PEDRO DA COSTA MELO VIEIRA

Aplicando técnicas de *Domain-Driven Design* para Reduzir Acoplamento e Aumentar a Coesão em uma Arquitetura de Microsserviços

São Paulo
2025

PEDRO DA COSTA MELO VIEIRA

Aplicando técnicas de *Domain-Driven Design* para Reduzir Acoplamento e Aumentar a Coesão em uma Arquitetura de Microsserviços

Versão Original

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de Software.

Área de Concentração: Tecnologia de Software

Orientador: Prof. Dr. Paulo Sérgio Muniz Silva

São Paulo

2025

FICHA CATALOGRÁFICA

VIEIRA, Pedro da Costa Melo

Aplicando técnicas de Domain-Driven Design para Reduzir Acoplamento e Aumentar a Coesão em uma Arquitetura de Microsserviços / P. C. M. VIEIRA - São Paulo, 2025.

69 p.

Monografia (MBA em Tecnologia de Software) - Escola Politécnica da Universidade de São Paulo. PECE – Programa de Educação Continuada em Engenharia.

1.Domain-Driven Design 2.Arquitetura de Software 3.Microsserviços I.Universidade de São Paulo. Escola Politécnica. PECE – Programa de Educação Continuada em Engenharia II.t.

Nome: VIEIRA, Pedro da Costa Melo.

Título: Aplicando técnicas de Domain-Driven Design para Reduzir Acoplamento e Aumentar a Coesão em uma Arquitetura de Microsserviços

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de Software.

Aprovado em: / /

Banca Examinadora

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

DEDICATÓRIA

À minha esposa, Luisa, e meus gatos, Fiona e Bento!
Venceremos, sempre.

AGRADECIMENTOS

Agradeço a Escola Politécnica da Universidade de São Paulo – EPUSP e a todo corpo docente do curso de Engenharia de Software, por todo o conhecimento compartilhado, o qual agrega muito em minha carreira.

Ao meu orientador Prof. Dr. Paulo Sérgio Muniz Silva, onde desde as aulas ministradas ao longo do curso até todos os momentos em que fui seu orientando, se mostrou absolutamente especialista nos assuntos que tangem a monografia, contribuindo muito com a evolução do meu conhecimento sobre o tema.

Aos meus amigos e colegas de trabalho que me incentivaram a ingressar no curso e que juntos conseguimos concluir essa jornada, além de contribuírem diretamente em minha evolução profissional: Arthur Machado, Fernando Godoy, Luan Sales e Renan Ferreira.

Por fim, à minha esposa Luisa, a qual me apoiou e apoia incondicionalmente em todas as novas etapas que surgem em minha vida.

RESUMO

VIEIRA, Pedro da Costa Melo. Aplicando técnicas de Domain-Driven Design para Reduzir Acoplamento e Aumentar a Coesão em uma Arquitetura de Microsserviços. 2025. 69. Monografia (MBA em Tecnologia de Software). Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo. São Paulo. 2025.

Este trabalho apresenta uma solução de projeto (*design*) utilizando técnicas do Projeto Dirigido pelo Domínio – *Domain-Driven Design* (DDD) – para refatorar um produto de *software* existente, implementado com arquitetura de microsserviços. Tal solução visa aumentar a coesão dos microsserviços e reduzir o acoplamento entre eles, alinhando seus modelos de domínio de forma mais acurada às capacidades do negócio (*business capabilities*). A pesquisa foi motivada por problemas identificados em um mau projeto de modularização, que resultaram em ambiguidades conceituais, baixa coesão e alto acoplamento. Foram utilizadas as técnicas de *design* estratégico e tático do DDD, com ênfase na identificação de subdomínios de negócio, mapeamento de contextos delimitados e realinhamento de seus modelos de domínio. Os artefatos resultantes das soluções são constituídos por modelos expressos na linguagem *UML* e apoiados por justificativas fundamentadas que mostram como a adequação da solução de projeto (*design*) utilizando as técnicas do DDD pode ajudar a melhorar a expressividade, organização dos modelos de domínio e a modularização de *software*. Por fim, conclui-se que a aplicação de DDD ajuda não apenas a mitigar problemas técnicos, mas também facilita a comunicação entre equipes e promove maior alinhamento semântico e estrutural dos elementos de *software* e as capacidades do negócio.

Palavras-chave: *Domain-Driven Design*; *Microsserviços*; *Acoplamento; Coesão*.

ABSTRACT

VIEIRA, Pedro da Costa Melo. Aplicando técnicas de Domain-Driven Design para Reduzir Acoplamento e Aumentar a Coesão em uma Arquitetura de Microsserviços. 2025. 69. Monografia (MBA em Tecnologia de Software). Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo. São Paulo. 2025.

This work presents a design solution using Domain-Driven Design (DDD) techniques to refactor an existing software product implemented with a microservices architecture. The proposed solution aims to increase microservices cohesion and reduce coupling, aligning their domain models more accurately with business capabilities. The research was motivated by issues identified in a poorly modularized design, which resulted in conceptual ambiguities, low cohesion, and high coupling. Strategic and tactical DDD techniques were applied, focusing on the identification of business subdomains, mapping bounded contexts, and realigning their domain models. The resulting artifacts are composed of UML models supported by well-founded justifications, demonstrating how the proposed design solution using DDD techniques can improve the expressiveness, organization of domain models, and modularization of the software. Finally, it is concluded that applying DDD not only mitigates technical problems but also facilitates communication between teams, promoting greater semantic and structural alignment between software elements and business capabilities.

Keywords: *Domain-driven Design*; *Microservices*; *Coupling; Cohesion*.

LISTA DE ILUSTRAÇÕES

Pág.

LISTA DE TABELAS

Pág.

LISTA DE ABREVIATURAS E SIGLAS

[DDD	Domain-Driven Design]
[UML	Unified Modeling Language]
[SRP	Single Responsibility Principle]
[CD	Contexto Delimitado]
[CDs	Contextos Delimitados]
[GO	Gerenciador de Ordens]
[PO	Processador de Ordens]
[GF	Gerenciamento de Fundos]
[GP	Gestão Patrimonial]

SUMÁRIO

Pág.

1. INTRODUÇÃO

1.1 Motivações

Adotada por gigantes da tecnologia, como Netflix e Amazon (NEWMAN, 2021), uma abordagem de arquitetura que se tornou um padrão de mercado na última década é o estilo arquitetônico de microsserviços. De acordo com Fowler (2014), arquitetura de microsserviços é uma abordagem que permite o desenvolvimento de uma aplicação como um conjunto de pequenos serviços, cada um executando em seu próprio processo e se comunicando por mecanismos leves, como APIs HTTP. Esses serviços são construídos em torno de capacidades de negócio, sendo independentes em termos de implantação e gerenciamento, com o mínimo de centralização.

(NEWMAN, 2021) complementa essa definição afirmando que a arquitetura de microsserviços é um tipo de arquitetura orientada a serviços, que possui uma definição clara a respeito das fronteiras a serem traçadas de modo a permitir implantações independentes. Tal arquitetura tem como grande vantagem ser independente de tecnologias, podendo ser desenvolvida com diferentes linguagens de programação, por exemplo.

Um dos grandes desafios ao construir uma arquitetura de microsserviços é decompor adequadamente o sistema em serviços distribuídos com baixo acoplamento entre eles. Isto é, conseguir construir os serviços individualizados de forma que reflitam as necessidades específicas de negócio que cada um deve realizar. Ao não atingir esse objetivo, esta decomposição resultará em serviços com responsabilidades mal definidas (ou até mesmo conflitantes entre eles), dificultando sua manutenção e evolução. Pontos como esses podem, inclusive, levar empresas a migrar produtos de volta para arquiteturas monolíticas, como foi o caso da Amazon em 2023 (SU, LI e TAIBI, 2024).

Conhecida no mundo do desenvolvimento de software como Lei de *Conway,* ela diz que "*organizações que projetam sistemas irão produzir designs que são cópias de sua estrutura de comunicação*" (CONWAY, 1968 - tradução livre). Esse princípio sugere que a estrutura de um sistema ou software é um reflexo da estrutura organizacional da comunicação da equipe que o desenvolve, impactando o modo como os softwares serão desenvolvidos e como,

considerando um ecossistema onde há múltiplos serviços, tais serviços se comunicarão. Segundo (FOWLER, 2022), há três respostas possíveis à lei de Conway, descritas na Tabela 1.

Tabela 1 – As três respostas à lei de Conway, segundo Fowler.

Ação	**Descrição**
Ignorar	Não levar em conta por não saber de sua existência ou não considerá-la relevante.
Aceitar	Reconhecer seu impacto e garantir que sua arquitetura não entre em conflito com os padrões de comunicação dos times.
Manobra de Inversão de Conway	Alterar os padrões de comunicação para encorajar a arquitetura de software desejada.

Fonte: (FOWLER, 2022), adaptado pelo autor.

Fowler (2022) observa que, apesar de ser uma ferramenta útil, a lei de Conway não é uma solução que irá resolver instantaneamente os problemas enfrentados no desenvolvimento de software da organização. Para cenários onde já existe uma estrutura de comunicação definida, é necessário mudar tanto a base de código como a própria organização.

A terceira resposta à lei de Conway é uma abordagem interessante para se construir uma arquitetura de microsserviços, a qual busca modificar intencionalmente a forma de comunicação entre times, quebrando os silos que atrapalham uma comunicação eficiente entre eles (LEROY J., SIMONS M., 2010).

Introduzida por Eric Evans (EVANS, 2003), a abordagem de projeto (*design*) de software dirigido pelo domínio (*Domain-Driven Design - DDD*) visa construir sistemas adotando intencionalmente as estruturas organizacionais de comunicação da empresa, de forma a construir seus sistemas separando suas responsabilidades e limitando suas fronteiras de

maneira bem definida, refletindo explicitamente suas competências centrais do negócio (VERNON, 2016).

1.2 Objetivo

Esta monografia apresenta uma solução de projeto (*design*) utilizando técnicas do Projeto Dirigido pelo Domínio – *Domain Driven Design* (DDD) – para refatorar um produto de software existente implementado com arquitetura de microsserviços, visando alinhá-lo mais acuradamente às capacidades do negócio (*business capabilities*), de modo a permitir o aumento da coesão dos microsserviços e a redução do acoplamento entre eles.

Refatora-se aqui parte de um produto de software complexo e estratégico existente, espelhado de modo anonimizado de uma empresa real, implementado com uma arquitetura de microsserviços. A refatoração ocorre em dois níveis: tanto no plano estratégico, realinhando o projeto (*design*) do software, com determinadas capacidades do negócio, como no plano tático, minimizando efeitos colaterais negativos, como baixa coesão e alto acoplamento, decorrentes de mau projeto de modularização.

Aqui, as soluções de refatoração restringem-se somente a soluções de *design*. Os artefatos resultantes das soluções são constituídos por modelos expressos na linguagem *UML* apoiados por justificativas explicativas. Assume-se, portanto, duas limitações importantes que restringem o escopo do presente trabalho: tanto a falta de discussão de alternativas de implementação das soluções de *design* propostas como a falta de uma análise de compensações (*trade-off*) dos custos envolvidos nas soluções.

1.3 Justificativas

A revisão bibliográfica do capítulo 2 evidencia que o tema de modelagem de microsserviços, visando uma decomposição que utiliza técnicas como o DDD para tratar problemas de coesão e de acoplamento é um assunto relevante hoje em dia.

A decisão para uma decomposição arquitetônica de um produto de software, que mantenha especialmente sua evolução alinhada à evolução do negócio ao qual dá suporte, implica

projetá-la de modo a obter o mínimo de acoplamento possível entre seus componentes, o que implica o aumento da coesão de cada um deles. Comparativamente a uma arquitetura não distribuída, a natureza distribuída um produto de software implementado em microsserviços requer cuidados adicionais para sua decomposição. Por exemplo, (SU, LI E TAIBI, 2024) discorrem sobre o movimento da decisão do retorno para uma arquitetura monolítica a partir de uma arquitetura de microsserviços. Neste estudo são abordados cinco principais motivos pelos quais uma empresa opta por fazer esse retorno. Dois deles são mais aderentes à esta monografia:

- **Complexidade**: Ter diferentes times trabalhando em serviços que, muitas vezes, possuem, inclusive, linguagens de programação diferentes, traz uma complexidade de controle maior do que é desenvolvido. Além disso, é necessário que um engenheiro precise ter o entendimento de diversos contextos diferentes para ter o conhecimento do produto como um todo.

- **Organização**: A dificuldade no gerenciamento de times pode ter um grande impacto no *design* e manutenção de uma arquitetura de microsserviços. Em determinado caso descrito por (SU, LI E TAIBI, 2024), há um cenário de uma equipe que não apresenta um tamanho apropriado para lidar com as diversas responsabilidades de diferentes microsserviços, linguagens de programação, etc., levando de volta o produto de software para uma arquitetura monolítica.

No contexto prático, é possível perceber, a partir dos motivos acima, que empresas enfrentam dificuldades ao realizarem a transição de uma arquitetura monolítica para uma arquitetura de microsserviços, muitas vezes por falta de uma metodologia clara que guie a modelagem adequada de seus domínios de negócio. A aplicação de técnicas de *DDD*, como Mapas de Contexto, Linguagem Ubíqua, *Design* Estratégico e *Design* Tático, surge como uma solução promissora para melhorar a coesão dos subdomínios do negócio e dos serviços dos produtos de software que lhes dão suporte, minimizando o acoplamento entre estes últimos ao possibilitar um alinhamento consistente entre subdomínios e serviços.

De acordo com Vernon (2016), as ferramentas do *design* estratégico do DDD podem ajudar as equipes de software a fazerem melhores escolhas para a decomposição do software e a tomarem decisões coerentes de integração entre seus componentes. Dessa forma, os modelos

de *design* do produto de software refletirão explicitamente os potenciais de capacidade do negócio (*business capabilities*).

1.4 Contribuição

O presente estudo oferece uma contribuição em dois planos: prático e conceitual. No plano prático, aplicam-se técnicas de *DDD* em um contexto de uma empresa anonimizada onde já existe uma arquitetura de microsserviços implantada. Identificam-se pontos de melhoria na decomposição do software tanto no nível de modelagem de *design* quanto na sua aderência ao contexto de negócio.

A pesquisa também tem potencial para oferecer benefícios práticos a empresas que enfrentam problemas semelhantes, ao mostrar como o uso de *DDD* pode ser empregado para corrigir uma migração mal estruturada.

