

# Projet Logiciel Transversal

**Alexis LOISON – Eric TAN**

Option Informatique et Systèmes



*Capture d'écran du jeu*

## SOMMAIRE

<b>1</b>	<b>Objectif.....</b>	<b>3</b>
1.1	Présentation générale.....	3
1.2	Règles du jeu .....	3
1.3	Conception Logiciel .....	3
<b>2</b>	<b>Description et conception des états.....</b>	<b>4</b>
2.1	Description des états.....	4
2.1.1	Etat éléments fixes .....	4
2.1.2	Etat éléments mobiles .....	4
2.1.3	Etat général .....	4
2.2	Conception logiciel.....	5
2.3	Conception logiciel : extension pour le rendu.....	5
2.4	Conception logiciel : extension pour le moteur de jeu .....	5
2.5	Ressources .....	5
<b>3</b>	<b>Rendu : Stratégie et Conception.....</b>	<b>8</b>
3.1	Stratégie de rendu d'un état .....	8
3.2	Conception logiciel.....	8
3.3	Ressources .....	8
3.4	Exemple de rendu .....	9
<b>4</b>	<b>Règles de changement d'états et moteur de jeu .....</b>	<b>11</b>
4.1	Changements extérieurs .....	11
4.2	Changements autonomes .....	11
4.3	Conception Logiciel .....	12
<b>5</b>	<b>Intelligence artificielle .....</b>	<b>14</b>
5.1	Stratégies .....	14
5.1.1	Intelligence minimale.....	14
5.1.2	Intelligence intermédiaire .....	14
5.1.3	Intelligence avancée .....	14
5.2	Conception Logiciel .....	15
<b>6</b>	<b>Modularisations.....</b>	<b>17</b>
6.1	Organisation des modules .....	17
6.1.1	Répartition sur différents threads .....	17

# 1 Objectif

## 1.1 Présentation générale

Le jeu proposé est basé sur « Advance Wars », un jeu de guerre militaire en 2 dimensions avec la possibilité d’avoir plusieurs joueurs qui puissent y jouer. Nous travaillerons avec 2 joueurs ou bien 1 joueur contre 1 IA.



Illustration 1 : Capture d’écran du jeu

## 1.2 Règles du jeu

Les joueurs doivent capturer toutes les infrastructures ennemies ou uniquement le quartier général.

Si le joueur perd toutes ses infrastructures ou son quartier général il perd la partie.

Chaque joueur peut créer des unités à partir de bâtiments de production qui seront construits grâce à des ressources récoltées/perçues en jeu.

Chaque unité a des points de vie, des avantages et des faiblesses face à d'autres unités. Elle pourra des points de vie en retournant à l'usine(terrestres) ou à l'aéroport(aériens).

## 1.3 Conception Logiciel

**Package state** : il s’agit du package contenant toutes les informations pour représenter un état de notre jeu.

**Package render** : celui-ci permet d’afficher un état de notre jeu en version graphique.

**Package engine** : son but est de permettre à une personne ou une IA de jouer au jeu. Ainsi ce package permet de modifier un état de jeu en fonction d’événements comme une action au clavier ou à la souris par exemple.

On peut ainsi représenter le diagramme des packages suivant pour notre projet :

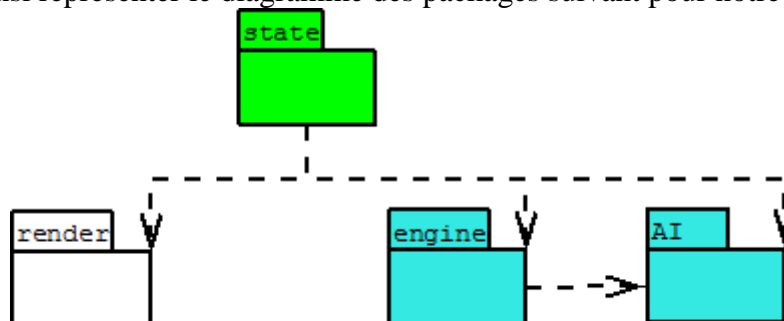


Illustration 2 : Diagramme des packages

## 2 Description et conception des états

### 2.1 Description des états

L'ensemble des états fixes et mobiles sont composés de :

- Coordonnées (x,y) dans la grille de chaque entité
- Identifiant du type d'élément : la nature de l'élément et sa couleur

#### 2.1.1 Etat éléments fixes

Notre carte sera composée de « cases ». La taille est définie par défaut. Chaque case ne pourra accueillir qu'un maximum de 10 unités du même élément à la fois.

