Projet Logiciel Transversal

Lotus

Illustration 1 - Exemple du jeu Dofus

LEMGADAR Hamza

LEPOIVRE Alexandre

DAMA Abdoulaye

EDDAAIF Rachid

Table des matières

1 Objectif

1.1 Présentation générale

L’objectif de ce projet est la réalisation d’un dofus like simplifié. Dofus est un jeu en ligne massivement multijoueur, développé par Ankama. Nous faisons évoluer notre personnage au sein d’un open-world en augmentant son niveau, développant son équipement et ses sorts.

Nous nous limiterons au lancement et à la gestion du combat. Seul le un contre un avec l’IA sur une map sera disponible. Le système de combat se fait en tour par tour et le joueur possède des points d’actions et de mouvements qu’il peut utiliser afin de se déplacer et attaquer l’adversaire.

1.2 Règles du jeu

Le joueur peut se déplacer sur une map sur laquelle des monstres à combattre apparaissent. Le joueur peut cliquer sur un monstre afin de lancer un combat.

Pendant le combat, le joueur peut lancer des sorts afin de combattre le monstre et se déplacer à l’aide des points de mouvements. Le déplacement est définis par des une grille de cases. Les monstres peut lancer des attaques sur le joueur. Si les monstres ou le joueur meurt, le combat se termine. Lors de la fin de combat, le joueur revient sur la map d’origine et un nouveau groupe de monstre est généré sur la map.

Le joueur et les monstres possède des statistiques diverses : PV, points d’action, points de mouvement.

Le joueur possède un niveau qui lui permet d’obtenir des nouveaux sorts. Son niveau augmentent grâce à l’expérience qu’il gagne à chaque fin de combat.

1.3 Ressources

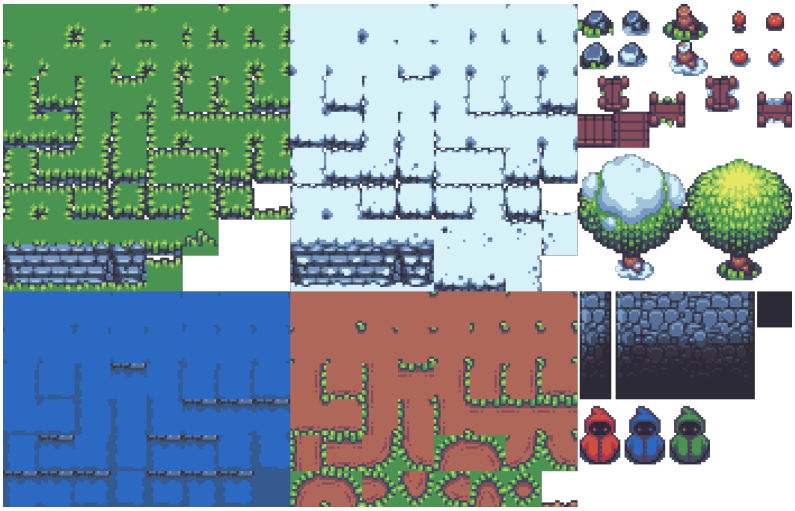

Illustration 2 - Ressources pour création de map

Lien de la ressource : https://kenney.nl/assets/isometric-blocks

Illustration 3 - Ressources pour les personnages

Lien de la ressource : https://opengameart.org/content/lpc-medieval-fantasy-character-sprites