1.5 Método de Pesquisa

Para realizar a confecção deste projeto, foi utilizado um método de pesquisa de Literatura Multivocal (GAROUSI, FELDERER e MÄNTYLÄ, 2019), que consiste em uma forma de Pesquisa Sistemática (PS) que, juntamente com uma revisão da literatura acadêmica (artigos científicos e materiais formais), utiliza uma pesquisa de publicações da denominada "literatura cinzenta", isto é, em postagens virtuais e vídeos, por exemplo, de autores que reconhecidamente contribuíram e contribuem para o progresso da engenharia de software.

Na revisão da literatura acadêmica, foram utilizadas as fontes de busca: *ResearchGate*, *IEEE*, *Google Scholar*, *MDPI* e *Science Direct*, para a pesquisa de artigos relevantes para o tema deste trabalho. Termos como "*microservices*", "*ddd*", "*software architecture*" foram utilizados, juntamente com combinações como "*microservices decomposition*" e "*microservices ddd*" para resultados mais focalizados para responder à questão de pesquisa.

Para a considerada literatura cinzenta, foram utilizados como fontes de busca sites de autores que estabeleceram as bases conceituais e práticas para o DDD, como Martin Fowler e Vaughn Vernon, dentre outros.

Finalmente, destaca-se que foram utilizadas publicações em formato de livro especificamente focalizadas no tema deste trabalho, como: (EVANS, 2003), (VERNON, 2016), (VERNON, 2016) e (NEWMAN, 2022).

1.6 Estrutura do Trabalho

O trabalho é composto por 5 capítulos.

O capítulo INTRODUÇÃO apresenta as motivações, objetivo, justificativas, contribuições oferecidas e o método de pesquisa utilizado para a confecção da monografia.

No capítulo REVISÃO BIBLIOGRÁFICA são abordadas as referências teóricas utilizadas, onde são destacados conceitos importantes a respeito de Arquitetura de *Software*, Microsserviços e *Domain-Driven Design*, os quais são a base para a solução dos problemas destacados nos capítulos subsequentes.

O capítulo ALGUNS PROBLEMAS NO DESIGN DE UMA ARQUITETURA DE MICROSSERVIÇOS destaca alguns problemas dentro do contexto de negócio de uma empresa fictícia, que utiliza como base implementações reais de forma anonimizada.

O capítulo APLICAÇÃO DAS TÉCNICAS DE DDD NO DESIGN ATUAL apresenta a aplicação das técnicas do DDD para resolver os problemas destacados no capítulo anterior. As soluções de *design* propostas são apresentadas por meio de diagramas *UML* com explicações fundamentadas.

Por fim, no capítulo CONSIDERAÇÕES FINAIS apresenta a conclusão do trabalho, consolidando os benefícios obtidos pela aplicação da solução proposta, suas contribuições e as algumas sugestões de trabalhos futuros.

2. REVISÃO BIBLIOGRÁFICA

O presente capítulo apresenta de forma detalhada os fundamentos teóricos utilizados como base para o desenvolvimento do trabalho apresentado nos capítulos subsequentes.

2.1 Arquitetura de Software

Sendo uma disciplina extremamente importante no mundo de desenvolvimento de software, (BASS, CLEMENTS E KAZMAN, 2021) descrevem a arquitetura de software como a estrutura ou estruturas de um sistema, composta por elementos de software, suas relações, e as propriedades de ambos. Esta definição inclui não apenas a organização estrutural do sistema, mas também as decisões de design que afetam diretamente os atributos de qualidade.

(BASS, CLEMENTS E KAZMAN, 2021) destacam, também, a importância de uma estrutura de arquitetura e como ela pode prover esclarecimentos dado o poder analítico que carrega consigo. A partir desse ponto de vista, é possível compreender que quando uma arquitetura é construída de forma apropriada em relação ao problema que o software está se propondo a resolver, ela também torna explícitos os atributos de qualidade inerentes à estrutura utilizada.

Além de aspectos de qualidade por si só, a arquitetura de software também possibilita enxergar *trade-offs* que podem ajudar nas decisões arquitetônicas a serem tomadas para determinado produto. Como descrito por (FOWLER, 2019 - tradução livre), "*uma arquitetura mal realizada resulta no crescimento de "sujeiras" - elementos no software que impedem um claro entendimento pelos desenvolvedores de software*".

E levando em conta o aspecto de que, segundo (BASS, CLEMENTS E KAZMAN, 2021), não há existe algo como uma arquitetura inerentemente boa ou ruim, mas que deve ser construída para se adequar a um propósito, é possível enxergar a relação com a citação de Fowler (2019), pensando que uma arquitetura que produz "sujeiras" provavelmente não é uma arquitetura ideal para determinada solução.

2.2 Arquitetura de Microsserviços

2.2.1 Definições

(NEWMAN, 2022) descreve microsserviços como serviços que podem ser implantados de forma independente, sendo modelados com base em um domínio de negócio. Tais serviços podem apresentar funcionalidades diversas e independentes, encapsulando a complexidade de cada uma delas e disponibilizando-as por pontos de entradas específicos.

Microsserviços aderem ao conceito de ocultação de informações. Segundo (PARNAS, 1972), tal conceito visa ocultar o máximo possível de informações dentro de um componente e expor o mínimo de informações por meio de uma interface externa.

Diferente de uma arquitetura orientada a serviços (*Service-Oriented Architecture - SOA*), que busca utilizar uma abordagem com diversos serviços em colaboração em busca de oferecer um conjunto final de recursos (NEWMAN, 2022), conceitualmente os microsserviços favorecem um fraco acoplamento entre si por meio de fronteiras bem definidas de responsabilidades, possibilitando uma alta coesão de suas responsabilidades.

Esse mesmo autor sintetiza os conceitos de coesão e acoplamento, e sua relação recíproca, da perspectiva de microsserviços. Uma coesão forte é alcançada quando os comportamentos relacionados estão centralizados em um único módulo ou serviço, minimizando a comunicação entre fronteiras. Em sistemas baseados em microsserviços, isso significa que cada serviço deve encapsular totalmente suas responsabilidades de negócio, reduzindo interdependências entre serviços e evitando a fragmentação de funcionalidades relacionadas. Por outro lado, há uma forte relação entre coesão e acoplamento, conceitos que, embora distintos, se complementam. Enquanto a coesão descreve o grau de relacionamento entre os elementos dentro de uma mesma fronteira, o acoplamento focaliza o relacionamento entre elementos que atravessam essas fronteiras, conectando diferentes módulos ou serviços.

2.2.2 Características de Microsserviços

Microsserviços trazem uma série de características consigo, que devem ser levadas em conta ao se construir uma arquitetura baseada em microsserviços. Segundo (BASS, CLEMENTS E KAZMAN, 2021), uma arquitetura não pode ser considerada inerentemente boa ou ruim, e os aspectos específicos de microsserviços são ferramentas importantes de análise para determinar se sua utilização é adequada ou não. Em outras palavras, uma arquitetura de microsserviços não deve ser levada como uma bala de prata, e suas características devem ser cuidadosamente avaliadas.

(NEWMAN, 2022) descreve alguns conceitos essenciais dos microsserviços:

- **Implantações independentes**

  A ideia de implantações independentes é garantir que, ao realizar uma alteração em um microsserviço, implantá-lo e disponibilizar a alteração para os usuários que o consomem não implica a implantação de outro microsserviço. Para garantir esse objetivo, é necessário garantir que os microsserviços possuam baixo acoplamento.

- **Responsáveis pelo próprio estado**

  Os microsserviços devem conter todas as responsabilidades pelo acesso de seus dados por consumidores externos, por exemplo, possuindo bancos de dados específicos para cada um. Para isso, devem trabalhar com o conceito de ocultação de seu estado interno. A vantagem desta característica é a redução do acoplamento do microsserviço e o consequente aumento de sua coesão.

- **Tamanho**

  O autor descreve que esse é um aspecto muito contextual, e que não deve ser medido de forma quantitativa, como o número de linhas de código, por exemplo. Seu tamanho e complexidade varia também de acordo com o conhecimento do programador em relação ao sistema. (RICHARDSON, 2018) sugere que um microsserviço deve ser grande o suficiente para encapsular uma unidade de negócio significativa, e pequeno o suficiente para ser desenvolvido e gerenciado por uma pequena equipe autônoma.

- **Flexibilidade**

  Pensando na ideia de se precaver de problemas que possam surgir no futuro, a flexibilidade é uma importante característica de uma arquitetura de microsserviços,

considerada em diferentes eixos de flexibilidade, como organizacional (no incentivo à descentralização de equipes de desenvolvimento, alinhadas aos domínios de negócios), técnico (na possibilidade de escolher tecnologias que se adequem com as necessidades de cada serviço), escala e robustez (na possibilidade de escalar apenas serviços de acordo com suas necessidades).

- **Modelagem com base em um domínio de negócio**
  Microsserviços podem ser modelados utilizando os conhecimentos e necessidades específicas de cada contexto do negócio. Dessa forma, as equipes de desenvolvimento responsáveis por seus microsserviços possuem conhecimentos específicos sobre o domínio de negócio em que estão atuando.

(FOWLER e LEWIS, 2014) complementam as características acima com princípios para a construção de microsserviços:

- **Componentização via Serviços**
  A componentização via serviços busca estruturar sistemas como um conjunto de componentes independentes. Para os autores, um componente consiste em uma unidade de software que pode ser atualizada ou substituída de forma independente. Sendo assim, uma de suas principais vantagens é a de implantações independentes, como afirma Newman (2022), assim permitindo que mudanças nos serviços, de forma geral, não exijam a reimplantação de todo o sistema. Isso reduz o acoplamento e facilita a evolução do software, desde que os contratos de serviço sejam bem definidos e as fronteiras sejam coesas.

- **Organizados por Capacidades de Negócio**
  A fim de evitar silos organizacionais, como ocorre quando times se organizam por meio de camadas tecnologias, organizar times por meio de suas responsabilidades (ou, capacidades de negócio) tem papel fundamental em uma arquitetura de microsserviços. Desta forma, os microsserviços passam a possuir fronteiras mais claras em relação ao negócio, promovendo times multifuncionais e independência operacional.

- **Serviços Inteligentes, conexões simples**

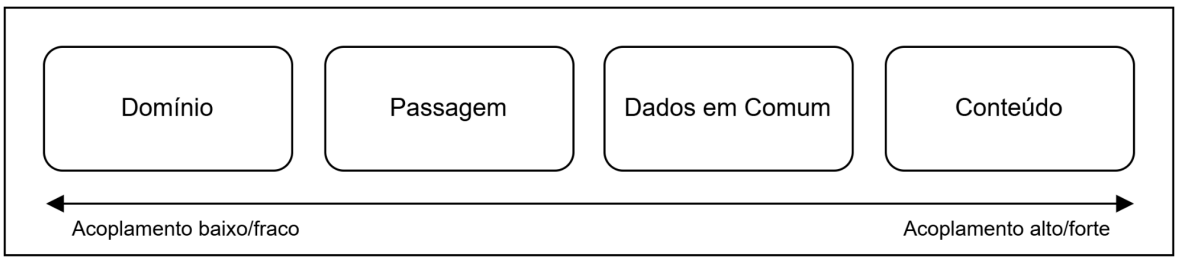
Este conceito reforça a ideia de que microsserviços devem concentrar suas respectivas lógicas de negócio e processamento de domínio, sendo o mais desacoplados e coesos possível. Portanto, suas conexões devem ser feitas de forma leve e simples. Os protocolos mais utilizados para isso são o HTTP e mensageria, utilizando ferramentas básicas como *RabbitMQ* ou *ZeroMQ*, que funcionam como roteadores assíncronos.
Os serviços agem como filtros que processam requisições, aplicam lógica e produzem respostas. Este processo auxilia no desacoplamento e na coesão dos serviços, evitando que comunicações excessivamente complexas sejam realizadas.

- ***Design* Evolutivo**
  No *design* evolutivo, a decomposição de serviços possui um papel importante para permitir que os desenvolvedores controlem as mudanças em suas aplicações sem desacelerar o ritmo de desenvolvimento, permitindo que tais mudanças sejam realizadas de forma frequente, rápida e controlada. Um exemplo em que este princípio pode ser usado é em aplicações financeiras, em que novos serviços são criados para oportunidades de mercado de curta duração.

2.2.3 Tipos de Acoplamento

Levando em conta que o acoplamento é uma característica esperada em uma arquitetura de microsserviços, é crucial entender os diferentes tipos de acoplamento para mitigar os riscos associados. (NEWMAN, 2022) descreve quatro tipos de acoplamento, organizados do nível mais baixo (desejável) para o mais alto (indesejável), como mostrado na Figura 1.

Figura 1 - Visão resumida dos tipos de acoplamento



Fonte: Newman (2021), adaptado pelo autor.

No nível mais fraco, o Acoplamento de Domínio ocorre quando um microsserviço depende das funcionalidades de outro para realizar suas operações, o que é quase inevitável, pois os microsserviços colaboram entre si.

O Acoplamento de Passagem, no segundo nível, surge quando um serviço envia dados para outro apenas porque esses dados serão usados por um terceiro serviço subsequente. Esse tipo de acoplamento cria dependências desnecessárias, exigindo que o serviço original compreenda detalhes da lógica de serviços subsequentes.

O terceiro nível de Acoplamento de Dados em Comum se dá quando dois ou mais microsserviços fazem uso de dados em comum, por exemplo, utilizando um banco de dados compartilhado.

Por fim, o Acoplamento de Conteúdo, o mais indesejável, ocorre quando um serviço acessa diretamente o estado interno de outro, violando o princípio de encapsulamento. Isso pode levar a falhas imprevisíveis e a um alto grau de dependência.

2.3 Projeto Dirigido pelo Domínio (*Domain-Driven Design*)

Uma característica importante para a construção de microsserviços é a identificação de suas fronteiras, e uma maneira de obter isso é organizá-los por suas capacidades de negócio, como descrito na seção 2.2.2. (NEWMAN, 2022) complementa essa visão ao utilizar como principal método de identificação de fronteiras o uso do próprio domínio como base, fazendo uso do *DDD*.

(FOWLER, 2020) descreve *DDD* como uma abordagem de desenvolvimento de software que foca na criação de um modelo de domínio que possua um entendimento profundo dos processos e regras do domínio ao qual o sistema está relacionado.

O termo foi inicialmente criado por Eric Evans (2003), o qual apresenta um conjunto de abordagens e técnicas que auxiliam a tarefa de tomar decisões de *design*. De acordo com o autor, quando a complexidade foge do controle, os desenvolvedores não podem compreender

claramente o sistema para alterá-lo de maneira fácil e segura. Para ele, a maior complexidade se dá no próprio domínio de negócio. Com isso, o *DDD* apresenta um conjunto de práticas, técnicas e princípios de *design*, para obter uma aceleração de projetos de software que possuem domínios complexos.

