



Projet Logiciel Transversal

Jeu au tour par tour multijoueur :

« Guerre des gangs »

Romain Loury
Matthieu Debarge

Sommaire

Introduction.....	3
1. Objectif	3
1.1. Présentation générale	3
1.2. Règles du jeu	3
1.3. Conception logiciel	4
2. Description et conception des états.....	4
2.1. Description des états.....	4
3. Rendu: Stratégie et conception.....	6
3.1. Stratégie de rendu d'un état	6
3.1.1. Les Textures.....	6
3.1.2. Affichage de la map.....	7
3.1.3. Les views.....	7
3.1.4. Les animations.....	8
4. Changement d'état et règle de jeu.....	10
4.1. Changement extérieur.....	10
4.2. Changement intérieur	10
4.3. Conception logiciel	11
5. Intelligence Artificielle.....	12
5.1. Stratégie	12
5.1.1. Intelligence artificielle simple.....	12
5.1.2. Intelligence basé sur heuristiques.....	12
5.1.3. Intelligence par recherche dans un arbre	13
5.2. Conception logiciel	13
6. Modularisation	14
6.1.1. Répartition des threads sur la machine	14

Introduction

Ce rapport traite du projet « Guerre des gangs ». Il s'agit d'un jeu vidéo développé dans un objectif d'apprentissage de la programmation logicielle et de ses principes. Un travail de conception, de programmation, d'optimisation et de service réseau sera mis en place au sein de ce projet. Celui-ci revêt un caractère transversal dans la mesure où il sera la mise en application de savoirs provenant de différentes matières telles que le génie logiciel, l'algorithmique, la programmation parallèle. Le projet est construit sur un total de 112 heures de travaux pratiques encadrés, et se conclut par un produit fini, ayant des fonctionnalités avancées, à la fois robuste et évolutif.

1. Objectif

1.1.Présentation générale

L'objectif du projet « Guerre des gangs » est la réalisation d'un jeu de stratégie pouvant se jouer seul contre une intelligence artificielle ou à plusieurs. Le jeu se déroule sur une carte. Il s'agit d'un jeu au tour par tour, l'état du jeu n'est modifié qu'à la fin d'un tour après que chaque joueur ait renseigné son coup. Le principe est simple, conquérir les territoires des gangs adverses en déplaçant ses recrues sur une carte.

1.2.Règles du jeu

La carte est composée d'un ensemble de territoires (une trentaine) et de quartiers généraux. Chacun des joueurs (physique ou intelligence artificielle) possède un quartier général, le but du jeu est de conquérir les QG de ses adversaires. Chaque joueur commence avec un certain nombre de recrues dans son QG. Pour conquérir d'autres territoires, le joueur peut envoyer une partie de ses recrues sur des territoires adverses, il s'agit d'une attaque. Lorsqu'un joueur envoie un nombre de recrues supérieur au nombre de recrues présent sur un territoire alors il obtient ce territoire. En revanche si un joueur attaque un territoire adverse ayant plus de recrues, le joueur attaquant se voit perdre toutes les recrues qu'il a engagées dans l'attaque et n'obtient pas le territoire qu'il souhaitait, le territoire attaqué conserve toutes ses recrues, ou une partie en fonction du nombre de recrues l'ayant attaqué. En cas d'égalité le territoire gagnant est attribué aléatoirement. A chaque tour, tous les joueurs possèdent 3 déplacements d'un territoire vers un autre. Tout l'intérêt du jeu et de la stratégie est d'essayer d'anticiper les coups des adversaires, pour savoir où placer ses recrues. Après chaque tour, chaque joueur se voit attribuer une certaine valeur d'argent lui permettant d'acheter de nouvelles recrues. Plus le joueur possède de territoires, et de QG, plus il obtient d'argent. Pour ajouter de la stratégie au jeu, il existe des cartes, qui peuvent aussi être achetées par le joueur. Les cartes sont plus ou moins puissantes en fonction de leur prix, il existe trois catégories de cartes de puissance différentes, le joueur peut en acheter une seule à chaque tour et ne connaît pas ses effets lorsqu'il l'achète, il connaît uniquement la puissance de la carte. Les cartes ont des effets sur le jeu, voici une liste non exhaustive d'effets introduits

par application d'une carte : réduction de l'effectif de recrues adverse d'un certain coefficient, ajout d'un certain nombre de recrues à un secteur, annulation des attaques des ennemies sur un ou plusieurs territoires, et d'autres. Le jeu s'arrête lorsque tous les QG sont conquis par un joueur. Lorsqu'un joueur n'a plus de QG la partie est perdue pour lui, même s'il lui reste un territoire qui n'est pas un QG. Il y a 9 cartes de 3 types différents, A, B, ou C

