Dossier du Projet

GROUPE :
Mehdi SABER
Tanina BOUZIDI
Florian PASQUIER

Ce fichier reprend la description des documents demandés pour le projet. Vous devez le compléter et le mettre à disposition sur git.

Tous les diagrammes devront être fournis avec un texte de présentation et des explications.

1:Documents pour CPOO

Chaque diagramme utilisé à des fins de documentation doit être focalisé sur un objectif de communication. Il ne doit par exemple pas forcément montrer toutes les méthodes et dépendances, mais juste ce qui est nécessaire pour montrer ce que le diagramme veut montrer. Chaque diagramme doit être commenté.

Architecture (5/60 points)

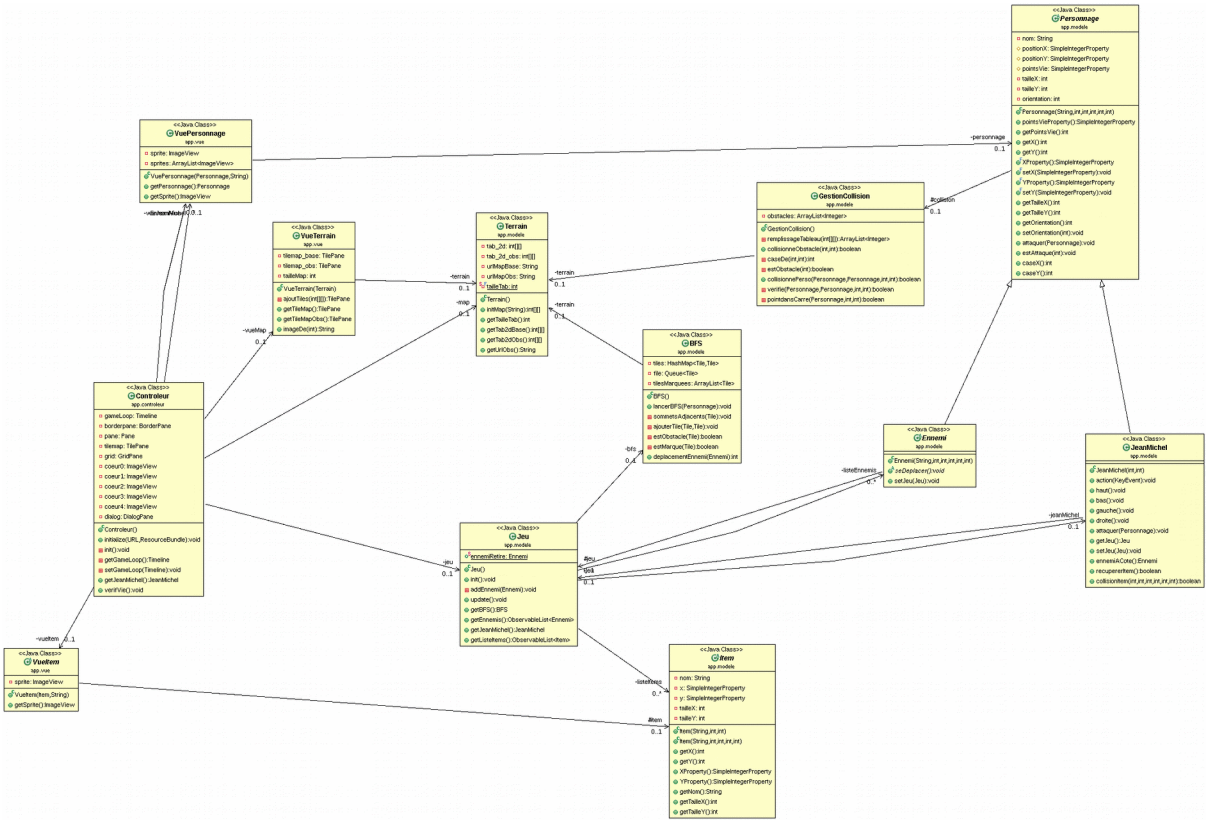
Un texte présentant vos choix principaux d'architecture, ainsi que des diagrammes de classe commentés permettant de la visualiser. Des diagrammes de paquetage peuvent être utilisés pour présenter les éléments d'architecture de haut niveau (par exemple l'architecture MVC).

