SAE S3.01

Groupe : Shinwari Said Kama, Chabane Juba, Ngatchou Antoine et Xiang Luc

Choix des modifications et des design pattern

Localisation des différentes branche :

Branche initiale sans modification : master

Branche avec les modifications (design pattern et refactoring) : final

SAE S3.01

Groupe : Shinwari Said Kama, Chabane Juba, Ngatchou Antoine et Xiang Luc

Problème(s) résolus :

Acteur :

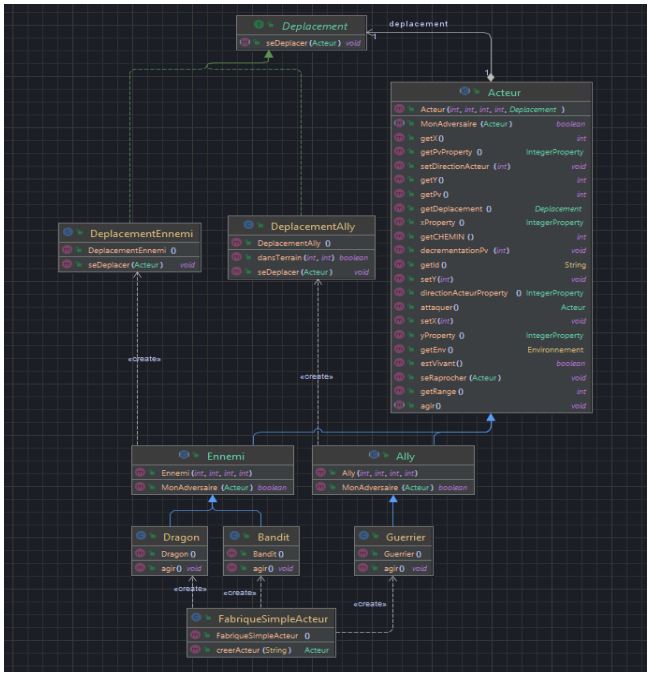
a) Design pattern

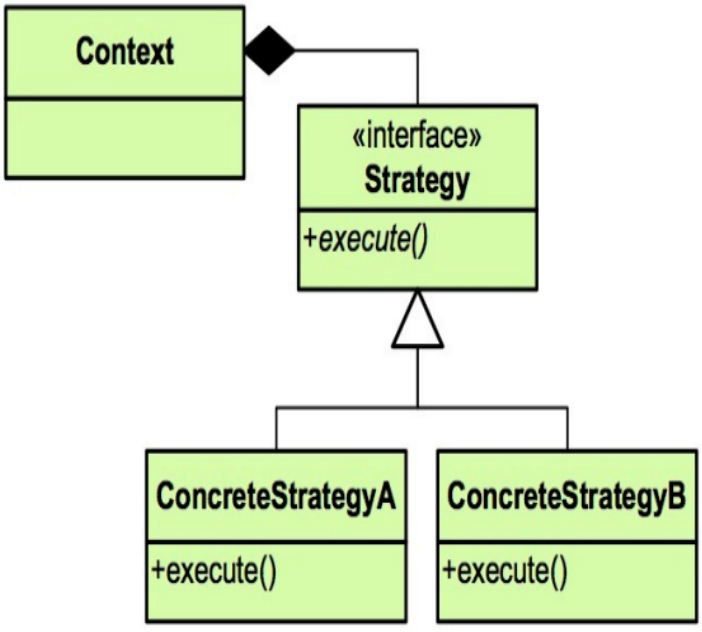
1) Déplacement des acteurs

Problématique	Sur
À la création de chaque sous-classe d'acteur, nous avons utilisé une méthode dans la sous-classe Ally marché sur chemin et dans la sous-classe ennemi seDèplacer.	Acteur

Solution

Design pattern stratégie.





Pourquoi ?
L'avantage de cette conception est que lors de l'introduction de nouveaux types d'alliés ou d'ennemis, nous n'aurons pas besoin de modifier le code existant. Cela respecte le principe de conception dit "ouvert/fermé", qui préconise que les classes devraient être ouvertes à l'extension, mais fermées à la modification.

