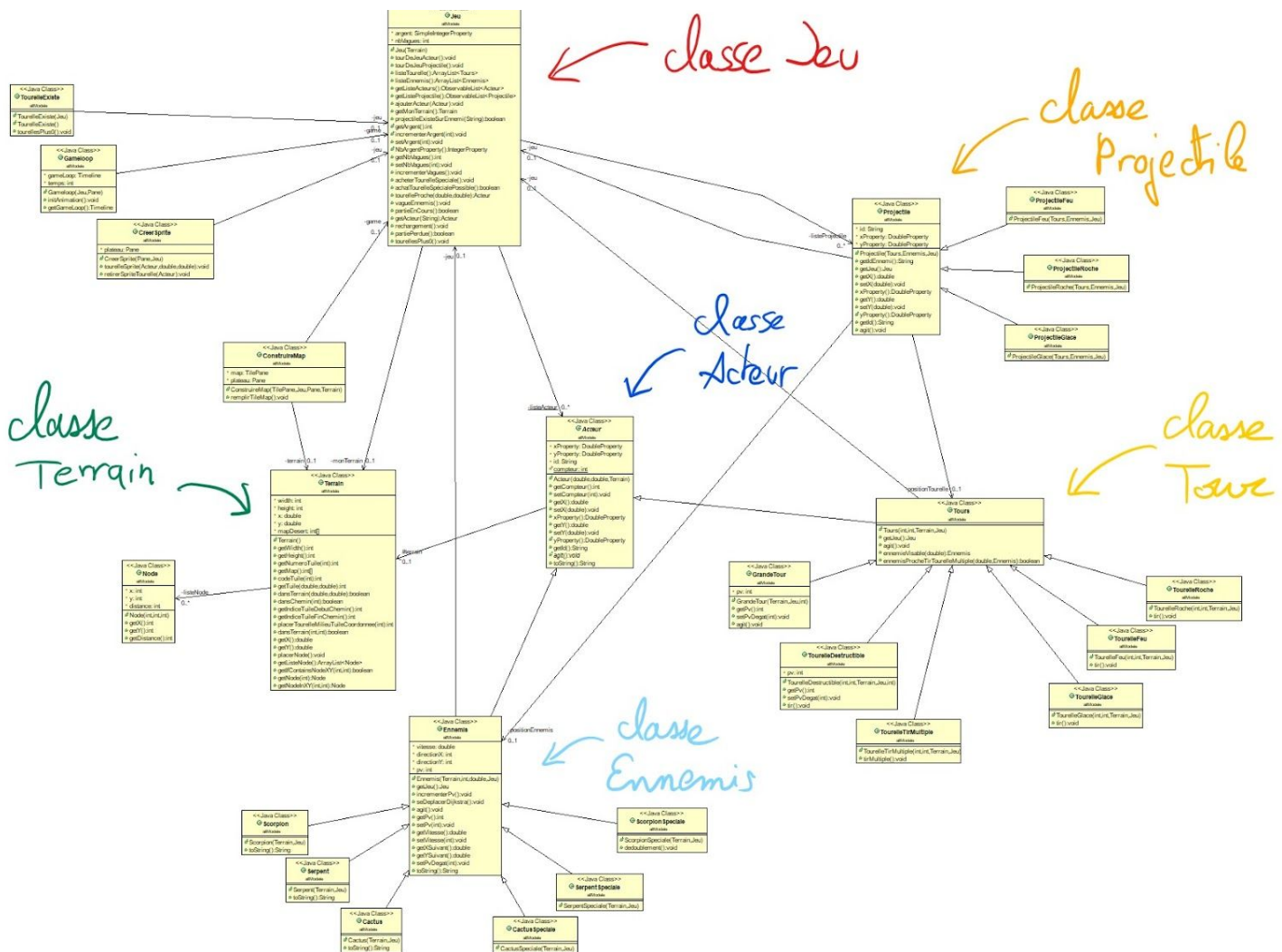
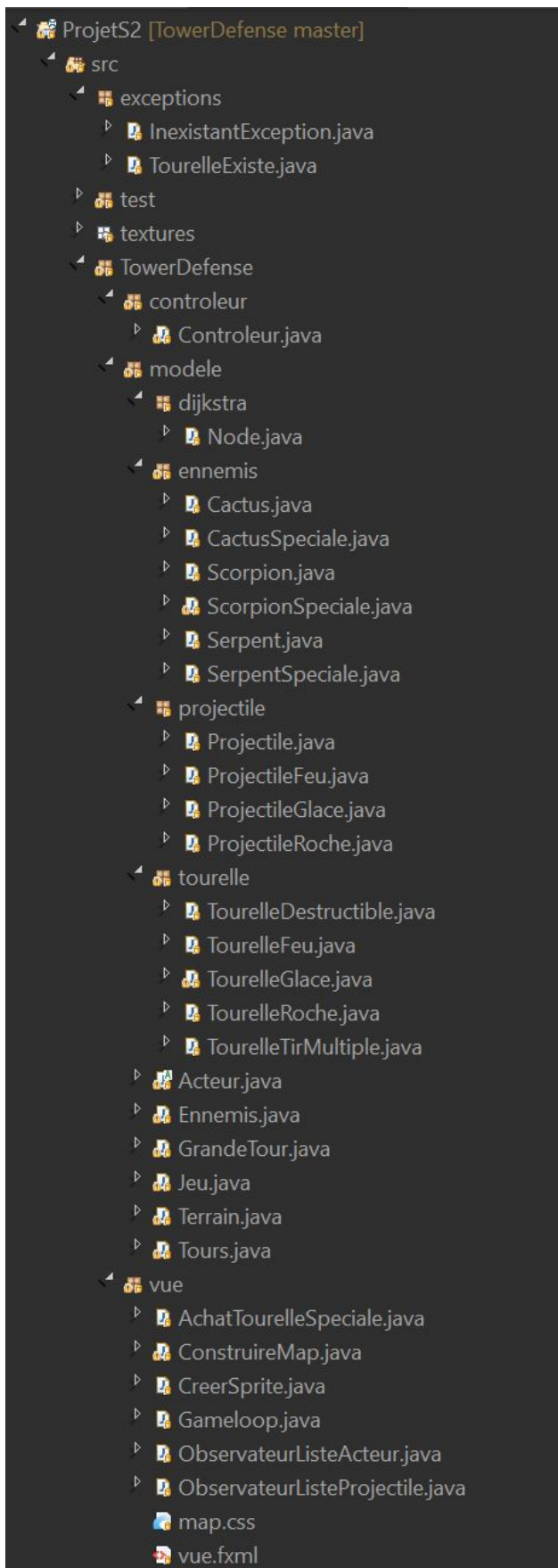


Dossier du Projet (19-20)

1.1 Architecture

Voici à quoi ressemble le diagramme de l'intégralité de notre notre package modèle. Par son agencement, vous pouvez très facilement identifier les différents sous-ensembles.