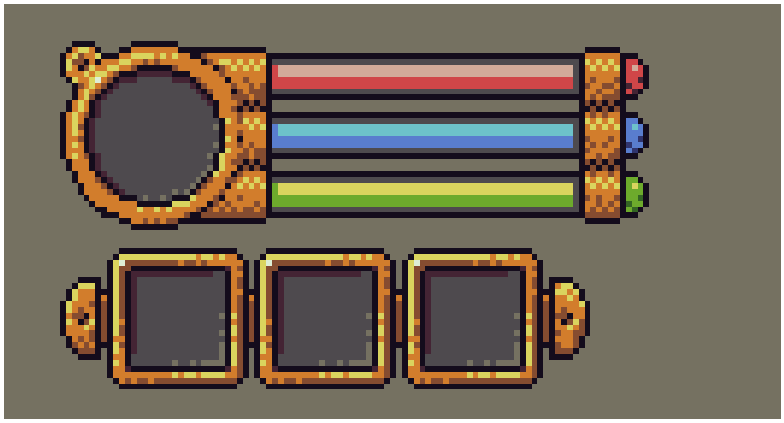

Illustration 4: Ressources pour l'interface

Lien de la ressource : https://opengameart.org/content/golden-ui

Illustration 5 - Ressources pour les sorts

Lien de la ressource : https://opengameart.org/content/attack-icons-wesnoth

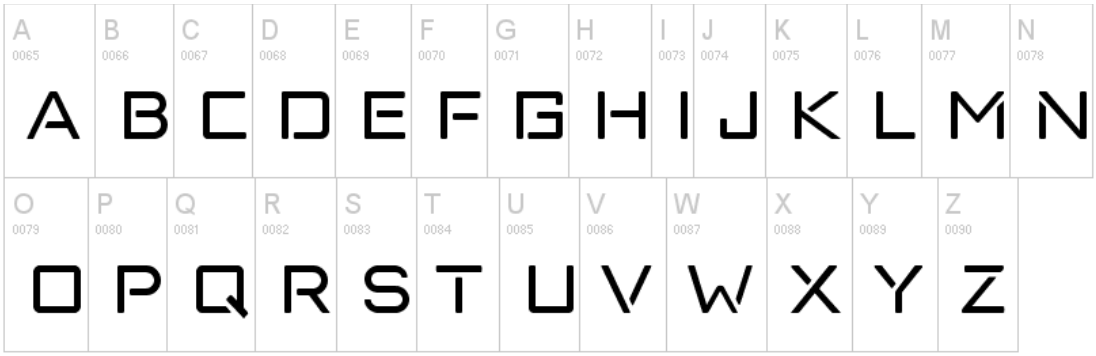

Illustration 6 - Ressource pour la typographie

Lien de la ressource : https://www.dafont.com/neo-latina.font?text=D%E9but+du+combat

2 Description et conception des états

2.1 Description des états

Notre jeu se compose par différents éléments mobiles composés des personnages et monstres. Ces éléments interagissent sur une map contenant des éléments fixes. Les interactions s'effectuent grâce à la position de chaque élément de notre map.

L'ensemble des éléments possède les propriétés suivantes :

- Un type d'élément
- Des coordonnées x et y associées à une case de notre grille

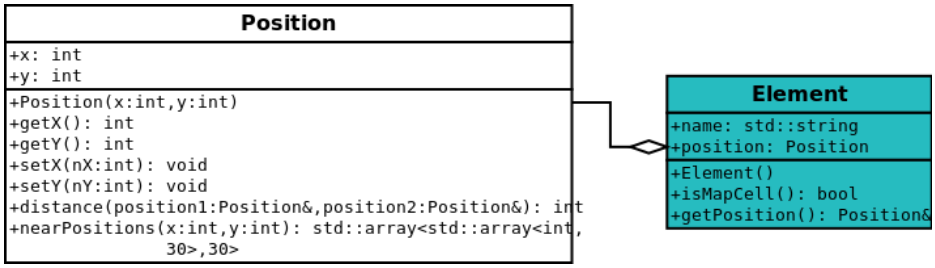

Illustration 7 - Classe Element

2.1.1 Eléments mobiles

L'élement mobile « Character » regroupe à la fois le monstre et le personnage joué par le joueur. Le « Character » possède différentes statistiques, regroupées au sein de la classe « Stats » : sa vie « health », ses points d'actions « actPoints » et de mouvements « movPoints », son expérience « experience » et son niveau « level ». Ces caractéristiques seront assignés à chaque joueur et monstre(s) présents lors d'un combat.

Le « Character » possède des sorts, regroupés au sein de la classe « Spell », qu'il pourra effectuer. Chaque sort possède les caractéristiques suivantes : un identifiant propre au sort « ID », des dégâts « Damage », une portée « Range » et un coût « Cost » qui limitera les actions du joueur pendant son tour.

Le « Character » possède un statut qui lui est propre « CharacterStatusID ». Lorsque le joueur est en combat et joue son tour son statut est défini sur « Playing ».Il est définit sur «Waiting » dans le cas inverse. En fin de combat, l'état du character passe à « Win » en cas de victoire et « Death » en cas de défaite. Dans le cas où le joueur n'est pas entré en combat, son statut est définit par « Wander ».

La classe « Character » regroupant à la fois les joueurs et les monstres, nous différencions ces deux entités à l'aide d'un ID au sein de « CharacterTypeID » : «PLAYER» pour notre joueur et « MOB» pour nos monstres. Le type de « Mob » est définit par l'énumération « MobTypeID » qui regroupe les différents monstres présents dans notre jeu.

