CENTRO UNIVERSITÁRIO DO PARÁ - CESUPA
ESCOLA DE NEGÓCIOS, TECNOLOGIA E INOVAÇÃO - ARGO
CURSO DE ENGENHARIA DA COMPUTAÇÃO

MURILO DIAS FERREIRA FARIAS
RUBENS MITITAKA KISHIMOTO

IMPLEMENTAÇÃO DA MODELAGEM DOMAIN DRIVEN DESIGN EM SISTEMA DE AGENDAMENTO WEB: UM RELATO DE EXPERIÊNCIA

BELÉM
2021

MURILO DIAS FERREIRA FARIAS
RUBENS MITITAKA KISHIMOTO

IMPLEMENTAÇÃO DA MODELAGEM DOMAIN DRIVEN DESIGN EM SISTEMA DE AGENDAMENTO WEB: UM RELATO DE EXPERIÊNCIA

Trabalho de conclusão de curso apresentado à Escola de Negócios, Tecnologia e Inovação do Centro Universitário do Estado do Pará como requisito para obtenção do título de Bacharel em Engenharia da Computação na modalidade MONOGRAFIA.

Orientador: Prof. Moshe Dayan Sousa Ribeiro

BELÉM
2021

MURILO DIAS FERREIRA FARIAS
RUBENS MITITAKA KISHIMOTO

IMPLEMENTAÇÃO DA MODELAGEM DOMAIN DRIVEN DESIGN EM SISTEMA DE AGENDAMENTO WEB: UM RELATO DE EXPERIÊNCIA

Trabalho de conclusão de curso apresentado à Escola de Negócios, Tecnologia e Inovação do Centro Universitário do Estado do Pará como requisito para obtenção do título de Bacharel em Engenharia da Computação na modalidade MONOGRAFIA.

Data da aprovação: / /
Nota final aluno I:_________
Nota final aluno II:_________
Nota final aluno III:_________

Banca examinadora

____

Prof. Moshe Dayan Sousa Ribeiro
Orientador e Presidente da banca

____

Prof(a). Daniele Moura de Queiroz
Examinador

RESUMO

Esta monografia apresenta um relato de experiência na implementação da metodologia Domain Driven Design (DDD) em um sistema de agendamento. O problema surge na forma de aplicação da metodologia baseada em domínio, pois o DDD oferece um conjunto de boas práticas e padrões, mas suas determinações são muito gerais, que podem mudar dependendo da complexidade do negócio. Diante das considerações supracitadas surge o questionamento dado por este trabalho que é como implementar o DDD no desenvolvimento de um sistema de agendamento web para uma barbearia. O objetivo é desenvolver uma metodologia baseada no DDD para criação de sistema de agendamento web oferecendo como resultado as etapas de desenvolvimento utilizando as modelagens estratégica e tática, desde o estudo do domínio, separação de contextos, mapa de contextos, padrões do DDD e arquitetura de software. Logo, a proposta final deste trabalho proporciona todas as fases do desenvolvimento do método Domain Driven Design para um negócio específico com intuito de demonstrar uma forma de realizar este projeto tendo em mente que é possível aplicar em inúmeros cenários com diferentes complexidades de negócio na construção de um software.

Palavras-chave: Domain Driven Design. Modelagem de Software. Desenvolvimento de Software.

ABSTRACT

This final paper presents an experience report on the implementation of the Domain Driven Design (DDD) methodology in a scheduling system. The problem arises in the application of the domain-based methodology, as DDD offers a set of good practices and standards, but its determinations are very general, which can change depending on the complexity of the business. In view of the mentioned considerations, the question raised by this work arises, which is how to implement DDD in the development of a web scheduling system for a barber shop. The objective is to develop a DDD-based methodology for creating a web scheduling system offering as a result the development stages using strategic and tactical modeling, from domain study, context separation, context map, DDD patterns and architecture of software. Therefore, the final proposal of this work provides all stages of the development of the Domain Driven Design method for a specific business in order to demonstrate a way to carry out this project keeping in mind that it is possible to apply in countless scenarios with different business complexities in the development of a software.

Keyword: Domain Driven Design. Software Modelling. Software Development.

SUMÁRIO

1 INTRODUÇÃO **7**
1.1 SITUAÇÃO PROBLEMA 7
1.2 OBJETIVOS DO ESTUDO 8
1.3 JUSTIFICATIVA 8
1.4 METODOLOGIA DA PESQUISA 9
1.4 ESTRUTURA DO TRABALHO 9
2 REFERENCIAL TEÓRICO **10**
2.1 REVISÃO DA LITERATURA 10
2.1.1 Introdução ao Domain Driven Design **10**
2.1.2 Modelo de Negócio e Levantamento de Requisitos **12**
2.1.3 Modelagem Estratégica e Modelagem Tática **13**
2.1.4 Contextos Limitados **15**
2.1.5 Linguagem Ubíqua **16**
2.1.6 Integração de Contextos **16**
2.1.7 Arquitetura de Software **17**
2.1.8 Padrões (Building Blocks) do DDD **19**
2.2 TRABALHOS RELACIONADOS 22
3 DESENVOLVIMENTO DA PROPOSTA **24**
3.1 VISÃO GERAL DA IMPLEMENTAÇÃO DDD 24
3.2 ENTENDER O NEGÓCIO E LEVANTAMENTO DE REQUISITOS 26
3.3 IDENTIFICAR ESPAÇO DO PROBLEMA 31
3.4 DEFININDO ESPAÇO DA SOLUÇÃO 32
3.5 MAPA DE CONTEXTO 34
3.6 COMPARAÇÃO DAS POSSIBILIDADES DO MODELAMENTO ESTRATÉGICO 36
3.7 DEFINIÇÃO DA ARQUITETURA 37
3.8 APLICAÇÃO DE PADRÕES DO DDD 38
4 ANÁLISE DOS RESULTADOS 43
5 CONSIDERAÇÕES FINAIS 44
REFERÊNCIAS **46**

1 INTRODUÇÃO

Estamos em um período interessante, um número significativo de empresas estão se tornando negócios de software, independente da indústria, empresas de ramos tradicionais como manufatura e finanças precisam se tornar empresas de software, pois o software deixou de ser uma forma de operar o negócio para se tornar um negócio em si (DELGADO, 2018). Essa demanda pela aplicação de tecnologias da informação para aumentar a competitividade no mercado está mudando os meios de trabalho atuais. Sendo implantado em diferentes áreas de negócio com alta complexidade, sistemas que crescem cada vez mais e junto com ele a necessidade de um nível de gerenciamento superior. Logo, fica imprescindível maior controle e compreensão de todo o escopo, vindo assim a utilização do *design*.

1.1 SITUAÇÃO PROBLEMA

Segundo Kamakura (2013), o design é o projeto, o plano e a forma que vai obter durante o desenvolvimento de software. O design está amplamente associado com o planejamento de atividades, se tornando assim um guia que organiza e encaminha todos os aspectos do negócio como uma utilidade a nível de construção do software. Um código que não segue nenhuma boa prática é um projeto difícil de manter e incrementar, surge então o método DDD (Domain Driven Design) que se tornou uma alternativa para nortear o desenvolvimento e gerar mais valor ao negócio.

