Projet Logiciel Transversal

Dragons & Licornes

Kevin TOOMEY – Chloé LODOLO

Table des matières

1 Présentation générale

1.1Archétype

L'objectif de ce projet est de réaliser un jeu que l'on peut trouver sur internet sous le nom de World Wars, basé sur l'archétype du Risk, mais en s'éloignant de l'approche militaire du jeu et en rendant le thème plus léger. Ainsi, au lieu d'une bataille entre missiles et tanks, notre jeu mettra en scène le combat entre une armée de licornes et une armée de dragons.

1.2Règles du jeu

Chaque joueur est le dirigeant d'une armée : l'un possède une armée de licornes, l'autre une armée de dragons. Le but du jeu est de conquérir tous les territoires ennemis.

Au début du jeu, les armées de deux joueurs sont égales en quantité, mais les territoires possédés par chaque armée et le nombre de soldats présents sur chaque territoire sont placés aléatoirement.

Le joueur actif choisit le territoire qui attaque ainsi que le territoire ennemi qu'il veut attaquer, tout en sachant qu'il ne peut attaquer qu'un territoire frontalier. Un lancer de dés a alors lieu : chacun des deux joueurs a autant de lancers de dés qu'il a de soldats sur son territoire. Les résultats des lancers de chaque joueur sont ensuite sommés.

Si celui qui fait le score le plus haut est l'attaquant, alors celui-ci devient propriétaire du territoire ennemi et tous les soldats du territoire attaquant sauf un migrent vers le territoire nouvellement conquis, offrant un territoire en plus pour l'attaquant et une nouvelle répartition des soldats.

En revanche si celui qui fait le score le plus haut est le défenseur alors le territoire attaquant perd tous ses soldats sauf un.

Lorsque le joueur considère qu'il a fini son tour, il passe la main à l'autre joueur.

1.3Ressources

Pour réaliser ce projet, nous utiliserons les trois textures suivantes :

Figure 1 – Textures des « soldats »

Figure 2 – Textures des bulles indiquant le nombre de soldats sur un territoire

Figure 3 –
Textures des différents territoires : inaccessible, occupé par les dragons, occupé par les licornes

2 Description et conception des états

2.1 Description des états

Un état du jeu est formé d'éléments que l'on a divisé en deux catégories : les éléments fixes et les éléments variables. Les territoires (ou cellules) du plateau sont des éléments fixes tandis que l'armée occupant un territoire et le nombre de soldats présents sur ce territoire sont des éléments variables. Tous les éléments possèdent des coordonnées (x,y) permettant de les repérer sur le plateau et un identifiant qui indique sa nature (fixe ou variable).

2.1.1 Eléments fixes

Le plateau du jeu est formé par un ensemble de cellules de forme hexagonale représentant les territoires pouvant appartenir à l'un ou l'autre des joueurs.

Certains territoires sont inaccessibles : aucun des deux joueurs ne peut envahir ce type de territoire. Ils ont pour but de permettre la modification du plateau à chaque partie différente et de complexifier légèrement le jeu en constituant des obstacles. Ils sont représentés par une cellule avec des montagnes.

Le reste des territoires appartient forcément à l'un ou à l'autre des joueurs. Si le territoire appartient aux licornes, il est bleu ; s'il appartient aux dragons, il est rouge.

2.1.2 Eléments variables

Selon l'évolution du jeu, un territoire – à partir du moment où il est accessible – peut appartenir soit à l'armée des licornes, soit à l'armée des dragons. Les données concernant l'armée qui occupe un territoire et le nombre de soldats présents sont variables.

En effet le nombre de soldats évolue forcément lors d'une attaque et peut aller de 1 à 8 soldats, et après un combat un territoire des licornes peut devenir propriété des dragons et inversement.

2.1.3 Etat général

L'état général est décrit par deux tableaux, l'un contenant les éléments fixes (les territoires) et l'autres les éléments variables (le type de l'armée et son nombre). Ces deux tableaux permettent de décrire l'état du jeu à un instant donné.

2.2 Conception logiciel

Le diagramme des états est présenté en Illustration 1. On distingue deux types de classes : les éléments et les conteneurs d'éléments.

