Projet Logiciel Transversal

Antoine Delavoypierre – Nicolas Leteinturier

Table des matières

1 Objectif

1.1 Présentation générale

Le but de ce projet est de programmer un jeu basé sur l'archétype du Risk. Le jeu aura pour but de respecter le cahier des charges:

-être un jeu en Multijoueur afin de pouvoir avoir des application serveur et programmer une IA

-le jeu doit inclure une IA avec 3 Niveaux de difficultés

-le jeu doit être un jeu à état, pour Stocker les données du jeu a tout moment

-le jeu doit pouvoir être mis en réseau.

Le Risk est un eu de stratégie en tour par tour qui permet de jouer a plusieurs joueur les uns contre les autres. Ceci inclue donc une possibilité de jouer contre un ordinateur. Le but du jeu étant d'élaborer une stratégie militaire afin de gérer des troupes et prendre le contrôle du monde.

1.2 Règles du jeu

Tout en se basant sur les règles de l'archétype, cette version du jeu va prendre certaines libertés. Comme par exemple l'implémentation de plusieurs type d'unité : des unités plus défensives, et des unités plus offensives qui permettent d'avoir un coefficient multiplicateur sur les jets de dès en fonction de la situation.

Au début de la partie:

-Une carte est générée composée de N tuiles

-La carte est divisé entre les joueurs dans la partie

-Un même nombre d'unités militaires sont attribuées a chaque joueur à repartir sur ses territoires.

En Partie chaque joueur joue tour par tour. Les tours se déroule de la façon suivante:

-Renforts:

en fonction du nombre de territoires que possède le jour, il reçoit des renforts à distribué sur les territoires de son choix

-Technologie:

Si le jour a assez de points d'expérience il peut investir ses points dans une technologies pour gagner un bonus

-Attaque:

si un joueur a plus de une unité sur un territoire il peut attaquer un territoire voisin. Le combat se fera en fonction de jet de dès en appliquant les bonus technologiques

-Déplacement:

Pour finir le tour le joueur peut déplacer des unités d'un territoire A vers un territoire B si les territoires son relié par du territoire du joueur.

Fin de partie:

-Un joueur perd si il n'a plus de territoires.

-Un joueur gagne si il possède 90% des territoires

1.3 Conception Logiciel

Afin de pouvoir programmer le jeu. Le sprite sheet a été créé. Une carte du monde a été divisé en 14 territoires (le nombre a été limité pour simplifier le jeu) chaque territoire a été isolé afin de pour pouvoir le traiter individuellement au niveau de la couleur. En effet la couleur du territoire sur l'interface graphique sera déterminé par quel joueur possède le territoire. Afin d'éviter de devoir préparer les sprites sheet de chaque couleur. La carte globale sera une superposition de territoires coloré individuellement grâce a SFLM.

2 Description et conception des états

2.1 Description des états

L'état du jeu a tout instant est défini par la possession de chaque territoire par un unique joueur, et par le nombre d'unité que ce joueur possède sur chaque territoire. Ainsi que l'ordre de passage des joueurs, et le joueur actuel qui doit jouer. Ainsi a tout instant il est possible d'interrompre et reprendre le jeu uniquement grace à :

-la liste de joueurs.
-la liste des pays avec le nombre d'unité dessus
-le joueur actuel

2.2 Conception logiciel

Pour définir ces états nous avons défini plusieurs classes.

Dans un Premier temps la classe GameState, qui contient le vecteur des joueurs donc l'ordre de passage, le vecteur des pays (ces vecteurs contiendront des pointeurs vers des objets définis dans les classes Player et Country), et le joueur actuel qui doit jouer.

La classe GameState gére les ajoutes et suppressions de joueurs soit lors de la création de la partie ou ultérieurement lors d'une défaite géré par le moteur de jeu. Une autre méthode dans la classe permet de déplacer des unités.

La classe Player défini par un identifiant unique.

Les méthodes de cette classe permettent de gérer les renforts reçu par un joueur en fonction du nombre de territoire qu'il possède.