- **Cases « Obstacles ».** Celles-ci sont infranchissables par certains éléments mobiles ou franchissables avec pénalités.
  - ➔ « Rivière » : franchissable uniquement par les éléments mobiles « aériens »
  - ➔ « Forêt/Bois » : franchissable par les éléments mobiles « aériens » et les éléments « infanteries » avec pénalité
- **Cases « Franchissables ».** Toutes les unités mobiles peuvent franchir ces éléments.
  - ➔ « Vide » : case ayant pour texture esthétique : une route ou de l'herbe
  - ➔ « Mine » : case rapportant des ressources de métal au détenteur
  - ➔ « Immeuble » : procure une diminution des dégâts reçus pour les infanteries
  - ➔ « Quartier Général » : permet aux éléments mobiles de récupérer des points de vie. La capture de cette case par un élément mobile ennemie procure la défaite de son ex-détenteur.
  - ➔ « Usine » : permet de créer des éléments « terrestres » mobiles ou leur permet récupérer des points de vie
  - ➔ « Aéroport » : permet de créer des éléments « aériens » ou leur permet récupérer des points de vie

#### 2.1.2 Etat éléments mobiles

Les éléments mobiles possèdent :

- Un **déplacement** : vers la gauche, vers la droite, vers le haut, vers le bas
- Une **vitesse de déplacement** définissant le nombre de cases que peut parcourir un élément mobile en 1 tour de jeu
- Une **position** [x,y] définit sur le cadre
- Des **dégâts**
- Des **points de vie** : entier allant de 0 à 10
- Une **couleur** : 1 pour le rouge, 2 pour le bleu et 3 pour le gris
- Un **rang** caractérisé par un chiffre : 1 pour une promotion et 0 pour une non-promotion
- Ces éléments peuvent être « aérien » ou « terrestre ».

#### 2.1.3 Etat général

Les propriétés suivantes sont ajoutées :

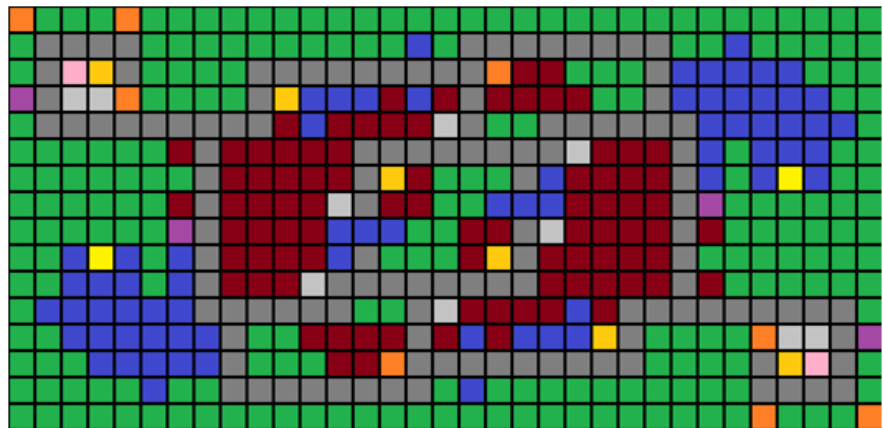
Epoque : « l'horloge » du jeu, l'heure depuis le commencement de la partie



