

# Projet Logiciel Transversal

Marlyatou DIALLO – Tian HAOMING



Figure 1 civilisation : conquête et pillage

## Table des matières

1	Objectif Civilization : Conquête et pillage.....	3
1.1	Présentation générale :.....	3
1.2	Règles du jeu .....	3
1.3	Conception Logiciel .....	4
2	Description et conception des états .....	5
2.1	Description des états .....	5
2.1.1	Etat éléments fixes .....	5
2.1.2	Etat éléments mobiles .....	5
2.1.3	Etat général .....	6
2.2	Conception logiciel.....	6
2.3	Conception logiciel : extension pour le rendu.....	6
2.4	Conception logiciel : extension pour le moteur de jeu .....	6
2.5	Ressources .....	6
3	Rendu : Stratégie et Conception .....	9
3.1	Stratégie de rendu d'un état.....	9
3.2	Conception logiciel.....	9
3.3	Conception logiciel : extension pour les animations .....	10
3.4	Ressources .....	10
3.5	Exemple de rendu .....	10
4	Règles de changement d'états et moteur de jeu .....	12
4.1	Horloge globale .....	12
4.2	Changements extérieurs.....	12
4.3	Changements autonomes .....	12
4.4	Conception logiciel.....	13
4.5	Conception logiciel : extension pour l'IA .....	14
4.6	Conception logiciel : extension pour la parallélisation .....	14
5	Intelligence Artificielle.....	16
5.1	Stratégies.....	16
5.1.1	Intelligence minimale.....	16
5.1.2	Intelligence basée sur des heuristiques .....	16
5.1.3	Intelligence basée sur les arbres de recherche .....	17
5.2	Conception logiciel.....	17
5.3	Conception logiciel : extension pour l'IA composée.....	18
5.4	Conception logiciel : extension pour IA avancée.....	18
5.5	Conception logiciel : extension pour la parallélisation .....	18
6	Modularisation.....	19
6.1	Organisation des modules.....	19
6.1.1	Répartition sur différents threads.....	19
6.1.2	Répartition sur différentes machines .....	19
6.2	Conception logiciel.....	19
6.3	Conception logiciel : extension réseau .....	19
6.4	Conception logiciel : client Android .....	19

# 1 Objectif Civilization : Conquête et pillage

## 1.1 Présentation générale :

Il s'agit d'un jeu au tour par tour destiné à créer sa propre armée de régime, à développer et à occuper des terres. Chaque joueur doit explorer des terres ou piller le château ennemi pour gagner des pièces d'or qui peuvent être utilisées pour acheter différents types d'unités de combat ou construire de châteaux.

A la fin du nombre de tours fixés, le vainqueur sera celui qui a le grand nombre de châteaux laissés et de pièces gagnés.

## 1.2 Règles du jeu

### -Unité de Combat :

Le jeu comprend cinq unités: **infanterie, ingénieurs, archers, cavalerie et lanciers**. Les ingénieurs sont réprimés par l'infanterie et la cavalerie peut réprimer l'infanterie, tandis que la cavalerie peut être réprimée par les archers et les lances. Le lancier et l'archer sont mutuellement répressifs. Toutes les unités auront un bonus d'attaque lorsqu'elles attaqueront des unités qu'elles suppriment.

Au début de la partie, chaque joueur ne possède qu'une seule équipe d'ingénieurs.

### -Occupation château et gain d'or :

Chaque équipe d'ingénieurs explore les terres dont certaines contiennent des pièces d'or. Ces pièces d'or explorées par les ingénieurs seront comptées dans le montant total des pièces. Au début de la partie chaque joueur possède un château initial qui rapporte à chaque tour des pièces tant qu'il n'est pas détruit par l'ennemi. Les pièces d'or pourront servir à la construction de châteaux ou à l'achat d'une unité de combat.

Lors de l'attaque d'un château, on peut simplement l'occuper (avoir des pièces à chaque tour et veiller à sa défense) ou le détruire (obtenir les pièces qui valent ce château).

### - Ce que le joueur peut faire à chaque tour :

Les joueurs peuvent effectuer quatre actions par tour: attaque, défense, déplacement et construction. Chaque unité de combat ne peut attaquer que deux fois par tour. Il faudra quatre tours pour construire un nouveau château. Le joueur a le droit de passer ce tour.

### -Mouvement et vision

Le jeu sera joué sur une carte en forme de damier, chaque unité se déplaçant d'un certain nombre de carrés par tour, le mouvement de l'unité à chaque tour consomme une certaine quantité de pièces d'or.

Au début de l'ouverture, la majeure partie de la carte est une zone inconnue et chaque unité peut déverrouiller les informations de la carte dans le champ de vision.

