Macoratti.net OOP - O princípio da substituição de Liskov (LSP)

Neste artigo vou tratar de um assunto que é um dos *pilares da programação orientada a objetos*: O **princípio da substituição de Liskov.** (LSP)

SOLID é um acrônimo para as 5 principais padrões de desenho :

1. ***S*RP** The Single Responsibility Principle: -- Uma classe deverá ter apenas um , e somente um , motivo para mudar;
2. ***O*CP** The Open Closed Principle: -- Você deverá poder estender o comportamento de uma classe sem modificá-la;
3. ***L*SP** The Liskov Substitution Principle: -- A classes derivadas devem poder ser substituídas pelas suas classes bases;
4. ***I*SP** The Interface Segregation Principle: -- make fine grained interfaces that are client specific.
5. ***D*IP** The Dependency Inversion Principle -- Dependa de abstração de não de implementação.

O princípio **LSP** foi definido por **Barbara Liskov** da seguinte forma: (*http://pt.wikipedia.org/wiki/Princ%C3%ADpio_da_substitui%C3%A7%C3%A3o_de_Liskov*)

> ***"Se q(x) é uma propriedade demonstrável dos objetos x de tipo T. Então q(y) deve ser verdadeiro para objetos y de tipo S onde S é um subtipo de T."***

Nas entrelinhas:

"Se você pode invocar um método q() de uma classe T (base), deve poder também invocar o método q() de uma classe T'(derivada) que é derivada com herança de T (base)."

De forma bem simples o princípio de **Liskov** sugere que :

"Sempre que uma classe cliente esperar uma instância de uma classe base X, uma instância de uma subclasse Y de X deve poder ser usada no seu lugar."

Em outras palavras: "***Uma classe base deve poder ser substituída pela sua classe derivada.***"

O objetivo é ter certeza de que novas classes derivadas estão estendendo das classes base mas sem alterar o seu comportamento.

Por que o princípio de LisKov é importante ?

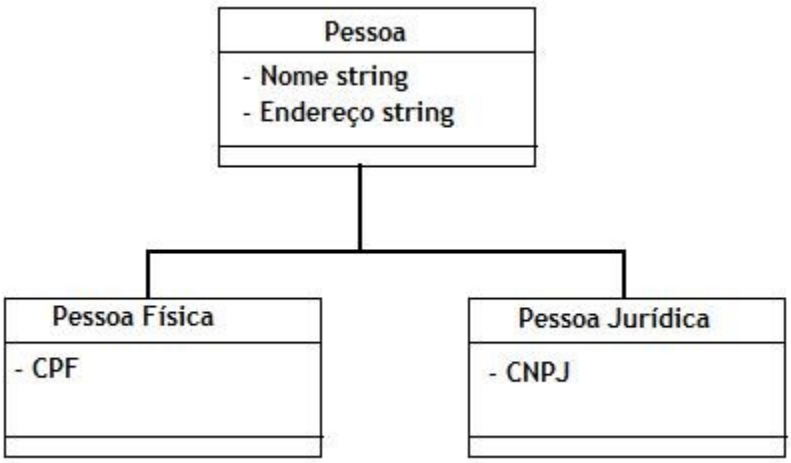
Porque sem aplicar o princípio de Liskov a hierarquia de classes seria uma bagunça e os testes de unidade para a superclasse nunca teriam sucesso para a subclasse.

Uma rápida revisão de conceitos

Herança

- Estende atributos e métodos de uma classe;
- Classe Pai, classe Base ou Superclasse é a classe que foi herdada;
- Classe Filha ou Sub-Classe é a classe que herda da classe Pai;
- Generalização - Obtém similaridades entre classes e define novas classes. As classes mais genéricas são as classes Pai;
- Especialização - Identifica atributos e métodos não correspondentes entre classes distintas colocando-os na classe filha;
- **É um**;

Exemplo:



Pessoa - classe Pai, classe Base ou SuperClasse

- PessoaFisica - classe Filha ou sub-classe
- PessoaJuridica - classe Filha ou sub-classe

- A classe **Pessoa** é a classe genérica;
- As classes PessoaJuridica e Pessoa Fisica são especializações;

- PessoaFisica **É uma** pessoa;
- PessoaJuridica **É uma** pessoa;

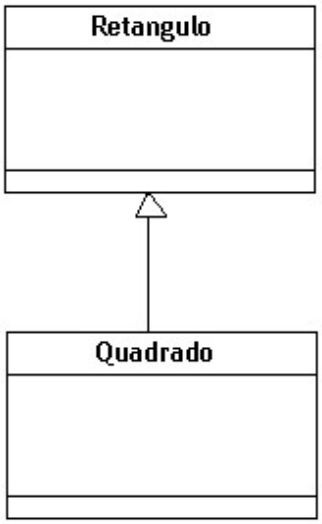
Vamos lembrar os conceitos de herança: *Quando uma classe herda de outra classe dissemos que temos a relação "É Um".*

Quando a classe **Pessoa Juridica** ou **Pessoa Fisica** herda da classe **Pessoa**, ela passa a ser uma classe **Pessoa** estendida, pois contém todos os métodos e propriedades de Pessoa *(nome e endereço*) e mais as suas próprias funcionalidades (*CPF ou CNPJ)*.