GGGGGSGRRRRRRRRGSGGGGGGGGGRRRRG  
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGFGGGF

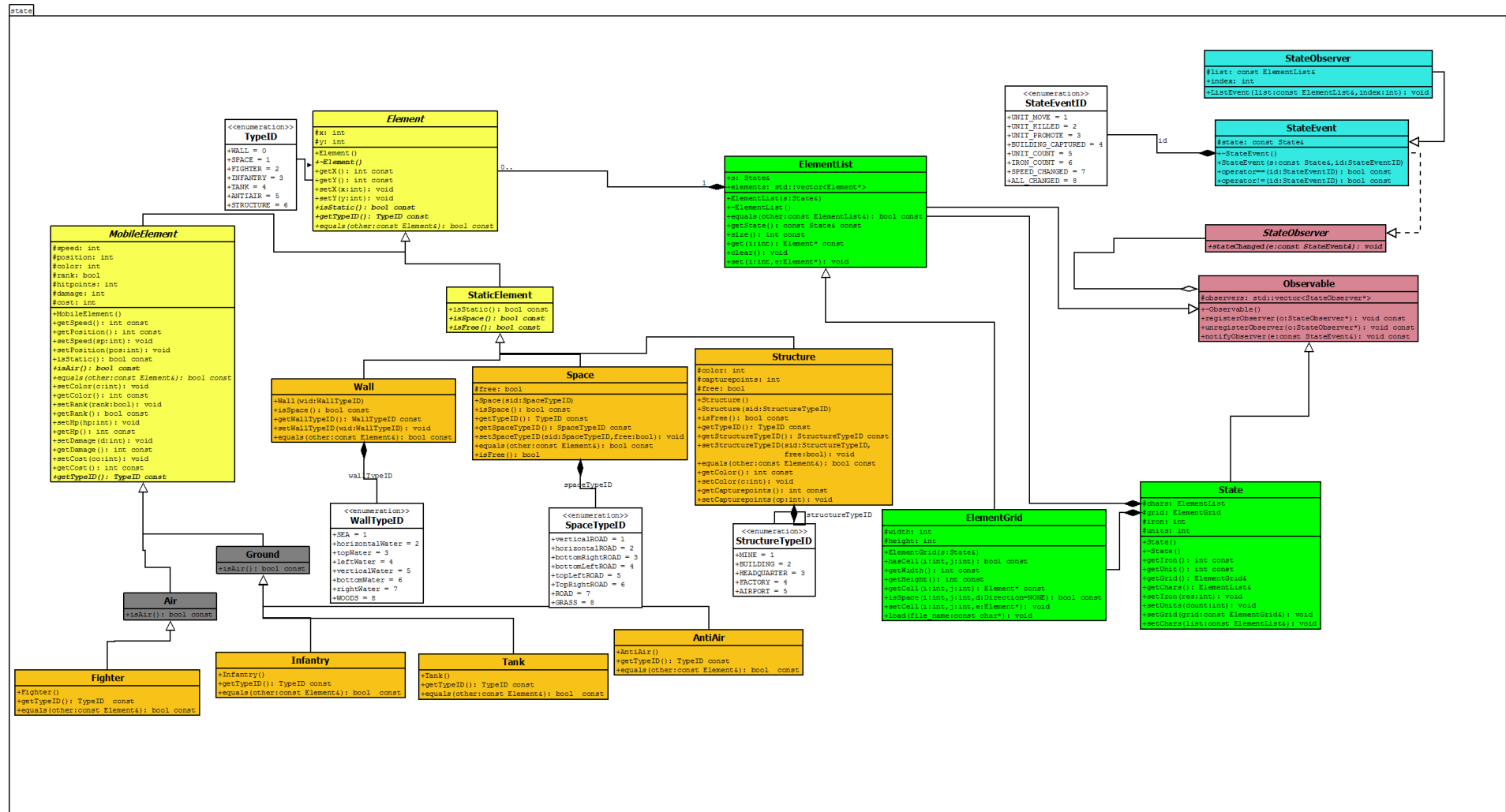
*Pour y voir plus clairement*

R/Road/Gris  
G/Grass/Vert  
M/Mine/Jaune  
B/Building/Gris clair  
H/Headquarter/ Rose  
F/Factory/Orange  
A/Airport/Violet  
S/Sea/Bleu/Marron  
W/Woods



*Illustration 3 : carte de jeu*

*Illustration 4 : Diagramme des classes d'état*



## 3 Rendu : Stratégie et Conception

### 3.1 Stratégie de rendu d'un état

Pour faire le rendu d'un état, nous utilisons SFML : Simple and Fast Multimedia Library. Celle-ci va nous permettre d'alléger le traitement des tuiles faites par la carte graphique de l'ordinateur.

Le principe utilise un système avec des vertex contenant plusieurs informations : une position, une couleur et une paire de coordonnées pour un jeu en 2 dimensions. Lorsque l'on groupe ces vertex ensemble (primitives), on obtient un tableau qui nous donne une forme.

Dans notre jeu, on affiche d'abord toute la carte avec les textures correspondantes aux bons tableaux de vertex. Ensuite, on peut par la suite changer les attributs des vertex qui nous intéressent pour modifier le rendu (par exemple signaler un changement d'unité/bâtiment d'une case de notre terrain).

