

Projet Logiciel Transversal

Nouhou KANE – Mustafa KARADAG – Alexandre LOUIS



PROFUS

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

PROFUS est un jeu vidéo inspiré du populaire MMORPG DOFUS. Comme nous ne pouvions pas reproduire toutes les fonctionnalités de DOFUS, nous avons fait le choix de nous concentrer sur les combats présents dans le jeu. Ainsi, dans PROFUS, nous avons pour objectif de permettre à un ou plusieurs joueurs de combattre l'un contre l'autre ou ensemble face à des IA. Il s'agit de combats au tour par tour dans un univers 2D isométrique donnant l'impression d'évoluer dans un monde en 3 dimensions. Le but est d'y vaincre son ou ses adversaires en faisant tomber leurs points de vie à 0, et ainsi accumuler de l'expérience pour monter de niveau et devenir plus puissant. Pour ce faire les personnages disposent d'un ensemble d'attaques, de portées et puissances différentes dont l'utilisation est limitée par un nombre de points d'actions. Mais nous allons voir cela plus en détail avec les règles du jeu.

1.2 Règles du jeu

Splash screen :

Phase de lancement du jeu : logo du jeu au centre avec une barre de chargement en dessous par exemple.

Menu principal :

Page d'accueil au lancement du jeu. On pourrait concevoir un petit diaporama pour que le fond ne soit pas statique ou rester sur quelque chose de plus simple avec une image de fond classique. + Paramètres

Choix de l'arène et du personnage :

Au début on se contentera d'une seule map et d'un seul personnage mais si on a le temps on pourrait laisser le choix entre 4 ou 5 personnes et 2 ou 3 maps/arènes. Donc dans cette section, le joueur pourra naviguer d'une possibilité à une autre avec des flèches et passer à l'étape suivante grâce à un bouton en bas de page. Le joueur pourra avoir plus d'informations sur le personnage sélectionné (classe, stats, histoire) en cliquant sur le logo (!).

Phase de combat :

Cette phase se base intégralement sur le déroulement d'un combat sur Dofus. Au lancement du combat, le joueur a la possibilité de se placer. Il a le choix avec un certain nombre d'emplacements prédéfinis. Il peut ainsi se rapprocher le plus possible de son adversaire ou décider au contraire de s'en éloigner. Par défaut, le joueur est placé de manière aléatoire. Il a également la possibilité de changer son orientation. En effet un coup reçu de face fera moins de dégât qu'un coup reçu dans le dos.

Durant son tour, chaque joueur dispose de 60/90 secondes, durant lesquelles il pourra réaliser une série d'actions (se déplacer, attaquer, etc..), avant que ce ne soit le tour de son adversaire.

Les points importants à prendre en compte sont les suivants :

- Chaque joueur dispose d'un certain nombre de points de vie (PV) (dépendant de la classe du personnage), une fois les PV à 0, le personnage a perdu.
- Les déplacements des joueurs sont limités par leur nombre de points de mouvements (PM) qui dans Dofus s'élèvent par défaut à 3 (pas pour les mobs). En consommant des PM le joueur peut se déplacer d'une case à l'autre de la map.
- Les attaques/sorts réalisables sont limités par le nombre de points d'action (PA), par défaut les joueurs en ont 5 (pas pour les mobs).

A noter que les points d'actions et de mouvements sont intégralement restitués à chaque début de tour.

Il sera nécessaire d'établir un certain nombre de sorts et pour cela nous pourrions reprendre les sorts existants sur Dofus afin de ne pas réinventer la roue.

Projet Logiciel Transversal – Nouhou KANE – Mustafa KARADAG – Alexandre LOUIS

Par exemple : <https://www.dofus.com/fr/mmorpg/encyclopedie/classes/8-iop>

A noter que les sorts ont des portées et zones de dégât différentes, qu'ils nécessitent plus ou moins de PA (parfois même de PM), qu'il peut y avoir une limite d'utilisation d'un sort durant un même tour voire un délai d'un certain nombre de tours entre 2 utilisations.

De plus, pour attaquer son adversaire, celui-ci ne doit pas se trouver derrière un obstacle.

En effet, un obstacle situé entre les 2 personnages, bloque la "vue" et empêche de lancer un sort. Cependant certains sorts ne prennent pas en compte la présence d'obstacle. Il a 2 manières de mettre fin à son tour. Soit le temps accordé de 60/90 secondes est écoulé, soit le joueur y a mis fin manuellement. Il est en effet possible de terminer son tour en cliquant sur le bouton associé (dans l'interface de Dofus il s'agit d'une flèche). Le but est donc de vaincre son adversaire en faisant tomber ses PV à 0.

Tuto combat Dofus: <https://www.youtube.com/watch?v=PAsw9IOE3pg>

Ex de combats Dofus: <https://www.youtube.com/watch?v=gmiAcgAm9DQ>

Type de Personnages :

A l'heure actuelle, nous proposons au joueur le choix entre 2 classes :

- **HERO** : il s'agit d'une classe privilégiant les attaques à distance. Le personnage utilisé pour cette classe se prénomme Vala, elle utilise une arbalète pour infliger des dégâts à ses adversaires.

et

- **DEMON** : il s'agit d'une classe privilégiant les attaques au corps à corps. Le personnage utilisé pour cette classe se prénomme demon, ayant des ailes elle possède une meilleure dextérité et utilise une faux pour infliger des dégâts à ses adversaires.

La classe HERO dispose davantage de points de vie que la classe DEMON (80 contre 70 au niveau 1) et également d'une plus grande résistance(shield) (25 contre 20 au niveau 1).

De l'autre côté la classe DEMON a davantage de points de mouvement (4 contre 2 au niveau 1) et inflige plus de dégâts (attack) (30 contre 25 au niveau 1).

Sorts :

Pour le moment chacun des personnages proposés ne possède que 2 attaques :

- Une attaque que l'on pourrait qualifier de classique, elle n'inflige pas beaucoup de dégâts mais ne nécessite que peu de points d'action et peut donc être utilisée plusieurs fois dans une même tour.

et

- Une attaque puissante, qui inflige davantage de dégâts mais consomme également plus de points d'action et qui par conséquent ne peut généralement être utilisée qu'une seule fois par tour.

Pour la classe HERO, les sorts disponibles sont les suivants :

- **Shoot** qui consiste à projeter une flèche à l'aide de l'arbalète de Vala pour atteindre des ennemis situés jusqu'à 5 cases de distance. Elle consomme 2 points d'actions et inflige 20 dégâts.
- **powerfulShoot** qui projette une flèche surpuissante infligeant 45 dégâts à un ennemi situé à 1 ou 2 cases de distance. Cette attaque consomme 5 points d'actions.

Quant à la classe DEMON, elle dispose des sorts suivants :

- **Strike** qui consiste à trancher l'ennemi à l'aide de la faux de Demon. Il s'agit d'une attaque de proximité avec une portée de 2 cases. Elle consomme 3 points d'actions et inflige 20 dégâts.
- **powerfulStrike** un coup dévastateur au corps à corps, consommant 6 points d'actions pour infliger 50 dégâts à son adversaire.

Calcul des dégâts :

Les dégâts infligés par les joueurs sont calculés de la manière suivante :

$$\text{dégatsInfligés} = \text{shieldJoueurAttaqué} - \text{dégatsAttaque} - \text{attackJoueurAttaquant}$$

A noter que les dégâts de l'attaque sont multipliés par **1.5** si l'attaque est réalisée dans le **dos** de l'ennemi et de **1.25** si elle est réalisée sur l'un de ses **flancs**.

Modes de jeu :

Nous souhaitons implémenter les modes de combats suivants :

- **JOUEUR VS JOUEUR** : Nous commencerons par permettre à deux joueurs réels de pouvoir s'affronter, puis si le temps nous le permet nous feront en sorte que les combats puissent opposer plus de deux joueurs.

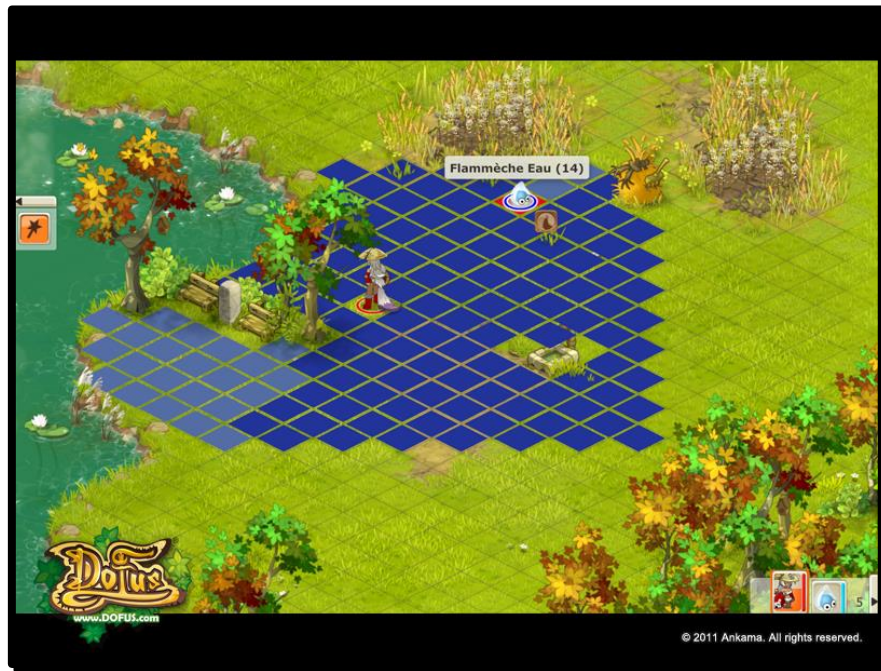
et

- **JOUEUR VS IA** : De même dans un premier temps ce mode opposera qu'un joueur à une IA ou une IA à une autre IA mais à terme nous souhaitons permettre des combats mixtes et plus conséquents, où IA et joueur réels pourront faire partie d'une même équipe.

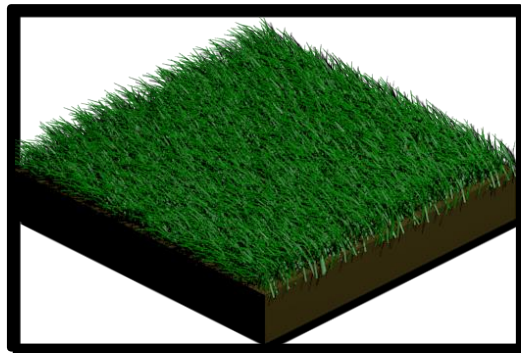
1.3 Conception Logiciel

- **Archetype (DOFUS):**





- Isometric grass :



Source : <https://opengameart.org/content/isometric-grass-tile>

Pour le sol de notre map nous pensons utiliser une texture comme celle ci-dessus. Elle constituera la base graphique sur laquelle nous ajouterons les autres éléments.

