# rapport TP MOCI

BONANCIO SKORA, Lucas Eduardo

Décembre 2023

## 1 Introduction

Ce document explique le développement d'un prototype de jeux vidéo de rôle pour approfondir les connaissances sur les patrons de projet orientés objet dans le cadre de la discipline de «Méthode et Outils de Conception Informatique» dans le cadre du deuxième année du diplôme d'ingénieur Telecom Nancy.

Le projet a été développé à partir des classes couches en apportant des personnages basiques et, progressivement, des patrons de conception ont été utilisés pour ajouter des nouvelles fonctionnalités jusqu'au développement d'un prototype de jeu. La figure 1 montre le diagramme avec ces premières classes.

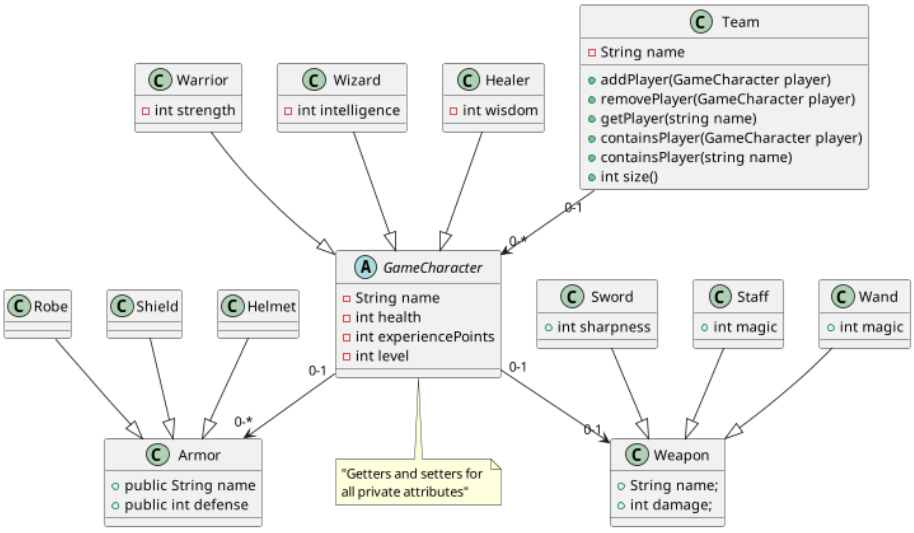


FIGURE 1 – Diagramme de classes initial

# 2 Patrons

## 2.1 Singleton

Le patron Singleton a été utilisé pour ajouter une classe centralisé de paramètres du jeu, appelée `GameConfiguration`. L'utilisation du patron (et non de une classe normal) est important pour garantir que le programme entière utilise les mêmes paramètres, et diminue le nombre d'objets utilisés.

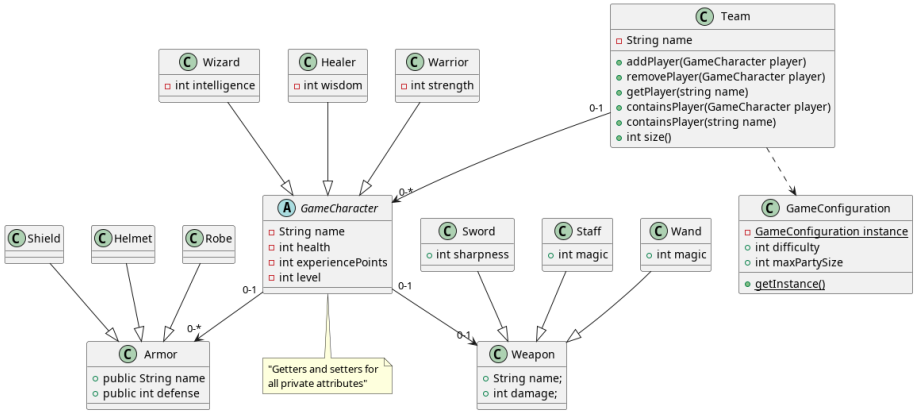


FIGURE 2 – Diagramme de classes après l'inclusion de Singleton

Au moment de l'inclusion, la seule classe que l'utilisait était `Team`, pour voir si l'inclusion d'un personnage respecte la taille maximale d'une equipe.

