Projet Logiciel Transversal

Nouhou KANE – Mustafa KARADAG – Alexandre LOUIS

PROFUS

Table des matières

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 pour 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 fâce 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 pourrons reprendre les sorts existants sur Dofus afin de ne pas réinventer la roue.

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 dégats (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 qualifiée de classique, elle n'inflige pas beaucoup de dégâts mais ne nécessite que peu de points d'action et peut donc être utilisé 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 :

$$dégatsInfligés = shieldJoueurAttaqué - dégatsAttaque - attackJoueurAttaquand$$

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 :

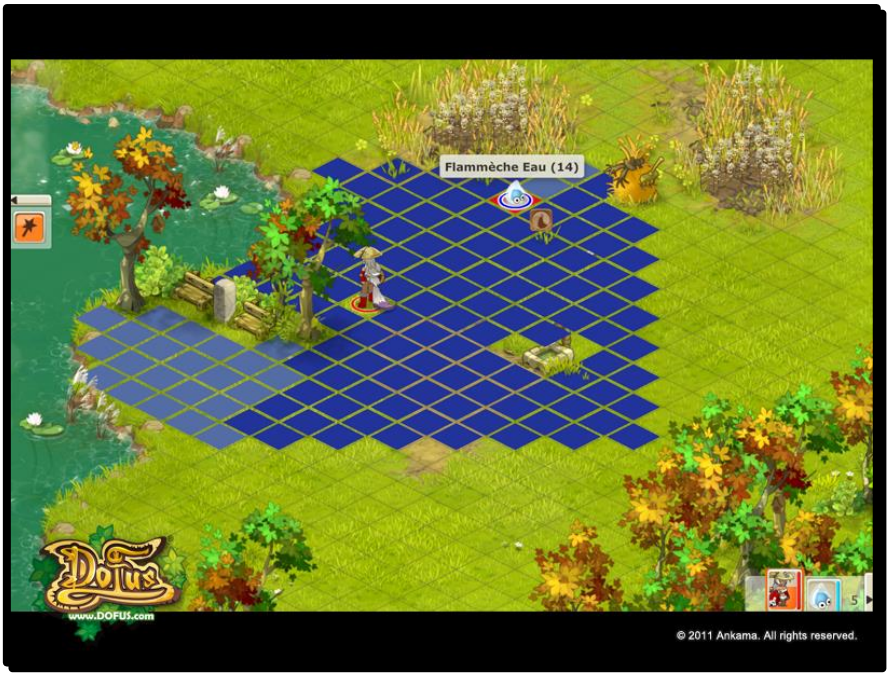
- **JOEUR VS JOUEUR** : Nous commencerons par permettre à deux joueurs réels de pouvoir s'affronter, puis si le temps nous le permet nous feront en sorte que les combats puissent opposer plus de deux joueurs.

et

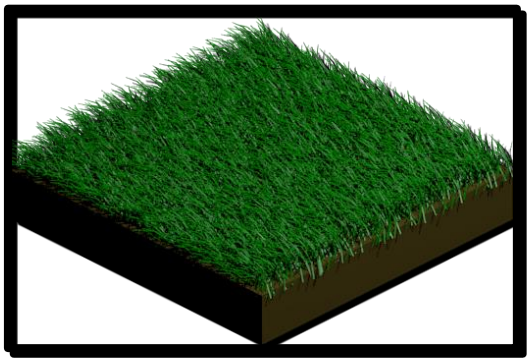
- **JOUEUR VS IA** : De même dans un premier temps ce mode opposera qu'un joueur à une IA ou une IA à une autre IA mais à terme nous souhaitons permettre des combats mixtes et plus conséquents, où IA et joueur réels pourront faire partie d'une même équipe.

1.3 Conception Logiciel

- **Archetype (DOFUS):**



- **Isometric grass :**

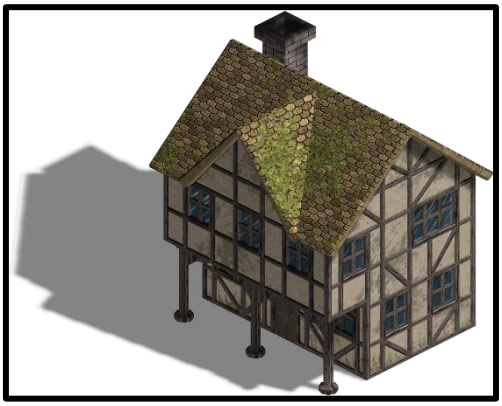


Source : https://opengameart.org/content/isometric-grass-tile

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 :**



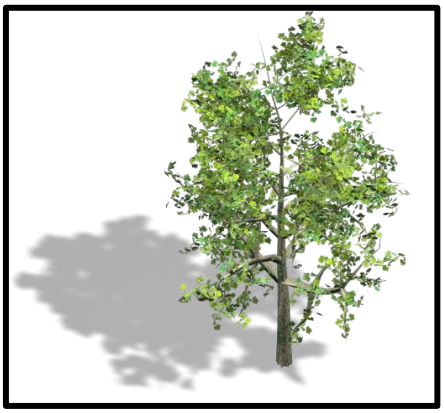
Source : https://opengameart.org/users/bleed

- **Timbered House_16 :**

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

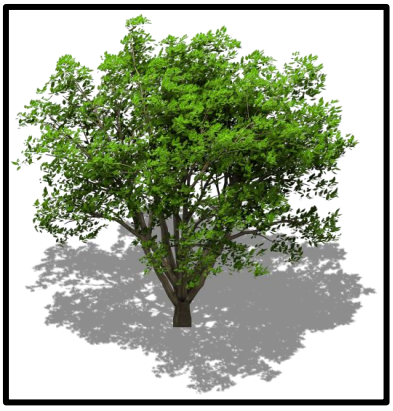
Nous avons sélectionné un peu de verdure avec différents types d'arbres :

- **_01 :**



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

- **Animated Swietenia :**



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

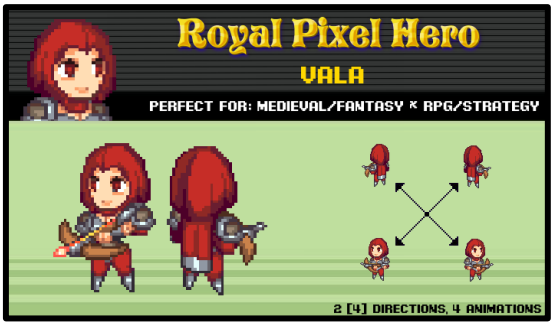
Et des éléments plus décoratifs mais qui serviront eux aussi d'obstacles comme **:**

- **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.

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 attaquent 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

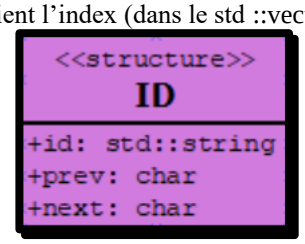
State est la classe principale du diagramme d'État. Elle permet de définir l'état dans lequel le jeu se trouve à tout instant et permet d'initialiser ces derniers.
Elle possède 16 attributs :

- *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.

ID

ID est une structure permettant d'identifier les joueurs mais également de faciliter également le passage d'un joueur à un autre et donc le changement de tour. Ses attributs sont :

- *id* contient le std ::string qui constitue l'identifiant du joueur, c'est-à-dire le nom donné au joueur plus (concaténation) '0' ou '1' en fonction de leur équipe.
- *prev* de type char contient l'index (dans le std ::vector players) du joueur précédent.
- *next* de type char contient l'index (dans le std ::vector players) du joueur suivant.