(VERNON, 2013) descreve as diversas vantagens ao se utilizar as técnicas de *DDD*:

- **A empresa ganha modelos úteis de seu domínio**

  Aplicar o *DDD* visa focalizar o aspecto mais importante da empresa, seu domínio nuclear (*core*), essencial. Apesar de outros domínios importantes existirem, eles servem para apoiar o domínio nuclear, o qual gera valor e diferencial de mercado para a empresa. Sendo assim, por meio do *DDD*, é possível construir modelos que realmente representem as capacidades de negócio da empresa.

- **O modelo de negócio é entendido de uma forma refinada e precisa**

  Ao se usar a Linguagem Ubíqua no domínio nuclear da empresa, é possível não apenas melhorar o desenvolvimento de software, mas refinar a definição e o entendimento da própria organização sobre seu negócio e missão. À medida que o modelo de domínio é refinado, surge um entendimento mais profundo do negócio, que pode ser utilizado como uma ferramenta analítica estratégica e tática.

  A colaboração entre os especialistas de domínio e times técnicos auxilia na análise de valor das direções presentes e futuras da empresa.

- **Especialistas de domínio contribuem para o desenvolvimento de software**

  Os chamados especialistas de domínio podem discordar sobre terminologias, dado suas diversas experiências sobre o negócio. Porém, o uso do *DDD* permite que eles ganhem um consenso sobre os conceitos, o que fortalece a empresa como um todo.

  Os desenvolvedores também passam a possuir um conhecimento de negócio em comum com os especialistas de domínio com quem trabalham. Dessa forma, consegue-se extrair diversos benefícios, como a redução de casos nos quais apenas algumas pessoas conhecem profundamente sobre determinado domínio.

Ao final, especialistas de domínios e desenvolvedores (antigos e novos) compartilham um conhecimento em comum, tendo como objetivo adotar uma linguagem compartilhada com qualquer pessoa da organização.

- **Fronteiras bem definidas são alocadas ao redor de modelos específicos**
  Os times técnicos são desencorajados a tomar decisões baseados em decisões puramente técnicas, visando direcionar o foco para a eficácia da solução, concentrando os esforços onde eles são mais relevantes do ponto de vista do negócio. Esse propósito está ligado à compreensão do Contexto Delimitado do projeto, de forma a garantir que o desenvolvimento permaneça alinhado com os objetivos estratégicos da empresa.

- **A arquitetura empresarial é melhor organizada**
  Quando os Contextos Delimitados são bem desenhados e entendidos, todos os times interessados sabem onde e o porquê cada integração é necessária. Suas fronteiras e relacionamentos entre si são bem definidos. Times que implementam módulos compartilhados realizam Mapeamento de Contextos para definir estratégias formais de integração entre eles. Ao fim, pode ser possível ter uma compreensão geral de toda a estrutura organizacional de comunicação da empresa.

- **Novas ferramentas, estratégicas e táticas, são adotadas**
  Os Contextos Delimitados estabelecem limites claros ao se modelar soluções dentro de um domínio de negócio. Dentro deste contexto, uma equipe desenvolve uma Linguagem Ubíqua, que será usada na comunicação e no modelo do software. Para formalizar o estilo de comunicação entre os Contextos Delimitados, diferentes equipes criam Mapeamentos de Contexto, segregando estrategicamente os contextos. Ao final, ferramentas de modelagem tática são aplicadas dentro de cada Contexto Delimitado, como Agregados, Entidades, Objetos de Valor e Eventos de Domínio.

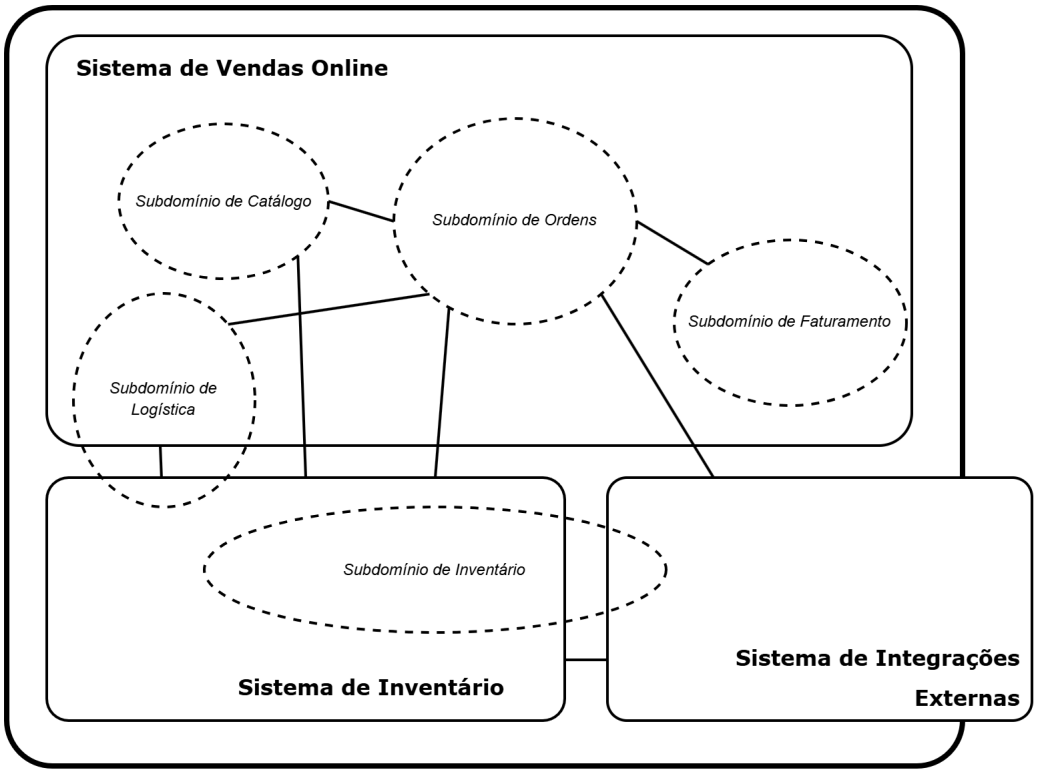
2.3.2 Representação do Domínio

Para (VERNON, 2013), o domínio pode ser definido como “o que” uma empresa faz e em qual contexto ela está inserida, onde cada empresa possui seu conhecimento associado e sua forma de fazer as coisas. Para (EVANS, 2003), a representação do domínio é central para

o desenvolvimento de software alinhado aos objetivos do negócio. Ele defende que o domínio deve ser representado por um modelo claro e coeso, que reflete a realidade do negócio. Para isso, ele apresenta a ideia da utilização de modelos. Para ele, um modelo de domínio não se trata de um diagrama específico ou o conhecimento de um especialista de domínio, mas uma abstração rigorosamente organizada e seletiva daquele conhecimento, fazendo com que as suas informações possuam um sentido.

No entanto, o termo "modelo de domínio" pode ser confuso dado a palavra "domínio", podendo se referir tanto ao domínio principal do negócio, quanto à apenas uma área de atuação dentro dessa empresa. (VERNON, 2013) utiliza termos como subdomínio, domínio principal, entre outros, para se referir a áreas de negócio específicas. Segundo ele, praticamente todos os domínios possuem subdomínios associados. A Figura 3 é uma representação dos subdomínios de uma empresa de comércio eletrônico, destacando as divisões específicas que cada subdomínio representa. A linha contínua externa, com maior espessura, indica o domínio principal de atuação da empresa; as linhas contínuas internas demarcam seus contextos delimitados; as linhas pontilhadas representam seus subdomínios; e as linhas contínuas entre os subdomínios e contextos delimitados indicam a integração de relacionamento entre eles.

Figura 2 – Representação de subdomínios em uma empresa de comércio eletrônico



Fonte: Vernon (2013), adaptado pelo autor.

2.3.3 Design Estratégico

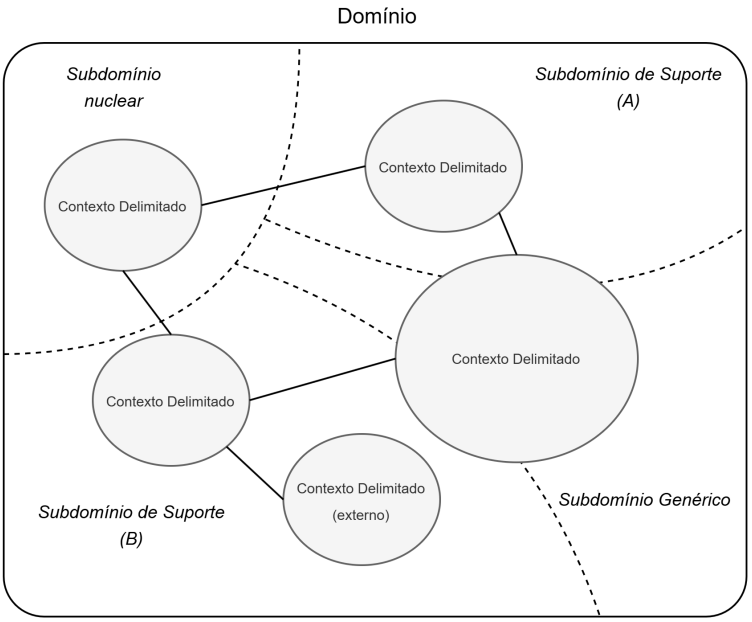
Segundo (VERNON, 2016), esse tipo de *design* destaca o que é estrategicamente importante para o negócio, apresentando forma de dividir o trabalho por importância e como integrar melhor de acordo com a necessidade do negócio. Ele ainda destaca que o design estratégico é utilizado como uma forma de visão ampla, antes de entrar nos detalhes da implementação.

(KHONONOV, 2022) complementa essa visão destacando que a visão estratégica do DDD se preocupa em responder às questões sobre qual software está sendo construído e o por que está sendo construído.

2.3.3.1 Subdomínio

Subdomínios podem ser considerados como partes menores do domínio principal de atuação de uma empresa que, em conjunto, são utilizados para atingir os objetivos e metas de seu domínio de negócio principal (KHONONOV, 2022). De acordo com (VERNON, 2016), os subdomínios podem ser utilizados para dividir todo o domínio de negócio de uma maneira lógica, facilitando o entendimento do espaço do problema em um projeto grande e complexo, por exemplo. A Figura 5 demonstra a divisão de um domínio de negócio abstrato em quatro subdomínios, seus respectivos contextos delimitados e o mapeamento de suas integrações pelas linhas contínuas entre eles.

Figura 3 – Domínio de negócio abstrato dividido em subdomínios



Fonte: VERNON (2016), adaptado pelo autor.

Quanto aos tipos possíveis de subdomínio, eles caracterizados no *DDD* de acordo com sua importância e relevância de negócio, sendo:

- **Subdomínio Nuclear (*core*)**

Representa a principal atividade da empresa, a qual gera o seu diferencial de mercado e valor de negócio. Nele, acontecem os principais investimentos em profissionais e recursos tecnológicos, de forma mais cuidadosa e estratégica.

- **Subdomínio de Suporte**

  Como o nome sugere, são subdomínios que oferecem suporte aos subdomínios principais da empresa, mas que não geram um diferencial de negócio ou vantagem competitiva. (KHONONOV, 2022) complementa este conceito destacando que uma característica singular de subdomínios de suporte se dão por sua lógica de negócio que, no geral, devem ser simples.

- **Subdomínio Genérico**

  O subdomínio genérico, por sua vez, representa um tipo de subdomínio que, além de não gerar um diferencial competitivo para o negócio, também consiste em atividades comerciais amplamente conhecidas e utilizadas. Esse tipo de subdomínio pode ser terceirizado, ou até mesmo construído internamente, mas sem um grande investimento, como é feito em um subdomínio principal, por exemplo.

2.3.3.2 Contextos Delimitados e Linguagem Ubíqua

De acordo com (VERNON, 2016), um CD - Contexto Delimitado (*Bounded Context*) é definido por um limite conceitual semântico, onde cada componente interno é semanticamente motivado e possui um significado claro, alinhado com as necessidades de negócio.

O modelo criado nas discussões de *design* deve refletir a linguagem falada pelo time que trabalha no Contexto Delimitado, a qual é chamada de Linguagem Ubíqua (*Ubiquitous Language*), e deve ser essa a linguagem usada na implementação do modelo do software. (VERNON, 2016) reforça que tal linguagem deve ser estrita, exata, rigorosa e rígida. (KHONONOV, 2022) complementa dizendo que a linguagem não deve ser ubíqua no sentido de ser utilizada de maneira universal em toda a organização mas, sim, apenas dentro dos limites de seu CD.

2.3.3.3 Mapeamento de Contextos

Considerando que diferentes Contextos Delimitados de uma empresa precisarão se comunicar de alguma forma, há também a técnica de Mapeamento de Contextos, os quais são representações visuais dos Contextos Delimitados de um sistema e a integração entre eles (KHONONOV, 2022).

(VERNON, 2021) destaca também que, acima de tudo, o Mapeamento de Contextos expressa qual tipo de relacionamento entre equipes e de integração entre contextos é representado pela linha entre os contextos. Ele ainda complementa que uma definição clara de suas fronteiras e contratos ajudam a criar mudanças controladas com o passar do tempo.

Há alguns estilos na representação dos mapeamentos, tanto de equipes como de contratos de comunicação entre CDs (mapeamento técnico), alguns deles representando ambos. Segue suas descrições resumidas (VERNON, 2021) e (KHONONOV, 2022):

- **Parceria**: Quando duas equipes trabalham em sincronia tendo um objetivo em comum entre os dois. Com essa integração, nenhuma das duas dita as regras de definição de contrato. Normalmente, uma parceria não dura por um longo prazo, mas apenas enquanto fornece vantagens para o negócio, podendo ser remapeada posteriormente.

- **Cliente-Fornecedor**: O estilo Cliente-Fornecedor, ao contrário da Parceria, descreve um padrão onde ambas as equipes podem possuir sucesso de forma independente um do outro, porém há uma relação de influência entre ambos. Isso é representado pela letra D (*Downstream* – a jusante) para o cliente e U (*Upstream* – a montante) para o fornecedor, denotando que o cliente planeja com o fornecedor para atender suas expectativas, mas o fornecedor decidirá o como e o que o cliente irá receber.

- **Conformista (CF)**: Quando há um desequilíbrio de poder entre as equipes, de forma que a equipe ascendente não possui motivação para atender as necessidades da equipe descendente. Caso a equipe descendente deseje se adequar às informações recebidas pela equipe ascendente, essa é uma relação de conformidade.

