LEONARDO HENRIQUE DE BRAZ

DOMAIN DRIVEN DESIGN:

VANTAGENS E DESVANTAGENS EM SEU USO

LAVRAS – MG

2021

LEONARDO HENRIQUE DE BRAZ

DOMAIN DRIVEN DESIGN:
VANTAGENS E DESVANTAGENS EM SEU USO

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Bacharel.

Prof. Dr. Ricardo Terra

Orientador

LAVRAS – MG

2021

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Braz, Leonardo Henrique de

Domain Driven Design : Vantagens e desvantagens em seu uso / Leonardo Henrique de Braz. 1^a ed. – Lavras : UFLA, 2021.

52 p. : il.

Monografia (bacharel)–Universidade Federal de Lavras, 2021.

Orientador: Prof. Dr. Ricardo Terra.

Bibliografia.

1. TCC. 2. Monografia. 3. Dissertação. 4. Tese. 5. Trabalho Científico – Normas. I. Universidade Federal de Lavras. II. Título.

CDD-808.066

LEONARDO HENRIQUE DE BRAZ

DOMAIN DRIVEN DESIGN: VANTAGENS E DESVANTAGENS EM SEU USO

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Bacharel.

APROVADA em 19 de novembro de 2021.

Prof. Dr. Maurício Ronny de Jesus Almeida UFLA
Prof. PhD. Gustavo Jansen de Souza Santos UTFPR

Prof. Dr. Ricardo Terra
Orientador

LAVRAS – MG
2021

AOS MEUS PAIS

Que me ensinaram que não há nada nesse mundo que seja impossível ou fora do meu alcance. E que nenhuma dificuldade é motivo de desistir.

AGRADECIMENTOS

Agradeço primeiramente a Deus a minha vida e saúde. Agradeço também aos meus pais, Leo e Rosimeire, o incentivo prestado desde o momento em que decidi sair de sua casa para me dedicar aos estudos e que me deram apoio nos momentos difíceis, compreendendo a minha ausência enquanto eu me dedicava à realização deste trabalho. E aos meus irmãos, sempre presentes e me inspiram a continuar.

À minha namorada, Kleissiely, pela ajuda durante toda minha trajetória até aqui desde antes da minha graduação. Além de sempre me motivar, compartilhou ao meu lado os momentos responsáveis por moldar a pessoa que sou hoje.

Agradeço ao meu orientador Ricardo Terra o ensinamento e orientação, sempre com muita paciência e dedicação, me incentivando a evoluir cada dia mais e me encorajando a dar o meu melhor para este trabalho, demonstrando acreditar na minha ideia.

Aos meus professores do Instituto Federal de Rondônia (IFRO), durante meu ensino técnico, os maiores incentivadores ao meu ingresso no ensino superior, em especial para o Jackson Henrique e Juliano Fischer.

Também quero agradecer a formação oferecida pela Universidade Federal de Lavras. Aos seus membros da universidade, em especial para a professora Juliana Galvani Greghi, a orientação em outras atividades e o oferecimento de grandes oportunidades durante a faculdade.

A todos os meus familiares próximos, pelo incentivo nos momentos difíceis e por compreenderem a minha ausência enquanto eu me dedicava à realização deste trabalho, em especial para Alfredo, Cidalina, Idalina, Irene, Isabel, Leomar, Lucineia e Valda.

Aos meus amigos de longa data, pela ajudara a manter o foco e participaram de grandes momentos de descontração, mesmo que a quilômetros de distância, em especial para Daniele, Helder, Heloisa, Iago, Lucas Maroto e Pedro.

E, por último mas não menos importante, agradeço as amizades criadas durante meu trajeto na universidade e que, com toda certeza, serão para toda vida. Destaco aqui meu agradecimento para Gabrielle Cuba, Guilherme Melo, Geovani Iruam, Helena Muniz, João Paulo (JP), Luis Felype, Maurício Ronny, Matheus Almeida, Ruan Marcos e Wagner Moretti. Em especial para o Guilherme Ochikubo e sua mãe, Marlene Barbosa, que sempre se mostraram pessoas incríveis, me recebendo com muito carinho e sempre estiveram dispostos a me ajudar, parceiros para todas as horas.

Uma mãe de verdade entende o que o filho não disse.
Autismo Diário

RESUMO

Com a evolução das capacidades de projeto, desenvolvimento e sustentação de um software, além da alta demanda de uso e da procura alavancada de sistemas sob demanda, tornou-se um desafio para os especialistas e praticantes da engenharia de software gerenciar e atender às necessidades de crescimento e organização de projetos. Nesse cenário, surgiram vários ideais e metodologias destinados a auxiliar no levantamento das partes críticas e essenciais de um software, dentre as quais tem-se o *Domain Driven Design* (DDD), cuja finalidade é descomplicar a modelagem e entendimento do problema a solucionado pelo sistema. Tendo isso em vista, o objetivo deste trabalho é servir como um guia de decisão para o uso do DDD em um projeto de software, apresentando comparativos entre diversos critérios e auxiliando na decisão de sua aplicação e uso. Nesse sentido, este artigo justifica-se pelo entendimento de que o uso do DDD sem avaliar a real necessidade de sua utilização resulta em complicações e dificuldades na sua aplicação, o que pode diminuir a produtividade da equipe e gerar problemas no desenvolvimento do software. Para a consecução do objetivo proposto utilizou-se as pesquisas de Evans (2004) e Vernon (2013) para sustentar, respectivamente, as discussões sobre metodologias de desenvolvimento introduzindo o DDD e aplicações práticas do DDD em projetos de software. Sendo assim, para o desenvolvimento desta pesquisa qualitativa, foram realizados estudos de cunho teórico, a fim de produzir comparativos com diversos parâmetros voltados ao desenvolvimento do software, em comparação com um projeto sem o uso do DDD. Espera-se, com este trabalho, definir um catálogo de análises para garantir que sua utilização traga benefícios à equipe do projeto e aos demais envolvidos.

Palavras-chave: DDD. Domain Driven Design. Engenharia de Software.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Com o crescimento da demanda de desenvolvimento de software, ocasionado pela necessidade de criação de novos sistemas de diferentes áreas e finalidades, gerenciar suas criações e respectivas evoluções tornou-se um desafio para os especialistas e praticantes da engenharia de software, necessitando de grande empenho e organização do projeto (EVANS, 2004). Além disso, segundo Vernon (2013), com as inúmeras opções de tecnologias disponíveis no mercado que podem ser utilizadas para criar um software, o foco do desenvolvimento acaba voltado aos aspectos técnicos de um sistema (definido por ele como "detalhe" da aplicação) ao invés da parte fundamental e mais importante na solução: a dor do cliente.

No decorrer do tempo, vários conceitos e ideais surgiram para auxiliar a equipe de desenvolvimento no levantamento de requisitos e entendimento das partes críticas e essenciais de um software, auxiliando na elaboração e definição do seu escopo baseado nos principais objetivos de sua construção (EVANS, 2004). Dentre essas, o *Domain Driven Design* (DDD), abordado neste trabalho, tem como finalidade descomplicar a modelagem e compreensão dos problemas de negócio que deverão ser resolvidos pelo software, cujo foco é voltado ao escopo do problema e entendimento da necessidade do cliente.

No entanto, o DDD não é a solução para todos os problemas encontrados no desenvolvimento e gerenciamento de um software. Como qualquer outra metodologia e conceito existente, é necessário uma motivação ou necessidade para seu uso e que seja de conhecimento da equipe o real motivo de sua utilização. Tal escolha pode ser realizada a partir de análises e levantamento dos benefícios em um projeto, garantindo assim que o desenvolvimento seja beneficiado por essa decisão (VERNON, 2013; EVANS, 2004; MILLETT; TUNE, 2015).

Sendo assim, o objetivo deste trabalho é definir um guia de decisão para o uso do DDD em projetos de software. Para isso, este estudo compara implementações com e sem a utilização do DDD sob diversos aspectos visando apontar suas vantagens e desvantagens. Como contribuição, este trabalho provê um catálogo de referências para garantir que a escolha em adotar ou não o DDD seja benéfica à equipe e aos demais inclusos no projeto, evitando assim complexidade adicional no desenvolvimento e entendimento do problema.

Para realização das análises, foram construídas duas implementações a partir da modelagem de um sistema alvo específico, apresentando o mesmo funcionamento. A diferença entre ambas está na sua forma de desenvolvimento, sendo uma construída a partir dos conceitos descritos pelo DDD e a outra sem a sua utilização. Foram realizadas, para cada implementação,

análises e classificações de prós e contras em sua utilização, associados aos seguintes aspectos de desenvolvimento: organização de código, desempenho, manutenibilidade, dificuldade de criação inicial, curva de aprendizado, produtividade e escalabilidade. Espera-se, com esta pesquisa, auxiliar equipes de desenvolvimento a tomar decisões sobre a utilização do DDD para seus projetos a partir das análises presentes, além de promover a ação de análises de benefícios ao utilizar uma metodologia ou formato de desenvolvimento.

Como referência para a consecução do objetivo proposto, alguns dos principais autores, tais como Evans (2004), Vernon (2013) e Millett (2015), foram extremamente utilizados para entender os conceitos e fundamentos que o DDD apresenta, além das técnicas de modelagem e decisões em projeto.

Este trabalho está organizado da seguinte forma. A Seção 2 introduz o *Domain Driven Design*. Em seguida, a Seção 3 descreve a metodologia e as decisões tomadas neste trabalho. A Seção 4 define o sistema alvo utilizado e a Seção 5 detalha as implementações. A Seção 6 realiza os comparativos entre as implementações. Já na Seção 7 são apresentados os trabalhos relacionados e, por fim, na Seção 8, há as considerações finais deste estudo.

2 *DOMAIN DRIVEN DESIGN*

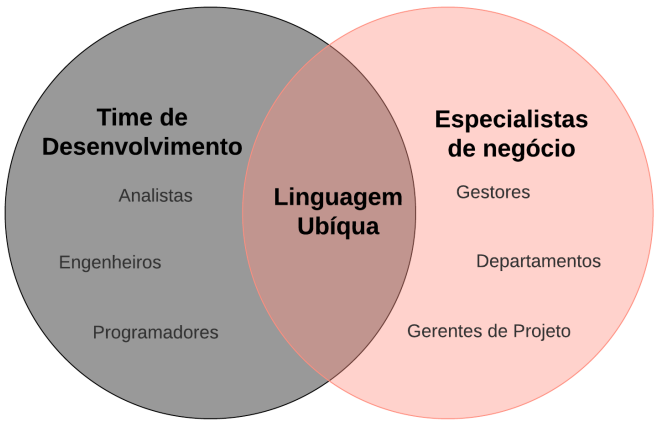
O *Domain Driven Design*, de acordo com Evans (2004), é uma abordagem de desenvolvimento de software voltada ao modelo de produção focada diretamente no negócio (área ou problema que o software deve resolver) de alto nível de comprometimento. O DDD é conhecido por ser um conjunto de princípios generalistas com foco voltado ao problema de domínio, exploração de casos de uso e definição da comunicação entre os membros de uma equipe via linguagem Ubíqua, baseando no contexto do projeto.

O esquema do DDD é dado a partir de três grandes pilares essenciais para um entendimento inicial de como aplicar e trabalhar com sua utilização: linguagem ubíqua, *Contextos Delimitados* e mapas de contexto.

2.1 Linguagem Ubíqua

A linguagem ubíqua (representada pela Figura 2.1) consiste em definir uma comunicação efetiva com as pessoas que estão participando do desenvolvimento. Sejam elas desenvolvedores ou *experts*, é necessária uma forma de comunicação única, sendo criada entre a colaboração da equipe técnica e pessoas de negócio (EVANS, 2004; VERNON, 2013).

Figura 2.1 – Esquema de representação da *Linguagem Ubíqua*



Fonte: do autor.

Segundo Evans (2004), essa forma de conversação torna-se um artefato vivo dentro do projeto, tendo necessidades de evolução e refinamento conforme crescimento ou modificações do mesmo, além da sua própria utilidade, pois a presença de um modelo onipresente de conversação demonstra a sua própria necessidade de torná-lo explícito para que todos entendam os seus termos, a partir de catálogos, documentações ou glossários compartilhados.

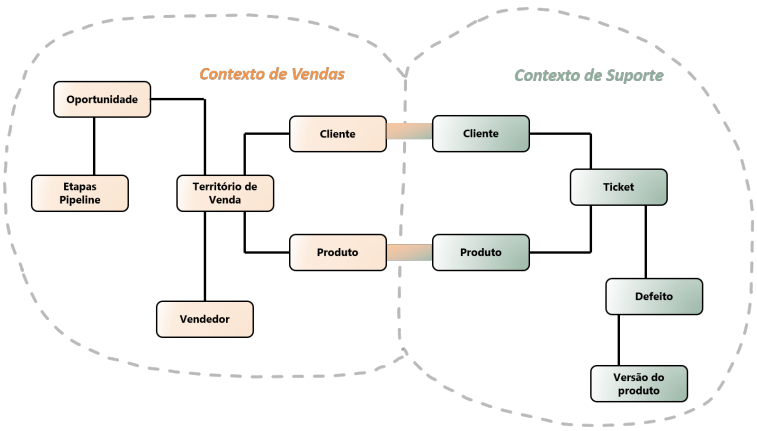
2.2 *Contextos Delimitados*

No *Domain Driven Design*, como o próprio nome diz, o foco de trabalho se encontra no **domínio** do problema, que representa a realidade da aplicação, referindo assim ao contexto de conhecimento na qual as pessoas e a empresa estão inseridos (EVANS, 2004; MILLETT; TUNE, 2015).

