Projet Logiciel Transversal

Fodus

Table des Matières

Table des Illustrations

I Objectif

I.A *Présentation générale*

L'objectif de ce projet est la réalisation d'un jeu type « tactical RPG » (système de combat similaire à Dofus).

Figure 1 : Dofus

Dofus est doté de 2 mécanismes de jeu : la phase d'exploration, et la phase de combat. Nous allons nous inspirer de la phase de combat, avec une phase d'exploration propre à notre jeu et son univers.

I.B Règles du jeu

Contexte

Le joueur incarne le stagiaire du nécromancien d'un donjon : il lui a confié la tâche de nettoyer les salles du donjon après le passage d'aventuriers.

Le joueur va donc affronter des monstres survivants, ainsi que potentiellement des aventuriers perdus. Mais étant un stagiaire, il ne peut combattre directement et il doit invoquer des serviteurs mort-vivants pour combattre à sa place ; Il peut également corrompre les monstres et les aventuriers, qui changeront de camp et combattrons pour lui : mais certains aventuriers pourront également corrompre des monstres pour les rallier à leur camp.

Le jeu sera constitué d'une première phase dite d'exploration. Dans cette phase du jeu, le joueur sélectionne une zone de jeu sur laquelle combattre.

Lorsqu'une zone de combat est choisie, le jeu passe en mode combat : il s'agit là d'un combat au tour par tour, où le joueur affronte les monstres et/ou les aventuriers de la zone. Le joueur se déplace comme tout autre personnage (monstres et aventuriers), mais ne peut les attaquer (mais peut se faire attaquer) : il ne peut que corrompre les autres personnages pour les rallier à son camp et invoquer des minions pour l'aider. Un combat est gagné quand tous les monstres sont vaincus, un combat est perdu quand le personnage principal du joueur est mort.

Le jeu possède un système d'expérience où le joueur sera récompensé à la fin de ses combats victorieux ; il pourra gagner de niveau après un certain nombre de points d'expériences, et se verra gratifier d'amélioration de caractéristiques et de compétences.

I.C Conception Logiciel

II Description et conception des états

II.A Description des états

Le jeu se divise en deux grandes parties : la partie exploration et la partie combat. La partie exploration consiste en une carte générale du donjon où l'on peut se déplacer de section en section. Une suite de mission demande au personnage de se rendre successivement dans différentes zones indiquées. Une fois arrivé dans ces zones le combat se lance, et l'on passe à la seconde phase du jeu.

Le combat se passe dans une zone divisée en cases sur laquelle se trouvent le personnage principal et un certain nombre d'ennemis. L'objectif de chaque combat est de tuer tous les ennemis présents, tandis qu'une défaite à lieu si le personnage du joueur meurt. Les commandes se font au tour par tour, d'abord le personnage du joueur, puis les ennemis. Chaque tour le joueur peut effectuer un déplacement puis une action, de même pour chaque adversaire.

Partie combat :

Etat :

Représente l'état du combat à un instant précis. On y trouve ainsi une liste de tous les éléments du combat, que ce soit des éléments fixes comme des cases ou des personnages mobiles. On y trouve également la taille de la zone de combat.

Les éléments "case" et "personnage" possèdent tous les deux les attributs "posX" et "posY" représentant leur position sur la grille de jeu.

Les cases :

Les différentes cases constituent la zone de jeu. Chaque case possède des attributs indiquant si elle est : un mur ou une case libre, un attribut indiquant si elle est occupée ou non, et d'un attribut indiquant si elle est piégée.

Elles possèdent également l'attribut "départ", qui correspond au fait que la case soit ou non une case de départ pour le personnage ou un ennemi.

Les personnages :

Regroupe tous les personnages du jeu, que ce soient les zombies, le personnage principal ou les ennemis. Tous ces personnages ont les mêmes propriétés, qui sont :

- "santeMax", représentant le total de points de vie.
- "sante", représentant la somme actuelle de points de vie.
- "deplacement", correspondant au nombre de case dont le personnage peut se déplacer chaque tour.
- "classe", représentant pour les aventuriers la classe à laquelle ils appartiennent. Les classes possibles sont : Guerrier, Magicien, Prêtre, Archer.

- “niveau”, représente le niveau et donc la puissance globale du personnage.
- “status”, qui correspond à l’état du personnage : bien, poison, étourdi, ralenti, ou une combinaison quelconque.
- “corruption”, qui indique combien de fois le personnage à été corrompu.
- “competences”, qui liste les compétences auxquelles le personnageaaccès.
- “limiteZombie” qui indique quel est le nombre maximal de Morts-vivants pouvant être invoqué à la fois par le personnage.
- “defense”, “puissance” : représentant des caractéristiques du personnage influençant respectivement les dégâts reçus et infligés.
- “effets” liste les effets qui affectent en permanence le personnage.
- “direction” indique dans quelle direction le personnage est dirigé

Les compétences : utilisables par les personnages, les compétences possèdent un certain nombre d’attributs les définissant :

- “degats” correspond au dégâts infligés par la compétence (ou soin)
- “postee” indique à combien de case peut être utilisées la compétence
- “zone” représente la zone qui sera affectée par la compétence (0 pour une case unique par exemple)
- “cible” représente quel type de cible peut être affecté par la compétence : les ennemis, les alliés ou les deux.
- “effets” liste les effets supplémentaires de la compétence

Partie exploration :

Carte :

Représente la carte du jeu dans cette phase. Le personnage s'y déplace pour se rendre dans les différentes zones, ainsi que des groupes d'aventuriers qui peuvent être affrontés par le joueur. Cet élément liste les différentes zones du jeu, les différents groupes d'aventuriers et le personnage.

Zone :

Correspond aux différentes zones où le joueur peut se rendre. Chacune d'entre elle peut potentiellement donner lieu à un combat contre les monstres du donjon, ou contre des

aventuriers si un groupe se trouve sur la même zone que le joueur. Chaque zone possède plusieurs attributs :

- "occupation" correspond au fait que la zone est occupée ou non, que ce soit pas un groupe d'aventurier ou par le joueur.
- "difficulte" représente la puissance des monstres présents dans la zone.
- "position" correspond à la position de la zone sur la carte.

Personnage :

Ne concerne ici que le personnage joueur. Ces différents attributs sont :

- “sante” qui est la sante totale du personnage
- “defense”, “puissance” sont des caractéristiques du personnage comme précédemment
- “Déplacement" exprime le nombre de case que peut parcourir le personnage en combat.
- “niveau” correspondant au niveau du personnage, et donc à sa puissance totale
- “expérience” correspond au nombre de points d’expérience engrangé par le personnage et permettant de passer au niveau supérieur.
- “posX” et “posY” représentent les coordonnées du personnage sur la carte.
- “inventaire”correspond à l’inventaire du personnage
- "emplacement" représente l'endroit où se trouve le personnage dans la carte.

Inventaire :

Regroupe tous les objets que le personnage a pu obtenir et ceux qui sont en cours d’utilisation.

- “equipe” liste tous les objets actuellement équipés sur le personnage
- “sac” liste tous les objets en possession du joueur.
- “limite” fixe une limite au nombre d’objet transportables par le joueur
- “or” correspond à l’argent accumulé par le joueur au cours du jeu.

Objet : représente tous les objets pouvant être utilisés par le joueur. Ils sont définis selon les attributs suivants :

- “type” représente la catégorie de l’objet, et donc l’emplacement où il sera équipé.

- “modificateur” représente l’efficacité de l’objet, et donc son influence sur les caractéristiques du personnage.
- “prix” représente la quantité d’or que le personnage peut obtenir en revendant l’objet.

Aventuriers :

Chacun de ces objets correspond à un groupe d'aventuriers se déplaçant dans le donjon et risquant de rencontrer le personnage et donc de l'affronter. Ils sont visibles sur la carte par le joueur, et sont décrits de la manière suivante :

- "niveau" représente le niveau global des aventuriers du groupe, et donc de la difficulté du combat en cas d'affrontement.
- "emplacement" correspond à la zone dans laquelle se trouve le groupe à un instant donné.