Type A: *-nombre de recrues des secteurs adjacents $\times 0.25$*
-nombre de recrues des secteurs adjacents + 10
-Annulation des attaques provenant des secteurs adjacents

Type B: *-nombre de recrues des secteurs adjacents $\times 0.50$*
-nombre d'alliés de l'attaque $\times 2$
-Annulation des cartes ennemies joués dans les secteurs adjacents

Type C: *-nombre de recrues du secteur attaqué $\times 0.50$*
-nombre d'alliés d'une attaque + 10
-Annulation des cartes jouées dans le secteur attaqué

Plusieurs carte de même type et ayant les mêmes effets peuvent être possédés et joués par différent gangs sur la carte

1.3. Conception logiciel

2. Description et conception des états

2.1. Description des états

Un état du jeu est défini par l'état de l'ensemble de ses éléments. Tous les éléments sont fixes, la carte ne se déplace pas au cours du jeu, ni les autres éléments. Au cours du jeu, certains attributs évoluent et d'autres sont fixes. Voici la liste des éléments du jeu.

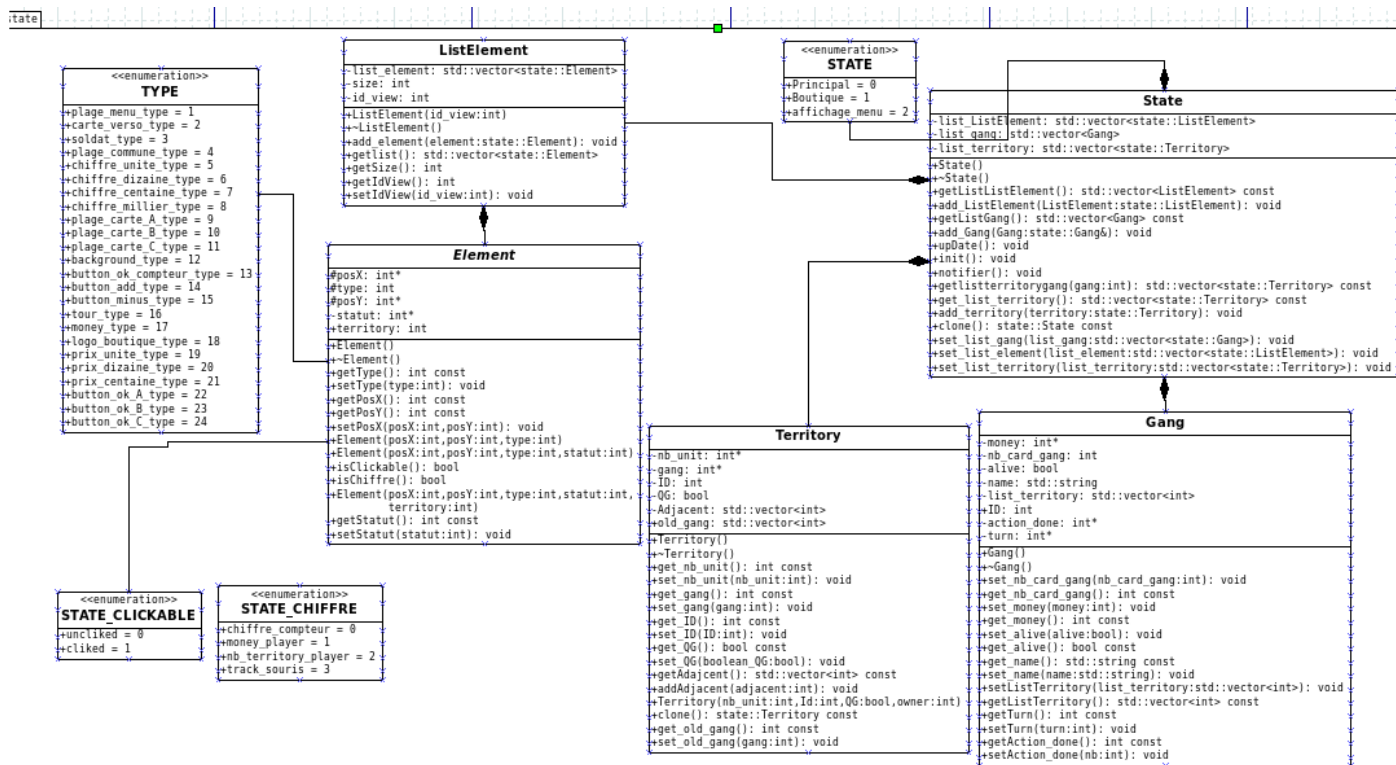
-Élément « **Gang** » : Les gangs et sont définis par 6 attributs, l'argent disponible gagné au fil des tours, les cartes possédées, le nom du gang, un booléen définissant si le Gang est encore en vie, la liste des ID des territoires conquis et un ID qui permet de reconnaître le gang.

