Projet Logiciel Transversal

Gaétan RAYNAUD – Mathis OUDIN

Table des matières

1 Objectif

1.1 Présentation générale

Ce projet à pour but de la réalisation d'un jeu de type Battle for Wesnoth, avec des règles simplifiées.

1.2 Règles du jeu

Le jeu voit s'affronter deux équipes. Le jeu se déroule sur une carte découpée en cases, les cases sont réparties selon plusieurs types, ayant chacun leurs caractéristiques propres. Le jeu est découpé en tour, les joueurs jouent chacun leur tour.

Chaque équipe dispose d'une unité principale, le général, qui est l'unité principale de l'équipe. Si le général meurt, l'équipe perd la partie.

Chaque équipe commence avec une quantité d'or et peut en gagner à chaque début de tour en revendiquant une case de type maison, en plaçant une unité dessus. Si le général est situé sur une case de type château, il peut recruter d'autres unités de type épéiste, sur les cases de type caserne adjacentes.

A chaque tour, toutes les unités disposent d'une distance maximale de déplacement qui leur sont propre. Si une unité est sur une case adjacente à une unité ennemie, elle peut l'attaquer. Si une unité n'a plus de vie, elle meurt et disparaît du plateau.

1.3 Ressources

Le projet utilise plusieurs ressources pour l'affichage. Chaque texture est découpée en image de 72x72pixels.

Il y a huit textures pour les différents types de cases :

Figure 1: Texture des cases

Il y a six types d’unités, avec plusieurs animation pour les différentes actions, toutes les textures sont regroupées dans un même fichier, mais que l’on a découpé pour le mettre dans le rapport :

Figure 2: Texture du général humain

Figure 3: Texture de l'épéiste humain

Figure 4: Texture de l’archer humain

Figure 5: Texture du général orc

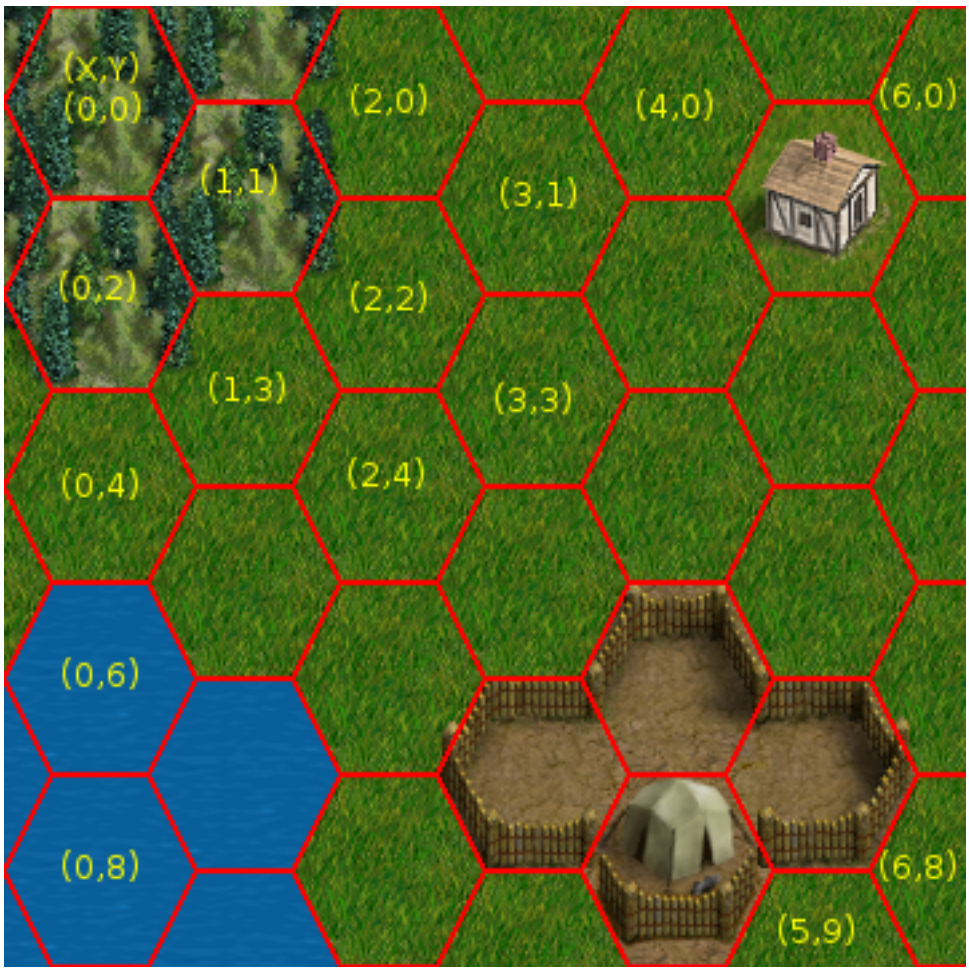
Figure 6: Texture de l'épéiste orc

Figure 7: Texture de l'archer orc

2 Description et conception des états

2.1 Description des états

Le jeu est formé d'une grille divisée en cases hexagonales avec différentes textures pour chaque case de la grille.



Sur cette grille des personnages mobiles se déplacent sur les cases à chaque tour. Les coordonnées (x, y) de chaque personnage correspondent à sa position dans la grille.

