

Projet Logiciel Transversal

Juliette COME - Charly BION

Table des matières

1 Présentation Générale	1
1.1 Archétype	1
1.2 Règles du jeu	1
1.3 Ressources	2
2 Description et conception des états	4
2.1 Description des états	4
2.1.1 Etat du jeu	4
2.1.2 Etat de la carte	5
2.1.3 Etat du deck	6
2.1.4 Etat des entités	8
2.2 Conception Logiciel	10
3 Rendu : Stratégie et Conception	18
3.1 Stratégie de rendu d'un état	18
3.2 Conception logiciel	20
4 Règles de changement d'états et moteur de jeu	22
4.1 Règles	22
4.1.1 La carte	22
4.1.2 L'intérieur des salles	22
4.2 Conception logiciel	24
5 Intelligence Artificielle	27
5.1 Stratégies	27
5.1.1 Intelligence aléatoire	27
5.1.2 Intelligence basée sur des heuristiques	27
5.1.3 Intelligence artificielle basée sur des arbres de recherche	28
5.2 Conception logiciel	29

6 Modularisation	31
6.1 Record/Replay	31
6.2 Répartition sur différents threads	31
6.3 Répartition sur différentes machines : Lobby	31
6.4 Conception logiciel	33

1 Présentation Générale

1.1 Archétype

Notre projet sera basé sur le jeu "slay the spire". C'est un jeu de deck building, rogue-like, pour le moment encore en early accesss.

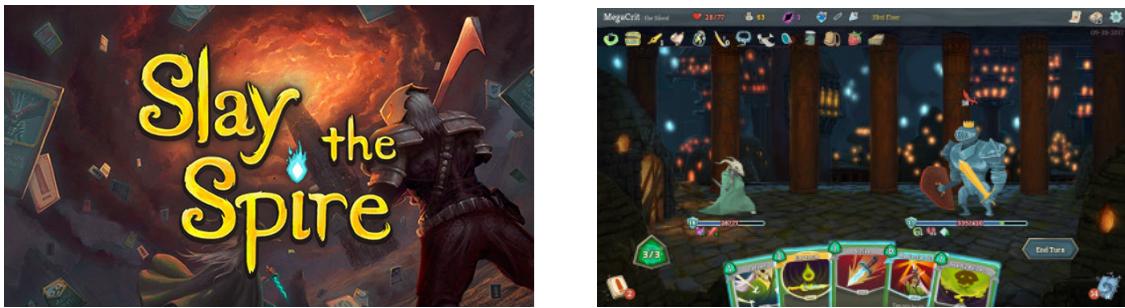


FIGURE 1 – Jeu slay the spire

1.2 Règles du jeu

Dans ce jeu, le joueur commence avec un deck basique de départ, puis progresse de salle en salle jusqu'à arriver au boss final. Il peut compléter son deck de cartes en fonctions des salles qu'il croise.

- Les salles sont divisées en deux groupes majeurs : les salles d'ennemis, plus fréquentes, et les salles spéciales. Entrer dans une salle d'ennemis déclenche un combat entre le joueur et les ennemis présents dans la salle. Les salles spéciales peuvent avoir des effets positifs ou négatifs sur le joueur. Les effets peuvent être par exemple acquérir une carte puissante, ou obtenir un bonus ou un malus pour le combat suivant.
- Le joueur et les ennemis possèdent un nombre de point de vie et une force précise. La puissance des ennemis augmentera avec le nombre de salles parcourues. Vaincre un ennemi consiste à réduire ses points de vie à 0. De même, le joueur perd la partie si sa vie est réduite à 0.
- Le deck sera limité à 15 cartes, chaque carte ajoutée doit en remplacer une autre si cette limite est atteinte.
- A chaque tour, le joueur possède un taux précis d'énergie et pioche une nouvelle main de 5 cartes. Le joueur peut jouer ses cartes en dépensant de l'énergie, et finir son tour à tout moment. Toutes les cartes non jouées sont défaussées et sont mises dans la pile de défausse. Au tour suivant, toute l'énergie du joueur est regénérée. Il pioche une nouvelle main de 5 cartes issues de sa pioche. Lorsque sa pioche est vide, la défausse du joueur est mélangée et forme la nouvelle pile de pioche. Pendant le tour du joueur, il peut voir les actions qui seront réalisées par les ennemis. Les ennemis jouent toujours leurs actions après le joueur.
- Les cartes peuvent être piochées, jouées ou défaussées. Elles dépensent de l'énergie lorsqu'elles sont jouées. Elles peuvent être utilisées pour attaquer l'ennemi, ajouter du block au joueur, ou réaliser d'autres actions plus spécifiques.

- Les cartes et les ennemis possèdent un élément précis (air, eau, terre, feu). Certains éléments seront plus puissants contre d'autres et inversement. Les cartes pourront également apporter des bonus et malus au joueur et aux ennemis.
- Les ennemis relachent des cartes après qu'ils soient vaincus, que le joueur peut ajouter à son deck. Ces cartes augmentent en puissance en fonction du nombre de salles parcourues.
- Le jeu est découpé en étages et en salles. Chaque étage se termine par un combat de boss, un ennemi plus puissant que les autres, et représentant un élément spécifique. Chaque boss achevé permet la progression du joueur vers l'étage suivant, sauf pour le dernier boss du jeu. Quatre étages, représentant chacun des éléments définis, sont prévus.

1.3 Ressources

Ressources nécessaires :

- Images des joueurs



FIGURE 2 – Exemples de deux sprites de joueur

- Image des ennemis



FIGURE 3 – Exemples de sprites d'ennemis

— Fonds pour les salles

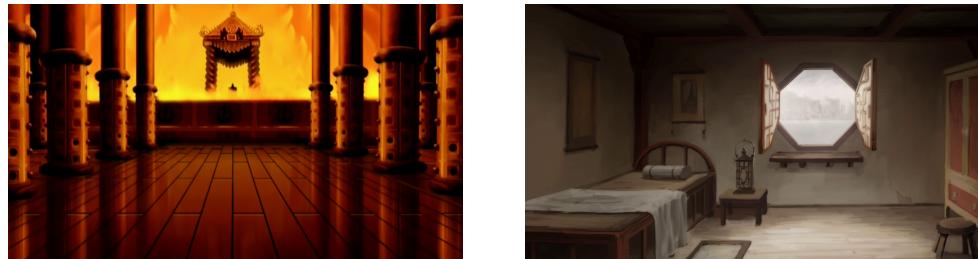


FIGURE 4 – Exemple de deux salles (une salle de combat et une salle de repos)

— Icônes : cartes, bonus/malus, types d'attaque

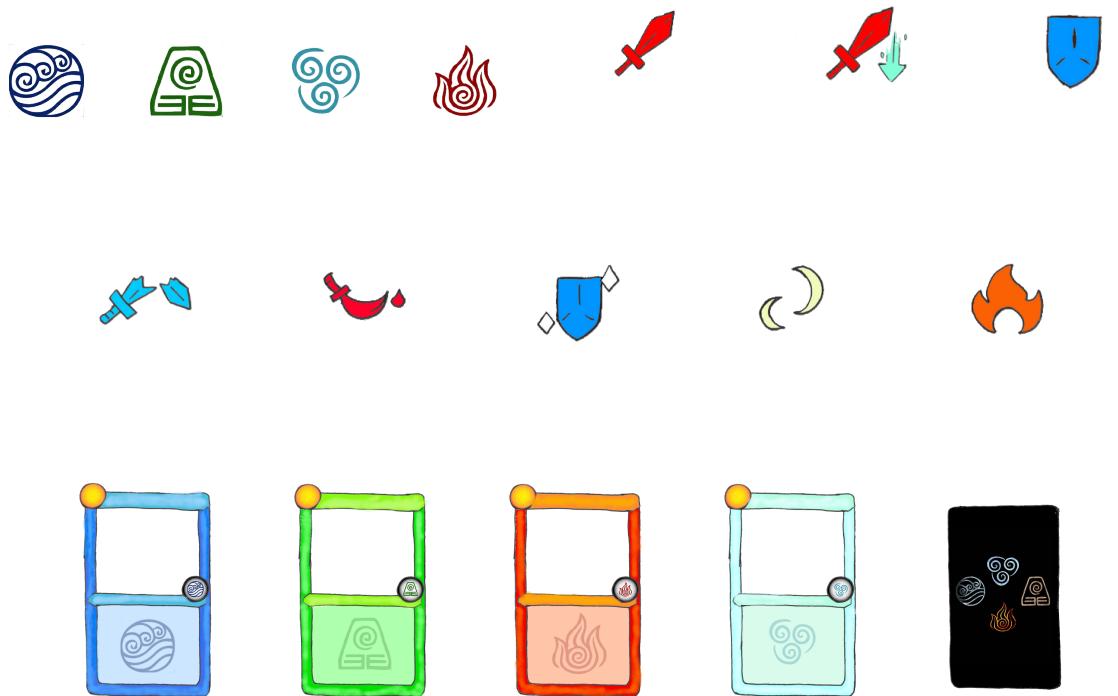


FIGURE 5 – Exemple de sprites pour les cartes et les différentes icônes utilisées

— Fond de la carte et icônes

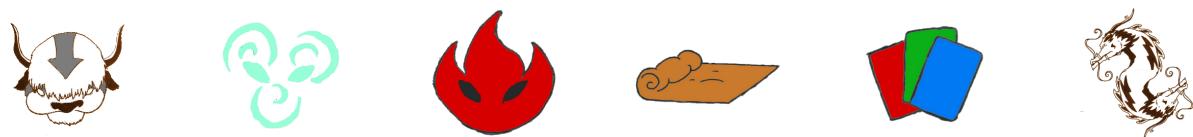


FIGURE 6 – Exemple d'icônes pour la carte

2 Description et conception des états

2.1 Description des états

Le diagramme des états possède pour l'instant 18 classes. Tous les attributs présentés seront private sauf si précisé autrement.

2.1.1 Etat du jeu

InfoPlayer

Cette classe contient les informations d'initialisations d'un joueur. Elle est utilisée pour initialiser le joueur, son deck, et également la map créée.

- int firstElement : élément de départ du joueur. Les valeurs autorisées sont 1 (Air), 2 (Water), 3 (Earth), 4 (Fire). La valeur par défaut est 1.
- bool playerControlledByAI : indique si le joueur est contrôlé par une intelligence artificielle ou non.

Rules

Cette classe contient les informations du/des joueur(s) et si le jeu est fini ou non.

- bool isGameLost : Indique si les joueurs ont gagné ou non. Implique que le jeu est terminé. Le jeu est perdu lorsque tous les joueurs sont morts.
- bool isGameOver : Indique si le jeu est terminé ou non. Un jeu est terminé lorsqu'il est perdu, ou lorsque les joueurs ont traversé la dernière salle du dernier étage.
- int nbPlayers : indique le nombre de joueurs qui participeront au jeu. Les valeurs autorisées sont 1 ou 2. Ces joueurs joueront en coopératif et affronteront les ennemis ensemble.
- std::vector<std::unique_ptr<InfoPlayer>> infoPlayer : vecteur de pointeurs vers un objet InfoPlayer. Contient les informations du joueur 1 et 2 (s'il existe) respectivement. sa taille maximale est de 2, sa taille minimale est de 1.

GameState

Contient des pointeurs vers tous les objets du jeu.

- std::vector<Player*> players : Un vecteur de pointeurs vers les joueurs. Sa taille minimale est de 1, sa taille maximale est de 2.
- std::unique_ptr<Map> map : Un pointeur vers la carte du jeu. La carte est composée d'étages (Floor), eux-mêmes composés de salles (Room).
- std::unique_ptr<Rules> rules : un pointeur vers un objet Rules qui définit les règles du jeu.
- bool isInsideRoom : Indique si les joueurs se trouvent dans une salle ou à l'extérieur. S'ils se trouvent à l'extérieur, c'est qu'ils sont en train de choisir une salle. Ils ont alors vue sur l'ensemble du graph des salles. Sinon, ils se trouvent à l'intérieur et effectuent les actions correspondantes. Les deux joueurs sont toujours en même temps à l'intérieur ou à l'extérieur d'une salle.

2.1.2 Etat de la carte

Map

Contient des pointeurs vers les différentes étages.