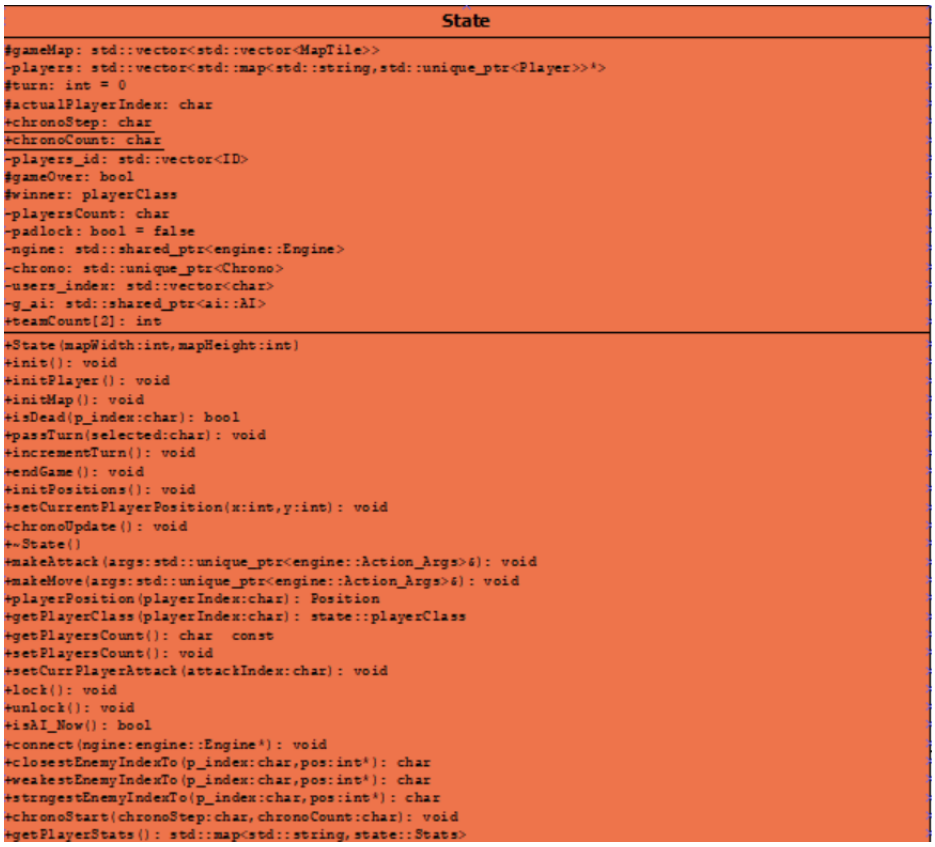


- *gameOver*, est un booléen mis à false initialement et qui passe à true quand le jeu est terminé, c'est-à-dire quand l'un des 2 joueurs n'a plus de HP.
- *winner*, de type playerClass contient la classe du joueur vainqueur (DEMON ou HERO). Cette variable est primordiale pour afficher la scène de fin 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.
- *users_index* de type std ::vector<char> contient les indices des joueurs réels, ceux qui ne sont pas des IA.
- *g_ai* de type std::shared_ptr<DeepAI> pointe sur l'IA avancée afin d'actualiser l'affichage au fur et à mesure du combat.
- *teamCount[2]* est un tableau de 2 entiers qui correspondent au nombre de joueurs par équipe.

En plus de son constructeur et son destructeur, elle possède 58 méthodes :

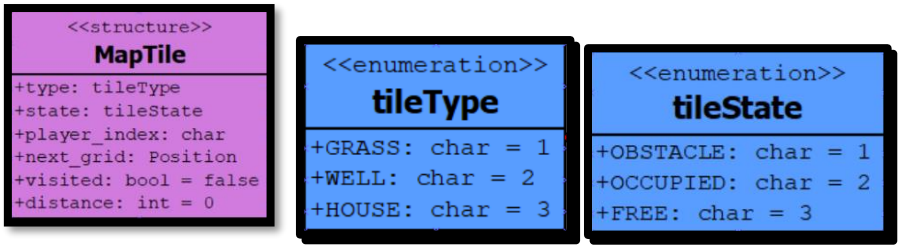
- init() pour initialise le state au lancement du jeu.
- initPlayer() pour créer les joueurs (objets Player) au lancement du jeu.
- initMap() pour charger la carte en début de partie.
- initPositions() initialise la position des joueurs en début de partie.
- isDead() vérifie les points de vie des joueurs, 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 par rapport au joueur dont on passe l'index.
- weakestEnemyIndexTo() renvoie l'index de l'ennemi le plus faible par rapport au joueur dont on passe l'index.
- strngestEnemyIndexTo() renvoie l'index de l'ennemi le plus fort (avec le niveau le plus élevé) par rapport au joueur dont on passe l'index.
- 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.
- pull_AP_THP() récupère l'AP du joueur actuel et les HP de sa cible.
- get_AP() renvoie les AP du joueur dont l'index est passé en argument.
- get_MP() renvoie les MP du joueur dont l'index est passé en argument.
- get_Shield() renvoie le shield du joueur dont l'index est passé en argument.
- get_Attack() renvoie l'attaque du joueur dont l'index est passé en argument.
- getCurrAttackIndex() renvoie la valeur *currentAttackPlayer* du joueur dont l'index est passé en argument.
- enemyWithLessHp_Of() renvoie l'index de l'ennemi ayant le moins de points de vie.
- enemyWithLessMp_Of() renvoie l'index de l'ennemi ayant le moins de points de mouvement.
- damage() retourne les dégâts qu'infligerait un joueur à un autre pour une attaque donnée.
- simu_attack() permet de simuler les conséquences d'une attaque.
- cancel_move() va permettre d'annuler un déplacement.
- cancel_attack() va permettre d'annuler une attaque.
- cancel_select() va permettre d'annuler la sélection d'un sort.
- cancel_passTurn() permet de revenir en arrière et annuler le passement de tour d'un joueur.
- BFS_Shortest_Path() prend en paramètres une position de départ et une position d'arrivée et détermine le chemin le plus court pour réaliser ce trajet en prenant en compte les obstacles.
- operator[] est un opérateur nous permettant d'accéder plus simplement et plus rapidement à la variable gameMap. En effet, si on se trouve dans la classe State, il suffit désormais de faire (*this)[position] pour accéder à la case de la map souhaitée.
- Les fonctions inMap() permettent de vérifier qu'une position se trouve au sein de la map. Elle renvoie true si c'est le cas, false sinon.

- isFree() permet de vérifier qu'une case de la map est libre ou non. Elle renvoie true si c'est le cas, false sinon.
- getAttackIndex() prend en argument l'index du joueur qui nous intéresse et renvoie l'index de son attaque actuellement sélectionnée.
- getAttack() prend en argument l'index du joueur qui nous intéresse et renvoie l'attaque (l'objet) actuellement sélectionnée par le joueur.
- turn_in_AI() permet de changer le statut du joueur dont l'index est passé en argument est de faire de lui un IA.
- turn_all_in_AI() permet de changer tous les joueurs qui ne le sont pas en IA.
- restore_user() permet de changer le statut du joueur dont l'index est passé en argument est de faire de lui un joueur réel, qui n'est pas une IA.
- restore_all_user() permet de changer tous les joueurs qui n'étaient pas des IA à l'origine, de nouveau en joueurs réels.
- hasEnough_AP() vérifie que le joueur dont on passe l'index en argument a suffisamment de PA pour réaliser l'attaque dont l'index est passé en argument.
- clear() permet de réinitialiser le statut 'visited' (utilisé pour le déplacement des IA) des cases de la map à false
- enemiesCount() renvoie le nombre d'ennemis du joueur dont l'index est passé en argument.



```
+getPlayerStats(sel:char): state::Stats
+getPlayersID(): std::vector<ID>
+getPlayersAttacks(): std::map<std::string,std::vector<Attack>>
+pull_AP_THP(x:int,y:int,ap_thp[2]:int): void
+get_HP(p_index:char): int
+get_MP(p_index:char): int
+get_AP(p_index:char): int
+get_Shield(p_index:char): int
+get_playerPower(p_index:char): int
+getCurrAttackIndex(p_index:char): char
+enemyWithLessHp_Of(p_index:char,pos:int*): char
+enemyWithLessMp_Of(p_index:char,pos:int*): char
+damage(p_index:char,attacker_index:char,attack:Attack): int
+cancel_move(args:std::unique_ptr<engine::Action_Args>&): void
+cancel_attack(args:std::unique_ptr<engine::Action_Args>&): void
+cancel_select(args:std::unique_ptr<engine::Action_Args>&): void
+cancel_passTurn(selected:char): void
+BFS_Shortest_Path(src:Position,dst:Position): bool
+operator[](p:Position): MapTile&
+operator[](point[2]:int): MapTile&
+operator[](id:std::string): std::unique_ptr<Player>&
+inMap(p[2]:int): bool
+inMap(p:Position): bool
+isFree(p:Position): bool
+get_Attack(p_index:char): Attack
+get_Attack(p_index:char,attack_index:char): Attack
+getAttackIndex(p_index:char): char
+simu_attack(p_index:char,t_index:char,atck_index:char,p_stats:state::Stats&,t_stats:state::Stats&): void
+turn_in_AI(p_index:char): void
+turn_all_in_AI(): void
+restore_user(p_index:char): void
+restore_all_users(): void
+hasEnough_AP(p_index:char,attack_index:char): bool
+inMap(x:int,y:int): bool
+clear(): void
+enemiesCount(p_index:char): int
```

La structure **MapTile** associée à **State** va permettre de décrire chacune de ces cases à l'aide de ses différents attributs:
-*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.**
-*player_index* est char contenant l'index du joueur qui occupe la case dans le cas où state = OCCUPIED.
-*next_grid* contient la position de la prochaine case de l'itinéraire sur laquelle l'IA doit se déplacer.
-*visited* est un booléen valant true si l'IA est déjà passée par cette case au cours de son itinéraire de déplacement, false sinon.
-*distance* est un entier correspondant à la distance séparant la case actuelle de la destination souhaitée.
Elle possède également un attribut *player_id* qui permet d'idendifier 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 :

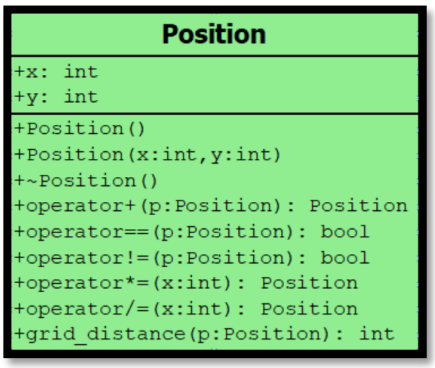
- *name*, un std ::string qui correspond au nom choisi pour son personnage.
- *playing* est un booléen permettant d'indiqué si c'est le tour du joueur en question ou non, s'il est en train de jouer.
- *level* de type int contient le niveau du joueur, il peut varier entre 1 et 100. Le niveau du joueur correspond à son avancement dans le jeu. Cela affecte ses stats et les dégâts qu'il peut infliger avec ses attaques.
- 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.
-

Position

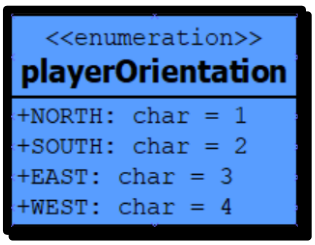
- sa *position* : de type **Position** est un objet ayant deux attributs entiers x et y, correspondant aux coordonnées du joueur sur la map.

