mviegas / **SOLID**

Estudos com exemplos em C# dos princípios do SOLID.

11 commits | **1** branch | **0** releases | **1** contributor

Branch: **master** | New pull request | Create new file | Upload files | Find File | Clone or download

mviegas DIP — Latest commit f7849bd on 10 May 2017

README.md — DIP — 2 years ago

README.md

SOLID

O SOLID é um acrônimo criado por criado por Michael Feathers a partir dos cinco primeiros princípiocs ("five first principles") da programação orientada à objetos identificados por Robert C. Martin no início dos anos 2000.

Esses princípios tem como objetivo obter as vantagens da orientação a objetos através de um código de alta qualidade e permita:

- Leitura, testes e manutenção fáceis
- Extensibilidade com o menor esforço possível
- Reaproveitamento
- Maximização do tempo de utilização do código

A aplicação correta dos princípios SOLID permite um código altamente estruturado e desacoplado de modo que são evitadas, devido à uma estrutura frágil:

- Dificuldades no teste
- Dificuldades no isolamento de funcionalidades
- Duplicação de código
- Quebra de código em vários lugares após alterações

A seguir vamos estudar cada um dos 5 príncipios que compõem o SOLID.

Single Responsability Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Single Responsability Principle

> "A class should have one, and only one, reason to change"

O Princípio da Responsabilidade Única (Single Responsability Principle) diz que "uma classe deve ter um único motivo para ser modificada". Em outras palavras, uma classe deve ter uma única responsabilidade, um único motivo de existir.

Veja o exemplo da classe Usuário abaixo.

Violação do SRP

```
public class Usuario
{
    public int Id { get; set; }
    public string Nome { get; set; }
    public string Email { get; set; }
    public string CPF { get; set; }

    public Tuple<bool, string> InserirUsuario()
    {
        if (CPF.Length != 11)
            return new Tuple<bool, string>(false, "O CPF deve ter 11 caracteres");

        using (var connection = new SqlConnection())
        {
            var command = new SqlCommand();

            connection.ConnectionString = "Local";

            command.Connection = connection;
            command.CommandType = System.Data.CommandType.Text;
            command.CommandText = $"INSERT INTO USUARIO (ID, NOME, EMAIL, CPF) VALUES (@id, @nome, @email, @cpf)";

            command.Parameters.AddWithValue("id", Id);
            command.Parameters.AddWithValue("nome", Nome);
            command.Parameters.AddWithValue("email", Email);
            command.Parameters.AddWithValue("cpf", CPF);

            connection.Open();
            command.ExecuteNonQuery();
        }

        return new Tuple<bool, string>(true, "Usuário inserido com sucesso");
    }
}
```

Da maneira como foi implantada, essa classe viola o SRP uma vez que uma classe não deve delegar responsabilidades à ela própria. No caso do exemplo, a classe não deve validar a si mesma ou se adicionar ao banco. Outro exemplo de violação está no fato de que um método de Inserção de um determinado registro não deve ter a responsabilidade de validar esse registro.

Logo, podemos concluir que uma classe construída da maneira demonstrada acima possui múltiplas responsabilidades, o que vai contra o SRP.

SRP da maneira correta

Para apicar o SRP de uma maneira correta então, devemos separar as responsabilidades descritas acima, de modo que haja exclusivamente uma classe para cada responsabilidade:

- Manter as propriedades e indicar se estas são válidas ou não.

```
public class Usuario
{
    public int Id { get; set; }
    public string Nome { get; set; }
    public string Email { get; set; }
    public string CPF { get; set; }
    public bool IsValid { get { return UsuarioValidation.Validar(); } }
}
```

- Executar essa validação.

```
public static class UsuarioValidation
```

```
{
    public bool Validar(Usuario usuario)
    {
        return usuario != null && usuario.CPF != null && usuario.CPF != String.Empty && usuario.CPF.Length == 11;
    }
}
```

- Interagir com o Banco.

```
public class UsuarioRepository
{
    public bool Adicionar(Usuario usuario)
    {
        try
        {
            using (var connection = new SqlConnection())
            {
                var command = new SqlCommand();

                connection.ConnectionString = "Local";
                command.Connection = connection;
                command.CommandType = System.Data.CommandType.Text;
                command.CommandText = $"INSERT INTO USUARIO (ID, NOME, EMAIL, CPF) VALUES (@id, @nome, @email, @cpf)";

                command.Parameters.AddWithValue("id", Id);
                command.Parameters.AddWithValue("nome", Nome);
                command.Parameters.AddWithValue("email", Email);
                command.Parameters.AddWithValue("cpf", CPF);

                connection.Open();
                command.ExecuteNonQuery();
            }

            return true;
        }
        catch (Exception)
        {
            return false;
        }
    }
}
```

