Rapport de Projet d'Info4B

Mélissandre GALLOIS, Hugo GUYOT

Vendredi 15 Avril 2022

Table des matières

1 Introduction **3**

2 Présentations **3**
2.1 Présentation du jeu d'origine . . . 3
2.2 Présentation du sujet . . . 4

3 Analyse fonctionnelle **4**
3.1 Règles de fonctionnement . . . 4

4 Spécification des classes principales et leurs méthodes essentielles **4**
4.1 La classe Game . . . 4
4.1.1 Ses méthodes essentielles . . . 4
4.2 La classe Affichage . . . 5
4.2.1 Ses méthodes essentielles . . . 5
4.3 La classe Map . . . 5
4.3.1 Ses méthodes essentielles . . . 5
4.4 La classe Player . . . 6
4.4.1 Ses méthodes essentielles . . . 7
4.5 La classe Bot . . . 7
4.5.1 Ses méthodes essentielles . . . 8
4.6 La classe Command . . . 8
4.6.1 Ses méthodes essentielles . . . 8
4.7 La classe Menu . . . 8
4.7.1 Ses méthodes essentielles . . . 9
4.8 La classe Function . . . 10
4.8.1 Ses méthodes essentielles . . . 10
4.9 La classe Block . . . 10
4.9.1 Ses méthodes essentielles . . . 11
4.10 La classe Points . . . 11
4.10.1 Ses méthodes essentielles . . . 11
4.11 La classe Vie . . . 11
4.11.1 Ses méthodes essentielles . . . 11
4.12 La classe Serveur . . . 12
4.12.1 Ses méthodes essentielles . . . 12
4.13 La classe Client . . . 12
4.13.1 Ses méthodes essentielles . . . 12

5 Description des structures de données **12**
5.1 Solutions envisagées . . . 12
5.2 Solutions retenues . . . 12

6 Architecture logicielle détaillée **13**

7 Jeu de test **14**

8 Conclusion **14**

1 Introduction

Nous sommes deux étudiants en deuxième année de Licence Informatique-Électronique à l'UFR Sciences et Techniques de Dijon. Dans le cadre de notre module d'Informatique consacré aux principes des systèmes d'exploitation, il nous a été demandé de réaliser un projet sur un des trois sujets donnés.
Le sujet que nous avons choisi est le numéro 2.

Le projet est basé sur un jeu existant : Lode Runner.

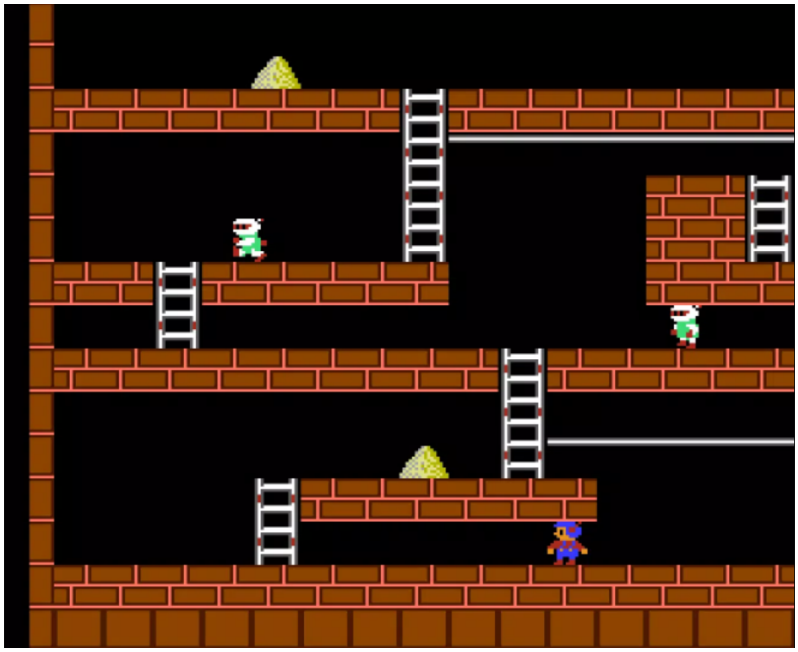

FIGURE 1 – Le jeu existant : Lode Runner

2 Présentations

2.1 Présentation du jeu d'origine

Lode Runner, ce petit jeu rétro de plateforme et de réflexion a été développé par Douglas E.Smith et publié par l'éditeur Brøderbund en 1983 sur la plateforme Apple II (pour la génération Z, un Apple II est un des premiers ordinateurs personnels 8-bit au monde, l'Apple II a été mis en vente à partir de 1977).

FIGURE 2 – Un Apple II 8-bit

Lode Runner fut un franc succès lors de son époque, notamment au Japon. Depuis, il a été réadapté sur plusieurs supports tels que PS, Gameboy, Atari, Wii(U), Xbox360, Nintendo DS, etc..

2.2 Présentation du sujet

Ce projet consiste donc à coder en Java, de créer des classes principales accompagnées de leurs méthodes, mais également de suivre une conception architecturale comme celle abordée dans nos cours.

Nous démarrons le jeu sur une page d'accueil qui vérifiera le type de jeu sollicité, le nombre de joueur et qui nous permettra d'accéder à un des niveaux disponibles.