Il faut également noter que nous aurons deux « fenêtres », une qui affiche le terrain et une autre qui montre les informations du joueur (ressources, couleur) ainsi que celle d'une unité/bâtiment qu'il aura sélectionnée. Ainsi, on peut définir une scène qui sera découpée en plusieurs plans : un pour les unités mobiles, un pour les cases franchissables et obstacles et un dernier servant à afficher les infos utiles au joueur. Chaque plan possède donc les caractéristiques suivantes : la position de ses tuiles ainsi que leurs textures.

### 3.2 Conception logiciel

Dans le cadre de notre jeu, nous allons utiliser le diagramme des classes ci-après.

Pour afficher notre rendu, nous devons utiliser une classe *Tile* pour générer des tuiles pour ensuite les afficher. Cette classe peut être séparée en une classe mère : *StaticTile* la liste des tuiles de notre jeu.

Pour chaque animation plusieurs tiles seront affectées, toutes nos tiles auront les mêmes animations. Pour gérer l'ensemble nous utiliserons une classe *Layer* qui fournira les informations bas-niveau nécessaires pour la carte graphique, ces informations seront dans notre cas traitées par le biais de la librairie SfmL implanté dans *Surface*. Le traitement de ses informations pour l'affichage est fait par la classe *Surface* qui sera en charge du chargement et du placement des textures.

Tout changement de l'état entraînera une réaction de *StateObserver* qui engendrera une actualisation du rendu.

Le rendu sera composé de trois plans : une qui s'occupera de l'affichage des éléments immobiles, une pour les éléments mobiles et une autre pour l'interface.

### 3.3 Ressources

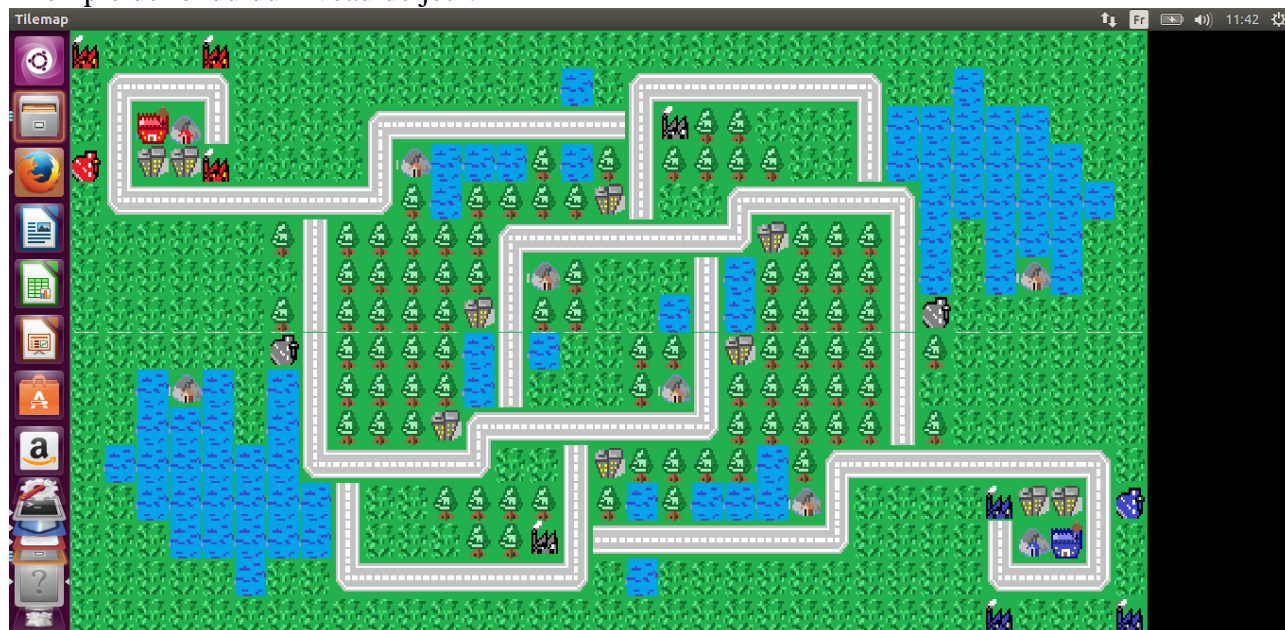
Voici les ressources utilisées pour notre jeu :





