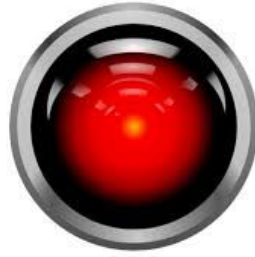


Documentation HAL 9000 Defender



Réalisation:

Cédric Fontaine
Illan Makhloufi
William Rousseau

C POO		IHM		Projet Tuteuré		Gestion de Projet	
diagrammes d'architecture	9	ergonomie et contenus	15	soutenance	10	document utilisateur	8
diagrammes de classes	9	programmation événementielle et J avaFX	35	niveau et qualité du produit fini	15	Trello	6
diagrammes de séquence	9			notes de sprints	15	Git	6
POO	24						
Structures de données	6						
gestion des erreurs	5,5						
tests	5,5						
algos	17						
ampleur et qualités du code	25						
	110		50		40		20

SOMMAIRE

Présentation	4
Utilisation	6
Architecture	8
Modèle	8
Entités	8
Virus	8
Antivirus	9
Projectile	9
Environnement	10
Graphe	10
Location	10
SpriteSheet	10
Tile	11
TileMap	11
Vecteur	11
Environnement	11
Game	11
Diagramme de séquence	13
Classes	16
Entity	16
Tower	18
Virus	20
Projectile	22
Game	23
MapController	25
Interfaces	27
Algorithmes	28
BFS	28
Move	30

Présentation

HAL 9000 Defender est un jeu au titre inspiré par l'IA du même nom dans le film 2001: A Space Odyssey. Ici, HAL 9000 est le CPU de l'ordinateur d'un ado peu précautionneux, et est menacé par des attaques et le joueur a la responsabilité de le défendre avant qu'il ne soit corrompu par des virus. Ceux-ci se déplacent en vagues sur la carte mère de l'ordinateur et cherchent à atteindre le CPU. Pour le défendre, le joueur a accès à un panel d'antivirus et autres logiciels de maintenance. Tous ont leurs avantages et inconvénients, et permettent de mettre en place différentes stratégies.

Le jeu est appréciable de par son interface et une ambiance rétro rappelant les bornes d'arcades aux jeux rythmés par des chip music 8bits. Le rythme du jeu évolue à mesure que l'on progresse et demande de la concentration et de la réactivité. En effet, chaque virus a un rôle clé pouvant mettre à mal la partie des joueurs les moins attentifs.

Les virus sont au nombre de 4.

Adware

En soutien, les adwares désactivent les défenses environnantes en les inondant de messages intempestifs (ads, ou popups). Leur point faible est leur courte portée, mais à plusieurs, ils peuvent percer les défenses et permettre aux autres virus de continuer leur progression pendant que le joueur doit supprimer les spams pour réactiver les antivirus.

Ransomware

Plus sournois que l'adware, les ransomwares ont pour fonction de désactiver la totalité du système en trouvant une backdoor sur leur chemin. Celle-ci peut se trouver à la toute fin de la route, mais également au tout début, faisant du ransomware une menace imprévisible. Si celle-ci atteint la backdoor, elle bloquera le système, forçant le joueur à dépenser des points pour régler la rançon.

Zombie et Trojan

Principaux attaquants, les zombies pavent la route au reste des virus en détruisant les pare-feux et infligeant des dégâts au CPU.

Le trojan fonctionne de la même manière, si ce n'est qu'à sa suppression il fait apparaître des zombies supplémentaires sur le terrain.

Ces deux unités provoquent la surchauffe du CPU, et donc la fin de la partie.

Pour se défendre, le joueur possède 4 anti virus et autant de bonus (un seul bonus peut être actif à la fois, à l'exception du pare-feu).

Afast

Défense de base, cet antivirus tire des projectiles à trajectoire statique dans la direction des virus détectés. Ceux-ci s'activent une fois à destination, endommageant les virus se trouvant aux alentours. Le placer à distance permet de le protéger des Adwares, mais le rend moins précis. Pratique pour tenir des embuscades.

GoodwareBytes

En soutien, il posera sur le chemin le plus proche un piège ralentissant le virus qui passera dessus.

IVG

Ce puissant antivirus génère autour de lui un champ de force de faible portée neutralisant les virus présents dans son périmètre, pendant un certain temps. Vulnérable aux adwares et ransomwares pendant son temps de recharge.

KilobitDefender

Ce dernier antivirus tire des projectiles à trajectoire dynamique. Les virus détectés sont assurément atteints, le rendant efficace dans de nombreuses situations.

Firewall

Le pare-feu est un bonus qui bloque la route des virus, les empêchant de progresser. Défense éphémère, il s'affaiblit avec le temps et peut être détruit par les virus hostiles.

AdBlock

Ce bonus empêche l'apparition des popups, rendant inefficace les adwares et ransomwares pendant un court instant.

SudVPN

SudVPN relocalise les virus à leur point d'apparition. Efficace lorsqu'une vague est trop proche du CPU.

CKleaner

Ce bonus de la seconde chance purge le système en supprimant la totalité des virus présents.

Utilisation

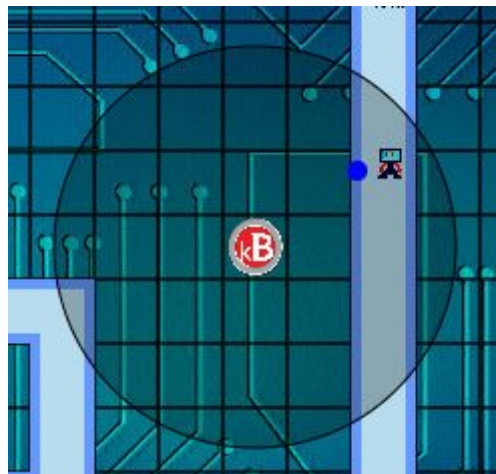
Pour commencer le jeu, il suffit de cliquer sur le bouton prévu à cet effet sur l'écran d'accueil.



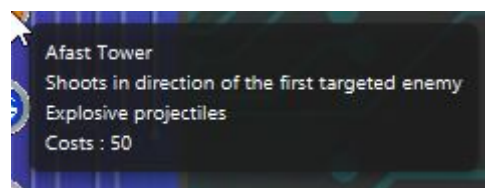
La partie se met en place en donnant des points de départ permettant d'acheter les défenses sur le panel de droite.



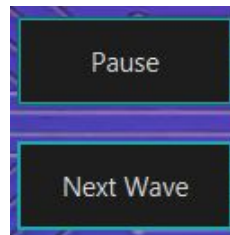
Pour placer des antivirus, il suffit d'en sélectionner un dans le registre, puis de cliquer sur la carte où se déroule le jeu. Il n'est pas possible de les poser sur le circuit des virus, à l'exception du pare-feu.