- std : `:vector<std : share_ptr<Floor>>` floors : vecteur de pointeurs vers les différents étages (Floor). Il y a forcément 4 étages (un pour chaque élément), donc ce vecteur est de taille 4.
- int currentFloor : la référence vers l'étage actuel. Forcément comprise entre 0 et 3 inclus.

Floor

Contient des pointeurs vers les différentes salles. Les salles sont organisées à l'aide d'un graphe orienté.

- int floorNumber : le numéro de l'étage. Compris entre 0 et 3 inclus. En plus de référencer l'étage, ce nombre est utilisé pour instancier les salles et les monstres afin d'avoir une difficulté graduée.
- std : `:shared_ptr<Room>` currentRoom : un pointeur vers la salle à laquelle se situent les joueurs.
- int element : indique l'élément de l'étage. Les valeurs autorisées sont 1 (Air), 2 (Water), 3 (Earth), 4 (Fire). Sert à instancier la majorité des ennemis de cet étage avec cet élément.
- std : `:string` floorImage : chaîne de caractère qui indique le chemin vers l'image associée à l'étage. Ce chemin doit référencer une image du dossier du projet. Il y a une image différente pour chaque étage.
- std : `:shared_ptr<Room>` firstRoom : un pointeur vers la première salle du graphe orienté. Cette salle est unique et sera considérée comme l'origine du graphe. Aucune salle ne pointe vers elle.

Room

Il existe 3 types de salle : les salles d'ennemis (EnemyRoom), les salles de repos (SleepRoom) et les salles spéciales (SpecialTrainingRoom).

- int element : l'élément de la salle (Room). Les valeurs autorisées sont 1, 2, 3 ou 4 pour chacun des éléments. Une salle peut être d'un élément différent que celui de l'étage dont elle fait partie.
- std : `:string` imageMapRoom : chaîne de caractères qui indique le chemin vers l'image associée à la salle. Ce chemin doit référencer une image du dossier du projet. Cette image sera utilisée pour l'affichage de la salle sur la carte (lorsque les joueurs seront en dehors de la salle).
- std : `:string` imageInsideRoom : chaîne de caractères qui indique le chemin vers l'image associée à la salle. Ce chemin doit référencer une image du dossier du projet. Cette image sera utilisée lors de l'affichage de l'intérieur de la salle (lorsque les joueurs seront dans la salle).
- bool isSpecialTrainingRoom : un booléen indiquant si la salle est une salle spéciale (true) ou non (false). Doit être false si au moins un parmi isEnemyRoom ou isSleepRoom est true.
- bool isEnemyRoom : un booléen indiquant si la salle est une salle d'ennemis (true) ou non (false). Doit être false si au moins un parmi isSpecialTrainingRoom ou isSleepRoom est true.
- bool isSleepRoom : un booléen indiquant si la salle est une salle de repos (true) ou non (false). Doit être false si au moins un parmi isEnemyRoom ou isSpecialTrainingRoom est true.
- std : `:shared_ptr<Room>` nextRoom : pointeur vers la salle suivante dans le graphe. Il n'y a pour l'instant qu'un unique chemin pour le graphe orienté.

SpecialTrainingRoom

Hérite de la classe Room. Permet aux joueurs d'ajouter des cartes à son deck.

- std : :vector<Card*> cardReward : un vecteur de pointeurs vers des cartes. La taille de ce vecteurs est 3. Les cartes peuvent être générées avec n'importe quel élément.

SleepRoom

Hérite de la classe Room. Permet aux joueurs de se reposer et de regagner de la vie.

- int heal : indique la valeur de vie que récupèrera un joueur en se reposant. Valeur positive requise.

EnemyRoom

Hérite de la classe Room. Les joueurs doivent affronter un ou plusieurs ennemis et les vaincre pour pouvoir passer à la salle suivante.

- std : :vector<std : :unique_ptr<Enemy>> enemies : un vecteur de pointeurs vers des objets de classe Enemy. Ce vecteur possède une taille minime de 1 et maximale de 3.
- std : :vector<std : :unique_ptr<DeckParts>> drawPiles : un vecteur de pointeurs vers des objets de type DeckParts. Le vecteur possède une taille minimale de 1 et maximale de 2. Référence la pioche (drawPile) de chacun des joueurs.
- std : :vector<std : :unique_ptr<DeckParts>> hands : un vecteur de pointeurs vers des objets de type DeckParts. Le vecteur possède une taille minimale de 1 et maximale de 2. Référence la main (hand) de chacun des joueurs. Une seule main sera affichée à la fois.
- std : :vector<std : :unique_ptr<DeckParts>> discardPiles : un vecteur de pointeurs vers des objets de type DeckParts. Le vecteur possède une taille minimale de 1 et maximale de 2. Référence la défausse (discardPile) de chacun des joueurs.
- int turn : nombre de tours écoulés depuis le début du combat. Valeur positive requise.
- int entityTurn : indique l'identifiant (Id) de l'entité en train de jouer. Si cet identifiant correspond à un joueur, sa main sera affichée et il pourra jouer des cartes. Si cet Id correspond à un ennemi, il effectuera son action puis choisira sa suivante en fonction des actions précédentes des joueurs.
- bool isGameLost : indique si les joueurs ont perdu. Le combat continue tant que les joueurs n'ont pas gagné, perdu ou abandonné. Les joueurs gagnent ce combat s'ils réduisent à 0 la vie des monstres adverses, perdent si leur vie est réduite à 0 ou s'ils abandonnent. Si les joueurs gagnent, ils sortent de la salle.

2.1.3 Etat du deck

Card

Définie les cartes jouées dans le jeu.

- std : :string name : le nom de la carte. Doit être un nom de carte existant.
- int cost : le coût en énergie de la carte. Les valeurs possibles sont comprises entre 0 et 3 inclus.
- int target : indique à quel type d'entité s'applique la carte. Les valeurs possibles sont 0 (unique joueur), 1 (unique ennemi), 2 (tous les ennemis), 3 (tous les joueurs).
- std : :string image : chemin vers l'image de la carte. Cette image doit appartenir au dossier.
- int element : l'élément de la carte. Les valeurs possibles sont 0 (None), 1 (Air), 2 (Water), 3 (Earth), 4 (Fire).
- int attack : valeur d'attaque de la carte. Indique le nombre de points de vie perdus par la cible lorsque la carte est jouée (hors bonus ou malus du joueur). Valeur positive requise.
- int block : valeur de block de la carte. Indique le nombre de block gagné par la cible lorsque la carte est jouée (hors bonus ou malus du joueur). Valeur positive requise.

- int draw : indique le nombre de cartes piochées lorsque cette carte est jouée. Les cartes piochées passent de la pile de pioche à la main. Si la main est pleine, on ne pioche pas davantage. Valeur positive requise.
- int discard : indique le nombre de cartes défaussées lorsque cette carte est jouée. Les cartes défaussées vont de la main à la défausse. Valeur positive requise.
- int heal : valeur de soin de la carte. Soigne la cible du montant indiqué lorsque la carte est jouée. Valeur positive requise. Principe différent de l'attribut "heal" de la classe Buff.
- Debuff debuff : objet Debuff contenant des malus à appliquer aux cartes au moment où elles sont jouées.
- Buff buff : objet Buff contenant des bonus à appliquer aux cartes au moment où elles sont jouées.

CardManager

Singleton permettant de créer les cartes de façon unique, puis d'y avoir accès facilement et enfin de les détruire facilement.

- static CardManager* inst : Unique instance du CardManager
- std::vector<std::unique_ptr<Card>> cards : Vecteur de cartes

Deck

Contient des pointeurs vers des cartes.

- std::vector<Card*> cards : contient un vecteur de pointeurs vers des cartes. Certains pointeurs peuvent pointer vers la même carte. Ce vecteur possède une taille positive (par défaut 15) égale à sizeMax.
- int size : la taille du sous-vecteur à considérer. Si un deck est incomplet ou que des cartes ont été retirées, on ignore simplement les pointeurs correspondant et on considère un vecteur de taille plus petite. Valeur positive nécessaire.
- int sizeMax : la taille maximale du deck autorisée. Valeur positive nécessaire. Par défaut 15.

DeckParts

Permet de créer la pioche, la main et la défausse pour une salle d'ennemis.

- std::vector<Card*> cards : un vecteur de pointeurs vers des cartes. Certains pointeurs peuvent pointer vers la même carte. Possède une taille positive, égale à sizeMax.
- Player* player : pointeur vers le joueur auquel la pioche/main/défausse appartient.
- bool isHand : indique si la partie de deck est une main ou non. Une partie de deck ne peut être qu'une seule de ces instances : pioche, main, défausse.
- bool isDiscardPile : indique si la partie de deck est une défausse ou non. Une partie de deck ne peut être qu'une seule de ces instances : pioche, main, défausse.
- bool isDrawPile : indique si la partie de deck est une pioche ou non. Une partie de deck ne peut être qu'une seule de ces instances : pioche, main, défausse.
- int size : la taille du sous-vecteur à considérer. Valeur positive requise.
- int sizeMax : taille maximale du vecteur de cartes (cards) possible. Valeur positive requise.

Buff

Contient les buffs à appliquer à une entité lorsqu'elle attaque ou à chacun de ses tours.

- int blockPlus : Valeur positive requise. Augmente le nombre de block réalisé de toute carte jouée de 50% pendant le nombre de tour indiqué par blockPlus. blockPlus diminue à chaque début du tour du joueur.
- int attackPlus : Valeur positive requise. Augmente l'attaque réalisée de toute carte jouée de

50% pendant le nombre de tour indiqué par attackPlus. attackPlus diminue à chaque début du tour du joueur.

- int heal : Valeur positive requise. Soigne le joueur à la fin de son tour de la valeur indiquée par heal. Cette valeur décrémente à chaque début de tour. (Pour heal = 4, le joueur se soigne de 4 au tour n, puis de 3 au tour n+1, etc).
- int evade : Valeur positive requise. Esquive un nombre d'attaque indiqué par evade. Les attaques esquivées ne sont pas choisies, ce sont les prochaines attaques lancées par le monstre. Elles sont esquivées même si la valeur du block était suffisante pour parer entièrement l'attaque.
- int retaliate : Valeur positive requise. Inflige 5 points de dégats par attaque reçue à l'attaquant. S'applique pendant les actions de l'ennemi, pendant le nombre de tour indiqué par retaliate. Cette valeur diminue au début du tour du joueur.

Debuff

Contient les debuffs à appliquer à une entité lorsqu'elle attaque ou à chacun de ses tours.

- int blockMinus : Valeur positive requise. Diminue la valeur de block réalisée de toute carte jouée de 50% pendant le nombre de tour indiqué par blockMinus. blockMinus diminue à chaque début du tour du joueur.
- int attackMinus : Valeur positive requise. Diminue la valeur d'attaque réalisée de toute carte jouée de 50% pendant le nombre de tour indiqué par attackMinus. attackMinus diminue à chaque début du tour du joueur.

2.1.4 Etat des entités

Entity

Contient les informations principales d'une entité. Ses identifiants, sa vie... Se décompose en joueur (Player) ou ennemi (Enemy).

- std::string name : nom de l'entité. S'il s'agit d'un ennemi, ce nom doit référencer un nom d'ennemi existant.
- int id : identifiant de l'entité. Valeur positive requise. Deux entités ne peuvent pas posséder le même identifiant, même si par exemple se sont deux ennemis possédant les mêmes caractéristiques. Les identifiants des joueurs seront 0 ou 1, ceux des monstres auront des valeurs supérieures strictement à 1.
- int life : la vie d'une entité. Valeur positive nécessaire. Une entité perd de la vie si elle reçoit des attaques alors qu'elle n'a pas suffisamment de block. Si la vie d'une entité passe à 0 elle est considérée comme morte.
- int element : élément de l'entité. Valeurs possibles : 0 (None), 1 (Air), 2 (Water), 3 (Earth), 4 (Fire). Ces valeurs changeront en fonction des cartes jouées (pour les joueurs seulement).
- std::string image : chemin vers l'image utilisée lors de l'affichage de l'entité. Ce chemin doit référencer une image existante du dossier.
- int statAttack : stat de base de l'attaque. Valeur positive requise. La statAttack s'additionne à l'attribut attack d'une carte (pour les joueurs) ou d'un skill (pour les ennemis) pour le calcul final des dégâts causés à la cible. La statAttack de base des ennemis est fixée en fonction du numéro de l'étage et de la salle actuels.
- int statBlock : stat de base du block. Valeur positive requise. La statBlock s'additionne à l'attribut attack d'une carte (pour les joueurs) ou d'un skill (pour les ennemis) pour le calcul final du block obtenu par la cible. La statBlock de base des ennemis est fixée en fonction du

numéro de l'étage et de la salle actuels.

