



---

## Projet Logiciel Transversal

Patrick Contin, William Duval Bourlignieux, Bastien Guillard, Abdel-Oihed Houta

---



# Table des matières

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Présentation Générale</b>                         | <b>1</b>  |
| 1.1      | Archétype . . . . .                                  | 1         |
| 1.2      | Règles du jeu . . . . .                              | 1         |
| 1.2.1    | Stat des personnages . . . . .                       | 1         |
| 1.2.2    | Classe des personnage . . . . .                      | 1         |
| 1.3      | Ressources . . . . .                                 | 1         |
| <b>2</b> | <b>Description et conception des états</b>           | <b>2</b>  |
| 2.1      | Description des états . . . . .                      | 2         |
| 2.1.1    | L'état de la carte du jeu . . . . .                  | 2         |
| 2.1.2    | L'état du joueur . . . . .                           | 2         |
| 2.1.3    | L'état des personnages . . . . .                     | 2         |
| 2.1.4    | L'état général du jeu . . . . .                      | 2         |
| 2.2      | Conception Logiciel . . . . .                        | 3         |
| <b>3</b> | <b>Rendu : Stratégie et Conception</b>               | <b>5</b>  |
| 3.1      | Stratégie de rendu d'un état . . . . .               | 5         |
| 3.2      | Conception logiciel . . . . .                        | 5         |
| <b>4</b> | <b>Règles de changement d'états et moteur de jeu</b> | <b>7</b>  |
| 4.1      | Règles . . . . .                                     | 7         |
| 4.1.1    | Actualisation . . . . .                              | 7         |
| 4.1.2    | Règles extérieurs . . . . .                          | 7         |
| 4.1.3    | Règles automatiques . . . . .                        | 7         |
| 4.2      | Conception logiciel . . . . .                        | 9         |
| <b>5</b> | <b>Intelligence Artificielle</b>                     | <b>11</b> |
| 5.1      | Stratégies . . . . .                                 | 11        |
| 5.1.1    | Intelligence aléatoire . . . . .                     | 11        |
| 5.1.2    | Intelligence heuristiques . . . . .                  | 11        |
| 5.1.3    | Intelligence avancé . . . . .                        | 11        |
| 5.2      | Conception logiciel . . . . .                        | 12        |
| 5.2.1    | IA . . . . .   | 12        |
| <b>6</b> | <b>Modularisation</b>                                | <b>14</b> |
| 6.1      | Organisation des modules . . . . .                   | 14        |
| 6.2      | Conception logiciel . . . . .                        | 15        |

# 1 Présentation Générale

## 1.1 Archétype

Le jeu est un tactical RPG, basé sur des jeux tels que Final Fantasy tactics, Fae tactics ou encore la série des Fire Emblem.

## 1.2 Règles du jeu

Le jeu se déroule sur une map la forme d'une grille, celle-ci voit s'affronter 2 équipes de plusieurs personnages. Chaque personnage peut se déplacer et interagir (attaquer, soigner, etc...) avec les autres sur la map. La victoire est déclarée quand l'ensemble des membres d'une équipes sont KO (PV = 0).

Les personnages commencent avec 0 de mana et en gagnent un peu en début de chaque tour, les sorts ont différents couts de mana en fonction de leur puissance.

### 1.2.1 Stat des personnages

| Nom en jeu        | Effet  |
|-------------------|--|
| PV(Point de Vie)  | Point de vie du personnage, il est KO s'ils tombent a 0.       |
| PM(Point de Mana) | Point de Mana, consommés par les capacités.                    |
| Attaque           | Attaque physique d'un personnage.                              |
| Armure            | Défense physique d'un personnage. Résiste aux dégats physique. |
| Magie             | Puissance d'effets des capacités qui consomme du mana.         |
| Ténacité          | Défense magique d'un personnage. Résiste au dégats magique     |
| Vitesse           | Permet de déterminer l'ordre des tours                         |
| Mobilité          | Nombre de case pouvant être parcouru en 1 seul tour.           |
| Esquive           | Chance d'esquive du personnage.                                |

### 1.2.2 Classe des personnage

Les classes sont réparties en plusieurs categories selon leur utilités (degat, tank, support) et leur portées (mêlé et porté). De base il y aurait 3 classes :

- un guerrier tank : peu de mobilité, vitesse, beaucoup de défense et de vie. Son attaque de base est un coup d'épée au corps a corps, avec un sort de protection, et un sort qui force un ennemi a l'attaquer.
- un mage support : peu de mobilité, peu de défense, peu de dégât, moyenne portée, ses sorts permettent d'augmenter les stats des alliées et de les soignées.
- un archer qui fait des dégats : peu de défense, vitesse moyenne, beaucoup de portée et de dégats, un sort qui fait beaucoup de dégât sur une seule cible, et un sort qui fait des dégât de zone.