II.B Conception logiciel

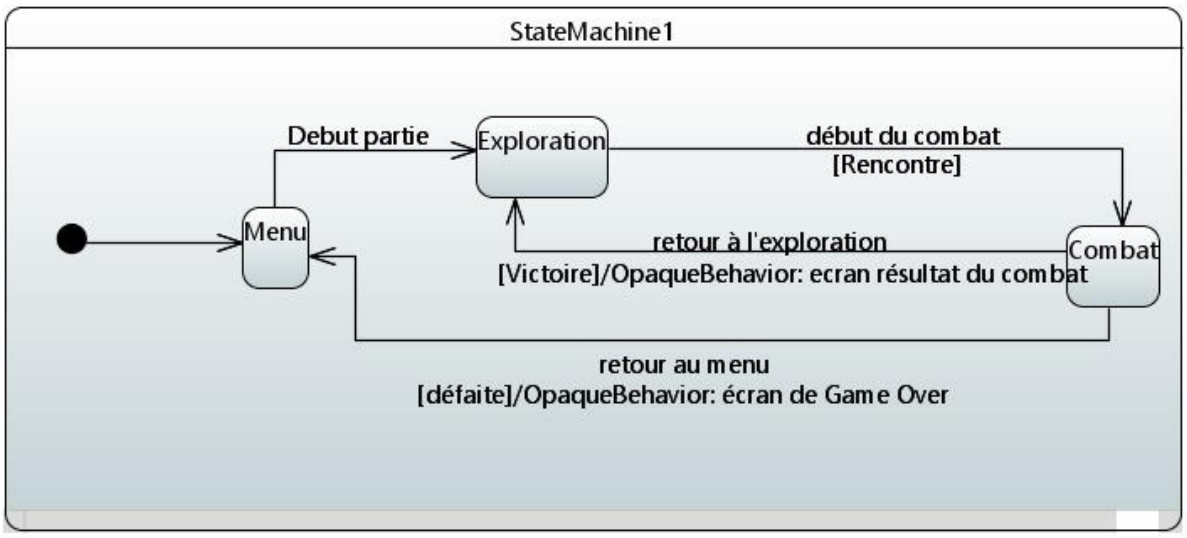


Figure 2 : Machine à état décrivant le déroulement du jeu

Représentation du workflow du jeu et des transitions entre différentes phases.

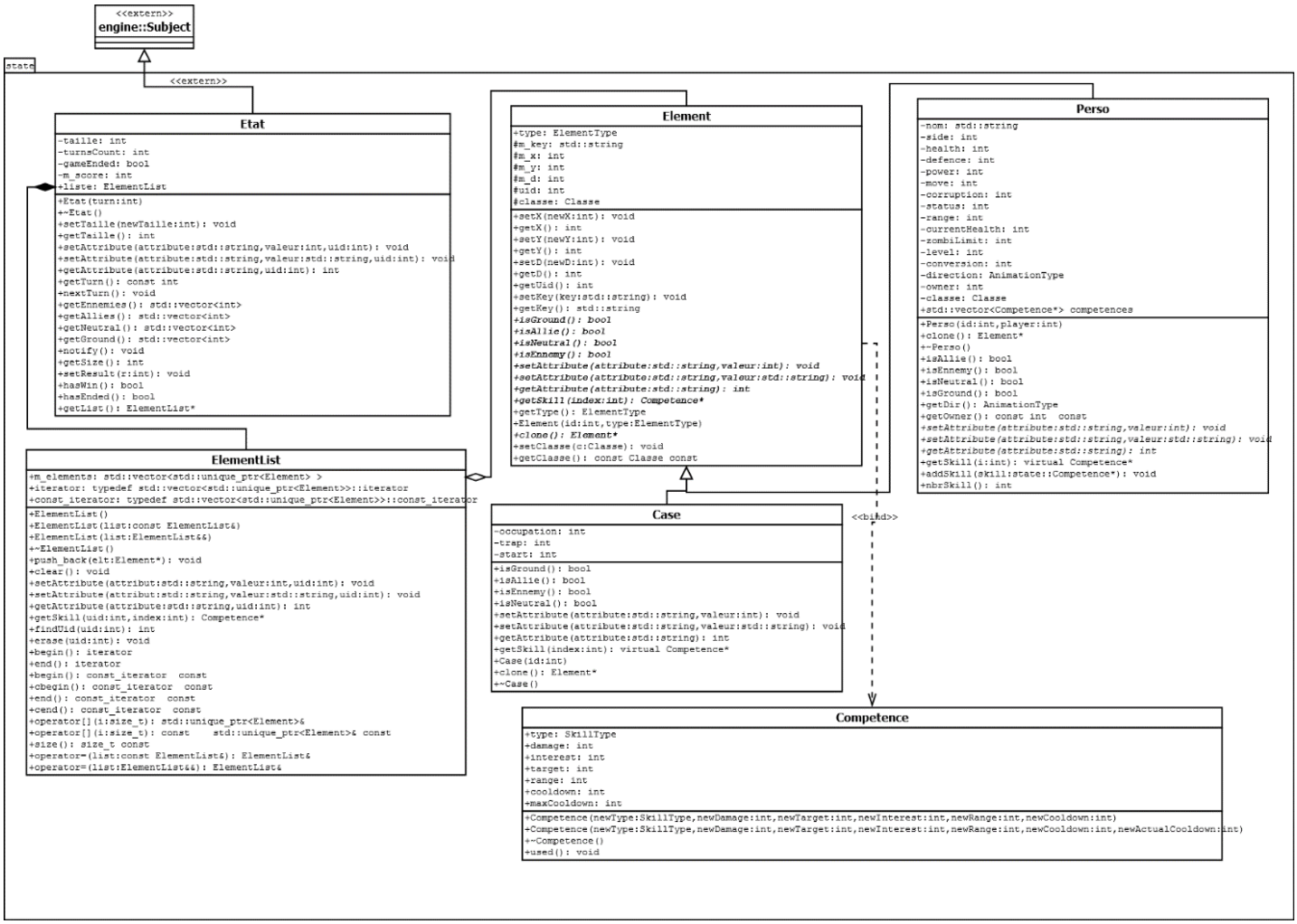


Figure 3 : Diagramme de classe d'état pour la phase de combat

Représentation des différentes classes utilisées lors de la phase de combat et des différents attributs de celles-ci, ainsi que des différents liens existants entre elles.

II.C Conception logiciel : extension pour le rendu

II.D Conception logiciel : extension pour le moteur de jeu

II.E Ressources

L'ensemble des ressources (autre que les fichiers images et sons) est codé en JSON, ce qui inclus le fichier de déclaration des tiles, et les fichiers de niveau (ainsi que tout fichier de configuration). Celle permet de n'avoir aucun élément (ou presque) hardcodé dans le code du jeu, et d'offrir une souplesse de développement.

Exemple : fichier de Tiles, qui permet de générer les tiles en fonction des éléments


```
"header": {
"version": "0.2",
    "name": "Default Tileset"
},
  "tiles": {
"UNDEFINED": {
      "y": 0,
      "x": 0
    },
    "VOID": {
"y": 0,
"x": 2
}
```


Exemple : Fichier d'un niveau


```
{
    "header": {
"version": "0.0",
"name": "My Little Level",
"desc": "Misc"
    },
    "level": [
        [
            [
                {
                    "key": "WALL_DUNGEON_TOP_LEFT"
                },
                {
                    "key": "VOID"
                }
            ],
            [
                {
                    "key": "WALL_DUNGEON_TOP_MID"
},
                {
                    "key": "VOID"
                }
            ],
            ...
```

],

[...],

...

]

}

III Rendu : Stratégie et Conception

III.A Stratégie de rendu d'un état

Pour rendre un état, nous avons regroupé tout le rendu en Scène, qui est l'image de l'état du jeu, et nous l'avons découpée en plans. Ces plans se répartissent différents types de données : un plan pour les informations, 2 plans pour les élément fixes (2 niveaux de profondeur), un pour les éléments mobiles (personnages), et un plan de debug. Chaque plan contient une texture et une matrice donnant la position des éléments dans la fenêtre de jeu, et les coordonnées de sa partie de la texture. Ces 2 éléments sont envoyés à la carte graphique ce qui permet un affichage rapide et optimisé des éléments (plutôt que de tracer sprite par sprite)

Actuellement, seuls les éléments fixes sont été implémentés, pas les mobiles ni les informations.

III.B Conception logiciel

Le diagramme des classes pour le rendu est en [Figure 4 : Diagramme de classe du moteur de rend].

Utilisant la bibliothèque SFML, nous avons choisi d'implémenter l'interface sf ::drawable à nos objets à rendre : scene et ses layers. Cette implémentation permet de simplifier les appels graphiques, puisqu'il nous suffira de faire un App.draw(scene) dans la boucle principale pour rendre l'intégralité des plans.

Layer

Un layer (plan) est un objet réalisant le lien entre une liste d'éléments et leur position spatiale dans la fenêtre ainsi que leur représentation (texture). Layer hérite de sf ::drawable :

un appel à App.draw(layer) permet de dessiner tout le layer. Dans le cas de l'ElementLayer (fixe ou mobile), le layer possède une propriété de profondeur (pour gérer plusieurs couches de tiles, ex. des décorations sur un mur), et une TileFactory, qui s'occupe de gérer la correspondance Element – Texture. Une fonction update() est déclenchée par l'Observer de l'état du jeu pour mettre à jour les éléments du plan.

Scene

Elle contient tous les plans, et son implémentation de draw() appelle les draw() de ses layers.

Elle possède une fonction update(ElementList), qui est appelée lors d'une modification de l'état (pattern Observer), et qui de façon analogue à draw() appelle les update(ElementList) de ses layers.

Elle s'occupe également de la gestion des animations en affichant les sprites animés, et en s'occupant des frames et de leur horloge.