Elle possède 5 opérateurs en plus de son constructeur et son destructeur :
operateur+(p :Position), operateur==(p :Position), operateur !=(p :Position),
operateur !=(p :Position) et operateur !=(p :Position) qui permettent de faciliter les opérations sur des objets Position et rendre le code plus lisible.
Par exemple : pos1 != pos2 va permettre de comparer deux positions (c'est-à-dire réaliser pos1.x != pos2.x et pos1.y != pos2.y)

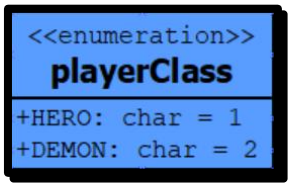
Enfin elle possède une méthode grid_distance() qui permet de déterminer la distance entre la position actuelle (couple des attributs x et y) et une position donnée en paramètre.



- 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**.



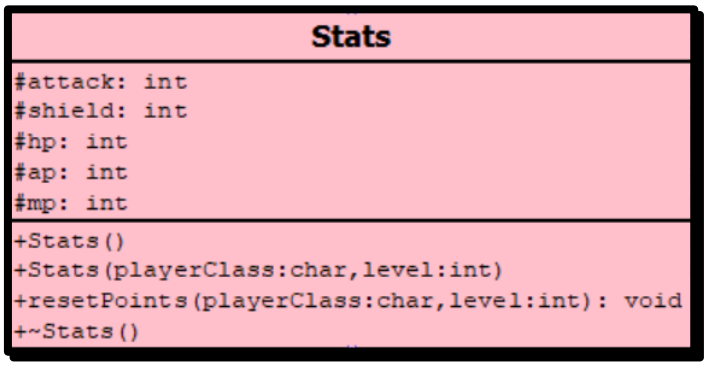
- sa *pClass* : Le joueur a le choix entre jouer un *Hero* ou un *Demon*. On les retrouve dans l'énumération associée **playerClass**.



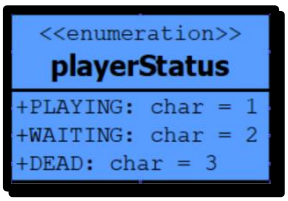
- ses stats (caractéristiques liées au combat): Définies par la classe **Stats**. Le joueur dispose de *mp* (movement point) consommables pour se déplacer ; de *ap* (action point) consommables pour effectuer une action comme une attaque ; de *hp* (health point = points de vie) représentant son niveau 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.

  Elle possède une méthode en plus de ses constructeurs et de son destructeur. Il s'agit de resetPoints qui comme son nom l'indique permet de restituer les points de mouvement (MP) et d'action (AP) du joueur à chaque fois

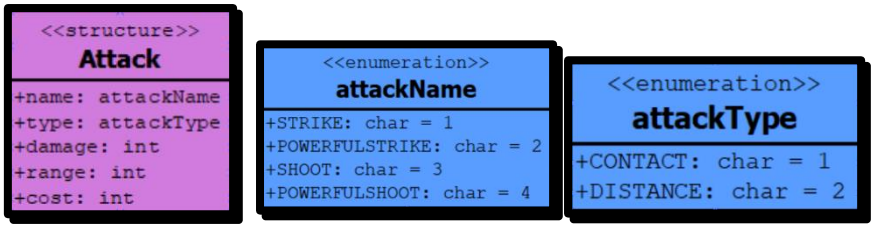
que son tour commence.



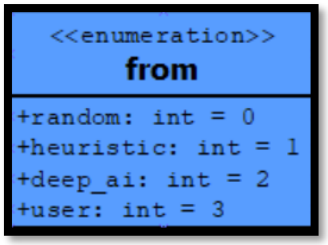
- son status en combat : Indique si le joueur joue, s'il est en attente de son tour de jeu ou s'il est mort. On les retrouve dans l'énumération associée **playerStatus**.



- ses *attacks* en combat : Chaque attaque est une stucture 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**).



- son *input* , un entier permettant de distinguer les différents joueurs en fonction de leur nature : utilisateur réel ou intelligence artificielle. On retrouve les valeurs que peut prendre cet attribut dans l'énumération associée **from**.
-

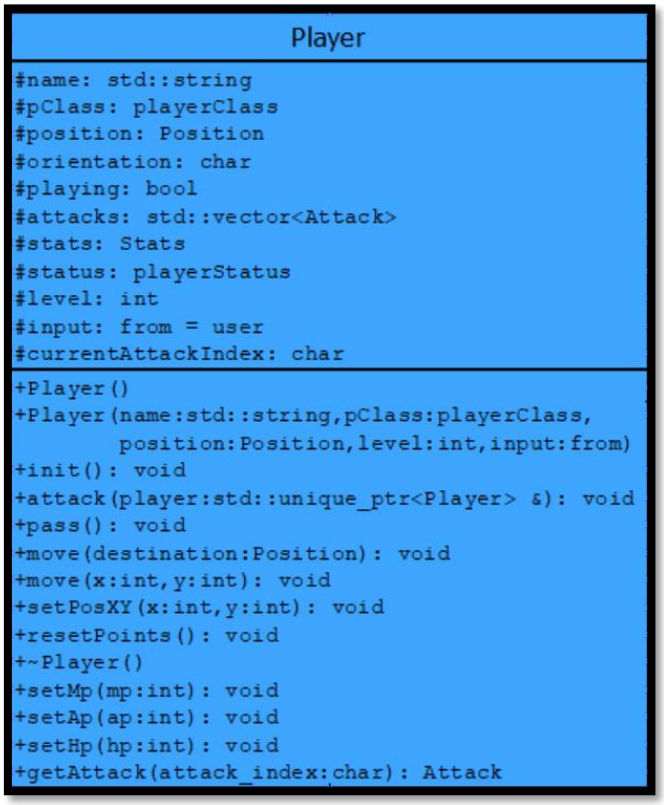


En plus de son constructeur et son destructeur, elle possède 9 méthodes :

- 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.
- setMp() permet de modifier les MP du joueur.
- setAp() permet de modifier les AP du joueur.
- setHp() permet de modifier les HP du joueur.
- getAttack() permet de renvoyer une attaque de la liste d'objets Attack du joueur en fonction de l'index est donné en paramètre.



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 :

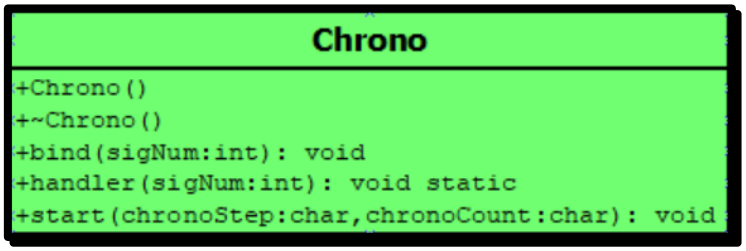
1. Lorsqu'une commande est générée, elle est enregistrée dans l'engine.
2. Si le padlock est ouvert, l'engine exécute la commande puis verrouille le padlock (lock()). L'engine attend alors que l'update du render ait lieu.
3. Le render actualise le rendu, déverrouille le padlock (unlock()) et provoque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

