Entendendo SOLID com exemplos em C#

Como os cinco princípios escritos por Uncle Bob podem ajudar seu código

Carlos Henrique Zansavio Follow
Jan 23, 2018 · 6 min read

Robert C. Martin (Uncle Bob) escreveu em 2000 os principais fatores que ele observou em códigos bem escritos, esses fatores, ele chamou de SOLID— acrônimo para 5 práticas essenciais que devem ser seguidas para construir sistemas que possam crescer com uma boa manutenção.

A engenharia de software nos diz que do tempo que gastamos construindo software, 20% é criando e 80% dando manutenção. Então com isso em mente, é essencial colocar esforço e tempo para que ele seja melhor escrito possivel.

Um dos primeiros conceitos que precisamos saber para emergir nesse

mundo, é conhecido como SOLID:

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Esse artigo tem o objetivo de explicar e dar exemplos claros de como aplicar esses cinco princípios.

1. Single Responsibility Principle

O primeiro princípio nos diz que devemos construir classes com apenas um objetivo. Se sua classe precisa de mais de uma razão para mudar, ela está fazendo mais de uma coisa—quebrando esse principio.

Código ruim:

```
class Usuario
{
   public string Nome { get; private set; }
   public string Email { get; private set; }

   public Usuario(string nome, string email)
   {
      this.Nome = nome;
      this.Email = email;
   }

   public void PersistirUsuario()
   {
      SqlConnection conexao = new SqlConnection("string de
conexao");
      SqlCommand cmd = new SqlCommand();
      // cmd.CommandText...
      // ...
      // ..
      conexao.Close();
   }
```

Como podemos ver, esse codigo é ruim por vários motivos.
Primeiro que a classe Usuario não deve saber como ela deve ser persistida. E além do mais, temos SQL no meio desse método. Esse SQL precisa ser encapsulado e mesmo se usassemos um padrão como o Repository, ele não deveria saber como fazer essa conexão.

Uma possível solução, como foi dito, é criar classes para essa persistência. Podemos usar o padrão Repository para tal.

```
class Usuario
{

  public string Nome { get; private set; }

  public string Email { get; private set; }

  public Usuario(string nome, string email)
  {

     this.Nome = nome;

     this.Email = email;

  }

}

class UsuarioRepository

{

   public void Persistir(Usuario usuario)

   {

      ConexaoSQL conexao = new ConexaoSQL();

      // Salva usuario no banco

      // Fecha conexao

   }
```

```
}

class ConexaoSQL

{

   public void AbrirConexao()

   {

      //..

   }

}
```

Nesse exemplo, podemos ver que cada classe tem sua propria função. Caso um programador que chegasse no projeto agora quisesse mudar uma string de conexão, ele facilmente saberia que precisaria ir apenas na classe de conexão do banco de dados.

Esse código pode ser melhorado e será nos proximos princípios utilizando-se de Interfaces.

2. Open/Closed Principle

Esse princípio nos diz que as entidades devem estar abertas apenas para expansão e fechadas para mudanças/modificações.

Ou seja, devemos poder expandir as entidades sem que sejam preciso fazer mudanças para que caibam essas expansões.

Vamos a um exemplo para deixar isso mais claro

```
class Paralelograma

{

      public int Area(int largura, int comprimento, string
```

```
tipo)

      {

      if(tipo == "retangulo")

         return largura * comprimento;

      else if(tipo == "quadrado")

         return largura * largura;

      else

         return -1;

   }

}
```

Nesse exemplo simples podemos ver que se caso o paralelograma trabalhado fosse outro, teriamos que fazer mais um if. Com isso teriamos que modificar a classe quebrando esse princípio.

Vamos voltar a esse exemplo mais pra frente com uma elegante solução.

3. Liskov Substitution Principle

Esse princípio parece complicado a primeira vista, mas ele diz basicamente que se uma entidade X é herdada de uma entidade Y, então a Y deve se comportar igual a X. Ou seja, a Y é a entidade base e ela deve responder por todas as entidades filhas dela, incluindo a X.

Com esse princípio, podemos resolver milhares de problemas já que se criarmos uma Interface, as classes que a implementam vão responder como se fosse a Interface.

Esse princípio será exemplificado mais para frente.

4. Interface Segregation Principle

O quarto princípio diz que não devemos ter preguiça de escrever interfaces. Uma interface que faz tudo é um sintoma de que essa prática está sendo quebrada.

