

BYTE DEFENSE

DOSSIER DU PROJET

2019 - 2020

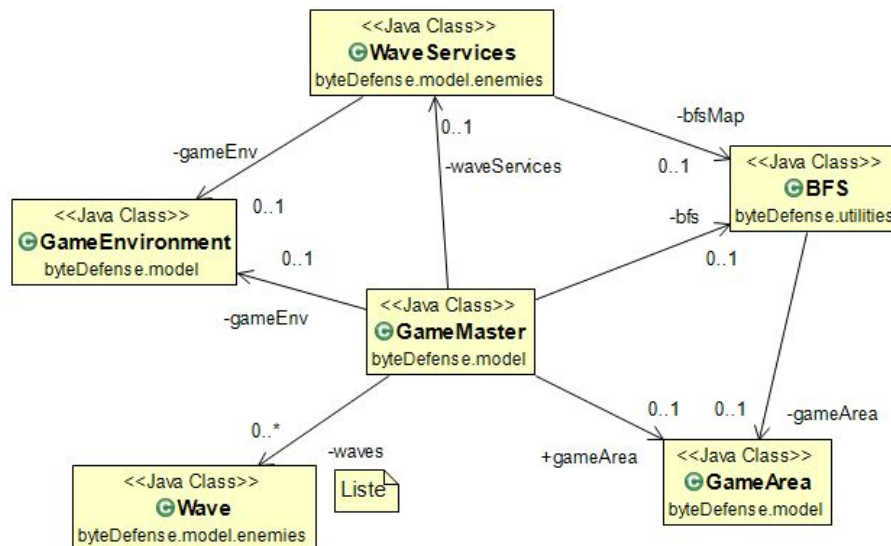
Sommaire :

Documents pour CPOO	3	Documents pour Gestion de projet	18
Architecture	3	Document utilisateur	18
Détails : diagrammes de classe	5	Screenplay	18
Diagramme 1 : GameMaster, ses dépendances et responsabilités	5	Tableau des relations	18
Diagramme 2 : GameEnvironment ou l'environnement de jeu	6	Les ennemis	19
Diagramme 3 : GameObject et ses sous-classes	7	Adware	19
Diagramme 4 : Découpage de la classe Enemy	8	Rootkit	19
Diagramme 5 : Effets propres et effets infligés aux LivingObject	9	Bot	19
Diagrammes de séquence	10	Ransomware	19
Déplacement des ennemis :	10	Spyware	19
Drag and drop achat d'une tourelle :	11	Trojan Horse	19
Structures de données	12	Vague des ennemis	19
La représentation du terrain	12	Les tourelles	20
Le chemin fourni par le BFS	12	AdCube	20
Gestion des vagues	13	Antivirus	20
Exception	14	Authentication Point	20
GameMaster.java	14	Firewall	20
EnemyFactory.java	15	SudVPN	20
Utilisation maîtrisée d'algorithmes intéressants	15	L'ordinateur	20
Le pathfinding grâce au BFS (Breadth-First-Search)	15	L'interface utilisateur	21
Lecture et construction des listes de tuiles	16	Magasin	22
Units	17	Déplacement et vente	22
		Barre d'infection	22
		Messages	22
		Minuteur	22
		Etat de la partie	23
		Boutons	23
		Scoring	23

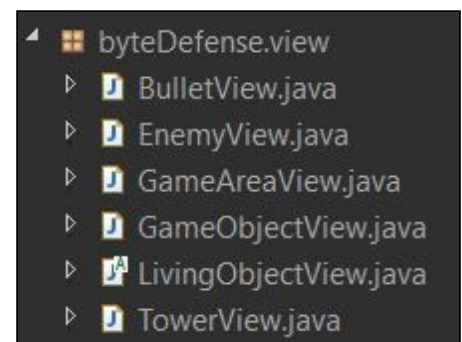
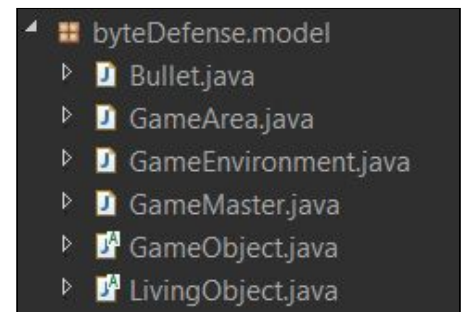


Architecture

GameMaster, ou “maître de jeu”, est la classe mère du modèle : c’est elle qui permet le déroulement des actions du jeu à partir des données des autres classes :

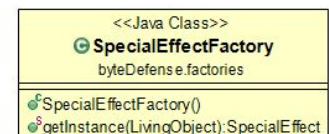
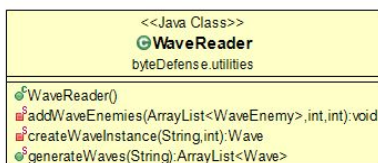
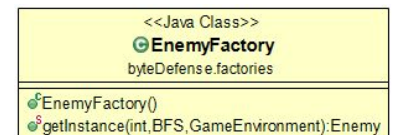
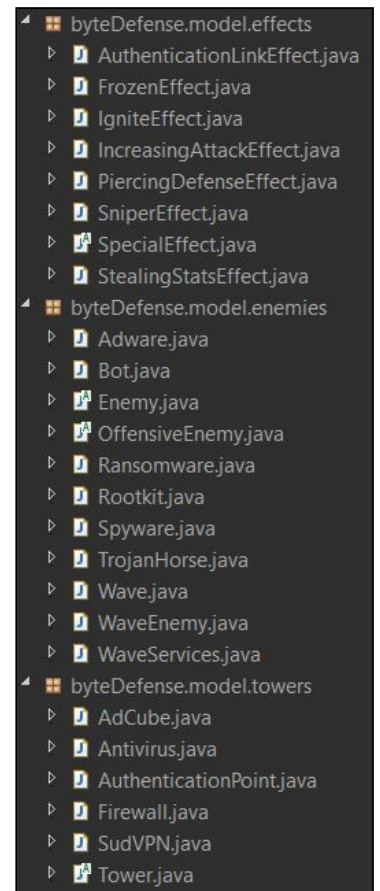


- **GameEnvironment** l’environnement de jeu dans lequel les ennemis (**Enemy**), tourelles de défense (**Tower**) et tirs (**Bullet**) évoluent;
- les ennemis et les tourelles sont des “objets vivants”, ou **LivingObject**, elles forment avec les tirs des objets de jeu, ou **GameObject**. Parallèlement, ces classes ont leurs semblables dans la partie vue (sauf GameEnvironment) pour bien marquer les différences et similitudes des traitements de leur affichage.
- le plateau de jeu (**GameArea**), ainsi que le chemin suivi par les ennemis et construit par le **BFS** sont également stockés dans le GameMaster.



Trois sous-paquets découpent l'architecture côté modèle :

- le sous-paquet *effects* contient la super-classe **SpecialEffect** et ses sous-classes pour les effets de chaque type d'ennemis qui en possèdent (AuthenticationLinkEffect, FrozenEffect, IgniteEffect, IncreasingAttackEffect, PiercingDefenseEffect, SniperEffect et StealingStatsEffect).
- les sous-paquets *enemies* et *towers* possèdent la même structure pour leurs types d'ennemis et de tourelles : ils contiennent la super-classe **Enemy** ou **Tower** et leurs sous-classes pour chaque type d'ennemis (Adware, TrojanHorse et **OffensiveEnemy**, duquel dépendent Bot, Ransomware, Rootkit et Spyware) ou de tourelles (AdCube, Antivirus, AuthenticationPoint, Firewall et SudVPN).
- dans le sous-paquet *enemies*, la classe **Wave**, correspond à une vague d'ennemis à ajouter au jeu (des **WaveEnemy**), dont l'ajout est géré par **WaveServices**.