O Domain Driven Design (DDD) é uma metodologia de desenvolvimento de software que visa implementar um design baseado em modelos que refletem as competências da organização (DELGADO, 2018). Servindo como caráter auxiliar, pois orienta a organização em compreender o que deve se distinguir, buscando o fator de distinção através da identificação do domínio principal, e isso só é possível com colaboração intensa de especialistas de domínio e programadores.

O problema surge na forma de aplicação da metodologia baseada em domínio, pois o DDD oferece um conjunto de boas práticas e padrões, mas suas determinações são muito gerais, contando também que para diferentes projetos com suas complexidades a forma de abordagem deve mudar. Para ter melhor proveito do caso em específico de negócio, existem considerações particulares a serem feitas e a proposta deste trabalho se trata de um relato de experiência da aplicação deste método em um contexto específico, o processo de agendamento em barbearias. Diante das considerações supracitadas surge o questionamento dado por este trabalho, como

implementar o DDD no desenvolvimento de um sistema de agendamento web para uma barbearia?

1.2 OBJETIVOS DO ESTUDO

1.2.1 Objetivo Geral

Desenvolver um modelo em software de um sistema de agendamento web utilizando o Domain Driven Design.

1.2.2 Objetivo Específicos

- Analisar etapas de desenvolvimento da abordagem DDD.
- Modelar o software conforme as informações obtidas no domínio do negócio.
- Aplicar as soluções analisadas para o andamento do sistema de agendamento.

1.3 JUSTIFICATIVA

Devido seu caráter abstrato e geral, o DDD é mais bem compreendido a partir de exemplos e aplicações. Já existem diversas aplicações do DDD, nas mais variadas áreas, como na autenticação de eleitores para sistemas eletrônicos de votação (BASED et al., 2016), em sistemas para a administração de terras na Holanda (OUKES et al., 2021) e até na análise e design de um sistema de manutenção de equipamentos (SHENGLIN et al., 2019) , e, nesse sentido, esse trabalho propõe aumentar esse acervo de experiências expondo mais uma implementação, para uma organização hipotética que atua em um domínio corriqueiro, na prestação de serviço por agendamento web.

Dentre várias abordagens de desenvolvimento possíveis, por que então escolher o DDD? Uludağ et al. (2018) discute que o DDD provê conceitos básicos para a arquitetura do projeto que podem ser benéficos não só para times ágeis, mas para o programa todo da organização no geral. Assim, o DDD, a partir de suas modelagens básicas, oferece o suporte necessário para que times ágeis consigam trabalhar em empreendimentos de larga escala em sintonia com os tomadores de decisão que também conseguem se beneficiar diretamente a partir da modelagem estratégica do DDD, principalmente.

Por meio de uma revisão bibliográfica, Vicente (2018) mostrou como a linguagem ubíqua, que é um dos fundamentos do DDD, pode ser uma poderosa ferramenta contra o problema de comunicação dentro do projeto de desenvolvimento de software que ele aponta, a

partir de estatísticas do PMI (*Project Management Institute* Brasil) de 2010, como um dos principais problemas que leva a falha de projetos de desenvolvimento de software. Logo, utilizar o DDD pode facilitar a comunicação dentro do projeto, ajudar no processo de trabalho dos tomadores de decisão e dar ferramentas importantes para que o projeto possa escalar de forma consistente.

1.4 METODOLOGIA DA PESQUISA

A metodologia utilizada consiste na pesquisa aprofundada sobre o tema e sua aplicação em um caso escolhido em específico para este trabalho, passando por etapas de busca de trabalhos correlatos, livros e artigos para embasamento teórico, planejamento necessário para o desenvolvimento e por fim executar na construção de um sistema de agendamento web em barbearias.

A pesquisa para a construção deste trabalho foi realizada mediante o trabalho e o livro, Domain Driven Design: atacando as complexidades no coração do software (2003), de Eric Evans, autor e idealizador da metodologia baseada em domínio, junto com o livro Implementando o Domain Driven Design (2013) do autor Vaughn Vernon, uma obra mais focada na aplicabilidade e nas formas que a metodologia pode tomar dentro do projeto. É importante citar também, as bases de dados virtuais como monografias e artigos científicos do Google Acadêmico e as fontes em matérias e resenhas de sites eletrônicos que auxiliaram na estruturação e aprimoramento de toda a metodologia deste trabalho.

1.5 ESTRUTURA DO TRABALHO

O trabalho consiste em um primeiro capítulo contendo a introdução em relação ao desenvolvimento de software, onde é apresentado o tema, a situação problema, a pergunta que norteou o estudo e os objetivos. No segundo capítulo, abordou-se o referencial teórico e as suas categorias – Introdução ao Domain Driven Design; Modelo de negócio e levantamento de requisitos; Modelagem estratégica e modelagem tática; Contextos limitados; Integração de contextos; Arquitetura de software e Padrões do DDD– finalizando este capítulo com um tópico sobre os trabalhos correlatos. No terceiro capítulo, apresenta-se o desenvolvimento da proposta/metodologia, baseado nas fundamentações teóricas vistas no capítulo anterior e utilizando como modelo o sistema online de agendamento em barbearias. O quarto capítulo discorreu sobre a análise dos dados obtidos durante o desenvolvimento do sistema, enquanto o quinto capítulo de considerações finais, trata de uma discussão e relato do experimento realizado em que é retratado as dificuldades e análises feitas diante deste trabalho.

2 REFERENCIAL TEÓRICO

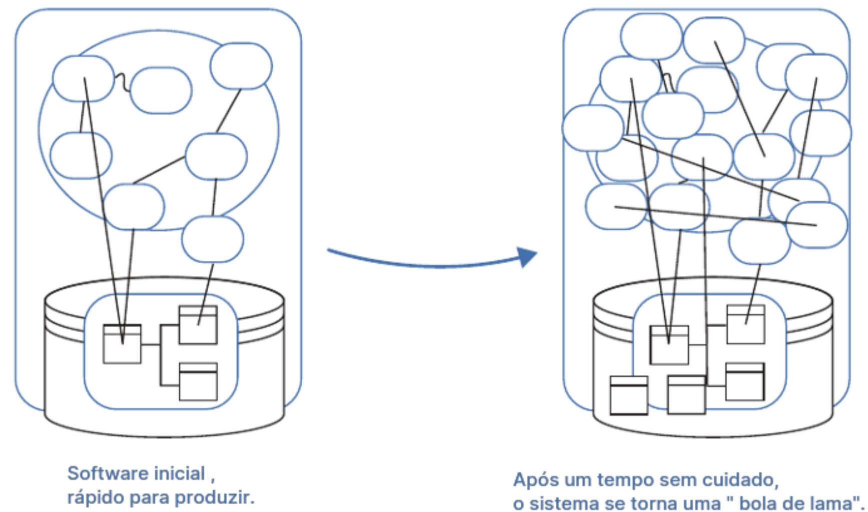
2.1 REVISÃO DA LITERATURA

2.1.1 Introdução ao Domain Driven Design

O Domain-Driven Design (DDD) é uma abordagem para desenvolvimento de software focado no domínio do problema, o seu maior objetivo é fazer com que o sistema mantenha uma qualidade de estado com fácil manutenção e compreensão, tanto pela equipe de desenvolvedores, quanto pelos interessados no negócio em que software está inserido (EVANS, 2003). O objetivo do Domain Driven Design é focar na relação de desenvolvimento com o domínio do problema, na qual o software será inserido, tornando as equipes de desenvolvedores e especialistas do negócio com alto nível de compreensão e entregando um software de alta qualidade.