2.4 Conception logiciel : extension pour le moteur de jeu

Chrono

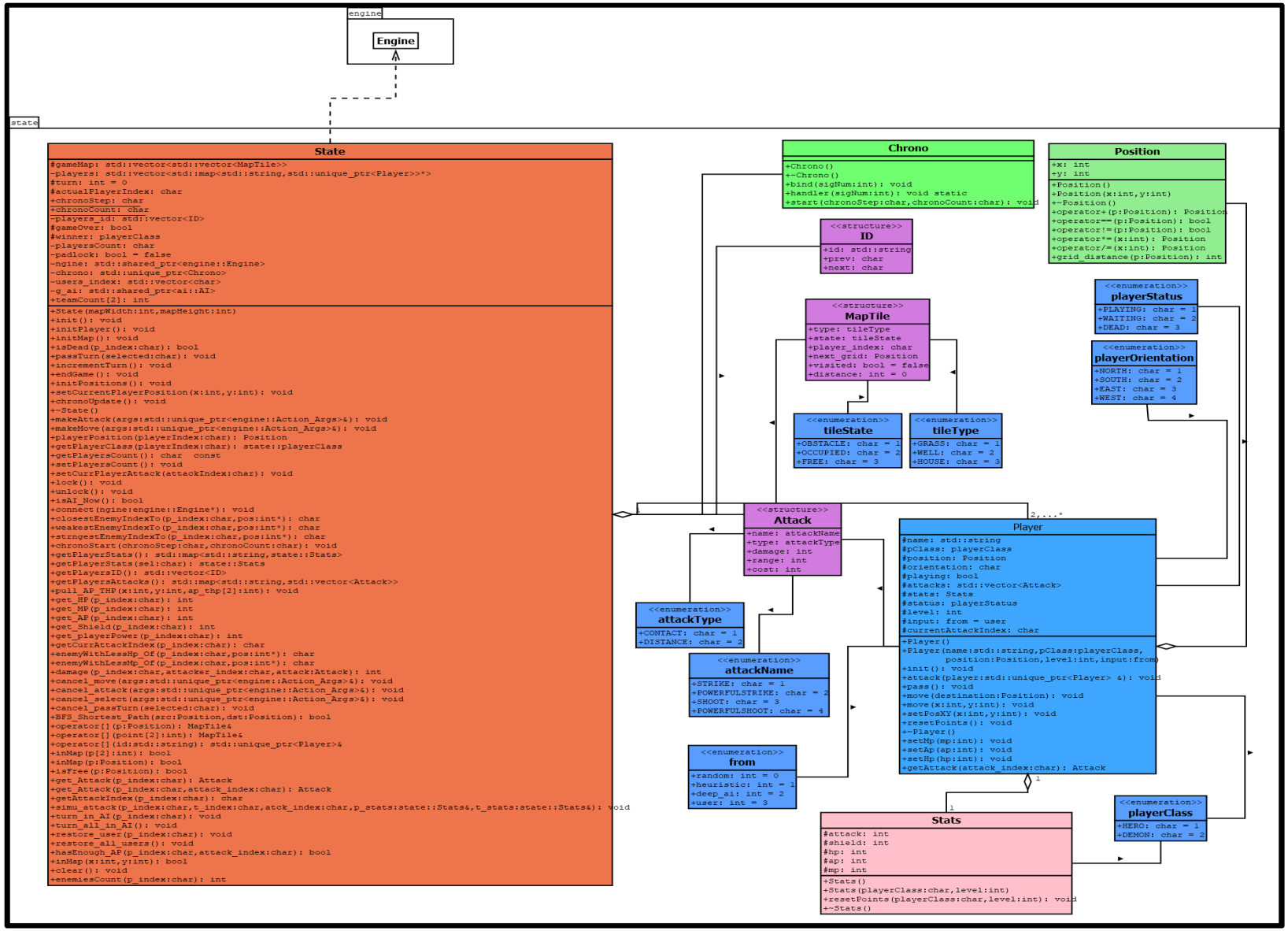
Cette classe est celle destinée à surveiller le temps passé et prévenir l'engine toutes les 60 secondes pour passer le tour du joueur.
Elle ne possède pas d'attribut et a trois méthodes en plus de son constructeur et son destructeur :

- 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.



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 render. 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.
- *heigh* 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 :

- 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 telles 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.

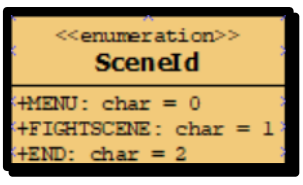
(l'association de ces deux dernières fonctions permet ainsi de se déplacer comme on le souhaite sur la map)

- 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.

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.



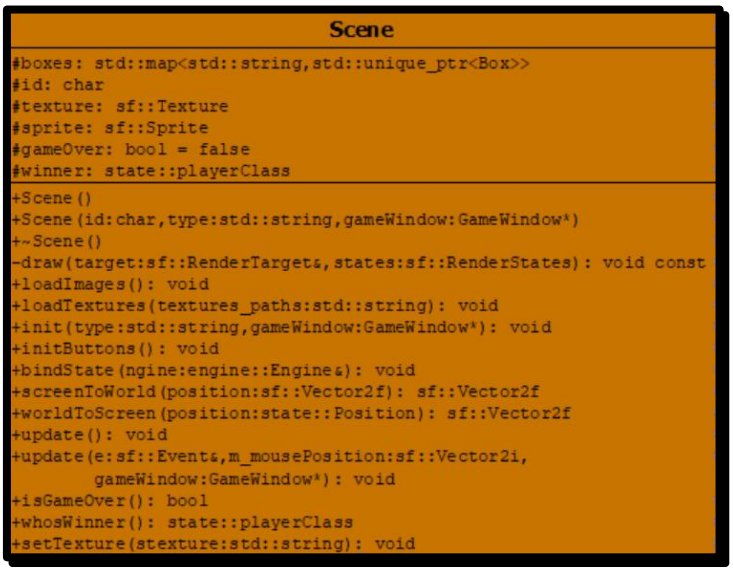
Scene

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* est 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.



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 :

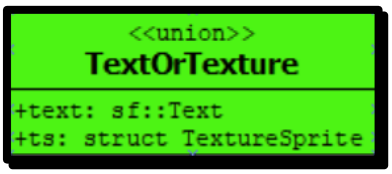
- *m_position* de type sf ::Vector2i qui contient les coordonnées de l'objet **Box** afin de pouvoir le dessiner à l'endroit souhaité via la méthode draw().
- *dimensions* de type sf ::Vector2f est un couple qui contiendra la longueur et la largeur de la Box à afficher.
- *m_text*, est le sf ::Text que 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.

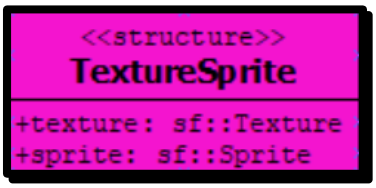
TextOrTexture

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.



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.



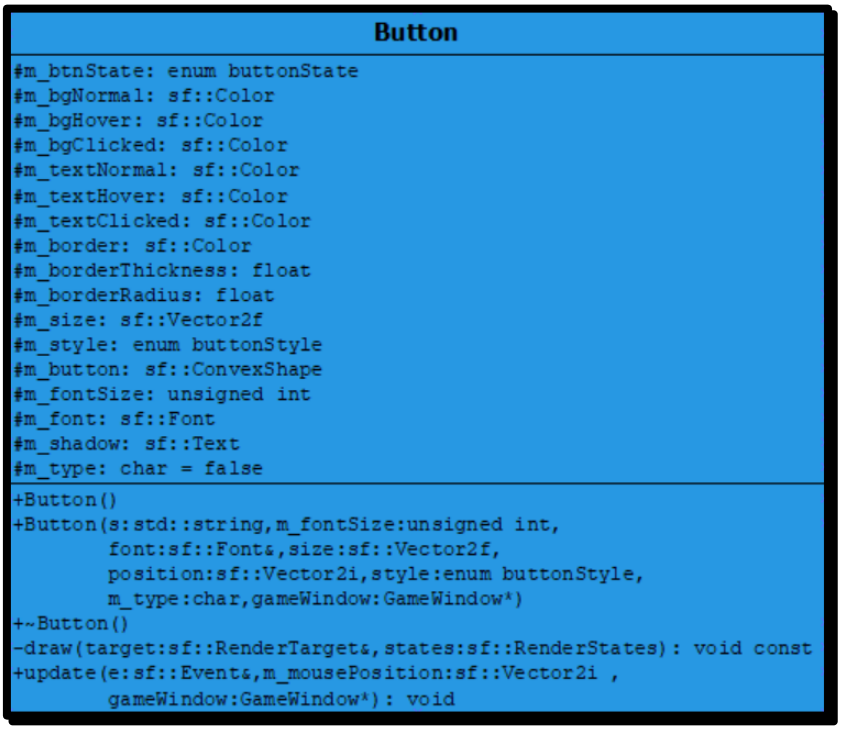
Button

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 qui indique 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
- Enfin, *m_type*, un char correspondant à un code unique permettant de distinguer les différents boutons.

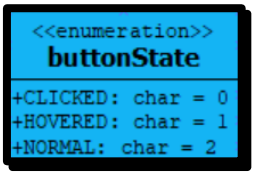
En plus de ses constructeurs et son destructeur, elle possède 1 constructeur à paramètres ainsi que 2 méthodes :

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



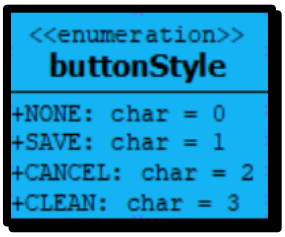
buttonState

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.