-Élément « **Elément** » : Cette classe permet de créer tous les élément qui composent la fenêtre ainsi que leurs paramètres. Chaque élément possède une position en X et Y. On différencie cependant plusieurs grands groupes d'éléments :

-les chiffres : ils peuvent appartenir à des territoires, ont un statut suivant leurs fonctions.

-les clickables : Le constructeur des cliquables prend en compte leur états : 1 si ils sont cliqués, si ils ne sont pas cliquer . Ce statut sera le lien avec les contrôles, lorsqu'un élément est cliqué, on vient changer les états en conséquence.

-les autres



Graphe d'état state

3. Rendu: Stratégie et conception

3.1. Stratégie de rendu d'un état

Dans un état particulier, le joueur doit pouvoir être informé de l'ensemble des variables de cet état. Le rendu graphique permet d'afficher les variables d'un état faisant ainsi le lien entre les données et l'affichage. On vient donc charger un rendu graphique qui dépend entièrement de l'état. A chaque fois que l'état change, on charge le rendu graphique correspondant à cet état.

3.1.1. Les Textures

Pour afficher les éléments, on vient tout d'abord charger leur texture via la classe textures.

-Classe « **Textures** » : On vient charger la texture d'un élément en lui faisant correspondre un Sprite suivant son statut, sa position, son type. On obtient alors le Sprite de l'élément.

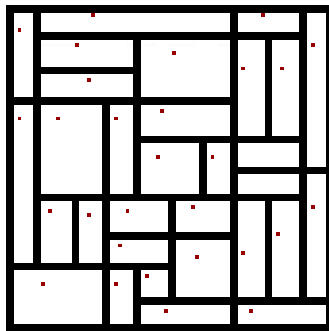
Une fois que l'on a les textures, il nous reste à pouvoir les afficher, ceci est réalisé dans la classe View.

3.1.2. Affichage de la map

L'élément central du jeu est la carte, parce qu'elle permet au joueur de définir ses actions, sa stratégie. Pour créer la carte, nous utilisons le logiciel paint.net qui nous permet d'éditer une image. L'image au format PNG est ensuite traitée par la classe Tile qui est ensuite utilisée par la classe TileMap qui permet de dessiner la carte.

-Classe « **Tile** » : avec un argument « `sf::Image` », créer un pointeur sur un tableau à deux dimensions correspondant aux numéros des tuiles associées à chaque pixel de l'image. Les tuiles sont positionnées suivant le placement des frontières. Il nous faut des frontières de deux pixels soit deux tuiles pour créer celle-ci. Une autre fonction de Tile renverra à partir de la même image, la position des nombres permettant d'afficher le nombre de recrues possédées par chaque secteur ainsi que les tours. Le fait d'utiliser une fonction traitant une image permettra de modifier la carte sans avoir à modifier tout le code source et de façon plus simple et rapide. On récupère de la même façon les positions des tours des secteurs ainsi que leurs numéros associés.

-Classe « **TileMap** » : est un élément généré à partir du tableau de tuile. Cette fonction permet de spécifier la texture de la map. Cet élément est drawable afin de pouvoir l'afficher via la librairie SFML.



Carte du niveau envoyée par paint

3.1.3. Les views

Les views permettent de diviser l'écran et donc de séparer un affichage. Elles sont générées à partir de la classe View. Elles correspondent en quelque sorte à des couches que l'on superpose.

