Clean Code - Guia e Exemplos

Clean Code ou código limpo se refere a um conjunto de boas práticas na escrita de software que você pode aplicar para obter uma maior legibilidade e manutenabilidade do seu código.

Índice

O que é o Clean Code?

Por que estamos falando tanto sobre código limpo (Clean Code) e por que isto é tão importante para nós? De fato a manutenção de um software é tão importante quanto sua construção.

Como relatado por Robert C. Martin em seu livro clássico, *Clean Code*, um Best Seller da nossa área, algumas práticas e visões são importantíssimas para mantermos a *vida* do nosso software.

> ***IMPORTANTE*** *Este artigo não descarta a leitura do livro, que é muito mais denso e profundo sobre o assunto.*

As empresas investem milhões em softwares todo ano, mas com tantas mudanças no time e nas tecnologias, como fazer este investimento durar? Como garantir uma boa manutenção, durabilidade, vida ao software? Segundo Uncle Bob, as práticas abaixo são o caminho.

Regras gerais

Siga as convenções

Se você começou agora em um projeto ou acabaram de definir suas convenções, siga-as! Se utilizam por exemplo constantes em maiúsculo, enumeradores com **E** como prefixo, não importa! Siga sempre os padrões do projeto.

KISS

Mantenha as coisas simples! Este conceito vem até de outro livro, e particularmente acho que é a base de uma boa solução. Normalmente tendemos a complicar as coisas que poderiam ser muito mais simples.

Então, Keep It Stupid Simple (Mantenha isto estupidamente simples - KISS)!

Regra do escoteiro

"Deixe sempre o acamapamento mais limpo do que você encontrou!" O mesmo vale para nosso código. Devolva (Check in) sempre o código melhor do que você o obteve. Se todo desenvolvedor no time tiver esta visão, e devolver um pedacinho de código melhor do que estava antes, em pouco temos teremos uma grande mudança.

Causa raiz

Sempre procure a causa raiz do problema, nunca resolva as coisas superficialmente. No dia-a-dia, na correria, tendemos a corrigir os problemas superficialmente e não adentrar neles, o que muitas vezes causa o re-trabalho!

Tente sempre procurar a causa raiz e resolver assim o problema de uma vez por todas!

Regras de design

Mantenha dados de configuração em alto nível

Algo que toda aplicação tem são suas configurações, como as conhecidas **ConnectionStrings**. Tente sempre deixar estas configurações ou o *parse* delas em um nível mais alto possível.

Evite sobrescrever configurações em métodos dentro de **Controllers** ou algo do tipo. Se possível, mantenha esta passagem no método principal, no início da aplicação e não mexa mais nisto!

Exemplo

Em diversas aplicações que trabalho, crio sempre uma classe **Settings** no projeto base e depois no **Startup** das aplicações populo ela com as configurações. Isto garante que não teremos estas configurações sendo escritas em todo lugar e também que não precisaremos do **IConfiguration** que fica no **ASP.NET** em projetos que não são Web.

MeuProjeto.Domain

```
public static class Settings {
    public static string ConnectionString { get; set; }
}
```

MeuProjeto.Api

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
    Settings.ConnectionString = Configuration.GetConnectionString("connectionString
}
```

MeuProjeto.Infra

```
using(var connection = new SqlConnection(Settings.ConnectionString)) {
    ...
}
```

Polimorfismo no lugar de IFs

Um **IF** ou condicional, como o nome diz, traz uma tomada de decisão a nossa aplicação, o que implica no aumento da complexidade da mesma. No geral devemos evitar o uso excessivo destes.

Nestes cenários, opte sempre pelo polimorfismo ao invés de tomar decisão em todo método que cria.

Exemplo

Vamos tomar como base uma classe **Pagamento**, onde temos pagamento via Boleto ou Cartão de Crédito, porém nos pagamentos via Boleto, caso o dia do vencimento seja sábado ou domingo (Final de semana), o mesmo pode ser pago no próximo dia útil.

IMPORTANTE *Esta regra não está 100% correta ou eficiente, é apenas uma demonstração*