- int block : la valeur de block. Valeur positive requise. Le block permet de parer des attaques. Au lieu de perdre des points de vie, une entité perd de son block. Elle perd le reste des points de vie si son block passe à 0. Le bloc est initialisé à 0 au début de chaque tour d'une entité.
- Buff buff : les buffs appliqués à l'entité.
- Debuff debuff : les debuffs appliqués à l'entité.
- bool isPlayer : indique si l'entité est un joueur (true) ou un ennemi (false).
- bool isEntityAlive : indique si l'entité est en vie. Une entité n'est pas en vie si son total de vie est nul. Une entité n'est pas affichée si elle n'est pas en vie, et ne peut plus agir pour le reste du jeu/combat. Si aucun des joueurs n'est en vie, la partie est terminée.
- int maxLife : la valeur maximale de la vie de cette entité.

Enemy

Hérite de Entity. Une classe pour les ennemis.

- std::vector<Card*> reward : un vecteur de pointeurs vers les cartes. Ce vecteur est de taille 3. Les cartes pointées peuvent être ajoutées au deck d'un joueur lorsque le combat est terminé. Un joueur ne pourra ajouter qu'une seule carte parmi toutes les cartes proposées (même s'il y a eu plusieurs ennemis).
- std::vector<EnemySkill*> skills : un vecteur de pointeurs vers les capacités (Skill) d'un ennemi. Ce vecteur possède une taille minimale de 1 et maximale de 10. Détermine les capacités pouvant être effectuées par un ennemi.
- int intent : un entier. Fait référence au pointeur de la capacité qui va être utilisée par l'ennemi lors de son tour. Valeur comprise entre 0 et la taille de skills - 1, soit au moins comprise entre 0 et 10.

EnemySkill

Capacité des ennemis. Semblable aux cartes pour les attributs.

- int attack : Valeur d'attaque de la capacité (buffs/debuffs non appliqués). Valeur positive requise.
- int block : Valeur de block de la capacité (buffs/debuffs non appliqués). Valeur positive requise.
- int heal : Valeur de soin de la capacité. Valeur positive requise.
- std::unique_ptr<Buff> buff : bonus que la capacité va appliquer sur la cible. Ils vont s'additionner à ses bonus actuels.
- std::unique_ptr debuff : malus que la capacité va appliquer sur la cible. Ils vont s'additionner à ses malus actuels.
- std::string intentImage : chemin vers l'image de référence de l'attaque. Cette image apparaîtra au dessus de l'ennemi en combat et indiquera au joueur le type de capacité utilisée. Ces images sont de deux catégories : celles représentant une capacité menée sur un joueur (ou tous les joueurs) et celles menées sur un ennemi (ou tous les ennemis). Capacités sur les joueurs : attaque, attaque et debuff. Capacités sur les ennemis : bloque, buff, soigne et toutes les combinaisons de ces paramètres. On a au total 9 images différentes pour les images d'intention.
- int cooldown : Valeur positive requise. Un entier qui indique le nombre minimum de tours requis avant de pouvoir réutiliser cette capacité.
- int turnsBeforeUse : Valeur positive requise. Un entier qui indique le nombre de tours restant avant de pouvoir réutiliser cette capacité. Après l'utilisation de la capacité, turnsBeforeUse prend la valeur cooldown, puis est décrémentée à chaque tour jusqu'à atteindre 0.
- int target : indique à quel type d'entité s'applique la capacité. Les valeurs possibles sont 0

(unique joueur), 1 (unique ennemi), 2 (tous les ennemis), 3 (tous les joueurs).

SkillManager

Singleton permettant de créer les compétences des ennemis de façon unique, puis d'y avoir accès facilement et enfin de les détruire facilement.

- static SkillManager* inst : Unique instance du SkillManager
- std::vector<std::unique_ptr<EnemySkill>> cards : Vecteur de compétences.

Player

Hérite de la classe Entity. Contient les informations relatives au joueur.

- int energy : valeur positive requise. Le nombre d'énergie restante du joueur. Cette énergie est utilisée pour jouer des cartes, une carte ne peut pas être jouée si son coût est supérieur à l'énergie du joueur. L'énergie est initialisée à 3 au début de chaque tour du joueur.
- std::unique_ptr<Deck> deck : un pointeur vers un deck de cartes.

PlayerManager

Singleton permettant de créer les personnages des joueurs de façon unique, puis d'y avoir accès facilement et enfin de les détruire facilement.

- static PlayerManager* inst : Unique instance du PlayerManager
- std::vector<std::unique_ptr<Player>> cards : Vecteur de joueurs

2.2 Conception Logiciel

Les figures qui suivent présentent notre diagramme de classe sous forme de diagramme UML.

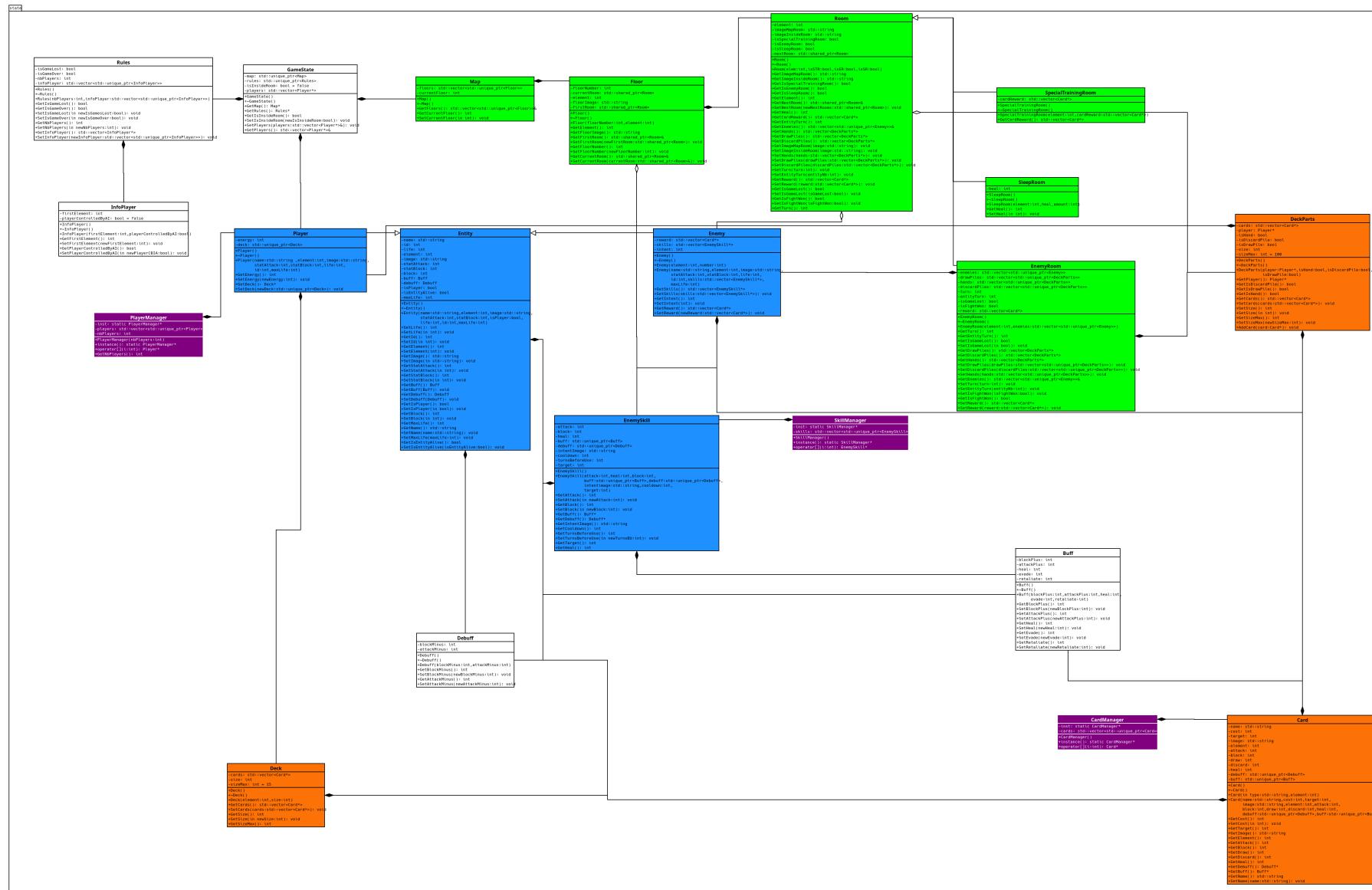


FIGURE 7 – Diagramme des classes d'état.

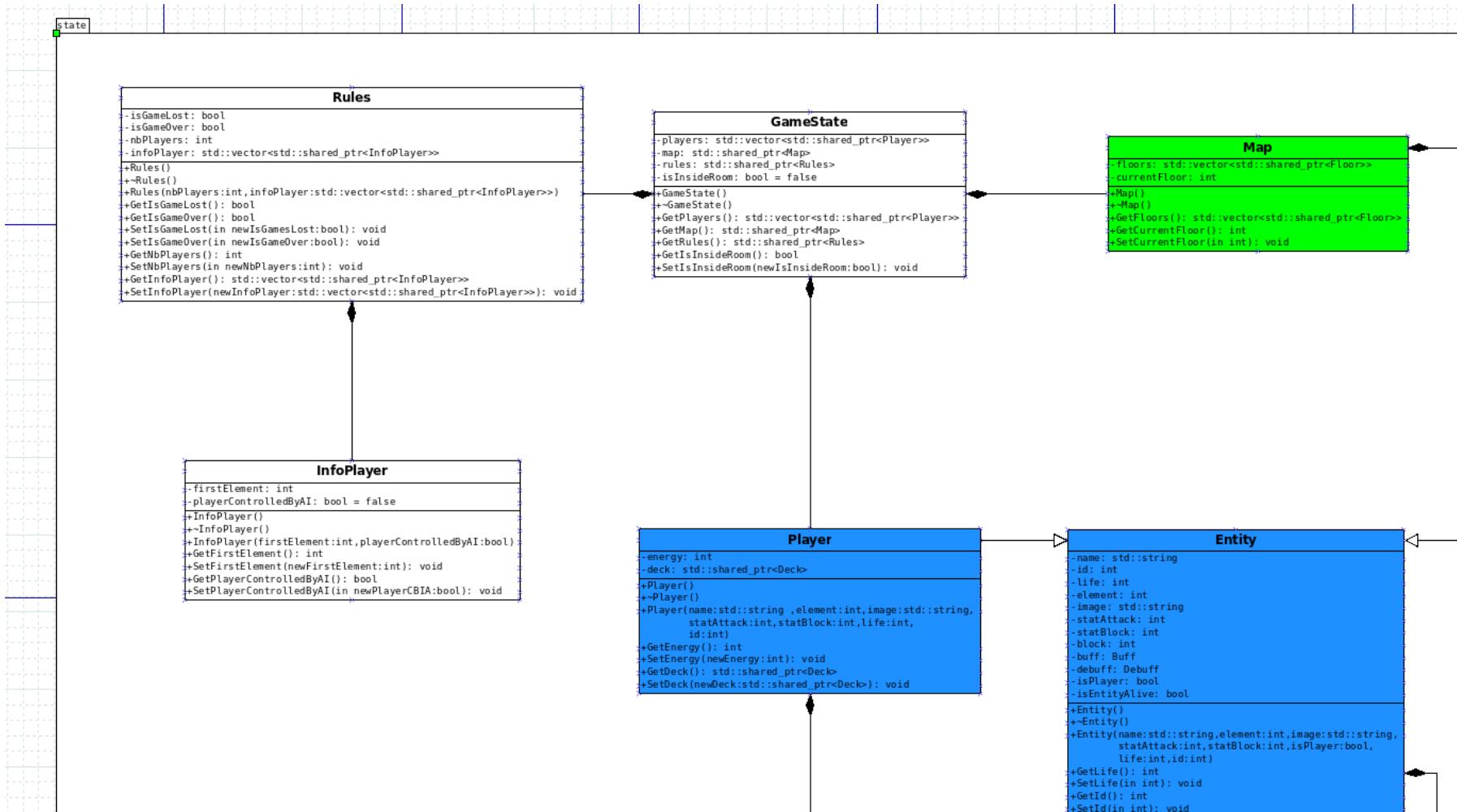


FIGURE 8 – Diagramme des classes d'état - état du jeu.