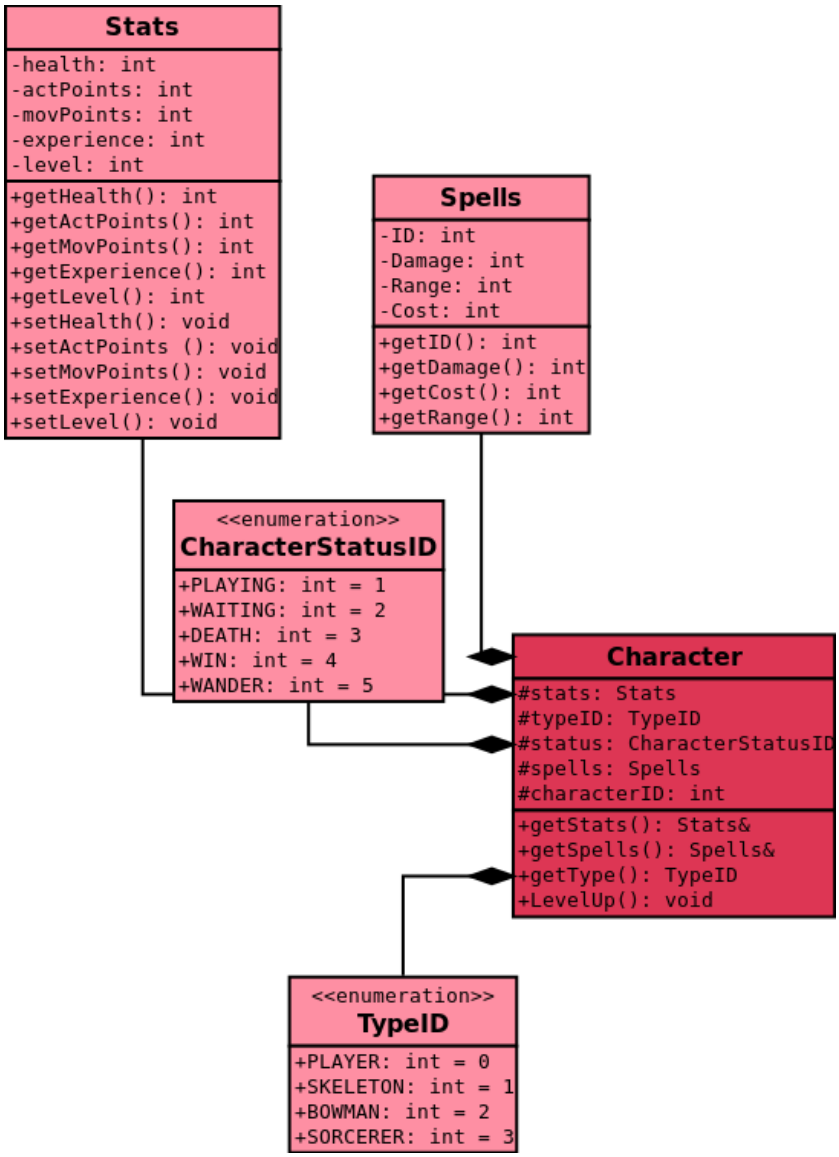

Illustration 8 - Classe Character

2.1.2 Eléments fixes

Notre map se compose d'éléments fixes remplissant une grille de cases. Ces cases permettent à notre joueur et aux monstres de se déplacer. Deux types d'éléments constituent notre map :

- Les cases obstacles, infranchissables par nos characters
- Les cases libres, franchissables par nos characters

Ces différents éléments ont été attribués lors de la création de notre map.

Les obstacles sont constitués par des :

- Arbres « TREE »
- Souches d'arbres «STUMP»
- Murs « WALLS »
- Eaux « WATER »

Les cases vides de notre map sont constituées de :

- L'herbe « GRASS »
- La terre « DIRT »

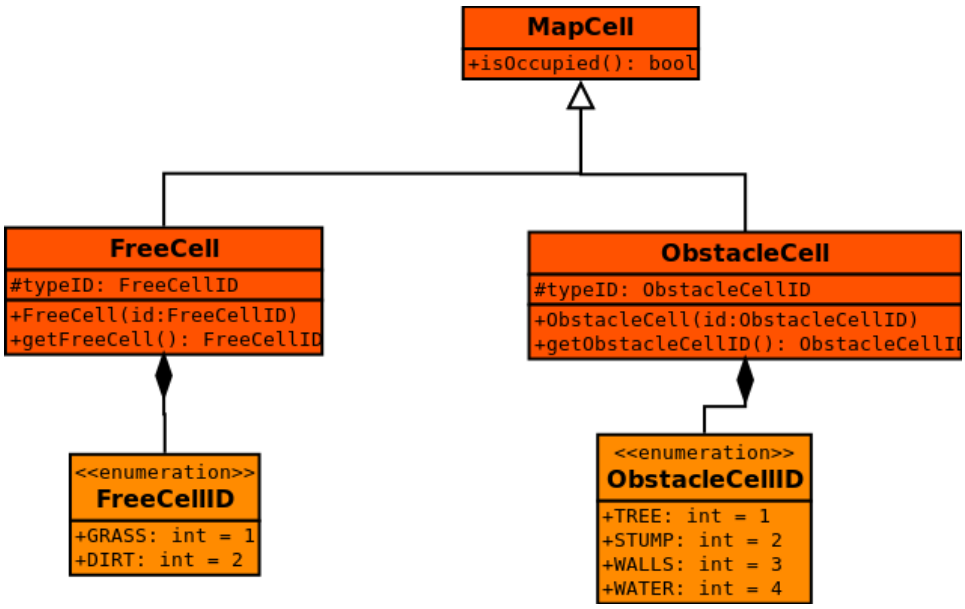

Illustration 9 - Classe MapCell

2.2 Conception logiciel

L'ensemble des actions effectuées par le joueur s'effectueront à l'aide de la souris représentée à l'écran par le curseur. La classe Cursor permet d'obtenir les positions du curseur de la souris mais aussi de connaître si le joueur est entrain de cibler une case fixe de la map décrite ci-dessus.