```
public class Pagamento {
    public bool PodeSerPago() {
        if(tipo == ETipoPagamento.Boleto)
        {
            if(vencimento.Day != IsWeekend())
                return true;
        }

        if(tipo == ETipoPagamento.CartaoCredito)
            ...
    }
}
```

Note que temos duas tomadas de decisão dentro do método **PodeSerPago**, onde a primeira se refere apenas a pagamentos do tipo Boleto. Caso hajam mais formas de pagamento futuramente, como trataríamos este código? Encheríamos de **IF**?

A solução mais plausível é derivar da classe base **Pagamento** criando o **PagamentoBoleto** que sobrescreve o método **PodeSerPago**, dando uma nova funcionalidade a ele.

```
public class Pagamento {
    public virtual bool PodeSerPago() {
        ...
    }
}

public class PagamentoBoleto : Pagamento {
    public override bool PodeSerPago() {
        if(vencimento.Day != IsWeekend())
            return true;
    }
}
```

Mult-thread

Sempre que necessário utilize processamento em Threads separadas. Já temos suporte a multi-threads e paralelismo no C# faz um bom tempo e o próprio Async/Await já ajudam nisso.

Exemplo

Sem async/await

```
[HttpGet("cursos")]
public IActionResult Index([FromServices] IContentRepository repository)
{
    ViewBag.Courses = repository.GetContents(EContentType.Course);
    return View();
}
```

Com async/await

```
[HttpGet("cursos")]
public async Task<IActionResult> Index([FromServices] IContentRepository repository
{
    ViewBag.Courses = await repository.GetContentsAsync(EContentType.Course);
    return View();
}
```

Separe os códigos mult-thread

Seguindo o mesmo exemplo acima, é uma boa prática manter o que é assíncrono separado do que é síncrono, para não forçar um método a ser ou não assíncrono por conta de outro trecho de código.

Exemplo

```
public async Task<IEnumerable<Model>> GetAsync()
{
    var model = new Model();
    model.Courses = await _context.Courses.ToListAsync();
    model.Tags = _context.Courses.ToList(); // Não async

    return model;
}
```

Utilize Async como sufixo

Se um método é assíncrono, utilize sempre o sufixo **async** para identificá-lo.

```
public async Task<IEnumerable<Model>> GetAsync()
```

Evite configurações desnecessárias

Evite deixar configurações no sistema só por que alguém ainda não definiu como aquilo deve ser. Isto polui o código e traz uma complexidade desnecessária.

Exemplo

```
public void ConfiguraUsoMySql()
{
    // ainda não sabemos se vamos ou não suportar MySQL também
    throw new NotImplementedException();
}
```

Utilize injeção de dependência

Sempre que possível utilize injeção de dependência, ele vai tornar seu código mais limpo e desacoplado.

Exemplo

Lei de Demeter

A Lei de Demeter (LoD) ou princípio do menor conhecimento é um princípio que prega os seguintes pontos.

- Cada unidade deve ter conhecimento limitado sobre outras unidades: apenas unidades próximas se relacionam.

- Cada unidade deve apenas conversar com seus amigos.
- Não fale com estranhos, apenas fale com seus amigos imediatos

Exemplo

```
public class Order()
{
    public Discount Discount { get; set; }
}

public class Discount()
{
    public decimal Amount { get; set; }

    public void Apply() { ... }
}
```

Mau exemplo

```
public class OrderHandler()
{
    var order = new Order();
    order.Discount.Apply(); // <-
}
```

Bom exemplo

```
public class Order()
{
    public Discount Discount { get; set; }

    public void Place()
    {
        Discount?.Apply();
    }
}

public class OrderHandler()
{
    var order = new Order();
    order.Place();
}
```

Regras sobre entendimento do código

Seja consistente

Se você executa algo de uma forma, execute todo o resto desta mesma forma. Seja consistente na forma com que aplica o código. Siga sempre o padrão definido.

Exemplo