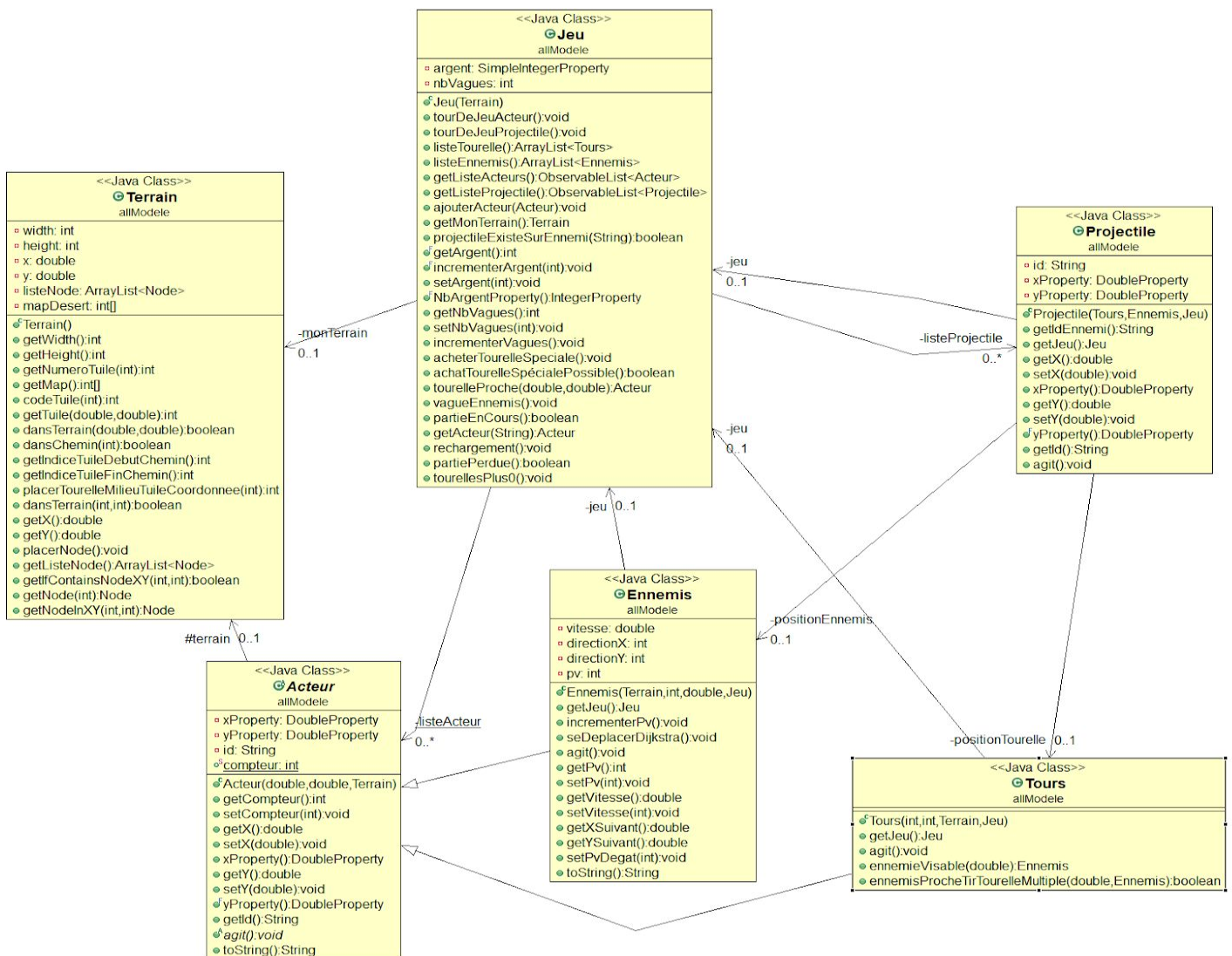
Notre architecture du modèle comptant 26 classes, nous n'allons nous attarder que sur les plus intéressantes.

Mais avant ça, voici comment est agencé notre le projet au sens du package explorer et notamment de l'architecture MVC. Comme vous pouvez le voir nous avons fait dans le package du modèle des sous-packages pour une plus grande lisibilité.

Nous dès à présent discuter de l'architecture des principales classes (classes indiquées en couleur sur le diagramme général juste au-dessus).

