BEN EL MOSTAPHA Mohamed
XU Nic
Licence informatique 2e année
Université Paris-Cité

POO - 2024

Projet Tower Defense

Table des matières

Introduction 3
En quoi consiste le projet 3
Comment se joue le Tower Defense 3
Outils utilisés 3
Plateforme collaborative 3
L'utilisation de l'IA 3
Choix de l'IDE 4
Contraintes (demandées par le sujet) 4
Utilisation de Java SwingFX 4
Architecture 4
Conception 4
Utilisation du MVC et explication de son principe 4
Structure du projet (+ diagramme en annexe) 5
Encapsulation 5
Principe de Responsabilité Unique 5
Efficacité 5
Utilisation de Design Pattern 5
Implémentations 6
Gestion de la carte 6
La map 6
L'éditeur de la map 6
Gestion des tours de défense 6
Fonctionnement général 6
Système d'achat et d'amélioration 6
Gestion des ennemis et système de tir 7
La difficulté du jeu 7
Les déplacements ennemis 7
Les projectiles 7
Gestion des vagues 8
Test 8
Améliorations possibles 8
Conclusion 8
Annexe 10

Introduction

En quoi consiste le projet

L'objectif de ce projet est de créer un Tower Defense, avec comme seule base, les connaissances acquises lors du premier semestre de la deuxième année de licence informatique en Programmation Orientée Objet. L'état final de ce projet témoignera ainsi de la compréhension des principes fondamentaux (l'héritage, le MVC, …) des membres du groupe.

Comment se joue le Tower Defense

Dans ce projet, le joueur pourra éditer sa propre map et ainsi son propre chemin* et placer le point d'apparition des monstres (le Spawn) et la tour (la Target). Une fois la map finie, les monstres suivront le chemin dessiné dans le but d'atteindre la tour. L'objectif du joueur sera de neutraliser les monstres en achetant des tours de défense qu'il devra placer dans des endroits stratégiques.

*Le chemin ne doit pas être coupé entre le Spawn et la Target, le jeu perd son intérêt si les monstres ne peuvent pas atteindre la cible.

Outils utilisés

Plateforme collaborative

Github a été utilisée comme plateforme de codage collaborative, discord a été utilisée comme moyen de communication.

Parsec a été également utilisée occasionnellement pour des sessions de codage en simultané.

L'utilisation de l'IA

L'aide de l'Intelligence Artificielle a été sollicitée durant ce projet, notamment le plus connu actuellement, ChatGPT qui donne la plupart du temps de bonnes suggestions mais ses réponses deviennent génériques et inutiles lorsque les questions deviennent plus spécifiques, cela s'explique par son fonctionnement qui dépend de la quantité de données analysées pour fournir les réponses. C'est d'ailleurs pour cela qu'il a tendance à mieux nous satisfaire lorsqu'il s'agit de web design et surtout pour du css, car le principe reste plus ou moins la même peu importe le projet.

Github Copilot a également été utilisé, majoritairement pour sa fonctionnalité d'auto-complétions qui analyse le code / le projet et propose automatiquement une suite du code que le programmeur a entamé en fonction de ce qu'il vient de faire ou du commentaire/nom de la fonction qu'il vient de mettre. Permettant ainsi de gagner du temps.

Github Copilot générant la suite du code

Choix de l'IDE

Le projet a été codé avec l'IDE Microsoft Visual Studio Code avec le pack d'extensions Java et Git lens/graph* pour pouvoir commit/push, pull et merge directement depuis l'IDE, le rendant ainsi plus polyvalent.

*L'application Github Desktop a également été utilisée pour commit/push/pull pour son interface graphique le rendant simple et intuitif.

Contraintes (demandées par le sujet)

Utilisation de Java SwingFX

L'utilisation autre que Swing est prohibée, seule l'utilisation de Swing (et d'autres librairies vues en cours) est autorisée. L'utilisation des librairies vues dans les autres UE informatiques est interdite (comme Gradle)

Architecture

Le projet devra être structuré de sorte à séparer le modèle de la vue, cette fragmentation rend une éventuelle évolution du projet beaucoup plus simple à faire.

Conception

Utilisation du MVC et explication de son principe

Afin de répondre aux contraintes structurelles, nous avons décidé d'utiliser le MVC (Modèle – Vue – Contrôleur) qui est un design pattern (patron de conception) souvent utilisé pour le web. La vue ne pouvant pas directement interagir avec la base de données, elle se charge des formulaires remplis par l'utilisateur, qui sont récupérés par le contrôleur. Le contrôleur va ainsi interroger le modèle avec les données du formulaire qui va à son tour interroger la BDD (base de données). Les données renvoyées par la BDD suivent le chemin inverse.