Pour rester dans le thème original de Dofus, nous avons choisi des bâtiments médiévaux/fantasy en 3D :

- **Medieval Building 03 :**



Source : <https://opengameart.org/users/bleed>

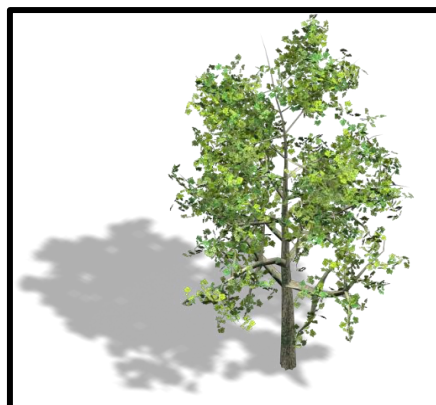
- **Timbered House_16 :**



Source : <https://opengameart.org/users/bleed>

Nous avons sélectionné un peu de verdure avec différents types d'arbres :

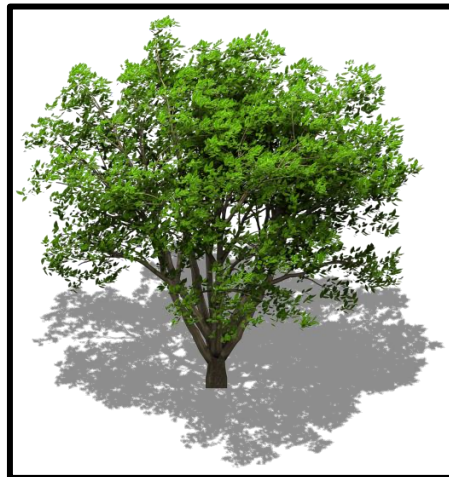
- **_01 :**



Source : <https://opengameart.org/users/bleed>

Projet Logiciel Transversal – Nouhou KANE – Mustafa KARADAG – Alexandre LOUIS

- **Animated Swietenia :**



Source : <https://opengameart.org/users/bleed>

Et des éléments plus décoratifs mais qui serviront eux aussi d'obstacles comme :

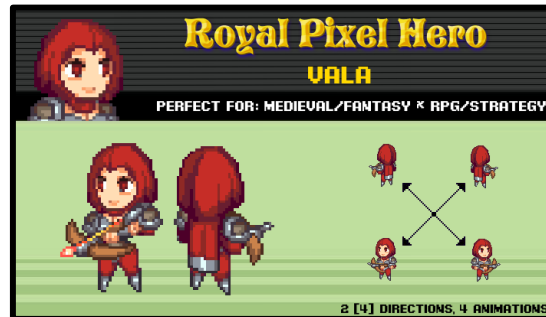
- **Well:**



Source : <https://opengameart.org/users/bleed>

Concernant les personnages, pour le moment nous sommes partis sur 2 possibilités :

- **Vala (HERO) :**



Source : <https://www.gamedevmarket.net/member/badim/>

- **Demon (DEMON) :**



Source : <https://www.gamedevmarket.net/member/badim/>

Le joueur ayant la possibilité de changer l'orientation de son personnage, nous avons recherché des personnages ayant des sprites dans les 4 directions.

2 Description et conception des états

2.1 Description des états

Notre projet comporte plusieurs classes. Chaque classe décrit un état ou un objet représentatif précis. L'état du jeu à un instant t est géré par une classe que l'on appelle State. Cette classe enregistre des informations telles que les joueurs de la partie (Héros, ennemis), le nombre de tours de jeu (le nombre de fois qu'on est passé d'un joueur à un autre), les identifiants de tous les joueurs et celui du joueur actuel, le temps de jeu passé et l'état de la map du jeu etc.

Projet Logiciel Transversal – Nouhou KANE – Mustafa KARADAG – Alexandre LOUIS

Chaque joueur est un objet de la classe Player identifié par un nom unique (attribué à l'exécution selon le choix du joueur {Héros} ou de manière automatique {ennemi}). Chaque joueur possède ses statistiques (points de vie, points de mouvement, niveau etc.), ses attaques (type et puissance d'attaque etc.) son orientation, sa position, son statut (mort, en train de jouer etc.).

Déroulé de jeu

La map est générée, les joueurs sont initialisés, chaque joueur a sa classe, l'ordre de jeu est défini et le combat peut commencer. Tuer la totalité des adversaires permet de remporter le combat. On meurt lorsque nos points de vie (*hp*) atteignent 0. Les joueurs se déplacent et effectuent des attaques avec les *ap* et *mp* qui leurs sont associés. Ils devront faire preuve de tactique pour infliger plus de dégâts que l'adversaire.

Les joueurs peuvent se déplacer sur toutes les cases libres, et non sur les obstacles et les cases occupées par une autre entité.

Les joueurs peuvent attaquer sur toutes les cases libres et les cases occupées et non sur les obstacles. Les obstacles obstruent la vue donc un joueur ne peut pas forcément attaquer une entité se trouvant derrière.

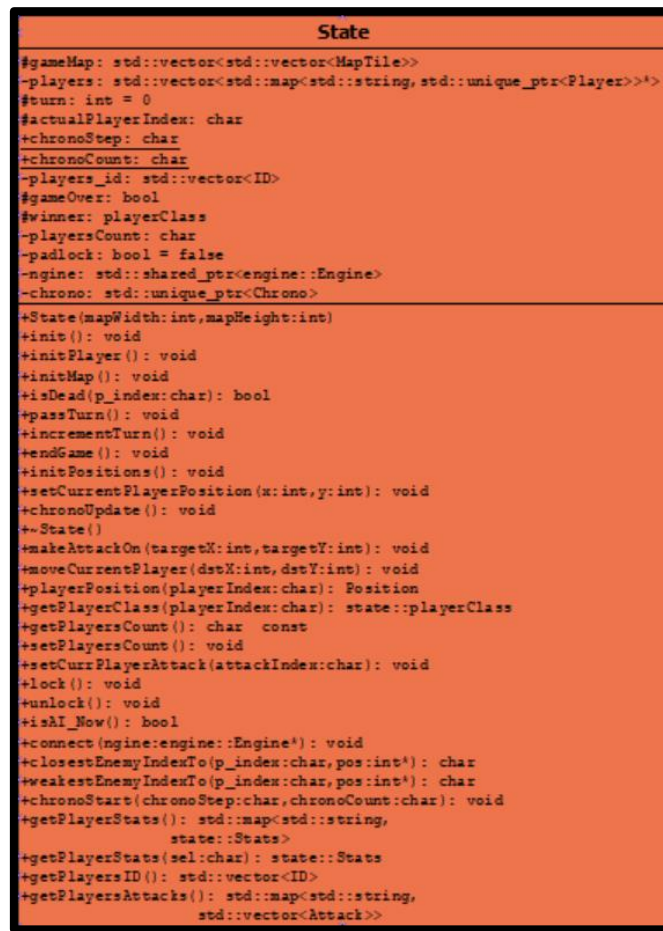
La fin de combat octroie des récompenses (augmentation de niveau du personnage)

2.2 Conception logiciel

Remarque : Les **getters et setters** des variables définies en tant que **protected** (précédées du symbole #) sont **générés automatiquement** lors de la compilation. Ils ne sont donc pas indiqués sur le diagramme de classe mais seront bien présents dans les .h de notre code.

Le diagramme d'état est composé de 6 classes, 2 structures ,8 énumérations et une interface. **State** et **Player** sont les 2 classes principales.

State

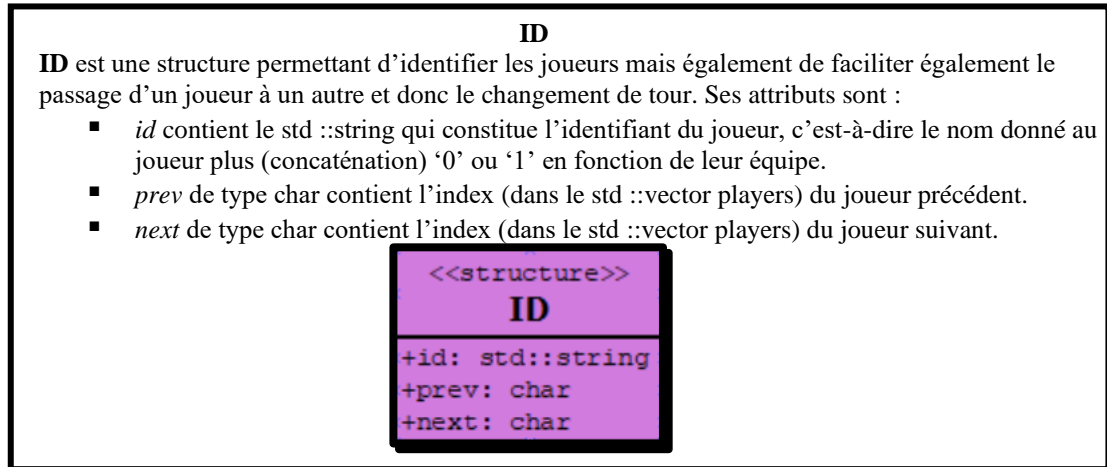


State est la classe principale du diagramme d'État. Elle permet de définir l'état dans lequel le jeu se trouve à tout instant et permet d'initialiser ces derniers. Dans cette classe, on trouve les vecteur *heroes* et *enemies* dans lesquels on liste les types de héros et ennemis.

Elle possède 13 attributs :

- *gameMap* de type `std::vector<std::vector<MapTile>>` est un vecteur de vecteur de **MapTile** qui va permettre de définir chaque "tiles" ou "cases" de jeu pour obtenir une cartographie entière de notre zone de jeu. (vecteur de vecteur = les cases doivent être vues comme constituant un plateau, comparable à un échiquier, où l'on peut identifier une case par un couple (x,y))
- *players* de type `std::vector<std::map<std::string, std::unique_ptr<Player>>*>` contient l'ensemble des joueurs de la partie. Les éléments du vecteur sont des `std::map` entre un `std::string`, le nom/identifiant du joueur et un `std::unique_ptr<Player>` qui pointe sur ce joueur.
- *turn* de type `int`, contient le numéro du tour de jeu actuel.
- *actualPlayerIndex* de type `char` contient l'index (dans le `std::vector players`) du joueur dont c'est le tour.
- *chronoStep* de type `char` est le pas temporel d'incrémentation du chrono (si *chronoStep* vaut 1, *chronoCount* sera décrémenté toutes les secondes)

- *chronoCount* de type `char` constitue le temps alloué à chaque joueur lors de leur tour.
- *players_id* de type `std::vector<ID>` est la liste des ID des joueurs.



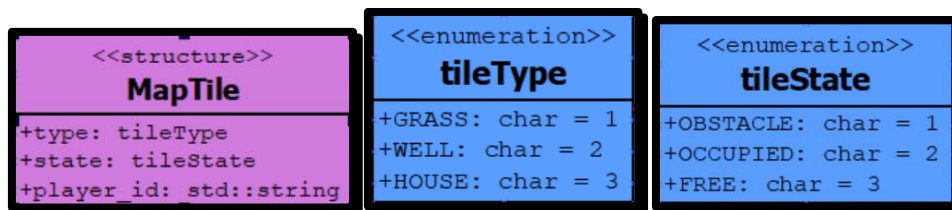
- *gameOver*, est un booléen mis à `false` initialement et qui passe à `true` quand le jeu est terminé, c'est-à-dire quand l'un des 2 joueurs n'a plus de HP.
- *winner*, de type `playerClass` contient la classe du joueur vainqueur (DEMON ou HERO). Cette variable est primordiale pour afficher la scène de fin appropriée.
- *playersCount* est un `char` contenant le nombre de joueurs.
- *padlock* est un booléen initialisé à `false` permettant d'indiquer à l'engine de ne pas exécuter de commande quand le render est en train de faire un update.
- *ngine* de type `std::shared_ptr<engine::Engine>` pointe sur l'engine afin qu'il puisse exécuter les actions requises.
- *chrono* de type `std::unique_ptr<Chrono>` pointe sur l'objet Chrono l'horloge globale de notre jeu.