L'architecture qui nous a été imposée est l'architecture Modèle-Vue-Controleur, dite MVC. Elle consiste à diviser le programme en 3 parties distinctes :

- Le Modèle : il s'agit de la partie « logique » du programme, celle où l'on crée tous les objets sans prendre en compte l'affichage
- La Vue : elle s'occupe de toute la partie graphique du jeu, et lie des objets à leur propre classe Vue.
- Le Controleur : c'est la partie qui s'occupe de lier les deux parties précédentes, en « construisant » le programme à partir du modèle et de la vue.

Nous avons donc tenté de faire un programme suivant le plus possible cette architecture. Vous trouverez ci-bas un diagramme de classe ne montrant que les classes « principales » de notre programme, afin de vous montrer comment nous avons appliqué le MVC ainsi que différentes règles de programmation comme l'héritage ou le polymorphisme.

Diagramme de classe:



Détails : diagrammes de classe (5/60points)

Vous choisirez des parties du modèle que vous considérez particulièrement intéressantes du point de vue de la programmation à objets (héritage, composition, polymorphisme…) ou de la programmation événementielle (gestion des événements, écouteurs, bindings...). Vous expliquerez vos choix et les illustrerez par des diagrammes de classe.

Diagramme du lien Vue-Objet (page suivante):

Grâce à ce diagramme nous pouvons voir que la structure MVC est respectée, puisque seules les vues ont accès à leur modèle respectif (et pas dans l'autre sens).