La classe Country contient les informations sur chaque territoire. Comme le nombre d'unité et leurs types sur un territoire (sous forme d'un vecteur de Type d'unité). Afin de pouvoir attaquer les pays voisin pour chaque territoire un vecteur contenant les pointeurs vers les pays voisins et defni. Lors de la phase d'attaque le moteur autorisera seulement les attaques sur les territoires voisins ennemies. Ainsi le Pays doit aussi contenir les informations sur le propriétaire du pays.

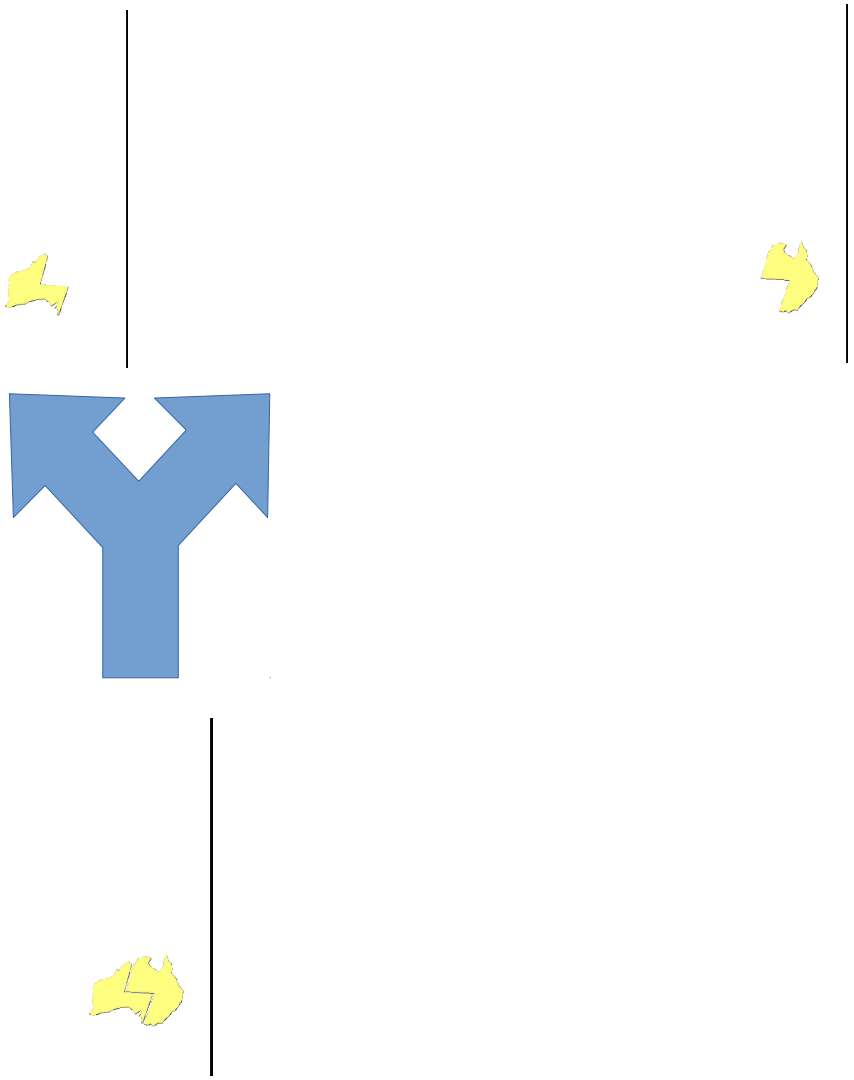
Les méthodes de la classe Country ont pour but de gérer les unités pressentes sur le territoire. Soit ajouter des unités en cas de renfort. Éliminer des unités en cas de perte en attaque ou défense. Afin d'initialiser la carte et la liste des voisins, une méthode permet d'ajouter des territoires voisins.

La classe Unité et Type afin de gérer les différents type d'unités que le joueur pourra choisir pour élaborer une stratégie.

