TP7 : Principe Ouvert/Fermé (OCP)

Objectif

Comprendre et appliquer le principe Ouvert/Fermé (Open/Closed Principle), le deuxième des principes SOLID.

Rappel du Principe

"Les entités logicielles (classes, modules, fonctions) devraient être ouvertes à l'extension mais fermées à la modification."

Ce principe stipule que vous devriez pouvoir étendre le comportement d'une classe sans la modifier. En d'autres termes, le code existant ne doit pas être modifié lorsque des nouvelles fonctionnalités sont ajoutées.

Problème

Le code suivant représente un système de dégâts pour un jeu. Le système doit calculer les dégâts infligés à différents types d'ennemis, en prenant en compte leur type (terrestre, volant, aquatique) et leurs résistances/faiblesses.

```
// DamageCalculator.cs - Code problématique
public class DamageCalculator
{
    public enum DamageType
    {
        Physical,
        Fire,
        Ice,
        Lightning,
        Poison
    }

    public enum EnemyType
    {
        Ground,
        Flying,
        Aquatic
    }

    // Calcule les dégâts en fonction du type de dégâts et du type d'ennemi
    public float CalculateDamage(float baseDamage, DamageType damageType, Enemy
enemy)
    {
        float finalDamage = baseDamage;

        // Ajuster les dégâts en fonction du type d'ennemi
        if (enemy.Type == EnemyType.Ground)
        {
            // Les ennemis terrestres ont des modificateurs spécifiques
```

```
            switch (damageType)
            {
                case DamageType.Physical:
                    finalDamage *= 1.0f;  // Dégâts physiques normaux
                    break;
                case DamageType.Fire:
                    finalDamage *= 1.2f;  // Vulnérables au feu
                    break;
                case DamageType.Ice:
                    finalDamage *= 0.8f;  // Résistants au froid
                    break;
                case DamageType.Lightning:
                    finalDamage *= 1.1f;  // Légèrement vulnérables à
l'électricité
                    break;
                case DamageType.Poison:
                    finalDamage *= 1.3f;  // Très vulnérables au poison
                    break;
            }
        }
        else if (enemy.Type == EnemyType.Flying)
        {
            // Les ennemis volants ont des modificateurs spécifiques
            switch (damageType)
            {
                case DamageType.Physical:
                    finalDamage *= 0.8f;  // Résistants aux dégâts physiques
                    break;
                case DamageType.Fire:
                    finalDamage *= 1.0f;  // Dégâts de feu normaux
                    break;
                case DamageType.Ice:
                    finalDamage *= 1.5f;  // Très vulnérables au froid
                    break;
                case DamageType.Lightning:
                    finalDamage *= 2.0f;  // Extrêmement vulnérables à
l'électricité
                    break;
                case DamageType.Poison:
                    finalDamage *= 0.7f;  // Résistants au poison
                    break;
            }
        }
        else if (enemy.Type == EnemyType.Aquatic)
        {
            // Les ennemis aquatiques ont des modificateurs spécifiques
            switch (damageType)
            {
                case DamageType.Physical:
                    finalDamage *= 1.0f;  // Dégâts physiques normaux
                    break;
                case DamageType.Fire:
                    finalDamage *= 0.5f;  // Très résistants au feu
                    break;
```

```
                case DamageType.Ice:
                    finalDamage *= 1.0f;  // Dégâts de froid normaux
                    break;
                case DamageType.Lightning:
                    finalDamage *= 1.8f;  // Très vulnérables à l'électricité
                    break;
                case DamageType.Poison:
                    finalDamage *= 1.2f;  // Légèrement vulnérables au poison
                    break;
            }
        }

        // Appliquer d'autres modificateurs spécifiques à l'ennemi
        finalDamage *= enemy.DamageResistance;

        return finalDamage;
    }
}

// Classe de base pour les ennemis
public class Enemy
{
    public DamageCalculator.EnemyType Type { get; protected set; }
    public float Health { get; protected set; }
    public float DamageResistance { get; protected set; } = 1.0f;

    public Enemy(float health, DamageCalculator.EnemyType type, float
damageResistance = 1.0f)
    {
        Health = health;
        Type = type;
        DamageResistance = damageResistance;
    }

    public virtual void TakeDamage(float amount, DamageCalculator.DamageType
damageType)
    {
        // Utiliser le calculateur pour ajuster les dégâts
        DamageCalculator calculator = new DamageCalculator();
        float finalDamage = calculator.CalculateDamage(amount, damageType, this);

        Health -= finalDamage;

        if (Health <= 0)
        {
            Die();
        }
    }

    protected virtual void Die()
    {
        // Logique de mort de l'ennemi...
    }
}
```

```
// Exemple d'utilisateur du système
public class GameManager
{
    public void AttackEnemy(Enemy enemy, float attackPower,
DamageCalculator.DamageType damageType)
    {
        enemy.TakeDamage(attackPower, damageType);
    }
}
```

Problèmes avec ce code

1. Si vous devez ajouter un nouveau type de dégâts (comme `Holy` ou `Dark`), vous devez modifier la classe `DamageCalculator`
2. Si vous devez ajouter un nouveau type d'ennemi (comme `Undead`), vous devez également modifier la classe `DamageCalculator`
3. La méthode `CalculateDamage` est trop complexe et contient trop de conditions imbriquées
4. Le code n'est pas facilement testable car la logique est dispersée
5. Les modifications requises pour de nouvelles fonctionnalités risquent d'introduire des bugs dans un code fonctionnel

Exercice

Refactorisez le code en appliquant le principe Ouvert/Fermé :

1. Créez une structure de classes qui permette d'ajouter facilement de nouveaux types de dégâts
2. Créez une structure de classes qui permette d'ajouter facilement de nouveaux types d'ennemis
3. Assurez-vous que l'ajout de nouveaux types ne nécessite pas de modifier les classes existantes
4. Assurez-vous que le code est plus facile à tester

Avantages attendus

- Pouvoir ajouter de nouveaux types de dégâts sans modifier le code existant
- Pouvoir ajouter de nouveaux types d'ennemis sans modifier le code existant
- Un code plus modulaire et plus facile à maintenir
- Des responsabilités mieux réparties entre les classes

Conseils

- Utilisez des interfaces ou des classes abstraites pour définir des contrats
- Pensez à utiliser le pattern Stratégie ou un autre pattern de conception approprié
- Utilisez le polymorphisme pour éviter les conditions switch/case