

# Documents Pour les SAEs S2 2022-2023

## Structures de données

**Dans la classe BFS, voici les structures de données utilisées:**

**LinkedList** : Utilisée comme une file, FIFO pour stocker les cases à traiter lors de l'algorithme de recherche en largeur. La file est déclarée comme *LinkedList<Case> fifo* et est initialisée dans la méthode *grilleBFS()*. Elle est utilisée pour ajouter et retirer des éléments de la file dans la boucle principale de l'algorithme.

**Map** : Utilisée pour stocker les prédécesseurs des cases. La map est déclarée comme *Map<Case, Case> prédécesseurs* et est initialisée dans la méthode *grilleBFS()*. Elle est utilisée pour associer chaque case à son prédécesseur lors du parcours en largeur. La méthode *cheminVersSource()* utilise cette map pour reconstruire le chemin à partir de la case cible jusqu'à la source.

Ces structures de données sont utilisées pour faciliter la mise en oeuvre de l'algorithme de recherche en largeur dans la classe BFS. LinkedList est utilisée pour maintenir une file d'attente de cases à explorer, tandis que Map (HashMap) est utilisée pour stocker les relations entre les cases et leurs prédécesseurs.

**Dans la classe Grille, les structures de données utilisées sont les suivantes :**

**Map** : Utilisée pour stocker les cases adjacentes à chaque case. La HashMap est déclarée et initialisée avec *Map<Case, Set<Case>> listeAdj*.

**Set** : Utilisée pour stocker les cases adjacentes à chaque case dans la grille. Elle est utilisée à plusieurs endroits dans la construction de la grille et dans les méthodes *adjacents()* et *estDeconnecte()*.

Ces structures de données sont utilisées pour représenter la grille du jeu. La Map *listeAdj* est utilisée pour associer chaque case à un ensemble de cases adjacentes. Le Set est utilisé pour stocker les cases adjacentes à chaque case et pour retourner les cases adjacentes dans la méthode *adjacents()*. Ces structures de données permettent de représenter les relations entre les cases de la grille et de gérer les obstacles.

## algorithmique

### Dans la classe **BFS**, voici les algorithmes intéressants présents :

1. Recherche en largeur (BFS) : L'algorithme BFS est implémenté dans la méthode ``grilleBFS()``. Il effectue une recherche en largeur sur une grille à partir d'une case source. Il utilise une file d'attente (fifo) pour stocker les cases à traiter et un dictionnaire (predecessors) pour stocker les prédécesseurs des cases. Cet algorithme permet de parcourir la grille de manière itérative et de trouver les cases accessibles depuis la case source.
2. Calcul du chemin vers la source : L'algorithme pour calculer le chemin vers la source est implémenté dans la méthode ``cheminVersSource()``. Il utilise le dictionnaire de prédécesseurs (predecessors) pour retracer le chemin depuis une case cible jusqu'à la source. Il retourne un ArrayList de cases représentant le chemin trouvé.

### Dans la classe ``GénérateurVague``, voici l'algorithme intéressant présent :

1. Génération des vagues d'ennemis : L'algorithme pour générer les vagues d'ennemis est implémenté dans la méthode ``vaguePourChaqueTour()``. Cet algorithme gère la génération progressive des ennemis au fil des tours de jeu. Il vérifie d'abord si la dernière vague est terminée et si aucune vague n'est en cours. Ensuite, il vérifie si suffisamment de tours se sont écoulées depuis la fin de la dernière vague pour commencer une nouvelle vague. À chaque multiple de 20 tours, un nouvel élément de la vague est généré en appelant la méthode ``genererUnElementDeLaVague()``. L'algorithme compte également le nombre d'ennemis créés dans la vague en cours. Lorsque le nombre d'ennemis créés atteint un certain seuil (la taille de la vague actuelle + 5), l'algorithme marque la fin de la vague en cours en appelant la méthode ``finDUneVague()``.

### Dans la classe ``Balle``, voici l'algorithme intéressant présent :

1. Déplacement de la balle : L'algorithme de déplacement de la balle est implémenté dans la méthode ``seDeplacer()``. Cet algorithme met à jour la position de la balle à chaque itération en fonction de sa vitesse et des directions ``directionX`` et ``directionY``. La balle se déplace dans la direction déterminée par ces valeurs.