O domínio da aplicação compõe todas as regras necessárias para a execução e solução dos processos, em conjunto aos comportamentos e toda a abstração de entendimento das operações que estão presentes. O domínio é a base do negócio e a razão de existência do software. Ele se refere a área de conhecimento e atividade na qual a organização e as pessoas estão inseridas (EVANS, 2004). Uma boa aplicação desses conceitos necessita da filosofia de *Contextos Delimitados* (*Bounded Contexts*), os quais se referem a divisões realizadas dentro da linguagem presente, estabelecendo assim contextos diferentes a cada modelo de processo, área ou os envolvidos no negócio. Cada uma das partes definidas possuem sua própria abstração dos problemas, mantendo a integridade e responsabilidade detalhada em comparação aos outros contextos, tendo funções únicas, resultando assim na presença (caso seja necessário) de uma linguagem ubíqua para cada.

A Figura 2.2, sobre os *Contextos Delimitados*, demonstra a existência de dois contextos para um determinado sistema, sendo ambos caracterizando uma representação de `Produto` e `Cliente`. É possível perceber que baseando do mesmo modelo real, ambos tratam-se de aspectos diferentes e classificados a partir do seu escopo de modelagem. O `Cliente` definido no Contexto de `Vendas` não possui a mesma semântica quando comparado a importância no Contexto de `Suporte`.

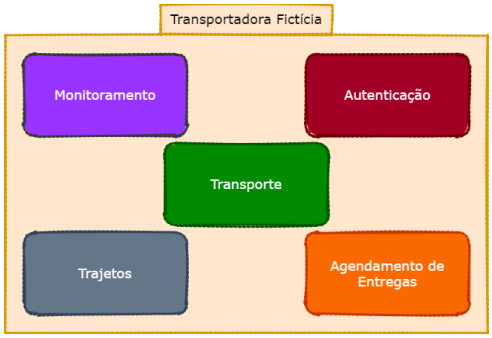
Figura 2.2 – *Contextos Delimitados* fictícios com entidades em comum



Fonte: Fowler (2014). Traduzido pelo autor.

Já a Figura 2.3 demonstra a modelagem dos contextos voltados a um sistema de controle de uma Transportadora de Mercadorias. Considere que os contextos foram levantados pela equipe de desenvolvimento dessa aplicação. A partir desse levantamento é possível notar que cada domínio representado é responsável por tratar diretamente sobre um nicho específico em transportes.

Figura 2.3 – Domínios da transportadora fictícia.



Fonte: do autor.

O domínio principal, aquele que é a parte fundamental do negócio e tem como objetivo resolver os problemas presentes, é o de `Transporte` (por se tratar de uma transportadora de cargas). Os outros domínios identificados são focados em trabalhar em outras áreas de atuação dentro do negócio, descritos pela Tabela 2.1.

Tabela 2.1 – Descrição dos *Contextos Delimitados* da empresa fictícia.

Contexto	**Descrição**
Transporte	Responsável por definir as principais rotas de entrega da transportadora seguindo suas respectivas logísticas.
Agendamento de Entregas	Utilizado pelos clientes da transportadora para realizar agendamento de entrega a partir de datas processadas pelas regras definidas.
Monitoramento	Responsável por prover um meio de acesso aos que necessitam obter informações de localização e monitoramento específico sobre os veículos presentes na transportadora.
Autenticação	Responsável por realizar a identificação dos usuários que acessam o sistema, incluindo as respectivas permissões.

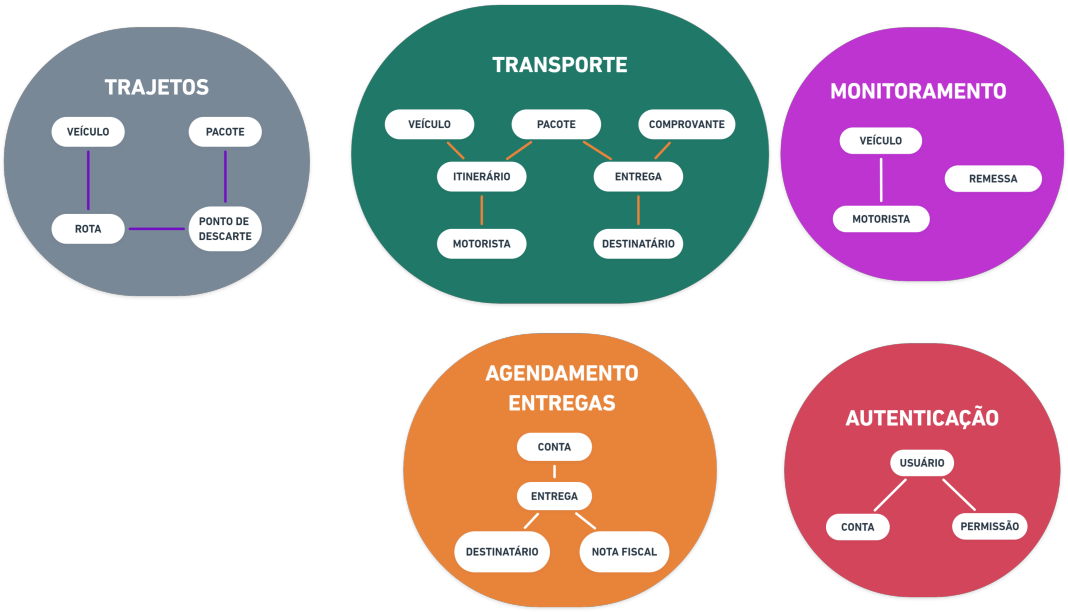
Fonte: do autor.

Uma modelagem válida dos *Contextos Delimitados* para o exemplo da empresa fictícia pode ser visto na Figura 2.4. Cada um deles podem ter representações internas totalmente independente dos outros, incluindo seus modelos, entidades, objetos de valor, serviços ou padrões

de projeto (GAMMA et al., 1994). Mesmo que todos lidem com o mesmo formato de representação no modelo real (o produto da entrega, o entregador ou o destinatário), cada um poderá ter sua própria abstração.

É possível perceber, nessa modelagem, a delimitação exata da Linguagem Ubíqua dentro de cada contexto, conforme analisado pela entidade `Veiculo`, presente no domínio de `Trajeto` e também pelo `Monitoramento`, contendo características distintas de acordo com a necessidade. No `Trajeto`, as informações de capacidade de carga em volumes (ou a quantidade de pacotes) e os gastos de combustível por distância percorrida são importantes para definir um bom algoritmo de roteamento das entregas. Já o `Monitoramento` necessitaria de informações de localização do veículo, placa, motorista e quaisquer outros *objetos de valor*[1] que resultam em informações de localização.

Figura 2.4 – *Contextos Delimitados* elaborados para a transportadora fictícia



Fonte: do autor.

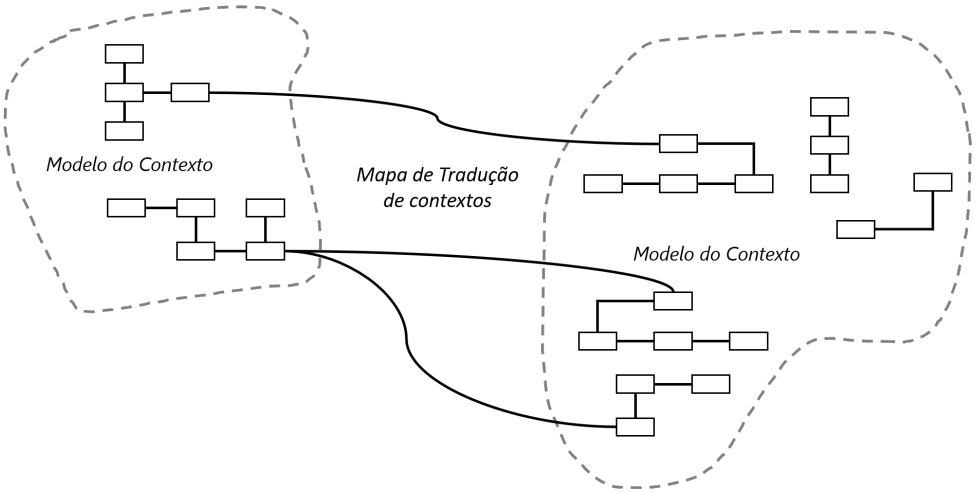
2.3 *Mapas de Contexto*

O último pilar presente nessa base de entendimento é denominado como Mapas de Contexto (*Context Maps*), recurso responsável por realizar o mapeamento da comunicação e funcionamento de todos os *Contextos Delimitados* definidos na aplicação. Esse mapeamento é capaz

[1] Objetos de valor são objetos imutáveis, existentes apenas para compor informações de uma entidade por meio de composição, sem a presença de um ciclo de vida próprio e gerenciável (EVANS, 2004).

de estabelecer a forma de comunicação entre os contextos já existentes e sua interação, seja com regras próprias definidas pela equipe de desenvolvimento ou então por padrões de projeto já existentes.

Figura 2.5 – *Context Maps* e sua importância (EVANS, 2004). Adaptado pelo autor.



Fonte: Evans (2004). Adaptado pelo autor.

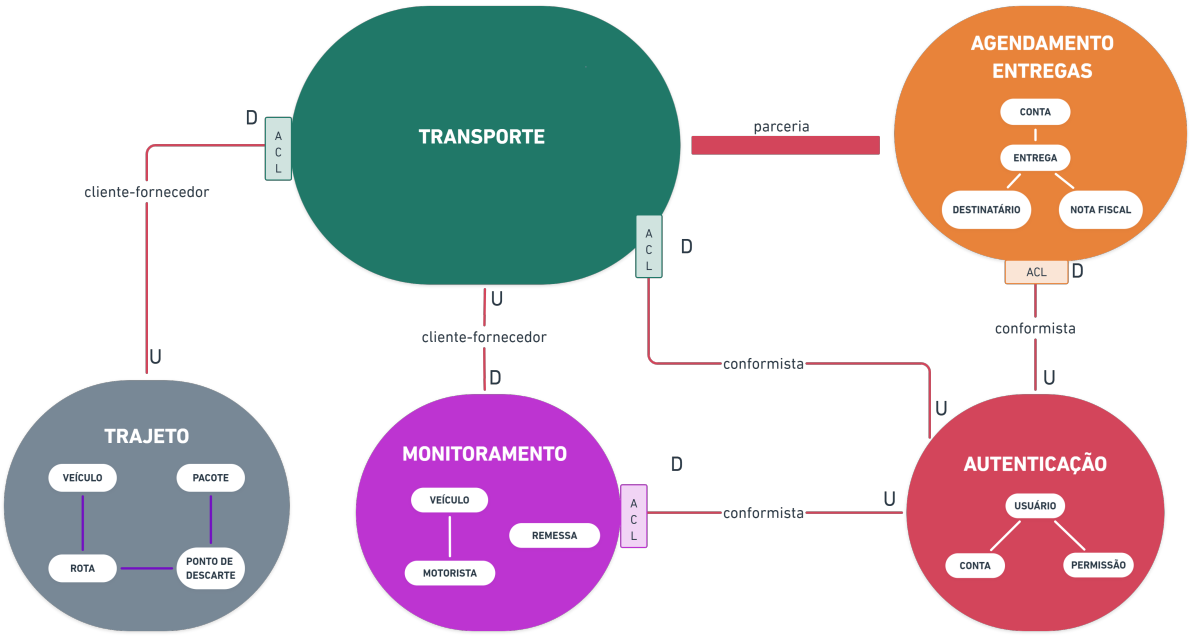
É comum que exista a necessidade de comunicação entre os *Contextos Delimitados*, pois seu trabalho, em conjunto com os demais, molda o sistema como um todo. Essas comunicações são amplamente estudadas a partir da modelagem estratégica e formas de acesso entre os contextos presentes em cada um dos escopos definidos no negócio (EVANS, 2004; FOWLER, 2014). Esses formatos são definições catalogadas e detalhadas entre as equipes para comunicação entre contextos, necessidades de dependências e o compartilhamento de informações. A organização dos artefatos e entendimento geral direciona o projeto à produção de um código limpo, maduro, eficaz, evolutivo e manutenível (MARTIN, 2008).

Nos *Mapas de Contexto*, ficam evidentes as relações de comunicação e/ou integrações diretas entre cada um dos contextos e suas utilizações (cliente-fornecedor, parceria, etc.). A Figura 2.6[2] mostra um mapeamento de contexto simples para os *Contextos Delimitados* encontrados anteriormente na Figura 2.4.

Existem várias formas de realizar a comunicação e interação entre os *Contextos Delimitados*. De acordo com Evans (2004), cada uma dessas estratégias segue um formato específico para cada tipo de relacionamento presente entre os contextos e ideais almejados pelas equipes de desenvolvimento. Em seu trabalho, Vernon (2013) destaca que essas classificações não são fixas

[2] A utilização dos rótulos nas linhas que representam os relacionamentos entre os domínios são ilustrativas para facilitar o entendimento, pois não há uma definição de obrigatoriedade em seu uso.

