

# RAPPORT THE LEGEND OF PENGO

Yann TROU et Wassim DJELLAT

Mai 2020

## SOMMAIRE

### *I. Fonctionnalités principales*

### *II. Analyse*

**A) Architecture générale**

**B) Map et génération du labyrinthe**

**C) Joueur et IA**

**D) Serveur/Client**

### *III. Implémentation*

**A) Arboréscence et description des packages**

**B) Données annexes**

### *IV. Test et utilisation*

### *V. Conclusion*

### *VI. Bibliographie*

# 1 Introduction

Tout d'abord, nous avons décidé de choisir Pengo parce que c'était le plus intéressant des trois sujets proposés. Pour faire ce projet à notre sauce, on a fait un mélange entre les mécaniques de gameplay de Pengo, et les textures de The Legend of Zelda.

## 1.1 Les Fonctionnalités principales

Les fonctionnalités principales de notre projet sont d'avoir un Link et des ennemis. Pour gagner Link doit aligner les trois blocs de triforce horizontalement ou verticalement, ou alors tuer tous les ennemis. Pour les tuer, il faut faire glisser un bloc de glace dans leur direction pour les écraser ou bien les étourdir en tapant contre un mur pour les rendre vulnérables et les tuer.

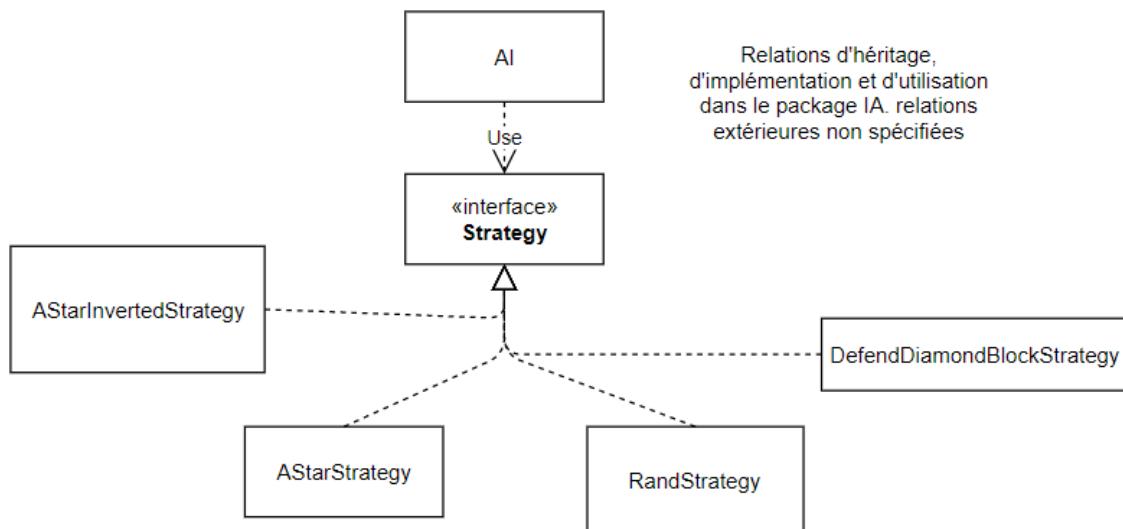
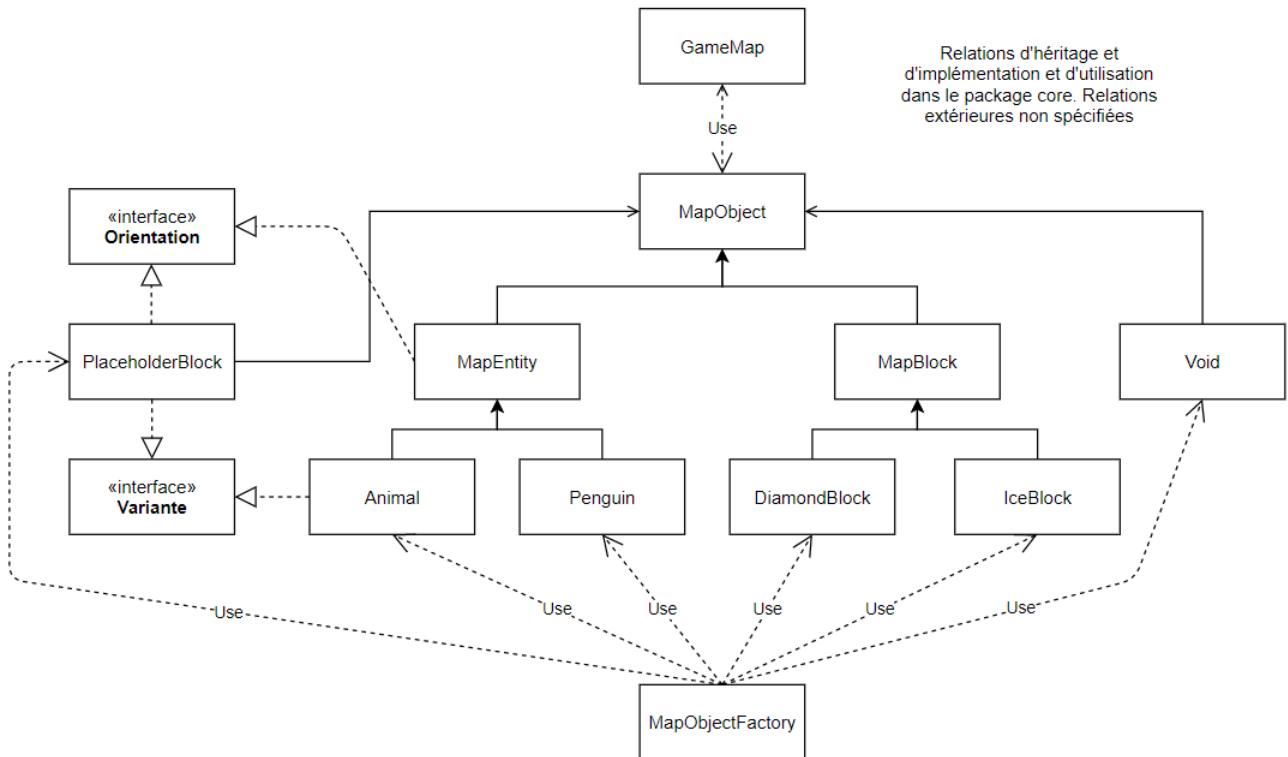
Ensuite vient la partie beaucoup plus compliquée, la partie réseau. Elle consiste à créer deux équipes sur un serveur. Une équipe de Link et une équipe d'ennemis, définies par l'hôte. Les joueurs se positionnent dans l'équipe de leur choix, puis la partie commence. Notre projet a été conçu pour que le joueur peut jouer en réseau (local) à plusieurs. Les fonctionnalités sont les mêmes que le mode solo.