## 2.2 Factory Method

Le patron Factory Method a été utilisé pour simplifier la création de nouvelles personnages de chaque type avec l'interface `CharacterCreator` et ses implantations. Les paramètres utilisés sont le nome du personnage, niveau du personnage, et s'il porte une arme et/ou une armure. La détermination du nombre de points de expérience et santé et le type et qualité des équipements sont détermines par le niveau désiré du personnage.

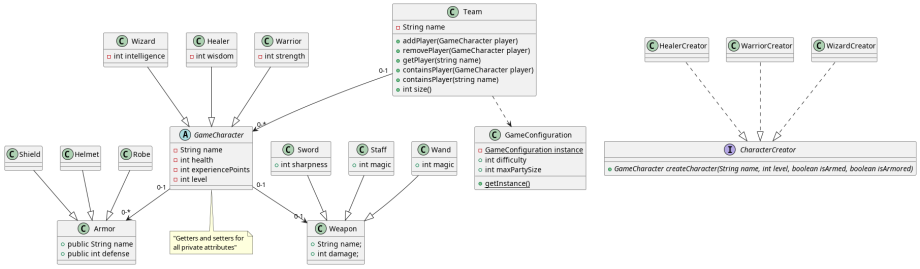


FIGURE 3 – Diagramme de classes après l'inclusion du Factory Method

## 2.3 Visitor

Le patron visitor a été utilisé pour ajouter des nouvelles opérations sur des personnages de jeu et ses Teams sans faire une classe plus lourde. Le patron visitor offre le meilleure rapport entre facilité d'ajout des nouvelles opérations et couplage du code. Au moment d'ajout de l'interface `CharacterVisitor`, les implantations étaient des visitors pour améliorer, soigner ou endommager les personnages.

Même si normalement le patron visitor est implanté a partir d'une interface, une classe abstraite a été utilisé pour pouvoir ajouter le méthode `visitTeam`, commun à tous spécialisations.

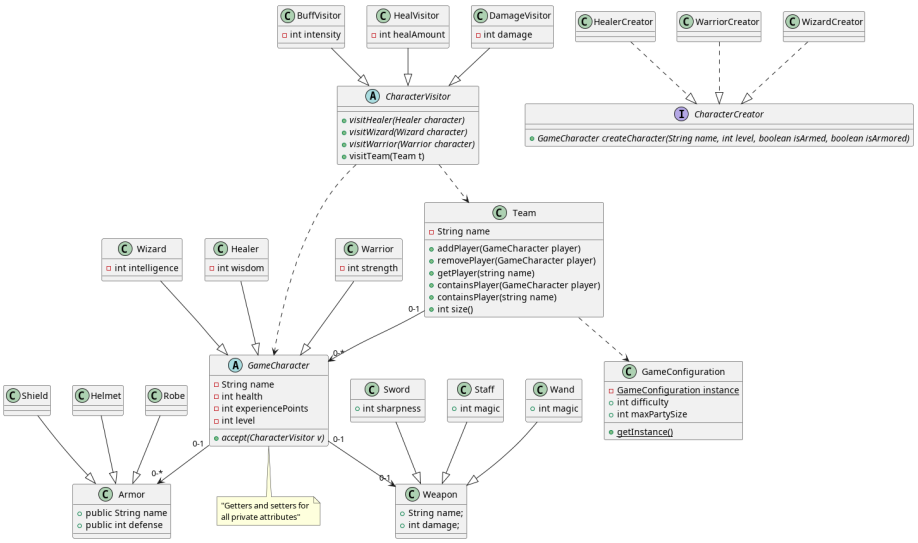


FIGURE 4 – Diagramme de classes après l'inclusion du Visitor

## 2.4 Strategy

Le patron Strategy a été utilisé pour varier le comportement des personnages selon les envies du jouer. Les stratégies possibles sont neutre, offensif (le personnage reçoit et donne damage double) et défensif (le personnage reçoit et donne la moitié de damage).