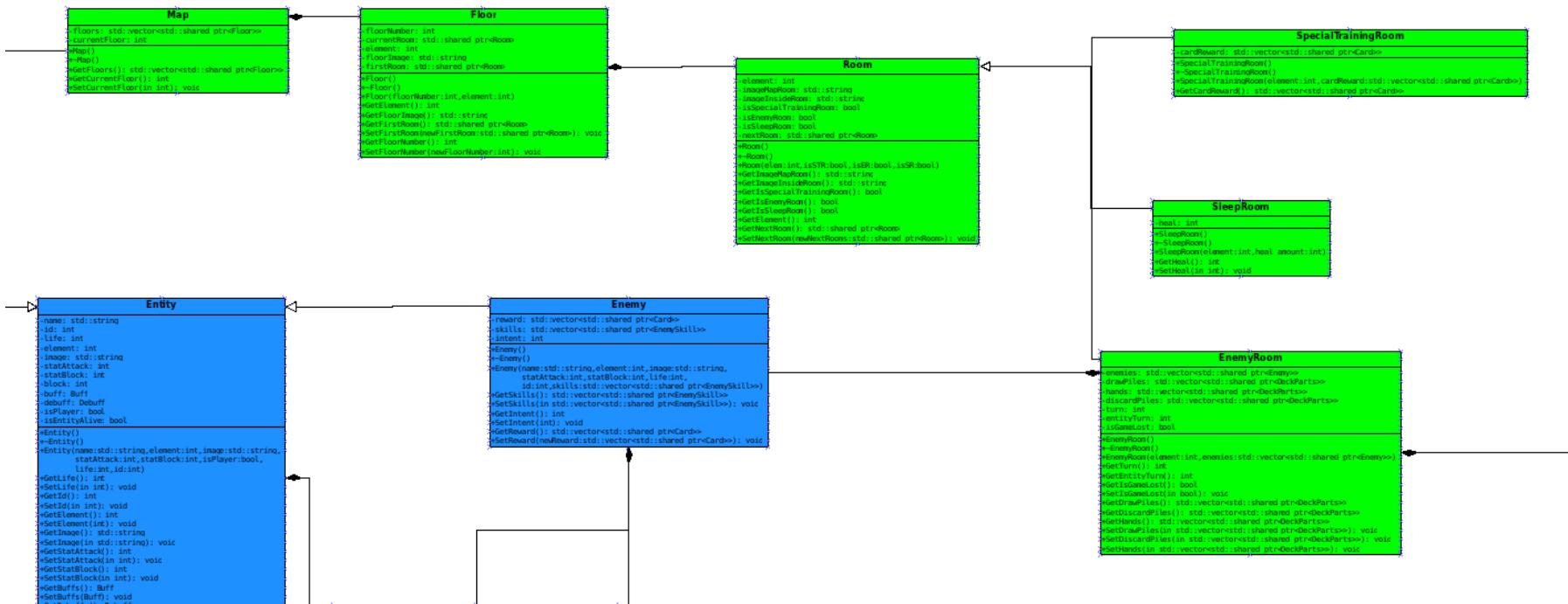


FIGURE 9 – Diagramme des classes d'état - état de la carte.

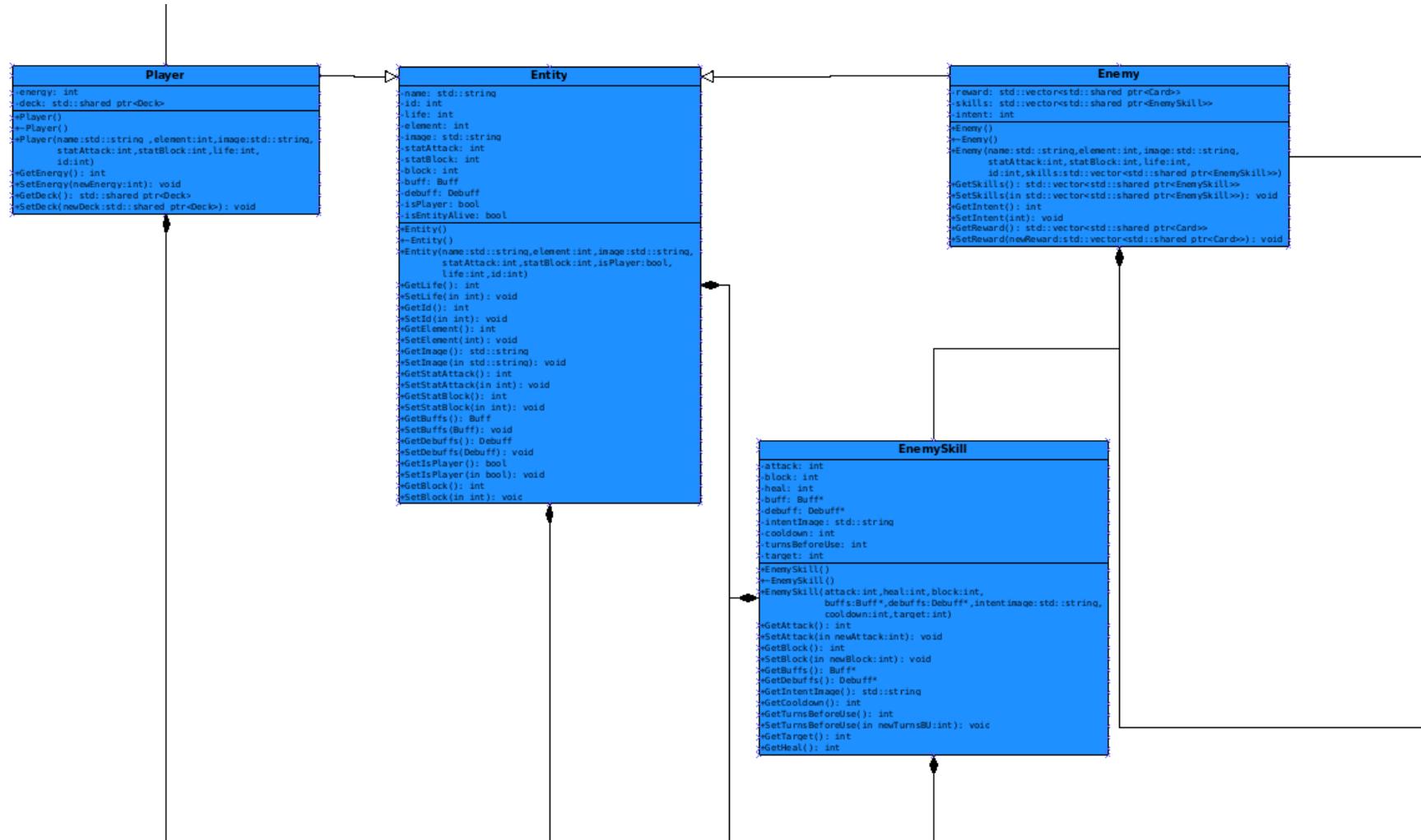


FIGURE 10 – Diagramme des classes d'état - état des entités.

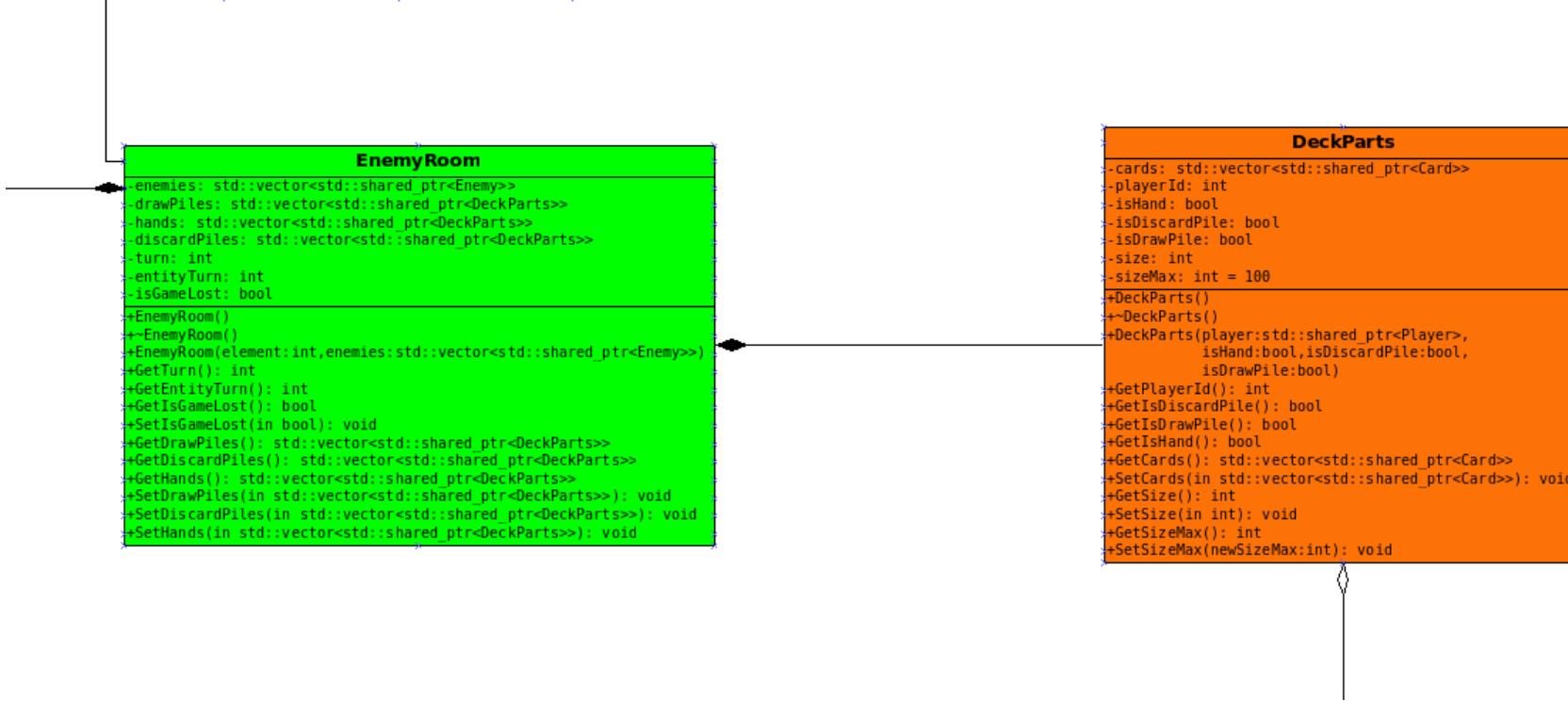


FIGURE 11 – Diagramme des classes d'état - état du deck 1.