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.



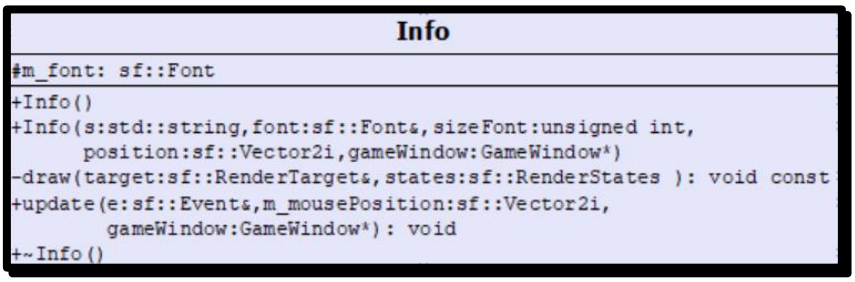
Info

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.



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 un fichier dans lequel sont inscrites les postions 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 le 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 le 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.

3.3 Conception logiciel : extension pour les animations

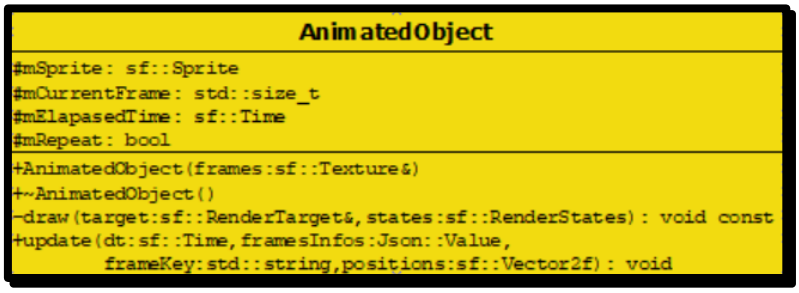
AnimatedObject

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 correspond 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).
- u*pdate()*, qui permet de mettre à jour l'objet animé, en actualisant la frame à afficher et le temps écoulé.

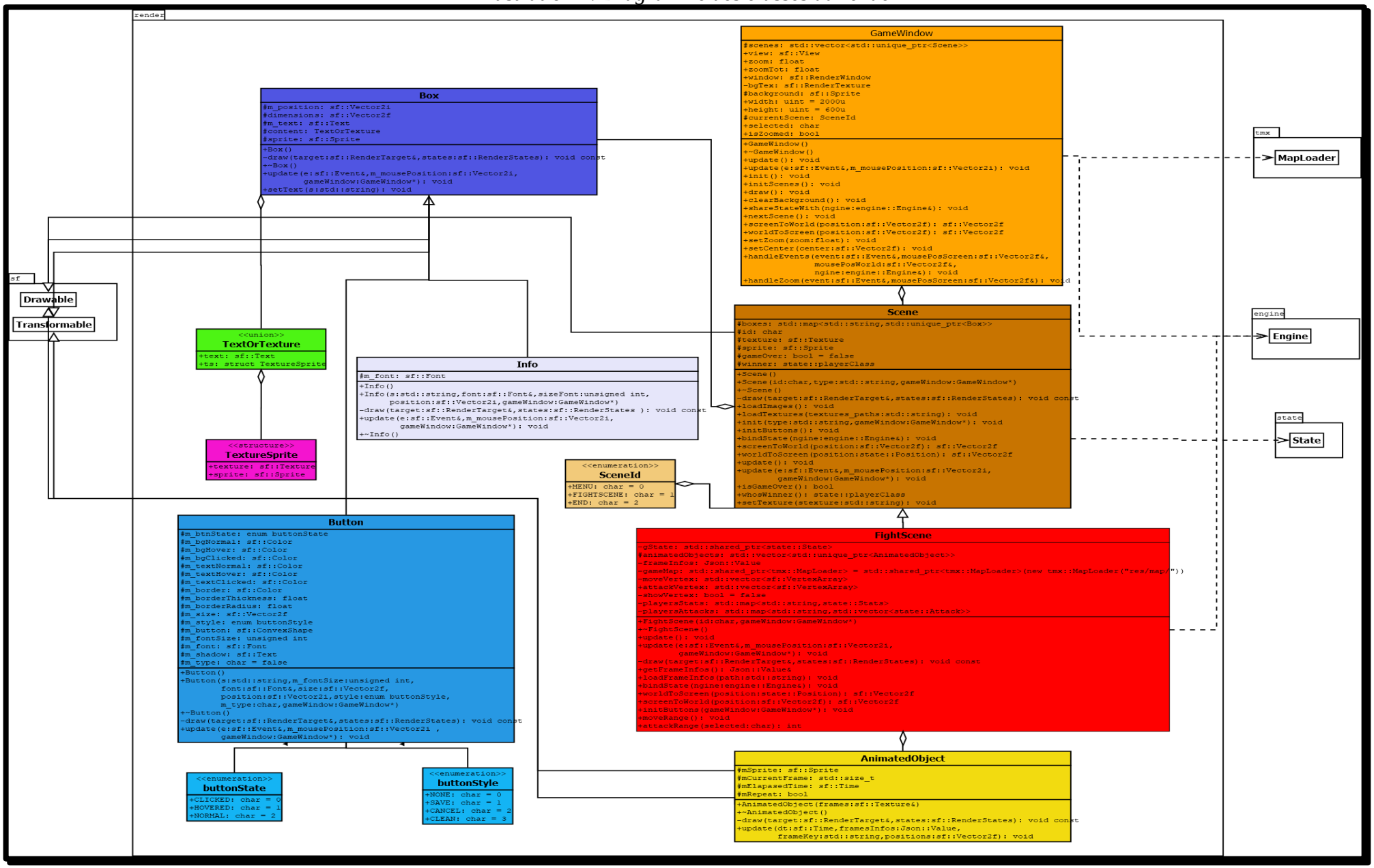


3.4 Ressources

3.5 Exemple de rendu

Exemple d'état de jeu

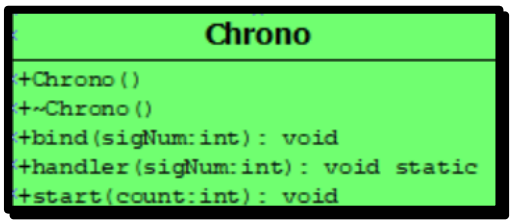
Illustration 1: Diagramme des classes du render



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.



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 lancer que sur les cases entourant son personnage dans un périmètre de maximum *valeur_portée_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

Changement de tour :

Enfin, un changement d'état à évidemment lieux 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 lieux. 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.

4.4 Conception logiciel

4.5 Conception logiciel : extension pour l'IA

Engine

Cette classe est celle destinée à représenter le moteur de jeu et contenir ses fonctionnalités.
Elle possède 11 attributs :

- *fuel* est un booléen servant d'indicateur on/off de l'engine.
- *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 remplie et se vide continuellement.
- *cmdUndid* de type std::deque<std::unique_ptr<Command>> a été ajoutée pour implémenter le rollback, cette variable contient les commandes qui ont été annulés par un retour en arrière
- *cmdHistory* de type std::stack<std::unique_ptr<Command>> a été ajoutée pour implémenter le rollback, cette variable correspond à l'historique des commandes pour permettre un retour en arrière.
- *cmd* est une liste de fonction de gestion de commande
- *void(*selection[4][2])(std::unique_ptr<Action_Args>&)* est une liste de fonctions associées aux différents action de sélection (clic sur bouton) ainsi que les fonctions d'annulation d'action associée. Nous avons separé 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 18 méthodes en plus de son constructeur et son destructeur.

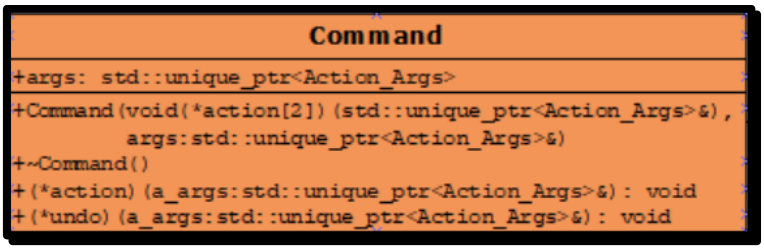
- init*()*, pour correctement initialiser l'engine au lancement du jeu.
- save*()*, pour sauvegarder l'avancement du jeu (tour actuel, stats des joueurs, etc..)
- setState*()*, pour mettre à jour *currentState* et c'est-à-dire actualiser l'état de jeu dans l'engine.
- resume*()*,pour charger et reprendre 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.
- timeOut() permet d'avertir l'engine quand le temps alloué au tour d'un joueur est atteint.
- bind() permet de partager le state et l'engine aux IA.
- undo() permet comme le ferait un ctrl+Z de revenir en arrière durant un tour, en annulant les commandes qui ont été réalisées.
- redo() permet comme le ferait un ctrl+Y de rétablir les commandes annulées.
- move_cmd() et atck_cmd() génèrent des commandes selon une action (move : move_cmd() ou attaque: atck_cmd()).

