Rapport de projet
Sujet n°3

Anthony PHOTHIRATH
Kévin DANG

Sommaire :

I°) Architecture du code

II°) Fonctionnalité et répartition des tâches

III°) Perspective d’amélioration

I°) Architecture du code

Les principes SOLID

L'intégralité de l'architecture du code a été conçue en essayant de suivre le mieux possible les principes **SOLID** tout en continuant à intégrer au fur et à mesure de nouvelles fonctionnalités. Ces principes visent à rendre le code plus lisible, facile à maintenir, extensible, réutilisable et sans répétition. Nous allons maintenant voir avec quelques exemples comment ces principes ont été intégrés dans le code.

Responsabilité unique (Single responsibility principle)
Une *classe*, une *fonction* ou une *méthode* doit avoir une et une seule **responsabilité**.

Ouvert/fermé (Open/closed principle)
Une entité applicative (classe, fonction, module ...) doit être *fermée* à la **modification directe** mais *ouverte* à **l'extension**.
Exemple : La classe Personnage du projet, une fois fixé est fermé à la modification. Pour modifier son comportement il faut donc passer par l'héritage, ce qui est réalisé par les différentes classes de personnages.

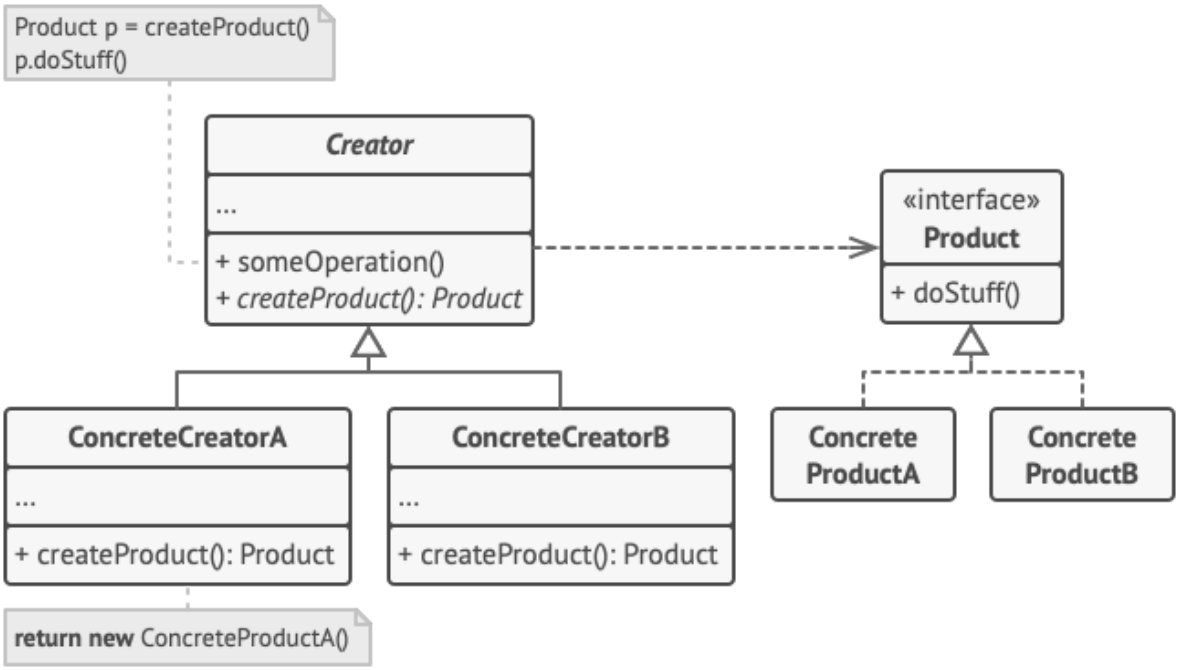
Substitution de Liskov (Liskov substitution principle)
Une *instance de type T* doit pouvoir être remplacée par une *instance de type G*, tel que *G sous-type de T*, sans que cela ne modifie **la cohérence du programme**.
Exemple: Ce *principe* est respecté dans le code via le *polymorphisme par sous-typage* impliqué par l'*héritage*. Comme les classes fille implémentent les classes mère, on peut remplacer une instance de Personnage par une instance de Guerrier par exemple sans que cela cause des problèmes.

Ségrégation des interfaces (Interface segregation principle)
Préférer plusieurs **interfaces** spécifiques pour chaque client plutôt qu'une seule interface générale
Exemple: Principe assez peu présent dans le code mais qui peut-être résumé par une bonne séparation du code en fonction des fonctionnalités avec les classes abstraites.