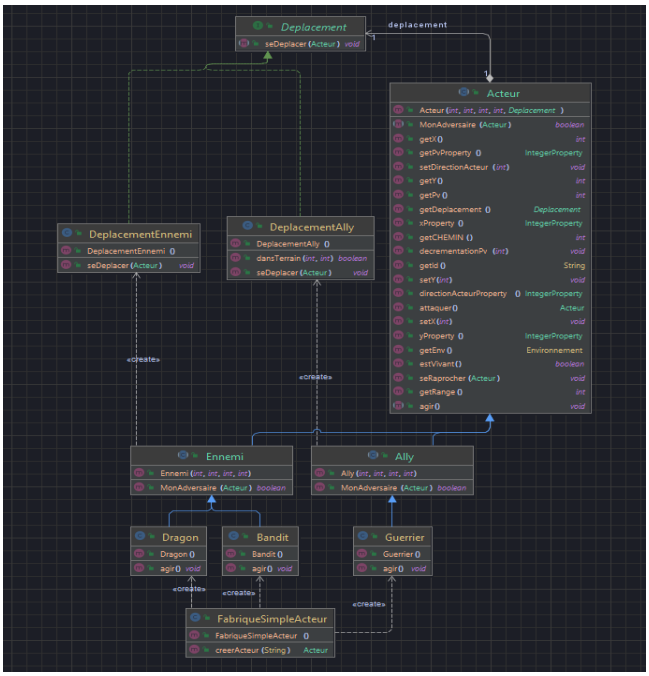
2) Attaque des acteurs

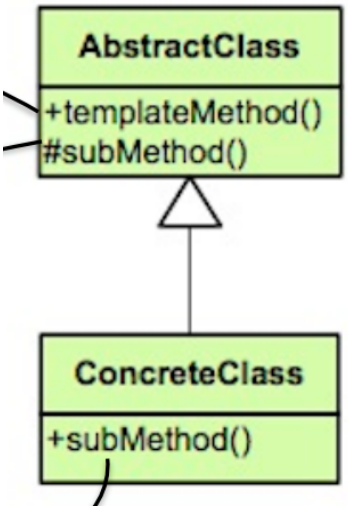
Problématique	Sur
A la création de chaque sous classe de Acteur on devait re-coder la méthode abstract attaquer et ça faisait parfois de la redondance de code sur certains acteurs.	Acteur

Solution

Design pattern template.

Maintenant la méthode abstract attaquer devient une méthode normale qui récupère une méthode(MonAdversaire)qui est codée dans Ally et Ennemi, et permet également de supprimer les “instance of” pour différencier les acteurs.





Pourquoi ?

Redéfinir la méthode abstraite "attaquer" pouvait non seulement engendrer des redondances, mais aussi compliquer la maintenance du code. En effet, toute modification apportée au code nécessitait une adaptation conséquente dans les redéfinitions de cette méthode, augmentant ainsi la charge de travail liée à la gestion des évolutions du code.

