeu, c'est-à-dire le moteur de jeu, le moteur de rendu et l'intelligence artificielle. Elle rend le moteur de jeu actif uniquement lorsqu'une commande est générée par l'intelligence artificielle ou le joueur. Elle s'occupe aussi d'envoyer des notifications de mise à jour du rendu.

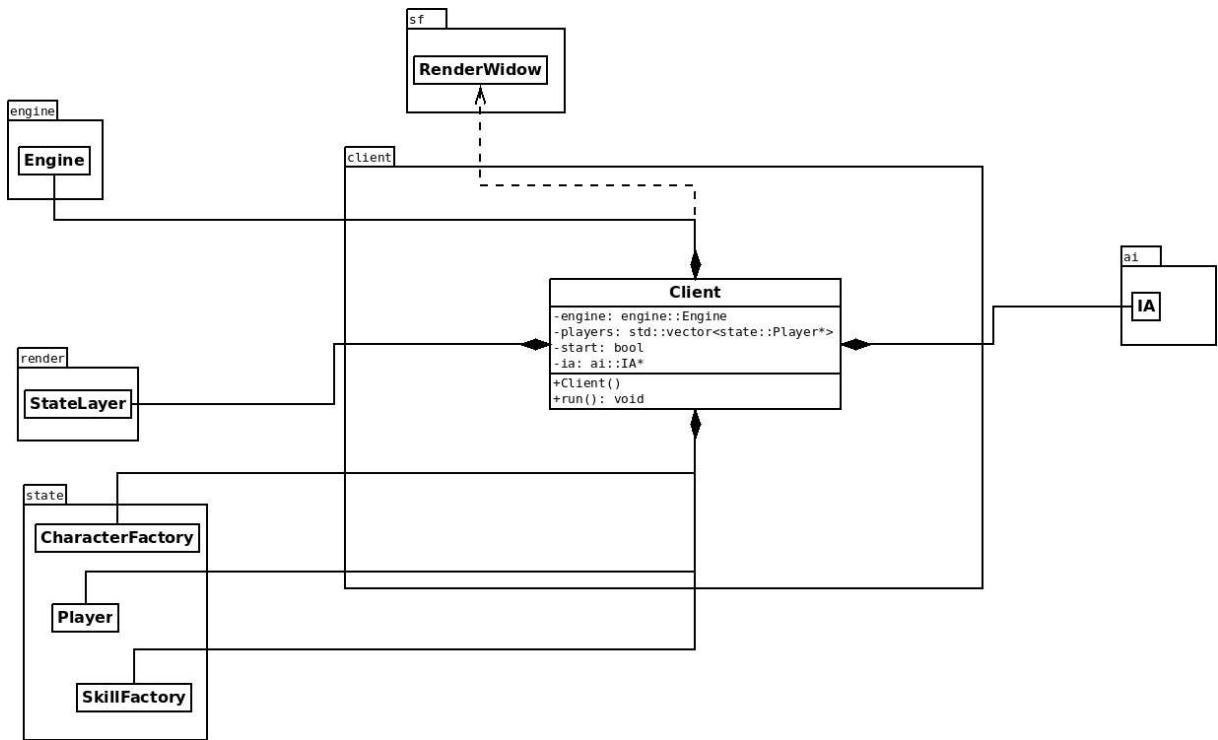


Illustration 5: Diagrammes des classes pour le client

### 6.3 Conception logiciel : extension réseau

**Classe Game** : La classe Game contient les éléments qui composent une partie du jeu. On y trouve la liste des joueurs présents dans la partie.

**Classe Player** : La classe Player décrit les caractéristiques de chaque joueur de la partie, c'est-à-dire leur nom et leur présence (free).

**Classes Services** : Plusieurs classes de service sont présentes pour les différentes opérations à exécuter. Elles héritent de la classe **AbstractService** et sont gérées par la classe **ServiceManager** qui choisit le bon service en fonction de l'URL et de la méthode HTTP. On trouve les services :

- **VersionService** : qui renvoie la version actuelle de l'API pour éviter les conflits de version.
- **PlayerService** : qui permet d'ajouter, modifier, consulter et supprimer des joueurs.