-Classe « **View** » : elle permet grâce à sa fonction init, d'initialiser toutes les vues. Chaque view possède un ID qui est le même que ceux utilisés pour les listes d'éléments qui les composent. Les différents ID sont associés aux vues :

- la map
- la plage commune
- la boutique

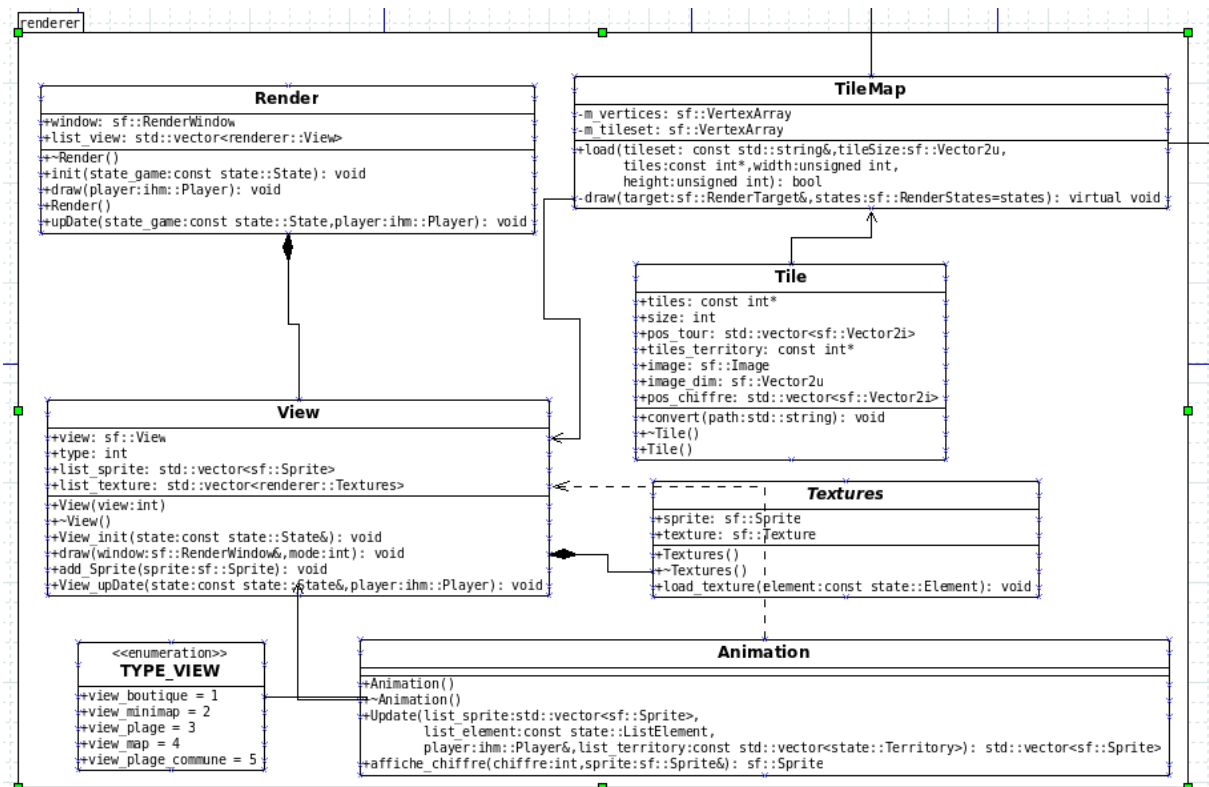
- le menu
- la minimap

On initialise toutes les vues suivant leur ID avec la méthode init. On vient créer aussi des liste de Sprite correspondant au listes d'élément de chaque vu qui sera utilisé dans la méthode draw ().