En JAVA, tout est sous forme d'objets qui sont alors accessibles partout (sans prendre en compte public, private, protected, …), le contrôleur n'est donc plus essentiel, cependant, le MVC n'est pas pour autant inutile en JAVA. En effet le code MVC permet de :

- Séparer le traitement des données et l'affichage
- Prévoir le futur et favoriser l'évolutivité
- Mieux diviser le code pour faciliter la compréhension et l'édition en mode projet
- D'alléger certains fichiers (car le projet a d'abord été en MV)

En outre, le MVC/MV rend le code plus lisible, plus compréhensible, plus évolutive, mais quant à son implémentation, elle n'en est pas moins compliquée. En effet, savoir comment structurer son code en amont est une difficulté majeure lorsque l'on crée un projet en partant de zéro. De plus, certaines sessions de codages, notamment aux premières semaines, ont été inutiles et ont dû être supprimés à la suite d'une restructuration complète du code.

Une session Parsec a été initiée lors d'une seconde phase de restructuration du code, visant à améliorer la structure du projet sans perturber excessivement le code préexistant et à éviter les défis rencontrés lors de la première restructuration. Cette démarche vise également à faciliter la resynchronisation des membres du groupe.

Une fois le code structuré, le code devient plus explicite et l'implémentation de nouvelles fonctionnalités devient plus triviale.

Structure du projet (+ diagramme en annexe)

Encapsulation

Chaque champ dans notre projet est soit privé, soit protégé (sauf pour les champs primitifs statiques finaux). Leur accès par les classes clientes est contrôlé à l'aide de getters, setters, itérateurs, etc. afin de s'assurer qu'il n'y a aucune violation de l'encapsulation.

Principe de Responsabilité Unique

Chaque classe a une responsabilité clairement identifiable et une nomenclature explicite, par exemple :

- MessagesView : gère l'affichage des messages à l'écran (alertes, messages informatifs...).

- HealthBar : gère l'affichage des barres de santé, réutilisable pour d'autres classes (EnemiesView, BaseView...).

- StringHelper : gère l'affichage des chaînes de caractères sur l'interface utilisateur. Le respect de ce principe a rationalisé le débogage et facilité la détection des erreurs, rendant le code plus utilisable.

Efficacité

Les HashMaps ont été utilisées pour charger des images et les réutiliser. Les floats ont été utilisés à la place des doubles lorsque la précision n'était pas nécessaire.

Utilisation de Design Pattern

Factory

Le modèle de conception de la méthode d'usine a été utilisé pour la création d'ennemis, de tours et de types de tuiles (voir les figures 1 à 3). Cela a rendu le code réutilisable et facile à comprendre. L'encapsulation de la logique de création dans une classe différente facilite l'extension.

State

Le projet a mis en œuvre ce modèle à la fois dans la vue et dans le modèle. L'éditeur de cartes utilise ce modèle en raison de la nature mutable de ses états. Les états de l'éditeur implémentent une interface fonctionnelle appelée MapEditorState, puis substituent la fonction handleState. La classe de l'éditeur de cartes a une référence à un objet implémentant l'interface MapEditorState, utilisé pour implémenter la logique de l'éditeur. (fig.4)
L'utilisation de state a également été utile pour les états de vagues. (fig.5)

Observer

Les tuiles représentent l'observable, et dès qu'un ennemi est sur une tuile, il notifie tous ses observateurs (qui sont des tours). De plus, lorsqu'un ennemi

quitte une tuile, il notifie également les observateurs de la tuile sur laquelle l'ennemi était avant de se déplacer.

Implémentations

Gestion de la carte

La map

La carte est un agrégat de tuiles, un bloc statique dessine la carte par défaut et place des fleurs aléatoirement autour. La classe de la carte fournit des accesseurs et des modificateurs pour les types de tuiles et les tuiles.

L'éditeur de la map

L'éditeur de cartes gère tout ce qui concerne la modification d'une carte, que ce soit le placement de tours, le changement de points d'apparition ou la modification de tuiles. De nouveaux états peuvent être ajoutés facilement à l'avenir, car l'éditeur de carte utilise l'interface MapEditorState pour gérer sa logique.

Gestion des tours de défense

Fonctionnement général

La gestion des tours de défense repose sur une organisation structurée sur plusieurs classes. Nous avons TowerModel, classe abstraite qui constitue la base pour toutes les tours. Elle détermine des attributs essentiels tels que la portée, le coût, le niveau, et le type de projectile utilisé. La gestion des tours est ensuite gérée par TowerManager qui établit les modalités de sauvegarde/placement/suppression des tours. Et enfin TowerFactory qui classe fournit une liste prédéfinie de types de tours ainsi que des méthodes pour obtenir le nombre total de tours en jeu. Les tours qui étendent de la classe modèle ont également un projectile qui leur est propre, déterminant ainsi leurs spécificités.

