# Projet

**Attigui Youness - Quellier Louis**

FIGURE 1 – Exemple du jeu

# Table des matières

# 1 Présentation Générale

## 1.1 Archétype

Pour ce projet nous nous inspirons du gameplay du jeu "Dofus". Il s'agit d'un jeu de stratégie au tour par tour où il est possible de se déplacer de carte en carte pour combattre des ennemis et gagner de l'équipement et de l'expérience permettant d'améliorer son personnage et d'avancer plus loin dans le monde.

## 1.2 Règles du jeu

Vous incarnez un héros auto-proclamé qui vient de fonder sa guilde d'aventurier dans le but de libérer le royaume de l'emprise du roi démon.

L'aventure commence avec une arme basique au choix, combiné avec un élément de prédilection. Ces armes pourront être trouvées ou achetées et gagneront des niveaux selon le nombres d'ennemis vaincus, débloquant ainsi des bonus de statistiques ainsi que des nouvelles actions, pouvant également être améliorées. Ce niveau est inhérent à l'arme et sera alors conservé lors du transfert vers un autre personnage.

Tous les personnages du jeu possèdent en plus de leur arme un ensemble de caractéristiques propres à chacun :

- pv : point de vie (s'ils tombent à 0, le personnage meurt et lâche son matériel sur la case sur laquelle il se trouvait)
- pa : point d'action (permettent d'utiliser les actions associées aux armes)
- pm : point de mouvement (permettent de se déplacer)
- éléments : eau/terre/feu/air

Les différents éléments (l'eau, la terre, le feu et l'air) font partie intégrante de ce monde et sont au coeur des stratégies de combat. Des bonus et malus de type élémentaire sont appliqués selon l'attribut du joueur, de son arme et de la case du terrain sur lequel il est placé. C'est à vous d'utiliser le terrain et vos compétences à votre avantage.

Au cours de l'histoire, d'autres aventuriers rejoindront votre guilde afin de vous aider dans votre tâche et vous serez engagés pour effectuer des missions variées afin de gagner de l'or pour acheter de nouvelles armes ou recruter de nouveaux mercenaires.

En tant que guilde indépendante vous êtes libres de vous déplacer à votre guise dans le royaume pour accomplir les missions dans l'ordre que vous souhaitez. Néanmoins, dans ce monde il n'existe pas d'organisme régissant les missions, celles-ci vous sont confiées lorsque vous arrivez sur les lieux.

Par moments vous devrez défendre une ville contre une attaque adverse, parfois vous devrez protéger un voyageur surpris par une embuscade ennemie ou souhaitant une escorte, d'autres fois vous devrez détruire une base fortifiée mais le plus souvent vous devrez simplement éliminer tous les adversaires présents dans une zone.

Un combat se déroule sur un des terrains de la carte du monde. Chaque terrain est composé d'un nombre fixe de cases. Chaque case est associé à un élément (eau, terre, feu, air, neutre). Le déploiement des personnages s'effectue au début du combat sur une zones prédéterminée dépendant de la direction d'arrivée, la disposition à l'intérieur de cette zone étant libre.

Au cours de votre aventure votre guilde acquiert une certaine notoriété dans le royaume, celle-ci peut aussi bien être positive que négative, influant sur vos interactions avec les différents personnages. Au fil de vos rencontres, vous pourrez refuser certaines missions pour diverses raisons mais cela fera diminuer votre réputation. Le degré de réussite ou d'échec des missions acceptées modifiera également cette réputation.

Si votre guilde a trop mauvaise réputation, vous serez exécutés en tant que criminel et l'aventure s'arrête là. Au contraire une très bonne réputation vous donne le privilège d'accéder à l'armurerie de Vulc contenant des armes surpuissantes mais également excessivement chères...

L'aventure se termine après avoir vaincu l'un des 4 boss élémentaires, vous permettant débloquer son arme associée. Vous pouvez cependant perdre la partie si tous vos personnages meurent ou via le système de réputation. Les armes rencontrées au cours de l'aventure seront stockées dans votre armurerie personnelle, vous permettant de choisir une nouvelle arme de départ en recommençant une aventure. La progression d'expérience est perdue mais l'or est gardé entre chaque partie.