A consolidação do tema foi feita pelo autor do livro Domain Driven Design: atacando a complexidade no coração do software de Evans em 2003 e mostra como esse método é mutável e aplicado para aplicações com mais complexidade. Atualmente, no mundo dos microsserviços é amplamente utilizado por causa do seu grande potencial, já que há uma grande harmonia entre as duas (WILLIAMS, 2019). Segundo Millet (2015), o Domain Driven Design se trata de um conjunto de práticas e princípios para criação do modelo de software, denominado por ele como uma "filosofia de desenvolvimento". Tais princípios são aplicados desde a modelagem do negócio conforme seus requisitos específicos, onde o software será utilizado, até em padrões de desenvolvimento a nível de código. Um grande problema em sistemas feitos sem um modelo de negócio bem definido é o padrão BBoM (Grande bola de lama do inglês Big Ball of Mud). A arquitetura BBoM, representada na Figura 1, é descrito como um software que cumpre seus requisitos inicialmente, porém, quando são implementadas mudanças e adições de funcionalidades, tais deveres se tornam cada vez mais complexos e aumentam as dificuldades de ler e entender o código feito para a aplicação (MILLETT, 2015). Tornando assim, basicamente, um sistema que faz algo útil e funcional, mas com alta dificuldade de entender. Logo, as consequências serão o número reduzido de manutenções, baixa escalabilidade e maleabilidade para mudanças.

Figura 1 - Big Ball of Mud



Fonte: Adaptado de Costa; Hild(2019)

Quando o assunto é DDD, logo se pensa na arquitetura, nos padrões (*building blocks*) do DDD e nos *designs patterns* relacionados. Porém, para trabalhar com DDD precisamos desenvolver diferentes habilidades, aprender a modelar de maneira efetiva, e, para isso, precisamos nos tornar o que o Eric Evans chama de *Knowledge Crunchers* — processadores de conhecimento — alguém preparado para obter, filtrar e estruturar uma grande quantidade de informação (HECK, 2018). Para organizar tanta informação é necessário ter as definições bem claras, como identificar os elementos e planejar a melhor alternativa. Tais conceitos essenciais para a modelagem são:

Domínio

Uma esfera de conhecimento, influência ou atividade. A área de assunto para a qual o usuário aplica um programa é o domínio do software (EVANS, 2003). O domínio é o coração do negócio que está sendo trabalhado, um conjunto de ideias que são aplicadas diante um problema estabelecido, conhecimentos e processos realizados que dão razão do negócio existir. Toda empresa tem um domínio principal e sem esse parâmetro, não há razão para o desenvolvimento do software.

Modelo

Um modelo é, de acordo com Evans (2003), um sistema de abstrações que descreve aspectos selecionados de um domínio e pode ser usado para resolver problemas relacionados a esse domínio. Com um modelo conseguimos simplificar o negócio para que seja possível a

construção do software de forma colaborativa entres as equipes, visando boa comunicação e garantindo que as funcionalidades especificadas sejam, de fato, implementadas.

O modelo é evolutivo, ou seja, a cada interação entre especialistas de domínio e a equipe técnica, o modelo se torna mais complexo e expressivo, mais cheio em utilidades, e os desenvolvedores transferem essa fonte de valor para o software (HECK, 2018). Dessa forma, o software cresce gradativamente de acordo com as demandas do negócio e mandados para o código, o sistema cresce em complexidade, mas bem estruturado para entendimento geral do projeto.

Linguagem Ubíqua ou Onipresente

A linguagem ubíqua, ou onipresente, é um dos pilares do DDD e descreve precisamente quais objetos, comportamentos e casos de uso estão em seu projeto para que todos possam entendê-lo, desde os proprietários do projeto sem nenhum conhecimento técnico até o desenvolvedor recém-integrado (CRIULANSCY, 2019). Seu propósito é definir os termos comuns do projeto para evitar falhas na comunicação entre todas as partes envolvidas. A linguagem ubíqua é refinada conforme o seu conhecimento do domínio avança.

Contexto Delimitado

Um contexto define um limite conceitual claro em torno de um aplicativo como um todo ou partes dele. Fora do contexto limitado, a mesma palavra pode significar, e certamente significa, algo totalmente diferente (CRIULANSCY, 2019). A linguagem ubíqua citada anteriormente só faz sentido em um contexto específico.

2.1.2 Modelo de Negócio e Levantamento de Requisitos

De acordo com Macedo (2009), sem a compreensão do negócio não existe a possibilidade de implementar o DDD. Dentro do time de projeto existem basicamente dois papéis essenciais: o time de desenvolvedores e os especialistas de domínio que entendem e vão guiar o time de desenvolvimento tirando dúvidas e esclarecendo as regras e processos desde a nomeação dos termos até a motivação do software existir. Dessa forma, o maior desafio neste momento é o entendimento do negócio analisado por parte dos desenvolvedores, pois na construção de software é preciso conhecer bem os detalhes, todas as regras de negócio e fazer um levantamento preciso de seus requisitos.

O processo de levantamento de requisitos é a fase mais importante na evolução de um sistema, pois se torna a estrutura fixa na qual toda a solução vai ser construída. Cabe ao analista

ou especialista fazer um bom e cuidadoso trabalho, para garantir alta confiabilidade no andamento do projeto (SOUSA, 2018). Por isso, o especialista de domínio deve enxergar todos os contextos possíveis que possam causar erros durante a criação das funcionalidades. Tais requisitos são separados em três tipos diferentes como os requisitos funcionais, que tratam de funções que o sistema deve executar; os requisitos não-funcionais, que lida com os recursos como restrições, segurança, validações entre outras características essenciais do sistema e por último as regras de negócio que são as políticas e normas que o sistema criado deve obedecer.

Para realizar um levantamento de requisitos detalhado e conciso, o especialista dispõe de algumas técnicas que são utilizadas de acordo com o perfil do cliente como entrevistas, questionários e as mais usadas técnicas de prototipação (SOUSA, 2018). É comum o uso de prototipação onde os requisitos obtidos até certo momento se tornam um protótipo da solução e apresentado ao cliente para que seja validado. A vantagem desta técnica é prover ao cliente uma prévia da solução final, podendo assim, após sua análise sugerir mudanças e implementações imediatas e não durante o desenvolvimento do software.

