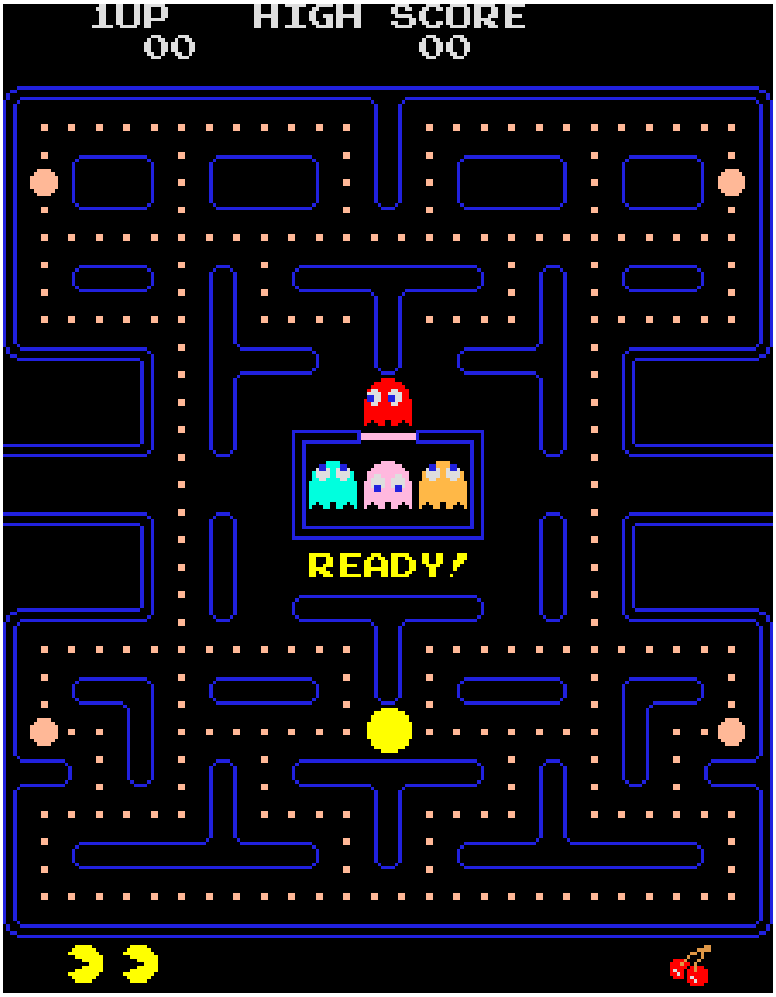


Projet Pac-man

POO EN JAVA

Matthieu CHERRIER | Jennifer MARTINEZ

Table des matières

I – Avant-propos

Nous sommes deux étudiants en L3 informatique, chacun ayant un parcours différent, ainsi que des compétences qui nous sont propres. Malheureusement, chacun de nous avons des lacunes en Programmation Orientée Objet, ce qui nous a conduit à faire de multiples tentatives infructueuses avant de parvenir à un résultat.

Nous avons fait un premier essai, quasi-fonctionnel bien que brouillon. Toutefois, en cours de route, nous nous sommes rendus compte que notre programme ne respectait pas vraiment les standards de la Programmation Orientée Objet, notamment le modèle Modèle-Vue-Contrôleur, vu en cours. C'est pour cette raison que nous avons préféré reprendre le projet à zéro, pour repartir sur de bonnes bases.

II – Notre premier Pac-man

1°) Liste des fonctionnalités

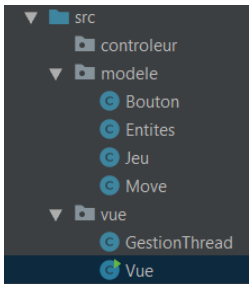
Notre premier Pac-man est quasi-fonctionnel, au lancement de l'application, une fenêtre s'affiche avec un bouton Play dessus, en appuyant sur celui-ci le jeu se lance.

Nous contrôlons donc Pac-man, représenté par un rond jaune, qui peut se déplacer librement dans un labyrinthe composé de murs et de couloirs. Pac-man se déplace entre chaque case de couloir, mais est bloqué lorsqu'il essaie de traverser un mur. Il peut également passer par les tunnels aux extrémités du labyrinthe, ce qui a pour effet de le téléporter à l'autre bout de la carte.