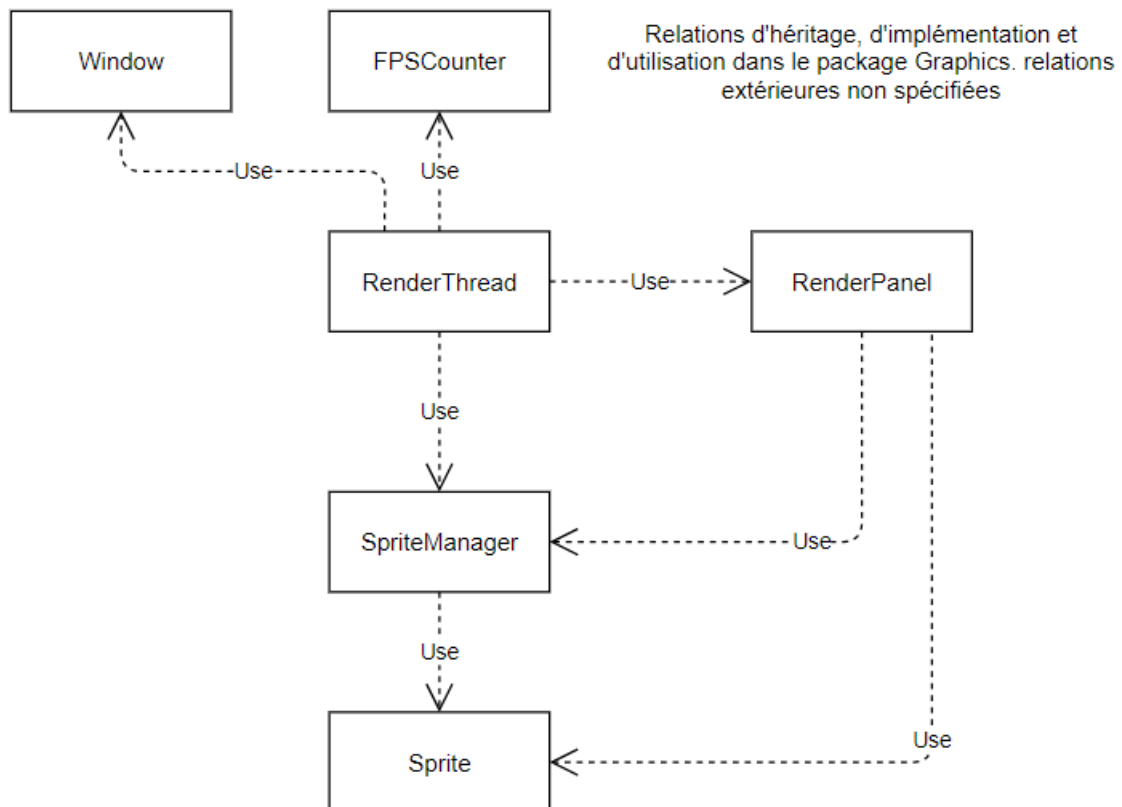
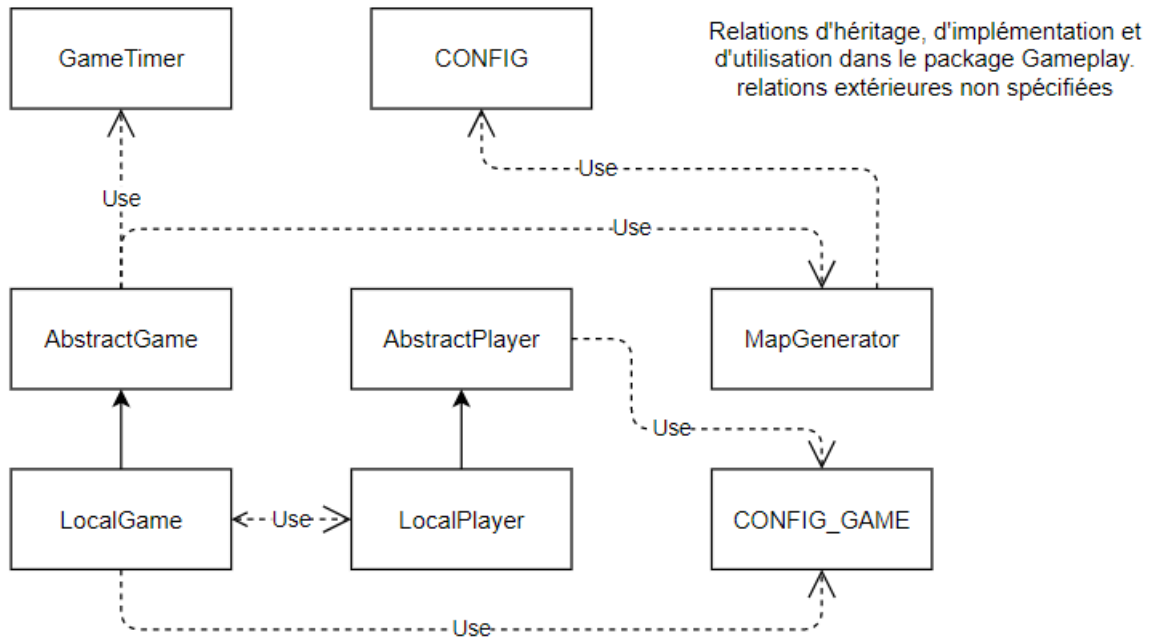
Nous avons aussi implémenté plusieurs stratégies pour l'IA, fonctionnant aussi bien en solo qu'en multijoueur. L'application présente un système de profils, agissant comme des pseudos et également une sauvegarde des profils et du classement dans leurs fichiers respectifs. Le mode multijoueur permet de jouer dans toutes les configurations possibles : Coop contre IA, Joueur contre Joueur, avec chaque équipe pouvant contrôler soit les ennemis, soit Link.