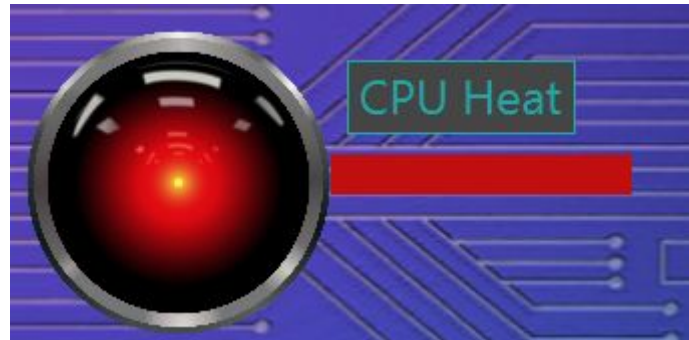
Des informations complémentaires sont disponibles en passant le curseur sur la défense souhaitée.



Les vagues sont déclenchées depuis le menu en bas de l'interface. De même, il est possible de mettre la partie en pause à tout moment, et de la relancer.



L'état du CPU y est indiquée dans le coin gauche, et le numéro de la vague en cours, à droite.



Au lancement du programme, une musique est lancée. Il est possible de changer les paramètres audio depuis la barre de menu en haut de la fenêtre, en choisissant une autre musique, ou en l'arrêtant.

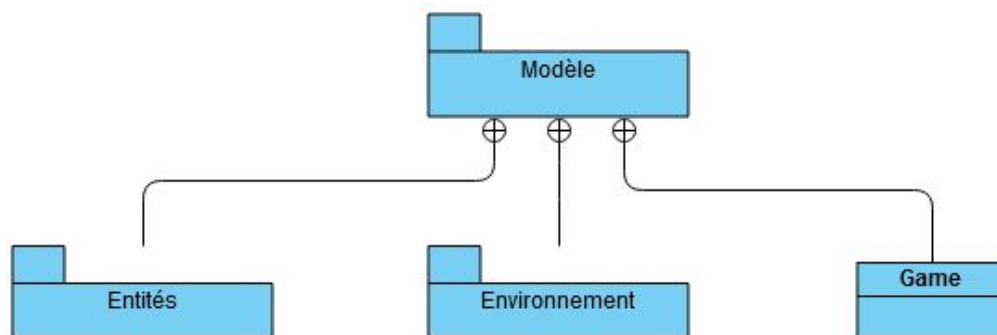


Architecture

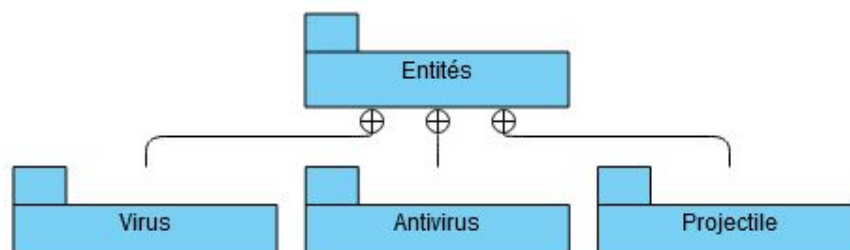
Le projet est basé sur une structure **MVC**.

Modèle

Le modèle est l'ensemble des données du jeu, donc des classes d'entités et d'algorithmes.



Il est composé de 2 sous-dossiers qui contiennent les classes qui leur correspondent.



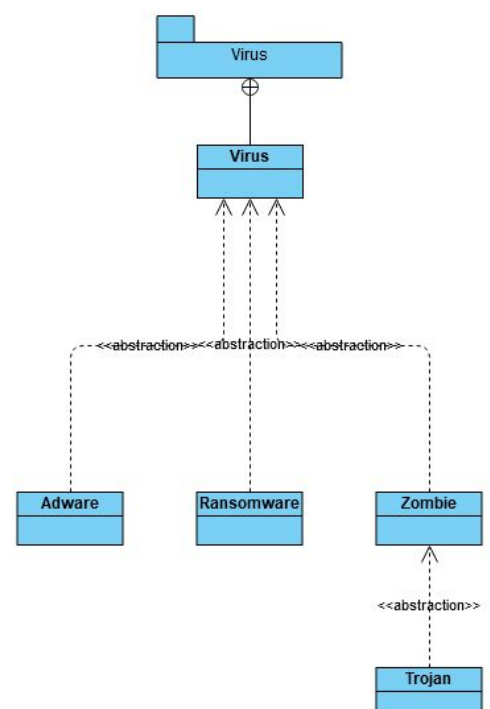
Entités, d'abord, comprend l'ensemble des acteurs présents en jeu, répartis en sous-catégories.

Entités

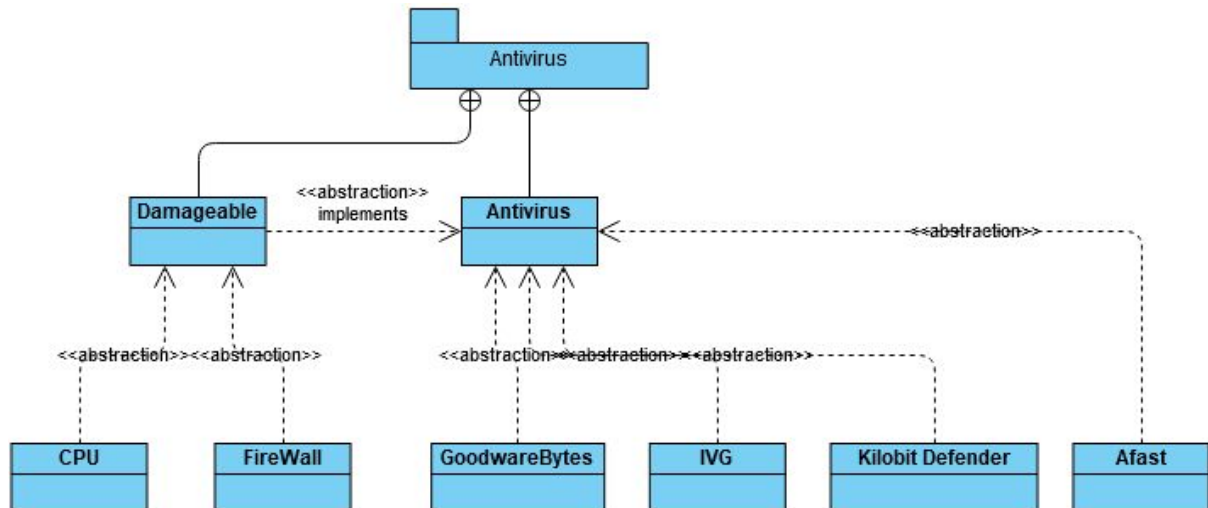
Virus

La classe mère **Virus** contient les différents corps de méthodes communes à toutes les entités de type Virus. Ces méthodes peuvent être écrasées selon le fonctionnement de certains virus. Par exemple, le **Ransomware** et l'**Adware** arrêtent de bouger dans certaines situations. Leur méthode de déplacement appelle donc la méthode de la classe mère uniquement si toutes les conditions le permettent.

La classe **Trojan** est une extension de **Zombie** dans la mesure où elle agit de la même manière, à l'exception de son "coup spécial" qui fait apparaître 2 entités à sa destruction. Cela évite la duplication du code (l'attaque du **CPU** et des **pare-feux**), puisque seule la classe **Zombie** (in extenso **Trojan**) inflige des dégâts, et **Trojan** écrase la procédure de "mort" de **Virus**



Antivirus



Le CPU et le Firewall sont 2 unités à points de vie, qu'il est possible de détruire. Elles implémentent donc une interface qui leur permet de prendre des dégâts. Les 4 antivirus utilisent le même algorithme de détection mais possèdent chacun leur méthode de tir, par exemple GoodwareBytes qui cherche dans son périmètre la tuile de type chemin la plus proche. Ou encore IVG, qui ne tire pas mais détruit les entités dans sa ligne de vue.