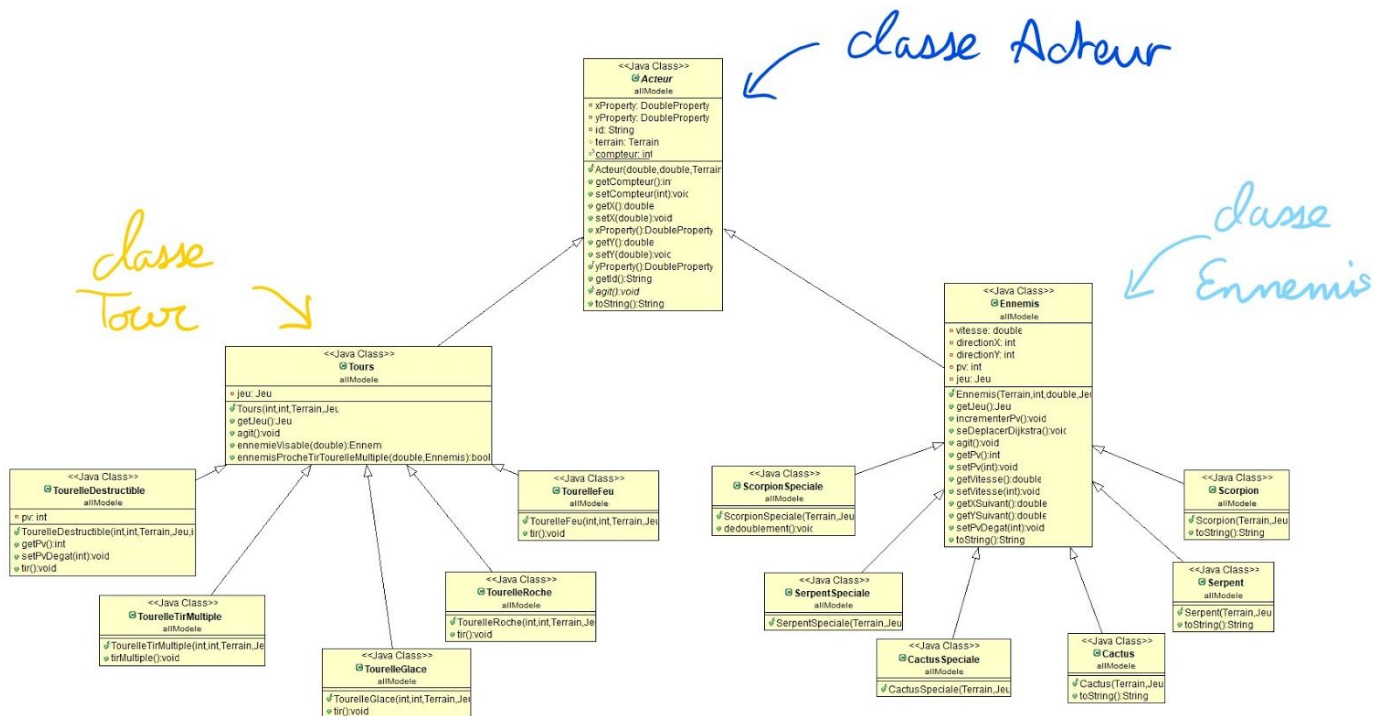
La classe Jeu est la classe “maîtresse” du jeu, en effet c’est elle qui va gérer les tours de jeu des différents acteurs et des projectiles, mais aussi gérer les ressource, le système d’argent du jeu, sans oublier les condition pour perdre la partie. Par conséquent, la classe Jeu possède a un Terrain, des Acteurs et des Projectiles sous forme d’observables listes.

Les Tours et les Ennemis étant des Acteurs héritent du code de la classe Acteur, notamment de la méthode abstraite *agit* qui est codée respectivement dans leur classe (Ennemis et Tours) pour leurs permettre d’agir différemment. Nous avons donc déclaré la méthode *agit* (*public abstract void agit();*) dans la classe Acteur pour que les classes héritières puissent faire un *@Override*. Ces Acteurs prennent en attribut le Jeu pour pouvoir accéder à leur liste respectives.



Maintenant, intéressons-nous à l'architecture des classes liées à la classe Acteur.

Comme dit précédemment, les classes Tours et Ennemis héritent du code de la classe Acteur pour pouvoir agir différemment. Mais ces deux classes sont aussi donatrices de code, la classe Acteur est une *SuperClass* des classes Tours et Ennemis, qui sont elles-mêmes des *SuperClass* respectivement de la classe Cactus ou encore TourelleFeu.



Pourquoi avons-nous choisi des *SuperClass* et non pas des *Interfaces* ?

Ce choix à été fait pour éviter la redondance de code, en effet, prenons les classes héritières de la classe Ennemis du diagramme ci-dessus. Ces classes et notamment les ennemis vont agir de la même façon : se déplacer en suivant un itinéraire donné. Cette méthode de déplacement doit donc être codée dans la classe donatrice pour ne pas avoir à répéter ce code dans les six classes héritières (à cet instant notre TowerDefense ne possède que ces six différents ennemis, mais si l'on entreprenait de faire évoluer le code en en ajoutant de nouveaux, la redondance serait encore plus lourde et le code ne serait pas optimisé ce qui sera un problème). Même principe pour la classes Tours et ses héritières.

Face à cet aspect, notre projet ne comporte aucune *Interface*, car, pour reprendre l'exemple, les différents ennemis ont les même responsabilités : se déplacer, donc nous n'avions pas besoin de coder des méthodes d'agissement différentes à ce niveau là.

Dans la classe Acteur n'est codé aucune méthode, il y a seulement les accesseurs ou *Getter* et *Setters* et la déclaration de la méthode *agit*. Les méthodes essentielles aux ennemis et aux tourelles sont codée respectivement dans les classes Ennemis et Tours.

Hormis, pour les ennemis spéciaux et tourelle spéciales, les classes des différents ennemis et tourelles ne comportent que leur constructeur dont la valeur des attributs varie en fonction de chacun ; et éventuellement un *toString()*, étant donné qu'ils héritent de leur classes donatrices respectives.

1.2 Diagrammes de classe

Classe Jeu



Tout d'abord nous avons la classe Jeu, c'est la classe maître, au dessus de tout le monde, elle est très importante dans la conception de notre jeu. Elle contient tous les éléments nécessaire pour la construction du jeu, c'est-à-dire toutes les listes importantes, le terrain de jeu, l'argent et les vagues d'ennemis. Ces éléments sont les attributs de cette classe.

Concernant les listes, ce sont des *ObservableList*, il y en a deux. Une *ObservableList* pour les acteurs et une deuxième pour les projectiles. Ces deux listes sont utilisés dans deux classes de la Vue, ces classes sont *ObservateurListeActeur* et *ObservateurListeProjectile*. Utilisé des *ObservableList* nous a permis la création des *Sprites* des ennemis et des projectiles de manière automatique dès lors que nous ajoutons un ennemi à la liste d'acteur et un projectile à la liste de projectile.