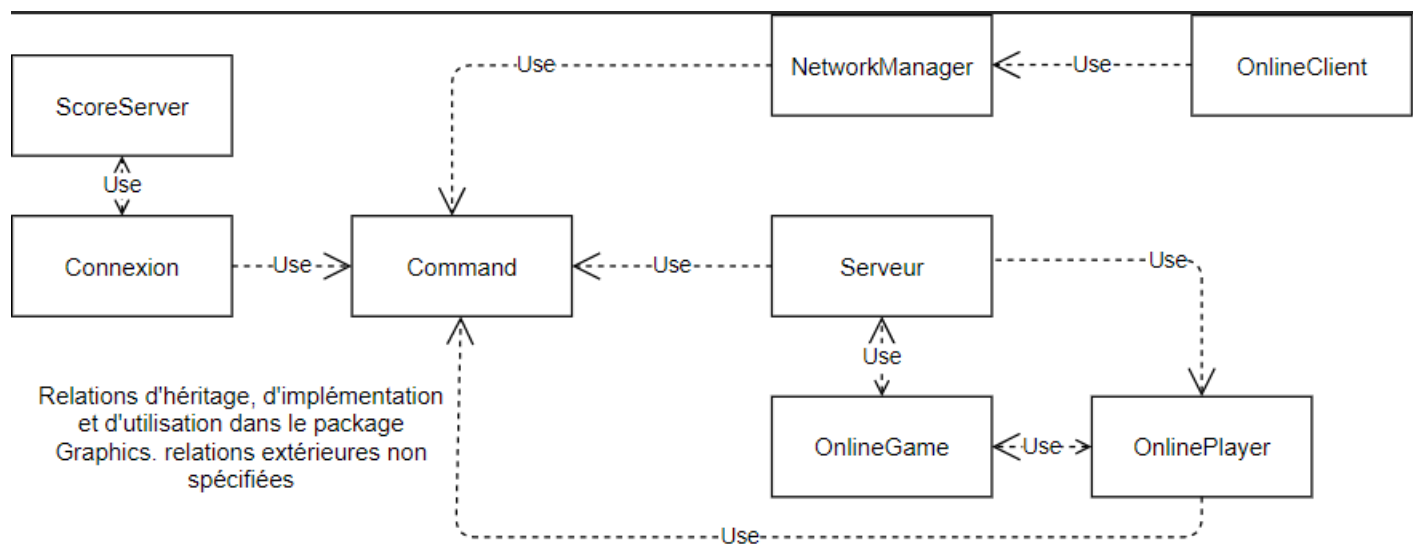
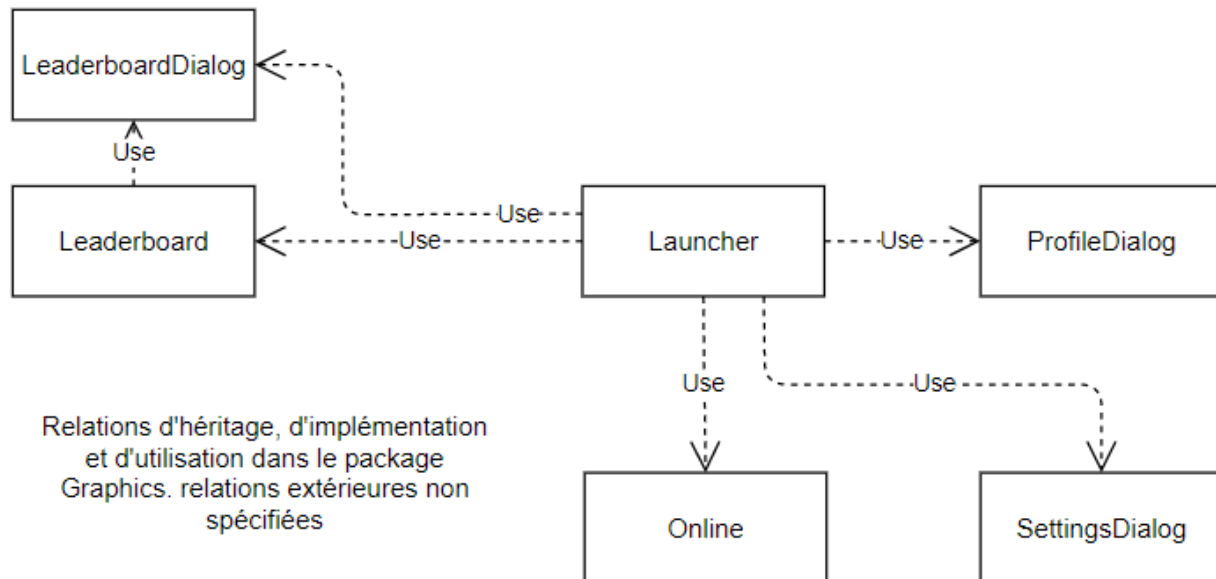
Nous disposons d'un mode basse résolution pour les écrans à faible résolution, simplifiant grandement l'aspect du jeu et la taille de la fenêtre.

