

Projet Logiciel Transversal

Pierre CHABAUD – Alexandre CHAMINAS



FIGURE 1 – Capture d'écran du jeu robostrike.com

Table des matières

1 Objectif.....	3
1.1 Présentation générale.....	3
1.2 Règles du jeu.....	3
1.3 Conception Logiciel.....	3
2 Description et conception des états.....	4
2.1 Description des états.....	4
2.2 Conception logiciel.....	4
2.3 Conception logiciel : extension pour le rendu.....	4
2.4 Conception logiciel : extension pour le moteur de jeu.....	4
2.5 Ressources.....	4
3 Rendu : Stratégie et Conception.....	6
3.1 Stratégie de rendu d'un état.....	6
3.2 Conception logiciel.....	6
3.3 Conception logiciel : extension pour les animations.....	6
3.4 Ressources.....	6
3.5 Exemple de rendu.....	6
4 Règles de changement d'états et moteur de jeu.....	8
4.1 Horloge globale.....	8
4.2 Changements extérieurs.....	8
4.3 Changements autonomes.....	8
4.4 Conception logiciel.....	8
4.5 Conception logiciel : extension pour l'IA.....	8
4.6 Conception logiciel : extension pour la parallélisation.....	8
5 Intelligence Artificielle.....	10
5.1 Stratégies.....	10
5.1.1 Intelligence minimale.....	10
5.1.2 Intelligence basée sur des heuristiques.....	10
5.1.3 Intelligence basée sur les arbres de recherche.....	10
5.2 Conception logiciel.....	10
5.3 Conception logiciel : extension pour l'IA composée.....	10
5.4 Conception logiciel : extension pour IA avancée.....	10
5.5 Conception logiciel : extension pour la parallélisation.....	10
6 Modularisation.....	11
6.1 Organisation des modules.....	11
6.1.1 Répartition sur différents threads.....	11
6.1.2 Répartition sur différentes machines.....	11
6.2 Conception logiciel.....	11
6.3 Conception logiciel : extension réseau.....	11
6.4 Conception logiciel : client Android.....	11

1 Objectif

1.1 Présentation générale

L'objectif de ce projet consiste en la réalisation du jeu multijoueur robostrike(Figure 1) lui même inspiré du jeu de société RoboRally(1994). Les joueurs doivent manœuvrer de manière stratégique un robot sur un quadrillage comprenant obstacles, pièges et bonus afin d'atteindre en premier des balises.

1.2 Règles du jeu

Lors d'une manche, les joueurs choisissent six actions ordonnées (avancer, reculer, droite, gauche, attaquer, ...) avant la fin du temps imparti. Une fois que les joueurs ont tous entré leurs ordres ou que le temps est écoulé, le jeu exécute les actions des joueurs tour par tour.

Les manches s'enchaînent jusqu'à la victoire finale de l'un des joueurs. Le gagnant est le joueur qui atteint le premier tous les objectifs ou qui est le dernier survivant.

A l'exception du départ, plusieurs joueurs ne peuvent pas se trouver sur la même case. Si deux joueurs se retrouvent sur une même case, une collision a lieu et le joueur présent initialement sur la case disputée se fait pousser sur une case adjacente.

Chaque joueur commence la partie avec cinq points de vie et toute attaque entraîne la perte d'un point de vie pour le joueur touché.

Les joueurs obtiennent des bonus tels que des soins, des pièges ou des améliorations d'attaque en passant sur des cases bonus.

1.3 Conception Logiciel

Les dépendances et paquets utilisés sont communs à tous les projets 3IS et sont disponibles sur le dépôt github « github.com/cbares/plt ». La bibliothèque graphique retenue est la SFML. Le schéma ci dessous (Figure 2) montre l'aspect final du projet avec une partie serveur et une partie client. Le serveur sera basé sur micro_httpd qui est un serveur HTTP minimaliste.