### 3.4 Exemple de rendu

Exemple de rendu du niveau de jeu :



```

classDiagram
    class ElementListLayer {
        +update(elementList:const state::ElementList&, i:int): void
        +stateChanged(e:const state::StateEvent&): void
    }
    class Layer {
        #tileset: const TileSet*
        -surface: Surface*
        +Layer()
        +-Layer()
        +getTileSet(): const TileSet* const
        +setTileSet(tileset:const TileSet*): void
        +setSurface(surface:Surface*): void
    }
    class Surface {
        +-Surface()
        +clear(): void
        +loadTexture(image_file:const char*): void
        +setSpriteLocation(i:int,x:int,y:int): void
        +setSpriteTexture(i:int,tex:const StaticTile*): void
    }
    class TileSet {
        +m_tileset: sf::Texture
        +-TileSet()
        +getCellWidth(): int const
        +getCellHeight(): int const
        +getImageFile(): const std::string& const
        +getElementTile(e:const state::Element*): const Tile* const
        +getCharTile(c:char): const Tile* const
    }
    class Tile {
        +-Tile()
        +isAnimated(): bool const
    }
    class StaticTile {
        #x: int
        #y: int
        #width: int
        #height: int
        +StaticTile(x:int=0,y:int=0,w:int=16,h:int=16)
        +isAnimated(): bool const
        +getX(): int const
        +getY(): int const
        +getWidth(): int const
        +getHeight(): int const
        +setX(x:int): int const
        +setY(y:int): int const
        +setWidth(w:int): int const
        +setHeight(h:int): int const
    }
    class Scene {
        #layers: std::vector<Layer*>
        #width: int
        #height: int
        +m_vertices: sf::VertexArray
        +Scene()
        +-Scene()
        +getWidth(): int const
        +getHeight(): int const
        +setLayer(index:int,layer:Layer*): void
        +getLayer(): int const
        +setSurface(index:int,surface:Surface*): void
        +registerObservers(s:const state::State*): void
        +unregisterObservers(s:const state::State*): void
        +StateChanged(e:const state::StateEvent&): void
    }
    class SceneLayer {
        <<enumeration>>
        +GRID_LAYER = 1
        +UNITS_LAYER = 2
        +INFO_LAYER = 3
    }
    ElementListLayer --> Layer
    Layer --> Surface
    Layer --> TileSet
    Layer --> Tile
    Layer --> StaticTile
    Scene --> Layer
    Scene --> TileSet
    Scene --> Tile
    Scene --> StaticTile
    SceneLayer --> Scene

```

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

Notre jeu a besoin d'un moteur de jeu afin de passer d'un état à un autre.

### 4.1 Changements extérieurs

Les changements extérieurs sont réalisés par des commandes extérieurs (appuis sur des touches par exemple).

Commandes principales : « LOAD » : on définit un état initial comme début de jeu

### 4.2 Changements autonomes

On parle de changement autonome lorsqu'un état se met à jour après l'effet de changements extérieurs. On a :

1. Si une unité se déplace sur une case sur laquelle :
  - Une unité ennemie est présente, celle-ci ira à une case adjacente la plus proche de l'ennemi (depuis sa position) et l'attaquera
  - SPACE est présent : celui-ci se placera sur la case si cela est possible
  - Un bâtiment allié sans unité est présent : celui-ci se place sur la case du bâtiment
  - Un bâtiment ennemi sans unité est présent : celui-ci se place sur la case du bâtiment et le capture
  - Un bâtiment ennemi avec unité est présent : celui-ci se place à une case adjacente la plus proche de lui et attaque
2. Si les points de vie d'une unité tombent à 0, celui-ci disparaît de l'écran.
3. Compteur d'unité : si une unité meurt, le compteur du joueur le possédant est décrémenté de 1. A l'inverse, celui-ci s'incrémente de 1 lorsqu'une unité est créée.
4. Compteur de ressources : à chaque début de leur tour, le joueur reçoit une quantité de ressources en fonction du nombre de mine(s) qu'il possède.
5. Lorsque le *Quartier Général* est capturé par une unité ennemi, le bâtiment change alors de couleur et le joueur ennemi remporte la partie.
6. La vitesse de déplacement de Infantry est diminuée lorsqu'il se trouve dans une case WOODS.
7. Lorsqu'une unité est créée, elle doit se trouver sur le bâtiment concerné. De plus, il faut que la ressource « Iron » soit mise à jour.