En plus de son constructeur et son destructeur, elle possède 28 méthodes :

- *init()* pour initialise le state au lancement du jeu.
- *initPlayer()* pour créer les joueurs (objets `Player`) au lancement du jeu.
- *initMap()* pour charger la carte en début de partie.
- *initPositions()* initialise la position des joueurs en début de partie.
- *isDead()* vérifie les points de vie des joueurs, si un des joueurs n'a plus de vie, la fonction repère le vainqueur et appelle la fonction *endGame()*.
- *endGame()*, cette fonction entame la fin du jeu, elle passe notamment la variable *gameOver* à `true`.
- *passTurn()*, permet de passer le tour du joueur actuel et donner la main au joueur suivant.
- *incrementTurn()*, incrémente la valeur *turn* quand on passe un tour.
- *setCurrentPlayerPosition()* permet de modifier la position du joueur dont c'est le tour.
- *chronoUpdate()* met à jour la valeur de la variable *chrono* en fonction de la valeur de *chronoStep*.
- *makeAttackOn()* permet durant le tour d'un joueur de lancer une attaque à une position passée en argument. L'attaque n'aboutira que si un joueur se trouve à cette position.
- *moveCurrentPlayer()* permet à un joueur de se déplacer à une position passée en argument.
- *playerPosition()*, cette fonction retourne la position du joueur dont l'index a été passé en argument.
- *getPlayerClass()*, cette fonction retourne la classe du joueur dont l'index a été passé en argument.
- *getPlayersCount()* permet d'accéder au nombre de joueurs.
- *setPlayersCount()* permet de changer la valeur de *playersCount*, le nombre de joueurs.
- *setCurrPlayerAttack()* permet de changer l'attaque sélectionnée/actuelle du joueur dont c'est le tour.
- *lock()* passe la variable *padlock* à `true`.
- *unlock()* passe la variable *padlock* à `false`.
- *isAI_Now()* permet de savoir si le joueur dont c'est le tour est une IA ou non.
- *connect()* permet d'initialiser la variable *ngine* avec l'engine du jeu.
- *closestEnemyIndexTo()* renvoie l'index de l'ennemi le plus proche.
- *weakestEnemyIndexTo()* renvoie l'index de l'ennemi le plus faible.
- *chronoStart()* démarre l'horloge globale (lance le *chrono*).

- getPlayerStats()(avec ou sans paramètres) permet d'obtenir une partie ou toutes les stats d'une partie ou tous les joueurs.
- getPlayersID() permet d'obtenir un std::vector contenant les structures ID des différents joueurs.
- getPlayerAttacks() permet d'obtenir un std::map<std::string, std::vector<Attack>> qui associe à l'identifiant du joueur (nom concaténé à '0' ou '1') à sa liste d'attaques.

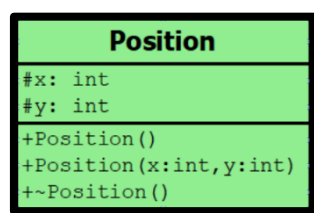
La structure **MapTile** associée à **State** va permettre de décrire chacune de ces cases à l'aide de ses attributs *type* et *state* :
 -type = choix du type de case (de l'herbe, un puit, une maison) dans **tileType**
 -state = disponibilité de la case (libre, occupée ou obstacle) dans **tileState**.
 Elle possède également un attribut *player_id* qui permet d'identifier le joueur présent sur une case quand elle est « OCCUPIED ».



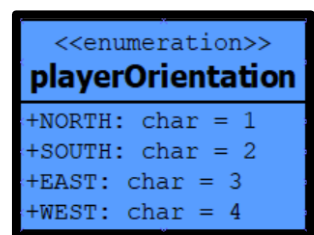
Player

La classe **Player**, comme son nom l'indique, va gérer tout ce qui concerne un joueur. Elle possède 11 attributs :

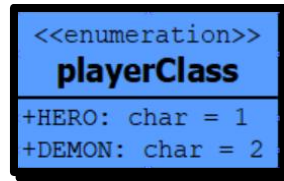
- *name*, un std::string qui correspond au nom choisi pour son personnage.
- *playing* est un booléen permettant d'indiquer si c'est le tour du joueur en question ou non, s'il est en train de jouer.
- *level* de type int contient le niveau du joueur, il peut varier entre 1 et 100. Le niveau du joueur correspond à son avancement dans le jeu. Cela affecte ses stats et les dégâts qu'il peut infliger avec ses attaques.
- la variable *isAi* va nous permettre de faire la distinction entre les joueurs réels (*isAi = false*) et les intelligences artificielles (*isAi = true*).
- *currentAttackIndex* contient l'index de l'attaque courante, c'est-à-dire l'attaque sélectionnée par le joueur quand il s'apprête à attaquer un ennemi.
- sa *position* : de type **Position** est un objet ayant deux attributs entiers x et y, correspondant aux coordonnées du joueur sur la map.



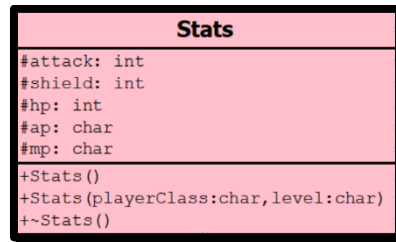
- son *orientation* : Le personnage peut être tourné vers le nord, le sud, l'est ou l'ouest. On retrouve ces valeurs dans l'énumération associée **playerOrientation**.



- sa *pClass* : Le joueur a le choix entre jouer un *Hero* ou un *Demon*. On les retrouve dans l'énumération associée **playerClass**.



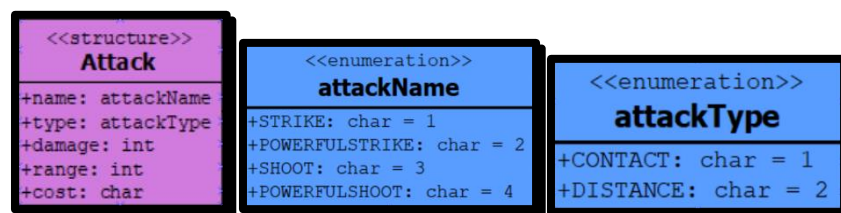
- ses stats (caractéristiques liées au combat): Définies par la classe **Stats**. Le joueur dispose de *mp* (movement point) consommables pour se déplacer ; de *ap* (action point) consommables pour effectuer une action comme une attaque ; de *hp* (health point = points de vie) représentant son niveau santé, s'ils tombent à 0 le joueur est vaincu ; de *shield* (bouclier) qui réduit les dégâts subis ; d'*attack* qui augmente les dégâts infligés par ses attaques.



- son status en combat : Indique si le joueur joue, s'il est en attente de son tour de jeu ou s'il est mort. On les retrouve dans l'énumération associée **playerStatus**.



- ses *attacks* en combat : Chaque attaque est une structure définie par son *cost* (le nombre de points d'action nécessaires pour l'utiliser), ses *damage* (le nombre de point de vie retiré à l'adversaire), son *range* (la distance maximale à laquelle le joueur peut lancer son attaque), son *type* (il peut s'agir d'une attaque à distance ou de contact, cf **attackType**) et son *name* (nom de l'attaque, cf **attackName**).



En plus de son constructeur et son destructeur, elle possède 8 méthodes :

- `init()` pour initialiser les différents aspect du joueur en début de partie.
- `attack()`, cette méthode prend en argument un `std::unique_ptr` pointant sur le joueur à attaquer. Elle vérifie que l'attaquant puisse attaquer (qu'il possède suffisamment de points d'action) et mettre à jour les stats du joueur attaqué (diminuer ses points de vie) mais également celles du joueur attaquant (diminuer ses AP).
- `pass()` permet de passer le tour du joueur. Elle agit en réalité sur les variables *playing* (`<=false`) et *status* (`<=WAITING`) du joueur.
- Les deux méthodes `move()` permettent de déplacer le joueur d'une position à un autre. Il s'agit en réalité de mettre à jour la position du joueur avec celle passée en paramètres. Tout comme pour `attack()`, la fonction vérifie que le joueur a suffisamment de points de mouvement pour se déplacer avant d'effectuer le changement.
- `changeOrientation()` permet de modifier l'orientation du joueur selon l'orientation désirée passée en argument.

- setPosXY() permet de modifier les coordonnées (x,y) du joueur.
- resetPoints() permet de restituer les points de mouvement (MP) et d'action (AP) du joueur à chaque fois que son tour commence.

```

class Player
{
#name: std::string
#pClass: playerClass
#position: Position
#orientation: char
#playing: bool
#attacks: std::vector<Attack>
#stats: Stats
#status: playerStatus
#level: int
#isAI: bool = false
#currentAttackIndex: char

+Player()
+Player(name:std::string,pClass:playerClass,
        position:Position,level:int,isAI:bool)
+init(): void
+attack(player:std::unique_ptr<Player> &): void
+pass(): void
+move(destination:Position): void
+move(x:int,y:int): void
+changeOrientation(orientation:playerOrientation): void
+setPosXY(x:int,y:int): void
+resetPoints(): void
+~Player()

```

2.3 Conception logiciel : extension pour le rendu

Pour cette partie nous avons rajouté dans la classe State, un système de verrou à l'aide de la variable padlock pour permettre une attente de la part du moteur quand le rendu doit être mis à jour. Le fonctionnement est le suivant :

1. Lorsqu'une commande est générée, elle est enregistrée dans l'engine.
2. Si le padlock est ouvert, l'engine exécute la commande puis verrouille le padlock (lock()). L'engine attend alors que l'update du render ait lieu.
3. Le render actualise le rendu, déverrouille le padlock (unlock()) et provoque l'exécution de la prochaine commande disponible.

2.4 Conception logiciel : extension pour le moteur de jeu

Chrono

Cette classe est celle destinée à surveiller le temps passé et prévenir l'engine toutes les 60 secondes pour passer le tour du joueur.

Elle ne possède pas d'attribut et a trois méthodes en plus de son constructeur et son destructeur :

- bind(), lie le handler à l'alarme, de sorte que, quand il y a une interruption d'alarme, le handler est exécuté.
- handler(), met à jour le chrono quand le timer a fini de compter.
- start(), qui lance le comptage du temps.

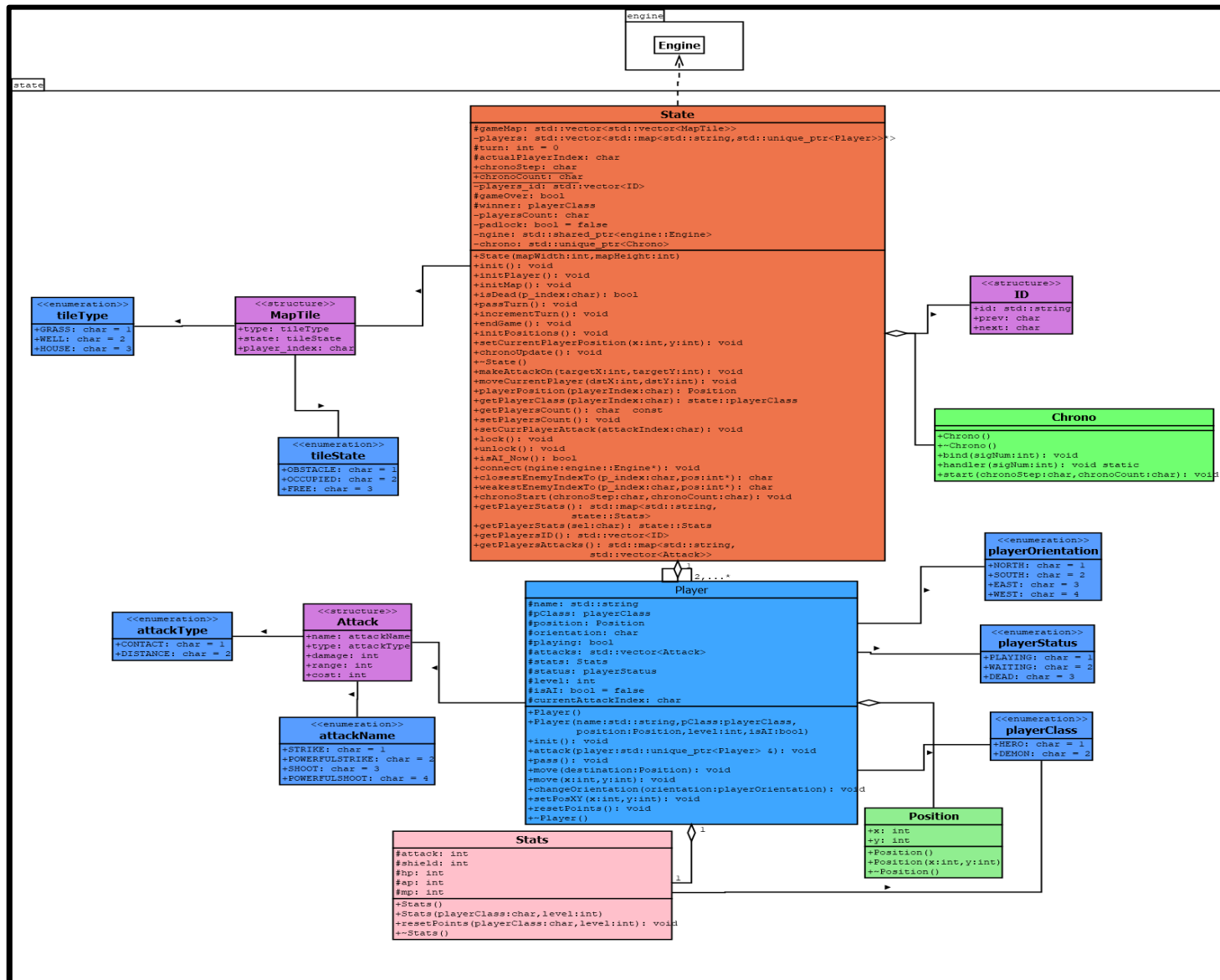
```

class Chrono
{
+Chrono()
+~Chrono()
+bind(sigNum:int): void
+handler(sigNum:int): void static
+start(count:int): void