```
// Codificando em inglês
public class CustomerRepository { ... }

// Agora mudou para "portuglês"
public class ProdutoRepository { ... }

// Agora é português
public class RepositorioUnidadeMedida { ... }
```

```
// Utilizou sufixo ASYNC no método assíncrono
public async Task<Product> GetAsync() { ... }

// Agora não usou mais =/
public async Task<Course> Get() { ... }
```

Utilize variáveis concisas

Opte por variáveis concisas, mesmo que resultem em um nome maior. Elas devem ser auto-explicativas, sem a necessidade de comentários ou informações adicionais.

Exemplo

```
// Total do que?
decimal total = 0;

// Total do carrinho de compras
decimal shoppingCartTotal = 0;
```

Obsessão primitiva

No dia-a-dia tendemos a nos focar apenas em tipos primitivos (Built-in), causando uma obsessão pelos mesmos. Podemos criar e usar objetos de valor (Value Objects) para suprir melhor esta necessidade.

Exemplo

Mau exemplo

```
public class Customer
{
    public string Email { get; set; }

    public Customer
    {
        // Valida E-mail
    }
}

public class Employee
{
    public string Email { get; set; }

    public Customer
    {
        // Valida E-mail novamente
    }
}
```

Bom exemplo

```
// Value Object
public class Email
{
    public string Address { get; set; }

    public Email
    {
        // Valida E-mail
    }
}

public class Customer
{
    public Email Email { get; set; }
}

public class Employee
{
    public Email Email { get; set; }
}
```

Evite dependências lógicas

Não escreva métodos cujo funcionamento correto dependa de algo contido em sua classe.

Exemplo

Mau exemplo

```
public class Student
{
    public bool IsSubscriber { get; set; }

    public void Xpto()
    {
        if(IsSubscriber)
            ... // Só executa se for assinante
    }
}
```

Bom exemplo

```
public class Student
{
    ...
}

public class Subscriber : Student
{
    public void Xpto()
    {
        ...
    }
}
```

Evite condicionais negativas

No C# a negação é dada por um sinal de exclamação (**!**) que muitas vezes pode ser imperceptível, ocasionando na má leitura do código.

Exemplo

```
// Evite
if(!IsSubscriber) { ... }

// Utilize
if(IsSubscriber) { ... }
```

Regras de nomes

Escolha nomes descritivos

Esolher bons nomes para classes, variáveis e métodos é essencial para um código limpo. Lembre-se que se você precisa explicar seu código, então algo pode ser melhorado nele.

Exemplo

```
// Evite
var x = 256;

// Duração do que? Qual a métrica?
int duration = 25;

// Muito mais expressivo
int durationInMinutes = 25;
```

Faça distinções significantes

Utilize sempre nomes nos quais quem estiver lendo seu código possa diferenciar seu significado de outros possíveis nomes.

Exemplo

```
// Evite
var salario = 7500M;

// Tem um significado maior
var salarioEmReais = 7500M;
```

Utilize nomes pronunciáveis e buscáveis

Evite utilizar nomes difíceis de pronunciar ou inventar nomes e conveções para variáveis, classes e métodos. Lembre-se sempre da linguagem ubíquoa e da importância dela no código.

Exemplo

```
// Evite
var strTexto = "Meu texto aqui";

// Evite
public void GenerateBoletoInLote() {}

// Evite
public void Cadastry() {}
```

Evite uso excessivo de strings

Quem nunca perdeu horas procurando um BUG que era apenas um problema de comparação de string? Evite digitar a mesma string várias vezes, utilize constantes para isto.

Exemplo

```
// Evite
if(environment == "PROD")
    ...