### 4.3 Conception Logiciel

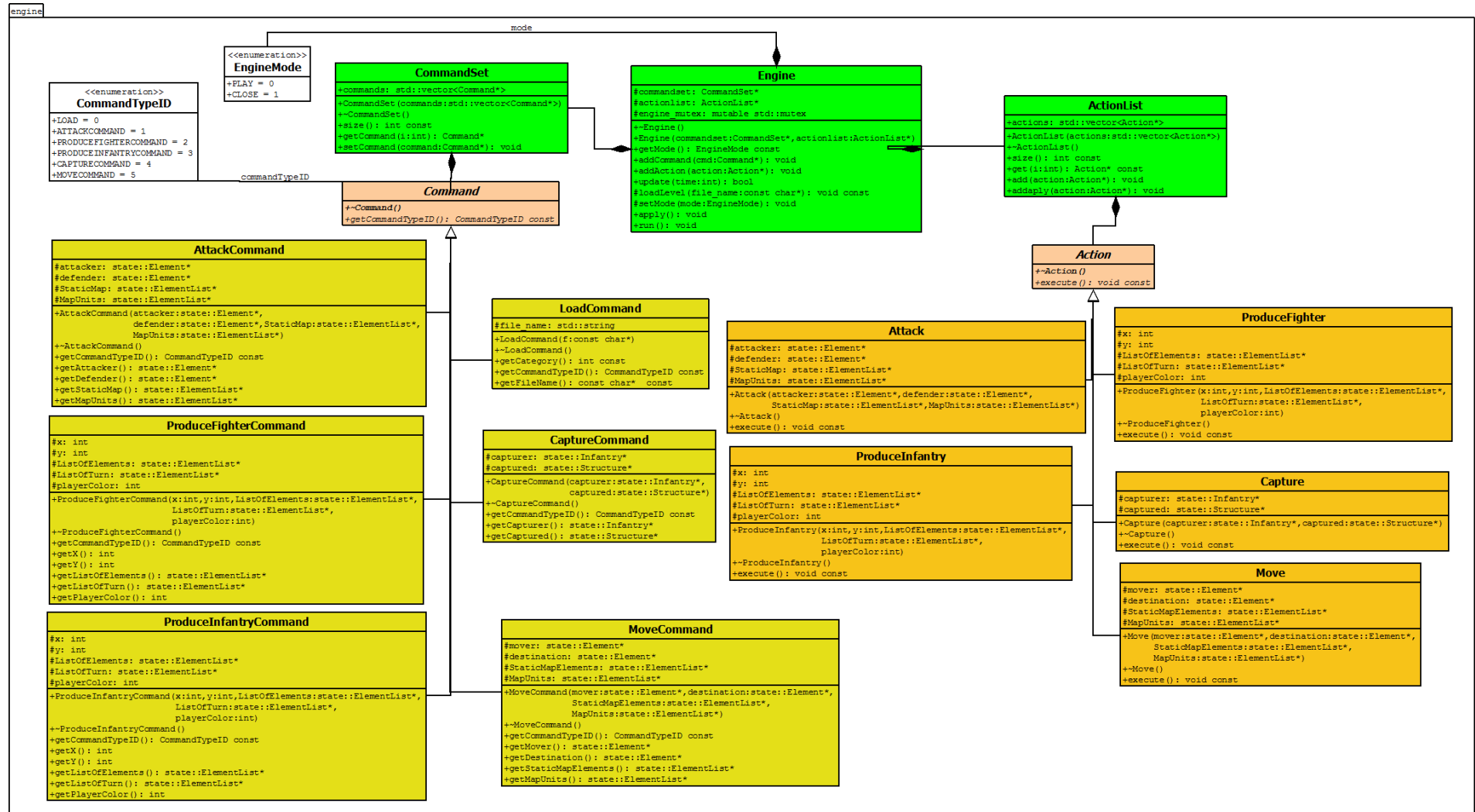
Le diagramme des classes que nous allons utiliser est présenté ci-après.

L'ensemble du moteur de jeu repose sur la classe Command principalement, c'est elle qui va permettre de gérer les commandes extérieures qui influent sur l'état du jeu.