- Executar o processo de adição de um novo Usuário.

```
public class UsuarioService
{
    public bool Inserir(Usuario usuario)
    {
        if (usuario.IsValid())
            return false;

        var repository = new UsuarioRepository();
        repository.Adicionar(usuario);
    }
}
```

De uma maneira bem grosseira, corrigimos a violação do SRP separando uma única classe Usuário, que antes era responsável por verificar se suas propriedades eram válidas, criar uma conexão com o banco e efetuar a inserção, em 4 classes com responsabilidades únicas. Dessa forma, desacoplamos o código, facilitando alterações e testes do mesmo.

Imagine, por exemplo, que os seus dados hoje guardados em um banco SQL passem a ser guardados em um arquivo XML. A única alteração a ser feita será no Repositório, mantendo todo o resto do nosso código intacto, uma vez que o Serviço e a Validação não precisam saber de que maneira é feita a conexão e a inserção na base de dados.

Open-Closed Principle

> "Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification"

Esse princípio permite que o código seja extendido sem se preocupar com as classes, métodos legados. Uma vez que estas classes e métodos foram criados, eles não devem ser mais modificados, mas sim devem estar abertas para extensão.

Violação do OCP

Seja o seguinte exemplo: um serviço de impressão de notas fiscais em um sistema de controle de estoque. O estoque é responsável por receber as mercadorias e devolvê-las caso elas tenham algum problema, por exemplo.

Suponha que agora o nosso serviço precise contemplar um novo tipo de nota fiscal como uma nota de saída de mercadoria, emitida quando alguma mercadoria sai para entrega, por exemplo. Seguindo a mesma lógica implementada acima, o código ficaria assim:

```
public class NotaFiscal { }

public class NotaDeRecebimentoDeMercadoria : NotaFiscal
{
    public void GerarNotaDeRecebimento() { }
}

public class NotaDeDevolucaoDeMercadoria : NotaFiscal
{
    public void GerarNotaDeDevolucao() { }
}

public class ImpressaoDeNotas
{
    public void ImprimirNotas(IEnumerable<NotaFiscal> notas)
    {
        foreach (var notaFiscal in notas)
        {
            if (notaFiscal is NotaDeRecebimentoDeMercadoria)
                (notaFiscal as NotaDeRecebimentoDeMercadoria).GerarNotaDeRecebimento();
            else if (notaFiscal is NotaDeDevolucaoDeMercadoria)
                (notaFiscal as NotaDeDevolucaoDeMercadoria).GerarNotaDeDevolucao();
        }
    }
}
```

Além de criar uma nova classe e implementar um novo método, a classe de ImpressaoDeNotas teria que ser alterada, mais precisamente o método Imprimir notas teria que ser alterado, para contemplar a impressão do novo tipo de Nota Fiscal. Daí, pode-se concluir que esse método não segue o OCP.

OCP da maneira correta

Para implementar o OCP corretamente, algumas coisas devem ser mudadas. Valendo-se de coneceitos como herança e polimorfismo, é interessante tornar classe NotaFiscal abstrata e implementar nela um método abstrato GerarNotaFiscal que será sobreescrito pelas classes filhas, garantindo assim que as classes filhas tenham esse método com a mesma chamada, mas com implementações diferentes.

```
public abstract class NotaFiscal
{
    public abstract void GerarNotaFiscal();
}

public class NotaDeRecebimentoDeMercadoria : NotaFiscal
```

```
{
    public override void GerarNotaFiscal() { /* Implementação da Nota de Recebimento */ }
}

public class NotaDeDevolucaoDeMercadoria : NotaFiscal
{
    public override void GerarNotaFiscal() { /* Implementação da Nota de Devolução */ }
}

public class NotaDeSaidaDeMercadoria : NotaFiscal
{
    public override void GerarNotaFiscal() { /* Implementação da Nota de Saída */ }
}

public class ImpressaoDeNotas
{
    public void ImprimirNotas(IEnumerable<NotaFiscal> notas)
    {
        foreach (var notaFiscal in notas)
        {
            notaFiscal.GerarNotaFiscal();
        }
    }
}
```

Desse modo, nosso serviço de impressão de notas está aberto para imprimir novos tipos de notas que possam surgir no controle do estoque, mas fechado para modificações.

Liskov Substitution Principle

> "Let q(x) be a property provable of objects x of a type T. Then q(y) should be provable objects of type S, where S is a subtype of T."