Le jeu offre trois types de tours distincts, un bunker caractérisé par une portée étendue et une cadence de tir élevée, un personnage doté d'une cadence de tir plus lente mais d'une portée moyenne avec un projectile infligeant des dégâts importants, et enfin, un chat polyvalent.
Les tours ont également la possibilité d'être améliorées.

```
@Override
public void upgrade(){
    if (level < 10) {
        if (ShmucklesModel.getShmuckles() - upgradeCost*count >= 0){
            projectile.upgrade();
            range += 1;
            cost += upgradeCost;
            level++;
            ShmucklesModel.setShmuckles(ShmucklesModel.getShmuckles() - upgradeCost*count);
            System.out.println(x:"Upgrade successful");
        }
        else {
            System.out.println(x:"Not enough shmuckles");
        }

    }
    else {
        System.out.println(x:"Tower is already at max level");
    }
}
```

Système d'achat et d'amélioration

Sélectionner une tour et la placer diminue le nombre de schmuckles du montant de la tour sélectionnée indiqué dans la section inférieure. Lorsqu'une tour est sélectionnée, deux boutons apparaissent dans la section inférieure : une flèche vers le haut et une autre vers le bas. En cliquant sur eux, vous pouvez améliorer/rétrograder la tour sélectionnée. Les tours déjà placées ne sont pas affectées par la mise à niveau.

Améliorer une tour signifie améliorer son projectile ainsi que d'autres statistiques. Le fait d'avoir séparé l'amélioration de la tour de l'amélioration du projectile dans le code offre beaucoup plus de flexibilité. Le fait d'avoir des projectiles en tant qu'entités distinctes nous permet de donner des projectiles différents à différentes tours à l'avenir.

Gestion des ennemis et système de tir

La difficulté du jeu

Le jeu propose trois niveaux de difficulté distincts, à savoir facile, moyen et difficile, que les joueurs peuvent sélectionner dans les paramètres. La complexité de chaque niveau est déterminée par une valeur numérique variant de 0.75 à 1.25. Cette valeur agit comme un multiplicateur, ajustant les attributs de certains monstres présents dans le jeu. Ainsi, cette mécanique permet de rendre ces créatures plus ou moins puissantes en fonction du niveau de difficulté choisi.

```java
public class MrBlobModel extends EnemyModel{
    private static int health = 30;
    private static int damage = 5;
    private static float speed = 0.01f  ;
    private static int attackSpeed = 1000;
    private static int id;

    public MrBlobModel() {
        super(health, speed * EnemyModel.getDifficultyMultiplierSpeed(), (int) (damage * EnemyModel.getDifficultyMultiplierSpeed()), attackSpeed);
    }
```

La difficulté ajuste la vitesse et les dégâts du Blob

Les déplacements ennemis

L'intelligence artificielle (IA) mise en œuvre suit des règles simples. Chaque ennemi est équipé d'un tableau bidimensionnel de booléens, ayant les mêmes dimensions que la carte du jeu. Chaque fois qu'un ennemi traverse une tuile (atteignant son centre), le booléen correspondant à sa position est défini sur True. Cette information est cruciale pour la logique de l'IA.

Logique de l'IA :

- Si une tuile a déjà été traversée, l'ennemi ne change pas de direction.
- Si la tuile n'a pas été traversée et que la tuile suivante n'est pas praticable, l'ennemi ajuste sa direction vers la première tuile praticable détectée, puis marque la tuile comme traversée.
- En l'absence de tuiles praticables, la direction est simplement définie sur AUCUNE.

Les projectiles

Implémentation

L'objectif était de diriger les projectiles dans une direction spécifique. Initialement, trouver une solution à ce problème était complexe. Cependant, après avoir réalisé que l'utilisation de vecteurs était appropriée dans ce contexte, les attributs xVelocity et yVelocity ont été introduits.

Explications mathématiques

```java
public void targetEnemy(EnemyModel enemy) {
    if(enemy==null){
        return;
    }
    // let B be the point where the projectile is fired from, and A be the point where the enemy is
    float xDistance = enemy.getX() - x; // xDistance represents the x coordinate of the vector AB
    float yDistance = enemy.getY() - y; // yDistance represents the y coordinate of the vector AB
    float distance = (float) Math.sqrt(Math.pow(xDistance, 2) + Math.pow(yDistance, 2)); // this is the norm of AB
    // we divide the xDistance and yDistance by the norm of AB to get the unit vector of AB (i.e. the direction of AB)
    xVelocity = speed*xDistance / (distance* 32); // we multiply by 32 because 1 Unit in our game is 32 pixels
    yVelocity = speed*yDistance / (distance *32);

}
```

En considérant A comme la position de l'ennemi et B comme la position initiale du projectile, le vecteur AB a été examiné :

- Obtenir la direction de AB équivaut à trouver la direction du projectile.

- Puisque AB = ||AB|| * u, où u est le vecteur de direction, calculer AB/||AB|| fournit une direction.
- En multipliant ||AB|| par la taille unitaire de notre jeu, nous obtenons le vecteur de direction en pixels.
- En multipliant AB/||AB|| * taille_en_pixel par une valeur "vitesse", la vitesse du projectile est définie en pixels par image.

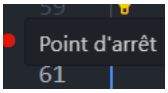
Ainsi, cette approche fournit une méthode efficace pour cibler les ennemis.