### Dans la classe ``LanceMissile`` :

1. Méthode ``sniper()`` : Cette méthode a été ajoutée pour gérer le comportement spécifique du lance-missile. Elle recherche les ennemis à portée de la tourelle et sélectionne le premier

ennemi trouvé (`ennemis.get(0)`). Ensuite, elle recherche les ennemis à portée de ce premier ennemi dans une zone de dégâts en chaîne définie par `32` de rayon et `5` de portée. Si des ennemis sont trouvés dans cette zone, elle crée une nouvelle instance de `Balle` spécifique (`Missile`) en utilisant la méthode `creerBallesDansTourelle()` et l'ennemi cible, puis inflige des dégâts à tous les ennemis trouvés dans la zone de dégâts en chaîne.

2. Méthode `attaquer()` : Cette méthode override la méthode `attaquer()` de la classe parent `Tourelle`. Elle a été modifiée pour inclure la gestion du cooldown du lance-missile. Si le "cooldown" est égal à `0`, cela signifie que la tourelle peut attaquer. Dans ce cas, elle appelle la méthode `sniper()` pour effectuer une attaque spécifique du lance-missile. Ensuite, elle décrémente le "cooldown" de `1`. Si le "cooldown" est égal à `TEMPS - 1` et la balle actuelle (`balleActuelle`) est différente de `null` et a atteint sa cible, elle remet la balle actuelle à `null` pour permettre au lance-missile de tirer à nouveau.

#### **Dans la classe `NuageRalentisseur` :**

1. Méthode `ralentir()` : Cette méthode est ajoutée pour gérer l'effet de ralentissement appliqué par le nuage ralentisseur. Elle utilise la méthode `chercherEnnemisDansPortee()` de l'environnement pour trouver les ennemis à portée du nuage. Ensuite, pour chaque ennemi trouvé, elle vérifie s'il est déjà dans la liste `ennemisDansZone`. Si ce n'est pas le cas, elle réduit la vitesse de l'ennemi en multipliant par le facteur de ralentissement (`ralentissement`) et ajoute l'ennemi à la liste `ennemisDansZone`. La méthode vérifie également si les ennemis dans la liste `ennemisDansZone` sont toujours vivants ou s'ils ont quitté la zone d'effet du nuage. Si un ennemi est mort, il est supprimé de la liste. Si un ennemi a quitté la zone d'effet, sa vitesse est rétablie à la normale et il est également supprimé de la liste.

2. Méthode `accélérerAvantDisparition()` : Cette méthode est ajoutée pour rétablir la vitesse normale des ennemis ralentis lorsque le nuage ralentisseur est supprimé. Elle parcourt la liste `ennemisDansZone` et augmente la vitesse de chaque ennemi en inversant le facteur de ralentissement (`1/ralentissement`).

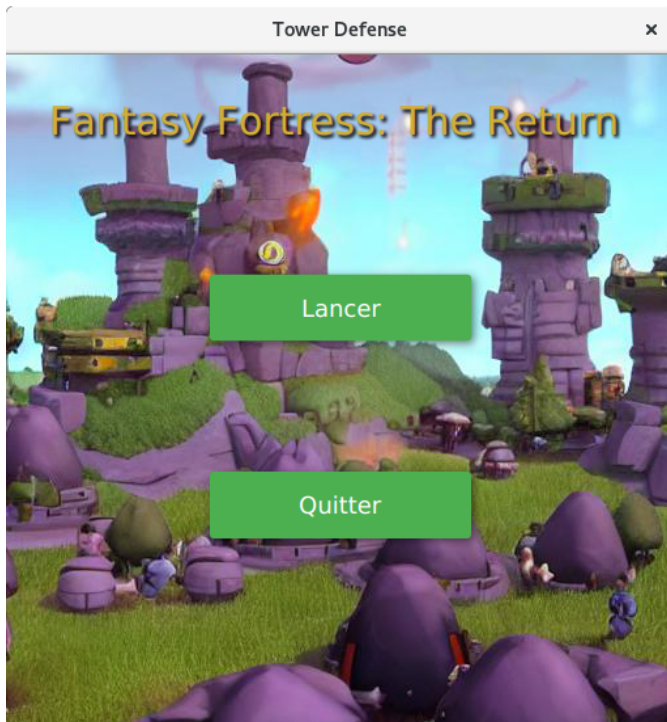
Document utilisateur (SAE **S2.05**) :

