SAE S3.01

Documentation

Rayan KHEROUA, Arsan ABDI, Horeb SILVA

CONSIGNE:

1 : Le document doit passer en revue (classe par classe et aussi au niveau de la structure générale) les différents défauts que vous avez repérés dans votre code. A priori on ne s'intéresse qu'au Modèle. Cependant, si vous avez eu besoin de retoucher l'architecture MVC, vous pouvez nous en parler.

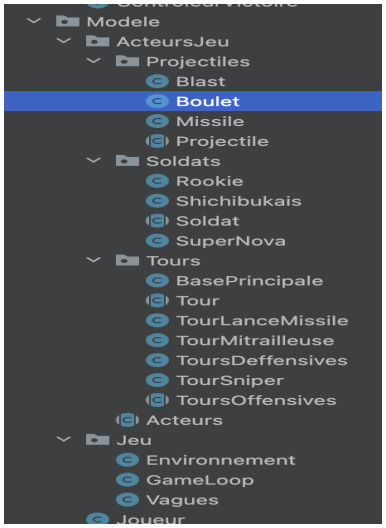
2 : Le document doit expliquer comment vous avez remédié à ces défauts ou comment il faudrait le faire au cas où vous n'avez pas eu le temps de tout traiter.

3 : Le document doit présenter les différents design patterns que vous avez utilisés et en quoi ils apportent quelque chose à votre programme.

PRÉSENTATION ET OBJECTIF:

Ce document a pour but de montrer l'évolution de notre projet JAVA en termes de refactoring.

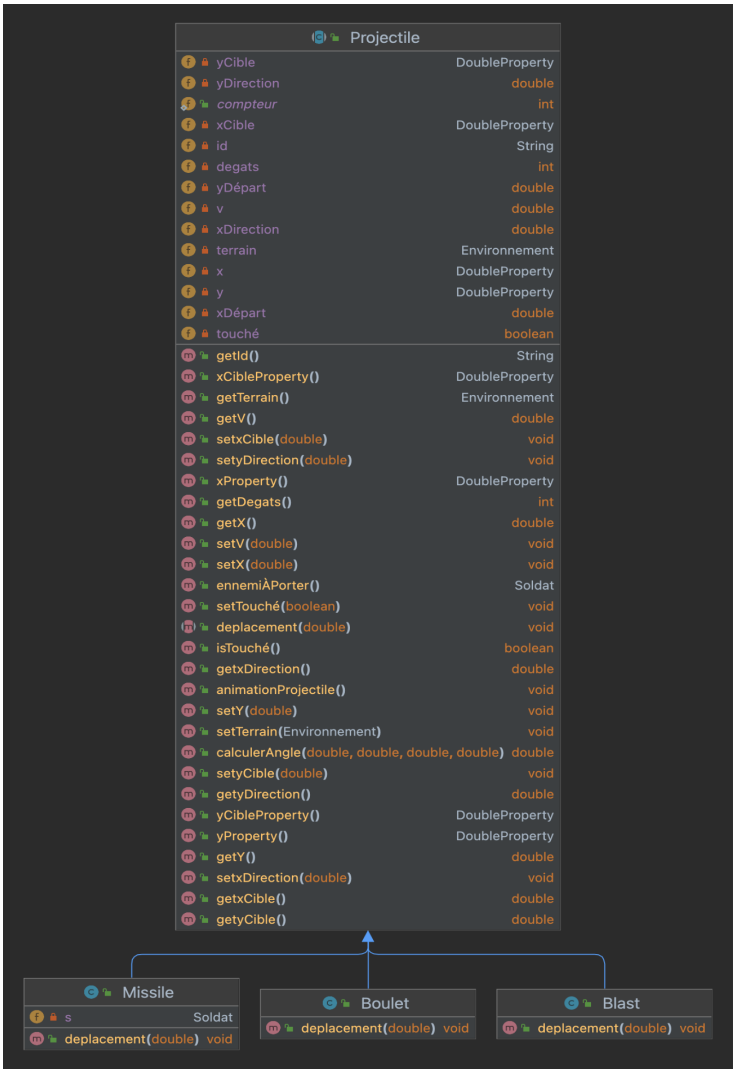
PRÉSENTATION DU JEU:



Tout d'abord, voici le modèle de notre projet, nous allons vous présenter, dans un premier temps, les différents problèmes trouvés, puis un peu plus bas, dans un second temps, nous parlerons des solutions trouvées pour y remédier.

PACKAGE Projectiles : Problèmes

Voici le diagramme de classe de ce package (représenté par la classe projectile et ces sous classes) :



Comme on peut le voir ci dessus, les sous classes de projectiles sont au nombre de trois, avec trois méthodes supposés “différentes”. Mais il y a de gros problème avec ces classes.

Tout d'abord, la classe Blast et la classe Boulet sont identiques :

```java
public class Blast extends Projectile {
    public Blast(double x, double y, double xCible, double yCible, double v, int degats, Environnement terrain) {
        super(x, y, xCible, yCible, v,degats,terrain);
    }

    public void deplacement(double elapsedTime) {
        double deltaX = super.getxDirection() * super.getV()*elapsedTime;
        double deltaY = super.getyDirection() * super.getV()*elapsedTime;

        if (!(super.getX()==super.getxCible()) || (super.getY()==super.getyCible())) {
            setX(getX() + deltaX);
            setY(getY() + deltaY);
        }
    }
}
```

```java
public class Boulet extends Projectile {
    public Boulet(double x, double y, double xCible, double yCible, double v, int degats, Environnement terrain) {
        super(x, y, xCible, yCible, v,degats,terrain);
    }

    public void deplacement(double elapsedTime) {
        double deltaX = super.getxDirection() * super.getV()*elapsedTime;
        double deltaY = super.getyDirection() * super.getV()*elapsedTime;

        if (!(super.getX()==super.getxCible()) || (super.getY()==super.getyCible())) {
            setX(getX() + deltaX);
            setY(getY() + deltaY);
        }
    }
}
```

Même code, même attribut, leur seule différence est leur essence, ce ne sont pas les mêmes classes et c'est tout.