La sémantique des couleurs utilisées est la suivante :

- Le rose symbolise les classes d'éléments. Le rose le plus foncé correspond à la classe mère, abstraite ; le rose plus clair correspond aux deux classes filles
- Le vert a quant à lui été utilisé pour les classes de conteneurs d'éléments

2.2.1 Les éléments

Nous avons trois classes : la classe Element, la classe Territory et la classe Team.

La classe Element est la classe mère des deux autres classes. Elle contient deux attributs qui sont les coordonnées x et y de chaque élément dans la grille du plateau de jeu, un constructeur et un destructeur, et une méthode nommée isStatic(). Cette dernière méthode est abstraite, et nous permet de savoir si un élément est variable ou fixe ; nous avons créé cette méthode car il n'y a pas de possibilité d'introspection d'une instance en C++.

La classe Territory a été créée pour représenter les cellules constituant les territoires du jeu. Elle possède un unique attribut, territoryStatus, dont la vocation est d'indiquer si la cellule peut être occupée par une armée ou non : l'attribut peut valoir accessable ou impossible ; ansi qu'un constructeur, un getter pour obtenir la valeur de l'attribut territoryStatus et la redéfinition de la méthode isStatic().

La classe Team sert à représenter l'armée. Elle a donc deux attributs. L'un indique de quelle armée il s'agit : c'est l'attribut teamStatus, dont les deux valeurs possibles dragons et unicorns ; l'autre, nommée nbCreatures, donne le nombre de soldats occupant la case : un entier entre 1 et 8. Elle possède également quelques méthodes : un constructeur, les getters et setters de teamStatus et nbCreatures et la redéfinition de la méthode isStatic().

2.2.2 Les conteneurs d'éléments

On distingue la classe ElementTab de la classe State.

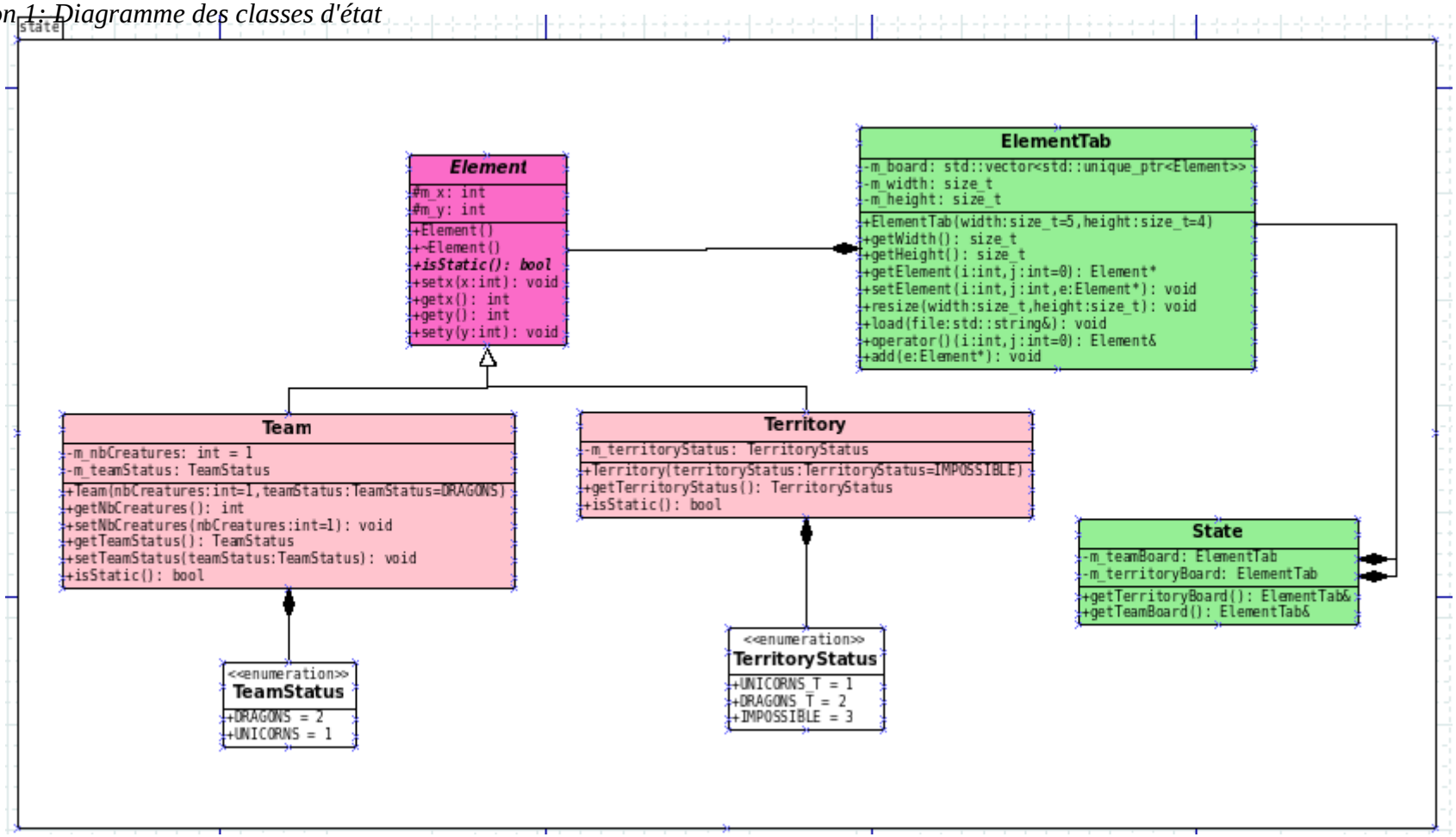
La première nous permet de construire des tableaux à deux dimensions contenant des objets de type Element, offrant la possibilité de créer notre plateau de jeu à base des cellules de type Territory.