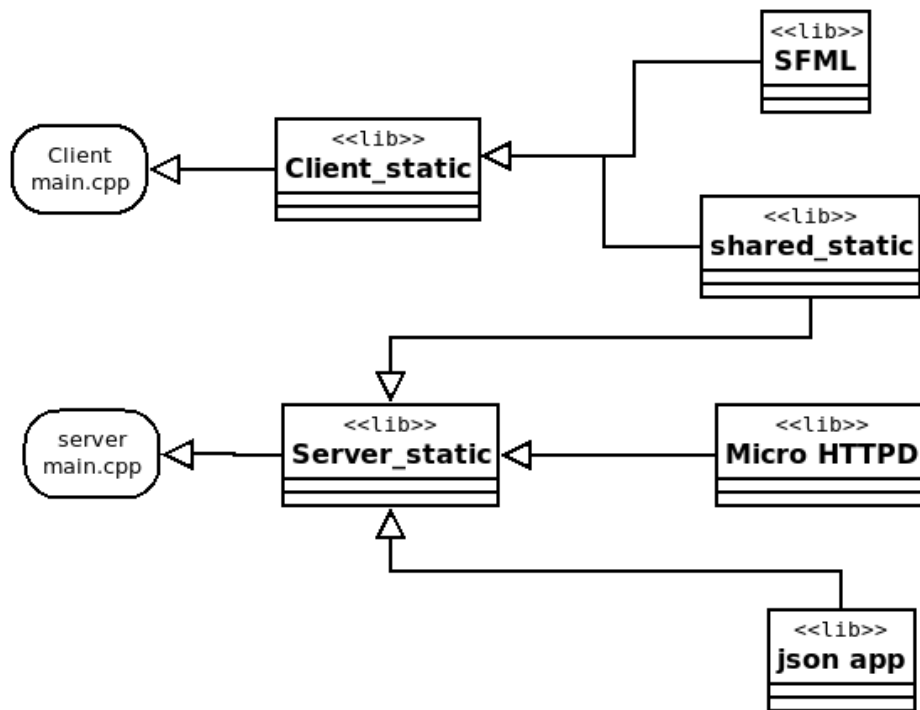


FIGURE 2 – Schéma de la structure logicielle du projet

2 Description et conception des états

2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (la carte de jeu) et d'éléments mobiles (les robots). Tous les éléments possèdent les propriétés suivantes :

- Coordonnées (x,y) dans la grille
- Identifiant de type d'élément : ce nombre indique la nature de l'élément (ie classe)

2.1.1 État éléments fixes

Le terrain de jeu est formé par un ensemble d'éléments « tuiles » composant la carte de jeu. La taille de cette carte est fixée au démarrage du niveau et ne changera pas au cours de la partie. Les types de tuiles sont :

Tuile « Normale ». Les tuiles « normales » sont des éléments neutres et donc franchissables sans risque par les robots.

Tuile « Objectif ». Les tuiles « objectif » correspondent aux objectifs que le joueur doit atteindre en totalité pour gagner la partie. Ces éléments sont numérotés de 1 à N. La tuile objectif dont le numéro est le plus élevé sert de point de réapparition après la mort d'un joueur et le départ de la partie est donné sur l'objectif n°1. De manière purement esthétique, le joueur voit les objectifs d'une couleur différente une fois qu'il est passé dessus.

Tuile « Trou ». Les tuiles « trou » sont des éléments franchissable qui causent la mort instantanée

du robot qui s'y est engagé ou qui a été poussé dedans.

Tuile « Mur ». Les tuiles « mur » sont des éléments infranchissable pour les éléments mobiles.

Tuile « Bonus ». Les tuiles « bonus » sont des éléments franchissables pour les éléments mobiles. Cette tuile peut contenir :

- Une bombe : explose à la fin de la manche en infligeant deux points de dégât aux joueurs touchés dans un rayon de deux cases et les immobilise pour le tour suivant
- Une trousse de soin : régénère toute la vie de vie dès utilisation
- Un tir en croix : inflige deux points de dégât aux joueurs les plus proches dans les directions Nord, Sud Est et Ouest dès utilisation.
- Un bouclier : protège le joueur dès activation et jusqu'à la fin de la manche
- Une attaque automatique : inflige un point de dégât au joueur en face a chaque action de manière automatique et augmente d'un point la puissance des attaques de face et en croix jusqu'à la fin de la manche

Tuile « Rotation ». Les tuiles « rotation » sont des éléments franchissables pour les éléments mobiles et on pour capacité de modifier l'orientation du robot d'un quart de tour dans le sens horaire ou anti-horaire selon le type.

Tuile « Tapis roulant ». Les tuiles « tapis roulant » sont des éléments franchissables pour les éléments mobiles et on pour capacité de modifier l'orientation du robot ainsi que sa position sur la grille.