Nous avons donc un joueur qui contrôlera un petit personnage, lâché dans la grande faune d'un tableau 2D parsemé d'ennemis et de liasses d'argent à récupérer. Il peut éviter les ennemis en empruntant un parcours spécifique en s'aidant d'échelles et de tyroliennes.

Le jeu étant également accessible pour jouer avec ses amis en multijoueur, nous aurons donc à mettre à disposition un serveur accompagné de ses clients.

Vous avez à votre disposition, pour plus de détails, notre GitHub ci-joint :

GitHub du projet

3 Analyse fonctionnelle

3.1 Règles de fonctionnement

Notre application doit pouvoir prendre en compte plusieurs classes et fonctions. Nous allons définir les ressources et les acteurs de chaque classe, ainsi que leur(s) éventuel(s) problème(s) de concurrence. Nous avons 11 classes et une classe qui sert d'exécutable, la classe **Runner.java**.

Débutons avec la classe **Game** qui impose une partie de jeu. Nous poursuivrons avec la classe **Affichage** et ses sous-problèmes. Par la suite nous accéderons à la classe **Map**, puis avec la classe **Player** nous l'implémenterons de la classe **Command**. Nous aborderons également les classes **Menu** et **Function** et c'est enfin avec les classes **Block**, **Vie** et **Points** nous achevons cette analyse fonctionnelle, que nous reverrons plus en détails dans l'architecture logicielle.

4 Spécification des classes principales et leurs méthodes essentielles

4.1 La classe Game

Cette classe doit gérer le déroulement d'une partie de jeu. La classe Game est un thread qui possède les ressources et les acteurs suivants :

Ressource	Acteur
Map lvl	Player p
Map currentMap	Affichage aff
Function f	

FIGURE 3 – Ressource(s) et Acteur(s) de la classe

Cette classe ne rencontre pas de problème d'accès concurrents aux ressources.

4.1.1 Ses méthodes essentielles

La classe Game possède une méthode essentielle et son constructeur.
Sa méthode essentielle est updateMap() qui actualise la Map courante avec la Map de base. Ensuite nous ajoutons le(s) joueur(s),les ennemis ainsi que les éléments ramassables.

Son constructeur passe en paramètres un entier nommé i qui signifie le numéro du niveau joué. Nous affichons le niveau lu grâce à une fonction récupérée dans la classe Function. Nous faisons apparaître le joueur sur son lieu d'apparition.

4.2 La classe Affichage

Cette classe doit gérer l'affichage d'un niveau ainsi que la vie et les points de la partie en cours. C'est un thread qui possède les ressources et les acteurs suivants :

Ressource	Acteur
boolean lock	Game partie

FIGURE 4 – Ressource(s) et Acteur(s) de la classe

Ce thread rencontre un problème de concurrence dans son exécution. La section critique est la suivante :

```
while(lock)
{
    try
        {
            System.out.println(this.show());
            this.sleep(50);
        }
        catch (Exception e){e.printStackTrace();}
}
```

Pour gérer ce problème, plusieurs solutions s'offrent à nous.

- Un verrou
- Un moniteur
- Une sémaphore

Nous aborderons les solutions envisagées et celles retenues après avoir abordé toutes les classes principales.

4.2.1 Ses méthodes essentielles

La classe possède 2 méthodes. Une méthode show() pour afficher le niveau en cours suivi du nombre de vies restants au joueur et son score.

Sa seconde méthode est son programme d'exécution qui va afficher le niveau toutes les x millisecondes : cela correspond à la fréquence 1/T dont le temps est écrit dans le sleep(). Le tout dans une boucle while avec un verrou pour que l'affichage s'automatise le temps de la partie.

4.3 La classe Map

Cette classe doit gérer la taille et la création d'un niveau, avec les caractéristiques d'apparitions qui conviennent selon le niveau lu. Cette classe n'a qu'un attribut qui est un tableau 2D correspondant à un niveau.

Ressource	Acteur
int[][] map	

FIGURE 5 – Ressource(s) et Acteur(s) de la classe

Cette classe ne rencontre pas de problème d'accès concurrents aux ressources.

4.3.1 Ses méthodes essentielles

La classe Map possède trois constructeurs et 2 méthodes. Ces deux méthodes sont toutes deux des affichages concernant le niveau. La méthode show() sert juste à afficher le niveau sous format d'entiers (c'est plutôt une méthode de débogage car elle n'a aucune utilisation dans le programme).

C'est la méthode affiche() qui est essentielle car elle affiche le niveau sous format de caractères qui représente bien plus communément les objets du jeu.

4.4 La classe Player

Cette classe doit gérer le joueur et ses aléas lors d'une partie. Le joueur doit prendre en compte ses déplacements grâce à la classe Command, la gravité qui l'entraîne inexorablement vers le sol, la collision avec les ennemis ou bien avec un élément ramassable. Chaque joueur possède également un pseudo mais également un id sous format d'un entier attribué via **int ind**.