Gestion des vagues

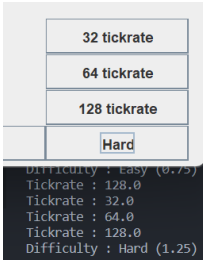
```
public static void calculatePercentageOfCTierEnemies() {

    // at least 20% of enemies are C tier, at most 60% of enemies are C tier. Their percentage decreases by 40% each wave

    numberOfCTierEnemies = Math.max((20 * numberOfEnemies) / 100, numberOfEnemies - (40*numberOfEnemies)/100);
    System.out.println("C-tier ennemies : " + numberOfCTierEnemies);
}

public static void calculatePercentageOfBTierEnemies() {

    // 3/4 of the remaining enemies are B tier

    if(waveNumber>5/GameModel.getDifficulty()) numberOfBTierEnemies = (3*(numberOfEnemies - numberOfCTierEnemies))/4;
    else numberOfBTierEnemies = numberOfEnemies - numberOfCTierEnemies;
    System.out.println("B-tier ennemies : " + numberOfBTierEnemies);
}

public static void calculatePercentageOfATierEnemies() {

    // the rest are A tier if and only if the wave number is greater than 5

    if(waveNumber>5/GameModel.getDifficulty()) numberOfATierEnemies = (numberOfEnemies - numberOfCTierEnemies)/4;
    System.out.println("A-tier ennemies : " + numberOfATierEnemies);
}
```

L'implémentation des ennemis s'articule autour de trois catégories définies en fonction de leur niveau de difficulté, à savoir A, B et C-tier. Cela permet d'ajuster non seulement le nombre d'ennemis mais également la difficulté des vagues au fur et à mesure de la progression.

Test



À la différence du projet de pré-professionnalisation, les tests unitaires et d'intégration n'ont pas été employés dans cette phase. (À noter qu'une fois la structuration faite, repérer les erreurs devient bien plus rapide.)



Des déclarations d'impression (print) ont été utilisées comme moyen de vérification du bon fonctionnement des fonctionnalités. Lorsqu'une erreur n'était pas immédiatement identifiable, des points d'arrêt (break points) ont été utilisés pour isoler le problème.

Améliorations possibles

- Dimensions dynamiques : pour des raisons pratiques, nous avons choisi d'avoir une taille fixe pour notre jeu. Cependant, après avoir effectué davantage de recherches après le projet, nous aurions pu utiliser WrapLayout pour avoir des dimensions dynamiques.
- Conventions de nommage : il existe quelques incohérences dans nos conventions de nommage, qui pourraient certainement bénéficier d'un travail supplémentaire.
- Plus de documentation : le code est lisible, mais il pourrait utiliser plus de commentaires pour s'assurer que le code est entièrement compris et facilement utilisable.
- Meilleure détection des ennemis : utiliser le modèle d'observateur pour observer les ennemis est une approche intelligente, mais étant donné que les tuiles notifient uniquement les tours de l'entrée et du départ, placer une tour à côté d'un ennemi immobile constitue un point aveugle (peut-être implémenter la notification de manière à ce qu'elle notifie continuellement les tours jusqu'à ce que les ennemis soient définis comme cibles).

Conclusion

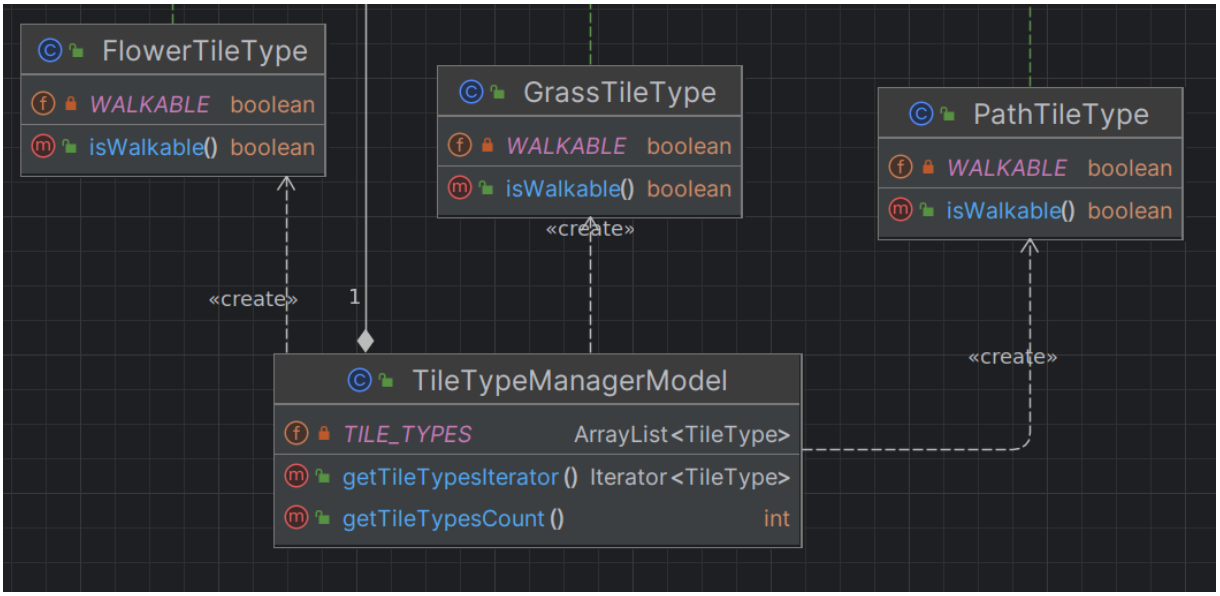
Ce projet de Tower Defense constitue une immersion approfondie dans le monde de la programmation orientée objet, mettant en lumière les principes fondamentaux tels que l'héritage, le modèle MVC, et l'utilisation de librairies comme Java SwingFX. L'approche modulaire adoptée, avec des

classes bien définies comme TowerModel, TowerManagerModel, et TowerFactory, offre une gestion flexible et extensible des tours de défense.