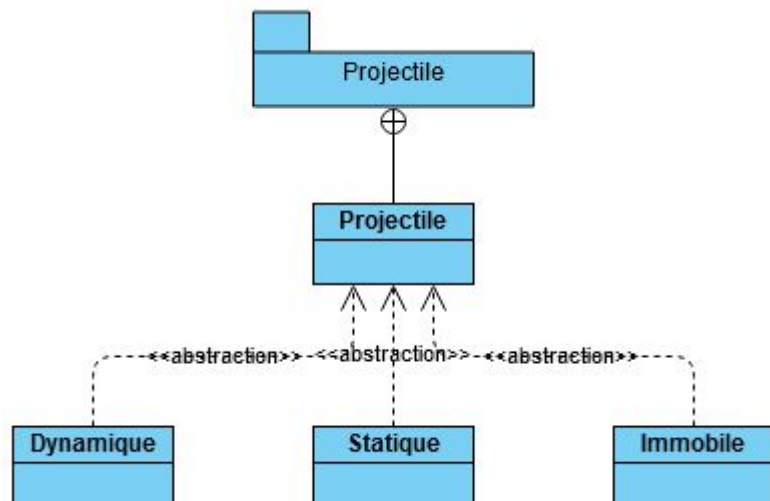
Projectile

La grande différence entre les projectiles réside dans leur méthode de déplacement.

Le projectile dynamique reste bloqué sur une cible et la suit jusqu'à la collision avec celle-ci.

Le projectile statique explose en atteignant des coordonnées fixées lors d'une détection de virus.

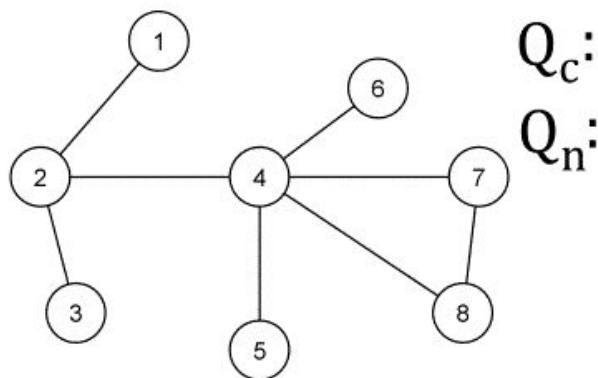
Le projectile immobile ne bouge pas et s'active en présence de cibles.



Environnement

Graphe

La classe graphe permet l'implémentations d'algorithme(s) de recherches (donc de trouver un chemin entre 2 nœuds d'un graphe). Elle utilise un algorithme de recherche en largeur reliant n'importe quelle position à la position de fin.



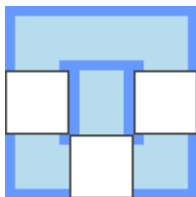
Un gif montrant l'exécution de l'algorithme BFS

Location

Cette classe générique encapsule un simple couple de valeurs observables (x, y) ou (colonne, ligne) permettant de situer des éléments dans un tableau en deux dimensions. Elle représente donc la position des entités et tuiles du jeu. Toute modification de ces valeurs entraîne donc une mise à jour aux objets correspondants dans la vue.

SpriteSheet

Cette classe génère une banque d'images à partir d'une feuille de sprites, en leur assignant un ID unique. Cela permet de charger les images une seule fois et de récupérer rapidement les sprites souhaités. Pour ce faire, on charge une feuille de sprite regroupant plusieurs images. Ici, des cases de taille 32x32 dans une grille de 96x96.



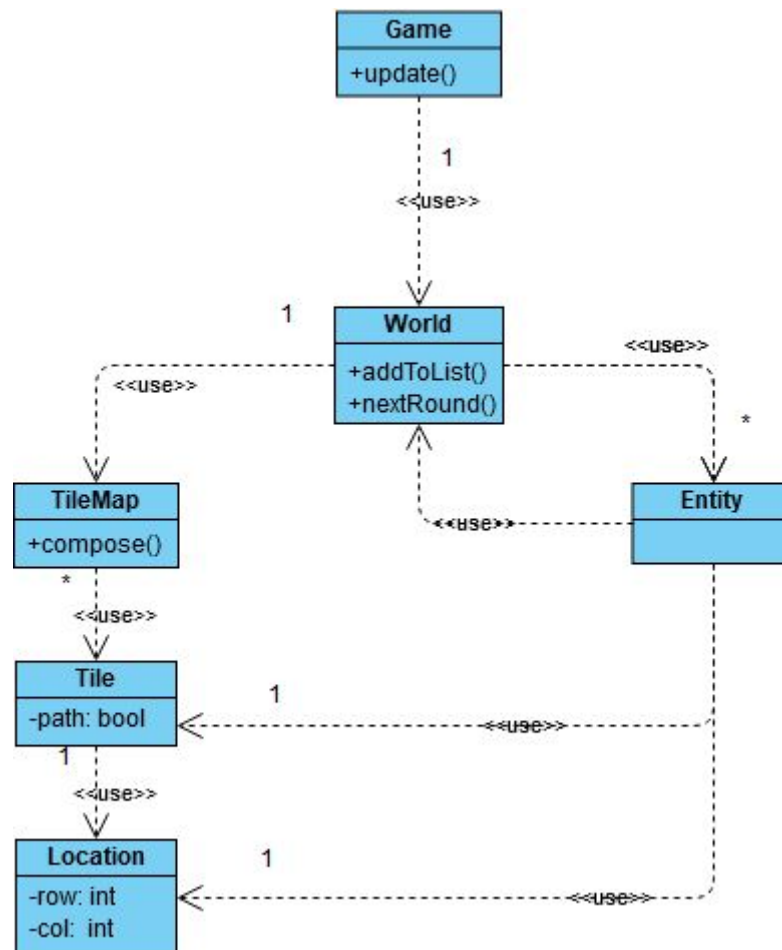
On part du principe que chaque élément a pour ID un entier s'incrémentant à partir de 0. La première case a donc l'indice 0, la seconde 1, etc. On détermine les dimensions de la feuille grâce à un accesseur ainsi que la taille d'une case, donc on peut déterminer le nombre de colonnes et lignes dans la grille.

En l'occurrence ici, on a une grille de 3x3 puisque $96 / 32 = 3$, donc 9 itérations (9 cases). On sait que pour accéder à l'image 0, on utilise les indice $x = 0$ et $y = 0$ (tableau à deux dimensions), et on la délimite grâce à sa taille.

On obtient donc les pixels entre (0, 0), et (32, 32), que l'on peut copier dans une nouvelle image, récupérable depuis une map grâce à son ID.

Les ID sont générées sur le logiciel Tiled, qui permet de faciliter l'agencement des sprites sur la feuille et de s'assurer que le sprite 0 (par exemple) corresponde bien à une tuile de type "coin supérieur gauche", à la lecture du fichier contenant les informations de la carte.

