Curso: Fundamentos de Engenharia de Software

Sumário

[01 – Introdução 5](#_Toc64917722)

[01-01 – Apresentação 5](#_Toc64917723)

[01-02 - O que é arquitetura 5](#_Toc64917724)

[Evolução 5](#_Toc64917725)

[Conceitos 5](#_Toc64917726)

[02 - Perfis de Arquitetos 5](#_Toc64917727)

[02-01 - Perfis de arquitetos 5](#_Toc64917728)

[O arquiteto 5](#_Toc64917729)

[02-02 - Arquiteto Corporativo 6](#_Toc64917730)

[02-03 - Arquiteto de Negócios 6](#_Toc64917731)

[02-04 - Arquiteto de Soluções 7](#_Toc64917732)

[02-05 - Arquiteto de Software 7](#_Toc64917733)

[02-06 - Outros perfis de arquitetos 7](#_Toc64917734)

[03 - Perfil do Arquiteto de Software 7](#_Toc64917735)

[03-01 - Responsabilidades 7](#_Toc64917736)

[03-02 - Requisitos Técnicos 7](#_Toc64917737)

[03-03 - Requisitos Pessoais 8](#_Toc64917738)

[03-04 - Mitos sobre o Arquiteto de Software 8](#_Toc64917739)

[04 - OOP 8](#_Toc64917740)

[04-01 - Pilares da Programação Orientada a Objetos 8](#_Toc64917741)

[04-02 - Estado e Comportamento 9](#_Toc64917742)

[04-03 - Herança 9](#_Toc64917743)

[04-04 - Abstração 9](#_Toc64917744)

[04-05 - Polimorfismo 9](#_Toc64917745)

[04-06 - Encapsulamento 9](#_Toc64917746)

[Modificadores de acesso 9](#_Toc64917747)

[04-07 - Interface x Implementação 9](#_Toc64917748)

[04-08 - Herança x Composição 9](#_Toc64917749)

[05 - SOLID 9](#_Toc64917750)

[05-01 - Princípios SOLID 9](#_Toc64917751)

[05-02 - SRP - Single Responsability Principle 10](#_Toc64917752)

[05-03 - OCP – Open Closed Principle 10](#_Toc64917753)

[05-04 - LSP- Liskov Substitution Principle 10](#_Toc64917754)

[05-05 - ISP - Interface Segregation Principle 10](#_Toc64917755)

[05-06 - DIP - Dependency Inversion Principle 10](#_Toc64917756)

[06 - Dependency Injection 11](#_Toc64917757)

[06-01 - Exemplo do 'mundo real' 11](#_Toc64917758)

[06-02 - Tipos de ciclo de vida 11](#_Toc64917759)

[06-03 - Registro de generics 11](#_Toc64917760)

[06-04 - Property Injection 11](#_Toc64917761)

[06-05 - Service Locator 'Pattern' 12](#_Toc64917762)

[06-06 - N Classes - 1 Interface 12](#_Toc64917763)

[07 - Clean Code 12](#_Toc64917764)

[07-01 - Apresentação 12](#_Toc64917765)

[07-02 - O que é um código limpo? 12](#_Toc64917766)

[07-03 - Desculpas e responsabilidades 13](#_Toc64917767)

[Desculpas: 13](#_Toc64917768)

[E de quem é a culpa? 13](#_Toc64917769)

[Verdades que não lhe dizem 13](#_Toc64917770)

[Quanto custa um código ruim? 13](#_Toc64917771)

[07-04 – Como medir um bom código? 13](#_Toc64917772)

[Medir bom código 13](#_Toc64917773)

[Nomes significativos 13](#_Toc64917774)

[Dicas: 14](#_Toc64917775)

[07-05 - Boas práticas 14](#_Toc64917776)

[Boas práticas 14](#_Toc64917777)

[Menos e mais 14](#_Toc64917778)

[Métodos 14](#_Toc64917779)

[07-06 - Devo comentar meu código? 14](#_Toc64917780)

[Comentários 14](#_Toc64917781)

[Quando comentar? 14](#_Toc64917782)

[07-07 - Tratamento de erros 15](#_Toc64917783)

[08 - Design Patterns 15](#_Toc64917784)

[08-01 – Apresentação 15](#_Toc64917785)

[Patterns do Gof 15](#_Toc64917786)

[Os padrões do GoF estão divididos em 3 famílias: 15](#_Toc64917787)

[08-02 - Creational Patterns 15](#_Toc64917788)

[Alguns Patterns 16](#_Toc64917789)

[08-03 - Abstract Factory 16](#_Toc64917790)

[08-04 - Factory Method 16](#_Toc64917791)

[08-05 – Singleton 16](#_Toc64917792)

[08-06 - Structural Patterns 16](#_Toc64917793)

[Alguns Patterns 16](#_Toc64917794)

[08-07 – Adapter 16](#_Toc64917795)

[08-08 – Facade 16](#_Toc64917796)

[08-09 – Composite 16](#_Toc64917797)

[08-10 - Behavorial Patterns 17](#_Toc64917798)

[Alguns patterns 17](#_Toc64917799)

[08-11 – Command 17](#_Toc64917800)

[08-12 – Strategy 17](#_Toc64917801)

[08-13 – Observer 17](#_Toc64917802)

[08-14 - Evite o Patternite 17](#_Toc64917803)

[09 - Arquitetura de Software 17](#_Toc64917804)

[09-01 - Estilos Arquiteturais 17](#_Toc64917805)

[Estilos Arquiteturais 17](#_Toc64917806)

[Estilos 17](#_Toc64917807)

[SOA vs Microservices: 17](#_Toc64917808)

[Outros Estilos: 18](#_Toc64917809)

[09-02 - Padrões Arquiteturais 18](#_Toc64917810)

[09-03 - 3-Tier Architecture 18](#_Toc64917811)

[09-04 - Onion Architecture 18](#_Toc64917812)

[09-05 - Hexagonal Architecture 'Ports & Adapters' 19](#_Toc64917813)

[09-06 - CQRS - Command Query Responsibility Segregation 20](#_Toc64917814)

[Definição: 20](#_Toc64917815)

[Revisão dos conceitos de CQRS: 20](#_Toc64917816)

[Anotações 20](#_Toc64917817)

[09-07- Event Sourcing 20](#_Toc64917818)

[Definições 20](#_Toc64917819)

[09-08 - DDD - Domain-Driven Design 21](#_Toc64917820)

[Processo de “implementação” do DDD: 21](#_Toc64917821)