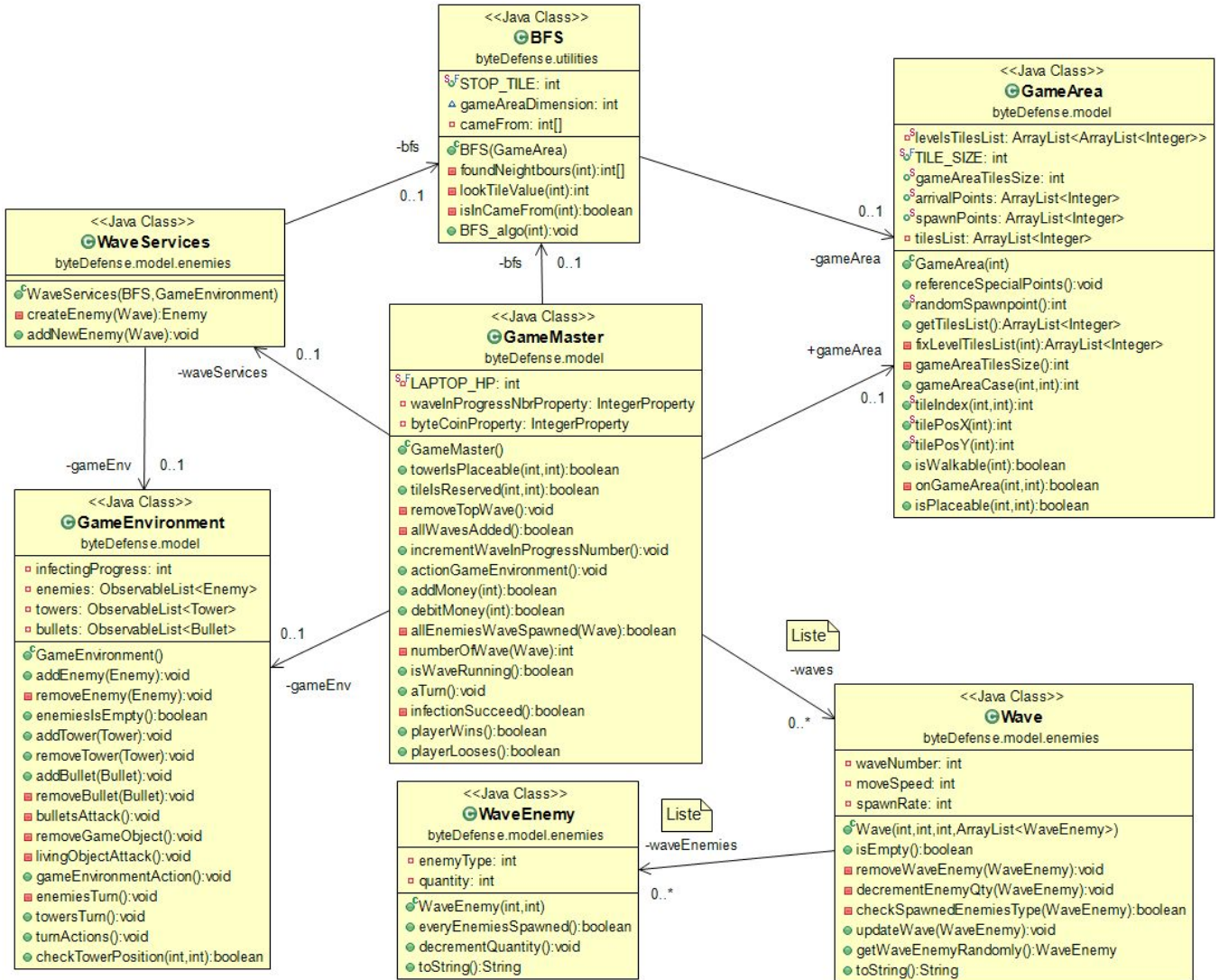


Les classes Enemy et SpecialEffect sont instanciées respectivement à partir des classes **EnemyFactory** et **SpecialEffectFactory** contenues dans le paquet *factory* du modèle.

Toutes les classes utilitaires de lecture de fichiers et de récupération de données (**GameAreaReader** et **WaveReader**), de construction du chemin à suivre par les ennemis (**BFS**) ou encore de gestion des tirs (**ShootUtilities**) se situent dans le sous-paquet *utilities* du modèle.

Détails : diagrammes de classe

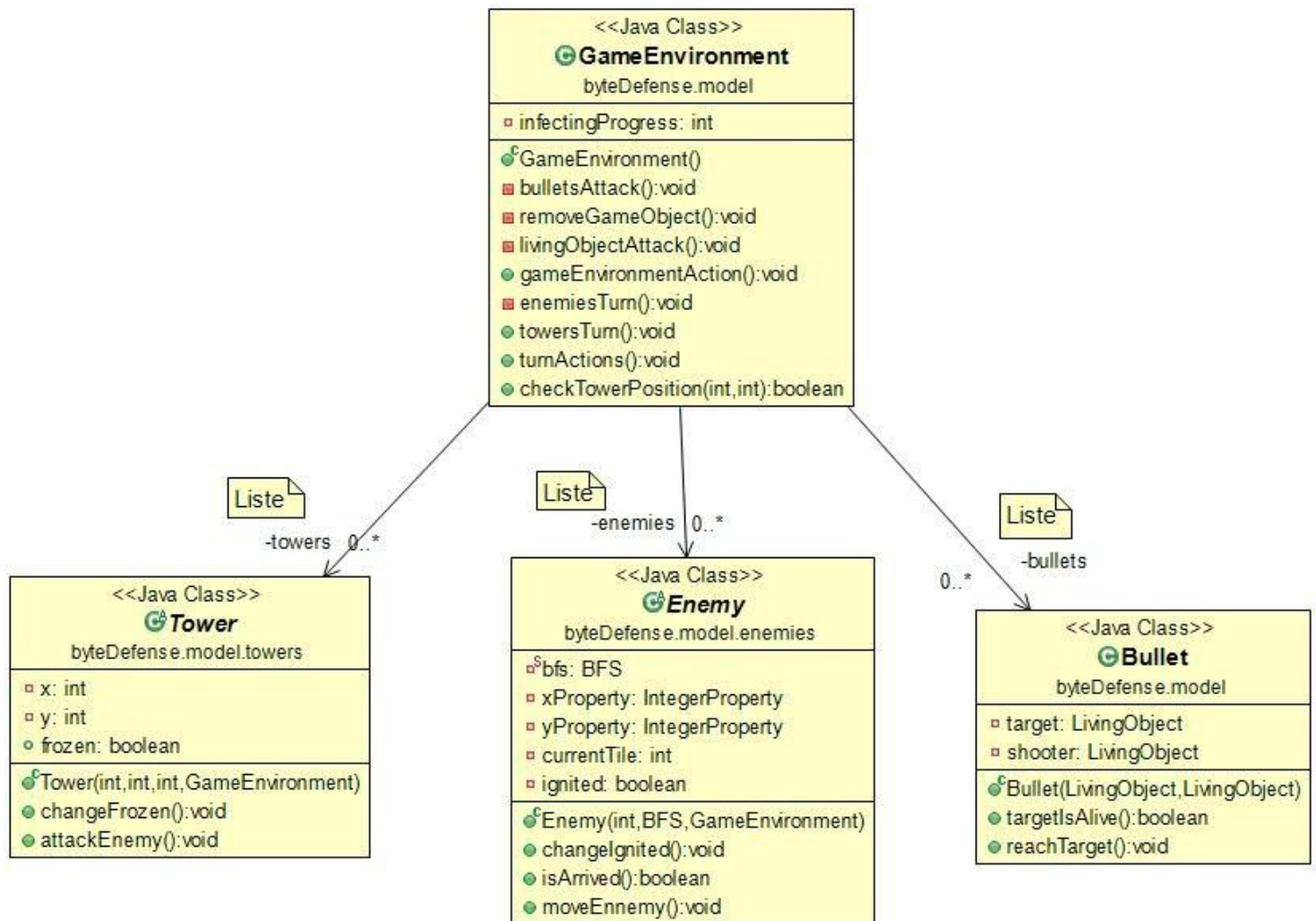
Diagramme 1 : GameMaster, ses dépendances et responsabilités



Commentaires : **GameMaster** gère un ensemble de tests et d'actions (**actions pendant un tour, conditions de victoire et de défaite, gestion de l'argent, de l'environnement de jeu et des vagues d'ennemis**). Cela permet de **regrouper** l'ensemble des données du modèle et de **contrôler** plus facilement leur évolution, sans pour autant les stocker et manipuler individuellement dans le **Controller**.