2.1.2 État éléments mobiles

Dans ce projet un seul type d'élément mobile est nécessaire : le type « Robot ». Il possède une orientation (nord, sud, est, ouest), une couleur, une liste d'objectifs visités, un nombre de points de vie, un nombre de vies et des indicateurs de possession et d'utilisation de bonus.

Élément mobile « Robot ». Cet élément est dirigé par les joueurs ou une IA, qui commande la propriété de déplacement, d'attaque et d'utilisation des bonus. De plus il possède la propriété « couleur », qui est purement esthétique et dont la seule règle est qu'elle est unique. On utilise également une propriété que l'on nommera « ActionStatus », et qui peut prendre les valeurs suivantes :

- Status « normal » : cas le plus courant, où le robot peut se déplacer sur la carte et cherche à atteindre les objectifs
- Status « paralysé » : cas où le robot a été touché par l'explosion d'une bombe à la toute fin de la manche précédente. Il est alors incapable de faire des actions et subit l'influence des éléments de jeu de la carte et les attaques des autres joueurs
- Status « mort » : cas où le robot est tombé dans un piège ou a perdu son dernier point de vie. Cet état intervient entre deux manches avant la réapparition ou dans l'attente de la fin de la partie (le joueur n'a plus de vies mais que les adversaire toujours en jeu la finissent).

2.1.3 État général

A l'ensemble des éléments statiques et mobiles, nous ajoutons les propriétés suivantes :

- **Epoque** : représente « l'heure » correspondant à l'état, ie c'est le nombre de « tic » de l'horloge globale depuis le début de la partie
- **Timer** : représente le nombre de « tic » de l'horloge globale depuis le début de la manche afin de donner la priorité de jeu au joueur ayant validé ses actions le premier. Dans un premier temps l'ordre sera établi en fonction de l'identifiant du robot.

2.2 Conception logicielle

Le diagramme des classes pour les états est présenté en Figure 3, dont nous pouvons mettre en évidence les groupes de classes suivants :

Classes Element. Toute la hiérarchie des classes filles d'Element (en jaune) permettent de représenter les différentes catégories et types d'élément. Nous n'avons pas utilisé le polymorphisme pour faire de l'introspection ne maîtrisant pas encore parfaitement ce concept. Cependant la méthode `isStatic()` ou `isReachable()` s'y prêterait tout à fait et nous travaillons activement à leur implémentation.

Conteneurs d'élément. La classes State permet de contenir des ensembles d'éléments. A partir cette classe nous pouvons accéder à toutes les données de l'état. Par exemple, « `twoDTab` » est un tableau à deux dimensions d'éléments de type « `MapTile` » permettant de contenir la carte de jeu.

Fabrique d'éléments. Dans le but de pouvoir fabriquer facilement des instances d'Element et plus particulièrement de « `MapTile` » (« `Robot` » étant créé une seule au début de la partie et de manière simple) nous utilisons la classe MapFactory. Cette classe permet de créer n'importe quelle instance non abstraite à partir d'un caractère. L'idée est de l'utiliser, entre autres, pour créer un niveau à partir d'un fichier texte.

Observateurs de changements. La conception de ces classes suit le Design Pattern Observer. Le but de la classe Observable dont hérite « `State` » est d'enregistrer des observateurs puis de les notifier à chaque changement d'état. Par exemple la classe `StateLayer` (`render.dia`) est un observateur et met à jour les textures/sprites à chaque action d'un personnage

2.3 Conception logicielle : extension pour le rendu

2.4 Conception logicielle : extension pour le moteur de jeu

2.5 Ressources

Les ressources graphiques comprennent trois packs de textures de 64 par 64 pixels (figures 2, 3 et 4) ainsi que sur un fichier de police d'écriture « `roboto.ttf` ».