La classe State permet quant à elle d’accéder à la totalité des données de l’état.

2.3 Conception logiciel : extension pour le rendu

2.4 Conception logiciel : extension pour le moteur de jeu

Illustration 1: Diagramme des classes d'état



3 Rendu : Stratégie et Conception

Le diagramme du rendu est présenté en Illustration 2. On distingue quatres types de classes : les Layers, la Surface, la Tile et les TileSets.

La sémantique des couleurs utilisées est la suivante :

- Le bleu symbolise les classes Layers. Le bleu le plus foncé correspond à la classe mère, abstraite ; le bleu plus clair correspond aux deux classes filles
- Le vert a quant à lui été utilisé pour la classe Surface
- Le violet représente la classe Tile
- Le rose symbolise les classes TileSet. Le rose le plus foncé correspond à la classe mère, abstraite ; le rose plus clair correspond aux trois classes filles

3.1 Stratégie de rendu d'un état

Pour effectuer le rendu, nous avons choisi de découper la scène à rendre en 3 plans. Un plan pour afficher tous les territoires, les hexagones marrons, beiges et montagneux. Le second plan est constitué des icônes des dragons et des licornes à placer sur les bonnes tuiles. Le dernier a pour fonction d'afficher le nombre de créatures présentes sur chaque tuile.

Chaque plan contiendra deux informations qui seront transmises à la carte graphique : une unique texture contenant les tuiles, et une unique matrice avec la position des éléments et les coordonnées dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.

Pour la formation de ces informations bas-niveau, la première idée est d'observer l'état à rendre, et de réagir lorsqu'un changement se produit. Si le changement dans l'état donne lieu à un changement permanent dans le rendu, on met à jour le morceau de la matrice du plan correspondant. Pour les changements non permanents, comme les animations et/ou les éléments mobiles, nous tiendrons à jour une liste d'éléments visuels à mettre à jour (= modifier la matrice du plan) automatiquement à chaque rendu d'un nouveau frame.

3.2 Conception logiciel

Le diagramme des classes pour le rendu général est présenté en Illustration 2.

Layers. Le cœur du rendu réside dans le groupe (en bleu) autour de la classe Layer. Le principal objectif des instances de Layer est de donner les informations basiques pour former les éléments bas-niveau à transmettre à la carte graphique. Ces informations sont données à une instance de Surface, et la définition des tuiles est contenu dans une instance de TileSet. Les classes filles de la classe Layer se spécialise pour l'un des plans à afficher. Par exemple, la classe StateLayer va afficher le nombre de pastilles, et la classe ElementTabLayer peut afficher le niveau ou les personnages. La méthode initSurface() fabrique une nouvelle surface, lui demande de charger la texture, puis initialise la liste des sprites. Par exemple, pour afficher le niveau, elle demande un nombre de quads/sprites égal aux nombre de cellules dans la grille avec initQuads(). Puis, pour

chaque cellule du niveau, elle fixe leur position avec setSpriteLocation() et leur tuile avec setSpriteTexture().