Les interactions entre les classes d'entités et celles d'environnement peuvent se voir ainsi.



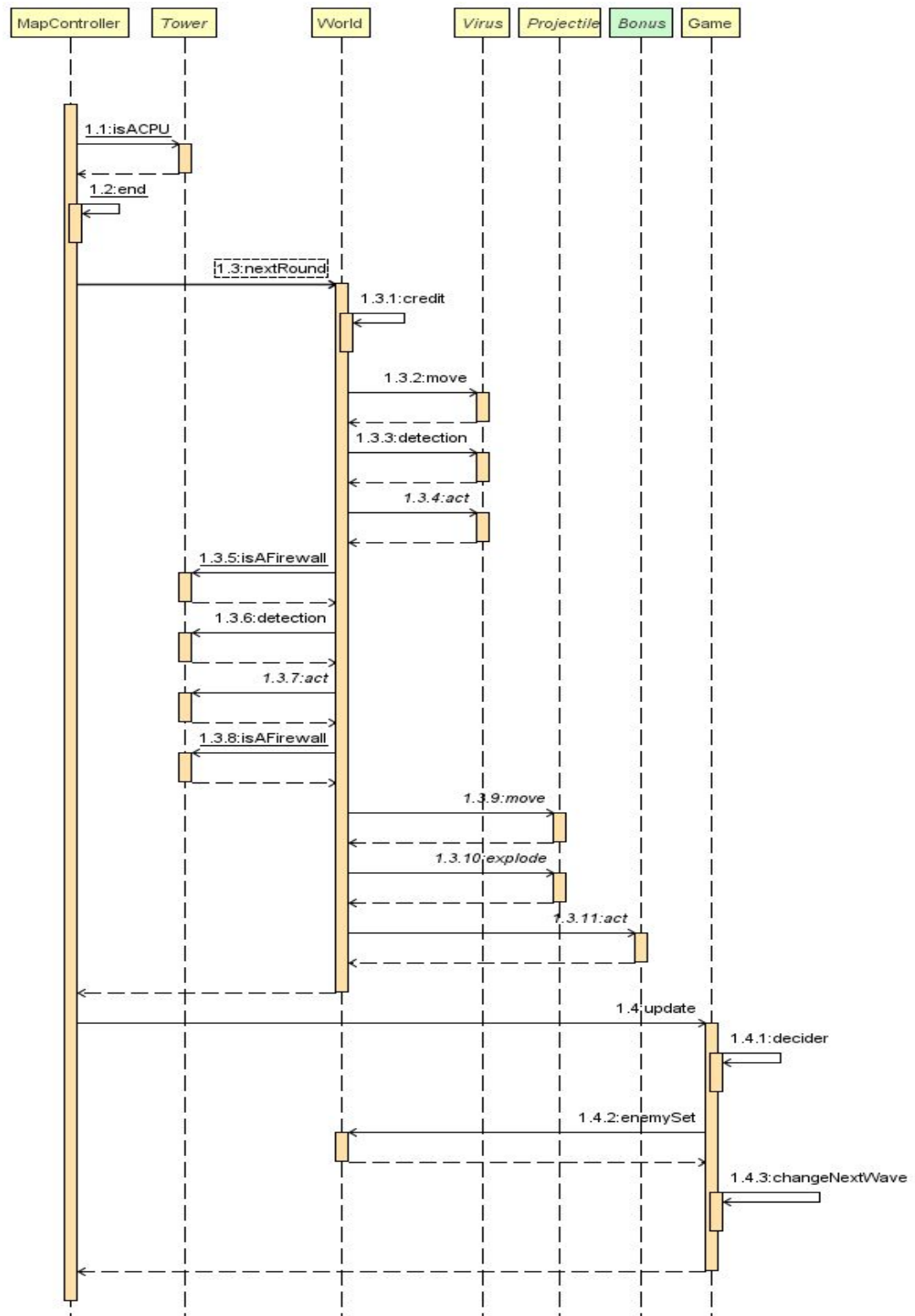
La **Game** possède un **environnement** composé d'une liste de **tuiles** et possédant des **entités**.

Les entités peuvent référencer la tuile sur laquelle ils se trouvent (notamment les virus qui peuvent ainsi accéder à la prochaine tuile pour déterminer leur direction).

Diagramme de séquence

La procédure **tick()** est implantée dans la classe MapController, et constitue la plaque tournante du jeu puisqu'elle fait appel à deux procédures charnières. Son contenu est encapsulé dans une condition qui consiste à vérifier l'état du jeu, s'il est en pause ou non.

Si celui-ci se trouve ne pas l'être, alors un processus de vérification de l'état du CPU se déclenche pour mettre fin au jeu au cas où celui-ci se verrait détruit.



Ensuite, elle fait appel à la première procédure importante, **nextRound()**, situé dans la classe World, qui a pour objectif principal de déclencher les différentes actions réalisables par chacune des entités actives et présentes dans le jeu:

Virus:

move(): Commune à tout type de virus, elle vérifie s'il n'y a pas d'obstacle (Firewall) dans la range et si tel est le cas, elle actualise la position de l'entité à l'aide de son vecteur.

detection(): Cherche parmi les tours présentent en jeu celles :

1. Qui sont dans la range et les ajoute dans une liste de targets
2. Qui ne sont plus dans la range et les supprime de cette même liste.

act(): Spécifique à chaque type de virus, selon les conditions, elle déclenche un effet qui leur est propre.

Tower:

Concernant ce type d'entité, le traitement est spécifique en raison de la classe Firewall qui se trouve être une exception puisque c'est un Bonus dont le fonctionnement est en tout point similaire à une Tower si ce n'est qu'il ne tire pas.

Dans le cas où il ne s'agit pas d'un Firewall. **nextRound()** appelle:

detection(): Actualise la liste de virus dans la range, en ajoutant les nouveaux et en supprimant ceux qui n'y sont plus.

act(): Spécifique à chaque type de tower, selon les conditions, elle déclenche un effet qui leur est propre.

Dans le cas contraire:

isAlive(): méthode applicable sur les classes qui implements l'interface Damageable, et qui indique si l'entité est encore en vie. Si ce n'est pas le cas, **nextRound()** la supprime avec **remove()**.

Projectiles:

move(): Spécifique à chaque type projectiles, selon les conditions, elle actualise la position de l'entité à l'aide de son vecteur (excepté pour Motionless).

Dans le cas où le projectile a atteint sa cible/destination:

explode(): Spécifique à chaque type projectiles, elle baisse les points de vie de(s) entité(s) visée(s) en fonction des dégâts qu'il implique.

nextRound() le supprime ensuite avec **remove()**.

Bonus:

act(): Spécifique à chaque type de Bonus, déclenche son effet.

Enfin, c'est au tour de la procédure **update()** d'être appelé. Présente dans la classe Game, son contenu est également encapsulé dans une condition de sorte à n'être accessible qu'en cas de nouvelle vague. Son but réside donc dans l'apparition d'ennemis et la gestion des rounds.