- **Camada de Anticorrupção (ACL)**: Continuando com a relação de desequilíbrio de poder decisório sobre os modelos de domínio, há casos em que a equipe descendente não deseja se adequar à equipe ascendente, seja pelo modelo ascendente ser ineficaz para as necessidades do descendente, ou pelo descendente desejar proteger seu conhecimento de domínio e/ou linguagem ubíqua. Nestes casos, o modelo descendente constroi um modelo que traduz os contextos externos por meio de uma camada de anticorrupção.

- **Serviço de Host Aberto (OHS)**: O Serviço de Host Aberto visa disponibilizar para os CDs descendentes uma interface aberta, a qual expõe um protocolo conveniente bem documentado para seus clientes.

2.3.4 Design Tático

Em continuidade ao projeto estratégico, que traz uma visão em alto nível, o projeto tático visa aplicar os conceitos obtidos em soluções de software, de forma a criar um modelo de *design* que seja fiel ao domínio, coeso e fácil de manter (EVANS, 2014), utilizando ferramentas como modelagem de Entidades, Objetos de Valor, Agregados, Repositórios, Serviços de Aplicação, Serviços de Domínio e Eventos de Domínio.

2.3.4.1 Entidade

Entidades são caracterizadas por modelos que representam um conceito do domínio que, mesmo sofrendo alterações de estado ao longo do tempo, mantêm sua identidade, sendo este seu principal ponto de distinção de outras ferramentas de modelagem – sua individualidade (VERNON, 2021).

(EVANS, 2004) complementa essa visão afirmando que uma entidade deve refletir apenas os comportamentos do conceito e os atributos necessários para dar suporte a eles.

2.3.4.2 Objeto de Valor

Um Objeto de Valor é um elemento que modela uma totalidade conceitual imutável. Essa ferramenta de modelagem não possui uma identidade única, e sua equivalência é

determinada apenas ao comparar os atributos encapsulados nela pelo seu tipo. Objetos de Valor são comumente utilizados para descrever, quantificar ou mensurar uma Entidade (VERNON, 2021).

2.3.4.3 Agregado

Agregados são ferramentas fundamentais na modelagem tática de um domínio que buscam encapsular toda a complexidade da criação de modelos por meio de Entidades e Objetos de Valor, estabelecendo um limite claro ao seu redor (EVANS, 2015). Cada agregado deve possuir uma entidade raiz (chamada de *raiz do agregado*), a qual deve ser o único ponto de acesso ao modelo criado. Essa estrutura garante que a raiz do agregado mantenha as propriedades e invariantes do agregado como um todo, simplificando o controle de consistência dos objetos relacionados.

(VERNON, 2021) descreve quatro ideias que servem como orientação ao se modelar agregados:

1. Proteger invariantes de negócios dentro dos limites do agregado;
2. Projetar agregados pequenos;
3. Referenciar outros agregados apenas pela identidade;
4. Atualizar outros agregados utilizando a consistência eventual.

2.3.4.4 Evento de Domínio

Eventos de domínio são objetos completos no modelo do domínio que funcionam como uma representação de uma ação que já ocorreu, e que é de interesse dos especialistas de domínio. Especialmente em sistemas distribuídos, em que o ciclo de vida de uma entidade pode ser alterado de formas assíncronas, esses eventos são úteis para tornar explícita a intenção de mudança de uma entidade, ajudando a compreender como o estado do sistema evoluiu (EVANS, 2015).

2.3.4.5 Serviço de Domínio

(EVANS, 2015) define que, quando um processo significante ou uma alteração no modelo de domínio não é uma responsabilidade natural de uma entidade ou objeto de valor,

uma operação deve ser criada para o modelo por meio de uma interface isolada declarada como um serviço. (VERNON, 2015) complementa essa visão, indicando que os serviços de domínio devem ser usados apenas quando realmente necessários, quando a lógica a ser encapsulada não se encaixa em outros componentes. Ele ainda sugere que os serviços sejam coesos e centralizem processos complexos ou cálculos que envolvam múltiplas entidades ou agregados.

2.3.4.6 Serviço de Aplicação

Serviços de Aplicação são comumente encontrados em projetos de *software* que possuem uma arquitetura dividida em camadas, as quais podem representar tanto separações físicas como lógicas. (EVANS, 2003) atribui à camada de aplicação a função de isolar as responsabilidades lógicas e de domínio. Para ele, os serviços de domínio são classes que agem como coordenadoras das requisições feitas pelo usuário. (FOWLER, 2009) destaca que a camada de serviço – a qual é um sinônimo da camada de aplicação – define o limite de uma aplicação com uma camada de serviços que estabelece um conjunto de operações disponíveis e coordena a resposta da aplicação em cada operação.

3. ALGUNS PROBLEMAS NO *DESIGN* DE UMA ARQUITETURA DE MICROSSERVIÇOS

Este capítulo apresenta certos problemas que podem ser encontrados na implementação de uma arquitetura de microsserviços, em um cenário onde a modelagem de suas das interações não foi realizada de maneira adequada, do ponto de vista de definição de fronteiras e responsabilidades claras de cada serviço, detalhando alguns dos problemas decorrentes de seu *design*. O capítulo também apresenta pontualmente as técnicas de *DDD* utilizadas como base para propor uma solução que busque minimizar os problemas encontrados.

3.1 Contexto do Negócio

Como descrito na seção 2.2.2, a arquitetura distribuída de microsserviços traz diversas vantagens para o ambiente de software corporativo, como maior escalabilidade e flexibilidade no desenvolvimento e manutenção de suas aplicações. No entanto, quando essa arquitetura ocorre sem uma modelagem cuidadosa, surgem problemas como baixo nível de coesão e alto grau de acoplamento entre os serviços, anulando muitos dos benefícios esperados.

Estudos como os de (OUMOUSSA e SAIDI, 2024) destacam que a complexidade na decomposição de sistemas monolíticos para uma arquitetura de microsserviços é um dos maiores desafios nessa transição. Mesmo com técnicas conhecidas para identificar microsserviços, faltam métricas amplamente aceitas para avaliar a qualidade das definições de serviço, o que limita a capacidade das equipes de medir e otimizar a arquitetura. (ZHONG et al., 2024) complementam, demonstrando que a definição de contextos delimitados, conceito central no *DDD*, é sujeita a interpretações diferentes, o que resulta em inconsistências no *design*.

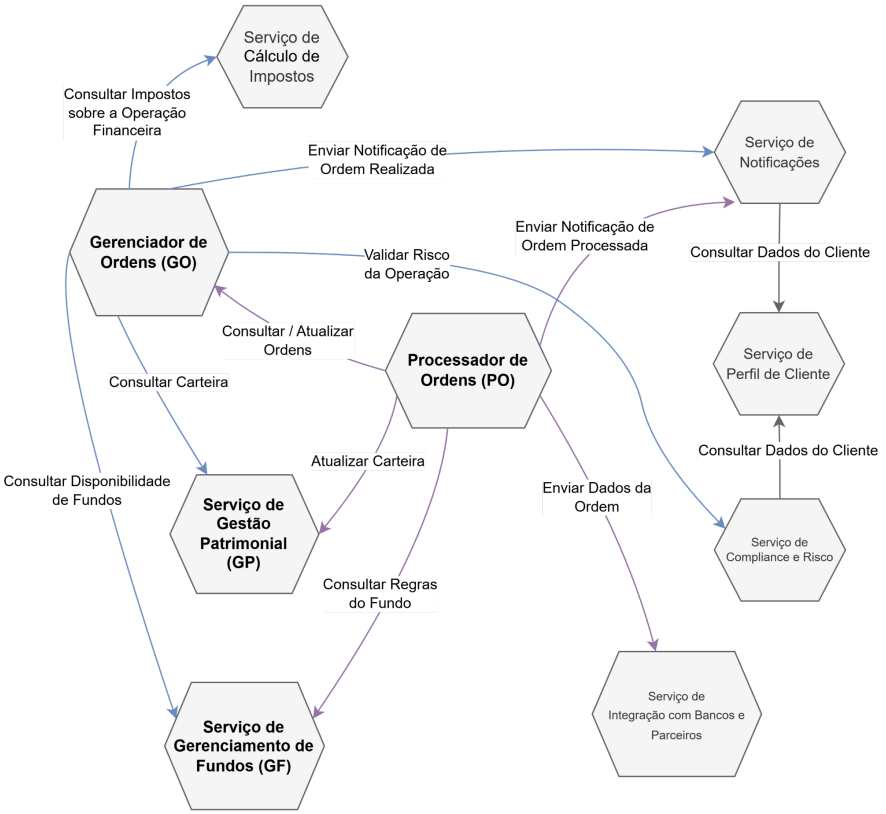
Esses desafios comprometem não apenas a evolução e a manutenção do sistema, mas também a capacidade de atender de maneira eficiente as necessidades do negócio (VERNON, 2013).

Considerando os fatores abordados anteriormente, descreve-se um cenário de uma empresa fictícia, chamada *InvestmentCorp*, calcado em um exemplo real que anonimiza e abstrai

algumas características de negócio, que opera como gestora de fundos de investimento. Sua principal meta de negócio é possibilitar que clientes consigam realizar compras e vendas de cotas em fundos de investimento por meio de um aplicativo móvel ou portal *web*. Para tanto, a *InvestmentCorp* possui capacidades de negócio que dão suporte à sua estratégia de operação principal, como administração dos fundos de investimento disponíveis, gerenciamento do patrimônio de seus clientes, realização de integrações com sistemas terceirizados, entre outras.

A fim de disponibilizar um ecossistema tecnológico que permitisse a construção de produtos resilientes e escaláveis, além de utilizar a colaboração de diferentes times especialistas em cada produto, a *InvestmentCorp* construiu sua arquitetura utilizando microsserviços. Por meio deles, a *InvestmentCorp* busca garantir que as transações financeiras realizadas pelos clientes sejam recebidas, processadas e integradas, tanto às suas carteiras quanto a organizações externas interessadas, como entidades governamentais e bancos, por exemplo. Representados pelas figuras hexagonais, a Figura 3.2 apresenta os principais microsserviços responsáveis por atender as capacidades de negócio da *InvestmentCorp*, bem como suas interações.

Figura 4 - Representação Geral dos Microsserviços do Módulo de Gerenciamento de Operações e suas Principais Interações



Fonte: o Autor.

Na figura, as setas representam as interações entre os microsserviços e seus nomes denotam a natureza das colaborações entre eles. Estão representadas as colaborações que pretendem garantir que as necessidades de negócio sejam atendidas.

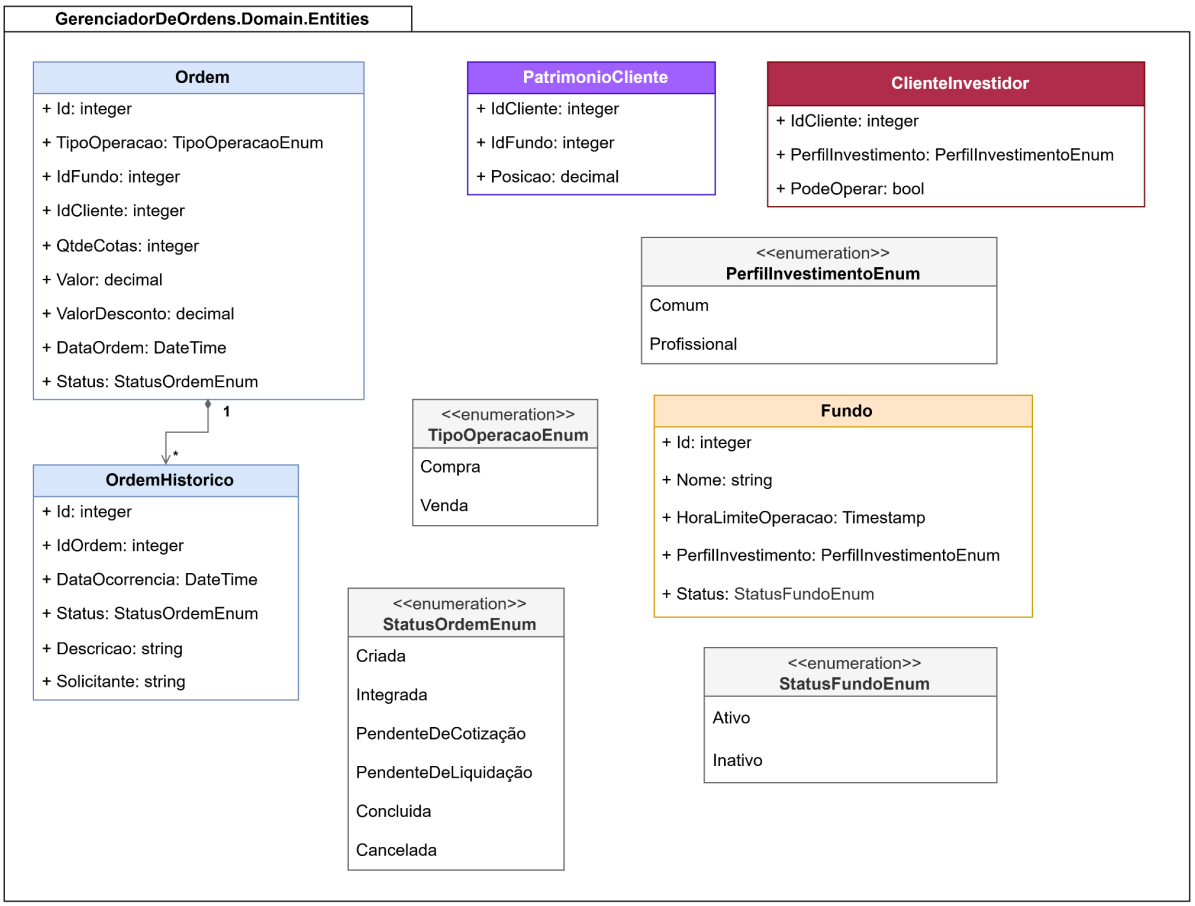
O serviço Gerenciador de Ordens (GO) tem um papel fundamental no fluxo operacional da *InvestmentCorp*. Ele serve como porta de entrada para as transações financeiras realizadas pelos clientes, além de disponibilizar as funcionalidades de entrada e consulta de ordens em diferentes canais da companhia. O GO interpreta o conceito de *Ordem* como uma entidade que representa uma operação de compra ou venda de cotas em um fundo de investimento pelo cliente.

Para possibilitar a criação de ordens, o GO se comunica com outros serviços, como o Serviço de Gestão Patrimonial (GP) e Compliance e Risco, a fim de realizar todas as validações necessárias que garantem sua consistência. Além disso, ele também consome dados do

Serviço de Cálculo de Impostos para obter a informação sobre a dedução de impostos no valor da operação financeira. Suas responsabilidades consistem em disponibilizar a consulta das ordens realizadas por meio de interfaces públicas (*APIs*) e atualização de status, funcionalidades essas que serão utilizadas posteriormente por serviços como o Processador de Ordens (PO).

