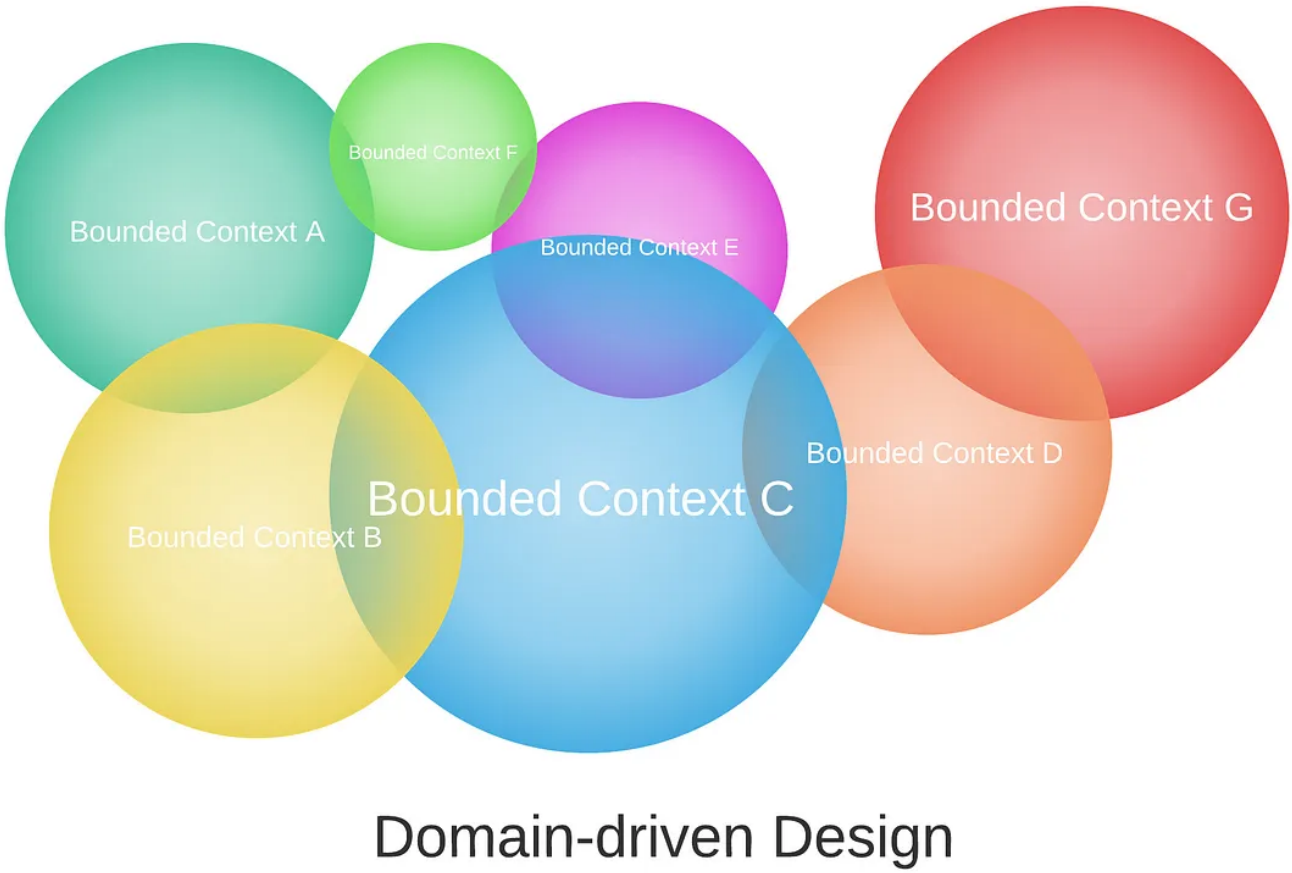
Marcelo M. Gonçalves Follow
Dec 20, 2019 · 11 min read

Domain-driven Design (DDD) em uma Arquitetura de Microsserviços



O processo de construção de componentes de software, implementados utilizando uma arquitetura de microsserviços, não se trata apenas em nos preocuparmos com a estrutura de arquitetura e engenharia interna, nem de aplicar as últimas tendências em tecnologia, tampouco utilizar padrões de design corporativos (**Enterprise Design Patterns**).