FIGURE 2 – Textures pour les tuiles du décor



FIGURE 3 – Textures pour les robots



FIGURE 4 – Textures pour les commandes de déplacement et d'attaque

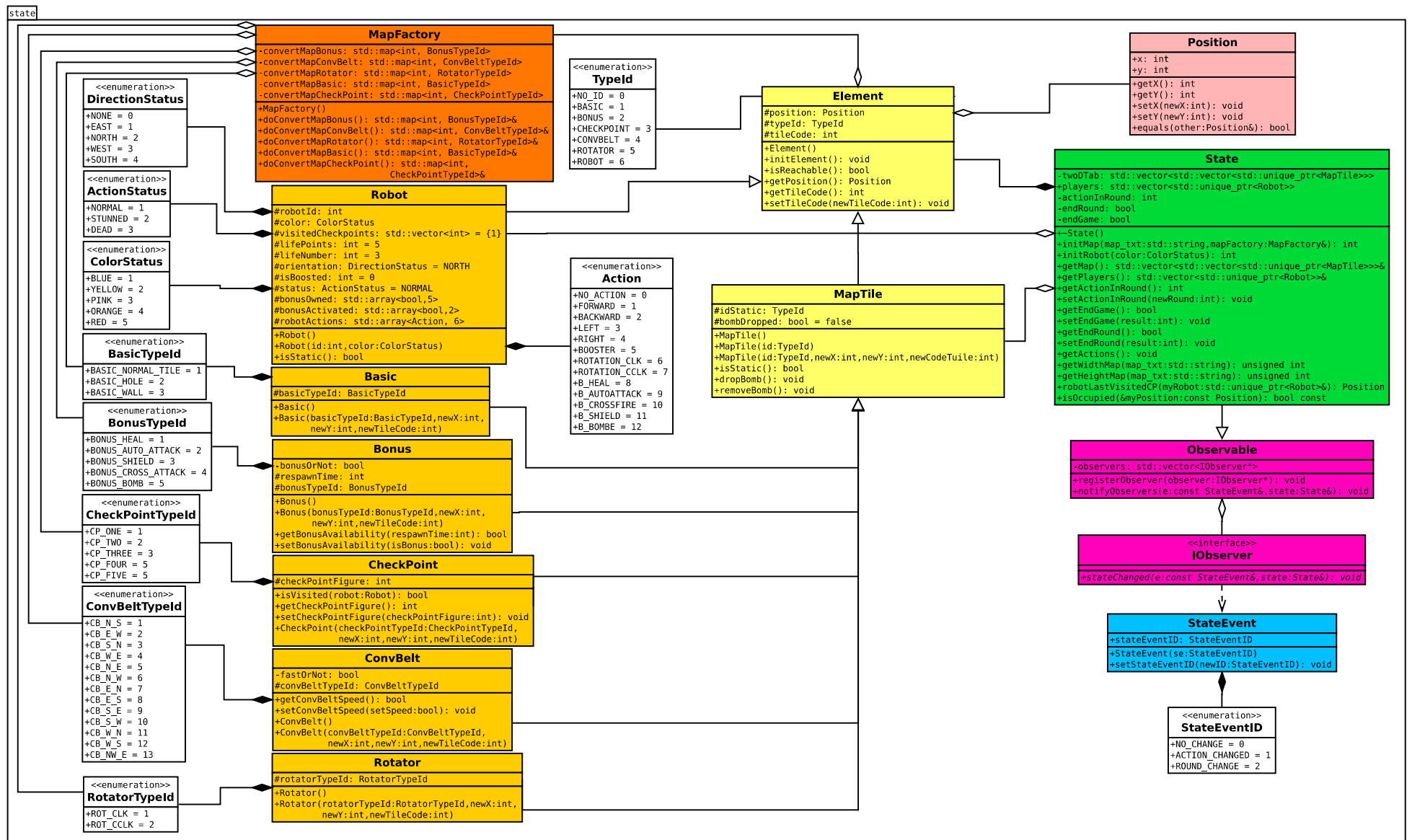


FIGURE 5 – Diagramme de classes du moteur d'états

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Le rendu de la carte de jeu est effectué sous forme de tuiles carrée générées à l'aide de la bibliothèque SFML.

La fenêtre de jeu est divisée en deux parties distinctes. L'aire de jeu se trouve à gauche et comporte les différentes tuiles. Un zoom est appliqué pour rester dans des valeurs de fenêtre raisonnables en cas de map trop grande ou trop petite. A droite est affichée la partie commande ainsi que les informations en textes sur les statistiques de la partie (niveau de vie des joueurs et nombre de vie des joueurs restants).