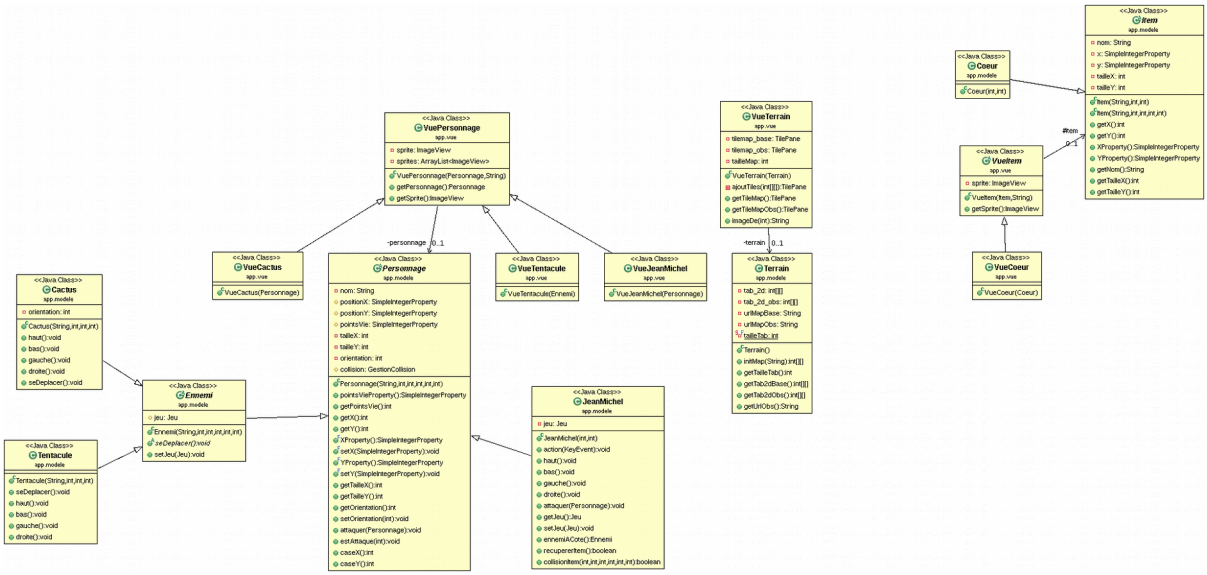
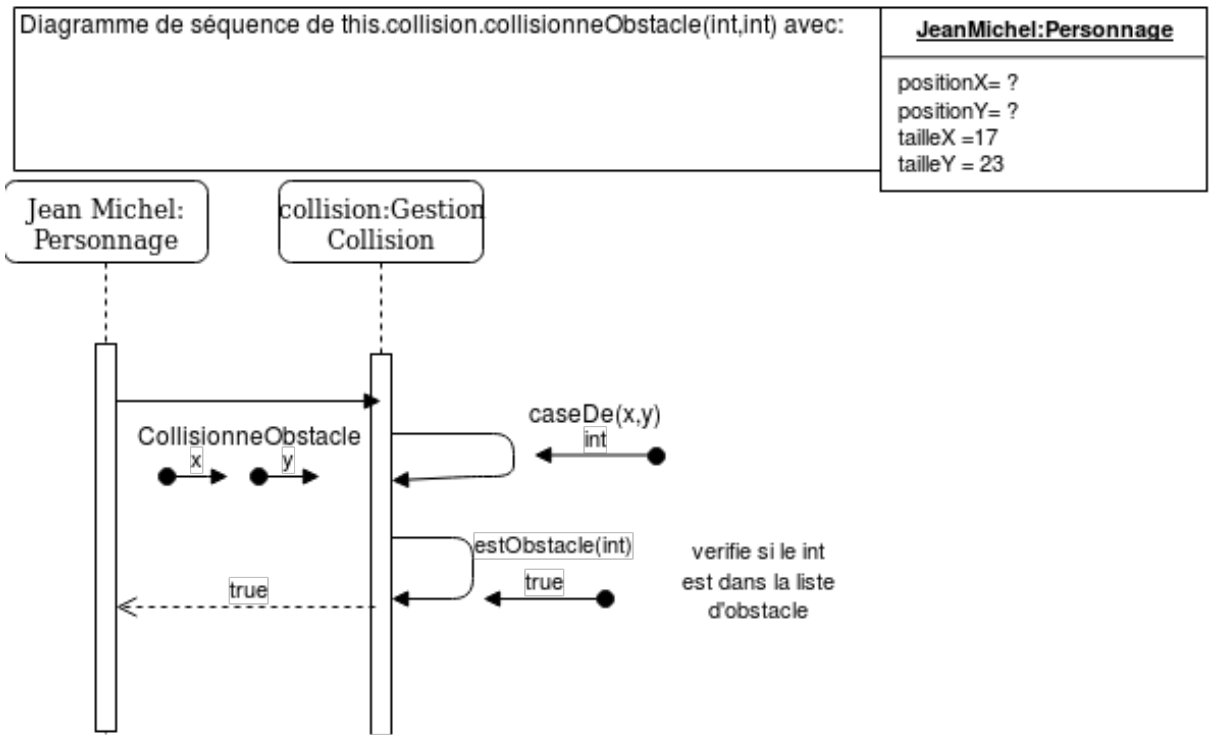


Diagramme détaillant la superclasse Personnage: (page suivante)
Nous avons choisi de présenter ce cas car il nous paraissait plus pertinent. En effet, nous avons la classe *Personnage* qui contient tous les getters et setters ainsi que certaines autres méthodes communes. *JeanMichel* étant spécifique, il est directement extends au *Personnage* alors que pour les classes *Cactus* et *Tentacule* nous avons choisi de les extends d'une classe abstraite *Ennemi* car leur déplacement est différent mais elles ont certaines méthodes communes.


<<Java Class>>
Personnage
app.modele
nom: String
positionX: SimpleIntegerProperty
positionY: SimpleIntegerProperty
pointsVie: SimpleIntegerProperty
tailleX: int
tailleY: int
orientation: int
collision: GestionCollision
Personnage(String,int,int,int,int,int)
pointsVieProperty():SimpleIntegerProperty
getPointsVie():int
getX():int
getY():int
XProperty():SimpleIntegerProperty
setX(SimpleIntegerProperty):void
YProperty():SimpleIntegerProperty
setY(SimpleIntegerProperty):void
getTailleX():int
getTailleY():int
getOrientation():int
setOrientation(int):void
attaquer(Personnage):void
estAttaque(int):void
caseX():int
caseY():int
<<Java Class>>
JeanMichel
app.modele
jeu: Jeu
JeanMichel(int,int)
action(KeyEvent):void
haut():void
bas():void
gauche():void
droite():void
attaquer(Personnage):void
getJeu():Jeu
setJeu(Jeu):void
ennemiACote():Ennemi
recupererItem():boolean
collisionItem(int,int,int,int,int,int):boolean
<<Java Class>>
Ennemi
app.modele
jeu: Jeu
Ennemi(String,int,int,int,int,int)
seDeplacer():void
setJeu(Jeu):void
<<Java Class>>
Cactus
app.modele
orientation: int
Cactus(String,int,int,int)
haut():void
bas():void
gauche():void
droite():void
seDeplacer():void
<<Java Class>>
Tentacule
app.modele
Tentacule(String,int,int,int)
seDeplacer():void
haut():void
bas():void
gauche():void
droite():void