Desenhar camadas inerentes à infraestrutura e redes da aplicação (**service mesh**), estando ciente das complexidades presentes em sistemas distribuídos, juntamente com a preocupação nos dados constituem alicerces fundamentais para a implantação bem sucedida de componentes com natureza distribuída. Portanto, quando nos referimos aos nossos microsserviços, a modelagem e transformação dos dados em nosso domínio de negócio desempenham papel crucial durante todo o ciclo de vida deste processo.

> *A etapa de **design do modelo do domínio de dados** apresenta alta complexidade, tratando-se de uma tarefa crítica em direção ao sucesso na adoção do **DDD** ao sistema sendo desenvolvido.*

Certamente, modelar o núcleo de dados de nossa aplicação, atacando todos detalhes existentes ao desbravar o modelo de negócio, não se trata de uma tarefa trivial e para isso existe um conjunto de técnicas e práticas avançadas visando auxiliar no processo de modelagem e qualificação do domínio de negócio.

Estas técnicas propõem-se a entregar uma representação mais precisa do nosso modelo de dados bem como dos limites transacionais em torno destes. Nos referimos a este conjunto de práticas de modelagem como **Domain-driven Design (DDD)**, e neste artigo iremos explorar em detalhes alguns dos Design Patterns pertencentes ao DDD com o objetivo de entender estas técnicas do ponto de vista prático.

Design Estratégico e Tático

Domain-driven Design, descrito em seu nível mais alto de abstração, refere-se basicamente a duas grandes camadas de design dos dados: **design estratégico e design tático**. Desta forma, não sendo possível obter sucesso na implementação de técnicas de design tático sem aplicar primeiramente práticas do design estratégico.

A continuidade e aperfeiçoamento nas etapas do fluxo de modelagem do domínio ocorre conforme o avanço no processo de análise e definição dos objetos identificados no domínio de negócio. As etapas de design, estratégica e tática, agrupam seus respectivos conjuntos de padrões e técnicas de acordo com a fase de maturidade da modelagem.

> *Assim sendo, a etapa estratégica se aproxima da fase inicial e mais rudimentar da visão sobre os dados ao mesmo tempo em que a etapa de design tática começa a ser explorada*

> *quando já existe uma visão mais concreta do domínio e relações entre os componentes do modelo.*

A etapa de design tático pode ser vista como uma lapidação fina dos objetos, iniciando uma transição em direção a transformarem-se em algo mais concreto e definitivo, refletindo com a maior precisão o domínio de negócio. Assim, quando construímos nosso Domain Model (modelo de dados) utilizando a terminologia DDD (Domain-driven Design), transitamos entre a etapa de design estratégico, identificando **Bounded Contexts** (Contextos Delimitados) e Ubiquitous Language (**Linguagem Ubíqua**).

Passando para a etapa subsequente, evolutivamente abrangendo características relacionadas ao design tático, na prática identificamos **Entities** (Entidades), **Value Objects** (Objetos de Valor) e Aggregates (Agregados). Sendo os Aggregates, representando o último nível de abstração da modelagem dos dados.

Design estratégico se refere a quebrar a complexidade do trabalho a ser realizado em grupos de maneira a avaliar as melhores formas de integrá-los. A etapa de design estratégico inicialmente oferece Design Patterns como **Bounded Context** e **Ubiquitous Language**. Domain-driven Design, em sua forma mais pura, trata-se de modelar uma Ubiquitous Language em um Bounded Context.

Ao fazermos isso, explicitamente definimos componentes internos sendo representados e referenciados dentro do contexto por meio dos termos cunhados e existentes na linguagem universal (ubíqua) definida. Assim, necessitamos aprender a separar nossos modelos utilizando-os exclusivamente dentro de contextos delimitados.

A semântica presente nos **Bounded Contexts** expressam os significados dos termos utilizados no contexto, resolvendo problemas de ambiguidade, atuando como um delimitador entre os contextos. Significa que dentro de cada contexto, para cada componente de software existe um significado específico, o qual é referenciado por realizar determinadas tarefas ao relacionar-se com os demais componentes dentro da fronteira transacional a qual pertence.

Espaço do Problema e da Solução

Quando iniciamos o processo de modelagem, o **Bounded Context** é algo conceitual residindo no **Problem Space (Espaço do Problema)** e nomeado como Subdomínio.