// Utilize
const string ENV = "PROD";

if(environment == ENV)
    ...
```

Não use prefixo ou caracteres especiais

Não utilize prefixo com o tipo da variável, classe ou método e NUNCA use espaços ou caracteres especiais nestes itens.

Exemplo

```
// Evite
public class clsCustomer { ... }

// Evite
string strNome = "André";

// Evite
var situação - "Pendente";
```

Regras para funções ou métodos

Pequenas e com apenas um objetivo

Mantenha suas funções ou métodos o menor possível. É mais fácil ter métodos menores e reutilizáveis do que tudo dentro de um método só.

Exemplo

```
// Evite
public void RealizarPedido()
{
    // Cadastra o cliente
    // Aplica o desconto
    // Atualiza o estoque
    // Salva o pedido
}

// Utilize
public void SaveCustomer() { ... }
public void ApplyDiscount() { ... }
public void UpdateInventory() { ... }
public void PlaceOrder() { ... }
```

Utilize nomes descritivos

A mesma regra dos nomes anteriormente vista aqui se aplica para este cenário. Mantenha nomes concisos, sem caracteres especiais.

Exemplo

```
// Evite
// Calcular o que?
public void Calcular() { ... }

// Utilize
// Calcula o ICMS
public void CalcularICMS() { ... }
```

Opte por poucos parâmetros

Evite exigir muitos parâmetros para construção do objeto, assim como use e abuse dos **Optional Parameters** do C#.

Exemplo

```
// Evite
public void SaveCustomer(string street, string number, string neighborhood, string

// Melhorando
public void SaveCustomer(Address address) { ... }
```

Cuidado com efeitos colaterais

Evite que uma função altere valores de outra classe sem ser a dela. Isto é chamado de efeito colateral.

Exemplo

```
// Evite
public class Order
{
    public decimal Total { get; set; }
}

var order = new Order();

// Qualquer um fora da classe Order
// pode atualizar seu total
order.Total = 250;
```

```
// Utilize
public class Order
{
    public decimal Total { get; private set; }

    public void CalculateTotal() { ... }
}

var order = new Order();

// Total é privado, ninguém de fora consegue
// modificá-lo, evitando efeitos colaterais
order.Total = 250; // ERRO
```

Não tome decisões desnecessárias

Não utilize os famosos "flags" para tomar decisões dentro dos métodos, divida-os em vários métodos ou até mesmo outras classes.

Exemplo

```
// Evite
public class CustomerRepository
{
    public void CreateOrUpdate(Customer customer, bool create)
    {
        if(create)
            ...
        else
            ...
    }
}
```

```
// Utilize
public class CustomerRepository
{
    public void Create(Customer customer) { ... }
    public void Update(Customer customer) { ... }
}
```

Regras de comentários

Um código bom é expressivo

Teoricamente, se você precisa comentar uma parte do seu código, é por que algo está errado com ele, ele não está expressivo o suficiente.

Não seja redundante

Evite comentários que não fazem sentido algum ao contexto ou cenário.

Exemplo

```
// Evite

// Função principal do sistema
public void Main() { ... }
```

Não feche os comentários

Não há necessidade de fechar os comentários.

Exemplo

```
// Evite

// Comentário // <- Desnecessário
public void Main() { ... }
```

Evite códigos comentados

Não deixe sujeira em seu código, ao invés de deixar algo comentado, remova ele. Hoje temos versionadores de código, você pode "voltar no tempo" facilmente.

Exemplo

```
// Evite
public void MinhaFuncao()
{
    // string texto = "1234";
    // public void Metodo() {... }
}
```

Inteção

Um bom uso de comentários é sobre a intenção de um método, classe ou variável (Variável nem tanto).

Exemplo

```
// Utilize

// Retorna a lista de produtos inativos
// para o relatório de fechamento mensal
public IList<Product> ObtemProdutosInativos()
{
    ...
}
```

Esclarecimento

Outro uso interessante para os comentários são esclarecimentos sobre o código.

Exemplo

```
// Utilize
public void CancelarPedido()
{
    // Caso o pedido já tenha sido enviado
    // ele não pode mais ser cancelado.
    if(DataEnvio > DateTime.Now)
    {
        AddNotification("O pedido já foi enviado e não pode ser cancelado");
    }
}
```

Consequências

Podemos utilizar comentários para alertar sobre trechos do código que podem ter consequências mais sérias. Neste caso recomendo o uso de um comentário em XML mais elaborado.

Exemplo