[Linguagem Ubíqua: 21](#_Toc64917822)

[Modelagem Estratégica 21](#_Toc64917823)

[Modelagem Tática: 21](#_Toc64917824)

[09-09 - Arquiteturas Evolutivas 22](#_Toc64917825)

[09-10 - Sempre considere a complexidade! 22](#_Toc64917826)

[09-11 - Conway's Law 22](#_Toc64917827)

[09-12 - Agilidade e o Manifesto Ágil 22](#_Toc64917828)

[Princípios por Trás do Manifesto Ágil 22](#_Toc64917829)

[09-13 – DevOps 23](#_Toc64917830)

[09-14 - Principios DRY, KISS e YAGNI 23](#_Toc64917831)

[Don’t Repeat Yourself 23](#_Toc64917832)

[Keep It Simple, Stupid 23](#_Toc64917833)

[You Ain’t Gonna Need It 23](#_Toc64917834)

[09-15 - Leituras recomendadas 23](#_Toc64917835)

# 01 – Introdução

## 01-01 – Apresentação

...

## 01-02 - O que é arquitetura

### Evolução

Arquitetura é a **organização** fundamental de um sistema incorporada em seus **componentes**, seus **relacionamentos** com o ambiente, e os princípios que conduzem seu **design** e **evolução.** [ISSO/IEEE 42010-2011]

Como conceituar a arquitetura?

Como definir a arquitetura?

Como gerir uma governança de arquitetura?

### Conceitos

* TOGAF (The Open Group Architecture Framework)
* ISO/IEC/IEEE 42010

# 02 - Perfis de Arquitetos

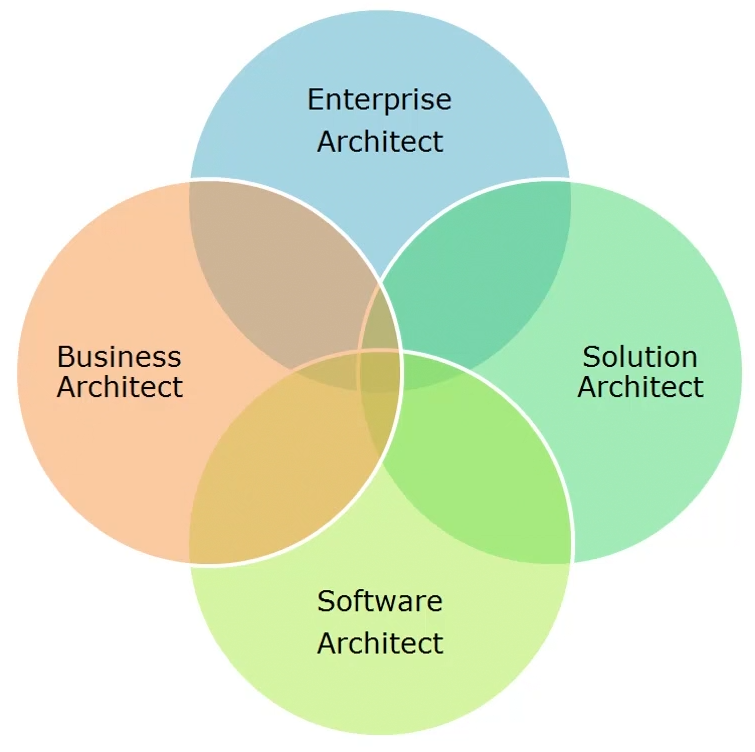
## 02-01 - Perfis de arquitetos

### O arquiteto

Um especialista em soluções para o desenvolvimento. Possui uma visão madura e aguçada

De acordo com a IASA (Global IT Achitects Association) existem diversos perfis de arquitetos, cada um com conjuntos de responsabilidades e focos profissionais específicos.

<http://iasagloba.org>



The Architect Journal – A study of Architect Roles By IASA Sweden

## 02-02 - Arquiteto Corporativo

Está mais ligado a empresa do que a entrega de software

* Objetivo
  + Garantir a entrega do negócio com soluções em TI
* Artefatos
  + Estratégias de TI, mapas de capacidade, estratégias de integração. Análise as-it/to be, analise de gap
* Conceito chave
  + Alinhamento entre TI e negócios

The zachman framework

## 02-03 - Arquiteto de Negócios

* Objetivo
  + Modelagem do negócio da organização entendo os processos atuais sugerindo melhorias
* Artefatos
  + Mapa de processos, casos de uso e modelos informais
* Conceito Chave
  + Entender o funcionamento da organização

BPMN Bussines Process Model and Notation

## 02-04 - Arquiteto de Soluções

Arquiteto mais perto do desenvolvedor

* Objetivo
  + Projetos de soluções em TI baseado nos requisitos do negócio, utilizando as capacidades de TI existentes na organização
* Artefatos
  + Diagrama de aplicações, mapas de sistema, interfaces de serviços, interfaces de serviços técnicos
* Conceito chave
  + Suportar a estratégia de negócio

## 02-05 - Arquiteto de Software

* Objetivo
  + Projetar aplicações utilizado conceitos de arquitetura e boas práticas de desenvolvimento
* Artefatos
  + Frameworks, arquiteturas / modelos de referência, análise de cenários / desafios técnicos
* Conceito Chave
  + Qualidade, Flexibilidade, Performance, Reuso, Testabilidade, Escalabilidade e Segurança

Imagem: Arquitetura Hexagonal

## 02-06 - Outros perfis de arquitetos

Outros perfis

* Arquiteto de Dados
* Arquiteto de TI
* Arquiteto de Nuvem
* Et..

Em geral estes perfis são criados pelas próprias empresas para enquadras seus profissionais conforme sua visão.

Muitos destes perfis se encaixam em algum dos previamente apresentados

# 03 - Perfil do Arquiteto de Software

## 03-01 - Responsabilidades

* Objetivo principal de um arquiteto de software é projetar uma solução compatível com os requisitos atuais da corporação empregadora, que tenha flexibilidade suficiente para comportar mudanças futuras ou novos requisitos resultantes de sua evolução ao longo do tempo.
* Ser responsável pela qualidade técnica do entregável, atuar no apoio técnico de toda equipe e ajudar nas decisões necessárias.
* Promover a comunicação do time, envolvendo e delegando as decisões de arquitetura com toda equipe.