Nous avons également mis en place un système d'achat de tourelle spéciales, pour mettre en place ces achats il nous fallait donc un système d'argent, le seul moyen de gagner cette argent est de tuer les ennemis, en mettant leurs points de vie à zéro.

Dans la même idée que les *ObservableList*, l'argent était un *SimpleIntegerProperty*, on a utilisé un *bind* pour lié le label "nbArgent" de la

Vue à l'attribut "argent" de la classe Jeu et donc mettre à jour la valeur. Nous avons créée une classe à part dans la Vue qui informait le joueur qu'il avait assez d'argent pour acheter une tourelle spéciale, cette classe implémente un *ChangeListener* et ainsi synchroniser les données. Le joueur devait voir la quantité d'argent qu'il possédait, c'est pour cela que nous avons utilisé un *ChangeListener*, ainsi le joueur était tenu au courant de la quantité d'argent qu'il possédait tout au long de la partie puisque l'attribut "argent" était écouté.

Pour ce qui est des méthodes de la classe Jeu, au delà des *Getters*, des *Setters* et des méthodes qui retournait des listes de tourelles et d'ennemis. La classe contenait toutes les méthodes utilisées dans la *GameLoop*. Elle sont au nombre de quatre, deux d'entre elles concernaient les

tours de jeu, elles faisaient agir les acteurs et les projectiles. Ces deux méthodes sont les deux méthodes les plus importantes du jeu, sans elles il ne se passe rien : les ennemis ne se déplacent pas et les tourelles ne tirent pas. Les deux autres méthodes présentes dans la *Gameloop* gèrent les vagues d'ennemis et une méthode spéciale d'un des ennemis.

Nous n'avions dans un premier temps créé qu'une seule méthode qui contenait tout, une méthode qui faisait agir à la fois les acteurs et les projectiles. Mais nous avons trouvé qu'il était plus pertinent de découper ces méthodes en plusieurs. De la sorte, nous pouvions dans la *Gameloop* utiliser des temporalités différentes ; l'appel de la méthode `tourDeJeuProjectile` était ainsi appelé beaucoup plus souvent que la méthode `tourDeJeuActeur`, cela rendait le jeu et surtout l'aspect des tirs beaucoup plus dynamique.

La méthode `tourelleProche` est utilisée lorsque le joueur souhaite déposer une tourelle, elle vérifie si il n'y en a pas une à proximité et donc empêche le joueur d'en poser une trop près ou par dessus une tourelle. Si de prime abord elle semble devoir être plutôt placée dans la classe `Tourelle`, il nous a paru plus sensé de la mettre dans la classe `Jeu` car nous nous plaçons non pas du point de vue d'une tourelle mais du point de vue du joueur lui-même.

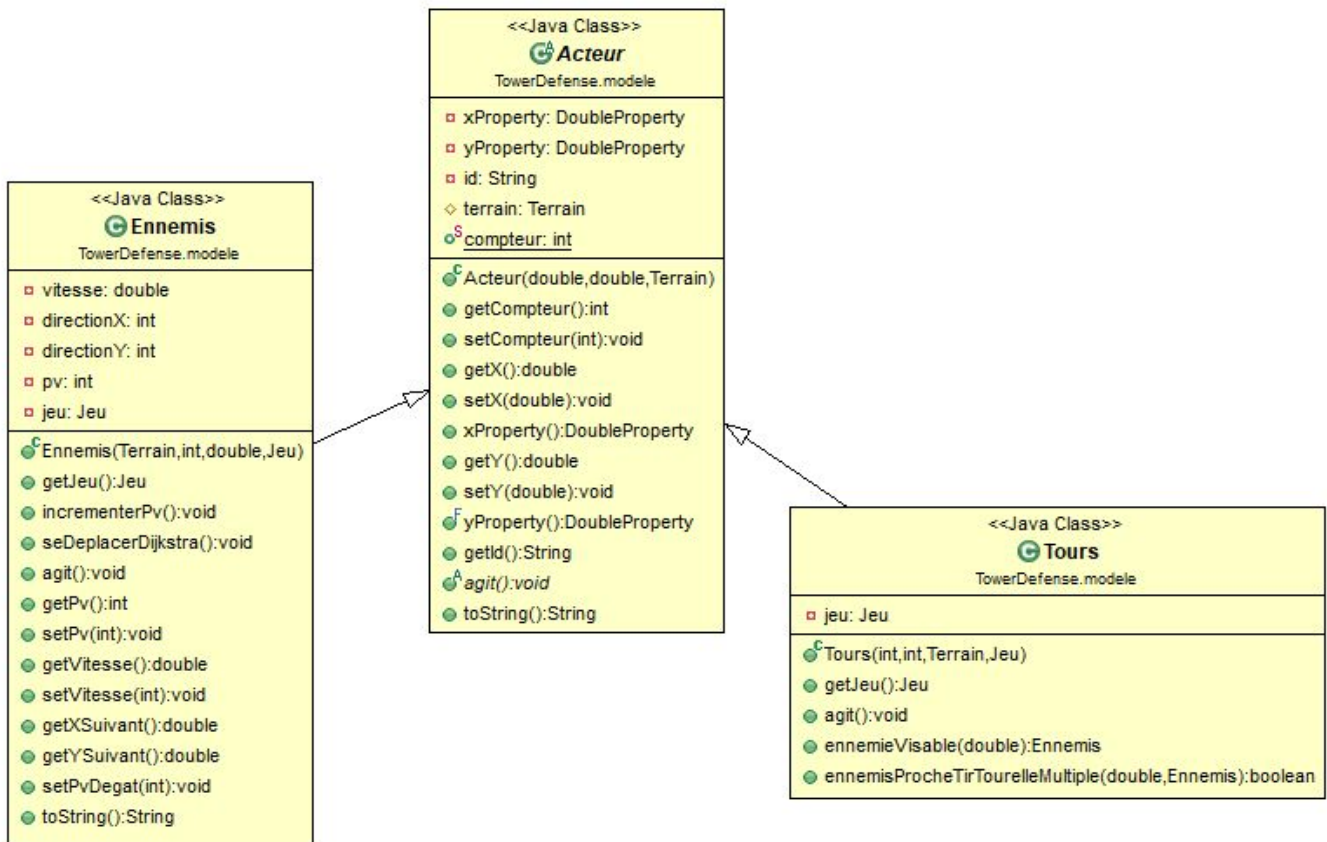
Beaucoup des méthodes des autres classes du modèle avaient besoin d'avoir accès au contenu de la classe `Jeu`, c'est pour cela que pratiquement toutes les classes possèdent un attribut `Jeu`.