Lors de ses déplacements, Pac-man croisera sur chaque case de son chemin des Pac-gommes et des Super Pac-gommes, qu'il ramassera automatiquement. Malheureusement, ces dernières n'ont pas d'effet modificateur rendant le personnage invincible, ni d'incrémentation d'un score.

Finalement, chaque partie contient 4 fantômes, qui se déplacent de manière aléatoire dans tout le labyrinthe. A défaut d'avoir réussi à implémenter des threads pour chacun des fantômes, nous les faisons se déplacer à chaque fois que le joueur se déplacera lui-même. A noter que, malgré nous, les fantômes persistent à dévorer les Pac-gommes/Super Pac-gommes, comme le fait notre personnage. De plus, les collisions avec Pac-man ne sont pas fonctionnelles.

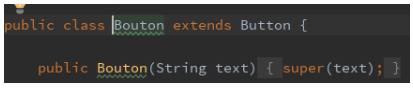
2°) Exploration du code.



Capture d'écran 1 - Arborescence.

La capture d'écran 1, expose l'arborescence de notre première version du code. Nous n'avons pas utilisé le package "controleur" car, finalement, nous voulions l'incorporer avec le package vue. Dans le package "modele", nous avons créé quatre classes.

Une classe "Bouton", qui ne servait qu'à créer le bouton "Play" du début, en héritant des fonctionnalités de la classe déjà existante "Button".



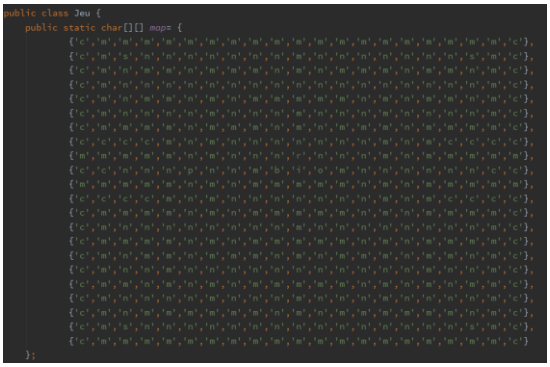
Capture d'écran 2 – Classe Bouton

```java
public class Entites {

    private static int width = 30;
    private static int height = 30;
    private static int widthGomme = 10;
    private static int heightGomme = 10;

    public static Rectangle mur(int i, int j) {
        Rectangle rectangle = new Rectangle(width,height);
        rectangle.setFill(Color.BLUE);
        rectangle.setLayoutX(i);
        rectangle.setLayoutY(j);
        return rectangle;
    }
```

Capture d'écran 3 – Classe Entites

La classe “Entites”, concrètement, elle initialise les différentes entités : les couloirs, les murs, le Pac-man, les gommes et les fantômes. Elle leur associe également une forme.

Dans “Jeu”, nous stockons uniquement le tableau de caractère définissant où se trouve chaque entité sur la grille, chacun ayant un caractère qui lui est propre.



Capture d'écran 4 – Classe Jeu

La classe “Move”, contient les différentes fonctions qui permettent à notre Pac-man et nos fantômes d’avancer.

```java
private static int[] getPMPosition(char who)
    char[][] map = Jeu.map;
    int i, j;
    int[] r = {-1, -1};

    for(i=0; i<map.length; i++) {
        for(j=0; j<map[0].length; j++) {
            if(map[i][j] == who) {
                r[0] = i;
                r[1] = j;
            }
        }
    }

    return r;
}
```

Capture d'écran 5 –
Fonction getPMPosition

Nous avons ainsi codé une fonction permettant de récupérer la position des personnages, nous parcourons le tableau de caractères “map” initialisé dans “Jeu” et grâce au caractère qui leur a été attribué, nous pouvons savoir quelle est la position de chacun d’entre eux.

```java
public static void goUP(char who) {
    int[] pos = getPMPosition(who);
    if(!isWall( dir: "u", who)) {
        Jeu.map[pos[0]][pos[1]] = 'c';
        pos[0] -= 1;
        move(who, pos);
    }
```