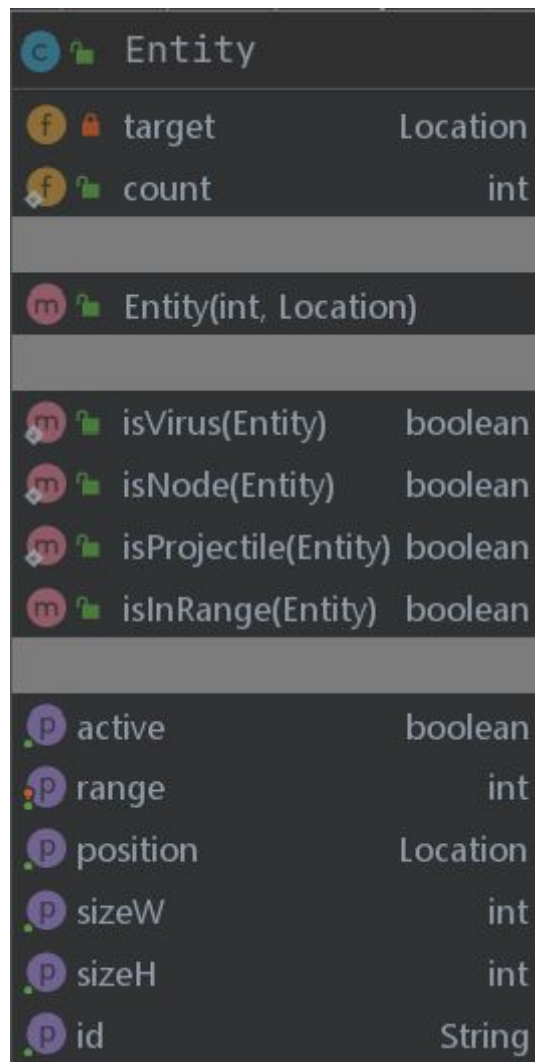
Dans un premier temps, elle fait apparaître chaque ennemi avec un intervalle de temps. Pour cela, elle fait appel à **enemySet()**, de la classe World, avec pour paramètre, le résultat renvoyé par la méthode **decider()**, qui génère aléatoirement un entier, celui-ci indiquant à **enemySet()** le type de Virus à créer.















Ensuite, dans le cas où tous les virus sont morts, elle configure la vague suivante, en mettant à jours son numéro ainsi que sa difficulté donc son nombre d'ennemis.

La procédure **tick()** étant appelée 30 fois par secondes, cela permet une actualisation du jeu constante.

Classes

Entity



	Entity	
	target	Location
	count	int
	Entity(int, Location)	
	isVirus(Entity)	boolean
	isNode(Entity)	boolean
	isProjectile(Entity)	boolean
	isInRange(Entity)	boolean
	active	boolean
	range	int
	position	Location
	sizeW	int
	sizeH	int
	id	String

La classe Entity est une des classes centrale du projet. Cette classe nous sert à gérer les éléments actifs de notre jeux. Ainsi, cette classe est la classe mère de Virus, Tower et Projectile.

Elle contient les attributs suivants :

- **private int range** qui est un entier déterminant la portée d'action de l'entité
- **protected Location position** qui définit un couple de valeurs (deux entiers) et qui, ici, permettent de définir la position de l'entité dans son environnement
- **private Location target** qui définit un couple de valeurs qui permettent de définir la position de leur cible (si existante)
- **private final boolean isActive** est un booléen qui définit le statut de l'entité, soit elle est active et peut donc agir, soit elle est inactive

- **private final int sizeW et sizeH** permettent de définir les dimensions de l'entité
- **private final String id** représente l'identifiant de l'entité, permettant de l'identifier au besoin
- **public static int count** est un entier initialisé à 0 qui permet d'obtenir un id différent pour chaque entité en incrémentant automatiquement

Son constructeur ne prend que deux paramètres qui sont la range et la position de l'entité, les autres attributs sont modifiés par des setters au besoin dans les différentes classes filles.

La classe possède aussi des méthodes de vérification pour ses classes filles (**isVirus**, **isNode**, **isProjectile**) qui prennent en paramètre une Entity et vérifient si celle-ci est une instance de la classe fille en renvoyant un booléen.

Elle dispose également d'une procédure **isInRange(Entity)** qui va renvoyer un booléen, vérifiant qu'une entité est à portée d'une autre.

Ainsi, la classe *Entity* est une classe très importante au sein de l'architecture du projet car c'est elle qui va définir les bases de tous les éléments actifs du jeu.

Tower

Tower		
f	upgradePrice	int
f	reloadingTime	int
f	spawningTime	int
f	env	World
f	projectileDamages	int
m	Tower(int, Location, int, int, int, World)	
m	Tower(Location)	
m	Tower(int, Location, World)	
m	isAFirewall(Tower)	boolean
m	isACPU(Tower)	boolean
m	disable()	void
m	enable()	void
m	hasTarget()	boolean
m	addRangedVirus(Virus)	void
m	delRangedVirus(Virus)	void
m	act()	void
m	detection()	void
p	target	Virus
p	active	boolean
p	inRangeVirus	ArrayList<Virus>
p	price	int

La classe Tower est une classe abstraite fille de la classe Entity. Elle permet de créer des objets que l'on peut poser dans l'environnement. Ce sont des bâtiments, il comprennent donc toutes les tours, le CPU et le Firewall.

Étant une sous-classe de *Entity*, elle dispose d'un accès à ses méthodes et attributs.

Elle contient les attributs suivants :

- **protected int reloadingTime** correspond à un cooldown entre les salves d'actions de la tour
- **protected int spawningTime** correspond au temps qui sépare deux actions d'une tour
- **private final boolean active** est un booléen qui définit le statut de l'entité, soit elle est active et peut donc agir, soit elle est inactive

- **private Virus target** est le virus qui est pris pour cible par la tour
- **private ArrayList<Virus> inRangeVirus** est une liste contenant tous les virus qui sont à portée d'action de la tour
- **protected World env** est une variable qui contient l'environnement dans lequel se trouve afin qu'il puisse interagir avec
- **protected int projectileDamages** est un entier qui contient les dégâts que les projectiles envoyés par la tour feront à leur cible

La tour possède plusieurs constructeurs:

- Un qui prend en paramètre une range (int), une position (*Location*), un prix d'upgrade (int), un cooldown entre les salves (int), un cooldown entre les actions (int) et un environnement (*World*). Elle fait appel au constructeur de la classe *Entity* qui prend la range et la position.
- Un qui prend en paramètre une range (int), une position (*Location*) et un environnement (*World*). Elle fait également appel au constructeur d'*Entity*.
- Un qui prend en paramètre une position (*Location*) et qui fait appel au constructeur d'*Entity* en initialisant la range à 0.

Tout comme la classe *Entity*, la classe *Tower* possède des méthodes de vérifications pour les tours particulières (**isACPU**, **isAFirewall**) qui prennent en paramètre une *Tower* et vérifient si celle-ci est une instance de la classe fille en renvoyant un booléen.

Elle possède des procédures pour changer le statut de la tour (la passant d'activée à désactivée), **enable()** et **disable()**. Les procédures **addRangedVirus(Virus)** et **delRangedVirus(Virus)** permettent de mettre à jour la liste des virus à portée d'action de la tour.

Elle dispose également d'une procédure abstraite **act()** inhérente à ses classes filles, elle contient ce que les tours doivent exécuter à chaque tick du programme, ainsi que d'une méthode abstraite **getPrice()** qui renvoie un entier correspondant au prix d'une tour.