## 1.3 Ressources

FIGURE 2 – Tileset personnages

FIGURE 3 – Exemple tileset ennemis


FIGURE 4 – Exemple tileset terrain

FIGURE 5 – Arbres

FIGURE 6 – Rochers

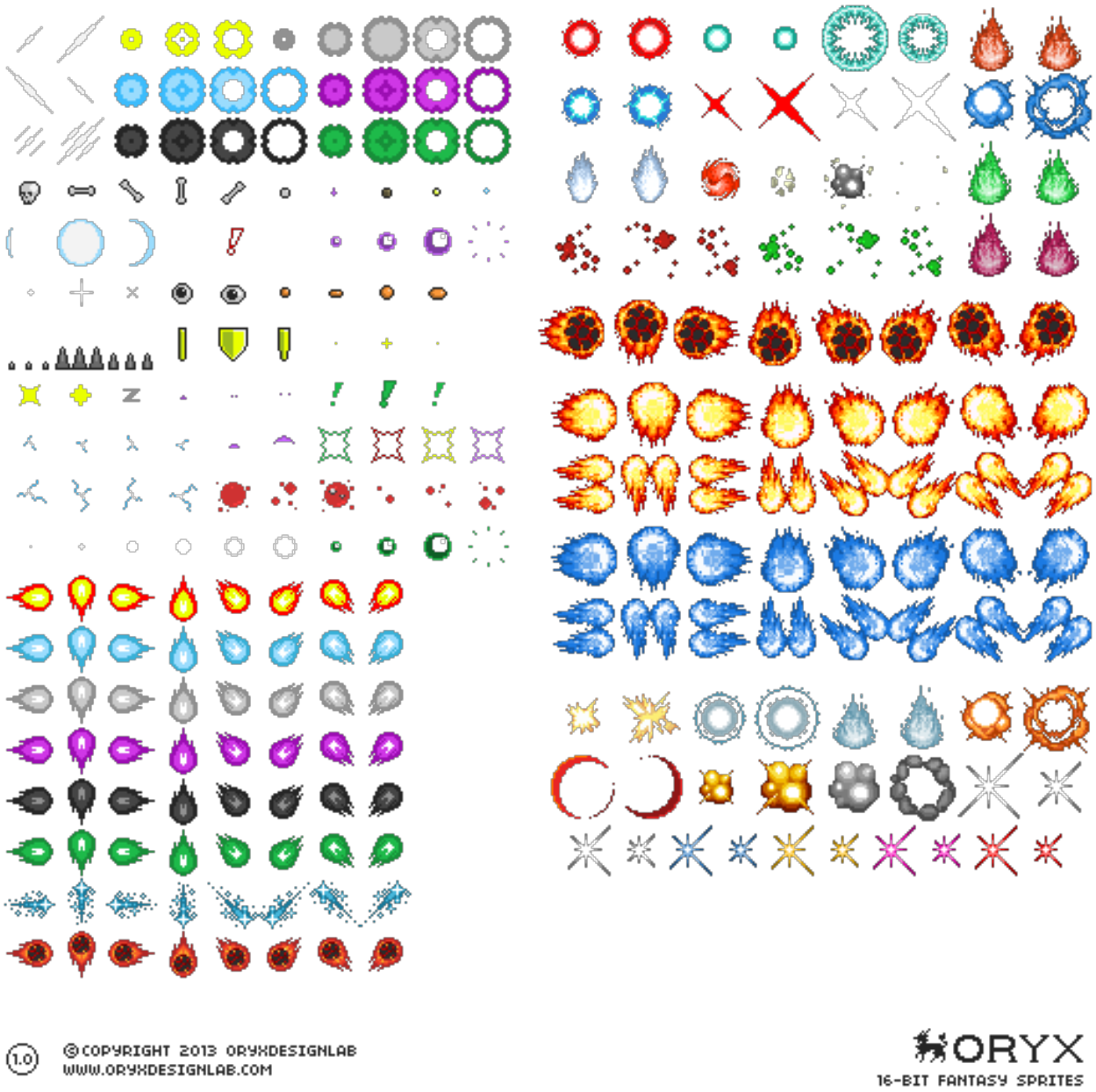


FIGURE 7 – Exemple animations attaques

# 2 Description et conception des états

## 2.1 Description des états

Un état de jeu est formé de la carte du monde possédant des cases d'un certain élément, repérées par ses coordonnées dans le monde, pouvant éventuellement posséder un obstacle infranchissable inamovible ou un personnage mobile. Ce personnage fait partie d'une équipe possédant un inventaire contenant des objets, ceux-ci étant principalement des armes pouvant être équipées individuellement et possédant diverses capacités et caractéristiques. L'équipe du joueur peut combattre et obtenir des quêtes fournissant des récompenses lors de leur accomplissement. Le détail des attributs et opérations des différentes classes est présenté ci-dessous.

### 2.1.1 State

— L'état possède une matrice de pointeurs de cellules qui contiennent toutes les information liées au terrain et à la position des personnages

— Il possède aussi toutes les équipes de personnages, ainsi que les quêtes et le combat éventuel qui se déroule.

— L'heure correspondant au nombre de fronts d'horloge, ainsi que la vitesse à laquelle le jeu est actualisé permet d'associer un temps à chaque état du jeu

— Un booléen indique si l'inventaire est ouvert

### 2.1.2 Cell

— La case doit contenir un élément choisi parmis une énumération et peut contenir un obstacle infranchissable par les personnages

### 2.1.3 Character

— Les coordonnées du personnage et la direction dans laquelle il se tourne

— Le personnage a un nom et appartient à une race qui sera utile par la suite pour les quêtes notamment

— Il possède également des attributs utiles au combat lui permettant de se déplacer, d'attaquer et de pouvoir encaisser des attaques

— Chaque personnage possède enfin une arme lui permettant d'effectuer diverses actions