Classe Action : la classe au joueur d'interagir avec l'état de son jeu concernant les actions qu'il pourra effectuer en jeu.

On ajoute également l'observer, dès qu'un événement est présent (sf::Event), un ordre est envoyé sur le rendu pour que celui-ci s'actualise.

Illustration 6 : Diagramme des classes du moteur



## 5 Intelligence artificielle

### 5.1 Stratégies

Afin de créer notre intelligence artificielle, nous allons procéder par plusieurs niveaux d'intelligence. D'un niveau simple et basique à une intelligence plus robuste et développée. Les différences entre celles-ci relèvent donc des réactions de l'IA en fonction de l'environnement. Son fonctionnement sera également illustré par un diagramme des classes (Illustration 7).

#### 5.1.1 Intelligence minimale

Dans un premier temps, l'IA implémentée sera simple et ne tiendra compte que de très peu de paramètres de l'environnement.

- ➔ L'IA avance jusqu'à rencontrer un obstacle puis se retourne et fait le chemin inverse à l'aide d'une unité.

#### 5.1.2 Intelligence intermédiaire

Pour composer notre IA de niveau intermédiaire, nous allons définir les propriétés suivantes :

- Durant chaque tour, l'IA crée le plus d'unités possibles, une par bâtiment de construction. L'IA prendra par défaut l'unité la plus couteuse à chaque fois.
- On fait ensuite en sorte que chaque unité créée attaque dès que possible une unité ennemie à portée (si plusieurs unités sont possibles, une est tirée au sort aléatoirement)
- Si aucune unité ennemie n'est à portée, alors les unités déjà créées doivent se déplacer vers une unité ennemie la plus proche.
- Si l'ennemi n'a aucune unité alors il s'approche à chaque tour un peu plus du QG ennemi pour aller le capturer. Le deuxième cas pour qu'une unité s'approche du QG ennemi est que celui-ci est à portée de mouvement de l'unité qui doit se déplacer.
- A chaque fois qu'un bâtiment neutre apparaît comme étant à portée d'une infantry alors celui-ci tente de le capturer si la case n'est pas déjà occupée par un allié. (De même s'il y en a plusieurs, l'unité capturera un bâtiment neutre de manière aléatoire)

L'algorithme qui permet de calculer le plus court chemin dans notre cas sera donc développer avec l'algorithme de Dijkstra. (On cherche à prendre le chemin le plus court pour s'approcher d'une unité ennemie ou du QG ennemi)