## 03-02 - Requisitos Técnicos

Hard Skills

* O principal requisito técnico de um arquiteto de software é o profundo conhecimento em programação e componentização. Conhecimentos adquiridos através de experiências sólidas em projetos.
* Dominar amplamente alguma plataforma de desenvolvimento, frameworks, ferramentas e modelagem técnica.
* Conhecer abordagens de arquitetura e padrões para soluções de problemas.
* Conhecer as técnicas de design de código e qualidade para permitir uma fácil manutenção e testabilidade.

## 03-03 - Requisitos Pessoais

Soft Skills

* Liderança
* Comunicação
* Proatividade
* Mindset empreendedor
* Humildade
* Tomar boas decisões
* Estar aberto para críticas e sugestões
* Ser um profundo pesquisador
* Ser didático
* Saber delegar tarefas
* Saber questionar
* Saber conduzir discussões
* Saber conversar com diferentes níveis hierárquicos
* Estar sempre aberto às novas tecnologias
* Estar sempre atualizado com o mercado
* Ter maturidade e equilíbrio emocional
* Ter pensamento estratégico
* Reconhecer e assumir os próprios erros e de sua equipe

## 03-04 - Mitos sobre o Arquiteto de Software

Mitos:

* O arquiteto não programa, apenas projeta, documento e lidera.
* Um desenvolvedor sênior pode assumir o papel de arquiteto.
* O arquiteto é um desenvolvedor com muito tempo de experiência.
* Para ser arquiteto basta estudar técnicas de arquitetura de software.
* O arquiteto trabalha em ambiente isolado ou diferente da equipe.
* O arquiteto é a autoridade técnica e não deve ser questionado.
* O arquiteto é um gerente técnico.
* O arquiteto é um professional com graduação em arquitetura de software
* O arquiteto é o profissional que está na liderança de um projeto

# 04 - OOP

# 04-01 - Pilares da Programação Orientada a Objetos

É essencial possuir um claro conhecimento dos princípios da orientação a objetos para poder aplicar as melhores práticas de design de código, padrões e abordagens de arquitetura.

Pilares fundamentais da OOP

* Estado
* Comportamento
* Abstração
* Herança
* Polimorfismo
* Encapsulamento
* Estado

Na orientação a objetos devemos sempre buscar o baixo acoplamento e a alta coesão entre os objetos

* Acoplamento:
  + Dependência direta entre um objeto e outro. Quando um objeto muda o outro muda por consequencia.
* Coesão:
  + Objetos executados uma única responsabilidade livres de uma dependência direta com um outro objeto.

## 04-02 - Estado e Comportamento

Prático..

## 04-03 - Herança

Prático..

## 04-04 - Abstração

Prático..

## 04-05 - Polimorfismo

Prático..

## 04-06 - Encapsulamento

### Modificadores de acesso

* public:
  + O acesso não é restrito.
* protected:
  + O acesso é limitado à classe ou aos tipos derivados da classe que os contém.
* internal:
  + O acesso é limitado ao assembly atual.
* protected internal:
  + O acesso é limitado ao assembly atual ou aos tipos derivados da classe contida.
* private:
  + O acesso é limitado ao tipo de conteúdo.
* private protected:
  + O acesso é limitado à classe ou tipos de conteúdo derivados da classe de conteúdo no assembly atual. Disponível desde C # 7.2.

## 04-07 - Interface x Implementação

Prático..

## 04-08 - Herança x Composição

Prático..

# 05 - SOLID

## 05-01 - Princípios SOLID

SOLID é o acrônimo dos cinco primeiros princípios da programação orientada a objetos e design de código identificados por Robert C. Margin (ou Uncle Bob) por volta do ano 2020

Os princípios SOLID devem ser aplicados para obter os benefícios da orientação a objetos, tais como:

* Seja fácil de se manter, adaptar e se ajustar as alterações do escopo;
* Seja testável e de fácil entendimento;
* Seja extensível para alterações com o menor esforço necessário;
* Que forneça o máximo de reaproveitamento;
* Que permaneça o máximo de tempo possível em utilização

Utilizando os princípios SOLID é possível evitar muitos problemas comuns

* Dificuldade na testabilidade / criação de testes de unidade;
* Código macarrônico, sem estrutura ou padrão;
* Dificuldades de isolar funcionalidades;
* Duplicação de código, uma alteração precisa ser feita em N pontos;
* Fragilidade, o código quebra facilmente sem vários pontos após alguma mudança.

## 05-02 - SRP - Single Responsability Principle

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

Uma classe deve ter um, e apenas um, motivo para ser modificada

## 05-03 - OCP – Open Closed Principle

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

“Entidades de software (classes, módulos, funções, etc) devem estar abertas para extensão, mas fechadas para modificação.”

Utilização do Strategy Pattern

## 05-04 - LSP- Liskov Substitution Principle

“Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S, where S is a subtype of T.”

“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.” “Uma classe base deve poder ser substituída pela sua classe derivada.”

*Se nada como um pato, voa como um pato, porém precisa de baterias, provavelmente você possui um problema de abstração*

Seria a “Prova Real” da Herança

## 05-05 - ISP - Interface Segregation Principle

“States that no client should be forced to depend on methods it does not use “

“Clientes não devem ser forçados a depender de métodos que não usam.”

*Muitas interfaces específicas são melhores do que uma interface única.*

Segregação de interfaces

## 05-06 - DIP - 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. “

“Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações; Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.”

Dependa de uma abstração e não de uma implementação.

# 06 - Dependency Injection

## 06-01 - Exemplo do 'mundo real'

Código...

Containers no .NET

IoC Container Benchmark – Performance comparison

* <https://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison>

*Código..*

## 06-02 - Tipos de ciclo de vida

Documentação

* <https://docs.microsoft.com/pt-br/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2#service-lifetimes>

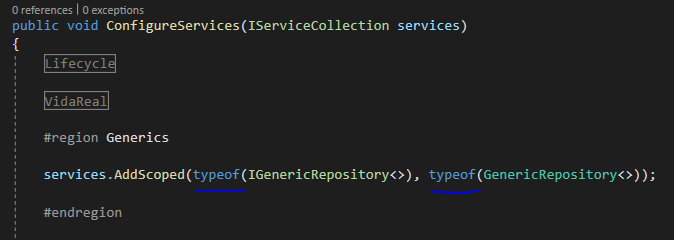
Tipos:

* Transient
* Scoped
* Singleton

*Código..*

## 06-03 - Registro de generics

*Código..*

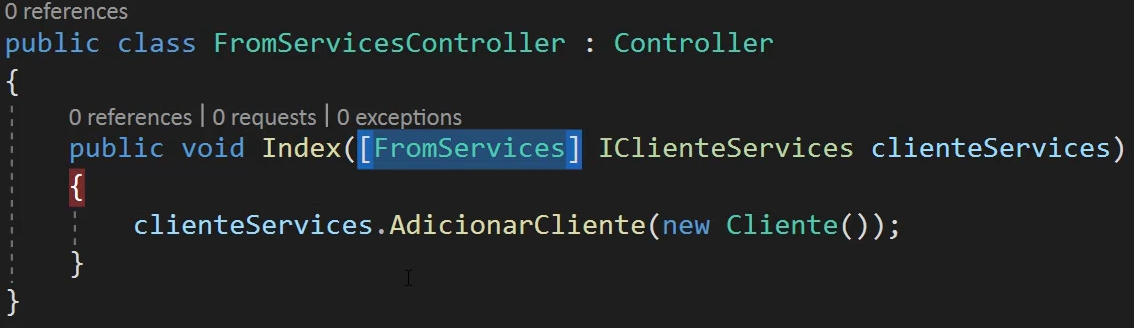


## 06-04 - Property Injection

Cenários para usar Property Injection

* <https://simpleinjector.readthedocs.io/en/latest/advanced.html#property-injection>

Usar [FromServices] antes da tipagem dos paramentos



## 06-05 - Service Locator 'Pattern'

Forma de fazer inversão de controler, utilizando injeção de dependência, mas de uma forma diferente

* Inteface: IServiceProvider
  + <https://docs.microsoft.com/pt-br/dotnet/api/system.iserviceprovider?view=net-5.0>

É uma forma de você ter acesso ao container e você mesmo obter o serviço.

## 06-06 - N Classes - 1 Interface

Documentação:

* <https://simpleinjector.readthedocs.io/en/latest/howto.html#resolve-instances-by-key>

Nota:

* A necessidade de registro codificado pode ser uma indicação de ambiguidade no design do aplicativo e um sinal de uma violação do [Princípio de Substituição de Liskov](https://en.wikipedia.org/wiki/Liskov_substitution_principle) . Dê uma boa olhada se cada registro com chave não deve ter sua própria interface exclusiva, ou talvez cada registro deva implementar sua própria versão de uma interface genérica.

# 07 - Clean Code

## 07-01 - Apresentação

**Livro Clean Code**

* Uma leitura obrigatória para desenvolvedores de qualquer plataforma e nível de experiência profissional.

Definições para código limpo que alguns programadores bem conhecidos e experientes dizem:

* Elegante e eficiente. Código limpo faz bem uma coisa
  + Bjarne Stroustrup
* Simples e direto. Pode ser lido como uma conversa
  + Grady Booch
* Parece ter sido escrito por alguém que se importa
  + Michael Feathers
* Cada rotina que você lê faz o que você espera
  + Ward Cunningham

## 07-02 - O que é um código limpo?

Características

* Simples
* Direto
* Eficiente
* Sem duplicidade
* Elegante
* Feito com cuidado
* Fácil de ler

Qualquer tolo consegue escrever código que um computador entenda. Bons programadores escrevem código que humanos possam entender. – Martin Fowler

## 07-03 - Desculpas e responsabilidades

### Desculpas:

* Mas o cronograma está apertado!
* Meu chefe me pressiona a entregar logo!
* Quero mostrar produtividade!
* Não ganho o suficiente para escrever o melhor código do mundo!
* A empresa não valoriza bom código e sim a entrega!

### E de quem é a culpa?

* É de todos

### Verdades que não lhe dizem

* A sua carreira é sua responsabilidade!
* Leia, Estudo e Pratique
* Vá a conferências
* Faça cursos

Não é da responsabilidade do seu empregador lhe oferecer cursos, oportunidades ou até mesmo um projeto com a tecnologia que pretende se especializar.

“Você recebe para trabalhar 40 horas por semana e rsolver os problemas da sua empresa, não os seus ...” – Uncle Bob

### Quanto custa um código ruim?

* Alta rotatividade
* Demo na entre de novas funcionalidades
* Dificuldade na manutenção
* Alta indecência de bugs
* Perda de confiança do cliente
* Desmotivação profissional
* Mais tempo depurando código do que escrevendo

## 07-04 – Como medir um bom código?

### Medir bom código

* Linhas de código
* Número de métodos
* Número de classes
* Linhas de código por método
* Complexidade ciclomática (cada if)
* Número de estrutura de decisão

### Nomes significativos

* Escolha os nomes que revelam intenção!
* Por que existe?
* O que faz?
* Como é usado?

### Dicas:

* Usar nomes fáceis de se encontrat
* Usar nomes pronunciáveis
* Evitar siglas ou acrônimos
* Não economizar palavras
* Revelar a intenção do código
* Evitar palavras que podem ser variáveis ou palavras reversadas em outras plataformas
* Evitar dar nomes domo “doubleValorPromocional”, o tipo não precisa estar no nome
* Evitar trocadilhos, não misture idioma, não mesclar nomes.

## 07-05 - Boas práticas

### Boas práticas

* Nome de classes devem ser substantivos e não devem conter verbos:
  + *ClienteRepository*
* Nome dos métodos deve conter verbos de preferência no infinitivo.
  + *AdicionarCliente*
* Não seja genérico.
  + Ex: Ao invés de Process(), usar ProcessarFolhaPagamento()

### Menos e mais

* A primeira regra dos métodos é que eles devem ser pequenos. A segunda regra é que eles devem ser menores ainda. – Uncle Bob
* Método <= 20 linhas
* Linha <= 100 caracteres
* Classe <= 500 linhas

### Métodos

* Extraia trechos em métodos privados.
* Métodos devem fazer apenas uma coisa, fazê-la certa e somente fazê-la.
* Evitar muitos parâmetros
* Não deixe o método mentir dizendo que faz uma coisa e faz outras “escondidas”
* Se o método tiver mais de uma responsabilidade extraia em dois ou mais.
* Leia seu método de cima para baixo como uma narrativa, ele deve fazer sentido.
* Aplique uma boa identação.