2.1.1 Éléments statiques : Terrain

La grille est formée de cases hexagonales de taille fixe. La grille est construite à partir de différentes cases possédant chacune une texture. La grille est générée au démarrage du niveau. Chaque case possède des coordonnées (x, y) représentants sa position dans la grille notée comme l'illustration ci-dessus, il ne peut y avoir qu'un seul terrain sur une position précise.
Chaque case possède un coup de déplacement pour les unités mobiles, spécifique à chaque type de case.
Les différents types de cases sont :

- **Les cases « Default » :** Les cases « Default » sont des éléments franchissables par les personnages et qui n'ont pas de propriétés particulières. Le choix du type change seulement le coût de déplacement du personnage ainsi que l'apparence de la case.
  Il existe plusieurs types de cases « Default » :
  - Les cases « Grass » classiques, avec un coût de déplacement de 1.
  - Les cases « Forest » avec une apparence de forêt et classiques, un coût de déplacement de 2.
  - Les cases « Water » qui représentent l'eau, classiques, avec un coût de déplacement de 5.
- **Les cases « House » :** Les cases « House » sont des éléments franchissables par les personnages. Lorsqu'un personnage termine son tour sur une case « House », celle-ci est revendiquée par l'équipe de l'unité. Chaque « House » revendiquée rapporte de l'or à l'équipe qui la possède, au début de son tour.
- **Les cases « Castle » :** Les cases « Castle » sont des éléments franchissables par les personnages. Si un personnage « Leader » se situe sur une case « Castle », il peut alors recruter d'autres personnages pour son équipe. Soit des « Swordman » soit des « Bowman », en contrepartie d'un prix en or.
- **Les cases « Wall » :** Les cases « Wall » sont des éléments franchissables par les personnages. Les cases « Wall » sont associées à une case « Castle ». Lorsqu'une case « Wall » est collée à une case « Castle », le personnage « Leader » qui se troue sur la case « Castle », peut recruter un personnage pour son équipe sur les cases « Wall » vides. Le « Leader » ne peut recruter de personnage que sur les cases « Wall » ce qui permet de limiter le nombre de personnages recrutés à chaque tour.

2.1.2 Éléments mobiles : Unit

Les éléments mobiles possèdent des coordonnées qui correspondent à une position sur la grille, il ne peut y avoir qu'un seul élément mobile sur une position précise. Chaque unité possède également des points de vies, une vitesse correspondant au nombre de cases qu'il reste à l'unité pour se déplacer pendant son tour, un identifiant d'équipe, et une liste d'armes.
Les armes disponibles sont une épée et un arc.
Les différents types d'unités sont :

- **Les unités « Leader » :** Les unités « Leader » sont des unités spéciales, qui sont plus résistantes et puissantes que les unités « Swordman » et « Bowman ». Elles possèdent deux armes, un arc et une épée.
- **Les unités « Swordman » :** Les unités « Swordman » sont des unités basiques, qui possèdent uniquement une épée.
- **Les unités « Bowman » :** Les unités « Bowman » sont des unités plus avancées, qui possèdent un arc et une épée.

2.1.3 Éléments complémentaires : Weapon et Team

En plus des éléments statiques et mobiles, il existe encore deux types d'éléments.

- **Les « Team » :** Les éléments « Team » représentent les équipes du jeu, elles possède plusieurs attributs :
  - « gold » qui représente l'or que possède une équipe.
  - « nbHouse » qui indique le nombre de maisons revendiquées par l'équipe et permettra de savoir combien d'or l'équipe gagnera au début de son tour.
  - « race » qui définit la race à laquelle appartient l'équipe, humaine, ou orc, qui a pour but de différencier les deux équipes visuellement, avec des apparences différentes selon la race.
- **Les « Weapon » :** Les éléments « Weapon » sont les différents types d'armes disponibles, il en existe deux, l'arc et l'épée. Elles possèdent un unique attribut, « damage » qui représente les dégâts infligés par l'arme. Et une méthode « isCloseCombat » qui permet de savoir s'il s'agit d'une arme au corps-à-corps ou à distance.

2.1.4 Éléments généraux :

À tous les éléments déjà présents s'ajoute un élément global :

- **Le « Turn » :** L'élément « turn » est un compteur qui représente le numéro du tour de jeu en cours.
- **Le « Winner » :** L'élément « winner » contient l'id de l'équipe qui a gagné la partie (l'adversaire à perdu son « Leader »).

2.2 Conception logiciel

Pour ce jeu nous avons utilisé une structure présentée par la figure 8. Ce diagramme des classes, se découpe en plusieurs parties.

- **Les classes de type d'objets « Unit », « Weapon », « Terrain » et « Team » :** Ces quatre classes sont les classes qui définissent les différents types d'objets présents dans le jeu. Il y a quatre types d'objets, les « Unit » qui représentent les personnages gérés par le joueur. Les « Weapon » sont les armes équipées sur les différents personnages, chaque personnage possède sa ou ses propres armes. Les objets du type « Terrains » sont les objets représentant les éléments statiques du décor. Enfin les « Team » représentent les équipes, et contiennent les informations relatives à celles-ci.
- **Les classes objets « Sword », « Swordman », « Castle »… :** Les classes d'objets sont des classes dérivées des types d'objets. Ce sont des types avec des éléments spécifiques, comme le « Swordman » qui est une « Unit » avec une épée du type « Weapon », qui est différent d'un arc lui aussi du type « Weapon ». On les créé grâce à un système d'héritage, qui permet de regrouper dans « Weapon » toutes les caractéristiques communes à toutes les armes. Pour ensuite faire les spécificités dans « Sword » pour la différencier de « Bow ».
- **Les classes de conteneurs « Board » et « State » :** La troisième partie contient la classe « Board » qui est une classe qui contient tous les éléments du jeu. Elle est composée, de trois listes, une pour la liste des équipes du jeu, une pour la liste des unités, et une de tous les terrains disposés sur le plateau. Ce « Board » est un tableau à doubles entrées, de taille imposée, et chaque case du tableau correspond à une case du plateau de jeu. Une case contient un terrain, et peut contenir un personnage. La classe « State » est une classe représentant un état du jeu, composé d'un plateau.

- **Clonage et comparaison :** Au sein des différentes classes qui constituent l'état, deux

méthodes ont été ajoutées :

- « clone » renvoie un pointeur vers une nouvelle instance de la classe identique à l'instance actuelle, sous-classes comprises.
- « equals» permet de comparer une instance d'une classe à une autre instance de la même classe, sous-classes comprises, et renvoie un booléen avec le résultat.

