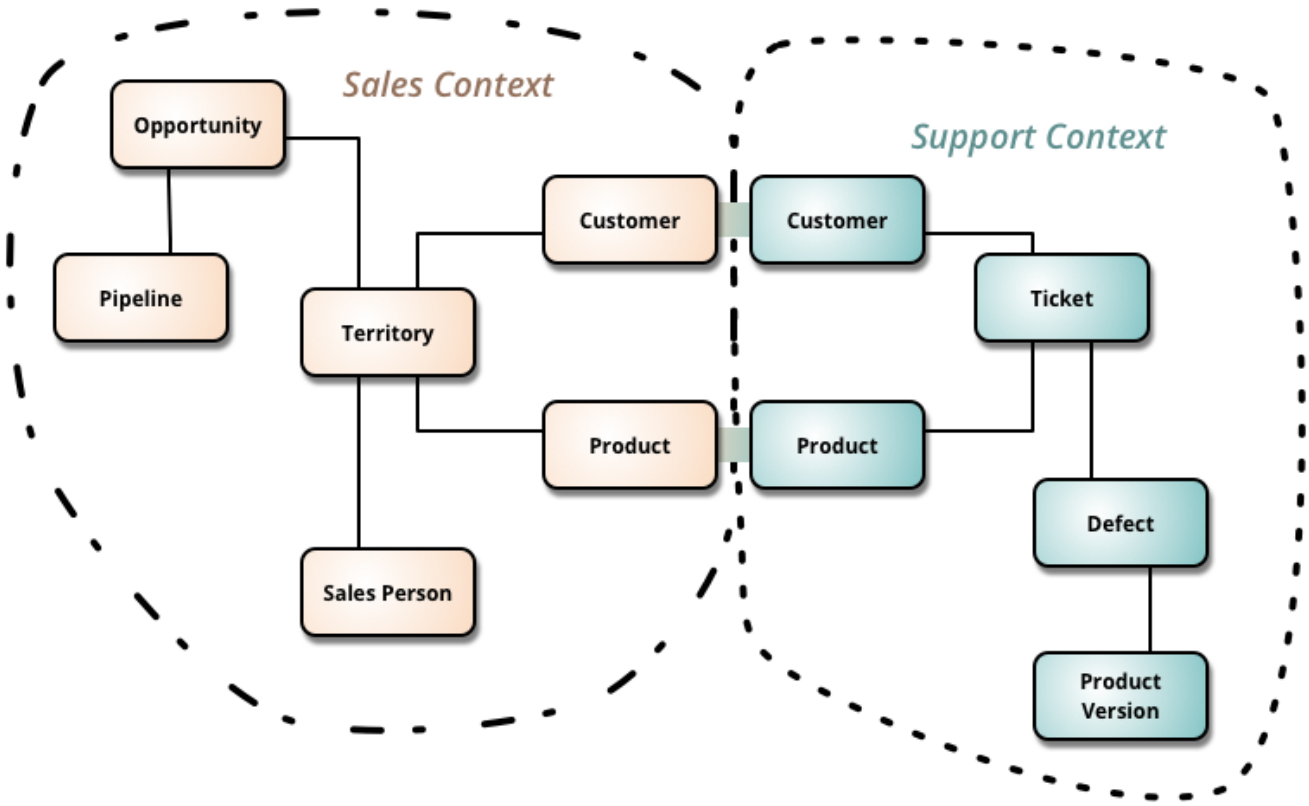
DDD – Bounded Context

Bounded Context (contexto delimitado) é uma prática no DDD que visa facilitar e dar coerência ao desenvolvimento em cima de um modelo extenso no qual entidades possam ter significados e responsabiliaddes diferentes dependendo do contexto no qual elas se aplicam.



Na imagem acima (roubada desse post do Martin Fowler) há dois contextos dentro de um mesmo sistema:

- Contexto de Vendas (Sales Context)
- Contexto de Suporte (Support Context)

Cada um deles tem seus próprios módulos (módulos no DDD podem ser simplesmente namespaces ou pastas) e entidades. Portanto observe que Produto e Cliente são entidades presentes em ambos contextos! Nesse caso elas representam o mesmo conceito (não necessariamente precisam representar o mesmo conceito), porém sua modelagem pode ser completamente diferente.

Um **Produto** no contexto de suporte pode ser uma entidade com apenas um `Id` e `Nome` para que a equipe de **suporte** saiba com qual produto está

lidando. Já no contexto de vendas, a equipe do **comercial** necessita de mais detalhes do Produto no qual está tentando vender, como por exemplo, o `Preço`. Imagine qual o sentido de no contexto do suporte haver um preço no Produto: eles iriam realmente precisar daquela informação para fazer o atendimento de um defeito no produto? Apesar do Produto nesse caso ter entidades diferentes em cada contexto, nada impede que no banco de dados as informações sejam armazenadas na mesma tabela, mas isso é transparente e não tem importância para a camada de domínio.