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 deux méthodes 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 le position du joueur, etc..)
- (*undo)(), il s'agit également d'un pointeur sur une fonction qui va permettre d'annuler une action en réalisant l'effet inverse (dans le cas de l'annulation d'une commande **move** d'un point A vers un point B, il s'agit de réaliser un move du point B vers le point A)

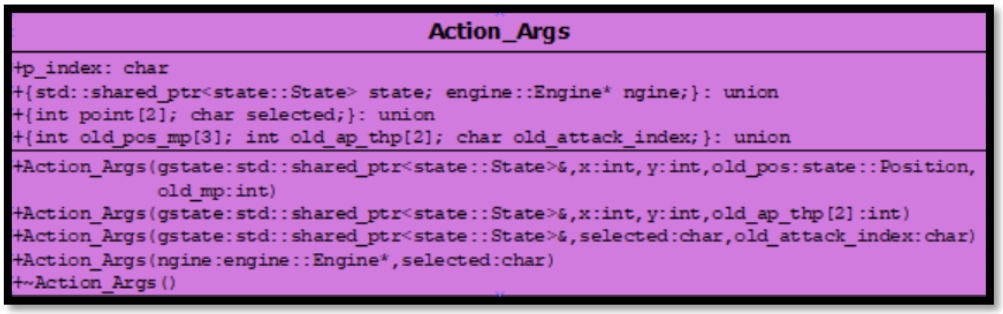


Action_Args

Cette classe a pour rôle de
Elle possède 4 attributs :

- Le premier *p_index* stocke l'index du joueur pour lequel il faut exécuter l'action.
- Le troisième 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 deuxième 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.
- Le quatrième est lui aussi une union, qui contient :
    - soit un int old_pos_mp[3], permettant de sauvegarder les informations d'un joueur avant un mouvement : les deux premiers éléments du tableau stockant sa position et le troisième élément ses MP ;
    - soit un int old_ap_thp[2], permettant de sauvegarder les informations de l'attaquant et l'attaqué avant une attaque : le premier élément du tableau stockant les AP de l'attaquant et le deuxième élément les HP de l'attaqué ;
    - soit un char old_attack_index, contenant l'ancien index d'attaque qu'avait sélectionné le joueur.

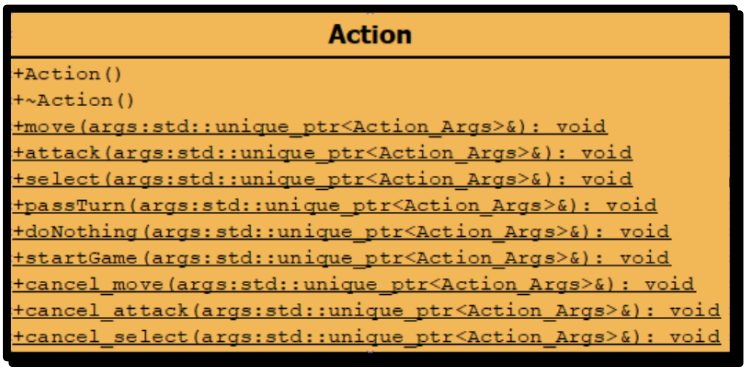
Elle ne possède pas de méthodes mais plusieurs constructeurs et un destructeur.



Action

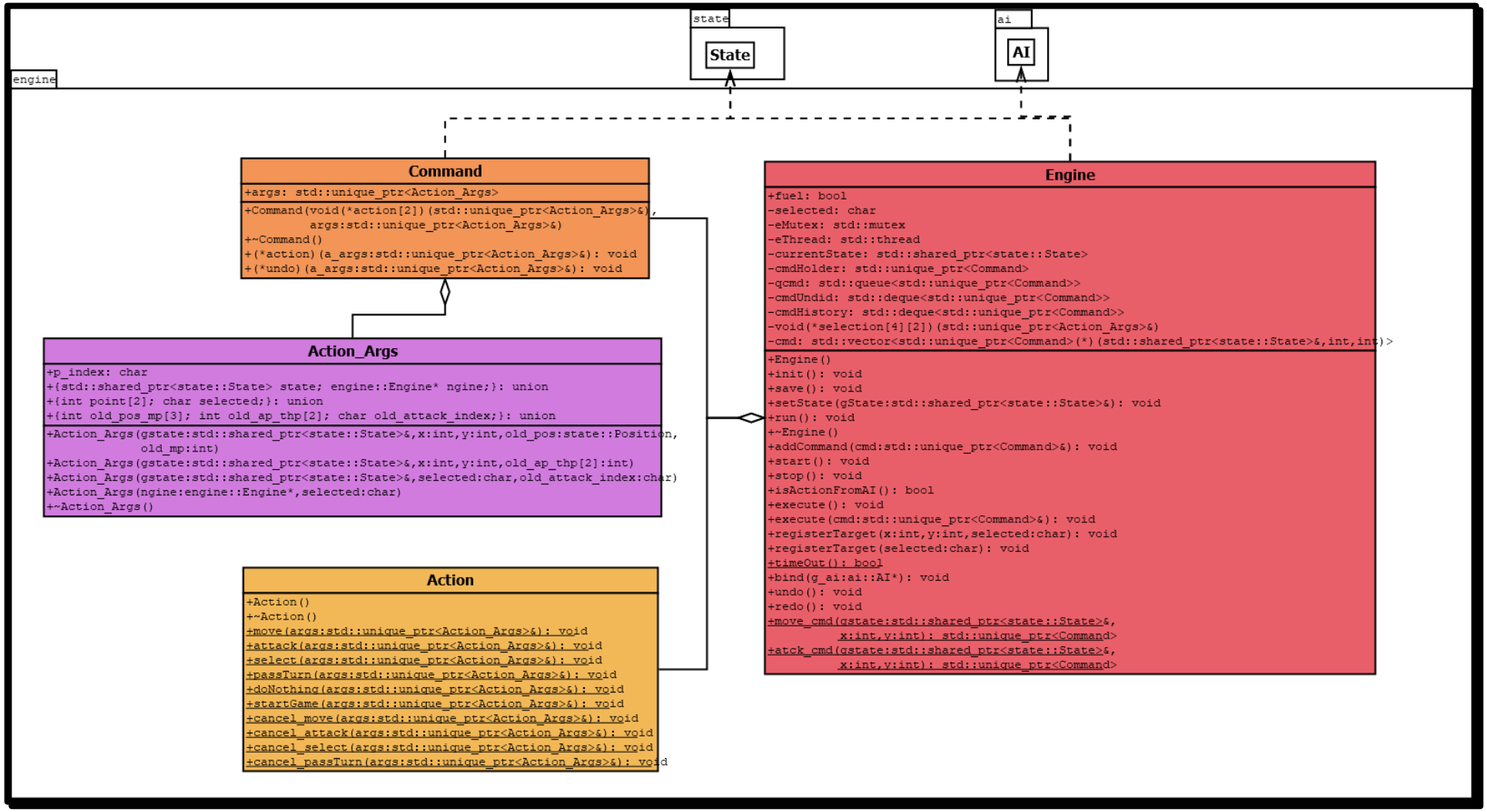
Cette classe ne possède pas d'attribut, son rôle est de contenir les méthodes exécutées par les commandes suite aux actions des joueurs. Ce sont les méthodes sur lesquelles pointe, le pointeur de fonction de la classe Command et qui vont mettre à jour state en fonction des actions des joueurs.
Elle possède donc 10 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.
- cancel_move() va permettre d'annuler un déplacement.
- cancel_attack() va permettre d'annuler une attaque.
- cancel_select() va permettre d'annuler la sélection d'un sort.
- cancel_passTurn() permet de revenir en arrière et annuler le passement de tour d'un joueur.