TileFactory

Cette Factory (pattern Factory donc) génère une tileà partir d'une clef d'identification. Ces clefs sont les identifiants des éléments (ex. FLOOR_WOOD_1), ce qui permet de trouver l'emplacement de leur texture via le chargement préalable du fichier tiles.json. Ce fichier est l'index de toutes les clefs d'éléments avec leurs coordonnées de texture associés : il n'y a pas de données en dur dans le code.

Tile

Cet objet est simplement une représentation de la clef et les informations de texture associée.

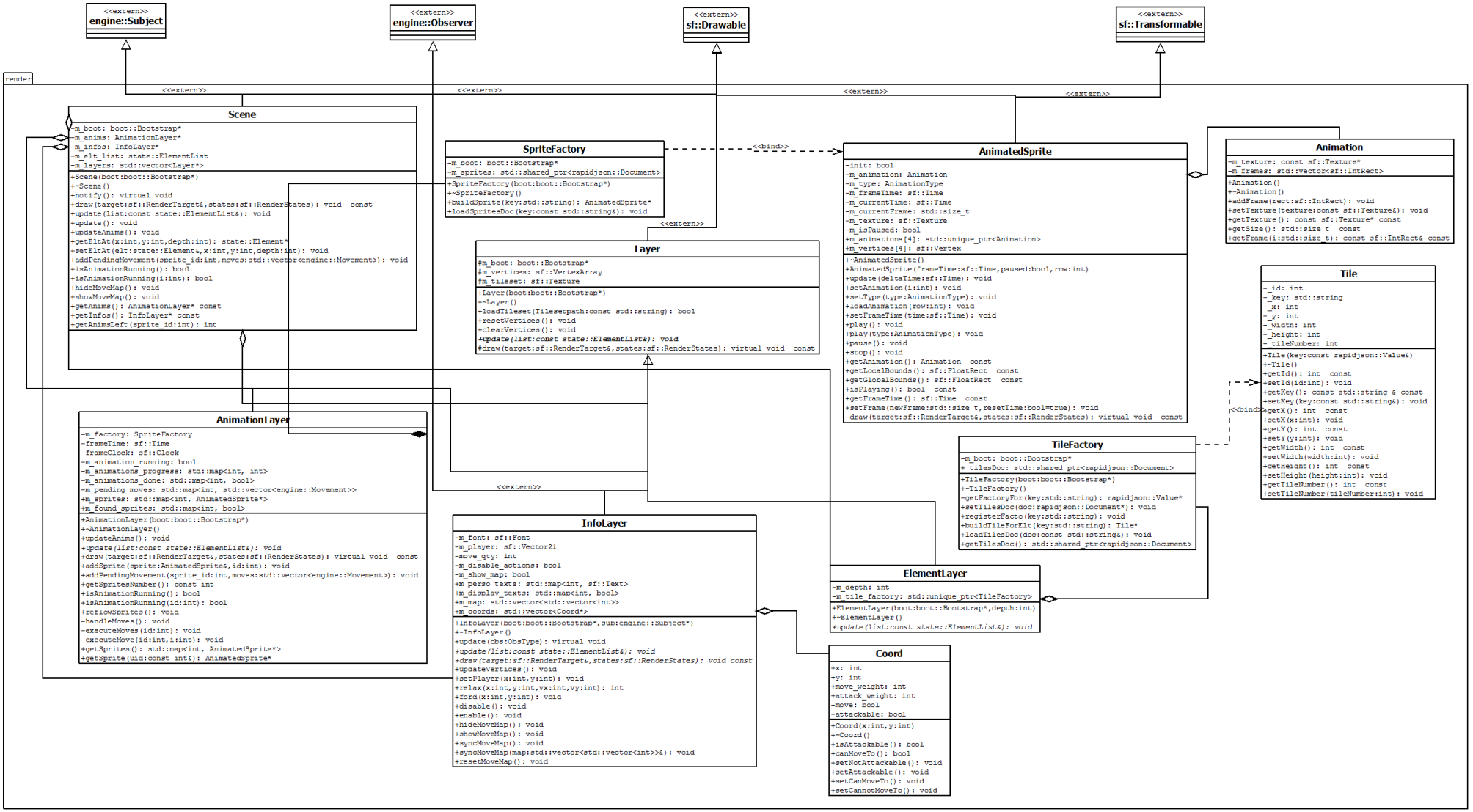


Figure 4 : Diagramme de classe du moteur de rend

III.C Conception logiciel : extension pour les animations

Les animations sont pour le moment gérées dans la classe Scene, mais seront encapsulés dans une classe dédiée dans une prochaine version.

Lors du parcours de l'ElementList si un Mobile Element (Perso) est détecté, un sprite correspondant est alors ajouté. Ensuite, lors des updates, le moteur de rendu analyse la liste des mouvements des éléments mobiles, et ajoute une liste de déplacements le cas échéant dans la scène. Celle-ci prend alors en compte l'intégralité du cycle de l'animation du sprite : direction, vitesse, frame, horloge, etc. Les mouvements d'une case à une case sont coupés en 8 frames pour avoir un déplacement fluide tout en n'updatant pas trop régulièrement le sprite. La gestion des animations est un Work in Progress, mais un POC est disponible.

III.D Ressources

Afin de gérer les ressources JSON, 2 programmes ont été réalisés : un script Python 3 permettant d'ajouter des tiles à l'index, et un éditeur graphique de niveau.

III.E Conception logiciel : éditeur

L'éditeur est pleinement intégré au jeu et peut être choisi au démarrage du programme.

Il contient 2 fenêtres : celle de droite permet de sélectionner une tile à appliquer, et celle à gauche est une vue du niveau en cours de modification, et est géré par une unique classe Editeur qui regroupe l'ensemble des fonctions, et ne déborde donc pas dans les fichiers « classiques » du reste du jeu.

Il contient la liste des éléments du niveau en cours (chargée depuis son JSON), la liste des tiles possibles (chargées depuis l'index des tiles), et permet de modifier les tiles du niveau (une par une ou plusieurs par drag and drop). La sauvegarde est faite automatiquement à la sortie de l'éditeur.

III.F Exemple de rendu

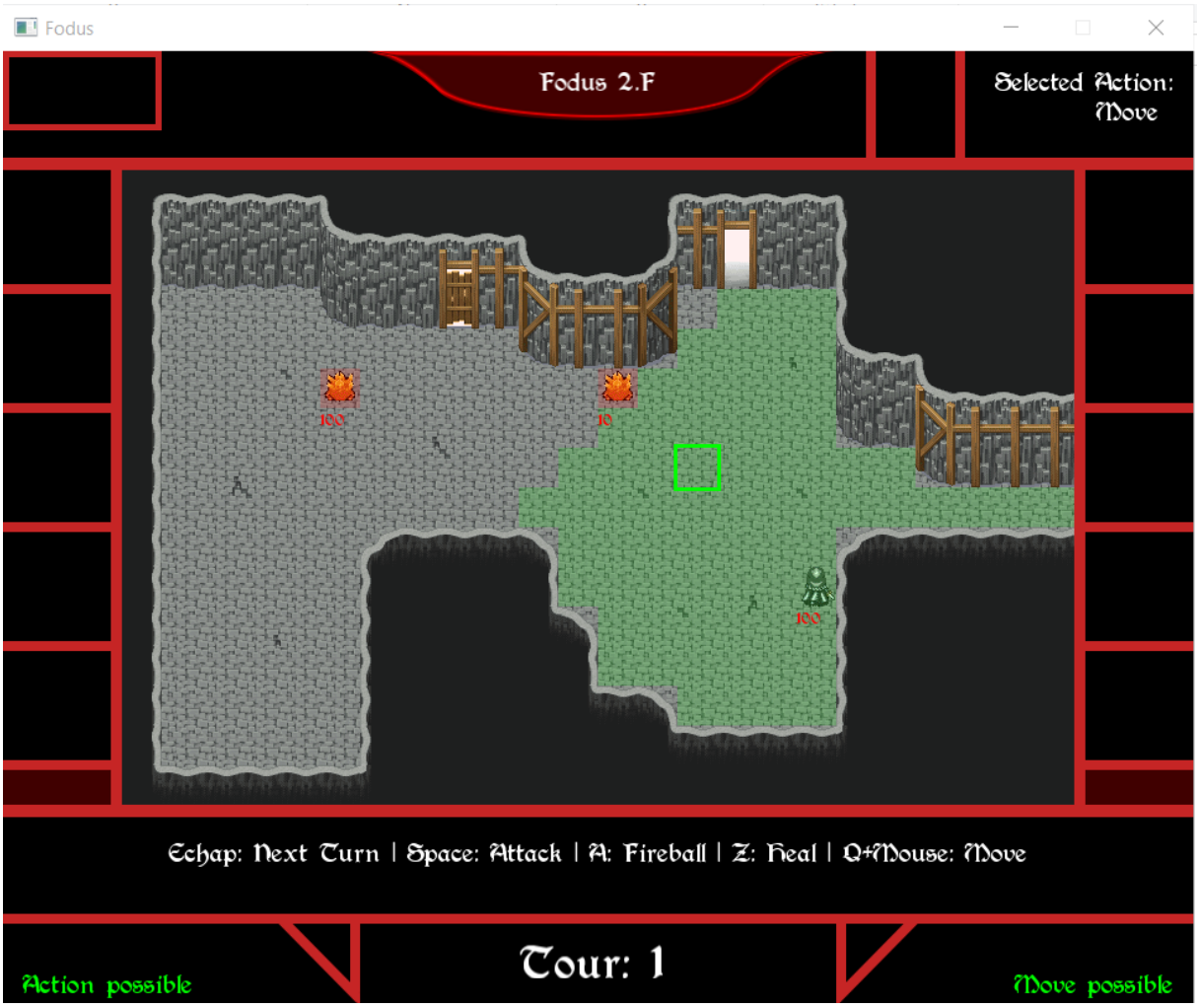


Figure 5 : Exemple de rendu

IV Règles de changement d'états et moteur de jeu

IV.A Horloge globale

Le jeu est un jeu à états finis : le jeu est rythmé par une horloge globale qui cadence l'exécutions des actions et donc les mises à jour de l'état.