La classe *Tower* est également entièrement responsable de la détection des virus, chose qu'elle fait avec la procédure **detection()** qui va analyser la position des différents virus par rapport à la tour et estimer si oui ou non ils sont à portée (en utilisant la méthode **isInRange(Entity)** de la classe mère) et en fonction du retour qu'obtient la méthode, elle actualise la liste **inRangeVirus** et la **target** de la tour. Elle est également en charge du cas particulier du CPU, une tour qui peut prendre des dégâts et qui par conséquent nécessite une gestion des virus particulière (gestion de ses points de vie et des virus qui se "suicident" dessus).

La procédure **detection()** est une méthode clé de la classe *Tower*, permettant la mise à jour de plusieurs attributs et par conséquent, de mettre la tour à jour par rapport à ce qu'il se passe dans l'environnement.

Virus

Virus		
f	sizeW	int
f	sizeH	int
f	direction	Vector
f	targets	Set<Entity>
m	Virus(int, Location, Tile, int, int, World)	
m	detection()	void
m	lookForFirewall()	boolean
m	move()	void
m	addTarget(Entity)	void
m	removeTarget(Entity)	void
m	act()	void
m	die()	void
m	hit(int)	void
p	world	World
p	currentHP	int
p	alive	boolean
p	current	Tile
p	spawnTile	Tile
p	virusID	int
p	speed	int

Virus est une classe abstraite qui va gérer les différents ennemis du jeu.

Étant une classe fille d'*Entity*, elle dispose d'un accès à ses méthodes et attributs.

Elle contient les attributs suivants :

- **protected int sizeW et sizeH** qui correspondent aux dimensions que le virus possède, ceci étant nécessaire par rapport aux méthodes de détection de range
- **protected int currentHP** est une variable qui contient les points de vie actuels du virus
- **protected Tile current** contient la tuile de jeu actuelle du virus
- **private Vector direction** correspond à un vecteur définissant la direction que le virus devra prendre au prochain tick pour actualiser sa position
- **private int speed** est un coefficient qui modifie la "vitesse" du virus, il est multiplié par le vecteur direction

- **protected Set<Entity> targets** est une liste sans doublon des instances d'*Entity* que le virus a pris pour cible
- **private int virusID** est un entier qui permet d'identifier le sprite correspondant au virus créé
- **protected Tile spawnTile** est un objet de classe *Tile*, donc une case de la map, qui correspond au point de spawn des virus
- **protected World world** correspond à l'environnement dans lequel évolue le virus

La classe *Virus* ne possède qu'un constructeur et prend en paramètre une range (int), une position (*Location*), une tuile (*Tile*) qui correspond à sa tuile d'apparition et à sa tuile actuelle (à la création de l'instance), une vitesse (int), un identifiant pour les sprites (int) et un environnement (*World*). Elle fait appel au constructeur de la classe *Entity* qui prend la range et la position.

Elle possède des méthodes pour mettre à jour son set de **targets**.

Elle dispose également d'une procédure abstraite **act()** inhérente à ses classes filles, elle contient ce que les tours doivent exécuter à chaque tick du programme.

À l'instar de la classe *Tower*, elle dispose également d'une procédure **detection()** afin de répertorier les tours qui sont à portée d'action des virus.

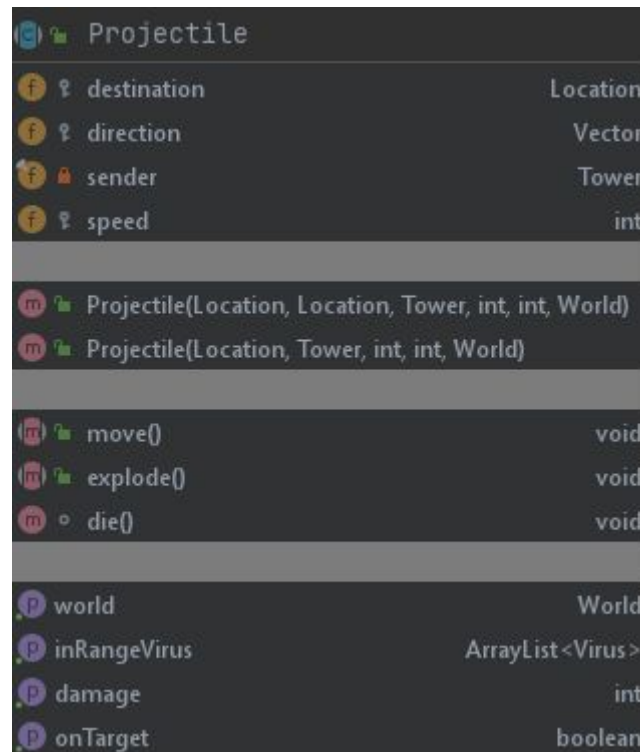
Les instances de *Virus* ont besoin de se déplacer de manière "autonome" sur la map afin de suivre le chemin. Pour cela, la classe dispose d'une procédure **move()** qui, en se basant sur la position actuelle du virus, définira les coordonnées du vecteur **direction** et les ajoutera aux coordonnées actuelles du virus. Elle est également programmée de telle sorte à ce que la direction du virus ne se mettent qu'à jour à condition qu'il soit arrivé au centre de sa tuile actuelle.

Le corps de **move()** est encapsulé dans un *if* qui fait appel à une méthode **lookForFirewall()**, méthode qui va vérifier qu'il n'y a pas de *Firewall* (mur bloquant le chemin des virus) sur le chemin. Si ce n'est pas le cas, la procédure **move()** s'exécute bien. Sinon, elle ne s'exécute pas, laissant le virus immobile.

La classe possède aussi une procédure **hit(int)** qui permet d'actualiser **currentHP** en soustrayant un certain nombre, infligeant ainsi des dégâts aux virus. Elle dispose également d'une procédure **die()** qui va mettre **currentHP** du virus à 0, le tuant donc.

La classe *Virus* est donc la classe centrale concernant les ennemis, peu importe leurs effets étant donné que ceux-ci sont définis dans la procédure **act()** des sous-classes.

Projectile



Projectile	
destination	Location
direction	Vector
sender	Tower
speed	int
Constructors:	
Projectile(Location, Location, Tower, int, int, World)	
Projectile(Location, Tower, int, int, World)	
Methods:	
move()	void
explode()	void
die()	void
Private Fields:	
world	World
inRangeVirus	ArrayList<Virus>
damage	int
onTarget	boolean

Projectile est une classe abstraite qui va gérer les projectiles tirés par les tours.

Étant une classe fille d'*Entity*, elle dispose d'un accès à ses méthodes et attributs.

