TP5 : Relations Entre Classes - Système d'Inventaire

Objectif

Comprendre et appliquer les différents types de relations entre classes (association, agrégation et composition) à travers la création d'un système d'inventaire pour un jeu.

Contexte

Dans un jeu de rôle, les joueurs collectent des objets, les utilisent et les gèrent dans un inventaire. Le système d'inventaire doit permettre de:

- Stocker différents types d'objets (armes, potions, équipement, etc.)
- Associer ces objets à un personnage
- Gérer les propriétés de l'inventaire comme le poids maximum, les emplacements, etc.

Problème

Le code actuel du système d'inventaire est mal structuré et ne respecte pas les relations appropriées entre les classes, rendant le système difficile à maintenir et à étendre.

Voici un aperçu des classes principales avec une implémentation problématique:

```
// Item.cs - Problème
public class Item
{
    public string name;
    public string description;
    public float weight;
    public int value;
    public string itemType; // "Weapon", "Potion", "Armor", etc.

    // Propriétés spécifiques aux armes
    public int damage;
    public float range;

    // Propriétés spécifiques aux potions
    public int healthRestored;
    public float duration;

    // Propriétés spécifiques aux armures
    public int defense;
    public string armorType; // "Helmet", "Chest", "Boots", etc.

    public void UseItem(Player player)
    {
        if (itemType == "Weapon")
        {
            // Logique d'utilisation d'une arme
```

```
            player.Attack(damage);
        }
        else if (itemType == "Potion")
        {
            // Logique d'utilisation d'une potion
            player.RestoreHealth(healthRestored);
        }
        else if (itemType == "Armor")
        {
            // Logique d'équipement d'une armure
            player.EquipArmor(this);
        }
    }
}

// Inventory.cs - Problème
public class Inventory
{
    public Item[] items = new Item[20]; // Taille fixe d'inventaire
    public int itemCount = 0;

    public void AddItem(Item item)
    {
        if (itemCount < items.Length)
        {
            items[itemCount] = item;
            itemCount++;
        }
    }

    public void RemoveItem(int index)
    {
        if (index >= 0 && index < itemCount)
        {
            // Décaler tous les éléments
            for (int i = index; i < itemCount - 1; i++)
            {
                items[i] = items[i + 1];
            }
            items[itemCount - 1] = null;
            itemCount--;
        }
    }

    public float GetTotalWeight()
    {
        float totalWeight = 0;
        for (int i = 0; i < itemCount; i++)
        {
            totalWeight += items[i].weight;
        }
        return totalWeight;
    }
}
```

```
// Player.cs - Problème
public class Player
{
    public string name;
    public int health;
    public int maxHealth;

    // L'inventaire est directement intégré dans la classe Player
    public Inventory inventory = new Inventory();

    // Des références directes aux objets équipés
    public Item equippedWeapon;
    public Item equippedHelmet;
    public Item equippedChest;
    public Item equippedBoots;

    public void Attack(int damage)
    {
        // Logique d'attaque avec l'arme équipée
        System.Console.WriteLine($"{name} attaque pour {damage} points de
dégâts!");
    }

    public void RestoreHealth(int amount)
    {
        health = System.Math.Min(health + amount, maxHealth);
        System.Console.WriteLine($"{name} restaure {amount} points de vie!");
    }

    public void EquipArmor(Item armor)
    {
        if (armor.itemType == "Armor")
        {
            if (armor.armorType == "Helmet")
            {
                equippedHelmet = armor;
            }
            else if (armor.armorType == "Chest")
            {
                equippedChest = armor;
            }
            else if (armor.armorType == "Boots")
            {
                equippedBoots = armor;
            }
        }
    }
}
```

Problèmes avec cette approche

1. La classe `Item` contient des propriétés qui ne s'appliquent qu'à certains types d'objets
2. La logique d'utilisation d'un objet est centralisée dans la classe `Item` et dépend de conditions basées sur le type
3. La classe `Inventory` utilise un tableau de taille fixe, ce qui limite la flexibilité
4. Il n'y a pas de séparation claire des relations entre `Player`, `Inventory` et `Item`
5. L'équipement est géré avec des références simples sans organisation logique

Exercice

Refactorisez ce code en utilisant les relations appropriées entre classes:

1. **Héritage et polymorphisme** pour les différents types d'objets
2. **Composition** pour la relation entre `Equipment` et `Player` (un joueur a toujours un équipement)
3. **Agrégation** pour la relation entre `Inventory` et `Item` (un inventaire contient des objets qui peuvent exister indépendamment)
4. **Association** pour les relations temporaires (par exemple, utiliser un objet)

Types de Relations à Mettre en Œuvre

Composition

- Relation forte "est une partie de"
- L'objet composé contrôle la durée de vie des objets qui le composent
- Si l'objet composé est détruit, ses composants sont également détruits
- Ex: Un joueur a un équipement qui n'existe pas sans lui

Agrégation

- Relation "a un" ou "contient"
- Les objets agrégés peuvent exister indépendamment
- Ex: Un inventaire contient des objets, mais les objets peuvent exister hors de l'inventaire

Association

- Relation plus faible entre objets
- Les objets se connaissent et interagissent
- Un objet peut être associé à plusieurs autres objets
- Ex: Un joueur utilise temporairement un objet

Conseils

- Utilisez l'héritage pour créer une hiérarchie de classes pour les différents types d'objets
- Utilisez des interfaces pour définir des comportements communs (comme `IUsable`)
- Pensez aux responsabilités de chaque classe (qui devrait gérer l'utilisation d'un objet?)
- Considérez les cycles de vie des objets pour déterminer le type de relation approprié