2.3 Conception logiciel : extension pour le moteur de jeu

- **Les classes d'observateurs :** Les classes d'observateurs permettent de générer des événements à partir des modifications des éléments d'un état. Les observateurs récupèrent ces événements et peuvent ainsi s'adapter aux modifications des états. Ces classes suivent la paterne « Observer ».

2.4 Chargement de la carte

Pour la génération d'un niveau du jeu, c'est à dire les différents types de cases, on a choisi de passer par un fichier. Cela permet de facilement modifier le niveau et d'en créer de nouveaux rapidement.

Pour cela, on a ajouté une méthode « LoadTerrainsFromFile » à notre class « Board », à la quelle on donne en paramètre le chemin du fichier, et va remplir sa liste de terrains en fonction du fichier et avec les bonnes coordonnées.

Pour remplir ce fichier, il faut savoir plusieurs choses. Le première est la taille en nombre de tuile du niveau, que l'on retrouve dans le « board ». On remplit le fichier avec autant de caractères que l'on veut mais seul le nombre de tuile correspondant à la taille du niveau seront chargés. Le fichier se complète avec des caractères, chaque caractère correspondant à un type de terrain.
Les types de terrains sont :

- 'G' : Herbe
- 'F' : Forêt
- 'W : Eau
- 'H' : Maison
- 'C' : Château
- 'L' : Mur gauche
- 'T' : Mur haut
- 'R' : Mur droit

2.5 Ressources

```
GGGGFFWWGGG
GGGGFFFWFGG
GTGGFFWFFFF
GLRGGFWWFHF
GCGGGFWFFFF
GGGGGGWGFFF
GGGGWWGGGGG
GGGGWWWGGGG
GGGGHWGGGGG
FFGGWWGGGGG
FFFFWGGGGGG
FFFFWWFGGTG
FHFFWFFGLRG
GGFFWFFFGCG
GGFWWFFGGGG
GGGFWWFFFGG
```

Figure 8: Exemple de fichier pour la génération d'un niveau

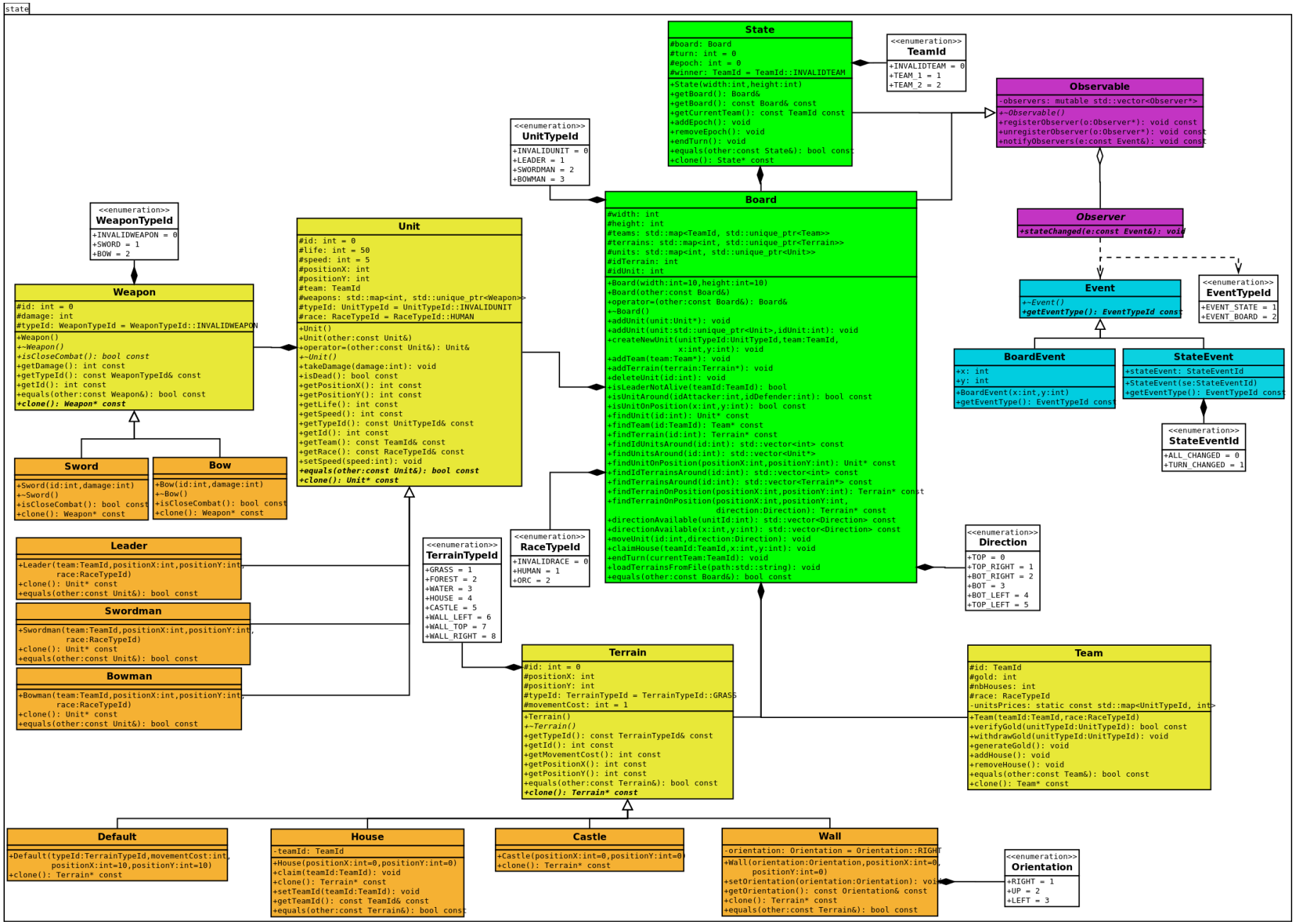


Figure 9: Diagramme des classes de l'état

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Pour notre projet nous avons opté pour un rendu d'état, le rendu ce fait à partir d'un état fixe calculé à l'avance. À chaque changement d'état le rendu est mis à jour.
Pour le rendu, nous découpons le jeu en quatre plans (layers) :

- Un plan pour le le terrain avec les différentes cases (forêt, eau, château …),
- un plan pour les unités des deux équipes,
- un plan pour les menus,
- un plan pour le « debug ».