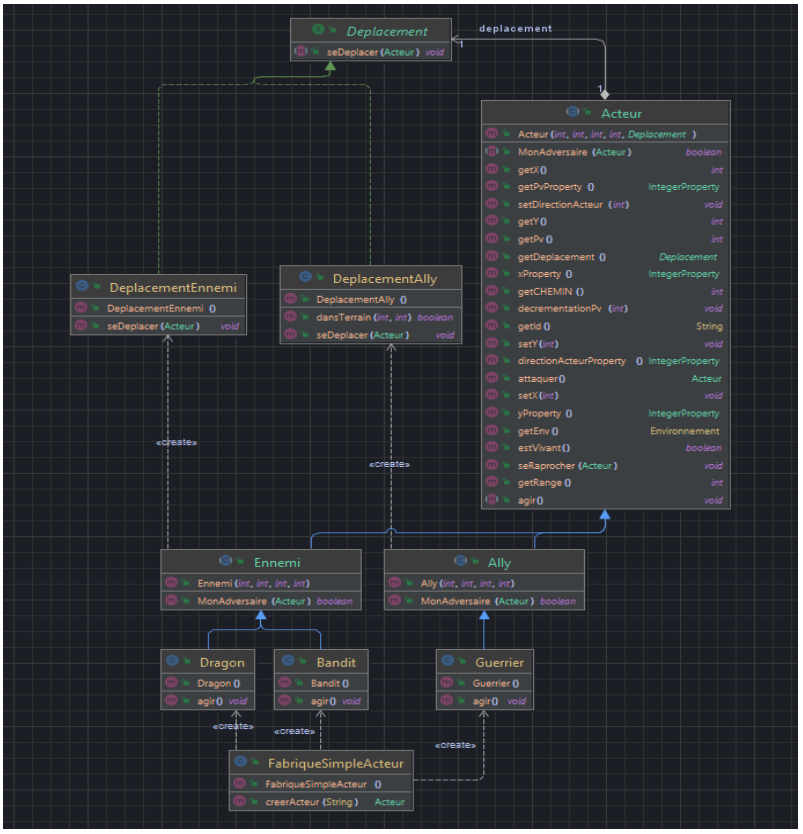
3) Factory simple

Problématique	Sur
Dans le projet, nous avons procédé à l'instanciation de nouveaux acteurs à divers endroits du code et on aimerait	Acteur

avoir une classe générale qui instancie tous les acteurs.	

Solution

Design pattern Simple fabrique



Pourquoi ?

Afin d'assurer une cohérence dans le processus d'instanciation tout en favorisant la maintenabilité du code, nous avons opté pour l'utilisation d'un design pattern de type fabrique simple. L'objectif est d'établir une instanciation générale centralisée permettant la création uniforme de nos acteurs.

b) Refactoring et ajout potentielle

1) Constructeur des acteurs

Problématique	Sur
Les acteurs tels que les bandits, guerriers ou dragon ont tous des paramètres à saisir mais les données sont toujours pareilles et c'est assez embêtant d'initialiser les mêmes paramètres à chaque fois.	Acteurs et ses sous classes.

Solution
On va déclarer directement les paramètres dans le constructeur, ainsi on aura plus besoin de saisir les paramètres à la création d'un acteur et cela va nous faciliter la tâche à la création des acteurs

2) Suppression de variable non utilisé

Problématique	Sur
Les acteurs possèdent une vitesse, sauf que nous ne l'utilisons pas dans le programme.	Acteurs et ses sous classes.

Solution
Étant donné que la variable 'vitesse' n'est pas utilisée, sa présence dans le code introduit de la redondance et ne respecte pas le principe de refactoring et c'est pourquoi nous l'avons retiré du code.

VagueEnnemi :

a) Design pattern

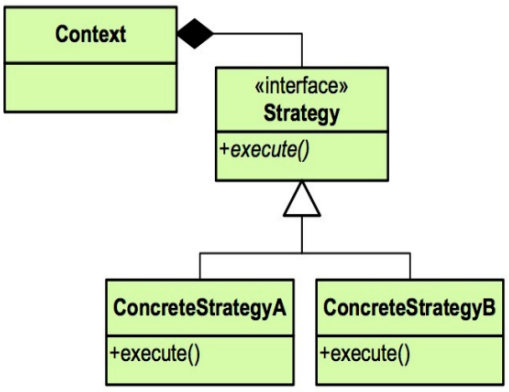
1) Réalisation de plusieurs type de vague

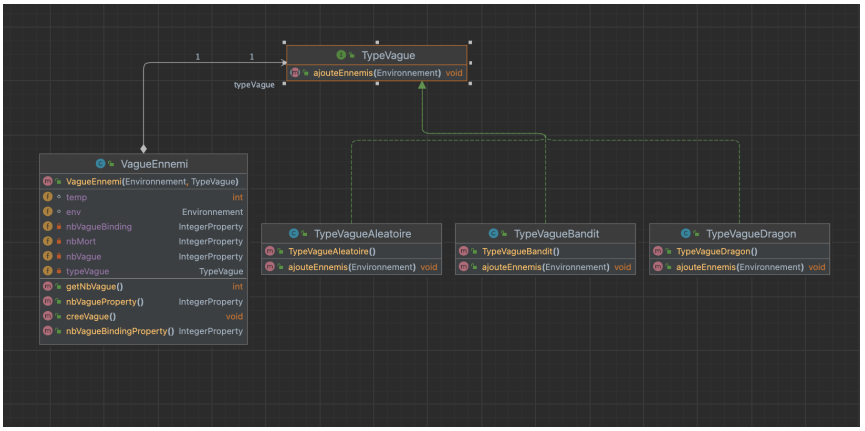
Problématique	Sur
Pour l'instant notre programme a qu'une seule fonction pour la création de vague qui initialise des ennemis dans l'environnement de manière aléatoire, le problème est que si on souhaite une autre façon d'insérer des ennemis dans notre vague on devra modifier notre code et cela est contre le principe ouvert-fermé.	VagueEnnemi