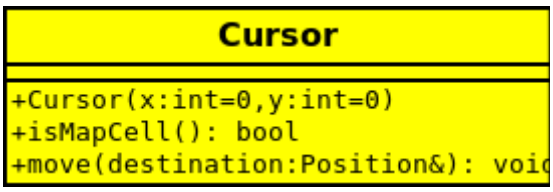

Illustration 10 - Classe Cursor

La classe State décrit les différents éléments de nos états de jeu. Elle se compose d'une map, des characters en jeu ainsi que des actions en cours.
Elle référence les deux états possibles de notre jeu :

- L'état « Wander » qui permet au joueur d'engager le combat avec un monstre
- L'état « Fight » qui correspond au combat avec le monstre.

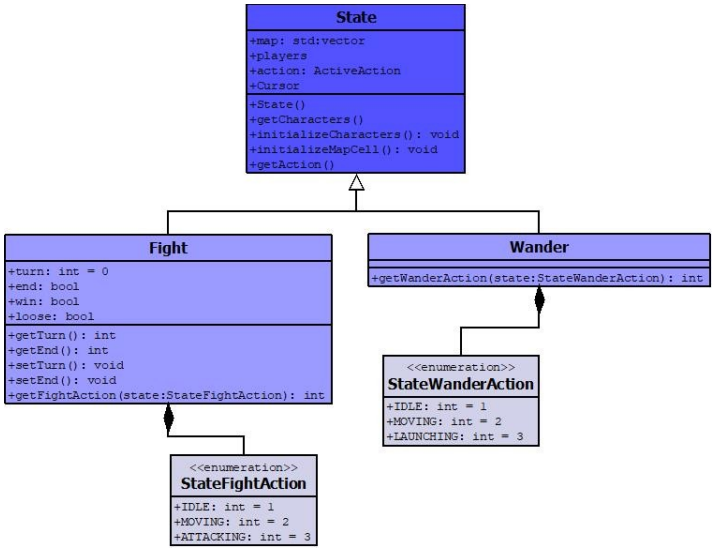

Illustration 11 - Classe State

L'ensemble des changements réalisés au sein de nos états sont gérés au sein de la classe Observable. On notifie toute action effectuée par l'utilisateur ainsi que par les autres classes. Ces informations sont stockées au sein de d'un registre de notifications.