Capture d'écran 6 – Fontion goUP

Nous avons ensuite codé quatre fonctions remplaçant, à chaque mouvement, l’ancienne position des personnages par un couloir. Le caractère de la nouvelle position est, quant à lui, remplacé par le personnage correspondant, supprimant ainsi la gomme, s’il y en avait une. Pour ce faire, nous récupérons la position, vérifions que le mouvement est possible, c’est-à-dire, qu’il n’y a pas un mur, grâce à la fonction “isWall”, qui vérifie si la prochaine coordonnée contient un mur ou non et retourne vrai ou faux en fonction. Ensuite, nous modifions le tableau de caractères pour correspondre, par le biais de la fonction “move”, qui détermine quel est le personnage et y place son caractère.

```java
private static boolean isWall(String dir, char who) {
    int[] pos;
    switch (dir) {
        case "u":
            pos = getPMPosition(who);
            if(Jeu.map[(pos[0]-1)][pos[1]] == 'm') {
                return true;
            }
            break;
```

Capture d'écran 7 – Fonction isWall

```java
public static List<Character> availableMoves(char who) {
    int[] pos = getPMPosition(who);
    List<Character> r = new ArrayList<>();
    if(Jeu.map[pos[0]+1][pos[1]] == 'c' || Jeu.map[pos[0]+1][pos[1]] == 's' || Jeu.map[pos[0]+1][pos[1]] == 'n') {
        r.add('d');
    }
```

Capture d'écran 8 – Fonction availableMoves

Pour finir, la fonction “availableMoves”, détermine les directions disponibles parmi les quatre et les stocke afin que les fantômes puissent en choisir une. Toutefois, elle ne gère pas le cas où aucune direction n’est disponible (un fantôme coincé entre trois murs et un autre fantôme, par exemple).

Dans le package “vue”, nous avons une classe fonctionnelle : “Vue”, elle contient les fonctions principales de notre Pac-man. Une fonction “initialiser”, qui créée un tableau de rectangles de même taille que le tableau de caractères contenu dans “Jeu”. Elle va pouvoir ainsi déterminer quelle entité insérer et à quel endroit.

```java
public static Rectangle[][] initialiser() {
    char[][] map = Jeu.map;
    Rectangle[][] r = new Rectangle[map[0].length][map.length];
    for(int i=0; i<map.length; i++) {
        for(int j=0; j<map[0].length; j++) {
            switch (map[i][j]){
                case 'm':
                    r[j][i] = Entites.mur(i, j);
                    break;
```

Capture d'écran 9 – Fonction initialiser

```java
@Override
public void start(Stage primaryStage) {

    GridPane gPane = new GridPane();

    BorderPane border = new BorderPane();

    border.setLeft(bouton);
    border.setCenter(gPane);

    stage = primaryStage;
    Scene s = new Scene(border, width: 800, height: 800, Color.BLACK);
    stage.setTitle("Pac-Man");
    stage.setScene(s);
    stage.show();

    bouton.setOnAction((event) -> {
        System.out.println("Bouton Play activé");
        update(gPane);
    });

    s.addEventHandler(KeyEvent.KEY_PRESSED, (key) -> {
        if(key.getCode()==KeyCode.UP) {
            Move.goUP( who: 'p');
            ghostMove();
            update(gPane);
        }
```

Capture d'écran 10 – Fonction Override start

La fonction “start” override de la classe déjà existante “Application”, va créer notre grille, faire les ajouts qu’il faut (bouton, etc.) et gérer les évènements possibles. Ceux-ci étant : un clique sur le bouton Play, permettant l’apparition du plateau, et la pression d’une flèche directionnelle, on détermine ensuite quelle flèche est pressée et on y associe quel mouvement faire.

```java
private static void randomMove(List<Character> availableMoves, char who) {
    Random rand = new Random();
    Character randomElement = availableMoves.get(rand.nextInt(availableMoves.size()));
    switch (randomElement) {
        case 'u':
            Move.goUP(who);
            break;
public static void ghostMove() {
    List<Character> o = Move.availableMoves( who: 'o');
    randomMove(o, who: 'o');
```

Capture d'écran 11 – Fonctions randomMove & ghostMove

Les fonctions “ghostMove” et “randomMove” permettent de déterminer aléatoirement quelles directions les fantômes doivent prendre respectivement.

Pour finir, nous avons la fonction “update”, qui met à jour la vue de la grille. A noter que le “main” contient uniquement le “launch(args)”.

