Exercice de développement logiciel en C# : Système de combat RPG (inspiré D&D & Final Fantasy)

Objectifs pédagogiques

Cet exercice, réalisé en **binôme**, vise à :

- Mettre en pratique les principes de la **programmation orientée objet** (encapsulation, héritage, polymorphisme).
- Appliquer les **bonnes pratiques du code propre** (nommage, lisibilité, simplicité, absence de duplication).
- Découvrir et utiliser les **principes SOLID**.
- Travailler en équipe avec **Git** (gestion de versions, branches, pull requests).
- Écrire des **tests unitaires** pour valider le comportement du code.
- Développer une **réflexion sur la modélisation** et l'**originalité des solutions**.

Énoncé du projet

Vous devez développer une application **console en C#** (**.NET 6+**) simulant un **système de combat au tour par tour** dans l'univers d'un jeu de rôle (inspiré de Donjons & Dragons pour la création de personnages et de Final Fantasy pour le système de combat).

L'application devra offrir les fonctionnalités suivantes :

1. Création de personnages joueurs et de monstres

Personnage joueur (héros) :

- Attributs : **nom**, **classe** (Guerrier, Mage, Voleur, Clerc, etc.), **niveau** (démarre à 1), **points de vie (PV)**, **points de mana (PM)** , **force**, **intelligence**, **agilité**, **défense**, **résistance magique**.
- Chaque classe possède des **caractéristiques de base** et une **progression** (par niveau).
- Un personnage peut apprendre des **compétences / sorts** (ex: Attaque, Boule de feu, Soin, Coup critique).
- Il peut porter de l'**équipement** (arme, armure, accessoire) qui modifie ses statistiques.

Monstre :

- Attributs similaires simplifiés (nom, PV, force, défense, etc.).
- Peut avoir des comportements d'IA simples (attaques aléatoires ou ciblées).
- Plusieurs types de monstres (Gobelin, Dragon, etc.) avec des caractéristiques différentes.

2. Gestion des compétences et des sorts

- Une compétence a un **nom**, un **coût en PM** (éventuellement 0 pour les attaques de base), une **puissance**, un **type** (physique / magique) et éventuellement une **portée** ou une **cible** (un ennemi, tous les ennemis, un allié).
- Les compétences peuvent avoir des **effets secondaires** (statut : poison, paralysie, etc.).
- Les sorts sont une sous-catégorie de compétences avec un coût en mana.

3. Système de combat au tour par tour

- Le combat oppose un **groupe de héros** (2 à 4 personnages) contre un **groupe de monstres** (1 à 4).
- Déroulement d'un tour :

  1. Calcul de l'**ordre d'action** (vitesse/agilité).
  2. Chaque combattant effectue une action (attaquer, utiliser une compétence, utiliser un objet, défendre, etc.).

- Lorsqu'un personnage attaque avec son arme, les dégâts sont calculés selon la formule : `(force + bonus arme) - défense ennemie` (ou variante avec aléatoire).
- Les sorts magiques utilisent l'intelligence et la résistance magique.
- Un personnage ne peut pas agir s'il est mort (PV 0).
- Le combat se termine quand tous les héros ou tous les monstres sont vaincus.

4. Gestion de l'inventaire et des objets utilisables en combat

- Objets : potion de soin (rend PV), potion de mana (rend PM), antidote, etc.
- Un héros peut utiliser un objet sur lui-même ou sur un allié pendant son tour.
- L'inventaire est commun au groupe (partagé) ou individuel (au choix).

5. Persistance et rejouabilité

- Les données (personnages, monstres, équipement, objets) sont conservées **en mémoire** pour le moment.
- Vous devez concevoir une **abstraction** pour le chargement des données (interface `IPersonnageRepository`, `IMonstreRepository`, `IEquipementRepository`…) afin de pouvoir changer de source facilement (fichier JSON, base de données) dans le futur.

6. Interface utilisateur (console)

- L'utilisateur doit pouvoir :
  - Créer une équipe de héros (choix parmi des classes prédéfinies).
  - Lancer un combat contre un groupe de monstres généré aléatoirement.
  - Visualiser le déroulement du combat (messages décrivant les actions, les dégâts, les morts).
  - À la fin du combat, afficher le résultat (victoire / défaite) et les statistiques (dégâts infligés, soins prodigués, etc.).

Contraintes techniques et de conception

- Utilisation exclusive de **C#** et du **framework .NET** (version récente).
- Le projet doit être organisé en plusieurs **fichiers/classes** selon les responsabilités.
- Vous devez appliquer les **principes SOLID** :
  - **S** : chaque classe a une responsabilité unique.
  - **O** : le code est ouvert à l'extension mais fermé à la modification (utilisez des interfaces, des classes abstraites).
  - **L** : les sous-classes doivent pouvoir se substituer à leurs classes de base.
  - **I** : préférez des interfaces spécifiques plutôt qu'une interface générale.
  - **D** : les modules de haut niveau ne dépendent pas des modules de bas niveau ; utilisez l'injection de dépendances.
- Code **propre** :
  - Noms significatifs (classes, méthodes, variables).
  - Méthodes courtes (idéalement moins de 20 lignes).
  - Pas de code dupliqué (principe DRY).
  - Commentaires uniquement pour expliquer le « pourquoi » pas le « comment ».
- Écrivez des **tests unitaires** (xUnit, NUnit ou MSTest) pour couvrir les cas principaux de la logique métier : calcul des dégâts, application des effets, déroulement d'un tour, etc. Visez une couverture raisonnable (au moins 50%).
- Utilisez **Git** dès le début du projet (dépôt sur GitHub, GitLab ou Bitbucket). Faites des **commits réguliers** avec des messages explicites. Utilisez des **branches** pour développer chaque fonctionnalité et faites des **pull requests** entre vous.