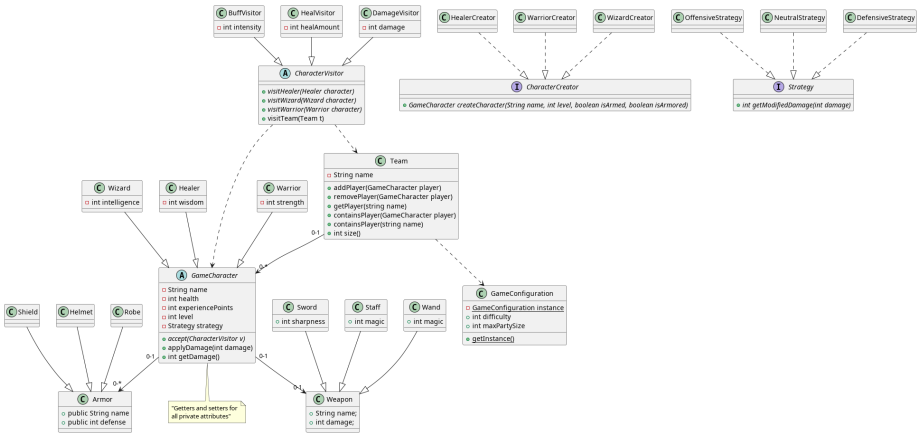


FIGURE 5 – Diagramme de classes après l'inclusion du patron Strategy

## 2.5 Observer

Le patron Observer a été utilisé pour permettre de centraliser certaines vérifications du jeu sans devoir les inclure directemment dans la classe `GameCharacter`. les objets qu'implémentent `GameCharacter` notifient ses *observers* quand certaines de ses attributs changent, notamment, le nombre de points de vie et le nombre de points d'expérience. Pour faire un modèle plus générique et réutilisable, la classe abstraite `Observable` et l'interface `Observer` ont été crée. Plus une fois, l'utilisation d'une classe abstraite en place d'interfaces a été fait pour permettre d'implémenter les méthodes que n'ont pas raison de changer entre implantations dans la portée de ce projet. Dans un projet plus grand et réel, l'utilisation d'une interface directement pourrait être mieux.

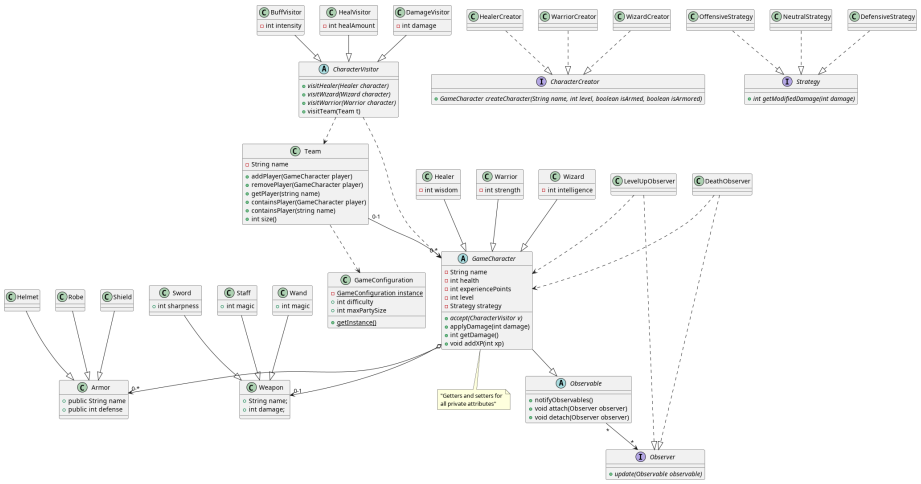


FIGURE 6 – Diagramme de classes après l'inclusion du patron Observer

## 2.6 Decorator

Le patron Decorator a été utiliser pour ajouter des nouvelles capacités ou actions aux personnages.