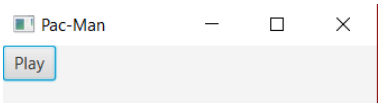
```java
private static void update(GridPane gPane) {
    Rectangle[][] rectangle = initialiser();
    for(int i=0; i<rectangle.length; i++) {
        for (int j=0; j<rectangle[0].length; j++) {
            gPane.add(rectangle[i][j], i, j);
```

Capture d'écran 12 – Fonction update

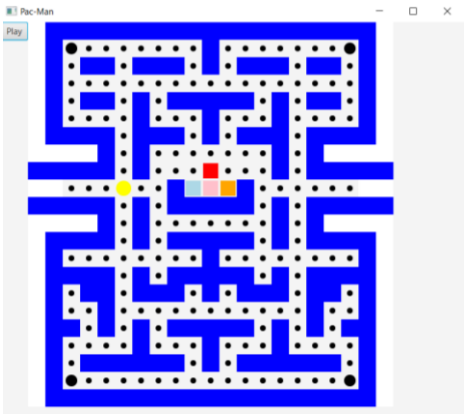
Nous avons également créé une classe “GestionThread” ainsi que l’ajout du code le lançant dans la vue mais cela s’est avéré un échec. Le thread tourne bien mais dès que nous voulons mettre la vue à jour, l’application crash.

Afin d’essayer de régler le problème des fantômes mangeant les gommes, nous avons tenté de stocker l’ancien contenu de la position où se trouve les fantômes actuellement et de la réinjecté au départ du dit fantôme (nous n’avons essayé que pour le fantôme orange). Cependant, nous n’arrivions pas à stocker correctement les informations, tout du moins, à ce qu’elles soient utilisées au bon moment. Nous avons également eu des problèmes de rafraîchissement de la grille, le fantôme restant en arrière-plan de la gomme et cette dernière se décalant dans la grille. C’est à ce moment-là (approximativement deux semaines après le début du projet) que nous avons réalisé notre erreur et sommes partis sur autre chose.

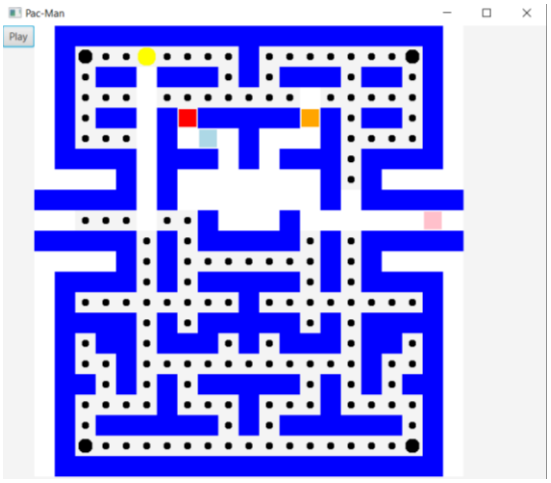
3°) Captures d’écran au lancement de l’application