La grille de la carte est créée à partir d'un fichier texte « map.txt » situé dans les ressources et dont de choix de la taille est laissée à l'appréciation du joueur. Un tableau stocké dans « MapFactory » permettant de relier un entier présent dans le fichier « map.txt », un type d'élément et un nombre permettant de choisir la position du sprite sur l'image « map.png ». Cela est réalisé à l'aide d'un `std::map` qui associe à chaque code de tuile un ID de terrain spécifique.

La méthode `initMap` de la classe `State` (package `state`) 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 `Terrain`).

Le reste de l'affichage est codé en « dur » car ne sera pas modifié d'une partie à l'autre (seul le nombre d'objet varie).

3.2 Conception logiciel

Classe `TileMap`: cette classe possède deux attributs : une texture et un tableau de `Vertex` (quads) contenant la position des éléments et leurs coordonnées dans la texture (map.png). Elle possède les méthodes `loadGrille`, `loadPersonnage` et `loadCurseur` lui permettant d'initialiser ses attributs à partir d'un tableau de `Terrain`, d'une liste de `Personnage` ou d'un `Curseur`. La méthode `Draw` a pour but de dessiner une texture pour ensuite permettre son affichage dans une fenêtre. Elle est codée en virtuelle pour surcharger la fonction `draw` de la SFML.

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 :

- `MAP_TILESET`
- `COMMAND_TILSET`
- `PAWN_TILESET`

En fonction de l'ID passé en `TileMap` adapte ses attributs `cellWidth`, `cellHeight` ainsi que l'arborescence `imageFile`. La classe possède deux attributs : une texture et un tableau de `Vertex` (quads) contenant la position des éléments et leurs coordonnées dans la texture (map.png).

Classe `StateLayer` : Cette classe est la classe principale du moteur de rendu et centralise les informations au de la même manière que la classe « `State` » pour le moteur d'états. Cette classe prend en paramètre 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 deux surfaces grâce à la méthode initSurface (grille de jeu et joueurs) et d'initialiser leurs textures. Cette classe est un observateur, elle implémente l'interface IObserver pour être avertie des changements d'état. Elle réagit ensuite en actualisant les textures.