Les principales classes sont **accessibles** à partir du GameMaster : la liste de **Wave** (vague d'ennemis), contenant chacune une liste de **WaveEnemy** (ennemi de la vague) ajoutés par l'intermédiaire de **WaveServices** à la **GameEnvironment**, aussi attributs du GameMaster; **GameArea** et le **BFS** gèrent respectivement les actions liées au plateau de jeu et au chemin que suivent les ennemis. L'argent du jeu, le **ByteCoin**, est également géré par le GameMaster.

Diagramme 2 : GameEnvironment ou l'environnement de jeu

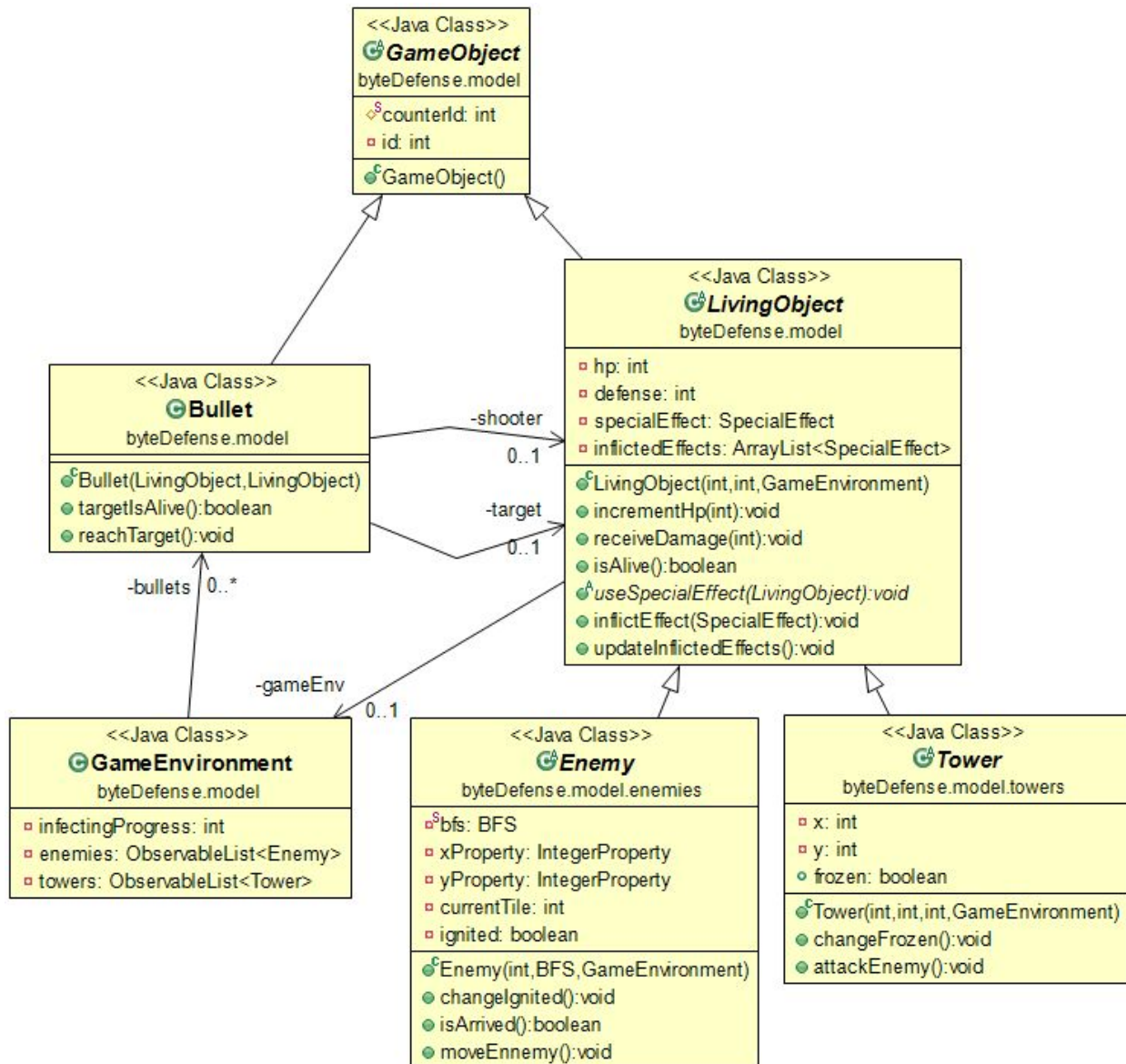


Commentaires : **GameEnvironment** représente l'environnement de jeu dans lequel les tourelles, ennemis et leurs tirs évoluent au fil des tours d'une partie.

Ces trois types d'entités sont stockées dans **trois listes distinctes** (ArrayList), ce qui permet de gérer **indépendamment** et dynamiquement (avec une approche orientée objet) les actions de chacune. Ces actions peuvent être celle d'**ajout**, de **suppression** et de **récupération** d'entités dans les listes, mais également d'attaques de tourelles sur les ennemis et d'ennemis sur les tourelles avec la méthode **livingObjectAttack()**, ou encore des tirs avec la méthode **bulletsAttack()**.

Les actions des ennemis et des tourelles lors d'un tour (correspond relativement au mouvement d'un ennemi) aussi sont gérées séparément car les deux n'effectuent **pas les mêmes actions** durant le tour. L'environnement gère également la progression de l'infection (**infectionProgress**), sous la forme d'un entier incrémenté avec l'attaque des ennemis qui atteignent le point d'arrivée.

Diagramme 3 : GameObject et ses sous-classes

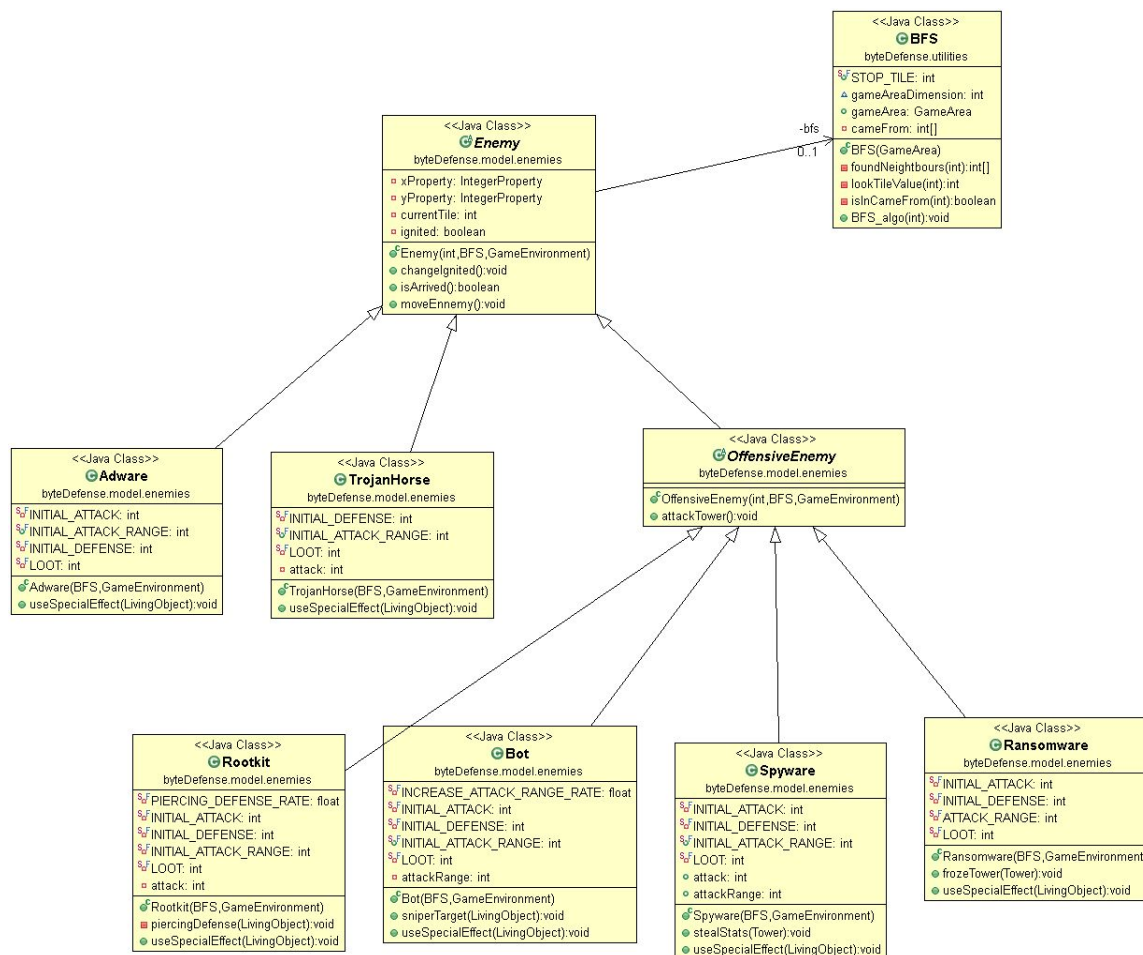


Commentaires : La classe **GameObject** (objet de jeu) est une super-classe parent des classes **Tower** (tourelle), **Enemy** (ennemi) et **Bullet** (tir) : c'est elle qui regroupe leur caractéristique commune, c'est-à-dire celle de pouvoir les **identifier** de façon unique par un identifiant récupérable.