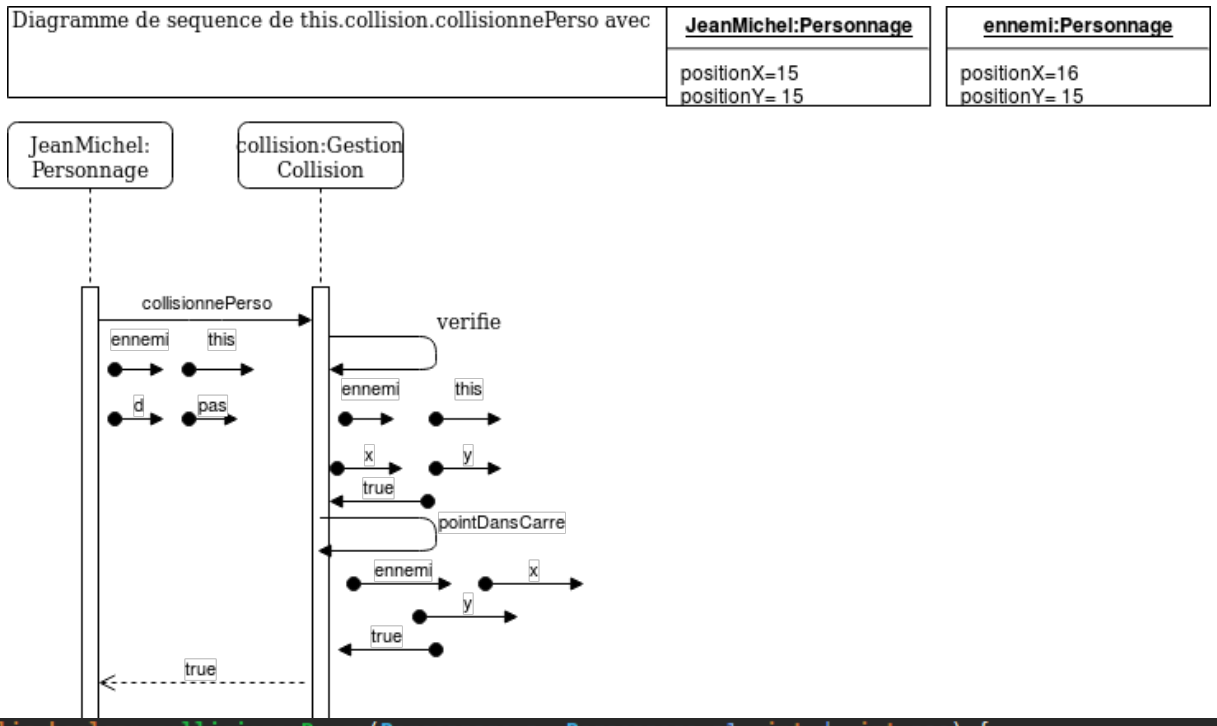
Diagrammes de séquence (5/60 points)

Vous choisirez des méthodes intéressantes du point de vue de la répartition des responsabilités entre les différentes entités du programme. Vous utiliserez des diagrammes de séquence pour expliquer l'exécution de ces méthodes.




```java
public boolean collisionneObstacle(int x, int y) {
    if (x < 0 || y < 0) return true;
    return estObstacle(caseDe(x, y));
}
private int caseDe(int x, int y) {
    try {
        return this.terrain.getTab2dObs()[y/16][x/16];
    } catch(Exception e) {
        return this.obstacles.get(0);
    }
}
private boolean estObstacle(int i) {
    for(int o : obstacles)
        if(i == o)
            return true;
    return false;
}
```