Chaque plan est composé d'une unique texture. On récupère la texture de l'élément à afficher dans la texture du plan grâce à une tuile contenant sa taille et sa position dans l'image du plan.
Concernant la synchronisation on a une horloge pour le rendu qui est synchronisé à 30Hz. Pour l'instant, on met à jour le rendu en fonction de l'état à chaque coup d'horloge du rendu. Le but étant, lorsque l'on aura les événement de changement d'état, de ne mettre à jour le rendu qu'a chaque changement d'état. Le plan du terrain lui, n'est pas mit à jour. On le calcul au lancement du jeu, et ensuite il est toujours affiché de la même manière, car le terrain ne change pas au cours de la partie.

3.2 Conception logiciel

3.2.1 Génération des plans

La génération du rendu est découpée en plusieurs partie. La première, en jaune sur le diagramme en Figure 10, est la partie qui gère les plans.

La classe « Layer » est la classe mère qui regroupe les informations générale d'un plan. De cette classe dérivent trois autres classes, une pour chaque plan du rendu. La surface associée à chaque classe, correspond au plan qui sera affiché. La méthode « initSurface » présente dans toutes les classes « Layer » est la méthode qui permet de créer la surface qui sera affichée en fonction soit de l'état, soit du « board ».

La classe Surface, contient plusieurs méthodes, qui permettent de charger le fichier de texture du plan concerné et ensuite de faire le lien entre les coordonnées entre les textures dans le fichier et les positions des différents éléments à l'écran. Pour cela on appelle la méthode « setSpriteLocation » avec la position d'origine (le coin supérieur gauche) ainsi que la taille de la texture pour la surface la place à l'écran.

Les classes « TileSet » sont les classes qui contiennent, les coordonnées des différentes textures dans les image des plan, par exemple dans la map terrains de « TerrainTileSet », il y a les coordonnées de toutes les textures des différents types de terrains. Pour justement que la surface puisse savoir ou se trouve la texture à afficher dans le fichier d'image du plan, en ne connaissant que le type d'élément à afficher la surface fait appel aux « TileSet », elle envoie l'élément à afficher à la méthode « getTile ». Qui va lui renvoyer une « Tile » qui est un objet qui contient la position dans l'image de l'élément ainsi que sa taille, ce qui permettra à la surface de « copier » depuis le fichier à l'écran le partie de la texture correspondant à l'élément à afficher.

3.3 Exemple de rendu

Figure 10: Exemple de rendu

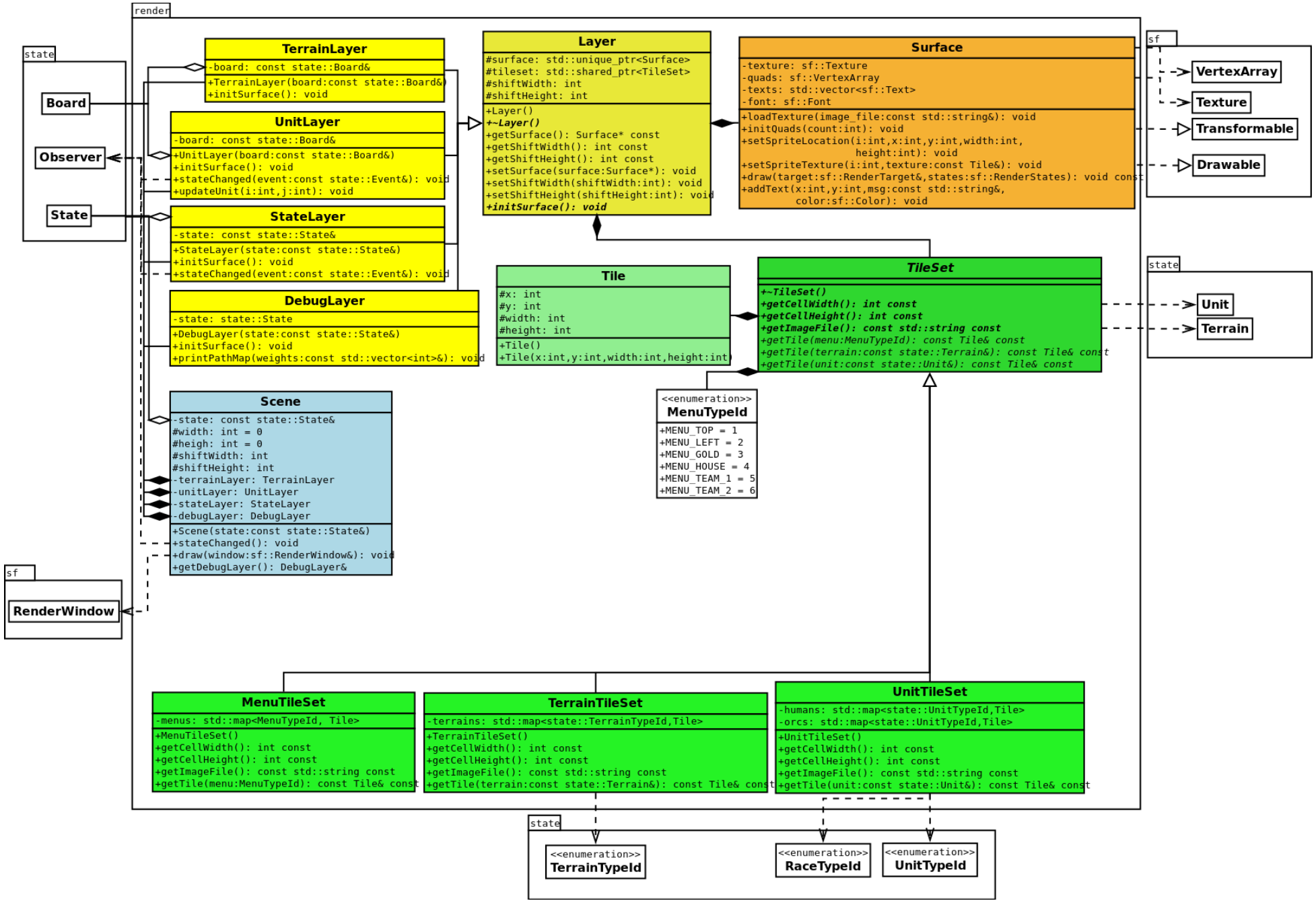


Figure 11: Diagramme des classes pour le rendu

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.