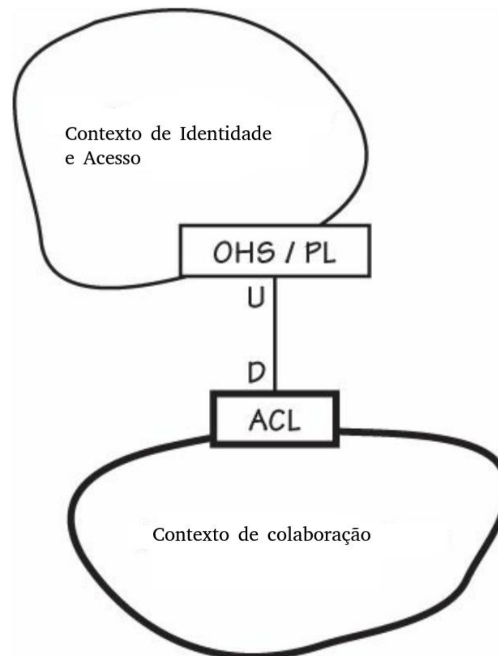
2.1.3 Modelagem Estratégica e Modelagem Tática

Modelagem Estratégica

Consiste em categorizar quais são as áreas onde os problemas, a serem resolvidos pelo software, encontram-se, estabelecer limites conceituais onde certos modelos do domínio são aplicáveis (VERNON, 2013), e integrar esses limites conceituais em uma solução que trabalhe de forma consistente e coerente com os objetivos da organização. Com a modelagem estratégica é possível visualizar de forma mais clara os objetivos específicos que cada um desses limites procura obter. Esses limites estabelecem regiões chamadas de contextos limitados que são soluções em software para um certo conjunto de problemas que a organização deve solucionar. Esse trabalho irá descrever logo adiante de forma mais detalhada o seu significado.

Na Figura 2, há um contexto de identidade e acesso que se preocupa em identificar cada usuário de um sistema e gerenciar seus privilégios nele, e um contexto de colaboração que se preocupa em estabelecer que usuário foi o autor de que tipo de atividade e quais outros usuários ele estabeleceu uma colaboração. Ambas as ideias trabalham juntas no sentido de gerar uma experiência coerente e consistente ao usuário final, mas são modeladas de forma delimitada no sentido de produzir um sistema mais modular e mais simples de visualizar as responsabilidades e dependências. Os elementos na figura 2 que conectam os contextos serão explicados na seção 2.1.6 que fala na integração entre contextos.

Figura 2 - Exemplo da integração de contextos.



Fonte: Vaughn Vernon (2013)

Modelagem Tática

A modelagem Tática atua dentro de um contexto limitado. Quando o DDD é aplicado para desenvolver uma solução da organização em software, o modelo lógico, que a organização usa para descrever os processos dessa solução, deve ser mapeado para um modelo em software a fim de ser implementado. A modelagem tática transforma o modelo lógico em modelo em software por meio do uso de blocos construtivos padrões que incorporam neles fundamentos e boas práticas do DDD.

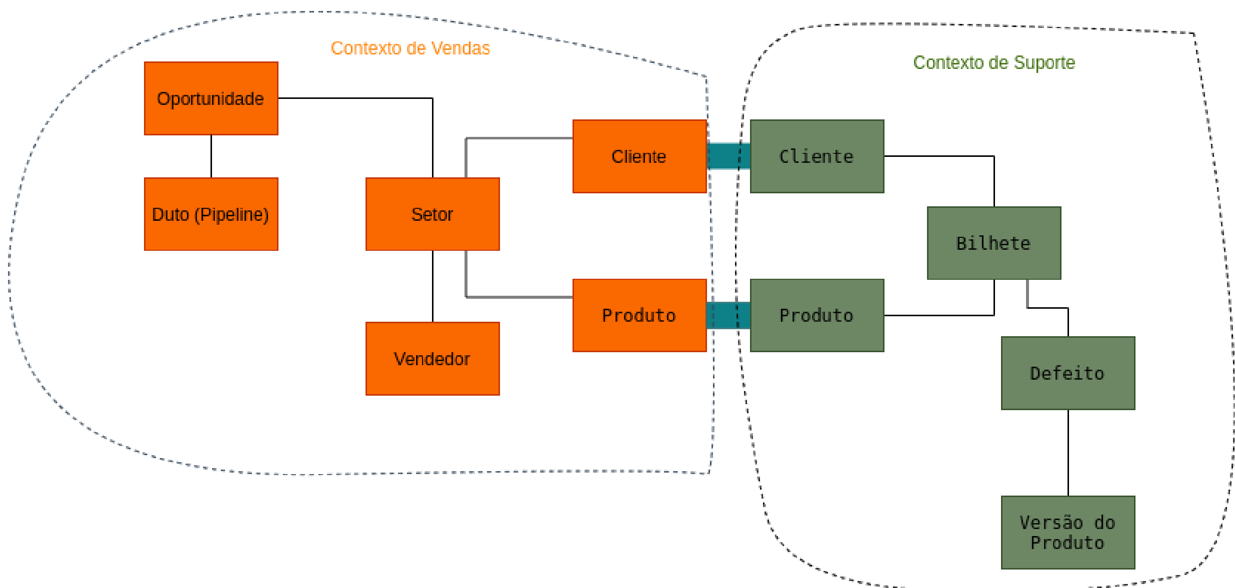
Como exemplo de bloco construtivo básico temos o conceito de entidade. Um conceito do domínio deve ser desenvolvido como entidade quando a sua individualidade é algo importante a ser considerada (VERNON, 2013). Isso é comum quando é um conceito em que suas mudanças de estado devem ser rastreadas durante todo seu ciclo de vida e quando é importante diferenciá-lo de qualquer outro objeto, como uma ordem de compra em um sistema de mercado online. Assim, essas estruturas vão ter pelo menos algum atributo que a identifique unicamente dos outros objetos de mesma natureza. Normalmente, isso é feito colocando atributos identificadores únicos em classes que são modeladas como entidades.

2.1.4 Contextos Limitados

São limites conceituais onde determinados modelos do domínio são aplicáveis. Eles fornecem um contexto para a linguagem que é utilizada no seu próprio modelo em *software*, a linguagem ubíqua. A Figura 3 mostra um exemplo para entender a sua ideia. Nela temos dois contextos, que são duas soluções em software específicas para áreas diferentes em que uma organização atua. No contexto de suporte, a ideia de produto (*product*) preocupa-se só com as características básicas da natureza do produto importantes para a sua atividade, no seu caso, poderia ser implementada com a inclusão de atributos como, além do seu nome, a sua versão e detalhes de possíveis problemas comuns desse produto, enquanto que “produto”, no contexto de vendas, teria um atributo referindo-se ao preço próprio e talvez informações sobre disponibilidade no estoque.

Também poderíamos isolar as informações sobre o estoque dos produtos em outro contexto responsável só pelo gerenciamento de estoque e assim utilizar algum método de integração para repassar essa informação para o contexto de vendas quando necessário. O Importante é saber que cada contexto vai englobar um determinado conjunto de objetos do modelo necessários para a sua solução e mesmo quando eles apresentam objetos similares, os dados que ele contém em cada contexto vão se referir somente às informações necessárias para as atividades que ele desenvolve.

Figura 3 - Exemplo de entidades similares em contextos diferentes.



Fonte: Adaptado de Brito (2019)

2.1.5 Linguagem Ubíqua

Essa linguagem é produto da forma como o time de desenvolvimento e os detentores do conhecimento sobre o domínio projetam a solução em software. Ela é desenvolvida para ficar o mais alinhada possível com a forma que os conhecedores do domínio descrevem as regras de negócio e seus elementos básicos, porém é comum que haja muitas incongruências com a forma que essas regras são modeladas de forma lógica.

