Liskov Yerine Geçme Prensibi

Tanım:

- alt sınıflardan oluşan nesnelerin üst sınıfın nesneleri ile yer değiştirdikleri zaman, aynı davranışı sergilemesini beklemektir.

- *Subtypes must be substitutable for their base types.*

  Alt sınıflar üst sınıflar ile yer değiştirebilir olmalı.

- *İstemci açısından fonksiyonelliği bozmayacak şekilde alt sınıf, üst sınıfın metodlarını override etmelidir.*

Örnek:

1. Dikdörtgen bilgilerini alan ve setleyen; alan hesaplama metodu olan bir dikdörtgen sınıfı oluşturunuz. (setWidth, setHeight, getArea)
2. Bu dikdörtgen sınıfını baz alan bir kare sınıfı oluşturunuz ver setleme işlemini buna göre düzenleyiniz.
3. Bir istemci sınıf üzerinden dikdörtgen nesnesi tanımlayın; bu nesnenin içeriğinin bir fabrika metodundan (RectangleFactory()) geldiğini varsayın.
4. Nesnenin yükseklik, genişlik özelliklerini setleyin; alan hesaplatın.

```
public class Rectangle
{
    protected int _width;
    protected int _height;
    public int Width
    {
        get { return _width; }
    }
    public int Height
    {
        get { return _height; }
    }

    public virtual void SetWidth(int width)
    {
        _width = width;
    }
    public virtual void SetHeight(int height)
    {
        _height = height;
    }
    public int getArea()
    {
        return _width * _height;
    }
}

public class Square : Rectangle  /* In an "is a" relationship, the derived class is clearly a kind of the base class */
{
    public override void SetWidth(int width)
    {
        _width = width;
        _height = width;
    }

    public override void SetHeight(int height)
    {
        _height = height;
        _width = height;
    }
}
```

```
public void AreaOfRectangle()
    {
        Rectangle r =
RectangleFactory();
// Returns the rectangle type object
        r.SetWidth(7);
        r.SetHeight(3);
        r.getArea();
    }
```

```
// Factory ------

public  Rectangle RectangleFactory()
    {
        return new Square();
    }
```

Yeni türetilen sınıflar ana sınıfın davranışını değiştirmeden genişletmelidir.

Örnekte setWidth ve setHeight metodlarının davranışları değişmiştir.

Doğrusu nasıl olmalıydı ?

```
public class Quadrilaterals
{
    public virtual int Height { get; set; }
    public virtual int Width { get; set; }
    public int getArea()
    {
        return Height * Width;
    }
}

public class Rectangle :Quadrilaterals
{

    public override int Width
    {
        get { return base.Width; }
        set { base.Width = value; }
    }
    public override int Height
    {
        get { return base.Height; }
        set { base.Height = value; }
    }

}

public class Square : Quadrilaterals{
    public override int Height
    {
        get { return base.Height; }
        set { SetWidthAndHeight(value); }
    }
    public override int Width
    {
        get { return base.Width; }
        set { SetWidthAndHeight(value); }
    }

    private void SetWidthAndHeight(int value)
    {
        base.Height = value;
        base.Width = value;
    }
}

public Quadrilaterals QuadrilateralsFactory()
   {
       return new Square();
   }
   public void AreaOfQuadrilateral()
   {
       Quadrilaterals r = QuadrilateralsFactory();
          r.Height=7;
          r.Width=3;
          r.getArea();
   }
```

Ödev:

- Bir çoklu sosyal medya düzenleme uygulaması yazıldığı düşünülsün.
- Bu uygulama kullanıcının A ve B gibi iki farklı sosyal web sayfasına kullanıcının iletisi, konumu ve anlık görüntüsünü yollamaktadır.
- Uygulama bilgisayar, tablet ve cep telefonundan gelen veri ile çalışmaktadır.
- Bilgisayardan anlık görüntü özelliği kullanılamamaktadır.
- Böyle bir uygulamanın sınıf yapılarını tasarlayınız, Liskov Yerine Koyma Prensibine uyup uymadığını kontrol ediniz.

Arayüz Ayrımı

- İstemciler, gereksiz yere kullanmayacakları metodlar bulunan arayüzlere zorlanmamalıdır.
- Bir büyük arayüz yerine daha basit birden fazla arayüz kullanmak bu durumu çözer.

