Modelando Domínios Ricos

Sumário

[01 – Introdução 5](#_Toc66547107)

[01-01 - Apresentação 5](#_Toc66547108)

[01-02 - Objetivos do Curso 5](#_Toc66547109)

[02 - Visão Geral do DDD 5](#_Toc66547110)

[02-01 - DDD não é arquitetura de camadas 5](#_Toc66547111)

[02-02 - Quanto devo implementar o DDD? 5](#_Toc66547112)

[Big Ball of Mud 5](#_Toc66547113)

[Como o DDD poderia resolver? 5](#_Toc66547114)

[02-03 - Visão global do DDD 5](#_Toc66547115)

[Modelagem estratégica 5](#_Toc66547116)

[Mapa de Contexto 6](#_Toc66547117)

[Definição de Arquitetura 6](#_Toc66547118)

[Definição da Arquitetura do Contexto 7](#_Toc66547119)

[Modelagem Tática 7](#_Toc66547120)

[03 - Linguagem Ubíqua 8](#_Toc66547121)

[03-01 - Apresentação 8](#_Toc66547122)

[Ubiquitous Language - Linguagem Ubíqua 8](#_Toc66547123)

[Tipos de Domínio: 8](#_Toc66547124)

[Motivação 8](#_Toc66547125)

[03-02 – Papeis dentro de um projeto 8](#_Toc66547126)

[Papeis 8](#_Toc66547127)

[Como extrair e definir a linguagem ubíqua? 8](#_Toc66547128)

[Ubíqua = Usada em todo lugar 8](#_Toc66547129)

[Usar o modelo como pilar da linguagem 8](#_Toc66547130)

[Como gerenciar? 9](#_Toc66547131)

[03-05 – Dicas essenciais 9](#_Toc66547132)

[Dicas: 9](#_Toc66547133)

[Onde é indispensável o uso de Linguagem Ubíqua 9](#_Toc66547134)

[04 - Modelagem Estratégica 9](#_Toc66547135)

[04-01 – Context Map 9](#_Toc66547136)

[04-02 – Bounded Context 9](#_Toc66547137)

[Modelagem estratégica 10](#_Toc66547138)

[Motivação de ter aplicação dividida em contextos 10](#_Toc66547139)

[04-03 - Definindo contextos delimitados 10](#_Toc66547140)

[Utilizado a UL como base 10](#_Toc66547141)

[Requisitos (Conversa com o): 10](#_Toc66547142)

[Identificação dos elementos: 10](#_Toc66547143)

[Identificação das possíveis ações: 10](#_Toc66547144)

[Definindo contextos e elementos chave 10](#_Toc66547145)

[Enquadrar os elementos dentro das categorias 11](#_Toc66547146)

[04-04 - Modelo de Negócio vs Modelo de Domínio 12](#_Toc66547147)

[Big Ball Off Mud 12](#_Toc66547148)

[Modelo de domínio 13](#_Toc66547149)

[04-05 - Integridade do Modelo 14](#_Toc66547150)

[Integridade do modelo em risco 15](#_Toc66547151)

[Integridade do modelo em risco 15](#_Toc66547152)

[Visão dos contextos delimitados 15](#_Toc66547153)

[04-06 - Tipos de relacionamento entre contextos 15](#_Toc66547154)

[Tipo de relacionamento entre contextos: 16](#_Toc66547155)

[Revisão: 16](#_Toc66547156)

[05 - Estilo e Padrões Arquiteturais 18](#_Toc66547157)

[05-01 - A evolução dos estilos arquiteturais 18](#_Toc66547158)

[05-02 - Definindo um estilo arquitetural 18](#_Toc66547159)

[05-03 - Transaction Script Pattern 19](#_Toc66547160)

[05-04 - Table Module Pattern 19](#_Toc66547161)

[05-05 - Domain Model Pattern 19](#_Toc66547162)

[05-07 - Arquitetura Hexagonal 19](#_Toc66547163)

[05-08 - Camadas sugeridas para atender o Domain Model 19](#_Toc66547164)

[Proposta de camadas do DDD 19](#_Toc66547165)

[05-09 - Camada de apresentação (Presentation Layer) 20](#_Toc66547166)

[Entregar aplicações modernas é crítico 20](#_Toc66547167)

[Aspectos da camada de apresentação 20](#_Toc66547168)

[05-10 - Camada de aplicação 21](#_Toc66547169)

[Responsabilidade 21](#_Toc66547170)

[Definição das camadas de negócio 21](#_Toc66547171)

[05-11 - Camada de Domínio 21](#_Toc66547172)

[Domain Model Pattern 21](#_Toc66547173)

[Serviços de Domínio 22](#_Toc66547174)

[05-12 - Camada de Infraestrutura 22](#_Toc66547175)

[06 - Modelagem Tática 22](#_Toc66547176)

[06-01 - Domain Module 22](#_Toc66547177)

[Abordagem clássica 22](#_Toc66547178)

[Clareando os equívocos 22](#_Toc66547179)

[Camada de Domínio 23](#_Toc66547180)

[E o que seria um Domain Module? 23](#_Toc66547181)

[06-02 - Objetos de Valor (Value Objetcs) 23](#_Toc66547182)

[06-03 - Entidades (Entities) 23](#_Toc66547183)

[06-04 - Agregações (Aggregate) 24](#_Toc66547184)

[Definição 24](#_Toc66547185)

[O que é comportamento? (Formas de utilizar) 24](#_Toc66547186)

[Fatores de uma agregação 24](#_Toc66547187)

[Responsabilidades 24](#_Toc66547188)

[06-05 - Serviços de Domínio 24](#_Toc66547189)

[Definição 24](#_Toc66547190)

[Exemplo de Serviço de Domínio / 1ª 25](#_Toc66547191)

[Exemplo de Serviço de Domínio / 2ª 25](#_Toc66547192)

[06-06 - Repositórios 25](#_Toc66547193)

[Definição 25](#_Toc66547194)

[Não existe uma forma errada de escrever um repositório. 25](#_Toc66547195)

[06-07 - Eventos de Domínio 25](#_Toc66547196)

[Exemplo: 26](#_Toc66547197)

[Disparar eventos de domínios para ações relevantes 26](#_Toc66547198)

[Exemplo melhorado 27](#_Toc66547199)

[07 - Mão na Massa - Modelagem Tática 27](#_Toc66547200)

[07-01 - Iniciando o projeto 27](#_Toc66547201)

[07-02 - Escrevendo a primeira entidade 28](#_Toc66547202)

[07-03 - Validações de entidades 28](#_Toc66547203)

[07-04 - Agregando com objetos de valor 28](#_Toc66547204)

[07-05 - Testando comportamentos 28](#_Toc66547205)

[07-06 - Persistindo com repositórios 28](#_Toc66547206)

[07-07 - Utilizando serviços de domínio 28](#_Toc66547207)

[07-09 - Opções de camada de application 29](#_Toc66547208)

[07-10 - Implementando a camada de application 29](#_Toc66547209)

[08 - CQRS 30](#_Toc66547210)

[08-01 – Apresentação 30](#_Toc66547211)

[CQRS – Command Query Responsabilty Segregation 30](#_Toc66547212)

[Revisão dos conceitos de CQRS: 30](#_Toc66547213)

[08-02 - Teorema CAP 30](#_Toc66547214)

[Em Resumo: 31](#_Toc66547215)

[Deve-se escolher duas opções 31](#_Toc66547216)

[08-03 - Como sincronizar as bases 31](#_Toc66547217)

[08-04 - Command Stack - Query Stack 32](#_Toc66547218)

[08-05 - Utilização de Sagas 32](#_Toc66547219)

[09 - Mão na Massa – CQRS 32](#_Toc66547220)

[09-01 - Setup do BC de Vendas 32](#_Toc66547221)

[09-02 - Objetos de Domínio 32](#_Toc66547222)

[09-03 - Command e Command Handler 33](#_Toc66547223)

[09-04 - Manipulando comandos na prática 33](#_Toc66547224)

[09-05 - Tratamento de notificações 34](#_Toc66547225)

[09-06 - Event e Event Handler 34](#_Toc66547226)

[09-07 - Implementando Queries (Query Stack) 34](#_Toc66547227)

[09-08 - Adicionando novas funcionalidades com comandos 34](#_Toc66547228)

[09-09 - Integração de BC's - Processamento do pedido 34](#_Toc66547229)

[09-10 - Integração de BC's – Pagamento 34](#_Toc66547230)

[09-11 - Integração de BC's - Finalização do pedido 35](#_Toc66547231)

[09-12 - Visão geral da implementação 35](#_Toc66547232)

[09-13 - Trabalhando com Sagas, Service Bus e Filas MSMQ 35](#_Toc66547233)

[10 - Event Sourcing 35](#_Toc66547234)

[10-01 - Sobre o que é 35](#_Toc66547235)

[Definições 35](#_Toc66547236)

[Utilização 35](#_Toc66547237)

[10-02 - Fatores essenciais 35](#_Toc66547238)

[Como funciona 35](#_Toc66547239)

[Complicado? 35](#_Toc66547240)

[10-03 - [Mão na massa] - Setup do ES no projeto 36](#_Toc66547241)

[10-04 - [Mão na massa] - Persistindo eventos 36](#_Toc66547242)

[10-05 - [Mão na massa] - Finalizando a implementação do ES 36](#_Toc66547243)

[11 - Encerramento 36](#_Toc66547244)

[11-01 - Palavras finais 36](#_Toc66547245)

[11-02 - Zerei a vida ou só comecei 36](#_Toc66547246)

[O que foi feito? 36](#_Toc66547247)

[O que fazer 36](#_Toc66547248)

[Anotações Extras 38](#_Toc66547249)

[Mediator 38](#_Toc66547250)

# 01 – Introdução

## 01-01 - Apresentação

## 01-02 - Objetivos do Curso

# 02 - Visão Geral do DDD

## 02-01 - DDD não é arquitetura de camadas

DDD não é uma receita pronta sobre como desenvolver uma arquitetura baseada em camadas do tipo:

* Pesentation
* Services
* Application
* Domain
* Infra

O DDD é uma abordagem de modelagem de software com foco na complexidade da aplicação

Através do conhecimento do domínio é possível facilitar a implementação de complexas regras / processos de negócios

Domain-Driven Design =e sobre Design. Design guiado pelo conhecimento do domínio

“Toda arquitetura é design, mas nem todo design é arquitetura” Grady Booh

## 02-02 - Quanto devo implementar o DDD?

Você precisa ter conhecimento suficiente para identificar por conta própria para identificar por conta própria se deve implementar DD ou não

Caso não esteja conseguindo identificar. Não implemente.

Link: <https://www.eduardopires.net.br/2016/08/ddd-nao-e-arquitetura-em-camadas/>

### Big Ball of Mud

Grande bola de lamê, indica um anti-padrão para arquitetos e desenvolvedores e refere-se a um software que possui todos os “problemas” que resultam em um código difícil de entender, manter e estender.

### Como o DDD poderia resolver?

* Capturar elementos conhecidos do negócio
* Organizá-los em um conjunto de princípios
* Criar uma modelagem de domínio com foco no desenvolvimento
* Abordagem diferenciada para desenvolver a regra de negócio

## 02-03 - Visão global do DDD

Livro Domain-Driven Design

Pontos chave:

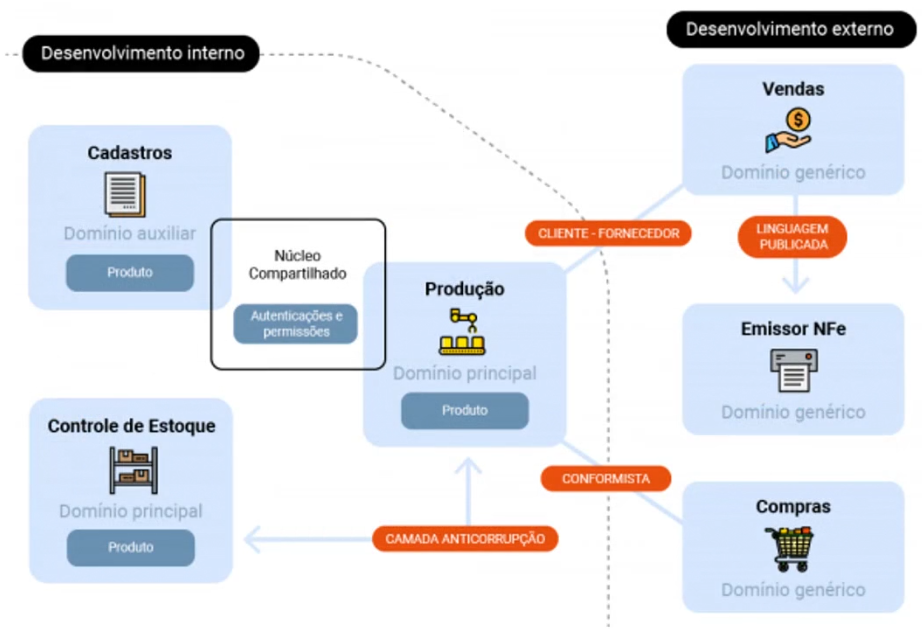
* Ubiquitous Languagem - Linguagem ubíqua
* Bounded Context - Contextos delimitados
* Context – Map Mapas de contextos

### Modelagem estratégica

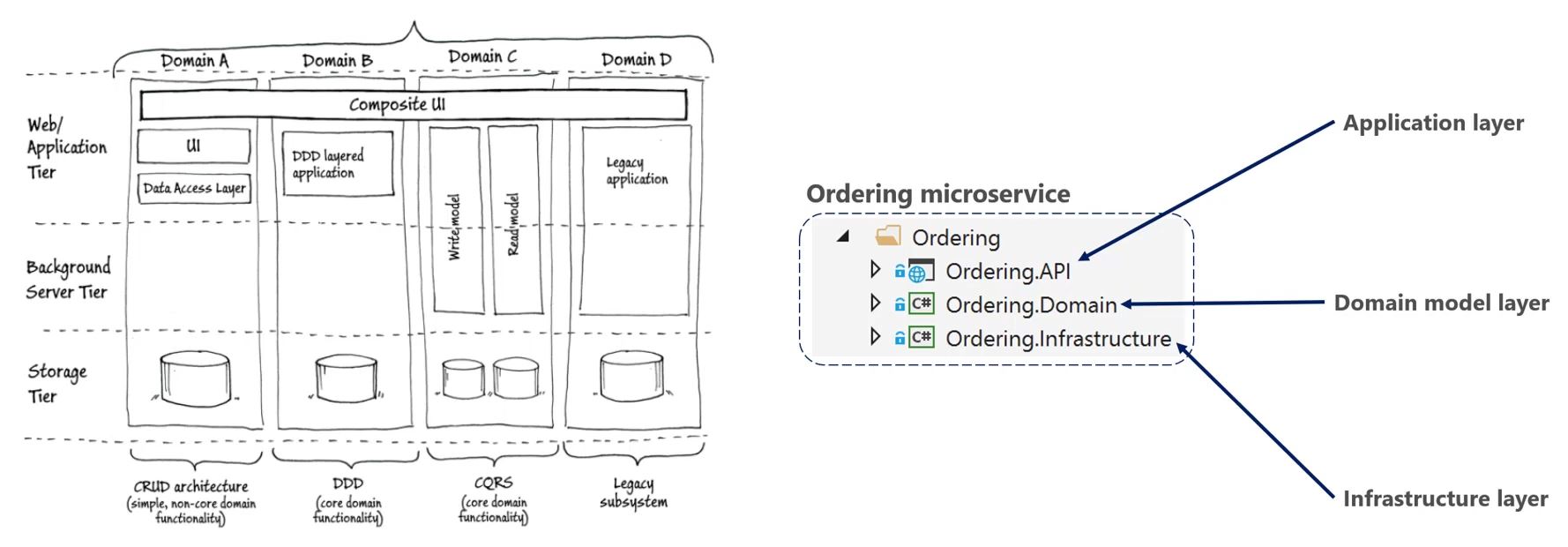
Exemplo:



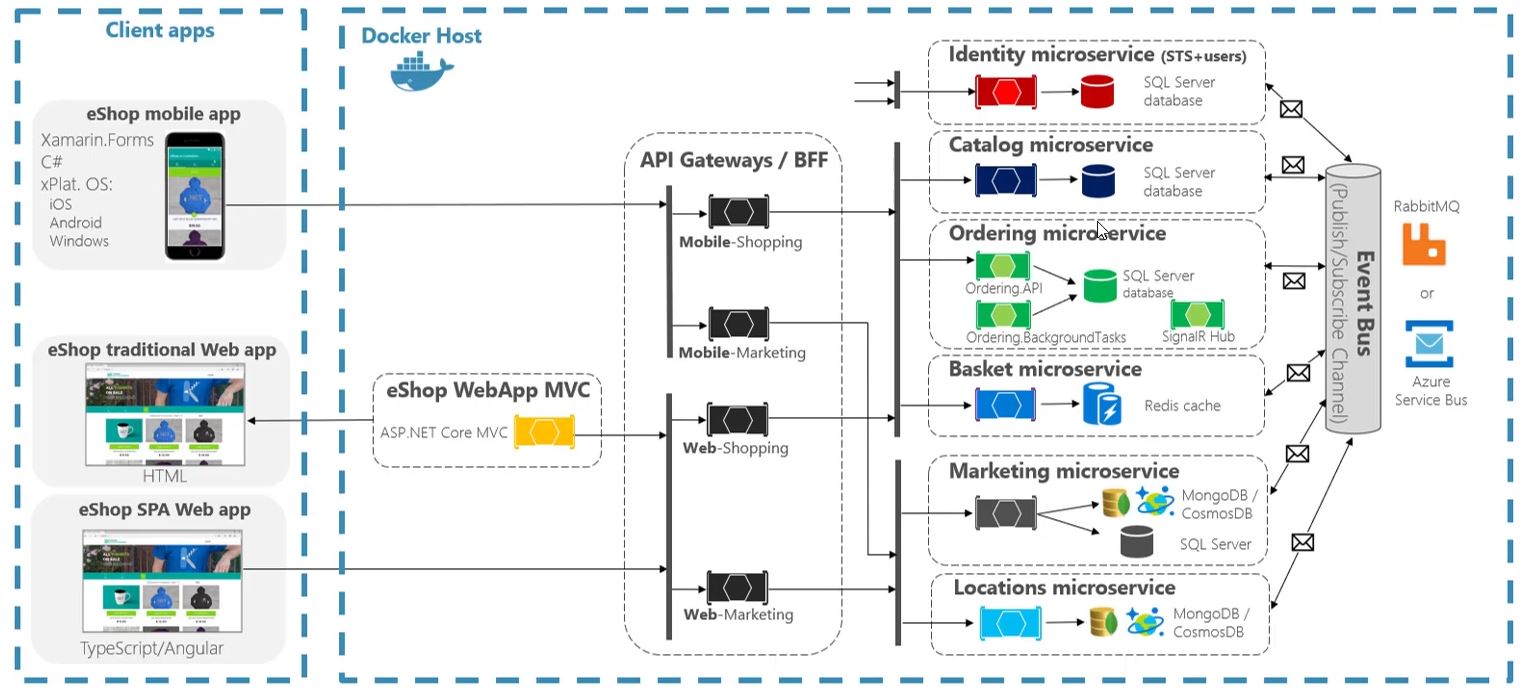
### Mapa de Contexto



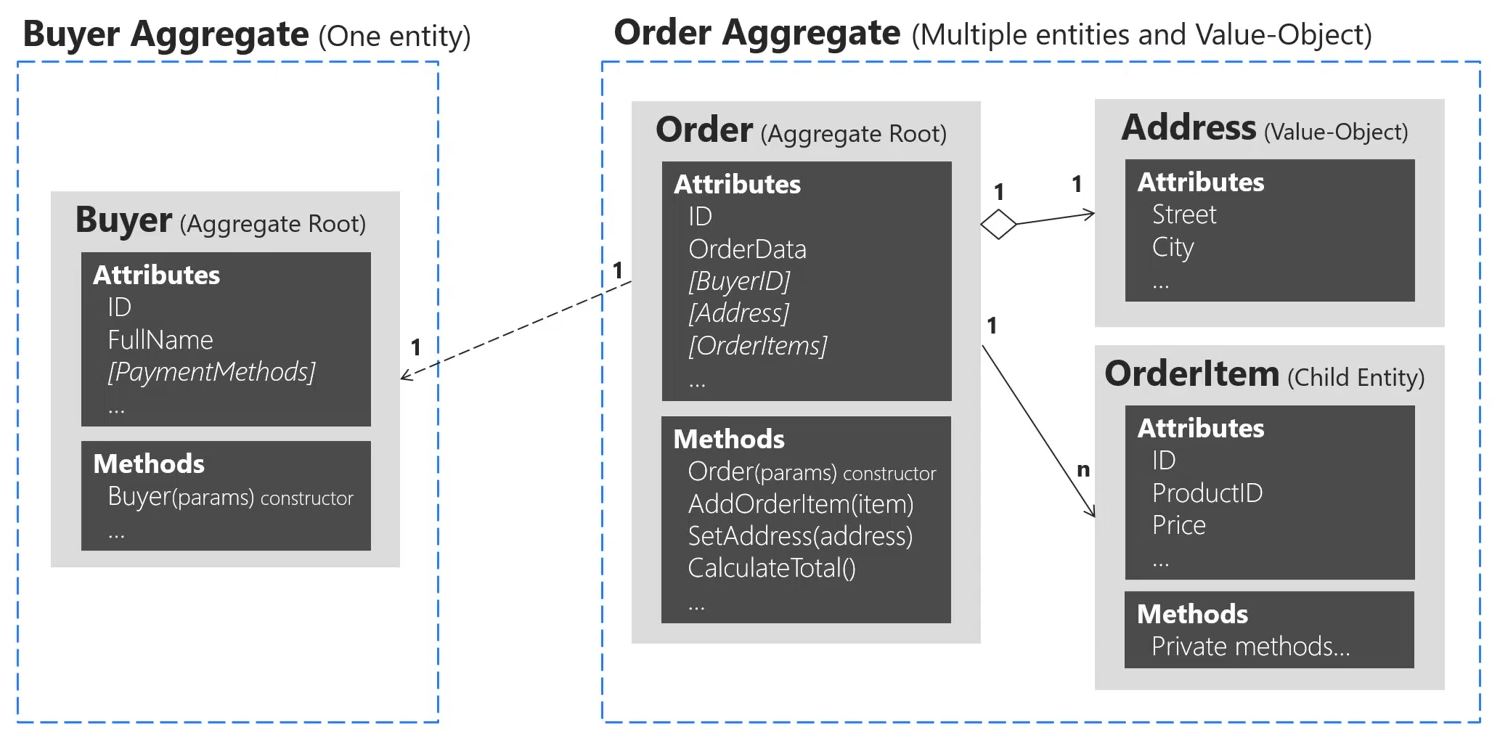
### Definição de Arquitetura



### Definição da Arquitetura do Contexto



### Modelagem Tática

****

# 03 - Linguagem Ubíqua

## 03-01 - Apresentação

### Ubiquitous Language - Linguagem Ubíqua

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

### Tipos de Domínio:

* Domínio Principal
* Domínio de Suporte
* Domínio Genérico

### Motivação

* Pessoas usam linguagens diferentes
* Terminologias em comum
* Ajudando os requisitos do usuário fazerem mais sentido

## 03-02 – Papeis dentro de um projeto

### Papeis

* **Domain Expert**
  + Conhece do negócio, os processos e seus termos
  + Define novos termos, processos e regras
  + Tem conhecimento da operação
* **Time de Desenvolvimento**
  + Precisa do apoio do Domain Expert para entender o processo, e as terminologias do negócio

### Como extrair e definir a linguagem ubíqua?

* Linguagem natural, não artificial
* Obtida em reuniões e brainstormings
* Composta e redefinida com o tempo
* Fluente e única
  + Atende ao entendimento do Domain Expert
  + Atenda ao entendimento da Equipe Técnica

### Ubíqua = Usada em todo lugar

* User Stories e Requisitos Funcionais
* Reuniões
* E-mail
* Documentação técnica
* Documentação do cliente
* O cliente conhece e usa
* Os departamentos relacionados usam
* O CÓDIGO FONTE usa

### Usar o modelo como pilar da linguagem

* Descobrir a linguagem ubíqua
* Compreender o domínio do negócio
* Desenvolver o modelo

Não pode haver sinônimos ou ambiguidades

### Como gerenciar?

* **Lista de termos utilizados salvos em documentos**
  + Glossário de termos completamente explicado
  + Disponível para todos (rede, núvem, intranet)
  + Parte da documentação do projeto
* **Atualizado constantemente**
  + Reponsabilidade do time de desenvolvimento

**Ferramenta para Gerênciar Glossário – Linguagem Ubíqua**

**Slimwiki** - <https://slimwiki.com/>

## 03-05 – Dicas essenciais

### Dicas:

* Contratar / Escalar Especialistas
* Lidar com Siglas
  + Muito específico
  + Difíceis de lembrar
  + Evitar se possível
* O idioma dos termos
  + Apenas Português?
  + Idiomas estrangeiros?
* Sincronizar o código fonte

### Onde é indispensável o uso de Linguagem Ubíqua

* Muitas definições difíceis de entender
  + Garantir que todos os termos estão muito claros
  + Nenhum outro termo é usado para indicar comportamento semelhante
* A lógica de negócio não está 100% definida
  + O negócio é novo e está crescendo
    - Exemplo: Startup
  + O domínio está sendo descoberto conforme o tempo

# 04 - Modelagem Estratégica

Postagem: <https://www.eduardopires.net.br/2016/03/ddd-bounded-context/>

## 04-01 – Context Map

....

## 04-02 – Bounded Context

* Contexto delimitado onde um elemento tem um significado bem definido
  + Todos os elementos da linguagem ubíqua
* Além dos limites do contexto a linguagem muda
  + Cada contexto delimitado tem sua própria linguagem ubíqua
* Domínio divido em uma teia de contextos interconectados
  + Cada contexto delimitado tem sua arquitetura e implementação

### Modelagem estratégica

Imagem..

### Motivação de ter aplicação dividida em contextos

* Remover ambiguidade e duplicação
* Simplificar o design dos módulos
* Integração dos componentes externos

## 04-03 - Definindo contextos delimitados

* **Bonded Context (Contexto delimitado)**
  + Linguagem Ubíqua
  + Arquitetura Independente
    - Ex: 4 camadas + CQRS
  + Interfaces de Serviços (para outros serviços)

### Utilizado a UL como base

### Requisitos (Conversa com o):

* A loja virtual exibirá um catálogo de produtos de diversas categorias
* Um cliente pode realizar um pedido contendo de 1 a N produtos.
* A loja realizará as vendas através de pagamento por cartão de crédito.
* O cliente irá realizar o seu cadastro para poder fazer pedidos.
* O cliente irá confirmar o pedido, endereço de entrega, escolher o tipo de frete e realizar o pagamento.
* Após o pagamento o pedido mudará de status conforme reposta da transação via cartão
* O correta a emissão da nota fiscal logo após a confirmação do pagamento do pedido

### Identificação dos elementos:

* A loja virtual exibirá um **catálogo** de **produtos** de diversas **categorias**
* Um **cliente** pode realizar um **pedido** contendo de 1 a N produtos.
* A loja realizará as **vendas** através de **pagamento** por **cartão de crédito**.
* O cliente irá realizar o seu **cadastro** para poder fazer pedidos.
* O cliente irá confirmar o pedido, **endereço** de entrega, escolher o tipo de **frete** e realizar o pagamento.
* Após o pagamento o pedido mudará de **status** conforme reposta da **transação** via cartão
* O correta a emissão da **nota fiscal** logo após a confirmação do pagamento do pedido

### Identificação das possíveis ações:

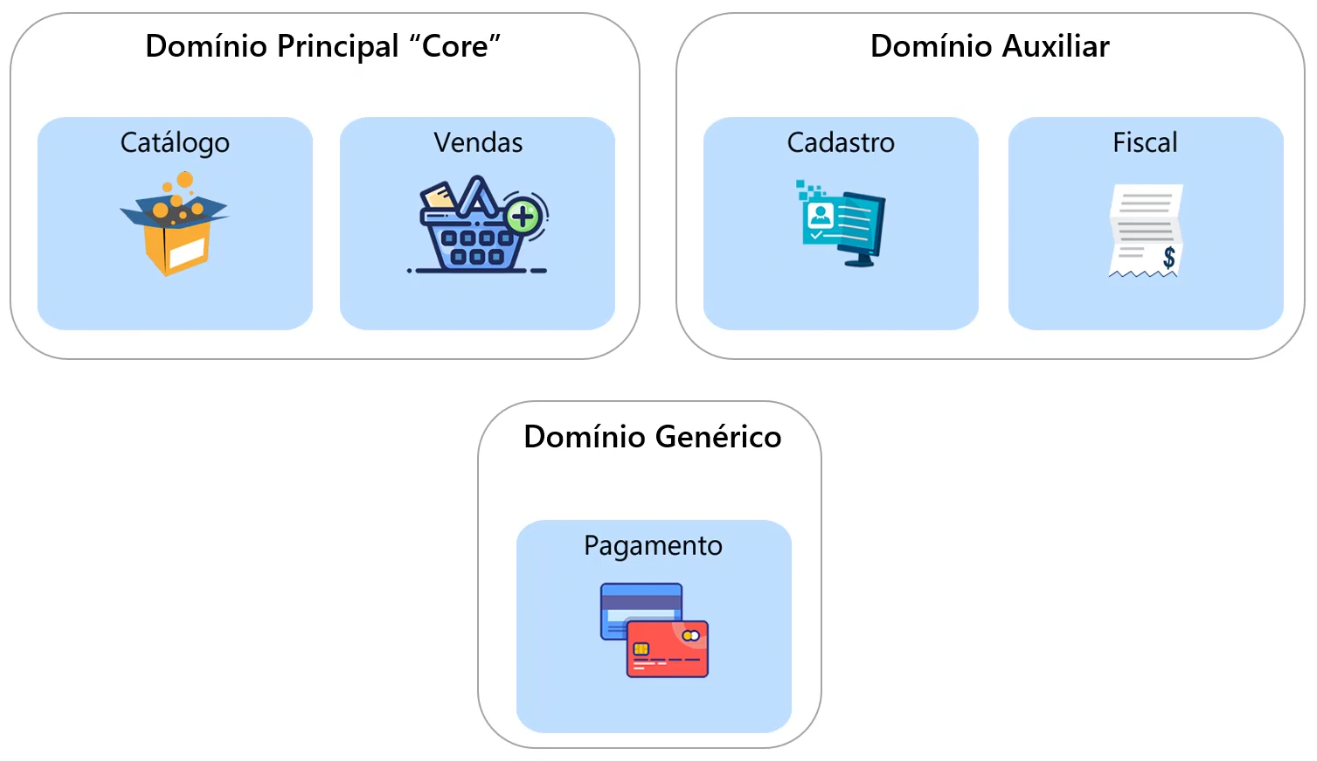
* A loja virtual **exibirá um catálogo de produtos de diversas categorias**
* Um cliente pode **realizar um pedido** contendo de 1 a N produtos.
* A loja **realizará as vendas** através de **pagamento por cartão de crédito**.
* O cliente irá **realizar o seu cadastro** para poder fazer pedidos.
* O cliente irá **confirmar o pedido**, endereço de entrega, **escolher o tipo de** **frete** e **realizar o pagamento**.
* Após o pagamento **o pedido mudará de status** conforme **reposta da transação via cartão**
* O correta a **emissão da nota fiscal** logo após a **confirmação do pagamento** do pedido

### Definindo contextos e elementos chave

Contextos:

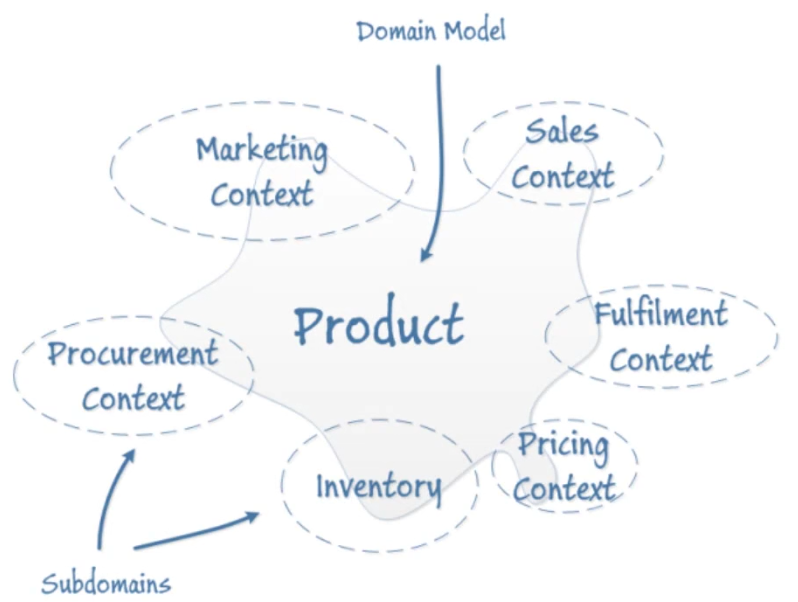


### Enquadrar os elementos dentro das categorias



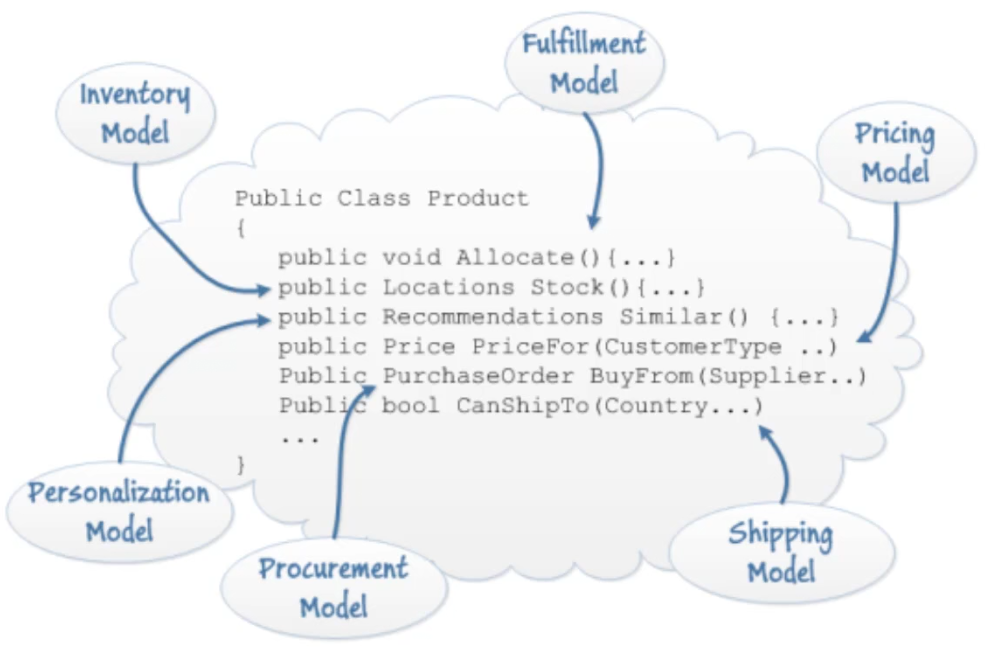
## 04-04 - Modelo de Negócio vs Modelo de Domínio

Modelo de Negócio



Tudo fala sobre produto

### Big Ball Off Mud



### Modelo de domínio



Estratégia 1 - Separar produto em diversas classes, porém com comportamentos diferentes

Todos os contextos persistindo em uma mesma tabela/banco

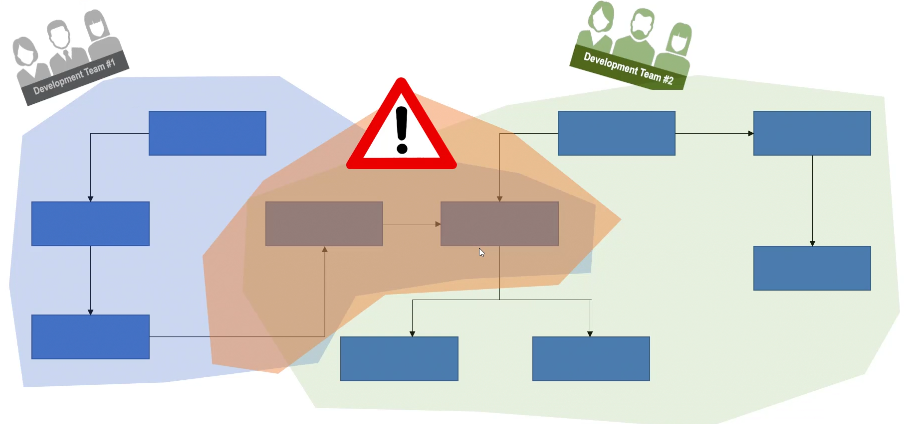
|  |  |
| --- | --- |
| **PRODUTO** | |
| **TABELA** | **CONTEXTO** |
| ID |  |
| Nome | Marketing Context |
| Categoria |
| Valor | Pricing Context |
| Valor Taxi |
|  |  |

Estratégia 2 - Cada contexto vai ter seu próprio banco de dados

* Onde cada banco terá a tabela Produto, porém, apenas com as colunas equivalentes ao seu contexto. Em todos os bancos o ID deve ser único.
* Usar CQRS
* Ter um banco de dados para cada contexto
* Um contexto não pode acessar o banco de outro contexto
* Ter um banco de dados somente para leitura de dados. (Banco desmoralizado)

## 04-05 - Integridade do Modelo

### Integridade do modelo em risco



A mesma entidade está sendo compartilhada entre dois contextos

### Integridade do modelo em risco

* O mesmo termo tem significados diferentes para times diferentes
* O mesmo termo representa elementos diferentes
* Dependência com sistemas externos
* Dependência com código legado
* Áreas funcionais da aplicação são melhores tratadas separadamente

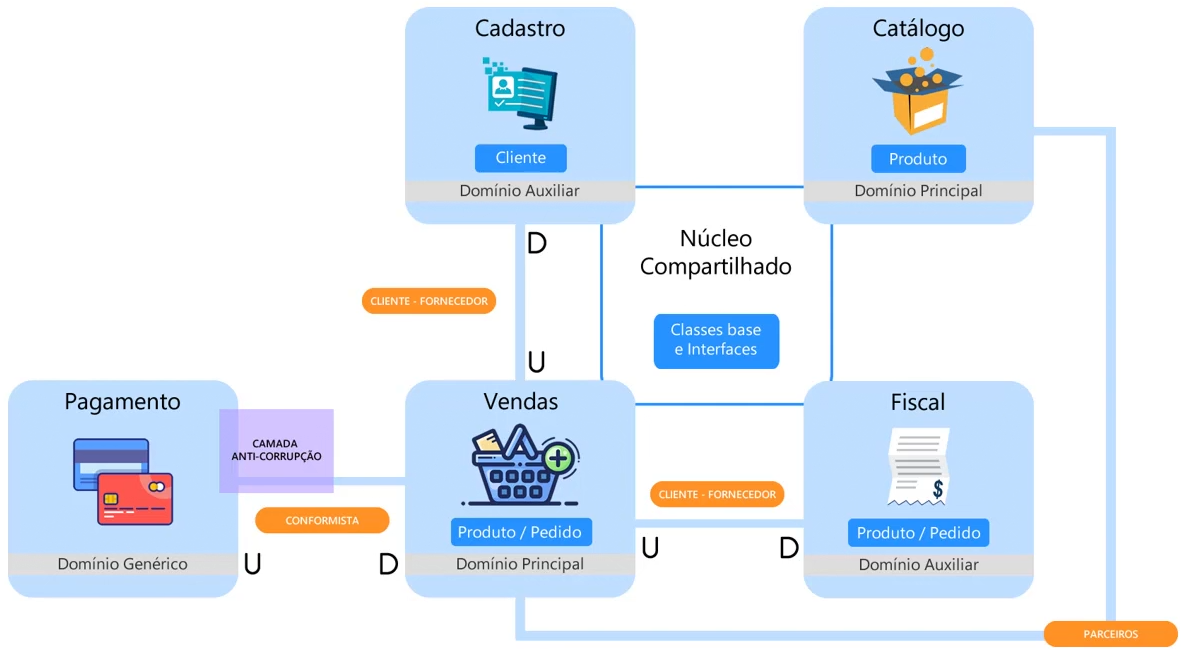
### Visão dos contextos delimitados

* A aplicação é resultado de uma composição de múltiplos contextos
  + Exemplo: Loja Virtual
  + Vendas
  + Fiscais
  + Logística
* O número de contexto delimitados refletem a organização física da empresa
  + Um Contexto delimitado para cada departamento da empresa

## 04-06 - Tipos de relacionamento entre contextos

Direção do Relacionamento

* Contextos **Upstream** influenciam contextos **Downstream**
  + Essas influências afetam: binários, mudanças, cronogramas



### Tipo de relacionamento entre contextos:

* **Cliente-Fornecedor (Customer-Supplier Development)**
  + Onde um contexto influencia no ouro
* **Parceiro (Partner)**
  + Colaboram, não existe uma direção. Os dois são iguais
  + Se um muda, o outro muda também.
* **Conformista (Conformist)**
  + É quando duas equipes possuem uma dependência mútua
  + Precisam, portanto, trabalhar juntos
* **Camada de anticorrupção (Anticorruption Layer)**
  + Nesse relacionamento a equipe **downstream** decide criar uma camada para proteger seu contexto das modificações **upstream**. É um típico cenário de sistemas legados
  + Garantir que tenha um único ponto de quebra
  + Fachada
* **Núcleo Compartilhado (Shared Kernel)**
  + Quando vários bounded context compartilham um mesmo domínio. Alterar significa que todas as equipes serão afetadas.
  + Classes base e Interfaces
  + PERIGO: Tudo que tiver no núcleo compartilhada, terá um forte acoplamento em todas os contextos

### Revisão:

* **Conformist**
  + Contextos Downstream dependem de Contextos Upstream
  + Nenhuma negociação é permitida
  + Criação camada de Anticorrupção
* **Customer / Supplier**
  + Contextos Customer dependem de Contextos Supplier
  + Oportunidade de levantar preocupações e abordar alguma solução
* **Partner**
  + Depenência mútua entre dois contextos
* **Shared Kernel**
  + Um modelo compartilhado não pode ser alterado sem consultar os times que dependem dele
* **Anti-Corruption layer**
  + Camada adicional dando ao contexto Downsteam uma inferface fixa independente do que acontece com contexto Upstream

# 05 - Estilo e Padrões Arquiteturais

## 05-01 - A evolução dos estilos arquiteturais

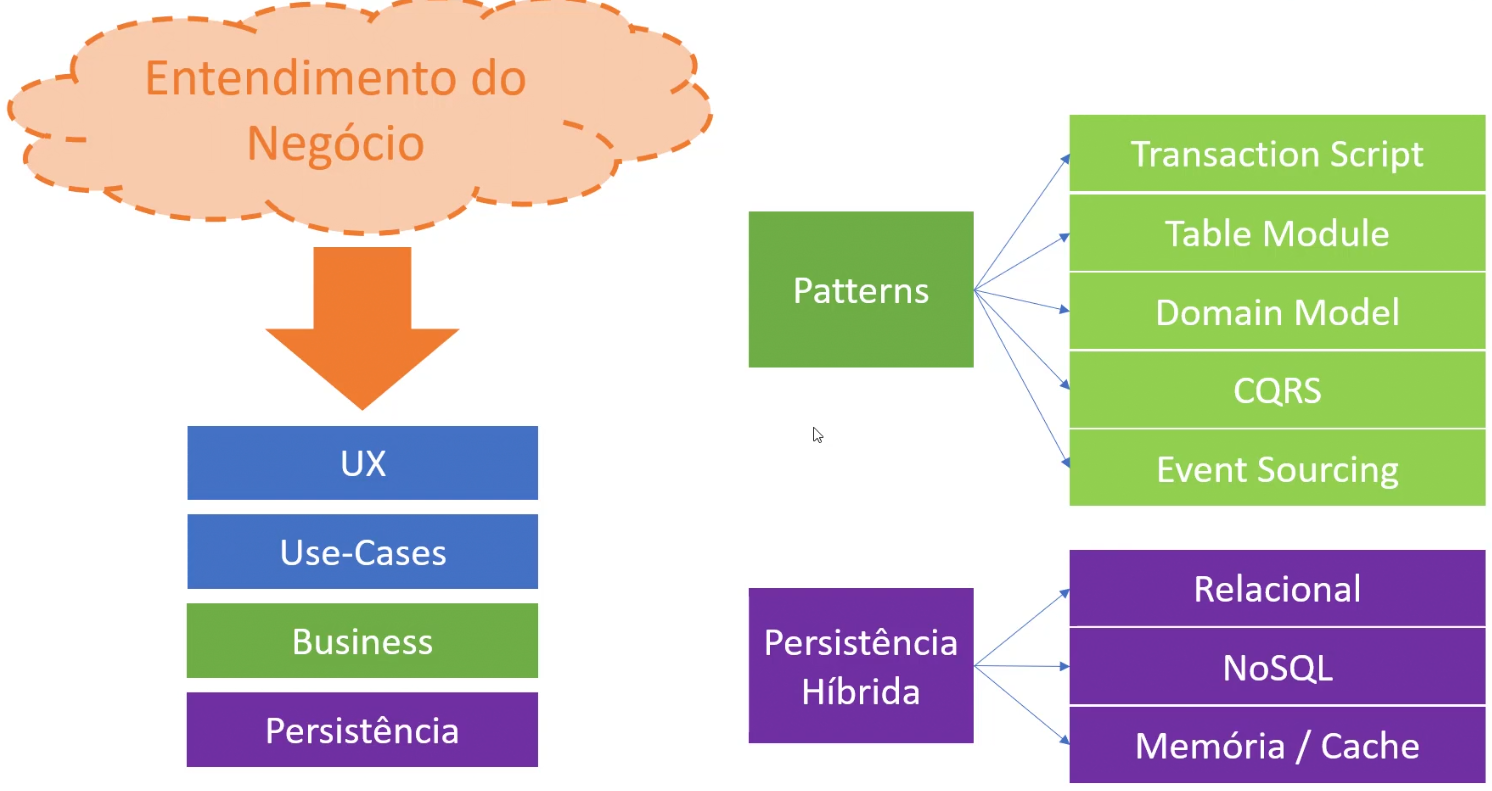
Modelo clássico em 3 camadas

* Presentation
* Bussiness
* Data

## 05-02 - Definindo um estilo arquitetural

Roteiro/Gabarito de como pensar na arquitetura

* **Entendimento do negócio**
  + UX
    - É api ou precisa de telas?
  + Use-cases / Use Story
    - A quantidade de casos de uso, ou história de usuários vão definir a quantidade de telas e processos a implementar
    - Ajudar a entender os desafios de negócios que será necessário implementar para resolver as situações para atender os casos de uso ou história de usuários
  + Bussinesss
* **Patterns**
  + Transaction Script
  + Table Module
  + Domain Module
  + CQRS
  + Event Sourcing
* **Modelo de Persistência / Persistência Hibrida**
  + Relacional
  + NoSQL
  + Memória / Cache



## 05-03 - Transaction Script Pattern

* Ações da aplicação
  + Cada processo engloba uma única ação
* Transação Lógica
  + Apresentação e dados
* Ações comuns
  + Dividida em processos delimitados para reuso

## 05-04 - Table Module Pattern

* Um módulo por tabela no banco de dados
* Módulos possuem todos os métodos que processam os dados
  + Requer queries e comandos
* Limitações de módulos para tabelas “significantes”
  + Tabelas com chaves estrangeiras para relacionamento

## 05-05 - Domain Model Pattern

* **Aggregate Objects**
  + Dados e comportamentos
* **Persistência Agnóstica**
  + Ignorante em relação a como persiste os dados
* **Alinhada com os serviços de domínio**
  + Realizam tarefas que não são possíveis serem realizadas nas entidades
  + Tarefas = Fluxos e regras

Todo agregado possue um serviço

**05-06 - Arquitetura Cebola**

Onion Architecture – Clean Architecture

Defende a ideia que precisa-se isolar as camadas e cada camada possuir sua responsabilidade, no qual existe uma direção de relacionamento.

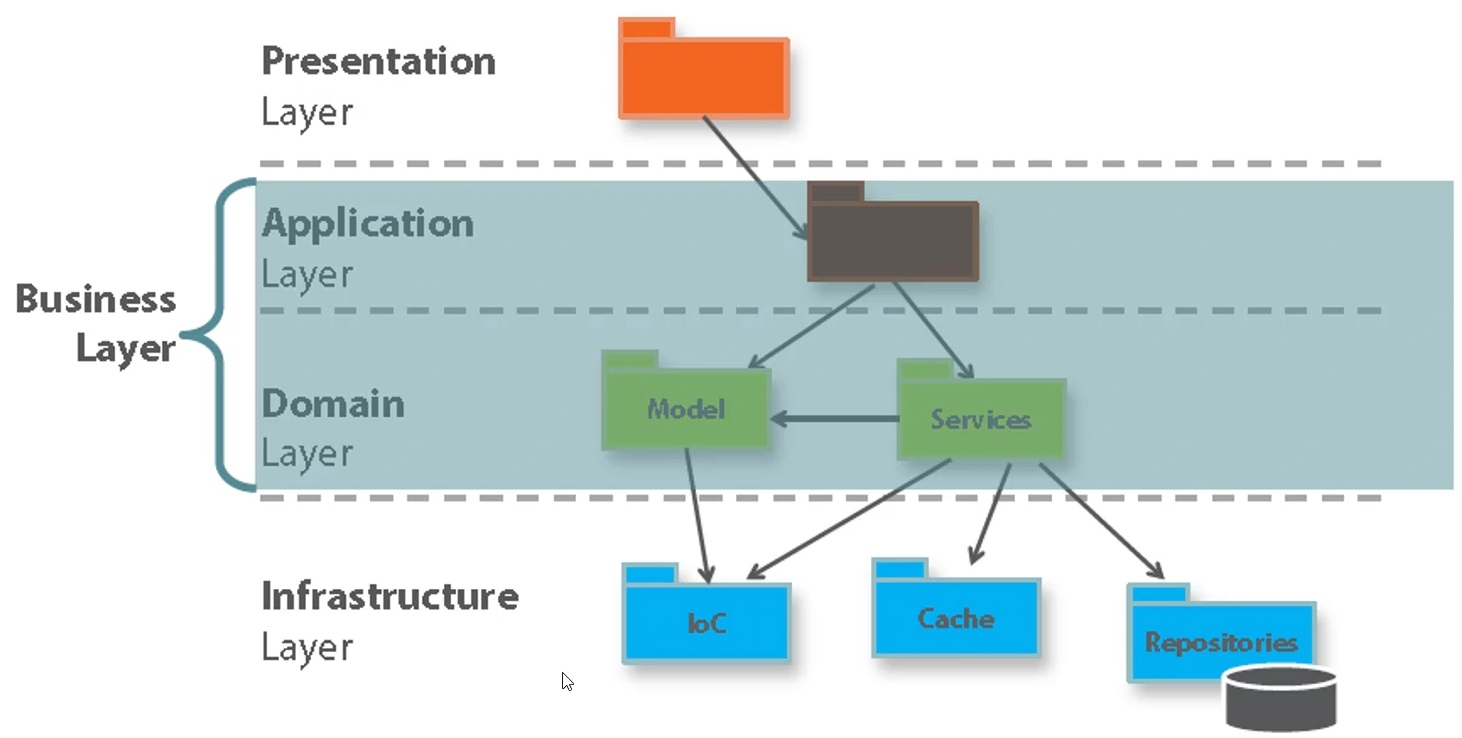
## 05-07 - Arquitetura Hexagonal

Hexagonal Architecture “Ports & Adapters”

Onion, DDD, Clean CQRSt udo junto...

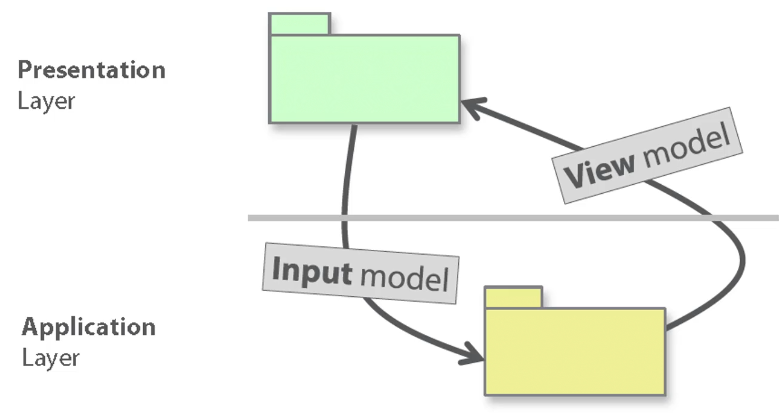
## 05-08 - Camadas sugeridas para atender o Domain Model

### Proposta de camadas do DDD



## 05-09 - Camada de apresentação (Presentation Layer)

* Faz o front com o usuário
* Pode ser MVC, WebApi, REST, Mobile, SPA
* Camada de Apresentação está duplamente ligada com a camada de aplicação



### Entregar aplicações modernas é crítico

* É necessário prover uma interface rica para executar todas as operações
  + É necessário trabalhar com diversos componentes e frameworks
* Responsabilidade de prover uma experiência de uso rica
  + Chamadas assíncronas, retorno silencioso, facilidade de encontrar a informação

### Aspectos da camada de apresentação

* Baseada em ações
* Responsiva (Device-friendly)
* User-friendly
* Compatível com o processo do mundo real

## 05-10 - Camada de aplicação

* Não precisa, necessariamente, ser um projeto na arquitetura.
* Pode ser um controller, simplesmente

### Responsabilidade

* Em qual camada deve ficar o código que formata os dados para a apresentação?  
  Possíveis respostas:
  + - Se os dados vão ser exibidos em apresentação é lá que deve ser tratado”
  + - Isso parece ser responsabilidade de negócio, deveria ser uma camada de negócios.
  + - Dados são dados, é a base de dados que retorna as informações tratadas
* Envia informação para apresentação
  + Fornece dados prontos para consumo conforme necessidade de exibição
* Orquestra cações disparadas pelos elementos de apresentação (Espécie de Workflow)
  + Casos de uso da camada de apresentação
* Duplamente ligada com a camada de apresentação
  + Pode ser estendida ou duplicada quando um novo frontend é adicionado

### Definição das camadas de negócio

**Representação 1**

* **Aplicação**
  + Depende dos casos de uso
    - Entidades de aplicação
    - Processos de aplicação
* **Domínio**
  + Independe de casos de uso
    - Entidades de negócio
    - Processos de negócio

**Representação 2**

* **Aplicação**
  + Depende dos casos de uso
    - Data Transfer Objects
    - Application Services
* **Domínio**
  + Independe de casos de uso
    - Domain Model
    - Domain Services

## 05-11 - Camada de Domínio

### Domain Model Pattern

* **Modelos do domínio**
  + Entidade baseada em OOP
  + Modelos Funcionais
* **Guia para classes entidades**
  + Convenções do DDD
    - Factories
    - Value types
    - Private setters
  + Dados e comportamento
* **Modelo Anêmico**
  + Apenas propriedades (dados do modelo)
  + Comportamentos e regras implementadas no serviço

### Serviços de Domínio

* Partes do domínio que não se encaixam em entidades existentes
* Classes que agrupam comportamentos diversos
  + Tipicamente trabalhando com diversas entidades
* Implementação de processos que:
  + Requerem acesso à persistência para ler e gravar
  + Requerem acesso a serviços externos

Caso seja implementado CRQS, esta camada terá também Comandos, ComandoHandlers, Events...

## 05-12 - Camada de Infraestrutura

Em resumo, é uma super camada que agrega o ferramental

Diga: Isole os detalhes da camada de infra

Fazer uso de facades para isolar detalhes tecnológicos

# 06 - Modelagem Tática

Arte de escrever as classes de domínio

## 06-01 - Domain Module

### Abordagem clássica

* **Camada de Domínio**
  + **Domain Models**
    - Aggregates
    - Entities
    - Value Types
    - Factories
      * Classes que criam outras classes
  + **Domain Services**
    - Cross-aggregate behavior
      * Comportamento envolvendo mais de 2 entidades
    - Repositories
      * Persistencia de dados
    - External services
      * Serviços externos

### Clareando os equívocos

* **Modelos de Domínio**
  + Mapear o contexto é fundamental
  + Modelar os objetos de domínio conforme identificados na UL
* **Database Agnostic**
  + O modelo deve ser fácil de persistir.
  + Persistir não deve ser a sua única preocupação!
  + A principal preocupação deve ser manter o sentido com o processo de negócio do domínio
* **Ubiquitous Language**
  + Entender a linguagem para entender o negócio
  + Manter a linguagem do negócio alinhada com o código

### Camada de Domínio



### E o que seria um Domain Module?

* Agregates (terá sempre)
* Entities
* Value Objects

Modulo representa uma parte do negócio

## 06-02 - Objetos de Valor (Value Objetcs)

* Coleção de dados individuais
* Destinado para uma coleção de atributos especializado numa classe mais rica
* Imutável
* Mais preciso que os tipos primitivos

Vantagens

* public string CPF {get; set;}
* public CPF CPF {get; set;}
  + Contrutores
  + Constantes Max e Min
  + Add hoc setters com lógica validação
  + Add hoc getters
  + Propriedades aidionais

## 06-03 - Entidades (Entities)

* Possuem Identidade
  + Ex: ID
* Deve ser exclusiva para o objeto mapeado
  + Não é possível existir um mesmo objeto mapeado para mais de uma Entidade
* Possui Estados e Comportamentos, logo não pode ser Anêmica
* Possui Lógica de Negócio
* Não faz persistência.
  + Se uma parte da regra do negócio depender de uma consulta ao banco, então essa regra não poderá ser aplicada nessa Entidade) ou seja, nunca deve usar o Pattern Active Record! kkkk

## 06-04 - Agregações (Aggregate)

### Definição

* Conjunto de Entidade que são referenciadas e usadas juntas.
* Um conjunto de entidades tratadas como uma quando o dado é alterado
* Possui uma única raiz de agregação (Aggregation Root)
  + que é a Entidade responsável pela consistência e representa todos os outros objetos que a compões
* A agregação (Agregate Root) mantém a integridade das classes filhas

Em resumo: é uma composição utilizada na UML.:D

### O que é comportamento? (Formas de utilizar)

* Métodos que validam o estado do objeto
* Métodos que invocam regras de negócio e alteram o estado do objeto
* Métodos que expressam processos do negócio envolvendo o objeto

### Fatores de uma agregação

* Limitar o acesso aos objetos filhos
* Certificar que o estado dos objetos filhos está sempre consistente
* Os limites reais de uma agregação são determinada pela regra de negócio
  + Agregações Distintas
    - Agregação no UML
  + Agregação
    - Composição no UML

### Responsabilidades

* Certificar sempre a consistência do estado dos objetos filhos
* Cuidar da integridade de todos os objetos filhos
* Intermedia a adição / edição / exclusão dos objetos filhos.
* Acesso aos objetos encapsulados devem ser sempre por Ações
* Utilizar um único repositório por agregação (Regra)

## 06-05 - Serviços de Domínio

### Definição

* Implementam lógica de negócios que não pertencem a um agregado em particular e trabalham com múltiplas entidades
* Coordenam a atividade dos agregados e repositórios com o propósito de implementar a ação de negócio
* Podem consumir serviços da infraestrutura, como enviar um e-mail, eventos ou mensagens.

Ações realizadas pelos serviços de domínio estão previstas nos **requisitos** e são permitidas e aprovadas pelo **domain experts**.

* Nomes utilizados nos serviços de domínio são estritamente parte da linguagem ubíqua

### Exemplo de Serviço de Domínio / 1ª

* Cenário:
  + Determinar se um cliente atingiu o status “platinum”
* Regra:
  + O cliente para ser Platinum precisa ter atingido 5000 em compras.
* Ações
  + Necessidade de consultar Pedidos e Produtos para determinar a somatória
  + Não é responsabilidade da Agregação acessar dados
  + Valores são consultados pelo Serviço e populado novas instâncias
  + Regra de negócio Cross-Aggregate

### Exemplo de Serviço de Domínio / 2ª

* Cenário
  + Alugando uma sala de reunião
* Regra
  + Alugar uma sala requer **validar** a disponibilidade da sala e **processar** o pagamento
* Ações
  + Reserva (DomainService)
    - Valida a disponibilidade da sala
    - Realiza a transação com o mecanismo
  + Reserva (Agregação)
    - Sala e objetos filhos
    - O repositório da agregação realiza a ação

## 06-06 - Repositórios

### Definição

* Apenas uma classe que persiste as entidades e agregações
* ..
* O tipo de classe mais popular do serviço de domínio
* Cuida de persistir os agregados
* Apenas um repositório por agregação
* Os repositórios possuem dependência direta com o meio de acesso a dados
* Um repositório é onde se pode tratar com comandos SQL e connection strings

### Não existe uma forma errada de escrever um repositório.

Ele pode:

* Ser Genérico
* Ser Especializado
* Conectar diretamente no banco e trabalhar com ADO
* Utilizar ORM
* Consultar Serviços Externos
  + Dica - Isolar a tecnologia atraves de Facade
* DEVE apenas retornar dados, fazer isto bem feito e somente fazê-lo

Cada aggregate possui um Domain Service para resolver as ações que envolvem persistência

## 06-07 - Eventos de Domínio

Não é muito utilizado, mas é muito interessante

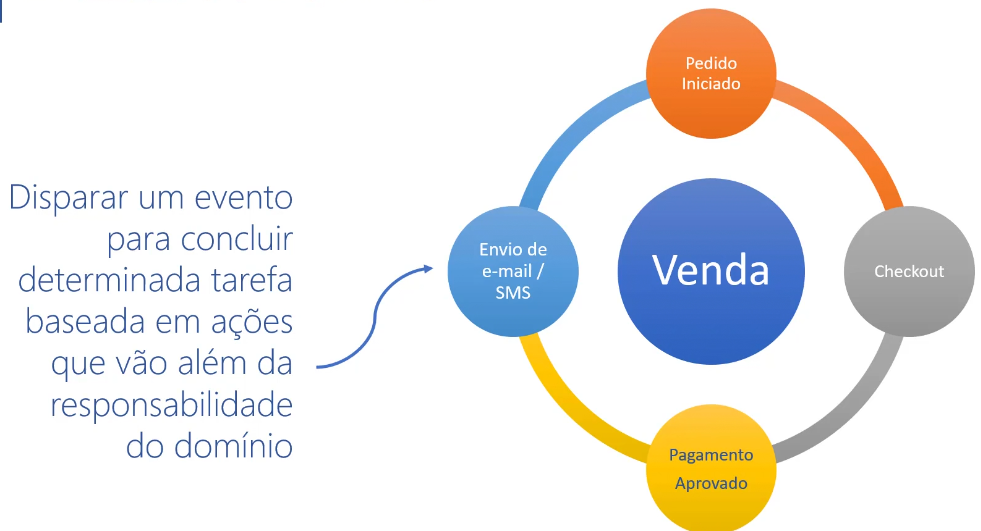
Você já considerou utilizar **eventos** em sua camada de domínio?

* Apesar de não serem obrigatórios, eles podem oferecer uma resiliência efetiva para expressar comportamentos do mundo real.

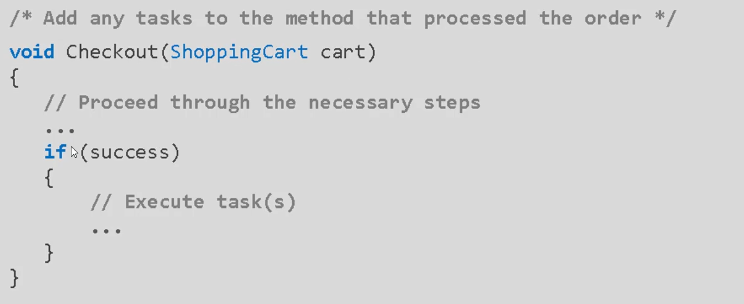
Simulação

Dica – Domínio não dispara e-mail

### Exemplo:



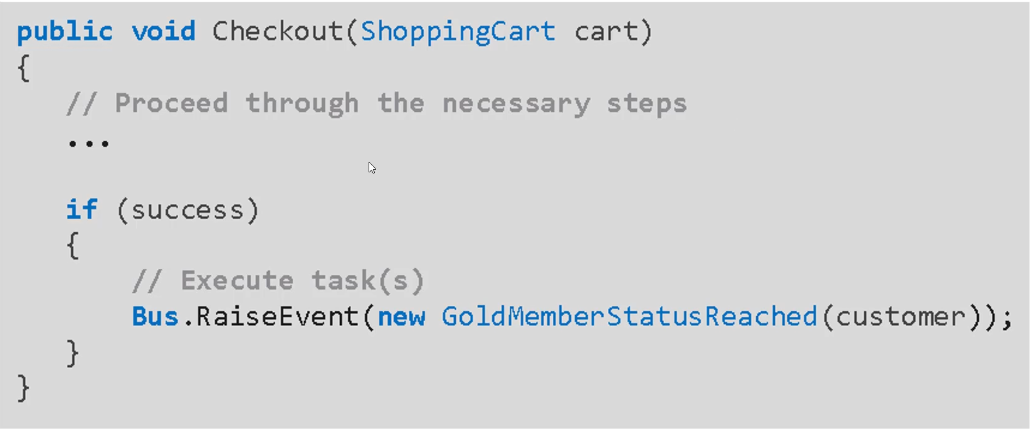
**Código monolítico não muito expressivo, necessidade de modificar o serviço no caso de mudanças:**



### Disparar eventos de domínios para ações relevantes

* Não é necessário ter todo o código em um único lugar
* Disparar o evento sem necessidade de conhecer o que ele faz
* Manipular o mesmo evento em diversos lugares

### Exemplo melhorado



# 07 - Mão na Massa - Modelagem Tática

## 07-01 - Iniciando o projeto

Modelagem tática na Prática

Montar o contexto de Catálogo

Criação da **Solution**

Criação de das Diretórios lógicos da Solution

* **Services** - Ficará toda modelagem dos contextos
* **WebApps** - Ficará as aplicações web

Criação de diretórios dento de Services

* **Cadastros** – Contexto
* **Catalogo** – Contexto
* **Core** – Núcleo Compartilhado (Shared Kernel)
* **Fiscal** – Contexto Auxiliar
* **Pagamentos** – Contexto
* **Vendas** – Contexto

Criação dos Projetos no diretório **Catalogo**:

Criação do Projeto **NerdStore.Catalogo.Domain**, do Tipo **Class Library** **Core** em **Catalogo**

Padrão do Nome dos Projetos

* **NerdStore.Catalogo.Domain**
  + Nome da Empresa ou Projeto
  + Onde ele Está (contexto)
  + Tipo de Porjeto (camada)

Criação do Projeto **NerdStore.Core** do tipo **Class Library Core** em **Core**

## 07-02 - Escrevendo a primeira entidade

Código ...

## 07-03 - Validações de entidades

AssertionConcerns - Validações

## 07-04 - Agregando com objetos de valor

...

## 07-05 - Testando comportamentos

...

## 07-06 - Persistindo com repositórios

Criar a interface `IProdutoRepository` em `NerdStore.Catalogo.Domain`

Ciar o diretório `Data` em `NerdStore.Core`

Criar a interface `IRepository` em `NerdStore.Core`

Criar a interface `IUnityOfWork` em `NerdStore.Core`

Criar o projeto `NerdStore.Catalogo.Data` do tipo ‘ClassLib’ em Catalogo

Ciar/Implementar o Contexto DB `CatalogoContext` em `NerdStore.Catalogo.Data`

Instalar os pacotes:

* Microsoft.EntityFrameworkCore
* Microsoft.EntityFrameworkCore.Design
* Microsoft.EntityFrameworkCore.Relational
* Npgsql.EntityFrameworkCore.PostgreSQL (não usarei o SQL server)

Criar o diretório `Mappings` em `NerdStore.Catalogo.Data`

Dentro do diretório `Mappings`, criar os mapeamentos para `ProdutoMapping` e` CategoriaMapping`

Criar o diretório `Repositories` em `NerdStore.Catalogo.Data`

Ciar/Implementar o repositório para `ProdutoRepository`

## 07-07 - Utilizando serviços de domínio

Serviços de Domínio só existe porque implementam regras de negócios que vão além da agregação

Em NerdStore.Catalogo.Domain:

Criar a interface IEstoqueService

Criar/implementar a classe EstoqueService

**07-08 - Lançando eventos de domínio**

Instalar o pacote a seguir, nos projetos NerdStore.Catalogo.Domain e NerdStore.Core:

* MediatR

Em NerdStore.Catalogo.Domain, criar o Diretório Events

No diretório Events, criar a classe ProdutoAbaixoEstoqueEvent

Em NerdStore.Core criar o diretório Messages

Dentro de Messages:

Criar a classe Message, é uma classe base que

Criar a classe Event, é a classe que possui o horário do evento

## 07-09 - Opções de camada de application

Código ...

## 07-10 - Implementando a camada de application

Código ...

Criar o projeto NerdStore.Catalogo.Application

Instalar o Pacote

* AutoMaper

Adicionar referencias a NerdStore.Catalogo.Data e NerdStore.Catalogo.Domain

Criar os Diretórios: AutoMapper, ViewModel e Services

Dentro do Diretório ViewModels

Criar classe ProdutoViewModel

Criar classe CategoriaViewMOdel

Dentro do diretório Servies:

Criar a Interface IProdutoAppServive

Criar/implementar a Classe ProdutoAppService

Dentro do Diretório Automapper:

Criar a classe DomainToViewModelMappingProfile

Criar a classe ViewModelToDomainMappingProfile

**07-11 - Conectando a camada de apresentação**

No diretório lógico `WebApps`, criar o projeto `NerdStore.WebApp.MVC`, do tipo `ASP.NET Core Web Application`.

* Escolher a opção `Web Application (Model-View-Controller)
* Em Authentication, clicar em Change e marcar a opção `Individual User Accounts`

Instalar os pacotes

* AutoMapper.Extensions.Microsoft.DependencyInjection
* MediatR.Extensions.Microsoft.DependencyInjection

Em Startup.cs

Configurar o AutoMapper

Configurar o Mediator

Criar o diretório Setup

Dentro do diretório Setup, criar o Arquivo DependencyInjection para resolver as dependências

Configurar a string de conexão com o banco de dados

OBS: Usarei o postgres ao invés de SQL Server

Abrir o Package Manager Console, selecionar o projeto `NerdStore.Domain.Data` executar os comandos:

add-migration Initial -Context CatalogoContext

update-database -Context CatalogoContext

07-12 - Desenvolvendo a apresentação - Validando o funcionamento

Criar dois Controllers

# 08 - CQRS

## 08-01 – Apresentação

### CQRS – Command Query Responsabilty Segregation

* 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 realizados 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.

## 08-02 - Teorema CAP

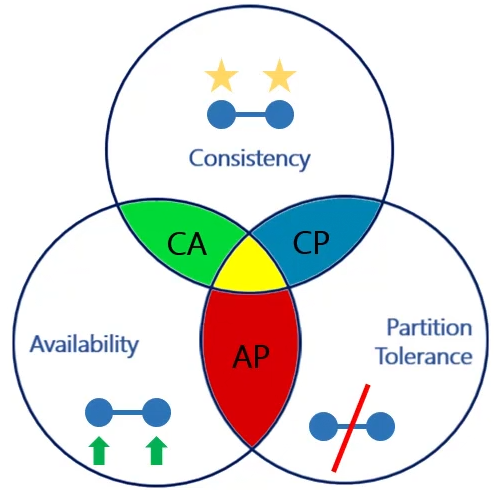
Três aspectos da aplicação que serão desejáveis

* Concistency / Consistência – Que os dados estão sempre consistentes
* Avaliabality / Disponibilidade –
* Partition Tolerance / Tolerancia a Falhas

### Em Resumo:

* **Consistência**
  + Cada leitura recebe a escrita mais recente ou um erro
* **Disponibilidade (Availability)**
  + Cada pedido recebe uma resposta (sem erro) - sem garantia de que contém a escrita mais recente
* **Partição tolerante a falhas**
  + O sistema continua a funcionar apesar de um número arbitrário de mensagens serem descartadas (ou atrasadas) pela rede entre nós

### Deve-se escolher duas opções

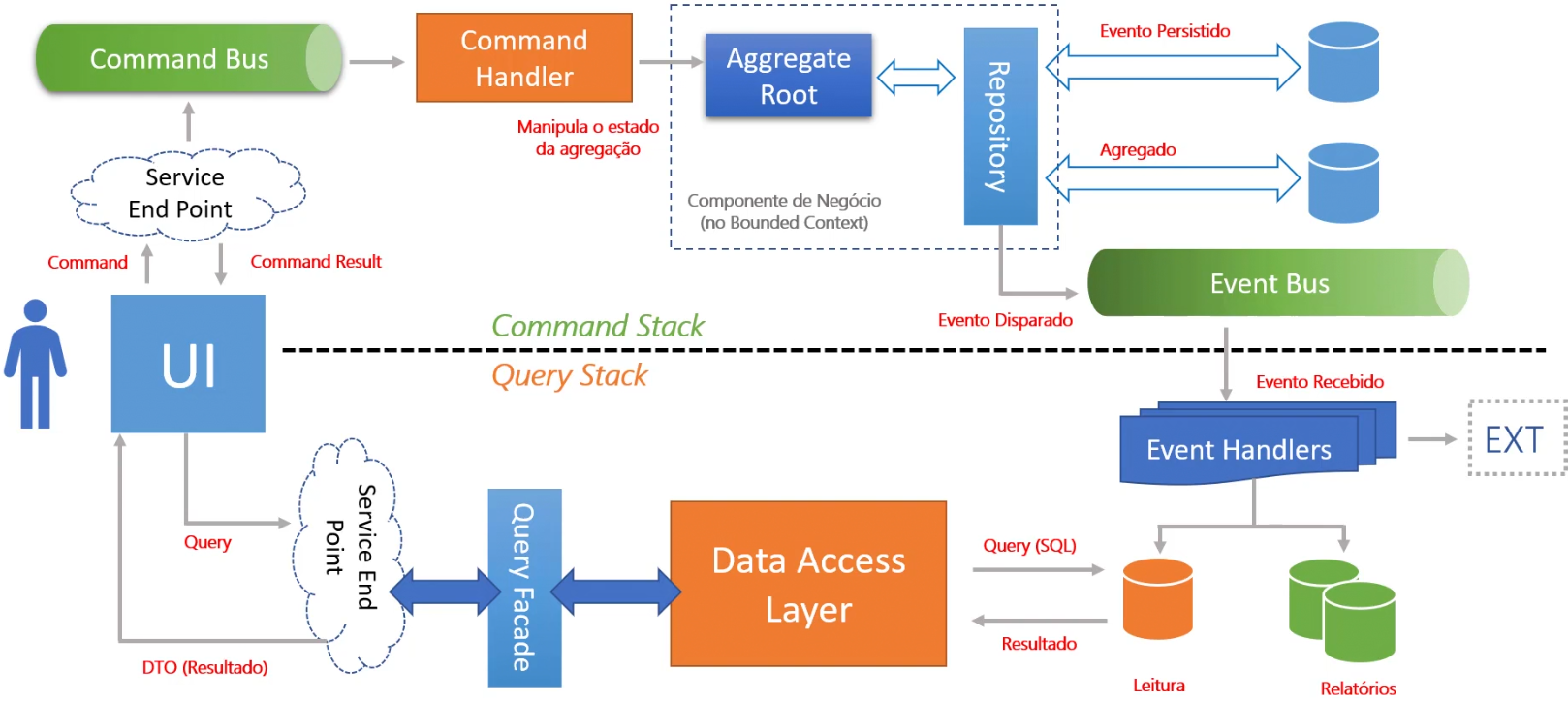


## 08-03 - Como sincronizar as bases

Estratégias para sincronizar as bases

* **Síncrono**
  + Todo comando gera atualizações síncronas
* **Assíncrono**
  + Todos comandos gera atualizações assíncronas
* **Agendado**
  + Um processo roda periodicamente e gera as astualizações
  + Obsolescência de dados
    - Quanto tempo se permite que os dados fiquem desatualizados no banco?
* **Por Demanda**
  + As atualizações são geradas via *request* (se o dado for antigo?

## 08-04 - Command Stack - Query Stack



## 08-05 - Utilização de Sagas

O Design Pattern (padrão de design) saga é uma maneira de gerenciar a consistência de dados em microserviços em cenários de transações distribuídas. Um saga é uma sequência de transações que atualiza cada serviço e publica uma mensagem ou um evento para disparar a próxima etapa da transação. Se uma etapa falhar, o saga executará transações de compensação que neutralizam as transações anteriores. (Microsoft)

<https://docs.microsoft.com/pt-br/azure/architecture/reference-architectures/saga/saga#:~:text=O%20padr%C3%A3o%20de%20design%20saga,microservices%20in%20distributed%20transaction%20scenarios>.

É um Pattern Arquitetural

Porque utilizar uma SAGA

# 09 - Mão na Massa – CQRS

## 09-01 - Setup do BC de Vendas

No contexto Vendas

Criar o projeto NerdStore.Vendas.Domain, do tipo ClassLibrary

Criar o projeto NerdStore.Vendas.Data, do tipo ClassLibrary

Criar o projeto NerdStore.Vendas.Application, do tipo ClassLibrary

Em NerdStore.Venda.Domain, criar os seguintes arquivos:

* Classe Entidade e Agregado **Pedido**
* Classe Entidade **PedidoItem**
* Classe Entidade **Voucher**
* Interface Repositório **IPedidoRepository**
* Enum **PedidoStatus**
* Enum **TipoDesscontoVoucher**

## 09-02 - Objetos de Domínio

Implementação dos arquivos criados na aula anteior

## 09-03 - Command e Command Handler

Instalar os pacotes em NerdStore.Core

* FluentValidation

Criar a classe base Command no diretório Messages, no projeto NerdStore.Core

Criar os seguintes diretórios no projeto NerdStore.Vendas.Application

* Commands
* Events
* Queries

Criar/implementar o Command AdicionarItemPedidoCommand em Commands

Criar/implementar o Handler

Registrar o Handler em DependencyInjection, em NerdStore.WebApp.MVC

## 09-04 - Manipulando comandos na prática

Implementar o Contexto DB `VendasContext` em `NerdStore.Vendas.Data`

Instalar os pacotes:

* Microsoft.EntityFrameworkCore
* Microsoft.EntityFrameworkCore.Design
* Microsoft.EntityFrameworkCore.Relational
* Npgsql.EntityFrameworkCore.PostgreSQL (não usarei o SQL server)

Criar o diretório `Mappings` em `NerdStore.Vendas.Data`

Dentro do diretório `Mappings`, criar os mapeamentos para

* ProdutoMapping
* PedidoItemMapping
* VoucherMapping

Criar o diretório `Repositories` em `NerdStore.Vendas.Data`

Ciar/Implementar o repositório para `ProdutoRepository`

Configurar a string de conexão com o banco de dados em Startup.cs do projeto NerdStore.WebApp.MVC

OBS: Usarei o postgres ao invés de SQL Server

Abrir o Package Manager Console, selecionar o projeto `NerdStore.Vendas.Data` executar os comandos:

add-migration Initial -Context VendasContext

update-database -Context VendasContext

- - - -

## 09-05 - Tratamento de notificações

Em NerdStore.Core, renomear o diretório Bus para Communication

Dentro de Communication, criar o diretório Mediator

Dentro de Domain Objects criar o diretório Messages

Dentro de Messages Criar o diretório CommonMessages

Dentro de CommonMessages criar os diretórios

* DomainEvents
  + Eventos de domínio, específico para notificar um handler que algo aconteceu
* Notifications
  + DomainNotification
* IntegratioEvents

## 09-06 - Event e Event Handler

*Código...*

## 09-07 - Implementando Queries (Query Stack)

*Código...*

## 09-08 - Adicionando novas funcionalidades com comandos

*Código...*

## 09-09 - Integração de BC's - Processamento do pedido

*Código...*

## 09-10 - Integração de BC's – Pagamento

No projeto `NerdStore.Pagamento.Data`:

Implementar o Contexto DB `PagamentoContext`

Instalar os pacotes:

* Microsoft.EntityFrameworkCore
* Microsoft.EntityFrameworkCore.Design
* Microsoft.EntityFrameworkCore.Relational
* Npgsql.EntityFrameworkCore.PostgreSQL (não usarei o SQL server)

Criar o diretório `Mappings`

Dentro do diretório `Mappings`, criar os mapeamentos para

* PagamentoMapping
* TransacaoMapping

Criar o diretório `Repositories` em `NerdStore.Pagamento.Data`

Ciar/Implementar o repositório para `PagamentoRepository`

Configurar a string de conexão com o banco de dados em Startup.cs do projeto NerdStore.WebApp.MVC

OBS: Usarei o postgres ao invés de SQL Server

Abrir o Package Manager Console, selecionar o projeto `NerdStore.Pagamento.Data` executar os comandos:

add-migration Initial -Context PagamentoContext

update-database -Context PagamentoContext

## 09-11 - Integração de BC's - Finalização do pedido

## 09-12 - Visão geral da implementação

## 09-13 - Trabalhando com Sagas, Service Bus e Filas MSMQ

Rebus

# 10 - Event Sourcing

## 10-01 - Sobre o que é

É sobre o passado!

### 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.

### Utilização

* O quanto é importante sabe em que ordem os itens foram adicionados?
* O quanto é importante saber que um item foi excluído?
* Bi, Machine Learning, Testes Automatizados, Simulação real de usuários.

## 10-02 - Fatores essenciais

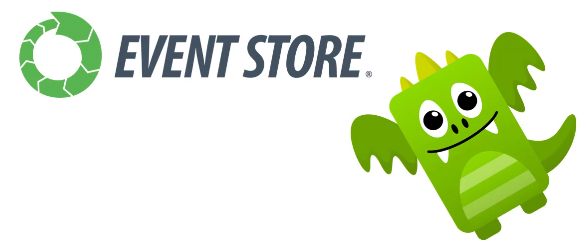
* Todos eventos dizem sobre o que houve no passado
* Eventos são uma expressão da linguagem ubíqua
* Eventos são imperativos e utilizam verbos no passado
* São sempre adicionados, nunca excluídos
* É possível realizar o “Replay” para conhecer o estado anterior da entidade

### Como funciona

...

### Complicado?

* Utilize um Event-base Data Store!



## 10-03 - [Mão na massa] - Setup do ES no projeto

eventstore.org

Criar um diretório lógico, no projeto, chamado BuildinBlocks

Criar um projeto do tipo ClassLibrary, com o nome de EventSourcing

Instalar o pacote

install package EventStore.Client

## 10-04 - [Mão na massa] - Persistindo eventos

## 10-05 - [Mão na massa] - Finalizando a implementação do ES

# 11 - Encerramento

## 11-01 - Palavras finais

* Refazer o curso
* Crie um projeto particular, usando o projeto criado no curso

## 11-02 - Zerei a vida ou só comecei

### O que foi feito?

* DDD
* CQRS
* Filas
* Lançar e Consumir Eventos

### O que fazer

* Aplicações distribuídas
  + Quebrar os contextos em API
* Service Bus
* Serviço de Núvem
* Servless Architecture
* Bancos NoSQl
* Padrões de Segurança na comunicação de rede
* Autenticação entre contextos
  + Token, Federação, Refresh Token, Certificado
* Trabalhar com sistemas escaláveis
  + Docker
  + Kubernets
  + Service Fabric
  + Configuração
  + Gerenciamento
  + Balanceamento
* Testes
  + Unidade
  + Integração
  + Automatizado
* Implementar as rotinas de Testes com DepOps
* Fazer Build automatizado
* Fazer Deploy automatizado
* GIthub
* Azure DevOps

# Anotações Extras

# Mediator

* **Request / Command**
  + Mensagem que será processada
  + Interface
    - *IRequest*
* **Handler / CommandHandler**
  + Processador da mensagem
  + Interface:
    - *IRequestHandler<TRequest>*
* **Mediator**
  + Faz a Intermediação. É responsável por receber um Request e invocar o Handler associado á ele.
  + Interface:
    - *IMediator*
* **Notification / Event**
  + Notificações
  + Interface
    - *INotification*
* **Handler / NotificationHandler or EventHandler**
  + Define um manipulador para uma notificação
  + Interface
    - *INotificationHandler*

**Send()** -> Envie de maneira assíncrona uma solicitação (request / command) para um único gerenciador (handler)

**Publish()** -> Envie de maneira assíncrona uma notificação (notification / event) para vários manipuladores (handler).