**INTRODUÇÃO AOS CONCEITOS DE DOMAIN DRIVEN DESIGN**

O desenvolvimento de *software* é aplicado com mais frequência na automação de processos existentes no mundo real ou no fornecimento de soluções para problemas reais de negócios; os processos de negócios são problemas automatizados ou reais. Devemos entender desde o início que o *software* é originado e profundamente relacionado ao domínio. Software é composto de código e podemos gastar muito tempo com o código e vê-lo simplesmente como objetos e métodos, no entanto, não se trata apenas disso.

(AVRAM, 2006) nos apresenta uma metáfora interessante. Vamos considerar o processo de fabricação de carros, os trabalhadores envolvidos na fabricação de automóveis podem se especializar na produção de peças do carro, mas ao fabricar tais peças, eles geralmente têm uma visão limitada do processo de fabricação como um todo. Eles passam a ver apenas como uma coleção de partes que devem trabalhar juntas para funcionar, mas um carro é muito mais do que isso.

Um bom carro começa com uma visão, com especificações bem definidas e isso continua com muitos *designs*, muitas vezes por meses e até anos, mudanças e refinamentos até chegar a um modelo final, o qual reflete a visão original. Esse processo todo não fica apenas no papel, isso inclui em desenvolver modelos do carro, testar em determinadas condições e verificar se ele funciona e assim o *design* é modificado com base nos resultados. Ao final desse processo, o carro é enviado para produção e as peças são criadas e posteriormente montadas juntas.

O desenvolvimento de *software* é semelhante, não podemos simplesmente sentar e escrever código, isso pode até funcionar em casos triviais, mas não é dessa forma que se criar sistemascomplexos. Para criar um bom *software*, você precisa saber o que ele é. Não é possível desenvolver um sistema bancário, a menos que você tenha uma boa compreensão de como funciona um banco, é preciso entender o domínio do sistema bancário.

O sistema bancário é bem compreendido pelas pessoas de dentro, por seus especialistas. Eles conhecem todos os detalhes, todas as capturas, todos os possíveis problemas, todas as regras. É aqui que devemos sempre começar: o domínio.

Quando começamos um projeto de *software*, devemos nos concentrar no domínio em que ele está operando. Todo o propósito é aprimorar um domínio específico. Para poder fazer isso, o *software* precisa se encaixar harmoniosamente com o domínio para o qual foi criado. Caso contrário, irá introduzir tensão no domínio, provocando mau funcionamento, danos e até mesmo caos.

Como podemos fazer o *software* se encaixar harmoniosamente com o domínio? A melhor maneira de fazer isso é transforma-lo em um reflexo do domínio. O *software* precisa incorporar os conceitos e elementos centrais do domínio e realizar precisamente as relações entre eles e tem que modelar o domínio.

**O QUE É DOMAIN DRIVEN DESIGN**

Dentro do desenvolvimento de *software*, *Domain Driven Design* – DDD é uma abordagem que reúne um conjunto conceitos, princípios e técnicas voltados ao domínio. A ideia básica está centrada no conhecimento do problema para o qual o *software* é proposto. Na prática, esse conjunto busca auxiliar o desenvolvedor na tarefa de construir aplicações que reflitam um entendimento do negócio. É a construção a partir da modelagem do domínio real.

**O QUE NÃO É DOMAIN DRIVEN DESIGN**

Não é uma tecnologia ou metodologia de desenvolvimento de *software*. Pode ser utilizado independente da linguagem de programação. Também não é uma arquitetura em camadas, seu principal objetivo é auxiliar a implementação de regras complexas ou processos de negócios. Como o próprio nome já diz é sobre *design* guiado pelo domínio (complexidade do negócio).

**EXISTE UM MODELO PARA IMPLEMENTAR DOMAIN DRIVEN DESIGN?**

A resposta é não! Não existe um modelo, passo-a-passo ou “receita de bolo” de como implementar DDD, mas podemos definir um resumo baseado na literatura e em conhecimentos disponíveis na internet.

**Domain Expert**

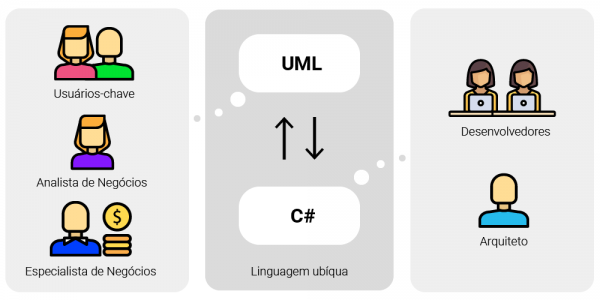
Sem entender sobre o negócio e suas regras complexas não é possível implementar o DDD. E o que se deve fazer então?