Les tourelles et ennemis possèdent d'autres responsabilités en commun (**la gestion des points de vie, de la défense, de l'effet propre à l'objet ou des effets qui lui sont infligés**) qu'ils héritent de la super-classe abstraite **LivingObject**. Chaque "objet vivant" évolue dans l'environnement de jeu (**GameEnvironment**), qu'il stocke comme attribut, et peut appeler les méthodes implémentées dans les niveaux inférieurs (**utilisation de l'effet, récupération des caractéristiques**).

Bullet contient sa cible (target) est son tireur (shooter), chacun sous forme d'un LivingObject.

Diagramme 4 : Découpage de la classe Enemy



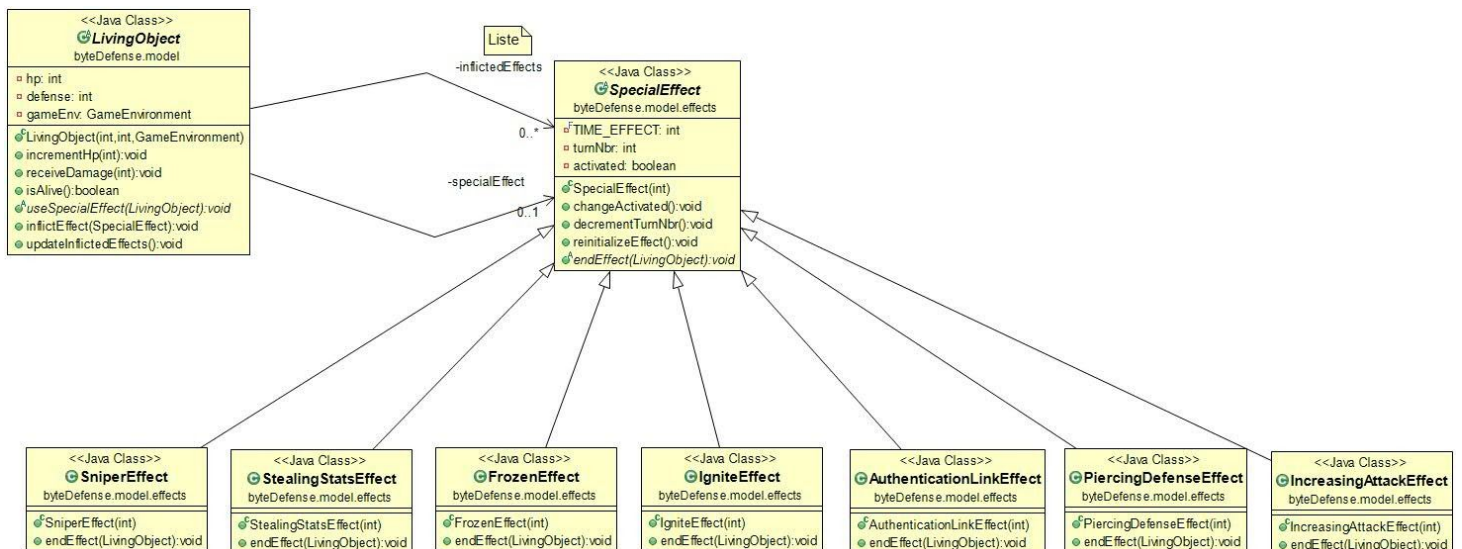
Commentaires : Il existe **6 types** ennemis différents qui peuvent être de **deux sortes** : soit ils tirent sur les tourelles, ils héritent alors de la classe **OffensiveEnemy** qui contient la méthode `attackTower()` (**Rootkit**, **Bot**, **Ransomware** et **Spyware**); soit ils ne tirent pas (**Adware**, **TrojanHorse**).

Tous les ennemis avancent tout le long du chemin afin d'infecter l'ordinateur. Ils possèdent alors la même super-classe abstraite **Enemy** qui leur permet de **se déplacer** (`moveEnemy()`), de vérifier si un ennemi a **atteint le point d'arrivée** (`isArrived()`), et de stocker le **BFS** qui construit le chemin. L'attribut **ignited** indique si l'ennemi a été "enflammé" ou non par une tourelle du type Firewall (pare-feu).

Pour chaque type d'ennemis, les caractéristiques sont les mêmes au départ. Les attributs récupérables de chaque type d'ennemis sont alors **statiques** (`INITIAL_ATTACK`, `INITIAL_DEFENSE`, `INITIAL_RANGE`, `LOOT`). Toutefois, certains ont des attributs **non statiques** (`attack` ou `attackRange` selon les classes). Chacunes des sous-classes implémente la méthode `useSpecialEffect()` dont les actions diffèrent selon l'effet propre au type d'ennemi.

Le découpage au niveau des tourelles est similaire à celui des ennemis, sauf que toutes les tourelles attaquent et qu'elles n'ont pas besoin du BFS puisqu'elles ont une position fixe.

Diagramme 5 : Effets propres et effets infligés aux LivingObject



Commentaires : Un effet spécial (**SpecialEffect**) n'a qu'une responsabilité chronologique et d'état : l'attribut **activated** indique si l'effet est actif; les attributs **TIME_EFFECT** et **turnNbr** correspondent à un nombre de tours, l'un à la durée globale de l'effet, l'autre à la durée qu'il reste avant que l'effet se dissipe. Ces valeurs sont réinitialisées lorsque leur durée est écoulée.