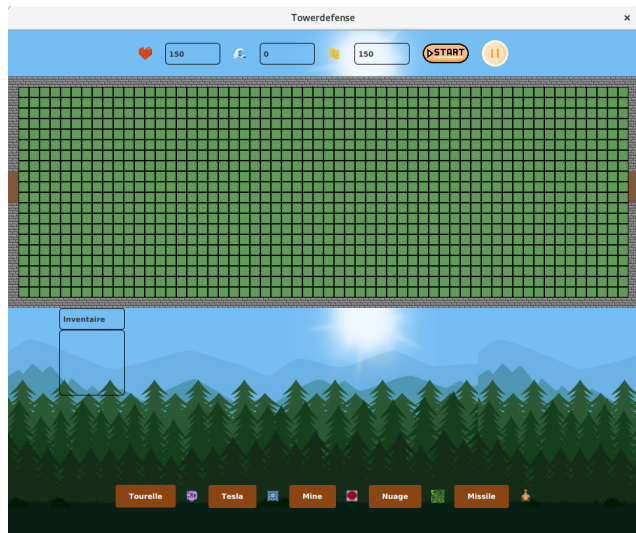
### Description de Fantasy Fortress: The Return :

Bienvenue dans l'univers de "Fantasy Fortress: The Return", un tower defense haut en couleur. Votre mission est de défendre votre base contre les vagues d'ennemis qui cherchent à s'y infiltrer, et cela pendant 50 vagues. Mais attention, dans ce jeu, les ennemis ne suivent pas un chemin prédéfini. Au contraire, le parcours de ces intrus est déterminé au fur et à mesure que vous posez vos tourelles. Plongez-vous dans un monde cartoon, avec des tourelles coloré et des ennemis tout aussi extravagants.

### Présentation du jeu :

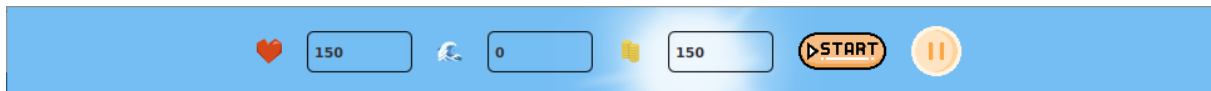
Comme on peut le voir lorsqu'on lance on tombe sur le Menu du jeu avec deux options Lancer ou Quitter:



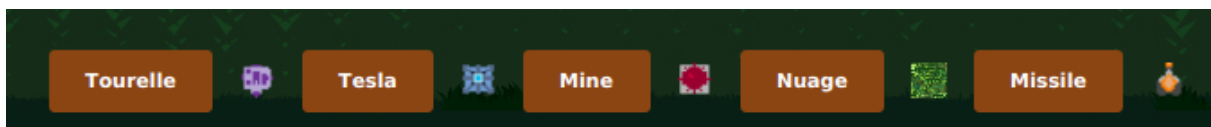


lorsqu'on clique sur Lancer le terrain apparaît

En haut de page on aperçoit la barre de vie du joueur, le nombre de vague actuelle et le nombre d'argent que le joueur possède



En bas de page on aperçoit les différentes défenses qui vont pouvoir être placées sur le terrain avec leur noms. Les défenses de type Tourelle on toutes une façon de tirer différentes c'est pour cela qu'elle n'ont pas de balle de la même couleur ou taille



Tir de la tourelle de base :



La tesla :



Le lance missile :



La mine :



Le nuage ralentisseur :




Il y a 4 types d'ennemis, les ennemis de base :



les éclaireurs :



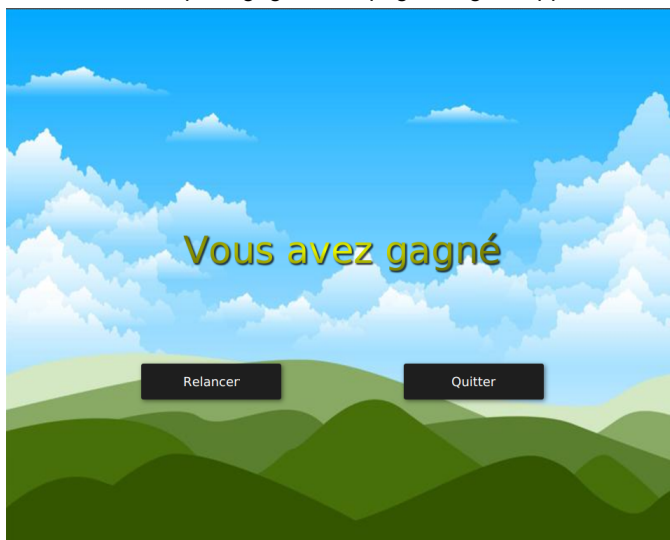
les mastodontes : 

les tanks : 

Lorsqu'on arrive à la fin du jeu, une page apparaît selon le résultat, si le joueur a perdu alors une page Game over s'affiche



Au contraire lorsqu'on gagne, une page "Gagné" apparaît



### Mécanique des actions utilisateur :

Menu : L'utilisateur peut faire 2 actions sur le menu, soit cliquer sur le bouton "Lancer" qui va démarrer la partie, soit cliqué sur le bouton "Quitter" qui va fermer le menu du jeu.