Basicamente é colocar o time de desenvolvimento para trabalhar em sintonia com os *Domain Experts*, eles vão guiar os desenvolvedores, tirando dúvidas, definindo regras e nomeando termos utilizados. Para que o resultado seja bom, é preciso saber do negócio, como visto anteriormente, não se faz um sistema bancário, sem nunca ter sido bancário.

**Linguagem Ubíqua**

Imaginemos agora que o time de desenvolvedores e os *Domain Experts* estão trabalhando juntos, quais dificuldades os times podem enfrentar? Sabemos que os desenvolvedores têm em suas mentes classes, métodos, algoritmos, *patterns*, POO, polimorfismo e muitos outros termos técnicos e os *Domain Experts* (pegamos o monitoramento de trafego aéreo como exemplo) sabem sobre voos, rotas, altitudes, latitudes e longitudes. Como eles vão se comunicar de forma clara e concisa, para que todos tenham o real entendimento sobre o assunto?

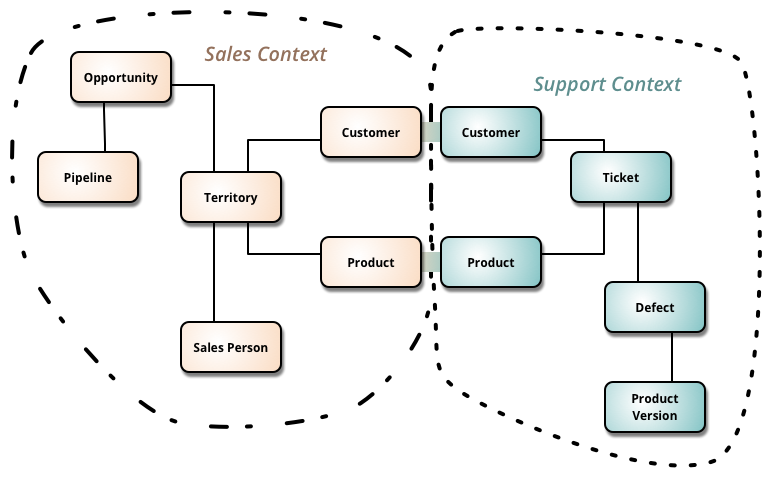
A resposta é: A linguagem ubíqua, ou seja, uma linguagem compartilhada e desenvolvida pelas duas equipes e todos devem fazer uso dela para expressar corretamente os processos do negócio onde termos estritamente técnicos são substituídos por termos que o todos compreendam. A figura 1 ilustra esse processo.



**Figura 1 - Linguagem Ubíqua. Fonte (BONIZI, 2017)**

**Bounded Context**

Os *Bounded Contexts* buscam delimitar um domínio complexo em contextos menores e baseados nas regras do negócio em questão, ou seja, você deve delimitar as intenções das entidades com base no contexto que ela pertence. Veja a figura 2, ela ilustra contextos delimitados onde *Customer* e *Product* aparecem mais de uma vez, no entanto, em contextos diferentes e com papeis e comportamentos diferentes.



**Figura 2- Bounded Context. Fonte (BRITO, 2019)**

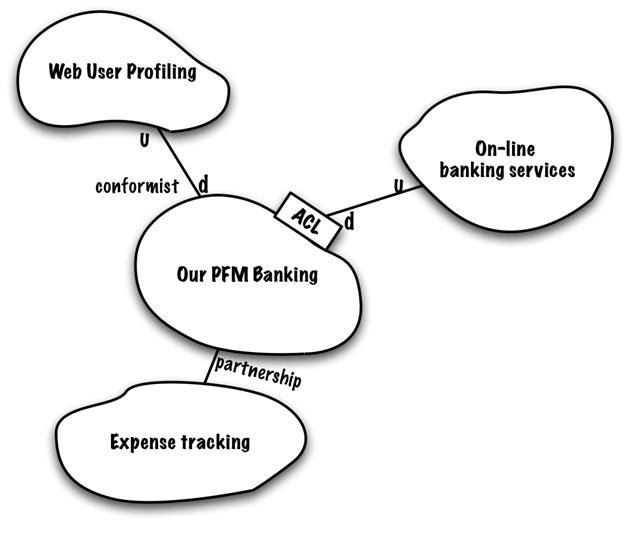
Analisando a figura 2 podemos compreender que no contexto de suporte, o cliente pode conter apenas o nome e telefone e assim a equipe de suporte saberá com quem está auxiliando e uma forma de contato. Já em vendas, existe a necessidade de mais informações como por exemplo endereço de cobrança e/ou entrega. A identificação desses detalhes requer uma boa harmonia entre os desenvolvedores e os *Domain Experts*.