Le if sert à s'occuper des valeurs interdites sinon *caseDe* est appelée. Cette fonction retourne la valeur correspondante du tableau du terrain. Puis *estObstacle* parcourt la liste d'obstacle avec cette valeur pour savoir si elle est présente ou non dans la liste.




```java
public boolean collisionnePerso(Personnage p, Personnage p1, int d, int pas) {
    switch(d) {
    case 0: return verifie(p, p1, 0, -pas);
    case 1: return verifie(p, p1, -pas, 0);
    case 2: return verifie(p, p1, 0, pas);
    case 3: return verifie(p, p1, pas, 0);
    }
    return false;
}
private boolean verifie(Personnage p, Personnage p1, int x, int y) {
    if(p == null || p1 == null) return false;
    if(pointdansCarre(p, p1.getX() + x, p1.getY() + y)
    || pointdansCarre(p, p1.getX() + x + p.getTailleX(), p1.getY() + y)
    || pointdansCarre(p, p1.getX() + x, p1.getY() + y + p.getTailleY())
    || pointdansCarre(p, p1.getX() + x + p.getTailleX(), p1.getY() + y + p.getTailleY()))
        return true;
    return false;
}
private boolean pointdansCarre(Personnage p, int x, int y) {
    if(x >= p.getX() && x <= p.getX() + p.getTailleX() && y >= p.getY() && y <= p.getY() + p.getTailleY()
    || x >= p.getX() && x <= p.getX() + p.getTailleX() && y >= p.getY() && y <= p.getY() + p.getTailleY())
        return true;
    return false;
}
```

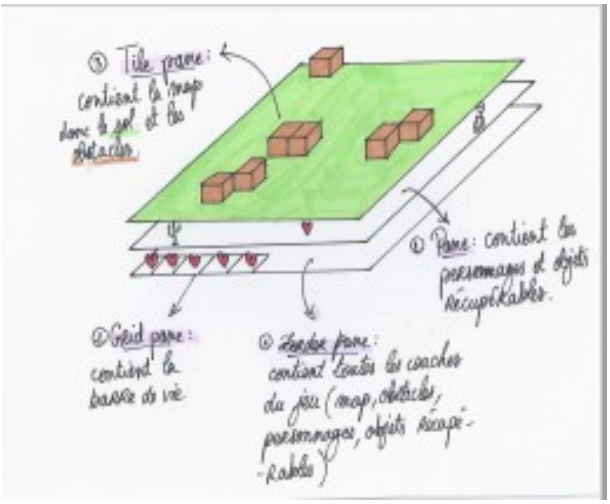
collisionnePerso appelle *verifie* avec des paramètres différents selon la direction du personnage. On vérifie d'abord les valeurs interdites puis *pointdansCarre* est appelée pour chaque sommet de l'image. Elle vérifie si les positions des sommets des images ne s'entremêlent pas.