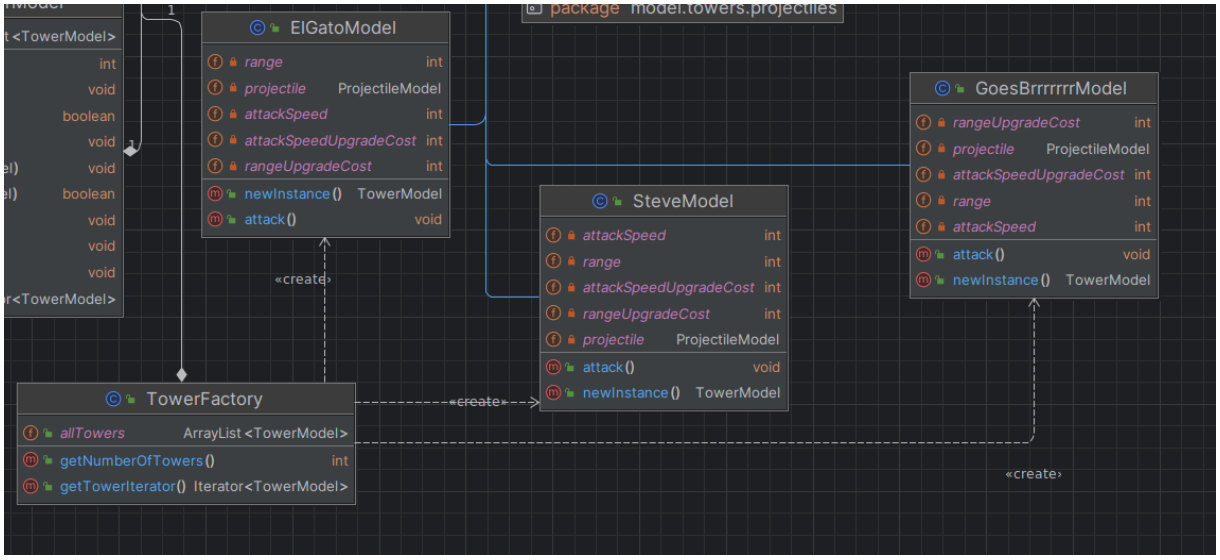
L'utilisation judicieuse de design patterns tels que Factory et State renforce la clarté du code et favorise la maintenabilité (surtout grâce au MVC). Les différentes implémentations.

Malgré les défis rencontrés, notamment lors de phases de restructuration du code, ces expériences ont été bénéfiques, offrant une vision plus claire de la conception logicielle. Les perspectives d'amélioration, comme l'intégration de tests unitaires et l'optimisation des déplacements ennemis, ouvrent la voie à des développements futurs. Ce projet représente ainsi une étape significative dans notre parcours d'apprentissage en programmation orientée objet.