La Classe Action permet de gerer toutes les actions/envenimement du jeu, c'est a dire les Attaques, les mouvement, les renforts…. Lors d'une attaque par exemple les troupes des deux territoires impliqué sont enlevé temporairement des deux territoires pour etre traité dans la classe action. Si elle sont victorieuse elles seront ainsi replacé à la fin dans le territoire ou elle sont supposé aller. Si elles sont detruite une methode les eliminera. Ainsi la classe Action permet de gerer quelle type d'action sera utilisé pour l'etat de jeu. Les autres classes comme Attaque, mouvement, renforcement et initialisation vont heriter de la classe Action.

2.3 Conception logiciel : extension pour le rendu

Pour le rendu, la classe Country étant défini par un identifiant associé a chaque territoire. Le rendu général sera défini par une superposition des différents territoires. Pour un certain territoire il faut pour le rendu également établir un code couleur en fonction du propriétaire du territoire. Ainsi utiliser une superposition de masque indépendant permet de donner une couleur a chaque territoire plus facilement. Voir exemple de sprite dans le dossier ressources. Ici nous avons l'exemple de l'oceanie qui est divisé en deux terrioires. Leur position respective sur la carte .png permet donc une supperposition. Puis en appliquant un filtre de couleur grace au rendu nous pourrons avoir l'interface.



2.4 Conception logiciel : extension pour le moteur de jeu

Les classes ont été établie pour faciliter l'extension pour le moteur de jeu. La définition d'un territoire par un identifiant et une liste de pays voisins permet au moteur d'avoir un accès facile a

quel territoire un joueur peut attaquer. Avoir également un accès rapide pour chaque territoire au nombre d'unités dessus et au propriétaire permettra d'avoir plus d'informations aux quelles l'IA aura accès ultérieurement dans la programmation de l'IA.