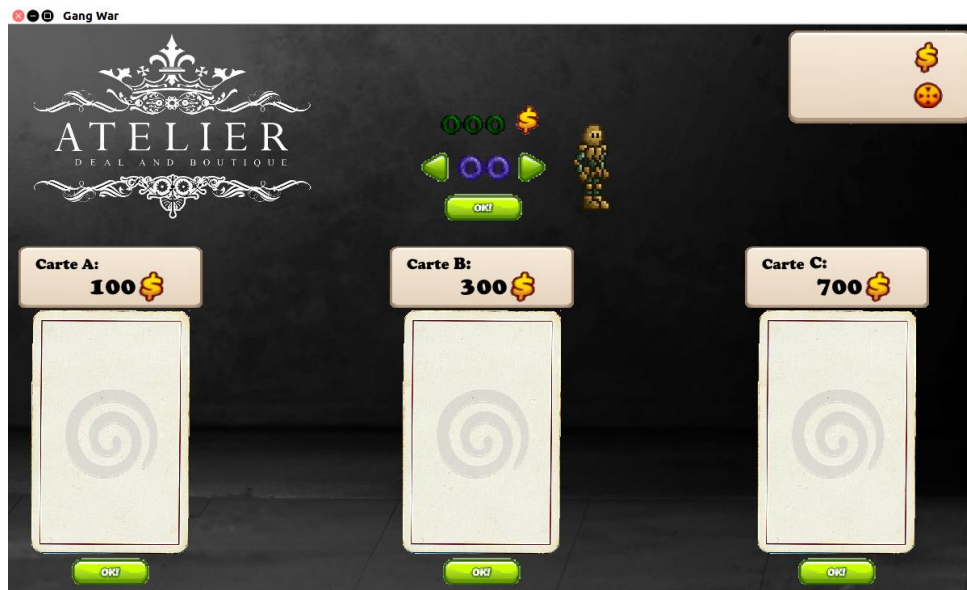
3.1.4. Les animations

On créer alors la classe animations qui permettent de changer les Sprite suivant l'état des différents éléments des différentes vues. Pour cela on a la classe Animation.

-Classe « **Animation** » : Suivant le Update de l'état, la classe mets à jour l'affichage des Sprite.



Le dia du package renderer



Rendu de la boutique (Mode=0)



Rendu de la Carte (Mode=1)

4. Changement d'état et règle de jeu

On définit les changements d'état dans le package engine . On a en réalité deux type de changement d'état : Les changements d'état du à l'IHM et les changements d'état du aux commandes.

4.1.Changement extérieur

On a trois modes de jeux :

Un mode de jeu qui est la map, sur laquelle on interagit pour déplacer les unités et voir comment le jeu avance.

Un mode de jeu qui est la boutique et qui a pour but de faire acheter des unités ou des cartes.

Un mode de jeu ou on interagit sur un territoire en gérant ses unités.

Le changement d'état entre boutique et map ce fait par l'intermédiaire du bouton B du clavier dans Clavier. Le changement d'état entre la map et la gestion du territoire se fait si on sélectionne une tour de la map.

4.2.Changement intérieur

On vient donc permettre plusieurs actions suivant le mode ou l'on est. Ces actions font encore partie de l'IHM. La détection des contrôles est faites dans le main et est gérer par la classe Souris et Clavier via leur méthode gestion. Ensuite, une fois cette détection faite, on vient changer l'état grâce à la méthode Update de la classe Engine.

Pour le mode boutique :

-Si on clique sur les boutons plus ou moins on incrémente le compteur d'unités

-Si on clique sur Ok, on vient enlever à notre argent soit le prix d'une carte soit le cout des unités achetées suivant le bouton.

Pour le mode map :

-Si l'on clique sur une tour, on signale que le joueur à sélectionner une tour, on passe par conséquent en mode 2.

-Si l'on touche les flèches du clavier, on vient bouger la position de la vue du joueur. Cette attribut est directement relia au package renderer.

Pour le mode gérer un territoire :

-Si l'on clique sur les boutons plus ou moins on incrémente le compteur

-Si l'on appuis sur Ok et que l'on a assez d'unité disponible, on change le nombre de d'unité pris par le joueur avec le nombre d'unité pris si le chiffre était nul avant. Si ce nombre était non nul, on met en fonction la commande Move_unit.

4.3.Conception logiciel

Dans le package engine, on distingue une partie de control qui gère le clavier et la souris et une partie Commande qui permet d'envoyer des commandes au moteur de jeu.