```

2.5 **Ressources**

Illustration 1: Diagramme des classes d'état



3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Nous avons conçu une map de 20 tiles sur 20 tiles, avec plusieurs dizaines d'obstacles. L'espace de combat n'est donc pas trop vaste et permet la mise en place de stratégie (se protéger d'une attaque en bloquant la ligne de vue de l'adversaire grâce à un mur par exemple). Nous avons souhaité nous rapprocher le plus possible de l'atmosphère de Dofus, par conséquent au-delà des textures médiévales, nous avons opté pour une map avec une orientation isométrique pour un effet de jeu en semi-3D. Toutes les données de notre map sont stockées sous forme d'un fichier tmx. Nous l'avons imaginée à l'aide du logiciel Tiled avant de l'exporter. Grâce au render SFML nous pouvons ensuite charger notre map mais également la modifier au fur à mesure de la partie. Pour chaque état nous avons 3 éléments à prendre en compte et à afficher :

- La map en tant que plateau de tuiles, surface statique sur laquelle les personnages se déplacent.
- Les personnages en animation constante grâce à la succession rapide de sprites et qui ont également la possibilité de changer d'emplacement sur la map. Nous envisageons également une animation d'attaque si possible.
- Et enfin les informations de jeu tels que les points de vie, d'actions et de mouvements ou le temps restant au joueur, mais aussi les statistiques des personnages ou les options de jeu.

Lorsque l'état de jeu est modifié dans le cas du déplacement d'un personnage par exemple nous allons venir modifier notre variable `state::map` et notre `render` va actualiser l'affichage de la map en conséquence.

3.2 Conception logiciel

Nous avons une nouvelle fois réalisé un diagramme à l'aide du logiciel Dia pour décrire l'architecture de notre rendu. Le diagramme se compose de 7 classes : **GameWindow**, **Scene**, **FightScene**, **Box**, **Info**, **Button**, et **AnimatedObject**; 3 énumérations : **buttonState**, **buttonStyle** et **SceneId**; enfin elle possède 1 union **TextOrTexture** et un structure **TextureSprite**.

GameWindow

Cette classe est celle destinée à générer et gérer la fenêtre de jeu.

Elle possède 12 attributs :

- *window* de type `sf::RenderWindow` va permettre de générer à la fenêtre globale dans laquelle sera contenue tous les autres éléments.
- *zoom* de type `float` est une valeur réelle permettant d'ajuster l'agrandissement du contenu de la fenêtre.*
- *zoomTot* de type `float`, contient le produit de tous les zooms (> 1 ou < 1) réalisées depuis le lancement du jeu. Elle permet de conserver le focus souhaité quand on passe d'une scène à un autre (notamment de FIGHTSCENE à END).
- *bgText* de type `sf::RenderTexture` qui correspond à la texture du background de notre map, c'est-à-dire la partie statique de notre map sur lequel évolueront les personnages.
- *background* la sprite à laquelle on va associer la texture précédente.
- *width* de type `uint` qui correspond à la largeur de la fenêtre.
- *height* de type `uint` qui correspond à la hauteur de la fenêtre.
- *scenes* regroupe les pointeurs sur de 3 objets **Scène**, qui sont les 3 pages qui susceptibles d'être affichées après le lancement du jeu : le menu/page d'accueil, la page de combat, la page de victoire ou de défaite.
- *currentScene* qui correspond à l'ID de la scène actuelle (MENU, FIGHTSCENE ou END).
- *selected* est un `char` contenant le code unique associé à un bouton de l'UI. Cette variable permet tout simplement de savoir sur quel bouton le joueur a cliqué.
- *isZoomed* un booléen permettant de savoir si le joueur a effectué un zoom/dézoom ou non.

En plus de son constructeur et son destructeur, elle possède 14 méthodes :

- `init()` pour initialise l'attribut *window* et donc la fenêtre qui sera affichée.
- `initScenes()` pour créer et initialiser les 3 scènes requises pour le menu, le combat et la fin de partie.
- `draw()` pour afficher la fenêtre avec les paramètres désirés ainsi que la scène actuelle.

- `clearBackground()` pour effacer le contenu de la fenêtre.
 - `update()` avec et sans paramètres permet d'actualiser le contenu de la fenêtre. La version avec paramètres permet de transmettre des informations tel que la position du clic ou le type d'évènement.
 - `shareStateWith()` permet de partager le state avec l'engine.
 - `nextScene()` permet de charger à l'écran la scène suivante. On change de scène dans la fenêtre.
 - `screenToWorld()` permet de passer des coordonnées orthogonales en pixel de la fenêtre aux coordonnées de la map (cases) du jeu.
 - `worldToScreen()` fait l'opération inverse.
 - `setZoom()`, comme son nom l'indique permet de modifier le zoom du rendu visuel.
 - `setCenter()`, permet de recentrer la vue à la position désirée.
- (l'association de ces deux dernières fonctions permet ainsi de se déplacer comme on le souhaite sur la map)
- `handleEvents()`, écoute les actions du joueur (clic, appui sur le clavier, mouvement de souris, etc..) et conduit au déroulement adéquat en appelant les fonctions appropriées.
 - `handleZoom()`, fonctionne de manière similaire à `handleEvents`, cependant cette fonction n'écoute que la molette de la souris, associée à l' action de zoom/dézoom.

```

GameWindow

#scenes: std::vector<std::unique_ptr<Scene>>
+view: sf::View
+zoom: float
+zoomTot: float
+window: sf::RenderWindow
+bgTex: sf::Texture
#background: sf::Sprite
+width: uint = 2000u
+height: uint = 600u
#currentScene: SceneId
+selected: char
+isZoomed: bool

+GameWindow()
+~GameWindow()
+update(): void
+update(e:sf::Events,m_mousePosition:sf::Vector2i): void
+init(): void
+initScenes(): void
+draw(): void
+clearBackground(): void
+shareStateWith(engine:engine::Engines): void
+nextScene(): void
+screenToWorld(position:sf::Vector2f): sf::Vector2f
+worldToScreen(position:sf::Vector2f): sf::Vector2f
+setZoom(zoom:float): void
+setCenter(center:sf::Vector2f): void
+handleEvents(event:sf::Events,mousePosScreen:sf::Vector2fs,
              mousePosWorld:sf::Vector2fs,
              engine:engine::Engines): void
+handleZoom(event:sf::Events,mousePosScreen:sf::Vector2fs): void

```

SceneId

Cette enumeration, comme nous l'avons mentionné à l'instant, permet de distinguer les différentes scènes à afficher dans le cadre de notre jeu.

Elle contient 3 valeurs :

- *MENU* pour la page d'accueil au lancement de jeu.
- *FIGHTSCENE*, la scène principale dans laquelle aura lieu les combats.
- *END* pour indiquer la victoire ou la défaite au joueur et lui permettre de relancer un combat.

```

<<enumeration>>
SceneId

+MENU: char = 0
+FIGHTSCENE: char = 1
+END: char = 2

```

Scene

Cette classe est celle destinée à générer et gérer les scènes, c'est-à-dire les différentes pages du jeu.

Elle possède 6 attributs :

- *boxes* qui est un ensemble de pointeurs sur des objets **Box** qui sont les formes et boutons ou simplement les textes qui seront affichés pour une scène donnée.
- *id* de type int est une valeur unique permettant de distinguer les différentes scènes.
- *texture* de type `sf::Texture` qui correspond à la texture utiliser pour une Scene donnée.
- *sprite* la sprite à laquelle on va associer la texture précédente.
- *gameOver*, est un booléen mis à false initialement et qui passe à true quand le jeu est terminé, c'est-à-dire quand l'un des 2 joueurs n'a plus de HP.
- *winner*, de type `playerClass` contient la classe du joueur vainqueur (DEMON ou HERO). Cette variable est primordiale pour afficher la scène de fin appropriée.

En plus de ses constructeurs et son destructeur, elle possède 13 méthodes :

- `init()` pour initialiser la scène.
- `initButtons()` permet de créer les boutons qui seront affichés pour une scène donnée. On indique notamment leur taille, leur position, leur contenu et leur couleur.
- `draw()` pour dessiner et afficher à l'écran les éléments de la scène.
- `loadImages()` pour charger les images qui vont être affichées.
- `loadTextures()` pour charger les texture des éléments à afficher.
- `update()` avec et sans paramètres permet d'actualiser le contenu de la fenêtre. La version avec paramètres permet de transmettre des informations tel que la position du clic ou le type d'évènement.
- `bindState()` pour partager le state avec les scènes.
- `screenToWorld()` permet de passer des coordonnées orthogonales en pixel de la fenêtre aux coordonnées de la map (cases) du jeu.
- `worldToScreen()` fait l'opération inverse.
- `isGameOver()` permet de connaître l'état du jeu, s'il y a Game Over, on pourra mettre à jour la variable `gameOver`.
- `whosWinner()` permet de connaître la classe du vainqueur. Cette fonction est appelée si `isGameOver()` renvoie true.
- `setTexture()` est utilisée afin de définir le background d'une scène.

```
Scene
#boxes: std::map<std::string, std::unique_ptr<Box>>
#id: char
#texture: sf::Texture
#sprite: sf::Sprite
#gameOver: bool = false
#winner: state::playerClass
+Scene()
+Scene(id: char, type: std::string, gameWindow: GameWindow*)
+~Scene()
+draw(target: sf::RenderTarget, states: sf::RenderStates): void const
+loadImages(): void
+loadTextures(textures_paths: std::string): void
+init(type: std::string, gameWindow: GameWindow*): void
+initButtons(): void
+bindState(engine: engine::Engine): void
+screenToWorld(position: sf::Vector2f): sf::Vector2f
+worldToScreen(position: state::Position): sf::Vector2f
+update(): void
+update(e: sf::Events, m_mousePosition: sf::Vector2i,
        gameWindow: GameWindow*): void
+isGameOver(): bool
+whosWinner(): state::playerClass
+setTexture(stexture: std::string): void
```

Box

Cette classe est celle destinée à générer et gérer les boutons, formes, informations qui seront affichés sur les différentes scènes.