#### 5.1.3 Intelligence avancée

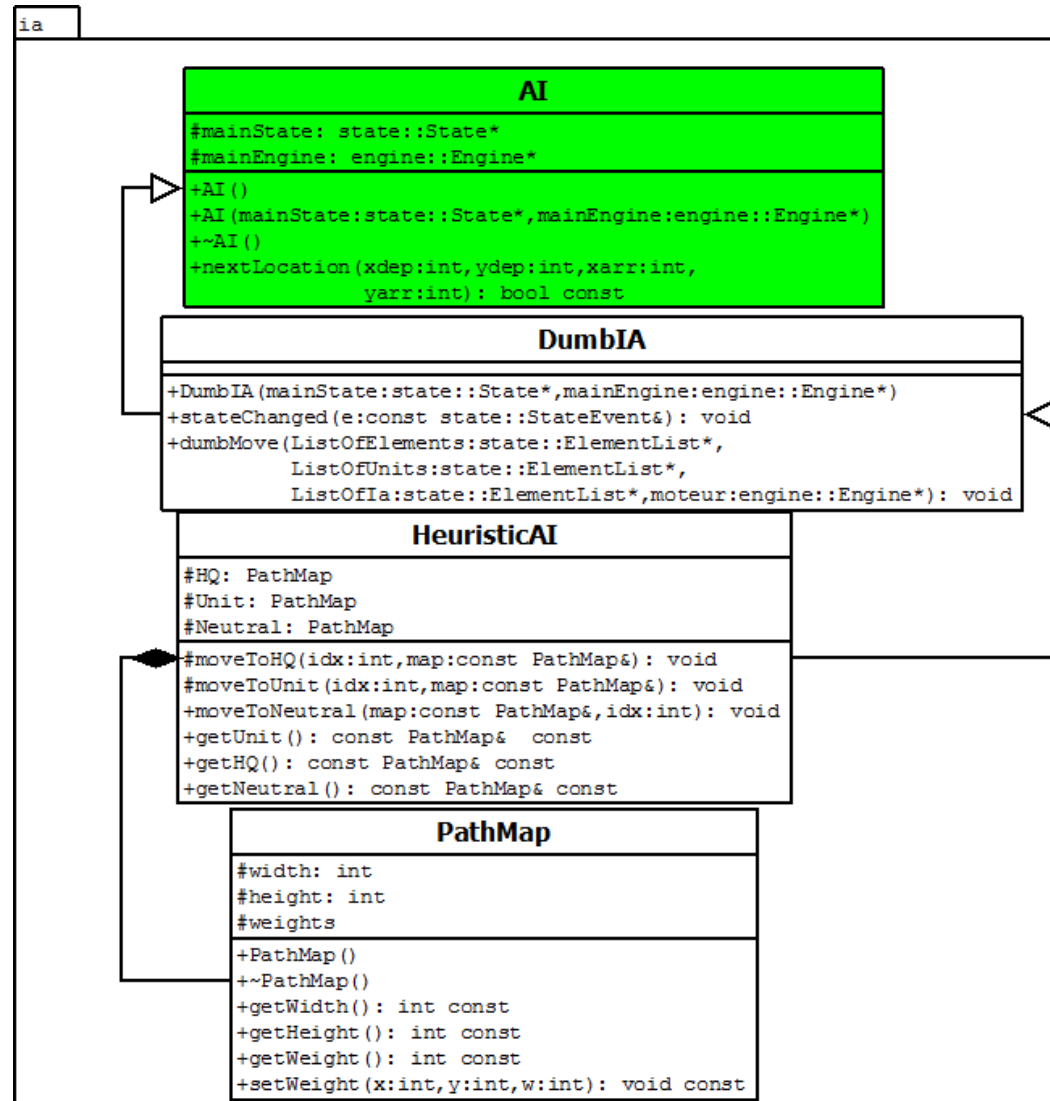
Afin de développer notre IA avancée, nous allons passer par un arbre de recherche. A chaque fois que l'on passera d'un niveau à un autre dans notre arbre, l'IA exécutera alors une série de commandes adaptées. Dans notre jeu, on a besoin que l'IA tues en priorité les unités avec le plus faible nombre de points de vies. Ainsi il doit être capable de choisir en priorité les unités les plus faibles qui sont à portée.

L'algorithme qui est utilisé dans cette partie est celui de Min-Max. A chaque fois que le premier joueur (supposons Max) jouera, il cherche à maximiser son coup. Lorsque l'IA cherchera ensuite son meilleur coup, il cherchera également à maximiser son coup mais en minimisant celui de son adversaire.

## 5.2 Conception Logiciel

Pour pouvoir gérer tous nos niveaux de difficultés pour l'IA, nous allons utiliser une classe *PathMap*. On va donner un poids pour chaque case de notre jeu. Ainsi selon la valeur du poids une unité effectuera par exemple une action d'attaque d'une unité ennemie à portée et non pas un déplacement vers le QG ennemi s'il est trop éloigné.

Illustration 7 : Diagramme des classes de l'IA





## 6 Modularisations

### 6.1 Organisation des modules

Afin de mieux travailler sur notre moteur de jeu et notre rendu, nous allons utiliser la notion de thread dans notre jeu.

#### 6.1.1 Répartition sur différents threads

Notre objectif est de placer le moteur de jeu sur un thread. Tandis que le deuxième thread sera lui sur le module de rendu. Les informations à considérer sont donc les commandes et les notifications de rendu. En organisant en plusieurs threads, on optimise ainsi la manière de fonctionner notre jeu en parallélisant au maximum les calculs nécessaires pour l'ordinateur.

Commandes : celles-ci sont générées à travers les **clics de la souris** effectués et les **saisies au clavier** par un joueur. On ajoute alors l'ordre de la commande saisie dans une première FIFO (*commands*). Lorsque cette FIFO possède 1 élément exactement, on transcrit alors l'action correspondante dans une deuxième FIFO (*actionslist*). Ainsi, on se met à vider la première FIFO pour avoir la seconde avec 1 seule action. Au final, on exécute alors l'action, une fois celle-ci achevée, on peut vider la seconde FIFO.