TP10 : Principe de Ségrégation des Interfaces (ISP)

Objectif

Comprendre et appliquer le principe de Ségrégation des Interfaces (Interface Segregation Principle), le quatrième des principes SOLID.

Rappel du Principe

"Les clients ne devraient pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas."

Ce principe stipule qu'une classe ne devrait pas être forcée d'implémenter des méthodes dont elle n'a pas besoin. Les interfaces volumineuses doivent être divisées en interfaces plus petites et plus spécifiques.

Problème

Le code suivant représente un système de gestion de personnages pour un jeu. Il utilise une interface monolithique `IGameCharacter` qui contient des méthodes pour tous les types de personnages, ce qui force certaines classes à implémenter des méthodes qui ne les concernent pas.

```
// IGameCharacter.cs - Interface monolithique problématique
public interface IGameCharacter
{
    // Propriétés communes
    string Name { get; }
    int Health { get; set; }
    Vector3 Position { get; set; }
    bool IsAlive { get; }

    // Méthodes de mouvement
    void Move(Vector3 direction);
    void Jump();
    void Crouch();
    void Swim();
    void Climb();
    void Fly();

    // Méthodes de combat
    void Attack(IGameCharacter target);
    void Defend();
    void UseSpecialAbility();
    void CastSpell(string spellName, IGameCharacter target);
    void Heal(IGameCharacter target, int amount);

    // Méthodes d'interaction
    void PickupItem(IItem item);
    void UseItem(IItem item);
    void Speak(string dialogue);
    void Trade(IGameCharacter trader);
```

```
    // Méthodes pour les personnages non-joueurs (PNJ)
    void SetPatrolPath(List<Vector3> path);
    void FollowPlayer(IGameCharacter player);
    void RespondToInteraction(IGameCharacter interactor);
}

// Vector3.cs - Structure simple pour les positions
public struct Vector3
{
    public float X { get; set; }
    public float Y { get; set; }
    public float Z { get; set; }

    public Vector3(float x, float y, float z)
    {
        X = x;
        Y = y;
        Z = z;
    }
}

// IItem.cs - Interface simple pour les objets
public interface IItem
{
    string Name { get; }
    void Use(IGameCharacter character);
}

// Player.cs - Personnage jouable
public class Player : IGameCharacter
{
    public string Name { get; private set; }
    public int Health { get; set; }
    public Vector3 Position { get; set; }
    public bool IsAlive => Health > 0;

    // Inventaire et autres propriétés spécifiques au joueur
    private List<IItem> inventory = new List<IItem>();

    public Player(string name, int health, Vector3 position)
    {
        Name = name;
        Health = health;
        Position = position;
    }

    // Implémentations des méthodes de l'interface

    // Méthodes de mouvement - Utilisées par le joueur
    public void Move(Vector3 direction) { /* Implémentation */ }
    public void Jump() { /* Implémentation */ }
    public void Crouch() { /* Implémentation */ }
    public void Swim() { /* Implémentation */ }
```

```
    public void Climb() { /* Implémentation */ }
    public void Fly() { /* Implémentation */ }

    // Méthodes de combat - Utilisées par le joueur
    public void Attack(IGameCharacter target) { /* Implémentation */ }
    public void Defend() { /* Implémentation */ }
    public void UseSpecialAbility() { /* Implémentation */ }
    public void CastSpell(string spellName, IGameCharacter target) { /*
Implémentation */ }
    public void Heal(IGameCharacter target, int amount) { /* Implémentation */ }

    // Méthodes d'interaction - Utilisées par le joueur
    public void PickupItem(IItem item) { inventory.Add(item); }
    public void UseItem(IItem item) { item.Use(this); }
    public void Speak(string dialogue) { Console.WriteLine($"{Name} says:
{dialogue}"); }
    public void Trade(IGameCharacter trader) { /* Implémentation */ }

    // Méthodes pour les PNJ - Non utilisées par le joueur, implémentations vides
ou levant des exceptions
    public void SetPatrolPath(List<Vector3> path) { throw new
NotImplementedException("Players don't patrol"); }
    public void FollowPlayer(IGameCharacter player) { throw new
NotImplementedException("Players don't follow other players automatically"); }
    public void RespondToInteraction(IGameCharacter interactor) { throw new
NotImplementedException("Players respond manually, not automatically"); }
}

// Enemy.cs - Personnage ennemi (PNJ)
public class Enemy : IGameCharacter
{
    public string Name { get; private set; }
    public int Health { get; set; }
    public Vector3 Position { get; set; }
    public bool IsAlive => Health > 0;

    private List<Vector3> patrolPath;

    public Enemy(string name, int health, Vector3 position)
    {
        Name = name;
        Health = health;
        Position = position;
    }

    // Méthodes de mouvement - Partiellement utilisées
    public void Move(Vector3 direction) { /* Implémentation */ }
    public void Jump() { /* Implémentation minimale */ }
    public void Crouch() { /* Implémentation minimale ou vide */ }
    public void Swim() { /* Peut-être non implémenté pour certains ennemis */
throw new NotImplementedException(); }
    public void Climb() { /* Peut-être non implémenté pour certains ennemis */
throw new NotImplementedException(); }
    public void Fly() { /* Seulement pour les ennemis volants */ throw new
```

```
NotImplementedException("This enemy cannot fly"); }

    // Méthodes de combat - Utilisées
    public void Attack(IGameCharacter target) { /* Implémentation */ }
    public void Defend() { /* Implémentation */ }
    public void UseSpecialAbility() { /* Implémentation */ }
    public void CastSpell(string spellName, IGameCharacter target) { /* Pourrait
ne pas être implémenté pour les ennemis non-magiques */ }
    public void Heal(IGameCharacter target, int amount) { /* Implémentation */ }

    // Méthodes d'interaction - Partiellement utilisées
    public void PickupItem(IItem item) { /* Généralement non implémenté pour les
ennemis */ throw new NotImplementedException(); }
    public void UseItem(IItem item) { /* Généralement non implémenté pour les
ennemis */ throw new NotImplementedException(); }
    public void Speak(string dialogue) { Console.WriteLine($"{Name} growls:
{dialogue}"); }
    public void Trade(IGameCharacter trader) { throw new
NotImplementedException("Enemies don't trade"); }

    // Méthodes pour les PNJ - Utilisées