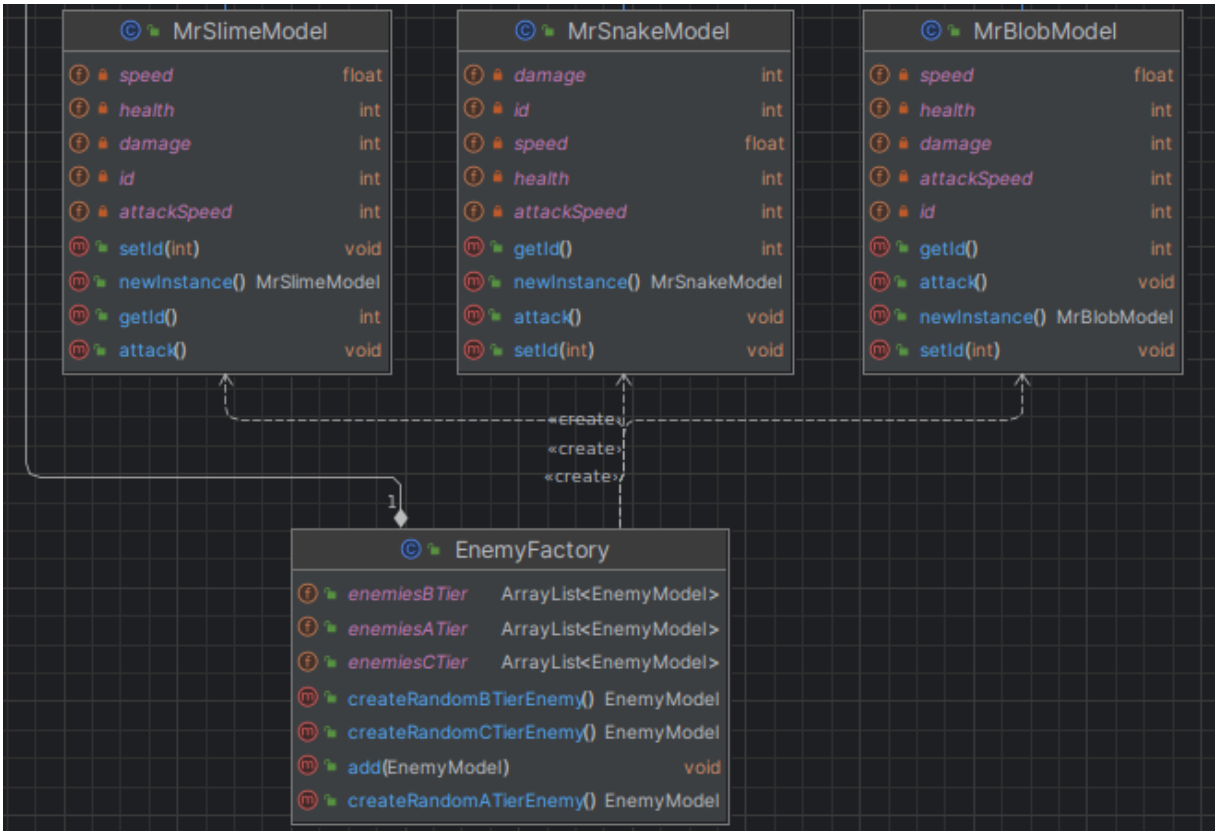
Annexe



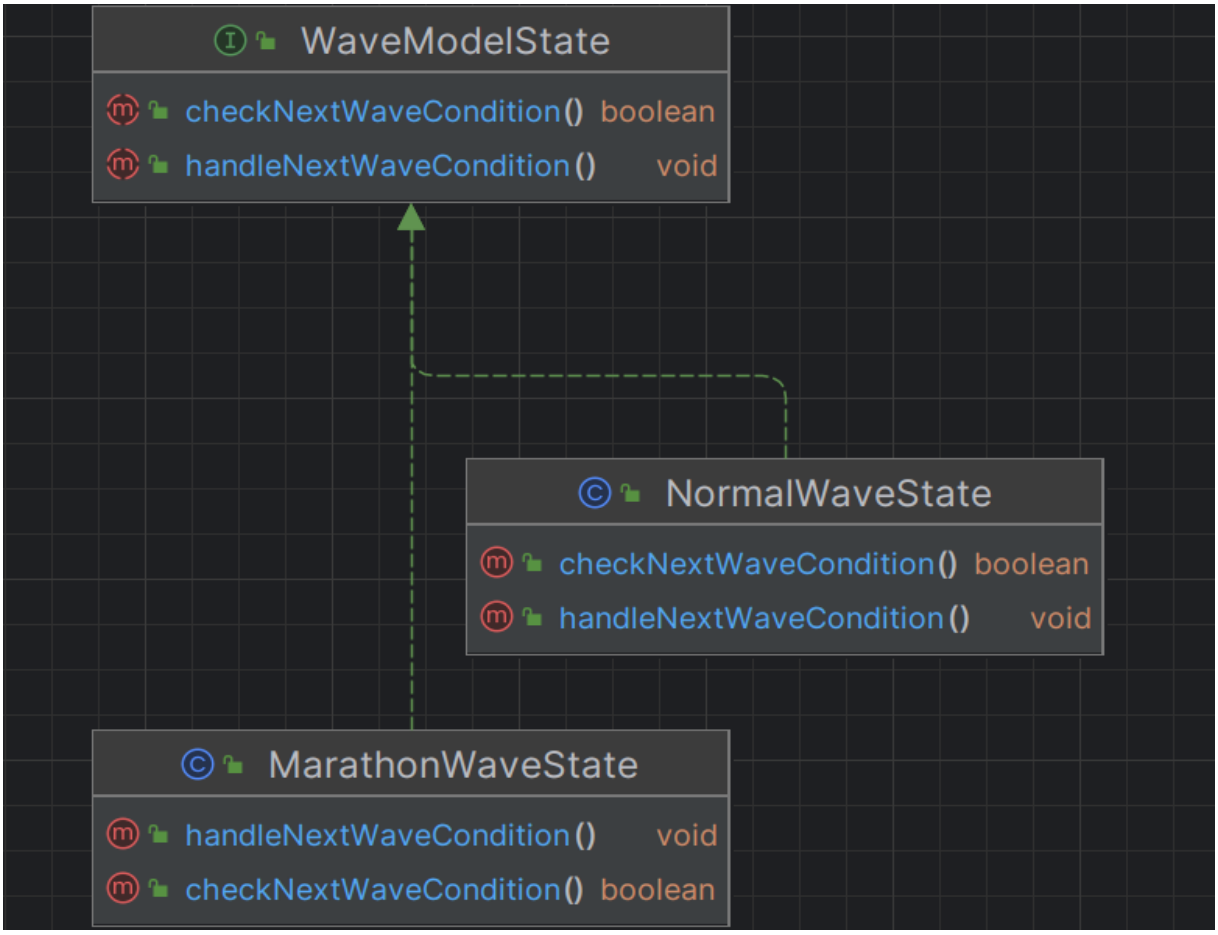
(Fig 1)