Na medida em que nosso modelo avança, adquirindo maior profundidade em suas definições e se tornando mais claro, seu Bounded Context visivelmente inicia uma transição para o **Solution Space (Espaço da Solução)**. Neste momento, passando a aproximar-se da modelagem em formato de implementação/código.

> ***Domain-driven Design*** *- Refere-se a* ***design*** *e não sobre* ***códigos*** *ou implementações explícitas.*

Ao aprofundar-nos nas práticas de Domain-driven Design, explorando os limites do **Problem Space**, realizamos análises estratégicas em um nível mais abstrato/básico, identificando objetivos e riscos identificados envolvidos.

De maneira que, o **Solution Space**, cujo contexto abriga a solução final, passamos a refletir sobre o resultado do esforço de análise estratégica e tática aplicados de forma conjunta ao **Problem Space (Core Domain)**. Assim, resultando em uma representação mais precisa de nossos componentes de software, expressados em **Aggregates** (Agregados) e refletindo o modelo de dados final permanentemente.

Bounded Contexts e Ubiquitous Language

O modelo de domínio é implementado dentro dos Bounded Contexts por meio de artefatos de software separados para cada contexto. O modelo definido dentro dos limites de cada contexto reflete a linguagem que é desenvolvida pelo time para trabalhar e se comunicar dentro de cada **Bounded Context**. A linguagem é chamada **Ubiquitous Language** pois ao mesmo tempo que é utilizada entre todos os membros do time (incluindo os **Domain Experts**), também é implementada como **Source Code**.

> *É essencial que a Ubiquitous Language seja rigorosa e precisa em manter os significados dos conceitos em seus respectivos Bounded Contexts (limites transacionais). Limites transacionais se referem a menor unidade atômica necessária a respeito das* ***Business Invariants*** *(Regras de Negócio).*

Devendo os limites transacionais dos componentes serem tão pequenos quanto possível, idealmente uma única transação para cada componente, dado que não podemos considerar transações atômicamente distribuídas, transcendendo os seus limites contextuais de cada Bounded Context. Desta forma, precisamos identificar **Transaction Boundaries** (Limites Transacionais), bem como definir suas fronteiras transacionais.

Ao praticar técnicas utilizando **Domain-driven Design**, aplicamos Design Patterns como **Aggregates** com o objetivo de expressar, em última instância, tais limites mencionados. Desta forma, tornando os Aggregates responsáveis por encapsular **Entities e Value Objects**, mantendo as Invariants e Constraints (Regras de Negócio) consistentes e íntegras.

O padrão **Aggregates** desempenha um papel fundamental ao relacionar terminologias utilizadas no **Source Code** da aplicação aos conceitos e termos presentes na **Ubiquitous Language**, onde podem existir diferentes significados entre contextos além de estarem fortemente acoplados às definições de limites de transação dos dados pertencentes a cada operação, podendo, os Bounded Contexts conter múltiplos Aggregates em sua composição.

Agilidade em Times Menores com DDD

Uma das razões para utilizarmos microsserviços é permitir o baixo acoplamento entre os times com o mínimo de impacto e com velocidades diferentes entre as equipes. Pois times autônomos, prontos para reagir a mudanças tão rápido quanto o surgimento das alterações em nosso domínio agregam maior valor ao negócio, reduzindo custos operacionais relacionados à comunicação entre as equipes promovendo maior agilidade.

Ao adquirirmos determinada agilidade dentro dos times, começaremos a nos aproximar de algo cujo formato se parece com microsserviços pois passamos a possuir maior autonomia na medida em que eliminamos dependências, ao passo de que, para conquistar tal nível de maturidade será necessário dedicação e esforço incomuns.

> *De maneira geral, implementar **DDD**, na prática, trata-se de um processo lento, complexo e cheio de armadilhas. Assim, demandando esforço dos times responsáveis por transformar um conjunto de dados inicial, sem relações e com pouca representatividade, em um modelo negócio conciso capaz de refletir com exatidão seu domínio de negócio.*

Inevitavelmente, a partir de um domínio de dados mais complexo, precisamos de algo como Domain-driven Design para entender os modelos que usaremos para implementar nosso sistema e desenhar os limites transacionais entre os componentes. Assim sendo, ao assumir a responsabilidade de analisar o domínio de negócio utilizando DDD (Domain-driven Design) ajudará a implementar uma

arquitetura de microsserviços de forma mais assertiva em direção ao sucesso da aplicação.