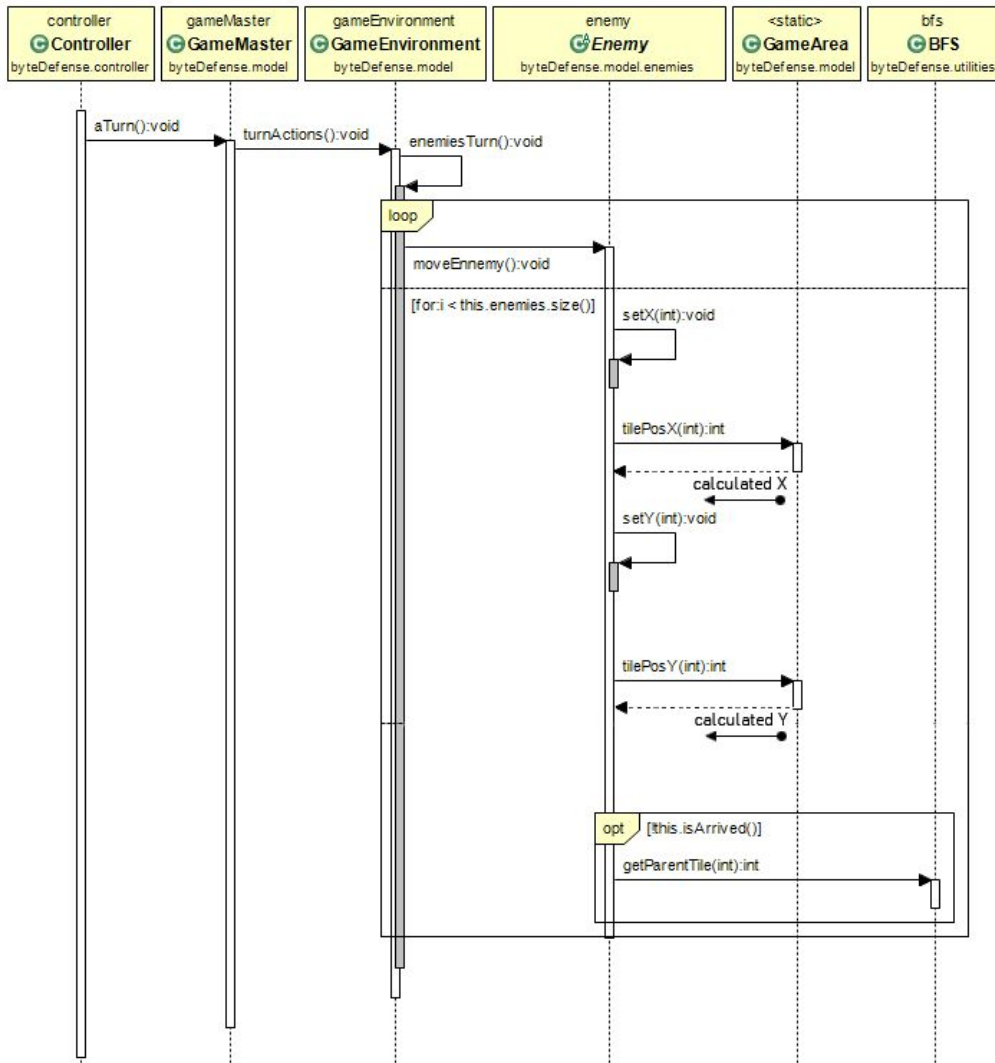
Les effets spéciaux sont **divisés en sous-classes de SpecialEffect pour chacun des types d'ennemis ou de tourelles auxquels ils correspondent** (chaque type de tourelles et ennemis possède un effet propre, sauf Adware, TrojanHorse et AdCube). Ces effets sont **désactivables** dans chacune des sous-classes de SpecialEffect avec la méthode abstraite **endEffect()** qu'ils implémentent.

Un LivingObject possède ou non un effet qu'il lui est propre, mais il a dans tous les cas une liste d'effets infligés (**inflictedEffect**) à lui, mise à jour avec la méthode **updateInflictedEffects()**.

Cette structure permet de **recupérer** pour chaque LivingObject son effet propre, ainsi que de pouvoir gérer en même temps les effets qui lui sont infligés, **dans l'ordre d'actions** des LivingObject dans leur environnement.

Diagrammes de séquence

Déplacement des ennemis :

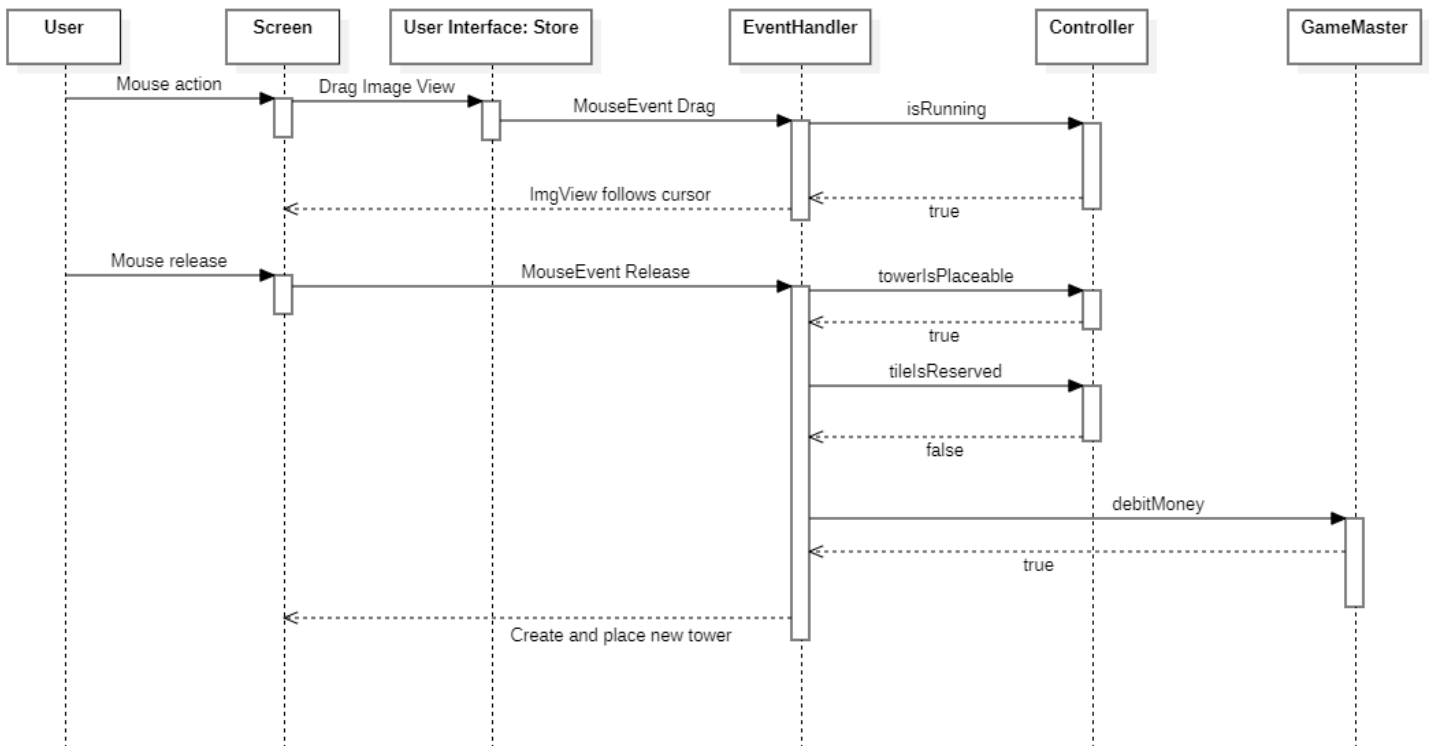


Commentaires : Ce diagramme montre l'enchaînement simple des méthodes pour déplacer les ennemis. Il commence l'appel de la méthode `aTurn()`, depuis le **Controller** qui possède en attribut le **GameMaster**. Dans cette méthode, une autre méthode `enemiesTurn()` permet de faire agir tous les ennemis en parcourant la liste d'ennemis de **GameEnvironment**. A chaque itération de la boucle, la méthode `moveEnemy()` détermine la prochaine position de l'ennemi, grâce à la méthode `getParentTile()` fournie par la classe **BFS**, et la fixe.

La vue est mise à jour grâce à un bind des coordonnées. Cette séquence est répétée à chaque actualisation de la gameloop dans le **Controller**, pendant qu'une vague est en cours et que le jeu n'est pas en pause. Le diagramme est centré sur le déplacement des ennemis, les autres appels de méthodes ne sont pas représentés.

Drag and drop pour l'achat d'une tourelle :

Diagramme de séquence simplifié de l'achat d'une tower



Commentaires : L'utilisateur peut faire un glisser-déposer d'une tourelle à partir du magasin vers un emplacement libre de la carte. L'événement handler défini dans la méthode `initialize()` du **Controller**, se déclenche si seulement la méthode `isRunning()` renvoie **vrai**, c'est-à-dire si le jeu est en play et qu'une partie est en cours.

Au relâchement de la souris **trois vérifications** ont lieu : la première vérifie si une tourelle **peut-être posée** aux coordonnées du curseur, la deuxième si l'**emplacement n'est pas déjà occupé** par une tourelle et la troisième si **le joueur possède assez de bytes coin** pour acheter la tourelle choisie.

La méthode `debitMoney(int amount)` effectuée l'achat seulement si elle est **réalisable** et renvoie **vrai**, sinon elle renvoie **faux** et la **tourelle n'est pas posée**.