Ensuite on a la classe Missile qui est différente mais qui n'en garde pas moins quelques similitudes :

```java
public class Missile extends Projectile {
    8 usages
    private Soldat s;
    public Missile(double x, double y, double v, int degats, Soldat s, Environnement terrain) {
        super(x, y, s.getX0Value(), s.getY0Value(), v, degats,terrain);
        this.s=s;
    }

    public void deplacement(double elapsedTime) {
        double deltaX = getxDirection() * getV()* elapsedTime;
        double deltaY = getyDirection() * getV()* elapsedTime;

        if (!(getX()==s.getX0Value()) || (getY()==s.getY0Value())) {
            setX(getX() + deltaX);
            setY(getY() + deltaY);
        }
        if(s.estVivant()) {
            double distance = Math.sqrt(Math.pow(s.getX0Value() - getX(), 2) + Math.pow(s.getY0Value() - getY(), 2));
            setxDirection((s.getX0Value() - getX()) / distance);
            setyDirection((s.getY0Value() - getY()) / distance);
        }
    }
}
```

La méthode de déplacement est très similaire et doit être remanier.

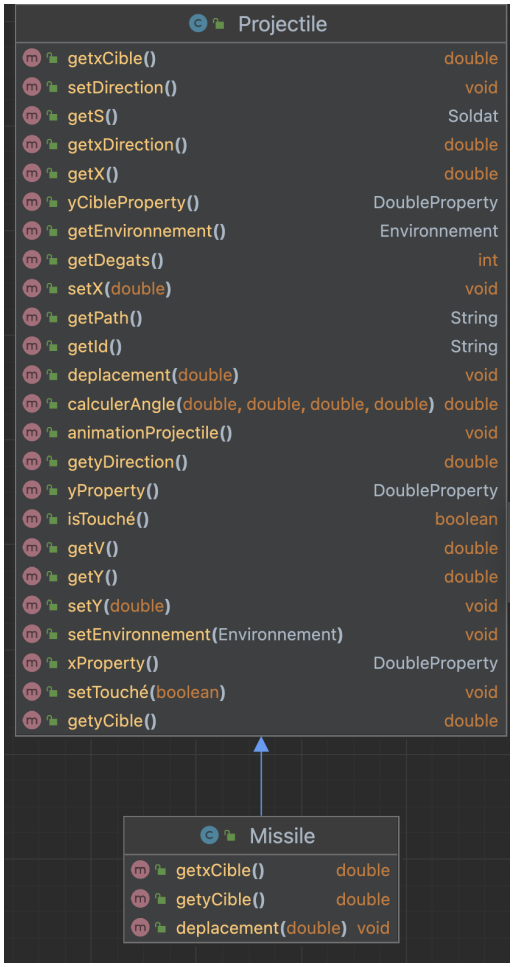
Ensuite, la classe projectile possédait quelques défauts, à corriger, notamment la méthode ennemiÀPorter :

```java
public Soldat ennemiÀPorter() {
    for (Soldat s : terrain.getSoldats()) {
        if (s.estVivant()) {
            double distanceX = Math.abs(s.getX0Value() - getX());
            double distanceY = Math.abs(s.getY0Value() - getY());
            double distanceTotale = distanceX + distanceY;
            if (distanceTotale <= 10) {
                return s;
            }
        }
    }
    return null;
}
```

La méthode de calcul de distance est souvent utilisée dans notre code, un problème auquel nous devons y remédier. En effet, si vous regardez la première version du jeu, vous constaterez qu'il y a plusieurs méthodes de calculs de portée, une méthode ci-dessus ennemiAPorter() dans la classe Projectile et une d'autres méthodes ennemiAPorter() et verificationEstAPorter() dans la classe Tour.

De plus, lorsqu'un projectile est lancé à un certain timing, il se peut que si il touche un ennemi, ce dernier va faire des dégâts sur l'ennemi qu'il visait au départ, et non à l'ennemi qui se trouve sur sa trajectoire.

PACKAGE Projectiles : Solutions



Tout d'abord, nous avons réduit le nombre de classe de projectile à deux : Projectile et Missile. Projectiles réunis désormais le code de la classe Blast et Boulet (en y codant la méthode déplacement directement dessus) , et nous avons effacé toute la redondance de code qu'il y avait dans la classe Missile .

```
public class Missile extends Projectile {

    1 usage   ± arsanabdi
    public Missile(double x, double y, double v, int degats, Soldat s, Environnement terrain, ToursOffensives tour, String path) {

        super(x, y, s, v, degats,terrain, tour, path);
    }
    2 usages   ± Rayankdz +1
    public void deplacement(double elapsedTime) {

        super.deplacement(elapsedTime);

        if(getS().estVivant()) {

            setDirection();
        }
    }
    4 usages   ± Rayankz
    public double getxCible() { return getS().getX0Value(); }
    4 usages   ± Rayankz
    public double getyCible() { return getS().getY0Value(); }
}
```

Aussi, les projectiles possèdent désormais un attribut de type tour, qui leur permet d'utiliser la méthode de calcul de portée de ces dernières.

Ainsi, nous avons supprimé la duplication de code, remanié le code au maximum pour garder le nécessaire et avoir un code plus lisible et réparer quelques problèmes d'action et de déplacement des projectiles. À noter que désormais, un projectile ne peut faire de dégât qu'à un seul ennemi, celui qu'il vise au départ, cela rend moins bien sur le jeu mais c'est la seule solution que nous avons trouver pour régler ce problème.