Surfaces. Chaque surface contient une texture du plan et une liste de paires de quadruplets de vecteurs 6 2D. Les éléments texCoords de chaque quadruplet contient les coordonnées des quatre coins de la tuile à sélectionner dans la texture. Les éléments position de chaque quadruplet contient les coordonnées des quatre coins du carré où doit être dessiné la tuile à l'écran.

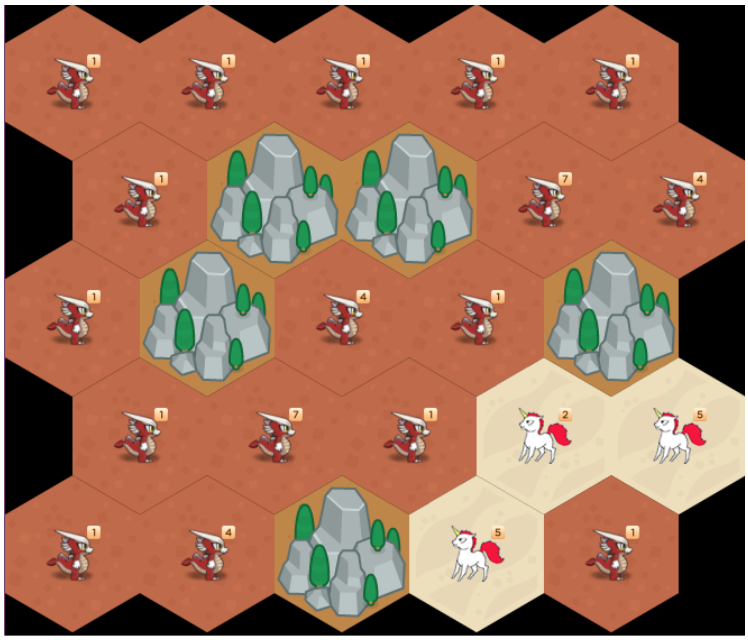
Tuiles. Les classes filles de TileSet regroupent toutes les définitions des tuiles d'un même plan. Par exemple, elle sait que les coordonnées de la tuile avec la créature dragon est (50,50) et de taille (50,50). Pour obtenir une telle information, un client de ces classes utilise la méthode getTile(). Par exemple si on passe à cette méthode une instance de la classe Team, alors elle va renvoyer une instance de Tile qui correspond à cet élément, et en fonction de ses propriétés TeamStatus.

3.3 Conception logiciel : extension pour les animations

Dans notre cas, nous avons aucune animation indispensable, tous les changements seront faits lors des phases du tour des joueurs.