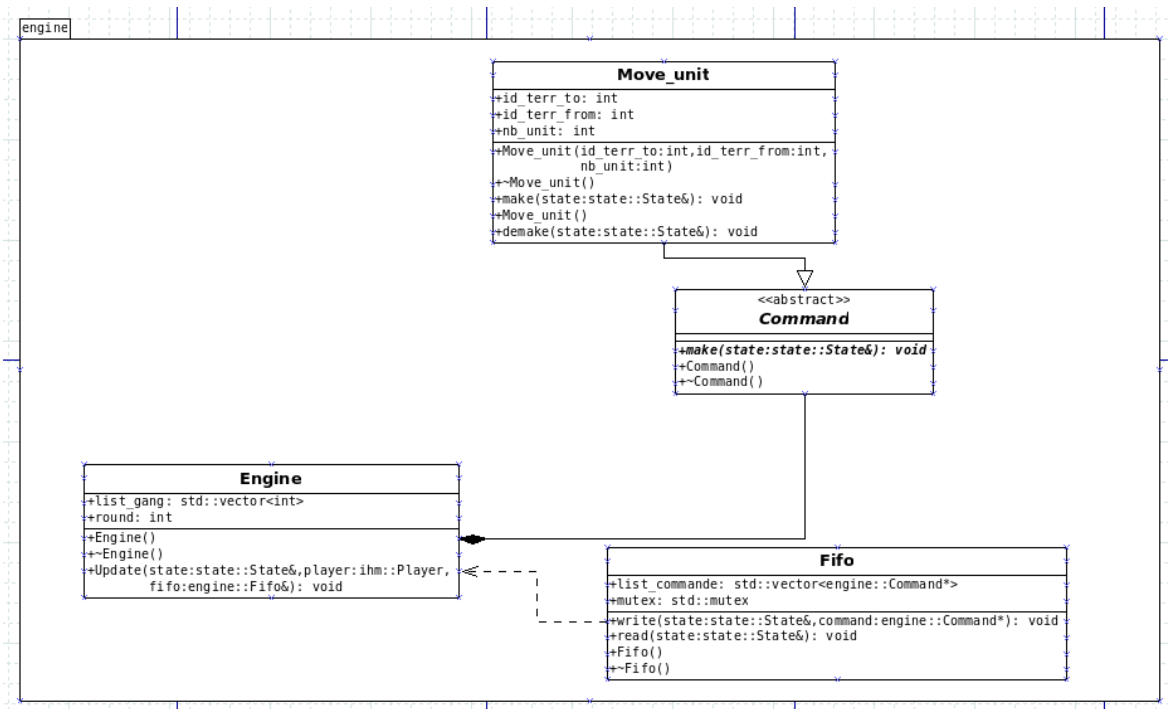
-Classe « **Clavier** » : Cette classe a pour but de gérer les actions du joueur sur le clavier. Si le joueur appuis sur une touche, la méthode gestion_clavier permet de faire la transition entre ce qu'a fait le joueur sur le clavier et le changement d'état associé. Grâce au clavier le joueur a accès à la boutique et peut naviguer sur la carte.

-Classe « **Souris** » : Cette classe a la même fonction que la classe Clavier sauf qu'ici on détecte la position du click de la souris afin de changer l'état en conséquence.

-Classe « **Move_unit** » : La classe Move_unit est une commande, la seule commande du jeu qui permet le game play. Cette classe a comme attribut : l'identifiant du territoire de provenance, identifiant du territoire cible et le nombre d'unité pris. Cette classe permet aussi l'application de cette commande via la méthode play() avec notamment les changements d'état associés.

-Classe « **Engine** » : Cette la classe qui permet de gérer les commandes et les changements d'états du à l'IHM.

Pour l'instant on a une IA qui génère des Move_unit aléatoirement en appuyant sur la touche A du clavier.



Dia du moteur de jeu

5. Intelligence Artificielle

Dans ce thème nous allons voir comment est créée l'intelligence artificielle. On constitue d'abord une IA simple basée sur le hasard qui permet de jouer des coups quoi qu'il arrive. Après on définit tous les coups possible et on définit un score avec des heuristiques qui est ensuite repris dans un MIN MAX.

Pour la déclencher, l'IA joue toute seule à la fin du tour du joueur, c'est-à-dire après que le joueur est effectué 3 move_unit. La gestion de la l'achat des unités est fait automatiquement.

5.1.Stratégie

5.1.1. Intelligence artificielle simple

Pour cette intelligence, elle est utilisée lorsqu'on a plus de coup de à jouer. Elle vient prendre un coup au hasard dans une liste de coup possible. Pour cela on vient scanner toute les possibilités de coups cependant on ne s'occupe uniquement des territoires pris pour cible et du territoire attaquant. La gestion du nombre d'unité sera vue après.