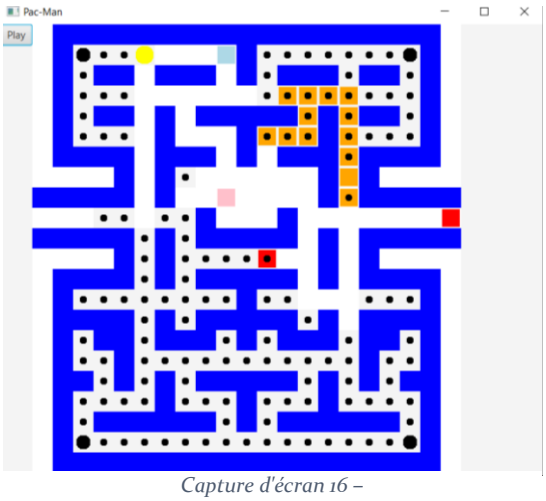
Capture d'écran 13 – Bouton Play



Capture d'écran 14 – Apparition de la grille



Capture d'écran 15 - Déplacements



Capture d'écran 16 –
Essai infructueux sur le fantôme orange

4°) Diagramme de classes

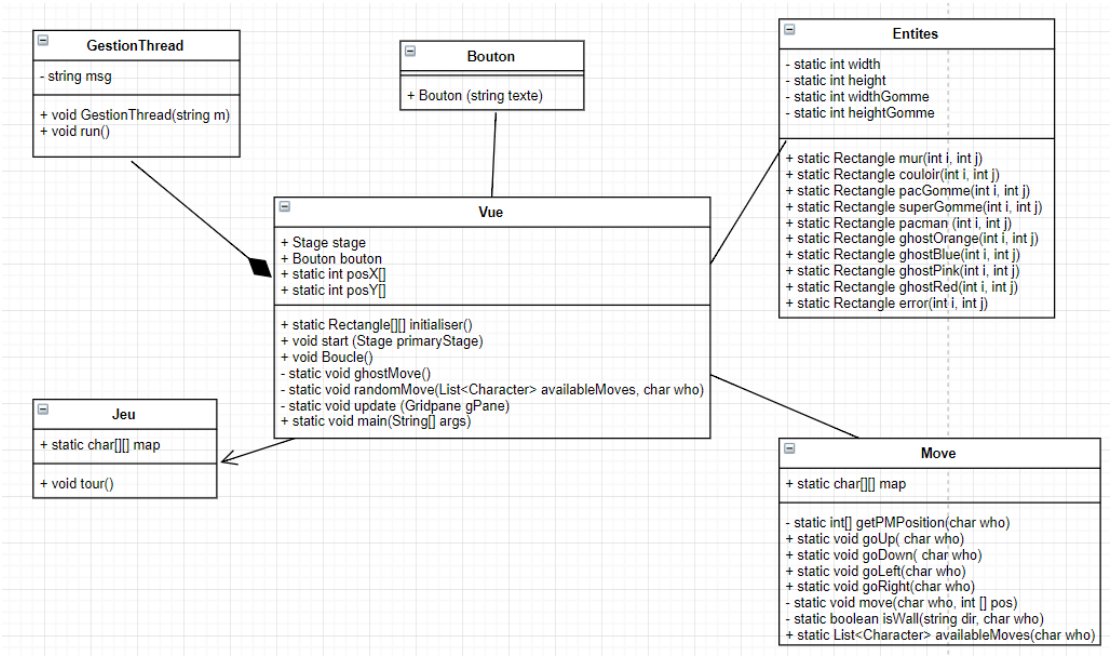


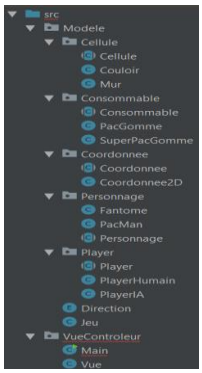
Diagramme 1 – Diagramme de classes Pac-man V1

5°) Conclusion

Notre premier essai, bien que quasi-fonctionnel, est beaucoup trop brouillon pour être considéré comme fini. Avec un peu de temps nous aurions pu aisément régler les derniers problèmes qui lui incombent, mais nous avons préféré le consacrer à refaire un projet neuf, en se basant sur ce que nous avons appris du premier.

III – Notre second Pac-man

1°) Liste des fonctionnalités



Capture d'écran 17 - Arborescence(2)

Lors de notre deuxième essai, nous avions une arborescence beaucoup plus axé MVC. Ainsi, notre package “VueControleur”, ne contient que deux classes, une classe “Main” et une “Vue”. Par manque de temps nous n’avons pas implémenter grand-chose dans cette dernière. Toutefois, “Main” contient quelques éléments d’implémentation. Nous avons essayé de faire un affichage, malheureusement, seule une fenêtre vide s’ouvre.

Dans cette version nous avons essayé de gérer l'update avec la classe déjà existante "Observable" et sa méthode "Observer". Toutefois, lorsque nous voulons créer notre grille, l'application plante. Pour ce faire, nous nous sommes basés sur le code fourni pour le rafraîchissement de la vue. Cependant, ce dernier utilisant des images et nous, non, nous voulions essayer de faire sans ImageView et ce qui en découlait. Nous avons créé une fonction "initialiser" pour sortir un morceau du code et voir si cela était la cause du crash, ce fut le cas.

```
@Override
public void start(Stage primaryStage){

    //tabC = initialiser();

    Coordonnee2D currentPos = new Coordonnee2D(tmpX, tmpY);

    Observer o =  new Observer() {
        @Override
        public void update(Observable o, Object arg) {
            for (int i = 0; i < tabC.length; i++) {
                for (int j = 0; j < tabC[0].length; j++) {
                    if (tabC[j][i] instanceof Couloir) {
                        for (int iterator = 0; iterator < 5; iterator++) {
                            if (((Couloir) tabC[j][i]).listPlayer.get(iterator) instanceof PlayerHumain) {
                                ((Couloir) tabC[j][i]).addPlayer(p);
                                currentPos.setY(i);
                                currentPos.setX(j);
                            }
                        }
                    }
                }
            }
        }
    };

    jeu.addObserver(o);
```