La classe Player est un thread qui possède les ressources et les acteurs suivants :

Ressource	Acteur
String name	fall()
Vie life	move(int dir)
Points score	
boolean evil	
int ind	
int x, int y	
int x_spawn, int y_spawn	
Map map	

FIGURE 6 – Ressource(s) et Acteur(s) de la classe

Ce thread rencontre des problèmes de concurrence dans son exécution. Les sections critiques sont les suivantes :

```
if (!(map.getCase(x, y) == 0 && map.getCase(x + 1, y) == 0 && map.getCase(x - 1, y) != 4))
{
    switch (dir) {
    case 1: /// s
            if (canMove(3)) {this.x++;}break;

    case 2: /// z
            if (canMove(4)) {this.x--;}break;

    case 3: /// d
        if (canMove(1)) {this.y++;}break;

    case 4: /// q
        if (canMove(2)) {this.y--;}break;

    default: break;
}
fall();
if (map.getCase(x, y) == 0){addScore(10);}
```

Ci-dessus, le code de la méthode move(int dir).

```
if (map.getCase(x, y) == 0 && map.getCase(x + 1, y) == 0 && map.getCase(x - 1, y) != 4)
{
    x++;
    fall();
}
```

Ci-dessus, le code de la méthode fall().

Pour gérer ce problème, plusieurs solutions s'offrent à nous.

- Un moniteur
- Une sémaphore

Nous aborderons les solutions envisagées et celles retenues après avoir abordé toutes les classes principales.

4.4.1 Ses méthodes essentielles

Cette classe possède plusieurs méthodes. Il est important de se pencher sur les quatre méthodes suivantes : **fall()**, **move(int dir)**, **boolean canMove(int i)** et **delBlock(int i)**.

Nous avons ici une ressource partagée : un thread Player modifie une variable partagée concernant les commandes, la méthode fall() et la méthode move(int dir) réquisitionnent le déplacement du joueur. Nous devons donc finir complètement une méthode avant de pouvoir réaliser un autre déplacement.
La section critique est celle contenue dans les deux méthodes :

- Dans la méthode fall nous avons les lignes de code qui suivent :

```
if ((map.getCase(x, y) == 0|| map.getCase(x , y) == 4)  && (map.getCase(x + 1, y)
== 0 || map.getCase(x + 1, y) == 4 || map.getCase(x + 1, y) == 7) &&
map.getCase(x - 1, y) != 4)
{
     x++;
     fall();
}
```

- Dans la méthode move(int dir) nous avons le code suivant :

```
if (!(map.getCase(x, y) == 0 && map.getCase(x + 1, y) == 0
&& map.getCase(x - 1, y) != 4))
{
// un switch() permettant d'aller dans une direction selon l'entier dir.
}
try {
      this.cmd.wait();
} catch (Exception e) {e.printStackTrace;}
fall();
```

Cette structure doit donc être protégée des accès concurrents. Pour gérer ce problème, plusieurs solutions s'offrent à nous. Nous aborderons les solutions envisagées et celles retenues après avoir abordé toutes les classes principales.

4.5 La classe Bot

La classe Bot doit gérer les déplacements des ennemis du jeu, cette classe est un thread qui possède les ressources et les acteurs suivants :

Ressource	Acteur
Player p1 boolean running	

FIGURE 7 – Ressource(s) et Acteur(s) de la classe

Pour simplifier notre programme nos ennemis seront des IA avec des déplacements aléatoires. C'est selon un boolean que nous décidons si le joueur est un vrai joueur ou un robot.

4.5.1 Ses méthodes essentielles

Ce thread ne possède que sa méthode d'exécution. Nous initialisons à zéro la direction et la valeur test à 9, la valeur test n'allant que de 0 à 10. Ainsi tant que l'ennemi robotisé doit continuer à bouger, nous lançons un dé de 10 faces pour qu'il puisse sélectionner une direction au hasard, de temps aléatoire. Il peut continuer la même direction pour un laps de temps très court ou très long, ou bien changer de direction tout le temps, ce qui le rend imprévisible.

4.6 La classe Command

Cette classe est **implements KeyListener** car elle est à l'écoute des touches du clavier via une JFrame. Nous n'observons que des entrées de touches lorsque nous sommes dans cette fenêtre. Autrement, l'entrée n'est pas prise en compte pour se déplacer via les touches Z,Q,S,D.

Ressource	Acteur
Player p	keyPressed(KeyEvent e)

FIGURE 8 – Ressource(s) et Acteur(s) de la classe

4.6.1 Ses méthodes essentielles

La seule méthode essentielle que possède cette classe est la récupération des touches pour déplacer le personnage, c'est-à-dire la fonction keyPressed() ci-dessous, où nous pouvons observer le Player p1 se déplacer grâce à la fonction move() avec pour chaque direction, un entier désigné.

```
@Override
        public void keyPressed(KeyEvent e) {

                if (e.getKeyCode() == KeyEvent.VK_Z)
                        p1.move(2);

                if (e.getKeyCode() == KeyEvent.VK_S)
                        p1.move(1);

                if (e.getKeyCode() == KeyEvent.VK_Q)
                        p1.move(4);

                if (e.getKeyCode() == KeyEvent.VK_D)
                        p1.move(3);
        }
```

4.7 La classe Menu

Cette classe registre tous les menus disponibles. Lors de l'élaboration de ce projet nous avons réalisé des schémas et des listes des objectifs à réaliser. Voici le cheminement à travers les différents menus à réaliser.

