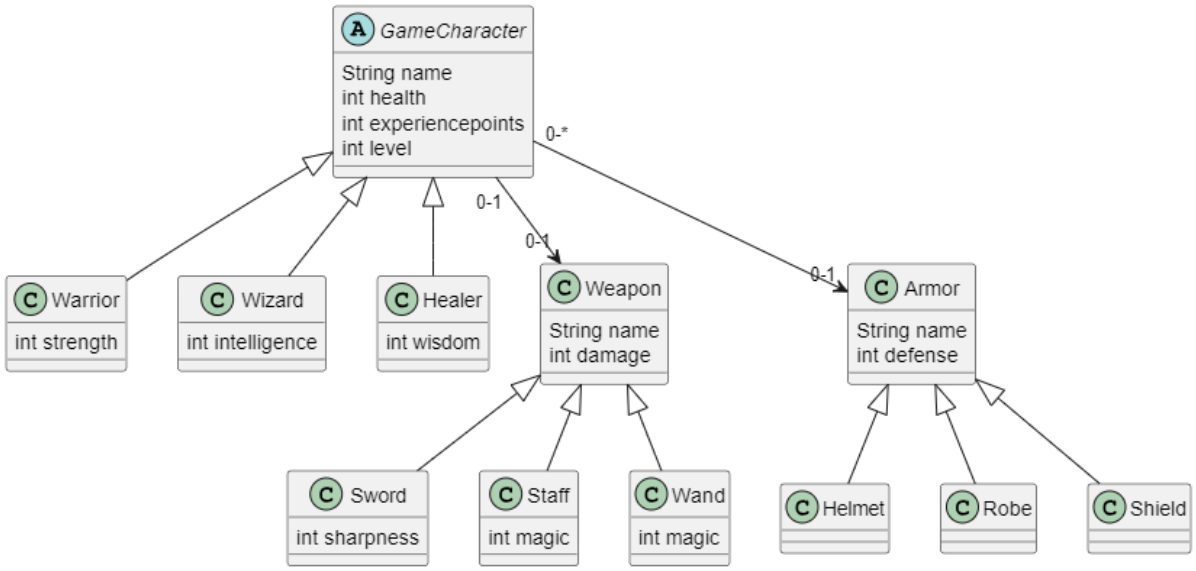
Sujet de TP : Construction d'un jeu de rôle simple avec design patterns

Contexte : Vous êtes chargé de développer un prototype pour un jeu de rôle (RPG). Votre objectif est d'implémenter un système modulaire qui permet de gérer les personnages, leurs interactions, et les différents éléments de gameplay de manière flexible.

But du TP : Comprendre et mettre en œuvre divers design patterns pour résoudre des problèmes de conception spécifiques dans le développement d'un jeu.

Pour montrer le fonctionnement de vos patterns, vous développerez des tests unitaires en utilisant la librairie Junit.



Partie 1: Mise en place des personnages et interactions de base

Singleton

- Créez un `GameConfiguration` en utilisant le pattern Singleton pour gérer les paramètres globaux du jeu. Le premier paramètre est le niveau de difficulté qui pourra être utilisé pour influencer la force des ennemis générés et leur nombre. Un autre paramètre est la taille maximum d'une équipe.

Factory Method

- Créez une interface `CharacterCreator` avec une méthode `createCharacter()`.
- Implémentez différentes "factories" pour chaque type de personnage chacune avec sa propre logique de création.
- Chaque créateur concret, tel que `WarriorCreator` ou `WizardCreator`, implémente `createCharacter` pour retourner une nouvelle instance de leur type spécifique de personnage.

Visiteur

- Développez des `CharacterVisitor` pour effectuer des opérations sur des ensembles de personnages, comme les `BuffVisitor`, `DamageVisitor`, et `HealVisitor`. BuffVisitor augmente la caractéristique du personnage en fonction de son type. Même chose pour Damage et Heal.
- Assurez-vous de tester les visiteurs sur les personnages pour vérifier les interactions. Vous pouvez mettre en place différents scénarios de visites consécutives.