Commencer une partie : Pour commencer la partie, il faut double-cliquer sur l'image start en haut de page.

Mettre en pause : Le joueur peut mettre sa partie en pause en double-cliquant sur l'icône pause qui se trouve à côté du start.

Placer une défense : Pour placer une défense, le joueur doit cliquer une fois sur l'image de la défense qu'il veut, celle-ci va ensuite apparaître et le joueur doit la déplacer avec sa souris pour la mettre sur le terrain.

## **Caractéristique des entités du jeu :**

### **Les Pièges**

La mine a un coût de 40 et fait 200 dégâts à tout type d'ennemis mais lorsqu'elle est posée ne peut faire de dégât qu'à un seul ennemi puisqu'elle disparaît juste après qu'un ennemi soit touché.

Le nuage ralentisseur a un coût de 40 et une durée de vie de 20 secondes durant ces 20 secondes les ennemis qui s'approchent du nuage sont ralentis.

### **Les Tourelles**

La tourelle de base a un coût de 60 et fait 2 de dégâts lorsqu'elle tire, des balles jaunes apparaissent, ce ne sont pas des balles à têtes chercheuses, lorsqu'une balle est lancée la balle suit une droite. Cette tourelle ne peut toucher qu'un ennemi à la fois.

La tesla a un coût de 150 et fait 3 de dégâts lorsqu'elle tire, des balles blanches apparaissent, se sont des balles à têtes chercheuses qui disparaissent lorsque la balle sort de la portée de la tour. Cette tourelle peut toucher jusqu'à 5 ennemis en même temps.

Le lance missile a un coût de 100 et fait 4 de dégâts lorsqu'elle tire, une balle rouge plus grosse apparaît, la balle fait des dégâts en chaîne, puisque lorsqu'elle est lancée si un ennemi se trouve dans un rayon de 30 pixels alors l'ennemi voit ses pv diminués. Cependant le lance missile a un cooldown et ne peut tirer que lorsqu'il n'est pas en cooldown.

### **Les Ennemis facile**

L'Ennemi de base a une vitesse de 2, 100 pv, et une prime de 10 lorsqu'il meurt à cause des défenses, il ne peut pas détruire les défenses.