IV.B Changements extérieurs

Les changements extérieurs sont des commandes exécutées par le moteur de jeu et provenant d'entrées utilisateur (ou du module IA par la suite) :

Commandes globales :

- Mode de démarrage : Editeur ou jeu
- Nouvelle partie, charger niveau, quitter
- …

Commandes d'action :

- Déplacement
- Attaque
- Magie
- …

IV.C Changements autonomes

Les changements autonomes sont un jeu de commandes automatiques exécutées lors de la mise à jour de l'état. Elles ne requièrent pas de commande d'entrée.

Il s'agit typiquement :

- Conditions de défaite : le personnage joueur n'a plus de vie (santé = 0)
- Vérifier la santé d'un personnage : si la santé est à 0, le détruire
- Conditions de victoire : il n'y a plus d'ennemis
- Appliquer la liste d'actions en attente
- …

IV.D Conception logiciel

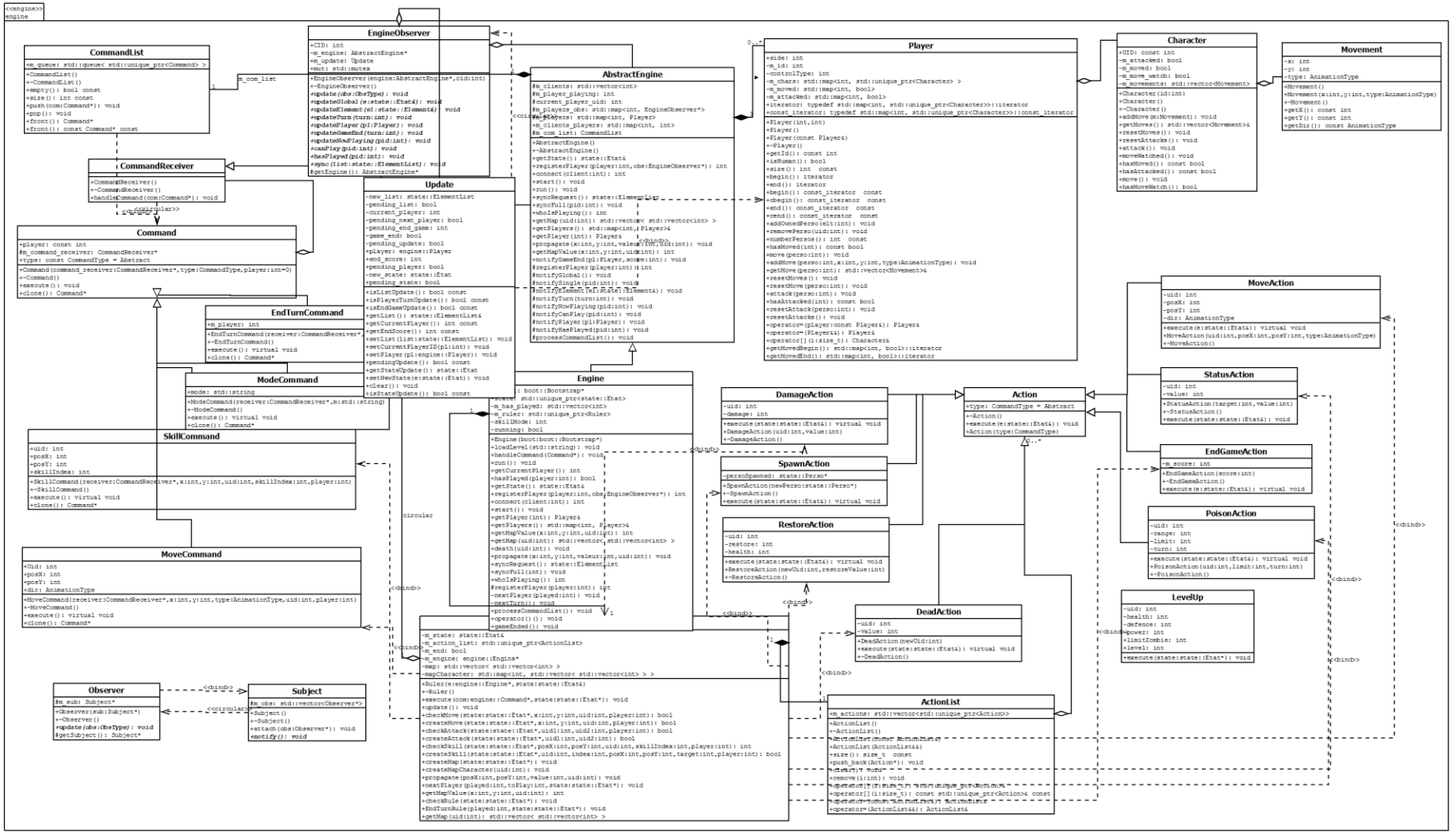


Figure 6 : Diagramme des classes du moteur de jeu

Le moteur du jeu repose sur le Pattern Command et sur l'exécution des commandes en différé, grâce à leur stockage dans une liste de commandes.

Classes Command :

Interface de commande suivant le Pattern Command. Elle implémente un attribut (objet de classe CommandReceiver), un attribut CommandType (enum) et une méthode abstraite execute(), qui appelle la fonction handleCommand(Command*) de l'objet Receiver.

Les différentes implémentations (ModeCommand, MoveCommand…) sont les différents types de commandes, avec leurs attributs associés.

Exemples :

MoveCommand : position X, position Y, UID (identifiant de l'objet se déplaçant), direction

ModeCommand : type de mode (editor / game)

Classes CommandList :

Conteneur personnalité faisant office de wrapper autour d'une std::queue< std::unique_ptr<Command> >

Cette classe sert de liste pour les commandes en attente de traitement par l'engine. Cette classe nécessite d'utiliser les mutex pour y accéder afin d'éviter les surprises pour le multithread.

Classe CommandReceiver :

Interface de receveur suivant le Pattern Command. Elle implémente une méthode abstraite handleCommand(Command*) qui est appelé pour le traitement des commandes.

Classe Engine :

Classe au cœur du moteur du jeu. Son rôle est de stocker les commandes dans la CommandList qui seront exécutées lorsqu'une mise à jour de l'état sera lancée (ie quand update() sera appelé à un rythme régulier). La classe est dans une boucle infinie pour traiter les commandes de la CommandList (avec un sleep bref entre chaque fois).

Les commandes donnant lieu à des actions en jeu sont envoyées au Ruler pour traitement, tandis que les autres peuvent donner lieu à des actions directes dans le moteur (ex. pause, quitter, chargement…) et ne sont pas soumises aux règles du jeu.

Afin de préparer la suite du projet, une interface AbstractEngine commence à être implémentée.

Classe Ruler :

Cette classe vérifie les commandes en fonction des règles du jeu qu'elle reçoit et en réalise un dynamiccast en fonction de leur type. Elle vérifie alors la faisabilité de la commande, et le cas échéant crée une Action qui sera appliquée sur l'état pour le modifier. C'est également elle qui gère les commandes autonomes.

Classe Action :

Cette interface représente une action validée par le Ruler, issue d'une commande externe ou de changements autonomes.

Exemple :

MoveAction : position X, position Y, UID (identifiant de l'objet se déplaçant), direction

Classe Player :

Cette classe tient à jour la liste des joueurs de la partie, et de leurs personnages associés. Il s'agit d'une méta donnée pure qui peut être reconstruite à partir de l'état du jeu.

Classe Character :

Cette classe tient à jour des métadonnées sur chaque personnage en jeu par tour : attaque effectuée, mouvement effectué, liste de déplacement. Ces métadonnées sont réinitialisées à chaque début de tour.