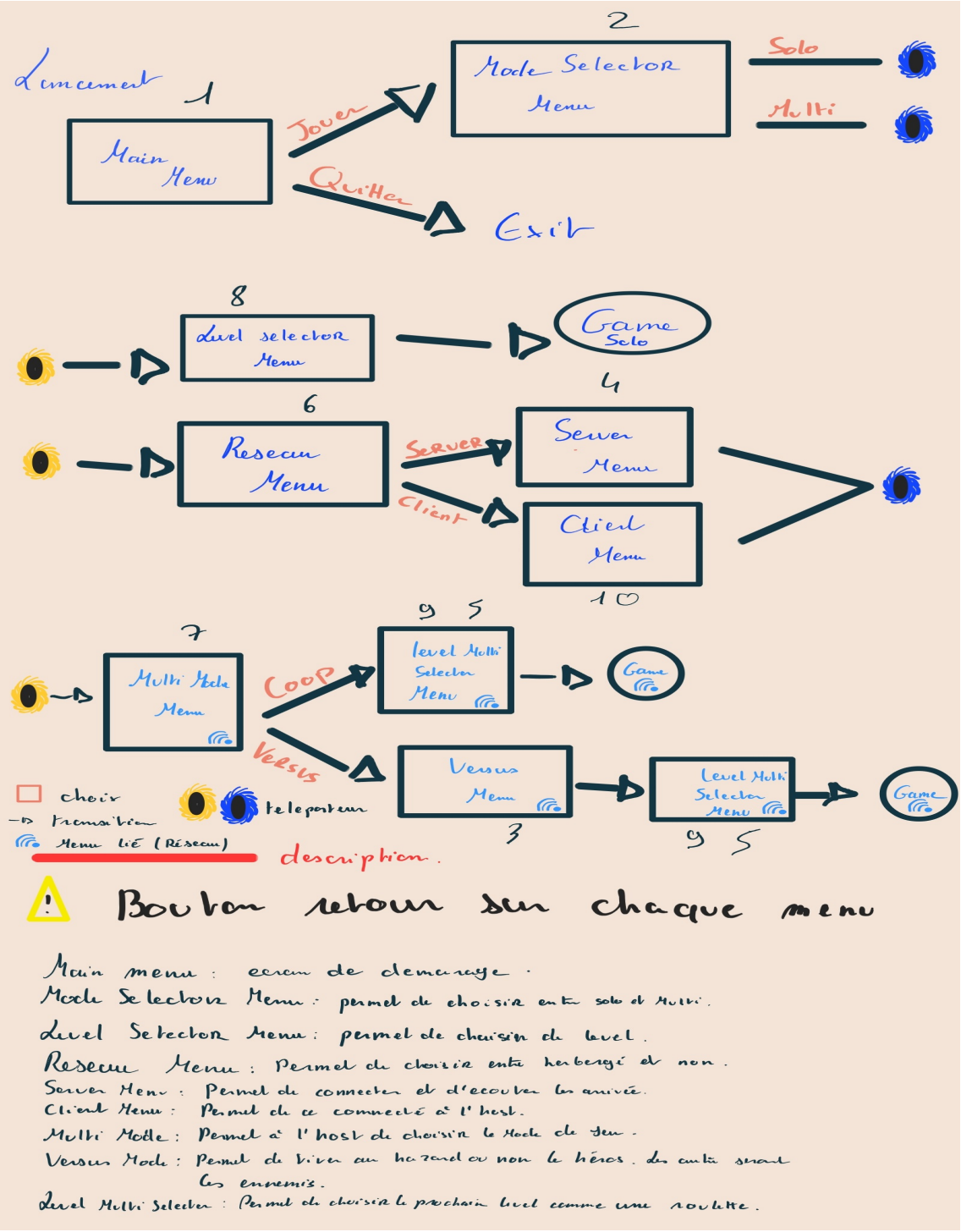

FIGURE 9 – Représentation des options des menus

Lors du mode multijoueur, nous observons après le MultiModeMenu deux options de jeu : en Coop et en Versus. Le mode Coop correspondrait à un mode de coopération entre les joueurs, le but de tous les joueurs est de récupérer les éléments ramassables du niveau en cours. Alors que le mode Versus n'intégrerait qu'un seul héros tandis que les autres joueurs joueront le rôle d'ennemi.

4.7.1 Ses méthodes essentielles

La classe Menu ne possède qu'une seule méthode assez essentielle puisqu'elle permet de naviguer à travers un menu et ses options. Cette méthode se nomme MenuManager() et est composée de simples conditions if..else.

4.8 La classe Function

Cette classe regroupe toutes les fonctions utiles à notre programme qui n'ont pas réellement de classe attribuée. Nous lui avons attribué deux méthodes. L'une de ces fonctions est ReadLevel. Une ancienne classe détachée auparavant que nous avons finalement reconsidérée.
Voici une ébauche d'un schéma de niveau avec les entiers définissants les principaux objets de notre jeu.