2.5 Ressources

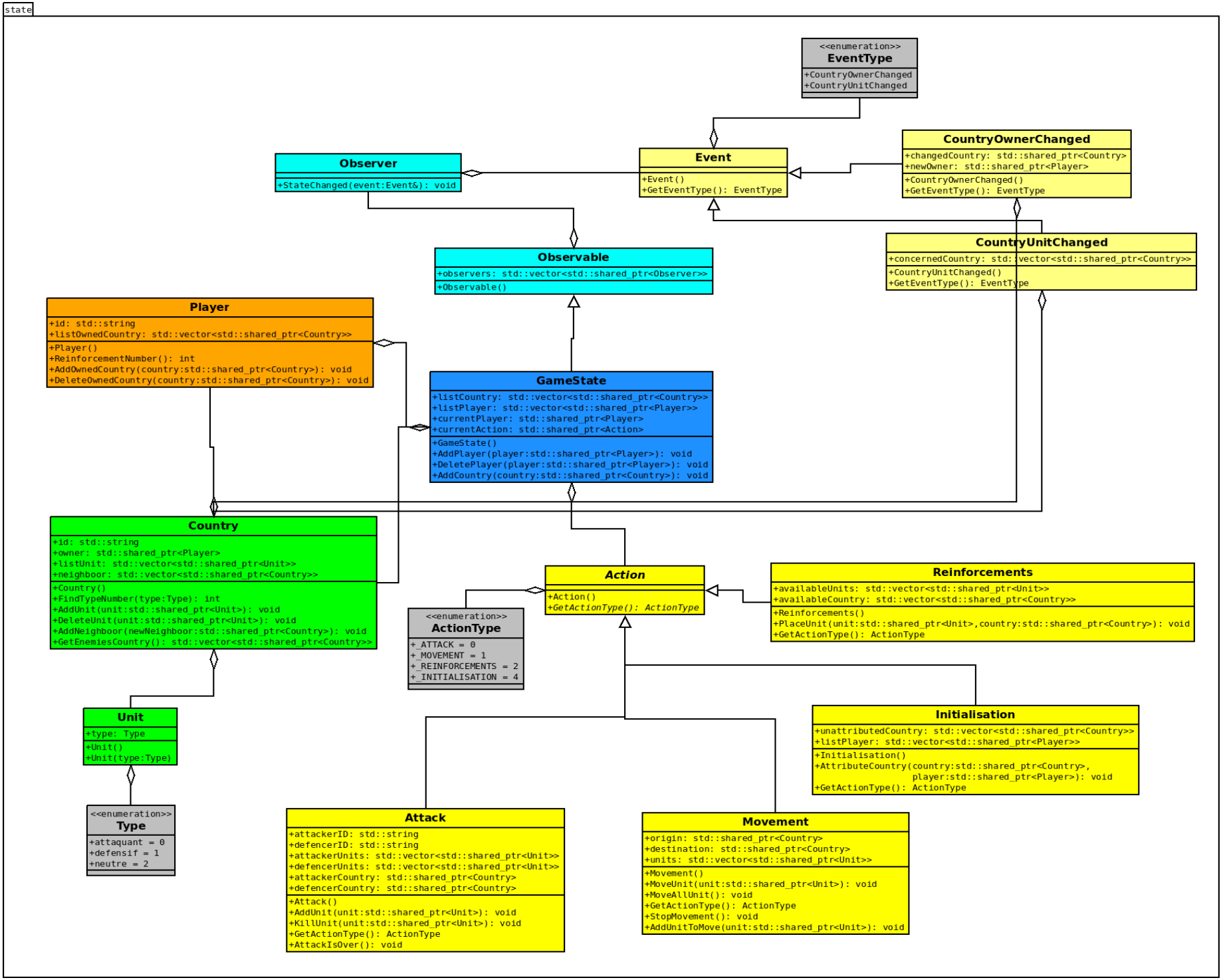


Illustration 1: Diagramme des classes d'état

3 Rendu : Stratégie et Conception

Afin d'obtenir un rendu nous avons décider d'organiser notre diagramme de classe pour le rendu de la facon suivante :

Une classe Scène gérera intégralité des données et de la synchronisation en ayant acces à l'état de jeu et aux autre classes. Scène pourra sélectionner quelle partie du rendu nécessite une mise a jour.

Les deux classe aux quelles ils peut faire appel sont WorldMap, et UnitRepresentation.

La classe WorldMap permet de gérer tout ce qui touche aux pays. C'est a dire la représentation des différents territoires. La classe importe donc dans un premiers temps tous les sprites de pays ce qui permet de construire le monde. Une fois le monde créé cette classe permet également de mettre en couleur individuellement chaque pays afin de visualisé quel joueur est propriétaire du territoire.

La classe UnitRepresentation de sont coté s'occupe d'extraire de l'etat de jeu le nombre d'unités sur chaque territoire afin de le représenter sur la carte. Pour ce faire il faut placer au bon endroit les valeurs des nombres d'unité .

Toute les positions et les addresses des sprites sont stocke dans un fichier « define.h » afin de ne pas avoir de code trop lourd avec des tableaux aux milieu du code.

La classe Layer permet de stocker tous les sprites et textures des deux classes précédentes.

3.1 Stratégie de rendu d'un état

Les classes décrites si dessus permet de traiter de facon independante le rendu des territoires et le rendu des unité ceci est important car le jeu est divisé en plusieurs phase. Dont des phase de renfort , où les territoires ne changent pas. Seul le nombre d'unités change. Ainsi en faisant ceci nous pouvons mettre a jour seulement la partie du scripte qui nous interesse. Ceci nous permet de l'imiter la complexité en temps de calcule et plus de flexibilité.

3.2 Conception logiciel

Afin de donner accès aux informations nécessaires aux bonnes classes les deux classes WorldMap et UnitRepresentation héritent toutes les deux de Layer. Layer ayant accès à certaines données de l'état de jeu. Ceci permet de pouvoir représenter toutes les informations nécessaires à l'utilisateur pour une prise de décision. La derniere classe GraphicElement est une classe pour traiter chaque element de Layer independement. Ainsi on peut pour chaque pays modifier de facon independante des autres sa couleur en fonction du propriétaire grace à un simple methode.
En parcourant la liste des pays issue de Scène la classe UnitRepresentation est capable de retrouver et representer le nombre de chaque type d'unités sur chaque territoire.