Elle possède 5 attributs :

- *m_position* de type `sf::Vector2i` qui contient les coordonnées de l'objet **Box** afin de pouvoir le dessiner à l'endroit souhaité via la méthode `draw()`.

- *dimensions* de type `sf::Vector2f` est un couple qui contiendra la longueur et la largeur de la Box à afficher.
- *m_text*, est le `sf::Text` que l'on souhaite afficher à l'écran.
- *content* de type `TextOrTexture` qui correspond à la texture ou le texte utilisé par la Box.
- *sprite*, de type `sf::Sprite` qui correspond à la sprite qui va contenir la Box.

En plus de son constructeur et son destructeur, elle possède 3 méthodes :

- `draw()` pour afficher la Box sur la scène à la position souhaitée.
- `update()` pour mettre à jour l'affichage de la scène.
- `setText()` permet de mettre à jour *m_text* et ainsi modifier le texte contenu par la box.

```

Box

#m_position: sf::Vector2i
#dimensions: sf::Vector2f
#m_text: sf::Text
#content: TextOrTexture
#sprite: sf::Sprite

+Box()
~Box()
-draw(target:sf::RenderTarget, states:sf::RenderStates): void const
+update(e:sf::Event, m_mousePosition:sf::Vector2i,
        gameWindow:GameWindow): void
+setText(s:std::string): void

```

TextOrTexture

TextOrTexture est une union qui permet de choisir entre une texture et un texte pour le contenu d'une Box donnée. Cela permet de ne pas avoir deux données en mémoire.

```

<<union>>
TextOrTexture

+text: sf::Text
+ts: struct TextureSprite

```

TextureSprite

Il s'agit d'une structure ayant 2 attributs *texture* et *sprite*, on a ainsi une texture et la sprite qui lui est associée.

```

<<structure>>
TextureSprite

+texture: sf::Texture
+sprite: sf::Sprite

```

Button

Cette classe est celle destinée à générer et gérer les scènes, c'est-à-dire les différentes pages du jeu.

Elle possède 17 attributs :

- *m_btnState* de type `buttonState` indiquant l'état du bouton (cliqué, survolé, normal).
- *m_bgNormal* de type `sf::Color` qui indique la couleur du background quand le bouton est dans l'état normal.
- *m_bgHover* de type `sf::Color` qui indique la couleur du background quand le bouton est dans l'état survolé.
- *m_bgClicked* de type `sf::Color` qui indique la couleur du background quand le bouton est dans l'état cliqué.
- *m_textNormal* de type `sf::Color` qui indique la couleur du text quand le bouton est dans l'état normal.
- *m_textClicked* de type `sf::Color` qui indique la couleur du text quand le bouton est dans l'état cliqué.
- *m_textHover* de type `sf::Color` qui indique la couleur du text quand le bouton est dans l'état survolé.
- *m_border* de type `sf::Color` qui indique la couleur de la bordure/ le contour du bouton.

- *m_borderThickness* de type float qui quantifie l'épaisseur de la bordure.
- *m_borderRadius* de type float qui permet de quantifier le degré d'arrondissement des coins du bouton.
- *m_size* de type sf::Vector2f qui correspond au couple longueur/largeur caractérisant les dimensions du bouton.
- *m_style* de type buttonStyle qui indique le type de style à appliquer au bouton.
- *m_button* de type sf::ConvexShape qui définit la forme du bouton.
- *m_fontSize* de type unsigned int qui indique la taille de la police de texte.
- *m_font* de type sf::Font qui indique la police de texte à utiliser.
- *m_shadow* de type sf::Text qui représente l'ombre du texte.
- Enfin, *m_type*, un char correspondant à un code unique permettant de distinguer les différents boutons.

En plus de ses constructeurs et son destructeur, elle possède 1 constructeur à paramètres ainsi que 2 méthodes :

- draw() pour dessiner le bouton.
- update() pour réactualiser le bouton.

```

Button

#m_btnState: enum buttonState
#m_bgNormal: sf::Color
#m_bgHover: sf::Color
#m_bgClicked: sf::Color
#m_textNormal: sf::Color
#m_textHover: sf::Color
#m_textClicked: sf::Color
#m_border: sf::Color
#m_borderThickness: float
#m_borderRadius: float
#m_size: sf::Vector2f
#m_style: enum buttonStyle
#m_button: sf::ConvexShape
#m_fontSize: unsigned int
#m_font: sf::Font
#m_shadow: sf::Text
#m_type: char = false

+Button()
+Button(s:std::string,m_fontSize:unsigned int,
        font:sf::Font,size:sf::Vector2f,
        position:sf::Vector2i,style:enum buttonStyle,
        m_type:char,gameWindow:GameWindow*)
+~Button()
-draw(target:sf::RenderTarget&,states:sf::RenderStates): void const
+update(e:sf::Event&,m_mousePosition:sf::Vector2i ,
        gameWindow:GameWindow*): void

```

buttonState

Il s'agit d'une énumération qui contient 3 valeurs de type char *CLICKED* (cliqué), *HOVERED* (survolé) et *NORMAL* (normal) représentant les différents états que peut prendre un bouton.

```

<<enumeration>>
buttonState

+CLICKED: char = 0
+HOVERED: char = 1
+NORMAL: char = 2

```

buttonStyle

Il s'agit d'une énumération qui contient 4 valeurs de type char *NONE* (aucun), *SAVE* (sauvegarder), *CANCEL* (annuler) et *CLEAN* (nettoyer) représentant les différents styles que peut prendre un bouton.

<<enumeration>> buttonStyle	
+NONE:	char = 0
+SAVE:	char = 1
+CANCEL:	char = 2
+CLEAN:	char = 3

Info

Cette classe est celle destinée à générer et gérer les scènes, c'est-à-dire les différentes pages du jeu. Elle possède 1 attributs :

- *m_font* de type `sf::Font` qui indique la police de texte à utiliser.

En plus de ses constructeur et son destructeur, elle possède 2 méthodes :

- `draw()` pour écrire l'info sur la scène.
- `update()` pour réactualiser l'info (le texte) affichée.

Info
<pre>#m_font: sf::Font +Info() +Info(s:std::string,font:sf::Font*,sizeFont:unsigned int, position:sf::Vector2i,gameWindow:GameWindow*) -draw(target:sf::RenderTarget*,states:sf::RenderStates): void const +update(e:sf::Event*,m_mousePosition:sf::Vector2i, gameWindow:GameWindow*): void +~Info()</pre>

FightScene

Cette classe héritant de la classe **Scene** est celle destinée à générer et gérer la scène de combat, elle s'occupera en autres de l'affichage de la map et des personnages.

Elle possède 9 attributs :

- *gState* de type `std::shared_ptr<state::State>` pointe sur l'état du jeu afin d'actualiser l'affichage au fur et à mesure du combat.
- *animatedObjectss* de type `std::vector<std::unique_ptr<AnimatedObject>>` qui est la liste des objets animés de la scène, c'est-à-dire les personnages.
- *framesInfo* de type `Json::Value` est fichier dans lequel sont inscrites les positions et les tailles des différentes frames des animations.
- *gameMap* sera chargée grâce à notre `MapLoader` et contient comme son nom l'indique notre map conçue sur `Tiled`.
- *moveVertex*, un `std::vector<sf::VertexArray>`, qui sera tracé afin d'afficher la zone de déplacement possible pour le joueur dont c'est le tour.
- *attackVertex*, un `std::vector<sf::VertexArray>`, qui permet de visualiser la portée d'une attaque. Le tracé est déclenché quand on clique sur un bouton de sort.
- *showVertex* est un booléen qui vaut `true` si le joueur décide d'afficher la portée de mouvement (appui sur la touche D) ou `false` dans le cas contraire.
- *playersStats* de type `std::map<std::string,state::Stats>`, contient les stats (HP, AP et MP) des joueurs afin de pouvoir afficher ces informations en haut de la fenêtre.
- *playersAttacks* de type `std::map<std::string,std::vector<state::Attack>>`, contient les attaques des joueurs afin de connaître la portée de ces dernières.

Elle possède 11 méthodes en plus de son constructeur et son destructeur.

- `draw()`, pour dessiner la scène dans la fenêtre.
- `update()` avec et sans paramètres permet d'actualiser le contenu de la fenêtre. La version avec paramètres permet de transmettre des informations tel que la position du clic ou le type d'évènement.
- `getFrameInfos()` et `loadFrameInfos()`, les getter et setter de `frameInfos`.
- `bindState()` pour partager le state avec la scène.
- `screenToWorld()` permet de passer des coordonnées orthogonales en pixel de la fenêtre aux coordonnées de la

map (cases) du jeu.

- worldToScreen() fait l'opération inverse.
- initButtons() permet de créer les boutons qui seront affichés pour une scène donnée. On indique notamment leur taille, leur position, leur contenu et leur couleur.
- moveRange() permet de tracer la portée de déplacement du joueur dont c'est le tour.
- attackRange() permet de tracer la portée de l'attaque qui a été sélectionnée par le joueur.

```
FightScene
-gState: std::shared_ptr<state::State>
#animatedObjects: std::vector<std::unique_ptr<AnimatedObject>>
-frameInfos: Json::Value
-gameMap: std::shared_ptr<tmx::MapLoader> = std::shared_ptr<tmx::MapLoader>(new tmx::MapLoader("res/map/"))
-moveVertex: std::vector<sf::VertexArray>
+attackVertex: std::vector<sf::VertexArray>
-showVertex: bool = false
-playersStats: std::map<std::string, state::Stats>
-playersAttacks: std::map<std::string, std::vector<state::Attack>>
+FightScene(id:char, gameWindow:GameWindow*)
+~FightScene()
+update(): void
+update(e:sf::Event&, m_mousePosition:sf::Vector2i,
        gameWindow:GameWindow*): void
-draw(target:sf::RenderTarget&, states:sf::RenderStates): void const
+getFrameInfos(): Json::Value&
+loadFrameInfos(path:std::string): void
+bindState(engine:engine::Engine&): void
+worldToScreen(position:state::Position): sf::Vector2f
+screenToWorld(position:sf::Vector2f): sf::Vector2f
+initButtons(gameWindow:GameWindow*): void
+moveRange(): void
+attackRange(selected:char): int
```

3.3 Conception logiciel : extension pour les animations

AnimatedObject

Cette classe est celle destinée à générer et gérer les objets animés.

Elle possède 4 attributs :

- *mSprite*, de type `sf::Sprite` qui correspond à la sprite de l'objet qui sera animé pour une frame donnée.
- *mCurrentFrame*, de type `sf::size_t` à la frame qu'il faut afficher et passer à *mSprite*.
- *mElapsedTime* de type `sf::Time` permet de savoir le temps qui a passé depuis le début de l'animation.
- *mRepeat* est un booléen qui nous indique si l'animation de l'objet doit être répétée ou non.

Elle possède deux méthodes en plus de son constructeur et son destructeur.