Structures de données

La représentation du terrain

```
private ArrayList<Integer> tilesList;
```

Cette structure est basée sur un fichier au format `.txt` qui fournit une carte éditée avec le logiciel TileMap Editor. C'est la classe utilitaire **GameAreaReader** qui va accéder au contenu du fichier pour la remplir de valeur numérique correspondant à une tile (cf. [lecture et construction des listes de tuiles](#)) Voici un exemple ci-dessous :

```
0 => coin de la map
2 => mur horizontal
7 => zone où les tourelles sont posables
8 => texture de l'ordinateur
9 => point d'apparition des ennemis
```

Utiliser une **ArrayList** à ce niveau permet de s'adapter aux différents types de terrain. Ainsi le fichier `tiles.txt` peut contenir plusieurs cartes qui pourront être chargées dans une structure comme ci-dessous.

```
private static ArrayList<ArrayList<Integer>> levelsTilesList;
// Cela permet de changer de carte rapidement et permet de faire évoluer le jeu et
d'ajouter des niveaux facilement.
```

Le chemin fourni par le BFS

```
private int[] cameFrom;
this.cameFrom = new int[this.gameAreaDimension];
```

Ce tableau est un attribut de la classe **BFS**, il est initialisé grâce à la dimension de la Game Area c'est-à-dire de la taille de `tilesList` : cela correspond au **nombre de tuiles** et donc au nombre de sommets sur lequel le BFS sera lancé.

Ce tableau d'entiers est d'abord rempli de 0 qui, au fur et à mesure de l'algorithme, sont remplacés par un indice qui correspond à une tuile de la `tilesList`. Voici ci-dessous un exemple :

```
this.cameFrom[12] = -1; // Si 12 est le point de lancement de l'algo et qu'aucun
voisin n'est trouvé pour ce point, alors le parent de 12 est -1 par défaut.
```



```

this.cameFrom[28] = 30;

/* Indice de la tuile
30 est la tuile parent de 28
Nous pouvons ensuite lire this.cameFrom[30] = 31 par exemple;
On retrouve donc le point de lancement de l'algo.
*/

```

Méthode permettant de récupérer le parent d'une tuile donnée :

```

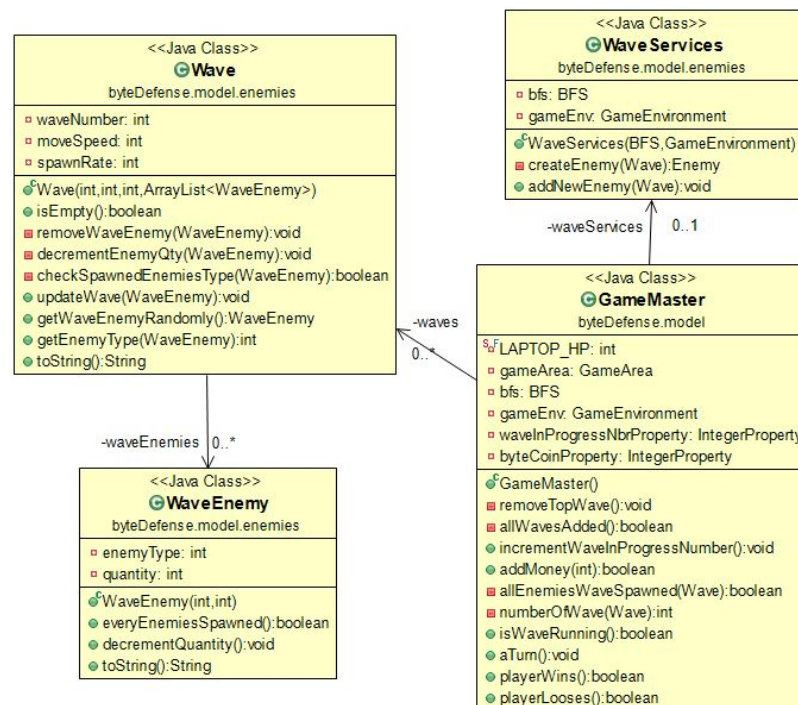
public int getParentTile(int tileIndex) {
    return this.cameFrom[tileIndex];
}

```

Gestion des vagues

Les responsabilités liées à la gestion des vagues et de ses données ont été **segmentés** en plusieurs classes, contenant des **listes** afin de gérer **dynamiquement** les **ajouts et suppressions d'ennemis des vagues** et de **tester facilement si une vague est en cours ou non**.

C'est la classe **Wave** qui représente une vague, d'un numéro précis (waveNumber) d'ennemis à ajouter à l'environnement de jeu. Une vague contient l'**ArrayList waveEnemies** de **WaveEnemy** (afin que de pouvoir récupérer **dynamiquement** les WaveEnemy même si le nombre de types d'ennemis d'une vague à l'autre change.) : un **WaveEnemy** (ou ennemi de la vague) correspond à un type d'ennemi (**enemyType**) et sa quantité (**quantity**) à ajouter pour une vague donnée.



L'ajout d'ennemis d'une vague dans l'environnement est géré par la classe **WaveServices** : elle récupère aléatoirement le type d'un WaveEnemy de la liste waves, puis décrémente sa quantité pour indiquer que l'ennemi a été ajouté (ou pas si le type n'existe pas). Lorsque la quantité pour un WaveEnemy est **nulle**, cela signifie que tous les ennemis de son type ont été ajoutés et le WaveEnemy est alors supprimé de la liste. Lorsque la liste des WaveEnemy est vide, alors tous les ennemis de la vague ont été ajoutés. C'est à ce moment là qu'elle est supprimée de la liste **waves** du **GameMaster**, qui contient l'ensemble des vagues d'une partie (c'est une ArrayList pour les mêmes raisons que la liste waveEnemies : le nombre de vague pouvant évoluer selon le fichier d'informations des vagues, le choix d'une liste permet de gérer dynamiquement les ajouts et suppressions).

Les ennemis qui seront ajoutés peu à peu à l'environnement de jeu sont ceux de la première vague de la liste waves. L'attribut **waveInProgressNbrProperty** indique le numéro de la vague en cours. Une vague est dite "en cours" ([isRunning\(\)](#)) lorsque le **waveInProgressNbrProperty** et le waveNumber de la première vague de la liste waves sont égaux. En effet, lorsque tous les ennemis d'une vague ont été ajoutés à l'environnement de jeu, la première vague de waves est supprimée, **waveInProgressNbrProperty** doit donc être incrémenté ([incrementWaveInProgressNbr\(\)](#)) pour lancer une nouvelle vague.

Une des conditions de victoire dépend de la liste waves du gameMaster : la méthode [playerWins\(\)](#) vérifie (avec la méthode [allWavesAdded\(\)](#)) si tous les ennemis de toutes les vagues ont été ajoutés, c'est-à-dire si **waves** est vide.

Exception

GameMaster.java

```
public boolean isWaveRunning() {
    try {
        return this.getWaveInProgressNbr() == this.numberOfWave(this.getTopWave());
    } catch (IndexOutOfBoundsException e) {
        return false;
    }
}
```

Cette fonction est utilisée pour savoir si **une vague est en cours ou non** (cf. [gestion des vagues](#)). Elle compare le numéro de vague actuelle `this.getWaveInProgressNbr()` à celui de la première vague de la liste des vagues `this.numberOfWave(this.getTopWave())`. En effet, la première vague de la liste étant supprimée à chaque fin de vague, afin d'ajouter les autres, cela crée un décalage qui permet le test. Dans ce cas, la liste est vide lorsque toutes les vagues ont été ajoutées. Cette exception `IndexOutOfBoundsException`, en retournant false, permet d'éviter l'arrêt du programme à la fin de la dernière vague.

EnemyFactory.java

```
public static Enemy getInstance(int enemyType, BFS bfsMap, GameEnvironment
gameEnv) throws Exception {
    Enemy enemy;

    switch (enemyType) {
        case 1:
            enemy = new Rootkit(bfsMap, gameEnv);
            break;
        ...
        default:
            throw new Exception("Aucun ennemi trouvé correspondant au type donné");
    }
    return enemy;
}
```

Cette méthode retourne un objet **Enemy**, à partir d'un type passé en paramètre sous forme d'entier, du **BFS** et du **GameEnvironment**. Elle est utilisée pour créer des ennemis dans le WaveServices (cf. [gestion des vagues](#)). Mais lorsque le type ne correspond pas à une condition du **switch**, alors le cas par défaut se déclenche et affiche un message d'exception **throw new Exception("Aucun ennemi trouvé correspondant au type donné")** : cette exception permet d'éviter l'arrêt du programme qui peut alors continuer son exécution.