	<b>infanterie</b>	<b>ingénieurs,</b>	<b>archers</b>	<b>cavalerie</b>	<b>lanciers</b>
<b>Distance de vision</b>	<b>4</b>	<b>8</b>	<b>6</b>	<b>4</b>	<b>4</b>
<b>Distance de mouvement</b>	<b>2</b>	<b>3</b>	<b>2</b>	<b>6</b>	<b>3</b>

### 1.3 Conception Logiciel

Système utilisé : machine virtuelle (Virtual box sous Ubuntu)

Environnement de développement utilisé : Eclipse ou Geany

Compilateurs C++ : g++

CMake : cmake

SFML : package libsFML-dev

Logiciel Tiled Map Editor

## 2 Description et conception des états

### 2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (les terrains, les pièces d'or générées sur les terrains) et un ensemble d'éléments mobiles (les pions). Tous les éléments possèdent des propriétés telles que :

- Une Coordonnées (x,y) sur la carte.
- Un code tuile
- un nom
- une puissance d'attaque et de défense pour certains et aussi un niveau de vie

#### 2.1.1 Etat éléments fixes

La carte de jeu est formée par une grille d'éléments de type **Field**, des éléments **GoldBonus** qui sont générés de façon aléatoires sur la grille et aussi des éléments **Castles** qui seront construits au fur et à mesure. Nous allons choisir une grille de 25 cases par 25 cases.

Les différents types de terrains (Field) sont :

-**Terrains non praticables** que nous avons nommé « NPraticable » : ils sont inaccessibles par les éléments mobiles et n'ont aucune influence sur eux. Ce sont :

- Les arbres « Tree »
- Les murs « Wall »
- L'eau « Water »

-**Terrains Praticables** que nous avons nommés « Praticable » : ceux-ci sont accessibles par les éléments mobiles et peuvent avoir de l'influence sur les statistiques du joueur. Ce sont :

- Terre Ferme « Land »: il peut y avoir de l'or caché sur une terre ferme et on peut également construire des châteaux sur elles.
- Les collines « Hill » : on peut construire des châteaux sur une colline mais il ne peut y avoir de l'or cachés sur une colline.

#### 2.1.2 Etat éléments mobiles

Ces éléments sont dirigés par les joueurs que nous avons défini par une classe **Player**. Chaque Joueur a un nom, manipulera des pions et châteaux et peut gagner des espèces d'or en bonus.

Chaque pion que nous avons nommé **UnityArmy** possède un niveau de vie (currentlife), un camp (camp), un champ d'attaque (attack\_field), un champ de mouvement (move\_field), un statut (status), un identifiant (type\_id).

Les statut represente l'etat d'un pion à un instant donné :

- Selectionné « SELECTED » : dans le cas ou le pion est selectionné pour faire une action
- Disponible « AVAILABLE » : dans le cas ou il peut etre selectionné mais n'a pas encore agi.
- Attente « WAITING » : dans le cas ou le pion ne peut etre selectionné car c'est le tour d'un autre joueur ou le pion a effectué toutes les actions qu'il peut faire en un tour.
- Mort « DEAD » : dans le cas ou son niveau de vie est à 0.

**Les Pions** sont de plusieurs catégories et sont identifiés par un type\_id :

- Cavalerie « CAVLRY »
- Archer « ARCHER »
- Ingénieur « ENGINEER »
- Lancer « LANCER »

### 2.1.3 Etat général

Sur l'ensemble des éléments mobiles et statiques, nous avons rajouté :

- Nombre de tours
- Numéro de tours
- Maximum de tours
- Fin : pour indiquer la fin de la partie.

## 2.2 Conception logiciel

Le diagramme d'état est constitué de plusieurs classes dont :

-Classe Eléments : toutes les classes filles(en bleu marine) issue de la classe mère **Elements** représentent les différentes catégories des éléments mobiles et statiques.

-Classe Conteneur d'éléments : il y'a les classe **State** et **Map2D qui contiennent l'ensemble des éléments**. Map2D est un tableau à deux dimensions d'éléments, par exemple pour contenir la grille des éléments du niveau.

Enfin, la classe State est le conteneur principal, à partir duquel on peut accéder à toutes les données de l'état et contient aussi la liste des joueurs.

-Classe Position : classe qui décrit la position de chaque élément sur la grille.

-Classe Correspondence : classe qui regroupe des sets de double composé d'un code tuile et d'un ID d'éléments.

-Classe **Observable** : la conception de ses classes suit un design pattern Observer. la classe **Observable** dont hérite **State** enregistre des observateurs et les notifie à chaque changement d'état.

**Classe Correspondances** : Classe qui regroupe des sets de doublets composés d'un code de tuile et d'un ID d'élément

### 2.3 Conception logiciel : extension pour le rendu

Les Classes State, Observable et Position seront des classes qui seront incluses dans le rendu et l'extension est render.dia