2 :Documents pour IHM

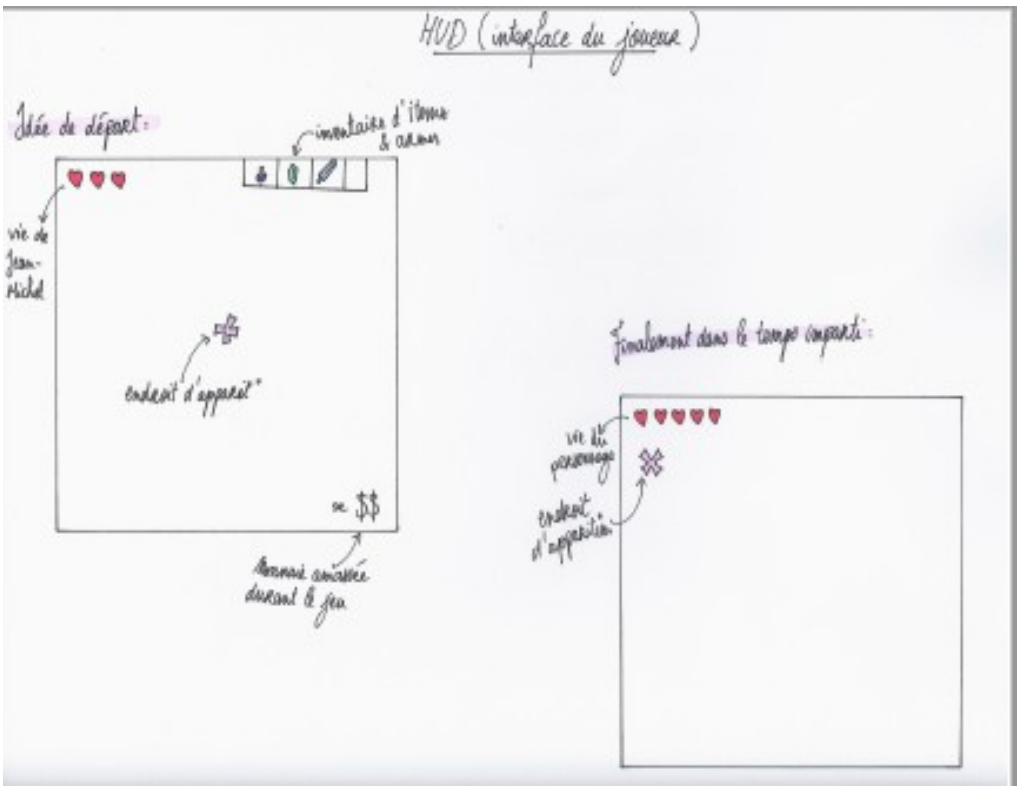
Maquettes (5 points/30)

Vous devez, avant de programmer les interfaces, les concevoir au moyen d'un outil (mockup, ou scenebuilder). Les maquettes doivent être expliquées et commentées.

Voici une maquette montrant les différentes couches de notre jeu :



Et voici la maquette de note interface utilisateur, ou HUD :



3:Documents pour gestion de projet

Cahier d'initialisation (2 points/20)

Description de votre jeu

- *la quête principale:* Le héros, Jean-Michel, doit battre le méchant capitaliste afin de sauver la faune et la flore de sa planète, parce qu'on en a qu'une, quand même.
- *les mécanismes:* ZQSD pour se déplacer, E pour interagir, autres interactions à définir sur la partie droite du clavier
- *Quelques items (armes, objets …) qui peuvent être découverts et comment ils influent sur le comportement de Jean-Michel:* poings comme arme de base, et 1 arme dans chaque zone qui permet de mieux progresser (p.e arme moins efficace)
- *Quelques ennemis avec leur mode de déplacement et d'attaque:* tentacule ou sushi comme ennemis de base
- Plusieurs zones différentes, comportant chacune 1 miniboss, qui, une fois battu, donne un objet important.

Cahier de spécifications fonctionnelles (8 points/20)

cette partie devra être mise à jour tout au long du projet pour décrire les spécifications fonctionnelles.

Map:

La map du jeu est d'abord créée sur le logiciel Tiled, un logiciel permettant de créer des maps faites de tuiles (ou tiles).
Cette map est ensuite exportée dans un fichier XML, dont la suite de nombres est copiée dans un fichier texte, avec un fichier texte différent pour chaque couche.
Un BufferedReader pour ce fichier texte dans la classe *Terrain* est utilisé pour convertir cette suite de nombres en un tableau d'entiers à deux dimensions.
Ce tableau est ensuite utilisé par la classe *VueTerrain* pour :

1. affecter à chaque valeur du tableau une image placée dans le dossier app/img.
2. créer un ImageView qui contiendra l'image précédemment créée.
3. ajouter ces ImageView dans un TilePane.

Ce procédé est le meme pour la couche des obstacles.

Collision:

Pour gérer les collisions de manière générale, nous avons créé une classe *GestionCollision*. Elle a comme attributs un *Terrain* et une ArrayList d'entiers servant à stocker tous les nombres provenant du fichier texte contenant les id des obstacles.
Dans le constructeur de *GestionCollision*, la liste est d'abord remplie de tous les nombres excepté 0, sans prendre compte des doublons.
Ensuite, on utilise un Set d'Integer comme "pivot" : le Set n'acceptant pas les doublons, on le remplit de tous les éléments de la liste.
Enfin, après avoir vidé toute la liste, on la remplit de nouveau avec tous les éléments du Set. Cela permet de conserver de la mémoire.