## 1.3 Ressources

Pour les ressources de ce projet, nous avons réaliser la carte du jeu avec le logiciel Tiled (Voir dossier res).

## **2 Description et conception des états**

### **2.1 Description des états**

Un état de jeu a besoin de 3 informations, la carte du actuelle du jeu, les personnages et leur état, ainsi que les joueurs et leur état.

#### **2.1.1 L'état de la carte du jeu**

La carte du jeu est composé d'une liste de cellule, qui compose la carte. Chaque cellule peut être, soit :

- Être vide, juste le sol de la case, avec rien dessus
- Avoir un personnage dessus, peut importe son état
- Avoir un obstacle (arbre, rocher, ou autre)

Il y a plusieurs carte de jeu prédéfini.

#### **2.1.2 L'état du joueur**

Le joueur possède une liste de personnage qu'il possède, avec lesquels il peut jouer. Il possède aussi un état, qui indique si :

- il est encore en jeu
- il a gagné sa partie
- il a perdu sa partie
- il ne joue plus au jeu (un timer compte, si le joueur prend trop de temps à jouer)

#### **2.1.3 L'état des personnages**

Les personnages de chaque joueur possède un ensemble de statistiques :

- Leur point de vie (PV), permet de déterminer combien de coup le personnage peut prendre avant de mourrir
- Leur attaque (ATK), permet de déterminer combien de dégats le personnage va faire avec son attaque de base
- Leur attaque magique (MAG), permet de déterminer combien de dégats le personnage va faire avec ses sorts
- Leur défense magique (RM), permet de réduire les dégats pris par les sorts
- Leur défense physique (DEF), permet de réduire les dégats pris par les attaques de base
- Leur vitesse (VIT), permet de déterminer l'ordre des personnages
- Leur mobilité (MOB), permet de déterminer la quantité de case que le personnage peut se déplacer
- Leur esquive (ESQ), permet d'éviter les attaques et les sorts

Chaque personnage possède également un nom, un compteurs de tours, ainsi qu'une liste de sorts qu'il peut lancer. Il possède aussi une liste d'effets qui va être remplie aux cours de la partie, au fur et a mesure que des effets lui sont appliqué.

#### **2.1.4 L'état général du jeu**

L'état général du jeu permet de compter le nombre de tours passé, de joueurs encore en jeu. Ainsi que l'index du joueur qui est en train de jouer, et une liste de tout les personnages encore en jeu.

## 2.2 Conception Logiciel

Le diagramme des classes pour les états est présenté en Figure 1. En rouge foncé on distingue la classe "état" principal, alors qu'en rouge plus clair on voit les classes lié aux personnages et la carte du jeu. En blanche on a les énumérations, qui sont utilisé pour ne pas utiliser de "magic numbers", et de permettre une transparence sur l'utilisation des variables et de leur différents états.