PACKAGE Tours : Problèmes

Pour la classe tour et ces sous classes, il y a plusieurs problèmes : Tout d'abord, comme dis juste au dessus, on a des méthodes qui recode la calcul d'une distance, ce qui pourrait être remanier, comme ces méthodes de tour :

```
public Soldat ennemiÀPorter() {
        for (Soldat s : terrain.getSoldats()) {
            if (s.estVivant()) {

                double distanceX = Math.abs(s.getX0Value() - getX0Value());
                double distanceY = Math.abs(s.getY0Value() - getY0Value());
                double distanceTotale = distanceX + distanceY;

                if (distanceTotale <= portee) {
                    return s;
                }
            }
        }
        return null;
    }
public boolean vérificationEstÀPorter(double x,double y){
        double distanceX = Math.abs(x - getX0Value());
        double distanceY = Math.abs(y - getY0Value());
        double distanceTotale = distanceX + distanceY;

        if (distanceTotale <= portee) {
            return true;
        }
        return false;
}
```

Ensuite on a les tours offensives qui elles ont énormément de duplication de code, notamment sur la méthode de création de projectile qui est identique à l'exception près qu'elles ne créent pas le même type de projectile : Voici le code dupliqué :

```
public void creationProjectile(Soldat s){
    Missile p = new Missile(getX0Value(), getY0Value(), getVitesseProjectile(), getDegatValue(),s, getTerrain());
    getTerrain().ajouterProjectile(p);
    p.animationProjectile();
}
```

```
public void creationProjectile(Soldat s){
    Boulet p = new Boulet(getX0Value(), getY0Value(), s.getX0Value(),s.getY0Value(), getVitesseProjectile(), getDegatValue(), getTerrain());
    getTerrain().ajouterProjectile(p);
    p.animationProjectile();
}
```

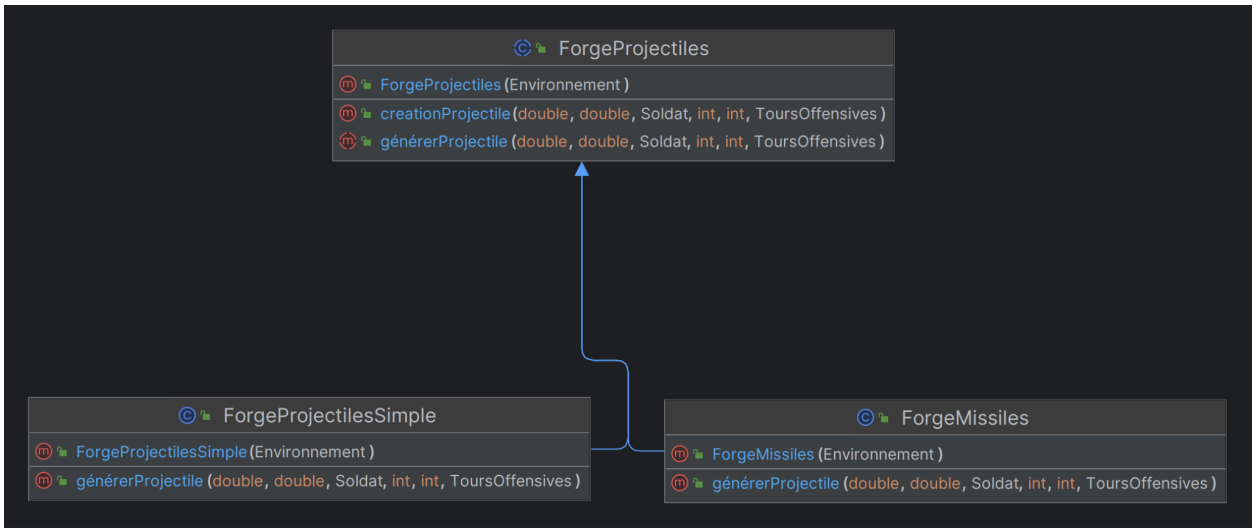
```
public void creationProjectile(Soldat s){
    Blast p = new Blast(getX0Value(), getY0Value(), s.getX0Value(),s.getY0Value(), getVitesseProjectile(), getDegatValue(), getTerrain());
    getTerrain().ajouterProjectile(p);
    p.animationProjectile();
}
```

Enfin, il y a la classe défensive. Il n'y a pas de problèmes concrets avec cette classe, seulement elle manque d'originalité, on pourrait pousser le concept de tour défensives aussi loin que celui de tour offensives (plusieurs types de ralentissement, d'autre mécanisme de défense, etc). Malheureusement ce point ne sera pas traiter sur nos travaux par manque de temps.

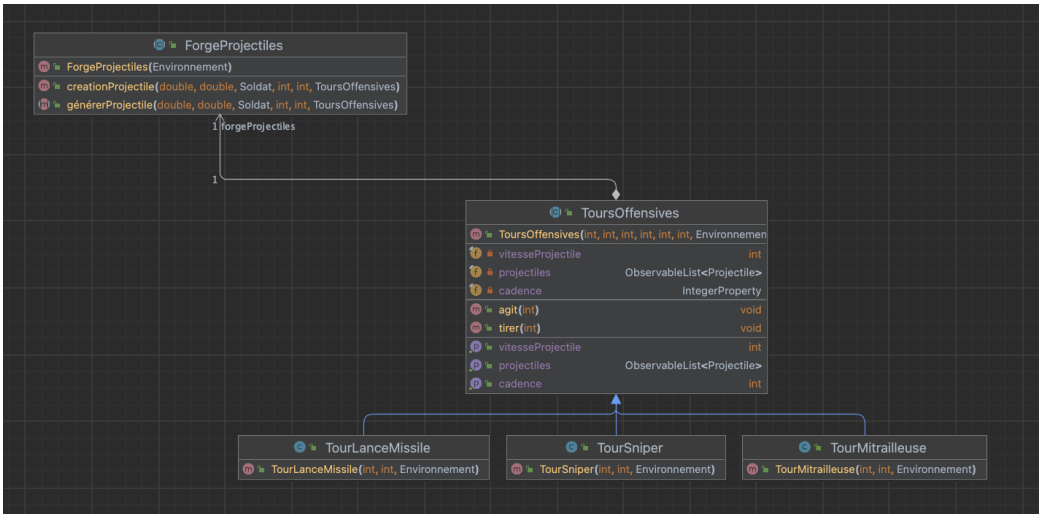
Ensuite, nous avons dû corriger un problème de type mvc, c'était la vue qui s'occupait de créer et ajouter les tours à l'environnement, ce qui est contraire à l'architecture mvc. Nous traiterons ce souci plus en détail à la fin.