- *draw()*, pour dessiner l'animation (afficher successivement les différents frames).
- *update()*, qui permet de mettre à jour l'objet animé, en actualisant la frame à afficher et le temps écoulé.

```
AnimatedObject
#mSprite: sf::Sprite
#mCurrentFrame: std::size_t
#mElapsedTime: sf::Time
#mRepeat: bool
+AnimatedObject(frames:sf::Texture&)
+~AnimatedObject()
-draw(target:sf::RenderTarget&, states:sf::RenderStates): void const
+update(dt:sf::Time, framesInfos:Json::Value,
        frameKey:std::string, positions:sf::Vector2f): void
```

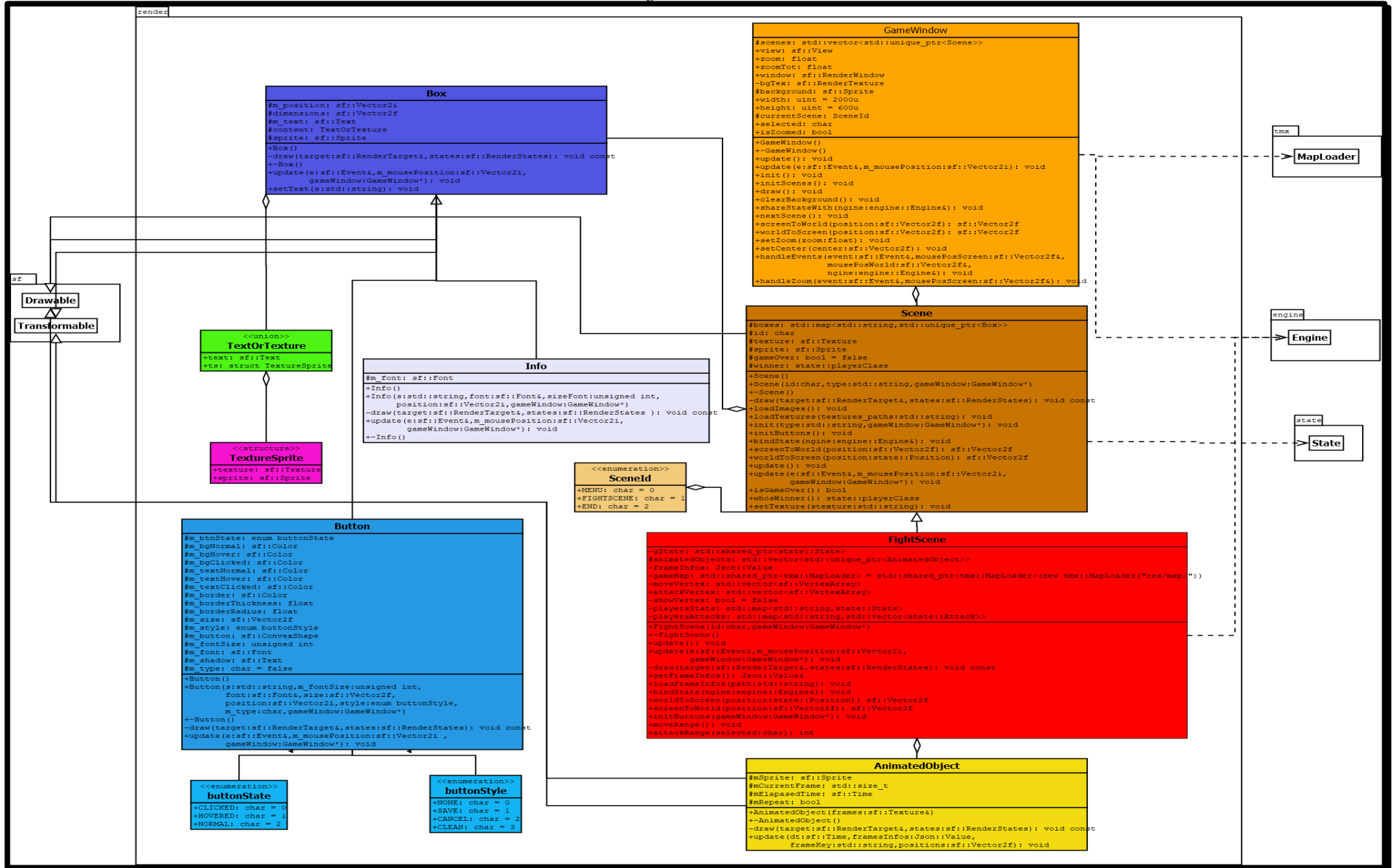
3.4 Ressources

3.5 Exemple de rendu



Exemple d'état de jeu

Illustration 1: Diagramme des classes du render



4 Règles de changement d'états et moteur de jeu

4.1 Horloge globale

Nous avons mis en place une horloge globale sous la forme d'un minuteur qui est décrémenté à pas d'une seconde (*chronoStep*) et qui à chaque début de tour est rétabli à la valeur *chronoCount* choisie.

```
Chrono
+Chrono()
+~Chrono()
+bind(sigNum:int): void
+handler(sigNum:int): void static
+start(count:int): void
```

Plusieurs évènements peuvent provoquer le passage d'un état à l'autre, nous les avons listés ci-dessous :

4.2 Changements extérieurs

- **Déplacement :**

Lors de son tour, comme nous l'avons vu, le joueur peut se déplacer d'une case à un autre en consommant des points de mouvements. Il ne peut pas se déplacer de plus de cases qu'il n'a de points de mouvements. Pour se déplacer, il peut ainsi cliquer sur les cases entourant son personnage dans un périmètre de maximum *nb_de_points_de_mouvement* cases. Lors d'un clic sur une case, nous récupérons alors la position du clic grâce aux fonctions disponibles dans la bibliothèque SFML. On vérifie que le joueur peut effectivement se déplacer à cette case et si c'est le cas on va générer une commande (**move**). Cette commande move va modifier la position du personnage concerné en agissant sur le state (*state::Player::position*). Il est donc essentiel que l'engine ait en permanence un accès au state afin de le mettre à jour. Enfin le render réactualisant sans cesse l'affichage, on observe le personnage se déplacer à l'écran.

- **Attaque :**

Durant son tour, le joueur peut également attaquer les ennemis auxquels il fait face en échange de points d'action. Il peut lancer un sort/une attaque uniquement si le coût du sort est inférieur au nombre de points d'action restants au joueur. Pour attaquer, il peut ainsi réaliser un premier clic pour choisir son attaque parmi les différents sorts, l'engine va alors modifier la variable *currentAttackIndex* du personnage attaquant dans le state, afin d'enregistrer le choix de l'attaque. Puis dans un second temps, il faudra réaliser un second clic, sur la case où l'on souhaite jeter le sort. Comme pour les déplacements, le sort ne peut être lancé que sur les cases entourant son personnage dans un périmètre de maximum *valeur_portee_attaque* cases. Lors d'un clic sur une case, nous récupérons alors la position du clic grâce aux fonctions disponibles dans la bibliothèque SFML. On regarde l'attribut *state* de la case et on agit de la manière suivante :

- S'il s'agit d'un obstacle ou que la case est vide, alors l'attaque est exécutée mais la seule conséquence est l'actualisation des points d'action du joueur. La commande **attack** est ainsi générée et va modifier dans le state le nombre de points d'actions du personnage concerné (*state::Player::stats.ap*).
- Si la case est occupée par un autre joueur alors dans ce cas en plus d'actualiser les points d'action du joueur attaquant, il faut mettre à jour les points de vie du joueur attaqué. Pour se faire, on va lire l'attribut *player_id* de la case, accéder aux joueurs correspondant dans notre liste de joueurs, et diminuer ses hp en tenant compte des stats offensives et défensives des deux joueurs.

4.3 Changements autonomes

5 Changement de tour :

Enfin, un changement d'état à évidemment lieu quand le joueur passe son tour (ou que les 60 secondes qui lui sont accordées sont écoulées). Si le joueur clique sur le bouton « Passer mon tour », on va générer la commande **passTurn**. Cette commande `passTurn` va modifier la variable `Player::playerStatus` à `WAITING` et `playing` passe à `false`. Ensuite, `State::actualPlayerIndex` est mis à jour pour donner la main au joueur suivant pour qui `Player::playerStatus` passera à `PLAYING` et `playing` passe à `true`.

De plus, lorsque les points de vie de l'un des joueurs tombent à 0, un changement d'état à également lieu. En effet, les points de vie des différents personnages étant surveillés continuellement, quand ils chutent à 0, la variable `Player::playerStatus` passe à `DEAD` et dans ce cas, la fonction `State::isDead()` retournera `true`, `State::gameOver` passera à `true` et le combat se terminera.

5.1 Conception logiciel

5.2 Conception logiciel : extension pour l'IA

Engine

Cette classe est celle destinée à représenter le moteur de jeu et contenir ses fonctionnalités.

Elle possède 9 attributs :

- *fuel* un mutex pour éviter que les threads n'interfèrent entre eux.
- *selected* est un char contenant le code unique associé à un bouton de l'UI. Cette variable permet tout simplement de savoir sur quel bouton le joueur a cliqué.
- *eMutex*, un mutex pour éviter que les threads n'interfèrent entre eux.
- *eThread*, le thread qui va contenir les tâches à réaliser.
- *currentState*, qui est un pointeur sur l'état actuel du jeu du package State.
- *cmdHolder*, permet de contenir une commande à exécuter.
- *qcmd* est une queue de commandes qui va contenir les commandes à exécuter. Elle se remplit et se vide continuellement.
- *action* de type `std::vector<void(*) (std::unique_ptr<Action_Args>&)>` est une liste de deux fonctions associées respectivement à l'action 'attaquer » et à l'action « se déplacer ». Une action dans ce contexte est un code dont le bit de poids fort est à 1.
- *selection* de type `std::vector<void(*) (std::unique_ptr<Action_Args>&)>` est la liste des fonctions à exécuter pour un clic sur un bouton donné. Chaque bouton de l'UI est ainsi lié à une fonction dans *selection* Une action dans ce contexte est un code dont le bit de poids fort est à 0.

Nous avons séparé les actions dans *action* de celles dans *selection* car celles dans *action* sont provoquées par un clic sur la map alors que celles dans *selection* sont provoquées par un clic sur un bouton de l'UI.

Elle possède 14 méthodes en plus de son constructeur et son destructeur.

- *init()*, pour correctement initialiser l'engine au lancement du jeu.
- *save()*, pour sauvegarder l'avancement du jeu (tour actuel, stats des joueurs, etc..)
- *setState()*, pour mettre à jour *currentState* et c'est-à-dire actualiser l'état de jeu dans l'engine.
- *resume()*, pour charger et reprendre une partie qui a été sauvegardée.
- *run()*, qui va mettre en route le moteur de jeu (écoute des actions des joueurs + mise à jour du state + exécution des IA).
- *addCommand()* afin d'ajouter une nouvelle commande à la queue actuelle.
- *start()* pour démarrer l'engine.
- *stop()* pour arrêter l'engine.
- *isActionFromAI()* pour vérifier si l'action provient d'une IA ou d'un joueur réel.
- *execute()* (avec ou sans arguments) permettent d'exécuter les commandes de la queue. *execute()* sans arguments exécute la commande en tête de file, *execute()* avec arguments prend en paramètre une commande et l'exécute.
- *registerTarget()* (avec ou sans arguments) permettent d'enregistrer les commandes dans la queue.
- *registerTarget()* permet d'avertir l'engine quand le temps alloué au tour d'un joueur est atteint.
- *bind()* permet de partager le state et soi même aux IA.

Engine
<pre> +fuel: bool -selected: char -eMutex: std::mutex -eThread: std::thread -currentState: std::shared_ptr<state::State> -cmdHolder: std::unique_ptr<Command> -qcmd: std::queue<std::unique_ptr<Command>> -action: std::vector<void(*) (std::unique_ptr<Action_Args>&)> -selection: std::vector<void(*) (std::unique_ptr<Action_Args>&)> +Engine() +init(): void +save(): void +setState(gState:std::shared_ptr<state::State>&): void +run(): void +~Engine() +addCommand(cmd:std::unique_ptr<Command>&): void +start(): void +stop(): void +isActionFromAI(): bool +execute(): void +execute(cmd:std::unique_ptr<Command>&): void +registerTarget(x:int,y:int,selected:char): void +registerTarget(selected:char): void +timeOut(): bool +bind(g ai:ai::AI*): void </pre>

Command

Cette classe est celle destinée à représenter une commande, transcription d'une action réalisée par un joueur. Elle possède un attribut : args un unique_ptr<Action_Args> qui permet de passer à l'action à exécuter des arguments tel que l'endroit ciblé par le joueur lors d'un clic. Dans le cas d'un clic, c'est en effet une information essentielle pour savoir où déplacer le joueur ou savoir qu'il souhaite attaquer, par exemple.