IV.E Conception logiciel : extension pour l'IA

IV.F Conception logiciel : extension pour la parallélisation

V Intelligence Artificielle

V.A Stratégies

V.A.1 Intelligence minimale

L'objectif de cette première intelligence artificielle très simple consiste à avoir une simulation d'un joueur qui ne joue pas de manière aléatoire, mais rien de plus. Les règles retenues sont très simples :

-Dans le cas où un ennemi est suffisamment proche pour pouvoir être attaqué, on se déplace à portée et on attaque. Cela permet de tenter de gagner la partie en éliminant tous les adversaires. Dans le cas de plusieurs ennemis présents, on en choisit un au hasard.

-Si aucun ennemi n'est à portée, alors on se rapproche (si possible au plus près) d'un ennemi au hasard.

Cela devrait permettre d'avoir un comportement plus cohérent que le hasard pur (sans mouvement erratique), mais ne permet pas encore de "jouer" de manière efficace.

V.A.2 Intelligence basée sur des heuristiques

L'IA commence à implémenter un comportement à l'approche heuristique : elle est dotée de différents comportements (pour le moment Fear et Aggressive) sélectionnés en fonction de critères. Elle incorpore un comportement pour un personnage indépendamment des autres : elle peut par exemple contrôler 1 personnage en Fear et 2 en Aggressive de manière découplée.

Si (points de vie) < (points de vie max)/2 Alors comportement = Fear

Sinon comportement = Aggressive

Ce comportement va alors influencer les actions de l'IA pour le personnage :

Si Fear Alors Fuite (= maximiser la distance entre ce personnage et l'adversaire le plus proche)

Si Aggressive Alors Minimiser la distance entre ce personnage et l'adversaire le plus proche et attaquer (= comportement basique de l'intelligence minimale)

Ces comportements sont mis à jour à chaque tour du personnage concerné ; donc si la santé d'un personnage passe sous la barre de la moitié de sa valeur maximale le comportement sera fixé sur Fear jusqu'à ce que le personnage soit soigné suffisamment.

Lorsque l'IA se retrouve bloquée, elle réalise des actions aléatoires pour la conduire à se débloquer naturellement. Elle prend en compte uniquement le camp de personnage le plus proche, et non pas le fait que le personnage soit le personnage principal comme c'était le cas au tout début. Par exemple maintenant en fixant les deux adversaires contrôlés par l'IA avec des camps différents les deux personnages vont se battre entre eux, sans se préoccuper du joueur tant qu'il n'est pas trop proche.

L'IA implémente l'utilisation pleine des capacités disponible : elle peut se soigner et attaquer.

V.A.3 Intelligence basée sur les arbres de recherche

V.B Conception logiciel

Classe AIPlayer :

L'IA implémente l'interface IGame : cela lui permet de se comporter de façon rigoureusement identique au client de jeu d'un joueur humain : données accessibles du moteur de jeu, fonctionnement… l'interface avec le moteur de jeu fourni les cartes des distances (utilisé notamment dans le moteur de rendu pour l'affichage des déplacements).

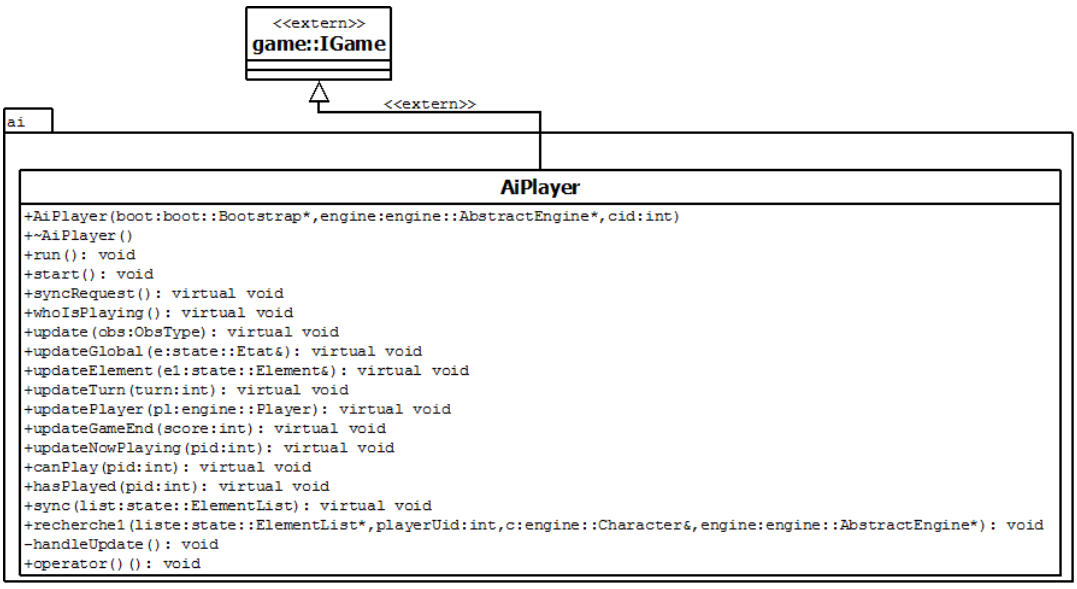


Figure 7 : Diagramme des classes de l'IA

Par conséquent, l'utilisation d'une IA est « plug'n'play » : l'IA n'a qu'à s'enregistrer auprès du moteur de jeu en tant que client pour un joueur X, et tout le fonctionnement classique d'un joueur sera assuré.

```
engine::Engine engine(this);
game::Game game(this,&engine, rand());
ai::AiPlayer aiP(this,&engine, rand());
```

```
engine.loadLevel(level);
game.start();
aiP.start();
engine.start();
game.run();
```

V.C Conception logiciel : extension pour l'IA composée

V.D Conception logiciel : extension pour IA avancée

V.E Conception logiciel : extension pour la parallélisation

VI Modularisation

VI.A Génération automatique des headers

Afin que les diagrammes UML soient conformes à l'implémentation réelle dans le programme, tous les headers sont extraits depuis les diagrammes UML qui sont convertis en headers C++.

Pour se faire, le programme dia2code s'est révélé insuffisant pour la tâche au vue de ses lacunes. Pour y palier, Timothé a réalisé un script Python 3 pour convertir les UML Dia en header C++ : pydia2code. Il réalise 2 transformations du XML du diagramme DIA pour le convertir en headers C++. Ce programme étant fait maison, il implémente les fonctionnalités dont nous avons besoin : namespaces, un fichier par classe, références circulaires avec forward declaration, inclusions de fichiers externes…

La configuration du générateur se fait en 2 étapes (nous sommes allés au plus rapide étant donné le temps que nous avions) :

- Les dépendances circulaires sont notifiées dans le fichier Dia par le nom / stereotype « circular » : le parser ajoutera alors une forward declaration pour éviter une dépendance circulaire.
- Les dépendances externes sont symbolisées par un stéréotype « extern » (Classes, Association, Dépendance…) : le parser les ignore.
- Les inclusions externes doivent être rajoutés dans le config.yml de pydia2code pour que le générateur les ajoute au header cible

Le script est réalisé en Python 3 et utilise les lib LXML et PyYAML (testé fonctionnel sur les CentOS de l'école). Code disponible sous GPL à https://github.com/AchilleAsh/pydia2code .

VI.B Organisation des modules

VI.B.1 Logger

Afin d'avoir un débogage plus efficace et une meilleure traçabilité des bugs et exécutions, le logiciel implémente la LibraryEasyLogging++, qui met à dispositions des macros permettant d'afficher dans la console et simultanément d'enregistrer dans le fichier global.log (même dossier que l'exécutable).

VI.B.2 Amorceur du logiciel

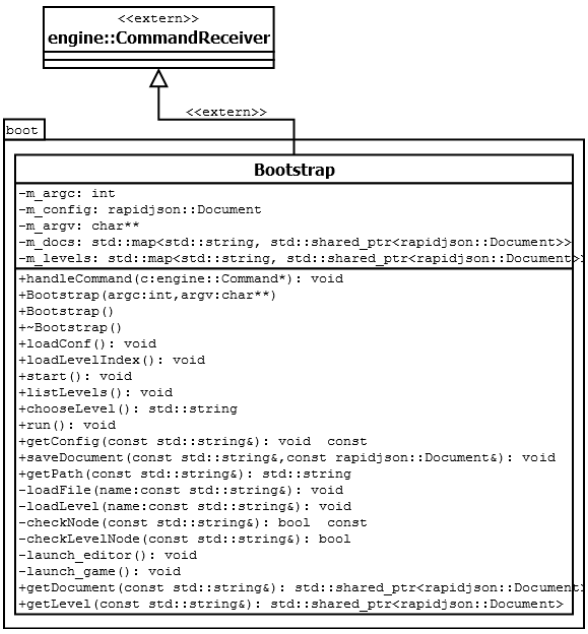


Figure 8 : Diagramme des classes de boot

Afin d'obtenir une organisation robuste et efficace, une classe d'amorce Bootstrap a été conçue. Lors de l'exécution du main(), celle-ci instancie le Bootstrap, l'initialise avec la fonction start() puis la lance avec run().