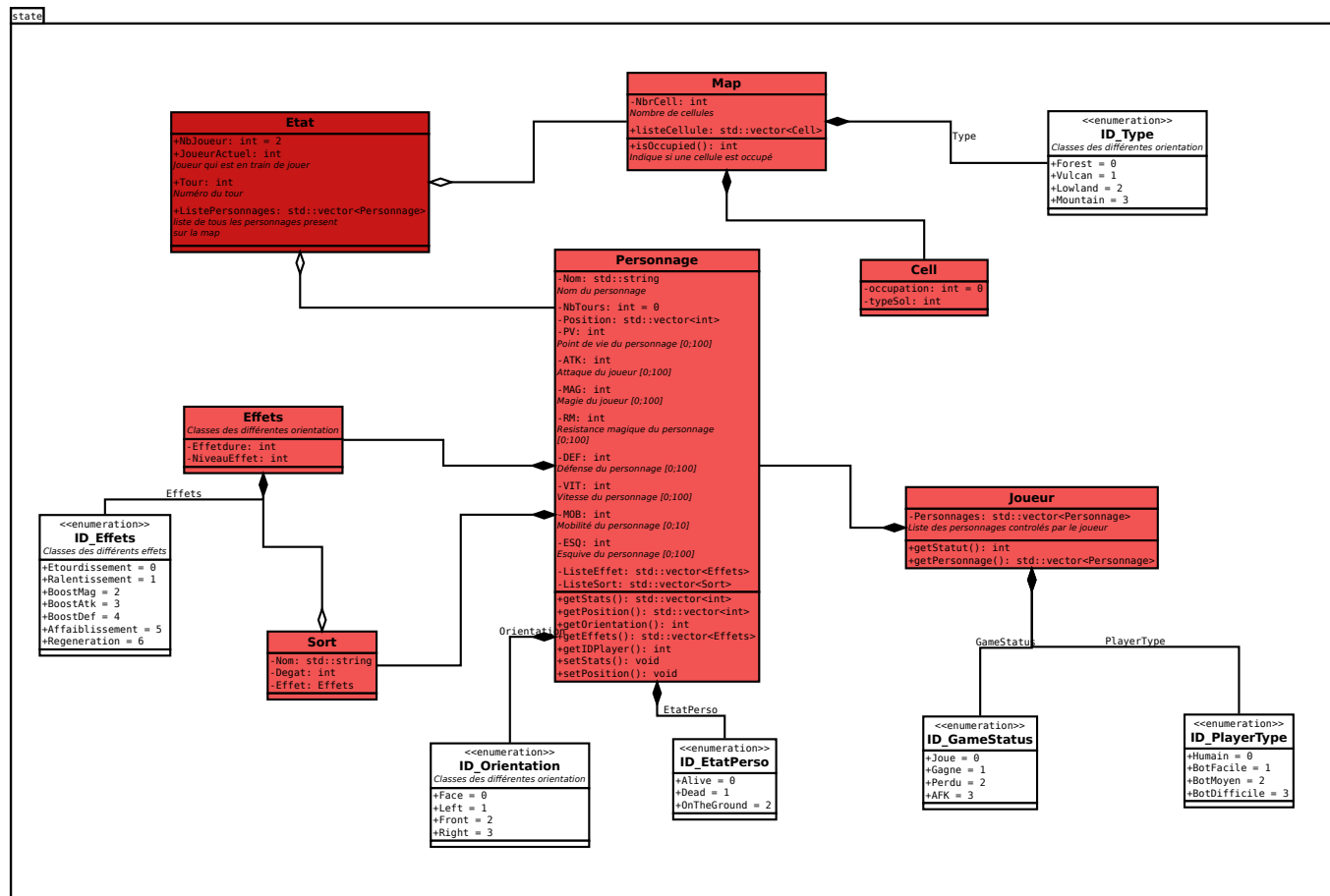


FIGURE 1 – Diagramme des classes d'état.

## 3 Rendu : Stratégie et Conception

### 3.1 Stratégie de rendu d'un état

La stratégie pour laquelle nous avons choisis d'opter est celle utilisée dans l'exemple de M.Gosselin. Cette stratégie simple est basée sur un développement de bas niveau avec des éléments simples comme des textures, couches, etc..

En effet, nous avons choisi de représenter un état suivant 3 étapes :

- la map
- les personnages
- les objets

**Map** La première étape concerne la map, nous avons utilisé le tutoriel dans la documentation SFML. Pour éviter de recharger la map à chaque coup d'horloge, nous décidons de l'afficher sur une couche (layer). La map sera quadrillée sous forme de matrice sur laquelle on vient attribuer des textures grâce à des tuiles. Ainsi les textures seront fixées et la map ne sera plus rechargée, ce qui permettra d'alléger la charge du CPU et de la carte graphique.

**Personnages** Les personnages sont créés grâce à une liste d'objets instanciés avec la classe personnage du state. Ces personnages seront placés sur la map selon leurs positions définies par les attributs de la classe Position du state. Concernant les textures elles seront chargées grâce à des sprites définis dans la bibliothèque SFML. Cela demande plus de ressources que la méthode pour afficher la map mais comme nous avons peu de personnages, ce n'est pas un problème. Le rendu s'actualise en permanence suivant une fréquence que l'on définira par la suite sachant que 50/60 Hz est la fréquence de rafraîchissement habituelle.

**Objets** Les objets seront traités uniquement si la partie map et personnages sont fonctionnelles. Les objets pourront affecter le rendu des personnages (un personnage peut changer de couleur s'il obtient un boost d'attaque) ce qui peut compliquer le code. Ainsi les objets seront traités derniers suivant la même méthode que les personnages (avec des sprites).

### 3.2 Conception logiciel

Le diagramme du rendu est disponible ici : 2.

La classe **StateLayer** va nous permettre de créer les différentes couches pour les différents niveaux. Elle permet aussi d'avoir un affichage graphique avec à l'instanciation d'une fenêtre graphique de la bibliothèque SFML. Elle récupère des informations de toutes les autres classes et gère l'ensemble du rendu.