Figura 2.6 – *Mapas de Contexto* da transportadora



Fonte: do autor.

e são definidas a partir de análises de projeto e suas principais decisões. Na Figura 2.6, grande parte dos contextos possuem os rótulos U e D associados, que significam, respectivamente, *upstream* e *downstream*. Cada um é apresentado separadamente, pois possuem significados específicos em relação ao formato de comunicação.

O relacionamento descrito entre o contexto de *Autenticação* com os contextos de *Agendamento de Entregas*, *Monitoramento* e *Transporte* podem ser classificados como **Conformistas**. Esse tipo de relacionamento destaca o consenso entre as equipes para definir que o contexto *upstream* (o contexto de *Autenticação*) não possui responsabilidades em relação ao acesso, fluxo, lógicas de projeto compartilhadas ou o próprio relacionamento com os contextos *downstream* (*Transporte*, *Monitoramento* e *Agendamento de Entregas*). A função que *Autenticação* exerce no software é de grande importância e afeta em grande parte a lógica do sistema, mas seu funcionamento e estratégias são decisões internas, sem dependências com os outros contextos presentes.

Já o relacionamento presente entre *Transporte* e *Agendamento de Entregas*, ilustrado por uma linha mais grossa que as demais, é denominado como **Parceria**. Nesse tipo de relacionamento torna-se presente uma cooperação entre as equipes de desenvolvimento de ambos os contextos, sejam em sucessos ou falhas, pois são amplamente conectadas e com regras complementares. O contexto de *Transporte* precisa ter acesso a informações (ou implementa-

ções) presentes em *Agendamento de Entregas*, e vice-versa. A cooperação torna mais ágil o desenvolvimento de recursos e interfaces de acesso dos recursos presentes em ambos os contextos (EVANS, 2004).

A comunicação entre os contextos de *Transporte* e *Monitoramento* é classificada como **Cliente-fornecedor**. Esse modelo descreve o envolvimento entre as equipes pelo formato de *upstream* para o fornecedor e *downstream* para o cliente da relação. O cliente vai se beneficiar dos serviços ou recursos que são prestados. O fornecedor é o mandante do relacionamento, definindo o que é disponível e modificando sempre que achar necessário. Nesse formato, a equipe de cliente trabalha com dependência direta para o contexto atrelado e seus recursos, mas é possível que o cliente solicite novos recursos ao fornecedor. No entanto, a decisão sempre fica a cargo do fornecedor. No exemplo da Figura 2.6, além do relacionamento entre *Transporte* e *Monitoramento*, também há uma relação de Cliente-fornecedor entre *Transporte* e *Trajetos*.

Entretanto, há momentos em que utilizar uma sujeição direta de um contexto para outro pode causar dependências estruturais e pode prejudicar o funcionamento do outro. Mesmo com as suas relações, é importante lembrar que cada um deles deve funcionar de acordo com suas delimitações de contexto, ou seja, deve atuar de acordo com as limitações do seu próprio escopo, definindo o mínimo de dependência direta possível, mantendo o menor nível de acoplamento. No exemplo que envolvem os contextos de *Transporte* com *Agendamento de Entregas* não há problema com esse comportamento, pois consideram-se como contextos altamente interligados e que, em conjunto, resolvem o problema de entrega (de acordo com a modelagem da equipe).

Mas existem casos que um contexto não deve ser prejudicado com a mudança de outro, pois sua dependência não impacta no seu funcionamento ao ser propenso a modificações. Para isso, é definida uma estratégia denominada como **Camada anti-corrupção** (ilustrada com um retângulo na ponta de cada contexto, com o nome de ACL) capaz de fornecer os recursos de um contexto para outro a partir de abstrações e mapeamentos customizados para os interessados. No exemplo abordando o contexto de *Transporte* e *Autenticação*, a equipe de *Transporte* decide implementar uma camada anti-corrupção sob os recursos que precisa utilizar do contexto de *Autenticação*. Nesse caso, caso o contexto de *Autenticação* mude sua estratégia de verificação dos usuários, o contexto de *Transporte* não será afetado, pois há uma camada de abstração a mais na sua comunicação, sendo necessário apenas prestar manutenção nesse único ponto de acoplamento, caso necessário. A mesma ideia aplica-se ao relacionamento entre *Autenticação* e *Agendamento de Entregas* e no relacionamento entre *Trajetos* e *Transporte*. Em relação ao

Transporte, seu funcionamento não deve ser afetado caso o contexto de *Trajetos* mude seu algoritmo de roteamento e caminhos de trânsito, por exemplo.

Manter o ritmo de desenvolvimento seguindo os princípios definidos pelo DDD não é uma tarefa fácil. Existem diversos princípios e boas práticas que podem ser utilizados como auxílio a manter o foco destinado ao domínio e ao problema motivador (VERNON, 2013; EVANS, 2004). Os principais utilizados e recomendados pela literatura são:

- *Módulos*: utilizado para agrupar as grandes áreas de sua aplicação, a partir de domínios;
- *Repositórios*: aplicado para abstrair implementações de fontes de dados da aplicação, facilitando mudanças, manutenções e desacoplamento ao domínio;
- *Serviços/casos de uso*: utilizado para manter lógica de *aplicação* que não são pertencentes às entidades;
- *Eventos de Domínio*: aplicado na execução de processos a partir de eventos emitidos por alguma ação ou consequência no sistema (por exemplo, usuário autenticado e trajeto definido);
- *Entidades*: aplicadas para representar objetos únicos e com identidade dentro de um domínio, contendo sua própria regra de negócio e ciclo de vida; e
- *Objetos de valor*: objetos sem identidade, imutáveis e diretamente aplicados às entidades, definindo sua estrutura e composição.

Vários padrões podem ser aplicados para estruturação do sistema (VERNON, 2013), sendo necessária análise da equipe de desenvolvimento como um todo para melhor auxiliar na evolução do software.

Na próxima seção são descritas as atividades desenvolvidas neste trabalho para obter um estudo mais aprofundado com as técnicas, ideias e pensamentos presentes no DDD, incluindo o desenvolvimento de um sistema alvo aplicando os seus ideais comparando aos métodos de desenvolvimento tradicionais.

3 METODOLOGIA

Para atingir os objetivos propostos neste trabalho, várias etapas foram planejadas com a finalidade de moldar uma estrutura inicial de estudos para ser utilizada como base de pesquisa, conforme a seguir:

1. Especificação de um sistema alvo;
2. Implementação na versão tradicional;
3. Implementação na versão com *DDD*;
4. Análise dos prós e contras de cada uma das implementações.

Para este trabalho, o objeto de estudo foi definido por um sistema de gerenciamento e controle de barbearias. Sendo projetado, especificado e implementado pelo autor, esse sistema propõe-se a controlar e administrar todo o fluxo de agendamento e disponibilidade dos prestadores de serviço do estabelecimento. Seu funcionamento e regras de negócio são descritas na Seção 4.

Após a definição do sistema alvo, foram desenvolvidas pelo autor duas implementações, obedecendo as regras de negócio e rastreamento do seu respectivo documento de requisitos. Ambas as implementações possuem o mesmo funcionamento e os recursos disponibilizados para uso são idênticos. A diferença entre eles está no modo em que foram implementados. A primeira implementação, denominada `SEM DDD`, segue o modelo tradicional, sem utilizar conceitos de modelagem estratégica ou detalhes de regras de negócio aprofundadas no DDD, diferente da segunda implementação, denominada `DDD`, que segue as técnicas e modelagem apresentadas, descritas na Seção 2.

Com as implementações definidas, conduziu-se a realização de diversos comparativos para demonstrar pontos positivos e negativos relacionados a diversos conceitos, em cada implementação, mostrando os seguintes aspectos: *organização de código, desempenho, manutenibilidade, dificuldade de criação, curva de aprendizado, produtividade e escalabilidade*, descritas na Seção 6.

Na Seção 4, descrita a seguir, são apresentados todos os detalhes de funcionamento, implementação, tecnologias e regras de negócio do sistema alvo estudado neste trabalho.

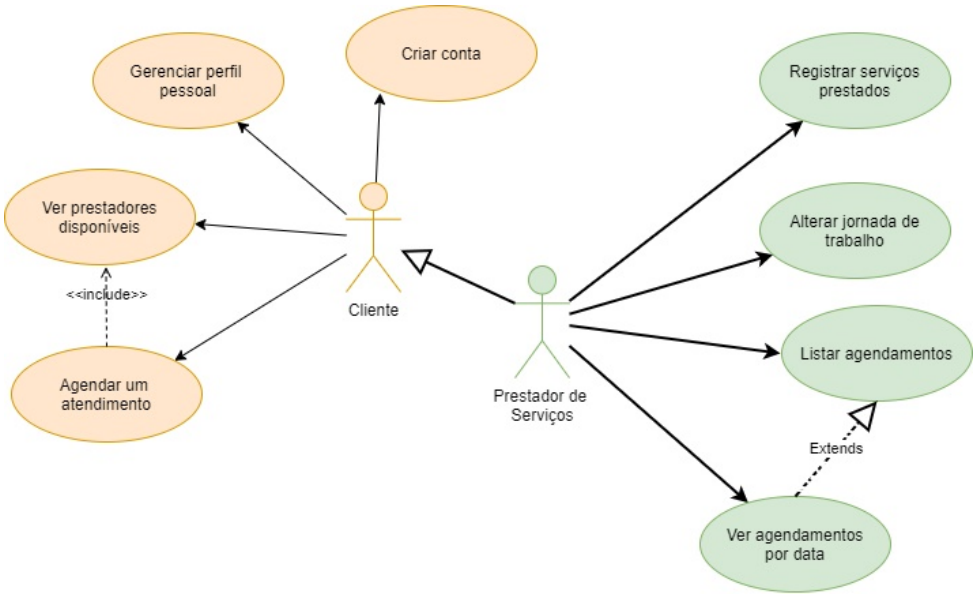
4 SISTEMA ALVO

Para realização dos estudos, foi-se necessária a definição de um sistema alvo com objetivo de ser utilizado como base de desenvolvimento e comparativos entre as implementações `SEM DDD` e `DDD`. Trata-se de uma aplicação voltada ao gerenciamento de barbearias, capaz de administrar o fluxo de agendamento, solicitações e atendimentos dos serviços que são prestados por elas. De modo geral, a aplicação é utilizada para controlar os horários de agendamento dos clientes com os prestadores de serviço que trabalham na barbearia, tornando possível que os clientes realizem agendamentos com os seus prestadores de preferência diretamente pelo sistema, além de conseguir visualizar a agenda de disponibilidade do prestador para atendimento no momento do agendamento.

4.1 Funcionalidades do Sistema

O sistema em questão é dedicado ao uso dos funcionários da barbearia (prestadores de serviço) e seus clientes, que vão marcar os horários de atendimento para um determinado serviço. A Figura 4.1 mostra o Diagrama de Caso de Uso da aplicação a partir de seus dois atores (`Clientes` e `Prestadores de Serviço`) e as funcionalidades presentes no sistema.

Figura 4.1 – Diagrama de Caso de Uso do Sistema Alvo



Fonte: do autor.

Cada funcionalidade presente na Figura 4.1 descreve em alto nível as possibilidades de uso que o sistema possui para realizar a gerência das tarefas, sendo detalhadas a seguir.

Criar conta: A criação de conta é destinada para ambos os atores do sistema, sendo utilizada para cadastrar clientes ou prestadores de serviço. É responsável por adicionar o registro da pessoa e permitir acesso às demais funcionalidades. Para o cadastro, é necessário informar o nome completo, endereço de e-mail e senha para ser realizado com sucesso. O endereço de e-mail deve ser uma informação única na aplicação, não permitindo valores duplicados. Ao finalizar a criação, o usuário fica bloqueado até efetivar a confirmação da conta, a partir do *link* de ativação enviado no endereço de *e-mail* informado no momento do cadastro. No caso do prestador de serviço, além das informações básicas utilizadas para cadastrar um cliente, também é solicitada a sua especialidade de atendimento.

Gerenciar perfil pessoal: Após a criação da conta, o usuário pode modificar suas informações de cadastro e também adicionar uma foto de perfil (denominada como avatar no contexto do sistema).

Ver prestadores disponíveis: Essa funcionalidade descreve a capacidade do cliente visualizar todos os prestadores de serviço disponíveis para atendimento na barbearia, visualizando a sua foto de perfil (avatar), nome completo e a sua especialidade. Esse recurso mostra os prestadores que podem realizar o atendimento de acordo com a data selecionada pelo usuário, caso o prestador tenha ao menos 1 (um) horário disponível.

Agendar um atendimento: A funcionalidade de agendar um atendimento oferece ao cliente a opção de marcar um serviço com um determinado prestador. Para tal, o usuário deve selecionar o prestador de sua preferência, além do horário desejado para realização do atendimento. É necessário que o horário em questão não esteja ocupado por outro agendamento e que se encaixe na jornada de trabalho do prestador, além de ser uma data válida (futura).