3.3 Exemple de rendu

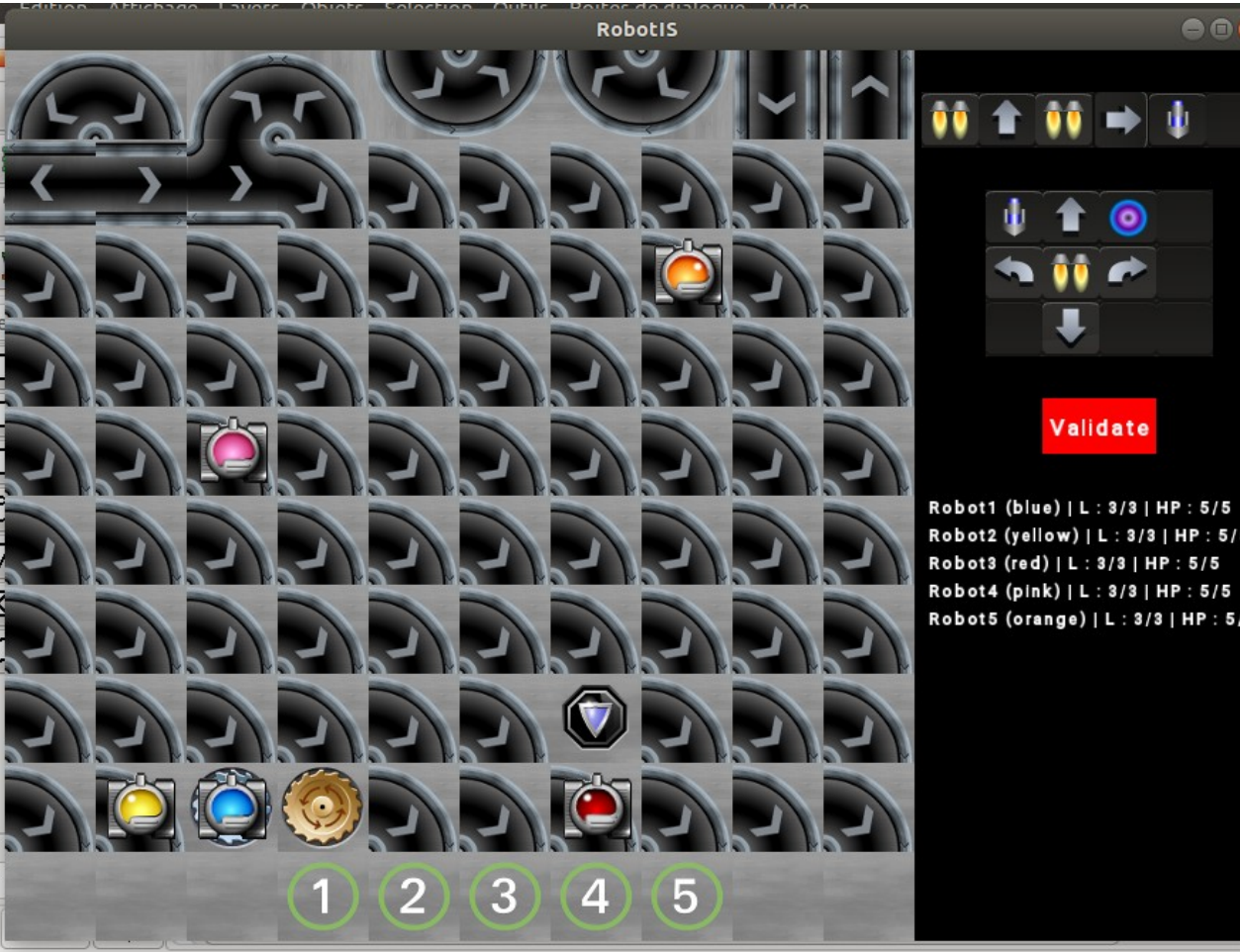


FIGURE 6 – Exemple d’affichage de la fenêtre de jeu

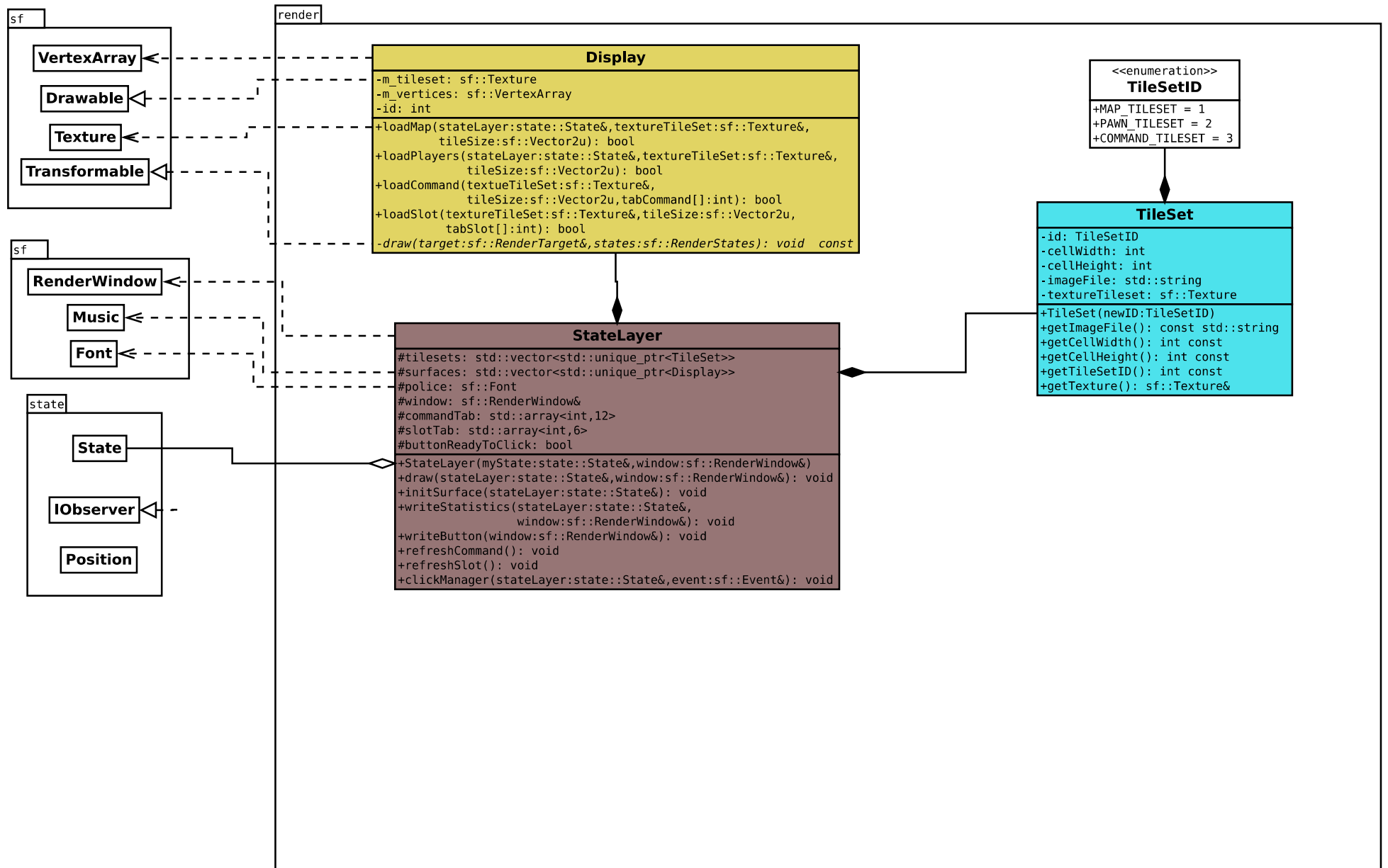


FIGURE 7 – Diagramme de classes du moteur de rendu

4 Règles de changement d'états et moteur de jeu

4.1 Changement d'état

Le jeu est découpé en manches où chaque robot effectue une série de six actions. Une manche débute lorsque tous les joueurs/robots ont validé leurs six actions ou que le temps imparti est écoulé (généralement 30 secondes). Les robots possèdent un statut qui définit leur capacité à effectuer les actions :

- **NORMAL** : le robot est capable d'effectuer ses actions sans contraintes particulières
- **STUNNED** : le robot a été touché lors de la manche précédente et ne peut donc pas effectuer d'action lors de la prochaine manche. Il reste cependant sur la carte et subi les actions des autres joueurs et de la carte de jeu
- **DEAD** : le robot n'a plus de points de vie. Ce statut peut survenir au cours de la manche jusqu'à la réapparition du joueur s'il lui reste des vies ou de manière plus durable jusqu'à la fin de la partie si ses vies sont consommées

Lorsque le robot a un statut **NORMAL**, le moteur de jeu prend en compte ses actions :

- Se déplacer : avancer, reculer, tourner, boost, ...
- Attaquer : attaque de face, attaque circulaire, attaques spéciales
- Utiliser un bonus : utilisation des cinq types de bonus. Ces bonus sont les suivants :
 - *Heal* : Sert à restaurer ses 5 points de vie au maximum
 - *Auto attack* : Le robot lancera une attaque à chaque action de la manche, en plus de son action habituelle
 - *Cross attack* : Attaque se lançant dans les 4 directions