Solution

Design pattern stratégie.

Avant dans "VagueEnnemi" la méthode "creerVague" faisait que créer une vague aléatoire, et maintenant nous avons notre interface "TypeVague" qui contient une méthode pour créer une vague qui sera appelé directement dans "creerVague".



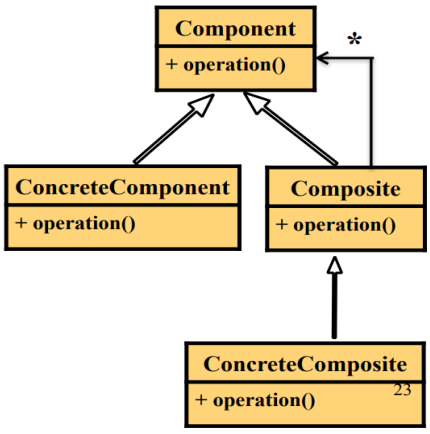


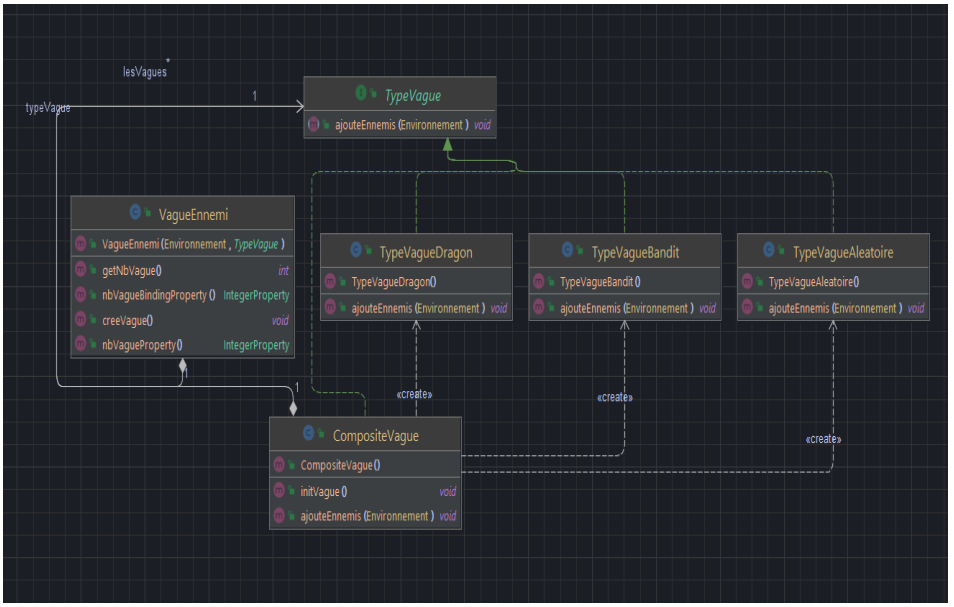
Pourquoi ?
Cela permet de créer des type vagues modulables avec plusieurs types de vagues comme des vagues aléatoires ou une vague composé uniquement de dragon, mais également car lorsqu'on va créer un type de vague on aura pas besoin de changer le code de la classe "VagueEnnemi" et donc c'est ouvert fermer.

2) Utilisé plusieurs type de vague dans le jeu

Problématique	Sur
On a pour l'instant qu'un seul type de création d'ennemi pour la génération de vague et on souhaite avoir plusieurs types de création d'ennemi pour avoir des vagues différentes.	VagueEnnemi

Solution
Design pattern composite.