4.2 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieurs, comme la pression sur une touche ou un ordre provenant du réseau, toutes les commandes extérieurs sont :

- **« Charger un niveau » :** On fabrique un état initial à partir d'un fichier
- **« Déplacer un personnage » :** Elle prend en paramètre l'id du personnage à déplacer ainsi que les positions de la destination, et déplace le personnage vers la position indiquée
- **« Attaquer avec un personnage » :** Elle prend en paramètre l'id du personnage qui attaque et de celui attaqué, ainsi que l'arme avec laquelle le personnage attaque, et inflige les dégâts de l'arme au personnage attaqué
- **« Recruter un personnage » :** Elle prend en paramètre la position du personnage qui recrute ainsi que le type de personnage à recruté. Si le recruteur est un « Leader » et est sur une case de type « Castle », et que l'équipe a assez d'or, il instancie un nouveau personnage pour son équipe sur une case adjacente de type « Wall ».

4.3 Conception logiciel

Le moteur du jeu, est basé sur le paterne « Command », et permet d'exécuter une série d'action de manière différée.

- **Les classes « Command » :** Le rôle de ces classes est de représenter une commande, chaque classe fille représente une commande différente, que l'on repère grâce à leur « CommandTypeId » :
  - **LOAD :** Charger un niveau,
  - **MOVE :** Déplacer un personnage,
  - **ATTACK :** Attaquer avec un personnage,
  - **SPAWN :** Recruter un personnage,
  - **END_TURN :** Termine le tour du joueur,
  - **HANDLE_WIN :** Vérifie si un joueur à gagné la partie.

La commande « HandleWin » n'est pas exécutée par un joueur, mais est effectuée à chaque mise à jour après l'exécution des autres commandes.
Les commandes peuvent être exportées au format JSON, à l'aide de la méthode serialize, et peuvent être recréées à partir d'un JSON, à l'aide de la méthode deserialize.

- **Les classes « Action » :** Les classes d'action représentent des actions à effectuer pour modifier l'état. Les commandes n'affectent pas l'état mais font appel à des actions, qui elles modifieront l'état. Cela permet d'isoler les actions qui modifient un état pour pouvoir ensuite faire l'action inverse (méthode undo). Cela va notamment être utilisé dans l'IA avancée, qui va pouvoir parcourir son arbre d'issue sans avoir à sauvegarder l'état entier à chaque fois, mais uniquement les actions effectuées, pour ensuite en appliquer l'inverse.

- **Engine :** C’est la classe qui va contenir toutes les commandes dans sa « map », la « map » permet de donner un indice de priorité à chaque commande. A chaque coup de l’horloge globale, on appel la méthode « update » du moteur, qui fait elle même appelle à la méthode « execute » de chaque commande enregistrée, puis vide la liste.

