DOMAIN-DRIVEN DESIGN

Domain-Driven Design - Conceitos básicos

BY BRUNO BRITO ON 14 DE MAIO DE 2019

READ IN 8 MIN

O Domain-Driven Design, ou DDD, foi criado para ajudar equipes a ter mais sucesso no desenvolvimento de software com alta qualidade. Quando implementado corretamente, o DDD entrega um design que traduz exatamente como o dominio funciona.

É inegável que times de desenvolvimento, comprometidos, se esforçam para produzir um software com qualidade.

Os princípios ágeis e as práticas de Scrum e Xp ensinaram equipes a entregar qualidade usando testes. Evitando entregas com grande quantidade de bugs. No entanto, mesmo que fosse possível entregar um software completamente sem bugs, não significa, necessariamente que existe um design de software bem projetado.

> Todo software possui um arquitetura. Boa ou ruim, ela está presente. Planejada ou não, ela se forma, consiste e persiste. Ela pode estar “escondida”, mal-entendida, corrompida, vilipendiada, ignorada ou, até mesmo, esquecida. Mas está ali.
> Elemar Jr

Entregar um software com poucos bugs é bom.

Ainda assim, entregar um software em que o modelo de domínio foi cuidadosamente elaborado. O design traduz a complexidade das regras de negócio e processos da organização. Resulta em benefícios a longo prazo. Agiliza os processos da empresa. Facilita a implementação de novas features. Aumenta a vida útil do software.

O que é DDD ?

O Domain Driven Design combina práticas de design e desenvolvimento. Oferece ferramentas de modelagem estratégica e tática para entregar um software de alta qualidade. O objetivo é acelerar o desenvolvimento de software que lidam com complexos processos de negócio.

Em seus princípios, DDD é sobre discussão, escuta e compreensão. Todo um esforço para centralizar o conhecimento.

O que não é DDD ?

O DDD não é uma tecnologia ou uma metodologia. Pode ser utilizado independente da linguagem. Não importa se é C# ou Java. Se é MVC ou Windows Forms.

Não é arquitetura em camadas e não impõe processos rígidos ao time.

Porque utilizar DDD

DDD é uma jornada, não o destino. A busca pela harmonia do domínio nunca termina. O domínio muda à medida que o negócio muda. Mantém o software alinhado com o negócio. Faz isso estratégica, tatica e filosoficamente.

Praticar DDD, significa aproximar os especialistas do domínio (Domain experts) do time de desenvolvimento. Criando uma linguagem única. Uma comunicação sem ruídos, limpa.

Domain Expert

É aquele que entende do negócio. Apoia o time de desenvolvimento na modelagem do domínio.

Ao colocar **domain experts** e desenvolvedores em sintonia através da linguagem ubíqua produz software que faz sentido para o negócio. O código expressa o negócio. E se faz sentido para o negócio é porque o caminho está correto.

Para criar um bom software, é necessário saber sobre o negócio. Não é razoável criar um software bancário, complexo, sem nunca ter sido bancário.

Quem sabe sobre banco? O arquiteto de software? O desenvolvedor? Eles apenas usam o banco para manter seu dinheiro seguro. Quem sabe sobre banco são os banqueiros. São eles os **domain experts**. Possuem condições de guiar a equipe para desenvolver um bom software.

DDD irá entregar um código que fala sobre o negócio. Faz sentido do ponto de vista de negócio. Se os domains experts fossem os devs, seria o código que eles fariam.

Uma jornada

Ninguém conhece todos os pormenores do negócio. O desenvolvimento com DDD é uma jornada de descoberta, todos aprendem. Os devs com os domains experts e estes com os devs.

Durante o desenho dos bounded context, ou conforme a progressão do projeto haverá situações não prevista pelos Domain Expert.

Pilares

Linguagem ubíqua e **bounded contexts** são os pilares do DDD. Andam lado a lado. Através da **linguagem ubíqua** é possivel identificar e delimitar os **bounded context**.

Linguagem ubíqua

A linguagem ubíqua não é a linguagem do negócio, não é um pattern. É a linguagem utilizada pela equipe do projeto.

Desenvolvedores pensam em classes, métodos, componentes. Padrões GOF, SOLID. Tendem a achar correspondência entre um conceito da vida real com programação. Entender quais objetos criar. Qual relacionamento entre eles. Pensam em herança, polimorfismo, OOP.

Os domains experts não entendem esses termos. Seu repertório exclui idéias sobre componentes, frameworks e persistência. Eles são especialistas do negócio. Conhecem e sabem como a empresa deve funcionar.

A Linguagem Ubíqua é uma linguagem de equipe. Compartilhada por todos na equipe. Não importa papel. Se está no time, deve utilizar a Linguagem Ubíqua.