### 2.1.4 Team

— L'équipe a un nom

— Elle comprend l'ensemble des personnages aptes au combat. Seul un combattant représente l'équipe hors combat lors du déplacement sur la carte du monde, et seul un nombre limité de personnages peuvent combattre simultanément

— L'équipe stocke également l'inventaire des objets récupérés, pouvant être affectées aux différents personnages de l'équipe

#### 2.1.5 Inventory

— L'inventaire comprend différents objets, entre autres les armes affectées aux combattants et celles en réserve

#### 2.1.6 Weapon

— L'arme peut posséder un nom
— L'arme possède des capacités permettant d'attaquer ou de se défendre
— Elle possède également un élément de prédilection répercuté sur son porteur

#### 2.1.7 Ability

— Les habilités peuvent avoir un nom, coûtent un certain nombre de points d'action, infligent un certain nombre de dégâts (ou de soin en cas de valeur négative) et possèdent éventuellement un temps de recharge de la capacité
— Elles possèdent un élément répercuté sur l'attaque en elle même. Elle sera souvent du même élément que l'arme, mais pourront parfois différer
— L'attaque possède une portée effective, et une zone d'effet dépendant du type d'arme utilisé
— Une réduction de dégât peut être envisagée pour les attaques de zone afin de moins affecter les personnages éloignés du point d'impact

#### 2.1.8 Fight

— Le combat enregistre le nombre de tours passés afin de définir la prochaine équipe à jouer
— Il possède les différentes équipes se battant afin que chaque membre de l'équipe puisse se déployer avant de commencer

#### 2.1.9 MainQuest

— Les quêtes principales s'obtiennent avant d'entrer en phase de combat en effectuant certaines actions et peuvent se décliner en divers objectifs à remplir à travers plusieurs combats à travers ses classes filles

### 2.2 Conception Logiciel

Le diagramme des classes pour les états est présenté en Figure 8 :

— La classe en vert contient les information globales sur le déroulement du jeu et permettent de déterminer l'état actuel du jeu

— Les classes en jaune caractérisent les personnages et le terrain dans lequel ils évoluent

— Les classes en bleu déterminent les actions possibles par les différents personnages

— Les classes en rouge permettent d'implémenter les phases de combat ainsi que les quêtes, permettant d'obtenir du matériel ou de nouveaux membres d'équipe.