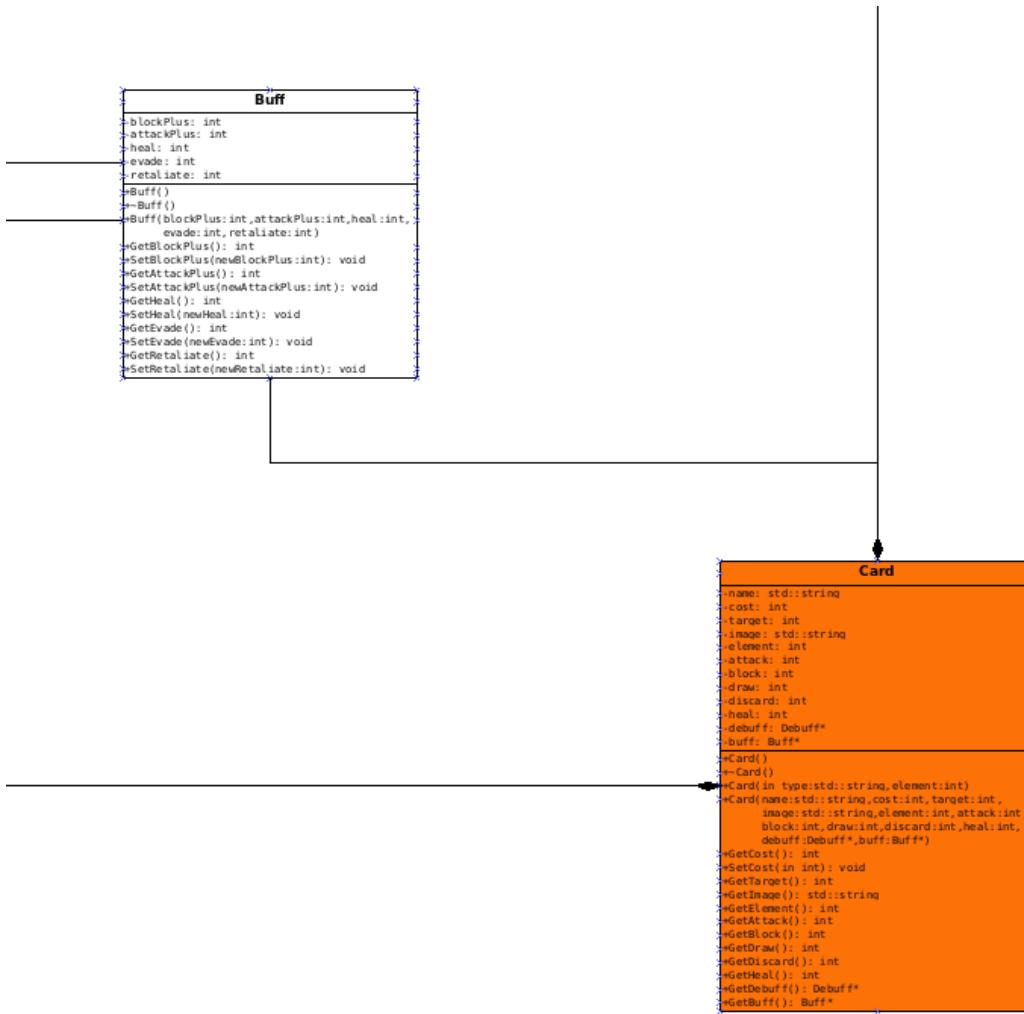


FIGURE 12 – Diagramme des classes d'état - état du deck 2.

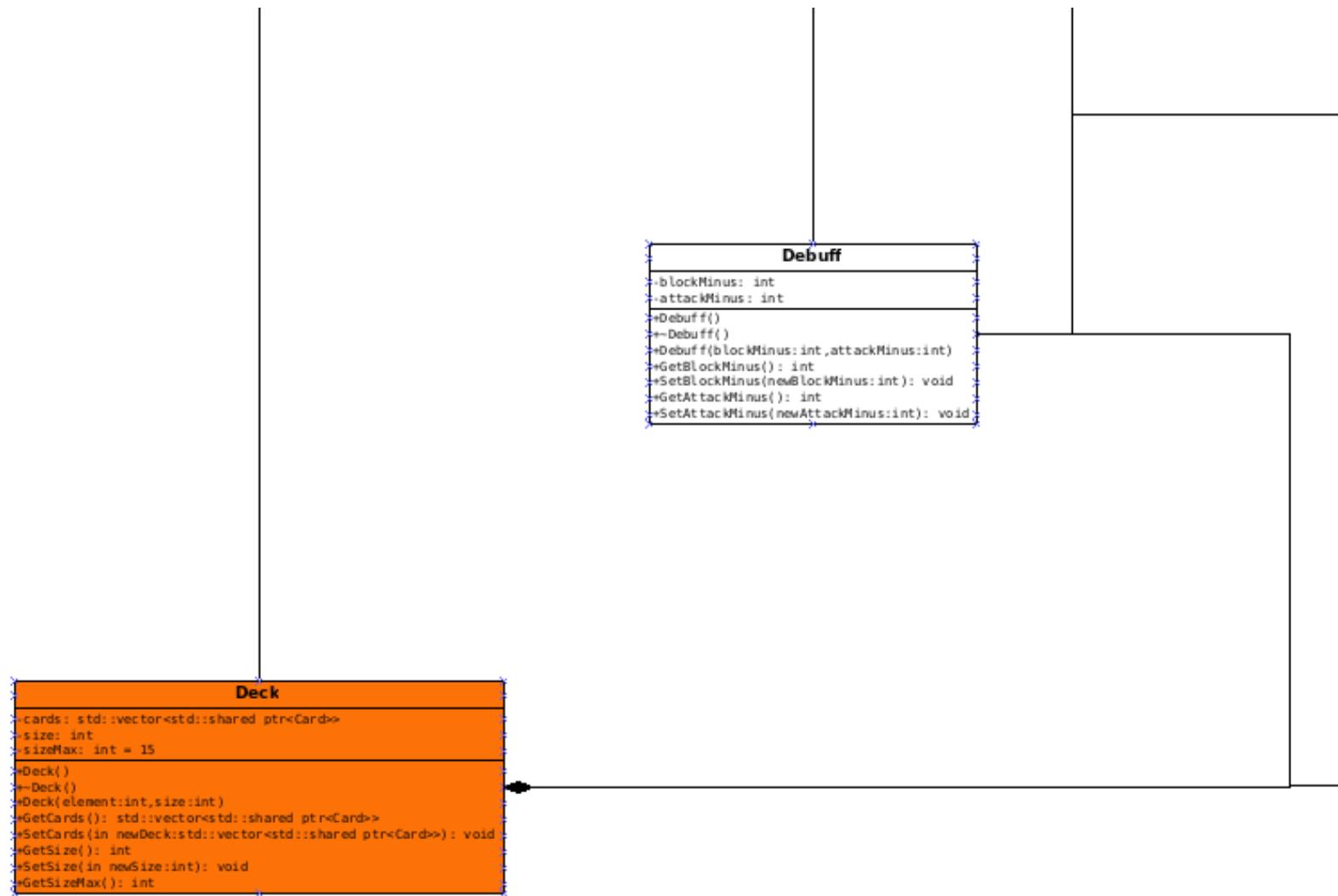


FIGURE 13 – Diagramme des classes d'état - état du deck 3.

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Nous pouvons diviser notre affichage en deux catégories principales : l'affichage de la carte, et l'affichage des salles.

Rendu des salles

Pour le rendu des salles, nous avons besoin d'afficher trois éléments principaux : des cartes (pour toutes les salles), des joueurs et des ennemis (pour la salle d'ennemis). Nous avons donc décidé de créer une classe "Editeur" afin de créer des textures de ces trois éléments, et une classe "Rendu" dont le rôle est de générer le rendu général de l'affichage.

Editeur

- int x : position de la texture sur l'axe des abscisses. Cet entier est utilisé afin de positionner la texture pour l'affichage final. Cette position est en pixels.
- int y : position de la texture sur l'axe des ordonnées. Cet entier est utilisé afin de positionner la texture pour l'affichage final. Cette position est en pixels.
- float scale : l'échelle à appliquer à la texture. Doit être précisée à la création ou à l'édition d'une texture.
- sf : :RenderTexture : texture correspondant à un joueur/ennemi ou carte. Elle est créée sur un fond transparent.

Pour la création de la texture des joueurs et des ennemis, les éléments suivants sont à prendre en compte :

- Le nom du joueur ou du monstre
- Sa vie
- Son élément
- Ses bonus ou malus, ainsi que le nombre de tours pendant lesquels ils s'appliquent
- La valeur de block
- L'énergie restante (pour le joueur)
- L'intention de l'ennemi (pour l'ennemi), c'est-à-dire une icône représentant l'action qu'il va réaliser, et si c'est une attaque, sa puissance.

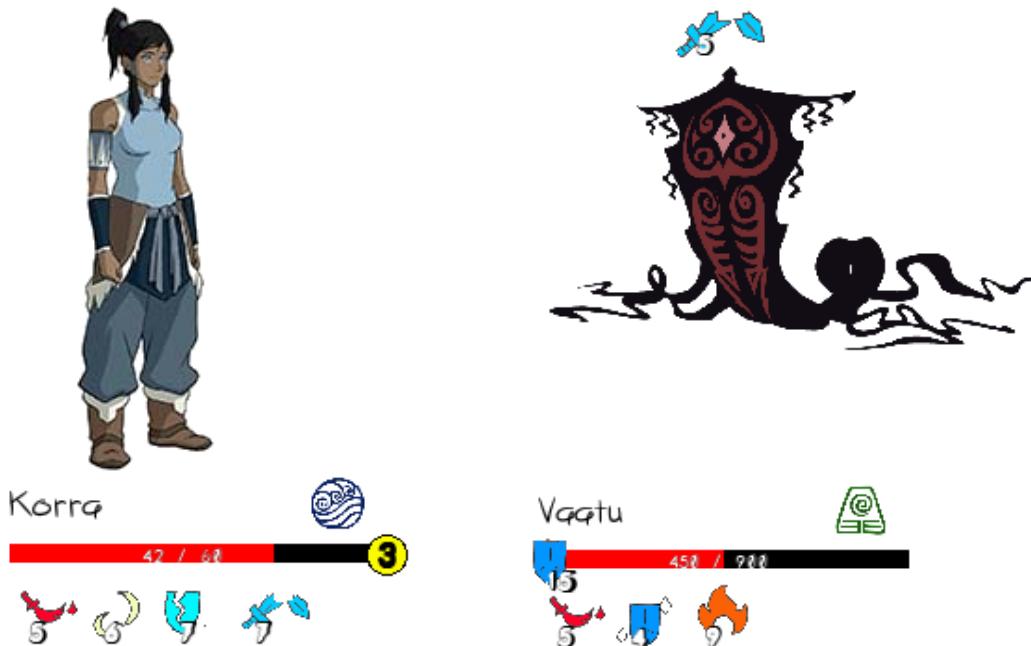


FIGURE 14 – Exemple de texture créé pour un joueur (à gauche) et un ennemi (à droite). Au dessus de la barre de vie se situent le nom du personnage ainsi que son élément. Les bonus ou malus sont indiqués en dessous de la barre de vie. La valeur de block pour ce tour est indiquée au côté gauche de la barre de vie, la valeur d'énergie (pour le joueur) est indiquée à droite de la barre de vie. L'intention de l'ennemi est indiquée au dessus de lui.

Rendu

Le rendu est utilisé afin de générer les différentes textures pour les salles et pour la carte. La classe Rendu utilise des objets de classe Editeur pour ce faire. La création d'une salle consiste en la création de tous les éditeurs nécessaires à la salle et de les placer au bon endroit. Par exemple, une salle d'ennemi aura besoin d'au moins une texture joueur, une texture ennemi, et 7 textures cartes (5 pour la main et 2 pour représenter la pioche et la défausse).

Les attributs de cette classe sont :

- int dimensionX : un entier indiquant la longueur de la fenêtre à générer en pixels
- in dimensionY : un entier indiquant la largeur de la fenêtre à générer en pixels.
- Gamestate * gamestate : un pointeur vers le gamestate.

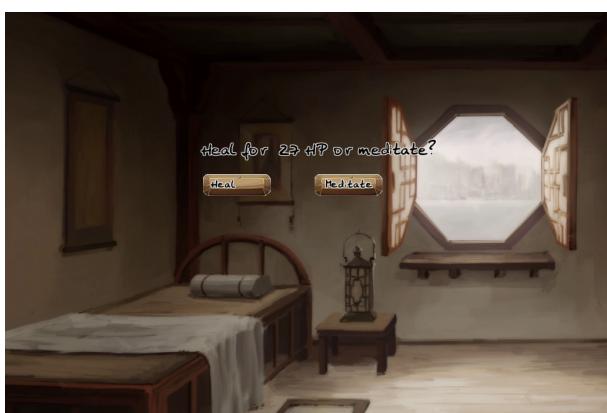


FIGURE 15 – Exemple de rendu final. En haut à gauche la carte, le cercle rouge indique la salle actuelle, en appuyant sur le bouton "enter room" on entre dans la salle. En haut à gauche une salle d'enemis, avec le(s) joueur(s) à gauche, le(s) ennemi(s) à droite, et les cartes du joueur actif en dessous. Les deux cartes au dos noirs indiquent la pioche et la défausse. En bas à gauche une salle de repos, on peut choisir de se soigner ou de méditer pour augmenter ses stats de base. En bas à droite une salle d'entraînement, on peut choisir une carte à ajouter au deck

3.2 Conception logiciel

L'image qui suit correspond à notre diagramme UML du Rendu d'un état.