Mas representar a mesma entidade em diversos contextos não seria duplicar código? A resposta é não, pois existe uma segregação de comportamentos da entidade conforme o contexto em que ela está definida. Isso não importa se os dados serão persistidos em uma mesma tabela ou em tabelas diferentes.

**Context Map**

Um *Context Map* é uma visão geral do software e serve para entender como é o relacionamento dos contextos. Os B*ouded Context* revelam como se comunica com os demais, a linguagem ubíqua guia o time para entender o negócio.

Desenhar é sempre a melhor forma de expressar e esse desenho é simples, o qual auxilia o time. A figura 3 ilustra como é simples desenhar um *Context Map.*

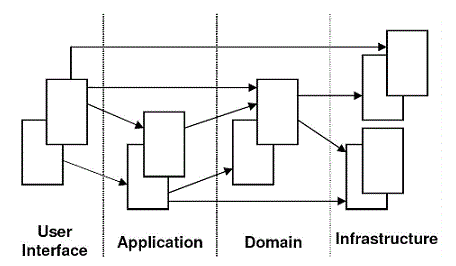


**Figura 3 - Context Map. Fonte (BRITO, 2019)**

**Layered Architecture**

A figura 4 ilustra uma sugestão para a arquitetura em camadas que pode ser utilizada em um modelo de domínio, onde:

* ***User Interface:*** Apresenta a informação ao usuário e interpreta os seus comandos.
* ***Application:*** É a camada que coordena atividade da aplicação. Não contém lógica de negócio nela, apenas mantem o fluxo das atividades.
* ***Domain:*** Contém toda informação sobre o domínio e é considerado o coração do projeto. Aqui é mapeado os objetos e comportamentos do mundo real para o software.
* ***Infraestructure*:** Atua como uma *library* onde oferece suporte para as outras camadas e realiza a persistência dos objetos de negócio.



**Figura 4- Arquitetura de camadas. Fonte (MACORATTI)**

**Modelagem Estratégica**

Segundo (PIRES, 2016), quando se trata de DDD a modelagem fica por conta do *Domain Model Pattern* que é uma abordagem de como escrever as classes que vão mapear os modelos do mundo real e seus comportamentos. O mesmo deve ser isolado dos detalhes da arquitetura como por exemplo a persistência de dados.

Ainda segundo (PIRES, 2016), a modelagem estratégica ou modelagem tática inclui os seguintes componentes:

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

**Resumindo**

Podemos entender como o principal objetivo desse artigo é demonstrar que o DDD vai muito além de apenas código, não é uma arquitetura em camadas e sim sobre negócio! É preciso sair da zona de conforto, aprender e entender sobre o negócio. Isso torna a equipe mais colaborativa e focada no que é realmente importante.

Para iniciar seu projeto em DDD é necessário ter um bom conhecimento teórico e prático (eu estou correndo atrás do meu!). E por isso eu recomendo que faça o curso “Modelando domínios ricos” do André Baltieri (<https://balta.io/>) e não poderia deixar de mencionar também os artigos e cursos do Eduardo Pires (<https://www.eduardopires.net.br/>). Recomendo também a leitura dos livros “Domain-Driven Design Quickly” by Abel Avram, “Domain-Driven Design: Tackling Complexity in the Heart of Software” by Erick Evans e “Implementing Domain-Driven Design" by Vaughn Vernon

**REFERÊNCIAS**

AVRAM, Abel. *Domain-Driven Design Quickly.* 1.Ed. - C4Media, Publisher of InfoQ.com Enterprise Software Development Community, 2006

PIRES, Eduardo. DDD não é arquitetura em camadas. 2016. Disponível em: <https://www.eduardopires.net.br/2016/08/ddd-nao-e-arquitetura-em-camadas/> Acesso em: 02/07/2019

MACORATTI, José Carlos. Resumo sobre Domain-Driven Design de Eric Evans. Disponível em: <http://www.macoratti.net/11/05/ddd_liv1.htm> Acesso em: 02/07/2019

BRITO, Bruno. Domain-Driven Design – Conceitos Básicos. 2019. Disponível em: <https://www.brunobrito.net.br/domain-driven-design/> Acesso em: 02/07/2019

BONIZI, Graziella. Desmistificando o DDD. 2017. Disponível em: <https://www.lambda3.com.br/2017/10/desmistificando-o-ddd/> Acesso em: 02/07/2019