As interações do time de desenvolvimento e dos experts da área do problema ajudam a encontrar esses pontos problemáticos e outros pontos cegos, e devem contribuir para gerar um modelo para a organização mais claro e consistente do domínio da aplicação. Por isso diz-se que o DDD ajuda a criar valor verdadeiro de negócio para a organização (EVANS, 2012), pois ele contribui para elucidar, validar, remodelar o modelo do domínio que a organização detém, e gera um modelo em software com uma linguagem semelhante à utilizada para descrever esse domínio pelos experts.

2.1.6 Integração de Contextos

Quando há mais de um contexto limitado solucionando o conjunto de problemas a serem resolvidos pela organização em software, essas soluções, normalmente, precisam comunicar-se de alguma forma. Nesse sentido, os diferentes "projetos" criam relações de integração para poder trocar os componentes particulares dos seus modelos e assim formar um espaço de solução para a organização coeso e consistente com as regras de negócio de todo o domínio de atuação dela.

Cada contexto tem uma forma de expor os dados dos seus elementos do modelo, o que cria diversas relações diferentes de integração. Pela necessidade de receber dados sobre objetos de outros contextos para realizar a sua lógica de atuação própria, os contextos produzem dependências entre si e as equipes de cada contexto precisam estabelecer maneiras de suprir essas dependências por meio de comprometimentos que geram estruturas particulares, para a integração entre as soluções, dependendo do modo que esse comprometimento ocorre. Esse comprometimento pode ser mútuo ou apenas unilateral, com uma das equipes podendo ter que se submeter totalmente à forma com que a outra expõem seus objetos do domínio.

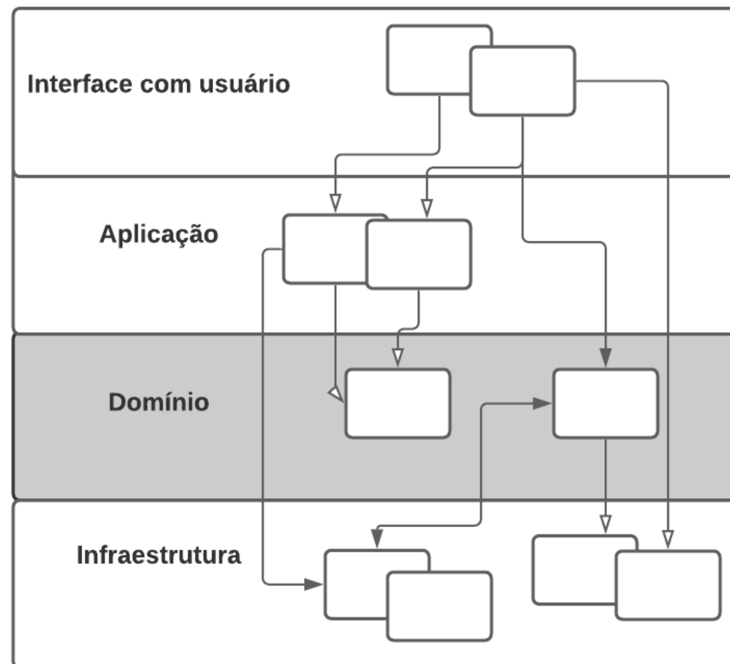
A Figura 2 mostra a representação de uma relação de integração entre dois contextos. O nome dessa representação chama-se mapa de contextos e serve para esboçar de forma simplificada como os diversos contextos, dentro de um espaço de problema da organização, estão conectados. Nele, os símbolos U e D determinam qual contexto depende do outro nessa

relação. O U é o que determina a forma como os seus dados vão ser expostos e o D é o que deve se conformar com essas determinações para utilizar esses dados internamente. Os retângulos juntos a cada contexto ligados por uma semirreta, representam a forma como um contexto relaciona-se com o outro. Nessa figura, o contexto de identidade e acesso integra-se com o contexto de colaboração disponibilizando um serviço aberto (*Open Host Service*), como por exemplo uma *web service*, e junto ele publica uma linguagem (*Published Language*) para que se comuniquem com esse serviço e ele possa compartilhar seu modelo com o outro contexto da forma necessária para a organização. A ACL (*Anti-Corruption Layer*) é uma relação de integração definida por uma camada de tradução de um modelo de um contexto para o outro que consegue compatibilizar as diferenças entre os dois modelos sem que haja uma grande cooperação entre as equipes dos dois contextos para isso.

2.1.7 Arquitetura de Software

Segundo Evans (2003), considerando que o desenvolvimento deve manter seu foco no modelo de domínio, a primeira etapa deve ser admitir uma arquitetura multicamadas com foco em 4 vertentes do sistema sendo eles: Interface, Aplicação, Domínio e Infraestrutura. A camada de interface apresenta e recebe as informações que seguem para a aplicação, responsável este em orientar os objetos da camada de domínio com a finalidade de atender os casos de uso do sistema. A camada de domínio contém as regras de negócio de todo o domínio do sistema. Por último, a camada de infraestrutura fica encarregada com as funcionalidades ditas genéricas que auxiliam as outras anteriores mediante integrações com persistência em base de dados, serviços e-mail ou comunicação com aplicativos terceiros.

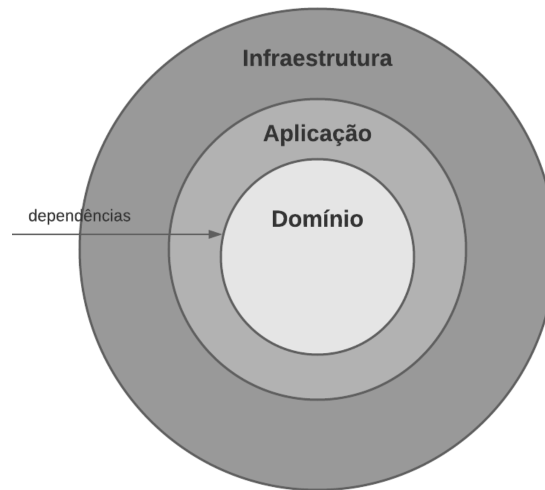
Figura 4 - Arquitetura em camadas.



Fonte: Adaptado de Evans (2003)

O Domain Driven Design não define a necessidade de uma arquitetura de camadas (as quatro citadas por Evans por exemplo), pelo contrário, o arquiteto de software tem total liberdade para decidir a melhor alternativa que satisfaça as demandas da aplicação, podendo utilizar modelos simples como *Table Module Pattern*, clássicos como *Onion Architecture* ou até os populares Microsserviços (PIRES, 2016). Apesar da flexibilidade do DDD, é preciso realizar boas práticas de um código limpo e conciso para que tenha como resultado algo independente e testável em diferentes cenários. Para isso, é preciso valorizar as regras de dependência mencionadas por Martin (2012) em *The Clean Architecture*, esta regra diz que as dependências do código-fonte só podem apontar para dentro. Nenhuma informação do círculo interno pode saber absolutamente de algo do círculo externo. Isso inclui funções, classes variáveis ou qualquer outra entidade de software nomeada. Logo, é incluído soluções como a injeção de dependências e a separação de camadas mediante suas responsabilidades, sendo esta última afetando ou não a arquitetura e o design de software.

Figura 5 - Dependências apontadas internamente.