Cette fonction run() permet de choisir le type de mode à lancer : éditeur ou jeu, ce qui instancie un objet Game ou Editor selon le choix et le lance.

Le Bootstrap réalise une couche d'abstraction entre les autres classes et par exemple les fichiers de configurations et de ressources, ceci afin de découpler au maximum toutes les classes du programme. Les classes ne savent même pas qu'ils manipulent des fichiers et ont directement accès au contenu (parsé si fichier JSON) ou au lien du fichier ressource : aucune configuration n'est écrite en dur dans le code.

Exemple :

Game : chargement du niveau « test » : appelle de bootstrap ::loadLevel(« test ») => Bootstrap : chargement depuis le disque de « res/levels/ »+ «test»+ « .json » , puis parsing du document JSON chargé, stockage dans une std ::map [« test »] et renvoie un shared_ptr sur ce membre de la map.

VI.B.3 Client du jeu

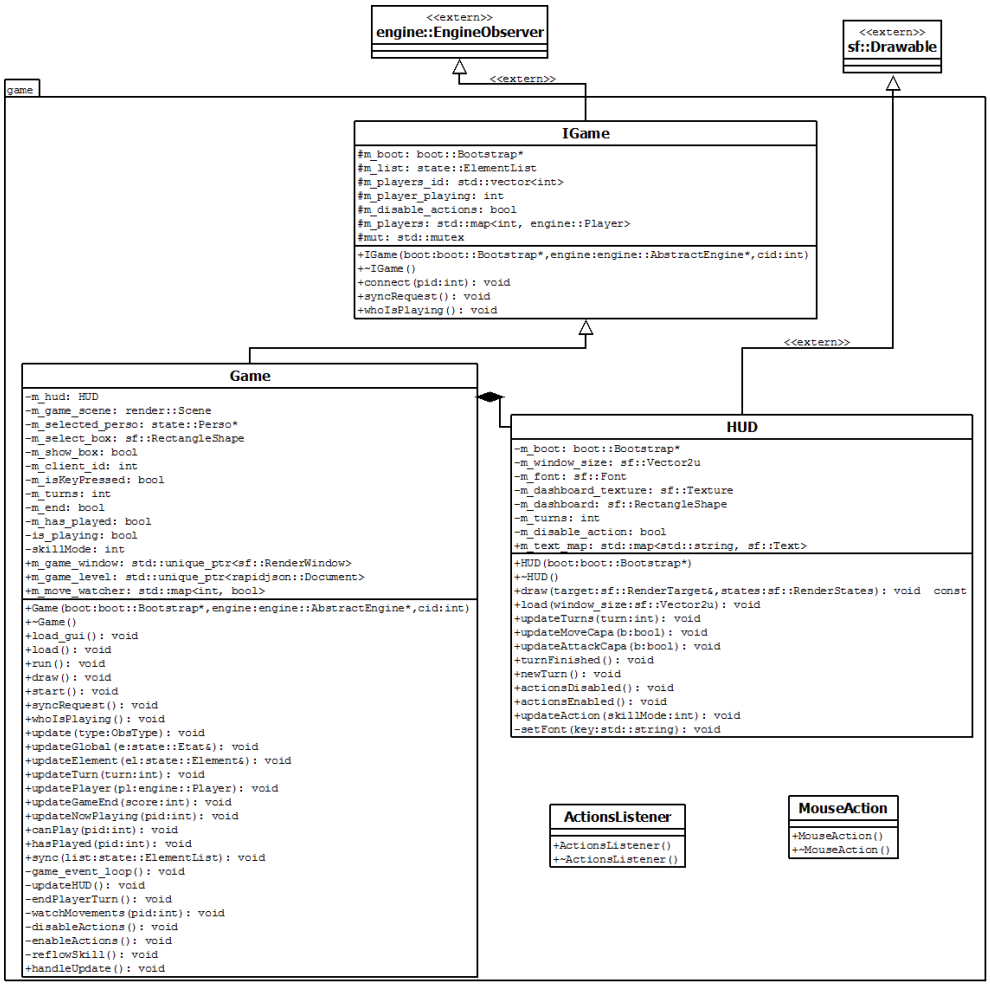


Figure 9 : Diagramme des classes de game

L'ensemble IHM-Moteur de rendu est encapsulé dans une classe Game, qui hérite de l'interface IGame qui hérite elle-même de l'interface EngineObserver.

L'interface EngineObserver répond au design pattern Observer, tandis que IGame est une implémentation d'EngineObserver, avec un nombre de méthodes supplémentaires abstraites. Le but est de proposer une API standardisée de communication entre le moteur de jeu et un client quelconque : réseau, local, ou IA.

L'implémentation d'IA est rendue aisée et modulaire, et totalement autonome. l'API actuelle n'exploite pas toutes les possibilités de l'architecture, mais permet une

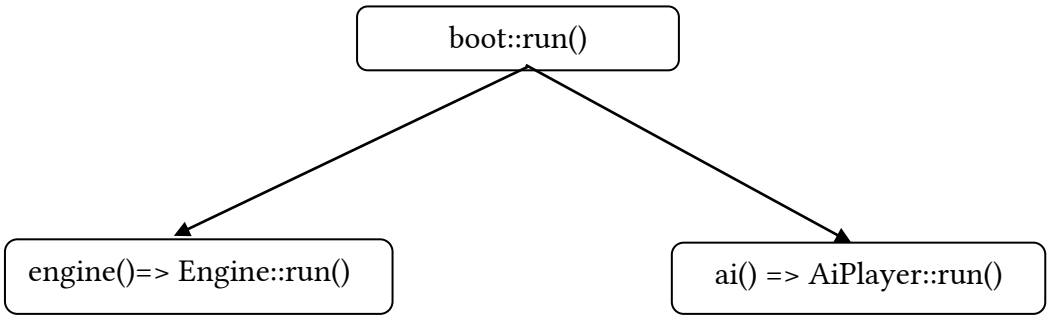
rétrocompatibilité avec les précédentes bouts de code qui n'exploitent pas encore les fonctions de l'API (exemple : un update global de toute l'ElementList est encore réalisée à chaque update). Game stocke une copie de l'ElementList et de la liste des joueurs et effectue l'analyse des mouvements pour en extraire les animations à effectuer.

En sus de tout le moteur de rendu, Game possède une classe HUD. Encore sous exploitée, elle inclue toutes les éléments en dehors de la scène du jeu (boutons, textes) et est responsable de leur mise à jour. Sera également inclus dans Game une classe ActionsListener, permettant d'enregistrer des callbacks à effectuer sur action d'une zone de jeu (souris) ou une touche (clavier).

VI.B.4 Répartition sur différents threads

Nous avons opté pour l'implémentation C++11 des threads de par sa facilité.

Le programme démarre dans son thread principal avec le Bootstrap, qui va lancer 2 autres threads lors du lancement du jeu :



Engine et AiPlayer sont des « functor » : ils surchargent l'opérateur () ce qui permet de les manipuler comme des fonctions avec les threads.

les run() de AiPlayer et Engine sont des boucles infinies avec un sleep, plus des opérations à réaliser lorsque des mises à jour sont présentes (nouvelles commandes pour l'engine, et nouvelles mise à jour pour l'AiPlayer).

Pour l'envoi des commandes, elles sont mises après un lock de mutex dans la CommandList (une file de Commands) de l'engine, qui les réalisera lors de son éveil.

A noter : le thread principal avec l'interface graphique reste le plus gourmand en occupation CPU : 90% pour ce thread, 10% pour les 2 autres (source : outil de profiling Visual Studio)

VI.B.5 Répartition sur différentes machines

VI.C Conception logiciel : extension réseau

VI.C.1 Introduction

L'extension réseau du jeu s'effectue simplement :

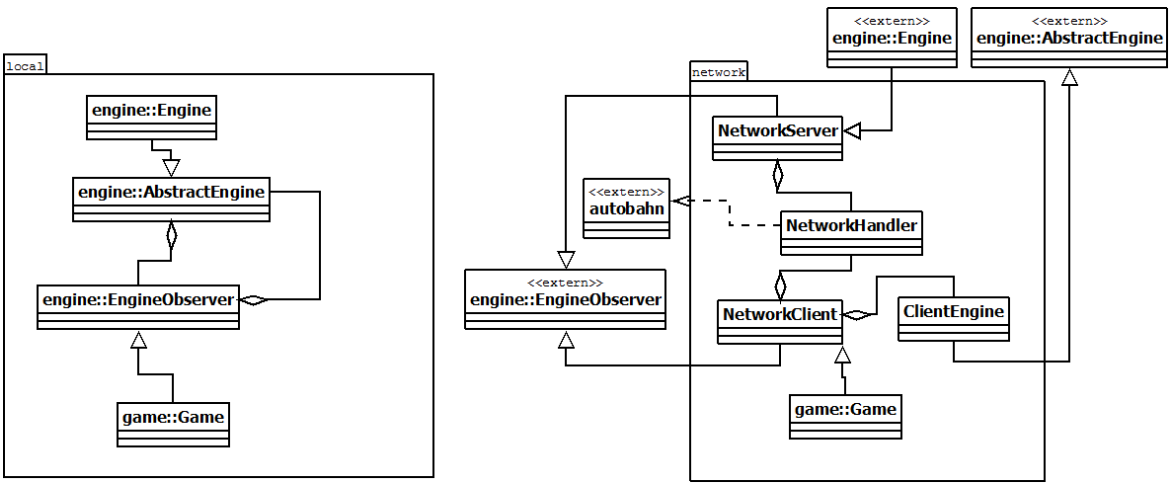


Figure 10 : Schéma de principe

A gauche la version « locale » précédente, à droite l'implémentation réseau.