```
// Utilize

/// <summary>
/// ATENÇÃO: Este método cancela o pedido e estorna o pagamento
/// </summary>
public void CancelarPedido()
{
    ...
}
```

Estrutura do código

Separe conceitos verticalmente

Mantenha uma estrutura de pastas saudável e organizada. Não precisa criar uma pasta para cada arquivo, mas pode haver uma separação por contexto ou feature.

Exemplo

- MeuApp
  - MeuApp.Domain
    - MeuApp.Domain.Contexts

- MeuApp.Domain.Contexts.PaymentContext
  - MeuApp.Domain.Contexts.PaymentContexts.Entities
  - MeuApp.Domain.Contexts.PaymentContexts.ValueObjects
  - MeuApp.Domain.Contexts.PaymentContexts.Enums
- MeuApp.Domain.Contexts.AccountContext
  - MeuApp.Domain.Contexts.AccountContext.Entities
  - MeuApp.Domain.Contexts.AccountContext.ValueObjects
  - MeuApp.Domain.Contexts.AccountContext.Enums

Declare variáveis próximas de seu uso

Não crie todas as variáveis juntas, no começo da class ou método, defina-as próximas de onde serão utilizadas.

Exemplo

```
// Evite
var total = 0;

public void CreateCustomer() { ... }

public void CreateOrder() { ... }

public void UpdateCustomer() { ... }

public void CalculateTotal()
{
    total = 250; // <- Só é utilizada aqui
}
```

```
// Utilize
public void CreateCustomer() { ... }

public void CreateOrder() { ... }

public void UpdateCustomer() { ... }

var total = 0;
public void CalculateTotal()
{
    total = 250;
}
```

Agrupe funcionalidades similares

Se uma função pertence a um grupo dentro de um objeto, mantenha-as sempre por perto.

Exemplo

```
// Evite
public void CreateCustomer() { ... }

public void CheckInventory() { ... }

public void CreateOrder() { ... }

public void UpdateCustomer() { ... }

public void CalculateTotal() { .. }
```

```
// Utilize
public void CreateCustomer() { ... }
public void UpdateCustomer() { ... }

public void CheckInventory() { ... }
public void CreateOrder() { ... }
public void CalculateTotal() { .. }
```

Declare funções de cima para baixo

Ordenar as funções também é importante. Além da sua ordem de grandeza, suas assinaturas também devem ter uma boa oganização.

Exemplo

```
// Utilize
public void CreateCustomer(string name) { ... }

public void CreateCustomer(string name, int age) { ... }

public void CreateCustomer(string name, int age, Address address) { ... }

public void CreateCustomer(string name, int age, Address address, bool active) { ..
```

Mantenha poucas e curtas linhas

Evite funções com linhas longas ou muitas linhas. Não existe um número correto, mas com certeza quanto mais código em uma função, mais difícil de mantê-la será.

Exemplo

```
// Utilize
public void CreateCustomer(string name)
{
    var customer = new Customer(name);
    _repository.Customers.Add(customer);
    _repository.SaveChanges();
}
```

Não use alinhamento horizontal

Não há necessidade de alinhar horizontalmente variáveis, constantes ou mesmo propriedades.

Exemplo

```
// Evite
private   Long            requestParsingTimeLimit;
protected Request         request;
private   FitNesseContent context;
this.context =            context;
input =                   s.getInputStream()
requestParsingTimeLimit = 900;
```

Use os espaços em branco corretamente

Utilize espaço em branco para associar ou não itens relacionados. Uma boa IDE já fará este trabalho por você.

Exemplo

```
// Utilize
private void meuMetodo(String parametro) {
  variavel++;
  int outraVariavel = algumArray.length();
  total += algumMetodo();
  outraClasse.algumMetodo(variavel, total);
  outroMetodo(total);
}
```

Não quebre a identação

Este item dispensa comentários. Um código não identado não pode ser enviado para o projeto.

Exemplo