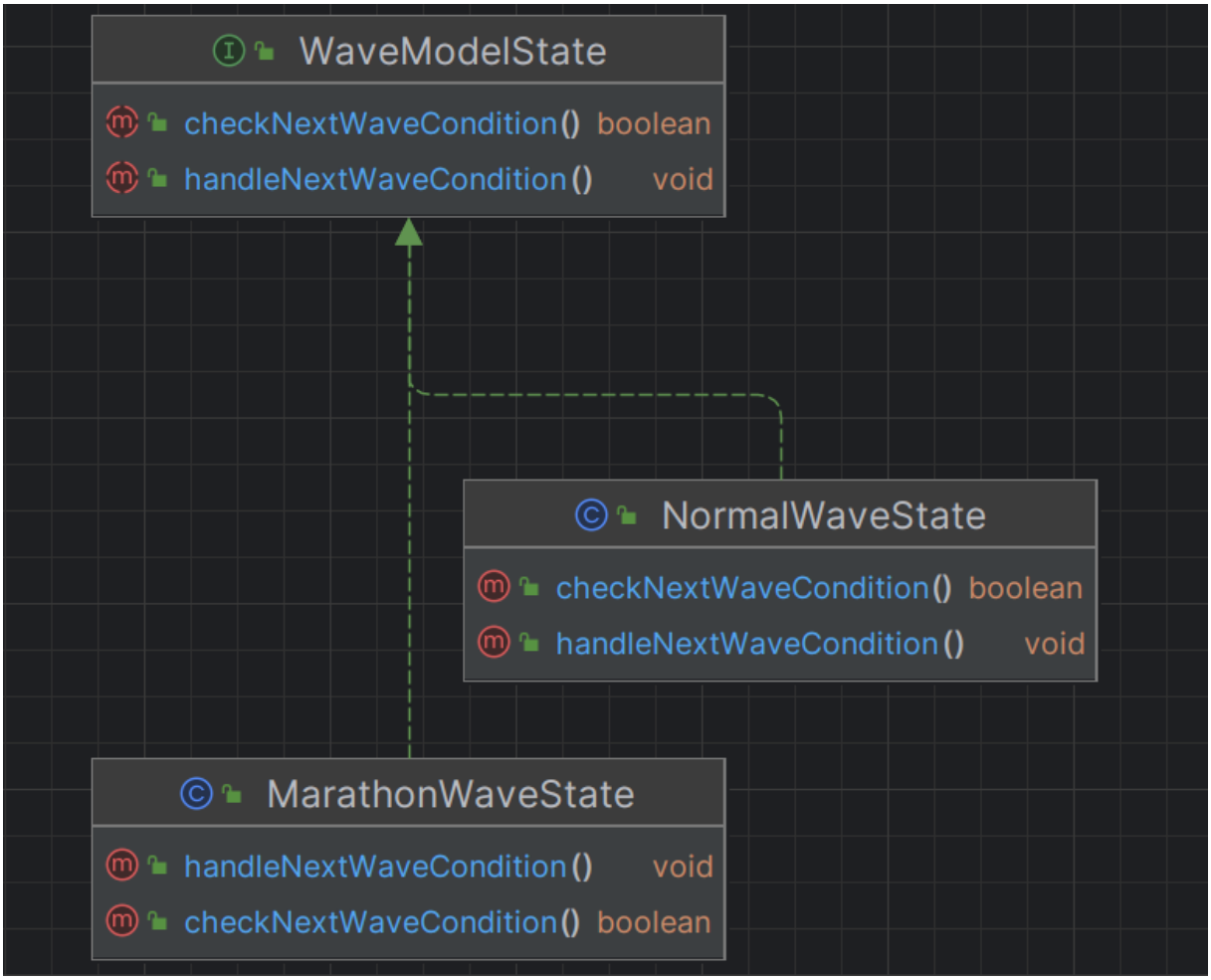
(Fig 2)


MrSlimeModel
speed float
health int
damage int
id int
attackSpeed int
setId(int) void
newInstance() MrSlimeModel
getId() int
attack() void
MrSnakeModel
damage int
id int
speed float
health int
attackSpeed int
getId() int
newInstance() MrSnakeModel
attack() void
setId(int) void
MrBlobModel
speed float
health int
damage int
attackSpeed int
id int
getId() int
attack() void
newInstance() MrBlobModel
setId(int) void
«create»
«create»
«create»
1
EnemyFactory
enemiesBTier ArrayList<EnemyModel>
enemiesATier ArrayList<EnemyModel>
enemiesCTier ArrayList<EnemyModel>
createRandomBTierEnemy() EnemyModel
createRandomCTierEnemy() EnemyModel
add(EnemyModel) void
createRandomATierEnemy() EnemyModel