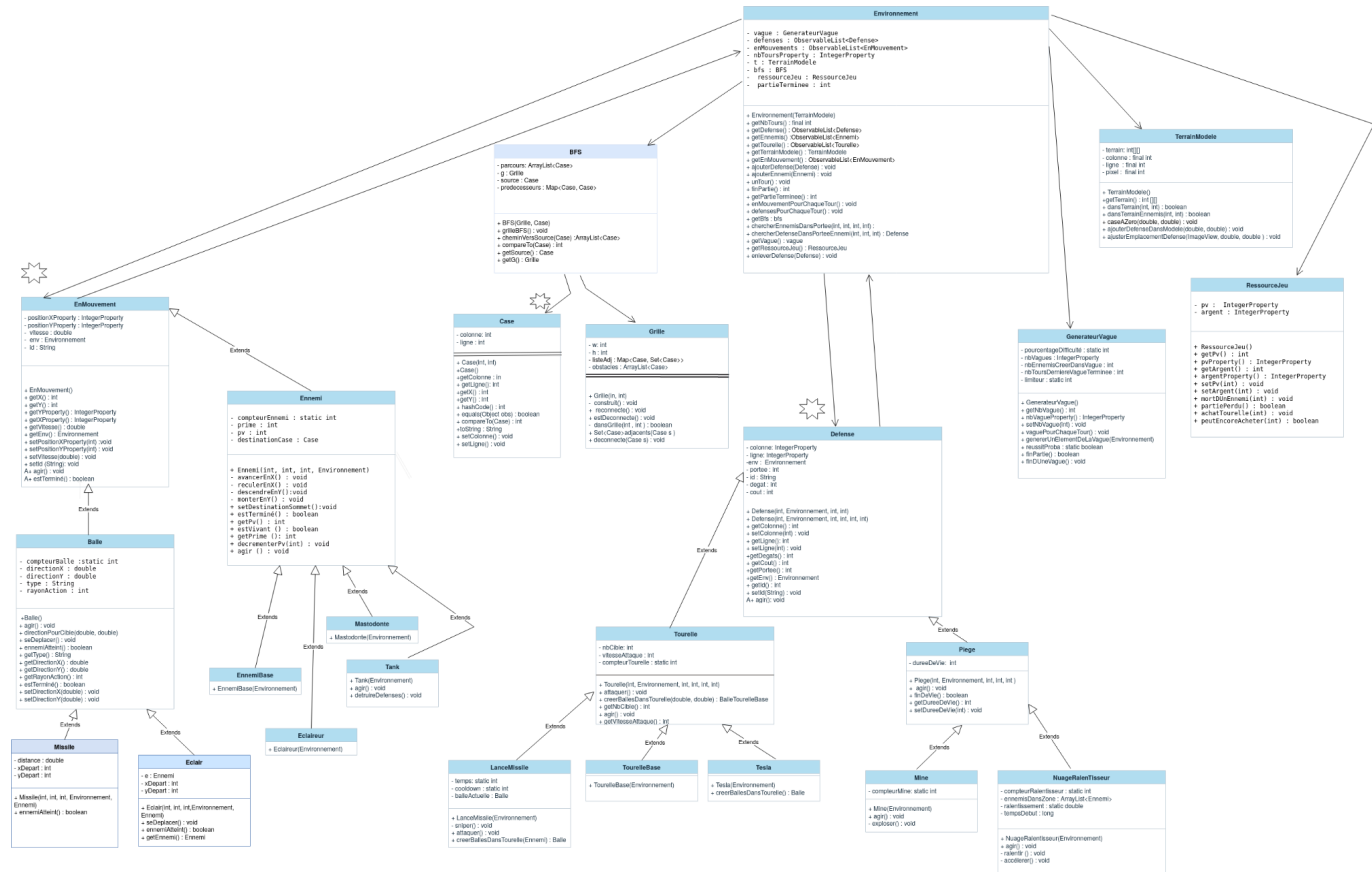
L'Eclaireur a une vitesse de 4, 100 pv et une prime de 20 lorsqu'il meurt à cause des défenses, il ne peut pas détruire les pièges.

### **Les Ennemis plus fort**

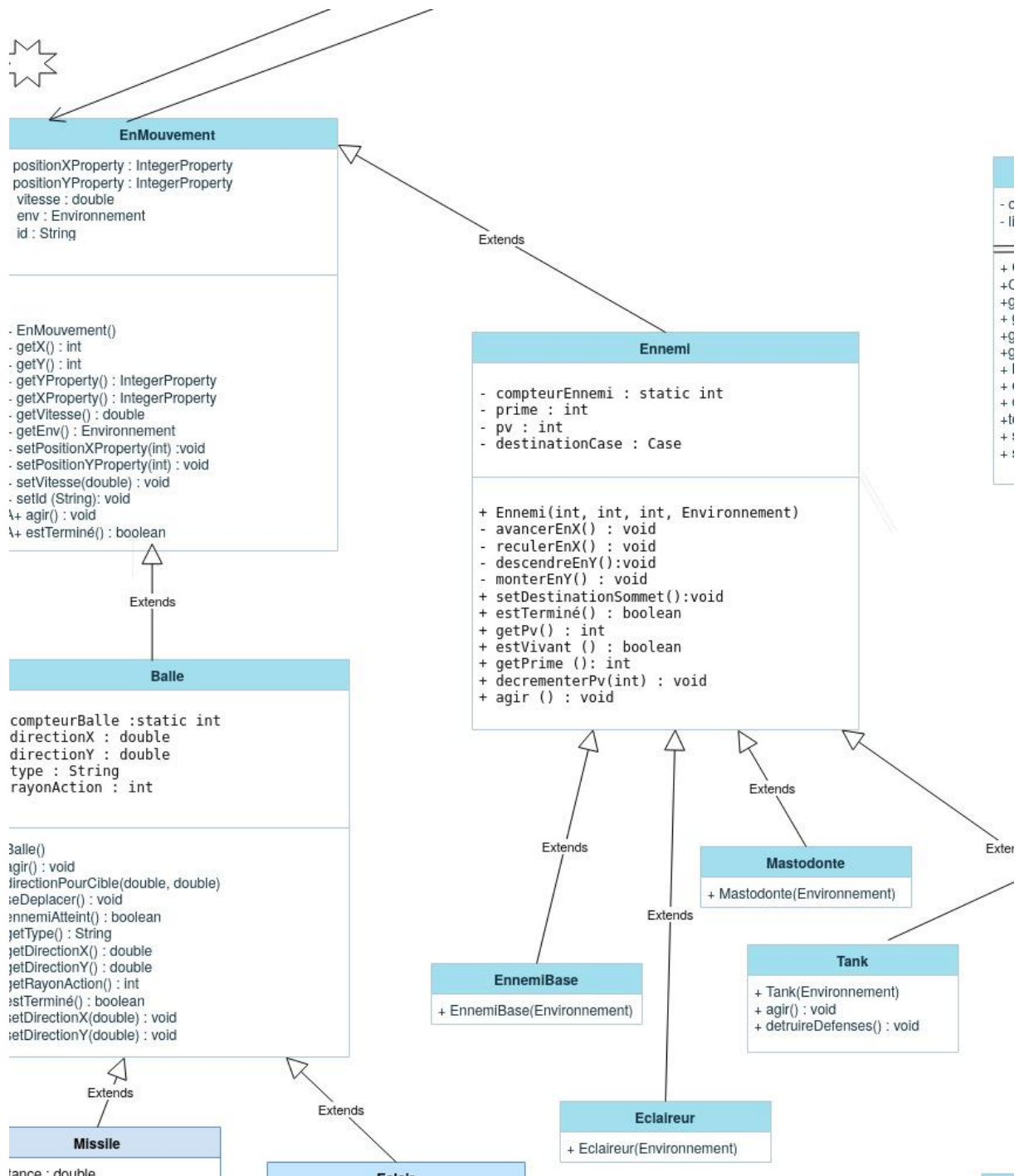
Le Mastodonte a une vitesse de 1, 300 pv et une prime de 50 lorsqu'il meurt à cause des défenses, il ne peut pas détruire les défenses.