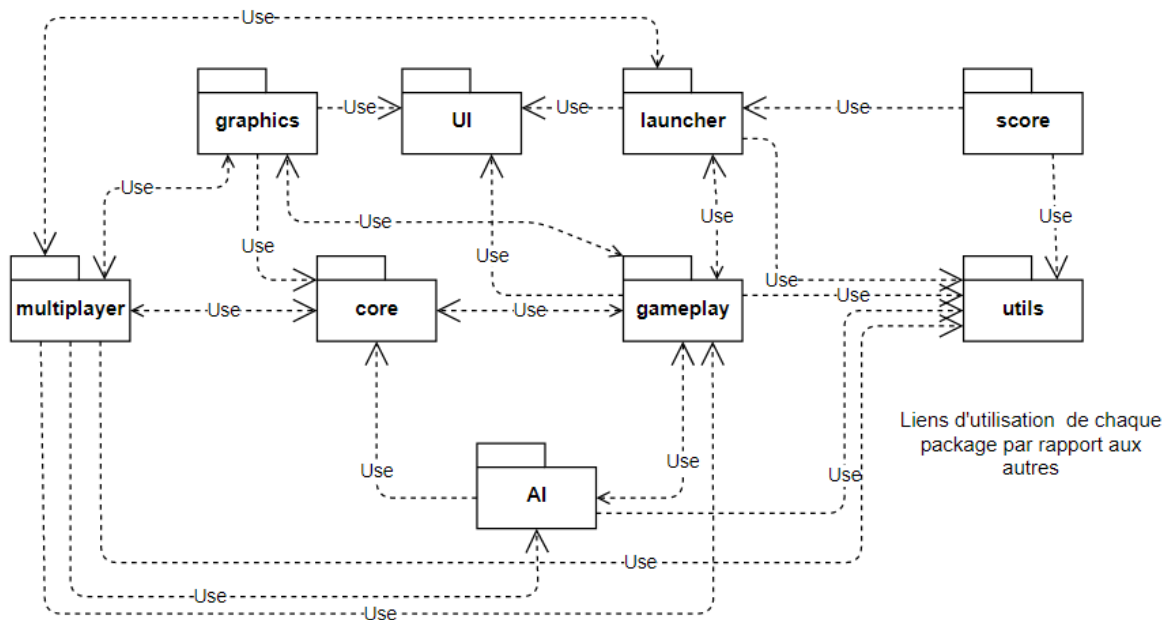
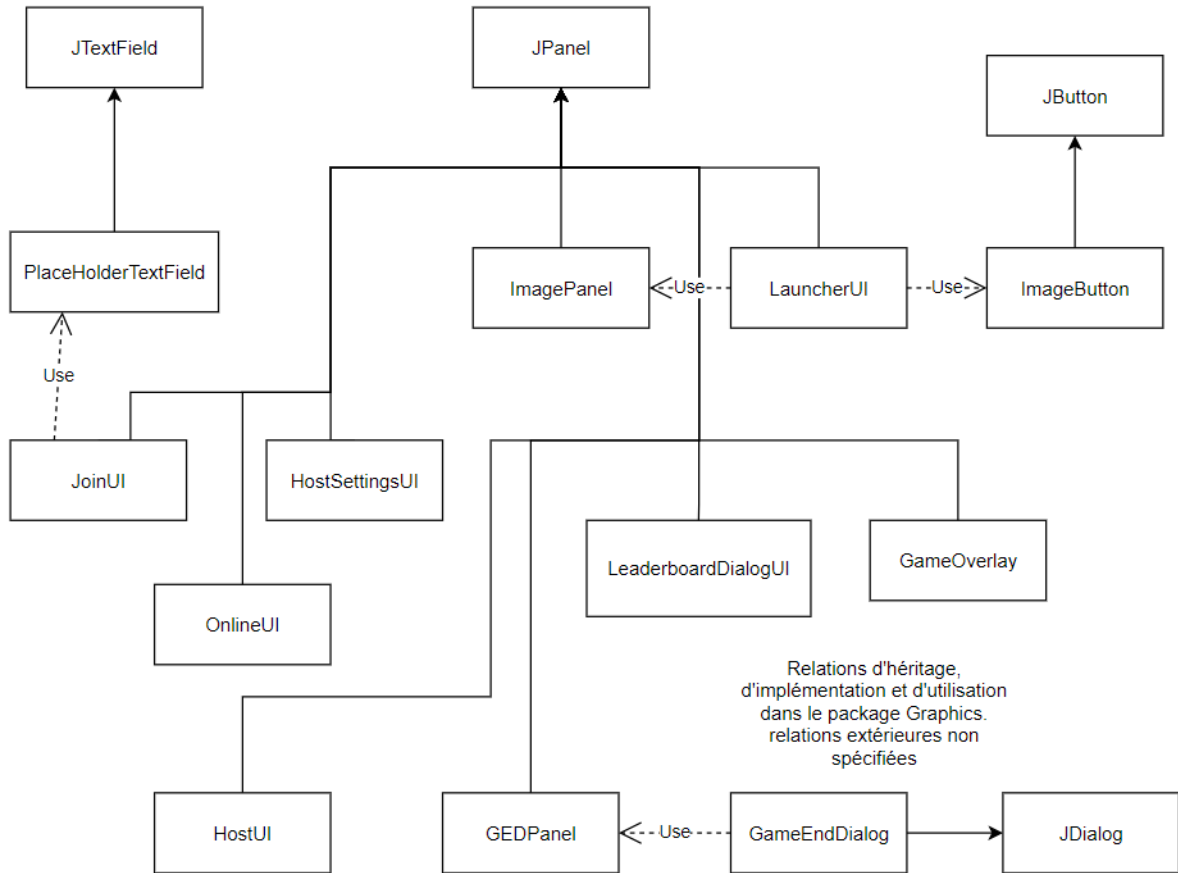
## 2 Analyse

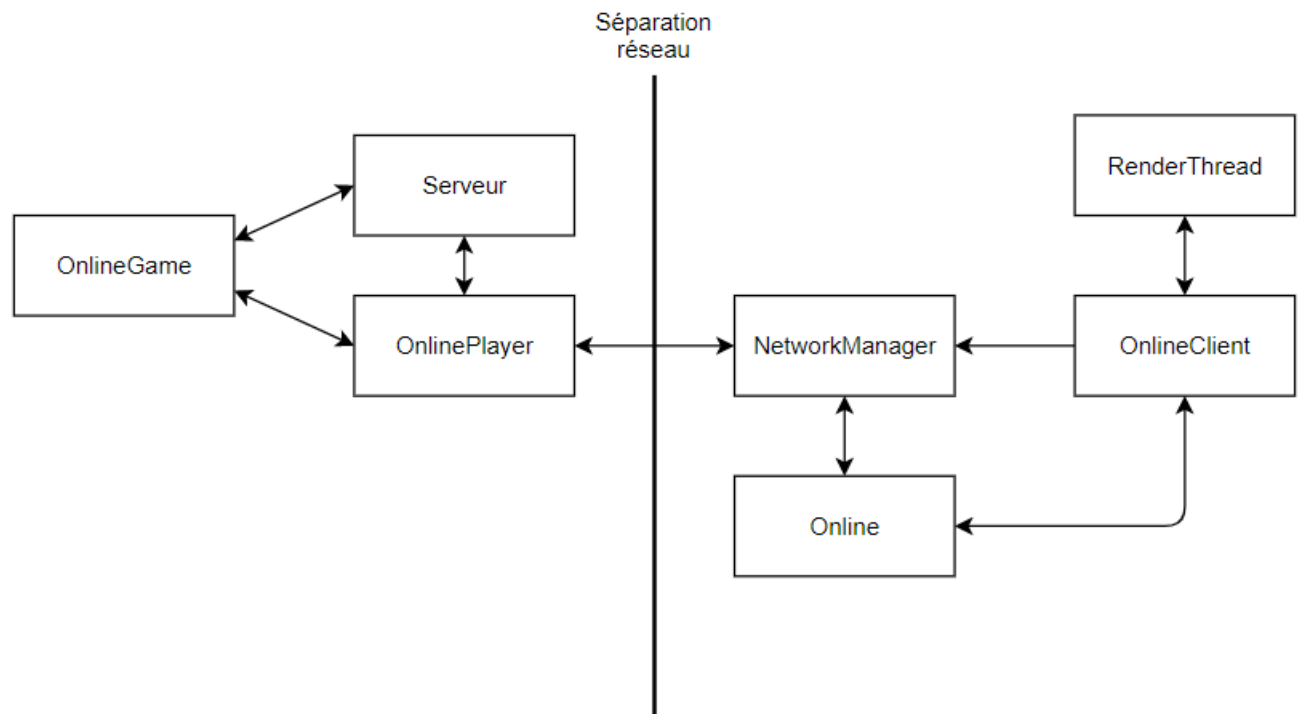
### 2.1 Architecture générale











Schémas de notre projet fait à la main grâce à l'application Draw.io.