5.1.2. Intelligence basé sur heuristiques

Pour l'instant, on a défini certaines règles :

-si on a un territoire avec plus d'unité qu'un autre territoire et qui est pas le nôtre, on l'attaque

-si on ne peut pas attaquer on joue un coup de défense afin de déplacer les unités des territoires les plus peuplé au moins peuplé.

Avec cette règle, l'IA est capable de battre le hasard mais aussi de se défendre face à un joueur réel.

De plus, on vient regarder la somme de unité adverse pour la première, deuxième et troisième périphérie afin d'ajuster le nombre d'unité pour que l'attaque soit la plus efficace.

5.1.3. Intelligence par recherche dans un arbre

On va créer un arbre. Pour cela on va devoir voyager dans un état qui n'est pas l'état du jeu, d'où la nécessité de la méthode clone() de la classe état . On créera donc l'arbre en appliquant les commandes sur cet état temporaire. De plus afin de permettre un voyage facile dans l'arbre on crée une méthode demake qui permet de défaire un make. La navigation dans l'arbre en est plus facile. A partir ça, va pouvoir créer une fonction récursive afin de pouvoir ajouter de la profondeur autant que l'on le souhaite. On crée donc un arbre avec cela. Ensuite, va devoir scorer les dernières feuilles. Pour cela on va utiliser une fonction score qui va établir le score en fonction du nombre de territoire possédé par l'IA. On vient alors résoudre tous les nœuds en regardant à quel gang est le tour. En sachant que l'on fait un max max max min min min. Le nœud parent vient prendre le score soit du plus petit si c'est au tour de l'adversaire de jouer soit du plus haut score si c'est à lui de jouer. On remonte l'arbre comme ceci et on envoie la référence de la commande à jouer.

5.2.Conception logiciel

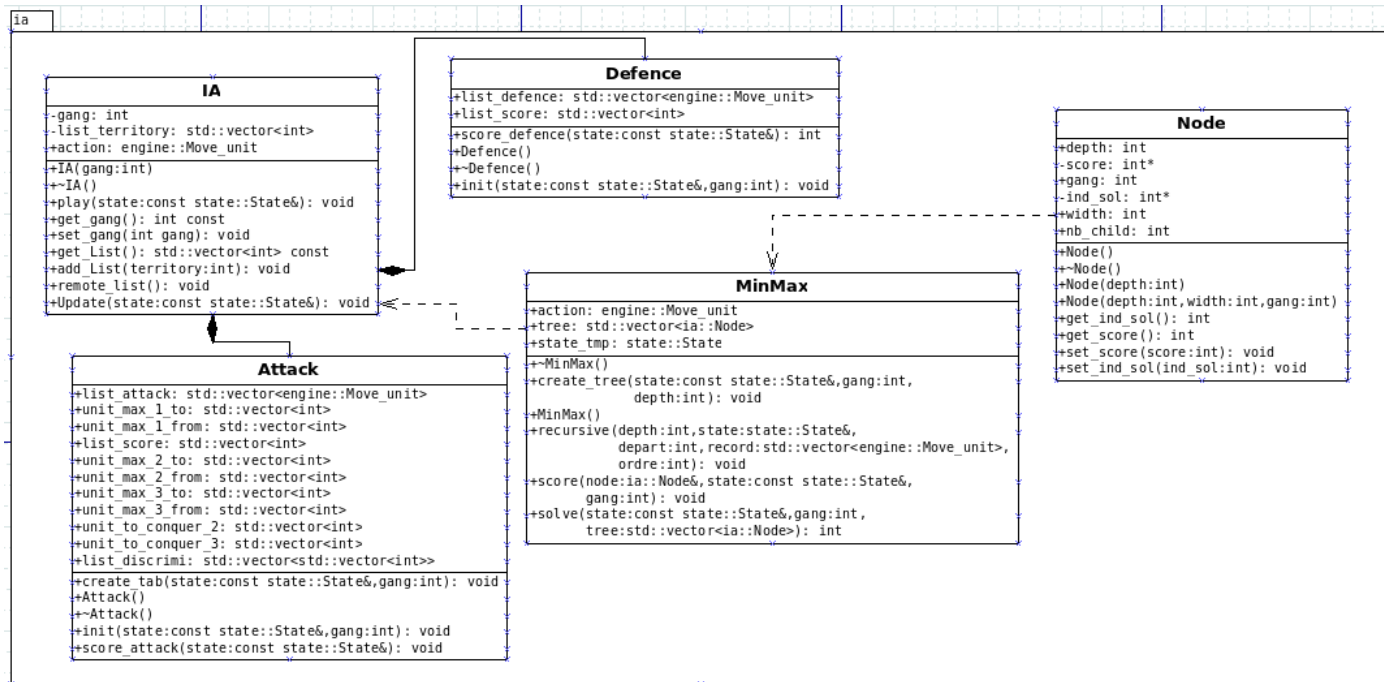
-Classe « **Attack** » : Cette classe a pour but de générer tous les types d'attaque possible. Les types d'attaque contiennent juste les cibles et les attaquants. Le nombre d'unité est par défaut 1+nb_unit de la cible. On a une classe qui permet de cb d'unité pour assurer que le territoire est en sécurité 1, 2 ou 3 tours. On assigne alors le nombre d'unité qui permet de perdre les territoires le plus tard possibles pour l'attaquant mais aussi pour le territoire attaqué.