Classe Projectile

<<Java Class>>	
Projectile	
TowerDefense.modele.projectile	
▣	positionTourelle: Tours
▣	positionEnnemis: Ennemis
▣	id: String
▣	xProperty: DoubleProperty
▣	yProperty: DoubleProperty
▣	jeu: Jeu
🔗	Projectile(Tours,Ennemis,Jeu)
🔍	getIdEnnemi():String
🔍	getJeu():Jeu
🔍	getX():double
🔍	setX(double):void
🔍	xProperty():DoubleProperty
🔍	getY():double
🔍	setY(double):void
🔍	yProperty():DoubleProperty
🔍	getId():String
🔍	agit():void

La classe `Projectile` n'hérite pas de la classe `Acteur`, on a décidé de la séparer pour ne pas que les projectiles fassent partie de la liste d'acteur est donc qu'ils agissent en même temps que les ennemis dans la *Gameloop*. Elle possède comme attribut une tourelle, un ennemi et des coordonnées (x,y). Ces coordonnées sont des *DoubleProperty*. Nous avons utilisé un *bind* dans la classe `ObservateurListProjectile` dans la *Vue* pour mettre à jour leur valeur aussi. L'idée est que les coordonnées (x,y) d'un projectile soient au départ les coordonnées de la tourelle. Un projectile est en réalité une tête chercheuse, dans la méthode *agit*, elle calcule, à chaque fois qu'elle est appelée, sa nouvelle position en fonction de celle de l'ennemi visé, et dans le cas où les coordonnées du projectile et de l'ennemi sont les mêmes, alors il lui inflige des dégâts, par exemple. Puis, ce dit projectile, est retiré de la liste `projectile` et grâce à l'*ObservableList* de `projectile` ; le *Sprite* disparaît aussitôt l'ennemi touché.

Classe Acteur-Ennemis-Tours



La classe Acteur est la classe la plus haute, elle contient les attributs dont toutes les classes qui héritent d'elle ont besoin, c'est-à-dire des coordonnées, un ID et un terrain. Etant donné que les ennemis, mais également, les tourelles agissent différemment, nous avons choisis de mettre la classe Acteur et la méthode *agit* abstraite, pour que les classes Ennemis et Tours puissent posséder cette méthode en *@Override*. Ainsi il n'y a qu'un seul appel de méthode à faire dans la méthode `tourDeJeuActeur` de la classe Jeu et non pas un par classe.

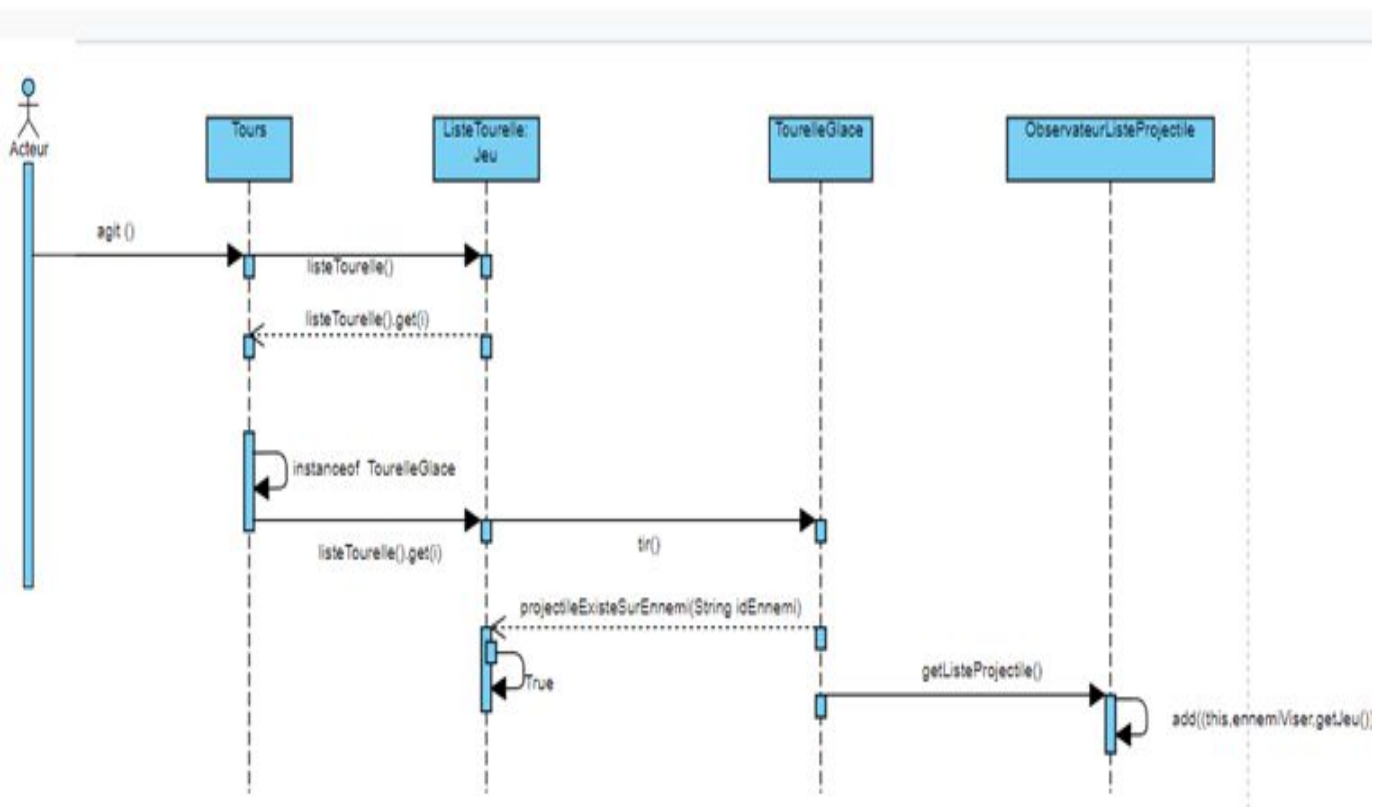
Nous avons six types d'ennemis découpés en six classes qui héritent de la classe Ennemis. Ces six ennemis se déplacent en direction de La Grande Tour ; qui leur inflige des dégâts une fois qu'il se trouvent dans sa zone d'attaque. Mais trois de ces ennemis ont des attaques spéciales : une présente dans la classe Jeu, une gérée dans la méthode *agit* de la classe Projectile et la dernière se trouve dans la classe de l'ennemi en question. Etant donné que tous les ennemis se déplacent, la méthode qui gère leur déplacements se trouve dans la classe Ennemis, celle-ci est appelée, ainsi que l'attaque de l'ennemi spéciale, dans la méthode *agit*.