### 2.4 Conception logiciel : extension pour le moteur de jeu

Engine.dia

### 2.5 Ressources



Figure 2: tuile UnityArmy

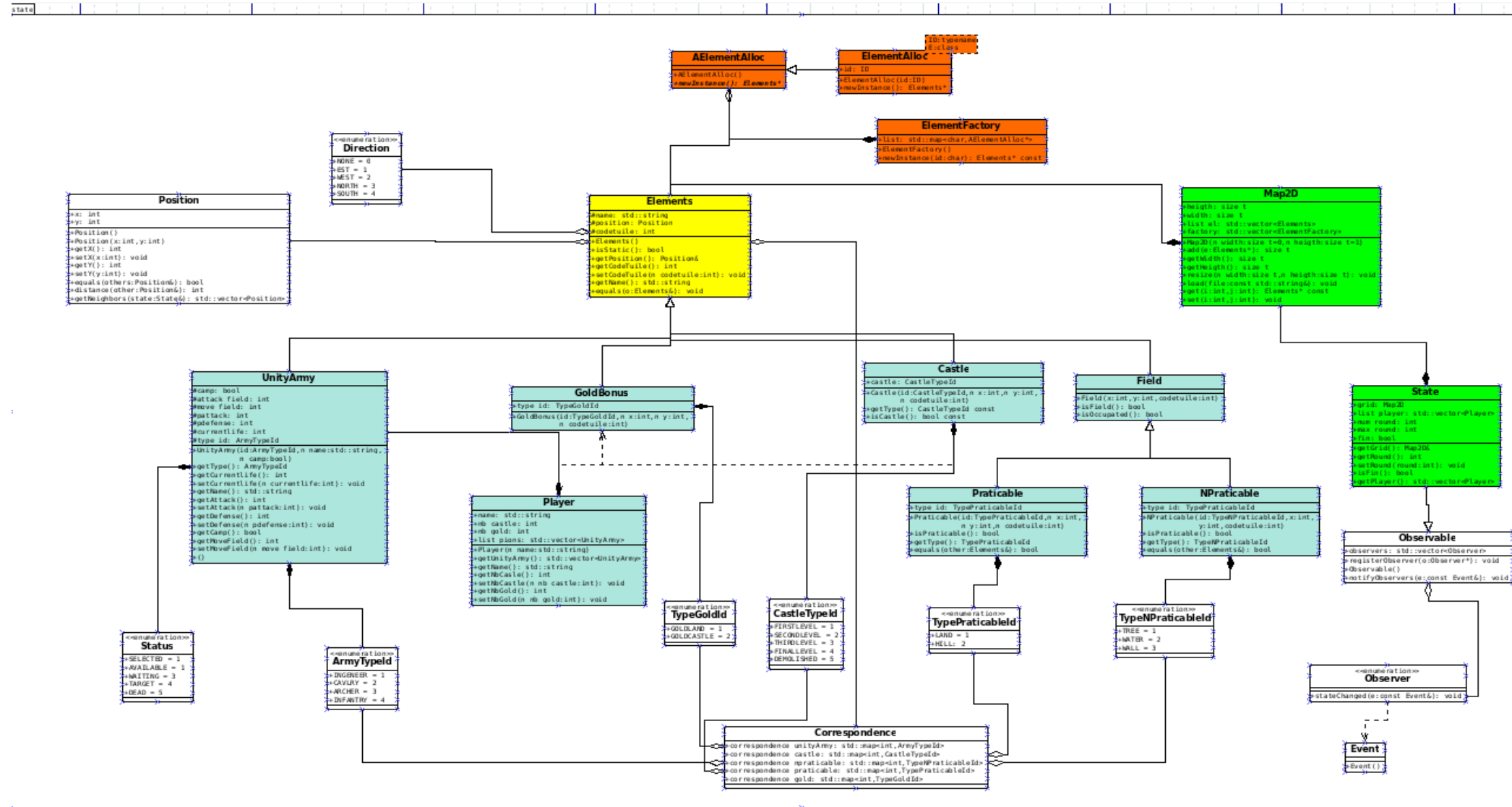


Figure 3:tuile niveau château



Figure 4:tuile Tree

Illustration 1: Diagramme des classes d'état





### 3 Rendu : Stratégie et Conception

NB : nous avons apporté beaucoup de modification sur le diagramme d'état et par conséquent des modifications ont été apporté sur ce qui a été rédigé précédemment.

#### 3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous avons fait le choix d'un rendu par tuile à l'aide de la bibliothèque SFML.

Nous divisons la scène à afficher en trois surfaces : une surface contenant les éléments statiques Field, une surface pour les éléments mobiles UnityArmy et une surface pour les informations de jeu (statistiques et nom d'un personnage, tour de jeu...).

La grille de Field est créée à partir du fichier texte « map.txt » composé de 25x25 codes de tuiles. La classe Correspondences (dans le package state) possède un tableau de correspondance pour les Praticables Field et un autre pour les NonPraticables Field. Ces std::map associent à chaque code de tuile un ID de terrain spécifique. Par exemple, la première tuile du tileset pour la grille porte le numéro 0 et représente un ensemble d'arbre, et cette association est donc répertoriée dans l'attribut correspondance\_P de la classe Correspondences. Pour modifier la grille il suffit donc de modifier les entiers dans le fichier « map.txt » (qui se trouve dans res/). Pour le moment nous avons juste utilisé un tableau contenant des codes tuiles et déclaré directement dans le main pour aller plus vite.

La méthode initGrille de la classe State (package state) nous permet de fabriquer pour chaque code tuile du fichier « map.txt » le terrain correspondant, de créer un pointeur unique vers cet objet et de l'ajouter à la grille (tableau à deux dimensions contenant des pointeurs de Field).