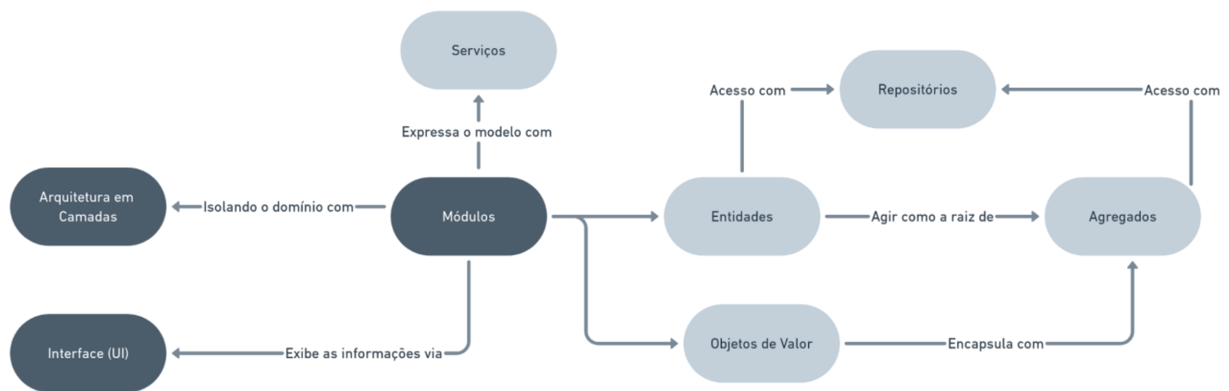


Fonte: Adaptado de Criulanscy(2019)

2.1.8 Padrões (Building Blocks) do Domain Driven Design

Dentro da metodologia DDD, é utilizado um conjunto de padrões auxiliares para a tomada de decisão e modelagem do projeto. Esses padrões ressaltam as boas práticas em programação orientada à objetos, sendo eles: Entidades, Objetos de Valor, Serviços, Módulos, Agregados, Fábricas e Repositórios.

Figura 6 - Fluxo realizado pelos padrões ou blocos de construção do DDD.



Fonte: Adaptado de Stemmler(2019)

Entidades

A identidade de um objeto deve ser única, não podendo dois objetos distintos possuir a mesma identidade (EVANS, 2003). As entidades são objetos mutáveis com identificadores, comumente definidos como *id*, e seu estado é modificado em resposta a alguma lógica de

negócios. Duas entidades são iguais se tiverem o mesmo identificador, por exemplo, um usuário com *id* = 7 será semanticamente o mesmo se recuperarmos esse usuário depois. As entidades pertencem à camada de domínio.

Segundo Macedo (2009), essa identidade pode ser atribuída com base em atributos que nunca alteram que distinguem um objeto de outros da mesma classe. Como alternativa, é comum a utilização de um identificador externo (como o CPF ou RG), na qual sua essência é imutável uma vez que foi atribuída a uma pessoa.

Objetos de Valor

Os Objetos de Valor são classes que não necessitam de uma identificação e são parte de uma Entidade. Um objeto de valor pode ser, por exemplo, o endereço de uma pessoa, não necessariamente esse endereço precisa ter uma identidade, pois, sem a Entidade a qual ele pertence, se torna inutilizável no contexto (EVANS, 2003). Conforme declarado no nome, os objetos de valor representam valores como propriedades, são imutáveis e contém métodos que implementam a lógica de negócios. Os Objetos de valor pertencem à camada de domínio do sistema.

Serviços

São classes que possuem lógica de negócio, são comumente utilizadas para realização de tarefas vinculadas relacionadas a algum objeto, porém longe de serem responsáveis por elas. (EVANS, 2003). Os serviços devem significar algo ao domínio representando funções baseadas nas regras de negócio e implicando que os serviços pertencem a camada de domínio, coordenando assim as operações que envolvem outros objetos. É importante citar que existem serviços nas três diferentes camadas: infraestrutura, aplicação e domínio. Sendo a maior diferença, que pode causar confusão inicialmente, é que nas camadas de infraestrutura e aplicação não é implementado nenhuma lógica de negócio e sim uma interface do serviço implementado dentro do domínio.

De acordo com Macedo (2009), um bom serviço é formado por características como não ser parte natural de nenhuma entidade ou objeto e a operação não armazenar nenhum tipo de estado, ocorrendo consultas e alterações de outros objetos, mas nunca possuindo estado próprio. Os serviços devem ser nomeados obedecendo a linguagem ubíqua determinada de acordo com o contexto que está inserido, padronizando assim o entendimento e facilitando a criação de novos serviços.

Módulos

Por causa da evolução sincronizada entre o modelo e o desenvolvimento, o número de conceitos nas quais o desenvolvedor deve se preocupar nessa evolução cresce gradativamente e de forma considerável. É difícil pensar em todas as classes e relacionamentos do sistema ao mesmo tempo. Para ajudar na organização dos conceitos são utilizados os Módulos. Os Módulos´ são abstrações utilizadas para agrupar as classes por um conceito do domínio (EVANS ,2003). Os módulos facilitam a subdivisão desses contextos carregados de funcionalidades para melhorar o andamento do sistema, tanto na sua adição de elementos como a sua revisão, respeitando um princípio de alta coesão e desacoplamento do código para que futuras alterações não afetem diferentes arquivos custando assim o desempenho da aplicação.

Cada módulo deve prosseguir apenas com classes que estejam semanticamente relacionadas entre si, a interação entre os módulos deve ser mínima para que não seja exigido muito para a compreensão do código (MACEDO, 2009). As aplicações com conceitos similares e quem tem uma relação semântica como um sistema de vendas e compras online ficariam mais bem realocadas no mesmo módulo.

Agregados

Um sistema implementado utilizando orientação a objetos pode ser interpretado como um grafo de classes conectados por suas associações. Geralmente esse grafo se torna grande e complexo, resultando em mudanças e evoluções de um sistema. Analisando o sistema e o domínio alvo, pode-se identificar limites no gerenciamento dos objetos. Os Agregados são um grupo de objetos em um limite de contexto, que possuem uma raiz, esta precisa ser necessariamente uma Entidade (EVANS, 2003). Agregados são objetos que juntam entidades e objetos de valor juntos. Um agregado é responsável por manter invariantes de negócios inviolados a qualquer momento e, portanto, representa um limite transacional.

Repositórios

O Repositório é a ponte de acesso à persistência de dados, geralmente bancos de dados relacionais e não relacionais, tirando essa responsabilidade do domínio. Seu intuito é o encapsulamento da lógica necessária para obter a referência de um objeto (EVANS, 2003). Os objetos não precisam lidar com a infraestrutura diretamente para obter informações da aplicação, o cargo de repositório surge para dar esta responsabilidade de buscar, entregar e muitas vezes filtrar as chamadas com o banco de dados. Um repositório é uma estrutura para recuperar e consultar agregados, usado para evitar o acoplamento estreito entre essa lógica de

consulta ou salvamento com a implementação técnica. As interfaces dos repositórios pertencem à camada de domínio e sua implementação concreta na camada de infraestrutura.

