Resenha: Design Estratégico em DDD (Capítulos 4, 5 e 6 da Referência de Eric Evans)

Introdução: Além do Modelo Único

O Domain-Driven Design (DDD) oferece um conjunto robusto de ferramentas para lidar com a complexidade no coração do software. Enquanto os padrões táticos (como Entidades, Agregados e Repositórios) fornecem os blocos de construção para um modelo rico, os capítulos 4, 5 e 6 da referência de Eric Evans nos elevam a uma perspectiva mais alta: o **Design Estratégico**. Essa abordagem reconhece uma realidade fundamental dos sistemas complexos: eles não são compostos por um único modelo unificado, mas por um ecossistema de múltiplos modelos. O Design Estratégico nos ensina a mapear, organizar e priorizar esse ecossistema para entregar o máximo de valor ao negócio.

Capítulo 4: Mapeando o Terreno (Context Mapping for Strategic Design)

O primeiro pilar do Design Estratégico aborda a inevitabilidade de múltiplos modelos em qualquer projeto de grande porte. Equipes diferentes, com objetivos distintos, naturalmente desenvolvem linguagens e modelos próprios. Tentar forçar um modelo único para toda a organização é uma receita para o fracasso. Em vez disso, o DDD nos instrui a abraçar essa diversidade e gerenciá-la explicitamente.

O padrão central aqui é o **Context Map** (Mapa de Contextos). Não se trata de um diagrama técnico formal, mas de uma ferramenta de visualização e comunicação que descreve as fronteiras entre os diferentes **Bounded Contexts** (Contextos Delimitados) e as relações entre eles. Cada contexto tem sua própria **Ubiquitous Language** (Linguagem Ubíqua) e seu próprio modelo. O mapa expõe como esses contextos interagem, permitindo que as equipes tomem decisões informadas sobre integração e colaboração.

Os padrões de relacionamento descritos neste capítulo oferecem um vocabulário para descrever essas interações:

- **Padrões de Colaboração:**

    - **Partnership (Parceria):** Duas equipes/contextos unem forças, pois seu sucesso ou fracasso está interligado. Exige colaboração intensa e sincronização constante.
    - **Shared Kernel (Núcleo Compartilhado):** Um subconjunto do modelo é compartilhado explicitamente entre dois contextos. É uma relação forte que exige alta coordenação para evitar que uma equipe quebre o modelo da outra.
- **Padrões de Cliente/Fornecedor (Upstream/Downstream):**
    - **Customer/Supplier (Cliente/Fornecedor):** Uma relação de dependência onde a equipe *downstream* (cliente) depende da equipe *upstream* (fornecedora). Idealmente, a equipe cliente tem poder de influência sobre as prioridades da equipe fornecedora.
    - **Conformist (Conformista):** A equipe *downstream* adota o modelo do *upstream* sem questionar. É uma solução pragmática quando a equipe *upstream* não tem interesse em atender às necessidades do cliente ou quando a integração é mais importante que a integridade do modelo *downstream*.
- **Padrões de Desacoplamento:**
    - **Anticorruption Layer (Camada Anticorrupção - ACL):** A equipe *downstream* constrói uma camada de tradução que isola seu modelo das peculiaridades do modelo *upstream*. Isso protege o modelo *downstream* e lhe dá autonomia, embora adicione complexidade de implementação.
    - **Open-Host Service (Serviço de Host Aberto):** A equipe *upstream* define um protocolo público e estável (como uma API REST) para que múltiplos clientes possam se integrar. O foco está em fornecer um serviço claro e bem documentado.
    - **Separate Ways (Caminhos Separados):** A decisão consciente de *não* integrar dois contextos. Às vezes, a complexidade da integração supera os benefícios.

A ausência de um mapa de contextos claro e de relações bem definidas geralmente leva ao antipadrão **Big Ball of Mud** (Grande Bola de Lama), onde as fronteiras são borradas, o código é emaranhado e a manutenção se torna um pesadelo. O mapeamento de contextos é, portanto, a primeira e mais crucial etapa para evitar o caos em sistemas de grande escala.

Capítulo 5: Encontrando o Coração (Distillation for Strategic Design)

Uma vez que o terreno foi mapeado, a próxima pergunta estratégica é: "Onde devemos concentrar nossos esforços?". Nem todas as partes de um sistema de software são igualmente importantes. A "destilação" é o processo de identificar e priorizar as partes do domínio que são verdadeiramente essenciais e que fornecem uma vantagem competitiva para o negócio.