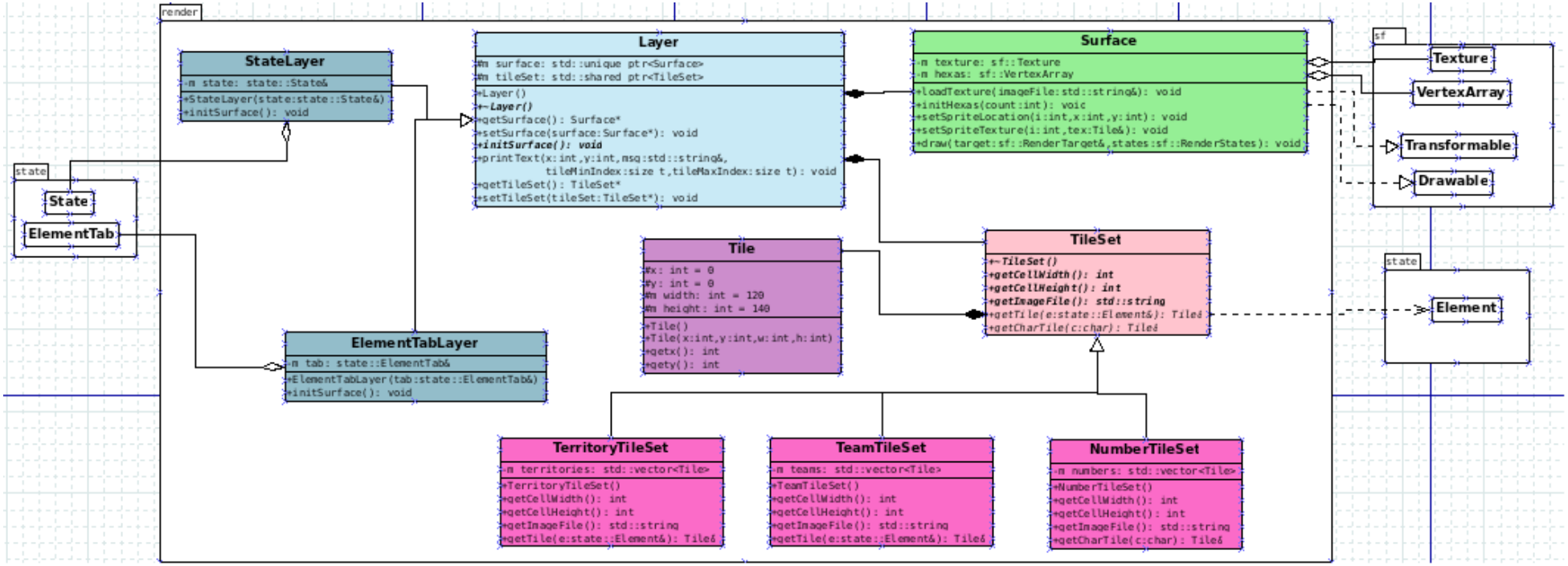
3.4 Ressources

3.5 Exemple de rendu



Exemple de rendu

Illustration 2: Diagramme de classes pour le rendu



4 Règles de changement d'états et moteur de jeu

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects lié au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

4.1 Horloge globale

On passe d'un état à un autre sans transition, il n'y a pas de notion d'état intermédiaire.

4.2 Changements extérieurs

Les changements exterieurs sont réalisés par l'utilisateur à l'aide de la souris. Dans une version suivante, il pourra selectionner la case avec laquelle il veut attaquer puis celle qu'il veut attaquer.

4.3 Changements autonomes

En ce qui concerne les changements autonomes, il y en a deux types : l'initialisation de l'état de départ ainsi que la gestion des créatures en renforts qui est géré en fin de tour, après les différentes attaques d'une équipe.

Le moteur est aussi fait tel que si on attaque une case que l'on est pas sensé pouvoir attaquer comme une case avec le titre de territoire IMPOSSIBLE ou un territoire de la même équipe que cele qui attaque, rien ne se passe, l'état n'est pas modifié.

Initialisation de l'état : Il est réalisé case par case d'une manière bien définie. Il sera plus tard, dans une version ultérieur, modifié pour que la carte soit générée aléatoirement.

Gestion des renforts : On ajoute une créature sur chaque case de l'équipe qui vient de finir ses attaques.

4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Illustration 3. L'ensemble du moteur de jeu repose sur un patron de conception de type Command, et a pour but la mise en œuvre différée de commandes extérieures sur l'état du jeu.

On distingue 3 types de classes : les Commandes, le moteur et l'ID de la commande sous forme d'énumeration.

La sémantique des couleurs utilisées est la suivante :

- Le bleu symbolise les classes Command. Le bleu le plus foncé correspond à la classe mère, abstraite ; le bleu plus clair correspond aux deux classes filles

- Le vert a quant à lui été utilisé pour la classe Engine .
- Le blanc représente l’énumération CommandTypeId.

Classes Command : Le rôle de ces classes est de représenter une commande, quelque soit sa source (automatique, clavier, réseau, ...). Notons bien que ces classes ne gère absolument pas l’origine des com- mandes, ce sont d’autres éléments en dehors du moteur de jeu qui fabriquerons les instances de ces classes. A ces classes, on a défini un type de commande avec CommandTypeId pour identifier précisément la classe d’une instance.

- InitBasicState Initialise l’état de départ du jeu.

- AttackCommand : Permet d’effectuer l’attaque d’une case à partir d’une autre. Elle calcul lequel des deux territoires gagne la bataille.

- GestionRenforts : Effectue l’attribution de renforts aux territoires de l’équipe qui vient de finir son tour.