Registrar serviços prestados: A funcionalidade de registrar serviços prestados é destinada aos prestadores de serviço da aplicação, sendo responsável por permitir que o prestador finalize um atendimento que foi agendado pelo cliente após a sua execução. Para tal, o prestador deverá selecionar o agendamento que deseja (a partir do cliente ou data de atendimento) e marcar como finalizado.

Alterar jornada de trabalho: Essa funcionalidade permite que o prestador de serviço modifique seus dados de atendimento na plataforma, definindo quais são as horas que ele destina para agendamentos na plataforma, a duração de cada atendimento e o intervalo (se houver) entre cada atendimento. Essa informação é utilizada para mostrar aos clientes os horários disponíveis.

Listar todos os agendamentos marcados: Em um painel de visualização, o prestador de serviço será capaz de visualizar todos os agendamentos que tem marcado para um determinado mês, dentro de sua jornada de trabalho, mostrando cada um dos clientes que realizaram a solicitação e o horário marcado em determinada data.

Ver agendamentos por data: Semelhante à funcionalidade de listagem dos agendamentos marcados, descritos previamente, ao ver os agendamentos por data, o prestador de serviço poderá verificar os lançamentos que estão registrado para um determinado dia, separando pelo horário de agendamento e cliente a ser atendido.

4.2 Tecnologias e Funcionamento

O sistema previamente descrito foi construído de acordo com as técnicas de acesso aos dados presentes em aplicações destinadas ao ecossistema WEB. Classificado como uma API *(Application Programming Interface)* fornecedora dos dados e validação das lógicas de negócio em um modelo centralizado e estruturado de acordo com o modelo REST ***(RE**presentational **S**tate **T**ransfer)* e padronização de estados da aplicação a partir do modelo RESTful[1].

Por se tratar de uma API, a aplicação disponibiliza recursos, acessíveis via requisições HTTP de acordo com cada funcionalidade presente no sistema. A nomenclatura de requisições é descrita na Tabela 4.1. Cada um desses recursos é descrito a partir de seu corpo de requisição, URI de solicitação e método HTTP respectivo para cada tipo de ação, conectadas com cada funcionalidade específica do sistema (ou funcionalidades de suporte) descritas na Seção 4.1.

Ambas as implementações foram construídas com a linguagem TypeScript (MICROSOFT, 2021), um *superset* do EcmaScript/JavaScript amplamente adotado com a principal finalidade de tornar o JavaScript uma linguagem *parcialmente tipada*. Todas as tecnologias

[1] Modelo RESTful é um modelo de serviços que possuem comportamento arquitetural REST (FIELDING, 2000), utilizando os mesmos princípios de seu *design* e com comportamento específico a cada tipo de requisição ao recurso.

Tabela 4.1 – Lista de *endpoints* da aplicação

Recurso	Método	Descrição
`/users/`	`POST`	Cadastra um novo usuário na API.
`/users/avatar/`	`PATCH`	Muda o *avatar* de um usuário.
`/passwords/forgot/`	`POST`	Envia e-mail de recuperação de senha.
`/passwords/reset/`	`POST`	Redefine a senha de acordo com *token* recebido por e-mail.
`/sessions/`	`POST`	Realiza autenticação na aplicação, retornando um *token* de acesso.
`/profile/`	`GET`	Busca informações do perfil do usuário autenticado.
`/profile/`	`PUT`	Atualiza informações de cadastro do usuário autenticado.
`/appointments/`	`POST`	Realiza um agendamento.
`/appointments/:id`	`GET`	Obtém informações de um determinado agendamento.
`/appointments/:id/finish`	`POST`	Marca um agendamento como efetivado.
`/appointments/me/`	`GET`	Mostra agendamentos do prestador autenticado em uma determinada data.
`/providers/:providerId/month-availability/`	`GET`	Mostra os horários disponíveis de um prestador em um determinado mês.
`/providers/:providerId/day-availability/`	`GET`	Mostra os horários disponíveis de um prestador em um determinado dia.

Fonte: do autor.

utilizadas na API são descritas via requisições recebidas em um servidor WEB, executado sob a plataforma NodeJS (NODE.JS, 2021). As requisições enviadas para a aplicação são interceptadas e executadas de acordo com a implementação do sistema.

Como parte de infraestrutura, ambas as implementações possuem esquemas de persistência utilizando as seguintes tecnologias:

1. PostgreSQL: Banco de dados relacional contendo todos os dados da aplicação, com as tabelas, relacionamentos e atribuições necessárias para suportar o armazenamento dos registros que são gerenciados pelo sistema;

2. MongoDB: Banco de dados NoSQL responsável por armazenar dados temporários e notificações que devem ser mostradas aos usuários na aplicação; e

3. Redis: Banco de dados no modelo chave/valor utilizado como armazenamento de cache de consultas com processamentos pesados.

5 IMPLEMENTAÇÕES

Para o sistema alvo apresentado na Seção 4, duas implementações foram criadas para satisfazerem as regras do sistema apresentadas, trabalhando no mesmo formato de *API* e disponibilizando os mesmos *endpoints* e funcionalidades. Como mencionado no Capítulo 3, cada implementação possui um modelo de *design* e estratégias de solução dos problemas respectivo aos ideais e metodologia utilizadas, no qual cada uma pode ser modificada e gerenciada de acordo com a necessidade do projeto.

Ambas as implementações, tanto a `SEM DDD` quanto a `DDD`, utilizam as mesmas tecnologias para realizar o processamento e gerenciar informações em sua lógica interna. Considerando que o domínio da aplicação utilizada para acesso externo na Internet seja definido como <https://barbearia.com.br>[1], os seguintes passos demonstram o ciclo de vida de uma requisição enviada para a aplicação, em ambas implementações:

1. Uma requisição é enviada ao domínio <https://barbearia.com.br>, informando com um recurso específico, método HTTP[2], corpo de requisição e cabeçalhos (se houver);

2. A aplicação interpreta a combinação realizada pelo identificador no formato método-recurso informado na URI, encontra o controlador de rota específico e executa o código registrado;

3. Durante a execução do código registrado para o identificador, são obtidas as principais informações enviadas pela requisição (corpo, *cookies*, parâmetros de requisição e de cabeçalho) e realiza sua lógica de negócio; e

4. Após a execução, o sistema retorna ao solicitante uma resposta de acordo com o processamento realizado, seguindo o código de convenção de respostas do protocolo HTTP.

Como exemplo, a funcionalidade de cadastrar um novo usuário na aplicação, descrita no processo de Criar conta e com requisição presente na Tabela 4.1 pode ser explorada com os seguintes passos:

[1] Endereço utilizado apenas por praticidade, ao invés do formato mais conhecido pela comunidade e padronizado pelas versões, como <https://barbearia.com.br/api/versao>.

[2] Método HTTP é um verbo que indica uma ação presente a um recurso no protocolo HTTP (W3C, 1999). Os principais verbos existentes são: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `OPTIONS`, `CONNECT`, `TRACE` e `HEAD`.

1. Uma aplicação cliente envia uma requisição, do tipo POST, para o endereço <https://barbearia.com/users> com o corpo da requisição (no formato JSON[3]) contendo as informações do nome, e-mail e senha do usuário;

2. Requisição é interpretada pela aplicação a partir de seu identificador (URI + método HTTP) e executa lógica definida pela aplicação (`SEM DDD` ou `DDD`);

3. Lógica de validação dos dados é aprovada (tal como verificação de e-mail duplicado, por exemplo) e o usuário é adicionado na base de dados do sistema; e

4. Sistema retorna para a aplicação cliente, em formato JSON, representação do usuário criado e o código 200 de *status* da requisição (sucesso).

Todas as funcionalidades das implementações do sistema são acessíveis seguindo a lógica apresentada anteriormente. O diferencial entre o `SEM DDD` e `DDD` é a forma de lidar com a execução das regras de negócio, utilizando implementação com e sem as práticas do DDD. É detalhado, nas Seções 5.1 e 5.2, o processo de construção de cada uma, destacando as principais estratégias de solução e decisões de projetos tomadas.

5.1 Implementação `SEM DDD`

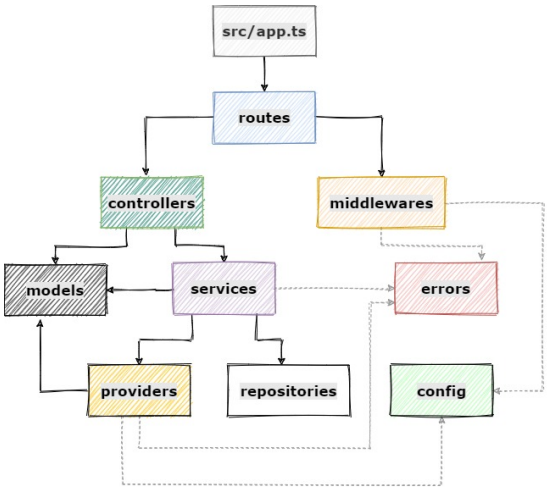
A implementação `SEM DDD` foi realizada utilizando os modelos mais adotados pela comunidade ao criar um sistema com especificações semelhantes ao sistema alvo. Nesse modelo, o foco da modelagem está descrito no formato dos componentes e suas principais responsabilidades. Sua estrutura é diretamente modelada às tecnologias e recursos utilizados para o funcionamento da aplicação.

A Figura 5.1 demonstra a forma em que o sistema foi projetado. Para esse modelo, cada módulo é representado pelos retângulos coloridos e suas comunicações pelas setas direcionadas. Cada um deles é responsável por gerenciar um tipo de componente que, ao final, resulta no processo completo de funcionamento do sistema.

As ações e funcionalidades presentes são definidas no módulo `services` (serviços), cujo objetivo é centralizar as regras de negócio e o acesso às outras camadas, por exemplo, para completar o processamento de uma determinada operação. O módulo `repositories` (repositórios) pode ser utilizado para manuseio dos dados que foram persistidos pela aplicação.

[3] O JSON (*JavaScript Object Notation*) é um modelo de documento estruturado, em formato de texto, baseado nos tipos de dados da linguagem JavaScript (ECMA, 2013).

Figura 5.1 – Modelagem da aplicação na implementação SEM DDD



Fonte: do autor.

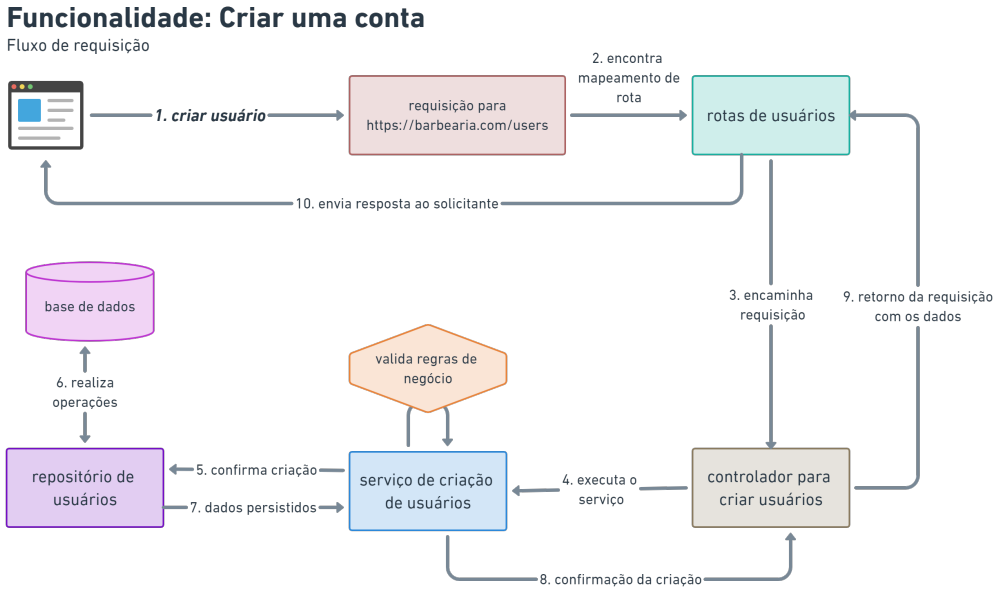
Por se tratar de uma API, há também o módulo `routes` (rotas), responsável por fazer o mapeamento de todas as rotas que o sistema disponibiliza para uso, além de controlar o que deve ser executado e suas respectivas dependências. A partir das rotas, o módulo `controllers` (controladores) é manuseado para garantir a execução do serviço específico e mapear os valores de entrada da requisição (parâmetros, cabeçalhos e corpo da requisição, por exemplo) e seus respectivos retornos, a partir dos códigos e estruturas de representação dos dados, se houver.

O código torna-se estável e bem estruturado de acordo com cada um dos módulos apresentados, responsáveis por auxiliar a equipe e o projeto fornecendo segurança e organização no entendimento da aplicação. Outros padrões de projeto, além dos que foram utilizados para cada módulo, podem ser aplicados com objetivo de facilitar o desenvolvimento e tornar um projeto mais confiável com boas soluções (GAMMA et al., 1994). A Figura 5.2 mostra o fluxo de execução de uma determinada funcionalidade quando a API recebe uma requisição a partir de uma ação realizada, seja de um usuário ou outro sistema, ocasionando a identificação das rotas e execução da solicitação para o controlador específico. O controlador fica responsável por fornecer os parâmetros necessários para a execução do serviço destinado a executar a operação, além de manipular os retornos obtidos a partir do processamento realizado.