O conceito central deste capítulo é o **Core Domain** (Domínio Principal). Este não é necessariamente o maior ou o mais complexo subdomínio, mas sim aquele que contém a lógica de negócio única e valiosa. É o coração da aplicação, a razão de sua existência. Em contraste, os **Generic Subdomains** (Subdomínios Genéricos) são partes necessárias, mas não diferenciadoras, do sistema (por exemplo, autenticação, envio de e-mails). Eles são candidatos ideais para serem resolvidos com soluções prontas (frameworks, bibliotecas) ou terceirização.

Os padrões de destilação ajudam a equipe a focar e proteger o Core Domain:

- **Domain Vision Statement (Declaração de Visão do Domínio):** Um resumo conciso que descreve o objetivo e o valor do Core Domain, alinhando a equipe e os stakeholders.
- **Highlighted Core (Núcleo Destacado):** Tornar o Core Domain explícito na documentação, na base de código e nas conversas da equipe, para que todos entendam sua importância.
- **Segregated Core (Núcleo Segregado):** Refatorar o código para separar fisicamente o Core Domain de outros elementos de suporte. Isso o torna menor, mais coeso e mais fácil de entender e evoluir.
- **Cohesive Mechanisms (Mecanismos Coesos):** Extrair frameworks técnicos complexos (como um mecanismo de persistência genérico) do Core Domain para que o modelo principal permaneça focado exclusivamente na lógica de negócio.

A destilação é um exercício estratégico contínuo que garante que os melhores desenvolvedores e os maiores investimentos estejam focados onde eles podem gerar o maior impacto.

Capítulo 6: Estruturando o Todo (Large-Scale Structure for Strategic Design)

Este capítulo aborda como impor uma ordem coerente em todo o sistema, especialmente em projetos muito grandes e de longa duração. Os padrões de estrutura em larga escala fornecem "plantas" de alto nível para organizar os diferentes contextos e componentes de software.

Ao contrário de uma arquitetura rígida definida no início, o DDD favorece uma **Evolving Order (Ordem Evolutiva)**, onde a estrutura em larga escala emerge e se adapta à medida que o entendimento do domínio se aprofunda. No entanto, alguns padrões podem guiar essa evolução:

- **System Metaphor (Metáfora do Sistema):** Uma analogia ou narrativa simples e poderosa que ajuda toda a equipe a compartilhar uma visão unificada sobre como o sistema funciona.
- **Responsibility Layers (Camadas de Responsabilidade):** Uma forma de arquitetura que organiza o modelo de domínio em camadas conceituais com base na sua responsabilidade e taxa de mudança. Por exemplo, uma camada pode lidar com "decisões" (regras de alto nível), enquanto outra lida com "capacidades" (processos de negócio) e uma terceira com "operações" (interações com sistemas externos). Isso difere das camadas técnicas (UI, Aplicação, etc.).
- **Knowledge Level (Nível de Conhecimento):** Um padrão sofisticado que separa um modelo em dois níveis: um que contém as instâncias operacionais (um "Cliente", um "Produto") e outro que contém as regras e classificações que governam essas instâncias (as "Regras de Desconto", os "Tipos de Cliente"). É útil para domínios onde as próprias regras de negócio são dinâmicas e configuráveis.
- **Pluggable Component Framework (Framework de Componentes Conectáveis):** Projetar o sistema de forma que subdomínios inteiros possam ser implementados como componentes que podem ser "plugados" na aplicação principal. Isso promove o desenvolvimento independente e a flexibilidade.

Esses padrões oferecem diferentes lentes para pensar sobre a coesão e o acoplamento em uma escala macro, permitindo que a arquitetura do sistema cresça de forma sustentável.

Conclusão

O Design Estratégico do DDD, conforme apresentado nos capítulos 4, 5 e 6, é uma disciplina essencial para arquitetos e líderes técnicos. Ele nos ensina a olhar para além do código e a pensar sobre sistemas como um ecossistema de modelos, equipes e objetivos de negócio. Ao fornecer um vocabulário para mapear contextos, um método para focar no que é mais importante e padrões para organizar a estrutura geral, o Design Estratégico transforma o desenvolvimento de software de um exercício puramente técnico em uma atividade estratégica alinhada ao sucesso do negócio.