**Classe ServiceException :** Elle permet de créer une exception pour interrompre l'exécution d'un service.

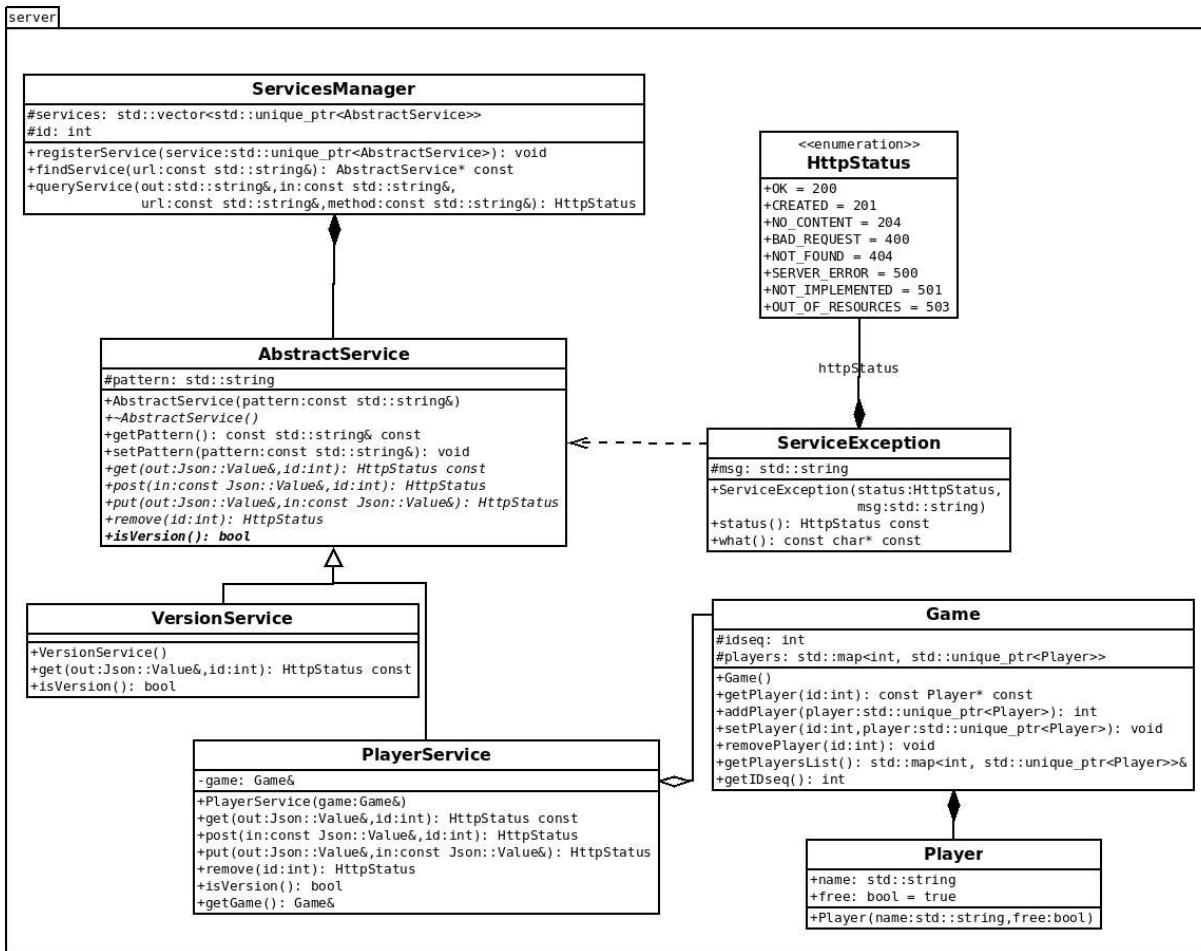


Illustration 6: Diagrammes des classes pour le client

## **6.4 Conception logiciel : client Android**

*Illustration 4: Diagramme de classes pour la modularisation*

