TP8 : Principe de Substitution de Liskov (LSP)

Objectif

Comprendre et appliquer le principe de Substitution de Liskov (Liskov Substitution Principle), le troisième des principes SOLID.

Rappel du Principe

"Les objets d'une classe dérivée doivent pouvoir remplacer les objets de la classe de base sans affecter le comportement du programme."

Ce principe stipule que les classes dérivées doivent respecter le contrat établi par la classe de base. En d'autres termes, si une classe B est un sous-type d'une classe A, alors les objets de type A peuvent être remplacés par des objets de type B sans altérer les propriétés désirables du programme.

Problème

Le code suivant représente un système de formes géométriques pour un logiciel de dessin. Il contient une violation du principe LSP.

```
// Shape.cs - Code problématique
public class Shape
{
    public virtual double Width { get; set; }
    public virtual double Height { get; set; }

    public Shape(double width, double height)
    {
        Width = width;
        Height = height;
    }

    public virtual double Area()
    {
        return Width * Height;
    }

    public virtual double Perimeter()
    {
        return 2 * (Width + Height);
    }

    public virtual void SetDimensions(double width, double height)
    {
        Width = width;
        Height = height;
    }
}
```

```
// Rectangle.cs
public class Rectangle : Shape
{
    public Rectangle(double width, double height) : base(width, height)
    {
    }

    // Hérite et utilise les implémentations de base
}

// Square.cs - Violation du principe LSP
public class Square : Rectangle
{
    public Square(double side) : base(side, side)
    {
    }

    // Override Width pour maintenir la contrainte du carré (width == height)
    public override double Width
    {
        get { return base.Width; }
        set
        {
            base.Width = value;
            base.Height = value; // Modifie également Height pour maintenir un
carré
        }
    }

    // Override Height pour maintenir la contrainte du carré (width == height)
    public override double Height
    {
        get { return base.Height; }
        set
        {
            base.Width = value; // Modifie également Width pour maintenir un carré
            base.Height = value;
        }
    }

    // Override SetDimensions pour maintenir la contrainte du carré
    public override void SetDimensions(double width, double height)
    {
        // Si width != height, utilise width (ou pourrait lever une exception)
        double side = width;
        base.SetDimensions(side, side);
    }
}

// Circle.cs
public class Circle : Shape
{
    private double radius;
```

```
    public Circle(double radius) : base(radius * 2, radius * 2)
    {
        this.radius = radius;
    }

    public double Radius
    {
        get { return radius; }
        set
        {
            radius = value;
            base.Width = value * 2;
            base.Height = value * 2;
        }
    }

    public override double Area()
    {
        return Math.PI * radius * radius;
    }

    public override double Perimeter()
    {
        return 2 * Math.PI * radius;
    }

    public override void SetDimensions(double width, double height)
    {
        // Un cercle ne peut pas avoir width != height
        double avgDimension = (width + height) / 2;
        radius = avgDimension / 2;
        base.Width = avgDimension;
        base.Height = avgDimension;
    }
}

// Exemple de code client
public class ShapeProcessor
{
    public void ProcessRectangle(Rectangle rectangle)
    {
        // Le client s'attend à ce comportement pour un rectangle:
        // La largeur et la hauteur sont indépendantes
        rectangle.Width = 5;
        rectangle.Height = 10;

        // Vérifier que l'aire est bien width * height (devrait être 50)
        double area = rectangle.Area();
       Debug.Log($"Aire du rectangle: {area}");

        // Vérifier que le rectangle a bien les dimensions définies
       Debug.Log($"Dimensions: {rectangle.Width} x {rectangle.Height}");
    }
```

```
}

// Test qui démontre la violation LSP
public class LSPTest
{
    public static void Main()
    {
        ShapeProcessor processor = new ShapeProcessor();

       Debug.Log("Test avec un Rectangle:");
        Rectangle rectangle = new Rectangle(2, 3);
        processor.ProcessRectangle(rectangle);
        // Résultat attendu: Aire = 50, Dimensions: 5 x 10

       Debug.Log("\nTest avec un Square (devrait se comporter comme un
Rectangle):");
        Square square = new Square(4);
        processor.ProcessRectangle(square);
        // Résultat: Aire = 100, Dimensions: 10 x 10
        // La hauteur a également changé lorsque la largeur a été modifiée,
        // ce qui viole les attentes du client sur le comportement d'un Rectangle
    }
}
```

Problèmes avec ce code

1. La classe `Square` hérite de `Rectangle` mais viole le principe LSP car elle modifie le comportement attendu d'un rectangle
2. Lorsqu'un client utilise un `Square` comme un `Rectangle`, le comportement est différent (changer la largeur affecte aussi la hauteur)
3. La classe `Circle` a également des problèmes avec sa relation à `Shape` et le fonctionnement de `SetDimensions`
4. L'héritage utilisé ici force des contraintes artificielles et des comportements non naturels

Exercice

Refactorisez le code en appliquant le principe de Substitution de Liskov :

1. Repensez la hiérarchie des classes pour éviter les violations LSP
2. Assurez-vous que chaque classe dérivée peut être utilisée partout où sa classe de base est attendue, sans surprises
3. Utilisez des interfaces ou des classes abstraites appropriées
4. Assurez-vous que le code client fonctionne correctement avec toutes les formes

Avantages attendus

- Code plus prévisible et fiable
- Suppression des comportements inattendus lors de l'utilisation du polymorphisme
- Meilleure organisation de la hiérarchie des classes

Conseils

- Réfléchissez à ce qu'est vraiment une forme géométrique et aux propriétés communes à toutes les formes
- Considérez si une relation "est un" (héritage) est vraiment appropriée entre un carré et un rectangle
- Utilisez des interfaces pour définir des contrats clairs