Les tourelles fonctionnent de la même manière, il y a plusieurs types de tourelle, chacune tirant leurs projectiles attirés et la méthode *agit* de Tours qui appelle chaque méthode de tir de chaque tourelle.

1.3 Diagrammes de séquence

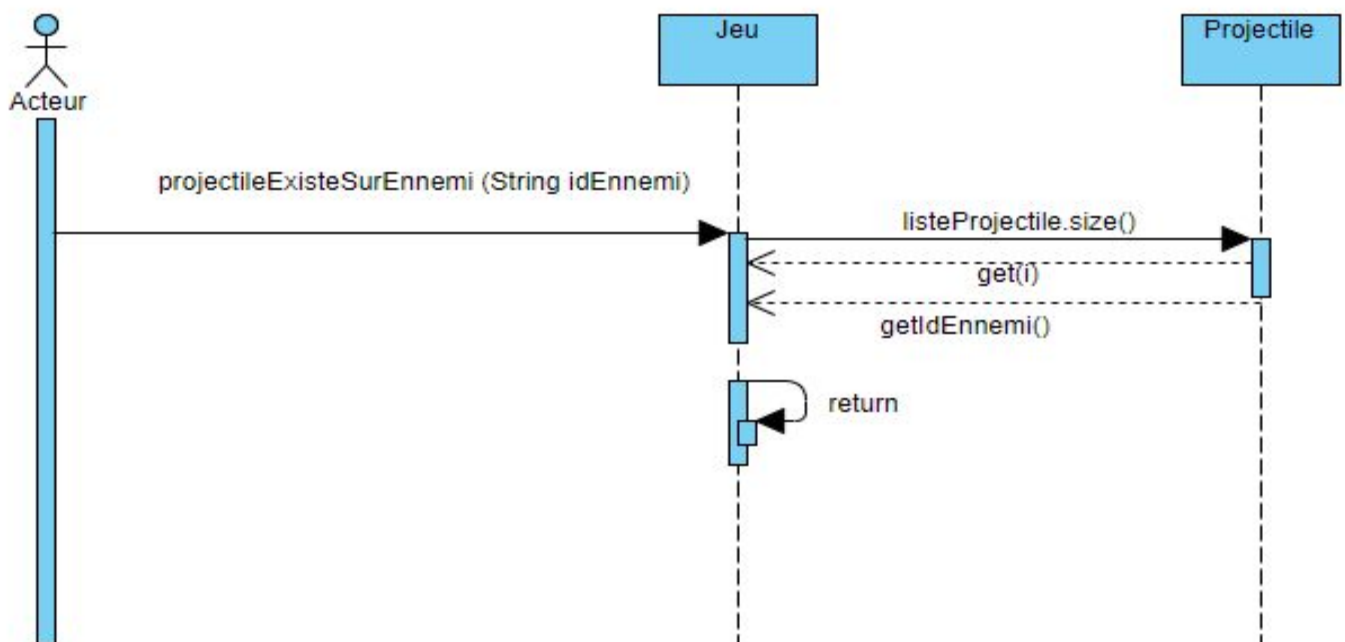
1ère méthode :

```
@Override
public void agit() {
    for(int i=0; i< jeu.listeTourelle().size();i++) {
        if(jeu.listeTourelle().get(i) instanceof TourelleFeu) {
            ((TourelleFeu) jeu.listeTourelle().get(i)).tir();
        }
        else if (jeu.listeTourelle().get(i) instanceof TourelleGlace) {
            ((TourelleGlace) jeu.listeTourelle().get(i)).tir();
        }
        else if(jeu.listeTourelle().get(i) instanceof TourelleRoche) {
            ((TourelleRoche) jeu.listeTourelle().get(i)).tir();
        }
        else if(jeu.listeTourelle().get(i) instanceof TourelleTirMultiple){
            ((TourelleTirMultiple) jeu.listeTourelle().get(i)).tirMultiple();
        }
    }
}
```