5.2 Implementação DDD

Já a implementação DDD foi construída utilizando todas as técnicas de análise de modelos, domínios e principais funcionalidades da aplicação, conforme sugere o *Domain Driven*

Figura 5.2 – Exemplo de fluxo na Implementação SEM DDD para criação de uma conta.

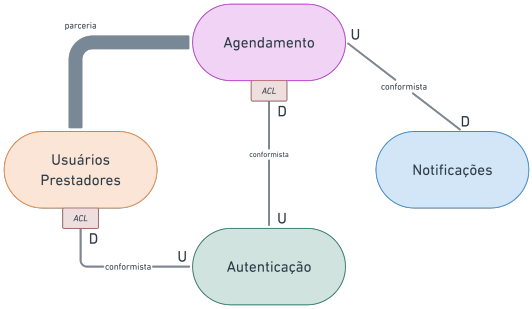


Fonte: do autor.

Design. Todo o fluxo foi baseado a partir do entendimento principal do Sistema Alvo, apresentado no Capítulo 4.

De acordo com as análises realizadas, foi possível definir um diagrama sobre a modelagem inicial do sistema, voltado aos conceitos aplicados com DDD, de seus *Contextos Delimitados* e suas respectivas interações, presente na Figura 5.3. O domínio principal, do gerenciamento de barbearias, possui os seguintes quatro subdomínios essenciais para cumprir o objetivo da aplicação: Agendamento, Usuários/Prestadores, Autenticação e Notificações.

Figura 5.3 – *Mapa de Contexto* para a barbearia.

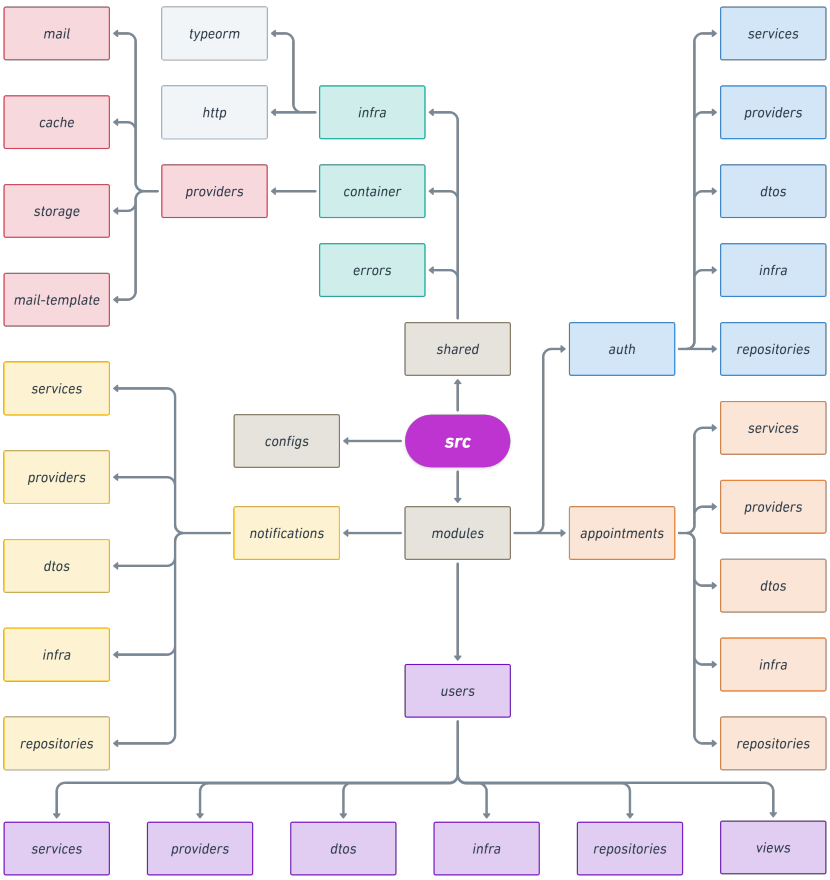


Fonte: do autor.

Cada um desses contextos possui responsabilidades específicas, com objetivos centralizados e completamente definidos. A comunicação entre eles é dada a partir dos conceitos presentes na Seção 2. A organização da arquitetura da Implementação DDD torna-se mais mo-

dularizada em relação aos seus principais domínios. A Figura 5.4 destaca as principais opções presentes dentro dessa implementação. Cada um dos diretórios possui responsabilidade específica em relação ao módulo que se encontra, sendo alguns semelhantes com a apresentação da Figura 5.1, mas aplicadas separadamente ao *Contexto Delimitado* que o envolve.

Figura 5.4 – Organização interna da implementação `DDD`.



Fonte: do autor.

Nessa estrutura há dois grandes pilares em sua representação: `modules` (para implementação dos módulos) e `shared` (para implementação dos recursos que são utilizados por todo o sistema). Para os recursos que são compartilhados (`shared`), existem os tratamentos de erros, códigos relacionados à infraestrutura (configuração de bibliotecas e serviços) e provedores de recursos (`providers`) que são utilizados por toda a aplicação e que não são propriedades de um único módulo, tal como serviço de envio e gerenciamento de *templates* de e-mail, controle de *cache* e armazenamento de arquivos em disco (em máquina local ou na Internet).

Já para os módulos do sistema (`modules`), estão definidas as implementações de serviços, camadas de acesso a dados, provedores de recursos (`providers`) de um módulo que poderão ser utilizados por outros e configurações específicas de recursos externos que farão

parte da infraestrutura, focado ao domínio em questão. Mesmo com o exemplo demonstrado na Figura 5.4, cada um dos módulos são isolados e independentes, podendo ter representações distintas entre si. As dependências entre os domínios são controladas via biblioteca de inversão de dependência disponibilizando as dependências solicitadas a partir dos elementos que foram registrados e configurados nos provedores de recursos (`providers`) presentes na aplicação (pelo `modules` ou `shared`), seguindo o acoplamento entre eles a partir das abstrações oferecidas pela camada de anti-corrupção.

Para o funcionamento da API, cada módulo da implementação registra, em sua camada de infraestrutura, todas as suas rotas que devem ser acessíveis. Com isso, a camada de recursos compartilhados (`shared`) também realiza o direcionamento das requisições para os seus respectivos módulos, presente na camada de infraestrutura. A Figura 5.5 mostra as etapas que são executadas quando uma ação é disparada ao sistema, semelhante ao descrito para a Figura 5.2, na Seção 5.1. A diferença está no formato percorrido durante a execução ao identificar os componentes dos módulos registrados para as ações do sistema, presentes em suas respectivas camadas internas. Até chegar no gerenciamento das rotas e controladores do domínio, por exemplo, é necessário o redirecionamento a partir das rotas gerais do sistema para continuar a execução.

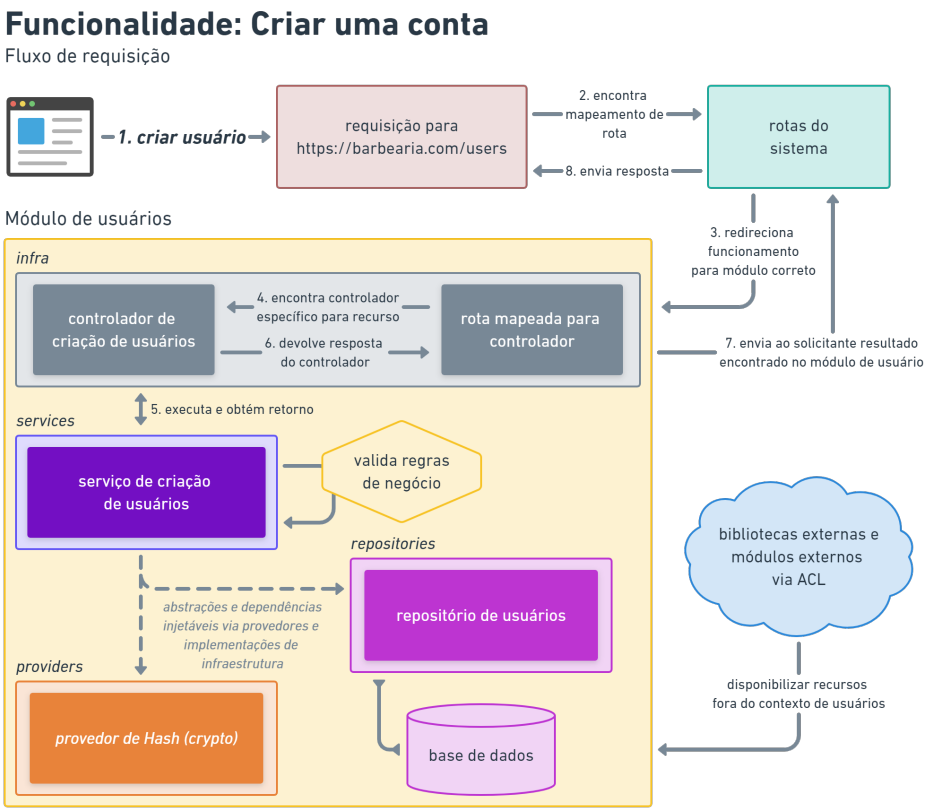
5.3 Considerações sobre as implementações

Tanto a implementação `SEM DDD` quanto `DDD`, apresentadas nas Seções 5.1 e 5.2, demonstram o funcionamento e detalhes em um maior nível de abstração, com a ausência de detalhes de código-fonte e operações reais. Tais seções transparecem os conceitos e estratégias definidas para cada implementação a partir de modelagens e fluxos de execução por meio de diagramas e exemplos.

No Capítulo 6, apresentado a seguir, são detalhadas diversas análises comparativas no que diz respeito aos aspectos chaves de cada implementação, tendo como objetivo catalogar e destacar seus prós e contras. Essas comparações englobam diversos aspectos presentes durante o desenvolvimento de software, sejam eles voltados às decisões de projeto ou viabilidade técnica de determinadas implementações, por exemplo, para cada uma das implementações apresentadas.

Nas análises técnicas, são apresentados trechos de código que podem compor as implementações, sem sua exibição completa ou de todos os seus respectivos arquivos. Os documentos

Figura 5.5 – Exemplo de fluxo na Implementação DDD para criação de uma conta.



Fonte: do autor.

utilizados neste trabalho, relativos a cada uma das implementações e documentos necessários em um projeto de software, encontram-se no repositório de código-fonte online, no seguinte endereço: <https://github.com/lhleonardo/tcc-ufla>.

6 COMPARATIVOS

As duas implementações apresentadas no Capítulo 5 são maneiras de desenvolvimento válidas para cumprir o principal objetivo: desenvolver uma solução para o sistema alvo, presente no Capítulo 4. Ambas tratam-se de pontos de vista específicos para as reais necessidades levantadas por uma equipe de desenvolvimento, além das regras presentes para o funcionamento do software. Cada uma dessas implementações possui seus prós e contras, pois não há uma regra exata que defina qual é o melhor modelo de implementação de software, seguindo o modelo utilizando `DDD` ou então o modelo `SEM DDD`. Na prática, um levantamento deve ser realizado com o objetivo de analisar a real necessidade de cada implementação.

As seções a seguir descrevem os prós e contras na utilização de certo modelo de implementação, de acordo com determinado contexto específico. Os seguintes comparativos são realizados: organização do código-fonte (Seção 6.1), desempenho (Seção 6.2), manutenibilidade (Seção 6.3), dificuldades de criação (Seção 6.4), curva de aprendizado (Seção 6.5), produtividade (Seção 6.6) e escalabilidade (Seção 6.7). Por fim, na Seção 6.8, é descrito um resumo dos principais aspectos que foram analisados.

6.1 Organização de código

A organização de código das implementações presentes neste trabalho, conforme apresentadas no Capítulo 5, torna-se eficiente para ambos, uma vez que o escopo do sistema alvo é pequeno. Para cada uma das implementações é possível realizar a seguinte análise, de acordo com seus prós e contras:

Implementação `SEM DDD`: Os módulos são separados de acordo com o componente[1] da arquitetura. Cada módulo possui uma função específica e direta, resultando na coesão das regras e orquestração das funcionalidades e recursos do sistema, pois sua centralização garante pequenas responsabilidades para cada componente. Porém, com o crescimento do sistema, a quantidade de arquivos e componentes específicos podem se tornar um empecilho para gerenciar. Para a pasta de serviços, por exemplo, seu crescimento é diretamente proporcional à necessidade de evolução e incremento das funcionalidades, gerando dificuldades na sua manutenção.

[1] Representa-se como componente o recurso capaz de realizar certa ação com seu objetivo específico (GHOLAMSHAHI; HASHEMINEJAD, 2019).

Implementação DDD: Utilizando o modelo gerenciado a partir dos domínios da aplicação, sua organização manipula os componentes baseado em módulos, centralizando cada uma das responsabilidades. Cada domínio (descrito na Seção 5.2) é representado dentro da sua pasta de `modules` e os recursos compartilhados pelo sistema em `shared`. A complexidade na inicialização de um projeto, além da alta curva de aprendizado para o seu entendimento, são fatores que podem prejudicar seu uso. Além disso, por apresentar domínios independentes, manter padrões arquiteturais e estruturais entre os domínios torna-se uma tarefa difícil de controlar. No entanto, a descentralização das regras em relação aos módulos e facilidade no entendimento ao comparar com uma aplicação já desenvolvida torna-se notável, pela facilidade de manutenção e crescimento.