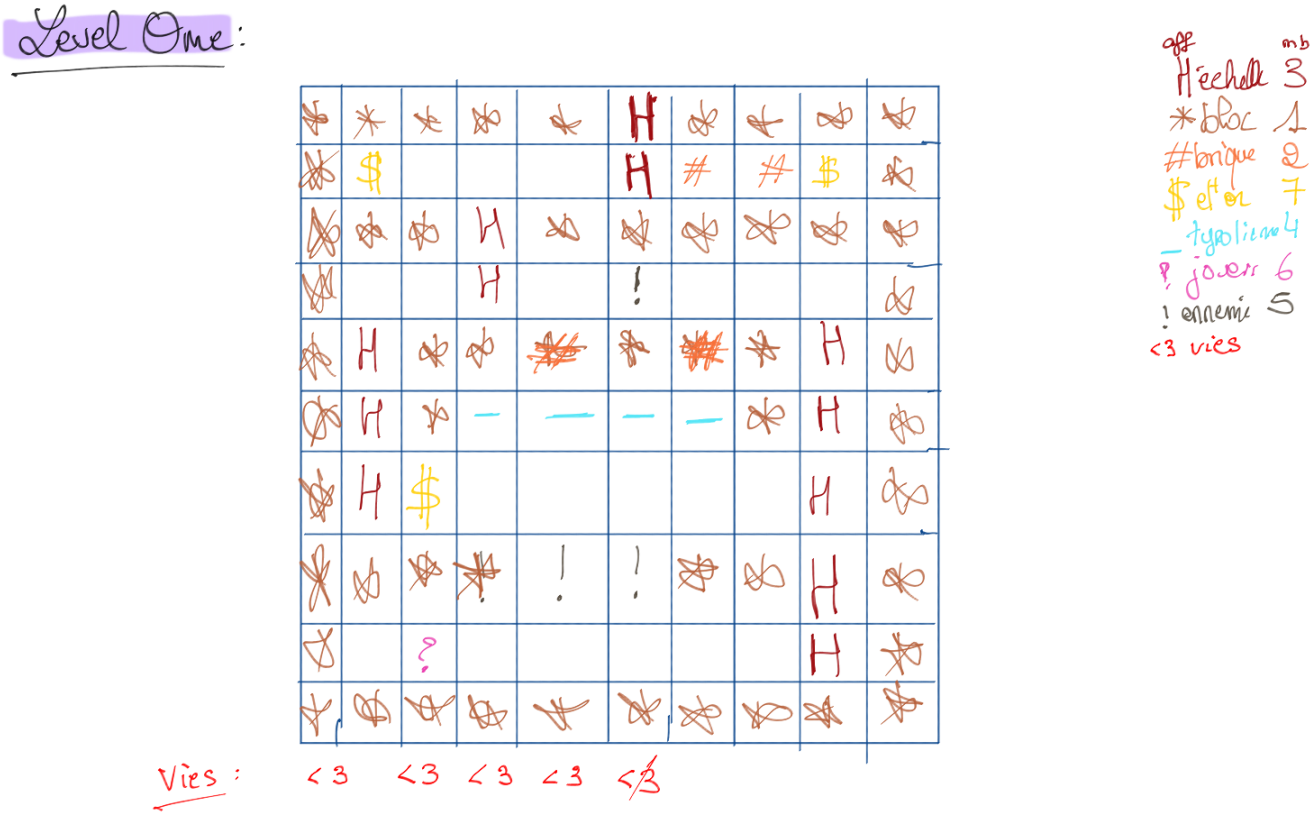

FIGURE 10 – Premier schéma de niveau

Comme vous pouvez le remarquer, il y a un détail flagrant : la taille de ce tableau laisse vraiment à désirer. Nous avions prévu du 10 par 10, ce qui différencie grandement de notre structure actuelle de 80 par 20.

4.8.1 Ses méthodes essentielles

La grande méthode de cette classe est donc ReadLevel. Cette méthode permet grâce à un chemin en paramètre, de lire la matrice d'entiers sous format .txt qui est rattachée à ce chemin.
Nous utilisons, comme vu en TD, l'introduction d'un fichier par un InputStream.

```
File file = new File(path);
InputStream fichierMatrice = new FileInputStream(file);
InputStreamReader lecture = new InputStreamReader(fichierMatrice);
BufferedReader textBuffer = new BufferedReader(lecture);
```

Remarquons également le BufferedReader qui après une boucle while dans laquelle nous allons lire ligne par ligne ce textBuffer, se fermera par l'assignation **textBuffer.close();** pour rendre à la mémoire l'allocation réalisée.

4.9 La classe Block

Cette classe est un thread avec 4 attributs. Nous retrouvons deux entiers correspondants à une localisation en x et y dans notre tableau 2D, une Map correspondant à notre niveau et un static int, c'est-à-dire un entier

non modifiable de valeur 4. Cette dernière valeur a été calculée pour obtenir à travers la méthode sleep() ; un résultat convenable.

4.9.1 Ses méthodes essentielles

Nous n'avons qu'un seul constructeur et la méthode d'exécution du thread. Dans son run() nous récupérons la brique à faire disparaître momentanément avant de la faire réapparaître.