Chaque personnage a en attribut la classe *GestionCollision*.
Voyons comment sont gérées les collisions :

Avec obstacles:

Si l'id devant le personnage sur la map appartient a la liste d'obstacles alors il ne peut avancer.
Pour vérifier s'il y a collision, on utilise une méthode *collisionneObstacle* prenant en paramètre une position x et y, et renvoyant un booléen.
Elle appelle deux méthodes en meme temps: *estObstacle* et *caseDe*.
La méthode *estObstacle* prend en paramètre un int, le compare avec la liste d'obstacles et renvoie true dans le cas où il se trouve bel et bien dans la liste.
Quant à la méthode *caseDe*, elle prend aussi en paramètre un int x et y, et renvoie un int. Elle renvoie la valeur de la case "[y/16][x/16]" du tableau d'obstacles du *Terrain* en attribut.

La méthode *collisionneObstacle* est appelée à chaque fois que *JeanMichel* ou un *Ennemi* veut se déplacer.

Entre personnages:

Dans la théorie, le principe de vérification est le même, mais en pratique, le procédé est totalement différent.

La méthode appelée est *collisionnePerso*, prenant en paramètre deux *Personnage*, un int représentant la direction, et un autre représentant la distance parcourue à chaque pas.
En fonction de la direction, la méthode vérifie, prenant les deux mêmes *Personnages* et deux int de position en paramètre, détermine si le personnage se trouve dans le "carré" occupé par le deuxième personnage, en appelant la méthode *pointdansCarre*, qui ne prend qu'un *Personnage* et deux int de position en paramètre.

Tout comme la méthode *collisionneObstacle,* la méthode *collisionnePerso* est aussi appelée à chaque fois que *JeanMichel* ou un ennemi veut se déplacer.

Se déplacer:

Jean-Michel:

JeanMichel, contrairement à ses ennemis, se déplace de 4 pixels à chaque pas, au lieu d'1 pixel.
A chaque fois que *JeanMichel* souhaite se déplacer, on vérifie si, dans la direction dans laquelle il se déplace, il y a un ennemi ou un obstacle.
Si oui, il ne se déplace pas. Si non, il peut donc se déplacer.

Cactus:

Le *Cactus* se déplace de manière très basique, ne changeant de direction que lorsqu'il rencontre un obstacle ou *JeanMiche*l.
Dès qu'il rencontre un obstacle, il se réoriente, et ce, jusqu'à sa mort.

Tentacule:

Il utilise le *BFS* pour se déplacer sur la map, en suivant toujours la position de *JeanMiche*l.
La classe *BFS* a comme attributs un *Terrain*, une HashMap de *Tile* représentant toutes les tiles de la map, une queue de *Tile* représentant la file, et une ArrayList de *Tile* pour stocker les *Tile* marquées.

Le *BFS* est appelé grace à la méthode *lancerBFS,* qui elle-même appelle d'autres méthodes: *sommetsAdjacents* qui ajoute toutes les *Tile* adjacentes dans la file, sauf si elles sont déjà marquées.

Attaquer:

Jean-Michel:

Lorsque *JeanMichel* appuie sur la touche E, la méthode *collisionnePerso* vérifie s'il y a des ennemis dans les cases tout autour de lui, et les attaque automatiquement.

Ennemis:

Les C*actus* n'attaquent pas.
Les *Tentacules* attaquent lorsqu'elles sont en contact avec *JeanMichel*: dans la méthode *seDeplace*r, après s'etre déplacé, la *Tentacule* vérifie s'il y a *JeanMichel* dans sa direction et l'attaque si c'est le cas.

Récuperer des objets:

Dès que *JeanMichel* passe sur un objet, il récupère l'objet.
Dans notre cas, *JeanMichel* ne récupère que des coeurs, ce qui a pour effet de restaurer sa vie.
Dans la méthode *initialize*, on ajoute un listener sur la property de la vie, et on appelle une méthode *verifVie* à chaque changement, qui se charge de modifier le nombre de coeurs en haut à gauche en fonction de la valeur de la vie.