La classe **LoadLayer** va nous permettre de texturer nos couches avec l'utilisation de quads, ensemble de 4 coins formant un rectangle, qui permet de cadrer la map pour y associer des textures.

La classe **Tiles** permet de gérer les tuiles. Mais surtout, elle permet de récupérer les ressources sous forme d'images que nous associons aux tiles. Sans ceci la texture est impossible.

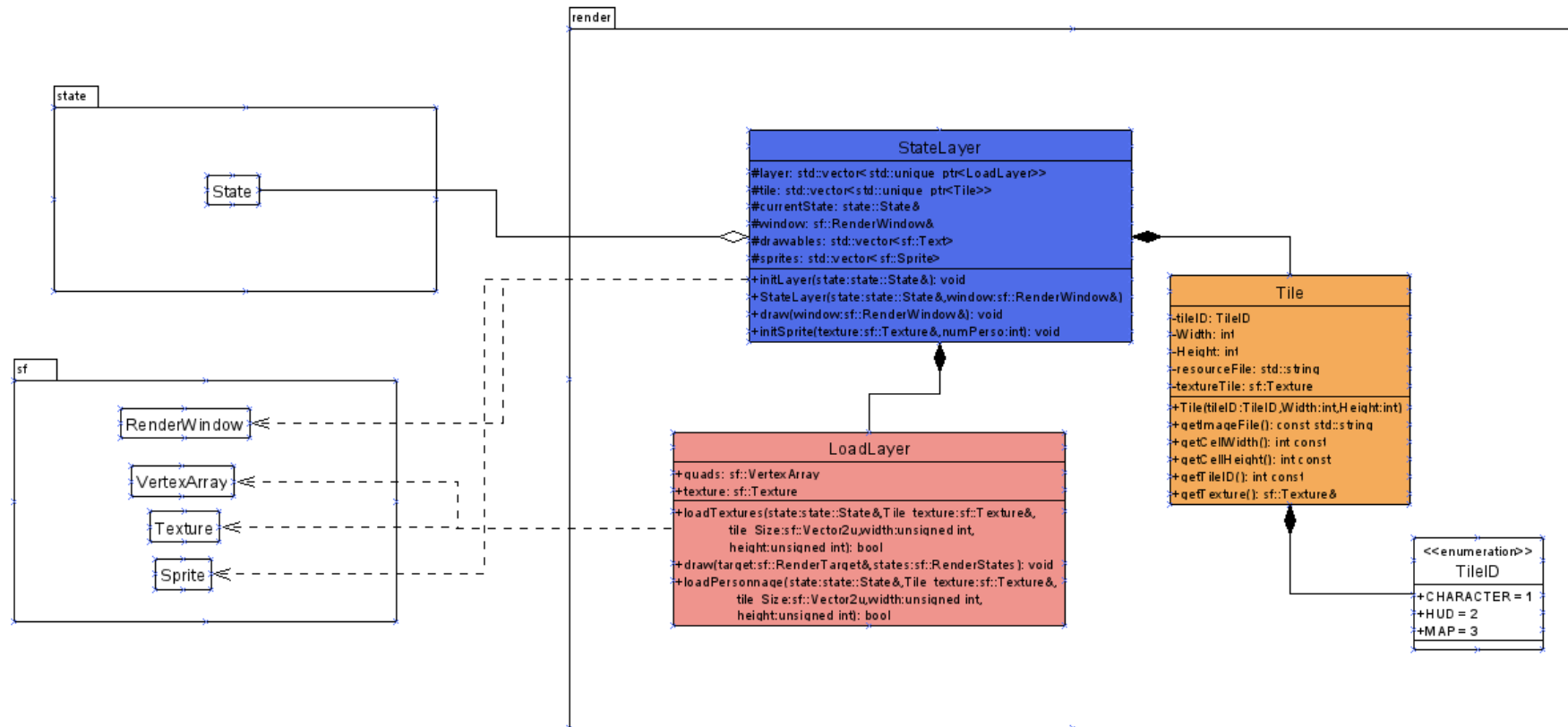


FIGURE 2 – Diagramme des classes de rendu.



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

### 4.1 Règles

#### 4.1.1 Actualisation

L'engine est complètement indépendant du render, le moteur de jeu ne sert qu'à mettre à jour l'état du jeu, en fonction des commandes extérieures (Clavier, souris ou serveur), et des règles automatiques, qui sont vérifiées à chaque changement d'état. Le moteur de jeu effectue une mise à jour de l'état à chaque commande extérieure, les règles automatiques n'ont besoin de s'activer qu'uniquement dans le cas d'un changement d'état (jeu tour par tour). Il existe aussi une règle autonome qui doit s'incrémenter entre les tours, afin de déterminer l'ordre de passage des tours des différents personnages. Cette commande sera appelée par une horloge.