-Classe « **Defence** » : Cette classe a le même but que la classe Attack sauf pour la défense. On vient regarder en plus quel est le coup de défense le plus adapté en lui donnant un score plus ou moins haut. La mise à jour de la liste est effectuée de la même façon.

-Classe « **Node** » : La classe permet définir les éléments nœuds qui seront ce qui composeront l'arbre. On les initialise avec une profondeur, un gang et une largeur. On a ensuite une méthode pour donner un score au nœud ainsi que la position de cet élément qui lui a donné son score (utile pour la profondeur 0).

-Classe « **MinMax** » : La classe permet de construire un arbre de profondeur depth. Cet arbre est construit de façon récursive grâce à l'état courant. On vient cloner l'état grâce à sa méthode clone(), puis on travaille sur cette état afin de ne pas perturber la partie . On a aussi une méthode score qui permet de donner un score aux feuilles finales qui est utilisé dans la méthode create_arbre() . Enfin la méthode solve() permet de résoudre l'arbre et de renvoyer l'id de la commande à jouer.

-Classe « IA » : La classe IA permet de générer un joueur qui envoie des commandes par la méthode play. Il a comme attribut son gang pour savoir quelle est son gang et sa commande ainsi que la liste de ses territoires.



Dia de l'IA

6. Modularisation

6.1.1. Répartition des threads sur la machine

Le but de cette partie est de permettre un multithreading entre le rendu et le moteur du jeu. Afin de pouvoir faire cela, nous devons créer un nouveau thread en plus du thread principal du main. Grâce au pattern commande, nous avons créé une classe FIFO dans le package « Engine » qui contient une liste de commande mais aussi un mutex. La liste des commandes sont l'empilement des commandes à exécuter. Peu importe la commande, ici il y a que une commande possible mais ceci est pour une extension possible, qui est envoyé elle est détypée en classe « Commande ». Cette pile doit avoir une méthode pour écrire dans la liste et une méthode pour lire la commande. C'est une FIFO. Pour lire la commande, on fait

intervenir la méthode make de la classe fille de la commande. En effet, la classe mère « command » est abstraite. Le mutex est là pour réguler l'écriture et la lecture dans la FIFO. Il assure que l'on ne peut pas écrire dans la liste en même temps que l'on lit dedans.

Pour ce qu'il y est du multithreading, le thread principal qui contient aussi la classe Engine vient écrire dans la liste FIFO en prenant le mutex de la liste. Des commandes venues du joueur ou de l'IA suivant le tour de jeu sont donc notées grâce à la méthode write. La lecture de la pile est faite dans le thread créé en dehors de la boucle du thread principal en utilisant la méthode read la classe FIFO. Celle-ci est une boucle avec une condition sur la taille de la liste qui doit être supérieur à 0 pour lire dans la liste. Si la condition est vérifiée on vient appliquer la méthode make de la commande en prenant le mutex et donc changé l'état. La commande est supprimée de la liste. Le rendu est fait dans le thread principal.

Note : Pour jouer c'est assez simple. On rentre dans le jeu, on a 0 argent mais un QG qui est au centre et c'est adjacent qui nous appartiennent. On est le joueur en vert. L'IA est en noir. On joue nous 3 coups et l'IA prend le relais, une fois qu'il a joué on a gagné de l'argent proportionnellement aux territoires possédés. On appuis sur B pour acheter les unités, les unités sont mises directement dans le QG puis on rejoue nos 3 coups etc...