 - *Bomb* : Pose une bombe qui explose en fin de manche
 - *Shield* Bouclier protégeant le robot des attaques adverses

Le moteur de jeu interprète les actions en fonction de l'ordre croissant des 6 actions entrées par les joueurs. Toutes les premières actions sont effectuées, puis le moteur traite les deuxièmes, etc. Les actions de même numéro sont effectuées par ordre de priorité suivant (1 étant la priorité maximale et 4 la minimale) :

1. Utilisation des bonus
2. Attaques (y compris issues du bonus *auto attack*)
3. Déplacements
4. Actions de la carte de jeu sur le robot (trou, plaque tournante, tapis roulant), ...

Les actions de même priorité sont exécutées en même temps à l'exception des actions de déplacement qui sont effectuées en fonction de la rapidité à jouer des robots.

Chaque action traitée par le moteur de jeu modifie l'état du jeu.

4.2 Conception logicielle

Le diagramme des classes pour le moteur de jeu (« **engine.dia** ») est présenté en Figure 8.

Classe Engine : Cette classe est centrale et permet de stocker un Etat. Elle a accès à toutes les actions entrées par les joueurs ainsi qu'à leurs attributs. Pour chaque ordre créé par les joueurs, le moteur crée 6 commandes et les exécute action après action. Lorsque la méthode « update » est appelée, le moteur appelle la méthode « executeOrder » de chaque commande puis supprime toutes les commandes une fois exécutées.

Classes Commandes : Les classes Attack, Environment, Move et UseBonus héritent de la classe Command et réalisent une action pour un joueur. Elles possèdent chacune une méthode « executeOrder » qui fait effectuer à un personnage l'action correspondante.