Se o negócio está em constante evolução a linguagem ubíqua também. Não é estática. Evolui de acordo com a evolução do software.

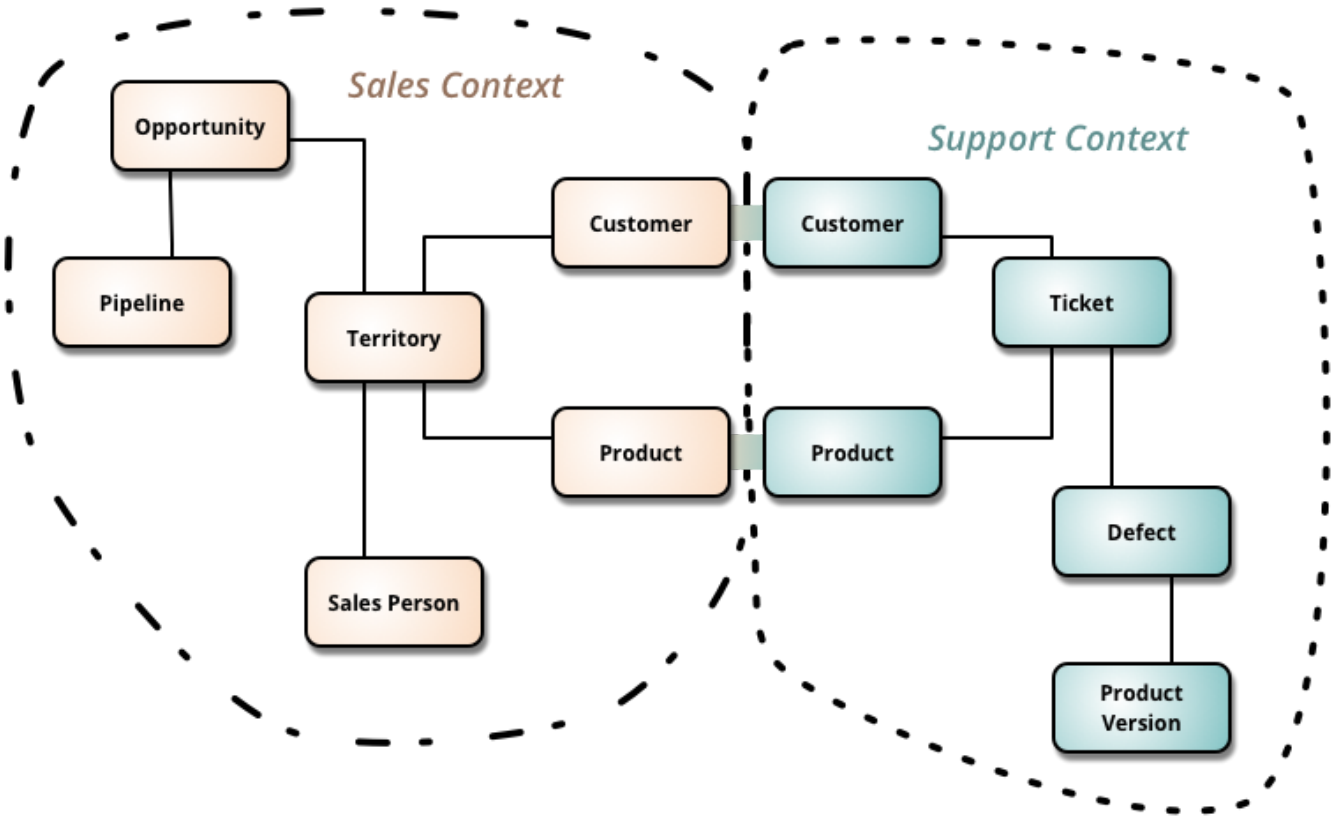
É comum haver desacordos entre um termo e outro. Durante o desenvolvimento de um novo software os conhecimentos dos domains experts serão explorados a fundo. E com isso alguns conceitos podem pode gerar desacordos entre os domains experts durante estas explorações.

Bounded context

Bounded context é um delimitador do domínio. Dentro do limite, todos os termos e frases da Linguagem Ubíqua têm um significado específico. E o modelo reflete a Linguagem Ubíqua.

> Bounded Context is a central pattern in Domain-Driven Design. It is the focus of DDD's strategic design section which is all about dealing with large models and teams. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.
>
> - Martin Fowler

Veja a imagem:



Repare que **Customer** e **Product** aparecem duas vezes. Porém em bounded contexts diferentes. Significa que possuem papéis e responsabilidades distintas dentro de cada contexto.

Product no contexto de *Suport* pode ser uma entidade com apenas um Nome. Assim a equipe de suporte sabe qual produto está lidando. Já em *Sales*. O comercial, por sua vez, precisa de mais informações. Além do nome do Product, como também, o Preço.