Esse princípio foi proposto por Barabara Liskov em um artigo científico no ano de 1988, e é fundamental para a aplicação de heranças na orientação à objetos. Em outras palavras, este princípio também é enunciado da seguinte maneira: "uma classe base deve poder ser substituída por sua classe derivada". Ou seja, caso existe uma função ou um método, em uma classe filha que tenha como herança uma classe base, então, se existir uma instância dessa classe filha, esta deve ter o comportamento igual ao da classe base.

Violação do LSP

Um exemplo clássico de violação do LSP seria o seguinte: sejam duas formas geométricas, um quadrado e um retângulo. Ambos são quadriláteros.

```
public class Retangulo
{
    public virtual int Altura { get; set; }
    public virtual int Largura { get; set; }

    public int CalcularArea { get { return Altura * Largura; } }
}

public class Quadrado : Retangulo
{
    public override int Altura
    {
        set { base.Altura = base.Largura = value; }
    }

    public override int Largura
    {
        set { base.Altura = base.Largura = value; }
    }
```

```
}
```

A classe base, Retângulo, tem Altura e Largura como duas propriedades que tem valores independentes um do outro. Quando a classe Quadrado herda de Retângulo e sobrescreve essas propriedades, dizendo que elas devem ter o mesmo valor, ocorre uma violação do LSP, uma vez que não temos mais o mesmo comportamento da classe base na classe filha.

LSP da maneira correta

Uma possível solução para isso seria criar uma classe abstrata Paralelogramo, pois todos os paralelogramos tem as mesmas características: possuem altura e largura e tem a área calculada como um produto dessas duas propriedades.

```
public abstract class Paralelogramo
{
    public Paralelogramo(int altura, int largura)
    {
        Altura = altura;
        Largura = largura;
    }

    public int Altura { get; private set; }
    public int Largura { get; private set; }
    public int Area { get { return Altura * Largura; } }
}

public class Quadrado : Paralelogramo
{
    public Quadrado(int altura, int largura) : base(altura, largura)
    {
        if (altura != largura)
            throw new Exception("Para um quadrado, a altura e a largura devem ser iguais.");
    }
}

public class Retangulo : Paralelogramo
{
    public Retangulo(int altura, int largura) : base(altura, largura) { }
}
```

Nessa solução específica, pode-se perceber como o LSP e o OCP andam lado a lado. Porque o tratamento da verificação dos lados do quadrado não foi implementado na classe Paralelogramo? Ao fazer isso, o OCP estaria sendo violado pois a classe Paralelogramo não estaria fechada para modificações. Da maneira como foi implementado, é garantido que Paralelograma está aberta para extensão e fechada para modificações.

Interface Segregation Principle

> Classes that implement interfaces should not be forced to implement methods they do not use.

Esse princípio diz que é melhor que existam várias interfaces com alguns métodos que serão todos utilizados, do que uma única interface que tenha vários métodos mas várias implementações que não utilizam todos eles. Ao implementar o ISP, cria-se um desacoplamento do código, facilitando a manutenção e trazendo coesão à implementação.

Violação do ISP

Pensando num exemplo mais próximo de uma aplicação real, seja um serviço de gravação e recuperação de dados no banco, onde cada dado desse seja uma entidade que herde de uma classe comum chamada EntityBase.

```
public interface IService<T> where T : EntityBase
{
    T Get();
    void Put(T entity);
}
```

```
public abstract class EntityBase
{
    public EntityBase(int id)
    {
        Id = id;
    }

    public int Id { get; private set; }
}
```

Suponha agora que existam as seguintes classes e que o serviço não deva colocar nenhum Usuário novo banco, mas apenas recuperar.

```
public class Usuario : EntityBase
{
    public string Nome { get; set; }
}

public class Acesso : EntityBase
{
    public DateTime DataEHora { get; set; }
}

 public class UsuarioService : IService<Usuario>
{
    public Usuario Get(int id)
    {
        /* Implementação da lógica de Buscar o usuário */
    }

    public void Put(Usuario entity)
    {
        throw new NotImplementedException();
    }
}

public class AcessoService : IService<Acesso>
{
    public Acesso Get(int id)
    {
        /* Implementação da lógica de Buscar o Acesso */
    }

    public void Put(Acesso entity)
    {
        /* Implementação da lógica de incluir o Acesso */
    }
}
```

A implementação acima viola o ISP, uma vez que existem métodos de IService que não são implementados em UsuarioService.