## 2.2 Représentation des objets du jeu

Les objets du jeu tels que les blocs, ennemis et link sont représentés par les classes du package core. Ce package contient tous les différents objets que l'on peut placer sur la carte. Cette implémentation fonctionne (presque) de la même façon que l'on soit en solo ou multijoueur. Tous les objets héritent de la classe abstraite `MapObject`, qui contient le strict minimum pour un objet : position x, y, type d'objet, et un pointeur vers la carte où il est placé. La classe contient aussi les méthodes pour se déplacer sur la carte dans les quatre directions. Ensuite, il y a encore un niveau d'abstraction. La classe abstraite `MapBlock` dont héritent le bloc de trident et le bloc de glace, contient la méthode récursive pour faire une glissade et détruire tous les ennemis qui oseraient croiser son chemin. Les classes `IceBlock` et `DiamondBlock` n'implémentent rien de plus si ce n'est leur type. L'autre classe abstraite, `MapEntity`, implémente l'interface `IOrientation`, permettant d'avoir différentes textures en fonction de l'orientation du bloc. De cette classe héritent `Animal` et `Penguin`. `Animal` implémente l'interface `IVariante`, permettant d'avoir différents sprites en fonction de la stratégie que l'ia utilise. Elle contient aussi le code pour détruire les blocs de glace et les pingouins. La classe `Penguin`, de son côté, implémente le code pour déclencher la glissade ou la destruction d'un `MapBlock` ainsi que le droit d'appeler `stun()` pour étourdir les ennemis.

Il y a aussi la classe `MapObjectFactory` : cette classe regroupe toutes les méthodes pour créer un `MapObject` et le placer sur une `GameMap`. Toutes ces méthodes sont statiques car cette classe n'a pas besoin d'être implémentée. C'est la seule façon par laquelle des `MapObject` devraient être créés.

Il y a également les classes `Void` et `PlaceholderBlock`. Ces deux classes héritent de `MapObject` et servent deux besoins très spécifiques : `Void` sert dans le cas où un emplacement sur la Map est vide. Plutôt que retourner `null`, on retourne un objet `Void`.

La classe `PlaceholderBlock` sert uniquement dans le cas du jeu multijoueur côté client. `OnlineClient` va utiliser exclusivement ces blocs car ils sont plus adaptés. Ce qui importe est que le `SpriteManager` reconnaisse ces placeholders comme ce qu'ils sont censés remplacer.

## 2.3 Map et génération du labyrinthe.

La map (classe `GameMap`) fonctionne comme une liste répertoriant tous les blocs dans le jeu. On peut lire un emplacement avec `getAt()`, supprimer un objet répertorié avec `release()` et placer un objet avec `place()`. C'est une ressource critique car quasiment tous les algorithmes l'utilisent d'une manière ou d'une autre pour mener à bien leurs calculs. C'est ici que la totalité des problèmes de concurrence intervient. Pour les résoudre, les méthodes permettant d'écrire ( `place` et `release` ) sont synchronisées.

Pour la génération de labyrinthe (classe `MapGenerator`), le premier algorithme qui a été utilisé est celui d'exploration exhaustive : (1\*). Néanmoins, il se révélera peu adapté car le concept de cloison n'est pas représenté dans l'implémentation. Mur et cellule sont la même chose, ainsi l'algorithme supprimait trop de blocs, ou alors pas assez. Finalement, après des recherches sur Google, on a trouvé un post : (2\*) détaillant un bug trouvé dans le code source original et montrant le pseudo code de l'algorithme utilisé. Nous l'avons donc repris et adapté pour notre projet. Il fonctionne ainsi : il commence par chercher un bloc dont l'une



des directions peut contenir un chemin. Il creuse dans une direction valide tirée au hasard et modifie sa position comme étant la ou il a creusé. Ensuite, il vérifie si il peut encore creuser a partir de cette nouvelle position, si c'est le cas il recommence, sinon il recherche un nouveau bloc pour commencer un nouveau chemin.

## 2.4 Le joueur et l'IA

On va commencer par le joueur. On peut abstraire le concept de joueur comme étant des points, des vies, un objet contrôlé et un pseudo dans une classe (AbstractPlayer) qui s'exécute dans un Thread (implémente Runnable). Au final, la seule méthode commune aux implémentations d'AbstractPlayer et celle qui permet d'ajouter les points. Dans le cas du mode solo, le joueur est représenté par la classe LocalPlayer. celle-ci ne contient pas grand chose, juste du code pour faire le lien entre les entrées clavier (classe InputHandler) et l'objet contrôlé. La classe OnlinePlayer, par contre, gère bien plus de choses. Utilisée par le Serveur dans le mode multijoueur, elle gère elle même l'envoi et la réception des commandes, ainsi que leur application. Son rythme de travail est cadencé par la réception de commandes, à l'instar du LocalPlayer, cadencé à 16ms. Les commandes qu'elle reçoit concerne presque tout, placement dans une équipe spécifique, déplacement, démarrage de la partie.

On peut décomposer l'IA en deux parties. La première, (classe AI) est sa représentation dans le code, avec son implémentation des délais d'étourdissement, de réapparition via des simples compteurs. L'autre partie, concerne les stratégies qu'elle peut utiliser. Les stratégies que nous avons implémentées fonctionnent de manière similaire. nous avons une méthode process détaillant le fonctionnement macro de la stratégie puis, chaque stratégie dispose de ses propres fonctions effectuant vraiment le calcul de la stratégie.

Commençons par détailler la stratégie qui chasse un joueur constamment (classe AStarStrategy). On avait pensé utiliser l'algorithme A Star, permettant de trouver un chemin entre l'IA et la cible. mais nous avons préféré le simplifier car il n'y a pas d'intérêt a calculer un chemin entier si on ne se déplace que d'une case a la fois. Au final nous utilisons de simples distances vectorielles pour trouver le bloc adjacent le plus proche de la cible et déplacer l'objet contrôlé dessus. Si il est coincé par un bloc de triforme, un simple tirage lui évite de l'être trop longtemps.

Ce n'est pas la seule stratégie qu'elle possède. Au lieu de lui donner un comportement d'attaque, nous avons imaginé l'inverse : une stratégie qui la ferait fuir et éviter le joueur a tout prix. pour cela, nous avons repris la méthode de A Star simplifiée utilisée au dessus, sauf qu'au lieu de se déplacer sur la case avec la distance la plus proche, on choisit celle avec la plus grande. également, cette stratégie fait en sorte d'éviter les blocs lancés par le joueur en essayant de se placer en diagonale de lui. La seule solution pour le tuer est d'aller plus vite que lui.