4.4 Conception logiciel : extension pour la parallélisation

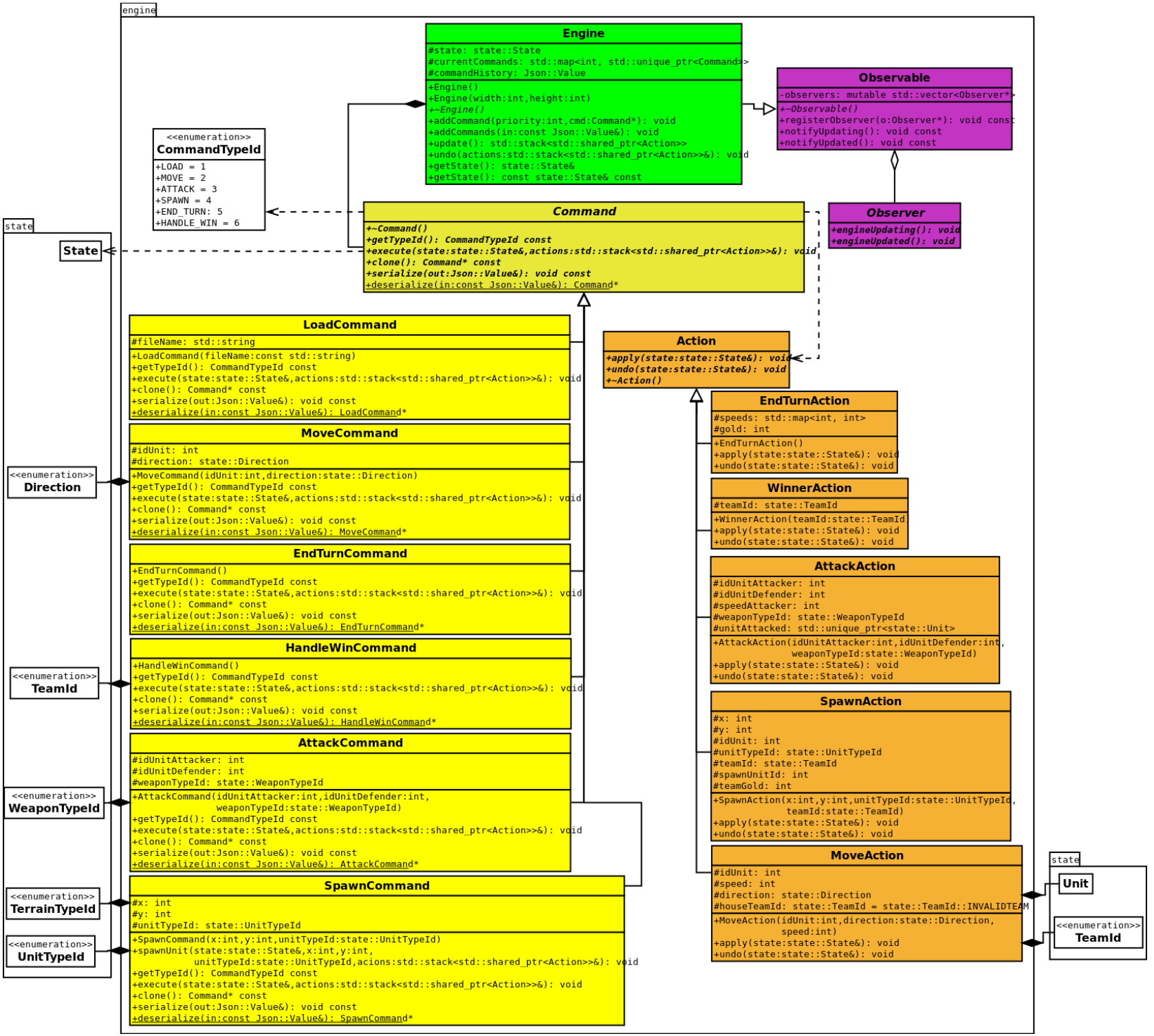


Figure 12: Diagramme des classes du moteur de jeu

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

Cette stratégie consiste à récupérer toutes les unités du joueur en cours et lister toutes les commandes possibles. Alors on choisit au hasard une commande dans la liste et l'exécute.

5.1.2 Intelligence basée sur des heuristiques

Nous implémentons une nouvelle stratégie plus avancée que le comportement aléatoire. Cette stratégie est basée sur une fonction heuristique.

- Chaque unité se dirige vers la case « House » non revendiquée ou l'unité ennemis la plus proche.
- Si une unité se trouve à coté d'une unité ennemis, elle l'attaque.
- Si un « Leader » se trouve sur une case « Castle », il recrute de nouvelles unités.

Les calculs de distance sont effectués à chaque action grâce à l'algorithme de Dijkstra.
On calcul une carte de Dijkstra pour les unités de chaque équipe et deux autres pour les cases « Houses » non revendiquées ou ennemis.

5.1.3 Intelligence basée sur les arbres de recherche

Pour une troisième intelligence artificielle, nous ajoutons, une stratégie basée sur l'exploration et la recherche dans un arbre. Un état de l'arbre de recherche, est un état du jeu à un moment donné. Les arcs entre les différents état de l'arbre sont les changements d'états générés par les commandes. Pour passer d'un état à un autre, on applique les commandes stockées par l'engine, en appelant la méthode apply de celui-ci.
Le score associé à un état du jeu, est représenté par la différence entre la somme de la vie de toutes les unités alliées, moins la somme de la vie des unités ennemies. Le but recherché est de maximiser cette valeur. Le meilleur choix est donc le chemin amenant à la plus grande différence entre la vie des alliés, moins celle des ennemies.

Pour réaliser cela, nous utilisons donc la recherche dans un arbre. Mais pour ne pas se retrouver, du fait du fort nombre de sommets possibles, à un dépassement de mémoire, on ne copie pas l'état en entier. On utilise les actions effectuées, si l'on souhaite descendre dans l'arbre, on applique les actions générées par les commandes pour passer à l'état suivant. Si l'on souhaite remonter dans l'arbre, on « undo » les actions qui ont permit d'arriver à cet état.

5.2 Conception logiciel

La classe « AI » : Les classes filles de la classe « AI » implante différentes stratégies d'IA, que l'on peut appliquer pour chaque joueur :

- **RandomAI** : Intelligence aléatoire
- **HeuristicAI** : Intelligence heuristique
- **DeepAI :** Intelligence avancée

La classe « PathMap » : La classe « PathMap » permet de calculer une carte des distances à un ou plusieurs objectifs. Pour chaque case sans unité du niveau, on obtient un poids qui représente la distance de l'objectif le plus proche. Pour s'approcher d'un objectif lorsqu'on est sur une case, il suffit de choisir la case adjacente qui a un plus petit poids. On a donc les différentes cartes de distance :

- unitTeam1PathMap : Distance par rapport aux unités de l'équipe 1.
- unitTeam2PathMap : Distance par rapport aux unités de l'équipe 2.
- houseTeam1PathMap : Distance par rapport aux maisons de l'équipe 1 et non revendiquées.
- houseTeam2PathMap : Distance par rapport aux maisons de l'équipe 2 et non revendiquées.

La classe « DeepAI » : La classe « DeepAi » représente l'IA avancée. Pour la résolution du problème, on utilise l'algorithme Alpha/Beta, une amélioration du minimax. Le but étant pour le joueur de maximiser la différence entre la somme de la vie de ses unités et la somme de la vie des unités adverses, et l'adversaire doit lui minimiser cette différence. Le jeu état assez complexe, et permettant à chaque état jusqu'à 6 actions par unité alliée sur le plateau, il est indispensable de limiter le nombre de recherches possibles. Pour cela, nous avons limité le nombre de commandes testées à un état ainsi que la recherche de profondeur dans l'arbre.