Perceba que para cada contexto Product possui caracteristicas distintas e destilando seus detalhes, também terá comportamentos únicos.

Identificar bounded context requer maturidade do negócio. Por isso a necessidade de trabalhar em conjunto com os domains experts. Com o tempo e amadurecimento da linguagem ubíqua o time cria mais condições de explorar os limites do contexto.

Modelagem Estratégica - Identificar e Mapear os Bounded Contexts

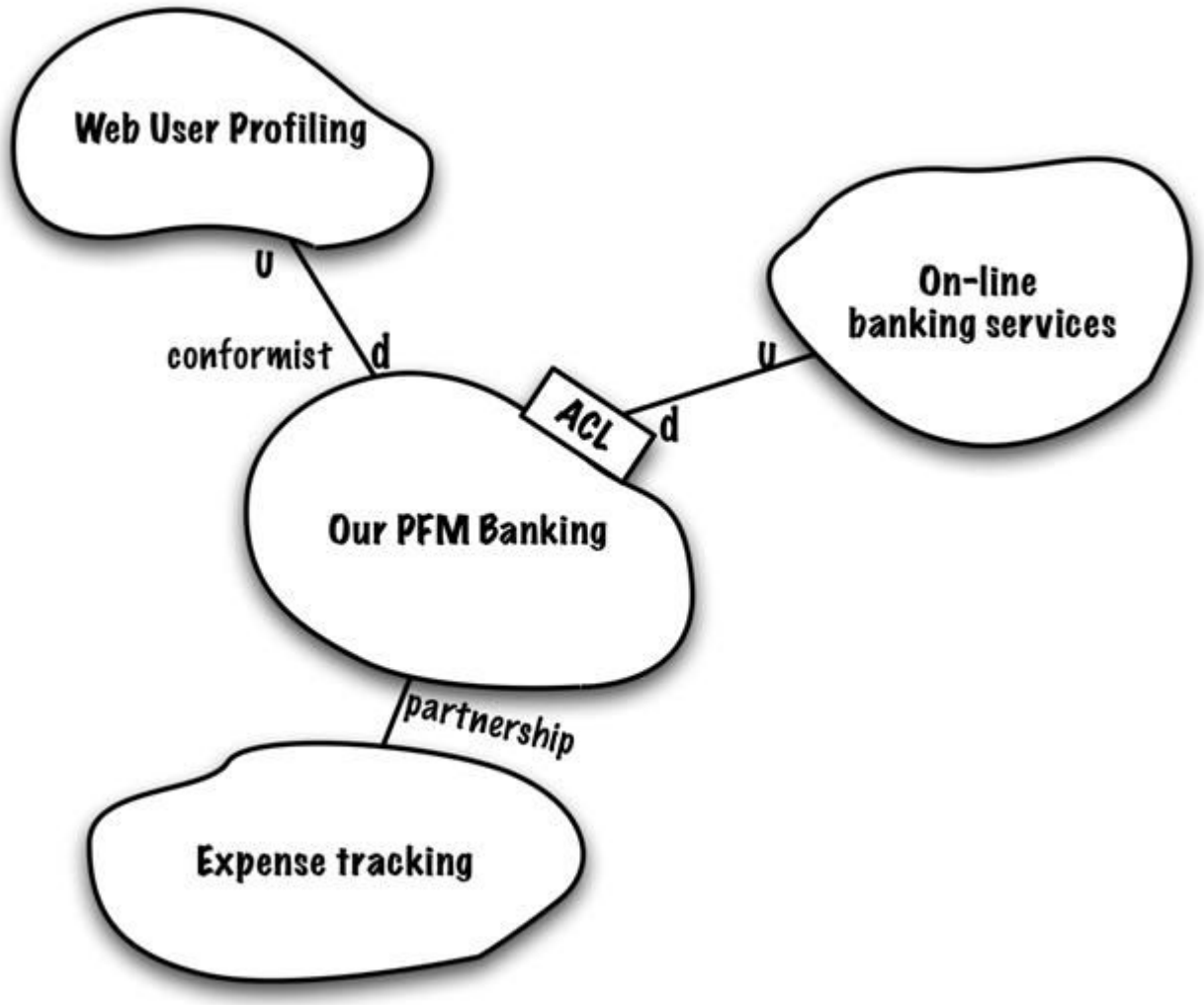
Um **Bounded context** é um limite conceitual. A linguagem ubíqua guia o time para entender o negócio e assim ajudar a identificar os contextos.

Utilize **Context Map** para entender como estes se relacionam. Desenhar é a melhor forma de entender o terreno.