Le Tank a une vitesse de 1,350 de pv et une prime de 100 lorsqu'il meurt à cause des défenses, il peut détruire les tourelles lorsqu'elles sont sur son chemin mais ne peut pas détruire les pièges.

## **Diagrammes de classe (SAE S2.01)**





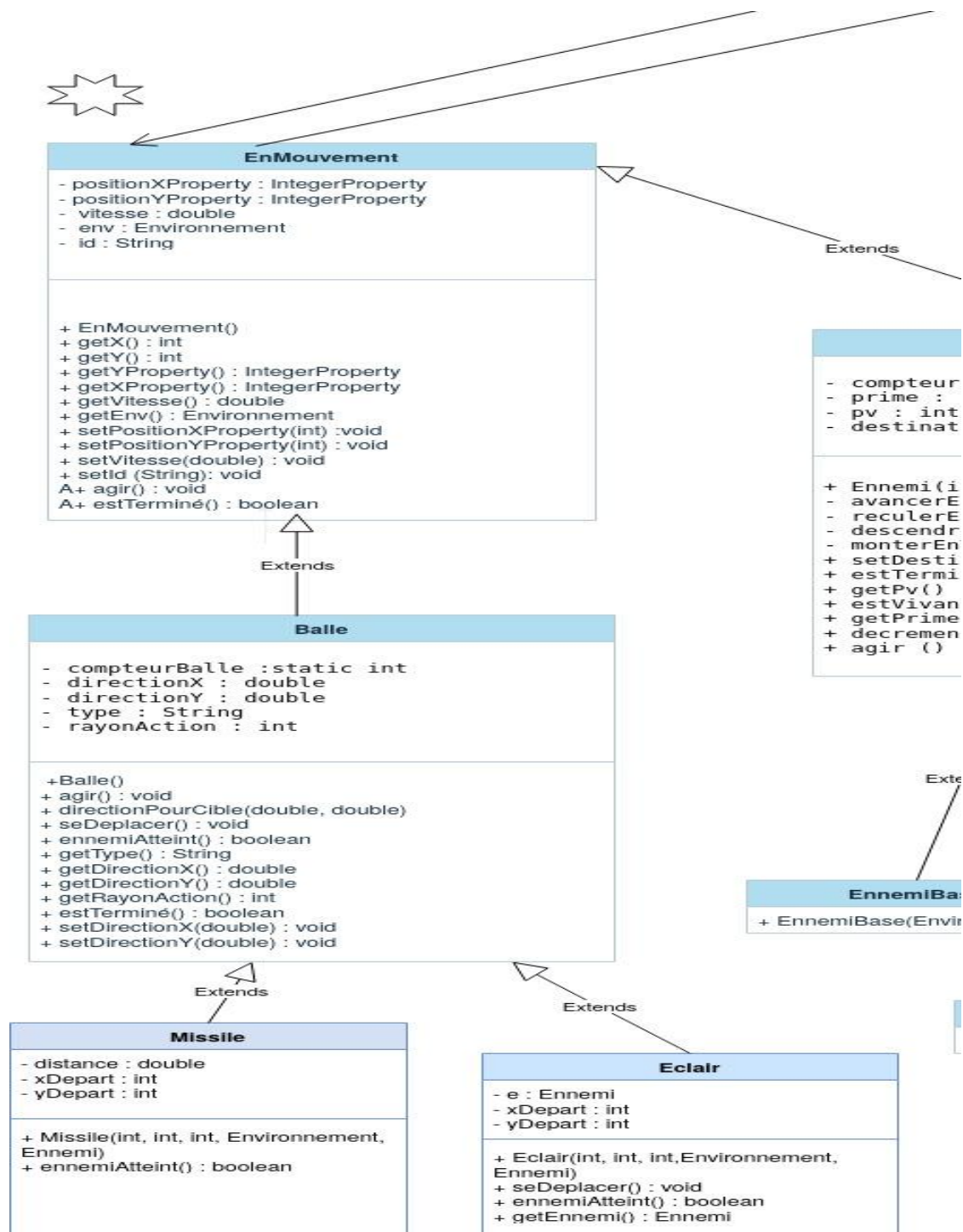


La classe abstraite "Ennemi" constitue la base pour tous les ennemis du jeu. Elle hérite de la classe "EnMouvement" et contient une référence à la classe "Environnement" pour interagir avec le jeu. Cette classe abstraite définit des attributs communs tels que la vitesse, la prime et les points de vie de l'ennemi. La méthode abstraite "agir()" est implémentée dans les sous-classes pour définir le comportement spécifique de chaque ennemi.