La première stratégie que nous avons implémentée est la plus simple. on tire une direction au hasard et on se déplace dans cette direction tant que l'on ne rencontre pas de mur ni de bloc de triforme. Aussi, il se peut qu'elle change de direction au hasard de son déplacement après tirage.

Pour finir, la stratégie de défense des blocs de triforme. l'algorithme fonctionne ainsi : on liste tous les blocs de triforme sur le plateau avec une double boucle imbriquée et on tire celui que

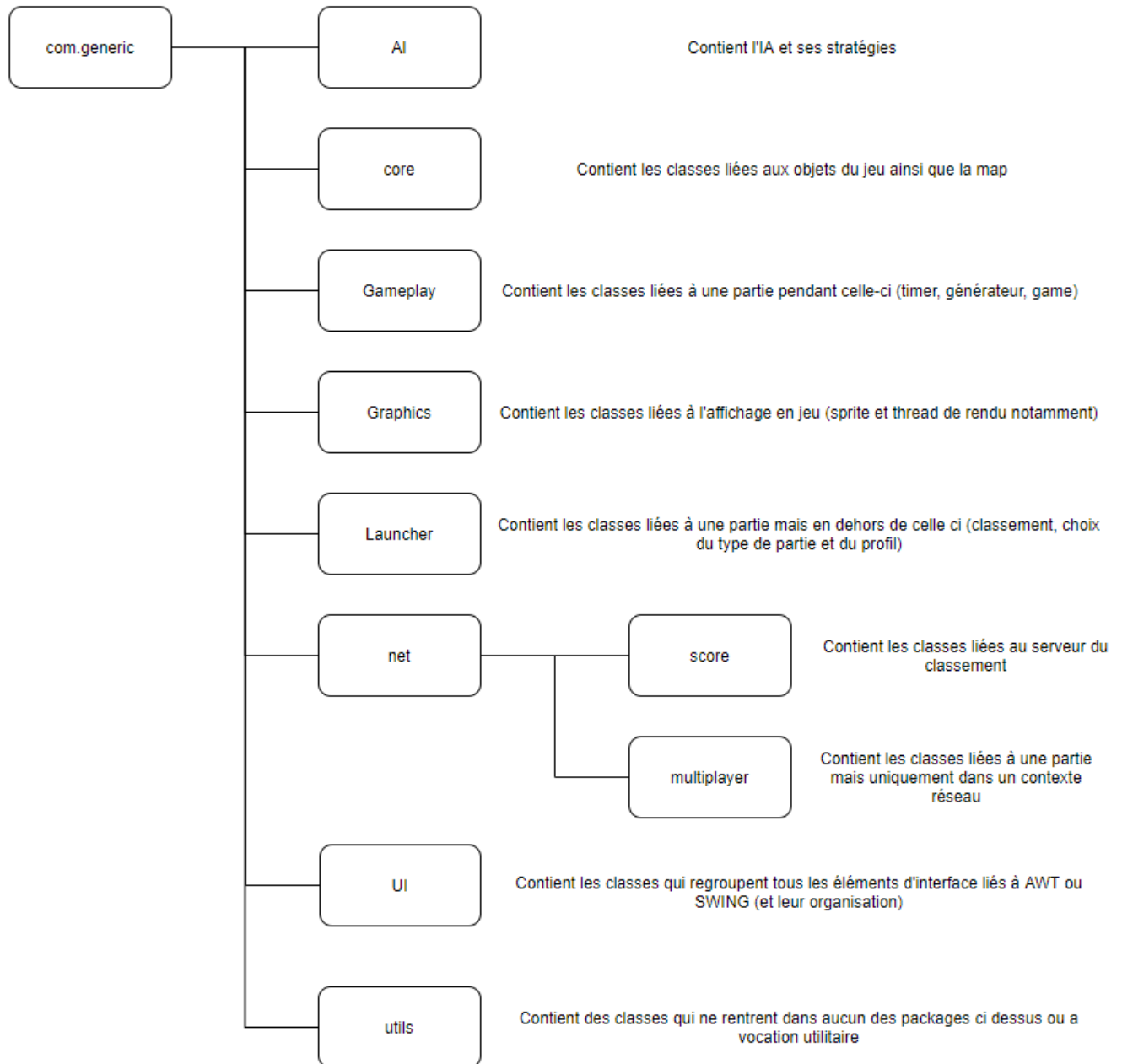
l'on va défendre au hasard. Ensuite, on tire un entier aléatoire qui dira le rayon dans lequel le bloc de glace à détruire se trouvera. On recherche le bloc, on le définit comme cible et on se déplace vers lui avec l'algorithme A Star modifié tant qu'il est encore sur le plateau. Une fois le bloc détruit, on recommence le processus de recherche.

## 2.5 Mode multijoueur

Du côté du client, il y a plusieurs classes qui entrent en jeu. Nous allons les traiter dans l'ordre où elles sont impliquées dans la création puis la réalisation d'une partie en ligne. La première classe, (classe Online) gère le GUI et le traitement des commandes reçues. quand l'utilisateur décide d'héberger une partie, un Serveur est démarré et commence de recevoir des connexions. La classe Serveur contient la composition des équipes au travers de deux HashMaps pour chaque équipe référençant les pseudos associés aux connexions. Si d'autres joueurs veulent rejoindre la partie, ils peuvent se connecter au serveur. La classe NetworkManager sert exclusivement à la réception et l'envoi de commandes côté client. c'est elle qui se connecte réellement au serveur. Elle possède des méthodes pour envoyer des commandes prédéfinies. Une fois que l'hôte a cliqué sur "Lancer la partie", une commande "GAME START" est envoyée à tous les utilisateurs connectés. La réception de cette commande déclenche la création d'une instance de OnlineClient et le vrai début de partie. La classe OnlineClient sert à faire le lien entre la partie sur le serveur et le joueur. C'est à dire : récupérer les entrées clavier, recevoir les mises à jour de la carte et les afficher dans la fenêtre, recevoir les mises à jour de l'overlay et afficher la fenêtre de fin de partie. La vraie partie, elle, se déroule sur le serveur avec la classe OnlineGame. c'est la variante multijoueur de AbstractGame, implémentant toutes les méthodes nécessaires au bon déroulement de la partie. Entre autres : gestion de l'étourdissement, réapparition des ennemis et liens dans chaque camp, initialisation des joueurs (et IA si nécessaire) , appel du tracage du labyrinthe, démarrage et arrêt de la partie. Une fois que la partie est finie, une commande "GAME END" est envoyée à tous les joueurs connectés, le résultat de la partie leur sera affiché, et ils retourneront dans le menu principal. Pour la création du Serveur/Client