PACKAGE Tours : Solutions (=> ForgeProjectiles)

Ce design pattern est utilisé pour régler le problème de redondance des tours :



1. **ForgeProjectiles** : Cette classe est responsable de la création de projectiles divers dans le jeu.
    - **creationProjectile(double, double, Soldat, int, int, ToursOffensives)** : Une méthode pour créer un projectile, en spécifiant les coordonnées de lancement, l'entité (Soldat) cible du projectile, la vitesse les dégâts, et l'objet `ToursOffensives` qui représente la tour qui tire le projectile.
    - **générerProjectile(double, double, Soldat, int, int, ToursOffensives)** : retourne un projectile simple ou un missile



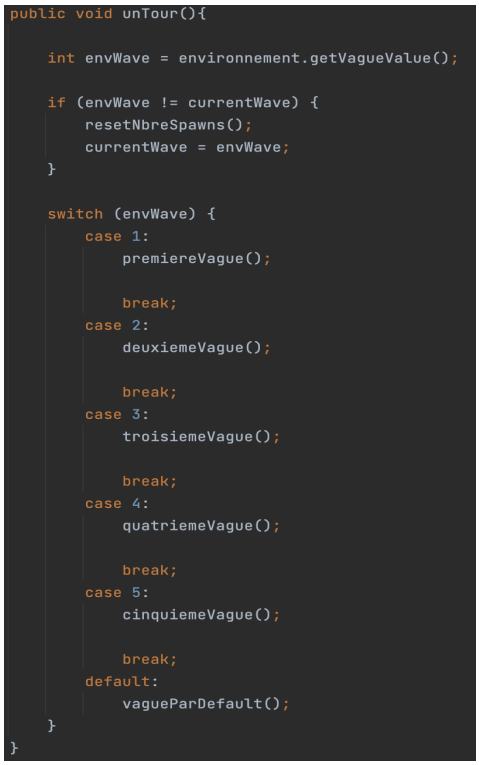
Désormais, la duplication de code de la méthode creation Projectile dans les classes est désormais supprimer, c’est maintenant un nouvel attribut de la classe Tours Offensives, Forges Projectiles qui se charge de la création du Projectile, création variant selon le type de cet attribut, solution idéales si l'on rajoute d’autre projectile et d’autre tour à l’avenir.

Comme autre solution pour la classe Tour, nous avons aussi effectué des suppressions de redondance :

Tout d’abord les méthodes ennemiÀporter() et verificationEnnemiÀporter() ont été remanié pour supprimer la duplication de code (calcul de la distance) qu’il y avait entre ces méthodes.

CLASSE Vagues : Problèmes

Il y a beaucoup de problèmes dans cette classe. Le premier étant sa trop forte responsabilité. Il faut qu'elle en délègue certaines. De plus, elle n'est pas assez objet et qu'elle pourrait être remaniée pour l'être encore plus (avec la gestion des vagues et l'apparition des ennemis.



```
public void unTour(){

    int envWave = environnement.getVagueValue();

    if (envWave != currentWave) {
        resetNbreSpawns();
        currentWave = envWave;
    }

    switch (envWave) {
        case 1:
            premiereVague();

            break;
        case 2:
            deuxiemeVague();

            break;
        case 3:
            troisiemeVague();

            break;
        case 4:
            quatriemeVague();

            break;
        case 5:
            cinquiemeVague();

            break;
        default:
            vagueParDefault();
    }
}
```

L'instruction switch est codée en dur pour traiter exactement cinq cas plus un cas par défaut. Cette conception n'est pas évolutive. Si le jeu doit prendre en charge davantage de vagues, la méthode doit être modifiée. Envisagez plutôt d'utiliser un modèle de stratégie dans lequel chaque vague possède sa propre classe implémentant une interface commune. Principe de la responsabilité unique : la méthode semble avoir plusieurs responsabilités : elle vérifie s'il y a un changement de vague, réinitialise les numéros de spawn si nécessaire, et exécute ensuite la logique pour la vague en cours. Idéalement, une méthode ne devrait faire qu'une seule chose. Envisagez de la diviser en plusieurs méthodes. Duplication : Il y a un schéma de répétition avec les appels à premiereVague, deuxiemeVague, etc.