Elle contient les attributs suivants :

- **protected Location destination** correspond au couple de coordonnées vers lequel le projectile doit se diriger
- **private Vector direction** correspond à un vecteur définissant la direction que le projectile devra prendre au prochain tick pour actualiser sa position
- **private final Tower sender** est la tour qui a envoyé le projectile
- **private final int damage** correspond aux dégâts infligés par le projectile à l'impact
- **private final World world** correspond à l'environnement dans lequel évolue le projectile
- **private int speed** est un coefficient qui modifie la "vitesse" du projectile, il est multiplié par le vecteur direction

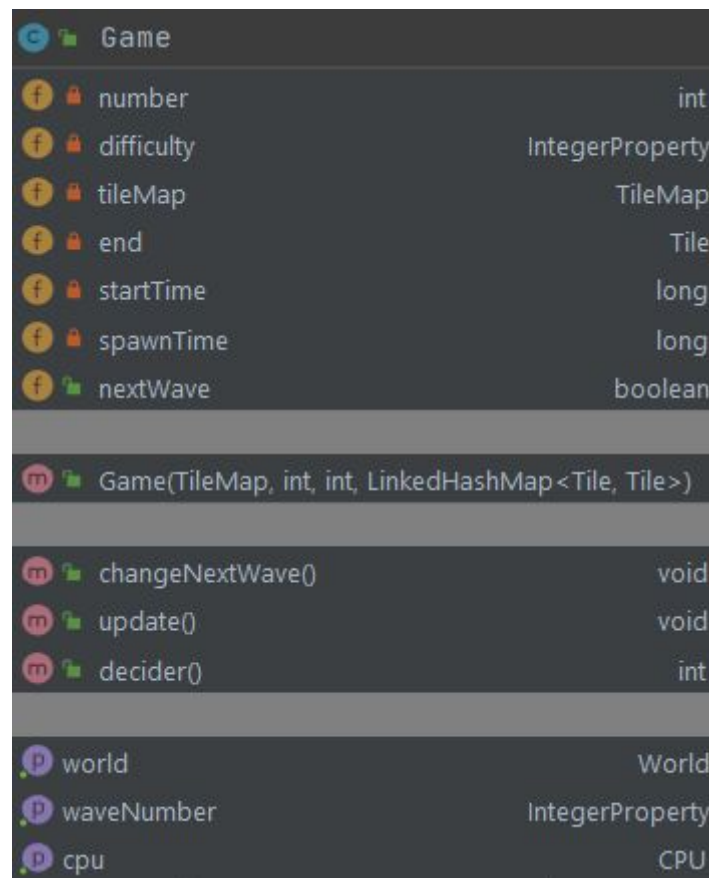
Projectile possède deux constructeurs:

- Un qui prend en paramètre une position (*Location*), une destination (*Location*), une tour qui tire le projectile (*Tower*), les dégâts (int), la range (int) et son environnement (*World*). On fait appel au constructeur d'*Entity*.

- Une autre qui prend les mêmes paramètres à l'exception de la destination qui n'existe pas.

Chaque projectile possédant sa manière de se déplacer dans l'environnement, la classe *Projectile* possède une procédure abstraite **move()** qui est définie dans ses sous-classes. D'autres méthodes abstraites qu'elle possède sont **isOnTarget()** qui renvoie un booléen si le projectile a atteint sa cible selon ses critères et la procédure **explode()** qui définit ce qu'il se passe lorsque le projectile atteint sa cible.

Game



Game est une classe centrale du programme, c'est elle qui va gérer les vagues d'ennemis, le choix des ennemis, les statistiques du joueur (nombre d'ennemi tués, ..). Avec la classe *World* ce sont les deux classes qui sont le "moteur" du programme.

Elle contient les attributs suivants :

- **private int number** correspond au nombre d'ennemis qui apparaissent par vague
- **private IntegerProperty difficulty** est le niveau de difficulté actuel de la vague
- **private IntegerProperty waveNumber** est le numéro de la vague en cours
- **private World env** correspond à l'environnement dans lequel la partie se déroule
- **private Tile end** est la tuile du jeu qui est considérée comme la fin du chemin

- **private long startTime et spawnTime** correspondent au temps actuel et au cooldown défini entre l'apparition de chaque ennemi
- **public boolean nextWave** est le statut de la vague, si elle est finie et qu'il n'y a plus d'ennemis, elle passe en *false*.
- **public CPU cpu** est l'entité finale du chemin et représente ce que le joueur a à défendre

















































































Game prend en paramètre une tilemap qui définit la carte du jeu (*TileMap*), une ligne (int) et une colonne (int) qui correspondent à la tuile finale du parcours ainsi qu'une liste de tuile ordonnées contenant le chemin (*LinkedHashMap<Tile, Tile>*).

Elle possède une méthode **decide()** qui à la responsabilité de gérer quel ennemi doit apparaître dans la vague. Elle tire un nombre aléatoire et fait apparaître un ennemi en fonction d'un pourcentage de chance prédéfini.

La procédure la plus importante de la classe *Game* est **update()**. Elle va dans un premier temps vérifier que la vague est active. Si c'est la cas, on va attendre un certain temps, et si il y a des ennemis à faire apparaître, alors on fait spawn un ennemi défini par un appel de **decide()**. Dans le cas où il n'y aurait plus d'ennemis à faire apparaître, on met à jour les valeurs du jeu (nombre de vagues, difficulté) et si on a fait un certain nombre de vagues la difficulté augmente. On réinitialise le nombre d'ennemis puis on déclare la vague comme étant terminée.

Cette méthode de classe est appelée par la procédure du contrôleur **tick()** qui permet de mettre à jour le jeu et est donc une des méthodes principales du programme.

MapController

MapController		
 	inPause	boolean
 	gameloop	Timeline
 	◦ root	BorderPane
 	kills	Label
 	imageAfast	ImageView
 	imageGb	ImageView
 	imageVG	ImageView
 	imageKbd	ImageView
 	firewall	ImageView
 	adblock	ImageView
 	sudvpn	ImageView
 	ckleaner	ImageView
 	heat	Label
 	grid	TilePane
 	world	Pane
 	bitcoins	Label
 	round	Label
 	tileMap	TileMap
 	game	Game
 	env	World
 	◦ initialize()	void
 	add_menu()	void
 	start(int)	void
 	shopDescription()	void
 	tick()	void
 	initLoop()	void
 	createTower(MouseEvent)	void
 	◦ setTowerOnAfast()	void
 	◦ setTowerOnGb()	void
 	◦ setTowerOnVG()	void
 	◦ setTowerOnKbd()	void
 	nextWaveChange()	void
 	setInPause()	void
 	◦ setTowerOnFirewall	void
 	◦ useAdblock()	void
 	◦ useSudVPN()	void
 	◦ useFlush()	void
 	end()	VBox
 	selectScreen()	VBox
 	map()	VBox

MapController est la classe centrale du programme, c'est elle qui va articuler entre elles les différentes classes et relier le modèle à la vue. Elle va gérer l'actualisation du modèle et lier celui-ci à la vue.

Étant la classe contrôleur, elle contient des attributs issus du FXML. Les images des tours, de bonus, les images de fond, etc .

Elle contient les attributs suivants :

- **private static boolean inPause** qui est le statut de la partie, donc en pause ou non
- **private Timeline gameloop** qui correspond au déroulement de la boucle de jeu, donc le nombre de fois où l'on met à jour le jeu par secondes
- **private TileMap tileMap** qui contient les différentes tuiles du jeu
- **private Game game** qui est la session de jeu actuelle et qui contient les informations sur la partie ainsi que le système de vague
- **private World env** qui est l'environnement dans lequel se déroule le jeu

La classe contient des méthodes permettant de set l'image de la tour ou utiliser un bonus. C'est dans le FXML que l'on définit l'appel des méthodes lors du clic sur l'image.