Pour réaliser la communication réseau, la classe EngineObserver (qui était la classe centrale liant l'engine et les clients) est scindée en 2 parties, dont l'une sera dans le client et l'autre dans le serveur. Il est également nécessaire d'adapter l'Engine, (utilisé pour les commandes par exemple).

Plusieurs patterns Adapter sont impliqués :

- ClientEngine qui adapte les getEngine() réalisé par le client du jeu
- NetworkClient qui adapte Game pour le réseau
- NetworkServer qui adapte Engine pour le réseau

VI.C.2 Stratégie Réseau

VI.C.2.a Cahier des charges

Pattern classique	Pattern réseau
Pattern Observer	Pub/Sub
Pattern Command	RPC

La solution réseau doit mettre en œuvre les RPC (Remote Procedure Call) et le système Pub/Sub pour pouvoir remplacer notre conception en quasi drop-in (moyennant des classes Adapter).

VI.C.2.b Choix

Nous avons choisi une solution basée sur le protocole WAMP, qui implémente notamment les RPC et le Pub/Sub.

WAMP est un protocole de communication temps réel principalement basé sur le protocole Websocket (canaux de communication full-duplex via TCP) mais peut utiliser d'autres transports : raw TCP, pipes, sockets UNIX. Initialement prévu pour les ressources Web pour navigateur internet en réponse à l'explosion des usages « Web 2.0 », la stack WAMP est implémentée dans la plupart des langages, en particulier C++ et peut donc être généralisé pour l'utilisation entre applications comme ici.

WAMP implémente les RPC (Remote Procedure Call) et le Pub/Sub (système d'abonnés - publication de contenu)

Les RPC de WAMP offrent les avantages suivants :

- Asynchrone et performant
- Paramètres et retour en JSON
- Agnostique du langage
- Fonctionne en local ou par internet

Le Pub/Sub est un pattern de publication – souscription de messages (inspiré du paradigme des queues de messages) où ces derniers sont émis dans les canaux mais en ne sachant pas les éventuels destinataires. De manière réciproque un abonné à un canal n'a pas connaissance (hormis si indiqué dans le contenu du message) de l'origine des messages publiés sur un canal.

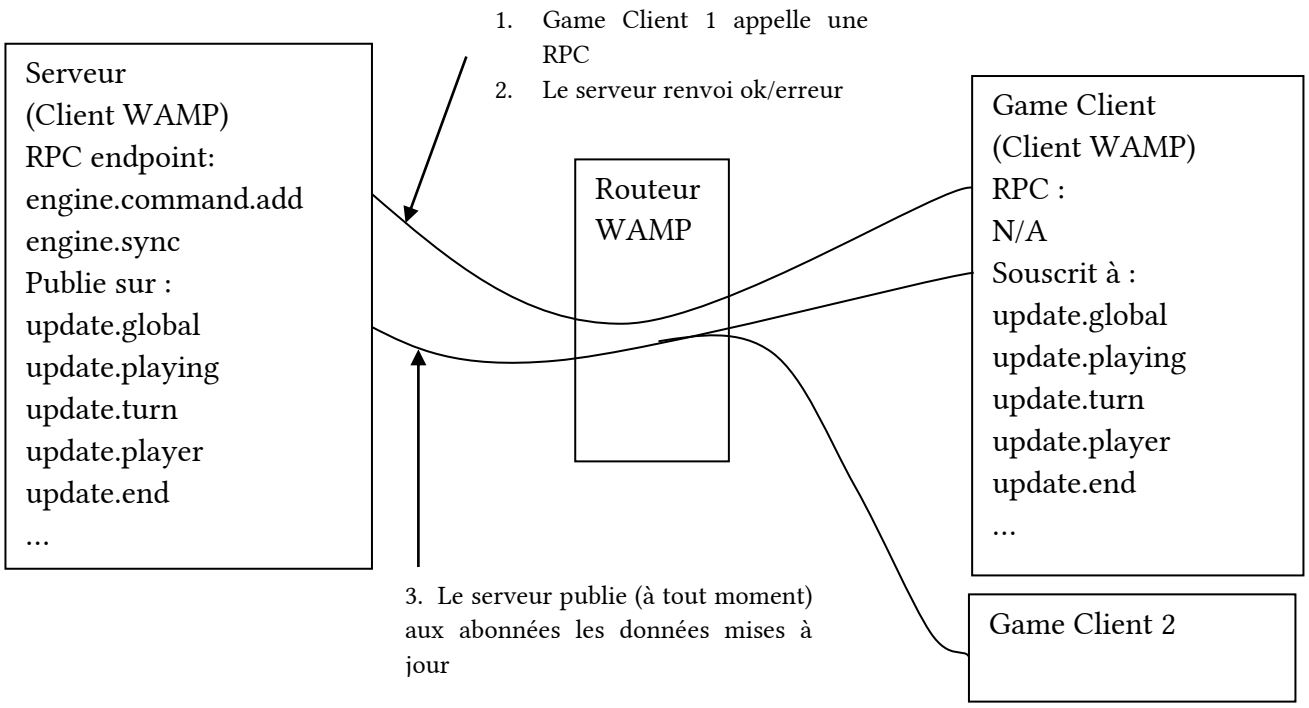
VI.C.2.c Architecture

L'ensemble des clients WAMP (dans notre cas le serveur de jeu et les clients) sont connectés au Routeur WAMP, qui comme son nom l'indique enregistre les clients et route les données (sans en analyser le contenu, hormis pour les droits d'accès éventuels). Seul le routeur a connaissance des clients, leurs canaux souscris, les RPC enregistrées…

L'ensemble des clients sont enregistrés sur un realm (royaume), et ne peuvent communiquer qu'avec d'autres clients du même royaume, en sachant qu'un routeur est à même de gérer plusieurs realms simultanément.

Nous n'allons utiliser qu'un seul realm (« fodus ») mais une implémentation possible serait d'établir un realm par partie, avec donc un seul routeur centralisé. Moyennant un peu d'API, il est possible de concevoir un salon d'accueil où sont listées les parties (= les realms)

et donner la possibilité au joueur de les rejoindre ou d'en créer une nouvelle. Mais pour commencer, nous n'implémentons qu'un unique realm par routeur (donc une seule partie simultanée par routeur)



VI.C.3 Conception

Le routeur WAMP est un programme autonome (Bonefish, en C++), qu'il est possible d'intégrer au sein du serveur. Nous choisissons néanmoins de le lancement en mode autonome avec un fork() puis exec.

Bonefish est compilé à partir des sources (intégré comme submodule git du dépôt), avec une configuration gérée via les arguments de lancement du programme.

Exemple :