Il y a beaucoup de duplication de code avec notamment les cinqs méthodes pour chaque vague différente comme vous pouvez le voir ci-dessous:

```java
public void deuxiemeVague(){

    int maxSoldiersType1 = 8;
    int maxSoldiersType2 = 6;

    totalSoldats = maxSoldiersType1 + maxSoldiersType2;

    Random random = new Random();

    if ((environnement.getNbrTours() % 15) == 0){

        if ((nbreSpawnsType1 < maxSoldiersType1) || (nbreSpawnsType2 < maxSoldiersType2)) {
            int soldierTypeToSpawn = random.nextInt(2) + 1; // Cela générere soit 1 soit 2

            if ((soldierTypeToSpawn == 1) && (nbreSpawnsType1 < maxSoldiersType1)) {
                System.out.println("Un nouveau Rookie apparait !");
                nouveauSpawnSoldat( typeSoldat: 1,  spawn: 9);
                nbreSpawnsType1++;
            } else if ((soldierTypeToSpawn == 2) && (nbreSpawnsType2 < maxSoldiersType2)) {
                System.out.println("Un nouveau Super Nova apparait !");
                nouveauSpawnSoldat( typeSoldat: 2,  spawn: 9);
                nbreSpawnsType2++;
            }
        }
    }
}
```

```java
public void troisiemeVague(){
    int maxSoldiersType1 = 10;
    int maxSoldiersType2 = 8;

    totalSoldats = maxSoldiersType1 + maxSoldiersType2;

    Random random = new Random();

    if ((environnement.getNbrTours() % 12) == 0) {

        if ((nbreSpawnsType1 < maxSoldiersType1) || (nbreSpawnsType2 < maxSoldiersType2)) {
            int soldierTypeToSpawn = random.nextInt(2) + 1; // Cela générere soit 1 soit 2

            if ((soldierTypeToSpawn == 1) && (nbreSpawnsType1 < maxSoldiersType1)) {
                System.out.println("Un nouveau Rookie apparait !");
                nouveauSpawnSoldat( typeSoldat: 1,  spawn: 12);
                nbreSpawnsType1++;
            } else if ((soldierTypeToSpawn == 2) && (nbreSpawnsType2 < maxSoldiersType2)) {
                System.out.println("Un nouveau Super Nova apparait !");
                nouveauSpawnSoldat( typeSoldat: 2,  spawn: 12);
                nbreSpawnsType2++;
            }
        }
    }
}
```

Le même schéma que la sélection des vagues a été reproduit avec la création des soldats, c’est-à-dire avec un switch-case:

```java
public Soldat selectionSoldat(int typeSoldat, double startX, double startY) {

    Soldat s;

    switch(typeSoldat) {
        case 1:
            s = new Rookie((int) startX, (int) startY, 89, 49);
            break;
        case 2:
            s = new SuperNova((int) startX, (int) startY, 89, 49);
            break;
        case 3:
            s = new Shichibukais((int) startX, (int) startY, 89, 49);
            break;
        default:
            s = new Rookie((int) startX, (int) startY, 89, 49);
    }

    return s;
}
```

La méthode effectue à la fois la sélection du type de soldat et la création de l'instance. Si les processus de sélection et de création sont susceptibles de se complexifier, les séparer pourrait être avantageux.

Si l'ajout de nouveaux types de Soldat est fréquent, l'utilisation d'un Factory Pattern pourrait simplifier la méthode en externalisant la logique de création dans une classe Factory dédiée.

De plus, il y a quelques soucis mineurs comme les noms de certaines méthodes qui ne sont pas assez descriptifs et qui pourraient l'être davantage comme dans la méthode qui suit:

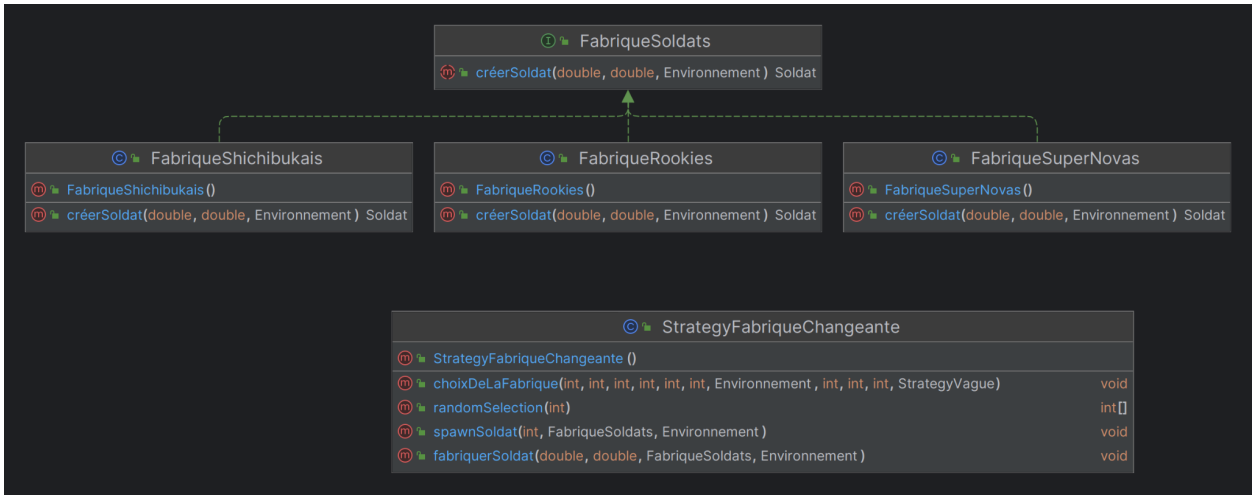
```
public Soldat afficherSoldat(double startX, double startY, int typeSoldat) {
    Soldat s = selectionSoldat(typeSoldat, startX, startY);
    listeSoldats.add(s);

    return s;
}
```

Pour finir sur cette classe, il y a un dernier problème: les attributs. Il y a des attributs qui ne servent à rien et ce n'est pas un problème seulement lié à cette classe mais un problème récurrent lié à la quasi-totalité des classes du jeu.

```
1 usage
private int numeroDeVague;
2 usages
private ListProperty<Soldat> listeSoldats;

public Vagues(Environnement environnement) {
    this.environnement = environnement;
    this.listeSoldats = environnement.getSoldatsProperty();
    this.numeroDeVague = environnement.getVagueValue();
    this.nbreSpawnsType1 = 0;
    this.nbreSpawnsType2 = 0;
    this.nbreSpawnsType3 = 0;
}
```

A l'exemple sur cette photo, les attributs numeroDeVague de type int et listeSoldats de type ListProperty<Soldat> ne servent à rien car on peut y accéder directement via l'attribut Environnement qui est dans cette classe.