Capture d'écran 18 – Fonction override update

```
Mur@71103db5
Mur@58805017
Exception in Application start method
Exception in thread "JavaFX Application Thread" java.lang.NullPointerException
```

Capture d'écran 19 – Erreur produite par le code associé à initialiser

```
Couloir@6fa47932
Mur@179d72ee
```

Capture d'écran 20 – Création des Cellules ?

```
public abstract class Cellule extends Node {
    protected Coordonnee coor;

    public void setCoor(Coordonnee coor) {
        this.coor = coor;
    }

    public Coordonnee getCoor() {
        return coor;
    }
```

Capture d'écran 21 – Classe abstraite Cellule

Dans le package "Modele", nous avons créée des classes mères desquelles découlent toutes nos classes concrètes (ou presque). C'est ainsi que fonctionne la plupart de nos classes. Par exemple, la classe abstraite "Cellule" a pour enfant "Mur" et "Couloir", toutes deux sont des Cellules mais ont des comportements différents. Ainsi, Couloir, contenant plus de possibilités d'altérations que Mur, est plus riche dans le contenu. Elle va ainsi avoir des fonctions pour ajouter un Player, un Consommable, etc.

```
public class Couloir extends Cellule {

    public List<Player> listPlayer;
    public Consommable conso;

    public Couloir()
    {
        listPlayer = new ArrayList<>();
        conso = null;
    }

    public void addConsommable(Consommable conso) { this.conso = conso; }

    public void removeConsommable() { conso = null; }

    public void addPlayer(Player p) { listPlayer.add(p); }

    public void removePlayer(Player p) { listPlayer.remove(p); }

    public Consommable getConso(){ return conso;}
```

Capture d'écran 22 – Classe concrète Couloir

```
public void appliquerEffetSPG(Player p)
{
    ((PacMan) p.p).invulnerable = true;
    ((PacMan) p.p).duree=duree;
public abstract class Player {
    public Direction dir;
    public Personnage p;
```

Capture d'écran 23 – Extrait fonction appliquerEffetSPG et classe abtraite Player

Les différences avec les autres classes apparentées, viennent surtout de leur fonction dans notre jeu. C'est pourquoi SuperPacGomme aura une fonction pour appliquer un effet. Player détermine le Personnage et bien d'autres choses.

Pour le reste, “Direction” est un “enum” car il ne contient que l’énumération de nos directions.

```
public void mainTurn() {
    finish = false;
    while (!finish) {
        for (Player p : listPlayer) {
            deplacerPlayer(p);
            decrementePlayerEffect(p);
        }
        finish = gameFinish();
        setChanged();
        notifyObservers();

        try {
            sleep( millis: 350);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

Capture d'écran 24 – Fonction mainTurn

“Jeu”, quant à elle, est plus riche. Elle contient les fonctions principales et primordiales à notre jeu. “MainTurn” appelle les fonctions pour déplacer et pour décrémenter les effets, tout cela, tant que personne ne lui indique que la partie est finie. Pour ce faire, nous avons la fonction “gameFinish”, celle-ci détermine s’il reste des gommes sur le plateau, si non, la partie est finie ou bien, si pac-man se déplace sur un Fantôme, auquel cas, la partie est également terminée.

```
private void decrementePlayerEffect(Player p) {
    for (int i = 0; i < ((PacMan) p.p).duree; i++) {
        ((PacMan) p.p).duree -= 1;
        if (((PacMan) p.p).duree == 0) {
            ((PacMan) p.p).invulnerable = false;
```

Capture d'écran 25 – Fonction decrementePlayerEffect

La fonction “decrementePlayerEffect”, décrémente la variable qui détermine la durée de l’effet, si celle-ci tombe à 0, elle ré-assigne la valeur “false” à l’effet du player. “deplacerPlayer”, permet de déterminer quel est le Player sur la case (Fantôme ou Pac-man), elle récupère ainsi le prochain mouvement de celui-ci et si le déplacement est possible (pas de mur ni de collision), le bouge. Elle a aussi pour fonctionnalité de retirer les gommes du passage de Pac-Man et ainsi de choisir l’effet si c’est une super.