    public void SetPatrolPath(List<Vector3> path) { this.patrolPath = path; }
    public void FollowPlayer(IGameCharacter player) { /* Implémentation */ }
    public void RespondToInteraction(IGameCharacter interactor) { /*
Implémentation */ }
}

// CivilianNPC.cs - Personnage non-joueur civil
public class CivilianNPC : IGameCharacter
{
    public string Name { get; private set; }
    public int Health { get; set; }
    public Vector3 Position { get; set; }
    public bool IsAlive => Health > 0;

    private List<Vector3> patrolPath;
    private Dictionary<string, string> dialogues = new Dictionary<string, string>
();

    public CivilianNPC(string name, int health, Vector3 position)
    {
        Name = name;
        Health = health;
        Position = position;
    }

    // Méthodes de mouvement - Partiellement utilisées
    public void Move(Vector3 direction) { /* Implémentation simple */ }
    public void Jump() { /* Les civils ne sautent généralement pas */ throw new
NotImplementedException(); }
    public void Crouch() { /* Les civils ne s'accroupissent généralement pas */
throw new NotImplementedException(); }
    public void Swim() { /* Les civils ne nagent généralement pas */ throw new
NotImplementedException(); }
```

```
    public void Climb() { /* Les civils ne grimpent généralement pas */ throw new
NotImplementedException(); }
    public void Fly() { /* Les civils ne volent certainement pas */ throw new
NotImplementedException(); }

    // Méthodes de combat - Non utilisées par les civils
    public void Attack(IGameCharacter target) { throw new
NotImplementedException("Civilians don't attack"); }
    public void Defend() { /* Implémentation minimale - fuir */ }
    public void UseSpecialAbility() { throw new NotImplementedException("Civilians
don't have special abilities"); }
    public void CastSpell(string spellName, IGameCharacter target) { throw new
NotImplementedException("Civilians don't cast spells"); }
    public void Heal(IGameCharacter target, int amount) { throw new
NotImplementedException("Civilians don't heal others"); }

    // Méthodes d'interaction - Partiellement utilisées
    public void PickupItem(IItem item) { /* Implémentation minimale ou non
implémentée */ }
    public void UseItem(IItem item) { /* Implémentation minimale */ }
    public void Speak(string dialogue) { Console.WriteLine($"{Name} says:
{dialogue}"); }
    public void Trade(IGameCharacter trader) { /* Implémentation pour les civils
marchands */ }

    // Méthodes pour les PNJ - Utilisées
    public void SetPatrolPath(List<Vector3> path) { this.patrolPath = path; }
    public void FollowPlayer(IGameCharacter player) { /* Implémentation */ }
    public void RespondToInteraction(IGameCharacter interactor) {
        // Répondre avec un dialogue aléatoire
        if (dialogues.Count > 0)
        {
            var randomDialogue = dialogues.ElementAt(new
Random().Next(dialogues.Count));
            Speak(randomDialogue.Value);
        }
    }

    // Méthodes spécifiques aux civils
    public void AddDialogue(string key, string text)
    {
        dialogues[key] = text;
    }
}

// StaticObject.cs - Objet statique interactif dans le jeu (comme un coffre)
public class StaticObject : IGameCharacter
{
    public string Name { get; private set; }
    public int Health { get; set; } // Représente la durabilité de l'objet
    public Vector3 Position { get; set; }
    public bool IsAlive => Health > 0; // Représente si l'objet est intact

    private IItem containedItem;
```

```
    public StaticObject(string name, int durability, Vector3 position, IItem
containedItem = null)
    {
        Name = name;
        Health = durability;
        Position = position;
        this.containedItem = containedItem;
    }

    // Méthodes de mouvement - Non utilisées par les objets statiques
    public void Move(Vector3 direction) { throw new
NotImplementedException("Static objects cannot move"); }
    public void Jump() { throw new NotImplementedException("Static objects cannot
jump"); }
    public void Crouch() { throw new NotImplementedException("Static objects
cannot crouch"); }
    public void Swim() { throw new NotImplementedException("Static objects cannot
swim"); }
    public void Climb() { throw new NotImplementedException("Static objects cannot
climb"); }
    public void Fly() { throw new NotImplementedException("Static objects cannot
fly"); }

    // Méthodes de combat - Non utilisées par les objets statiques
    public void Attack(IGameCharacter target) { throw new
NotImplementedException("Static objects cannot attack"); }
    public void Defend() { throw new NotImplementedException("Static objects
cannot defend"); }
    public void UseSpecialAbility() { throw new NotImplementedException("Static
objects don't have special abilities"); }
    public void CastSpell(string spellName, IGameCharacter target) { throw new
NotImplementedException("Static objects cannot cast spells"); }
    public void Heal(IGameCharacter target, int amount) { throw new
NotImplementedException("Static objects cannot heal"); }

    // Méthodes d'interaction - Partiellement utilisées
    public void PickupItem(IItem item) { throw new NotImplementedException("Static
objects cannot pick up items"); }
    public void UseItem(IItem item) { throw new NotImplementedException("Static
objects cannot use items"); }
    public void Speak(string dialogue) { throw new NotImplementedException("Static
objects cannot speak"); }
    public void Trade(IGameCharacter trader) { throw new
NotImplementedException("Static objects cannot trade"); }

    // Méthodes pour les PNJ - Non utilisées par les objets statiques
    public void SetPatrolPath(List<Vector3> path) { throw new
NotImplementedException("Static objects don't patrol"); }
    public void FollowPlayer(IGameCharacter player) { throw new
NotImplementedException("Static objects cannot follow players"); }

    // La seule méthode réellement utile pour un objet statique
    public void RespondToInteraction(IGameCharacter interactor)
```

```
    {
        if (containedItem != null && interactor is Player player)
        {
            Console.WriteLine($"{Name} contains {containedItem.Name}");
            player.PickupItem(containedItem);
            containedItem = null;
        }
        else
        {
            Console.WriteLine($"{Name} is empty or cannot be interacted with");
        }
    }
}
```