Les méthodes **end()** et **selectScreen()** sont des méthodes affichant un écran de sélection, permettant de choisir entre diverses options telles que lancer la partie ou quitter. La méthode **end()** affiche un écran de fin lorsque le joueur a perdu.

setInPause() et **nextWaveChange()** sont des procédures appelées lors du clic sur les boutons assignés. Le premier met la partie en pause en bloquant le passage dans la procédure **tick()**. La seconde déclenche la vague suivante lorsque celle-ci est terminée.

La procédure **initLoop()** va créer la *Timeline* lui disant d'appeler la procédure **tick()** à un interval fixé (dans notre cas, trente fois par secondes) permettant d'actualiser le programme. La procédure **tick()** va faire appel aux fonctions qui mettent à jour le modèle, c'est à dire **update()** dans *Game* et **nextRound()** dans *World*. Elle vérifie également que le jeu n'est pas en pause et s'assure que les éléments qui doivent être visibles sont bien visibles après mise à jour du modèle.

createTower(MouseEvent) va être chargée de créer une tour en fonction de la tour sélectionnée et de l'ajouter à l'environnement tout en vérifiant que le joueur possède assez d'argent pour l'acheter et que celle-ci n'est pas placée sur le chemin.

start(int) est une procédure appelée par **initialize()** et permet d'initialiser le programme. Elle prend en paramètre un entier qui correspond à la map à utiliser (en l'occurrence, seulement 1). On initialise la plupart des variables, créons les listeners et affectons les binds aux attributs avant de finir en créant la *Timeline*.

Interfaces

Nous disposons de deux interfaces afin de gérer des cas particulier concernant les tours où pour pouvoir formater certaines classes (en l'occurrence, les bonus).

Bonus

```
package models.entities.bonus;

public interface Bonus {

    void act();

    boolean isActive();

    int getPrice();

}
```

Bonus est une interface faite pour préparer la forme des autres bonus. Elle contient les procédure **act()** dont le rôle sera de faire agir le bonus, appliquer ses effets à l'utilisation. Une méthode **isActive()** qui vérifiera le statut du virus (car on ne peut pas utiliser plusieurs bonus à la fois à l'exception du *Firewall* qui est un bonus particulier) et renverra un booléen. Et enfin la méthode **getPrice()** qui renverra un entier correspondant au prix du bonus.

Damageable

```
package models.entities.tower;

public interface Damageable {

    void getDamaged(int a);

    boolean isAlive();

}
```



Damageable est une interface créée afin de gérer les cas particulier de *Tower* que sont *Firewall* et *CPU*. Cette classe permet d'infliger des dégâts aux tours qui ont de la vie via la procédure **getDamaged(int)** et de vérifier leur statut (si elles sont "mortes" ou non) via **isAlive()**.

Algorithmes

BFS

Cet algorithme est situé dans la classe Graphe et sert à déterminer un chemin entre 2 tuiles. Il les reçoit en paramètre, ainsi que la map concernée, et parcourt celle-ci en largeur.

À partir du point de destination, on récupère les tuiles adjacentes, puis on les empile dans une liste ordonnée FIFO. Cela permet de remplir une HashMap ayant pour clé chaque tuile visitée, et pour valeur la tuile dont elle est issue.

```
FONCTION bfs: LinkedHashMap<Tile, Tile>
```

```
Variables:
```

```
temp : Tile
```

```
file : LinkedList
```

```
path : LinkedHashMap
```

```
tileMap : TileMap
```

```
Initialisation:
```

```
file ← ajout de end en première position
```

```
path ← ajoute end au path (<end: end>)
```

```
Début
```

```
  TantQue temp n'est pas égal à start Faire:
```

```
    Si la file est vide Alors:
```

```
      temp ← la dernière tuile supprimée de la file
```

```
      neighbors ← appel getAvailableNeighbors(tileMap, temp)
```

```
      Pour chaque tuile de neighbors Faire:
```

```
        file ← ajout de la tuile en première position
```

```
        path ← ajout de la paire <tile: temp>
```

```
      FinPour
```

```
    FinSi
```

```
  FinTantQue
```

```
  retourner path
```

```
Fin
```

FONCTION getAvailableNeighbors: Collection de Tile

Variables:

tile : Tile

tileMap : TileMap

neighbors : Collection de Tile

Initialisation:

neighbors ← création de la collection

Début

neighbors ← ajout du voisin du haut de tile

neighbors ← ajout du voisin du bas de tile

neighbors ← ajout du voisin de gauche de tile

neighbors ← ajout du voisin de droit de tile

Pour chaque tile de neighbors Faire:

Si tile indisponible Alors:

 neighbors ← retrait de tile

retourner neighbors

Fin

FONCTION getTile: Tile

Variables:

col : int

row : int

loc : Location

Initialisation:

loc ← nouvelle location (row, col)

Début

Pour chaque tile de tileMap Faire:

Si location de tuile = loc Alors:

retourner tile

retourner null

Fin

Move

Cet algorithme se situe dans la classe Virus et constitue le moyen de déplacement de ce type d'entité. En effet cette procédure s'accouple parfaitement avec l'algorithme du BFS abordé au-dessus puisqu'il s'agit maintenant de parcourir le chemin généré. Ainsi, en se basant sur les positions des tuiles adjacentes (actuelle et parent), on détermine un vecteur de direction .

PROCÉDURE move:

Variables:

parent : Tile

Initialisation:

parent \leftarrow La tile pointée par celle de l'entité dans la HashMap path.

Début

```
    Si aucun Firewall n'est détecté Alors
        Si le virus est au centre de sa tile Alors
            Si son X < X de parent Alors
                X de direction  $\leftarrow$  1
            SinonSi son X > X de parent Alors
                X de direction  $\leftarrow$  -1
            Sinon
                X de direction  $\leftarrow$  0
            FinSi

        Si son Y < Y de parent Alors
            Y de direction  $\leftarrow$  1
        SinonSi son Y > Y de parent Alors
            Y de direction  $\leftarrow$  -1
        Sinon
            Y de direction  $\leftarrow$  0
        FinSi

    FinSi
    position  $\leftarrow$  position(X,Y) + direction(X,Y)
FinSi
Fin
```

JUnit

Nous avons testé `isInRange()` de la classe `Entity`. Commune à toutes les entités, elle constitue $\frac{1}{3}$ de la mécanique de jeu puisque toute action événementielle indépendante du joueur est déclenchée lors d'une détection d'entité.

Chaque type de vérification applique 4 tests différents:

1. Quand l'entité n'est pas dans le rayon de détection de l'entité testée
2. Quand l'entité se trouve dans le rayon de détection
3. Quand l'entité se trouve sur la position exacte de l'entité testée
4. Quand l'entité se trouve sur la position exacte de l'entité testée mais sans rayon

Il nous a donc paru intéressant et censé de nous concentrer sur cette méthode afin de vérifier si le processus de détection était fonctionnel et ce même dans des cas de figure où le résultat importe peu puisque la mécanique n'est pas exploité dans le jeu, comme c'est le cas de la détection des Tower par les Projectiles.

Taux de participation (volume de code)