3.3 Conception logiciel : extension pour les animations

3.4 Ressources

Nous avons pour cette partie ajouter quelques nouvelles ressources afin d'optimiser le rendu. Dans une première partie nous avons ajouter une représentation pour les différents types d'unités. Qui est representé au centre de chaque pays. Nous avons donc du déterminer la position de chaque pays afin de positionner ce rendu. Les positions sont sauvegardées dans le ficher define.h. Un font de carte a été ajouté pour un aspect plus esthétique. Pour les differents territoires nous avons utilisé les

territoires décrit dans la partie precedente. Avec chaque territoire dans un fichier png. La carte est donc une supperposition de tous les fichiers traité individuellement par le moteur de rendu.

3.5 Exemple de rendu

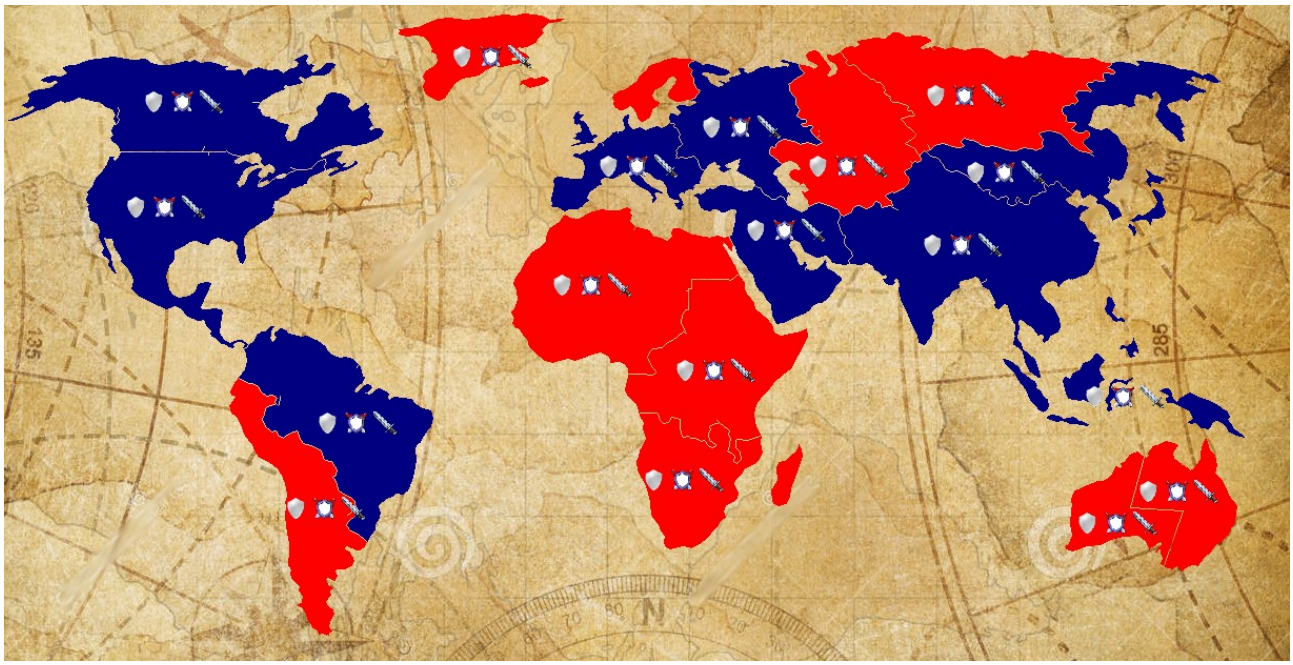


Illustration 2: Diagramme de classes pour le rendu

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

4.2 Changements extérieurs

4.3 Changements autonomes

4.4 Conception logiciel

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