A estrutura das implementações foram pensadas de acordo com o modelo implementado em questão.A relação de prós e contras, para o tópico em questão, pode ser resumida de acordo com a Tabela 6.1. Suas relações são classificadas de acordo com as duas implementações.

Tabela 6.1 – Relação de prós e contras: organização do código

Impl.	Prós	Contras
SEM DDD	• Facilidade de *setup* inicial do projeto; • Separação por responsabilidade; e • Modularizado via componentes.	• Complexo para gerenciar conforme evolução; • Módulos diretamente acoplados; e • Baseado em tecnologia, não em ecossistema.
DDD	• Modularização de acordo com domínio; • Propenso à evolução e manutenções; e • Facilidade de encontrar recursos específicos.	• Dificuldade para iniciar a implementação; • Curva de aprendizado e produtividade inicialmente baixa; • Precisam ter escopos delimitados corretamente; e • Manter padrões arquiteturais e estruturais entre os domínios.

Fonte: do autor.

6.2 Desempenho

O desempenho de um sistema é um ponto importante a ser avaliado por influenciar diretamente o seu funcionamento, independente da implementação escolhida, abordadas ou não neste trabalho. Avaliações de impactos de desempenho devem ser realizadas ao escolher um determinado tipo de implementação. Neste trabalho, considera-se para medir o desempenho o tempo gasto para executar uma determinada operação.

Como ambos os tipos implementados resolvem o mesmo problema, da mesma maneira, apenas com diferenças técnicas a partir de seu projeto arquitetural e fluxo de execução, torna-se um fator chave perceber o tempo gasto com objetivo de analisar a implementação em específico. Para ambas as implementações apresentadas neste trabalho, descritas no Capítulo 5, são analisados os seguintes casos:

Implementação `SEM DDD`: As dependências são diretamente conectadas e seus fluxos são parametrizados para seus respectivos componentes, conforme demonstra a Figura 5.2, seguindo o processo de chamadas tal como: requisição → rota → controlador → serviço → repositório.

Implementação `DDD`: o desempenho é propenso a ser pior por conta de suas camadas adicionadas e complexidades adicionais presentes na sua arquitetura. No entanto, pode ser um *overhead* consideravelmente pequeno, capaz de impactar minimamente a aplicação como um todo.

Para avaliações numéricas de desempenho, a Tabela 6.2 mostra um resumo de tempo gasto para as execuções de acordo com funcionalidades aplicadas em ambas as implementações, sendo executadas 2.000 (duas mil) vezes cada[2]. A medição das funcionalidades foi realizada a partir da interceptação de cada rota, aferindo o tempo de início e de conclusão do processamento da requisição. As execuções foram realizadas em uma máquina com processador Intel® Core™ i7-7700HQ, com memória RAM de 16GB (modelo DDR4 SDRAM de 2.400 MHz) e sistema operacional Ubuntu Server, versão 20.04 LTS.

A partir destes tempos mensurados, foram realizados testes estatísticos para obter maior confiabilidade das execuções. Após ser submetido ao teste de *Shapiro-Wilk*[3], foi possível con-

[2] Os registros de tempo mensurados podem ser vistos nos arquivos presentes no repositório apresentado na Seção 5.3.

[3] Teste estatístico destinado a validar se uma distribuição qualquer é semelhante a uma distribuição normal, a partir da validação da hipótese nula (SHAPIRO; WILK, 1965).

Tabela 6.2 – Tempo médio de execução gasto (em segundos) pelas implementações

Funcionalidade	SEM DDD	DDD
Criar Conta	0,718s	0,758s
Agendar um atendimento	1,581s	1,679s
Ver prestadores disponíveis	0,294s	0,330s
Listar todos os agendamentos marcados	0,188s	0,198s

Fonte: do autor.

cluir com 99% de confiança, de acordo com a Tabela 6.3, que as execuções não seguem uma distribuição normal, pois todos os *p-values* encontrados apresentam valores < 0.01.

Tabela 6.3 – Resultado obtido pelo teste de *Shapiro-Wilk*

Funcionalidade	*p-value*	W
Criar Conta	$3,689e^{-67}$	0,250
Agendar um atendimento	$4,604e^{-44}$	0,803
Ver prestadores disponíveis	$2,364e^{-66}$	0,281
Listar todos os agendamentos marcados	$6,559e^{-69}$	0,179

Fonte: do autor.

Logo, por conta dos registros não apresentarem normalidades, foi realizado o teste *Mann-Whitney (Wilcoxon rank-sum test)*[4] para realizar as comparações entre os tempos de execução das implementações DDD e SEM DDD, em cada uma das funcionalidades apresentadas. A Tabela 6.4 mostra a relação das funcionalidades e seus tempos de execução em um intervalo de confiança válido.

Tabela 6.4 – Intervalo de confiança das execuções a partir de *Mann-Whitney*

Funcionalidade	Int. de confiança SEM DDD	Int. de confiança DDD
Criar Conta	0.656s até 0.669s	0.688s até 0.702s
Agendar um atendimento	1.491s até 1.513s	1.575s até 1.598s
Ver prestadores disponíveis	0.281s até 0.285s	0.323s até 0.329s
Listar todos os agendamentos marcados	0.184s até 0.189s	0.193s até 0.196s

Fonte: do autor.

[4] Teste não-paramétrico utilizado para avaliar uma amostra não pareada e que não segue normalidade (SIEGEL; JR., 2006).

Sendo assim, com 99% de confiança, a hipótese nula que define que não há diferença de desempenho entre a implementação DDD e a SEM DDD foi rejeitada, possibilitando concluir os seguintes fatos para cada uma das funcionalidades:

1. *Criar conta*: com *p-value* de $5,106e^{-29}$, apresentou o tempo de execução entre 0.027s e 0,044s maior usando a implementação DDD em relação a SEM DDD, o que representa um *overhead* entre 3,82 a 6,09%;

2. *Agendar um atendimento*: com *p-value* de $2,196e^{-40}$, apresentou o tempo de execução entre 0.068s e 0.1s maior usando a implementação DDD em relação a SEM DDD, o que representa um *overhead* entre 4,31 a 6,33%;

3. *Ver prestadores disponíveis*: com *p-value* de $1,462e^{-39}$, apresentou o tempo de execução entre 0.018s e 0.029s maior usando a implementação DDD em relação a SEM DDD, o que representa um *overhead* entre 6,06 a 9,73%;

4. *Listar todos os agendamentos marcados*: com *p-value* de $9.623e^{-32}$, apresentou o tempo de execução entre 0.0064s e 0.0094s maior usando a implementação DDD em relação a SEM DDD, o que representa um *overhead* entre 3,43 a 5,04%;

Com isso, é possível perceber que, com a presença do DDD, há um aumento de tempo gasto em relação à implementação SEM DDD, destacada por conta dos *overheads* encontrados, resultante do aumento da complexidade da arquitetura do projeto, visando seguir os conceitos de domínios, modelagem estratégica e mapas de contexto presentes no DDD.

No entanto, em um sistema com maior escopo e que apresente funcionalidades com um maior nível de complexidade, o *overhead* encontrado na implementação DDD diminuirá em relação ao tempo gasto do processamento, visto que esse valor será consideravelmente pequeno, não influenciando assim o desempenho da implementação.

6.3 Manutenibilidade

No desenvolvimento de software é comum que o sistema e suas implementações necessitem ser mantidas no decorrer do tempo, seja com correção de erros ou adição de novas funcionalidades. Por isso, torna-se um ponto essencial a manutenibilidade do software e validar as suas facilidades para sanar as tendências de mudanças.

Cada uma das implementações apresentadas neste trabalho são analisadas no tratamento sobre os aspectos de manutenção e os respectivos prós e contras em sua utilização, presentes na Tabela 6.5. Suas particularidades e análises são descritas a seguir.

Implementação `SEM DDD`: Por conta do seu projeto arquitetural orientado aos componentes presentes da aplicação, a manutenção desses recursos torna-se fácil pela sua centralização e responsabilidade única para cada componente. Com isso, as modificações ou inclusões de novas funcionalidades são propensas a serem realizadas em uma ou mais categorias de componentes. As dificuldades aparecem conforme a escala de crescimento do sistema e mudanças das regras de negócios com muitas dependências entre si. Em uma base de larga escala, uma modificação pode, por exemplo, acarretar problemas em outros componentes, causando dificuldade nesse controle.

Implementação `DDD`: A segregação das responsabilidades do sistema em domínios na aplicação auxilia em aspectos de manutenção, já que domínio possui sua responsabilidade e funcionalidades separadas, além do baixo acoplamento entre os demais domínios, tornando possível a evolução ou modificação de um domínio, por exemplo, sem grandes impactos ou problemas que não estão previstos em suas dependências. No entanto, pela complexidade da arquitetura, para mantê-la e evoluí-la da forma correta (isto é, seguindo os padrões arquiteturais e mantendo uma base de código de qualidade) torna-se necessário um bom conhecimento da estrutura e princípios modelados em cada domínio, além de entender os conceitos do DDD para prosseguir com um desenvolvimento padronizado.

6.4 Dificuldade de criação

O começo de um projeto de software demanda um esforço inicial da equipe de desenvolvimento antes de começar a sua implementação, por conta de fatores que variam desde a definição arquitetural da aplicação até o seu modo de funcionamento, tecnologias, padronização, etc. Essa inicialização do projeto é uma etapa que demanda tempo e atenção de todos que estarão participando da construção do sistema.

Em relação às implementações, cada uma tem sua particularidade, necessidade e decisões de projeto que precisam ser analisadas durante sua criação, conforme descrito a seguir:

Tabela 6.5 – Relação de prós e contras: manutenibilidade

Impl.	Prós	Contras
SEM DDD	• Componentes e fluxos centralizados; • Arquitetura de fácil identificação de funcionalidades; e • Funcionalidades bem segregadas.	• Dificuldade em modificar componentes complexos e muito dependentes; • Modificações podem afetar, por consequência, outros componentes; e • Alterações e melhorias difíceis de lidar de acordo com a escala do projeto.
DDD	• Escopos de domínios bem definidos, facilitando modificações e implementações específicas; • Mudanças com pouco efeito colateral, por causa do baixo acoplamento; e • Facilidade no rastreamento de problemas pela alta coesão dos módulos.	• Necessário conhecimento da modelagem e estratégias utilizadas na sua criação; • Precisa de entendimento claro sobre o objetivo do software e suas ideologias; e • Requer conhecimento de DDD para garantir que o software continue seguindo os padrões e metodologias já definidos.

Fonte: do autor.

Implementação SEM DDD: Muito presente pela comunidade de desenvolvimento, alguns padrões de inicialização já existem para os modelos focados em componentes, como detalhado na Seção 5.1. Sua inicialização, por ser usual, há diversos modelos arquiteturais criados por outros desenvolvedores disponíveis para uso e que podem ser utilizados com o objetivo da implementação.

Implementação DDD: Por sua arquitetura ser altamente ligada aos detalhes da implementação, já que todo o software é modelado a partir do problema principal a ser resolvido (conforme descrito no Capítulo 2), uma solução inicial é elaborada com cautela e análises dos Contextos Delimitados e o Mapa de Contexto para comunicação entre os domínios. Ainda que todo o projeto esteja documentado e com escopo bem definido, tornam-se necessárias análises de viabilidade e escalabilidade de uma arquitetura voltada ao problema do software, tornando assim um modelo único e exclusivo para o problema em questão.

Diante disso, é importante destacar que cada análise da implementação é destinada a um problema específico e a sua escolha não torna os outros modelos existentes inviáveis ou inutilizáveis, uma vez que as decisões variam de acordo com as necessidades da equipe de desenvolvimento. A relação de prós e contras, para o tópico em questão, pode ser resumida de acordo com a Tabela 6.6, apresentando suas relações classificadas de acordo com ambas implementações.

Tabela 6.6 – Relação de prós e contras: dificuldade de criação

Impl.	Prós	Contras
SEM DDD	• Uso de modelos arquiteturais existentes; e • Facilidade de definição de arquitetura.	• Dificuldade em mudanças de arquitetura; • Altamente acoplado à tecnologia; e • Arquitetura muito rígida, dificultando modificações estruturais.
DDD	• Adaptável ao crescimento; e • Facilidade em realizar mudanças.	• Complexo para definir arquitetura; • Arquitetura muito rígida, dificultando modificações estruturais; e • Precisa de análises de acordo com o projeto.

Fonte: do autor.

6.5 Curva de Aprendizado

A construção de um sistema necessita de um aprendizado prévio sobre as estratégias e métodos utilizados em sua construção. Independente do modelo de implementação, apresentado ou não neste trabalho, a equipe de desenvolvimento precisa ter uma experiência básica ou um entendimento inicial sobre o problema e como solucioná-lo. Considerando que seja a primeira experiência, para ambas as implementações apresentadas no Capítulo 5, são descritos a seguir os principais pontos sobre a curva de aprendizado em relação a cada uma das implementações:

Implementação SEM DDD: A arquitetura desse tipo de implementação é caracterizada a partir dos seus componentes e suas responsabilidades, sendo utilizados para compor uma parte coesa

do sistema, conforme apresentado na Seção 5.1. Nesse modelo, torna-se necessário conhecer o funcionamento dos recursos da tecnologia (rotas, interceptadores, filtros e controladores, por exemplo) para a definição da arquitetura inicial. A partir disso, é possível separar as responsabilidades e compor as funcionalidades do sistema.