La classe "EnnemiBase" est une autre sous-classe de "Ennemi". Elle utilise l'héritage et le polymorphisme pour spécialiser le comportement de l'ennemi de base.

La classe "Mastodonte" hérite également de "Ennemi" et représente un ennemi lourd et puissant. Cette classe utilise l'héritage et le polymorphisme pour spécifier les valeurs spécifiques de vitesse, de prime et de points de vie du mastodonte.

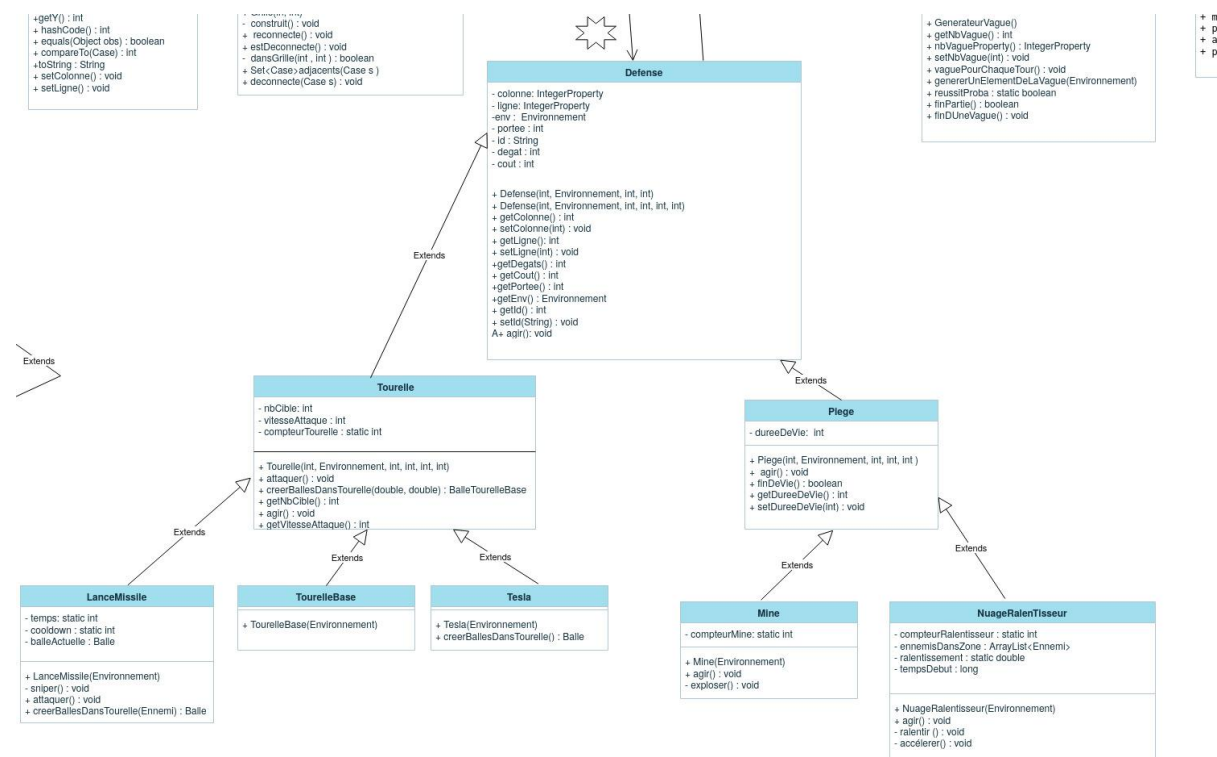
Enfin, la classe "Tank" est une sous-classe de "Ennemi". Elle redéfinit la méthode "agir()". Cette classe utilise l'héritage et le polymorphisme pour spécialiser le comportement du tank.