La consigne a suggéré de faire un `ArmoredDecorator` pour diminuer les dégâts faites à un personnage. Comment les classes d'armures étaient déjà faites pour suivre le diagramme de classes inclut dans la consigne, pour éviter la redondance, l'auteur a décidé de changer ça pour le `DualWielderDecorator`, qui permet a un personnage d'avoir une deuxième arme et attaquer avec les deux à chaque fois.

Pour bien implémenter ce patron (et le suivante), c'était claire que une interface séparé pour le caractères était nécessaire. `GameCharacter` devient une interface, et les implantations des méthodes dans cet classe ont été mises dans la classe abstraite `BaseCharacter`. Comme ça, `Decorator` peut implémenter l'interface. Ça assure que le patron est bien implémenté et que le Decorator peut vraiment remplacer un `GameCharacter`.

Comme la plupart des Decorator va changer peu de méthodes, la classe abstraite `Decorator` implémente tous les méthodes dans l'interface.

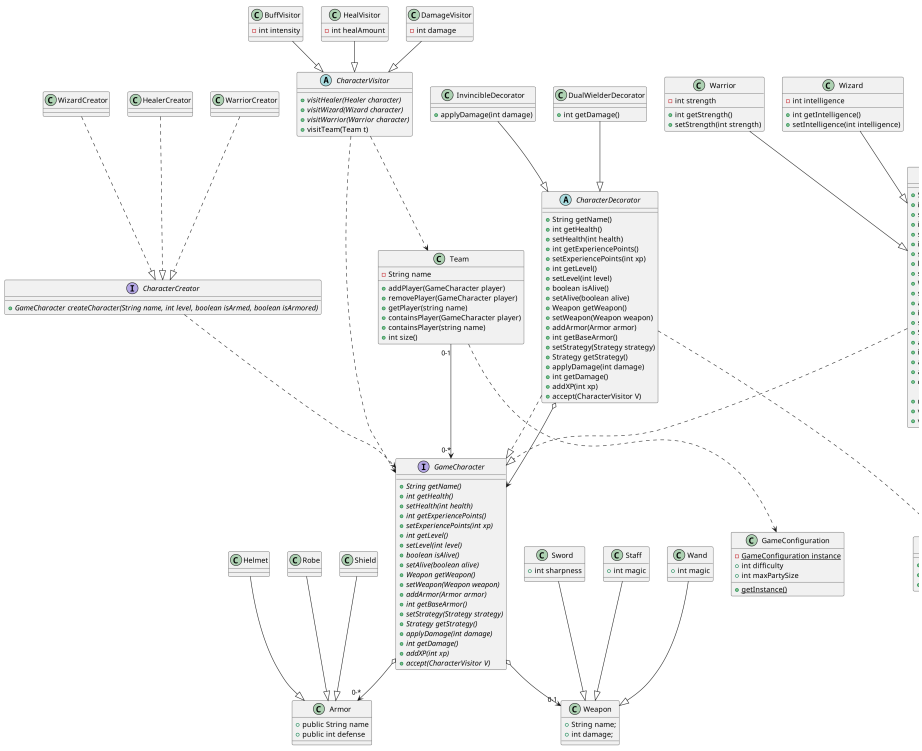


FIGURE 7 – Diagramme de classes après l'inclusion du patron Decorator

## 2.7 Composite

La présence de ce patron dans le sujet est curieux parce que, dans la partie du sujet pour le Visiteur, il demandait l'ajout d'une classe `Team`. Normalement, on implémentera une Composite avant d'un Visiteur, parce qu'une des avantages principaux de ses deux patrons est son synergie. à la fin, la seule chose qui est nécessaire pour implaniter une Composite est faire la classe `Team` implémenter l'interface `GameCharacter` et régler les visiteurs pour interpréter `Team` comme un type de `GameCharacter` :