Consignes de travail en équipe

- **Formez des binômes**. Vous pouvez vous organiser comme vous le souhaitez (pair programming, répartition des tâches).
- Définissez ensemble une **modélisation UML** simplifiée (diagramme de classes) avant de commencer à coder. Cela vous aidera à répartir le travail.
- L'utilisation de Git est obligatoire :
  - Un seul dépôt pour le binôme.
  - Chacun travaille sur sa propre branche.
  - Fusion via pull requests après relecture par l'autre membre.
  - Les commits doivent être atomiques et le message doit décrire le changement.
- **Rendu** : fournissez le lien du dépôt Git ainsi qu'un **rapport de quelques pages** (format PDF ou Markdown) qui présente :
  - Les choix de modélisation (diagramme de classes, explication des patterns utilisés).
  - Les idées originales que vous avez développées pour répondre aux problématiques (gestion des compétences, IA des monstres, système d'équipement, etc.). L'originalité sera valorisée.
  - Les difficultés rencontrées et comment vous les avez résolues.
  - La répartition du travail au sein du binôme et votre expérience de collaboration.
- **Soutenance orale** : en fin de projet, chaque binôme participera à une évaluation orale individuelle ou en groupe. Vous serez interrogés sur :
  - Le code que vous avez produit (explication de certaines parties, justification des choix techniques).
  - La théorie du cours associée (principes POO, SOLID, clean code, tests, gestion de versions).
  - Votre capacité à répondre à des questions de modification ou d'extension du projet.

Critères d'évaluation

Critère	Détails	Pondération
Fonctionnalités	Toutes les fonctionnalités demandées sont présentes et fonctionnent.	20%
Conception POO	Utilisation pertinente de l'héritage, polymorphisme, encapsulation. Cohérence du modèle.	25%
Principes SOLID / Clean code	Respect des 5 principes, lisibilité, absence de duplication, noms explicites, méthodes courtes.	25%
Tests unitaires	Présence de tests, pertinence, succès.	10%
Collaboration Git	Commits fréquents, messages clairs, utilisation de branches, pull requests, travail en équipe visible.	10%
Rapport	Qualité, clarté, profondeur de l'analyse, originalité des solutions proposées.	5%
Soutenance orale	Compréhension du code, capacité à argumenter les choix, maîtrise des concepts théoriques.	5%

Suggestions pour réussir

1. **Commencez simple** : implémentez d'abord le cœur du métier (classes `Personnage`, `Monstre`, `Competence`, `Equipement`) sans interface utilisateur. Validez avec des tests unitaires.

2. **Héritage vs composition** : une bonne approche est d'utiliser la **composition** pour les caractéristiques (`Stats`) et l'**héritage** pour les types de personnages (`Guerrier`, `Mage`). Mais respectez le principe **L** (Liskov) et préférez la composition si l'héritage devient trop rigide.
3. **Pattern Stratégie** : pour les comportements d'attaque, d'IA des monstres, de calcul de dégâts. Cela rendra le code facilement extensible.
4. **Pattern Fabrique** : pour créer des personnages, des monstres, des équipements à partir de données prédéfinies.
5. **Pattern Observateur** : pour les notifications (affichage des actions de combat) sans coupler la logique métier à la console. Une interface `IActionLogger` avec une implémentation `ConsoleLogger` peut suffire.
6. **Injection de dépendances** : construisez votre application en injectant les dépendances (repositories, logger, générateur de monstres) dans le constructeur des classes qui en ont besoin (ex: `CombatService`).
7. **Séparez la persistance** : créez des interfaces `IPersonnageRepository`, `IMonstreRepository`, `IEquipementRepository`. Une implémentation `MemoirePersonnageRepository` stocke les données dans des `List<T>`. Vous pourrez plus tard les charger depuis un fichier JSON.
8. **Tests unitaires** : testez le calcul des dégâts, l'application des soins, les états (poison, etc.), l'ordre d'initiative, la fin de combat. Utilisez des mocks pour simuler le `Random` si nécessaire.
9. **Évitez les classes Dieu** : ne faites pas tout dans `Program.cs`. Créez des services (`CombatService`, `PersonnageService`, `InventaireService`) qui orchestrent la logique.
10. **Interface utilisateur** : créez une classe `CombatUI` qui utilise les services et affiche les messages. Elle ne doit contenir **aucune règle métier**.