Partie 2: Gestion avancée des personnages

Strategy

- Chaque personnage doit pouvoir adopter différentes stratégies de combat (`AggressiveStrategy`, `NeutralStrategy`, `DefensiveStrategy`).
- Une attaque d'un personnage en mode agressif fera plus de dégats mais celui-ci subira plus de dommage en cas d'attaque.
- Une attaque d'un personnage en mode défensif fera moins de dégats mais celui-ci subira moins de dommages en cas d'attaque.
- Le comportement de combat peut être changé dynamiquement pendant le jeu. Implantez des tests montrant des changements de stratégie

Observer

- Mettez en œuvre un système d'observation où les personnages notifient des observateurs (`LevelUpObserver`, `DeathObserver`) lors des changements d'état.
- Implémentez la logique pour que les personnages puissent notifier les observateurs et que les observateurs puissent s'abonner.

Partie 3: Extension et amélioration du jeu

Decorator

- Utilisez le pattern Decorator pour ajouter des améliorations aux personnages, comme `ArmoredDecorator` (réduit les dégats) ou `InvicibleDecorator` (`Maintient les hp au dessus de 0`).
- Les décorateurs doivent modifier les attributs des personnages ou ajouter de nouvelles méthodes.

Composite

- (Bonus) Implémentez le pattern Composite pour créer des structures d'équipe où les personnages peuvent former des groupes pour combiner leurs forces.
- Le groupe de personnages doit pouvoir agir comme un seul personnage pour certaines actions.

Partie 4: Gestion des actions du jeu avec le pattern Command

Objectif : Intégrer le pattern Command pour gérer les actions des personnages de manière flexible et extensible. Avant ça vous devrez faire une facade pour le jeu qui fournir les opérations permettant d'ajouter une compagnie (Fellowship), d'en supprimer une, et de produire des attaques entre compagnies (1v1), de soigner une compagnie (Heal), d'améliorer une compagnie (Buff)

Command

- Définissez une interface `Command` avec une méthode `execute()` et une méthode `undo()` si vous souhaitez pouvoir annuler les actions.
- Créez des commandes concrètes pour différentes actions

Invoker

- Créez une classe `GameInvoker` qui est responsable de l'exécution des commandes. Elle peut également conserver un historique des commandes pour les annuler si nécessaire.
- La classe `GameInvoker` doit être capable de mettre en file d'attente plusieurs commandes et de les exécuter en séquence.

Client

- Le client, dans ce contexte, pourrait être l'interface utilisateur ou le système de contrôle qui crée des objets Command en fonction des entrées du joueur.

- Le client doit associer les commandes aux boutons ou aux actions du joueur.

Partie 5 : Exemple de Pattern State pour un RPG

Objectif : Implémenter des états internes qui changent le comportement du personnage.

StateInterface

- L'interface `State` définiit les méthodes qui changent en fonction de l'état, telles que `onEnterState`, `onUpdate`, `onTryToMove`, ou `onAttack`.

ConcreteState

- Chaque état concret, tel que `DeadState`, `WoundedState`, ou `ScaredState`, implémente `State`.
- Les transitions d'état sont déclenchées par des événements internes.

ContextClass

- La classe `GameCharacter` propose des méthodes qui délèguent au `State` actuel. Cependant, les transitions d'état sont gérées à l'intérieur des objets `State` eux-mêmes, possiblement par des événements du jeu.

Livrables :

- Un rapport expliquant chaque pattern utilisé, son rôle dans l'architecture du jeu, et comment il résout un problème de conception.
- Diagrammes UML montrant la conception des classes et l'interaction entre les patterns.
- Code source complet commenté.
- Tests unitaires pour chaque fonctionnalité.
- (Optionnel) Une interface utilisateur simple pour interagir avec le système de jeu.

Critères d'évaluation :

- Correcte mise en œuvre des patterns de conception.
- Qualité et clarté du code.
- Présence et pertinence des tests unitaires.
- Qualité du rapport et des diagrammes UML.
- (Bonus) Originalité et complexité des interactions dans le jeu.