Les unityarmy eux ne sont pas créés à partir d'un fichier, pour l'instant ils sont créés par la méthode initUnityArmy de la classe State.

#### 3.2 Conception logiciel

Le diagramme des classes pour le rendu général est présenté sur la figure ci dessous.

**Classe Surface :** cette classe possède deux attributs : une texture et un tableau de Vertex (quads) contenant la position des éléments et leurs coordonnées dans la texture. Elle possède les méthodes loadGrille, loadPersonnage et loadCurseur lui permettant d'initialiser ses attributs à partir d'un tableau de Field, d'une liste de UnityArmy et d'un fichier « spritee.png », « unityArmy.png ». La méthode Draw a pour but de dessiner une texture pour ensuite permettre son affichage dans une fenêtre.

**Classe TileSet :** Cette classe possède plusieurs attributs : un id de type TileSetID, des entiers cellWidth et cellHeight (qui représentent respectivement la largeur et la longueur en pixel d'une tuile) et une chaîne de caractères imageFile (chemin vers un « fichier.png »). L'ID peut prendre différentes valeurs :

- GRIDTILESET
- UNITYARMYTILESET
- INFOTILESET

Suivant l'ID passé en argument du constructeur de TileSet, les attributs cellWidth, cellHeight et imageFile sont initialisés différemment. De plus, les textures sont chargées depuis le fichier correspondant lors de l'appel au constructeur, et donc une seule fois par TileSet. Les méthodes getCellWidth, getCellHeight et getImageFile permettent de récupérer ces attributs.

**Classe StateLayer :** Cette classe possède un attribut state qui est une référence à une instance de State. Elle possède également un tableau de pointeurs de TileSet et un autre de pointeurs de Surface. Le but de cette classe est de créer trois surfaces grâce à la méthode initSurface et d'initialiser leur texture. Cette classe est un observateur, elle implémente l'interface Observer pour

être avertie des changements d'état. Elle réagit ensuite en actualisant les textures.  
Projet