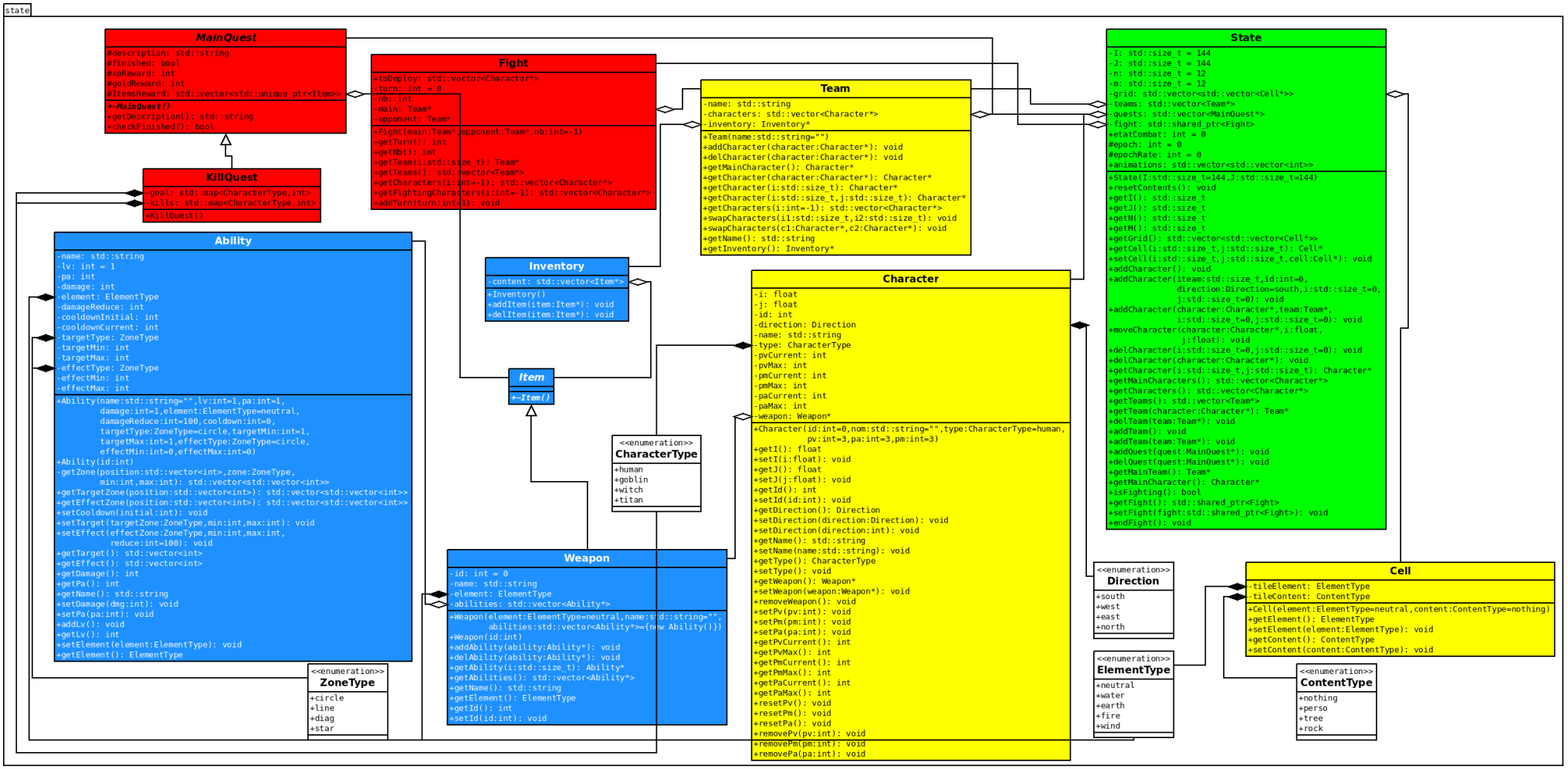


FIGURE 8 – Diagramme des classes d'état.

# 3 Rendu : Stratégie et Conception

## 3.1 Stratégie de rendu d'un état

Le rendu de l'état est possible grâce aux différentes informations récupérées de la classe World, c'est à dire les tuiles et les personnages. Il s'agit donc d'un rendu à 3 couches successives. Toutes les textures sont chargées et découpées afin de renvoyer le sprite correspondant à un personnage donné ou à une cellule du monde donnée (élément et contenu optionnel). Ce sprite est ensuite déplacé au lieu d'en générer un nouveau. Un système de vue centre la fenêtre sur une petite zone autour du personnage principal (encadré d'un carré blanc) et n'actualise l'affichage des sprites sur cette vue uniquement et pas sur le monde entier. Cela évite d'une part de charger des sprites inutilement et d'autre part de créer une carte du monde où les zones pas encore visitées ne seraient pas visibles (ce serait alors une vue du monde en entier).

## 3.2 Conception logiciel

Le diagramme de classes du rendu est présenté en Figure 10 :

— La classe Sprites est le gestionnaire des différentes ressources. Elle charge les différentes textures du jeu lors de l'instanciation et génère les différents sprites de résolutions paramétrables, compatibles avec l'état du monde, dépendant alors de la tuile à afficher (élément + contenu) et du personnage à afficher. Le SpriteGeneratorById permet de générer les sprites pour les différentes fenêtres d'interface joueur.

— Le Render réutilise alors les sprites générés pour afficher la carte grâce au RenderWindow de sfml et gère la View correspondante via display().

— l'interface utilisateur se base sur la classe Element. Cette classe permet de créer des conteneurs pour les objets/éléments sfml (text, sprite et rectangle). la classe éléments applique le design pattern composite, ainsi un élément peut être un "parent" et contenir des éléments "enfants" qui vont automatiquement se positionner et se dimensionner en fonction de la position et taille du parent. Un éléments parent peut avoir comme enfants d'autre élément parent, ainsi les éléments s'organise selon une structure d'arbre. La classe élement permet également d'encapsulter la gestion des évenements sfml. Voici quelques élements particulier : Window (affiche une fenêtre déplacable et fermable), CharacterStat (affiche les stats d'un perso), CharacterSheet (affiche les informations lié à un personnage)