Implementação DDD: Diferente do modelo SEM DDD, que se torna amplamente conectada aos aspectos tecnológicos, o principal foco dessa implementação está na solução do problema da aplicação, cujo esforço se aplica na resolução das dificuldades que compõem o objetivo da construção do software. Logo, é importante o conhecimento das técnicas e conceitos presentes no DDD e seus recursos, conforme descrito no Capítulo 2, além de uma base sólida da definição do problema, pois ambos fatores podem apresentar dificuldades para um primeiro contato (isto é, para o entendimento do DDD ou do sistema).

A Tabela 6.7 resume os principais prós e contras encontrados sobre a curva de aprendizado para cada implementação, de acordo com as análises realizadas sobre o seu uso inicial em um desenvolvimento. Cada ponto descrito refere-se aos aspectos técnicos e decisões que são enfrentadas pela equipe de desenvolvimento durante a construção e aprendizagem em cada modelo de implementação.

6.6 Produtividade

Um dos tópicos relevantes para análise é a produtividade no desenvolvimento de uma aplicação. Para fins de estudo, considera-se esse tempo o período gasto para construir as duas implementações, descritas no Capítulo 5. Ao realizar a criação de um sistema, as etapas de desenvolvimento, como a elaboração dos requisitos, testes, homologações e manutenções compõem o ciclo de construção e, consequentemente, possuem ou gastam um tempo específico para serem realizadas. As análises a seguir demonstram como o uso das implementações podem impactar diretamente na produtividade, de forma analítica, no desenvolvimento do sistema, a partir das etapas iniciais de estruturação, construção e manutenções.

Implementação SEM DDD: No início da produção do sistema, o tempo de desenvolvimento é afetado de forma positiva, pois os modelos arquiteturais e estrutura dos componentes são comuns, possibilitando ganho de produtividade com o reaproveitamento de outros projetos ou

Tabela 6.7 – Relação de prós e contras: curva de aprendizado

Impl.	Prós	Contras
SEM DDD	• Facilidade no entendimento da arquitetura e seus componentes; • Equipes podem ser divididas para desenvolver os componentes; e • Arquitetura pode ser baseada em *templates*, permitindo focar em outros componentes.	• Necessário conhecimento da tecnologia e outros recursos utilizados; e • Equipe pode perder o foco na implementação do problema ao resolver pendências tecnológicas.
DDD	• O rendimento evolui com o progresso da definição do escopo do problema e definição dos domínios; • As equipes podem ser divididas no desenvolvimento dos domínios; e • Auxilia no entendimento do problema por conta do foco do DDD.	• Necessário conhecimento sólido no DDD e sua metodologia; • Maior foco na elicitação dos requisitos e objetivo do software; e • Requer conhecimento na comunicação entre os domínios e eventos do sistema.

Fonte: do autor.

guias de outros *templates* existentes, fornecendo à equipe a capacidade de destinar seu trabalho às funcionalidades e recursos do sistema. Após isso, a produção tende a ser mais produtiva pela estruturação inicial realizada e os escopos bem definidos e, consequentemente, beneficiando de forma proveitosa no tempo de desenvolvimento, pois as implementações futuras já possuem seu escopo e onde cada componente deve ser inserido ou modificado. Sobre as manutenções, o tempo gasto no desenvolvimento é beneficiado por conta da centralização e coesão dos componentes existentes, capacitando o rápido rastreamento de problemas e formas de adição de novas funcionalidades.

Implementação DDD: A produtividade no início do desenvolvimento é desfavorável em razão das modelagens necessárias para definição de todo o ecossistema da aplicação, pois é necessário realizar a organização inicial dos módulos e suas respectivas comunicações antes da construção efetiva do sistema. O processo de definição dos domínios, regras de negócio, comunicação e utilização do sistema são essenciais para esse modelo, afetando diretamente o tempo de desenvolvimento das atividades. No decorrer do desenvolvimento do sistema, a produtividade é

beneficiada de forma vantajosa, uma vez que a arquitetura estará bem definida e a equipe pode ser segmentada para construir os domínios definidos anteriormente, pois o escopo da aplicação foi definido corretamente. As manutenções também serão beneficiadas, visto que haverá facilidade no rastreamento e inclusão de novos recursos ou funcionalidades por possuir um escopo bem definido.

No entanto, ambas as implementações estão propensas a serem prejudicadas, em relação à produtividade, em qualquer uma das etapas citadas anteriormente. O problema se dá às falhas de modelagem e estruturação das funcionalidades, a partir de uma análise inválida do objetivo central da construção do sistema, motivando assim em uma reestruturação da arquitetura ou, no pior dos casos, o propósito do software.

A Tabela 6.8 apresenta os principais prós e contras do uso de ambas as implementações, de acordo com as análises realizadas anteriormente, focado no aspecto de produtividade de um sistema. Cada um dos pontos forma um conjunto de opções que devem ser levadas em consideração para decidir o uso de uma implementação.

6.7 Escalabilidade

Com o crescimento da demanda de um software, aumentando sua carga de trabalho em relação ao processamento, quantidade de clientes utilizando o sistema e disponibilidade dos serviços prestados, a escalabilidade da aplicação torna-se necessária para sanar essas necessidades. A partir do momento em que torna-se necessário escalar, as implementações acabam sendo analisadas para encontrar os formatos viáveis, sem afetar seu funcionamento e muito menos o desempenho da aplicação.

Existem várias formas de aplicar as técnicas de escalabilidade em uma aplicação. De acordo com os formatos apresentados por Tanenbaum (2007), os principais modelos de escalabilidade para esses tipos de serviço são a partir do dimensionamento vertical ou horizontal. No dimensionamento vertical, a máquina dedicada (hospedeira) tem suas características físicas, em especificações de hardware, modificadas com o objetivo de aumentar sua capacidade em executar uma determinada carga de trabalho, tal como aumento de memória, armazenamento ou processador. Já no dimensionamento horizontal, a aplicação é executada em diversas máquinas dedicadas, cujo objetivo seja dividir a carga de trabalho entre os serviços e as requisições,

Tabela 6.8 – Relação de prós e contras: produtividade

Impl.	Prós	Contras
SEM DDD	• Utilização de arquitetura pré existente a partir de *templates* semelhantes; • Equipes distribuídas para desenvolver em paralelo; e • Fácil de rastrear e moldar componentes novos ou que precisam ser modificados.	• Necessárias análises para definir o escopo do sistema; e • Fragilidade ao rastrear problemas na arquitetura, ao precisar modificá-la.
DDD	• Agilidade ao identificar problemas ou adicionar novas funcionalidades nos domínios; • Equipe segmentada para dividir a produção pelos domínios; e • Manutenção do sistema descomplicada, pelo conhecimento da equipe sobre a arquitetura e escopo de domínio;	• Tempo necessário para definir os domínios, comunicação e arquitetura do sistema; e • Fragilidade ao encontrar problemas na arquitetura e ao precisar modificá-la.

Fonte: do autor.

com a presença de sistemas capazes de realizar e gerenciar o balanceamento de carga e suas sincronizações necessárias (TANENBAUM; STEEN, 2007).

Ambas as implementações, descritas no Capítulo 5.2, são aptas a serem dimensionadas em ambos os formatos de escalabilidade, principalmente pela característica de funcionarem seguindo as regras e o estilo de arquitetura REST/RESTful, conforme demonstra o Capítulo 4, sem armazenar informações de estado internamente da aplicação (sem manter sessões ou *cookies*[5], por exemplo).

As análises a seguir destacam pontos estratégicos no que tange o interesse de escalabilidade para sistemas, dedicado para ambas às implementações.

[5] *Cookies* são arquivos de texto criados a partir da aplicação servidora para um determinado cliente, mantendo-os durante a comunicação e, assim, sendo capaz de identificar uma conexão, seus *status* e demais informações (CAHN et al., 2016).

Implementação SEM DDD: para o modelo de dimensionamento vertical, a implementação consegue ser gerenciada facilmente, a partir dos recursos novos que serão alocados para oferecer melhor capacidade de execução da carga de trabalho atual. Já para o modelo de dimensionamento horizontal, alguns pontos podem resultar em dificuldades. A sincronização entre os recursos e atividades (bancos de dados, armazenamento e concorrências de ações), deve ser levada em consideração desde o início do desenvolvimento, possibilitando assim a segmentação das ações, garantindo a integridade das operações.

Implementação DDD: semelhante à implementação SEM DDD, o dimensionamento vertical é viável de ser implementado, já que não há concorrência e detalhes a serem analisados no sistema, apenas a melhoria da máquina dedicada que está sendo utilizada para executar a aplicação. No entanto, mesmo com as mesmas ressalvas presentes para o dimensionamento horizontal proposta para a implementação SEM DDD, a presença de domínios não-dependentes possibilita seu dimensionamento por apresentar seus escopos desacoplados e capacidade de utilizações de recursos externos que não precisam ser compartilhados. Logo, é possível estruturar a aplicação para trabalhar de maneira distribuída, sem a necessidade de adaptações para garantir a integridade das operações (replicações e sincronização, por exemplo). Além disso, tirando proveito da característica descentralizada dos domínios, é possível utilizar a base de código para a construção de sistemas específicos para arquiteturas distribuídas, tal como implementação pelo modelo arquitetural de microsserviços[6].

A Tabela 6.9 apresenta os principais prós e contras relacionados a escalabilidade para cada implementação utilizada neste trabalho, retratadas no Capítulo 5. Observações quanto ao dimensionamento vertical não estão inclusos, já que ambas as implementações são aptas para trabalhar com esse modelo sem ressalvas. Cada um dos itens presentes devem ser levados em consideração para decidir a melhor forma de utilizar uma implementação capaz de ser escalável, de acordo com a necessidade do projeto.

[6] A arquitetura de microsserviços é formada a partir de pequenas aplicações desacopladas com escopos bem definidos que compõem um sistema ao serem utilizadas em conjunto, a partir das comunicações entre os serviços (MARTIN; JAMES, 2014).

Tabela 6.9 – Relação de prós e contras: escalabilidade

Impl.	Prós	Contras
SEM DDD	• Capaz de operar com dimensionamento horizontal a partir de máquinas idênticas, com as mesmas especificações; • Diminuição da espera de requisições, quando sobrecarregado, com balanceamento; e • Quando implementado, capacitado a gerenciar replicações das bases e recursos externos.	• Propenso a falhas de replicação dos dados e sincronização; • Dificuldade de escalar recursos específicos apenas para uma máquina, no dimensionamento horizontal; e • Queda no desempenho pela complexidade adicionada para gerenciar os serviços escalonados.
DDD	• Capacidade de ser dimensionado horizontalmente de acordo com as regras de cada domínio; • Propenso a dimensionar recursos específicos não compartilhados entre os domínios, pertencente por apenas um; e • Fácil de migrar para arquiteturas distribuídas (como microsserviços) por conta das particularidade dos domínios.	• Replicações complexas em domínios com muitos recursos externos compartilhados; • Domínios não projetados para serem distribuídos; e • Queda no desempenho pela complexidade adicionada para gerenciar os serviços escalonados.

Fonte: do autor.

6.8 Considerações sobre as análises

Foram realizadas e demonstradas, neste capítulo, várias análises e comparativos para identificar os pontos positivos e negativos para cada uma das implementações trabalhadas neste trabalho, descritas no Capítulo 5. Cada aspecto analisado tem a sua importância durante o desenvolvimento de um sistema e representa um objeto de inspeção utilizado no momento de escolha de alguma implementação.

Os aspectos apresentados são iniciativas para critérios de decisão para utilizar uma determinada implementação. Cada um tem sua respectiva importância e os benefícios de sua utilização. A análise da implementação, então, torna-se mais precisa quando realizada pela

equipe, visto que a decisão é tomada de acordo com a necessidade no desenvolvimento e o sistema alvo.

Para ilustrar sua utilização, são descritos dois perfis de equipes de desenvolvimento que poderão ser beneficiadas em utilizar a implementação SEM DDD ou DDD, de acordo com os aspectos apresentados neste capítulo:

Perfil para SEM DDD: Para equipes que, independente da escala do sistema, possui definição completa do escopo e complexidade do sistema. Não precisa ser um sistema que terá grande crescimento. Facilidade em manter projetos simples e seguir boas estruturas arquiteturais para equipes que têm conhecimento do ecossistema e propósito do software.

Perfil para DDD: Para equipes que trabalham com domínios complexos e que estão sendo evoluídos com o decorrer do projeto, mas necessitam de atenção aos requisitos e regras de negócio. Com a presença de inúmeras regras de negócio e comunicação entre os domínios da aplicação, facilitando o rastreamento, manutenção e crescimento do software. Focado em desenvolver a solução independente de infraestrutura e tecnologias.