Os princípios conceituais presentes no DDD (Domain-driven Design) se referem a construção de uma linguagem compartilhada entre os envolvidos no processo (Ubiquitous Language entre Software e Business), identificando-se contextos e dentro de cada contexto definido (Bounded Context) adotar um significado para os termos utilizados. Podendo manter o foco em atacar a complexidade do domínio de negócio.

Definir os Bounded Contexts é a primeira coisa a ser feita por meio de discussões entre o time técnico e o pessoal de negócio. Após, é necessário dar início a construção da Ubiquitous Language (Linguagem Universal), dando sentido aos componentes nos contextos apropriados. Conceitualmente, deveremos ter um único time trabalhando por Bounded Contexts (Contextos Delimitados), com repositórios de Source Code (Código Fonte) separados por contexto, sendo ainda possível um time trabalhar em diversos Bounded Contexts porém múltiplos times não deveriam trabalhar em um único Bounded Context.

> *Cada time, responsável por determinado Bounded Context, define as interfaces de comunicação que serão utilizadas por quem tiver interesse em interagir com aquele contexto, sendo esta camada de isolamento um dos principais benefícios de se utilizar Domain-driven Design.*

O grande motivo para utilizar-se Bounded Contexts e promover uma separação enxuta dos componentes identificados no domínio, é que a maior parte dos times não sabem a hora de parar de jogar objetos para dentro do modelo de dados, e devido a esta falha acabam criando uma Big Ball of Mud (Grande bola de lama), obtendo-se um modelo anêmico, complexo e difícil de manter, resultando na criação de um monólito em sua pior forma, a qual não existirão motivos dos quais se orgulhar.

Complexidade do Domínio na Organização

Ao aplicar DDD (Domain-driven Design) na prática sua organização/empresa explicitamente reflete as principais características de seu domínio de negócio. O design tático e suas técnicas são direcionadas a uma etapa de lapidação dos componentes de software de forma mais elaborada.

O padrão Aggregate é utilizado nesta camada pois apresenta maiores detalhes sobre como agregar entidades e objetos de valor dentro de clusters com tamanhos precisamente adequados da maneira mais explícita, de forma que o modelo começa pequeno e gerenciável, crescendo naturalmente em termos de complexidade sem perder o controle sobre seus limites e relacionamentos entre os componentes.

Em um cenário cada vez mais competitivo, DDD (Domain-driven Design) auxilia você e seu time a entregar softwares que possuem design e implementação cada vez mais efetivos. Um design efetivo obriga a organização a pensar na construção de um modelo de domínio mais coerente. Conforme identificamos o tamanho e complexidade de nosso domínio, ao modelarmos seus objetos, teremos diversos problemas de ambiguidade e para isso precisamos de limites.

> *Para alcançar estes limites, delimitamos os contextos existentes dentro de nosso domínio, identificando e expressando seus artefatos utilizando Entities, Value Objects, e Aggregates gerando inicialmente um modelo contendo entidades que refletem o negócio.*

Desta forma, em cima do modelo de domínio gerado, delimitamos os contextos, limitando as entidades e agregados juntamente com seus valores envolvidos. Estes limites, em alguns casos, acabam se tornando nossos microsserviços naturalmente, em outros casos, componentes internos a estes Bounded Contexts refletem melhor cada microsserviços em granularidade mais fina. Em todos os casos, microsserviços se referem aos limites que damos aos artefatos que identificamos dentro de nosso domínio e suas relações.

Business Invariants (Regras de Negócio) nos Agregados

Ao modelar microsserviços utilizando DDD (Domain-driven Design) devemos nos preocupar com a garantia de consistência dos componentes, do ponto de vista das regras de negócio, representadas no DDD como Business Invariants (Invariantes/Regras de Negócio). Queremos manter as Business Invariants intactas em nosso domínio aplicando DDD (Domain-driven Design), modelando estas Invariants dentro de nossos agregados em transações únicas, ocorrendo o processo de validação da integridade dos dados relacionados a determinada instância em execução.