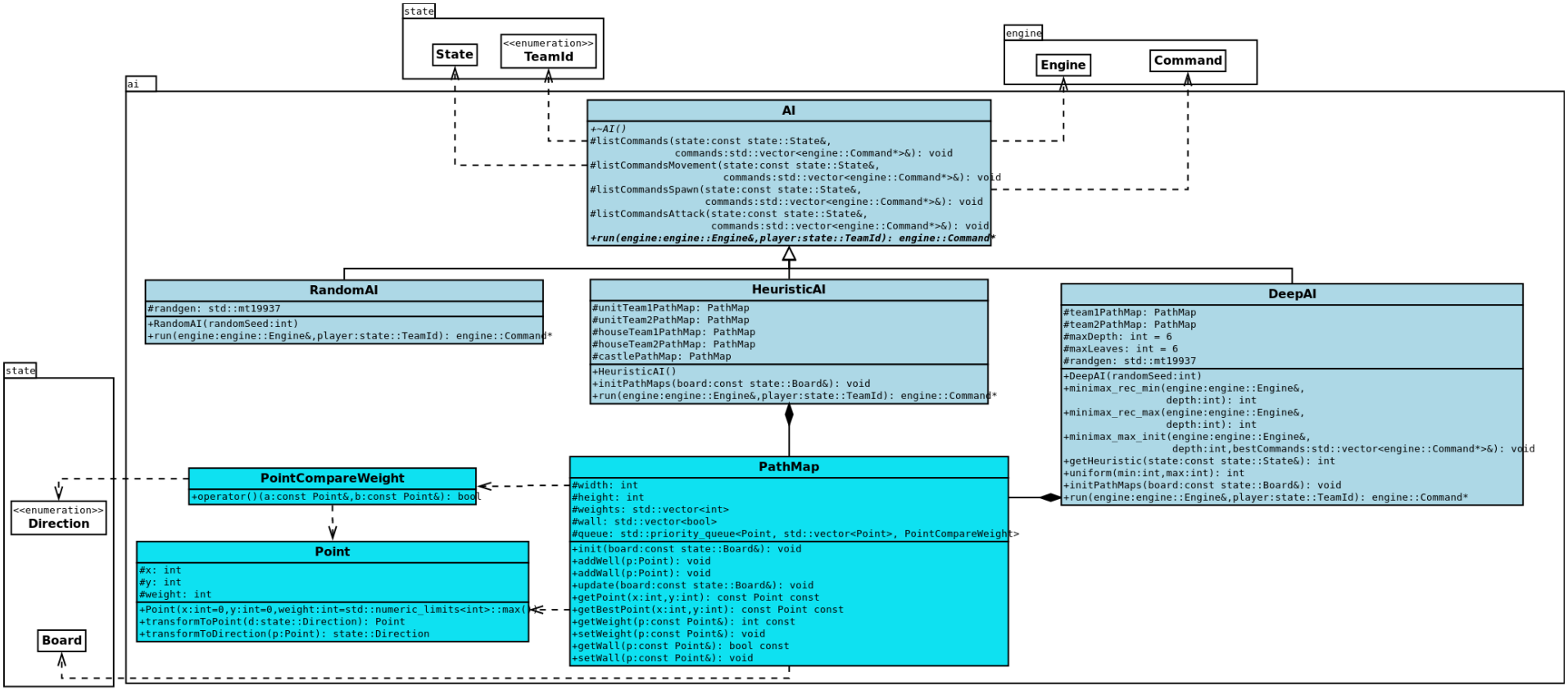


Figure 13: Diagramme des classes d'intelligence artificielle.

6 Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

Notre objectif ici est de placer le moteur de jeu sur un thread, puis le moteur de rendu sur un autre thread. Le moteur de rendu est nécessairement sur le thread principale (contraire matérielle), et le moteur du jeu est sur un thread secondaire. La modification de l'état du jeu est protégée à l'aide d'un mutex, pour ne pas modifier, et lire l'état en même temps. Une variable de condition transite entre les deux threads, pour les synchroniser.

6.1.2 Répartition sur différentes machines : rassemblement des joueurs

La première étape pour pouvoir jouer en réseau est la création d'une liste de client pour le serveur. Pour ce faire, nous formons des services CRUD sur la donnée "joueur" via une API Web REST :

Requête GET /player/<id>	
Pas de données en entrée	
Cas joueur <id> existe	Statut OK Données sortie : type: "object", properties: { "name": { type:string }, }, required: ["name"]

Cas <id> négatif	Statut OK
	Données sortie :
	
	type: "array",
	items: {
	type: "object",
	properties: {
	"name": { type:string },
	},
	},
	required: ["name"]

Cas joueur <id> n'existe pas	Statut NOT_FOUND Pas de données de sortie

Requête POST /player

Données en entrée :

```
type: "object",
properties: {
    "name": { type:string },
},
required: [ "name" ]
```

Cas il reste une place libre	Statut CREATED Données sortie : type: "object", properties: { "id": { type:number }, }, required: ["id"]
Cas plus de place libre	Statut OUT_OF_RESOURCES Pas de données de sortie

Requête PUT /player/<id>

Données en entrée :

```
type: "object",
properties: {
    "name": { type:string },
},
required: [ "name" ]
```

Cas joueur <id> existe	Statut NO_CONTENT Pas de données de sortie
Cas joueur <id> n'existe pas	Statut NOT_FOUND Pas de données de sortie

Requête DELETE /player/<id>

Pas de données en entrée

Cas joueur <id> existe	Statut NO_CONTENT Pas de données de sortie
Cas joueur <id> n'existe pas	Statut NOT_FOUND Pas de données de sortie

6.1.3 Répartition sur différentes machines : échange des commandes

Pour la gestion des commandes, tous les clients envoient leurs commandes moteur au serveur. Ils n'exécutent pas directement ces commandes, mais demandent de manière régulière au serveur si il y a de nouvelles commandes pour la prochaine époque. Si c'est le cas, les clients exécute ces commandes, puis attendent à nouveau les prochaines. Nous implantons cette stratégie via un service web CR sur la donnée "commandes par époque" : on ne peut qu'ajouter/créer un lot de commandes par époques, et consulter n'importe quel lot pour toutes les époques passées.