## 07-06 - Devo comentar meu código?

### Comentários

* Comentários não vão ajudar um código ruim ser melhor interpretado.
* Um código que requer comentário, precisa ser reescrito
* Não deixe trechos de código comentado

### Quando comentar?

* Alertar consequências que pode vir a causar
* Licença, direitos autorais, etc.
* Necessidade de explicar uma a regra de negócios interna.
* Decisões de design de código

## 07-07 - Tratamento de erros

Tratamento de erros

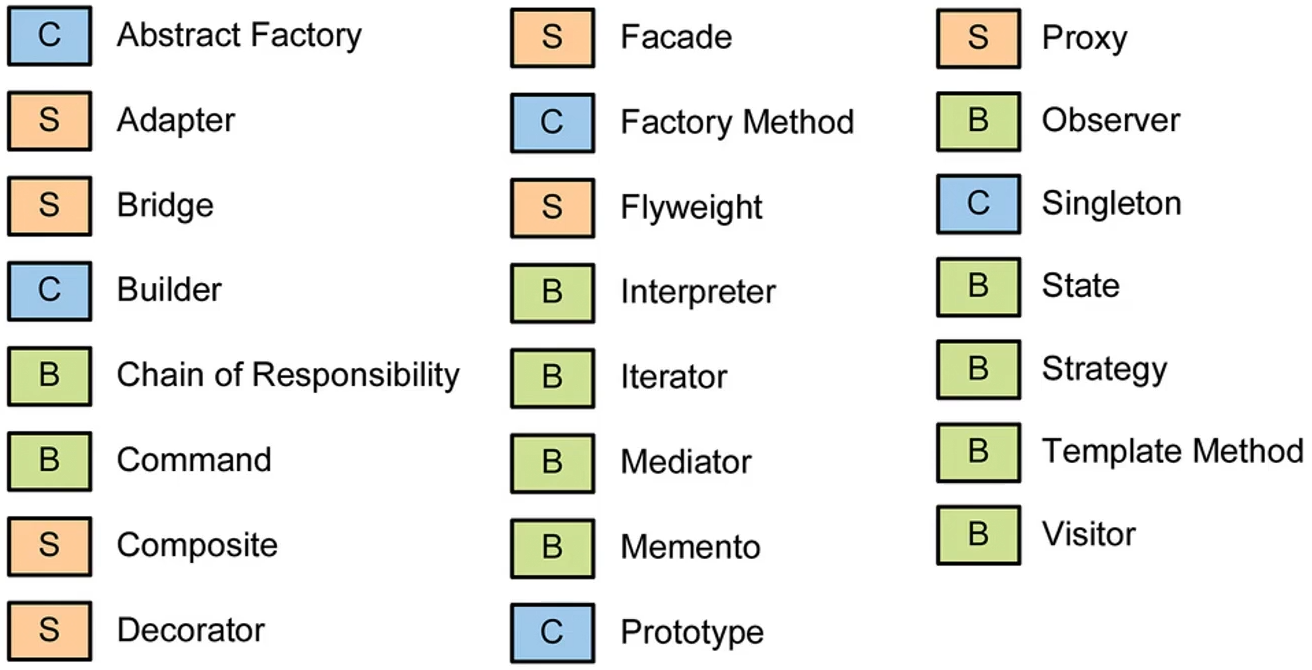
* Tratar e prever possíveis exceções é de responsabilidade do desenvolvedor.
* Retorne *Exceptions* e não códigos de erro.
* Informe o máximo que puder em sua *Exception*
* Se necessário crie *Exceptions* personalizadas para um problema específico.
* Não retorne null
* Regra dos escoteiros
  + “Deixe a área de acampamento mais limpa de como você a encontrou”

# 08 - Design Patterns

## 08-01 – Apresentação

* Design Patterns são padrões de código para solução de problemas conhecidos.
* O objetivo é não reinventar a roda e aplicar uma solução com um bom design de código
* O conceito de padrões foi introduzido por 4 desenvolvedores intitulados “Gang of Four”(GoF) e hoje conta com 23 padrões fundamentais.
* Atualmente existem mais de 80 padrões conhecidos que são em geral variações dos 23 patterns go GoF

### Patterns do Gof



### Os padrões do GoF estão divididos em 3 famílias:

* **Creational Patterns (Criacional)**
  + Fornecem meios de criação de um objeto e de como ele será instanciado
* **Structural Patterns (Estrutural)**
  + Tratam da composição de objetos por heranças e interfaces para diferentes funcionalidades.
* **Behavioral Patterns (Comportamental)**
  + Tratam das interações e comunicação entre os objetos além da divisão de responsabilidades.

## 08-02 - Creational Patterns

### Alguns Patterns

* **Abstract Factory**
  + Cria uma instância de diversas famílias de classes.
* **Factory Method**
  + Cria uma instancia de diversas derivações
* **Singleton**
  + Cria uma única instancia que será utilizada por todos os recursos.

Documentação

* <https://dofactory.com/net/design-pattern>

## 08-03 - Abstract Factory

Resumo: Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas

*Código...*

## 08-04 - Factory Method

Resumo: Defina uma interface para criar um objeto, mas deixe as subclasses decidirem qual classe instanciar. O Factory Method permite que uma classe adie a instanciação para as subclasses.

*Código...*

## 08-05 – Singleton

Resumo: Certifique-se de que uma classe tenha apenas uma instância e forneça um ponto global de acesso a ela

*Código...*

## 08-06 - Structural Patterns

### Alguns Patterns

* **Adapter**
  + Compatibiliza objetos de interfaces diferentes
* **Facade**
  + Uma única classe que representa um subsistema
* **Composite**
  + Compartilha um objeto em estruturas de árvores que representa hierarquiras

## 08-07 – Adapter

Converta a interface de uma classe em outra interface esperada pelos clientes. O adaptador permite que as classes trabalhem juntas de outra forma, devido a interfaces incompatíveis

*Código..*

## 08-08 – Facade

Fornece uma interface unificada para um conjunto de interfaces em um subsistema. Façade define uma interface de nível superior que torna o subsistema mais fácil de usar.

*Código...*

## 08-09 – Composite

Componha objetos em estruturas de árvore para representar hierarquias parte-todo. O Composite permite que os clientes tratem objetos individuais e composições de objetos uniformemente

*Código..*

## 08-10 - Behavorial Patterns

### Alguns patterns

* **Command**
  + Encapsula um *command* request em um objeto
* **Strategy**
  + Encapsula um algoritmo dentro de uma classe
* **Observer**
  + Uma forma de notificar mudanças a uma séria de classes

## 08-11 – Command

Encapsula uma solicitação como um objeto, permitindo, assim, parametrizar clientes com diferentes solicitações, solicitações de fila ou log e oferecer suporte a operações que podem ser desfeitas

*Código...*

## 08-12 – Strategy

Defina uma família de algoritmos, encapsule cada um e torne-os intercambiáveis. A estratégia permite que o algoritmo varie independentemente dos clientes que o utilizam

*Código...*

## 08-13 – Observer

Defina uma dependência um-para-muitos entre objetos para que, quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente

*Código...*

## 08-14 - Evite o Patternite

* Pertternite é o mal que o desenvolvedor sofre quando deseja aplicar todos os patterns conhecidos apenas para praticar ou por entender que quanto mais patterns melhor.
* Não utilize sem conhecer
* Apesar de conhecer só utilize se necessário.