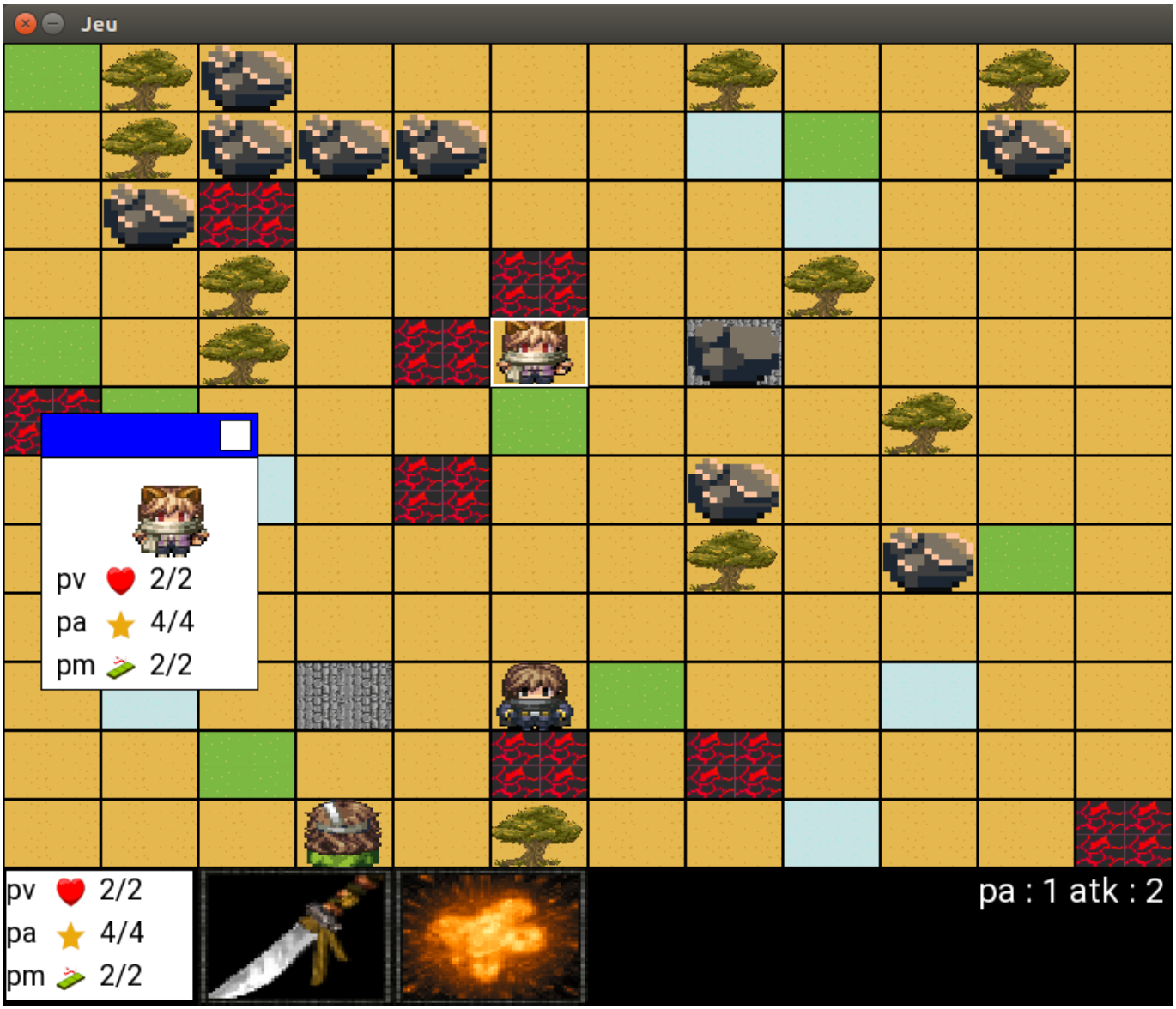


FIGURE 9 – Preview du rendu

```
N6render9RectangleE 11 Mouse over
*this abs:   (  0,   0) (700, 700)

    N6render6WindowE 12
    parent abs:  (  0,   0) (700, 700)
    *this rel:   ( 50,  50) (200, 100)
    *this abs:   ( 50,  50) (200, 100)

        N6render9RectangleE 13
        parent abs:  ( 50,  50) (200, 100)
        *this rel:   (  0,   0) (100%,  30)
        *this abs:   ( 50,  50) (200,  30)

            N6render4TextE 14
            parent abs:  ( 50,  50) (200,  30)
            *this rel:   (  5,   5) ( 10,  10)
            *this abs:   ( 55,  55) ( 10,  10)

            N6render9RectangleE 15
            parent abs:  ( 50,  50) (200,  30)
            *this rel:   ( -5,  -5) ( 20,  20)
            *this abs:   (225,  55) ( 20,  20)

        N6render9RectangleE 16
        parent abs:  ( 50,  50) (200, 100)
        *this rel:   (  0,  30) (100%, -30)
        *this abs:   ( 50,  80) (200,  70)

    N6render6WindowE 17
    parent abs:  (  0,   0) (700, 700)
    *this rel:   (400,  50) (200, 100)
    *this abs:   (400,  50) (200, 100)

        N6render9RectangleE 18
        parent abs:  (400,  50) (200, 100)
        *this rel:   (  0,   0) (100%,  30)
        *this abs:   (400,  50) (200,  30)

            N6render4TextE 19
            parent abs:  (400,  50) (200,  30)
            *this rel:   (  5,   5) ( 10,  10)
            *this abs:   (405,  55) ( 10,  10)

            N6render9RectangleE 20
            parent abs:  (400,  50) (200,  30)
            *this rel:   ( -5,  -5) ( 20,  20)
            *this abs:   (575,  55) ( 20,  20)

        N6render9RectangleE 21
        parent abs:  (400,  50) (200, 100)
        *this rel:   (  0,  30) (100%, -30)
        *this abs:   (400,  80) (200,  70)
```