Örnek

```
// interface segregation principle
interface IWorker {
            public void work();
            public void eat();
}

class Worker implements IWorker{
            public void work() {
                        // ....working
            }
            public void eat() {
                        // ...... eating in launch break
            }
}

class SuperWorker implements IWorker{
            public void work() {
                        //.... working much more
            }

            public void eat() {
                        //.... eating in launch break
            }
}

class Manager {
            IWorker worker;

            public void setWorker(IWorker w) {
                        worker=w;
            }

            public void manage() {
                        worker.work();
            }
}
```

Örnek

```
// interface segregation principle - good example
interface IWorker extends Feedable, Workable {
}

interface IWorkable {
	public void work();
}

interface IFeedable{
	public void eat();
}

class Worker implements IWorkable, IFeedable{
	public void work() {
		// ....working
	}

	public void eat() {
		//.... eating in launch break
	}
}

class Robot implements IWorkable{
	public void work() {
		// ....working
	}
}
```

```
class SuperWorker implements IWorkable, IFeedable{
	public void work() {
		//.... working much more
	}

	public void eat() {
		//.... eating in launch break
	}
}

class Manager {
	Workable worker;

	public void setWorker(Workable w) {
		worker=w;
	}

	public void manage() {
		worker.work();
	}
}
```

Soru:

```
// The code that violates ISP
interface IMachine
{
    public bool yazdir(List<Item> item);
    public bool zımbala(List<Item> item);
    public bool faksCek(List<Item> item);
    public bool tara(List<Item> item);
    public bool fotokopiCek(List<Item> item);
}
// Code implementing the IMachine interface.

class Machine : IMachine
{
   public Machine()
   {
   }

   public bool yazdir(List<Item> item)
   {
      // Dökümanları (item)  yazdırır.
      Console.WriteLine("All Items printed" + item.Count());
   }

   public bool zımbala(List<Item> item)
   {
      // Dökümanları zımbalar
      Console.WriteLine("Items stapled" + item.Count());
   }

   public bool faks(List<Item> item)
   {
      // Dökümanları fakslar
      Console.WriteLine("All Items Faxed" + item.Count());
   }

   public bool tara(List<Item> item)
   {
      // Dökümanları tarar
      Console.WriteLine("All Items Scanned" + item.Count());
   }

   public bool fotokopiCek(List<Item> item)
   {
      // Dökümanların fotokopisini çek
      Console.WriteLine("All Items Photo copied" + item.Count());
   }
}
```

Soldaki arayüz ile makinelerin kullanabileceği özellikler verilmiştir. Bu arayüzdeki metodları kullanabilen farklı makineler üretilebilmektedir.

Buna göre soldaki tasarım doğru mudur?

Ödev:

- Doğru tasarımı kodlayın ☺

Bağımlılıkları tersine çevirme

- Yüksek seviye sınıflar, düşük seviye modüllere bağlı olmamalı.
- Her ikisi de soyut (abstract) yapılara bağlı olmalı.

- Bu sayede sınıfların bağımlılıkları azaltılır.

Örnek

```
public class Email
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendEmail()
    {
        //Send email
    }
}

public class SMS
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendSMS()
    {
        //Send sms
    }
}

public class Notification
{
    private Email _email;
    private SMS _sms;
    public Notification()
    {
        _email = new Email();
        _sms = new SMS();
    }

    public void Send()
    {
        _email.SendEmail();
        _sms.SendSMS();
    }
}
```

Bu tasarımda hata var mıdır?

Örnek

- Notification sınıfı SMS ve Email sınıflarına bağımlıdır.
- Bağımlılığı soyut sınıfa bağlamalıyız.

- Dikkat: Notification sınıfı single responsibility prensibine de uymamaktadır bu haliyle.

Örnek-Doğru Tasarım

```
public interface IMessage
{
    void SendMessage();
}

public class Email : IMessage
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendMessage()
    {
        //Send email
    }
}

public class SMS : IMessage
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendMessage()
    {
        //Send sms
    }
}

public class Notification
{
    private ICollection<IMessage> _messages;

    public Notification(ICollection<IMessage> messages)
    {
        this._messages = messages;
    }
    public void Send()
    {
        foreach(var message in _messages)
        {
            message.SendMessage();
        }
    }
}
```

Örnek

```
class Imalat
{
    public void Olustur()
    {
        Araba araba = new Araba();
        araba.ArabaOlustur(true);
    }
}

class Araba
{
    public void ArabaOlustur(string renk)
    {
        //İşlemler
    }
}

//------
```

```
interface IArac
{
    void Olustur(string Renk);
}

class Imalat
{
    Iaraca arac;
    public Imalat()
    {
        arac = new Araba();
    }
    public void Olustur()
    {
        arac.Olustur(true);
    }
}

class Araba : IArac
{
    public void Olustur(string Renk)
    {
        //İşlemler
    }
}
```

Soru

- Metinden sesli okuma uygulaması yazılmak isteniyor. Pdf, Word ve Epub türü dökümanlardan sesli okuma yapılabilmektedir.
- Her doküman türü kendi içinde standart olarak Başlık, Yazar ve isbn numarası bilgilerini tutmaktadır.
- Ayrıca, dökümandan, düz metin bilgisinin elde edildiği bir dönüştürücü işlevine sahiptirler. Her dökümanın dönüştürücüsü kendine özeldir.
- Bu uygulamayı bağımlılıkları tersine çevirme prensibine göre tasarlayıp kodlayınız.