### 3.3 Conception logiciel : extension pour les animations

Le diagramme pour les animations sera engine.dia

### 3.4 Ressources

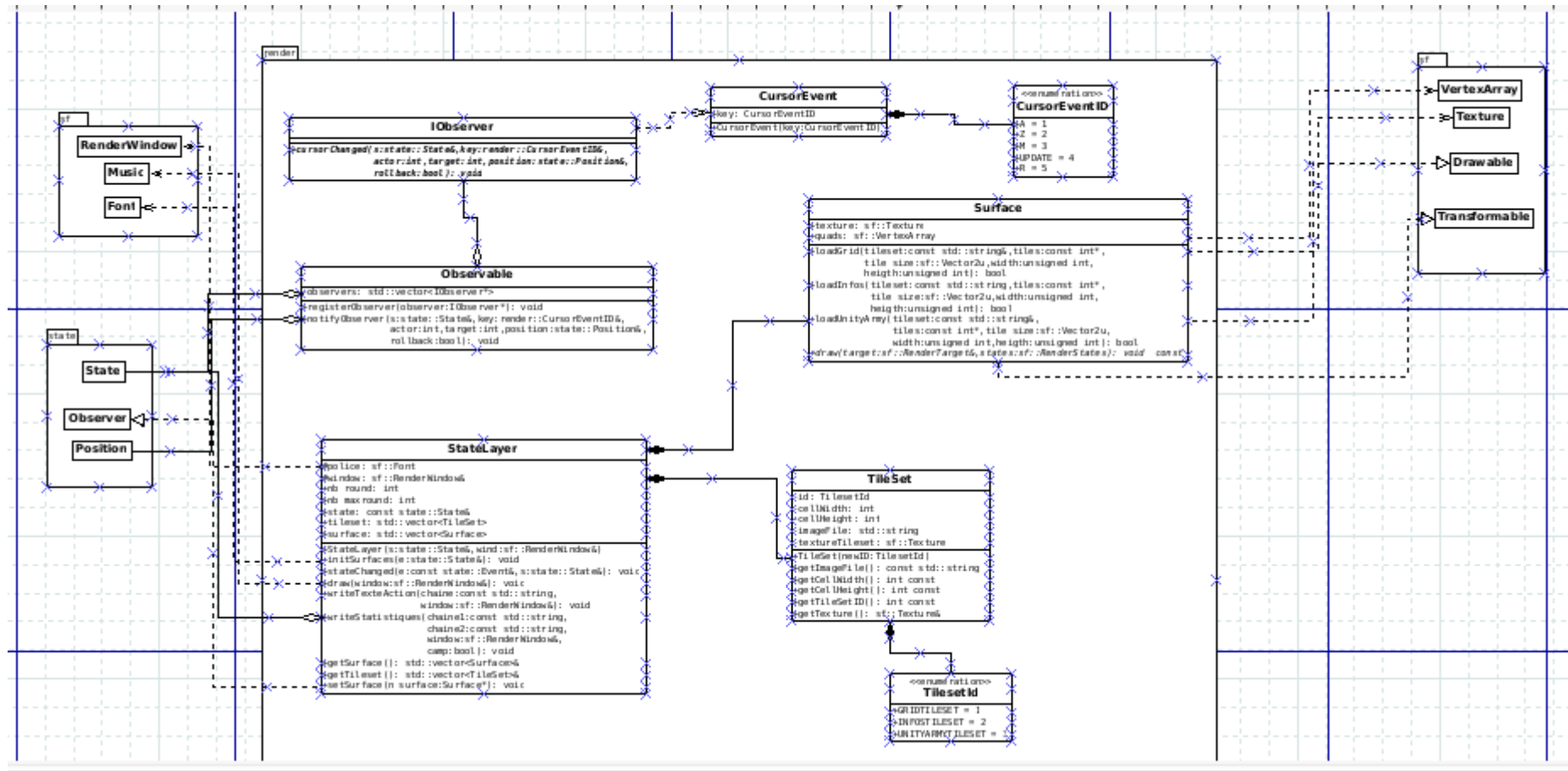


### 3.5 Exemple de rendu



Nous avons encore des choses à rajouté sur ce rendu mais etant pris par le temps nous allons devoir fournir en attendant cela.

*Illustration 2: Diagramme de classes pour le rendu*



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

### 4.1 Horloge globale

Les changements d'état suivent une horloge globale : de manière régulière, on passe directement d'un état à un autre. Il n'y a pas de notion d'état intermédiaire. Ces changements sont calibrés sur le temps qu'il faut pour un élément mobile à vitesse maximale pour passer d'une case à une autre. En conséquence, tous les mouvements auront une vitesse fonction de cet élément temporel unitaire. Notons bien que cela est décorrélé de la vitesse d'affichage : l'utilisateur doit avoir l'impression que tout est continu, bien que dans les faits les choses évoluent de manière discrète. Par exemple, s'il y a 60 images par seconde, et que le temps unitaire est de 200ms, alors pendant un temps unitaire, 12 étapes d'animation seront affichées pour les éléments en mouvement.

### 4.2 Changements extérieurs

Chaque action effectuée modifie l'état de jeu

- . Le choix de l'unityarmy sélectionné et des actions effectuées est défini par des commandes. Les commandes du joueur sont provoquées par une pression sur une touche.
- . Déplacer le curseur : flèches « haut/bas/gauche/droite ».
- . Sélectionner une unity : « Enter » (lorsque le curseur encadre le personnage).
- . Déplacer une unity : flèche « haut/bas/droite/gauche » (l'unityarmy doit avoir été sélectionné au préalable).
- . Attaquer : « A » puis déplacer le curseur avec les flèches sur la case d'un adversaire puis « Enter » pour confirmer l'attaque (le personnage doit avoir été sélectionné au préalable).
- . Annuler une attaque : Appuyer sur « N ».
- . Terminer le tour d'actions d'un personnage : « Z » (le personnage doit avoir été sélectionné au préalable).

### 4.3 Changements autonomes

En début de tour, toutes les unityArmy de type ENGINEER d'un joueur possèdent le statut DISPONIBLE. Ils peuvent donc tous être SÉLECTIONNÉS. Lorsqu'une unityArmy est SÉLECTIONNÉE par un joueur il peut effectuer selon les cas au moins une des 3 actions listées ci-dessous :

- Se déplacer
- Attaquer
- Construire
- Terminer ses actions (son tour)

Une unityarmy ne peut se déplacer que d'une case par une case (sur des terrains praticables et inoccupés) . Il possède des points de mouvements qui correspondent au nombre de déplacement maximum qu'il lui est possible de faire. Lorsque tous ses points de mouvement ont été utilisés, il lui est impossible de continuer son déplacement.

Une unityarmy ne peut attaquer une autre que lorsque celle-ci se trouve dans son champ d'attaque et appartient au camp adverse (les attaques alliées ne sont pas autorisées).

Lorsqu'une unityarmy termine ses actions, son statut devient WAITING. Cela signifie qu'il ne pourra plus être SÉLECTIONNÉ (et ne pourra donc plus effectuer d'actions) avant le prochain tour du joueur.

L'action « Terminer son tour d'action » doit obligatoirement être effectuée par chaque unityarmy encore actif à chaque tour du joueur (même lors du dernier tour lorsque tous les adversaires ennemis sont morts). Il existe 4 enchaînements d'actions possibles :

- Attaquer directement (lorsque cela est possible) ce qui termine automatiquement le tour d'un personnage
- Effectuer un ou plusieurs déplacements , attaquer puis terminer son tour automatiquement
- Effectuer un ou plusieurs déplacements, construire puis terminer son tour automatiquement
- Effectuer un ou plusieurs déplacements puis terminer son tour manuellement
- Terminer son tour directement sans avoir effectué aucune autre action

Le tour de jeu d'un joueur est terminé lorsque tous ses personnages sont en WAITING.

C'est alors le tour du joueur adverse.

Lorsqu'une unityarmy est attaquée par un ennemi, il tente toujours une contre-attaque (à moins d'avoir été achevé durant la première attaque).

Si une unityarmy perd tous ses points de vie, son statut évolue et prend la valeur DEAD.