Ambas as implementações apresentadas neste trabalho são modelos arquiteturais e de controle de projeto válidas para utilização no desenvolvimento de software. O propósito deste capítulo não está relacionado à indicação de uma implementação específica, mas sim, no suporte de escolha da equipe para, a partir de cada uma das análises, decidir qual modelo é interessante para o problema que busca ser solucionado.

No Capítulo 7, descrito a seguir, são apresentados trabalhos presentes na literatura que complementam os propósitos apresentados para cada aspecto dos comparativos. Cada um destes acrescentam no entendimento e fluxo de desenvolvimento para sistemas criados com o DDD, com diferentes níveis de complexidade e escopos distintos, cujo foco esteja associado às melhores soluções voltadas ao DDD e as características de implementação de determinados sistemas.

7 TRABALHOS RELACIONADOS

Neste capítulo são descritos trabalhos que fazem o uso ou se beneficiam do DDD em diversas aplicações voltadas ao desenvolvimento de software. Tais trabalhos foram selecionados com o objetivo de complementar os conhecimentos e aplicações do DDD em diversas áreas, tendo em vista a utilização de técnicas com maior nível de complexidade, aplicados a problemas de escopo mais amplos quando comparados com este trabalho.

Conforme apresentado no Capítulo 2, o DDD tem como objetivo trabalhar sob o problema principal que o software deve resolver. No entanto, pelo fato de a análise do problema conter uma complexidade alta, desde a elicitação dos requisitos até a construção do sistema, existem trabalhos cuja finalidade esteja em facilitar esse processo.

Existem ferramentas capazes de facilitar o processo de criação dos modelos de domínio que serão utilizados na solução do problema e, consequentemente, na construção do software. Dentre essas, existe um projeto capaz de definir e gerenciar os escopo dos módulos do sistema a partir da utilização de anotações[1], denominada como *aDSL (annotation-based Domain Specific Language)*, capaz de construir o modelo de domínio a partir das análises realizadas, utilizando abordagens generativas em conjunto das regras definidas presentes na modelagem do software (LE; DANG; NGUYEN, 2020). Tais regras são utilizadas para criar um modelo dimensionado dos principais recursos definidos (controladores, recursos de visualização, acesso aos dados, etc.) utilizando todas as informações presentes nas anotações. Ainda assim, existem outros modelos capazes de incrementar o processo de desenvolvimento com ferramentas auxiliares para o DDD, tal como a presença de geradores para modelos de domínio a partir de uma nomenclatura de modelagem de negócio pré-definida (CHEN et al., 2019).

Conforme mencionado na Seção 6.7, ao tratar sobre a escalabilidade das implementações apresentadas neste trabalho, a capacidade de desenvolver sistemas cuja arquitetura adequa-se na utilização de microsserviços, com modelagens resultantes do DDD, são benéficas, e cada vez mais utilizadas.

O motivo da adoção de microsserviços com o DDD se dá a possibilidade da modelagem dos domínios da aplicação em pequenos serviços separados e com escopos específicos. Segundo Rademacher et al. (2018), tal associação é resultante das características descentralizadas e escopos delimitados dos domínios serem equivalentes ao conceito de mínimo compartilhamento

[1] Anotação é um recurso presente em várias linguagens, como em Java, capaz de adicionar meta-dados no código-fonte, podendo ser lidos e processados posteriormente por alguma aplicação, seja um compilador ou interpretador, e executar alguma ação (NOSÁL'; SULÍR; JUHÁR, 2016).

possível de funcionamento e especificações, presente no projeto e construções de microsserviços e facilitando sua construção (RICHARDS, 2016).

No entanto, a construção dos microsserviços relacionados a modelagens com DDD torna-se um desafio em relação a suas diversas particularidades. De acordo com Rademacher et al. (2018), a modelagem dos contextos não é uma tarefa direta e necessita de atenção para não prejudicar o sistema como um todo. Alguns desafios, como a definição dos contextos da aplicação que deverão ser implementados como microsserviços, a modelagem dos componentes da infraestrutura da arquitetura e a definição e divisão das equipes para cada uma das implementações estão inclusos nas dificuldades de seu uso.

Portanto, quando aplicado com um propósito bem definido, o DDD provê inúmeros benefícios ao desenvolvimento do software (EVANS, 2004; VERNON, 2013). Inúmeras aplicações com regras de negócio de alta complexidade se beneficiam da utilização de seus ideais. Dentre essas, existe o caso de sucesso no desenvolvimento de um sistema de escrituras para administração de terras, especificamente aplicado nas dependências da Holanda. Segundo Oukes et al. (2021), já existia um sistema em funcionamento, entretanto, tal aplicação não estava preparada para acompanhar as novas regras impostas com o crescimento de demandas de terras. O novo sistema, implementado utilizando o DDD, foi modelado para o funcionamento dos órgãos presentes em utilização de uma especificação já existente para solução de escritura das terras, denominada como LADM (*Land Administration Domain Model*, ou Modelo de Domínio de Administração de Terras) e foi capaz de realizar o mesmo processo de gerenciamento, além da definição dos escopos com os princípios do DDD (OUKES et al., 2021). Tal trabalho pode ser utilizado como base para o entendimento das análises de um sistema com um escopo maior e complexo.

No capítulo a seguir são descritas as principais considerações e aplicações práticas das técnicas apresentadas neste trabalho, a partir dos comparativos apresentados entre as implementações do sistema alvo, definido no Capítulo 4, e suas respectivas análises.

8 CONSIDERAÇÕES FINAIS

O objetivo deste trabalho foi o estudo aprofundado sobre o *Domain Driven Design*, analisando seus principais pilares e metodologia de implantação para realizar um comparativo com outro modelo de implementação, frequentemente utilizado pela comunidade, definido na Seção 5.1. Além disso, a partir dos estudos, foi possível analisar a aplicação das técnicas de desenvolvimento utilizadas pelo DDD na construção de um software, cujo foco é descrito nas melhores práticas de criação, sustentação e evolução de um sistema.

A análise de um sistema e modelagem para os conceitos do DDD, mostrados na Seção 5.2, a partir de um sistema alvo, permitiu o entendimento prático dos conceitos apresentados por Evans (2004) em seu propósito ao idealizar os conceitos do DDD, esclarecendo tópicos específicos para tomada de decisões ao utilizá-lo. Com duas implementações para um mesmo sistema alvo, foi possível realizar um comparativo que fosse possível ser utilizado como base de seleção entre as implementações. Os tópicos que sofreram comparações são de alta importância para uma escolha de implementação, propondo uma distinção concisa para o público que tem interesse do uso do DDD. Os comparativos descritos no Capítulo 6 destacam os prós e contras para cada aspecto destacado, contrapondo ambas as implementações e detalhando cada uma delas.

A Tabela 8.1 destaca os principais pontos, de forma resumida, que foram concluídos a partir das análises, possibilitando o entendimento dos benefícios e malefícios ao escolher uma determinada implementação. Para tal, os itens são exibidos e classificados de acordo com a sua capacidade em promover um determinado aspecto, destacando-os como: promover completamente (✓), promover parcialmente quando aplicado algum esforço (✓*) e não promover (✗) um determinado aspecto. Todos os tópicos inclusos se referem aos principais pontos sobre organização do código, desempenho, manutenibilidade, dificuldade de criação do sistema, curva de aprendizado, produtividade e escalabilidade das aplicações.

Com isso, é possível concluir que a utilização do DDD é uma boa escolha. As vantagens em sua utilização em escopos de grande escala estão presentes pela construção de um sistema manutenível, evolutivo e rastreável. A metodologia presente auxilia no entendimento do problema em sua essência, possibilitando assim que o foco da equipe de desenvolvimento seja direcionado na construção do sistema. Mesmo com algumas limitações, como maior complexidade em iniciar e entender uma arquitetura, além do *overhead* de desempenho presente, seu ganho a longo prazo é benéfico para a saúde do sistema e evolução da equipe.

Tabela 8.1 – Resumo das análises para ambas implementações

#	**Aspecto esperado**	SEM DDD	COM DDD
1	Início rápido da estrutura do projeto	✓	✗
2	Funcionalidades separadas em componentes específicos	✓	✓
3	Utilização de conceitos de modularização e reúso de código	✓	✓
4	Baixo acoplamento entre os recursos e funcionalidades	✗	✓
5	Independência de tecnologias externas	✗	✓
6	Código que seja fácil de manter	✓	✓
7	Foco no melhor desempenho das operações	✓	✓
8	Curva de aprendizado (curto prazo)	✓	✗
9	Curva de aprendizado (longo prazo)	✓	✓
10	Facilidade em redefinir pedaços da arquitetura	✗	✓
11	Simplicidade (não precisa de muito estudo)	✓	✗
12	Facilidade para primeira experiência	✓	✗
13	Facilidade em gerenciar escopos complexos	✓*	✓
14	Capacidade de ser escalável	✓*	✓
15	Aptidão para utilizar microsserviços	✗	✓
16	Capacidade de dividir a equipe para implementar pedaços do sistema	✓	✓
17	Tempo de desenvolvimento curto	✓	✗
18	Conhecimento aprofundado sobre o problema	✗	✓

Fonte: do autor.

Para trabalhos futuros, pesquisas podem ser realizadas para complementar o conhecimento adquirido neste estudo. O desenvolvimento de um sistema de maior porte, comparado ao apresentado neste trabalho, é sugerido para possibilitar a análise dos prós e contras na utilização do DDD aplicado em uma aplicação de maior escopo, realizando os comparativos necessários para obter um bom guia de escolha.

Em complemento aos benefícios de utilização do DDD, pode ser realizada uma pesquisa com diversos profissionais na área, sejam especialistas de negócio ou desenvolvedores, para validar se foi realmente benéfica a utilização do DDD em seus projetos e, para os desenvolvedores

que anseiam sua utilização, se escolheriam aderir a utilização do DDD em seus projetos atuais ou futuros.

Além disso, a construção de um sistema aplicando técnicas avançadas de desenvolvimento e implementação de software, descritas por Vernon (2013), utilizando padrões de projetos mais focados ao DDD, tais como *Event Sourcing* e *Command Query Responsibility Segregation* (CQRS), conceitos de modelagem (entidades, objetos de valor, serviços de domínio e agregados) e modelos arquiteturais beneficiados por sua utilização (tal como microsserviços) é recomendada para avaliar aspectos ao desenvolver um software com maior escopo, sendo validadas por profissionais da área que utilizam o DDD.

REFERÊNCIAS

CAHN, A. et al. An empirical study of web cookies. In: **25th International Conference on World Wide Web (WWW16)**. [S.l.: s.n.], 2016. p. 891–901.

CHEN, C. et al. Business generator model based on domain driven design. **Journal of Physics: Conference Series**, v. 1168, 2019.

ECMA. **The JSON data interchange format**. 2013. Disponível em: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>.

EVANS, E. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. [S.l.]: Addison-Wesley, 2004.

FIELDING, R. **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doutorado) — University of California, 2000.

FOWLER, M. **Bliki: Bounded Contexts**. Web, 2014. Disponível em: <https://martinfowler.com/bliki/BoundedContext.html>.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.]: Addison-Wesley Professional, 1994.

GHOLAMSHAHI, S.; HASHEMINEJAD, S. M. H. Software component identification and selection: A research review. **Software: Practice and Experience**, v. 49, n. 1, p. 40–69, 2019.

LE, D. M.; DANG, D.-H.; NGUYEN, V.-H. Generative software module development for Domain-Driven Design with annotation-based domain specific language. **Information and Software Technology**, v. 120, 2020.

MARTIN; JAMES. **Microservices**. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. [S.l.]: Prentice Hall PTR, 2008.

MICROSOFT. **TypeScript - JavaScript With Syntax For Types.** Microsoft Corporation, 2021. Disponível em: <https://www.typescriptlang.org/>.

MILLETT, S.; TUNE, N. **Patterns, principles, and practices of Domain-Driven Design**. [S.l.]: John Wiley & Sons, 2015.

NODE.JS. **Node.js Official Page**. OpenJS Foundation, 2021. Disponível em: <https://nodejs.org/>.

NOSÁL', M.; SULÍR, M.; JUHÁR, J. Language composition using source code annotations. **Computer Science and Information Systems**, v. 13, n. 3, p. 707–729, 2016.

OUKES, P. et al. Domain-driven design applied to land administration system development: Lessons from the netherlands. **Land Use Policy**, v. 104, 2021.

RADEMACHER, F.; SORGALLA, J.; SACHWEH, S. Challenges of domain-driven microservice design: A model-driven perspective. **IEEE Software**, v. 35, n. 3, p. 36–43, 2018.

RICHARDS, M. **Microservices vs. Service-Oriented Architecture**. [S.l.]: O'Reilly Media, Inc, 2016.

SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). **Biometrika**, v. 52, n. 3-4, p. 591–611, 1965.

SIEGEL, S.; JR., N. J. C. **Estatística não paramétrica para ciências do comportamento**. [S.l.]: Artmed, 2006.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas distribuídos: Princípios e Paradigmas**. [S.l.]: Pearson Prentice Hall, 2007.

VERNON, V. **Implementing Domain-Driven Design**. [S.l.]: Addison-Wesley Professional, 2013.

W3C. **HTTP/1.1: Method Definitions**. World Wide Web Consortium (W3C), 1999. Disponível em: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>.