ISP da maneira correta

Uma solução seria separar a interface IService em duas: uma de leitura e uma de escrita. O código ficaria então da seguinte maneira:

```
public interface IReaderService<T> where T : EntityBase
{
    T Get(int id);
}

public interface IWriterService<T> where T : EntityBase
{
```

```
    void Put(T entity);
}

public class UsuarioService : IReaderService<Usuario>
{
    public Usuario Get(int id)
    {
        /* Implementação da lógica de Buscar o usuário */
    }
}

public class AcessoService : IReaderService<Acesso>, IWriterService<Acesso>
{
    public Acesso Get(int id)
    {
        /* Implementação da lógica de Buscar o Acesso */
    }

    public void Put(Acesso entity)
    {
        /* Implementação da lógica de incluir o Acesso */
    }
}
```

Dessa forma, o ISP não estaria sendo violado.

Dependency Inversion Principle

> High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Esse princípio tem como objetivo desacoplar as dependências do projeto, induzindo que módulos de alto e baixo nível dependam de uma mesma abstração.

Voltando à resolução do primeiro exemplo SRP, temos a seguinte classe estrutura para o cadastro de um usuário no banco de dados:

```
public class UsuarioRepository
{
    public bool Adicionar(Usuario usuario)
    {
        try
        {
            using (var connection = new SqlConnection())
            {
                var command = new SqlCommand();

                connection.ConnectionString = "Local";
                command.Connection = connection;
                command.CommandType = System.Data.CommandType.Text;
                command.CommandText = $"INSERT INTO USUARIO (ID, NOME, EMAIL, CPF) VALUES (@id, @nome, @email, @cpf)";

                command.Parameters.AddWithValue("id", Id);
                command.Parameters.AddWithValue("nome", Nome);
                command.Parameters.AddWithValue("email", Email);
                command.Parameters.AddWithValue("cpf", CPF);

                connection.Open();
                command.ExecuteNonQuery();
            }

            return true;
        }
        catch (Exception)
```

```
        {
            return false;
        }
    }
}

public class UsuarioService
{
    public bool Inserir(Usuario usuario)
    {
        if (usuario.IsValid())
            return false;

        var repository = new UsuarioRepository();
        repository.Adicionar(usuario);
    }
}
```

Note que para o serviço chamar o Repositório, ele instancia diretamente a classe UsuarioRepository. Isso viola o DIP e faz com que o Serviço dependa sempre da implementação concreta do Repositório. Uma possível solução seria:

```
public interface IUsuarioRepository
{
        bool Adicionar(Usuario usuario);
}

public class UsuarioRepository : IUsuarioRepository
{
    public bool Adicionar(Usuario usuario)
    {
        try
        {
            using (var connection = new SqlConnection())
            {
                var command = new SqlCommand();

                connection.ConnectionString = "Local";
                command.Connection = connection;
                command.CommandType = System.Data.CommandType.Text;
                command.CommandText = $"INSERT INTO USUARIO (ID, NOME, EMAIL, CPF) VALUES (@id, @nome, @email, @cpf)";

                command.Parameters.AddWithValue("id", Id);
                command.Parameters.AddWithValue("nome", Nome);
                command.Parameters.AddWithValue("email", Email);
                command.Parameters.AddWithValue("cpf", CPF);

                connection.Open();
                command.ExecuteNonQuery();
            }

            return true;
        }
        catch (Exception)
        {
            return false;
        }
    }
}

public interface IService<T> where T : class
{
        bool Inserir(T entity);
}

public class UsuarioService : IService<Usuario>
{
```

```
        private IUsuarioRepository _usuarioRepository;

        public UsuarioService(IUsuarioRepository usuarioRepository)
        {
                _usuarioRepository = usuarioRepository;
        }

    public bool Inserir(Usuario usuario)
    {
        if (usuario.IsValid())
            return false;

        _usuarioRepository.Adicionar(usuario);
    }
}
```

Desta forma, as interfaces IUsuarioRepository e IService funcionariam como contratos, seguindo a idéia de que quem contrata não precisa saber como o que foi contratado é executado, mas apenas que o que está no contrato seja atendido.

Nota: a criação de IService segue a mesma idéia da criação de IUsuarioRepository, que é disponibilizar apenas o contrato, e não sua implementação concreta, a quem for necessário. Seria possível ainda a criação de mais uma interface IRepository ou apenas IRepository que tivesse os métodos básicos compartilhados por todos os repositórios.

Existem alguns frameworks em .NET como Castle.Core e Unity, que possibilitam a injeção das dependências através de construtores, properties, fields e métodos.