O uso de repositórios foi proposto também por Fowler (2002), que reforçava a ideia de buscas mais sofisticadas em que as classes que necessitam fazer o uso da persistência não teriam conhecimento dos detalhes de mais baixo nível envolvidos, podendo também receber parâmetros altamente específicos. No cenário atual é bem comum a utilização de frameworks como *Hibernate* e *TypeORM*, que auxiliam no mapeamento de objeto-relacional utilizando o suporte a uma linguagem de consulta diferente das mais usadas em tabelas SQL.

2.2 TRABALHOS RELACIONADOS

No nosso trabalho, temos como proposta expor modelagens utilizando os conceitos do DDD para uma barbearia que presta serviço por agendamento web. A ideia nessa seção é observar um exemplo de trabalho em que os autores utilizaram os princípios do DDD para dar forma ao direcionamento de sua modelagem, visando seu caso específico, e outro que mostra o benefício do uso do DDD em projetos de software como forma de diminuir os problemas de comunicação no projeto. Os dois trabalhos são citados brevemente na justificativa deste trabalho.

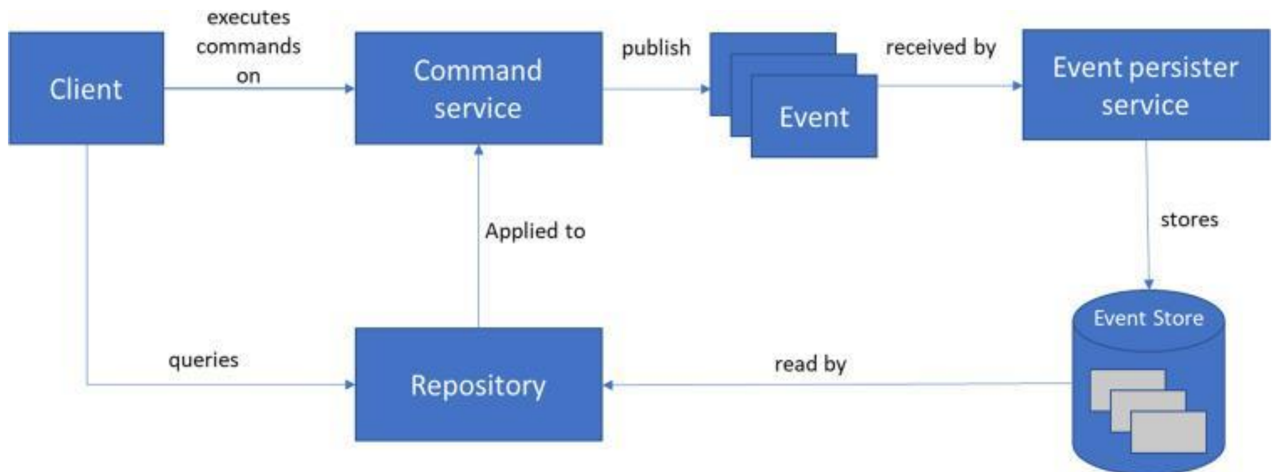
2.2.1 ***Domain-Driven Design applied to land administration system development: Lessons from the Netherlands.*** *Design Orientado ao Domínio aplicado ao desenvolvimento de sistemas para a administração de terras: Lições da Holanda.* (OUKES et al., 2021)

Desenvolveu-se um novo sistema baseado em DDD para a administração de terras na Holanda. Esse sistema proporciona tanto a gestão dos dados quanto o cadastro de terras de forma integrada e permite que todas as mudanças de estado sejam registradas como uma sequência de eventos pelo paradigma de persistência de dados conhecido como ES (*Event Sourcing*).

O sistema também implementa uma arquitetura CQRS (*Command Query Responsibility Segregation*), onde o principal foco é separar os mecanismos para ler e escrever dados. Essa arquitetura funciona em conjunto com o padrão ES e se complementam de forma a gerar informação com significados facilmente compreensíveis para o usuário em termos do modelo de administração de terras. Como exemplo, os eventos são sempre escritos no passado e representam o que foi aceito e mudado pelo sistema. Um exemplo de evento seria “Posse

transferida da pessoa A para a pessoa B” enquanto um exemplo de comando relacionado seria “Transferir posse”.

Figura 7 - Representação visual do processo que envolve a arquitetura CQRS e o ES



Fonte: OUKES et al., 2021

2.2.2 **ANÁLISE DO BENEFÍCIO DA ABORDAGEM DDD NA DIMINUIÇÃO DE FALHAS DE COMUNICAÇÃO NO DESENVOLVIMENTO DE PROJETOS DE SOFTWARE** (VICENTE, 2018)

Este trabalho preocupa-se em mostrar como os problemas relacionados com a comunicação são frequentes e uma das principais causas para falhas em projetos. Sendo também apontado que uma das habilidades mais valorizadas pelas empresas é justamente a capacidade de comunicação. Um dos principais motivos para a falha na comunicação em projetos de desenvolvimento de software é a diferença na linguagem utilizada pelos membros da equipe para descrever os processos e seus problemas. Essa diferença origina-se da discrepância de linguagem entre as áreas de atuação dos membros da equipe, das experiências deles e por outras questões culturais e sociais.

Como solução para essas diferenças de linguagens utilizadas, a adoção de uma linguagem ubíqua a fim de representar os elementos do domínio, é fundamental. A linguagem ubíqua direcionada para um contexto, que nesse caso seria representado pelo projeto, é um dos fundamentos do DDD e o processo de refinamento dela entre os membros da equipe,

desenvolvedores e detentores de conhecimento do domínio, é uma atividade importante dessa abordagem.

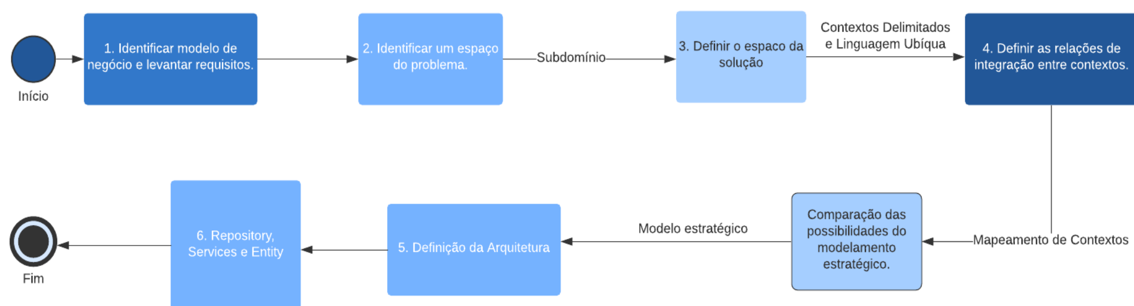
Com isso, o autor traça a importância de utilizar uma linguagem ubíqua e seu processo de refinamento para a melhor interação entre a equipe e como forma de estruturar o código, pela delimitação conceitual dos objetos no software e suas nomenclaturas de identificadores, através dessa linguagem. As interações usando a linguagem ubíqua seriam mais proveitosas e mais significativas, pois todos conseguiriam em geral entender os significados limitados dos termos dessa linguagem, e ela seria orientada para o domínio, o que ajudaria a esclarecer as regras de negócio para os desenvolvedores. O produto desse projeto seria um software mais alinhado com a forma que os detentores de conhecimento do domínio enxergam ele.

