Aluno: Thomás Ramos Oliveira

Domain-Driven Design (DDD)

O conceito de **Domain-Driven Design (DDD)**, consolidado pelo livro *Domain-Driven Design Reference*, publicado por **Eric Evans em 2015**, representa uma abordagem estratégica e aprofundada para o desenvolvimento de software que lida com domínios complexos. Diferentemente de abordagens que priorizam a tecnologia ou padrões de arquitetura isolados, o DDD coloca o **domínio do negócio no centro de todas as decisões de design**, argumentando que a compreensão profunda do negócio é a chave para sistemas de software eficazes, resilientes e adaptáveis.

A essência do DDD reside na ideia de que software complexo só pode ser realmente compreendido e modelado se os desenvolvedores colaborarem intimamente com **especialistas de domínio** – profissionais que conhecem profundamente as regras, processos e nuances do negócio. Essa colaboração contínua não visa apenas criar documentação ou diagramas, mas produzir um **modelo de domínio vivo**, que é diretamente refletido no código-fonte. Assim, o software deixa de ser apenas uma ferramenta tecnológica e se torna uma **extensão do conhecimento do negócio**, permitindo que regras, comportamentos e processos estejam codificados de maneira clara e consistente.

A Força da Linguagem Compartilhada

Um dos pilares do DDD é a **Linguagem Ubíqua (Ubiquitous Language)**, que representa um vocabulário compartilhado entre todos os membros da equipe, sejam eles técnicos ou de negócios. Essa linguagem deve permear **todas as formas de comunicação**: reuniões, documentação, diagramas, testes e até mesmo nomes de classes e métodos no código. Por exemplo, se no contexto do negócio uma transação é chamada de "devolução", o software deve refletir exatamente esse termo, como em `DevolucaoTransaction`. Isso **elimina ambiguidades**, evita mal-entendidos e cria uma coerência direta entre os conceitos do negócio e a implementação técnica.

Outro conceito central é o **Contexto Delimitado (Bounded Context)**. Em sistemas grandes ou multifuncionais, a mesma palavra ou conceito pode ter significados distintos dependendo da área de negócio. Por exemplo, um "Cliente" em um sistema de faturamento pode ter atributos relacionados a pagamentos e crédito, enquanto no sistema de atendimento pode incluir apenas informações de contato. O **Contexto**

Delimitado define claramente os limites em que um modelo é válido, garantindo que equipes possam trabalhar **independentemente e de forma consistente** em seus próprios modelos sem que conceitos se confundam.

A **interação entre diferentes Contextos Delimitados** é orquestrada por meio dos **Mapas de Contexto (Context Maps)**, que documentam relações, integrações e dependências entre modelos distintos. Esse mapeamento permite identificar quando conceitos precisam ser compartilhados, traduzidos ou adaptados, evitando que um modelo "vaze" para o outro e que regras de negócio sejam violadas inadvertidamente.

Estrutura e Praticidade: Blocos Táticos do DDD

Eric Evans detalha no documento os **blocos táticos**, que são padrões e estruturas fundamentais para implementar o DDD na prática. Esses blocos fornecem um **guia concreto** para transformar o conhecimento do domínio em código limpo, organizado e expressivo:

1. Entidades:
São objetos de domínio que possuem **identidade própria e ciclo de vida único**. Uma entidade não é definida apenas por seus atributos, mas por sua **continuidade no tempo e identidade única**, mesmo que suas características mudem. Exemplos clássicos incluem `Produto`, `Funcionario` ou `Pedido`. O foco principal é que a entidade representa **um conceito único do domínio**, persistente e reconhecível.

2. Objetos de Valor (Value Objects):
São objetos que **não possuem identidade própria** e são definidos exclusivamente por seus atributos. São geralmente **imutáveis**, o que significa que, em vez de alterar um objeto de valor existente, cria-se um novo. Exemplos incluem `Endereco`, `IntervaloDeTempo` ou `Dinheiro`. Esses objetos ajudam a expressar conceitos do domínio de forma precisa, garantindo consistência e segurança nos dados.

3. Agregados (Aggregates):
Agrupam **Entidades e Objetos de Valor** que estão logicamente conectados e devem ser tratados como uma unidade para fins de consistência e integridade. O **Aggregate Root** (Raiz do Agregado) é a única entidade acessível externamente, assegurando que todas as operações sobre o agregado respeitem as regras do domínio. Por exemplo, em um agregado `Pedido`, o `Pedido` seria a raiz, e os itens do pedido seriam entidades internas manipuladas apenas através do pedido.

4. Repositórios (Repositories):
Fornecem uma **abstração para persistência e recuperação** de agregados. Repositórios permitem que a lógica de negócio permaneça isolada da complexidade da camada de armazenamento (banco de dados, APIs externas etc.), oferecendo uma interface consistente para gerenciar o ciclo de vida dos agregados. Por exemplo, `PedidoRepository` pode fornecer métodos como `salvar(pedido)` ou `buscarPorId(id)`.

5. Serviços de Domínio (Domain Services):
São operações que encapsulam **lógica de negócio que não pertence naturalmente a nenhuma Entidade ou Objeto de Valor**. São tipicamente **sem estado** e atuam como orquestradores que coordenam múltiplos objetos de domínio. Um exemplo seria `CalculadoraDeDesconto`, que aplica regras complexas de desconto envolvendo múltiplos objetos `Pedido` e `Cliente`.

A Filosofia Além do Código

O DDD não se limita a aplicar padrões; ele representa **uma mentalidade e filosofia de desenvolvimento**, enfatizando o entendimento profundo do domínio e a adaptabilidade do software. Alguns princípios-chave incluem:

- **Design Flexível:** O código deve ser estruturado para acomodar mudanças no conhecimento do domínio sem grandes refatorações.
- **Refatoração Contínua:** À medida que a equipe aprende mais sobre o domínio, o modelo e o código devem evoluir.
- **Interfaces que Revelam Intenção:** Funções e classes devem ser claras e legíveis, refletindo exatamente o que o domínio espera.
- **Testes Automatizados:** Não apenas verificam o comportamento do software, mas **documentam as regras do domínio** de forma executável.

Para projetos de grande escala, o DDD propõe **estratégias arquiteturais**, como a **Arquitetura em Camadas**, que separa o domínio das preocupações de infraestrutura, apresentação e integração externa. Isso garante que a **lógica de negócio permaneça pura e independente de detalhes técnicos**, permitindo que o software se mantenha adaptável, resiliente e fácil de evoluir.

O resultado final é um sistema que **vai além de simplesmente funcionar**. Ele se torna um **modelo vivo e preciso do negócio**, capaz de evoluir conforme as necessidades mudam, refletindo de maneira confiável os conceitos, regras e processos do domínio central