#### 4.1.2 Règles extérieures

Les règles extérieures sont provoquées par les clics de souris à différents endroits, ou les ordres reçus par le réseau.

- Sélectionner un personnage, afin d'afficher les statistiques du personnage
- Déplacer les personnages
- Attaquer un ennemi
- Lancer un sort (pas encore implémenté)

#### 4.1.3 Règles automatiques

Les règles automatiques sont les checks qui sont lancés à chaque fois que l'on veut faire une action, afin de savoir si l'action est valide, elles sont exécutées en fonction de la commande extérieure qui a été exécutée auparavant. Un autre type de règle automatique est la barre d'action, qui détermine l'ordre des tours, cette barre d'actions est une commande qui va s'exécuter à intervalles réguliers.

Les différentes règles suivant les actions sont, pour :

- Incrémenter la barre d'action :
  - Tester si un personnage est arrivé au maximum de la barre
  - Si un personnage est arrivé au maximum : passer en mode tour par tour, lui permettre de jouer et lui donner son mana.
- Sélectionner un personnage :
  - Chercher les informations sur le personnage sélectionné
  - Afficher les informations sur le personnage sélectionné
- Déplacer un personnage :
  - Vérifier la disponibilité des cases
  - Calculer le pathfinding
  - Effectuer le déplacement
- Attaquer un personnage
  - Tester la possibilité de l'attaque (portée, ligne de vue)
  - Calculer les positions relatives, pour les dégâts directionnels
  - Calculer les dégâts infligés (défense, crit et autre)
  - Appliquer les dégâts au personnage
  - Tester si le personnage est mort
- Lancer un sort
  - Obtenir les informations sur le type de sort
  - Obtenir les informations sur la cible

— Réaliser l'action en fonction du sort (si c'est une attaque, on attaque, un soin, on soigne...)

## 4.2 Conception logiciel

Le diagramme de classe de moteur de jeu est représenté sur la figure Le jeu repose sur un patron de conception de type Command.

La classe commande est une classe abstraite qui parente toutes les autres classe commande.

Les classes commandes, ce sont les classes qui représentent les différentes commandes :

- ShowInfoCommand, prend une case en argument et renvoie les informations sur le personnage qui s'y situe
- AttackCommand, prend une case en argument et lance une attaque sur cette case, en vérifiant les autres règles
- MoveCommand, prend une case en argument et déplace, si possible, le personnage sur la case
- SpellCommand, prend une case en argument, et lance le sort sur cette case
- TickCommand, incrémente les ticks de tour, afin de faire avancer la barre des tours