Elle possède une unique méthode en plus de son constructeur et son destructeur.

- (*action)(), un pointeur sur une fonction qui selon le type de commande va modifier l'état de la manière souhaitée (dans le cas d'une commande **move** modifier la position du joueur, etc..)

Command
<pre> +args: std::unique_ptr<Action_Args> +Command(void(*action) (std::unique_ptr<Action_Args>&), args:std::unique_ptr<Action_Args>&) +~Command() +(*action) (a_args:std::unique_ptr<Action_Args>&): void </pre>

Action_Args

Cette classe a pour rôle de

Elle possède 2 attributs :

- Le deuxième est une union entre un int *point*[] et un char *selected*. Si on clique sur la map, l'union contient le point x,y correspondant, si on clique sur un bouton, il contiendra le code associé au bouton.
- Le premier est également une union, qui contient soit un std::shared_ptr<state::State> *state*, auquel cas la commande agit sur le state, ou un engine::Engine* *ngine*, et dans ce cas, la commande a un effet sur le fonctionnement de l'engine.

Elle ne possède pas de méthodes mais plusieurs constructeurs et un destructeur.

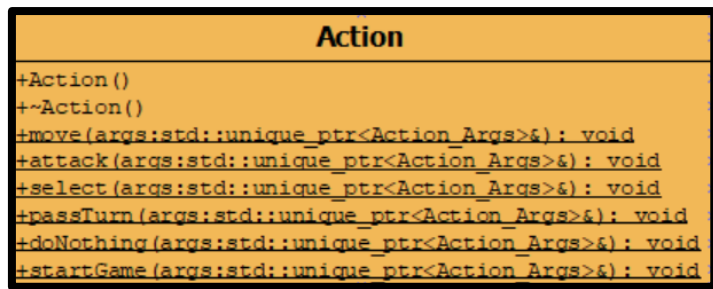
Action_Args
<pre> +{std::shared_ptr<state::State> state; engine::Engine* nengine;}: union +{int point[2]; char selected;}: union +Action_Args(gstate:std::shared_ptr<state::State>&, x:int,y:int) +Action_Args(gstate:std::shared_ptr<state::State>&, selected:char) +Action_Args(ngine:engine::Engine*,selected:char) +~Action_Args() </pre>

Action

Cette classe ne possède pas d'attribut, son rôle est de contenir les méthodes exécutées par les commandes suite aux actions des joueurs. Ce sont les méthodes sur lesquelles pointe, le pointeur de fonction de la classe Command et qui vont mettre à jour state en fonction des actions des joueurs.

Elle possède donc 5 méthodes en plus de son constructeur :

- move() pour modifier la position du joueur quand il désire se déplacer.
- attack() pour mettre à jour les points d'actions du joueur et diminuer les points de vie de l'adversaire concerné.
- select() permet de préparer une attaque en sélectionnant un sort avant d'attaquer l'ennemi.
- passTurn() permet de passer le tour du joueur actuel et donner la main au joueur suivant.
- doNothing() implémente l'action réalisée par le joueur en attente, quand ce n'est pas son tour
- startGame() permet d'initier le jeu après avoir cliquer sur le bouton JOUER du menu d'accueil.

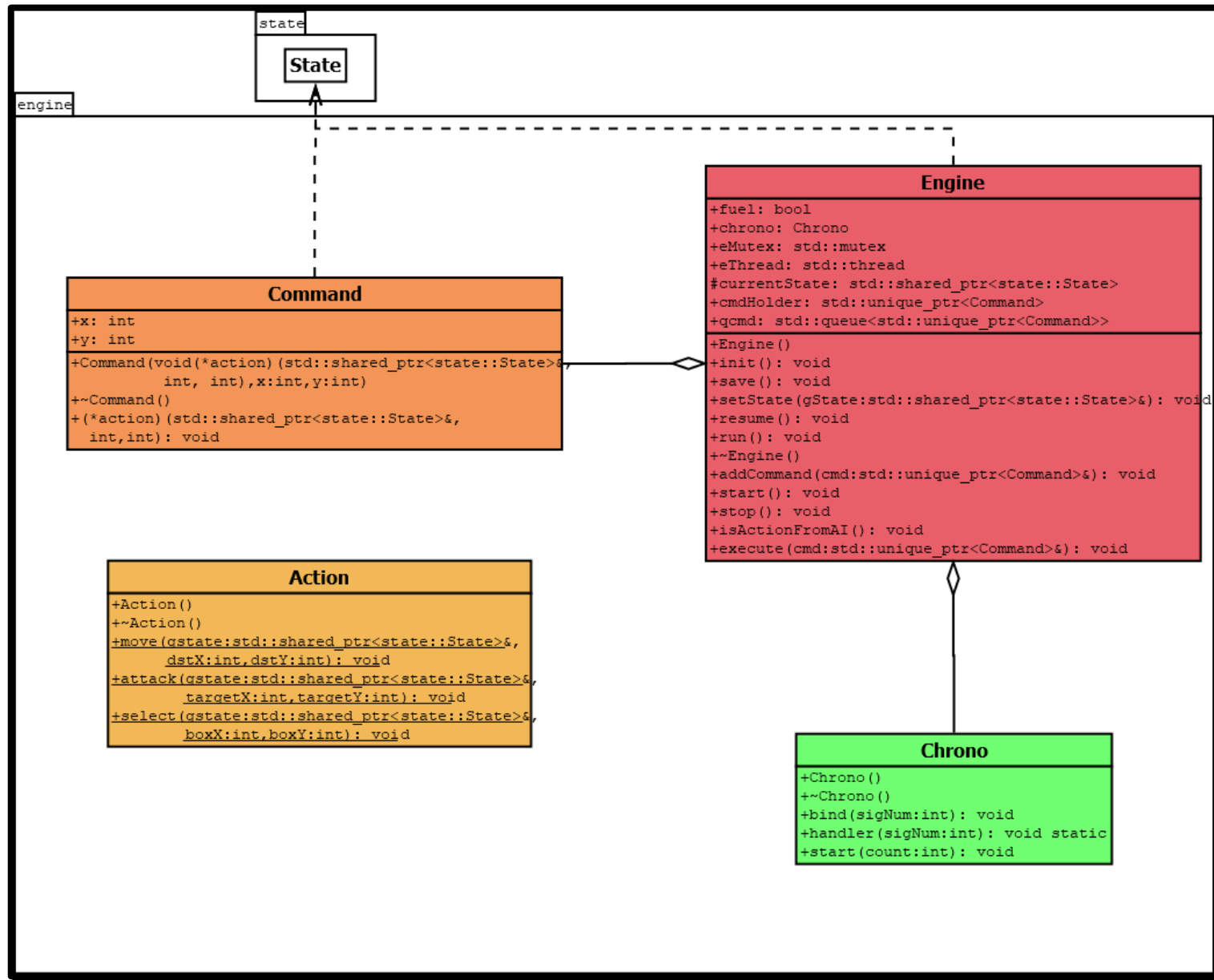


```
classDiagram
    class Action {
        +Action()
        +~Action()
        +move(args:std::unique_ptr<Action_Args>): void
        +attack(args:std::unique_ptr<Action_Args>): void
        +select(args:std::unique_ptr<Action_Args>): void
        +passTurn(args:std::unique_ptr<Action_Args>): void
        +doNothing(args:std::unique_ptr<Action_Args>): void
        +startGame(args:std::unique_ptr<Action_Args>): void
    }
```

The image shows a UML class diagram for the **Action** class. The class has a yellow header with the name **Action**. Below the header, the methods are listed in a yellow box with black text. The methods are: `+Action()`, `+~Action()`, `+move(args:std::unique_ptr<Action_Args>): void`, `+attack(args:std::unique_ptr<Action_Args>): void`, `+select(args:std::unique_ptr<Action_Args>): void`, `+passTurn(args:std::unique_ptr<Action_Args>): void`, `+doNothing(args:std::unique_ptr<Action_Args>): void`, and `+startGame(args:std::unique_ptr<Action_Args>): void`.

5.3 Conception logiciel : extension pour la parallélisation

Illustration 3: Diagramme des classes de l'engine



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

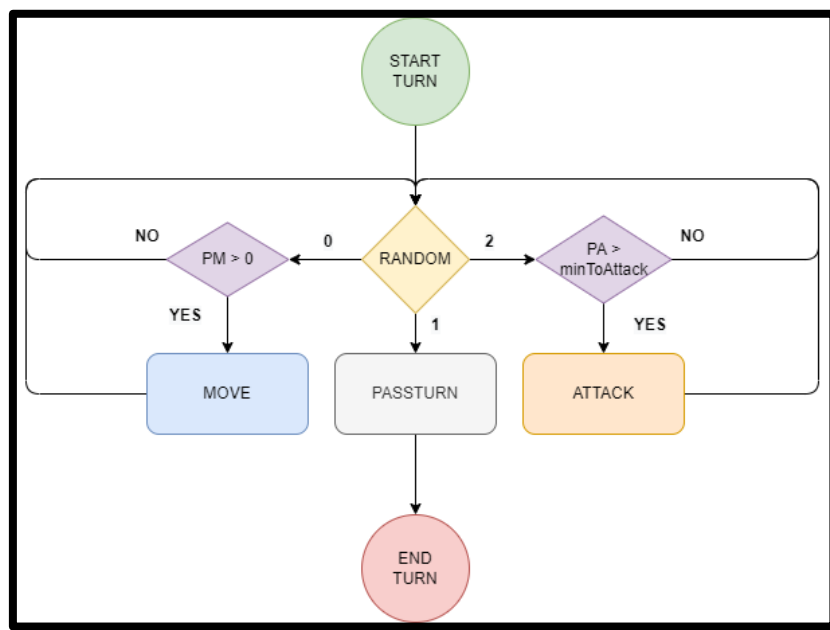


Figure 1 - Fonctionnement de l'IA minimale

Pour cette IA, la stratégie adoptée consiste simplement choisir aléatoirement entre 3 actions : se déplacer, attaquer ou passer son tour. Par conséquent, les scénarios réalisés par cette IA seront totalement imprévisibles et auront peu de chance de mener à sa victoire. L'IA fera ainsi des actions comme se déplacer même si elle est déjà en mesure d'attaquer et attaquer même si la cible n'est pas atteignable, frapper dans le vide, etc...Les actions sont réellement réalisées de manière aléatoire.

5.1.2 Intelligence basée sur des heuristiques

Pour cette IA intermédiaire, nous avons établi des règles simples mais qui assurent néanmoins un taux de réussite bien plus élevé que l'IA précédente. Ici, l'IA va toujours chercher à infliger des dégâts à ses adversaires.

Étape 1 : Elle s'approche le plus possible de sa cible en utilisant un minimum de points de mouvement.

Étape 2 : Si elle est suffisamment proche, elle attaque cette cible avec un sort choisi au hasard.

5.1.3 Intelligence basée sur les arbres de recherche

Enfin l'IA avancée, la stratégie adoptée consiste à se rapprocher de l'adversaire et à l'attaquer dès qu'il rentre dans le champ d'attaque de l'IA. Cependant, cette IA ne cherchera pas à s'enfuir ou se cacher derrière un obstacle pour éviter d'être attaquée au tour suivant.

La première étape pour cette IA consiste donc à vérifier si l'ennemi est atteignable en comparant la portée de ses sorts avec la distance qui les sépare. Si c'est le cas, elle l'attaque avec son sort le plus puissant, son objectif est d'infliger un maximum de dégâts. Elle continuera d'attaquer avec le sort le plus puissant à sa disposition tant qu'il lui reste des points d'actions.