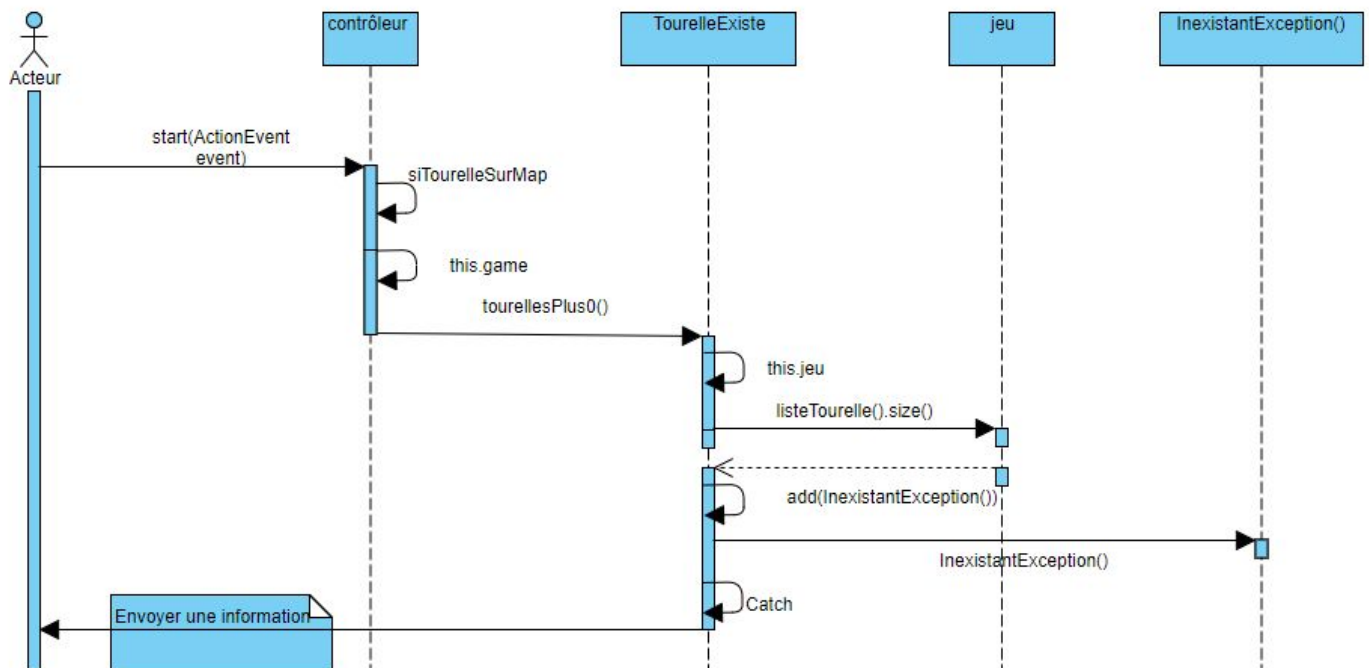
2eme méthode:

```
public boolean projectileExisteSurEnnemi(String idEnnemi) {  
    for(int i=0; i< listeProjectile.size(); i++) {  
        if (listeProjectile.get(i).getIdEnnemi()==idEnnemi) {  
            return true;  
        }  
    }  
    return false;  
}
```



3eme methode:

```
@FXML
void start(ActionEvent event) {
    boolean siTourelleSurMap = true;
    try {
        this.game.tourellesPlus0();
    } catch (InexistantException ie) {
        siTourelleSurMap = false;
    }
    if (siTourelleSurMap) {
        vue.getGameLoop().play();
    }
}
```



1.4 Structures de données :

Dans la classe Jeu, nous avons utilisé des *ObservableList* qui contiennent tous les acteurs, c'est-à-dire, les ennemis et les tourelles tout au long du jeu.

```
public class Jeu {  
  
    private Terrain monTerrain;  
    private static ObservableList<Acteur> listeActeur;  
    private ObservableList<Projectile> listeProjectile;  
    private SimpleIntegerProperty argent;  
    private int nbVagues;  
}
```

1.5 Exception :

La classe Jeu contient une gestion d'exception, elle envoie une exception si le joueur cherche à démarrer le jeu sans avoir posé au minimum une tourelle sur le terrain.

Quand on lance le main, il suffit d'essayer de lancer le jeu en appuyant sur le bouton "play" sans avoir posé de tourelle au préalable pour que le message géré par l'exception s'affiche.

```
public class TourelleExiste {  
    private Jeu jeu ;  
  
    public TourelleExiste(Jeu jeu) {  
        this.jeu = jeu;  
    }  
    public TourelleExiste() {  
  
    }  
  
    public void tourellesPlus0() throws InexistantException {  
        try {  
            if(this.jeu.listeTourelle().size()==1)  
                throw new InexistantException();  
  
        } catch ( InexistantException i) {  
            Alert alert= new Alert(AlertType.INFORMATION);  
            alert.setTitle("Tourelle");  
            alert.setHeaderText(null);  
            alert.setContentText("veuillez mettre au moins une Tourelle !");  
  
            alert.showAndWait();  
  
            throw i;  
        }  
    }  
}
```

1.6 Utilisation maîtrisée d'algorithmes intéressants :

Il y a principalement l'algorithme du BFS, contenu dans la classe Ennemis, appelé dans sa méthode *agit*. Mais avant que les ennemis puissent se déplacer sur le chemin, la méthode *placerNode*, dans la classe Terrain, associe chaque tuile du chemin sur lequel les ennemis peut se déplacer à une Node. Les Node ont leur propre classe, elles est située dans le package dijkstra.

```
public class Node {
    private int x;
    private int y;
    private int distance;

    public Node(int x, int y, int distance) {
        this.distance = distance;
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getDistance() {
        return distance;
    }
}
```



```

public void placerNode() {
    //recuperation coordonnees de la tuile finChemin
    int xFin = (this.getIndiceTuileFinChemin()%30);
    int yFin = (this.getIndiceTuileFinChemin()/30);

    Node noeudFin = new Node(xFin, yFin, 0);
    this.listeNode.add(noeudFin);

    int coordonneesNodeX = 0;
    int coordonneesNodeY = 0;

    //pour chaque i dans la liste de node on recupere les coordonnees X et Y
    for(int indiceNode = 0; indiceNode < this.listeNode.size(); indiceNode++) {
        coordonneesNodeX = this.listeNode.get(indiceNode).getX();
        coordonneesNodeY = this.listeNode.get(indiceNode).getY();
        //gauche
        if(this.dansChemin(coordonneesNodeY*30+coordonneesNodeX-1)) {
            if(this.getIfContainsNodeXY(coordonneesNodeX-1, coordonneesNodeY) == false) {
                Node noeud = new Node(coordonneesNodeX-1, coordonneesNodeY, this.listeNode.get(indiceNode).getDistance()+1);
                this.listeNode.add(noeud);
            }
        }
        //droit
        if(this.dansChemin(coordonneesNodeY*30+coordonneesNodeX+1)) {
            if(this.getIfContainsNodeXY(coordonneesNodeX+1, coordonneesNodeY) == false) {
                Node noeud = new Node(coordonneesNodeX+1, coordonneesNodeY, this.listeNode.get(indiceNode).getDistance()+1);
                this.listeNode.add(noeud);
            }
        }
        //bas
        if(this.dansChemin(coordonneesNodeY*30+coordonneesNodeX-30)) {
            if(this.getIfContainsNodeXY(coordonneesNodeX, coordonneesNodeY-1) == false) {
                Node noeud = new Node(coordonneesNodeX, coordonneesNodeY-1, this.listeNode.get(indiceNode).getDistance()+1);
                this.listeNode.add(noeud);
            }
        }
        //haut
        if(this.dansChemin(coordonneesNodeY*30+coordonneesNodeX+30)) {
            if(this.getIfContainsNodeXY(coordonneesNodeX, coordonneesNodeY+1) == false) {
                Node noeud = new Node(coordonneesNodeX, coordonneesNodeY+1, this.listeNode.get(indiceNode).getDistance()+1);
                this.listeNode.add(noeud);
            }
        }
    }
}
}