```
// Evite
public class MinhaClasse{
var valor=12;
Console.WriteLine(valor);
}
```

Objetos e estruturas

Esconda estruturas internas

Este tópico abrange uma discussão extensa. Esconder a estrutura de um objeto, ou seja, privar as propridades relacioadas a dados dele, vai sempre trazer resultados positivos e negativos.

Particularmente, gosto de tornar os `SET` privados, mas não é uma regra do meu código e não aplico em todas as propriedades. Como consequência, sempre precisamos de mais métodos para manipulação destes valores.

Se os dados não fazem sentido para os objetos externos, não há discussão, mantenha-os privados.

Exemplo

```
public class NotificationContext
{
    private List<string> _notifications;

    public void Add(string notification)
    {
        _notifications.Add(notification);
    }

    public bool IsValid() => _notifications.Any();

    public IEnumerable Notifications { get => _notifications.AsEnumerable(); }
}
```

Opte por estrutura de dados

Estruturas de dados representam a forma como os dados são organizados, podendo ser uma `class` ou um `struct`. Normalmente associamos as `struct` mais a estrutura de dados do que as classes, mas podemos estruturar dados com qualquer uma delas.

A diferença é que ao usar `class` (OOP) temos recursos como abstração, herança, polimorfismo, dentre outros.

Particularmente acho que a segmentação em objetos de valor é um ponto chave neste item.

Exemplo

```
// Usando estruturas
public struct Email
{
    public Email(string address)
    {
        // Permite apenas E-mails hotmail, gmail, yahoo...
    }

    public string Address { get; private set; }
}

public class Customer
{
    public Email Email { get; private set; }
}
```

```
// Usando classes
public class Email
{
    public Email(string address)
    {
        // Permite qualquer tipo de E-mail
    }

    public string Address { get; private set; }
}

public class CommonEmail : Email
{
    public Email(string address)
        : base(address)
    {
        // Permite apenas E-mails hotmail, gmail, yahoo...
    }
}

public class Customer
{
    public Email Email { get; private set; }
}
```

Nos dois casos temos estruturas representando um E-mail como objeto de valor, porém no segundo cenário, podemos criar extensões e ter uma maior flexibilidade.

Evite usar dados e objetos juntos

Este é outro ponto polêmico que muitos interpretam como manter nos objetos apenas propriedades enquanto seus comportamentos ficam em outros objetos.

Particularmente acho que a essência de um objeto é justamente o agrupamento de variáveis e funções (Propriedades e métodos). Neste ponto eu sempre mantenho os comportamentos nas entidades.

Em relação a manter parte com `object` e parte com `struct` eu confesso que a maior parte dos meus casos eu uso apenas o `object`. Pode ser vício ou puro comodismo, mas acho estranho esta mistura.

Talvez uma abordagem que aplique estes conceitos de uma forma legal seja novamente o uso dos **value objects**.

```
// Objeto de valor, representa um endereço, sua estrutura de dados
public class Address
{
    public string ZipCode { get; set; }
    public string Street { get; set; }
    public string Number { get; set; }
    public string Neighborhood { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }
}

// Objeto do cliente... com seus comportamentos
public class Customer
{
    public Address BillingAddress { get; private set; }
    public Address ShippingAddress { get; private set; }

    public void ChangeBillingAddress(Address address) { ... }
    public void ChangeShippingAddress(Address address) { ... }
}
```

Instanciar poucas variáveis

Evite instanciar muitas variáveis nos objetos e seus métodos. Faz uso maior das propriedades se possível.

Exemplo

```
public class ShoppingCart
{
    public decimal Total { get; private set; }

    public decimal CalculateTotal()
    {
        var total = 0; // Desnecessário
        foreach(var item in Items)
            total += item.Price;
    }
}
```

```
// Melhorando
public class ShoppingCart
{
    public decimal Total { get; private set; }

    private decimal CalculateTotal()
    {
        foreach(var item in Items)
            Total += item.Price;
    }
}
```

Classe base não deve saber sobre suas derivadas

Uma classe não deve saber sobre detalhes dos seus filhos. Nas verdade isto me soa tão estranho que não vejo um cenário onde uma classe pai consiga saber detalhes de seus filhos.

Exemplo