On a pas fait de “ConcreteComposite” car on pense que le jeu n’a pas besoin d’autre type de vague Composite et qu’un seul suffit pour le fonctionnement de notre jeu.

Pourquoi ?

L'intérêt est la possibilité d’exploiter plusieurs types de vague lors de notre jeu comme par exemple à chaque tour pair nous aurons des bandits, les impairs des vagues aléatoire puis tous les 5 tours des dragons.

b) Refactoring et ajout potentielle

1) Suppression du timer

Problématique	Sur
Sur la méthode creerVague auparavant un timer était utilisé pour générer une vague sauf que le timer n'est pas utile en soit car nous avons déjà un compteur sur le nombre de vagues et également sur le nombre de morts.	VagueEnnemi

Solution
Nous l'avons retiré du code car il n'était pas nécessaire et remplacé par la fonction de l'interface TypeVague qui génère les ennemis de notre vague.

Environnement :

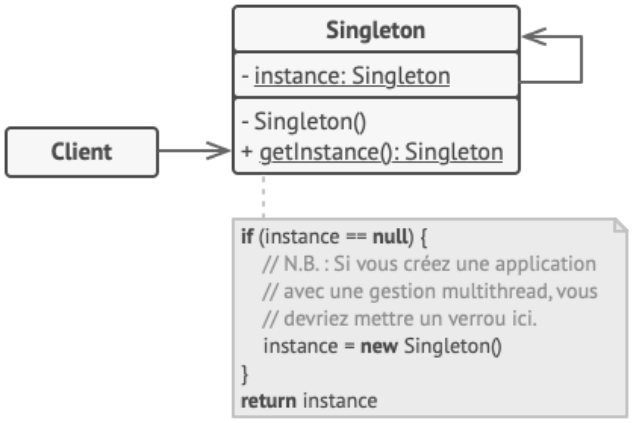
a) Design Pattern

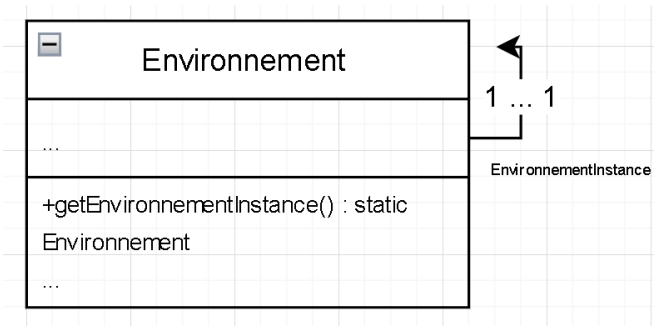
1) Avoir qu'une seule instance d'environnement

Problématique	Sur
On créer à chaque fois une nouvelle instance pour notre environnement, cela créer une sorte de redondance avec les "new".	La classe Environnement et tous les classes qui appellent l'objet Environnement.

Solution

Design pattern Singleton.





Pourquoi ?

On va utiliser un singleton car ça permet d'éviter certaines redondances de code en appelant environnement puis en le réinitialisant à chaque fois dans les différentes classes qui l'appel.

b) Refactoring et ajout potentielle

1) Ajout d'un compteur de mort

Problématique	Sur
On a besoin de compter nos ennemis tués lors d'une partie, sauf que rien ne permet de savoir combien d'ennemis nous avons tué.	La classe Environnement

Solution
Ajout d'un Observables integerProperty qui va permettre de compter le nombre d'ennemis tué et ça va permettre de pouvoir créer une vague en fonction du nombre d'ennemis vaincus.

2) Ajout d'une liste d'ennemi

Problématique	Sur
Après avoir changé la structure de nos acteurs et des vagues on avait un problème sur l'ajout des acteurs dans notre environnement, car le code initial utilise une liste générale qui en fonction des instances of permet l'ajout sur cette liste mais nous ne voulons plus en faire usage de ces "instance of".	La classe Environnement

Solution
Pour répondre à ce problème nous avons fait deux listes, une pour les acteurs alliés et l'autre pour les ennemis, ainsi nous avons plus besoin "d'instance of" et on a bien deux listes distinct.