### 3 IMPLEMENTATION

#### 3.1 Arborescence et description des packages

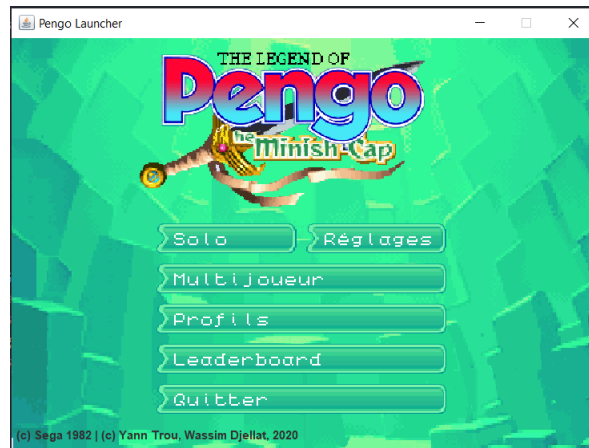


### 3.2 Données annexes

Dossier ressources : contient les ressources nécessaire au bon fonctionnement et au bon affichage des éléments du jeu. contient les textures des murs, du menu principal, de link, des ennemis, de l'overlay, en haute et basse définition ainsi que la police d'écriture dans le style pixellisé.

Dossier saves : contient deux fichiers Ladder.sav et PlayerProfiles.sav. Le fichier Ladder.sav est la sauvegarde du leaderboard effectuée par la classe ServeurScore. Le fichier PlayerProfiles.sav contient la liste des pseudo des joueurs précédents, sauvegarde effectuée par la classe Launcher et qui est chargée au démarrage du jeu pour remplir la partie profils du launcher.

## 4 TEST ET UTILISATION



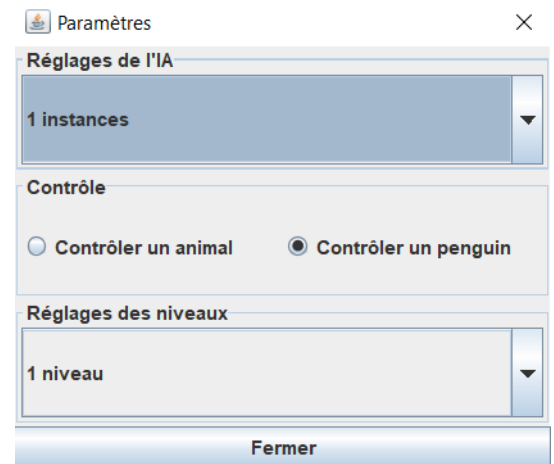
Voici le launcher, c'est à partir de celle-ci que nous allons pouvoir lancer le jeu avec différents paramètres possibles.

Si vous cliquez sur Solo, cela vous lancera une partie en local avec des paramètres par défaut.



Ensuite si nous allons dans les réglages, pour modifier la partie :

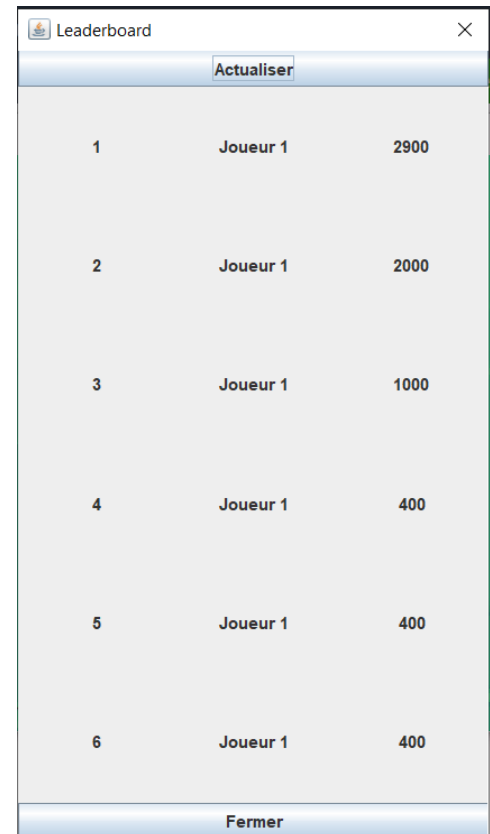
Donc avec les réglages solo, nous pouvons régler le nombre d'instance donc d'ennemis. Nous pouvons choisir si l'on veut contrôler le Link ou bien les ennemis. Nous pouvons aussi choisir le nombre de niveaux à jouer.



Quand nous cliquons sur Profils, nous atterissons dans le menu des choix de profils.  
Nous pouvons ajouter/supprimer/sélectionner un profil.



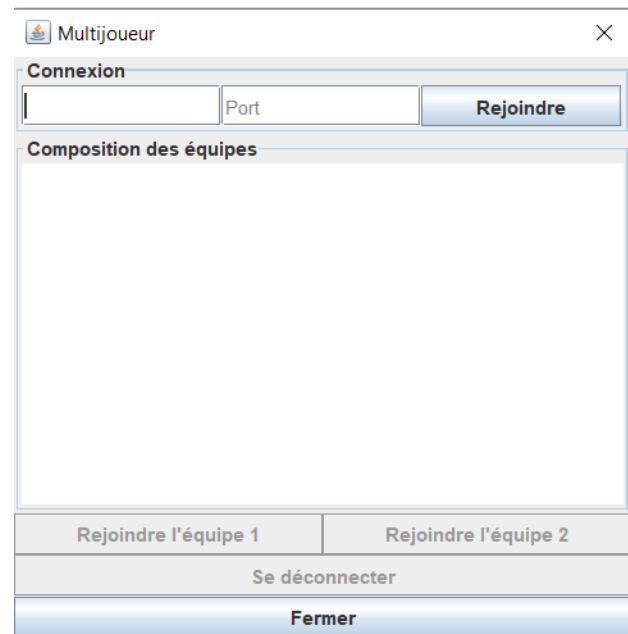
Dans le leaderbord est stocké le score de tous les joueurs ayant jouer, ce score est trié grâce à un classement des meilleurs joueurs.



Quand nous cliquons sur Multijoueurs, nous arrivons à un choix, soit vous hébergez la partie ou bien vous vous connectez à une partie.  
Nous allons voir les deux :



Quand vous vous connectez à une partie, vous pouvez choisir l'ip, le port et ensuite vous vous connectez à l'hébergeur de la partie.  
Vous pouvez aussi vous déconnecter et quitter le lobby.  
Ensuite vous pouvez rejoindre selon votre choix une des deux équipes.