```
public Coordonnee2D translatedirection(Direction d, Coordonnee2D currentpos) {
    int x = currentpos.getX();
    int y = currentpos.getY();
    Coordonnee2D coorToGo = new Coordonnee2D(x, y);

    switch (d) {
        case TOP:
            y -= 1;
            coorToGo.setY(y);
            break;
```

Capture d'écran 26 – Fonction translatedirection

“translateDirection”, détermine à partir d’une direction et d’un emplacement quelle est la coordonnée d’arrivée.

“collision” détermine juste si les directions du Pac-Man et des Fantômes ne se rejoignent pas. Elle renvoie un true si c’est le cas, false, sinon. Nous avons également voulu utiliser l’interface Runnable, sans grand succès.

2°) Diagramme de classes

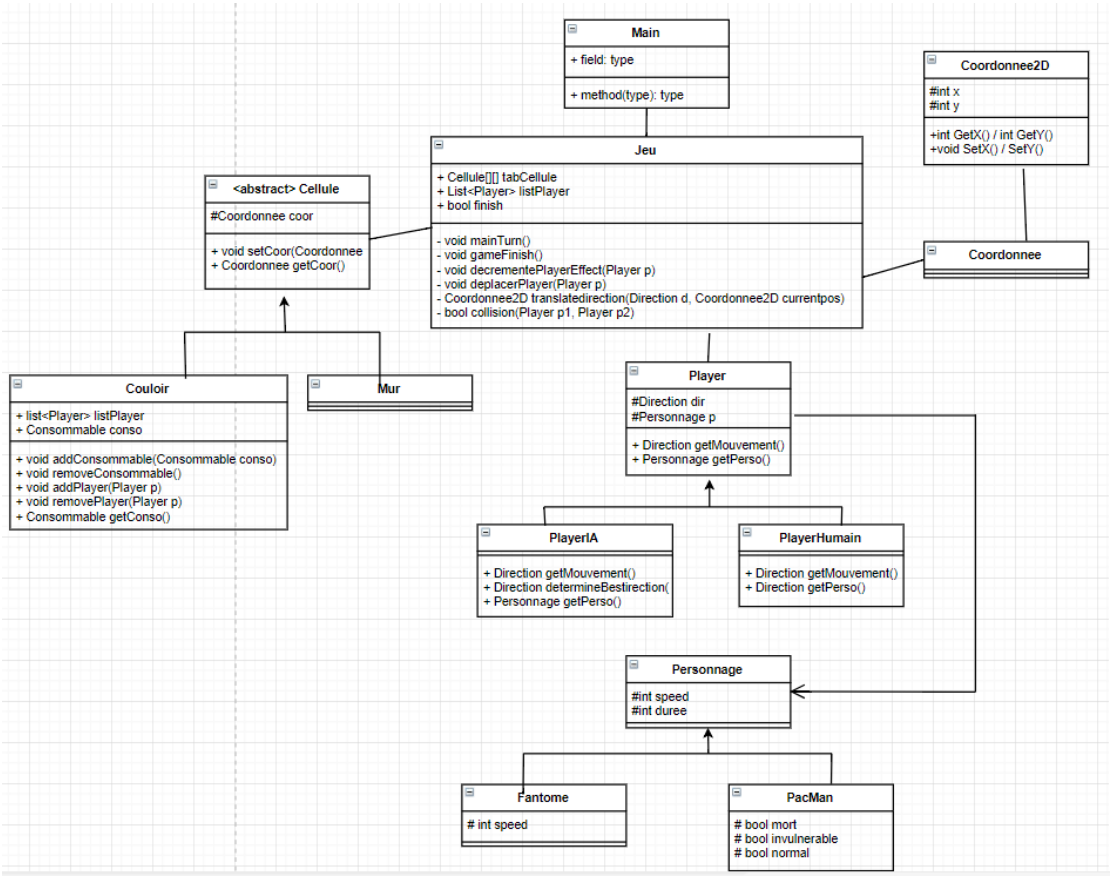


Diagramme 2 – Diagramme de classe Pac-man V2

3°) Conclusion

Si nous avions eu plus de temps ou mieux compris le MVC et l'orienté objet, nous aurions aimé implémenter bien d'autres choses, malheureusement, ce n'est pas un sujet qui nous est évident et nous avons grandement bataillé pour en arriver à ce résultat-là, ne ménageant pas nos efforts. Toutefois, cela nous a permis de nous remettre en question et de mieux comprendre le sujet que si nous l'avions travaillé seul, de notre côté. En effet, si nous n'avions pas pu collaborer à deux, jamais nous n'aurions pu réaliser nos erreurs et ainsi pouvoir essayer de les corriger et les comprendre.