Inversion des dépendances (Dependency inversion principle)
Il faut dépendre des abstractions, pas des implémentations
Exemple: Nous ne manipulons dans le code pratiquement jamais les sous-types. Nous déclarons toujours les types abstraits, le type effectif sera ensuite déterminé.

Le design pattern Factory

Nous avons beaucoup utilisé le design pattern **factory** notamment pour la création des personnages et des objets.



L'*interface* est déclarée par le *Produit* et est commune à tous les objets qui peuvent être conçus par le *créateur* et ses *sous-classes*.

Les **Produits Concrets** sont différentes implémentations de l'interface **produit**.

La méthode **fabrique** est déclarée par la classe **Créateur** et retourne les **nouveaux produits**. Il est important que son type de retour concorde avec l'interface *produit*.

Nous avons rendu la méthode *fabrique* **abstraite** afin d'obliger ses *sous-classes* à implémenter *leur propre version* de la méthode mais nous aurions pu également modifier la méthode fabrique de la classe de base afin qu'elle retourne un type de produit par défaut.

Il faut bien comprendre que malgré son nom, la création de produits n'est pas la *responsabilité principale* du créateur. La classe créateur a en général déjà un fonctionnement propre lié à la nature de ses produits. La fabrique aide à **découpler** cette logique des *produits concrets*. C'est un peu comme une grande entreprise de développement de logiciels : elle peut posséder un département spécialisé dans la

formation des développeurs, mais son activité principale reste d'écrire du code, pas de produire des développeurs.

Les **Créateurs Concrets** redéfinissent la méthode *fabrique* de la classe de base afin de pouvoir retourner les *différents types de produits*.

Notez toutefois que la méthode fabrique n'est pas obligée de créer tout le temps de nouvelles instances. Elle peut retourner des objets depuis un cache, un réservoir d'objets ou une autre source.

Ainsi, nous **détachons** la création des objets de l'utilisation, ce qui permet d'éviter une certaine redondance au niveau de la programmation.

Nous pouvons voir aussi que le fait de passer par des classes filles pour créer différents objets permet de répondre au **principe Dependency Inversion Principle**, qui consiste à dire que les objets de forte valeur métier ne doivent pas dépendre des objets de faible valeur métier.

Le code qui appelle la méthode fabrique (souvent appelé le code client) ne fait pas la distinction entre les différents *produits concrets* retournés par les sous-classes, il les considère tous comme des **Personnages** (ou Objets) **abstraits**. Le client sait que tous les **Personnages** sont censés avoir une telle méthode, mais son fonctionnement lui importe peu.

Appliquer ce *design pattern* sur les personnages nous a vraiment permis d'isoler la **construction** des sous-type de Personnage (ou Objet) dont les *constructeurs* sont par ailleurs *privés* afin d'éviter toutes mauvaises instanciations et donc de contrôler beaucoup plus finement la création de nos **Personnages**, notamment *l'initialisation de leurs statistiques* par exemple.

II°) Fonctionnalité et répartition des tâches

Kévin	Anthony
• Architecture du code • Factory des personnages • Factory des objets • Routine de la partie • Interaction hors combat (déplacement) • Création des premiers mobs • Nettoyage de l'interface	• Système et interaction en combat (sort, status, équilibrage, action etc) • Système de gestion des objets et d'équipement (effet, drop, autoloot etc) • Carte et pièce du jeu • Ajout de mobs supplémentaires

Parmi les méthodes de travail utilisées, nous avons notamment sollicité le peer programming et le code review.

III°) Perspective d’amélioration

Nous avons pu rendre dans les temps, un produit considéré comme fini et prêt à l'emploi mais il existe encore beaucoup de points à améliorer.

Tout d’abord sur la fin nous avons pu ajouter beaucoup d’extension à notre projet, ce qui a été facilité par l’architecture du code mais cela a été fait au détriment du maintien d’un code propre (responsabilité des classes ou méthodes notamment), un temps de **refactorisation** après l’ajout de chaque extension aurait été de mise ce que nous n’avons pas fait par manque de temps.

Egalement nous sommes déçus de ne pas avoir pu implémenter une fonctionnalité de sauvegarde dont nous avons une ébauche de conception, notamment le fichier qui serait sauvegardé puis parsé afin de retrouver toutes les données mais la présence de nouvelles extensions rendait illisible et compliqué le fichier à parser.

Enfin la présence de **test unitaire** n’aurait pas été de trop mais pour cela il faut un couplage faible des composants de l’application via l’inversion de contrôle/l’injection de dépendance ce qui n’a pas été totalement parfaitement réussi dans le projet.