4.6 Conception logiciel : extension pour la parallélisation

Illustration 3: Diagramme des classes de l'engine



5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence minimale

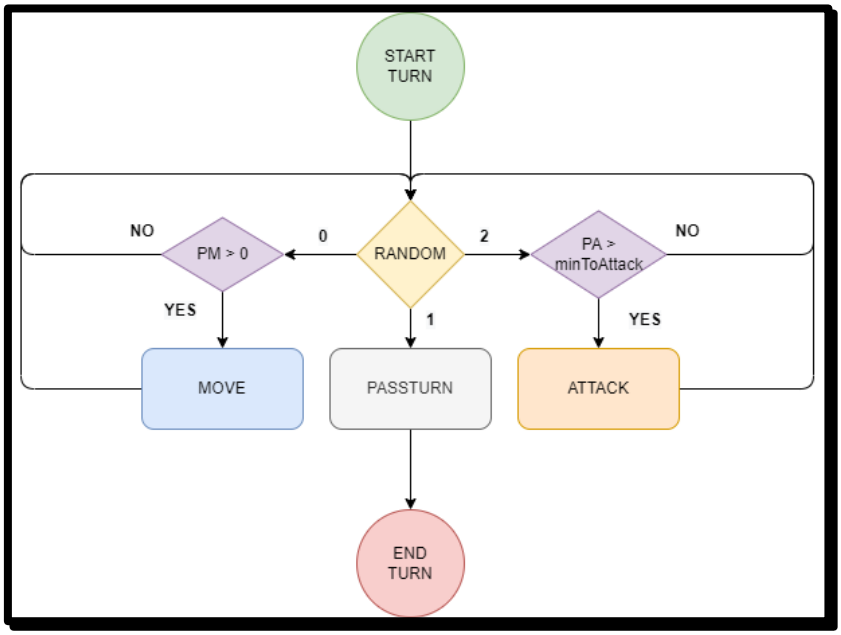


Figure 1 - Fonctionnement de l'IA minimale

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

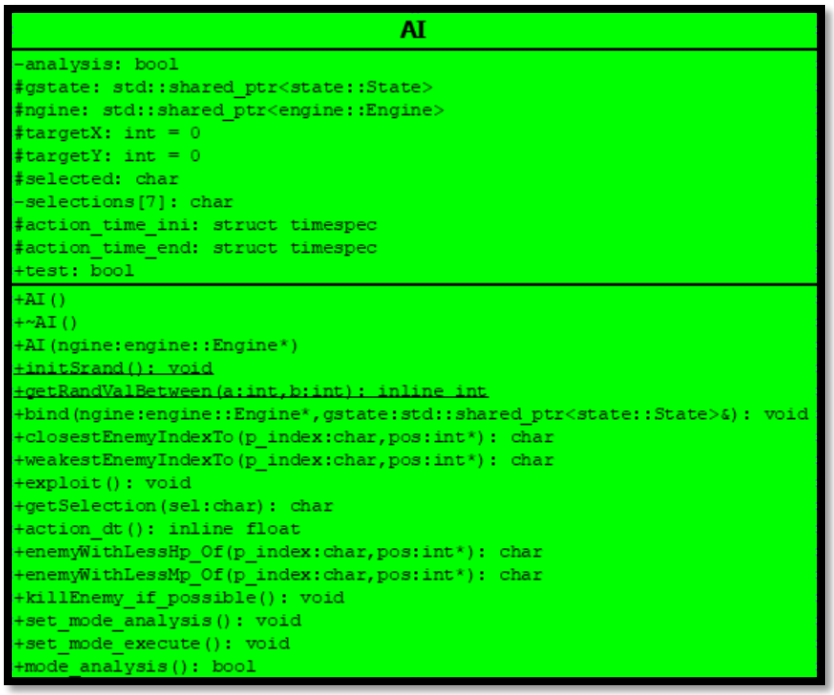
Cette classe est celle destinée à gérer le fonctionnement des différentes IA, il s'agit de la base sur laquelle sont construites les IA minimale, heuristique et avancée.
Elle possède 10 attributs :

- *analysis* un booléen qui vaut true quand l'IA est en mode analyse de l'état de jeu, et false sinon.
- *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'éxécuter. *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.
- *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.
- *test* un booléen qui vaut true quand l'IA essaye les différentes stratégies pour déterminer laquelle est la meilleure, et false sinon.

Elle possède 14 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.
- enemyWithLessHp_Of() renvoie l'index de l'ennemi ayant le moins de points de vie.

- enemyWithLessMp_Of() renvoie l'index de l'ennemi ayant le moins de points de mouvement.
- killEnemy_if_possible() demande à l'IA d'analyser l'état de jeu actuel, et notamment la vie des ennemies ainsi que les dégâts infligeables pour déterminer si un ennemi est éliminable, et le cas échéant procéder à son élimination.
- set_mode_analysis() permet de passer *analysis* a true et ainsi passer l'IA en mode analyse.
- set_mode_execute() permet de passer *analysis* a false et ainsi passer l'IA en mode exécution de stratégie.
- mode_analysis() renvoie la valeur du booléen *analysis* afin de déterminer si l'IA est en mode analyse ou non.

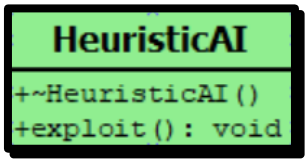


5.3 Conception logiciel : extension pour l'IA composée

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.



5.4 Conception logiciel : extension pour IA avancée

DeepAI

Cette classe hérite de **AI** et a pour rôle la gestion de l'IA avancée.
Elle possède 4 attributs :

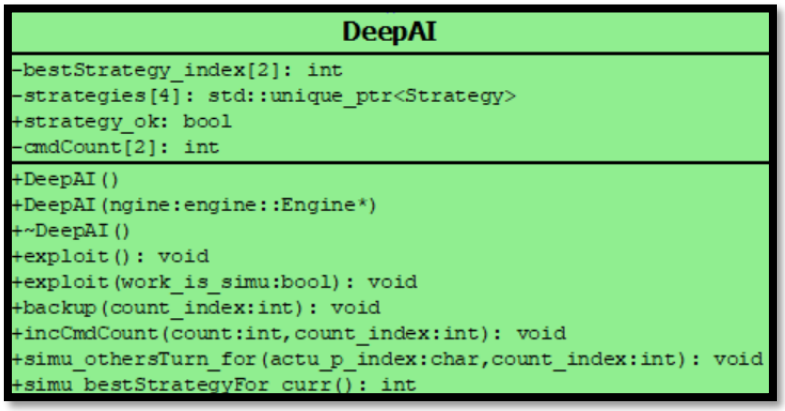
- *bestStrategy_index* de type int contient le numéro de la meilleur stratégie trouvée après analyse de l'état du jeu.
- *strategies* de type std::unique_ptr<Strategy> contient la liste des différentes stratégies que l'IA avancée peut

adopter.

- *mode* de type bool vaut true quand l'analyse du jeu est en cours et false quand l'IA passe en mode exécution.
- *cmdCount* est un int permet de compter le nombre de commandes qui ont été exécutées par l'IA. Cela permet notamment de savoir combien de commandes il faut annuler à la fin de la phase d'analyse.

Elle possède 6 méthodes en plus de ses constructeurs et son destructeur.

- incCmdCount() permet d'incrémenter la valeur de cmdCount lorsqu'un commande est exécutée.
- backup() annule les commandes exécutés selon le type de simulation effectuée (recherche de stratégie ou simulation d'action après avoir trouvé la meilleur stratégie).
- exploit() (avec ou sans paramètres) 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.
- simu_othersTurn_for() permet de permet de simuler le tour des autres joueurs en prenant comme référence le joueur dont l'index est passé en argument.
- simu_bestStrategyFor_curr() permet de tester les différentes stratégies et déterminer la meilleure pour l'IA actuelle.



Strategy

Cette classe hérite de **AI** et a pour rôle la gestion de l'IA avancée.
Elle possède 9 attributs :

- *iteration* se comporte comme un itérateur afin de pouvoir évoluer dans le déroulement de la stratégie.
- *g_ai* de type std::shared_ptr<DeepAI> pointe sur l'IA avancée afin d'actualiser l'affichage au fur et à mesure du combat.
- *judicious_attack* de type std ::queue<char> contient les index des attaques qui ont été privilégiées pour la stratégie.
- *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 requises.
- *p_index* stocke l'index du joueur qui représente l'IA.
- *t_index* stocke l'index du joueur ciblé par la stratégie.
- *t_pos* stocke la position du joueur ciblé par la stratégie.
- *savedPos* la position de destination qui a été privilégiée pour la stratégie.

Elle possède 7 méthodes en plus de son constructeur et son destructeur.

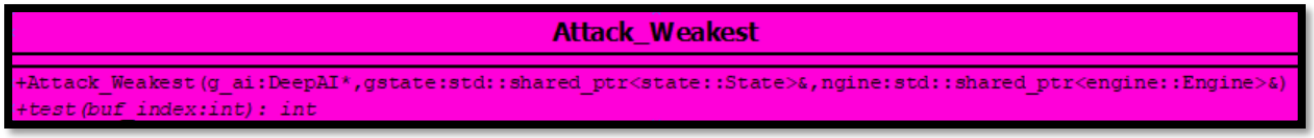
- apply() applique la meilleur stratégie trouvée par la fonction test().
- test() lance une analyse de l'état de jeu et essaye les différentes stratégies pour déterminer laquelle est la meilleure et doit être suivie.
- pick_GoodPosition() permet de trouver la position optimale à laquelle le joueur devrait se placer. Il s'agit de la position la moins couteuse en PM, qui permettrait d'attaquer l'ennemi ciblé par la stratégie.
- chooseBestAttck() permet de déterminer la meilleure attaque à utiliser pour la situation analysée par l'IA.
- simulate_attack() permet comme son nom l'indique de simuler une attaque en actualisant une copie des stats des personnages impliquées : mise à jour des HP du joueur attaqué et des PA du joueur attaquant.
- start_simulation() lance la simulation de la stratégie pour déterminer son efficacité.