# 09 - Arquitetura de Software

## 09-01 - Estilos Arquiteturais

### Estilos Arquiteturais

* Um estilo arquitetural é uma **abordagem** de como **projetar** e **entregar** uma aplicação.
* Trata-se de como **organizar** os **componentes** responsáveis de uma arquitetura, como eles irão **interagir** entre si e quais **aspectos** tecnológicos irmão atender.

### Estilos

* **Arquitetura Monolítica:**
  + Tudo em um
* **Arquitetura em camadas**
  + Dividir responsabilidades em camadas específicas
* **Arquitetura REST**
  + Dividir em APIs
* **Arquitetura de Microservices**

### SOA vs Microservices:

...

### Outros Estilos:

* Plugin Architecture
* Client-Server Architecture
* Pipes and Filters Architecture

## 09-02 - Padrões Arquiteturais

Definições:

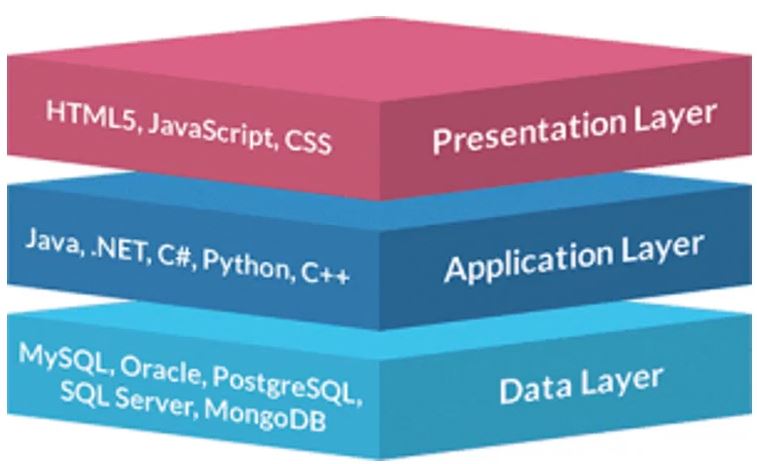
* Os Padrões Arquiteturais são **semelhantes** ao “Design patterns”, mas possuem um **escopo** diferente.
* Padrões Arquiteturais são **estratégias** de **alto nível** que dizem respeito a **componentes** de grande escala, as **propriedades** e **mecanismos globais** de um sistema.
* Um projeto de arquitetura pode conter **diversos estilos arquiteturais**. Um padrão arquitetural pode ser um **subconjunto** de um estilo arquitetural visando um **escopo** **específico**.
  + Ex: Arquitetura Web, Usando estilo Client-Server, Usando padrões, tipo MVC
* Um padrão arquitetural é uma **solução geral** e **reutilizável** para um problema em um contexto particular. É uma **solução recorrente** para um **problema recorrente**.
  + Ex:

## 09-03 - 3-Tier Architecture

Famosa arquitetura “3 camadas”

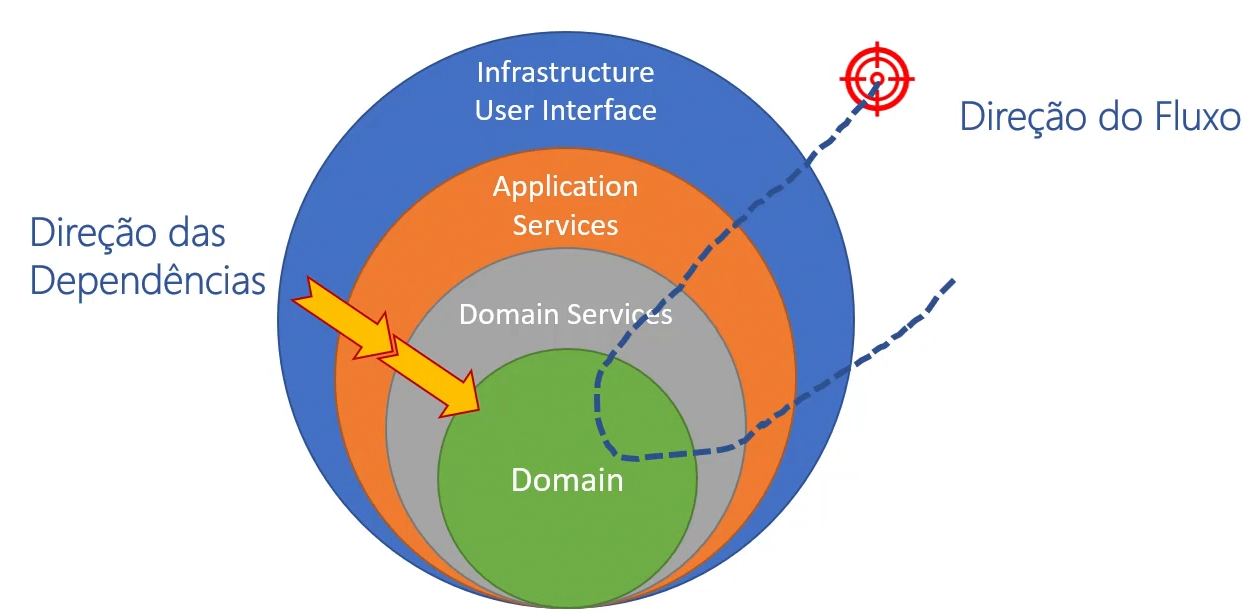
É um estilo e padrão arquitetural ao mesmo tempo. Estilo, no sentido de dividir em camadas. Padrão, porque se divide ela em três, como um responsabilidade cada.

* Clássica maneira de distribuir responsabilidades (apresentação, aplicação [negócios], acesso a dados).
* Não deve ser menosprezada, pois nem sempre a complexidade é uma solução para problemas simples.
* Pode ser aplicação em diversos cenários, porém geralmente é mais encontrada em aplicações com foco comercial (cadastros, regras e etc).



## 09-04 - Onion Architecture

Famosa arquitetura “cebola” – Clean Architecture



## 09-05 - Hexagonal Architecture 'Ports & Adapters'

A idea é unir Onion Arquitecutre, Domain-Driven Design, Conceitos de Clean Architecture, CQRS, Event Sourcing, Mensageria no mesmo lugar.

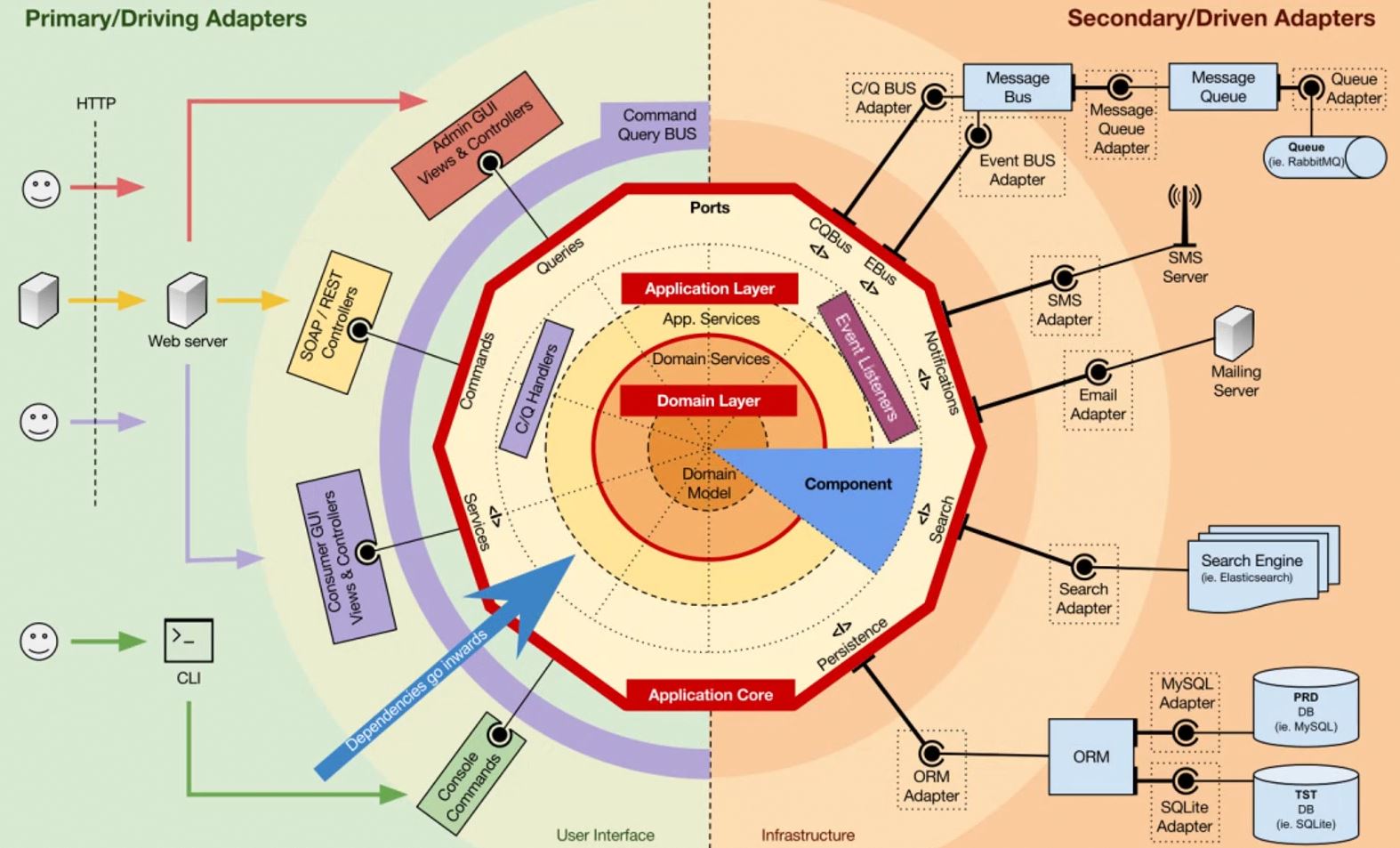
Arquitetura muito ampla.

Permite que se faça qualquer coisa

Ter qualquer nível de integração

Links:

* <https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/>



## 09-06 - CQRS - Command Query Responsibility Segregation

### Definição:

* Um **padrão arquitetural** onde o foco é seprar os meios de leitura e escrita de dados.
  + **Alterações** de dados são realizados via **Commands**
  + **Leitura** de dados são realizado via **Queries**
* O objetivo do CQRS é prover **expressividade** para aplicação, pois todos os Commands representam uma **intenção de negócio**
* CQRS promove a **consistência eventual**, que é quando possuímos um banco de **leitura** e outro de **escrita** com os mesmos dados, porém os dados não são consistidos exatamente no **mesmo momento**.
* Muito aplicado em arquiteturas hexagonais, microservices ou em aplicações que possuem uma **alta demanda** de consumo de dados.

### Revisão dos conceitos de CQRS:

* Commands:
  + Representam uma intenção de mudança no estado de uma entidade.
  + São expressivos e representam uma única intenção de negócio, ex:
    - AumentarSalarioFunacionarioCommand
* Queries
  + É uma forma de obter dados de um banco de origem de dados para atender as necessidades da aplicação.

### Anotações

Todo comando retorna um evento, sendo de sucesso ou de falha

## 09-07- Event Sourcing

### Definições

* Nós podemos buscar o estado de uma aplicação para encontrar o estado atual do mundo e isso responde muitas perguntas. Entretando há momentos há momentos que nós não só queremos ver onde nós estamos, mas também queremos saber como chegamos lá.” – Martin Fowler
* Event Sourcing assegura que todas as mudanças feitas no estado de uma aplicação são armazenadas como uma sequência de eventos. Não só podemos buscar esses eventos, mas também podemos usar este log de eventos para reconstruir estados passados e ajustar automaticamente o estado atual com mudanças retroativas
* A ideia central é persistir todos estados anteriores de uma entidade de negócio desde o momento de sua criação. Com estes dados é possível realizar o “replay” dos fatos passados para entender o comportamento do usuário, trabalhar com Big Data, Machine Learning, realizar testes de integração com cenários reais ou simplesmente recriar as entidades se necessário.

## 09-08 - DDD - Domain-Driven Design

* Introduzido em 2003 por Eric Evans
* Indicado para aplicações complexas, com muitas entidades e regras de negócio.
* Razoavelmente fácil de entender, difícil de aplicar.
* Um guia de como entender um negócio, organizá-lo em um conjunto de princípios, criar uma modelagem com base no negócio e implementar utilizando diversas boas práticas

### Processo de “implementação” do DDD:

* Entender o negócio
* Extrair a Linguagem Ubíqua
* Modelagem Estratégica
* Definir a Arquitetura
* Modelagem Tática

### Linguagem Ubíqua:

* Vocabulário de todos os termos específicos do domínio
  + Nomes, verbos, adjetivos, jargões, apelidos, expressões idiomáticas e advérbios
* Compartilhado por todas as partes envolvidas no projeto
  + Primeiro passo para evitar desentendimentos
* Usadas em todas as formas faladas e escritas de comunicação
  + A linguagem universal de um negócio é feira dentro da empresa

### Modelagem Estratégica

* Extrair a Linguagem Ubíqua vai colaborar na visão e entendimento do negócio como segregar seu domínio em partes menores e responsáveis.
* Para documentar estas segregações responsáveis utilizamos o Mapa de Contextos (Context Map) que pode ser representado através de imagens e uma simples documentação do tipo de relacionamento entre os contextos.
* Cada contexto delimitado possui uma própria Linguagem Ubíqua, pois em contexto diferentes, os termos podem ser significados diferentes.

### Modelagem Tática:

* **Aggregate Object**
  + Uma entidade que é a raiz agregadora de um processo do domínio que envolve mais de uma entidade.
* **Domain Model**
  + Uma entidade do domínio, possui estados e comportamentos, lógica de negócio, getters e setters AdHoc, etc.
* **Value Object**
  + Um objeto que agrega valor às entidades, não possui identidade e é imutável.
* **Factory**
  + Classe responsável por construir adequadamente um objeto / entidade.
* **Domain Service**
  + Serviço do domínio que atende partes do negócio que não se encaixam em entidades específicas, trabalha com diversas entidades, realiza persistência através de repositórios e etc.
* **Application Service**
  + Serviço de aplicação que orquestra ações disparadas pela camada de apresentação e fornece DTOs para comunicação entre as demais camadas e para o consumo da camada de apresentação.
* **Repository**
  + Uma classe que realiza a persistência das entidades se comunicando diretamente com o meio de acesso aos dados, é utilizado apenas um repositório por agregação.
* **External Service**
  + Serviço externo que realiza a consulta/persistência de informações por meios diversos.

## 09-09 - Arquiteturas Evolutivas

* “Um arquiteto permite que decisões importantes sejam adiadas e um bom arquiteto maximiza o número de decisões não tomadas.” – Uncle Bob
* “Uma arquitetura evolutiva suporta mudanças contínuas e incrementais como um primeiro princípio por meio de vários aspectos”. – Rebecca Parsons
* “Mudanças são inevitáveis. A evolução, no momento, é opcional.” – Tony Robbis

## 09-10 - Sempre considere a complexidade!

* **Acidental**
  + A complexidade acidente é aquela que sugere durante o processo de desenvolvimento, ou seja, ela é **CAUSADA** pela abordagem escolhida para resolver o problema.
* **Essencial**
  + Já a essencial é basicamente a complexidade que nosso “software” se propõe resolver.
  + SIM, infelizmente existem problemas complexos, e é neles que temos que focar.

## 09-11 - Conway's Law

* Qualquer empresa que projeta um sistema, inevitavelmente produz um projeto cuja estrutura é uma cópia da estrutura de comunicação da organização – Melvin Conway

## 09-12 - Agilidade e o Manifesto Ágil

<https://agilemanifesto.org>

* **Indivíduos e interações** mais que processos e ferramentas
* **Software em funcionamento** mais que documentação abrangente
* **Colaboração com o cliente** mais que negociação de contratos
* **Responder a mudanças** mais que seguir um plano

### Princípios por Trás do Manifesto Ágil

* Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada  
  de software com valor agregado.
* Aceitar mudanças nos requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adequam a mudanças, par que o cliente possa tirar vantagens competitivas.
* Entregar software funcionando, com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos.
* Pessoas relacionadas à negócios e desenvolvedores devem trabalhar em conjunto e diariamente, durante todo curso do projeto
* Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho.
* O Método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara.
* Software funcional é a medida primária de progresso.
* Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes.
* Contínua atenção à excelência técnica e bom design, aumenta a agilidade.
* Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feiro.
* As melhoras arquiteturas, requisitos e designs emergem de times auto-organizáveis.
* Em intervalor regulares, o time reflete em como ficar mais efetivo, então, se ajustam e otimizam seu comportamento de acordo.

## 09-13 – DevOps

* O termo DevOps deriva da junção das palavras “desenvolvimento” (development) e “operações” (operations), send uma prática de engenharia de software que possui o intuito de **unificar** o **desenvolvimento** de software (Dev) e a **operação** de software (Ops).
* A característica principal do movimento DevOps é defender fortemente a **automação** e **monitoramento** em todas as fases da **construção** do software, de integração, teste, liberação para implantação e gerenciamento e infraestrutura.
* DevOps pretende fornecer, em **ciclos** de desenvolvimento menores, frequência de **implantação** aumentada, **liberações** mais seguras, em alinhamento próximo com os **objetivos de negócio**.

## 09-14 - Principios DRY, KISS e YAGNI

### Don’t Repeat Yourself - Não repita a si mesmo

Cada parte do conhecimento deve ter uma representação única, não ambígua e definitiva dentro da aplicação.

### Keep It Simple, Stupid - Mantenha simples, estúpido

O KISS valoriza a simplicidade do projeto e defende que toda a complexidade desnecessária seja descartada.

Não adicionar complexidade desnecessária.

### You Ain’t Gonna Need It - Você não vai precisar

Uma orientação que sugere não adicionar funcionalidades ao código fonte de uma aplicação até que estas sejam necessárias

Não adicionar funcionalidades que não foram solicitadas. Faça apenas o que foi pedido!

## 09-15 - Leituras recomendadas

* Clean Code (Robert C. Martin)
* Clean Architecture (Robert C. Margin)
* Clean Coder (Robert C. Margin)
* Patterns of Enterprise Application Architecture (Martin Fowler)
* Computer Science (Robert Sedgewick Kevin Wayne)
* Domain-Driven Design (Eric Evan)
* Implementing Domain Driven Design (Voughn Veron)
* Domain-Driven Distilled (Voughn Veron)

## 10-01 – Palavras finais