O GO é responsável pela parte inicial do ciclo de vida de uma Ordem. Ela é de fato criada apenas quando a solicitação realizada por um cliente for validada com sucesso a partir de diversas etapas que compõem o fluxo de validação de uma Ordem, como a validação de risco da operação, status do Fundo de Investimento, posição disponível do cliente no momento da solicitação, entre outras. A Figura 3.3 representa, em alto nível de abstração, o modelo das principais entidades de domínio contidas no serviço GO.

Figura 5 – Modelo das classes do serviço Gerenciamento de Ordens (GO)



Fonte: o Autor.

Observa-se que a relação entre Ordem e *OrdemHistórico* reflete um aspecto importante do domínio: o rastreamento do ciclo de vida de uma ordem. Ao conectar diretamente uma Ordem ao seu histórico, eventos relevantes e mudanças de status podem ser registrados, possibilitando a rastreabilidade de seu estado.

O pacote *GerenciadorDeOrdem.Ordem*, que representa as principais responsabilidades do serviço no que tange o conceito de Ordem visto anteriormente, terá suas entidades, em sua maioria, serão instanciadas por meio do fluxo de entrada de Ordens.

O Processamento de Ordens (PO), por sua vez, é responsável por processar e integrar as novas ordens disponibilizadas por GO, tanto nas carteiras dos clientes quanto naquelas de interessados externos, como bancos, órgãos reguladores, etc. Diferente de GO, PO interpreta uma “Ordem” dentro de seu contexto específico como uma operação efetivada, representada

por uma transação financeira já integrada, que pode estar passando pelo fluxo de processamento, ou que já foi devidamente processada e integrada aos parceiros externos interessados (como órgãos regulamentadores e governamentais, por exemplo) como na posição patrimonial dos clientes.

PO tem um papel ativo, tanto para consultar as ordens realizadas em GO quanto para enviar os detalhes dessas ordens adiante no fluxo de operação, como nas integrações com serviços externos e carteiras dos clientes, por exemplo.

O serviço Gestão Patrimonial (GP) realiza o gerenciamento patrimonial dos clientes, controlando sua carteira de investimentos, saldos, e sua evolução patrimonial, com base nas configurações e regulamentações dos fundos aplicados. O GP disponibiliza a visão patrimonial para os clientes da *InvestmentCorp* por meio de diferentes canais, como aplicativo móvel e portal *web*. Nota-se também que GP atua de forma passiva em relação aos demais serviços interessados, porém possuindo suas próprias regras de negócio internas.

O Gerenciamento de Fundos (GF) também se destaca por atender um contexto muito importante da *InvestmentCorp*. Além de ser responsável por gerenciar as informações dos fundos de investimento, incluindo características como horários de operação, limites mínimos e máximos de aplicação e resgate, e regras de negócio específicas de cada fundo, também disponibiliza as informações de forma estruturada para outros serviços, como GO, e para os diferentes canais da companhia.

3.2 Ciclo de Vida de uma Ordem

Nos contextos de negócio abordados anteriormente, “Ordem” representa um dos conceitos mais importantes pois, consolidando as definições tanto em GO quanto PO, pode-se dizer aqui que Ordem é a entidade que representa uma transação financeira de compra ou venda de cotas em fundos de investimento, realizada por um cliente. Uma Ordem possui diversos estágios ao longo de seu ciclo de vida, os quais permitem que diferentes ações ou eventos sejam realizados. Por exemplo, uma Ordem de compra SOLICITADA pode ser CANCELADA, porém se estiver no estado PENDENTE DE COTIZAÇÃO isso já não é possível.

De forma a descrever as mudanças de estado de uma Ordem e suas ações possíveis, utiliza-se o diagrama de Máquina de Estado da UML, apresentado na Figura 8. A Tabela 3 descreve com detalhes cada estado e cada ação possível em cada estado.

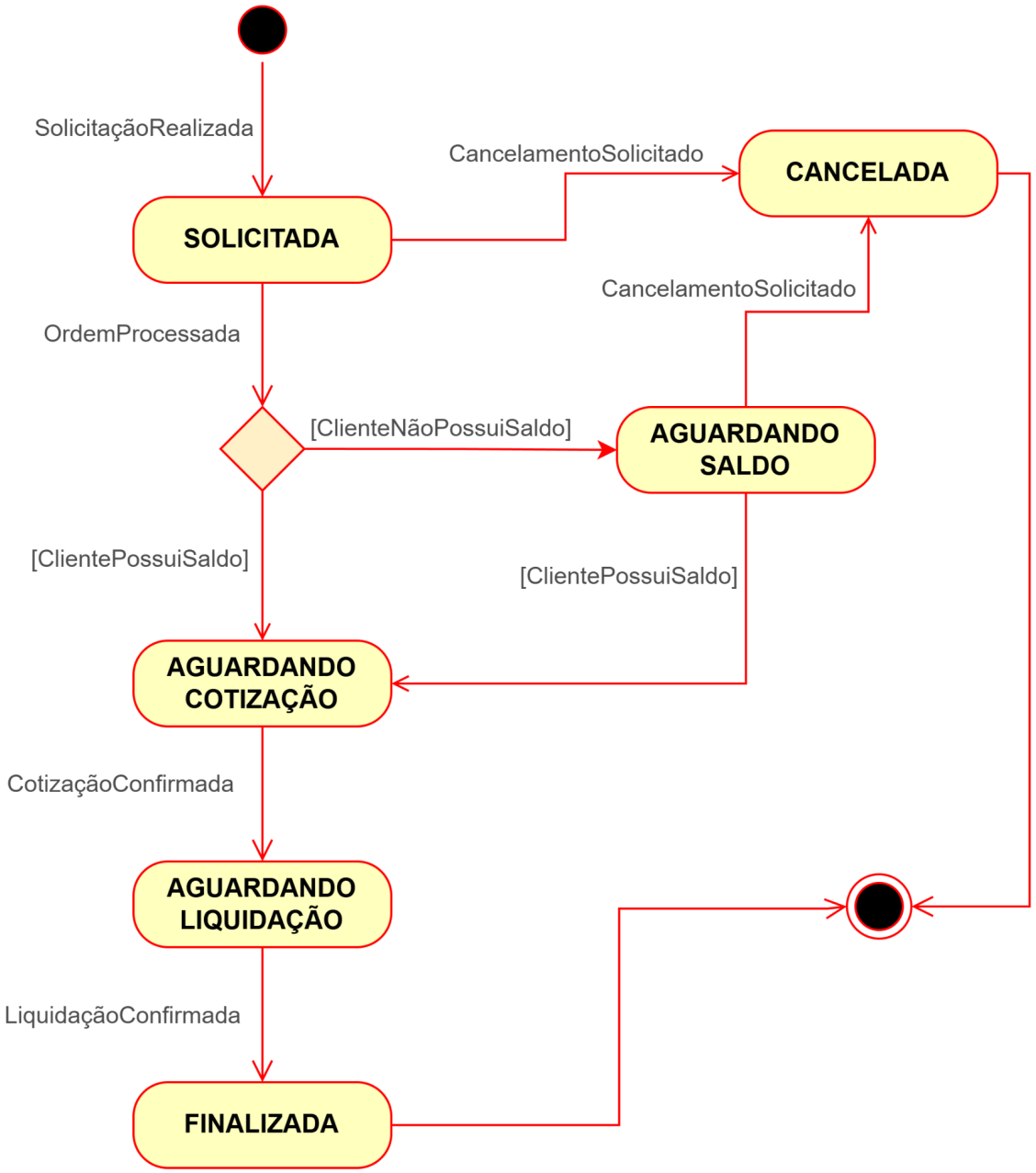
Tabela 2 – Descrição dos estados de uma Ordem.

Estado	Descrição	Ações
SOLICITADA	Estado inicial de uma ordem, atribuído assim que a solicitação realizada pelo cliente é validada e a ordem é criada. O estado se mantém assim até que a próxima janela de processamento se inicie e seu estado mude para AGUARDANDO COTIZAÇÃO ou AGUARDANDO SALDO (caso a conta do cliente esteja sem saldo), ou que haja um cancelamento e seu estado mude para CANCELADA.	Permite: Consultar saldo do Cliente, Consultar dados do Fundo e Cancelar. Bloqueia: Finalizar.
AGUARDANDO COTIZAÇÃO	Este estado é atribuído assim que ocorra uma janela de processamento e a ordem seja processada. Esse estado se mantém assim até que o prazo para cotização do regulamento do fundo seja finalizado. Caso seja uma ordem de compra, seu estado muda para FINALIZADA. Se for de venda, muda para AGUARDANDO LIQUIDAÇÃO	Permite: Consultar saldo do Cliente e Consultar dados do Fundo. Bloqueia: Finalizar, Cancelar.
AGUARDANDO LIQUIDAÇÃO	Após o prazo para liquidação de cotas previsto no regulamento do fundo finalizar, quando houver uma nova janela de processamento o estado da ordem será alterado para FINALIZADA.	Permite: Consultar saldo do Cliente e Consultar dados do Fundo. Bloqueia: Cancelar.
AGUARDANDO SALDO	Esse estado será atribuído durante uma janela	Permite: Consultar saldo

	de processamento caso o cliente não possua saldo em conta para realizar a transação. Isso pode ocorrer apenas em casos de ordens de compra onde a ordem possui uma data de agendamento. É possível alterar o estado da ordem para CANCELADA também.	do Cliente, Consultar dados do Fundo e Cancelar. Bloqueia: Finalizar.
FINALIZADA	Estado atribuído assim que todos os estados anteriores forem devidamente finalizados e a ordem seja concluída.	Bloqueia: Qualquer ação.
CANCELADA	Estado atribuído caso o cliente deseje cancelar a ordem.	Bloqueia: Qualquer ação.

Fonte: o Autor.

Figura 6 – Diagrama de estado do ciclo de vida de uma Ordem



Fonte: o Autor.

3.3 Descrição do Problema

Apesar de atenderem as necessidades de negócio da *InvestmentCorp*, é possível observar alguns pontos de atenção em relação à interação entre os microsserviços no esquema arquitetônico apresentado nas figuras acima. Ao observar a relação entre GO e PO, por exemplo, observa-se que ambos utilizam o conceito de "Ordem". No entanto, este conceito possui significados diferentes em cada serviço, apesar de ambos terem a ver com o ciclo de vida de uma Ordem, aqui entendida de forma geral como uma operação financeira realizada por um cliente. Sem uma definição precisa desse conceito, essa variação na definição indica uma coesão fraca, como apresentado na seção 2.2.3, podendo gerar ambiguidades no entendimento do domínio, prejudicar a evolução do sistema como um todo e, principalmente, deteriorar a comunicação entre as equipes.

Além disso, é observa-se que a interação entre GO e PO gera um acoplamento similar ao acoplamento de domínio, descrito no capítulo 2.2.5, pois PO precisa realizar consultas recorrentes em GO, para verificar se novas Ordens foram realizadas. Apesar de ser um tipo de acoplamento baixo, a análise deve considerar problemas de disponibilidade (caso GO esteja em falha operacional por algum motivo, PO apresentará erros no momento de consultar GO), de manutenibilidade (pois mudanças feitas em GO poderão afetar diretamente PO) e de escalabilidade, por exemplo.

Além dos problemas observados em relação à comunicação entre os serviços, pode-se observar problemas de *design* nos próprios serviços. Conforme observado no diagrama da Figura 7, os modelos em GO são agrupados por pacotes que representam os conhecimentos de domínio de cada classe. Com base nas convenções do padrão *UML*, nota-se que as apresentadas possuem acesso "público" às suas propriedades (representado pelo símbolo "+" que precede o nome das propriedades). Isso significa que não há o conceito de modularização, pois todas as classes podem ser acessadas por classes externas, o que implica problemas de manipulação indesejada no estado de tais entidades.

Entidades como *FundoInvestimento* e *ClienteInvestidor*, representadas em pacotes separados, não possuem um acoplamento direto com Ordem. Isso reflete o papel dessas entidades no contexto do GO, as quais são utilizadas apenas em consultas ou validações específicas, por

meio de integrações com serviços externos. Contudo, esse tipo de integração pode criar problemas de coesão, pois o GO acaba "conhecendo" indiretamente conceitos específicos de serviços externos, como os representados por *FundoInvestimento* e *PatrimonioCliente*.

Um exemplo disso é observado com a entidade *PatrimonioCliente*, no que diz respeito ao conceito de Posição. Dentro do serviço GO, *Posição* representa não apenas a posição financeira atual do cliente em um fundo, mas também incorpora o impacto de ordens pendentes de processamento, por exemplo. Essa definição, no entanto, diverge do conceito de Posição mantido pelo serviço externo GP, resultando em um problema de baixa coesão.

No geral, destacam-se diversos problemas que vão desde baixa coesão e alto acoplamento, até decisões de *design* consideradas inadequadas. A Tabela 4 consolida os principais problemas que surgem a partir da interação entre GO e os demais serviços, bem como os impactos encontrados no *design* atual deste serviço. Além do identificador do problema, há também sua descrição e tipo de impacto, os quais serão referenciados no capítulo 4 com suas respectivas soluções.

Tabela 3 – Compilação dos problemas do *design* atual.

Identificador	Descrição	Tipo de impacto
GO-01	O conceito de "Ordem" está distribuído em GO e PO, causando ambiguidade em seu entendimento.	Baixa coesão
GO-02	O conceito de "Posição" em GO é diferente do conceito de "Posição" em GF, causando ambiguidade em seu entendimento.	Baixa coesão
GO-03	Os pacotes *GerenciadorDeOrdem.FundoInvestimento* e *GerenciadorDeOrdem.Cliente* não isolam adequadamente os conceitos internos de GO em relação aos conceitos dos serviços externos.	Baixa coesão
GO-04	GO e PO possuem comunicação direta com diferentes microsserviços externos e, por compartilhar o conceito de	Fronteira semântica mal definida

	ordem, suas fronteiras não são suficientemente claras.	
GO-05	Apesar de possuir um acoplamento de domínio (conceitualmente baixo), GO é passivo em relação à PO, possibilitando, por exemplo, que falhas operacionais em PO provoquem efeitos colaterais indesejados no ciclo de vida de Ordem.	Acoplamento inadequado
GO-06	As propriedades de todas as entidades em GO possuem acesso público, possibilitando alterações indesejadas em seus estados, além de trazerem mais complexidade ao controle de seu estado.	Modularização inexistente
GO-07	GO possui um serviço de aplicação para realizar a coordenação de chamadas à serviços externos além de centralizar validações de domínio, prejudicando a manutenção e reduzindo a coesão.	Baixa coesão

Fonte: o Autor.