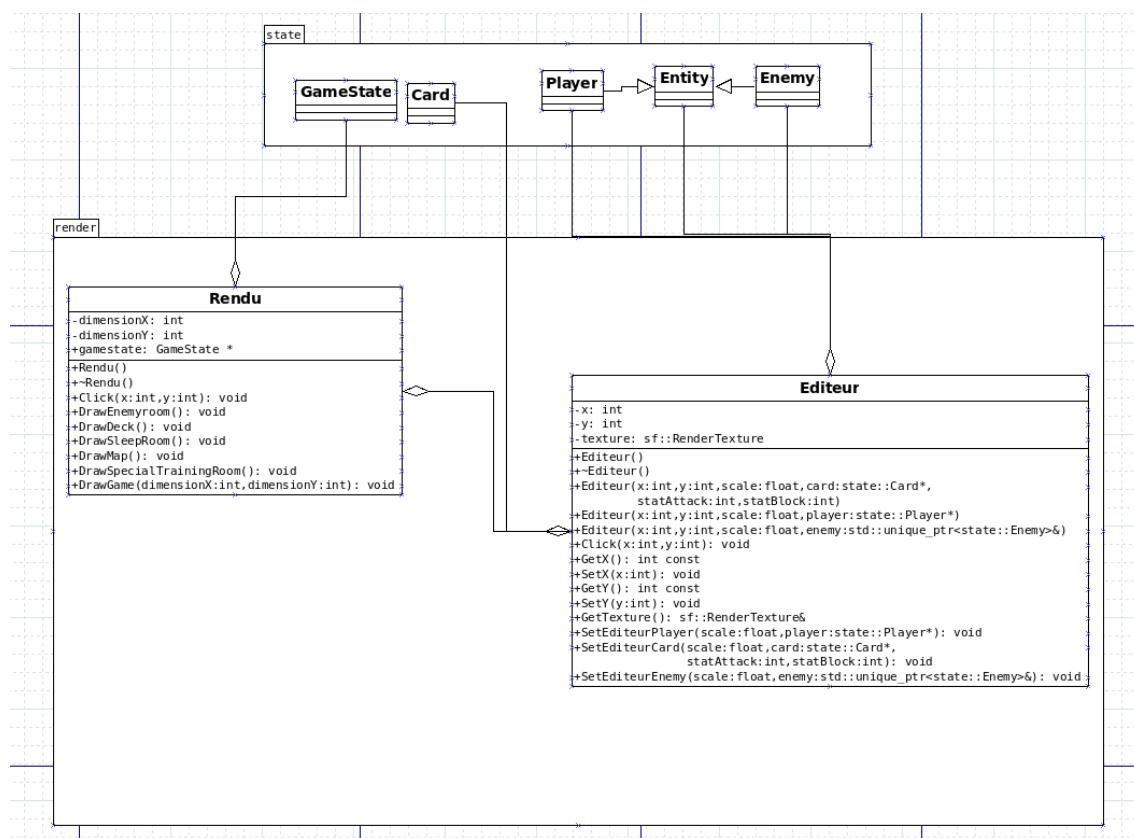


FIGURE 16 – Diagramme des classes de rendu.

4 Règles de changement d'états et moteur de jeu

4.1 Règles

Outre l'initialisation du jeu et la fin du jeu dont nous ne parlerons pas pour l'instant, on peut décomposer notre jeu en 2 états principaux : la carte, et l'intérieur des salles. Le jeu se termine lorsque toutes les salles sont parcourues sans mort des joueurs, ou lorsque que les joueurs sont morts.

4.1.1 La carte

A partir de la carte, seule une action est possible : passer à la salle suivante. Il n'y a pour l'instant qu'un unique chemin possible. Une fois que la salle suivante est sélectionnée, on rentre à l'intérieur de cette salle.

4.1.2 L'intérieur des salles

Salle d'ennemis

En entrant dans une salle d'ennemis, les joueurs affrontent 1 ou plusieurs ennemis. Les tours se déroulent selon la logique suivante : le joueur 1 joue son tour, puis le joueur 2 s'il existe, puis l'ennemi 1, l'ennemi 2 et 3 s'ils existent. Au début de son tour, un joueur voit son énergie réinitialisée à 3, ses buffs et débuffs baissent de 1, et il pioche 5 nouvelles cartes de sa pile de pioche (s'il n'y a plus assez de cartes dans la pioche, la défausse est mélangée et forme la nouvelle pioche) vers sa main. Les seules actions disponibles sont passer le tour ou jouer une carte.

Tant que le joueur possède l'énergie suffisante, il peut jouer des cartes. Pour jouer une carte, il sélectionne la carte en cliquant dessus, puis sélectionne la cible en cliquant dessus. S'il tente de jouer une carte avec un coût supérieur à son énergie, ou s'il sélectionne une mauvaise cible pour cette carte, elle ne sera pas jouée et restera dans sa main. Lorsqu'une carte est jouée, l'énergie du joueur baisse d'autant que le coût de la carte, ses effets sont appliqués (attaque, block, ajout de bonus ou malus, pioche, défausse, soin...), puis elle est défaussée. La taille maximale de la main est 7.

Lorsque le joueur décide de finir son tour (en appuyant sur "end turn"), toutes les cartes qu'il avait dans sa main sont mises dans la pile de défausse. L'entité suivante joue son tour.

Si c'est le tour d'un ennemi, l'ennemi réalise son action (attaque, block, application d'un bonus/-malus...) sur une cible valide.

Une fois que tous les ennemis sont morts (soit que leur vie est à 0), les joueurs peuvent sélectionner une carte de récompense. Le dernier ennemi tué définit les cartes obtenues en récompense. Une seule carte peut être sélectionnée par personne, mais cette carte peut être la même pour les deux joueurs. La carte sélectionnée est ajoutée au deck du joueur. Le deck possède déjà 15 cartes, alors le joueur doit définitivement enlever une carte de son deck avant de pouvoir ajouter la nouvelle. Les joueurs peuvent également choisir de ne pas sélectionner de carte. Après cette phase, les joueurs sortent de la salle et peuvent sélectionner la suivante depuis la carte.

Salle d'entraînement

En entrant dans une salle d'entraînement, trois cartes sont proposées. Les joueurs peuvent choisir une de ces cartes et l'ajouter à son deck (si le deck fait déjà 15 cartes, une carte doit préalablement être retirée). Cette carte peut être la même pour les deux joueurs. Ils peuvent aussi choisir de passer directement à la salle suivante en appuyant sur "pass". Ils passent alors à la salle suivante.

Salle de repos

En entrant dans une salle de repos, les joueurs ont deux choix possibles : se reposer et se soigner d'un nombre de PV indiqué, ou méditer et gagner un gain définitif en valeur d'attaque et de block.

4.2 Conception logiciel

Les figures qui suivent présentent notre diagramme de classe sous forme de diagramme UML.

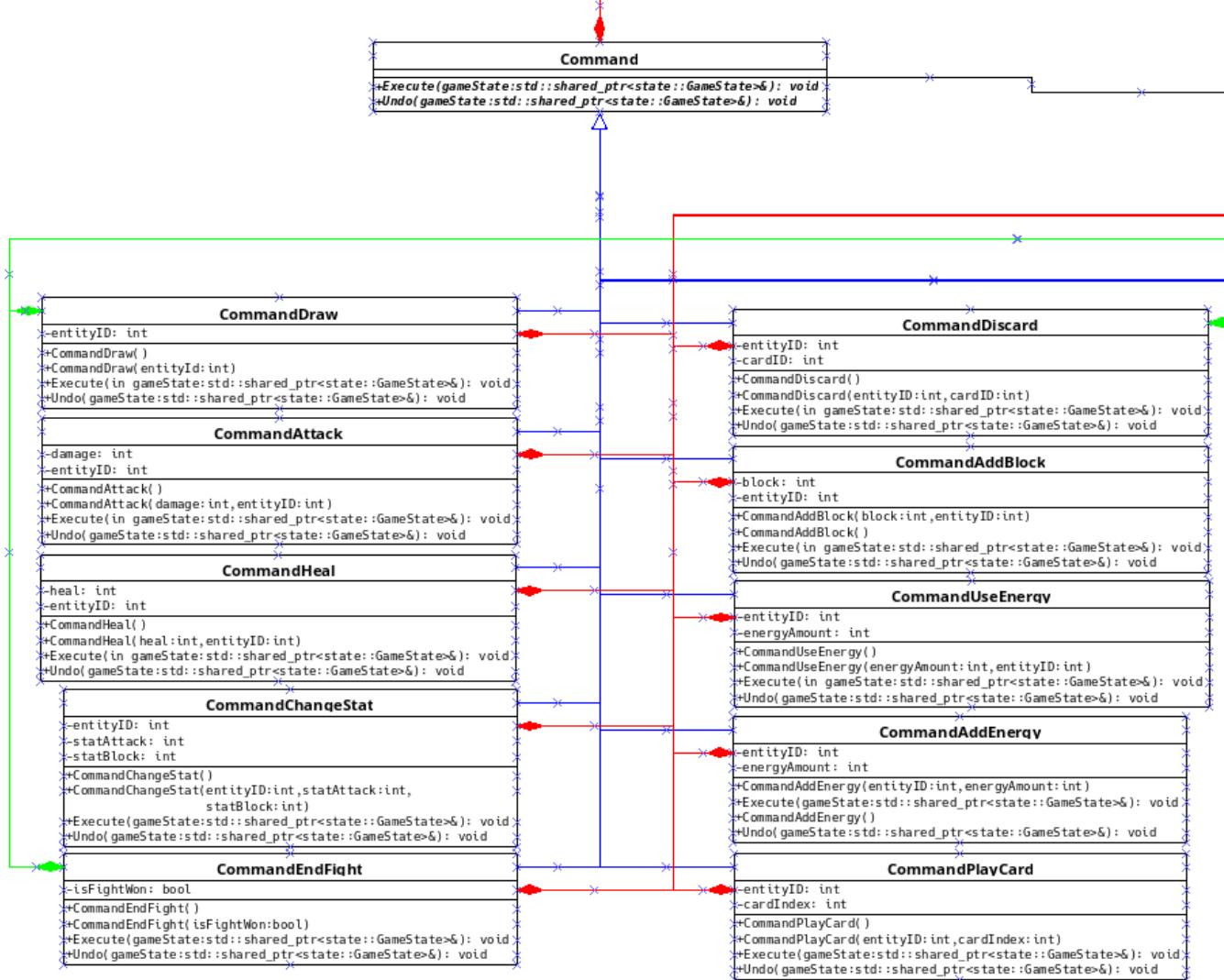


FIGURE 17 – Quelques commandes.

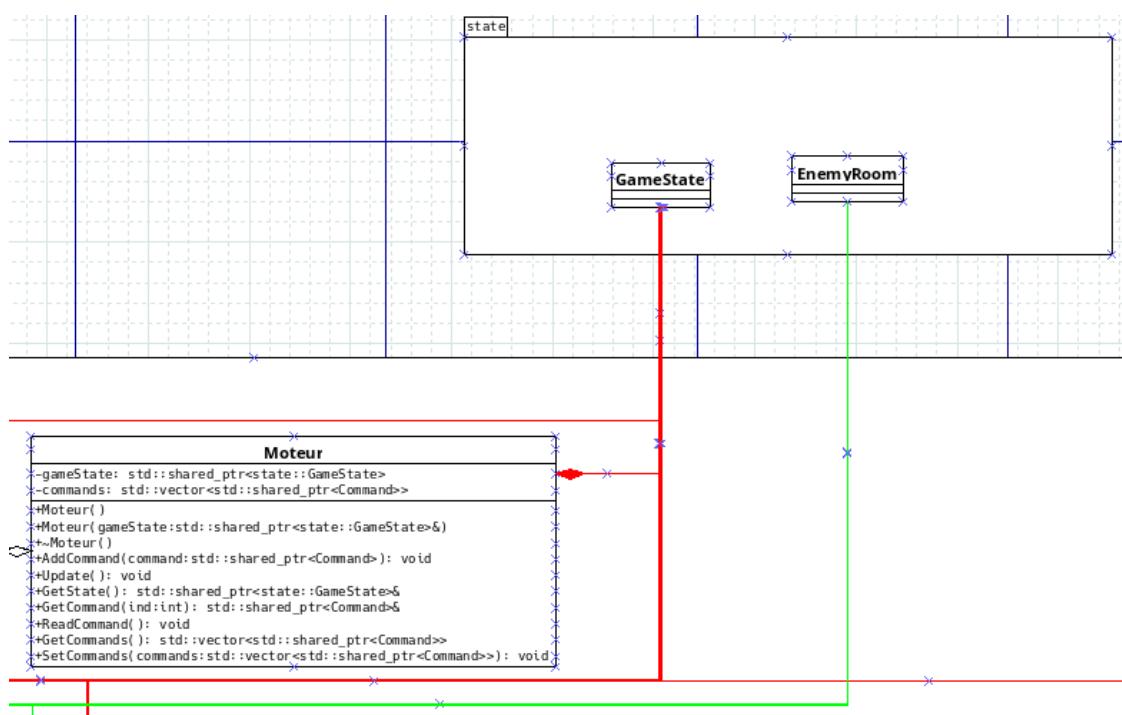


FIGURE 18 – Diagramme des classes de moteur de jeu.