Projectile :

a) Refactoring et ajout potentielle

1) Changement de package

Problématique	Sur
Le package Tours avait toutes les classes de Tour et Projectile.	Projectiles et tours

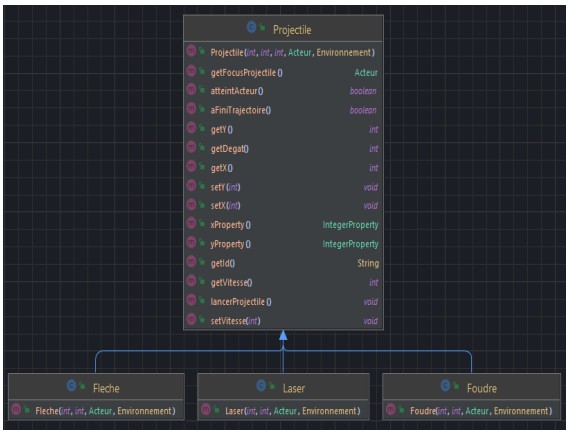
Solution
Création de deux nouveaux packages : Tours & Projectiles. Avant toutes nos classes ayant un rapport avec les projectiles étaient dans le package "Tours" désormais on les a séparés en deux packages : Tours et Projectile, celà apporte une meilleure compréhension du code.

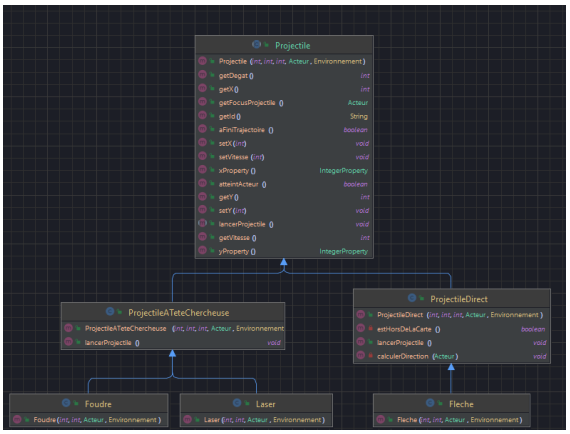
2)

Problématique	Sur
Tous nos projectiles sont à tête chercheuse.	Projectiles

Solution

Avant : **Après :**





-**Spécialisation** : Les classes “ProjectileDirect” et “ProjectileATeteChercheuse” permettent de spécialiser le comportement des projectiles sans dupliquer de code.

-**Réutilisation** : Les méthodes communes sont définies dans la super-classe “Projectile”, ce qui facilite la réutilisation du code et réduit la redondance.

-**Extensibilité** : Il est facile d'ajouter de nouveaux types de projectiles en créant de nouvelles sous-classes.

-**Maintenabilité** : Les modifications apportées à un type de projectile spécifique ne nécessitent des modifications que dans une classe spécifique, sans affecter les autres.

-**Abstraction** : La classe “Projectile” sert à définir de ce qu'un projectile doit être capable de faire, tandis que les sous-classes fournissent les détails d'implémentation.

-**Encapsulation** : Les détails de l'implémentation des différents types de projectiles sont cachés derrière leur classe respective, permettant de changer l'implémentation sans affecter les autres parties du code

3)

Problématique	Sur
Nous avons pas des types de projectiles diversifiés.	Projectiles et tours

Solution
Créer un autre type de Projectile. Avant tout nos projectiles étaient de type Projectiles malgrés qu’ils soient uniquement à tête chercheuse, maintenant on a un autre type de projectiles “ProjectileDirecte”, donc cette classe extend la classe Projectile.

4)

Problématique	Sur
Tous nos projectiles étaient des projectiles à tête chercheuse.	Projectile

Solution
Création d'un nouveau Type de projectile: projectileDirect La superclass Projectile avait trop de responsabilités, en plus d'apporter de la diversification dans notre jeu, désormais les Laser et la Foudre sont des projectilesATeteChercheuse et les flèches sont des projectile Directi

Refactoring général :

Nous avons retiré toutes les variables et méthodes non utilisées dans les différentes classes.