Entendendo o LSP na prática : Violando o LSP

Para ilustrar o princípio LSP eu vou me basear em um exemplo clássico que pode ser encontrado em diversos artigos na web e que eu adaptei para a linguagem C#.

Vamos supor que temos um projeto que implementa duas classes : **Retangulo e Quadrado**.



A classe **Quadrado** *é uma* **Retangulo** que tem uma característica especial: a largura e a altura são iguais.

Vamos definir duas classes para representar ambas as contas.

A notação UML ao lado ilustra o relacionamento de herança existente entre duas classes onde temos que
Quadrado é a classe derivada e **Retangulo** é a classe Base.

A primeira vista estamos no caminho certo e nosso desenho parece

ser consistente.

De forma até intuitiva concordamos que um quadrado é um retângulo (*pelo menos conceitualmente*) e
portanto é lógico modelar a classe quadrado como sendo derivada da classe Retangulo. Dessa forma
Retangulo é a classe base e **Quadrado** a classe derivada.

No entanto, esse tipo de pensamento pode levar a alguns problemas sutis, mas significativos.
Geralmente, estes problemas não estão previstos, até que executamos o código do aplicativo.

Vamos criar um pequeno projeto na linguagem C#, implementar as duas classes e testar se o nosso modelo não viola o principio **LSP**.

Crie um novo projeto usando o **Visual C# 2010 Express Edition** do tipo **Console Application** com o nome **LSP_Violacao**;

No menu **Project** selecione **Add Class**, informe o nome **Retangulo.cs** e clique em **Add**;

Em seguida defina o seguinte código na classe:

```
namespace LSP_Violacao
{
   class Retangulo
   {
      protected int m_largura;
      protected int m_altura;

      public virtual void setLargura(int
largura)
      { m_largura = largura;}

      public virtual void setAltura(int
altura)
      {  m_altura = altura;  }

      public int getLargura()
      { return m_largura;   }
```

Na classe **Retangulo** temos:

- Duas propriedades m_largura e m_altura
- O método **setLargura(int largura)**
- O método **setAltura(int altura)**

Observe a palavra-chave virtual no método **setLargura**();

A palavra-chave **virtual** é usada para modificar um método,
propriedade, indexador ou declaração de evento e permitir
que ele seja sobrescrito em uma classe derivada.

Na classe **Quadrado** iremos sobre-escrever o método **setLargura**();

```
        public int getAltura()
        { return m_altura;    }

        public int getArea()
        { return m_largura * m_altura; }
    }
}
```

Não estou usando as ***propriedades automáticas*** para que fique bem claro.

Vamos agora criar a classe **Quadrado**.

No menu **Project** selecione **Add Class**, informe o nome **Quadrado.cs** e clique em **Add**;

A seguir defina o seguinte código nesta classe:

```
namespace LSP_Violacao
{
    class Quadrado : Retangulo
    {
        public override void
setLargura(int largura)
        {
            m_largura = largura;
            m_altura = largura;
        }

        public override void
setAltura(int altura)
        {
            m_largura = altura;
            m_altura = altura;
        }
    }
}
```

A classe **Quadrado** herda da classe Retangulo.

O método **setLargura**() desta classe sobrescreve o método da classe Base (*setLargura*);

Use o modificador **override** para modificar um método, uma propriedade, um indexador, ou um evento. Um método de substituição fornece uma nova implementação de um membro herdado de uma classe base.

O método sobrescrito por uma declaração de **override** é conhecido como o método base sobrescrito. O método base sobrescrito deve ter a mesma assinatura que o método de substituição.

Você não pode substituir um método não-virtual ou estático.
O método sobrescrito da classe base deve ser virtual, abstract, ou override.

Como a classe **Quadrado** herda da classe **Retangulo** ela vai herdar os métodos ***setLargura e setAltura*** , o que para um quadrado não faz muito sentido visto que a largura e a altura são iguais. É por isso que estamos sobrescrevendo esses dois métodos na classe Quadrado e definindo a largura igual a altura.

Agora estamos prontos para testar a nossa aplicação e verificar se as definições do nosso projeto estiverem de acordo com o princípio LSP a utilização da classe **Quadrado**() deverá poder ser usada sem problema algum no lugar da classe base **Retangulo**.