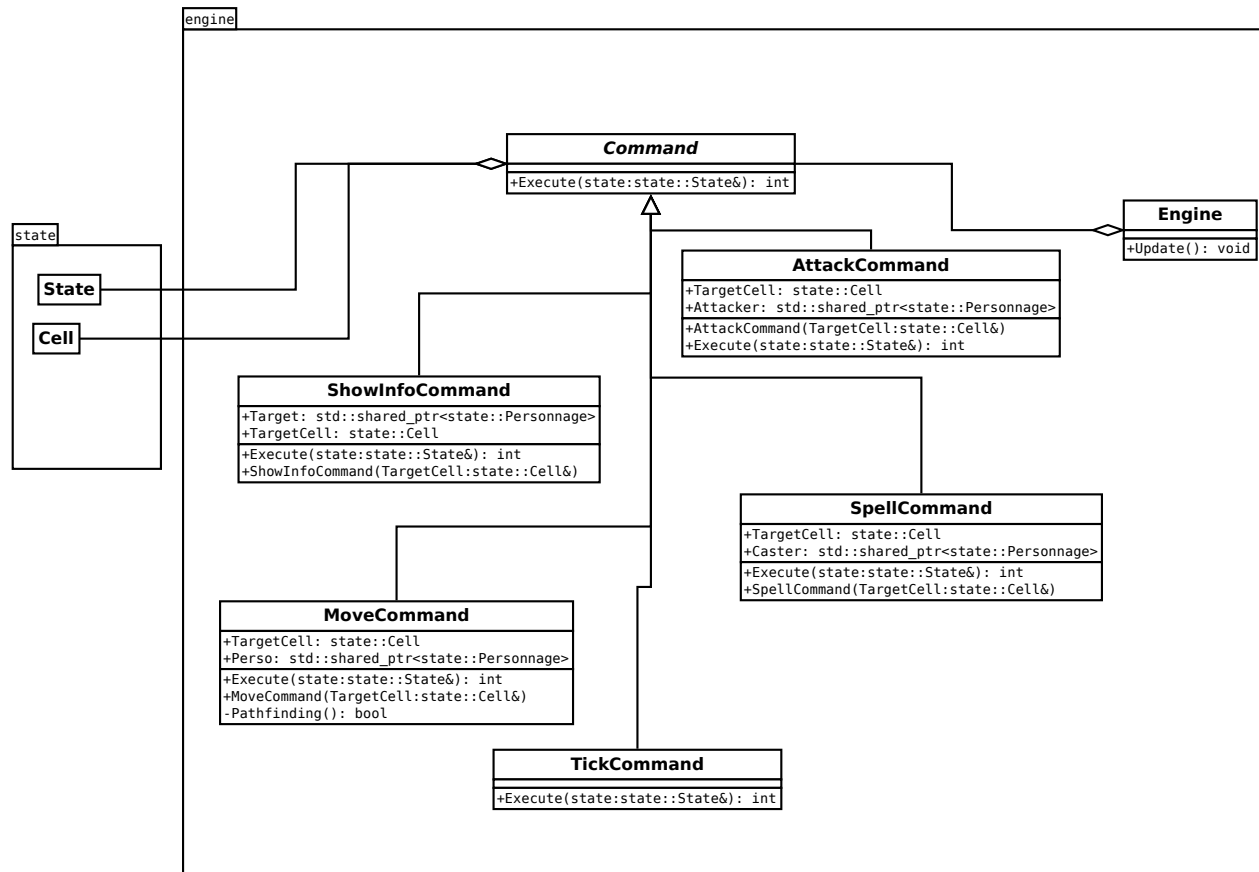


FIGURE 3 – Diagramme des classes de moteur de jeu.

## 5 Intelligence Artificielle

### 5.1 Stratégies

#### 5.1.1 Intelligence aléatoire

L'intelligence artificielle que nous implémentons consiste à sélectionner un personnage aléatoirement dans la liste des personnages encore vivant. Après avoir sélectionner le personnage, l'intelligence va choisir aléatoirement une action entre "déplacer le personnage" et "attaquer un personnage".

#### 5.1.2 Intelligence heuristiques

Pour avoir un meilleur comportement que le hasard, nous décidons d'implémenter un ensemble d'heuristiques afin que l'intelligence artificielle puisse répondre au problème suivant :

**Tuer tous les personnages de l'adversaire.**

Pour commencer nous avons choisis les heuristiques suivantes :

- Si un personnage est à portée, on l'attaque !
- Sinon, on cible un personnage qui possède moins de 30% de ses PVMax pour s'en approcher.
- Sinon, on cible un personnage qui possède une défense inférieure à notre attaque.

#### 5.1.3 Intelligence avancé

Pour avoir une intelligence qui réfléchit par-elle même, nous avons opté pour une intelligence utilisant un arbre de recherche.

Notre IA avancé va commencer par faire une liste de tous les personnages faisant partie de l'équipe ennemie, puis, dans une copie du "state", elle va essayer de se rapprocher de chacun de ces ennemis et de les attaquer, on va ensuite laisser l'IA heuristique jouée quelques tours, et on note le résultat, si un allié est mort, on a une mauvaise note, si un ennemi est mort, alors on a une bonne note, on prend alors le coup qui a permis d'avoir la meilleure note, et on le joue.

## 5.2 Conception logiciel

### 5.2.1 IA

**RandomIA** Pour générer de l'aléatoire nous utilisons la fonction **rand()** de la librairie `<cstdlib>`. Elle renvoie une série de chiffres générés par des algorithmes donc on aura pas un nouveau chiffre après chaque compilation.

**HeuristiqueIA** Nous avons introduit un système de **node**, les nodes sont créés en parcourant la map et chaque cellule sera associée à une node, elles sont distinguées selon si un personnage est présent dans la case.

De plus à chaque node est associée un alentours qui est une liste de node "libre" autour d'elle. Ceci permet aux personnages de se déplacer tout en esquivant les autres personnages.

**DeepIA** Pour l'IA avancée nous nous sommes inspirés de l'IA heuristique, en y ajoutant un système d'état temporaire de simulation, et un système de note, qui permet de classer les différents coups.