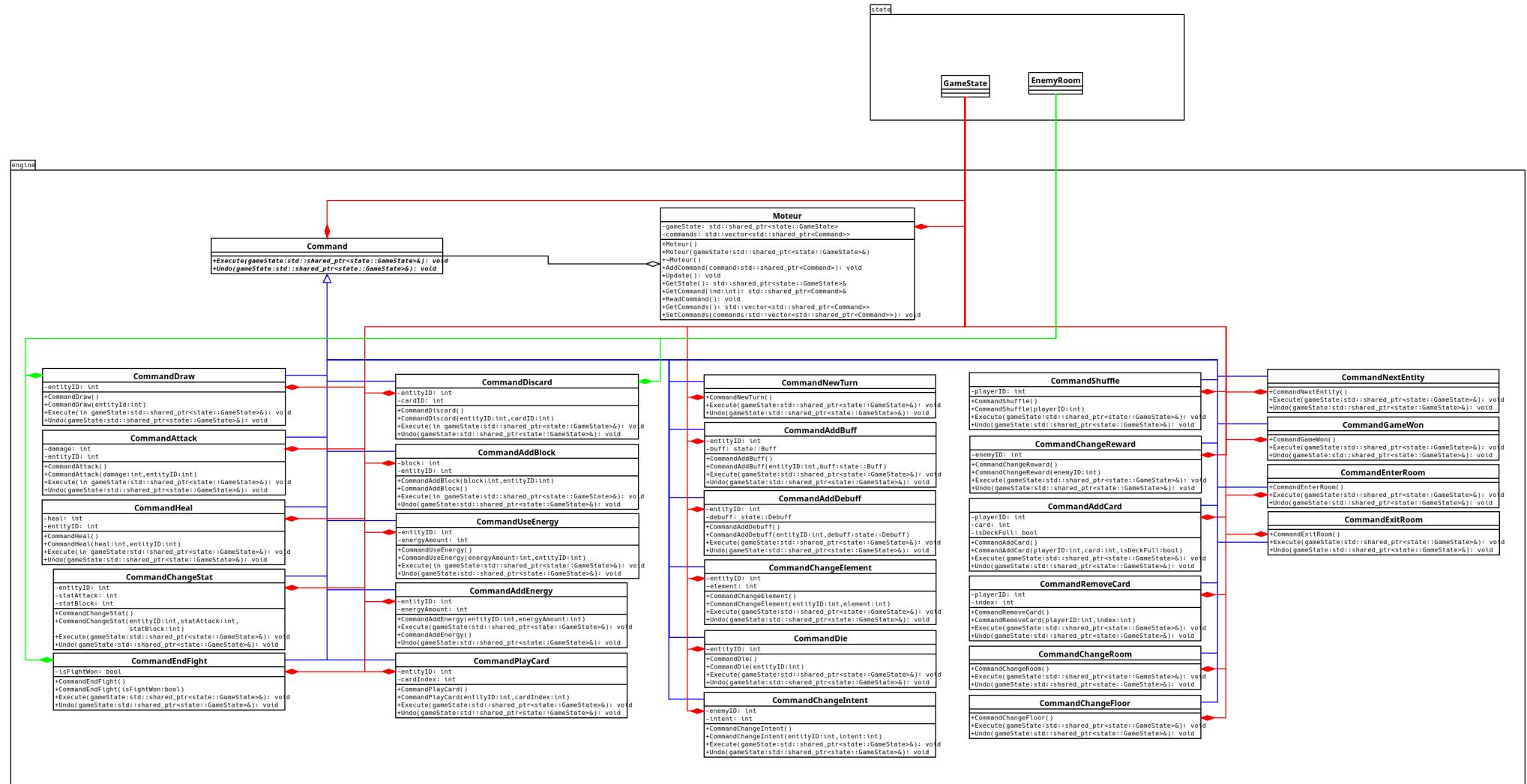


FIGURE 19 – Diagramme des classes de moteur de jeu.

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

Lorsque le personnage est dans une salle de combat, il joue aléatoirement des cartes jusqu'à une éventuelle fin de combat. Il peut aussi choisir de finir son tour. Lorsqu'il est dans la salle de repos, il choisira aléatoirement entre se soigner ou améliorer ses statistiques. Lorsqu'il est dans une salle d'entraînement ou si le joueur à remporté un combat, le personnage choisira aléatoirement une carte parmi les trois proposées pour rajouter à son deck, ou choisira de passer à la salle suivante directement. Si il est à l'extérieur d'une salle, et donc sur la carte, la seule option disponible est de rentrer dans la salle suivante.

5.1.2 Intelligence basée sur des heuristiques

Nous proposons un ensemble d'heuristiques pour chaque salle :

- **Salle de repos :** Si le joueur a moins de 50% de sa vie, il choisira de se soigner. Dans le cas contraire il méditera afin de gagner de l'attaque et du block définitifs.
- **Salle d'entraînement :** Dans cette salle, le joueur peut choisir entre 3 cartes. Il y a des cartes considérées comme offensives (les cartes qui attaquent, et/ou débuff les ennemis) et d'autres comme défensives (soin/block ainsi que les buffs). Afin de choisir une carte à rajouter au deck, le joueur va essayer d'avoir le même nombre de cartes offensives que de cartes défensives. Afin que son deck s'améliore, on oblige toujours l'IA à prendre une carte, même quand son deck est plein, et on retire une carte possédant le score le plus faible du deck. Ce score est choisi de manière arbitraire en fonction de toutes les caractéristiques de la carte, de telle sorte que les cartes du deck de départ soient les premières à être supprimées.
- **Salle d'ennemis :** Durant le combat, le joueur effectue un certain nombre d'étapes :
 - **Étape 1 :** Si le joueur est attaqué et qu'il n'a pas de buff "evade" sur lui, et si le joueur a dans les mains une carte donnant le buff "evade", il va la jouer.¹
 - **Étape 2 :** S'il n'a pas le buff evade appliqué sur lui, il va ensuite essayer de bloquer les dégâts que vont lui infliger les ennemis, jusqu'à ce qu'il n'ait plus d'énergie suffisante, de carte de block/soin, ou que l'intégralité des dégâts des ennemis soient bloqués.
 - **Étape 3 :** S'il en a la capacité, le joueur va ensuite d'attaquer les ennemis. Il va sélectionner le plus faible, et lui faire des dégâts s'il peut faire baisser sa vie (et non juste son block). Dans le cas contraire il n'attaquera pas.
 - **Étape 4 :** S'il lui reste de l'énergie après toutes ces étapes, il va jouer le reste de ses cartes aléatoirement.
- **Fin du combat :** En fin de combat, le joueur pourra choisir une carte à ajouter à son deck, il la choisira de la même manière qu'il choisit des cartes dans la salle d'entraînement.

L'élément du joueur et des ennemis est ici considéré. L'eau infligera plus de dégât à l'air, lui sur le feu, lui sur la terre et enfin cette dernière sur l'eau. De manière générale lors d'un combat, le joueur joue prioritairement les cartes dont les éléments lui sont défavorables, afin d'essayer de terminer le tour en étant d'un autre élément et ainsi ne pas fausser le calcul du block. Si à

1. La carte fournissant le buff "evade" étant unique et la plus coûteuse, elle ne serait pas jouée sinon, bien qu'elle donne un avantage certain.

l'étape 4, il ne lui reste que des cartes à jouer d'un élément défavorable, il n'en jouera aucune. On n'empêche cependant pas le joueur de jouer des cartes aux étapes 1, 2, 3.

5.1.3 Intelligence artificielle basée sur des arbres de recherche

Cette intelligence artificielle n'interviendra que dans le choix des cartes à jouer. Le reste des commandes effectuées se feront toujours avec le principe de l'ia heuristique (choix des cartes à ajouter au deck, dormir/méditer...).

Jouer des cartes se découpe maintenant en plusieurs étapes :

- Etape 1 : conception de l'arbre : l'arbre est construit à partir des cartes contenues dans la main du joueur (au nombre de 5). Toutes les cibles permises sont incluses. Les attributs d'un noeud de l'arbre est : son index, correspondant à l'index de la carte dans la main du joueur (< 0 si il ne correspond pas à une carte), sa cible (sous la forme d'un entityID), un booléen indiquant si la carte a été choisie, et un vector contenant les cartes/noeuds suivants. La racine de l'arbre possède l'index -2, tandis que les noeuds d'index -1 indiquent la commande "passer à l'entité suivante".
- Etape 2 : parcours de l'arbre et choix de la meilleure combinaison de cartes. Ce choix se fait basé sur un score. On parcourt l'arbre en récursif en clonnant les états. Comme on n'a que 5 cartes à prendre en compte ainsi que la commande passer le tour, ce choix ne semblait pas trop consommateur en mémoire. Le score est calculé à partir de :
 - la vie perdue par le joueur lorsqu'il a joué ses cartes et après calcul des dommages infligés par les ennemis
 - le nombre d'ennemis tués
 - le nombre de vie perdue par un ennemi s'il n'est pas tué, pour tous les ennemis
 - si le joueur est mort après avoir joué toutes ses cartes, imposant le score à $-\infty$
- Etape 3 : ajout des commandes correspondant aux cartes à jouer dans le moteur, ainsi que la commande fin du tour.

5.2 Conception logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 20.

AI_Random

Classe implantant l'intelligence aléatoire.

AI_Heuristic

Classe implantant l'IA basée sur une heuristique.

AI_Deep

Classe implantant l'IA basée sur les arbres de recherche.

Node

Classe implantant les noeuds de l'arbre de recherche étant utilisé dans la classe IA_Deep.