- popQueues() permet de retirer des différentes queue les informations situées à l'index passé en argument.

Comme nous l'avons évoqué pour cette intelligence artificielle plusieurs stratégies sont envisageables afin d'obtenir les meilleures performances lors du tour. Chaque méthode possède une méthode test() (voir description dans la classe parente **Strategy**) en plus de leur constructeur. La différence majeure entre ces stratégies est la cible, l'ennemi que l'IA doit chercher à attaquer et éliminer. Nous allons les passer en revue :

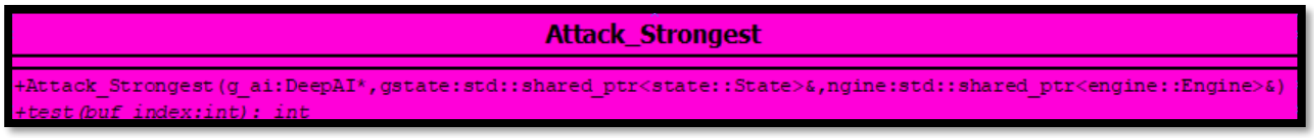
Attack_Weakest

Cette stratégie cherche à attaquer l'ennemi le plus faible c'est-à-dire celui qui a le niveau le moins élevé. En effet cette stratégie peut être particulièrement intéressante si l'on considère une situation comme la suivante :
Imaginons qu'une IA de niveau 40 est face à 2 ennemies, l'un est de niveau 50 mais n'a que 100 points de vie (ennemi A) tandis que l'autre est seulement de niveau 30 mais à 200 points de vie (ennemi B). Dans ce genre de jeu, la meilleure stratégie est d'éliminer les ennemis dès que possible pour réduire leur nombre et ainsi limiter les dégâts subis par la suite. On pourrait ainsi se dire qu'il faut attaquer l'ennemi A comme il ne lui reste que peu de HP, cependant étant de niveau 50 sa résistance/ son shield est bien plus élevé que celui du joueur B. L'IA infligera donc davantage de dégâts au joueur B qu'au joueur A, c'est pourquoi il est intéressant d'avoir une stratégie qui analyse les niveaux des joueurs et s'en prend au plus faible.



Attack_Strongest

Cette stratégie cherche à attaquer l'ennemi le plus fort, celui dont le niveau est le plus élevé. Si l'on reprend le fait qu'il faut chercher à éliminer au plus vite la menace la plus grande, on comprend rapidement en quoi cette stratégie est particulièrement adaptée. En effet, tout comme dans la situation précédente le joueur A avait plus de shield que le joueur B, son attaque est également bien plus élevée. Un joueur de haut niveau est une sérieuse menace comme il infligera d'importants dégâts lors de son tour, c'est pourquoi sauf cas exceptionnels (ennemi de faible niveau éliminable qui impliquerait donc la stratégie précédente) il faut privilégier une stratégie qui s'en prend à l'ennemi ayant le plus grand niveau.



Attack_Most_InDanger

Attack_Most_InDanger est une stratégie dans laquelle l'IA prend pour cible le joueur avec le moins de point de vie. Comme nous l'avons dit il faut éliminer les ennemies dès que possible, ceux ayant très peu de vie sont donc des cibles à privilégier. Cependant comme nous l'avons vu parfois s'en prendre à l'ennemi avec le moins de vie n'est pas forcément la stratégie la plus adaptée si la cible est de haut niveau et donc plus difficilement éliminable qu'un joueur avec plus de points de vie mais un niveau plus faible.

Attack_Closest

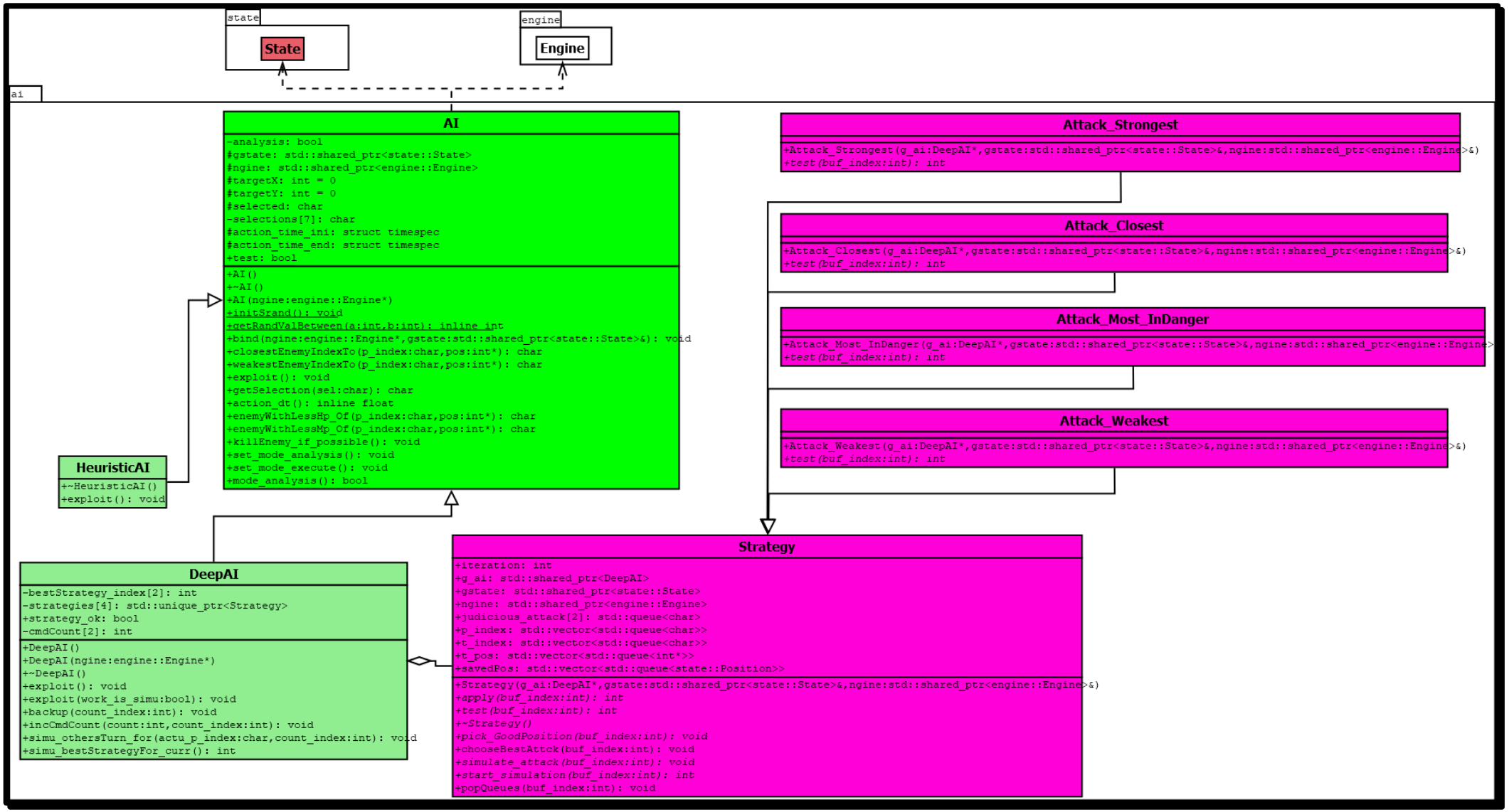
Enfin dans la stratégie Attack_Closest, l'IA va se tourner vers son ennemi le plus proche. Cette stratégie peut être particulièrement efficace notamment si le joueur joué par l'IA possède des sorts à distance. En effet dans ce cas, cette stratégie va permettre de limiter les déplacements vers la cible tout en leur infligeant des dégats et donner l'occasion de conserver la distance qui la sépare d'elle.

5.5 Conception logiciel : extension pour la parallélisation

Nous n'avons pas pu réaliser de parallélisation entre l'engine et le render dans la mesure où notre engine se base sur un système de pattern Command. La parallélisation poserait donc un problème dans la mise en place du rollback.
En effet, si on réalise du multithreading, on aura d'un côté le render qui tourne de manière continue et surveille sans cesse les changements du state, tandis que de l'autre côté l'engine est en train de réaliser les tests des stratégies pour l'IA avancée, crée et exécute des commandes, les annule jusqu'à trouver la bonne stratégie, et toutes ces actions qui sont faites, puisqu'elles modifient temporairement le state, seront rendues visibles à l'écran par le render et c'est ce que l'on souhaite éviter.

Néanmoins nous comptons intégrer de la parallélisation dans notre jeu entre le render et le réseau.

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