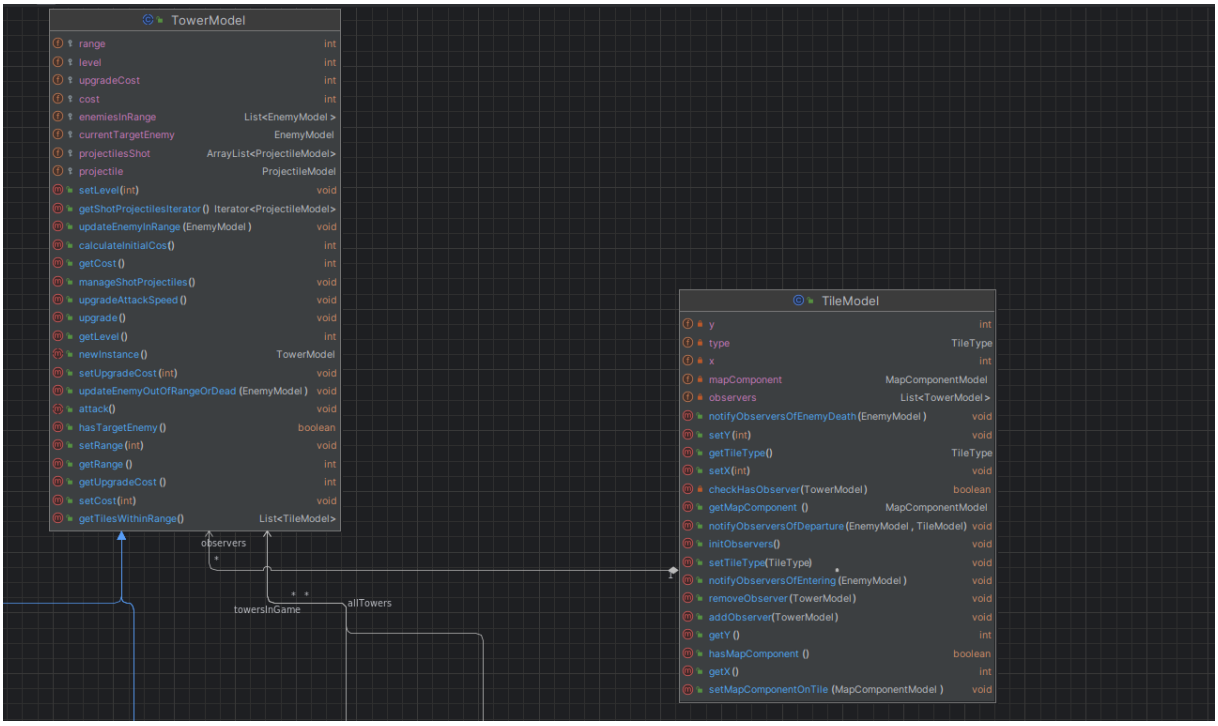
(Fig 3)


WaveModelState
checkNextWaveCondition() boolean
handleNextWaveCondition() void
NormalWaveState
checkNextWaveCondition() boolean
handleNextWaveCondition() void
MarathonWaveState
handleNextWaveCondition() void
checkNextWaveCondition() boolean

(Fig 4)


WaveModelState
checkNextWaveCondition() boolean
handleNextWaveCondition() void
NormalWaveState
checkNextWaveCondition() boolean
handleNextWaveCondition() void
MarathonWaveState
handleNextWaveCondition() void
checkNextWaveCondition() boolean

(Fig 5)


TowerModel
range int
level int
upgradeCost int
cost int
enemiesInRange List<EnemyModel>
currentTargetEnemy EnemyModel
projectilesShot ArrayList<ProjectileModel>
projectile ProjectileModel
setLevel(int) void
getShotProjectilesIterator() Iterator<ProjectileModel>
updateEnemyInRange(EnemyModel) void
calculateInitialCos() int
getCost() int
manageShotProjectiles() void
upgradeAttackSpeed() void
upgrade() void
getLevel() int
newInstance() TowerModel
setUpgradeCost(int) void
updateEnemyOutOfRangeOrDead(EnemyModel) void
attack() void
hasTargetEnemy() boolean
setRange(int) void
getRange() int
getUpgradeCost() int
setCost(int) void
getTilesWithinRange() List<TileModel>
observers
towersInGame
allTowers
TileModel
y int
type TileType
x int
mapComponent MapComponentModel
observers List<TowerModel>
notifyObserversOfEnemyDeath(EnemyModel) void
setY(int) void
getTileType() TileType
setX(int) void
checkHasObserver(TowerModel) boolean
getMapComponent() MapComponentModel
notifyObserversOfDeparture(EnemyModel, TileModel) void
initObservers() void
setTileType(TileType) void
notifyObserversOfEntering(EnemyModel) void
removeObserver(TowerModel) void
addObserver(TowerModel) void
getY() int
hasMapComponent() boolean
getX() int
setMapComponentOnTile(MapComponentModel) void

(Fig 6)