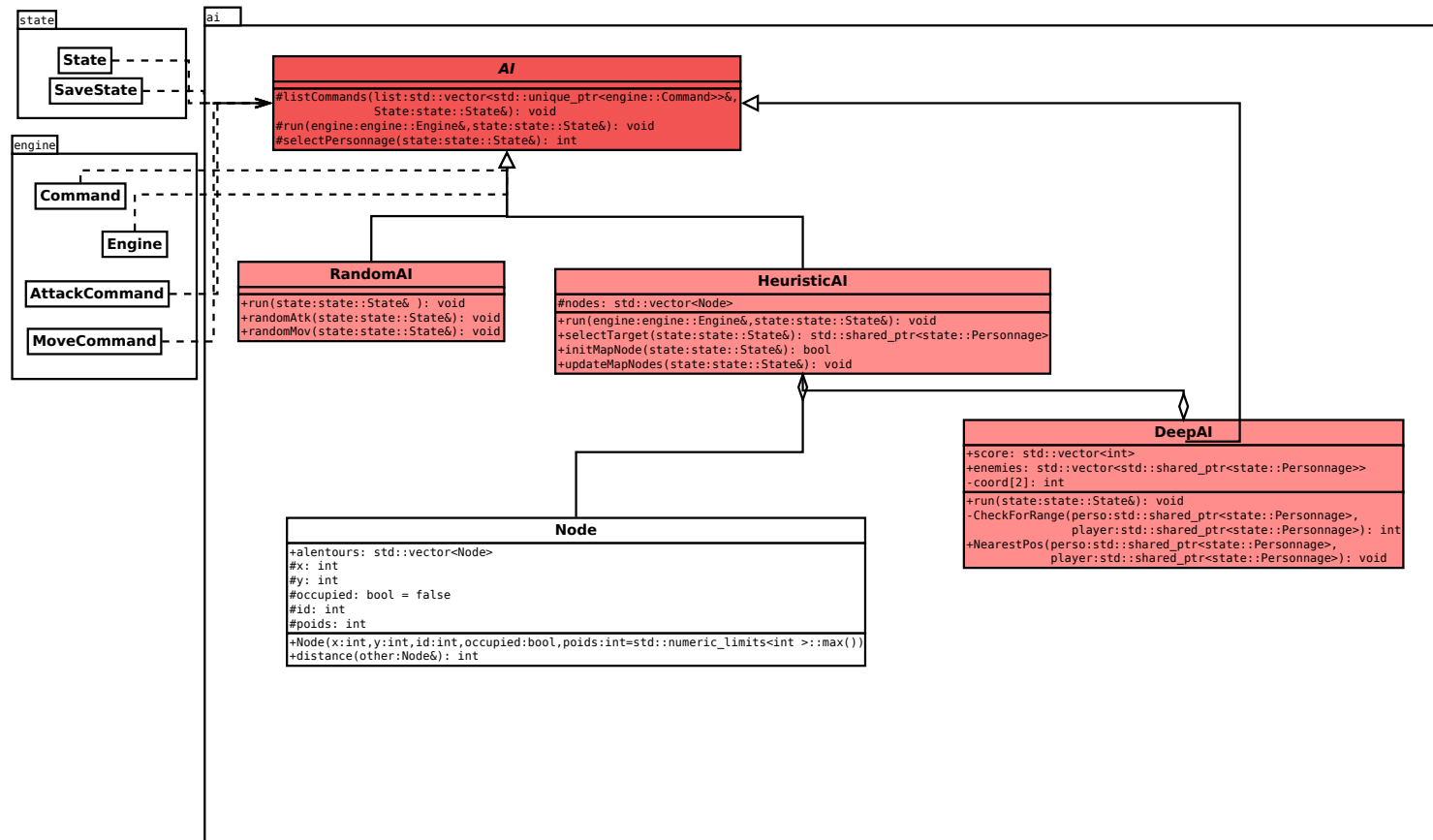


FIGURE 4 – Diagramme des classes d'intelligence artificielle.

## **6 Modularisation**

### **6.1 Organisation des modules**

Comme notre jeu a une partie "temps réel", nous avons été forcé d'implémenter un thread supplémentaire directement dans notre engine. Nous avons donc une partie de l'engine, qui est dans un thread a part, et l'affichage se charge uniquement lorsqu'il y a un changement dans le state, comme nous n'utilisons pas de d'animation d'attente ou "idle animation", on peut charger le render que lors d'un changement, pour optimiser la gestion des ressources. Le thread a part, lui, va incrémenter toutes les secondes la vitesse des personnages, lorsqu'on arrive a 100 pour un personnage, le thread tourne dans le vide, et attend que le tours du personnages soit finit, tout cela est gérer dans le thread principal.