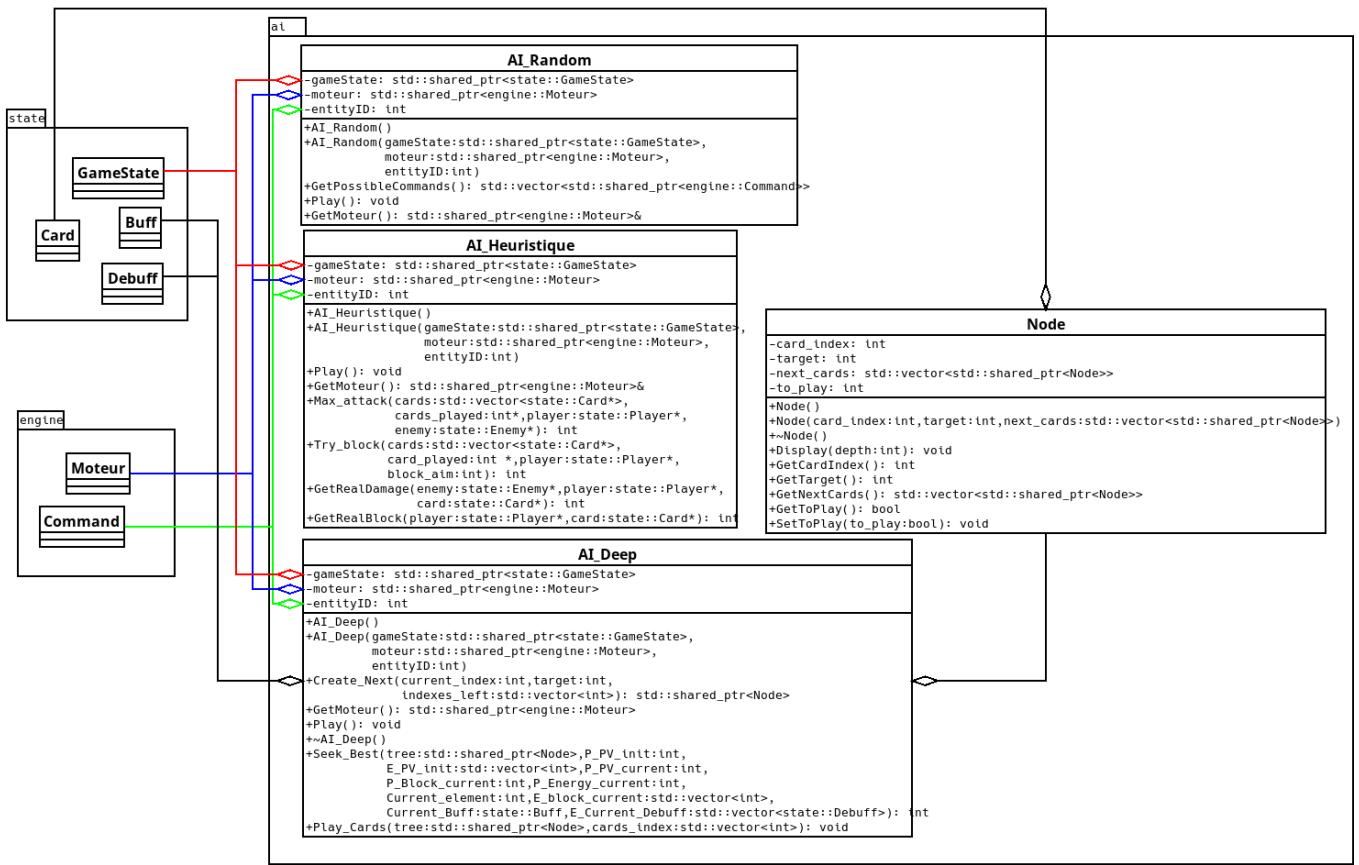


FIGURE 20 – Diagramme des classes d'intelligence artificielle.

6 Modularisation

6.1 Record/Replay

Lors de la fonction *update* du moteur, on ajoute la sérialisation des commandes à une Json : :Value *jsonval*. Cette Json : :Value est enregistrée dans le fichier *replay.txt* lors de l'appel du destructeur du moteur. On enregistre également une seed pour la fonction *rand()* utilisée dans le jeu. Afin d'enregister les commandes dans le fichier, on ajoute deux méthodes aux commandes : *Serialize* et *Deserialize*.

Serialize : défini et retourne une Json : :Value *val*. On ajoute à val tous les attribus nécessaires à la création d'une commande (par exemple, *entityID* pour la plupart des commandes). On ajoute également un champ commun à toutes les commandes : *typeCmd* qui définit le type de la commande sérialisée.

Deserialize : à partir d'une Json : :Value *in*, on désérialise une commande. On reprend tous les arguments correspondant à la commande pour les mettre dans les champs correspondants. Puis on renvoie la commande ainsi initialisée.

On ajoute une méthode *ReadCommand* au moteur afin de lire toutes les commandes du fichier *replay.txt*.

ReadCommand : prend une Json : :Value *val* en entrée. Décompose val en différentes commandes. On analyse le champ "typeCmd" afin de désérialiser la commande correspondante, les attributs variant d'une commande à l'autre. On enregistre toutes les commandes ainsi re-crées dans le vecteur de commandes du moteur, qui les réalisera à l'aide de sa fonction "update".

6.2 Répartition sur différents threads

Afin de préparer la séparation sur plusieurs machines (une se chargeant d'exécuter le moteur : le serveur, et l'autre gérant le rendu : le client) et d'optimiser l'exécution du jeu, nous allons séparer notre jeu en deux threads. Alors que le thread principal sera responsable du rendu graphique du jeu, le second se chargera de gérer le rendu.

Afin de ne pas afficher une ressource qui est en cours de modification, nous utilisons un mutex, ce qui permet que les parties critiques du code ne soient pas utilisées en même temps par plusieurs threads séparés. De plus, La communication entre les threads se fait par le biais de pointeurs vers des booléens et vers l'état du jeu. Lors de la mise en réseau du jeu, la communication se fera par l'intermédiaire de fichiers Json.

6.3 Répartition sur différentes machines : Lobby

Afin de jouer à un jeu en réseau, il faut que les joueurs puissent se connecter au serveur, ce qui consiste à faire une liste de clients côté serveur. Dans ce but, plusieurs requêtes du client vers le serveur ont été implémentées :

— **Requête GET** :

— /user/<id> si <id> >= 0 : Renvoie toutes les informations sur l'utilisateur correspondant

- /user/<id> si <id>< 0 : Renvoie la liste des utilisateurs connectés au serveur.
- **Requête POST** : /user/<id> change les informations de l'utilisateur correspondant
- **Requête PUT** : /user crée un nouvel utilisateur avec les données fournies dans la requête
- **Requête DELETE** : /user/<id> supprime l'utilisateur correspondant

6.4 Conception logiciel

Le diagramme des classes pour le serveur est présenté en Figure 21

Afin de rendre les communications client/serveur plus intuitives, nous avons choisi de créer une classe qui sera chargée de la communication du client : la classe networkManager (Figure 22). La méthode InitConnection permettant de mettre le nom du client vers le serveur et d'initialiser son ID. On peut mettre dans le destructeur du networkManager l'envoi d'une requête Delete avec son id afin de supprimer le joueur de la liste lors de sa déconnexion.

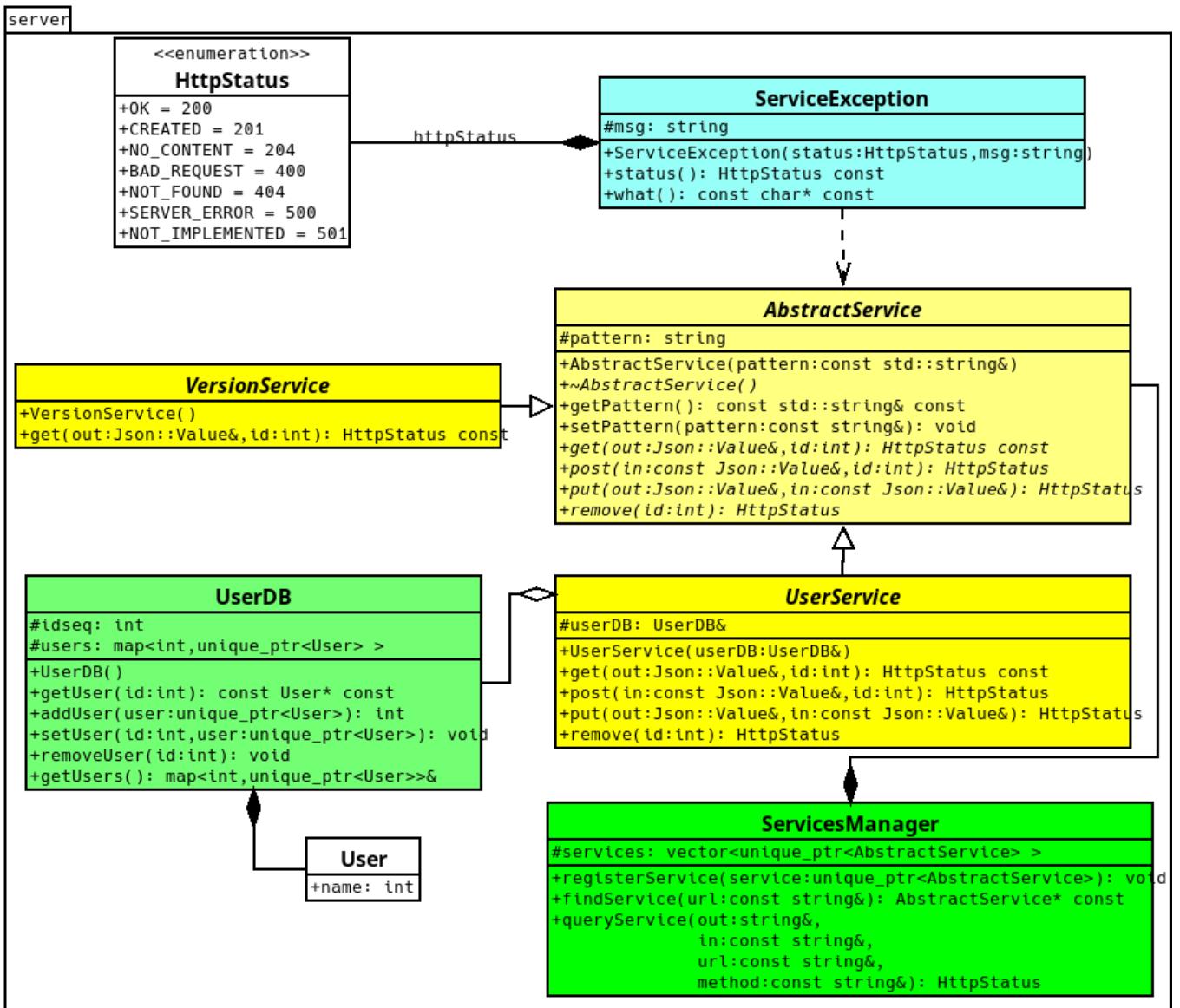


FIGURE 21 – Diagramme des classes pour le serveur.

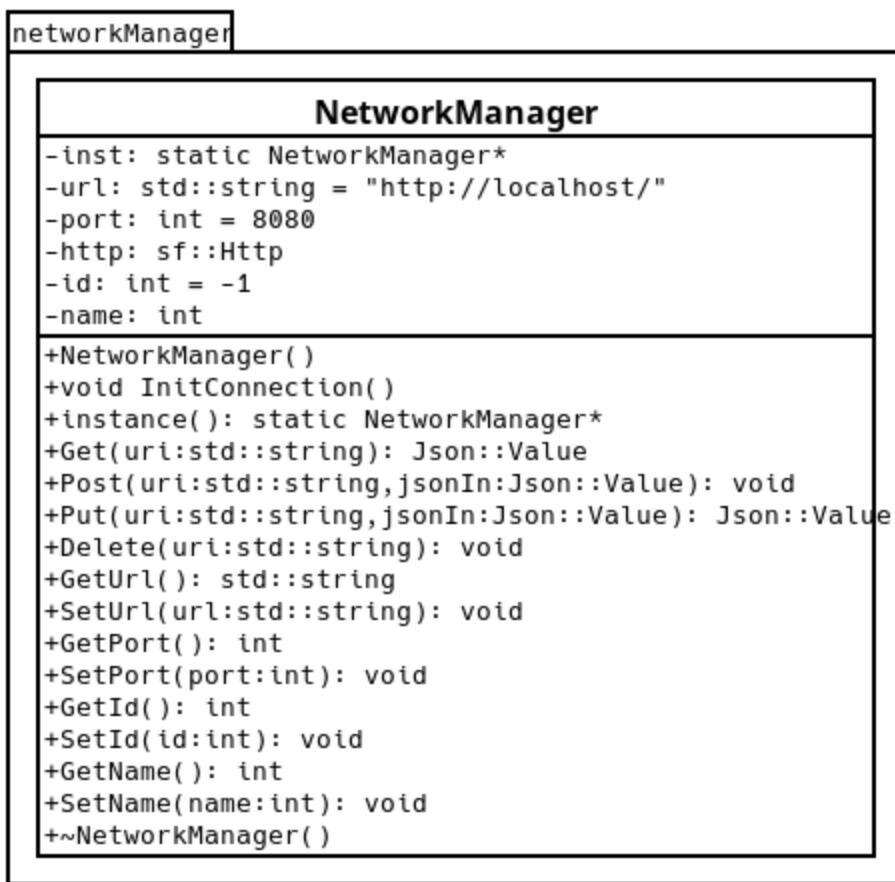


FIGURE 22 – Diagramme des classes pour le networkManager.