Problèmes avec ce code

1. L'interface `IGameCharacter` est trop volumineuse et contient des méthodes qui ne s'appliquent pas à tous les types de personnages
2. Les classes comme `CivilianNPC` et `StaticObject` doivent implémenter des méthodes qu'elles n'utilisent pas
3. De nombreuses méthodes lancent des exceptions `NotImplementedException` car elles ne sont pas pertinentes pour certaines classes
4. L'ajout de nouvelles fonctionnalités à l'interface affecterait toutes les classes, même celles qui ne devraient pas les utiliser

Exercice

Refactorisez le code en appliquant le principe de Ségrégation des Interfaces :

1. Divisez l'interface `IGameCharacter` en interfaces plus petites et plus spécifiques
2. Faites en sorte que chaque classe n'implémente que les interfaces dont elle a besoin
3. Organisez les méthodes selon leur fonctionnalité (mouvement, combat, interaction, etc.)
4. Assurez-vous qu'aucune classe n'est forcée d'implémenter des méthodes inutiles

Avantages attendus

- Des interfaces plus cohésives et plus ciblées
- Moins de code mort ou de méthodes lancant des exceptions
- Une meilleure organisation du code
- Une plus grande flexibilité pour étendre le système

Conseils

- Pensez aux responsabilités communes et spécifiques de chaque type de personnage
- Considérez quelles méthodes sont logiquement regroupées ensemble
- N'hésitez pas à créer de nouvelles classes si cela aide à mieux organiser le code