4.10 La classe Points

Cette classe ne possède qu'un seul attribut qui est un entier et considéré comme le nombre de points que possède le joueur.

Ressource	Acteur
int pointsJoueur	addPointsJoueur()

FIGURE 11 – Ressource(s) et Acteur(s) de la classe

Nous ne faisons qu'ajouter des points supplémentaires lorsque le joueur se positionne sur les coordonnées d'un élément à ramasser.
Cependant, nous aurions pu émettre un système de soustraction des points, par exemple lorsque le joueur touche un ennemi ou bien s'il ramasse un élément à collecter considéré comme piège, nous aurions pu avoir une fonction de régression des points. De tel sorte que nous aurions eu un problème de concurrence que nous aurions pu agréablement régler par un moniteur en ajoutant un synchronized au deux fonctions.

4.10.1 Ses méthodes essentielles

La classe possède 2 méthodes dans notre jeu : une méthode pour ajouter des points à notre ressource. Ainsi qu'une méthode d'affichage que nous retrouverons en dessous du niveau joué lors de notre partie.

4.11 La classe Vie

La classe Vie se compare quelque peu à la classe Points. Elle ne possède qu'une seule ressource, un entier, ici considérée comme le nombre de vies d'un joueur. Ce nombre de vies est modifiable, par défaut il est élevé à 5 vies.

Ressource	Acteur
int vieJoueur	toucheEnnemi()

FIGURE 12 – Ressource(s) et Acteur(s) de la classe

4.11.1 Ses méthodes essentielles

Cette classe ne possède que deux méthodes : une méthode d'affichage semblable à celle de la classe Points, mais également une méthode concernant la collision du joueur avec un ennemi qui provoque la perte d'une vie. Á nouveau, comme pour la classe précédente, nous aurions pu ajouter un système "opposé" en ajoutant des coeurs en ramassant par exemple un élixir de soin ou complètement en ajoutant une nouvelle touche au jeu : un marché.
Ainsi tous les deux ou cinq niveaux, nous aurions un passage chez le marchand avec lequel nous pourrions échanger de la vie supplémentaire en échange de points disponibles, ou bien choisir une mission à remplir et obtenir une récompense (vie supplémentaire ou points supplémentaires). Des exemples de missions auraient été : cassez 50 blocs, tuez 2 ennemis (en les bloquant via des briques), réalisez le niveau suivant en moins d'une minute, etc...

Tout cela pour dire, que nous aurions pu nous occuper de gérer d'autres problèmes d'accès concurrents à une même ressource commune. Mais parfois, autant ne pas se compliquer la vie.

4.12 La classe Serveur

Avant de débuter la classe Serveur qui sera suivie de la classe Client, laissez-nous vous introduire la notion de socket d'un système d'exploitation.

Un socket est une interface logicielle qui permet d'utiliser les services d'un protocole réseau. Nous avons deux types de sockets : TCP et UDP. Nous nous servirons du socket TCP car celui-ci assure un service de transmission des données fiable avec une détection et correction d'erreurs de bout en bout.
Toutes les entrées et les sorties sont gérés par des flux (comme InputFileStream), mais également tel que la lecture de saisie du clavier, la lecture et l'écriture dans un fichier, l'échange de données réseau, etc...
Un flux est donc une série de données envoyée sur un canal de communication entre deux machines.

Pour reprendre, cette classe Serveur possède un attribut de type final static int qui correspondra au port du serveur. Tant que l'entrée au serveur est accessible aux clients nous continuons d'être à l'écoute d'une possible connexion à laquelle nous attribuerons un ID pour pouvoir les différencier grâce à l'instance ServerSocket. Nous envoyons par la suite les données nécessaires au joueur pour s'intégrer à la partie et savoir jouer. Il est important de penser à importer la classe java.io !

4.12.1 Ses méthodes essentielles

La classe Serveur ne possède qu'un main() qui lui permet de créer un socket et d'être à l'écoute des nouvelles connexions possibles. Il envoie également les données dès lors d'une nouvelle connexion et ferme bien tout lors de la fin du programme avec la méthode .close() ; qui permet de rendre l'allocation et de ne pas provoquer de fuite.

4.13 La classe Client

La classe Client possède un attribut de type static int qui est équivalent au port de connexion du serveur. Tant que le serveur est à l'écoute d'une possible connexion nous pouvons demander la création d'un socket à destination du serveur avec le bon port. Nous n'avons plus qu'à jouer une fois le lien établit.
Dans le mode multijoueur, les connexions clients pourront soit jouer le héros ou bien un ennemi.

4.13.1 Ses méthodes essentielles

La classe Client ne possède qu'un main() également, qui lui permet de se connecter à un serveur et d'accéder à toutes les données nécessaires pour jouer.

5 Description des structures de données

Après avoir introduit les différentes classes de notre programme, nous avons pu observer les différents problèmes de concurrences qui nous ont mené à choisir une solution particulière pour chaque cas. Ci-dessous, les solutions envisagées pour répondre à ces problèmes mais également celles qui ont été retenues et pourquoi.

5.1 Solutions envisagées

- ◆ Concernant
- ✪ Concernant
- ➽ Concernant l'affichage des objets de la map, nous aurions voulu représenter les éléments à collecter sous le caractère '€'. Hors Windows n'apprécie pas grandement ce caractère.