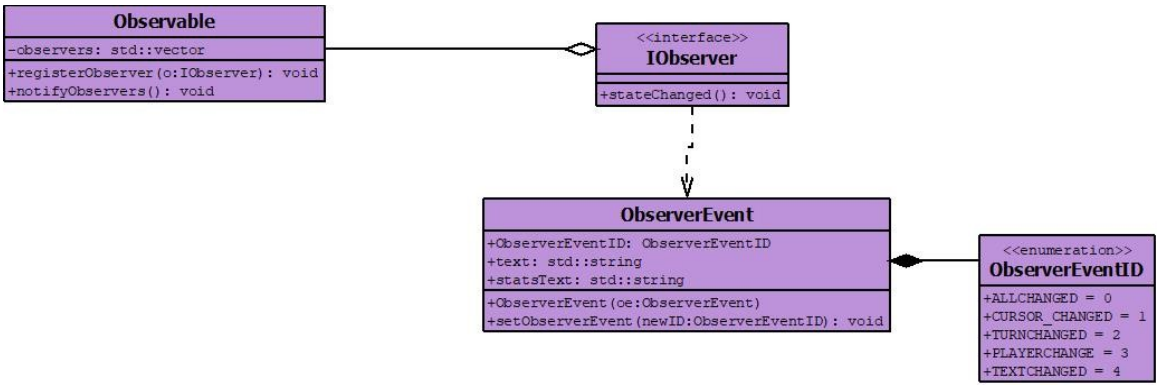

Illustration 12 - Classe Observable

2.3 Conception logiciel : extension pour le rendu

2.4 Conception logiciel : extension pour le moteur de jeu

2.5 Ressources

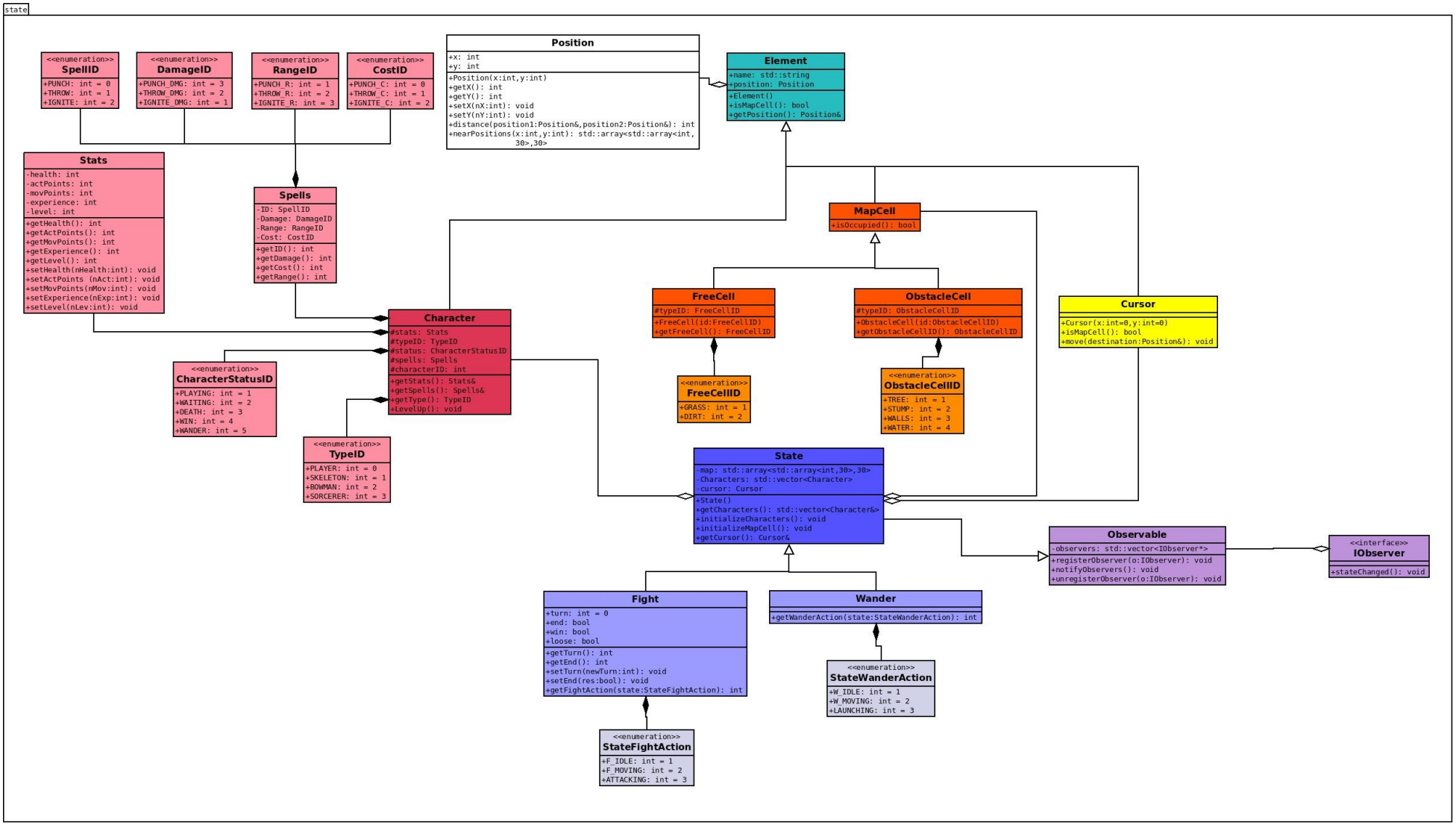

Illustration 13: Diagramme des classes d'état

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'implémente 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 définir une stratégie de rendu, il faut connaître l'ensemble des éléments qui définissent un état. Tout d'abord, nous avons **le terrain de jeu** qui est unique dans le sens où on a la même map dans une situation de combat ou d'exploration, **les personnages** ainsi que **les informations** liées à ces personnages (santé, points de mouvement, ...).

A l'aide de la bibliothèque SFML, nous avons choisi la méthode de rendu par tuiles pour un état donné. Notre surface de jeu est donc composée de 3 couche:

- Une couche contenant les éléments immobiles et inaccessibles de la map, comme le sol, les montagnes de décor, les lacs...
- Une couche contenant les personnages, le curseur et l'animation des sorts.
- Une couche de fond noir uniquement pour le décor mais qui n'est pas forcément nécessaire.

En plus de la surface de jeu on aura une surface pour les informations liées aux caractéristiques des personnages, notamment une barre de santé dynamique qui évoluera en fonction des dégâts subits qu'ils subiront ou la barre de points d'action qui évoluera en fonctions des mouvements effectués par les personnages.

La map est construite à partir d'un fichier texte contenant l'ID associé à chaque tuile. Si on souhaite rendre accessible une zone occupée par de l'eau (qui est inaccessible), il suffit de changer les IDs associés aux tuiles occupants cette zone en ID du sol. L'initialisation des personnes suit le même principe, elle charge les personnages en fonction de l'ID de la tuile et des coordonnées qui lui sont attribuées. Ces variables changent forcément au cours du jeu en fonction des choix de l'utilisateur où de l'IA, par conséquent le rendu visuel doit automatiquement se mettre à jour à chaque évolution d'état.