Listing 1 – Structure d'abre de Element

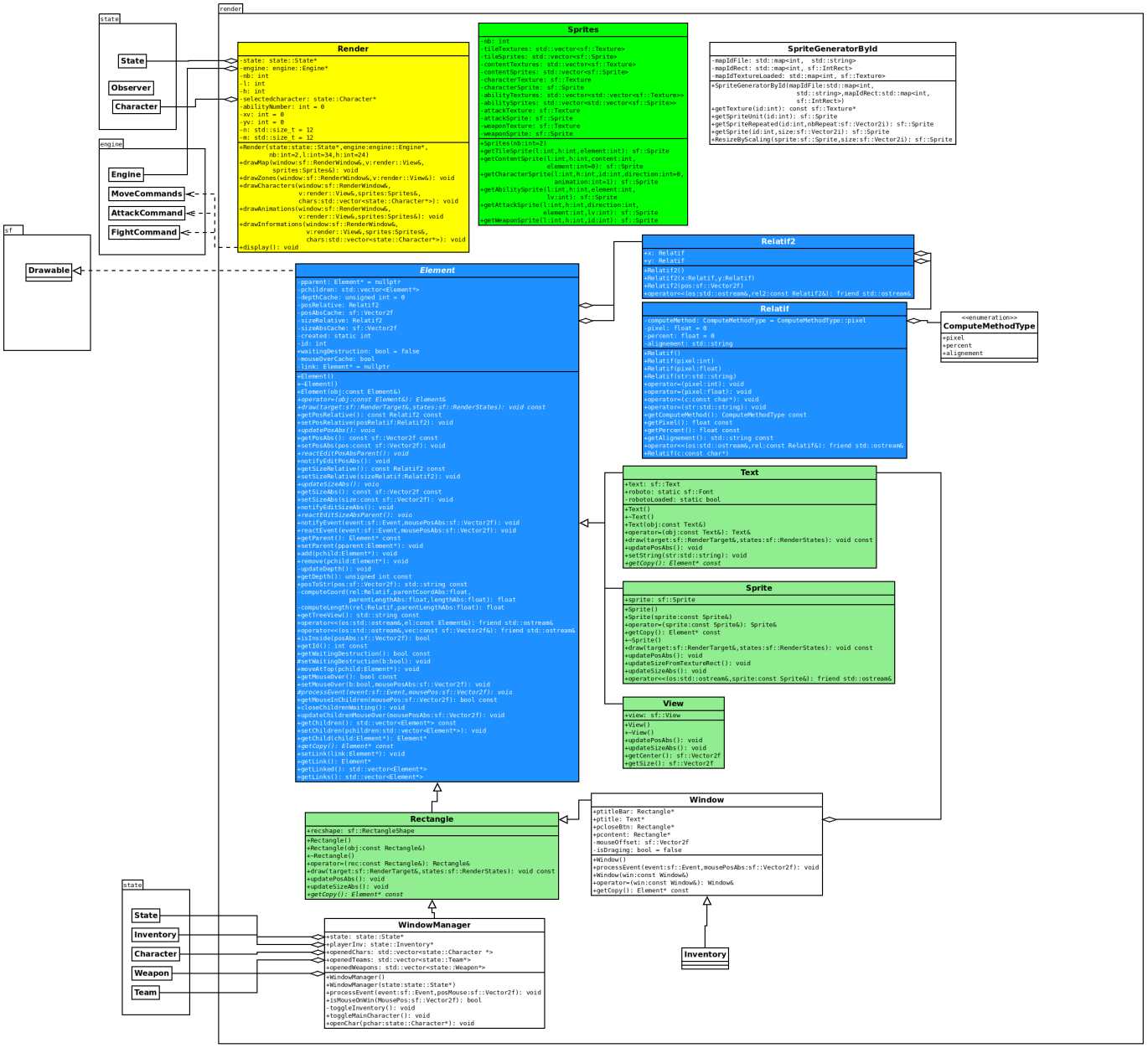


FIGURE 10 – Diagramme des classes de rendu.

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

## 4.1 Horloge globale

Les commandes sont exécutées à une fréquence de 30 Hz et chaque animation de déplacement est constitué d'une étape de changement de direction ainsi que de 12 étapes de déplacement pur séparés en 4 sprites différents.

## 4.2 Changements extérieurs

Les changements extérieurs sont le déplacement du personnage, que ce soit en combat ou non, ainsi que les phases d'attaque en combat.

## 4.3 Changements autonomes

Les changements autonomes sont le changement de direction du personnage lors de son déplacement, l'affichage des sprites fx lors de l'attaque ainsi que le démarrage de la phase de combat permettant de déployer tous les personnages de chaque équipe si un ennemi est rencontré lors de la phase d'exploration.

## 4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Figure 11. Le moteur de jeu repose sur le pattern Command et permet la mise en oeuvre différée des commandes modifiant l'état du jeu.

**Classes Command** Ces classes représentent les commandes exécutées par le moteur.

— MoveCommands permet de générer l'ensemble des commandes individuelles MoveCommand afin de créer une animation de déplacement en utilisant un pathfinding A* (rapide mais ne générant pas nécessairement le chemin le plus optimal) et FightCommand en cas de rencontre avec un ennemi, ce qui lance la phase de combat.

— AttackCommand permet aux différents personnages d'attaquer avec leurs capacités et ainsi d'infliger des dégâts à différentes cibles et génère des commandes DamageCommand pour générer les dégâts ainsi que les animations d'attaque.

**Engine** Le moteur exécute les commandes dans l'ordre d'apparition. Il enregistre également les commandes dans un rollback pour pouvoir les inverser pour l'IA avancée, ou encore pour proposer le choix d'annuler un tour au joueur.

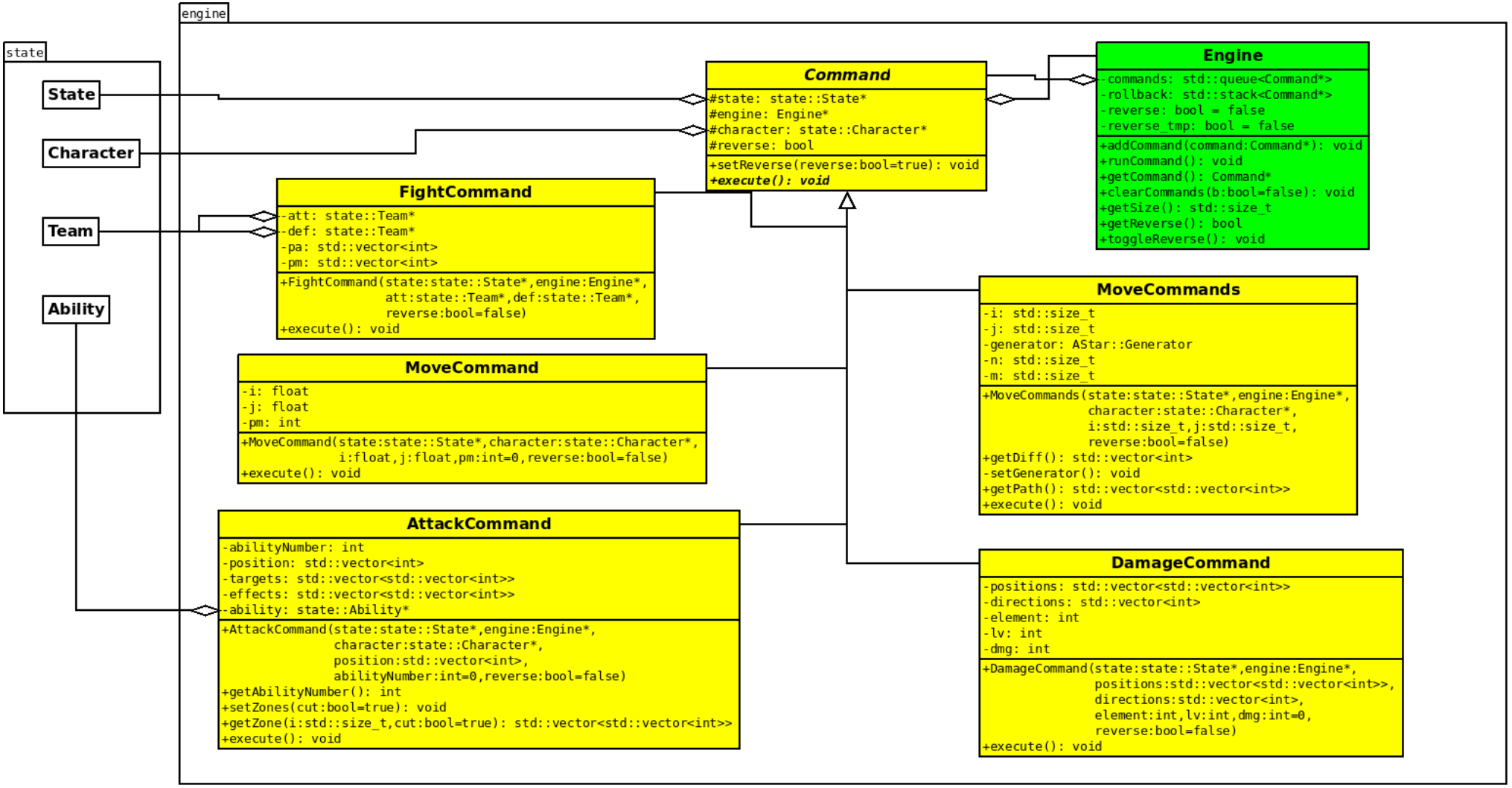


FIGURE 11 – Diagramme des classes du moteur.

# 5 Intelligence Artificielle

## 5.1 Stratégies

### 5.1.1 Intelligence aléatoire

L'IA execute des commandes aléatoires parmi le mouvement et l'attaque, s'il lui reste assez de pa/pm.

### 5.1.2 Intelligence basée sur des heuristiques

L'IA balaye toutes les positions possibles du personnage et pour chaque position va affecter un Score à chaque attaque possible à partir de cette position : pour chaque dégât réellement infligé à un ennemi (respectivement un allié) le score augmente (respectivement diminue) de ce montant (un soin est considéré comme des dégâts négatifs). Si un personnage meurt suite à cette attaque, le score est majoré (respectivement minoré), mais si les dégâts sont beaucoup plus élevés que la vie du personnage, un malus est appliqué pour éviter le gaspillage des ressources. Dans une plus faible proportion, plus l'action coûte de ressources en pa, plus le score diminue, mais plus l'IA se rapproche d'un ennemi, plus le score augmente. Ces différences de score sont minimes devant le score de dégât afin de différencier de manière plus précise des actions de score similaire, et permettent de se rapprocher des ennemis si aucune action n'est possible. Une liste de toutes les commandes de même score maximal est établie et une commande aléatoire parmi ces meilleures commandes (déplacement puis attaque) est envoyée au moteur jusqu'à épuisement des pa et pm de chaque joueur (ou la fin du combat).

### 5.1.3 Intelligence avancée

L'IA balaye plusieurs options parmi ses meilleurs coups (actions avec un score au dessus d'un certain seuil fixé) grâce à un arbre de profondeur fixé, représenté par la classe TreeNode, mais cette fois elle peut anticiper les mouvements adverses (qui soustraient leur score au noeud de l'arbre) et permettre de changer plusieurs fois de personnage au lieu de les faire jouer séquentiellement. Le système de score reste identique à celui de l'intelligence heuristique. Un système de rollback permet d'annuler les actions éxécutées de proche en proche, modifiant directement l'état, au lieu de stocker plusieurs copies de l'état actuel.

## 5.2 Conception logiciel

Le diagramme de classes pour l'intelligence artificielle est présentée en figure 12

**Classes AI** Les classes filles s'organisent selon un pattern strategy :

- RandomAI : IA aléatoire
- HeuristicAI : IA heuristique
- HeuristicAI : IA avancée

**Score** Cette classe calcule et stocke le score d'une action avec les critères expliqués dans la section de l'intelligence heuristique
**TreeNode** Chaque noeud possède un parent (sauf la racine de l'arbre), une liste d'enfants, un score et les commandes exécutées pour arriver à ce noeud dans l'arbre. Il est donc associé à un état virtuel du jeu.

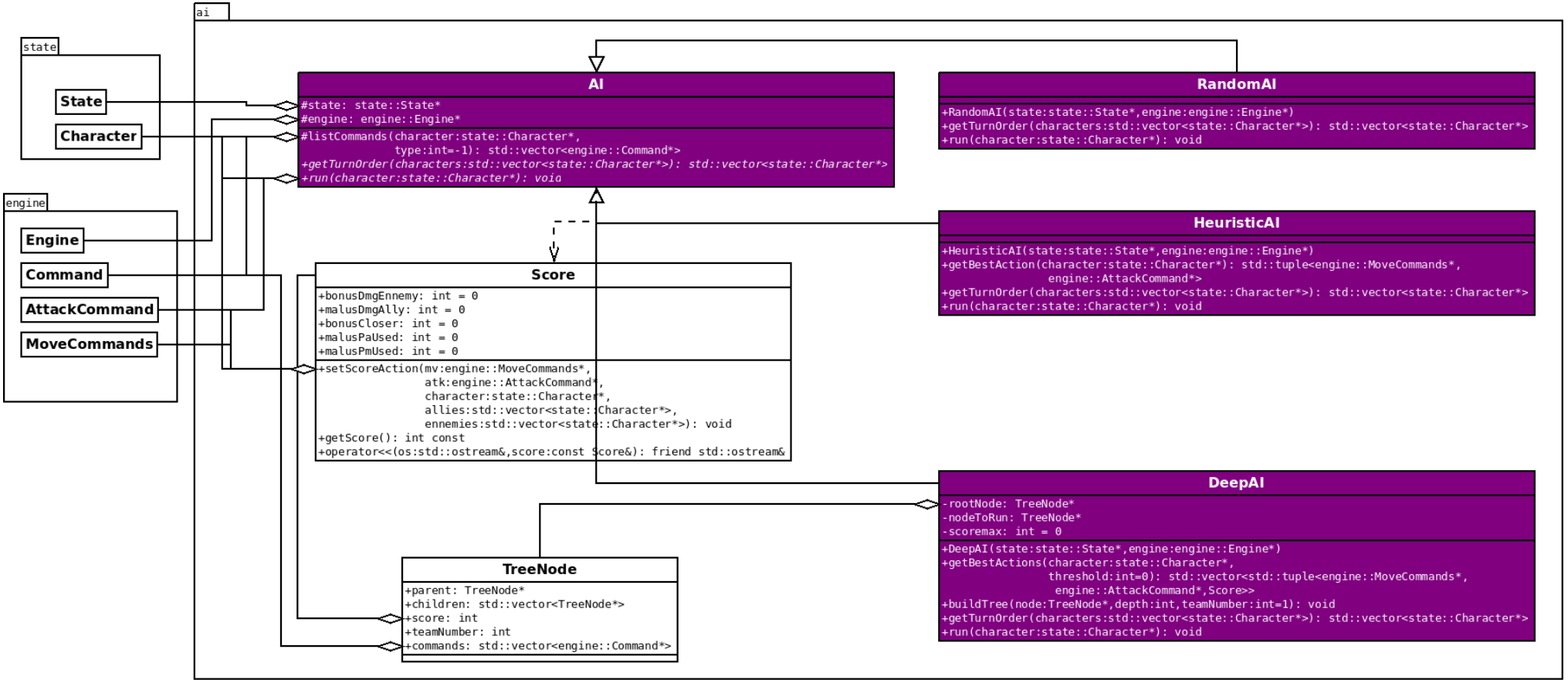


FIGURE 12 – Diagramme des classes d'intelligence artificielle.

# 6 Modularisation

## 6.1 Organisation des modules

## 6.2 Conception logiciel