4. APLICAÇÃO DAS TÉCNICAS DE *DDD* NO *DESIGN* ATUAL

Este capítulo discute as propostas de solução para os problemas destacados no capítulo 3 da perspectiva do DDD.

4.1 Visão geral das soluções propostas

Considerando o cenário atual da arquitetura de microsserviços da *InvestmentCorp*, analisado na seção anterior, constata-se que diversos problemas emergem a partir do *design* adotado. Embora a arquitetura atenda aos requisitos de negócio, ela apresenta problemas significativos, como alto acoplamento entre os serviços e baixa coesão dos microsserviços. Esses problemas impactam negativamente na evolução e manutenção do software, comprometendo inclusive princípios fundamentais para a adoção de microsserviços, discutidos na seção 2.2.2.

As soluções propostas visam reestruturar parte do *design* com base nos conceitos e técnicas do *DDD*, detalhados na seção 2.3. O objetivo principal é criar uma arquitetura cujos elementos apresentam alta coesão funcional, com fronteiras e responsabilidades bem definidas, alinhadas com as delimitações dos subdomínios. Procura-se aqui abordar os problemas identificados de duas formas:

- Diretamente, por meio da aplicação de práticas de *DDD* no *design* atual sob uma perspectiva estratégica, destacando os principais subdomínios, seus respectivos tipos (nuclear suporte, genérico), os contextos delimitados, as fronteiras entre os microsserviços e os tipos de comunicação de suas interações. Além disso, avalia-se da perspectiva tática os principais módulos causadores de problemas, propondo sugestões de melhoria.
- Indiretamente, por meio de propostas de mudanças arquitetônicas fundamentadas nos problemas evidenciados ao aplicar os conceitos do *DDD*, como a redistribuição de responsabilidades dos serviços e a introdução de mecanismos de mensageria.

Para revisar aspectos do *design* arquitetônico, sob um ponto de vista, principalmente, da comunicação entre os microsserviços, aplicam-se as diretivas do *Design* Estratégico. Quanto

ao *Design* Tático, discute-se exclusivamente neste trabalho o subdomínio Plataforma de Fundos de Investimento, por concentrar a maior parte dos problemas identificados anteriormente

4.1 *Design* Estratégico com Subdomínios e Contextos Delimitados

Na análise a seguir, são identificados os Subdomínios da empresa e definidos seus respectivos Contextos Delimitados (CD), mantendo uma correspondência de um para um (1:1) entre subdomínios e CDs. Além disso, explora-se como os microsserviços estão associados a esses Contextos Delimitados, destacando as responsabilidades e limites de cada um dentro da arquitetura da *InvestmentCorp*.

4.1.1 Identificação e Classificação dos Subdomínios

A primeira etapa na identificação dos subdomínios consiste em traçar suas fronteiras lógicas segundo as capacidades potenciais do negócio, como ponto de partida para alinhar a arquitetura às necessidades do negócio.

Como mencionado na seção 3.1, a principal linha de atuação da *InvestmentCorp* é possibilitar que clientes consigam realizar compras e vendas de cotas em fundos de investimento por meio de um aplicativo móvel ou portal *web*. Considerando essa meta de negócio e as responsabilidades atribuídas ao microsserviço GO, é possível identificar um subdomínio que circunscreve essas responsabilidades. Tal subdomínio representa de maneira geral uma plataforma para negociação de fundos de investimento, assumindo aqui que o termo "plataforma" refere-se a um sentido atual empregado corriqueiramente em boa parte das empresas: áreas do negócio e de seus sistemas que oferecem uma gama expansível de serviços. Esse subdomínio é nomeado aqui de **Plataforma de Fundos de Investimento**. Ele desempenha um papel central no diferencial competitivo da empresa, garantindo o sucesso das transações dos clientes e, consequentemente, a captação de novos recursos. Sendo este um subdomínio nuclear, ele dá suporte direto aos objetivos estratégicos da empresa.

Para sustentar o ciclo de vida de uma Ordem, o serviço PO é responsável por integrar as operações realizadas em GO aos demais sistemas necessários até o momento de sua

finalização. Considerando que esse é um processo que ocorre de maneira assíncrona em relação à entrada da ordem, o qual pode inclusive possuir intervenções manuais, define-se aqui o subdomínio de suporte ***Backoffice***, cujo papel será proteger e especializar a linguagem ubíqua já conhecida em PO, isto é, isolar seus conceitos de domínio de conceitos externos à ele, buscando garantir maior consistência conceitual, descrito com mais detalhes na próxima seção.

Levando em conta o conhecimento do serviço de Gerenciamento de Fundos de Investimento, é proposto aqui o subdomínio **Fundos de Investimento**, o qual controla informações essenciais sobre os fundos geridos pela *InvestmentCorp*, como características, regras operacionais, disponibilidade, entre outros. Esses dados são fundamentais para sustentar as transações realizadas pelos clientes, garantindo conformidade com as regras do mercado. Este também pode ser considerado também um subdomínio nuclear dentro da companhia, dado que a disponibilização de fundos estratégicos para o mercado representa diferenciais de negócio

Já o subdomínio de suporte **Patrimônio do Cliente** centraliza informações sobre a posição financeira dos clientes, incluindo saldos, extratos e evolução patrimonial, responsabilidades já apresentadas anteriormente no microsserviço de GP. Embora desempenhe um papel importante, ele não é diretamente responsável pelo diferencial competitivo da empresa, funcionando como um apoio às operações financeiras.

O subdomínio **Compliance e Risco** reúne as validações regulatórias e de risco associadas aos clientes e suas operações. Ele desempenha um papel essencial para garantir a conformidade com normas internas e externas, protegendo a integridade das transações, mas não apresenta diferencial competitivo, sendo classificado como subdomínio de suporte.

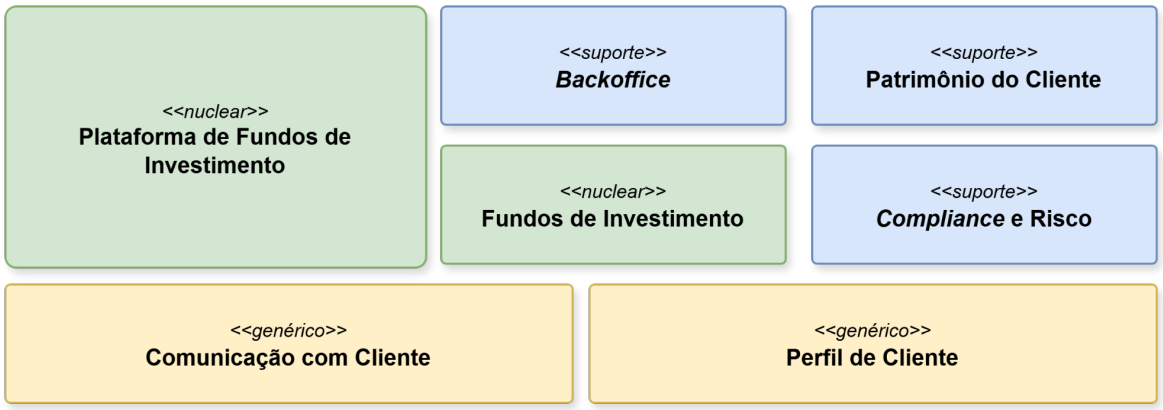
Por fim, há subdomínios que, embora sejam cruciais para garantir a experiência do cliente e o bom funcionamento do sistema de software, não representam diferenciais competitivos e, sobretudo, podem ter o suporte de produtos de prateleira configuráveis. Tais subdomínios são classificados como genéricos, como foi visto no capítulo 2. Tem-se aqui dois subdomínios desse tipo: **Perfil de Cliente**, que agrega e disponibiliza dados essenciais sobre o cliente, como informações de contato e perfil de investimento, e **Comunicação com Cliente**, que

centraliza as comunicações com os clientes, tanto em canais externos, como e-mail e telefone, quanto no envio de notificações nos canais internos, como aplicativo e portal *web*.

4.1.2 Definição dos Contextos Delimitados

Como definido acima, a relação entre subdomínios e CDs é de 1 para 1. Com base nos subdomínios identificados, a Figura 4.1 apresenta os Contextos Delimitados mapeados.

Figura 7 - Contextos Delimitados da *InvestmentCorp*



Fonte: o Autor

Os CDs estabelecem limites claros em relação às responsabilidades de cada funcionalidade, ao eliminar a ambiguidade de termos e promover maior coesão dentro dos subdomínios. Termos de mesmo nome (falsos cognatos), que antes estavam distribuídos entre diferentes serviços, agora são devidamente atribuídos a CDs que correspondem aos subdomínios que os definem univocamente nas suas respectivas linguagens ubíquas.

Por exemplo, o termo “Ordem”, anteriormente presente nos serviços GO e PO, torna-se um conceito específico dentro do CD Plataforma de Fundos de Investimento. Enquanto isso, no CD *Backoffice* (detalhado posteriormente na seção sobre o *Design* Tático), o modelo central passa a ser denominado “Operação Financeira”, com suas próprias características. Embora possa derivar características oriundas de "Ordem" (como o valor, tipo de operação, etc), a Operação Financeira é tratada como um conceito único dentro da linguagem ubíqua deste CD, sendo protegida e compreendida exclusivamente pelas equipes que trabalham neste contexto.

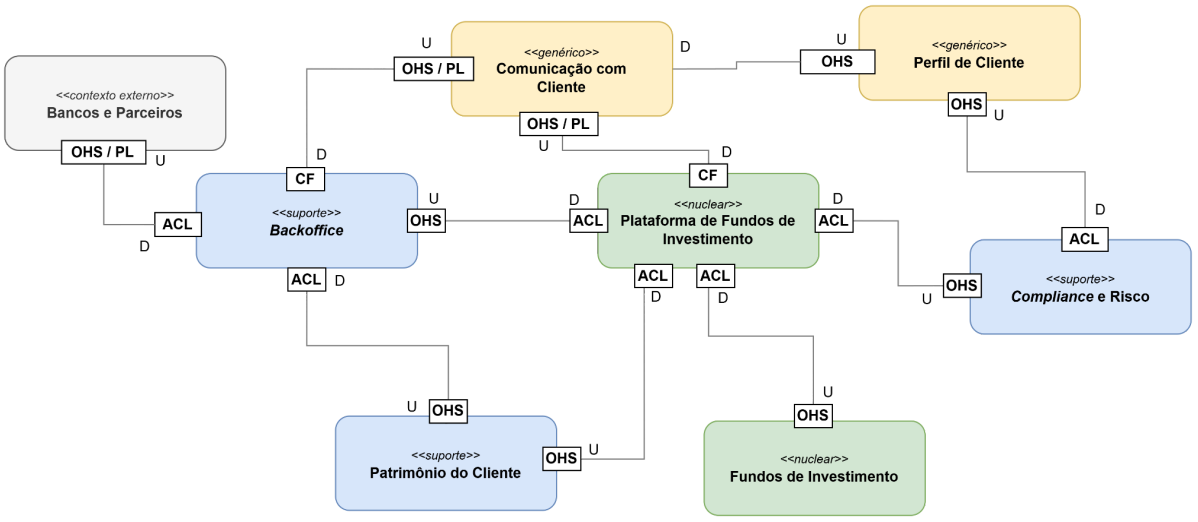
Outro exemplo é o termo “Posição”, que possui agora definições melhor alinhadas com as linguagens ubíquas de seus respectivos CDs. Em Plataforma de Fundos de Investimento, o conceito é representado como “Posição Disponível”, que reflete a posição financeira total de um cliente em determinado fundo deduzindo o valor referente às suas Ordens ainda não finalizadas. Por outro lado, no CD Patrimônio do Cliente, o conceito de Posição permanece protegido e representa exclusivamente o entendimento do que essa entidade significa dentro deste contexto.

Por fim, a identificação dos Subdomínios e Contextos Delimitados traz vantagens que ajudam a mitigar conceitualmente os problemas **GO-01** e **GO-02**, apresentados na Tabela 3, seção 3.2. Tais problemas ainda serão revisitados no nível de *design* tático posteriormente na seção 4.3.

4.2 Design Estratégico com Mapeamento de Contexto

Visando estilos de comunicação entre CDs que busquem não apenas evoluir a qualidade de comunicação entre as equipes envolvidas, por meio de definição de interfaces e contratos mais precisamente definidos, mas também diminuir de alguma forma o acoplamento entre certos CDs, a Figura 4.2 apresenta o mapeamento dos contextos delimitados propostos na seção anterior.

Figura 8 - Mapeamento dos Contextos Delimitados no Cenário Proposto



Fonte: Autor

Por se tratarem de representações de microsserviços, como visto na seção 3.1, a comunicação entre os CDs mantém o estilo de comunicação previamente utilizado entre os microsserviços. Sob o ponto de vista do DDD, utiliza-se o estilo de comunicação Serviço de *Host* Aberto (*Open Hosted Service*), representado pela sigla *OHS*. Como apresentado no capítulo 2, tal estilo de comunicação dá acesso ao seu Contexto Delimitado por meio uma interface aberta, com contrato claramente especificado, para que os interessados em seus dados consigam integrar com facilidade. Esse padrão é amplamente utilizado em várias relações no mapa, especialmente entre os CDs nucleares e de suporte.

O mapa destaca também os casos em que um contexto "a jusante" (*downstream*) – identificado pela letra D – consome informações de um contexto "a montante" (*upstream*) – identificado pela letra U –, traduzindo os modelos de integração para evitar impacto em seus modelos internos. Conforme descrito no capítulo 2, esse tipo de mapeamento se caracteriza por definir uma Camada de Anticorrupção (*Anti-Corruption Layer*) no CD a jusante, destacada no mapa como *ACL*. Os contextos principais apresentados no mapa utilizam esse tipo de mapeamento para se comunicar com os CDs de suporte, visando fazer com que os modelos de integração sejam adequados às suas necessidades, protegendo seus conceitos internos. Isso também ocorre na relação com o contexto externo Bancos e Parceiros.

É importante destacar que, sempre que possível, recomenda-se a introdução de uma *ACL* nesses casos para ajustar os modelos de integração às necessidades específicas do contexto a jusante (VERNON, 2016).