É comum não iniciamos um projeto já pensando e definido Bounded Contexts, talvez isso nem seja o ideal, pois identificar os contextos de um modelo requer uma certa maturidade do negócio, o que muitas vezes logo no início do projeto não temos. Normalmente chegamos a conclusão de usar contextos quando surge a necessidade de utilizá-los. Ou seja, quando o domínio começa a ficar confuso. Quando aplicar a linguagem ubíqua começa a se tornar um problema.

Em equipes de desenvolvimento grandes ou distantes, como no caso de mais de um fornecedor trabalhando na mesma aplicação para um determinado cliente, a separação de contextos traz o benefício de facilitar o isolamento do desenvolvimento. Isso pode evitar conflitos de merge e principalmente evitar que bugs surjam em funcionalidades existentes. Voltando ao exemplo anterior, vamos supor que não houvessem contextos separados para o Produto um desenvolvedor que está implementando a parte de **vendas** do sistema adicionou a propriedade `Preço` na entidade, porém cometeu algum erro qualquer no mapeamento pro banco de dados (que não foi pego por nenhum teste unitário) e que essa pessoa fez o commit da funcionalidade. Isso iria desencadear erros para as demais partes do sistema que já estivessem utilizando a entidade de Produto, como por exemplo, a parte de **suporte** no qual nunca necessitou do tal campo de `Preço` e que agora foi brindada com um bug completamente inesperado.

Mapa de contexto

Quando se opta por utilizar Bounded Context é importante que seja criado um mapa de contexto para expor claramente as fronteiras e limites de cada Bounded Context. O mapa de contexto nada mais é do que um documento, uma imagem, um rascunho anexado na parede, qualquer coisa que facilite o entendimento dos contextos da aplicação. A imagem do início desse post é um mapa de contexto!

Shared Kernel

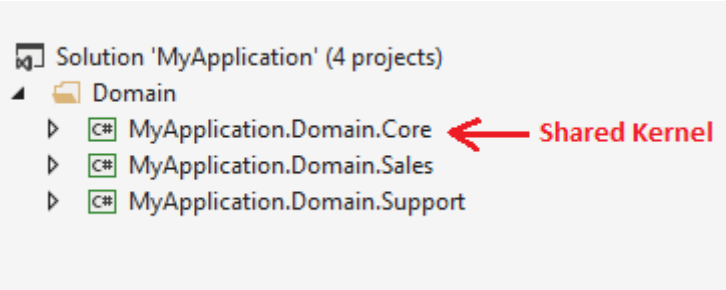
A separação de contexto também traz algumas dificuldades, como: evitar a replicação de código. No exemplo do Produto, o Id e Nome provavelmente seriam necessário nos dois contextos (Suporte e Vendas). Isso nem sempre é um problema, mas pode haver casos em que surja a necessidade de aplicar um Shared Kernel (núcleo compartilhado) para concentrar essas entidades “Core” do sistema. O Shared Kernel pode ser utilizado para interligar contextos ou simplesmente para reaproveitamento de entidades, pois todos os contextos podem ter referência para o Shared Kernel.

É importante salientar que o acesso de um contexto à outro é algo a ser evitado, mas há casos em que é necessário e faz sentido para o domínio essa interligação direta (sem utilizar o Shared Kernel). Se isso acontecer, essa ligação deve estar explícita no mapa de contexto, deixando claro o porquê da ligação.

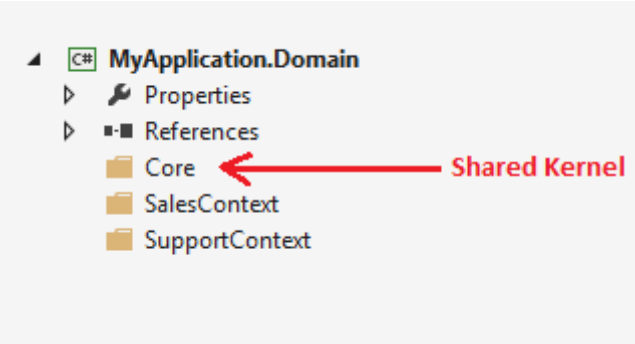
> **Exemplo:** em uma empresa de hardware fictícia, quando concluída a ***Venda*** de um computador (Produto) é necessário enviar um ticket para o setor ***Operacional*** para que eles disponibilizem esse computador com as exatas especificações que o cliente solicitou. Nesse caso a entidade Produto (do Shared Kernel) faria uma *“travessia”* do contexto de **Vendas** para o contexto **Operacional**. Essa comunicação poderia ser feita de forma síncrona (através da execução direta de código ou chamada de API) ou de forma assíncrona através de eventos, embora para soluções distribuídas apensar de ainda fazer sentido essa abordagem precise de uma análise mais profunda e leves ajustes (talvez eu aborde isso em um futuro post).

Como implementar Bounded Context

Não há uma única fórmula de como implementar o Bounded Context, pois isso pode variar de linguagem para linguagem. É necessário entender o conceito para depois decidir como melhor aplicar em seu projeto. Em .NET eu poderia organizar os contextos em projetos separados:



Ou até mesmo em pastas dentro de um único projeto:



Como disse, a implementação é a gosto. O importante é entender bem o conceito por detrás da técnica para discutir a necessidade e pontuar os benefícios que isso traria para o seu projeto.