```

```

public void seDeplacerDijkstra() {

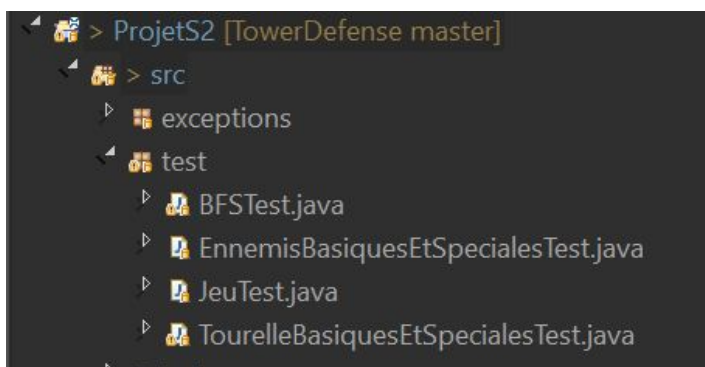
    int posX = (int) this.getX();
    int posY = (int) this.getY();
    int currentIndicePosEnnemi = (posY*30 + posX);
    int currentDistanceNode = this.terrain.getNodeInXY(posX,posY).getDistance();

    //a droite
    if(this.terrain.dansChemin(currentIndicePosEnnemi+1) && this.terrain.getIfContainsNodeXY(posX+1, posY)
        && this.terrain.getNodeInXY(posX+1,posY).getDistance() < currentDistanceNode) {
        this.directionX = 1;
        this.directionY = 0;
        this.setX(this.getX() + this.vitesse*this.directionX);
        this.setY(this.getY() + this.vitesse*this.directionY);
    }
    //a gauche
    else if(this.terrain.dansChemin(currentIndicePosEnnemi-1) && this.terrain.getIfContainsNodeXY(posX-1, posY)
        && this.terrain.getNodeInXY(posX-1, posY).getDistance() < currentDistanceNode) {
        this.directionX = -1;
        this.directionY = 0;
        this.setX(this.getX() + this.vitesse*this.directionX);
        this.setY(this.getY() + this.vitesse*this.directionY);
    }
    //en bas
    else if(this.terrain.dansChemin(currentIndicePosEnnemi+30) && this.terrain.getIfContainsNodeXY(posX, posY+1)
        && this.terrain.getNodeInXY(posX, posY+1).getDistance() < currentDistanceNode) {
        this.directionX = 0;
        this.directionY = 1;
        this.setX(this.getX() + this.vitesse*this.directionX);
        this.setY(this.getY() + this.vitesse*this.directionY);
    }
    //en haut
    else if(this.terrain.dansChemin(currentIndicePosEnnemi-30) && this.terrain.getIfContainsNodeXY(posX, posY-1)
        && this.terrain.getNodeInXY(posX, posY-1).getDistance() < currentDistanceNode) {
        this.directionX = 0;
        this.directionY = -1;
        this.setX(this.getX() + this.vitesse*this.directionX);
        this.setY(this.getY() + this.vitesse*this.directionY);
    }
}
}

```

1.7 Junits :

Les classes couvertes par nos JUnits sont les classes Acteur, Jeu, Tours, Ennemis avec le BFS et une méthode de la classe cactusSpeciale.



2 Documents pour Gestion de projet




2.1 Document utilisateur

- L'objectif du jeu est d'empêcher les ennemis d'approcher de trop près La Grande Tour. Les ennemis, s'ils arrivent à destination, vont infliger des dégâts à la Tour. Son univers est un désert, avec une seule route, qui est celle par laquelle les ennemis arrivent. Les tourelles ont un design plus technologiquement avancé.
- Il y a trois tourelles principales et deux spéciales. Dans les tourelles principales, la tourelle de feu est là pour faire de lourds dégâts, la tourelle de glace pour ralentir les ennemis, et la tourelle de roche pour les immobiliser. Dans les tourelles spéciales, il y a la tourelle destructible qui se détruit automatiquement (ne tire plus) au bout d'un nombre de tirs, et la tourelle multiple qui tire sur plusieurs ennemis à la fois.

Pour ce qui est des ennemis, il y en a trois, et chaque ennemi a une variante plus puissante: les scorpions, les serpents, et les cactus. Les cactus sont des ennemis équilibrés, avec une rapidité et une barre de vie moyenne. Les serpents sont des ennemis plus rapides, mais avec moins de vie, et les scorpions sont plus résistants mais plus lents.

Ci-dessous, un tableau représentant les relations de force et de faiblesse entr les ennemis et les tourelles.

Un **+** signifie que l'ennemi est fort face à cette tourelle, et à l'inverse un **-** signifie que l'ennemi est faible face à elle.

			
	-	+	
		-	+
	+		-

- Lorsque vous éliminez un ennemi, cela va vous rapporter de l'argent. Vous pouvez dépenser de l'argent pour acheter des tourelles spéciales une fois que vous en avez assez.