CLASSE Vagues: Solutions (=> FabriqueSoldats)



créerSoldat(double, double, Environnement) : Il s'agit d'une méthode définie dans les classes de fabrique qui prend des coordonnées et un objet `Environnement` comme paramètres pour créer une instance de `Soldat`.

FabriqueShichibukais, FabriqueRookies, FabriqueSuperNovas : Ces classes sont des "fabriques" dans le design pattern Factory. Leur rôle est de créer des objets `Soldat`. Chaque fabrique crée des soldats avec des caractéristiques différentes, comme indiqué par leurs noms distincts qui représente différents types de soldats dans le jeu.

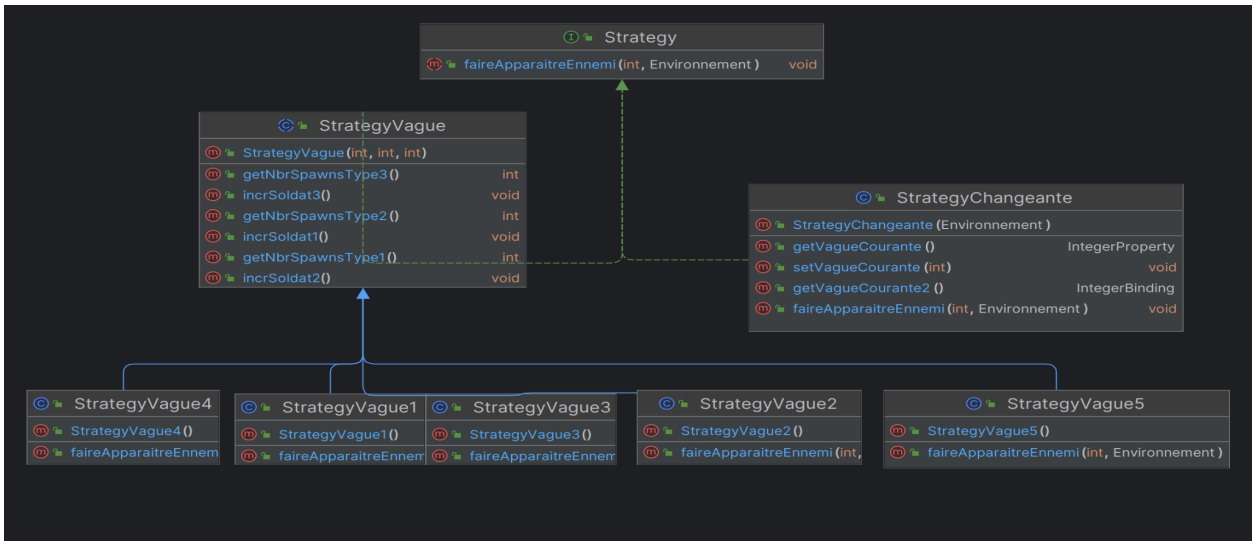
StrategyFabriqueChangeante : Cette classe implémente le design pattern Strategy, permettant de changer la stratégie de création des soldats. Elle contient des logiques pour choisir entre différentes fabriques de soldats en fonction de certaines conditions.

choixDeLaFabrique(int, int, int, int, Environnement, int, int, int, StrategyVague) : Cette méthode est responsable de la sélection d'une fabrique de soldats basée sur un ensemble de paramètres.

spawnSoldat(int, FabriqueSoldats, Environnement) : Cette méthode contrôle l'invocation de soldats dans l'environnement de jeu en utilisant une fabrique spécifique.

CLASSE Vagues: Solutions (=> StratégieDesVagues)

Solution idéales aux problèmes posés par notre précédente classe Vague. Plus objet, plus propre, plus lisible, solution ouvert fermés, etc. Supprime également la redondance de code qui était présent dans la classe vague.



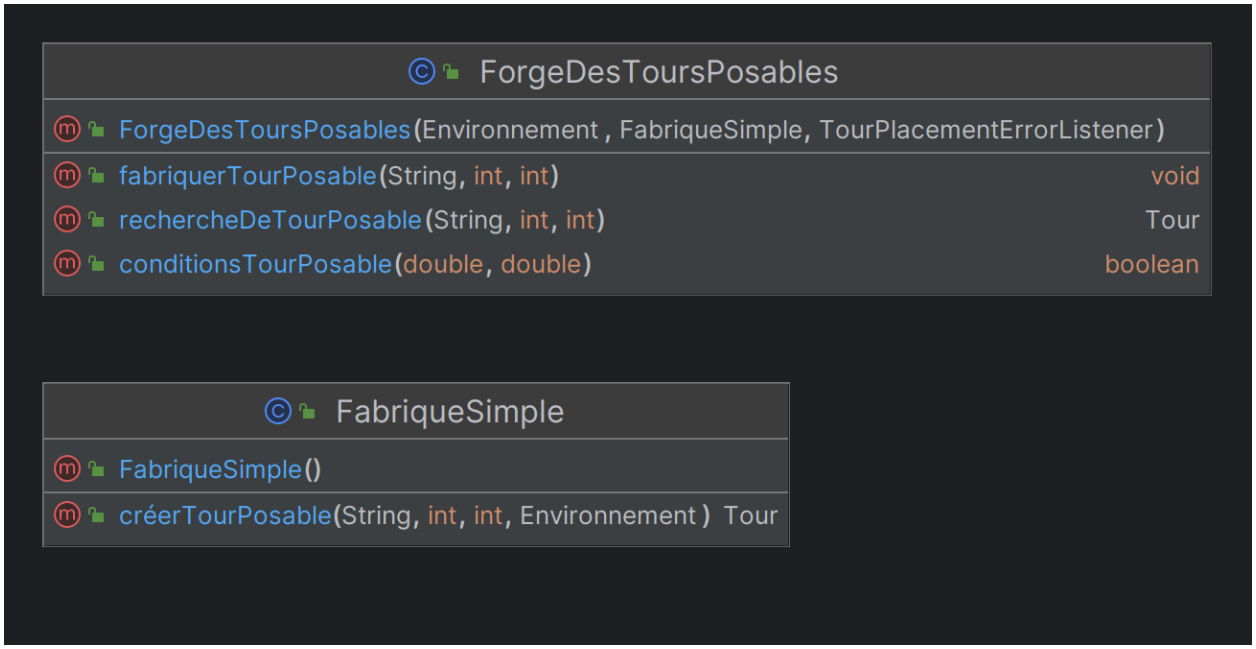
Dans ce package, nous avons utilisé le design pattern Strategy