Si toutes les unityArmy d'un joueur meurent avant qu'on n'atteigne le nombre de tour maximum fixés. et qu'il ne possède ni de châteaux, ni d'espèces d'or pour acheter des unityarmy la partie est terminée à la fin du tour adverse.

Si les unityArmy des deux joueurs sont encore vivants à la fin de la partie, le joueur qui gagne est celui qui a construit le plus de châteaux et qui a une quantité plus grande de pièces d'or.

## 4.4 Conception logiciel

Le diagramme des classes pour le moteur de jeu (« engine.dia ») est présenté sur la figure ci-dessous.

Le moteur de jeu repose sur le Design Pattern Order.

**Classe Engine** : C'est le cœur du moteur. Elle permet de stocker les commandes dans une std::map avec clef entière (avec « addOrder »). Ce mode de stockage permet d'introduire une notion de priorité : on traite les commandes dans l'ordre de leur clef (de la plus petite à la plus grande).

Lorsque la méthode « update » est appelée, le moteur appelle la méthode « execute » de chaque commande puis supprime toutes les commandes une fois exécutées. La méthode

« verificationStartTurn » change le joueur actif, réinitialise les points de mouvement de tous les unityarmy du joueur et procède à la récupération de la quantité d'or des unityarmy ayant construit des châteaux. La méthode « verificationEndTurn » incrémente le nombre de tours si tous les unityarmy restants du joueur actif sont en WAITING et déclare la partie terminée si tous les personnages du joueur adverse sont MORTS avant que le nombre de tours fixés soit atteint.

La classe Engine a été étendue pour permettre une exécution dans un thread séparé.

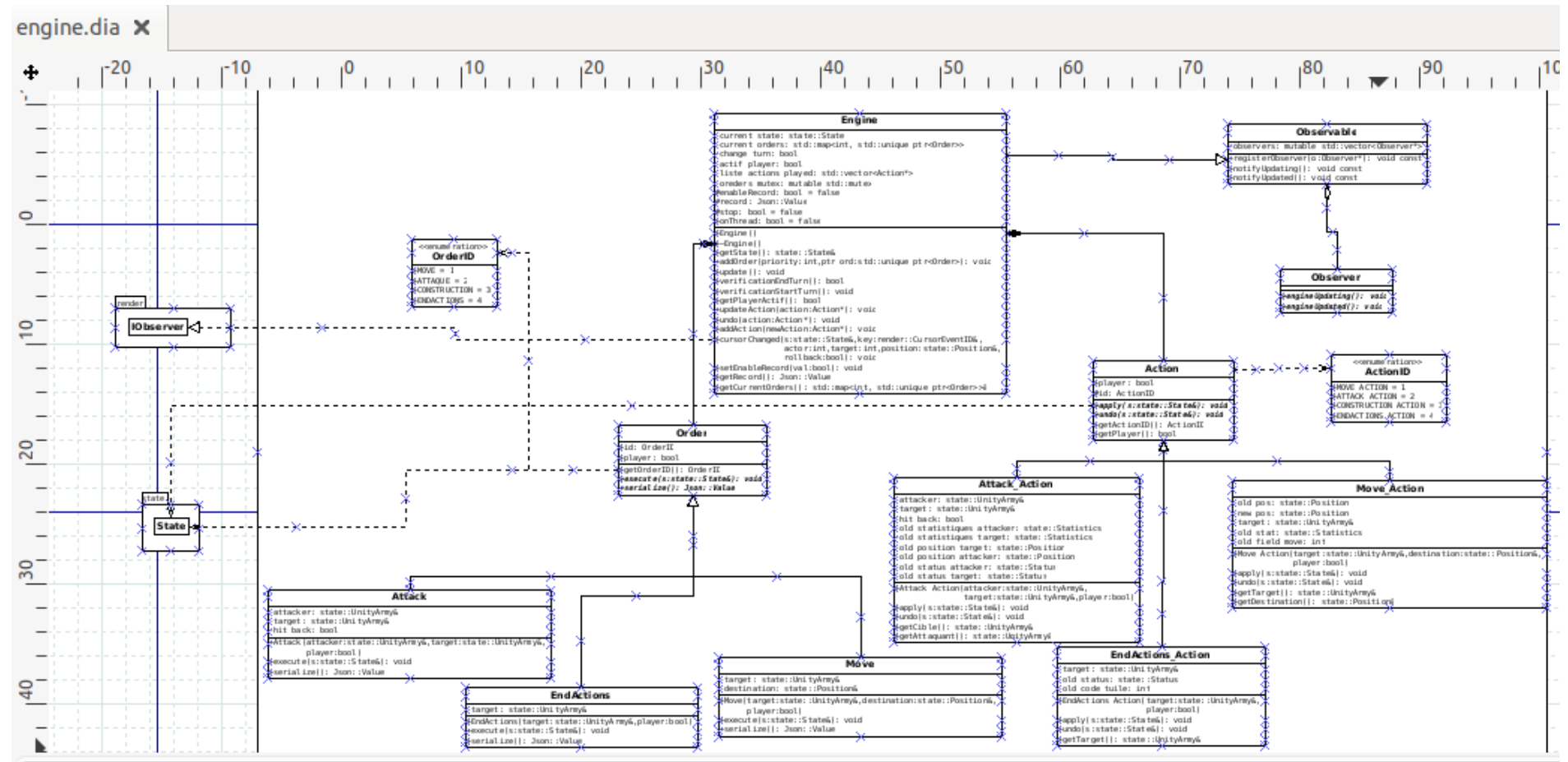
**Classes Commandes** : Les classes Attack, Move et EndAction, Construction héritant de la classe Order, possèdent chacune une méthode « execute » qui fait effectuer à un personnage l'action correspondante.