Além de que se uma das entidades não precisa de algum dos métodos que a interface obrigou a classe implementar, é certeza de que essa interface pode ser quebrada em duas.

```
interface ICadastro

{

   void ValidarDados();

   void PersistirBanco();

   void MandarEmail();

}

class Cliente : ICadastro

{

   public void ValidarDados()

   {...}

   public void PersistirBanco()

   {...}

   public void MandarEmail()

   {...}

}

class Produto : ICadastro
```

```
{
   public void ValidarDados()
   {...}
   public void PersistirBanco()
   {...}
   public void MandarEmail()
  {
     // apenas criamos o metodo mas deixamos vazio.
  }
}
```

Faz todo sentido mandar um email para Cliente, mas faz algum sentido mandar um email para Produto? Logo essa interface pode ser quebrada em duas ou mais interfaces.

```
interface ICadastroCliente
{
   void ValidarDados();
   void PersistirBanco();
   void MandarEmail();
}
interface ICadastroProduto
{
   void ValidarDados();
```

```
    void PersistirBanco();

}

class Cliente : ICadastroCliente

{

    public void ValidarDados()

    {...}

    public void PersistirBanco()

    {...}

    public void MandarEmail()

    {...}

}

class Produto : ICadastroProduto

{

    public void ValidarDados()

    {...}

    public void PersistirBanco()

    {...}

}
```

Interfaces especificas são melhores que interfaces gerais.

Por mais que pareça desperdício escrever mais codigo ou redundância de código , agora podemos escrever métodos especificos de produtos e clientes com uma melhor segregação.

Diga não a interfaces gerais!

5. Dependency Inversion Principle

Para que esse princípio seja bem implementado, é preciso ter feito todos os outros corretamente. Talvez, eu diria que esse seja o mais importante, mas por precisar dos outros bem construidos, ele seja tão importante quanto.

Ele diz que entidades devem depender de abstrações. As entidades de mais alto nivel não devem depender das de mais baixo nivel. E que não devemos depender de detalhes mais sim de abstrações.

No final, ele diz que devemos abstrair o máximo possivel :D

Podemos fazer isso utilizando de injeção de dependência (dependency injection).

Só para falar de injeção de dependência e ilustrar o princípio da inversão de dependência, poderia ser feito um post completo mas para efeito geral, vou falar ilustrar dois jeitos de faze-la.

Ele funciona da seguinte maneira: como C#, Java e outras linguagens são fortemente tipadas, devemos abstrair e implementar nossas classes concretas herdando de interface ou classes abstradas. Então quando chamamos o método, na compilação, ainda não sabemos exatamente de qual classe aquele objeto pertence. Só será decidido no runtime. Com isso eliminamos os ifs e outros códigos feios.

Primeiro, vamos solucionar aquele exemplo que deixamos aberto no princípio Open/closed. Aqui não injetamos a dependência por construtor mas apenas passamos no método.

```
interface IPoligono

{

   int Calcular();

}

class Retangulo : IPoligono
```

```
{

   public int Largura { get; set; }

   public int Comprimento { get; set; }

   public int Calcular()

   {

      return Largura * Comprimento;

   }

}

class Quadrado : IPoligono

{

   public int Largura { get; set; }

   public int Calcular()

   {
      return Largura * Largura;

   }

}

class Paralelograma

{

   public int Area(IPoligono poligono)

   {

      return poligono.Calcular();

   }

}
```

Claro que esse exemplo simples não faz muito sentido, mas é apenas um exemplo didático para entendermos melhor o uso da injeção de dependência. Agora podemos mudar o código de calcular sem que afete a classe Paralelograma ou até estende-lá sem precisar modificá-la.

Agora vamos entender uma injeção de dependência melhorando o primeiro exemplo passando-a no construtor.

```
class Usuario

{...}

interface IUsuarioRepository

{

   void Persistir(Usuario usuario);

}


class UsuarioRepositoryPgSql : IUsuarioRepository

{

   public void Persistir(Usuario usuario)

   {

      //implementamos como deve ser feito a implementação
para persistir

      //um usuario no banco de dados postgresql

   }

}

interface IUsuarioServices

{

   void AdicionarUsuario(Usuario usuario);
```

```
}

class UsuarioServices : IUsuarioServices

{

   private readonly IUsuarioRepository _usuarioRepository;

   public UsuarioServices(IUsuarioRepository
usuarioRepository)

   {

      _usuarioRepository = usuarioRepository;

   }

   public void AdicionarUsuario(Usuario usuario)

   {
      // poderiamos verificar se o objeto usuario é valido
antes de persistir

      _usuarioRepository.Persistir(usuario);

   }

}
```

Nós acabamos sendo redundantes mas obecemos o princípio da responsabilidade unica. Cada método não deve saber como o outro faz o seu serviço.

Se caso seu cliente chegasse para você amanhã e dissesse que não da para ser Postgresql mais, simplesmente mudariamos a classe de repositorio sem que outras classes fossem afetadas.

A partir daqui, você poderia injetar essa dependência utilizando Unity, Ninject ou outras bibliotecas. Fica fácil a partir do momento que seu codigo é desacoplado.

Bom, com isso agora você entende o que são os cinco princípios SOLID e como aplicá-los.

Os exemplos foram feitos em C# mas são facilmente aplicados em qualquer linguagem a partir do momento que voce entende esses princípios.

Happy coding!

E esse foi meu primeiro post no Medium. Contribua e dê um feedback do que como posso melhorar :)

```
this.Nome = nome;

this.Email = email;

}

public void PersistirUsuario()

{

SqlConnection conexao = new SqlConnection("string de conexao");

SqlCommand cmd = new SqlCommand();

// cmd.CommandText…

// …

// ..

conexao.Close();

}
```