Este desenho é simples. Serve como mapa para o time. É desenhado com o objetivo de dar perspectiva da solução.

Um **Context Map** é a visão global do software. Cada **Bounded Context** dentro do *Mapa* revela como se comunica com os demais. Deve desenhar o presente, não o futuro. O mapa é atualizado ao passo que o projeto progride.

Veja o exemplo de como é simples desenhar um **Context Map**



Pode ser desenhado a mão ou num quadro branco. É importante que seja simples, informal. Seu desenho é de alto nível. O objetivo é ter uma visão geral e não ser um documento oficial. É natural que com o tempo perca a sincronia com o projeto

Na figura também é possível ver algumas marcações que explicam o relacionamento entre os **bounded context**.

Mapeando o relacionamento entre Bounded Contexts

Ao fazer o **Context Map** é importante definir qual a relação entre os **bounded context**. Os tipos de relacionamentos implicam diretamente na estratégia de como os times do projeto irão trabalhar. Qual a dependência entre eles. Impacta na arquitetura. Na estratégia de comunicação. O tipo de dependência entre os bounded context gera consequências direta na organização das equipes.

- **Shared Kernel** - É quando vários **bounded context** compartilham um mesmo domínio. Alterar significa que todas as equipes serão afetadas.
- **Customer-Supplier Development** - Quando existe um relacionamento upstream (Supplier) e downstream (Customer), significa que a equipe upstream pode ter êxito interdependente da equipe downstream. Modificações no contexto upstream impactam a downstream.
- **Conformist** - O **bounded context** downstream está conformado que o modelo *upstream* não atende suas necessidades e as modificações impactam diretamente seu contexto. A equipe então se conforma com esse relacionamento e precisa estar atento a suas mudanças.
- **Partner** - É quando duas equipes possuem uma dependência mútua. Precisam, portanto, trabalhar juntos. (Foi sugerido por **Eric Evans** durante a apresentação do QCon London 2009.)
- **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.

Evite Big ball of mud

Big ball of mud são modelos onde os limites são inconsistente. Grande e misturado. Códigos macarrônicos e desleixados. É aquele sistema que ninguém quer mexer. Parece aquele cobertor curto, ao cobrir a cabeça, descobre os pés.

Ao modelar seus **bounded contexts** tenha cuidado. Evite criar um *Big Ball of Mud*. Para isso evite modelos abrangentes. Unicos.

Cuidado com modelos com conceitos e nomes que tenham significado global. Primeiro porque será quase impossível estabelecer um acordo entre os diferentes domains experts.

Saiba que conceitos possuem significados diferente entre os **bounded contexts**. Podem estar escondidos num primeiro momento. Mas com o passar do tempo e aprendizado do time, eles aparecem.

Resumo

O objetivo é mostrar que DDD vai muito além do código. DDD não é uma arquitetura em camadas, não é sobre código. É sobre negócio.

É necessário sair do Status Quo. Entender sobre o negócio. É uma jornada de aprendizado, diário.

Em algum momento da tua carreira você já fez ou consegue responder alguma dessas perguntas?

- Qual seu papel dentro da organização?
- Por que sua empresa existe?
- Quais são as estratégias para os próximos anos na sua organização?
- Em qual mercado ela está inserido?
- Qual o perfil de clientes que sua empresa atende?
- No último ano sua empresa teve lucro operacional?
- Qual o EBITDA dela?

- Qual a classificação do teu projeto atual? (Redução de custo, Expansão da capacidade de produção, expansão para novos mercados)

São difíceis de responder. E é normal. A graduação de tecnologia não há matérias sobre análise econômica ou finanças corporativas.

E perceba que se a organização não for de TI, nenhuma dessas respostas está relacionado com tecnologia.

Responder tais questões, certamente vão aprofundar teu conhecimento da empresa. E já deu para perceber que entender o negócio ajuda na modelagem dos **bounded context**. A consequência natural é a entrega de softwares melhores.

Espero que esse post tenha incomodado vocês de uma maneira boa. Ajude a construir uma outra visão. Entender que software é, na maioria das vezes, o meio. Não o fim. E que para entregar um software de qualidade, precisa além de capacidade técnica, esforço para aprender sobre o domínio que está lidando

Referencias

- Bounded Contexts - Martin Fowler
- Domain-Driven Design - Martin Fowler
- Implementing Domain-Driven Design - Vaughn Vernon
- VOCÊ NÃO É (OU, PODE NÃO SER) ARQUITETO DE SOFTWARE - Elemar Jr
- DDD Não é arquitetura em camadas - Eduardo Pires
- DDD Comunity

WRITTEN BY

BRUNO BRITO

Primeiramento sou pai! #dev fullStack e MCSA Web Application.

BRAZIL WEBSITE FACEBOOK