```
// N/A
```

Mais métodos, menos tomadas de decisão

Já comentamos bastante isto na parte de OOP dos cursos, mas fica aqui o reforço, sempre opte por ter mais métodos, mais sobrecargas do que tomadas de decisão.

Exemplo

```
// Evite
public class Order
{
    public void Pay(CreditCard card)
    {
        if(card == null)
            // Pagamento via boleto

        // Pagamento via cartão
    }
}
```

```
// Utilize
public class Order
{
    public void Pay()
    {
        // Pagamento via boleto
    }

    public void Pay(CreditCard card)
    {
        // Pagamento via cartão de crédito
    }
}
```

Evite métodos estáticos

Classes e métodos estáticos são difíceis de gerenciar, além de serem compartilhados entre a aplicação como um todo. Imagina que você tem uma classe estática que tem uma lista de notificações, esta lista seria compartilhada entre todas as requisições (Diversos usuários) em seu sistema.

```
// Evite
public static class NotificationContext
{
    public static IList<Notification> Notifications { get; set;}
}
```

```
// Utilize
public class NotificationContext
{
    public IList<Notification> Notifications { get; set;}
}
```

Testes

Um assert por teste

Utilize um e apenas um `assert` por teste. Mais de um assert pode confundir você e comprometer a escrita do seu teste.

```
// Evite
[TestMethod]
public void ShouldReturnTrue
{
    Assert.AreEqual(true);
    Assert.AreEqual(1);
}
```

```
// Utilize
[TestMethod]
public void ShouldReturnTrue
{
    Assert.AreEqual(true);
}
```

Legível

Trate seus testes como parte fundamental do seu código, não secundária. Os testes tem que ser organizados e bem escritos assim como o resto do seu software.

Exemplo

```
// N/A
```

Rápido

Um dos objetivos principais de um teste é cobrir uma pequena porção do nosso código. Normalmente estendemos esta ideia para a maior parte do código possível, ocasionando uma ampla gama de testes de unidade.

Dados estes testes, os mesmo são executados antes da publicação das nossas aplicações, garantindo que não enviaremos nada com bugs para produção.

Porém, em cenários mais críticos, o tempo dos deploys (Publicações) é extremamente importante, e se nossos testes demoram muito, podem impactar negativamente nisto.

Exemplo

```
// N/A
```

Independentes

Os testes não devem depender de entidades externas, nem de outros testes. Neste exemplo, volto a salientar o uso do DI e DIP.

Exemplo

Repetitível

Devemos ter a possibilidade de repetir o mesmo teste, mas com parâmetros diferentes.

Exemplo

```
[TestMethod]
[DataRow("email@valido.com", "email@balta.io")]
public void ShouldValidateEmail(string email)
{
    Assert.IsTrue(new Email(email).IsValid());
}
```

Code smells

Code Smells são alguns sintomas que podemos identificar e que nos remetem a uma má aplicação do Clean Code de uma forma geral.

Rigidez

Seu software é difícil de mudar. Qualquer mudança, por mínima que seja, causa uma cascata de outras mudanças.

Fragilidade

Uma simples mudança quebra seu software em diversos locais. É o famosos "cobre o pé, descobre a cabeça".

Imobilidade

Você não consegue reutilizar partes do seu código em outros projetos por que isto requer um esforço gigantes. Em resumo, tudo está muito acoplado.

Complexidade desnecessária

Você usa padrões e arquiteturas que tornam seu código mais burocrático do que efetivo. É o famoso "e se", onde pensamos em tudo que o software pode ter um dia e já "deixamos tudo pronto".

"E se eu quiser voar com meu carro um dia?", bem, se um dia você quiser voar, aí você constrói as asas, mas se não vai precisar voar agora, foca em construir apenas o carro.

Repetição desnecessária

Você precisa repetir o mesmo código em diversos lugares.

Opacidade

Seu código é difícil de entender.

Fontes

- Post em Inglês
- Livro Clean Code

Gostou deste artigo sobre **Clean Code?**