Utilisation maîtrisée d'algorithmes intéressants

Le pathfinding grâce au BFS (Breadth-First-Search)

Le **BFS** est l'algorithme que nous avons choisi de mettre en place pour le **pathfinding** qui guide les ennemis le long du chemin : il est mieux adapté au contexte de notre jeu car les ennemis ont un **objectif unique** (une source). Une autre alternative était l'algorithme de Dijkstra, qui aurait été plus intéressant pour des chemins ayant un poids et pouvant être bloqués par un obstacle, ce qui n'est pas le cas dans l'état actuel du projet.

Notre implémentation est réalisée sur une seule classe (BFS). Cela permet, avec une seule liste de tuiles et ses conditions initiales (points de départ et de sortie) fournies par la **GameArea**, de construire un **tableau représentant le chemin le plus court vers le point de lancement sans passer par un graphe** (cf. [le chemin fourni par le BFS](#)). Sur une carte avec plusieurs points de sortie, l'algorithme est lancé plusieurs fois. Pour chaque ennemi le chemin pour atteindre une sortie est défini à son apparition.

Focus sur l'algo :

```
public void BFS_algo(int start);
```

La méthode reçoit en paramètre un entier qui est l'indice de la tuile à partir de laquelle le pathfinding est lancé. L'algorithme crée une `LinkedList<Integer>` qui est une queue FIFO, elle est instanciée localement car elle est différente pour chaque paramètre. On ajoute ensuite ce paramètre à la queue, puis la structure `cameFrom[]` est remplie.

Dans la boucle qui parcourt le terrain, une variable qui stocke la dernière valeur sortante de la queue, puis on cherche la tuile voisine (haute, droite, basse ou gauche), stockée dans un tableau d'entier de taille 4. On parcourt le contenu de ce tableau : si le tableau contient une valeur différente de -1 à une position, cette valeur est l'indice de son voisin. Si les voisins déjà n'ont pas été marqués (ajouté à `cameFrom[]`), ils sont ajoutés à la queue et renseignés dans `cameFrom[]`.

Lecture et construction des listes de tuiles

GameAreaReader est une classe qui gère la récupération des données du fichier `tiles.txt` pour construire un plateau de jeu en une **ArrayList de tuiles** (cf. [la représentation du terrain](#)). Cette classe comporte deux fonctions, une fonction pour **construire une tilesList à partir des données d'une ligne du fichier** et une autre fonction pour construire **les plateaux de jeu des différents niveaux**.

```
private static ArrayList<Integer> createTilesList(String levelLine)
```

La première fonction prend en paramètre une ligne du fichier sous forme de chaîne de caractères. Elle la **découpe à chaque virgule** afin de séparer les données des tuiles entre elles. Ces données sont insérées dans une **ArrayList** pour être retournées dans la deuxième fonction.

```
public static ArrayList<ArrayList<Integer>> generateLevelsTilesList(String sourceFile)
```

La deuxième fonction prend en paramètre la source du fichier, elle est utilisée dans la classe **GameArea**. Sa responsabilité est de **construire une ArrayList contenant les listes de tuiles de chaque niveau**, issues de la première fonction `createTilesList(levelLine)`. Tout d'abord, elle construit une liste de toutes les lignes du fichier, puis entre dans **une boucle qui récupère une à une chaque ligne pour la passer en paramètre à la première fonction**. Une `tilesList` est récupérée et s'ajoute à la principale liste qui est retournée à la fin de la fonction. **Une ArrayList d'ArrayLists** permet de stocker dynamiquement plusieurs terrains récupérables facilement.

Junits

Les tests Junit couvrent la classe appelée **GameArea**. Le **GameAreaTest** vérifie le bon fonctionnement de certaines des méthodes les plus utilisées dans la classe.

Quatre méthodes sont mises en test :

1. `isWalkableTest();`

```
public boolean isWalkable(int index);
```

Cette fonction est assez simple dans son fonctionnement, mais reste néanmoins essentielle (pour le BFS notamment). Son test consiste à fournir les valeurs des **cas-limites** et d'autres cas d'utilisation, pour vérifier si la méthode retourne la bonne valeur booléenne. Pour ce faire, on utilise la méthode `assertTrue()`.

2. `tileIndexTest();`

```
public static int tileIndex(int x, int y);
```

La fonction `tileIndex()` permet de faire la **correspondance entre le terrain**, représenté dans une `ArrayList<Integer>`, **et la dimension 2D des positions** (coordonnées XY) des **GameObjects**. Les tests consistent seulement à tester si la fonction retourne les bonnes valeurs pour des paramètres donnés.

3 et 4. `tilePosX();` et `tilePosY();`

```
public static int tilePosX(int indTilePos); // Indice de la Tile  
public static int tilePosY(int indTilePos);
```

Ces deux fonctions servent à retrouver les coordonnées X et Y d'un objet. Elles se complètent avec la méthode `TileIndex`, pour retrouver les positions ou l'indice dans sur la liste des tuiles. Avec la méthode `assertEquals()`, on vérifie les valeurs retournées pour des paramètres précis.

Le code permet également de vérifier les invariants de classes pour minimiser les erreurs.

Documents pour Gestion de projet

Document utilisateur

Screenplay

Le jeu se déroule dans un **réseau informatique**, milieu dans lequel se propagent toutes sortes de **logiciels malveillants**. Mais heureusement des **moyens de protection** existent pour les traquer et garder sain l'**ordinateur**.

Les logiciels malveillants ne se contentent pas que d'infecter la machine, ils n'hésiteront pas à user de leurs **particularités** pour endommager et détruire les systèmes de sécurité du réseau.

Utilisez astucieusement les capacités de vos **protecteurs** (les tourelles), pour venir à bout de toutes les menaces, et vaincre le boss de fin, le **cheval de Troie**.

Tableau des relations

Ennemis						
Tourelles						
						
						
						
						
						







Tout le monde attaque tout le monde

Seulement les tourelles attaquent les ennemis





L'ordinateur qui est infecté par les menaces, n'est pas une entité à part entière, il représente seulement le concept de point arrivée et est associé à la progression de l'infection.

Les ennemis

Les ennemis sont des virus informatiques : par défaut ils ont 50 points de vie. Il y a deux types d'ennemis : **ceux qui ciblent les tourelles** et **ceux qui ne tirent pas**.






 <p>Adware Gain : 1</p> <p>Points d'attaque : 10 Points de défense : 5 Portée d'attaque : 0 Caractéristique : Aucune</p>	 <p>Rootkit Gain : 4</p> <p>Points d'attaque : 15 Points de défense : 5 Portée d'attaque : 2 Caractéristique : Perce la défense de la tourelle (points d'attaque +10 % de sa défense)</p>	 <p>Bot Gain : 6</p> <p>Points d'attaque : 15 Points de défense : 15 Portée d'attaque : 2 Caractéristique : Augmente sa portée d'attaque (x2)</p>
 <p>Ransomware Gain : 8</p> <p>Points d'attaque : 10 Points de défense : 30 Portée d'attaque : 3 Caractéristique : Gèle les tourelles (elle ne peuvent plus attaquer pendant 2 tours)</p>	 <p>Spyware Gain : 10</p> <p>Points d'attaque : 15 Points de défense : 25 Portée d'attaque : 2 Caractéristique : Vole les points d'attaque de sa cible s'ils sont supérieurs aux siens</p>	 <p>Trojan Horse Gain : 60</p> <p>Points d'attaque : 0 Points de défense : 800 Portée d'attaque : 0 Caractéristique : Infecte l'ordinateur en une fois</p>

Vague des ennemis

Ennemis						
Vagues						
1	5	5	0	0	0	0
2	5	3	2	0	0	0
3	5	3	2	3	0	0
4	4	3	2	3	2	0
5	5	4	5	5	4	0
6	7	6	7	7	6	0
7	0	0	0	0	0	1


Les tourelles

Les tourelles sont des protections pour l'ordinateur : par défaut elles ont 100 points de vie et tirent toutes sur les ennemis.