Nota-se que o mapa sugere uma mudança na comunicação entre equipes para tornar o modelo de negócio mais resiliente. Destacado pelo problema **GO-05** (Tabela 3, seção 3.1), o serviço GO era passivo em relação à PO. Em outras palavras, a equipe que gerenciava GO, um serviço essencial para a empresa, dependia diretamente do consumo de suas informações por outra equipe para dar andamento no ciclo de vida de Ordem. Agora, o CD Plataforma de Fundos de Investimento passa a ser uma peça ativa no ciclo de Ordem, tendo um controle de quando essa entidade deve seguir para as próximas etapas, informando o CD *Backoffice* quando isso acontecer. Essa mudança de comunicação visa resolver o problema **GO-05**, tornando ambos os CDs mais resilientes entre si. Além disso, isso abre margem para uma modelagem tática na criação de Eventos de Domínio, que é abordada posteriormente na seção 4.5.

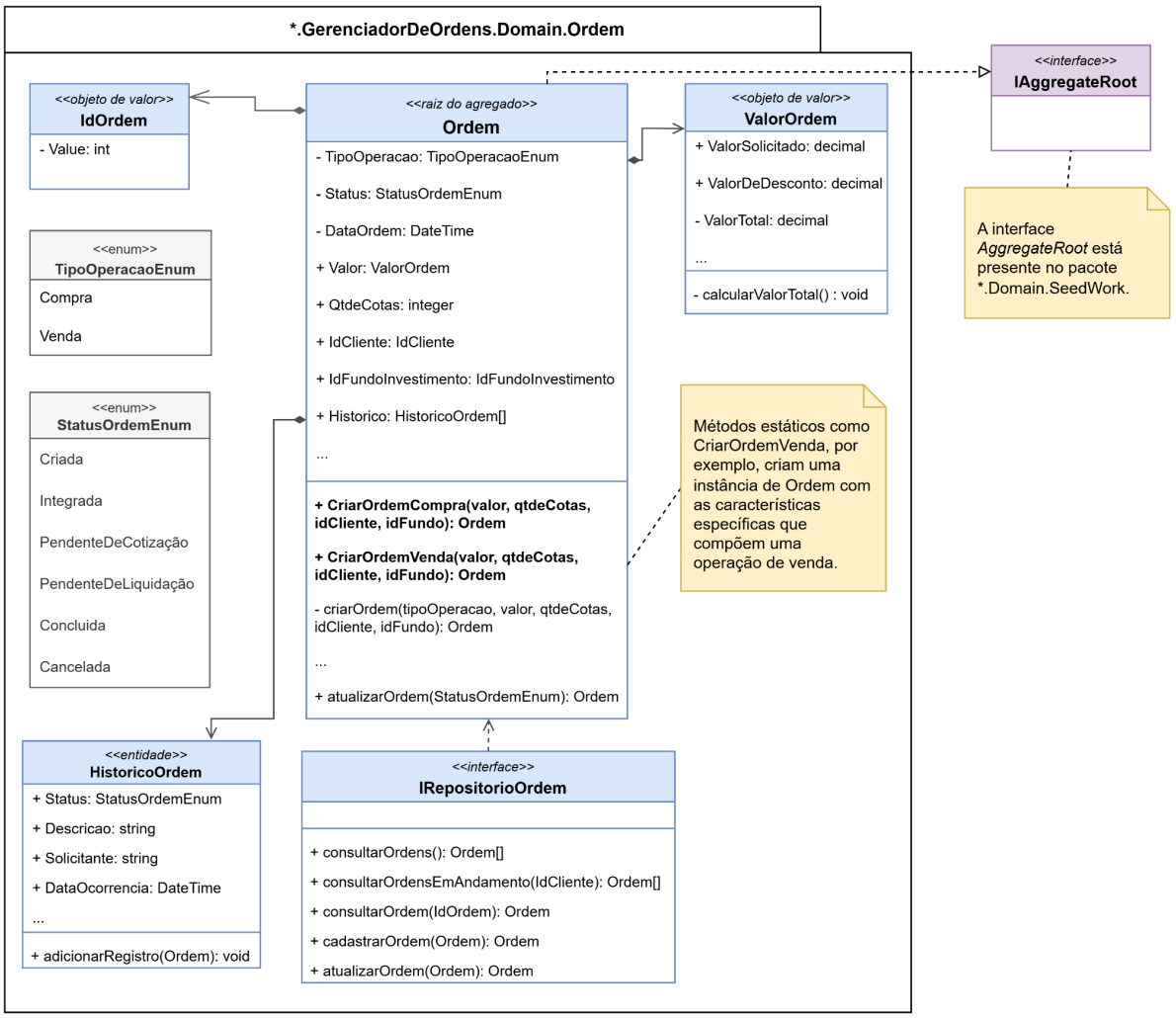
Por fim, são evidenciados cenários onde um contexto a montante não tem motivação para atender às demandas específicas de um contexto a jusante. Nesses casos, o contexto assume uma postura conformista, representada no mapa pela sigla *CF*, em relação ao modelo a montante. Considera-se aqui que a *InvestimentCorp* não possui o controle sobre o desenvolvimento dos projetos de *software* contidos no CD Comunicação com Cliente e Bancos e Parceiros (contexto externo). Por tanto, os relacionamentos para com estes CDs é conformista (*CF*).

4.3 Design Tático com Agregados

A seção 3.1 descreve como o *design* dos principais serviços nucleares foi modelado, e como cada um deles se comunica interna e externamente. A fim de promover uma melhor consistência transacional nas operações realizadas dentro do CD **Plataforma de Fundos de Investimento**, a Figura 11 apresenta o novo modelo de domínio da entidade Ordem em alto nível de abstração. Tal modelo agora leva em consideração a raiz do agregado Ordem, a qual serve como único ponto de acesso para sua lógica de negócio. Objetos de Valor específicos agora são criados, como *ValorOrdem* por exemplo, que contém as informações completas

referentes ao valor financeiro de uma Ordem, como o valor solicitado pelo cliente e os valores deduzidos da operação (quanto houverem), obtidos a partir da integração com o Serviço de Cálculo de Impostos, resultando no Valor Total da operação.

Figura 9 – Modelo de domínio de Ordem



Fonte: o Autor.

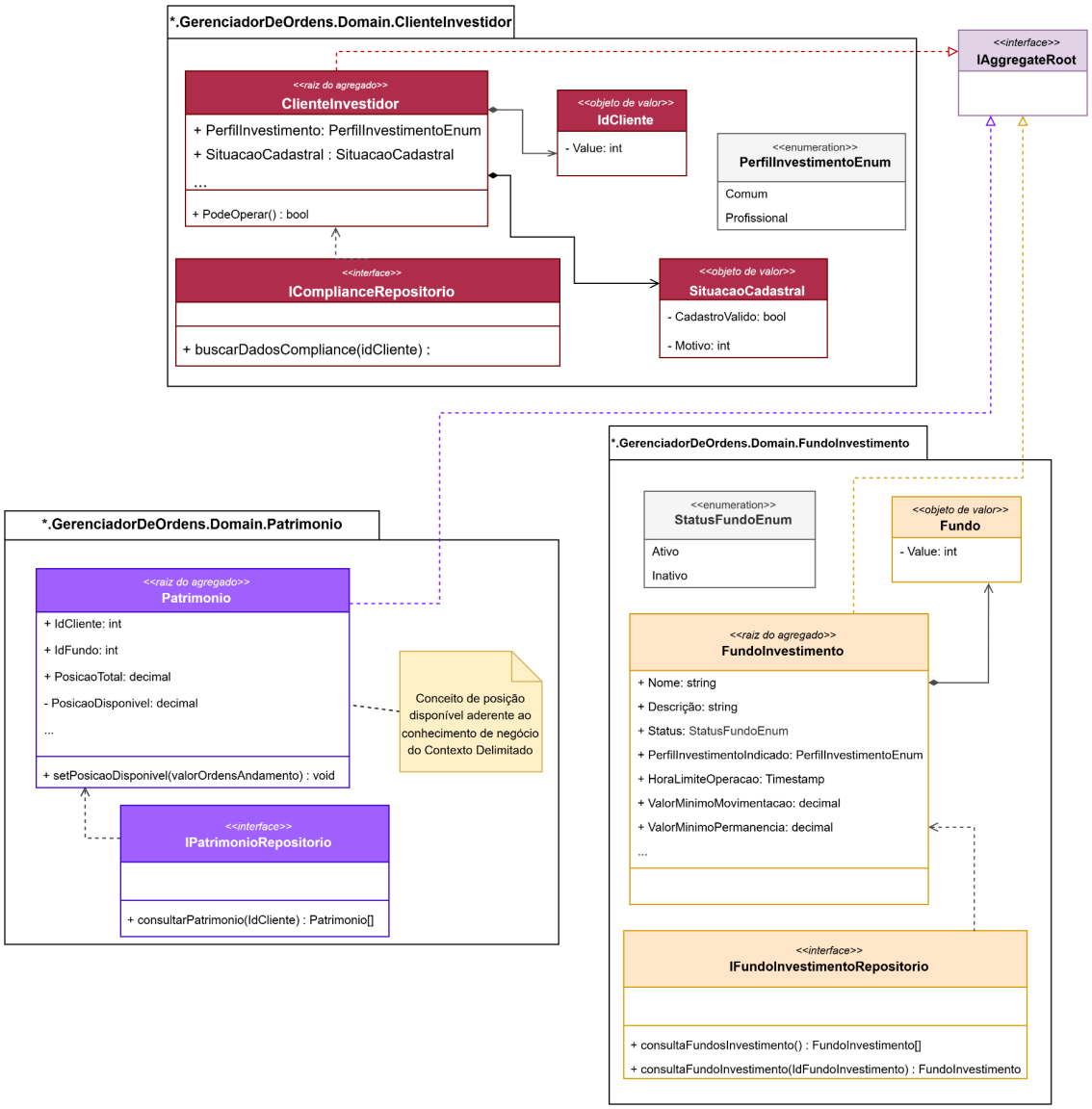
Nota-se, assim como o objeto de valor *OrdemValor*, que a entidade *HistoricoOrdem* não pode mais existir sem que seja a partir da entidade Ordem. Em termos de *design*, isso garante maior consistência transacional. E do ponto de vista de negócio, tal entidade é útil, pois possibilita o controle completo do ciclo de vida de uma Ordem.

Servindo como um pequeno subconjunto de classes, o pacote *SeedWork*[1] contém, a princípio, apenas a interface *IAggregateRoot* (raíz do agregado), a qual é implementada por todas as entidades consideradas como raiz do agregado. Novas classes serão adicionadas posteriormente na seção 4.5.

[1] Termo introduzido por Michael Feathers (2003), popularizado posteriormente por Martin Fowler (2003). Esse pacote também é conhecido por nomes como *Common*, *SharedKernel*, etc.

Assim como Ordem, os modelos de domínio *ClienteInvestidor*, *FundoInvestimento* e *Patrimonio* compõem um papel importante para a criação de uma Ordem. Cada um deles possui seu próprio agregado, os quais podem ser acessados por meio de sua raiz de agregado e referenciados com a utilização de seus identificadores. A Figura 12 descreve com mais detalhes o modelo de domínio das três entidades.

Figura 10 - Modelos de domínio de *ClienteInvestidor*, *FundoInvestimento* e *Patrimonio*



Fonte: o Autor

Pelos diagramas acima, é possível observar que a classe Patrimônio atua praticamente como um intermediador, que cruza as informações de identificação do Fundo de Investimento e Cliente, relacionando a posição total de um cliente em um determinado fundo de investimento. Para aderir à linguagem ubíqua presente no CD Plataforma de Fundos de Investimento, a entidade Patrimonio agora expõe uma nova propriedade, a PosicaoDisponível. A nova propriedade abriga o valor calculado da posição total do cliente deduzindo o valor financeiro das ordens ainda não finalizadas que este cliente possui.

A entidade *ClienteInvestidor* abriga as principais propriedades relacionadas ao cliente no que diz respeito à sua condição de realizar operações financeiras, a qual se baseia em diferentes combinações de propriedades, como sua situação cadastral, perfil de investimento, etc. Ao expor o método PodeOperar (boleano), é possível utilizar seu resultado nas diferentes validações realizadas pelo Contexto Delimitado.

Por fim, a entidade *FundoInvestimento*, que abriga informações importantes sobre o Fundo de Investimento em questão. Tal como ocorre em *ClienteInvestidor*, esta entidade possui um objeto de valor que representa seu identificador (IdFundoInvestimento), o qual pode ser utilizado nas demais implementações quando necessário.

4.4 Design Tático com Serviços de Domínio

Como descrito no capítulo 3, atualmente a validação de uma Ordem no serviço Gerenciador de Ordens é feita por um Serviço de Aplicação. Tal serviço é responsável por coordenar as chamadas aos serviços externos necessários, aplicar as regras de negócio de validação de Ordem e, por fim, criar a entidade Ordem, caso as validações sejam concluídas com sucesso.

Embora essa abordagem atenda às necessidades de negócio, ela apresenta limitações quanto à organização e clareza do domínio. Como identificado no Capítulo 3 por meio do problema **GO-07**, a centralização da lógica de validação de Ordens no serviço de aplicação pode dificultar a manutenção, reduzir a coesão do modelo de domínio, além de poder gerar inconsistências na lógica de negócio.

Portanto, propõe-se nesta seção a refatoração deste comportamento a partir da adoção de um Serviço de Domínio, sendo este o componente fundamental para coordenar as chamadas a de agregados de outros contextos delimitados e centralizar as principais validações responsáveis pela criação de uma Ordem. A mudança visa deslocar tais validações para o escopo do modelo de domínio, uma vez que as regras de negócio pertencem a este modelo. Assim, o serviço de aplicação atua apenas como um orquestrador.

4.4.1 Responsabilidades do Serviço de Domínio

Como apresentado no capítulo 2, serviços de domínio são componentes responsáveis por realizar invocações envolvendo múltiplos agregados ou que não podem ser atribuídas estritamente a uma única entidade. No caso da transação de criação de uma ordem, as seguintes validações devem ser realizadas previamente:

Coordenação de Modelos de Domínio: Consultar serviços externos para recuperar os dados necessários para a validação, como perfil do cliente, posição disponível e regras operacionais dos fundos.