5.2 Solutions retenues

- ▮ Concernant la section critique de la classe Affichage, nous avons décidé d'utiliser un verrou pour l'affichage du jeu.
- ✻ Concernant la méthode fall() et la méthode move(int dir) qui réquisitionnent toutes deux le déplacement du joueur. Pour faire en sorte que l'une puisse se terminer complètement avant de passer la main à l'autre. Nous avons décidé de synchronized les deux méthodes.
- ➽ Nous avons donc troquer l'affichage pour le caractère le plus ressemblant de l'alphabet : la lettre E. (comme Eléments, Euro, Etc..)

6 Architecture logicielle détaillée

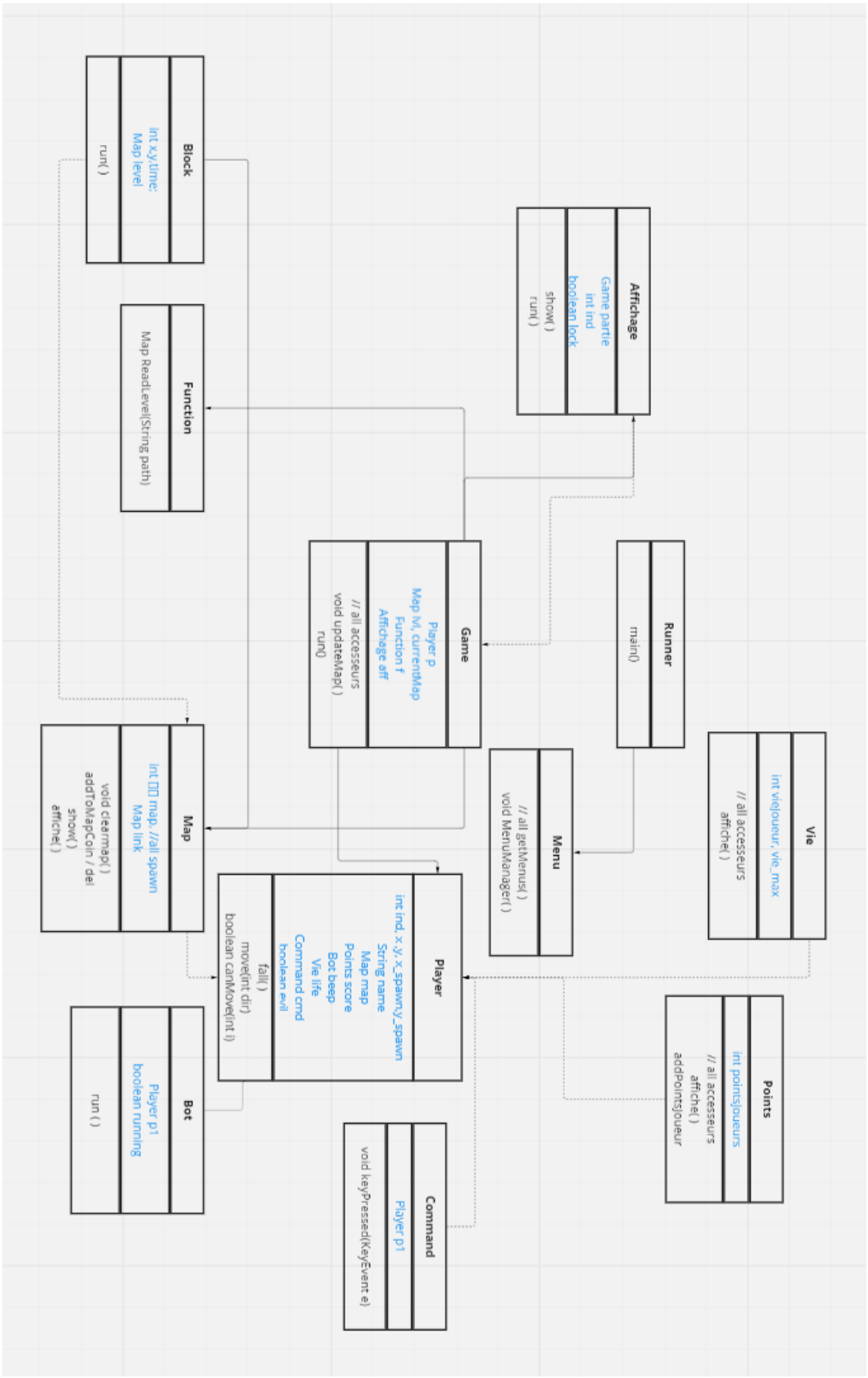

FIGURE 13 – Ressource(s) et Acteur(s) de la classe

7 Jeu de test

Imaginons que vous souhaiteriez lancer une partie de jeu sur ce programme sorti de nulle part. Sans aucune notice par ailleurs.