Maintenant si aucun ennemi n'est à sa portée, elle va chercher à se rapprocher de l'ennemi ayant le plus faible niveau en consommant des points de mouvement, jusqu'à être en mesure de l'attaquer. A niveau égal, elle choisira de se rapprocher de l'ennemi ayant le moins de points de vie.

La seule exception est lorsqu'elle a la possibilité d'achever un ennemi (quel que soit son niveau). Dans ce cas cet ennemi deviendra sa cible prioritaire.

Il est donc nécessaire de comparer la distance à parcourir pour atteindre les ennemis, leurs niveaux, de comparer les dégâts réalisables à ces ennemis en prenant en compte les stats de l'IA **ET** celles des ennemis, et enfin de comparer les points de vie restants des ennemis.

Pour un ennemi **atteignable** donné si les dégâts réalisables sont supérieurs aux points de vie restants de cet ennemi, il deviendra la cible de l'IA indépendamment de son niveau. Et si plusieurs joueurs atteignables peuvent être achevés, l'IA choisira d'achever celui ayant le niveau le plus élevé (celui susceptible de lui faire de plus de dégâts et qui est donc la plus grande menace).

Si plusieurs ennemis sont atteignables (à la portée de ses attaques) mais non achevables, l'IA attaquera en priorité le joueur ayant le niveau le plus faible afin de maximiser les dégâts infligés.

5.2 Conception logiciel

AI

Cette classe est celle destinée à gérer le fonctionnement des différentes IA, il s'agit de la base sur laquelle sont construites les IA minimale, heuristique et avancée.

Elle possède 8 attributs :

- *gstate* de type `std::shared_ptr<state::State>` *gState* de type `std::shared_ptr<state::State>` pointe sur l'état du jeu afin d'actualiser l'affichage au fur et à mesure du combat.
- *ngine* de type `std::shared_ptr<engine::Engine>` pointe sur l'engine afin qu'il puisse exécuter les actions des IA.
- *targetX* de type `int` est la coordonnée x comprise entre 0 et 21 de la cible de l'IA, cela peut être une case quand elle se déplace, ou un joueur lorsqu'elle attaque.
- *targetY* de type `int` est la coordonnée y comprise entre 0 et 21.
- *selected* est un `char` contenant le code unique associé à un bouton de l'UI. Cette variable permet de savoir sur quel bouton l'IA a « cliqué ».
- *selections* est la liste des codes associés aux boutons de l'UI.
- *action_time_ini* et *action_time_end* de type `struct timespec` permettent de mesurer le temps qu'a pris une action à s'exécuter. *action_time_ini* est un marqueur de temps pour le début de l'action et *action_time_end* pour la fin, en faisant la différence des deux on accède au temps mis par l'IA pour exécuter l'action.

Elle possède 8 méthodes en plus de son constructeur et son destructeur.

- *initSrand()* permet de démarrer le générateur de nombre aléatoire.
- *getRandValBetween()* retourne de manière aléatoire un entier entre un intervalle passé en argument.
- *bind()* permet partager le state ainsi que l'engine aux IA.
- *closestEnemyIndexTo()* renvoie l'index de l'ennemi le plus proche.
- *weakestEnemyIndexTo()* renvoie l'index de l'ennemi le plus faible.
- *exploit()* permet de simuler l'action que l'IA a choisi d'effectuer comme le fait de cliquer sur un bouton ou sur la map par exemple.
- *getSelection()* reçoit le code du bouton associé à l'indice passé en paramètre de la fonction.
- *action_dt()* permet de mesurer le temps entre deux actions successives de l'IA.

```

AI
#gstate: std::shared_ptr<state::State>
#engine: std::shared_ptr<engine::Engine>
#targetX: int = 0
#targetY: int = 0
#selected: char
-selections[7]: char
#action_time_ini: struct timespec
#action_time_end: struct timespec

+AI()
+~AI()
+initSrand(): void
+getRandValBetween(a:int,b:int): inline int
+bind(engine:engine::Engine*,gstate:std::shared_ptr<state::State>): void
+closestEnemyIndexTo(p_index:char,pos:int*): char
+weakestEnemyIndexTo(p_index:char,pos:int*): char
+exploit(): void
+getSelection(sel:char): char
+action dt(): inline float

```

5.3 Conception logiciel : extension pour l'IA composée

HeuristicAI

Cette classe hérite de **AI** et a pour rôle la gestion de l'IA heuristique.

Elle ne possède pas d'attributs supplémentaires et a une unique méthode en plus de son constructeur :

- exploit() permet de simuler l'action que l'IA a choisi d'effectuer comme le fait de cliquer sur un bouton ou sur la map par exemple.

```

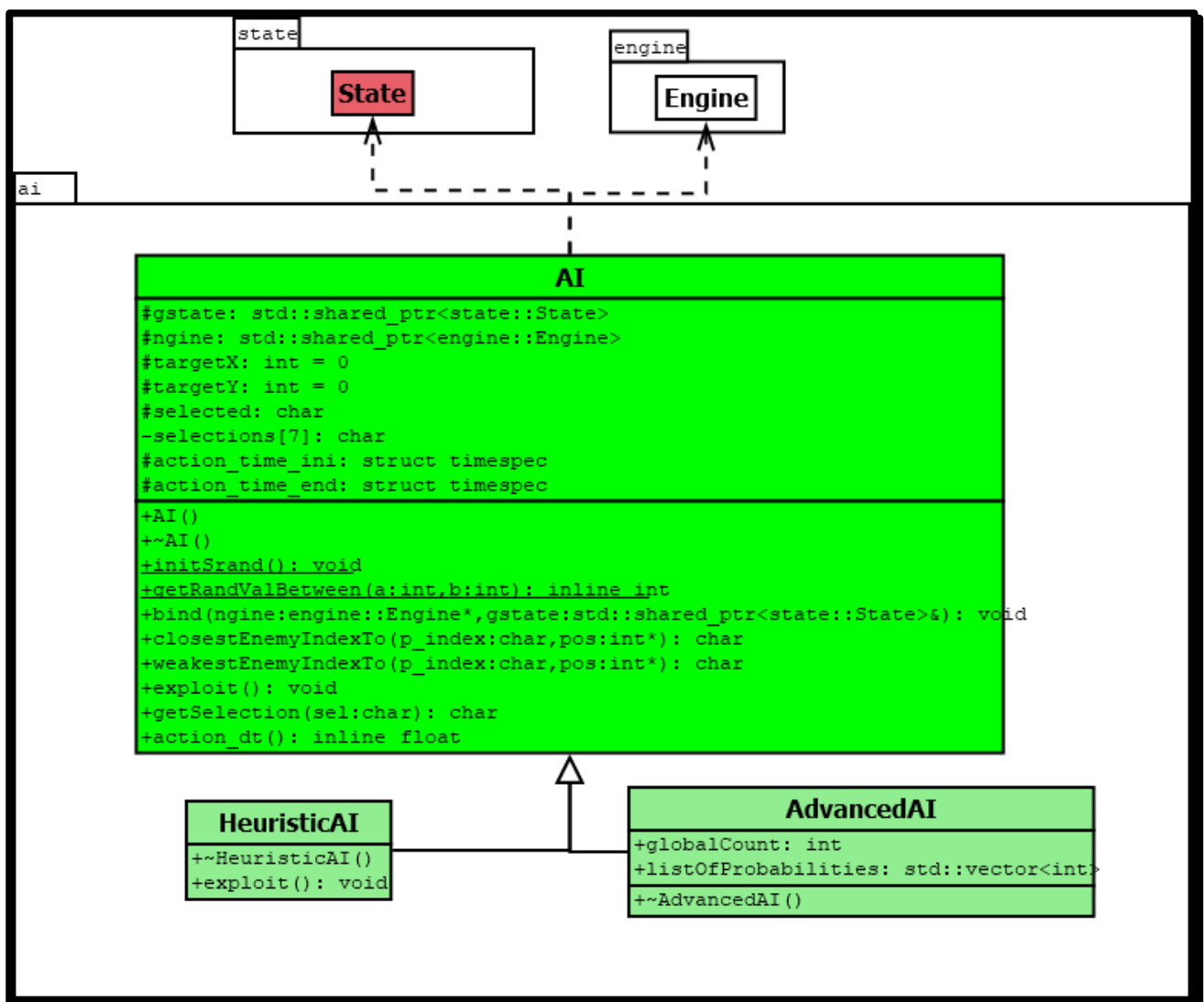
HeuristicAI
+~HeuristicAI()
+exploit(): void

```

5.4 Conception logiciel : extension pour IA avancée

5.5 Conception logiciel : extension pour la parallélisation

Illustration 4: Diagramme des classes de l'AI



6 Modularisation

La partie Modularisation concerne la parallélisation des processus du jeu, l'enregistrement et la reproduction d'une partie (sérialisation des données dans un fichier) ainsi que la mise en réseau d'une partie sur un serveur.

Concernant **la parallélisation** des processus nous avons choisi de nous concentrer sur celle du client et de l'engine.

Pour cela, la première étape a été de définir un thread principal, en l'occurrence ici, il s'agit de celui du render dont le rôle est d'afficher le jeu à l'écran, et un ou plusieurs threads secondaires pour l'engine. On souhaite ne pas avoir à attendre que l'affichage se fasse pour que le moteur de jeu continue de tourner, et inversement. Ainsi, selon les ressources requises pour chaque thread, le jeu sera capable de tourner plus efficacement.

Il faut alors réduire l'initialisation du jeu à un simple appel de fonctions qui permettent chacune de lancer un thread séparément, lesquels viendront ensuite renseigner les différents paramètres nécessaires au bon déroulement de la partie.

La deuxième étape concerne **l'enregistrement** d'une partie et le lancement d'une partie enregistrée. Sachant qu'une partie consiste en un enchaînement de commandes en tour par tour, nous avons ajouté pour l'ensemble des commandes pouvant être générées lors d'une partie, une fonction `serialize()`, qui permet de retranscrire au format texte standard les éléments nécessaires à l'exécution d'un tour de jeu.

Dans notre cas, il est nécessaire pour reproduire une partie de connaître les emplacements des joueurs avant les phases de déplacement et d'attaque, leurs stats (points de vie, attaque, défense, points d'actions et points de mouvements) puis de noter l'emplacement de la destination choisie et/ou les positions des attaques réalisées après ces phases. Par conséquent, les différents éléments nécessaires à l'enregistrement d'une partie sont au minima le numéro du tour accompagné des cases occupées par les joueurs ainsi que leurs statistiques (en réalité la classe, et le niveau du joueur suffisent), et les cases ciblées par leurs actions de déplacement ou d'attaque. Même si d'autres éléments peuvent être ajoutés tel que les instants des actions, les éléments cités plus haut constituent le minimum nécessaire à la reproduction et au suivi d'une partie. Pour **la reproduction de la partie**, le principe n'est pas bien différent du fonctionnement actuel, on va venir lire les informations contenues dans le fichier au format JSON et transmettre les différentes actions à l'engine comme le fait déjà l'IA et on pourra alors revoir le déroulement de la partie.

Dans un dernier temps, afin d'éviter que l'hébergement de la partie soit géré par nos propres machines et alléger leur charge nous avons souhaité externaliser cela sur un **serveur**. Ce serveur se charge ainsi de manipuler les données d'une partie et de les communiquer à une ou plusieurs machines connectées au réseau.

Ainsi, comme pour la parallélisation, on va séparer la réalisation des différents processus, cependant ici chaque processus ne se trouvera pas sur un thread différent de l'ordinateur mais sur des postes différents. On aura donc un serveur qui connaîtra toutes les informations liées à l'état du jeu (le state) et qui contiendra l'engine, tandis que l'utilisateur, lui, n'aura que son render sous forme de fenêtre de jeu sur son ordinateur.

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