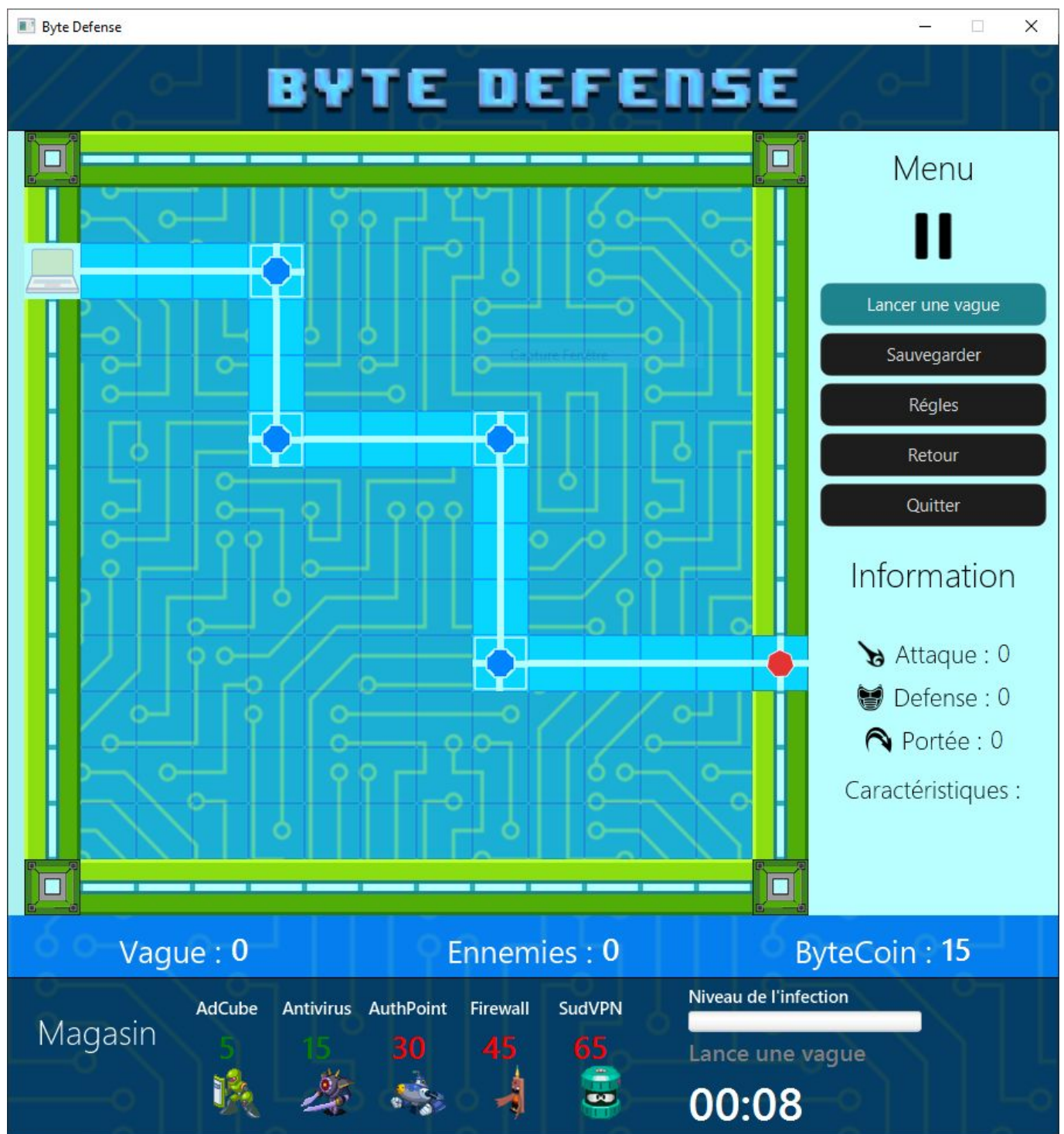
 <p>AdCube Coût: 5</p> <p>Points d'attaque : 8 Points de défense : 10 Portée d'attaque : 2 Coût : 5 Caractéristique : Aucune</p>	 <p>Antivirus Coût: 15</p> <p>Points d'attaque : 15 Points de défense : 0 Portée d'attaque : 3 Coût : 15 Caractéristique : Double tirs (tire sur deux ennemis en même temps)</p>	 <p>Authentication Point Coût: 30</p> <p>Points d'attaque : 10 Points de défense : 30 Portée d'attaque : 4 Coût : 30 Caractéristique : Ajoute les points d'attaque de l'ennemi à ses points d'attaque</p>
 <p>Firewall Coût: 45</p> <p>Points d'attaque : 25 Points de défense : 15 Portée d'attaque : 3 Coût : 45 Caractéristique : Enflamme les ennemis (3 points d'attaque par tours)</p>	 <p>SudVPN Coût: 65</p> <p>Points d'attaque : 15 Points de défense : 30 Portée d'attaque : 2 Coût : 65 Caractéristique : Inflige de gros dégâts instantanés (x3,50)</p>	

L'ordinateur

L'ordinateur se trouve à la fin du réseau informatique. Il perd de la vie au fur et à mesure que les ennemis l'infectent. Si le cheval de Troie infecte l'ordinateur, alors la partie est perdue.

	<p>Ordinateur Points de vie : 645</p>
---	--

L'interface utilisateur ou Game scene



Magasin

Le **magasin** est l'interface pour acheter des tourelles. Le **coût des tourelles est affiché** : il devient **vert** lorsque le joueur a assez d'argent pour l'acheter et **rouge** à l'inverse. Pour acheter une tourelle, il faut la **glisser sur une case de la zone des tourelles**. Lorsque l'on passe la souris sur une tourelle du magasin, ses informations de caractéristique sont affichées dans la **barre à droite**.



Information

SUDVPN

 Attaque : 15

 Defense : 30

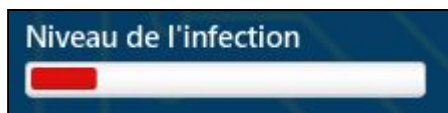
 Portée : 2

Caractéristiques :

Déplacement et vente

Entre deux vagues, le joueur peut **déplacer** ou **vendre** ces tourelles. Le coût pour déplacer une tourelle est de **50% du prix de base de la tourelle**. Pour la vente, le joueur gagne **30% du prix de base**. Pour déplacer une tourelle, il faut la **glisser** sur le terrain et pour la vendre, il faut **double cliquer** sur la tourelle.

Barre d'infection



La **barre d'infection** est le niveau d'infection de l'ordinateur, elle augmente au fur et à mesure que les ennemis infecte l'ordinateur. Le niveau d'infection ne doit pas atteindre 645 de points d'attaque (soit les points de vie de l'ordinateur).

Messages

Les **messages** sont affichés en dessous de la barre d'infection : il indique si le jeu est en pause, qu'il reprend, si une vague est en cours, s'il faut en lancer une nouvelle, ou alors si le joueur a gagné ou perdu la partie.



Minuteur

Le **minuteur** évolue pendant la partie, il commence directement lorsque le programme est lancé et s'arrête lorsque l'on met le jeu en pause ou à la fin du jeu. Il ne s'interrompt pas entre deux vagues.



Etat de la partie

Au-dessus du magasin est affiché le numéro de la vague, le nombre d'ennemis sur le terrain et les bytes coin que possède le joueur. Le ByteCoin est l'argent du jeu, il est fixé par défaut à 15.

Vague : 0 Ennemies : 0 ByteCoin : 15

Boutons



Play & Pause

Le bouton **play & pause** met en pause le jeu ou le relance. En pause, le joueur ne pourra plus acheter, déplacer ni vendre de tourelles. Lorsque l'on appuie sur le bouton pause, le bouton play s'affiche et inversement.

Lancer une vague

Le bouton **lancer une vague**, comme le nom l'indique, sert à lancer une vague. Une vague ne peut être lancée pendant qu'une autre vague est en cours ou lorsque le jeu est en pause.

Quitter

Le bouton **quitter** ferme le programme.

Scoring

Pour gagner une partie, le joueur doit réussir à tuer tous les virus sans que la barre d'infection soit complète ou que le cheval de Troie infecte l'ordinateur. Cela représente un total de 7 vagues et 104 ennemis. Le score correspond à l'argent qui lui reste, le niveau d'infection et le temps qu'il a passé.

À toi de jouer !