Requête GET /commands/<epoch>	
Pas de données en entrée	
Cas époque <epoch> existe	Statut OK Données sortie : liste des commandes sérialisées en JSON
Cas époque <epoch> n'existe pas	Statut NOT_FOUND Pas de données de sortie

Requête POST /commands	

Données en entrée :
 	
type: "array",
items: {
type: "object",
properties: {
"priority": { type:number },
"command": commande sérialisée
},
required: ["priority", "command"]	

Tous les cas	Statut CREATED Données sortie : type: "object", properties: { "epoch": { type:number }, }, required: ["epoch"]

6.1.4 Répartition sur différentes machines : statut de la partie

Nous avons également ajouté un service web qui permet d'obtenir le statut de la partie : CREATING (en cours de création), RUNNING (partie en cours). Celui-ci permet aux clients de savoir s'il peuvent rejoindre la partie, ou si la partie est déjà en cours.

6.2 Conception logiciel

La classe « NetworkClient » : La classe NetworkClient contient les informations pour faire fonctionner le jeu : Moteur de jeu (avec état intégré) et rendu. Elle ne contient une IA que pour le joueur qu'elle dirige. Elle commence par se connecter au serveur, et attends ensuite que la partie soit pleine, si au bout de une minute, la partie n'est pas pleine, le client s'arrête. Si la partie démarre, le client affiche le jeu et à chaque frame rendue, il demande au serveur si des commandes pour la prochaine époque sont disponibles. Si c'est le cas, il les exécute, puis reprend son rendu à 60 images par seconde. Si le serveur ne répond pas ou indique un statut différent que RUNNING, le client s'arrête.

La classe « Game » : La classe game et les classes associées représentent les éléments d'une partie. En premier lieu, nous trouvons la liste des joueurs, présent dans la partie. On peut ajouter, retirer ou modifier un joueur à tout moment.
On peut aussi demander à tout moment demander l'état de la partie au serveur à l'aide de « getGameStatus » Elle contient également un moteur de jeu, exécuté dans un thread séparé (le thread principal est réservé au HTTP). Au lancement de la partie, on attend que le nombre joueur requis pour lancer une partie soient présents, puis on démarre la partie. Une fois la partie démarrée, le moteur de jeu tourne en boucle, et ne fait aucune notification, se sont les clients qui demandent s'il y a eu des changements. Il ne s'arrête que lorsque la partie est terminée ou qu'un jouer a quitté la partie. Lorsqu'une commande à été ajoutée au serveur par un client, le serveur va ajouter une commande « HandleWinCommand », afin de vérifier si un jouer a gagné la partie, la commande sera exécutée par tous les clients. Pour être capable de donner les commandes, nous utilisons un système d'enregistrement qui mémorise les commandes en JSON. Ainsi, elles peuvent très facilement être délivrées, il suffit de prendre l'élément dans le tableau JSON correspondant à l'époque désirée.

La classe « Services » : Les services REST sont implantés via les classes filles de AbstractService, et gérés par la classe ServiceManager :

- **VersionService** : le traditionnel service qui renvoie la version actuelle de l'API. Indispensable dans toute API pour prévenir les conflits de version.
- **PlayerService** : fournit les services CRUD pour la ressource joueur. Permet d'ajouter, modifier, consulter et supprimer des joueurs.
- **CommandsService :** fournit les services CR pour la ressource "commandes par époque". Permet d'ajouter, et consulter les lots de commandes.
- **GameService :** fournit un service qui permet de connaître l'état de la partie.

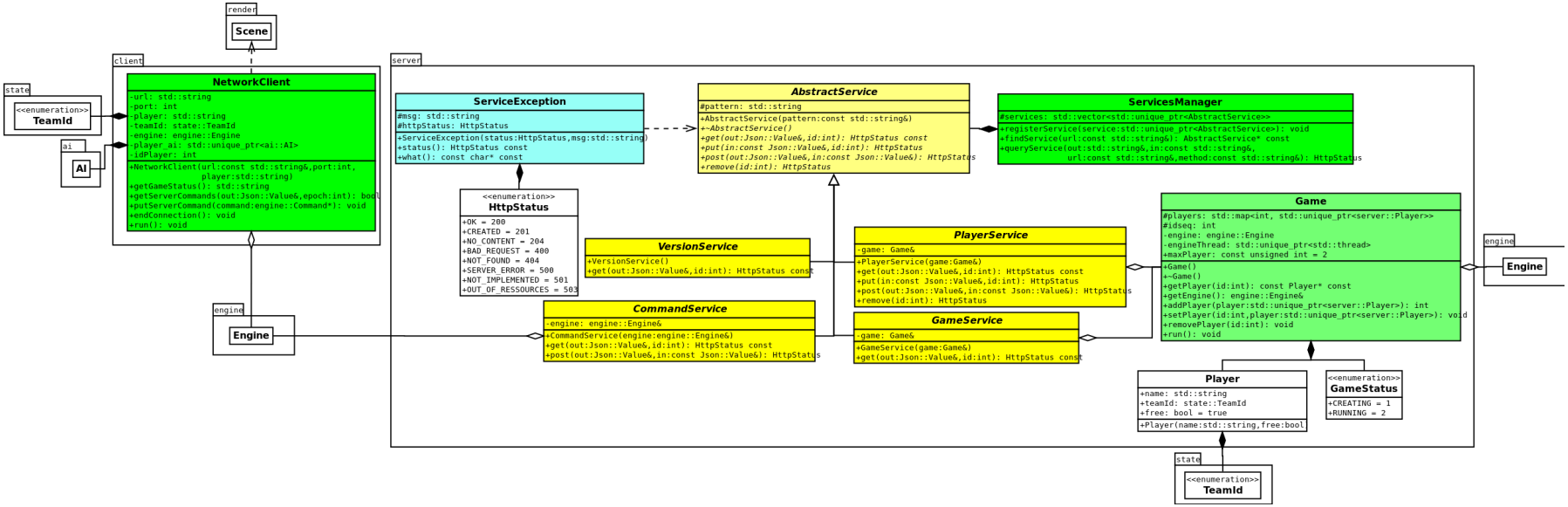


Figure 14: Diagramme des classes pour la modularisation.