Vamos incluir o código abaixo no evento **Main** do arquivo **Programa.cs**:

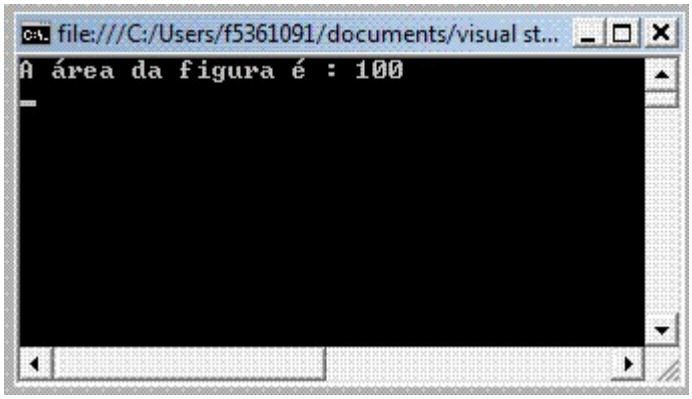
```
using System;

namespace LSP_Violacao
{
   class Program
   {
      private static Retangulo
getNovoRetangulo()
      {
         //um factory
         return new Quadrado();
      }

      static void Main(string[] args)
      {
            //vamos criar um novo
retangulo
            Retangulo r =
Program.getNovoRetangulo();

            //definindo a largura e
altura do retangulo
         r.setLargura(5);
          r.setAltura(10);
     // o usuário sabe que r é um
retângulo
     // e assume que ele pode definir
largura e altura
            // como para a classe
base(Retangulo)

         Console.WriteLine(r.getArea());
            Console.ReadKey();
     // O valor retornado é 100 e não
50 como era esperado
      }
   }
}
```



Ao executarmos o projeto iremos obter o resultado exibido na figura acima.

Não é o resultado esperado.

Ao usar uma instância da classe quadrado que sobrescreveu o método da classe
Base Retangulo, para calcular a área de um retângulo obtivemos um valor
incorreto.

Dessa forma o princípio LSP foi violado pois o resultado deveria ser o correto.

No exemplo temos que a uma instância da classe **Quadrado** é devolvida por uma *factory* com base em algumas condições e nós não sabemos exatamente que tipo de objeto será retornada.

Se o princípio LSP estivesse sendo corretamente aplicado o fato de usar a instância de Quadrado não implicaria em mudança de comportamento como veremos que irá ocorrer.

Estamos criando um novo retângulo definindo a altura de 5 e a largura de 10 para obter a área.

O resultado deveria ser **50 mas obtemos 100.**

Qual o problema ???

Ao definir a largura para 10, acabei definindo também a altura, deixando a área com valor de 100 e não 50 como era esperado.

Neste caso um quadrado não é um retangulo , e ao aplicarmos o princípio "É Um" da herança de forma automática vimos que ele não funciona para todos os casos.

A instância da classe **Quadrado** quando usada quebra o código produzindo um resultado errado e isso viola o principio de LisKov onde *uma classe filha (Quadrado) deve poder substituir uma classe base(Retangulo).*

Para obter o resultado correto basta alterar o código do método estático **getNovoRetangulo();**

```
private static Retangulo getNovoRetangulo()
{
//um factory
return new Retangulo();
}
```

Fazendo isso o resultado fica correto mas mostra que não podemos usar a instância de **Quadrado** como **Retangulo**.

Este exemplo mostra uma violação clara do princípio **LSP** onde a inclusão da classe **Quadrado** herdando de **Retangulo** mudou o comportamento da classe base violando assim o princípio **Open-Closed.**

O principio **LSP** é uma extensão do **Princípio Open Closed** e isso significa que temos que ter certeza de que as novas classes derivadas estão estendendo as classes base, sem alterar seu comportamento.

Eu sei é apenas OOP , mas eu gosto...

"Não se turbe o vosso coração;crede em Deus, crede também em mim." (João 14:1)

Referências:

- Padrões de Projeto
- Padrões de Projeto - O modelo MVC - Model View Controller
- O padrão Singleton
- VB.NET - Permitindo uma única instância da sua aplicação
- Design Patterns - o padrão Factory
- Conceitos sobre projetos - Decomposição
- Usando o padrão Strategy
- SRP - O princípio da responsabilidade única - Macoratti.net
- Boas Práticas - O padrão inversão de controle (IoC) - Macoratti.net
- **Seção Padrões de Projeto do site Macoratti.net**
- **Super DVD .NET - A sua porta de entrada na plataforma .NET**
- **Super DVD Vídeo Aulas - Vídeo Aula sobre VB .NET, ASP .NET e C#**

José Carlos Macoratti