## 6.2 Conception logiciel

Comme la modularisation est directement une partie de l'engine, alors le thread supplémentaire est directement intégré à l'engine. Dans l'engine, on a une classe "engine" qui s'occupe de d'interfacer le thread. On utilise un constructeur, pour initialiser l'engine sur un state particulier. On peut ensuite lancer l'engine avec la méthode start, qui va appeler la méthode increment dans un nouveau thread. C'est cette méthode increment, qui va s'occuper de gérer les différents tours des personnages. On peut ensuite mettre fin au tour du personnage, avec la fonction EndTurn. On a aussi la fonction IsGameOver, qui check pour savoir si la partie est terminée. Et finalement la fonction Stop, qui va attendre que le thread soit bien terminé, quand la partie est terminée.

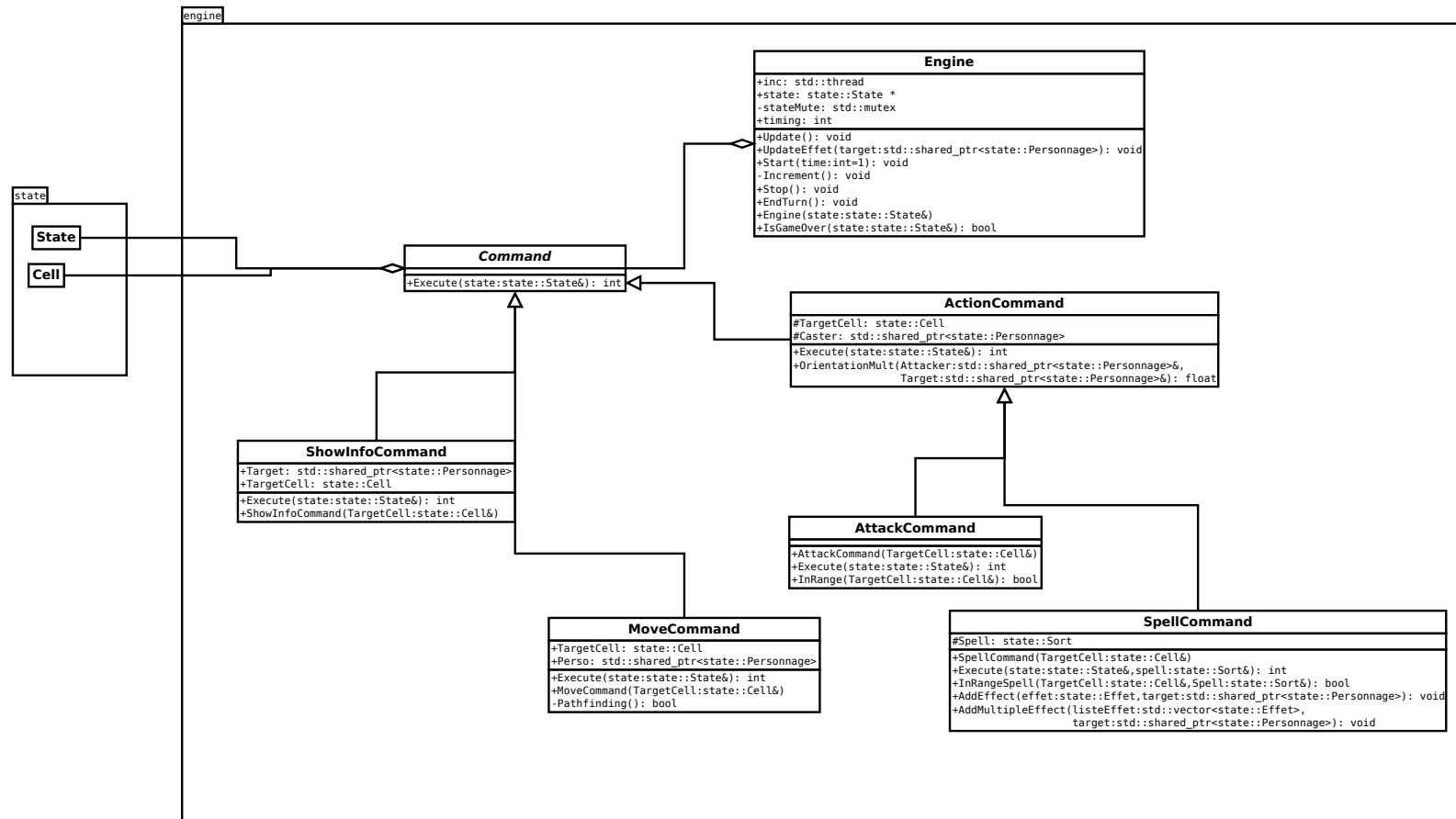


FIGURE 5. Diagramme des classes pour la modularisation