Lorsque vous lancez l'exécutable, votre invité de commandes va vous présenter un menu d'accueil. Vous devrez appuyez sur la touche numérotée correspondante au prochain menu vers lequel vous voulez vous dirigez. Vous aurez accès ici au menu d'accueil qui vous propose deux cheminements :

- ➥ En entrant le numéro 1, vous voulez jouer une partie, donc vous serez diriger vers un second menu pour choisir votre mode de jeu.
- ➥ En entrant le numéro 2, vous souhaitez quitter le jeu. Le programme s'achèvera ici.

Nous appuyons donc sur la touche "1", nous voilà de nouveau face à des choix :

- ➥ En entrant le numéro 1, vous voulez jouer une partie en mode solo, vous serez le seul joueur contre des ennemis à intelligence artificielle (d'où les déplacements douteux).
- ➥ En entrant le numéro 2, vous souhaitez jouer une partie en mode multijoueur : vous serez plusieurs joueurs.
- ➥ En entrant le numéro 3, vous reviendrez au menu précédent.

❒ Pour une partie en mode solo :

Il est donc temps de choisir un niveau à jouer ! Vous pouvez directement choisir votre niveau ou laissez le sort se charger de votre destin. Ou ne pas faire confiance au destin et choisir de retourner au menu précédent. Donc nous récapitulons :

- ➥ Entrez 1, pour choisir le niveau du jeu.
- ➥ Entrez 2, pour un niveau aléatoire.
- ➥ Entrez 3, vous reviendrez au menu précédent.

Après avoir attribué un niveau, c'est le moment de pouvoir s'amuser et augmenter sa réflexion en faisant en sorte de ramasser tous les éléments affichés par "E" afin d'accéder au niveau suivant ! C'est par la suite une récursion sans fin, à vous de faire le meilleur score !

Lors de la fin d'une partie, c'est-à-dire lorsque vous n'avez plus de vie, vous pouvez demander un fichier texte qui relatera votre suivi de partie et le temps que vous y avez consacré.

❒ Pour une partie en multijoueur :
Nous atterrissons face à un nouveau menu qui vous fait face à deux choix :

- ➥ Entrez le numéro 1, pour un jeu en coopération.
- ➥ Entrez le numéro 2, pour un jeu en mode combat.
- ➥ Entrez le numéro 3, vous reviendrez au menu précédent.

En coopération, nous avons nos quatre héros qui doivent accumuler tous les points. Tous les joueurs jouent le rôle de héros. Après connexion et hébergement des clients sur le serveur, il n'y aura plus qu'à avoir accès au menu de sélection du niveau en multijoueur et la partie est lancée !

En combat, nous avons un héros qui aura le même but mais avec à charge, les autres joueurs avec rôles d'ennemis (ils auront les commandes d'un vrai joueur).

Par la suite nous n'avons pas fini d'implémenter la partie multijoueur, mais nous avons essayé d'être le plus clair possible dans son fonctionnement.

8 Conclusion

Réalisons un petit récapitulatif de ce programme en se basant sur les 5 propriétés pour juger un système d'exploitation.

♦ Concernant l'Efficacité :

Les ressources utilisées ont été assez minimisées et les temps de réponse sont rapides. L'utilisateur n'a

pas à attendre (sauf pour attendre les connexions en multijoueur évidemment - ce n'est pas un problème machine pour cela -). Nous pouvons également lancer plusieurs joueurs en même temps sans aucun souci.

♦ Concernant l'Ouverture :

Nous pouvons communiquer avec d'autres machines grâce au mode multijoueur. L'important est simplement d'utiliser une machine avec un clavier. Autrement nous n'avons pas spécifier d'autres touches pour un autre type de console mais nous pourrions améliorer cela, en programmant une connexion avec une manette de Xbox et son fonctionnement.

♦ Concernant la Souplesse :

Ce jeu peut se jouer sur d'autres architecture différentes (Linux,Mac,Windows...), d'autant plus que la conception des claviers ne diffère pas tant que cela. La version de java doit cependant être à jour, pour que nous puissions utiliser les classes d'importations nécessaires.

♦ Concernant la Fiabilité :

Le protocole réseau étant respecté, il ne devrait y avoir aucune conséquence, cependant nous pourrions nous attendre à un décalage horaire entre les différentes machines. Ce qui pourrait entacher la progression des différents joueurs.

♦ Concernant l'Ergonomie :

Le programme permet grâce à l'interface visuelle du menu d'accueil, d'intégrer facilement les touches nécessaires pour jouer. L'affichage du niveau n'est certes pas très agréables à regarder (ce ne sont pas de réels graphismes) cependant il est suffisant pour que l'on puisse différencier les différents objets et personnages du niveau.

Ce projet en langage java a été des plus ressourçant. N'ayant pas l'habitude de programmer ce type de code, nous avons appris de nouveaux éléments et bien plus encore, le fonctionnement d'un serveur réseau et le fonctionnement des connexions qui s'y lient. Ce qui est très passionnant. Au début, cela était assez difficile à cerner, mais au bout du compte, rien de bien méchant à intégrer. C'est un projet qui a demandé beaucoup de temps, mais qui avec quelques semaines en plus, aurait intégré une toute nouvelle évolution du jeu (sur des machines bien plus performantes à notre époque).

Un énorme merci à nos professeurs qui nous ont encadré dans ce module !

Remerciements :

M. LECLERCQ
Mme. GILLET
M. HAMIDI