**Classes Actions** : Les classes Attack\_Action, Move\_Action et EndActions\_Action, ConstructionAction\_Actionn héritant de la classe Action, possèdent chacune une méthode « apply » qui change l'état de la même manière que le ferait une Commande, mais possèdent également une méthode « undo » qui remodifie l'état mais en annulant les changements opérés par l'action en question, grâce aux attributs de chaque Action qui correspondent aux changements que provoque l'action.

#### **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

### 5.1 Stratégies

#### 5.1.1 Intelligence minimale

L' IA contrôle une liste de unityarmy qu'elle va sélectionner les uns après les autres lors de son tour de jeu. Une fois une unityarmy sélectionné, un entier correspondant à une action précise est choisi au hasard parmi 4 : 0 (se déplacer), 1 (attaquer), 2 (Construire un château), 3 (terminer son tour d'actions).

Si « se déplacer » est choisi et que l'unityarmy sélectionné possède encore des points de mouvement, une destination va être choisie aléatoirement parmi une liste de destinations possibles.

Si la liste n'est pas vide, l'unityarmy se déplacera. Sinon il ne pourra pas se déplacer. Dans les deux cas, il restera SÉLECTED et l'IA pourra donc continuer à le manipuler (une nouvelle action aléatoire sera tentée après celle-ci).

Si « attaquer » est choisi, une cible va être déterminée aléatoirement parmi une liste de cibles possibles (qui diffèrent en fonction de la position du personnage et de son champ d'attaque). Si la liste n'est pas vide, l'unityarmy va attaquer puis terminer directement son tour, comme défini dans les règles du jeu. Il passera donc en WAITING ou DEAD (s'il est décédé des suites de son attaque), l'IA ne pourra plus le manipuler et sélectionnera le l'unityarmy suivant. Si la liste d'attaque est vide, l'attaque n'aura pas lieu, l'unityarmy restera SELECTED et l'IA pourra donc continuer à le manipuler (une nouvelle action aléatoire sera tentée après celle-ci).

Si « Construire » est choisi, l'unityarmy qui est capable de construire qui est l'ingénieur possède des points de mouvement pour se déplacer et construire, un terrain vide sera aléatoirement choisi parmi les terrains vides.

Si « terminer son tour d'actions » est choisi, le personnage va terminer son tour d'actions

Directement. Il passera donc en WAITING. L'IA ne pourra plus manipuler ce personnage et sélectionnera l'unity army suivant.

#### 5.1.2 Intelligence basée sur des heuristiques

Nous proposons ensuite un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard et donner une chance à l'IA de remporter la partie.

On privilégie presque toujours l'attaque au déplacement et le déplacement à la fin du tour d'actions.

Une fois une des unités sélectionné, on calcule un score pour chacun des personnages adverses à partir des statistiques du personnage sélectionné (son attaque, son taux de coup critique, nombres de pièces gagnés, châteaux construits), de celles de l'adversaire (sa défense, son taux d'esquive) ainsi que de la distance entre ces deux personnages.

Ainsi, plus une attaque peut être efficace contre un des personnages adverse plus le score sera élevé. Le personnage sélectionné prend donc ensuite pour cible l'adversaire pour lequel ce score est le plus élevé pour ce tour de jeu.

Si le personnage sélectionné a la possibilité d'attaquer sa cible c'est-à-dire si le personnage adverse ciblé se trouve dans son champ d'attaque, il le fera en priorité. Si plusieurs personnages adverses se trouvent dans son champ d'attaque il attaquera celui contre lequel il a le plus grand score calculé.

Si le personnage peut attaquer mais que son nombre de PV est trop faible (inférieur ou égal à 5 PV), il va tenter de se déplacer vers le refuge le plus proche (MAISON ou CHATEAUX) pour récupérer des PV. S'il n'y a aucun refuge sur la map ou si tous sont indisponibles, le personnage prendra le risque d'attaquer ou terminera son tour.

Si le personnage ne peut pas attaquer immédiatement sa cible, il va tenter de s'en approcher avec des déplacements définis après un calcul de chemin réalisé grâce à un algorithme A\*

(Algorithme de recherche d'un chemin de coût minimal). Si ces déplacements ne sont pas possibles,



Il attaquera les autres ennemis à sa portée qui le bloquent.

Si le personnage ne peut ni attaquer ni se déplacer vers son objectif (s'il est bloqué par des alliés ou des murs par exemple), il va rechercher un nouvel objectif et refaire tous les tests précédents pour s'en approcher.

Si aucun des objectifs n'est atteignable, il terminera son tour.