daemon/bonefish --realm "fodus" --websocket-port 9999 --rawsocket-port 8888

VI.C.3.a Description du workflow

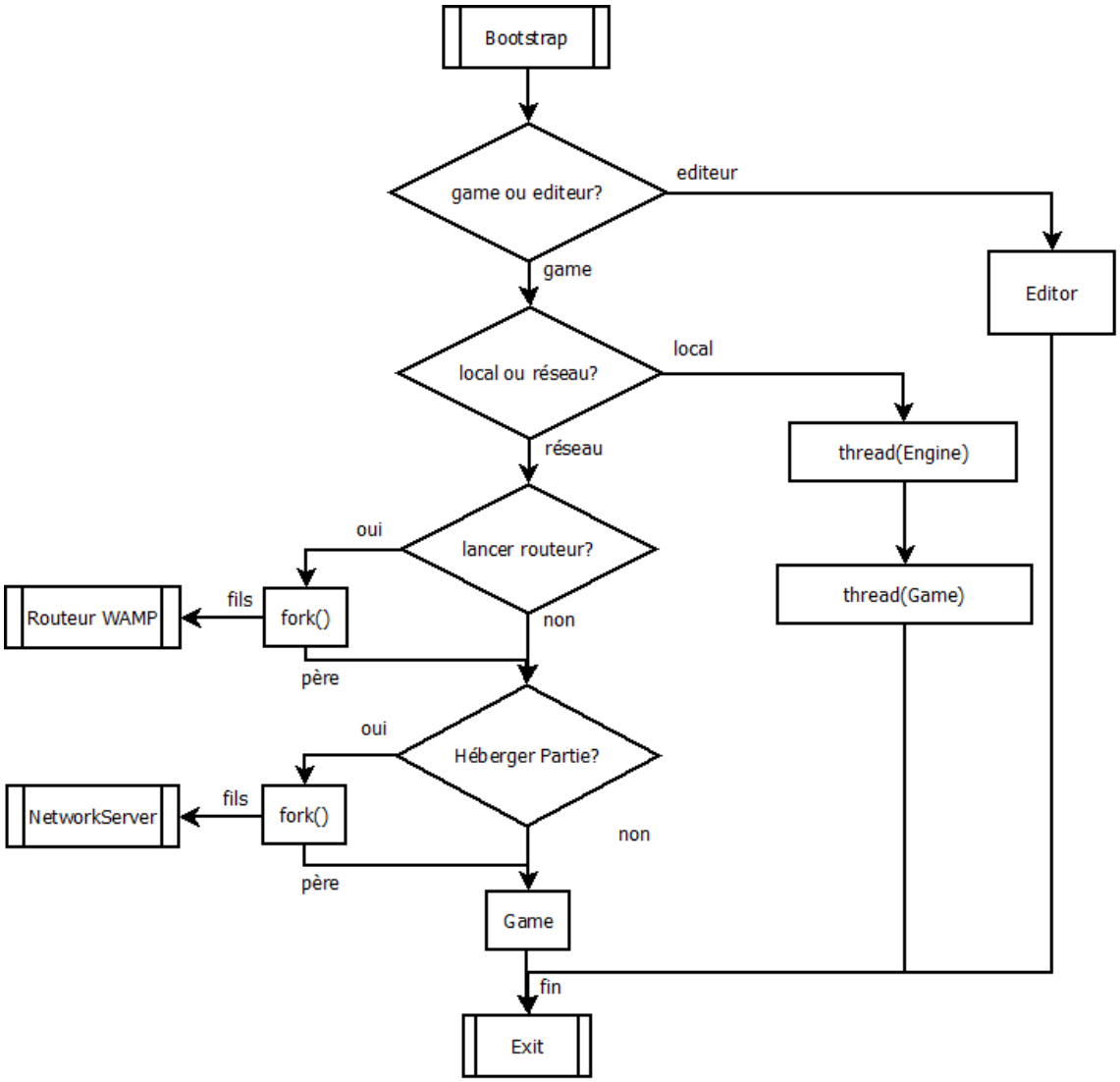


Figure 11 : workflow

Le lancement du serveur se fait au choix du mode dans le Bootstrap. Lors de la validation du mode réseau :

1. On renseigne la connexion au routeur
2. Si pas de connexion ou échec, on propose de lancer un routeur (ou de quitter le mode réseau)
3. Si le lancement du routeur est choisi, on fork puis on paramètre et on exécute Bonefish
4. On propose de rejoindre ou héberger une nouvelle partie
5. Si hébergement : fork, paramétrage + exécution du NetworkServer
   Sinon renseignement des infos de la partie à rejoindre
6. On lance un client du jeu paramétré pour rejoindre la partie (tout juste créée ou rejointe)

VI.C.3.b Description de l'API réseau

Les URI WAMP sont de la forme xxx.yyy.zzz (semblable au http : xxx/yyy/zzz)

VI.C.3.b.i RPC

RPC_NETWORK_SERVER_01_00 – Ajout d'une commande

#URI : engine.command.add

#Paramètres : int ClientID, int PlayerID, Command command(sérialisée)

#Retour : Code de statut

Soumission d'une Command au moteur de jeu du serveur.

END

RPC_NETWORK_SERVER_02_00 – Demande de synchronisation

#URI : engine.sync.request

#Paramètres : int ClientID, int PlayerID

#Retour : Code Statut + ElementList sérialisée

Soumission d'une Command au moteur de jeu du serveur.

END

RPC_NETWORK_SERVER_03_00 – Demande joueur courant

#URI : engine.sync.playing

#Paramètres : int ClientID, int PlayerID

#Retour : Code Statut + PlayerID

Récupération de l'ID du joueur actuellement en train de jouer

END

VI.C.3.b.ii PUB/SUB

CHANNEL_PUB_SUB_01_00 – Mise à jour globale

#URI : update.global

#Message : State etat (sérialisé)

Le moteur publie l'état actuel du jeu à tous les abonnés.

END

CHANNEL_PUB_SUB_02_00 – Mise à jour d'un joueur

#URI : update.player

#Message : Player player (sérialisé)

Le moteur publie l'état actuel d'un joueur à tous les abonnés.

END

CHANNEL_PUB_SUB_03_00 – Mise à jour fin de jeu

#URI : update.end

#Message : int WinningPlayerID, int score

Le moteur publie la fin du jeu et le vainqueur.

END

CHANNEL_PUB_SUB_04_00 – Mise à jour du joueur courant

#URI : update.player

#Message : int PlayerID

Le moteur publie l'ID du joueur jouant actuellement.

END

Les classes Adapter NetworkEngine et NetworkClient héritent de leur classe à adapter (Engine et Game respectivement), et implémentent les appels réseaux à NetworkHandler dans la redéfinition des fonctions impliquées, avant d'appeler la fonction originelle de leur parent.

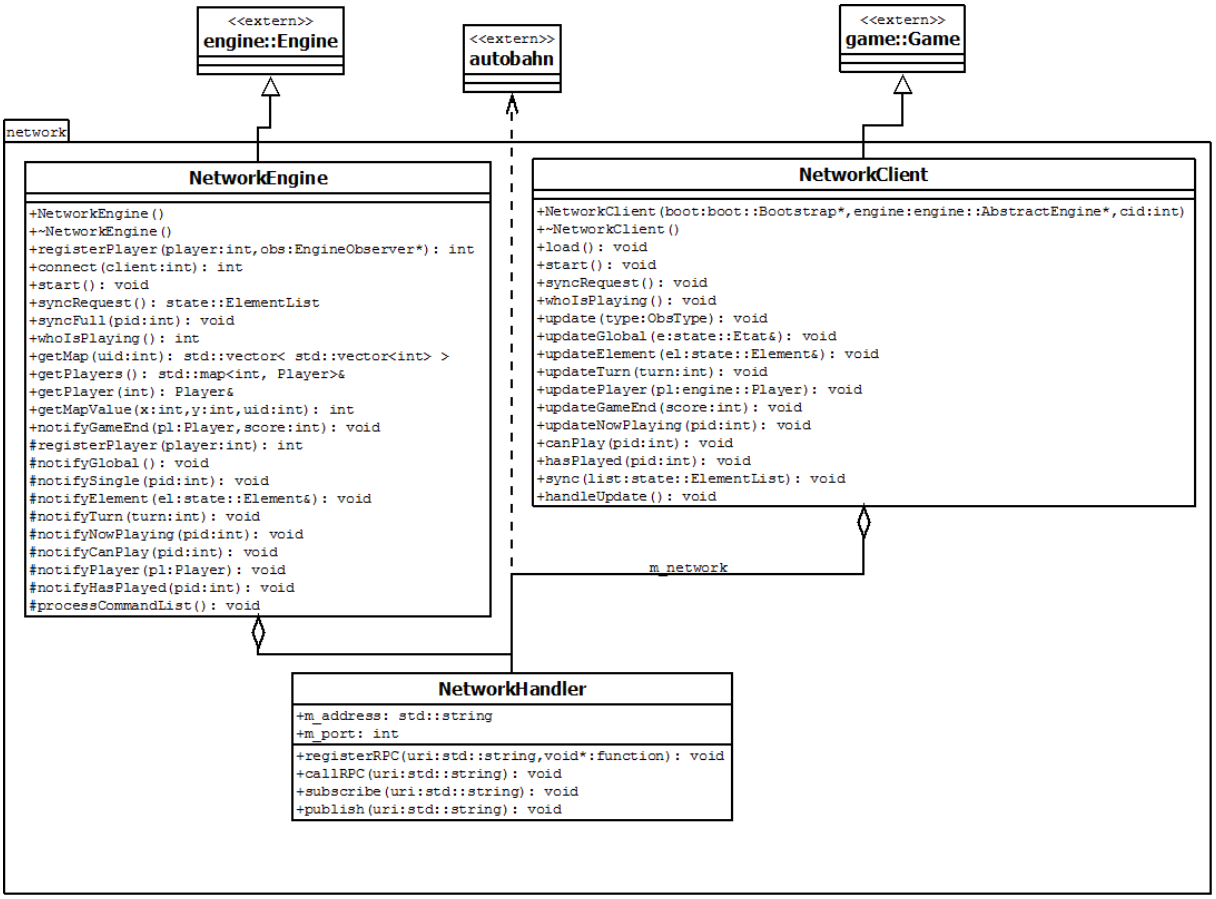


Figure 12 : Diagramme des classes de network