**Héritage :** Dans ces classes, on observe l'utilisation de l'héritage. La classe Balle hérite de la classe abstraite EnMouvement, ce qui signifie que la classe Balle hérite de tous les attributs et méthodes définis dans la classe EnMouvement. L'héritage permet à la classe dérivée (Balle) d'hériter du comportement de la classe de base (EnMouvement) tout en permettant de spécifier des attributs et des méthodes spécifiques à la classe dérivée.

**Composition** : La composition est également utilisée dans ces classes. Par exemple, la classe EnMouvement a une composition avec la classe Environnement, car elle contient une référence à un objet de la classe Environnement. Cela signifie que la classe EnMouvement utilise et dépend de la classe Environnement pour effectuer certaines opérations.

**Polymorphisme** : Le polymorphisme est présent grâce à l'utilisation de méthodes abstraites. Par exemple, la classe EnMouvement a une méthode abstraite agir(), qui est redéfinie de manière polymorphique dans les classes dérivées telles que Balle, Eclair et Missile. Cela permet de définir un comportement spécifique à chaque classe dérivée, tout en les traitant de manière polymorphique en tant qu'objets de la classe de base EnMouvement. Cela permet une meilleure modularité et extensibilité du code.



La classe abstraite **Piege** hérite de la classe **Defense** et sert de base pour la création de différents types de pièges. Le polymorphisme est présent dans cette

classe avec la méthode abstraite `agir()`, qui est déclarée mais pas implémentée. Les sous-classes concrètes devront fournir une implémentation spécifique de cette méthode.

La classe `Mine` est une sous-classe de `Piege` et hérite ainsi de son comportement général de défense. La spécialisation de la classe `Mine` se fait par le biais de l'héritage et du polymorphisme, où elle peut implémenter la méthode abstraite `agir()` selon ses propres règles et fonctionnalités spécifiques.

La classe `NuageRalentisseur` est également une sous-classe de `Piege`. Elle utilise une composition en maintenant une liste d'ennemis présents dans la zone du nuage ralentisseur. Cette composition permet d'appliquer des effets spécifiques aux ennemis dans cette zone. Le polymorphisme est utilisé pour spécialiser le comportement de la classe `NuageRalentisseur`, en redéfinissant la méthode `agir()` pour ralentir les ennemis présents dans la zone du nuage.

La classe `Tourelle` hérite de la classe `Defense` et représente une tourelle de base. Le polymorphisme est utilisé dans la classe `Tourelle` avec la méthode `creerBallesDansTourelle()`, qui est déclarée mais pas implémentée. Les sous-classes concrètes de `Tourelle` fourniront une implémentation spécifique de cette méthode pour créer les balles correspondantes à leur type.

La classe `Tesla` est une sous-classe de `Tourelle` et hérite donc de ses caractéristiques générales. Le polymorphisme est utilisé pour spécialiser le comportement de la classe `Tesla`. La méthode `creerBallesDansTourelle()` est redéfinie dans la classe `Tesla` pour créer des balles de type `Eclair` spécifiques à la tourelle `Tesla`.

La classe `TourelleBase` est une autre sous-classe de `Tourelle` et hérite de ses propriétés de base. Elle ne possède pas de composition spécifique. Le polymorphisme est utilisé pour spécialiser le comportement de la classe `TourelleBase`, lui permettant d'avoir ses propres caractéristiques et fonctionnalités distinctes.