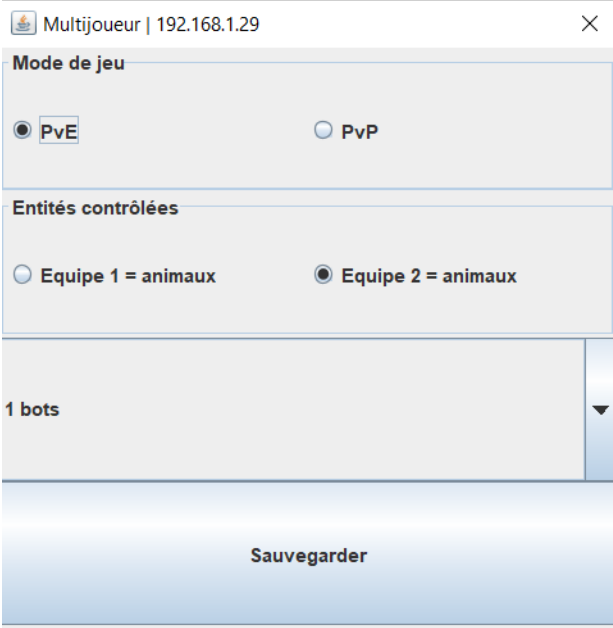


Quand vous voulez héberger une partie, vous avez le choix de rejoindre une des deux équipes. Vous pouvez utiliser les paramètres avant de lancer la partie.



The screenshot shows a window titled 'Multijoueur | 192.168.1.29'. It has two tabs: 'Lancer la partie' and 'Paramètres'. The 'Paramètres' tab is active, showing a section titled 'Composition des équipes'. Below this title, there is a large empty box labeled 'Joueur 1'. At the bottom of the window, there are three buttons: 'Rejoindre Equipe1', 'Rejoindre Equipe2', and 'Retour'.

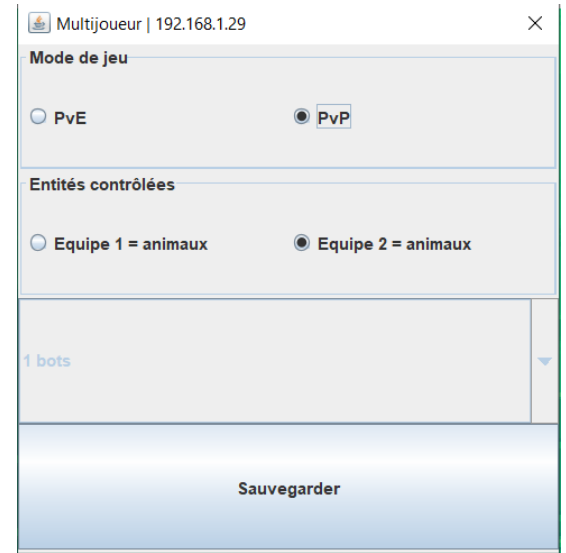
Dans ces paramètres nous avons le choix de faire du PvE ( une partie où les joueurs jouent contre des IA ). Quand nous sommes sur du PvE, nous avons le choix d'être soit des ennemis, soit des alliés ( penguin ). Nous pouvons aussi ajouter un nombre indéfini d'IA.



The screenshot shows the same 'Multijoueur | 192.168.1.29' window, but with the 'Paramètres' tab selected. The 'Mode de jeu' section has two radio buttons: 'PvE' (selected) and 'PvP'. Below this, the 'Entités contrôlées' section has two radio buttons: 'Equipe 1 = animaux' and 'Equipe 2 = animaux' (selected). At the bottom, there is a text input field labeled '1 bots' and a 'Sauvegarder' button.



Quand nous sommes sur les réglages PvP ( Joueurs contre Joueurs ) nous avons le choix de décider qui des deux équipes seront les ennemis. l'accès à l'IA est désactivé. Après le choix fait vous cliquez sur sauvegarder pour revenir au lobby avec les autres joueurs.



```
public class CONFIG_GAME {
    public static int PLAYER_INIT_LIVES = 2;
    public static int AI_INIT_LIVES = 6;
    public static boolean PLAYER_IS_ANIMAL = false;
    public static boolean PLAYER_IS_PENGUIN = true;
    public static int N_AI = 6;
    public static int N_NIVEAUX = 1;

    public static boolean TEAM_1_IS_ANIMAL = false;
    public static boolean TEAM_2_IS_ANIMAL = true;

    public static boolean PvE = true;
    public static boolean PvP = false;

    public static void setPlayerIsPenguin(boolean val) {
        PLAYER_IS_ANIMAL = !val;
        PLAYER_IS_PENGUIN = val;
    }
}
```

FIGURE 1 –

```
public final static String WINDOW_TITLE = "Pengo Remake ";

public static boolean LOW_RES_MODE = false;

public final static int WH_HIGH_RES = 940;
public final static int WW_HIGH_RES = 765;

public final static int WH_LOW_RES = 520;
public final static int WW_LOW_RES = 400;

public final static int AI_TICK_RATE = 350;
public final static int STUN_TIME = 2000;

public final static int GRID_WIDTH = 13; //650px
public final static int GRID_HEIGHT = 15; //750px

public final static int SPRITE_SIZE_HD = 50;
public final static int SPRITE_SIZE_SD = 25;

public final static Color BG_DEFAULT_COLOR = new Color(0, 24, 24);

public static void setLowResMode(boolean val) { LOW_RES_MODE = val; }
```

FIGURE 2 –

Ici nous avons les paramètres qui sont changés dans le code après le choix des réglages, elles sont réparties dans deux classes différentes : Config et ConfigGame. Config sert à la configuration strictement utilitaire (résolution, taille de la grille, taille des sprites, etc..) tandis que ConfigGame sert pour régler les paramètres de la partie. Une vidéo de démonstration sera mis à disposition joint avec ce rapport.

## 5 CONCLUSION

Pour conclure, ce projet nous a beaucoup apporté de connaissances en Java. Interfaces graphiques, programmation réseau, organisation, etc.. Mais on pourrait encore l'améliorer : permettre de jouer plusieurs parties consécutives en réseau, avoir de la musique, des effets sonore, des animations.

également, corriger certains bugs qui restent encore peu compréhensibles pour nous..

## 6 Bibliographie

1\* = [https://fr.wikipedia.org/wiki/Mod%C3%A9lisation\\_math%C3%A9matique\\_de\\_labyrinthe](https://fr.wikipedia.org/wiki/Mod%C3%A9lisation_math%C3%A9matique_de_labyrinthe)

2\* = [http://www.ukvac.com/forum/sega-pengo-bug-found-in-maze-generating-algorithm\\_topic335921.html](http://www.ukvac.com/forum/sega-pengo-bug-found-in-maze-generating-algorithm_topic335921.html)