3 DESENVOLVIMENTO DA PROPOSTA / METODOLOGIA

3.1 VISÃO GERAL DA IMPLEMENTAÇÃO DDD

O DDD (*Domain Driven Design*) se trata de uma abordagem de modelagem de software que acompanha diferentes práticas com o intuito de possibilitar a implementação de regras complexas e processos de negócios denominados como domínio. Não existe um passo-a-passo propriamente dito ou determinado, porém, é possível estabelecer uma ordem lógica de tudo que é necessário e ordenar em etapas de acordo com as necessidades de cada processo. Na Figura 8 abaixo, é ilustrado todo processo em sequência proposto neste trabalho, seguido nas próximas subseções uma breve descrição de cada etapa.

Figura 8 - Visão geral da implementação da metodologia DDD.



Fonte: Autores (2021)

3.1.1 Identificação do modelo de negócio e levantar requisitos

O primeiro passo da modelagem é o processo de aperfeiçoamento do conhecimento sobre o domínio no qual o software está inserido, sem entender o negócio não é possível implementar o DDD. Em um projeto existem dois papéis que trabalham em conjunto nesta etapa, o time de desenvolvimento e os especialistas do domínio, sendo este último responsável em explicar e guiar o time de desenvolvedores na definição de regras e processos da aplicação mediante um levantamento prévio de todos os requisitos funcionais, não funcionais e suas respectivas regras de negócio.

3.1.2 Espaço do problema e subdomínio.

Nesta etapa são identificadas as áreas dos problemas que precisam ser resolvidos pela aplicação. Assim, é feita a análise dos subdomínios, descrevendo a que cada subdomínio se refere em relação ao problema geral do domínio.

3.1.3 Espaço da solução, Contexto Delimitados e Linguagem Ubíqua

Há diversas formas de solucionar um determinado problema. Por isso, neste passo, deve-se ser analisadas possíveis situações levando em conta as boas práticas do DDD e suas eventuais soluções correspondentes, para ao final do modelamento estratégico, chegar-se na melhor solução a fim de resolver os problemas do domínio.

3.1.4 Integração entre contextos.

Se as situações descritas no passo anterior apresentarem solução geral com mais de um contexto, então é necessário definir as relações de integração de cada contexto.

3.1.5 Comparação das possibilidades do modelamento estratégico

Deve-se comparar cada situação levantada e trabalhada nos passos 3 e 4 de acordo com as boas práticas do DDD e escolher a situação que mais se adequa.

3.1.6 Definido a arquitetura

Tendo uma clara visão do *Map Context*, integração entre contextos, é possível avançar para a definição da arquitetura do sistema de acordo com cada contexto. Neste processo não é definido uma arquitetura estabelecida pois a estrutura deve ser definida de acordo com a decisão da equipe em relação às necessidades impostas, cada contexto pode ter de forma independente sua própria arquitetura. Neste trabalho será definido a arquitetura de acordo com os princípios de camadas descritos por Evans (2003) divididos em camada de aplicação, domínio e infraestrutura.

3.1.7 Aplicando padrões táticos do DDD.

Dentre os conceitos do DDD, existem uma série de padrões táticos de desenvolvimento a nível de código focados no domínio da aplicação. Tais padrões são boas práticas para a programação orientada a objetos e são subdivididas em Entidades, Repositórios, Serviços, Objetos de Valor, Módulos e Agregados. Neste processo nem todos serão aplicados, dependerá de acordo com a complexidade do sistema.

3.2 ENTENDER O NEGÓCIO E LEVANTAMENTO DE REQUISITOS

Os serviços de agendamentos através do aplicativo têm a ver com a identificação do público-alvo que está imerso em uma rotina onde o tempo é um recurso precioso e prefere não desperdiça-lo, portanto o agendamento proporciona uma forma de solucionar esse desperdício (BENINCASA, 2018). Um sistema de agendamento online provê diferentes benefícios para o negócio como a acessibilidade do usuário utilizar o serviço a qualquer hora em qualquer lugar,

automatização de processos, modernização do negócio, alto nível de controle da agenda e a melhoria na gestão em torno desses elementos. Além de facilitar na organização, um sistema de marcação online otimiza o tempo, melhora o atendimento e interfere positivamente na produção do estabelecimento. A aplicação apresentada neste trabalho consiste em um sistema de agendamento online para uma barbearia e a análise do modelo de negócio é baseado nos requisitos necessários como criação de usuário, gerenciamento do perfil de usuário, definição de prestadores de serviços (barbeiros), listagem de agendamentos por data e a realização de agendamento.

De maneira simplista é possível considerar que um requisito de software é toda abstração de um recurso, funcionalidade ou resultado esperado de um sistema. Sendo o processo de levantamento de requisitos a etapa responsável por identificar todas essas necessidades e demandas, documentar os requisitos de software e especificar os mesmos através de funcionalidades, comportamentos e características necessárias para atender tais solicitações (ALFF, 2020). As funcionalidades principais desta aplicação de agendamento serão separadas entre o sistema de gerenciamento de usuários, agendamento de serviços e as notificações de agendamentos realizados.

3.2.1 Gerenciamento de Usuários

O gerenciamento de usuário deve fornecer funcionalidades como a criação de novo usuário, atualização do seu perfil e a recuperação de senha.

Tabela 1 - Requisitos da funcionalidade de criação de usuário.

Criação de usuário	
Requisitos Funcionais (RF)	● O usuário deve poder criar uma conta inserindo e-mail, nome e senha.
Requisitos Não Funcionais (RNF)	● A usabilidade deve ser colocada em prioridade, dando importância para uma interface intuitiva e fluida. ● Suportar alto desempenho diante um cenário de alta escala com crescente número de usuários. ● Garantir a criação segura de usuários com senhas criptografadas e protegidas. ● Integração com banco de dados relacional PostgreSQL pela

	modularidade e alta escalabilidade.
Regras de Negócio	● O usuário precisa inserir a senha novamente para confirmar e validar sua escolha. ● O usuário não poderá criar uma conta utilizando um e-mail já utilizado.

Fonte: Autores (2021)

Tabela 2 - Requisitos da funcionalidade de recuperação de senha.

Recuperação de senha	
Requisitos Funcionais (RF)	● O usuário deve poder recuperar sua senha informando seu e-mail. ● O usuário deve receber um e-mail com instruções para recuperação de senha. ● O usuário deve poder resetar sua senha.
Requisitos Não Funcionais (RNF)	● Deve ser notificado em tela que o e-mail de recuperação foi enviado. ● O envio de e-mails deve acontecer em segundo plano. ● Utilização do Amazon SES (*Simple Email Service*) pela transparência dos custos e alta confiabilidade.
Regras de Negócio	● O link enviado para resetar senha deve expirar depois de 2 horas. ● O usuário precisa confirmar a nova senha ao recuperar sua senha.

Fonte: Autores (2021)

Tabela 3 - Requisitos da funcionalidade de visualizar e alterar informações do perfil.

Visualização e atualização do perfil	
Requisitos Funcionais (RF)	• O usuário deve poder visualizar e atualizar nome, email e senha; • O usuário deve poder visualizar e atualizar sua foto de perfil(avatar).
Requisitos Não Funcionais (RNF)	• Disponibilidade para melhoria na usabilidade e personalização de cada usuário. • