



Domain Driven Design

Construindo Sistemas Complexos

Aprenda a modelar sistemas alinhado com o negócio

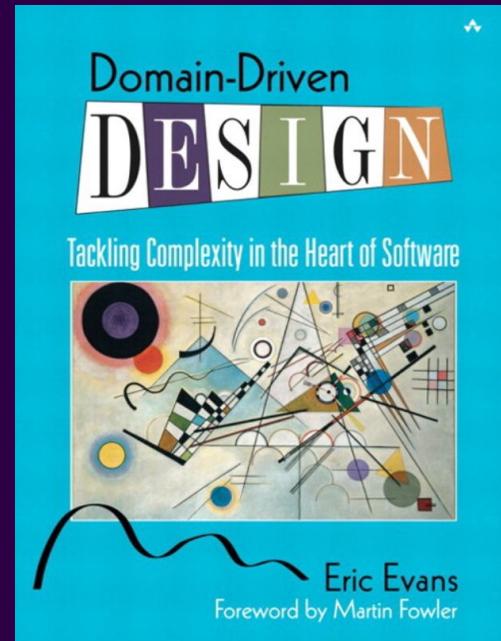
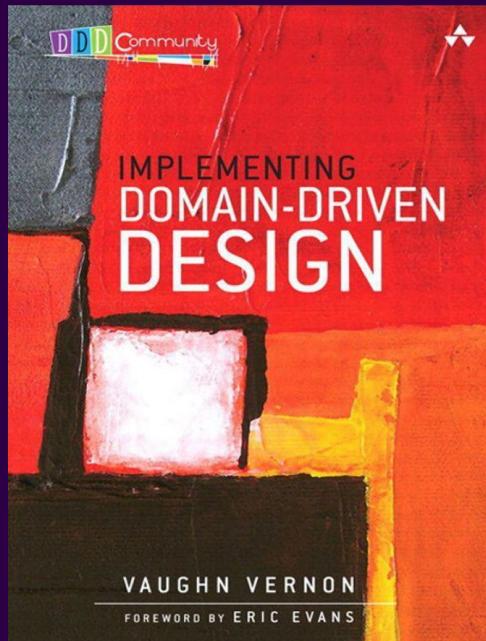
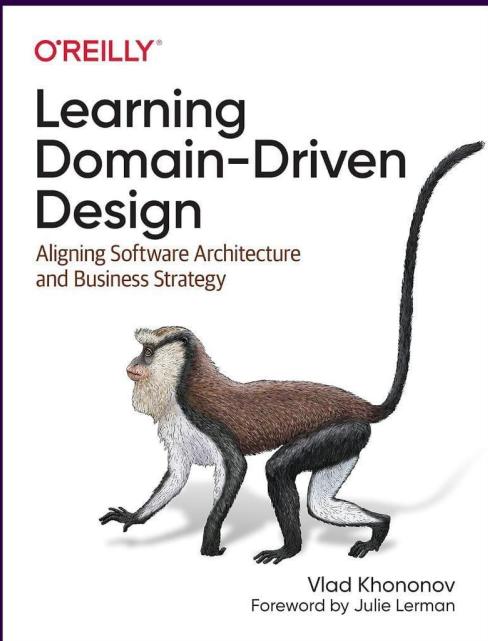


Olá

Eu sou **Willian Brito**

- ❑ Analista de Sistemas na [Aiko](#).
- ❑ Formado em [Análise e Desenvolvimento de Sistemas](#).
- ❑ Pós Graduado em [Segurança Cibernética](#).
- ❑ Certificação **SYCP** (Solyd Certified Pentester) v2018.

Este conteúdo foi baseado nestas obras



“**DDD** é uma abordagem ao design de software que coloca o domínio no centro do processo de desenvolvimento.”

—Erich Evans

Modelagem Estratégica



Tópicos abordados

01

Domínios e Subdomínios

Representa o conjunto de problemas que o software deve resolver.

02

Linguagem ubíqua

Linguagem compartilhada entre os envolvidos no projeto.

03

Contextos Delimitados

Representa uma área claramente definida.

04

Mapa de Contextos

Representação visual das relações e interações entre contextos delimitados.

01

Domínio

Representa o conjunto de problemas que o software deve resolver.

O que é DDD ?

Domain-Driven Design, ou "Design Orientado ao Domínio," é uma abordagem para o desenvolvimento de software que prioriza a compreensão e a modelagem de problemas complexos a partir do ponto de vista do domínio do negócio.

Criado por **Eric Evans**, o DDD propõe que o desenvolvimento de software deve girar em torno do conhecimento profundo das regras, processos e linguagens do domínio para construir modelos de software mais precisos e alinhados aos requisitos de negócio.

O DDD parte do princípio de que a complexidade nos sistemas de software geralmente decorre das regras e requisitos específicos do domínio, ou seja, da área de negócio à qual o sistema se destina. Para lidar com essa complexidade, o DDD sugere uma abordagem centrada no domínio, onde desenvolvedores e especialistas (como gestores e analistas) colaboram para definir uma "**linguagem ubíqua**" compartilhada por todos os envolvidos, e construam modelos de software que reflitam fielmente o domínio.

O que é Domínio ?



Em Domain-Driven Design (DDD), o conceito de **domínio** e **subdomínios** ajuda a dividir e organizar a complexidade de sistemas de software, especialmente aqueles que resolvem problemas específicos de negócios.

Domínio: O domínio representa o contexto central e o conjunto de problemas que o software deve resolver. Ele é a área do negócio ou o escopo funcional em que o sistema opera. Por exemplo, em uma aplicação financeira, o domínio pode incluir conceitos como transações, contas e usuários.

Subdomínios: Subdomínios são partes específicas e delimitadas do domínio. Eles representam divisões do problema geral, possibilitando que equipes trabalhem em áreas menores e mais manejáveis, ao invés de tentar entender e construir o sistema como um todo. No contexto de DDD, os subdomínios são classificados em três tipos principais:

Tipos de Subdomínios ?



1. Domínio Principal (Core Domain)

- O **Core Domain** é o coração do sistema e representa a principal fonte de valor para o negócio. Ele contém a lógica central e as regras mais complexas, sendo a parte mais estratégica e que diferencia a empresa no mercado.
- **Exemplos:** Em um sistema bancário, o core domain pode envolver o gerenciamento de transações e o cálculo de saldo de contas.
- **Importância:** É onde o time de desenvolvimento deve dedicar mais tempo e recursos para garantir que seja o mais robusto e otimizado possível.

Tipos de Subdomínios ?



2. Domínio de Suporte

- O **Domínio de Suporte** complementa o core domain, mas não é o foco principal do negócio. Ele contém funcionalidades importantes, mas que não diferenciam a empresa no mercado.
- **Exemplos:** Em um sistema bancário, o gerenciamento de relatórios internos e a administração de usuários são considerados suportes, pois auxiliam o negócio, mas não são a essência dele.
- **Importância:** Essas funcionalidades são necessárias e agregam valor, mas podem ser menos inovadoras ou estratégicas. O time de desenvolvimento pode buscar soluções de prateleira ou adotar soluções mais simples para esses domínios.

Tipos de Subdomínios ?



3. Domínio Genérico

- O **Domínio Genérico** compreende funcionalidades amplamente utilizadas, que não são específicas do negócio e que geralmente podem ser implementadas com soluções prontas, como bibliotecas ou serviços externos.
- **Exemplos:** Autenticação de usuários e log de auditoria são domínios genéricos, pois são comuns a vários tipos de sistema.
- **Importância:** Ao invés de investir muito tempo nesses domínios, o time pode utilizar ferramentas e bibliotecas existentes para reduzir esforço, deixando mais recursos disponíveis para o core domain.

Tipos de Subdomínios ?



Exemplo Prático

Em uma plataforma de e-commerce:

- **Domínio Principal:** Gerenciamento de pedidos, cálculo de preços e regras de desconto.
- **Domínio de Suporte:** Ferramentas de marketing e notificações aos clientes.
- **Domínio Genérico:** Autenticação de usuários e geração de relatórios.

Tipos de Subdomínios ?



Comparando Subdomínios -

Agora que já temos uma melhor compreensão sobre os três tipos de subdomínios, iremos explorar suas diferenças adicionais e ver como afetam as decisões de design de software.

Apenas **domínios principais** proporcionam **vantagem competitiva** a uma empresa. Os domínios principais são a estratégia da empresa para se diferenciar da concorrência, eles são **complexos** e difíceis da concorrência copiar, a lucratividade da empresa depende disso.

É por isso que, estratégicamente, as empresas procuram resolver problemas complexos em seus domínios principais.

Comparando Subdomínios -

Os **subdomínios genéricos**, por definição, não podem ser uma fonte de vantagem competitiva. Eles são soluções genéricas as mesmas utilizadas pela empresa e por sua concorrência.

Essas soluções **não são simples nem trivial** e deve haver uma boa razão para que outros já tenha investido tempo e esforço para resolver esses problemas, considere por exemplo, algoritmos de criptografia, mecanismos de autenticação e autorização e gateways de pagamentos.

Comparando Subdomínios -

Os **subdomínios de suporte**, têm barreiras de entrada baixa e também não proporcionam vantagem competitiva.

A lógica do subdomínio de suporte é simples. Eles são operações básicas de ETL e interfaces CRUD, e a lógica de negócio óbvia. Frequentemente, nada mais é do que validar inputs ou converter dados de uma estrutura em outra.

Conclusão

Esses tipos de **subdomínios** ajudam o time de desenvolvimento a alocar recursos com inteligência, concentrando-se em desenvolver o **core domain** de forma personalizada e estratégica e adotando abordagens **menos custosas** nos **outros tipos de subdomínios**. Essa divisão facilita o entendimento e a manutenção do sistema e contribui para o desenvolvimento orientado a negócios.

02

Linguagem Ubiqua

Linguagem compartilhada entre os envolvidos.

Linguagem Ubíqua



A **linguagem ubíqua** é um dos conceitos fundamentais no DDD. Ela é uma **linguagem compartilhada** entre todos os envolvidos em um projeto de software, incluindo desenvolvedores, analistas e especialistas do domínio (stakeholders, como clientes ou representantes do negócio).

Essa linguagem não é apenas uma forma de comunicação, mas também uma maneira de representar o domínio no próprio código, garantindo que o software **reflita fielmente as regras e os conceitos do negócio**.

O que é Linguagem Ubíqua ?

A **linguagem ubíqua** é um **vocabulário comum** que todos os membros do projeto devem usar para se referirem a conceitos, processos e objetos no domínio do negócio. Ela deve ser desenvolvida em conjunto entre a equipe técnica e os especialistas do domínio, garantindo que todos compartilhem o mesmo entendimento sobre o que cada termo significa.

- **Inclui termos do domínio:** Substantivos, adjetivos, verbos e expressões que são familiares aos especialistas do negócio.
- **É refletida no código:** Os termos usados no domínio devem ser os mesmos no código, em classes, métodos, interfaces e variáveis.
- **Elimina ambiguidades:** Ao padronizar a linguagem, evitam-se interpretações erradas ou inconsistentes de um mesmo conceito.

Qual a sua Importância ?



A **linguagem ubíqua** é fundamental para o DDD, pois permite a criação de um modelo de domínio preciso, coeso e fácil de entender. Algumas das principais razões para a importância dessa linguagem incluem:

1. Facilita a Comunicação

- Como todos os envolvidos no projeto usam a mesma linguagem, a comunicação entre especialistas e desenvolvedores se torna mais clara e precisa.
- Termos e definições são uniformes, eliminando o risco de mal-entendidos. Quando alguém menciona uma "**Ordem de Venda**" ou uma "**Transação**", todos entendem o mesmo conceito.

Qual a sua Importância ?



2. Aproxima o Domínio do Código

- A linguagem ubíqua aproxima o domínio do código, tornando o software mais compreensível e alinhado com o negócio. Ao olhar para o código, qualquer pessoa envolvida no projeto deve entender o que está sendo feito, pois os termos do negócio são os mesmos usados na programação.
- O modelo de domínio não é apenas uma representação abstrata, mas algo concreto, refletido diretamente no código e nas conversas diárias do time.

Qual a sua Importância ?

3. Reduz Ambiguidades e Erros

- Ambiguidades são comuns quando diferentes pessoas interpretam um mesmo termo de maneiras distintas. A linguagem ubíqua padroniza esses termos, garantindo que todos compartilhem o mesmo entendimento sobre o que cada conceito representa no sistema.
- Isso é especialmente importante em domínios complexos, onde pequenos mal-entendidos podem gerar erros significativos no comportamento do software.

Qual a sua Importância ?



4. Facilita a Evolução do Modelo de Domínio

- Conforme o domínio evolui, a linguagem ubíqua também evolui. A linguagem comum facilita essa adaptação, permitindo que mudanças no negócio sejam mais rapidamente refletidas no software.
- Se uma regra de negócio muda, os termos também são atualizados na linguagem, e o código pode ser adaptado para manter o alinhamento com o domínio.

Qual a sua Importância ?



5. Promove a Colaboração entre Times

- A linguagem ubíqua permite que desenvolvedores, analistas, especialistas e até usuários finais participem ativamente das discussões sobre o modelo. Todos têm o contexto necessário para colaborar, pois falam a mesma linguagem do domínio.

Qual a sua Importância ?



Exemplo Prático de Linguagem Ubíqua:

Imagine uma empresa de logística. No modelo de domínio, temos termos como "**Carga**", "**Entrega**", "**Rota**" e "**Motorista**". Esses termos são usados nos requisitos, nas discussões e, finalmente, no próprio código:

- A classe `Carga` representa a carga a ser transportada.
- A classe `Entrega` representa uma entrega específica.
- O método `CalcularRota` está em uma classe `Rota`, refletindo a lógica de cálculo de rotas.

Conclusão

A **linguagem ubíqua** é um dos pilares que sustentam o **DDD**. Ela não é apenas uma linguagem de comunicação, mas uma forma de **garantir** que o **modelo de domínio seja fiel ao negócio**, que o código seja comprehensível por todos os envolvidos e que o software possa evoluir sem perder a consistência com o domínio.

03 Contextos Delimitados

Representa uma área claramente definida.

Contexto Delimitado



Contextos Delimitados (Bounded Contexts) são um conceito fundamental em DDD e representam a divisão do domínio em **áreas claramente definidas**, cada uma com seu próprio modelo de domínio e linguagem ubíqua.

O objetivo dos contextos delimitados é ajudar a organizar e lidar com a complexidade de sistemas grandes ou com múltiplos aspectos, permitindo que diferentes partes do sistema evoluam de maneira independente e ao mesmo tempo mantenham consistência.

O que é Contexto Delimitado? -

Um contexto delimitado é uma **área do sistema onde um modelo de domínio específico é definido, consistente e usado de forma única**. Ele estabelece fronteiras claras dentro das quais os termos e regras de negócio têm um significado bem definido. Cada contexto delimitado tem sua própria linguagem ubíqua, evitando conflitos de nomenclatura e conceitos que poderiam surgir ao usar um único modelo em um domínio vasto.

Em outras palavras, contextos delimitados são a "**moldura**" dentro da qual um modelo de domínio faz sentido e é aplicável. Eles permitem que diferentes partes do sistema, que podem lidar com subdomínios distintos, usem modelos e linguagens adequados para cada parte do negócio.

Qual o seu Papel ?



Os contextos delimitados têm vários papéis importantes no DDD:

1. Gerenciar a Complexidade do Domínio:

- Dividir o domínio em contextos delimitados ajuda a tornar cada parte do sistema mais simples e organizada. Ao invés de tentar aplicar um único modelo para representar todos os aspectos do negócio, o DDD incentiva o uso de modelos específicos para cada contexto.
- Essa divisão facilita o entendimento e manutenção do código, reduzindo o impacto das mudanças em um contexto sobre os demais.

Qual o seu Papel ?



2. Isolar Linguagens Ubíquas:

- Cada contexto delimitado tem sua própria linguagem ubíqua, alinhada ao modelo e às regras de negócio específicas daquela área. Isso evita ambiguidades e conflitos de termos que podem surgir em sistemas grandes.
- Por exemplo, o termo "**Cliente**" pode ter um significado no contexto de "**Vendas**" (focado em transações e relacionamento) e outro no contexto de "**Supporte**" (focado em casos de atendimento e histórico de reclamações). Separar esses conceitos em contextos evita interpretações conflitantes.

Qual o seu Papel ?



3. Permitir Evolução Independente dos Contextos:

- Como cada contexto é autônomo, ele pode ser desenvolvido e evoluir de maneira independente dos outros. Mudanças em um contexto não afetam diretamente os outros, o que permite maior flexibilidade para adaptar o sistema às necessidades do negócio.
- Essa separação é particularmente útil em sistemas distribuídos ou que adotam arquitetura de microsserviços, pois permite que cada contexto funcione como um serviço isolado e focado em seu próprio modelo de domínio.

Qual o seu Papel ?



4. Definir Limites de Comunicação:

- Contextos delimitados definem onde e como os modelos interagem. Quando um contexto precisa se comunicar com outro, isso é feito por meio de integrações bem definidas, como APIs ou eventos, ao invés de compartilhar internamente modelos e dados.
- Essa abordagem minimiza o acoplamento entre os contextos, o que permite que cada um mantenha seu próprio modelo e evolua de maneira independente.

Qual o seu Papel ?



5. Auxiliar na Escalabilidade Times:

- Em grandes equipes de desenvolvimento, contextos delimitados facilitam a distribuição do trabalho, pois diferentes equipes podem se especializar em contextos específicos. Isso ajuda a equipe a ter maior foco e domínio sobre uma área particular do sistema.
- Ao definir claramente as fronteiras, fica mais fácil gerenciar a colaboração entre equipes e evitar conflitos de modelo e linguagem.

Relação com Subdomínios -

Contextos Delimitados e subdomínios são conceitos relacionados, mas diferentes:

- **Subdomínios** referem-se às divisões do problema no nível do domínio do negócio. Eles representam áreas funcionais que podem ser classificadas como **Principal, Suporte** ou **Genérico**.
- **Contextos Delimitados** são as implementações técnicas de um modelo de domínio. Em outras palavras, enquanto o subdomínio está focado no negócio, o contexto delimitado está focado na solução.

Em um sistema grande, pode haver um ou mais contextos delimitados implementando os mesmos subdomínios, ou um único contexto pode abranger vários subdomínios dependendo da complexidade e das necessidades do negócio.

Exemplo Prático

Em nossa plataforma de e-commerce, poderíamos ter os seguintes contextos delimitados:

- **Contexto de Vendas:** Lida com a criação e gerenciamento de pedidos. Aqui, termos como "**Pedido**", "**Cliente**" e "**Carrinho**" têm um significado específico para o fluxo de venda.
- **Contexto de Estoque:** Cuida do gerenciamento de inventário e disponibilidade de produtos. Os conceitos de "**Produto**", "**Quantidade em Estoque**" e "**Reabastecimento**" são aplicados de forma única.
- **Contexto de Pagamento:** Gerencia a integração com sistemas de pagamento. "**Transação**", "**Cartão de Crédito**" e "**Autorização**" são termos importantes dentro desse contexto.

Nesses contextos, um termo como "**Produto**" pode existir em ambos os contextos de Vendas e Estoque, mas seu comportamento e propriedades podem ser diferentes em cada um deles.

Conclusão



Contextos Delimitados são fundamentais no **DDD** para organizar e lidar com a complexidade de sistemas de software complexos. Eles oferecem uma maneira de dividir o domínio em áreas coesas, onde cada uma possui uma linguagem própria e um modelo específico.

Dessa forma, contextos delimitados ajudam a manter o código organizado, reduzem ambiguidades e facilitam a evolução do sistema sem comprometer a consistência do domínio como um todo.

04

Mapa de Contexto

Representação visual das relações e interações entre contextos delimitados.

Mapa de Contexto



Um **Mapa de Contexto (Context Map)** é uma ferramenta utilizada em **DDD** para representar as relações entre os diferentes **Contextos Delimitados** de um sistema. Ele fornece uma visão geral de como os contextos interagem e colaboram entre si, mapeando as dependências, fluxos de dados e interações entre os módulos.

Essa visualização ajuda as equipes a entender melhor a **arquitetura do sistema** e a identificar as áreas de cooperação e de isolamento entre os contextos.

O que é Mapa de Contexto ? -

O Mapa de Contexto é uma diagramação que destaca:

- Quais são os contextos delimitados existentes no sistema.
- Quais relações existem entre esses contextos, como comunicação de dados, colaboração ou dependência.
- Os tipos de relações e responsabilidades entre contextos, descrevendo quem depende de quem, como a informação é trocada e o que deve ser feito para manter a consistência dos dados e operações entre contextos.

O Mapa de Contexto ajuda a definir a **estratégia de integração** e **comunicação** entre as equipes de desenvolvimento, criando uma linguagem comum e eliminando possíveis ambiguidades e conflitos que poderiam surgir quando diferentes equipes trabalham em áreas interconectadas do sistema.

Padrões de Integração

No Mapa de Contexto, existem diferentes tipos de relações entre contextos delimitados. Cada tipo de relação define como a comunicação ocorre e quais são as responsabilidades de cada contexto. Alguns dos padrões de relacionamento mais comuns incluem:

1. Shared Kernel (Núcleo Compartilhado):

- Um conjunto de componentes ou lógica de domínio que é compartilhado entre dois ou mais contextos. Esses contextos dependem do mesmo núcleo de funcionalidades essenciais.
- **Exemplo:** Em uma empresa de vendas, tanto o contexto de “**Vendas**” quanto o de “**Faturamento**” podem compartilhar uma definição comum de “**Cliente**”.

Padrões de Integração



2. Customer-Supplier (Cliente-Fornecedor):

- Um contexto “**Cliente**” depende das informações ou serviços fornecidos por outro contexto, o “**Fornecedor**”. O contexto fornecedor desenvolve e mantém funcionalidades que o cliente usa diretamente.
- **Exemplo:** Um contexto de “**Pedidos**” depende do contexto de “**Estoque**” para verificar a disponibilidade de produtos antes de confirmar um pedido.

Padrões de Integração



3. Conformist (Conformista):

- O contexto conformista depende de outro contexto, mas não tem influência ou controle sobre ele. O contexto conformista adota e segue as definições do outro contexto.
- **Exemplo:** Em uma integração com um sistema externo onde a equipe deve seguir os padrões estabelecidos pela outra parte, como uma API de terceiros.

Padrões de Integração



4. Anticorruption Layer (Camada Anticorrupção):

- Um padrão que protege o contexto interno de influências externas indesejadas. A camada anticorrupção atua como um adaptador entre contextos, traduzindo dados e conceitos entre eles para evitar poluição conceitual.
- **Exemplo:** Em uma integração com um sistema legado, onde é preciso adaptar os dados para o modelo de domínio sem trazer inconsistências.

Padrões de Integração



5. Open Host Service (Serviço de Host Aberto):

- Um contexto fornece um conjunto de funcionalidades para outros contextos através de uma interface pública e bem definida, permitindo acesso controlado ao modelo de domínio.
- **Exemplo:** Um contexto que disponibiliza serviços para outros sistemas através de uma API REST.

Padrões de Integração



5. Separate Ways (Caminhos Separados):

- Contextos que funcionam de forma independente e não têm dependência ou colaboração direta entre si. Eles podem até resolver problemas semelhantes, mas não precisam compartilhar ou sincronizar dados.
- **Exemplo:** Um sistema de RH e um sistema de contabilidade que pertencem à mesma organização, mas funcionam de forma totalmente independente.

Qual sua Importância ?



O Mapa de Contexto oferece uma visão estratégica e prática para a arquitetura de sistemas em DDD. Sua importância inclui:

- 1. Clareza nas Interações Entre Equipes:** Cada equipe sabe exatamente como seu contexto interage com outros, evitando conflitos e alinhando as expectativas sobre as responsabilidades de cada um.
- 2. Melhora na Arquitetura do Sistema:** Ajuda na identificação de acoplamentos indesejados e possibilita a criação de soluções, como camadas anticorrupção, que minimizam problemas.
- 3. Definição de Fronteiras Claras:** Facilita a divisão do sistema em áreas coesas e alinhadas ao negócio, garantindo que as dependências e fluxos de dados sejam bem controlados.

Qual sua Importância ?



4. Aprimora a Comunicação e a Documentação: Serve como uma ferramenta de documentação visual para todos os envolvidos, incluindo novos membros da equipe, stakeholders e gerentes, ajudando a compartilhar uma visão comum do sistema.

5. Facilita a Evolução e Escalabilidade: Ao dividir o sistema em contextos bem definidos e desacoplados, o Mapa de Contexto permite que cada parte evolua de forma independente, promovendo maior escalabilidade e manutenção.

Exemplo Prático

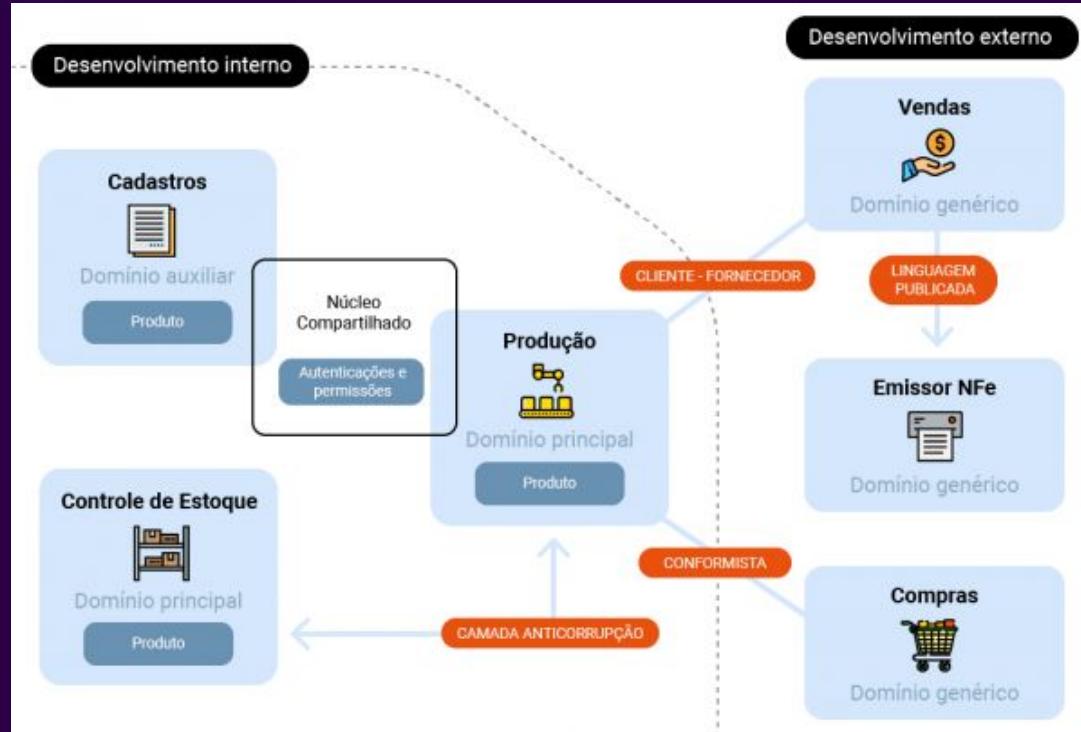
Imagine que nossa plataforma de e-commerce possui os seguintes contextos delimitados:

- **Vendas:** Responsável pelo processo de compras e pedidos.
- **Estoque:** Lida com a disponibilidade de produtos e atualização de inventário.
- **Pagamentos:** Gerencia transações financeiras e autorizações.
- **Entrega:** Cuida do transporte e rastreamento de pedidos.

No Mapa de Contexto:

- O contexto **Vendas** pode ser cliente do contexto **Estoque**, verificando a disponibilidade de itens.
- O contexto **Pagamentos** pode fornecer um serviço de “**Open Host Service**” para que o contexto **Vendas** processe os pagamentos.
- O contexto **Entrega** é acionado pelo contexto **Vendas** ao confirmar um pedido.

Exemplo Prático



Conclusão



O **Mapa de Contexto** é uma **representação visual** e estratégica das relações e interações entre contextos delimitados no DDD.

Ele ajuda a definir como as partes do sistema colaboram e onde estão as fronteiras de responsabilidade, permitindo que o software seja mais modular, escalável e fácil de manter.

Modelagem Tática



Tópicos abordados

05

Entidades

Objetos do domínio com identidade única.

06

Objetos de Valor

Classe de objetos que representam valores imutáveis

07

Agregados

Garante a consistência transacional.

08

Serviços de Domínio

Serviços com lógica de negócio compartilhada.

09

Eventos de Domínio

Representam algo que aconteceu no domínio.

10

Repositório

Forma de encapsular a lógica de acesso a dados.

05

Entidades

Objetos do domínio com identidade única.

Entidades



Em Domain-Driven Design (DDD), **Entidades** são um dos elementos fundamentais para representar o modelo de domínio.

Elas são objetos do domínio que possuem uma identidade única e persistente, que é o que as diferencia de outros objetos mesmo que tenham as mesmas propriedades.

Essa identidade permanece a mesma durante todo o ciclo de vida da entidade, independentemente de mudanças em seus atributos ou estado interno.

Características

- **Identidade:** Cada entidade deve ter um identificador único (como uma propriedade Id ou UUID). Esse identificador é a única forma de diferenciar uma entidade de outra, mesmo que todas as outras propriedades sejam iguais.
- **Ciclo de Vida:** As entidades geralmente possuem um ciclo de vida dentro do domínio e podem passar por diferentes estados.
- **Igualdade Baseada na Identidade:** Diferente de objetos de valor, as entidades são consideradas iguais com base em seu identificador e não em suas propriedades.

Exemplo Prático

Vamos codificar nossa plataforma de e-commerce, onde uma entidade central poderia ser **Order (Pedido)**. Este pedido possui várias propriedades que representam seu estado, mas o que o diferencia é seu identificador único.

```
public class Order
{
    public Guid Id { get; private set; }
    public DateTime OrderDate { get; private set; }
    public Customer Customer { get; private set; }
    public List<OrderItem> Items { get; private set; } = new List<OrderItem>();
    public decimal TotalAmount => Items.Sum(item => item.TotalPrice);

    public Order(Customer customer)
    {
        Id = Guid.NewGuid(); // Gera um identificador único para a ordem
        OrderDate = DateTime.UtcNow;
        Customer = customer;
    }

    public void AddItem(Product product, int quantity)
    {
        var orderItem = new OrderItem(product, quantity);
        Items.Add(orderItem);
    }

    public void RemoveItem(Guid itemId)
    {
        var item = Items.FirstOrDefault(i => i.Id == itemId);
        if (item != null)
        {
            Items.Remove(item);
        }
    }
}
```

Exemplo Prático

Aqui, cada **OrderItem** representa um item de um pedido e também tem um identificador. Este exemplo mostra a composição de uma entidade maior (Order) com uma entidade menor (OrderItem).

```
public class OrderItem
{
    public Guid Id { get; private set; }
    public Product Product { get; private set; }
    public int Quantity { get; private set; }
    public decimal UnitPrice => Product.Price;
    public decimal TotalPrice => Quantity * UnitPrice;

    public OrderItem(Product product, int quantity)
    {
        Id = Guid.NewGuid();
        Product = product;
        Quantity = quantity;
    }
}
```

Exemplo Prático

No exemplo abaixo, **Customer** também é uma entidade, pois possui uma identidade própria e persistente no sistema.

```
public class Customer
{
    public Guid Id { get; private set; }
    public string Name { get; private set; }
    public string Email { get; private set; }

    public Customer(string name, string email)
    {
        Id = Guid.NewGuid();
        Name = name;
        Email = email;
    }
}
```

Boas Práticas

- **Usar Identidades Únicas:** Utilize identificadores únicos e imutáveis para identificar as entidades. No exemplo, Guid é usado, mas outras opções podem ser apropriadas dependendo do contexto.
- **Proteger o Estado Interno:** Use modificadores private ou protected nos setters das propriedades para controlar como e onde o estado da entidade pode ser alterado. Isso ajuda a manter a consistência e encapsulamento.
- **Métodos de Comportamento:** Crie métodos que representem as operações de negócio, ao invés de expor diretamente as propriedades. Isso mantém a lógica de domínio dentro das entidades e evita que a lógica seja espalhada pelo código.
- **Igualdade Baseada no Id:** A comparação de igualdade entre entidades deve ser baseada no identificador. Assim, duas instâncias de Order com o mesmo Id representam o mesmo pedido, mesmo que tenham atributos diferentes.

Conclusão

Entidades são elementos centrais no DDD e representam objetos com identidade única no domínio. Compreender como projetá-las corretamente ajuda a manter o sistema coerente, organizado e focado no domínio de negócios.

Os exemplos demonstram como criar entidades com identidades únicas, encapsulando a lógica de negócio e protegendo o estado interno para garantir consistência e clareza no código.

06

Objetos de Valor

Classe de objetos que representam valores
imutáveis

Objetos de Valor



Em DDD, **Objetos de Valor (Value Objects)** são uma classe de objetos que representam uma quantidade ou descrição que **não possui identidade própria**, mas sim características ou valores pelos quais são definidos.

Diferente das entidades, que são identificadas por um ID único, os objetos de valor são identificados pela igualdade de seus atributos.

Características

- **Imutabilidade:** Um objeto de valor é normalmente imutável. Se houver uma mudança em seu valor, cria-se um novo objeto de valor ao invés de modificar o existente.
- **Sem Identidade Única:** Objetos de valor não têm um identificador próprio. Dois objetos de valor são considerados iguais se todos os seus atributos são iguais.
- **Descarte:** Objetos de valor podem ser criados e descartados livremente, uma vez que seu propósito é representar dados temporários ou imutáveis que pertencem a uma entidade.

Exemplo Prático

Vamos adicionar o campo **Email** na entidade **Customer** e implementar Email como um objeto de valor.

O objeto de valor Email validará seu próprio valor no momento da criação e evitará a criação de objetos inválidos.

```
public class Email
{
    public string Address { get; }

    public Email(string address)
    {
        if (string.IsNullOrWhiteSpace(address))
            throw new ArgumentException("O email não pode ser vazio.");

        // Regex básica para validação de email
        if (!Regex.IsMatch(address, @"^[\w\.-]+@[^\w\.-]+\.[^\w\.-]+$"))
            throw new ArgumentException("Formato de email inválido.");

        Address = address;
    }

    // Override de igualdade baseado no valor, não em identidade
    public override bool Equals(object obj)
    {
        if (obj is Email otherEmail)
            return Address == otherEmail.Address;

        return false;
    }

    public override int GetHashCode()
    {
        return Address.GetHashCode();
    }

    public override string ToString()
    {
        return Address;
    }
}
```

Exemplo Prático

Agora, vamos atualizar a classe **Customer** para utilizar o objeto de valor **Email** ao invés de um string.

```
public class Customer
{
    public Guid Id { get; private set; }
    public string Name { get; private set; }
    public Email Email { get; private set; }

    public Customer(string name, Email email)
    {
        Id = Guid.NewGuid();
        Name = name ?? throw new ArgumentNullException(nameof(name));
        Email = email ?? throw new ArgumentNullException(nameof(email));
    }

    public void UpdateEmail(Email newEmail)
    {
        Email = newEmail ?? throw new ArgumentNullException(nameof(newEmail));
    }
}
```

```
var email = new Email("example@example.com");
var customer = new Customer("João Silva", email);

Console.WriteLine($"Cliente: {customer.Name}, Email: {customer.Email}");

// Atualizar o email do cliente
var newEmail = new Email("novoemail@example.com");
customer.UpdateEmail(newEmail);

Console.WriteLine($"Email atualizado: {customer.Email}");
```

Vantagens

- **Confiabilidade:** Ao encapsular a validação e criação no objeto de valor, é possível garantir que o email esteja sempre em um formato válido.
- **Reduz Ambiguidade:** Ao nomear e definir um Email como objeto de valor, o código torna-se mais legível e explícito, representando a intenção de domínio.
- **Reutilização:** Email pode ser reutilizado em outras entidades, como Supplier ou Employee, que também precisem de um email validado.

Conclusão



Objetos de valor como **Email** são elementos essenciais para DDD, pois encapsulam dados que têm valor próprio mas não identidade única. Isso ajuda a manter o domínio organizado, com dados bem definidos e validados, permitindo que a lógica de domínio se torne mais expressiva e fácil de manter.

07 Agregados

Garante a consistência transacional.

Agregados



No Domain-Driven Design (DDD), **Agregados** representam um conjunto de entidades e objetos de valor que estão logicamente associados e precisam ser tratados como uma única unidade de consistência e alteração.

Em um agregado, há sempre uma entidade principal chamada **Raiz do Agregado (Aggregate Root)**, que gerencia a integridade e as regras de negócio de todo o agregado.

Outras entidades ou objetos de valor dentro do agregado só podem ser acessados através da raiz.

Qual sua Importância ?

- **Consistência e Integridade:** Agregados garantem que as regras de negócio sejam aplicadas de maneira consistente. Todas as operações devem passar pela raiz do agregado, assegurando que o estado de cada entidade esteja consistente.
- **Isolamento:** Os agregados são projetados para serem alterados e persistidos independentemente. Isso ajuda a definir limites de transação, facilitando o trabalho com dados em sistemas distribuídos ou em cenários de alta concorrência.
- **Escopo de Responsabilidade:** Cada agregado encapsula uma parte do domínio e as regras de negócio que lhe pertencem, promovendo organização e clareza.

Exemplo Prático



Vamos implementar um exemplo onde:

- Order (Pedido) é a **Raiz do Agregado**.
- OrderItem (Item de Pedido) é uma **Entidade associada a Order**, representando um item específico do pedido.
- Customer (Cliente) não faz parte do agregado Order mas é associado por uma referência, pois não precisa ser gerenciado internamente por Order.

Exemplo Prático

Entidade Customer (Referência Externa): Aqui, Customer é uma entidade, mas ela não faz parte do agregado Order. Em vez disso, Order possui uma referência a um Customer que representa o cliente que realizou o pedido. Esse relacionamento é tratado como uma associação e não como parte do agregado.

```
public class Customer
{
    public Guid Id { get; private set; }
    public string Name { get; private set; }
    public Email Email { get; private set; }

    public Customer(string name, Email email)
    {
        Id = Guid.NewGuid();
        Name = name ?? throw new ArgumentNullException(nameof(name));
        Email = email ?? throw new ArgumentNullException(nameof(email));
    }

    public void UpdateEmail(Email newEmail)
    {
        Email = newEmail ?? throw new ArgumentNullException(nameof(newEmail));
    }
}
```

Exemplo Prático

Objeto de Valor Email: O objeto de valor Email encapsula as validações de um endereço de email e é usado pela entidade Customer.

```
public class Email
{
    public string Address { get; }

    public Email(string address)
    {
        if (string.IsNullOrWhiteSpace(address))
            throw new ArgumentException("O email não pode ser vazio.");

        // Regex básica para validação de email
        if (!Regex.IsMatch(address, @"^[\w\.-]+@[^\w\.-]+\.[^\w\.-]+$"))
            throw new ArgumentException("Formato de email inválido.");

        Address = address;
    }

    // Override de igualdade baseado no valor, não em identidade
    public override bool Equals(object obj)
    {
        if (obj is Email otherEmail)
            return Address == otherEmail.Address;

        return false;
    }

    public override int GetHashCode()
    {
        return Address.GetHashCode();
    }

    public override string ToString()
    {
        return Address;
    }
}
```

Exemplo Prático

Entidade OrderItem: OrderItem representa um item em um pedido. Ele faz parte do agregado **Order** e só pode ser acessado ou modificado através de Order.

```
public class OrderItem
{
    public Guid Id { get; private set; }
    public Product Product { get; private set; }
    public int Quantity { get; private set; }
    public decimal UnitPrice => Product.Price;
    public decimal TotalPrice => Quantity * UnitPrice;

    public OrderItem(Product product, int quantity)
    {
        Id = Guid.NewGuid();
        Product = product ?? throw new ArgumentNullException(nameof(product));
        Quantity = quantity > 0 ? quantity : throw new ArgumentException("A quantidade deve ser maior que zero.");
    }

    public void UpdateQuantity(int quantity)
    {
        if (quantity <= 0)
            throw new ArgumentException("A quantidade deve ser maior que zero.");

        Quantity = quantity;
    }
}
```

Exemplo Prático

Raiz do Agregado (Order): A classe Order é a raiz do agregado e gerencia o ciclo de vida de OrderItem. Todas as operações de OrderItem, como adicionar ou remover itens, devem ser feitas por meio de Order.

```
public class Order
{
    public Guid Id { get; private set; }
    public DateTime OrderDate { get; private set; }
    public Customer Customer { get; private set; }
    private List<OrderItem> _items = new List<OrderItem>();
    public IReadOnlyCollection<OrderItem> Items => _items.AsReadOnly();
    public decimal TotalAmount => _items.Sum(item => item.TotalPrice);

    public Order(Customer customer)
    {
        Id = Guid.NewGuid();
        OrderDate = DateTime.UtcNow;
        Customer = customer ?? throw new ArgumentNullException(nameof(customer));
    }
}
```

```
public void AddItem(Product product, int quantity)
{
    if (product == null)
        throw new ArgumentNullException(nameof(product));

    var orderItem = new OrderItem(product, quantity);
    _items.Add(orderItem);
}

public void RemoveItem(Guid itemId)
{
    var item = _items.FirstOrDefault(i => i.Id == itemId);
    if (item != null)
    {
        _items.Remove(item);
    }
}

public void UpdateItemQuantity(Guid itemId, int quantity)
{
    var item = _items.FirstOrDefault(i => i.Id == itemId);
    if (item != null)
    {
        item.UpdateQuantity(quantity);
    }
    else
    {
        throw new ArgumentException("Item não encontrado no pedido.");
    }
}
```

Características

- **Encapsulamento:** Order encapsula a lista `_items`, expondo-a apenas como leitura (`IReadOnlyCollection`). Isso impede que itens sejam modificados diretamente fora da raiz do agregado.
- **Consistência:** Order fornece métodos como `AddItem`, `RemoveItem` e `UpdateItemQuantity` para gerenciar `OrderItem`. Assim, as regras de negócio para adicionar ou atualizar itens são aplicadas no nível do agregado, garantindo consistência.
- **Total do Pedido:** `TotalAmount` é uma propriedade calculada que soma o total dos preços dos itens. Essa propriedade é gerenciada internamente pela raiz do agregado e reflete o estado atual de Order e seus OrderItems.

Exemplo de Uso

```
var customer = new Customer(Guid.NewGuid(), "João Silva", new Email("joao.silva@example.com"));
var order = new Order(customer);

var product1 = new Product("Produto A", 100.0m);
var product2 = new Product("Produto B", 50.0m);

order.AddItem(product1, 2);
order.AddItem(product2, 1);

Console.WriteLine($"Total do pedido: {order.TotalAmount}");

order.UpdateItemQuantity(order.Items.First().Id, 3);
Console.WriteLine($"Total do pedido após atualização: {order.TotalAmount}");
```

Conclusão



Agregados são importantes para o DDD porque agrupam entidades e objetos de valor relacionados, garantindo que as regras de negócio sejam aplicadas de maneira **consistente**. No exemplo, Order é o agregado e sua raiz, controlando e encapsulando as operações em OrderItem.

Essa abordagem melhora a coesão do código, facilitando a manutenção e garantindo a integridade do modelo de domínio.



08 Serviços de Domínios

Serviços com lógica de negócio compartilhada.

Serviços de Domínio



Em DDD, **Serviços de Domínio** são componentes que encapsulam a lógica de negócio que não se encaixa naturalmente em uma entidade ou objeto de valor.

Quando uma regra de negócio não pode ser atribuída a uma entidade específica ou a um objeto de valor, mas ainda faz parte do domínio, é comum criar um serviço de domínio para representar essa lógica.

Características

- **Sem Estado:** Serviços de domínio geralmente não mantêm estado próprio, operando apenas com os dados passados para eles.
- **Focados em Regras de Negócio:** Diferente de serviços de aplicação, que tratam lógica de aplicação e fluxo de operações, serviços de domínio contêm regras de negócio que fazem sentido no contexto do domínio.
- **Nomenclatura Expressiva:** Devem possuir um nome que explique claramente o propósito do serviço, como **PaymentProcessingService**, **OrderDiscountCalculator**, etc.

Exemplo Prático



Vamos considerar um cenário em nossa plataforma de e-commerce, onde queremos **calcular o desconto** de um pedido (Order). Essa lógica de desconto pode depender de múltiplos fatores (quantidade de itens, valor total, cliente, etc.) e não se encaixa bem na entidade Order ou nos objetos de valor.

Por isso, implementaremos um serviço de domínio para encapsular essa lógica.

Exemplo Prático

```
public class Order
{
    public Guid Id { get; private set; }
    public DateTime OrderDate { get; private set; }
    private List<OrderItem> _items = new List<OrderItem>();
    public IReadOnlyCollection<OrderItem> Items => _items.AsReadOnly();
    public decimal TotalAmount => _items.Sum(item => item.TotalPrice);

    public Order()
    {
        Id = Guid.NewGuid();
        OrderDate = DateTime.UtcNow;
    }

    public void AddItem(Product product, int quantity)
    {
        var orderItem = new OrderItem(product, quantity);
        _items.Add(orderItem);
    }
}
```

```
public class Product
{
    public string Name { get; }
    public decimal Price { get; }

    public Product(string name, decimal price)
    {
        Name = name;
        Price = price;
    }
}

public class OrderItem
{
    public Product Product { get; private set; }
    public int Quantity { get; private set; }
    public decimal UnitPrice => Product.Price;
    public decimal TotalPrice => Quantity * UnitPrice;

    public OrderItem(Product product, int quantity)
    {
        Product = product ?? throw new ArgumentNullException(nameof(product));
        Quantity = quantity;
    }
}
```

Exemplo Prático

Serviço de Domínio (DiscountCalculator): Esse serviço contém a lógica para calcular um desconto em um pedido. Ele pode levar em conta várias regras de negócio para determinar o valor do desconto.

```
public class DiscountCalculator
{
    public decimal CalculateDiscount(Order order)
    {
        if (order == null)
            throw new ArgumentNullException(nameof(order));

        decimal discount = 0;

        // Regra de desconto baseada no valor total do pedido
        if (order.TotalAmount > 1000)
            discount += order.TotalAmount * 0.10m;

        // Regra de desconto baseada no número de itens
        if (order.Items.Count > 5)
            discount += order.TotalAmount * 0.05m;

        return discount;
    }
}
```

Exemplo Prático

```
var product1 = new Product("Produto A", 200.0m);
var product2 = new Product("Produto B", 300.0m);

var order = new Order();
order.AddItem(product1, 3);
order.AddItem(product2, 2);

var discountCalculator = new DiscountCalculator();
decimal discount = discountCalculator.CalculateDiscount(order);

Console.WriteLine($"Desconto aplicado: {discount:C}");
Console.WriteLine($"Total do pedido após desconto: {order.TotalAmount - discount:C}");
```

Considerações

- **Desacoplamento de Lógica de Negócio:** DiscountCalculator encapsula a lógica de cálculo de desconto, evitando acoplar essa lógica diretamente na classe Order. Isso facilita a manutenção e a testabilidade.
- **Coesão:** DiscountCalculator possui um único propósito, **calcular descontos para pedidos**, respeitando as regras de negócio. Isso aumenta a coesão e permite evoluir a lógica de desconto sem impactar outras partes do sistema.
- **Reutilização:** Esse serviço pode ser reutilizado em diferentes partes da aplicação, como ao criar um pedido ou ao atualizar o carrinho de compras, garantindo que a lógica de desconto seja consistente em todo o sistema.

Quando utilizar ?

- A lógica de negócio **não pertence diretamente** a uma única entidade ou objeto de valor.
- A lógica de negócio precisa ser **aplicada a múltiplas entidades**.
- É importante **encapsular regras de negócio complexas** fora das entidades, mantendo-as mais simples e focadas em suas responsabilidades principais.

Outros exemplos

- **PaymentProcessingService:** Lógica para processar pagamentos, onde os detalhes não fazem parte de uma entidade única.
- **ShippingCalculator:** Cálculo de frete com base em regras complexas, como localização, peso e tipo de entrega.
- **CustomerCreditEvaluator:** Avaliação de crédito do cliente com base em regras de domínio.

Conclusão



Os serviços de domínio ajudam a **organizar** e **isolar a lógica de negócio**, permitindo que o domínio seja mais flexível e fácil de evoluir conforme novos requisitos surgem.

09 Eventos de Domínio

Representam algo que aconteceu no domínio.

Eventos de Domínio



Eventos de Domínio representam algo que aconteceu no domínio do sistema e que é relevante para outros componentes ou partes da aplicação.

Em DDD, eventos de domínio são usados para notificar outros componentes sobre mudanças significativas no estado de uma entidade ou agregado, promovendo um estilo de arquitetura orientado a eventos e acoplamento fraco entre diferentes partes do sistema.

Importância dos Eventos

- **Desacoplamento:** Permitem que diferentes partes do sistema saibam sobre mudanças no domínio sem precisar de uma referência direta, facilitando uma arquitetura modular e mais fácil de manter.
- **Coesão:** Ao encapsular o processamento de um evento em classes específicas, o código das entidades e dos agregados permanece mais coeso e focado em suas responsabilidades principais.
- **Extensibilidade:** Eventos de domínio permitem adicionar novos comportamentos sem modificar o código original do domínio. Basta adicionar novos "ouvintes" (event handlers) para lidar com eventos conforme necessário.

Exemplo Prático



Vamos implementar um exemplo de evento de domínio para nossa plataforma de e-commerce. Quando um pedido (Order) é confirmado, um evento de domínio **OrderPlaced** será gerado para notificar que o pedido foi concluído.

Outros componentes, como a logística e o sistema de faturamento, podem então reagir a esse evento.

Exemplo Prático

Passo 1: Criar o Evento de Domínio OrderPlaced

O evento de domínio é uma classe que representa o fato ocorrido. Incluímos informações relevantes para que os manipuladores do evento (event handlers) possam reagir a ele.

```
public class OrderPlaced
{
    public Guid OrderId { get; }
    public DateTime PlacedAt { get; }
    public Guid CustomerId { get; }

    public OrderPlaced(Guid orderId, Guid customerId, DateTime placedAt)
    {
        OrderId = orderId;
        CustomerId = customerId;
        PlacedAt = placedAt;
    }
}
```

Exemplo Prático

Passo 2: Adicionar o Evento à Entidade Order

Vamos modificar a classe Order para que ela emita o evento OrderPlaced ao ser confirmada. Para simplificar, a lista de eventos gerados ficará na própria entidade, e podemos usar um sistema de event handlers para processá-los mais tarde.

```
public class Order
{
    public Guid Id { get; private set; }
    public DateTime OrderDate { get; private set; }
    public Guid CustomerId { get; private set; }
    private List<OrderItem> _items = new List<OrderItem>();
    public IReadOnlyCollection<OrderItem> Items => _items.AsReadOnly();
    public decimal TotalAmount => _items.Sum(item => item.TotalPrice);

    private List<object> _domainEvents = new List<object>();
    public IReadOnlyList<object> DomainEvents => _domainEvents.AsReadOnly();

    public Order(Guid customerId)
    {
        Id = Guid.NewGuid();
        OrderDate = DateTime.UtcNow;
        CustomerId = customerId;
    }

    public void AddItem(Product product, int quantity)
    {
        var orderItem = new OrderItem(product, quantity);
        _items.Add(orderItem);
    }

    public void ConfirmOrder()
    {
        // Lógica para confirmar o pedido
        // ...

        // Adicionar o evento de domínio
        var orderPlacedEvent = new OrderPlaced(Id, CustomerId, DateTime.UtcNow);
        _domainEvents.Add(orderPlacedEvent);
    }

    public void ClearEvents()
    {
        _domainEvents.Clear();
    }
}
```

Exemplo Prático

Passo 3: Manipulador de Eventos (Event Handler) `SendOrderConfirmationEmailHandler`

Agora vamos criar um manipulador de eventos para tratar o evento `OrderPlaced`. Esse manipulador será responsável, por exemplo, por enviar um e-mail de confirmação para o cliente.

```
public class SendOrderConfirmationEmailHandler
{
    private readonly IEmailService _emailService;

    public SendOrderConfirmationEmailHandler(IEmailService emailService)
    {
        _emailService = emailService;
    }

    public void Handle(OrderPlaced orderPlacedEvent)
    {
        // Simulação de envio de email
        _emailService.SendEmail(orderPlacedEvent.CustomerId,
            $"Seu pedido {orderPlacedEvent.OrderId} foi confirmado!",
            "Obrigado por sua compra!");
    }
}
```

Exemplo Prático

Passo 4: Disparar o Evento de Domínio

No fluxo de trabalho principal, ao confirmar o pedido, o evento OrderPlaced é gerado e pode ser processado por qualquer manipulador que esteja escutando esse evento.

```
public class OrderService
{
    private readonly List<object> _eventHandlers;

    public OrderService(List<object> eventHandlers)
    {
        _eventHandlers = eventHandlers;
    }

    public void PlaceOrder(Order order)
    {
        order.ConfirmOrder();

        // Processar todos os eventos de domínio que foram gerados
        foreach (var domainEvent in order.DomainEvents)
        {
            foreach (var handler in _eventHandlers.OfType<IEventHandler<OrderPlaced>>())
            {
                handler.Handle((OrderPlaced)domainEvent);
            }
        }

        // Limpar eventos processados
        order.ClearEvents();
    }
}
```

Características

- **Gerar o Evento:** Ao confirmar o pedido (ConfirmOrder), a entidade Order cria uma instância de OrderPlaced e a adiciona à lista de eventos.
- **Manipular o Evento:** No OrderService, ao confirmar o pedido, os manipuladores de evento (como SendOrderConfirmationEmailHandler) processam o evento e enviam a confirmação por email para o cliente.
- **Acoplamento Fraco:** A lógica de envio de e-mail está desacoplada da entidade Order e é tratada por SendOrderConfirmationEmailHandler. Novos comportamentos podem ser adicionados com mais manipuladores, sem alterar a entidade Order.

Quando Usar ?

- **Propagação de Mudanças:** Quando uma mudança em uma entidade deve ser conhecida por outros componentes.
- **Acoplamento Fraco:** Para desacoplar diferentes partes do sistema, mantendo responsabilidades separadas.
- **Arquitetura Orientada a Eventos:** Em sistemas distribuídos ou de alta complexidade, os eventos facilitam a coordenação de diferentes partes do sistema.

Outros Cenários

- **Inventário:** Ajustar o inventário quando uma ordem de venda é colocada.
- **Faturamento:** Gerar uma fatura quando um pedido é concluído.
- **Notificações:** Enviar notificações em tempo real quando um evento específico ocorre no sistema.

Conclusão



Os **eventos de domínio** promovem uma arquitetura extensível e organizada, ajudando a gerenciar a complexidade de sistemas grandes e orientados a mudanças.

10 Repositórios

Forma de encapsular a lógica de acesso a dados.

Repositórios



No Domain-Driven Design (DDD), o padrão **Repository** é uma **forma de encapsular a lógica de acesso a dados** para permitir que o código do domínio interaja com o armazenamento de dados de maneira mais abstrata e independente da implementação específica (por exemplo, banco de dados SQL, NoSQL, etc.).

Um repositório fornece métodos para manipular agregados, permitindo que o domínio recupere, adicione, remova e atualize dados sem precisar conhecer detalhes sobre como esses dados estão sendo persistidos.

Características

- **Abstração de Acesso a Dados:** O repositório atua como uma interface entre o domínio e o banco de dados, isolando a lógica de acesso a dados e simplificando o código do domínio.
- **Centralização das Operações de Persistência:** Toda a lógica de persistência é concentrada no repositório, facilitando a manutenção e a modificação.
- **Manipulação de Agregados:** Em DDD, repositórios geralmente operam em agregados – um conjunto de entidades que deve ser manipulado como uma unidade.

Exemplo Prático



Vamos criar um exemplo de repositório para a entidade Order, usando uma interface **IOrderRepository**. Esse repositório terá métodos para adicionar, buscar e salvar ordens, e vamos implementar essa interface para uma estrutura de dados em memória para simplificar o exemplo.

Passo 1: Criar a Interface IOrderRepository

A interface define as operações principais que o repositório expõe para o domínio. Essas operações geralmente incluem métodos como **Add**, **GetById** e **Save**.

```
public interface IOrderRepository
{
    void Add(Order order);
    Order GetById(Guid orderId);
    void Save(Order order);
}
```

Exemplo Prático

Vamos criar um exemplo de repositório para a entidade Order, usando uma interface **IOrderRepository**. Esse repositório terá métodos para adicionar, buscar e salvar ordens, e vamos implementar essa interface para uma estrutura de dados em memória para simplificar o exemplo.

Passo 1: Criar a Interface IOrderRepository

A interface define as operações principais que o repositório expõe para o domínio. Essas operações geralmente incluem métodos como **Add**, **GetById** e **Save**.

```
public interface IOrderRepository
{
    void Add(Order order);
    Order GetById(Guid orderId);
    void Save(Order order);
}
```

Exemplo Prático

Passo 2: Implementar a Classe InMemoryOrderRepository

Nesta implementação, usaremos uma lista em memória para simular o armazenamento.

Em um sistema real, essa implementação poderia usar um banco de dados relacional ou NoSQL.

```
public class InMemoryOrderRepository : IOrderRepository
{
    private readonly List<Order> _orders = new List<Order>();

    public void Add(Order order)
    {
        _orders.Add(order);
    }

    public Order GetById(Guid orderId)
    {
        return _orders.FirstOrDefault(o => o.Id == orderId);
    }

    public void Save(Order order)
    {
        var existingOrder = GetById(order.Id);
        if (existingOrder != null)
        {

            _orders.Remove(existingOrder);
            _orders.Add(order);
        }
    }
}
```

Exemplo Prático

Passo 3: Utilizar o Repositório em um Serviço de Domínio

Agora, vamos criar um serviço de domínio para interagir com o repositório, confirmando e salvando o pedido.

```
public class OrderService
{
    private readonly IOrderRepository _orderRepository;

    public OrderService(IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }

    public void PlaceOrder(Order order)
    {
        order.ConfirmOrder();
        _orderRepository.Save(order);
    }

    public Order FindOrder(Guid orderId)
    {
        return _orderRepository.GetById(orderId);
    }
}
```

Exemplo Prático

Passo 4: Exemplo de Uso

Finalmente, podemos simular o uso do repositório e do serviço em um cenário de aplicação.

```
// Criando o repositório e o serviço de pedidos
var orderRepository = new InMemoryOrderRepository();
var orderService = new OrderService(orderRepository);

// Criando um novo pedido
var order = new Order(customerId: Guid.NewGuid());
order.AddItem(new Product("Produto A", 100.0m), quantity: 2);
orderService.PlaceOrder(order);

// Recuperando o pedido pelo ID
var savedOrder = orderService.FindOrder(order.Id);
Console.WriteLine($"Pedido {savedOrder.Id} para o cliente {savedOrder.CustomerId} foi salvo com sucesso.");
```

Exemplo Prático

- **Interface `IOrderRepository`:** Define as operações básicas que o repositório deve oferecer.
- **Implementação `InMemoryOrderRepository`:** Contém a lógica para armazenar e recuperar pedidos de uma lista em memória.
- **Serviço de Domínio `OrderService`:** Usa o repositório para persistir e recuperar pedidos, mantendo o domínio desacoplado dos detalhes de persistência.

Vantagens

- **Facilita a Testabilidade:** Repositórios podem ser facilmente substituídos por implementações in-memory em testes, evitando a necessidade de dependências de banco de dados.
- **Apoia o Princípio da Inversão de Dependência:** O domínio depende de uma abstração (**IOrderRepository**) e não de uma implementação concreta, facilitando a injeção de diferentes repositórios conforme necessário.
- **Isolamento da Lógica de Persistência:** Toda a lógica de persistência é encapsulada no repositório, tornando o código do domínio mais limpo e coeso.

Quando Utilizar ?

- Quando o sistema possui lógica de domínio complexa que precisa ser independente de detalhes de infraestrutura.
- Para desacoplar a lógica de negócio da camada de persistência, facilitando a evolução de ambas de forma independente.
- Em projetos que adotam DDD ou uma arquitetura orientada a agregados.

Conclusão



O padrão **Repository** promove uma arquitetura mais limpa e modular, facilitando a manutenção e a evolução de sistemas complexos.

Obrigado

Alguma Pergunta ?

willian_brito00@hotmail.com
linkedin.com/in/willian-ferreira-brito
github.com/willian-brito