Classes UseBonus, Rotation, Attack et Move : Ces classes exécutent une action et mettent à jour l'état du jeu.

4.3 Horloge globale

4.4 Conception logiciel : extension pour l'IA

4.5 Conception logiciel : extension pour la parallélisation

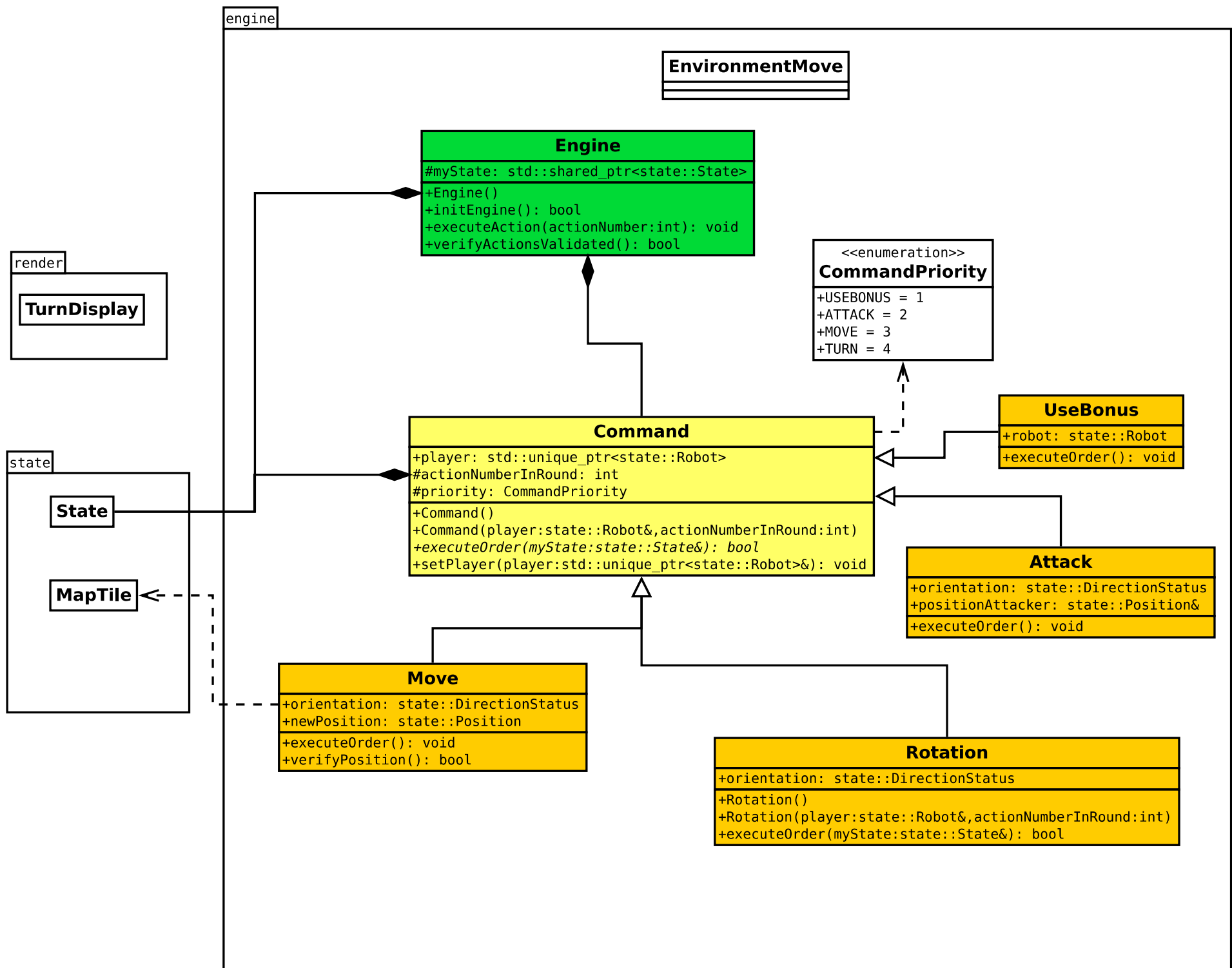


FIGURE 8 – Diagramme de classes du moteur de jeu

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 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 1: Diagramme de classes pour la modularisation