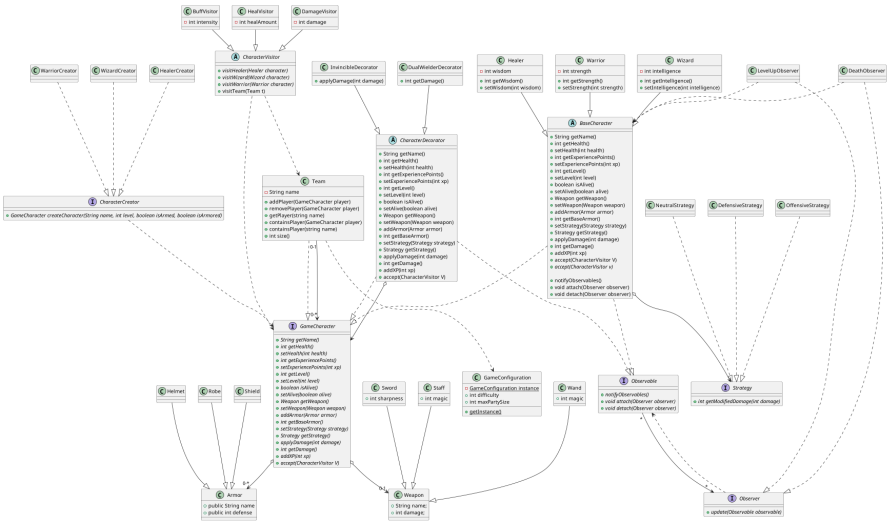


FIGURE 8 – Diagramme de classes après l'inclusion du patron composite

## 2.8 Command

Le patron de projet Command a été utilisé pour permettre d'accéder et manipuler les classes du jeu de façon centralisé. Pour faciliter ce but, une classe Facade pour le jeu a été aussi crée. La classe Main envoyé a la classe `GameInvoker`, qui l'exécutent.

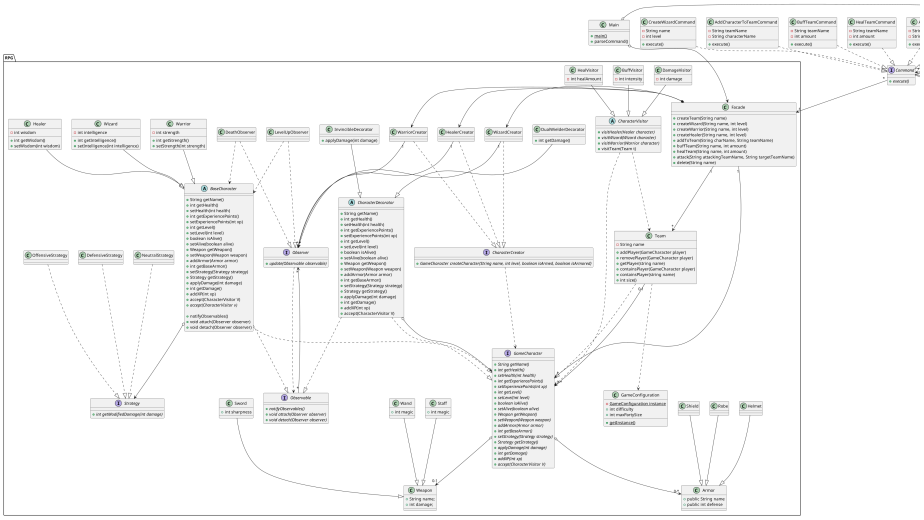

FIGURE 9 – Diagramme de classes après l'inclusion du patron command

# 3 Conclusion

Cet projet a été très intéressant pour mieux comprendre l'importance des patrons de projet et aussi les synergies et différences entre eux. Par exemple, *State* et *Strategy* sont des techniques semblables pour faire varier le comportement d'un Objet, et les patrons *Visitor* et *Composite* sont beaucoup plus interessantes si utilisés ensemble.