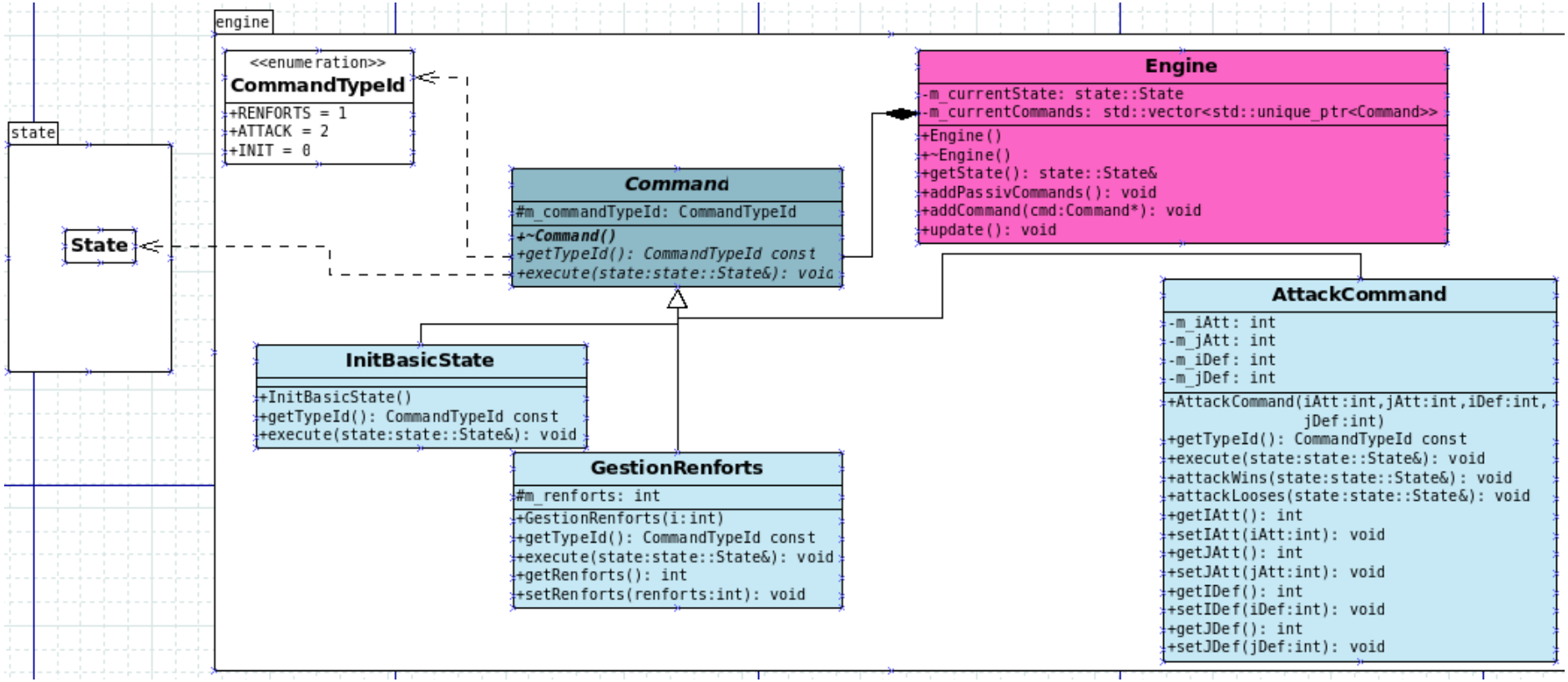
Engine : C’est le coeur du moteur. Elle stocke les commandes dans un std : :vector .

La méthode update() permet d’executer les commandes de la liste et ainsi la vider dans l’ordre ancien vers récent. Dans cette méthode, on fait appel aux méthodes execute() de chaque commande pour modifier l’etat.

4.5 Conception logiciel : extension pour l'IA

4.6 Conception logiciel : extension pour la parallélisation

Illustration 3: Diagrammes des classes pour le moteur de jeu



5 Intelligence Artificielle

Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.

5.1 Stratégies

5.1.1 Intelligence minimale

5.1.2 Intelligence basée sur des heuristiques

5.1.3 Intelligence basée sur les arbres de recherche

5.2 Conception logiciel

5.3 Conception logiciel : extension pour l'IA composée

5.4 Conception logiciel : extension pour IA avancée

5.5 Conception logiciel : extension pour la parallélisation

6 Modularisation

Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est un parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau

6.4 Conception logiciel : client Android

Illustration 4: Diagramme de classes pour la modularisation