3.2 Conception logiciel

3.2.1 StateLayer

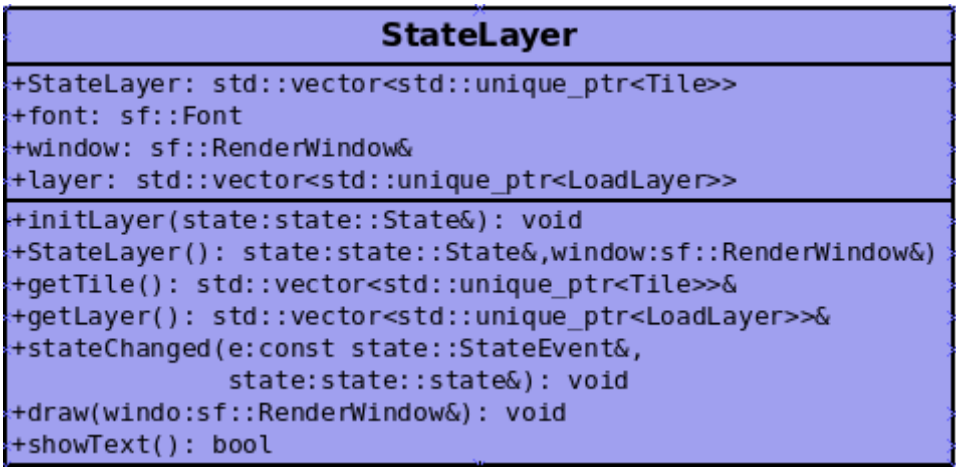

Illustration 14: StateLayer Class

Cette classe joue le rôle d'intermédiaire entre les packages **state** et **render.** Elle contient l'ensemble des différents tuiles, la typographie utilisée pour l'affichage des stats ou encore l'attribut « Window » pour la gestion de l'affichage graphique. En plus des *getters* on a 5 méthodes dans la classe :

- Le constructeur où on charge les fonts et les tuiles correspondantes.
- La méthode initLoadLayer(State) initialise les objets LoadLayer.
- La méthode *draw* qui prend en argument la fênetre d'affichage qui contient les différentes couches et éléments.
- La méthode *stateChanged responsable* d'écouter les avènements afin de pourvoir mettre à jour l'état présent du jeu.

3.2.2 LoadLayer

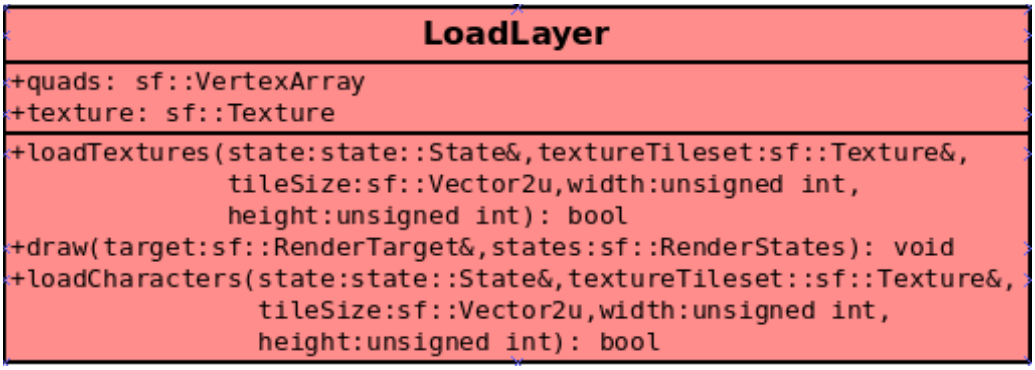

Illustration 15: LoadLayer Class

Cette classe utilise plusieurs classes de la librairie SFML. Elle permet de charger les textures et les personnages d'une couche donnée. L'attribut *quads*(type Vertex array) est un tableau de vertex, qui est la plus petite entité graphique qu'on peut manipuler, elle contient une position en 2d, une couleur et une paire de textures. Nous utilisons cette classe pour dessiner les tuiles de notre grille.

3.2.3 Tile

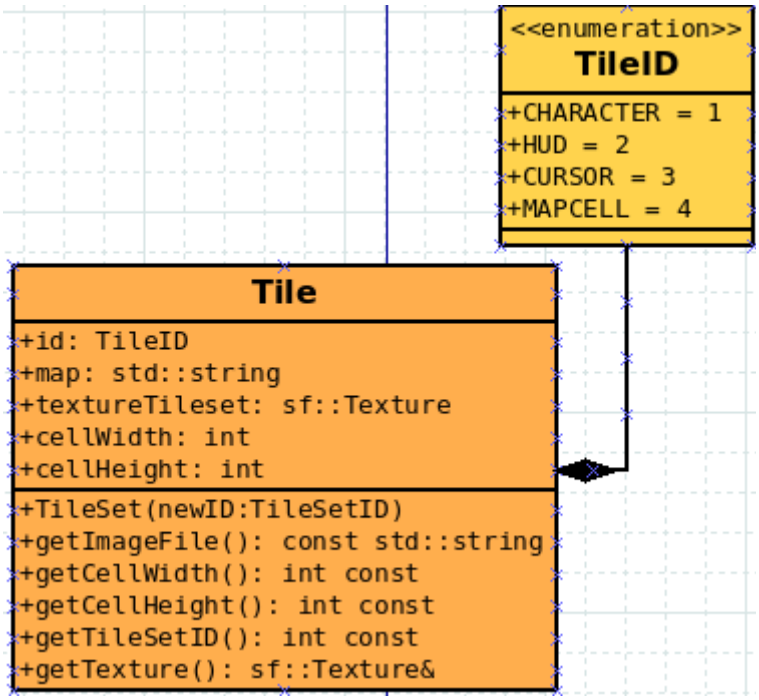

Illustration 16: Tile Class

Cette classe permet de définir la nature de chaque tuile, non seulement en lui attribuant la bonne texture mais aussi les bonnes dimensions (CellWidth, CellHeight). Chaque tuile a une dimension 32x32 pixels (reconfigurable). Le constructeur nous permettra juste de sélectionner une donnée et de la charger en mémoire pour pouvoir l'afficher par la suite.

3.3 Conception logiciel : extension pour les animations

3.4 Ressources

3.5 Exemple de rendu

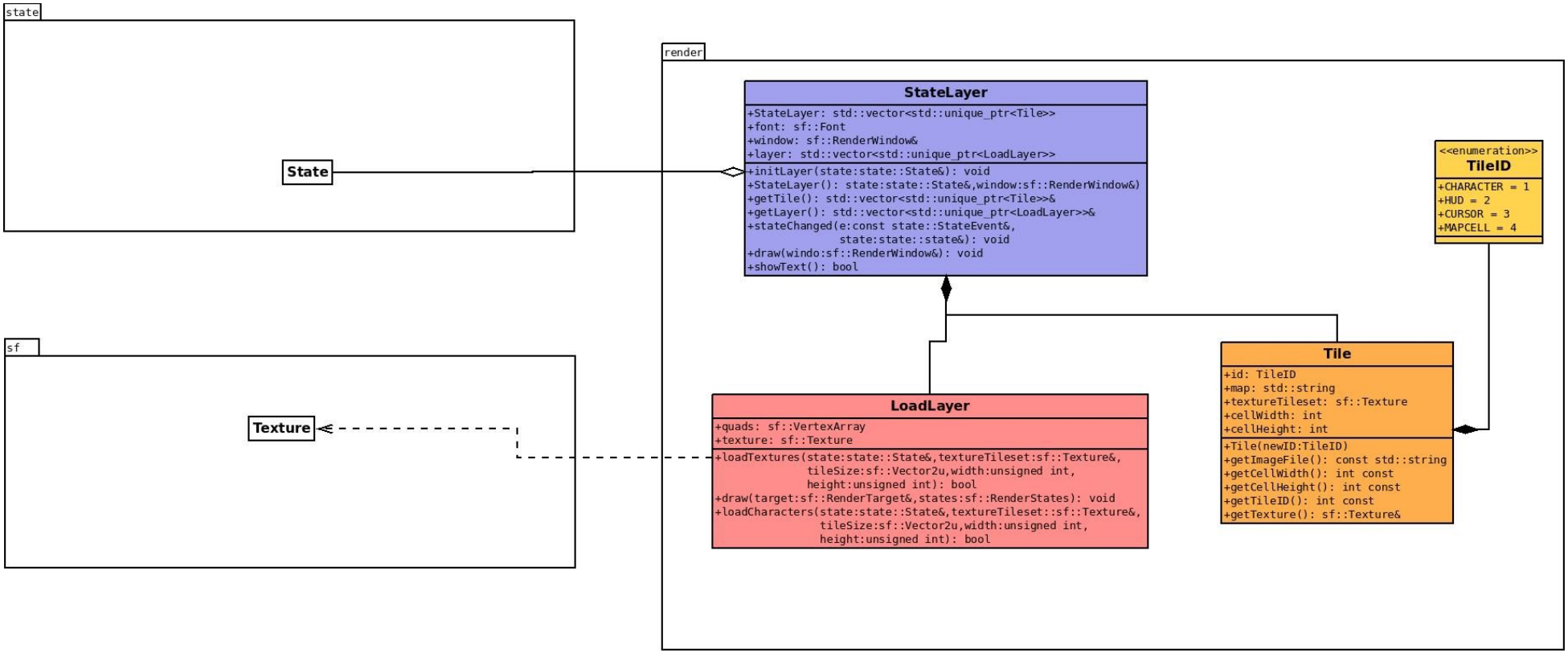

Illustration 17: Diagramme de classes pour le rendu

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 lié 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

Chaque changement d'état est régi par une horloge globale permettant le passage d'un état à un autre. Ces changements d'états sont définis dans notre cas pas le déplacement d'une case à une autre. L'ensemble des mouvements liés aux états aura donc la même temporalité.

4.2 Changements extérieurs

Les changements extérieurs correspondent aux changement d'états liés à des actions utilisateurs. Dans notre cas, il existe 3 types de changements extérieurs :

- Les commandes d'attaque
- Les commandes de déplacements
- Les commandes de sélection de personnage

Les commandes d'attaque permettent au joueur sélectionné d'attaquer avec ses attaques. Le joueur peut sélectionner une case sur laquelle il souhaite attaquer. Il choisit aussi l'attaque à envoyer et son statut est défini sur PLAYING.

Les commandes de sélection de personnage permet au joueur de sélectionner son personnage. Lorsque le personnage est choisi, son statut est défini sur PLAYING. Les autres personnages présents sont définis sur WAITING.

Les commandes de déplacements permettent au personnage de se déplacer selon le nombre de points de mouvements dont il dispose. Le nombre de points de mouvements est fixé à 3 pour notre personnage. Le joueur se déplaçant possède le statut PLAYING.

Lorsque le joueur a effectué son déplacement et son action, il peut mettre fin à son tour et son statut est modifié.

4.3 Changements autonomes

Les changements autonomes s'effectuent à chaque création ou mise à jour d'un état, après les changements extérieurs. Ces changements s'effectuent dans l'ordre suivant :

- Si le joueur meurt, on affiche LOOSE
- Les statistiques de chaque personnage sont mises à jour
- Si le joueur gagne, on affiche WIN

4.4 Conception logiciel

Afin de transformer les actions de l'utilisateur en modification de l'état, on utilise le patron Command.

La classe Command permet de regrouper les différentes commandes. Cette classe regroupe l'identifiant de la commande ainsi que exécution de la commande au moteur.

La classe CheckHealth permet de vérifier si l'un des personnages présent sur la map n'a plus de points de vie.

La classe SelectedCharacterCommand permet de connaître l'état d'un personnage.

La classe AttackCommand permet de connaître l'attaque du personnage.

La classe MoveCommand permettent de gérer le déplacement du personnage.

La classe SwitchTurnCommand permet de change le tour de jeu en cours.

La classe Engine est le moteur du jeu. Cette classe exécute les commandes qui lui sont fournis. Les tâches sont exécutés dans l'ordre.

4.5 Conception logiciel : extension pour l'IA

4.6 Conception logiciel : extension pour la parallélisation

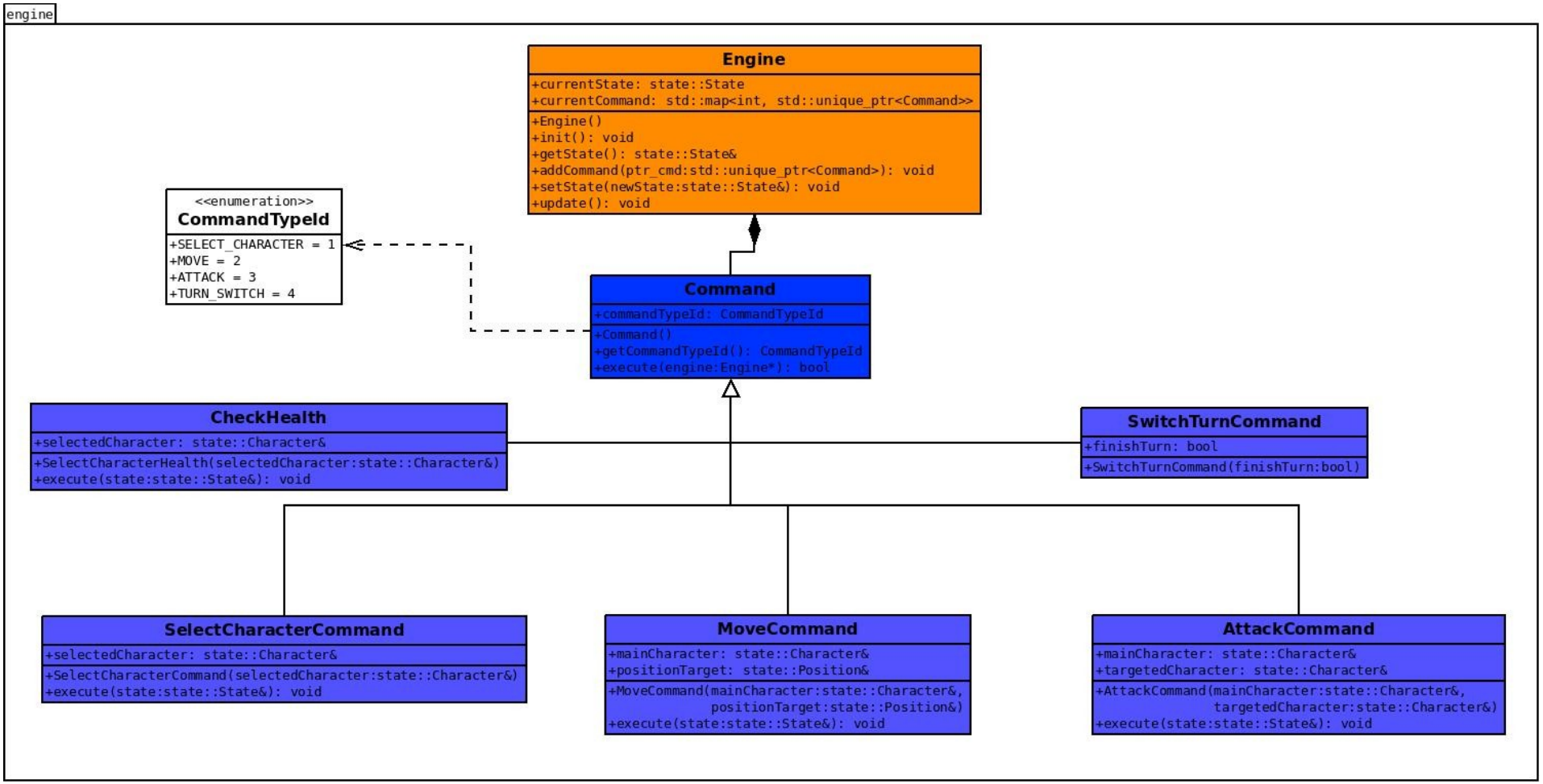

Illustration 18: Diagrammes des classes pour le moteur de jeu

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, il 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

5.2 Conception logiciel

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

6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau

6.4 Conception logiciel : client Android

Illustration 19: Diagramme de classes pour la modularisation