### 5.1.3 Intelligence basée sur les arbres de recherche

Nous proposons une intelligence plus avancée en suivant les méthodes de résolution de problèmes à états finis. On se base sur un algorithme minimax pour que l'IA détermine quels sont les choix de commandes optimaux en prenant en compte ce que va faire le joueur au prochain tour. La stratégie est la suivante : on va simuler toutes les suites de coups possibles de l'IA, puis pour chacune de ces suites d'actions, toutes les suites de coups possibles du joueur. Une fois la simulation du tour des deux adversaires terminée, on appelle une fonction d'évaluation qui attribue un score à cette suite d'actions en fonction de l'état du jeu après simulation. Plus ce score est élevé, plus le scénario simulé (suite de coups de l'IA puis réponse du joueur) est intéressant à jouer pour l'IA.

On simule alors un graphe virtuel où chaque noeud correspond à une action simulée et dont les noeuds suivants sont les actions possibles après cette action. Une branche est donc une succession d'actions de l'IA puis du joueur, soit la simulation d'un tour complet de l'IA puis d'un tour complet du joueur.

A chaque arrivée en fin de branche, on compare son score à celui qui est pour l'instant considéré comme optimal et si le nouveau score est meilleur, on garde en mémoire ce nouveau score et la suite d'actions qui a conduit à cet état optimal pour l'IA.

Pour passer d'une branche à une autre, une fois qu'une branche est simulée en entier et que son score est calculé, on annule la dernière action et on revient donc à l'état précédent, un noeud plus haut dans le graphe, de manière récursive.

En effet, lors de ces simulations, ce sont des Actions qui sont effectuées et non des

Commandes. Les Actions ont la particularité de pouvoir être annulées car chaque action garde en mémoire ce qu'elle a modifié dans l'état.

## 5.2 Conception logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 8.

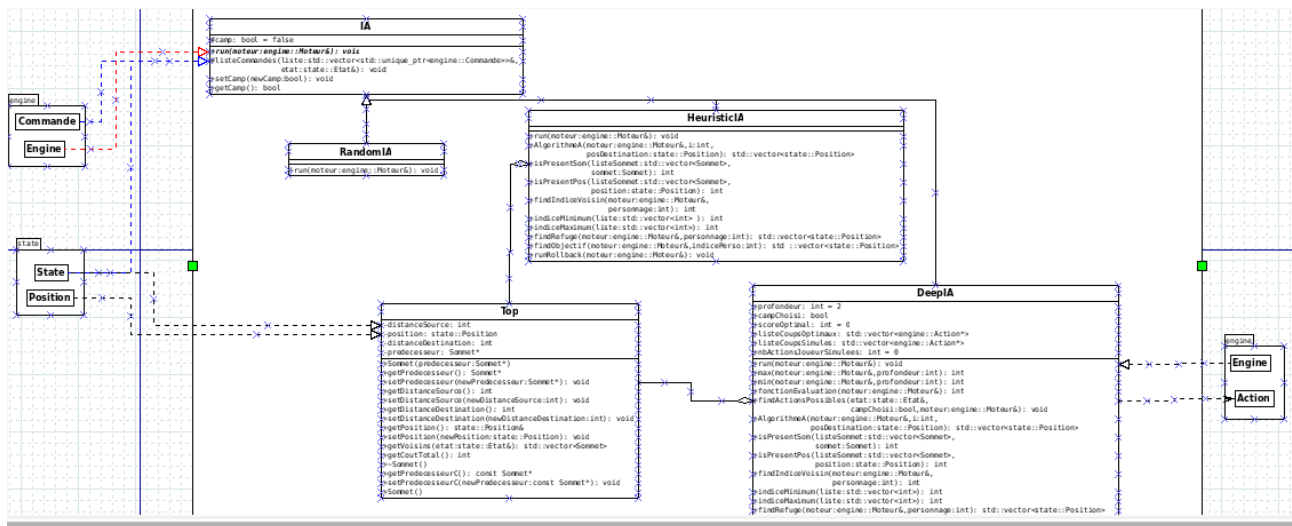
**Classe IA** : Les classes filles de la classe IA implémentent différentes stratégies d'IA. Elles Possèdent un champ et une fonction « run ».

**Classe RandomIA** : Classe qui implémente l'IA aléatoire.

**Classe HeuriticIA** : Classe qui implémente l'IA heuristique. La fonction findObjectif définit quel sera l'objectif du personnage actif, que ce soit la cible de sa prochaine attaque ou la position du refuge le plus proche si ses PV sont trop faibles. L'algorithme A\* permettra dans les deux cas de trouver rapidement le meilleur chemin vers l'objectif.

**Classe Top** : Un sommet est une Position qui possède un prédécesseur. Des instances de Sommet sont utilisées dans la recherche d'un chemin.

**Classe DeepIA** : Classe qui implémente l'IA avancée. Les déplacements sont définis comme pour l'IA Heuristique. La fonction findActionsPossibles liste toutes les actions possibles pour l'IA ou le joueur à un état donné et va simuler toutes les successions de coups possibles de manière récursive pour trouver quelle est la suite d'actions optimale pour l'IA avec un algorithme minimax.



### 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 une 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*