Verificação de Regras de Negócio: Avaliar as informações obtidas para garantir que a solicitação cumpra todas as condições necessárias. Por exemplo:

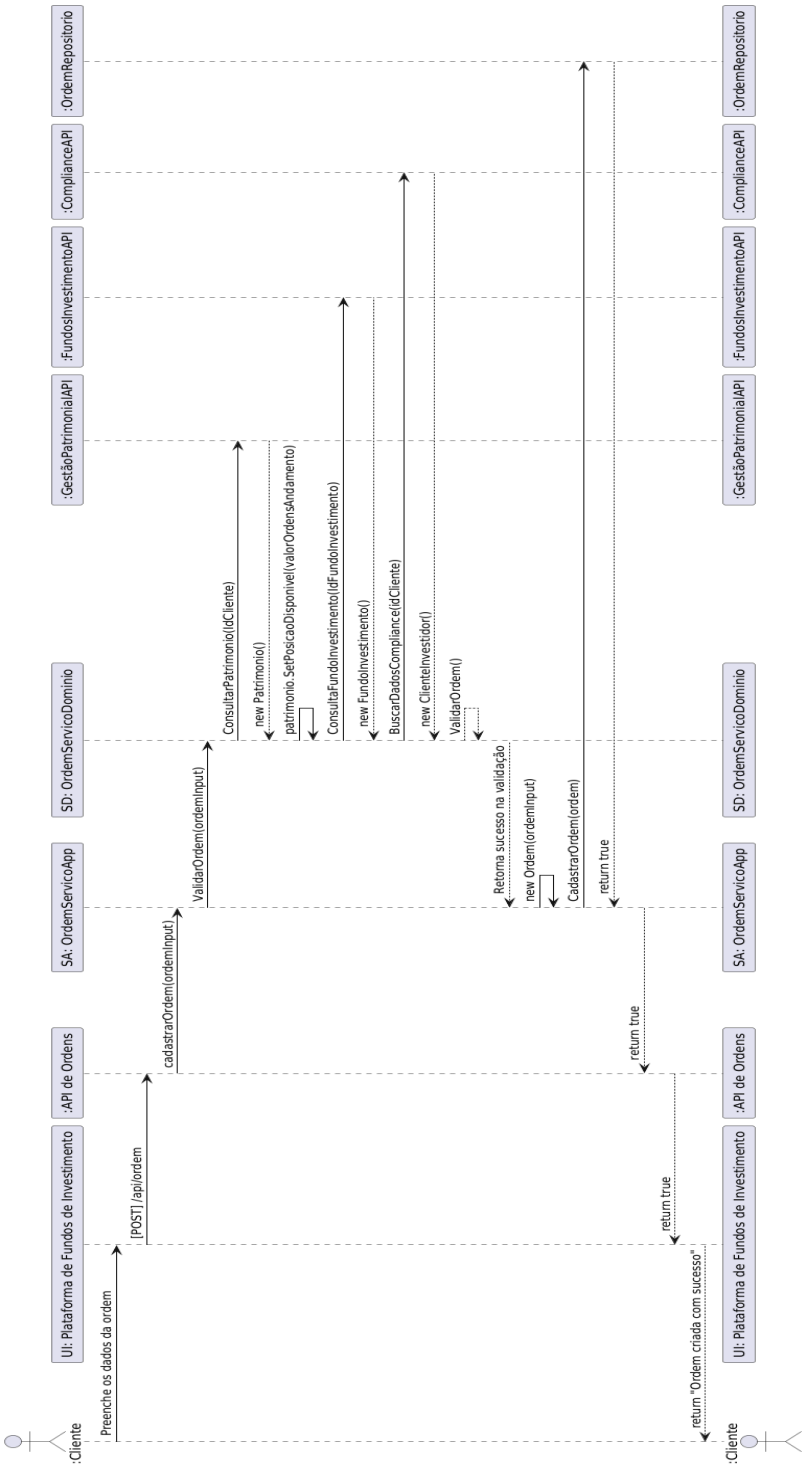
- Garantir que o cliente tenha posição suficiente em um fundo para realizar uma venda.
- Verificar se o fundo está ativo e dentro do horário de operação.
- Validar se o perfil do cliente está alinhado com as características do fundo.

Retorno: Após realizar as validações, o serviço de domínio retornará um resultado indicando o sucesso ou a falha das validações, detalhando quais regras foram violadas, caso aplicável.

Essa separação entre o Serviço de Aplicação e o Serviço de Domínio permite que o primeiro se concentre na orquestração das transações necessárias, como receber a requisição do cliente, chamar o serviço de domínio para validações e persistir a entidade de ordem em caso de sucesso. Já o serviço de domínio focaliza exclusivamente na aplicação das regras de negócio,

promovendo maior coesão no modelo de domínio. A Figura 13 ilustra como ocorre a interação entre esses dois tipos de serviço.

Figura 13 - Diagrama de Sequência do Serviço de Domínio (resumida)



Fonte: o Autor

4.4.2 Benefícios Esperados

Esta mudança visa conquistar benefícios em termos de *design* de software e alinhamento ao domínio da *InvestmentCorp*, como:

- **Maior Clareza e Organização do Modelo**: Distribuição das responsabilidades de cada componente, isolando as validações de uma Ordem em um serviço específico.
- **Evolução do software**: As regras atribuídas ao serviço de domínio podem ser reutilizadas em cenários futuros, se necessário.
- **Proteção do Modelo de Domínio**: Centralizar as regras de negócio no Serviço de Domínio evita que a lógica essencial se espalhe por diferentes partes do código, reduzindo o risco de efeitos colaterais adversos e inconsistências.

4.5 Design Tático com Eventos de Domínio

Retomando o problema apresentado em **GO-05** (Tabela 3, seção 3.1), os serviços **GO** e **PO** possuíam uma forma de comunicação direta, na qual a continuidade do estado de uma Ordem em **GO** dependia direta e imediatamente da ação de consulta de **PO**. Como visto na seção 4.2, foi apresentada uma proposta de mudança na comunicação entre os CDs responsáveis por GO e PO (Plataforma de Fundos de Investimento e *Backoffice*, respectivamente), de forma que GO é o principal responsável por controlar as ações a serem feitas a uma Ordem durante o início de seu ciclo de vida.

Visando tornar o ciclo de vida de uma Ordem mais coeso dentro dos contextos delimitados em que esse conceito aparece, são introduzidos no modelo de domínio do CD Plataforma de Fundos de Investimento os chamados Eventos de Domínio. Como discutido no capítulo 2, os Eventos de Domínio expressam uma ação já realizada a partir de um agregado (por exemplo: *Ordem Criada*, *Ordem Processada*, *Ordem Finalizada*, etc.). A Tabela 4 detalha os possíveis eventos que ocorrem tanto no CD Plataforma de Fundos de Investimento, durante o início do ciclo de vida de uma Ordem, quanto no CD *Backoffice*, durante as fases de processamento da Ordem.

Tabela 4 – Eventos de Domínio em GO.

Evento	CD	Descrição
OrdemSolicitada	Plataforma de Fundos de Investimento	Quando uma Ordem foi solicitada com sucesso (início do ciclo de vida).
OrdemCancelada	Plataforma de Fundos de Investimento	Quando uma Ordem foi cancelada com sucesso (fim do ciclo de vida).
OrdemCotizada	*Backoffice*	Quando uma Ordem foi cotizada com sucesso.
OrdemLiquidada	*Backoffice*	Quando uma Ordem foi liquidada com sucesso.
OrdemFinalizada	Plataforma de Fundos de Investimento	Quando uma Ordem foi finalizada com sucesso (fim do ciclo de vida).

Fonte: o Autor.

Com a inclusão dos eventos de domínio, espera-se atingir um maior grau de coesão dos modelos de domínio dos CDs em que são utilizados, de forma a expressar por meio de tais eventos as ações realizadas pelos agregados de forma mais clara.

5. CONSIDERAÇÕES FINAIS

5.1 Conclusões

Este trabalho teve como objetivo analisar e propor soluções de *design* para alguns problemas de modelagem identificados em produtos de software implementados em microsserviços, utilizando os conceitos de *Domain-Driven Design* (DDD). Para ser possível exemplificar tanto os problemas quanto a aplicação de técnicas de DDD, foi utilizada uma solução de arquitetura de software de uma empresa fictícia, a *InvestmentCorp*, espelhada anonimamente em um caso real. Ao longo do estudo, foi possível observar como um *design* inadequado pode impactar negativamente na coesão e no acoplamento dos serviços, além de dificultar a manutenção e evolução do sistema.

A análise do *design* do software existente, que implementa uma capacidade de negócio nuclear da empresa, destacou problemas como a ambiguidade de conceitos-chave, exemplificada pelas diferentes interpretações do termo "Ordem" entre os serviços, a baixa coesão entre os serviços, como evidenciado no uso do conceito de "Posição", além dos vários pontos com alto acoplamento entre elementos arquitetônicos. Um exemplo deste último, é a introdução de camadas anticorrupção, como decisão estratégica derivada do mapeamento de contextos.

Esses problemas foram abordados por meio da aplicação de técnicas de *design* estratégico, como a identificação de subdomínios e a criação de contextos delimitados, e de *design* tático, como a introdução de agregados, objetos de valor e serviços de domínio. O resultado, como mostrado com detalhes no capítulo 4, permitiu o aumento da coesão nos contextos delimitados examinados, assim como a diminuição do acoplamento entre elementos desses contextos e entre os próprios contextos, com a reformulação, no caso do Gerenciador de Ordens (GO).

Os resultados obtidos também mostram que a aplicação de DDD não apenas mitiga os problemas técnicos, mas também melhora a clareza e a expressividade do modelo, facilitando a comunicação entre equipes e alinhando o *design* técnico aos objetivos de negócio. Enfim, pode-se concluir que o presente trabalho evidencia aspectos do potencial do uso do DDD

como direcionador do *design* em ambientes de microsserviços, promovendo maior alinhamento semântico e estrutural dos elementos de software e as capacidades do negócio.

5.2 Contribuições do Trabalho

As principais contribuições deste trabalho podem ser sintetizadas em três pontos:

Diagnóstico detalhado de uma implementação existente de microsserviços. Foi realizada uma análise aprofundada do *design* atual, identificando problemas estruturais e organizacionais que impactam diretamente a coesão e o acoplamento dos serviços.

Aplicação prática de técnicas de *design* do DDD. O trabalho apresentou uma aplicação sistemática de conceitos de DDD, incluindo *design* estratégico e tático, com foco em resolver os problemas de modelagem das abstrações de software e fortalecer o alinhamento entre os serviços e o domínio de negócio.

Propostas de soluções potencialmente generalizáveis. As soluções propostas, como o uso de serviços de domínio e camadas anticorrupção, podem ser adaptadas para outros cenários de implementação semelhantes, podendo ser utilizadas como um guia para equipes que enfrentam desafios similares em arquiteturas de microsserviços.

5.3 Trabalhos Futuros

Embora o trabalho tenha abordado com detalhe os problemas de *design* identificados, algumas áreas ainda podem ser exploradas em estudos futuros:

- Realizar a implementação das soluções aqui propostas em um ambiente real, avaliando o impacto em métricas como tempo de desenvolvimento, clareza da base de código e redução da taxa de falhas, por exemplo.

- Realizar análise de custos comparativos entre o custo de uma implementação atual e de sua evolução com a aplicação do DDD, com a seleção de métricas que permitam decisões de projeto (*project*) baseadas em análise de compensação (*trade-off*) entre alternativas de solução.
- Realizar análise comparativa dos testes (de unidade e integração) entre o cenário atual e o cenário proposto após a aplicação das técnicas de DDD.
- Investigar como a aplicação de DDD influencia a dinâmica das equipes de desenvolvimento, especialmente no que diz respeito à comunicação e alinhamento de expectativas.

REFERÊNCIAS

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. **Software Architecture in Practice: Software Architect Practice_c3**. [s.l.]: Addison-Wesley, 2012.

PARNAS, David. **Information Distribution Aspects of Design Methodology**. IFIPS Congress. 71. 339-344, 1971.

EVANS, Eric. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. [s.l.]: Addison-Wesley, 2003.

EVANS, Eric. **Domain-Driven Design Reference: Definitions and Pattern Summaries**. [s.l.]: Dog Ear Publishing, 2014.

FOWLER, Martin. **Conway's Law**. Martinfowler.com. Disponível em: <https://martinfowler.com/bliki/ConwaysLaw.html>. Acesso em: 9 Dec. 2024.

FOWLER, Martin. **Domain Driven Design**. Martinfowler.com. Disponível em: <https://martinfowler.com/bliki/DomainDrivenDesign.html>. Acesso em: 9 Dec. 2024.

FOWLER, Martin. **Software Architecture Guide**. Martinfowler.com. Disponível em: <https://martinfowler.com/architecture/>. Acesso em: 9 Dec. 2024.

FOWLER, Martin; LEWIS, James. **Microservices**. martinfowler.com. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 9 Dec. 2024.

FOWLER, M. **Reducing coupling**. IEEE Software, v. 18, n. 4, p. 102–104, 2001.

KHONONOV, Vlad. **Learning Domain-Driven Design**. [s.l.]: "O'Reilly Media, Inc.," 2021.
NEWMAN, Sam. Criando Microsserviços – 2a Edição: Projetando sistemas com componentes menores e mais especializados. [s.l.]: Novatec Editora, 2022.

OUMOUSSA, Idris; SAIDI, Rajaa. **Evolution of Microservices Identification in Monolith**

Decomposition: A Systematic Review. IEEE Access, v. 12, p. 23389–23405, 2024.

SU, Ruoyu; LI, Xiaozhou; TAIBI, Davide. **From Microservice to Monolith: A Multivocal Literature Review**. Electronics, v. 13, n. 8, p. 1452, 2024.

VERNON, Vaughn. **Domain-Driven Design Distilled**. [s.l.]: Addison-Wesley Professional, 2016.

VERNON, Vaughn. **Implementing Domain-Driven Design**. [s.l.]: Addison-Wesley, 2013.

ZHONG, Chenxing; LI, Shanshan; HUANG, Huang; *et al*. **Domain-Driven Design for Microservices: An Evidence-Based Investigation**. IEEE Transactions on Software Engineering, v. 50, n. 6, p. 1425–1449, 2024.

RICHARDSON, Chris. **Microservices Patterns: With examples in Java**. [s.l.]: Simon and Schuster, 2018.

MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design.** [s.l.]: Prentice Hall, 2017.

PARNAS, D. L. **On the criteria to be used in decomposing systems into modules**. Communications of the ACM, v. 15, n. 12, p. 1053–1058, 1972.

FOWLER, Martin. **Seedwork**. Martinfowler.com. Disponível em: <https://martinfowler.com/bliki/Seedwork.html>. Acesso em: 14 Jan. 2025.

FOWLER, Martin. **Padrões de Arquitetura de Aplicações Corporativas**. 1. ed. [s.l.]: Bookman, 2009.

FEATHER, Michael. **Stunting a Framework**. Weblogs Forum. Disponível em: <https://www.artima.com/forums/flat.jsp?forum=106&thread=8826>. Acesso em: 14 Jan. 2025.

GAROUSI, Vahid; FELDERER, Michael; MÄNTYLÄ, Mika V. **Guidelines for including grey literature and conducting multivocal literature reviews in software engineering.** Information and Software Technology, v. 106, p. 101–121, 2019.