1. **Strategy** : L'interface fonctionnelle de base pour les stratégies des vagues.
    - **faireApparaitreEnnemi(int, Environnement)** : Une méthode qui est responsable de faire apparaître un ennemi dans l'environnement, où `int` est le nombre de tours.
2. **StrategyVague** : Une classe concrète qui implémente `Strategy`, spécifique à la gestion des vagues d'ennemis.
    - **getNbrSpawnsType3()**, **getNbrSpawnsType2()**, **getNbrSpawnsType1()** : Méthodes pour obtenir le nombre d'ennemis de chaque type à faire apparaître.
    - **incrSoldat3()**, **incrSoldat2()**, **incrSoldat1()** : Méthodes pour augmenter le compteur d'ennemis de chaque type, après qu'ils aient été générés.
3. **StrategyVague1**, **StrategyVague2**, **StrategyVague3**, **StrategyVague4**, **StrategyVague5** : Des classes dérivées de `StrategyVague`, chacune représentant une stratégie différente pour générer des vagues d'ennemis.
    - **faireApparaitreEnnemi(int, Environnement)** : Méthode héritée de l'interface de base, chargée pour implémenter des comportements spécifiques à chaque stratégie de vague.

4. **StrategyChangeante** : Une classe qui représente une stratégie changeante pour les vagues d'ennemis.
    - **faireApparaitreEnnemi(int, Environnement)** : Méthode héritée de `Strategy`, avec une implémentation qui permet de changer la manière dont les ennemis apparaissent en fonction de la vague courante.

Désormais chaque vague du jeu est devenue une classe à part entière, et pas seulement des méthodes, ce qui règle le problème de redondance et de flexibilité que posait la classe vague, améliorant donc la compréhension de ces dernières, leur réutilisation, leur organisation/encapsulation, etc.

PACKAGE ForgeDesToursPosables: Solutions



ForgeDesToursPosables : Cette classe est responsable de la création et de la gestion des "tours" dans un environnement de jeu, en utilisant un forge, un lieu où les objets sont créés.

fabriquerTourPosable(String, int, int) : Une méthode qui fabrique une tour avec des spécifications données, comme un type (String) et des coordonnées (int, int).

conditionsTourPosable(double, double) : Une méthode qui vérifie si les conditions pour poser une tour sont remplies, en utilisant des coordonnées.

FabriqueSimple : Cette classe est une fabrique plus simple que celle décrite précédemment pour la création de tours.

créerTourPosable(String, int, int, Environnement) : Une méthode pour créer une tour posable dans l'environnement donné, similaire à celle dans `ForgeDesToursPosables` mais avec moins de complexité.

CLASSE Environnement : Problèmes

Le premier problème de la classe Environnement est qu'elle se trouve incomber de trop de responsabilités. En effet, elle gère le quadrillage de la map, calcule le chemin du BFS et vérifie le passage d'un soldat dans une case précise comme vous pouvez le voir à l'exemple sur les images suivantes:

```java
1 usage
public void initQuadrillage() {
    quadrillage = new int[][]{
            {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            {0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            {0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            {0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            {1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
```

```java
1 usage
public void calculerChemin(int destX, int destY) {  // Méthod

    boolean[][] visited = new boolean[getYmax()][getXmax()];

    Queue<Integer> queue = new ArrayDeque<>();
    queue.offer(destX);
    queue.offer(destY);
    visited[destY][destX] = true;

    int[] dx = {0, 0, -1, 1};
    int[] dy = {-1, 1, 0, 0};

    while (!queue.isEmpty()) {
        int x = queue.poll();
        int y = queue.poll();

        for (int i = 0; i < 4; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];

            if (isValidMove(nx, ny) && !visited[ny][nx]) {
                queue.offer(nx);
                queue.offer(ny);
                visited[ny][nx] = true;
                distances[ny][nx] = distances[y][x] + 1;
            }
        }
    }
}
```

```java
private boolean isValidMove(int x, int y) {

    return x >= 0 && x < distances[0].length && y >= 0
            && y < distances.length && valeurDeLaCase(y, x) == 1;
}
```

CLASSE Environnement : Solutions

La solution apportée à ce premier problème fut la mise en place d'une classe BFS à part entière qui gère ses propres responsabilités et méthodes, comme vous pouvez le voir par vous même ci-dessous:

```
3 usages   aabdi +1 *
public class BFS {
    5 usages
    private Environnement environnement;

    4 usages
    private int[][] quadrillage;

    1 usage   aabdi
    public BFS(Environnement environnement) {...}

    1 usage   aabdi
    public void initQuadrillage() {...}

    1 usage   Rayankz +1
    public void calculerChemin(int destX, int destY) {...}

    2 usages   aabdi
    public boolean isValidMove(int x, int y) {...}

    4 usages   aabdi
    public int getYmax() {...}

    4 usages   aabdi
    public int getXmax() {...}

    2 usages   aabdi
    public int valeurDeLaCase(int i, int j) {...}
}
```

CLASSE Environnement : Problèmes

Le second problème de la classe Environnement est qu'elle fait encore trop de choses. Par exemple, elle s'occupe aussi de déplacer le soldat ce qui ne devrait pas être le cas. deplacementSoldats : ce qui est dans la boucle (si le soldat n'est pas piégé, le deplacer) devrait être une méthode de Soldat : c'est le soldat qui se déplace…Et puis la méthode deplacerSoldat devrait être dans Soldat.