> *Idealmente, devemos ter uma transação por **Aggregate**, podendo em alguns casos existir atualizações em múltiplos Aggregates dentro de uma mesma transação, excepcionalmente*

> *devendo ser contornada através dos* ***Process Managers (SAGA Pattern) com BASE Transactions****.*

Ainda, ao aplicarmos conceitos sobre Domain-driven Design em nossa aplicação, outros padrões de design interessantes emergem como opções de implementação conjunta, como o conhecido design pattern CQRS (Command Query Responsibility Segregation), possibilitando-nos direcionar comandos aos agregados, além de ser uma boa maneira de propagar as mudanças de dados (Data Changes) entre os Bounded Contexts.

Independente da maneira como o processo de modelagem utilizando DDD foi concebido em sua aplicação, o contexto é a chave que dará início ao processo de modelagem. Possibilitando aos humanos resolver qualquer problema de ambiguidade. Contextos são sobre a semântica dos dados, seus significados e representação dentro de cada universo, portanto, em uma arquitetura de microsserviços que promove o uso de DDD (Domain-driven Design), os dados são cidadãos de primeira classe.

Comunicação através de Eventos (Domain Events)

A comunicação entre os microsserviços, além dos limites de seus Bounded Contexts, em uma aplicação que promove a prática do DDD (Domain-driven Design), deveria ocorrer por meio de Domain Events (Eventos de Domínio). Tanto do ponto de vista interno quanto externo aos contextos de cada microsserviços, eventos são utilizados para garantir a consistência eventual entre Transaction Boundaries (Limites Transacionais) e Bounded Contexts (Contextos Delimitados).

Portanto, eventos tratam-se de estruturas imutáveis que capturam acontecimentos na linha do tempo podendo serem entregues aos componentes inscritos e interessados em reagir ao mesmo. Praticar estratégias de Domain-driven Design ajudam você e seu time a construir software obtendo benefícios sobre organização de domínio de forma mais coerente, maior consistência de seus dados, além de facilitar decisões sobre design e integrações futuras. Sendo possível tornar os componentes estrategicamente mais flexíveis e inteligentes quando aplicado este o conjunto de técnicas.

Uma vez que o processo tenha sido executado com sucesso, basta dar uma olhada a fundo em seu domínio de negócio, refletido nos modelos identificados e seus dados o ajudarão a enxergar onde estão seus microsserviços de forma intrínseca. Pois, o

simples fato de utilizarmos frameworks/tecnologias A ou B para desenvolver nossos componentes, não significa estarmos necessariamente implementando uma arquitetura de microsserviços tampouco apresentando um design consistente sobre o modelo de dados.

Considerações Finais

Entender os princípios conceituais do DDD trata-se do ponto de partida para compreensão dos conceitos propostos, para isso existem livros inteiros dedicados ao assunto com exclusividade e riqueza de detalhes, como Domain-Driven Design: Tackling Complexity in the Heart of Software escrito por **Eric Evans**, Implementing Domain-Driven Design e Domain-Driven Design Distilled escritos por **Vaughn Vernon**, Applying Domain Driven Design And Patterns escrito por **Jimmy Nilsson**, Patterns, Principles, and Practices of Domain-Driven Design escrito por **Scott Millett** e **Nick Tune**, entre outros.

> *Além destes autores com autoridade no assunto como **Eric Evans, Vaughn Vernon, Christian Posta, Martin Fowler, Martin Kleppmann, Jimmy Nilsson**, etc. é possível encontrar uma infinidade de artigos na internet dedicados ao assunto.*

Acompanhar a evolução de técnicas e práticas utilizadas na implementação do **Domain-driven Design** em seus projetos exige grande curva de aprendizado. Aprofundar-se nas definições citadas, bem como compreender os termos empregados, presentes nos livros de referência, depende exclusivamente de nós e do nosso interesse em evoluir e buscar cada vez mais informações visando ajudar nas tomadas de decisão em futuros projetos.

Ao adquirirmos determinadas responsabilidades precisamos construir uma base cada vez mais sólida, facilitando os trade-offs e restringindo as margens de risco. Particularmente, iniciativas envolvendo quaisquer tecnologias/padrões, e **Domain-driven Design** não é exceção, referem-se a algo muito individual, podendo guiar nossas carreiras profissionais em direção ao sucesso.

Ddd Domain Driven Design Microservices Microservice Architecture