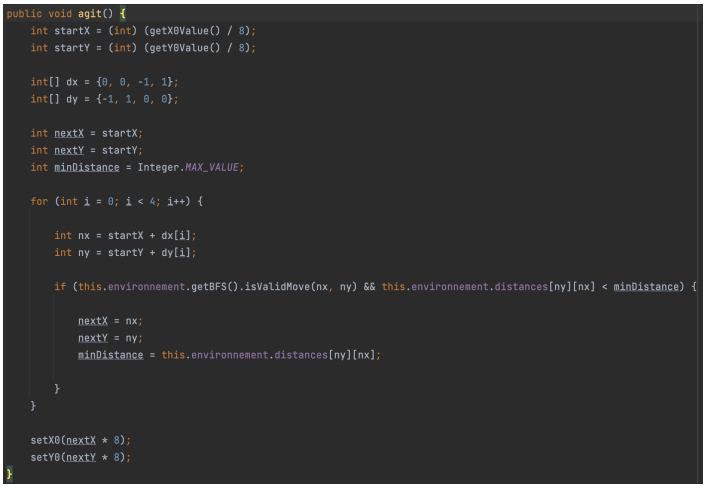
```
1 usage
public void deplacerSoldat(Soldat soldat) {
    int startX = (int) (soldat.getX0Value() / 8);
    int startY = (int) (soldat.getY0Value() / 8);

    int[] dx = {0, 0, -1, 1};
    int[] dy = {-1, 1, 0, 0};

    int nextX = startX;
    int nextY = startY;
    int minDistance = Integer.MAX_VALUE;

    for (int i = 0; i < 4; i++) {
        int nx = startX + dx[i];
        int ny = startY + dy[i];

        if (isValidMove(nx, ny) && distances[ny][nx] < minDistance) {
            nextX = nx;
            nextY = ny;
            minDistance = distances[ny][nx];
        }
    }
    soldat.setX0(nextX * 8);
    soldat.setY0(nextY * 8);
}
```

```
1 usage
public void deplacementSoldat(int n){
    if (!listeSoldats.isEmpty()) {
        for (Soldat soldat : listeSoldats) {
            if(!soldat.isEstPiégé() || n%2==0)
            deplacerSoldat(soldat);
        }
    }
}
```

CLASSE Environnement : Solutions

La solution apportée au second problème est le déplacement de la méthode déplacerSoldat dans la classe Soldat tout simplement et en simplifiant la méthode deplacementSoldat pour qu'elle ait moins de charges (qu'elle ne s'occupe pas du fait que le soldat soit piégé ou non par exemple comme vous pouvez le voir ci-dessous:

La méthode deplacerSoldat a été placée dans la classe Soldat (à la seule différence qu'elle a changé de nom et est devenue la méthode agit()).



```
public void agit() {
    int startX = (int) (getX0Value() / 8);
    int startY = (int) (getY0Value() / 8);

    int[] dx = {0, 0, -1, 1};
    int[] dy = {-1, 1, 0, 0};

    int nextX = startX;
    int nextY = startY;
    int minDistance = Integer.MAX_VALUE;

    for (int i = 0; i < 4; i++) {

        int nx = startX + dx[i];
        int ny = startY + dy[i];

        if (this.environnement.getBFS().isValidMove(nx, ny) && this.environnement.distances[ny][nx] < minDistance) {

            nextX = nx;
            nextY = ny;
            minDistance = this.environnement.distances[ny][nx];

        }
    }

    setX0(nextX * 8);
    setY0(nextY * 8);
}
```

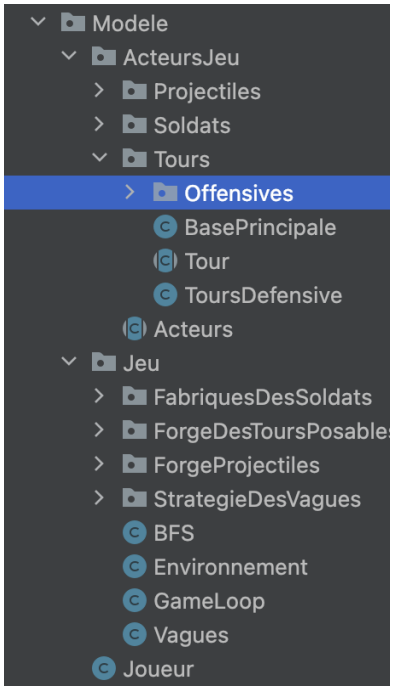
Nous avons ensuite conçu à la classe Soldat sa propre méthode de déplacementSoldat:

```
1 usage   aabdi *
public void déplacementSoldat(int n) {
    if(!this.isEstPiégé() || n % 2 == 0) {
        this.agit();
    }
}
```

Pour finir, voici la version finale de l'ancienne méthode deplacementSoldat de l'Environnement vue auparavant:

```
1 usage   aabdi +1
public void actionDesSoldats(int n) {

    if (!this.listeSoldats.isEmpty()) {

        for (Soldat soldat : this.listeSoldats) {

            soldat.déplacementSoldat(n);
        }
    }
}
```

Présentation du Jeu actuelle :



Voici notre code remanier qui désormais fait usage de design pattern,avec des responsabilités un peu plus étalées entre les classes, beaucoup de duplications de code et nous avons fait le nécessaire pour rendre le code plus lisible possible.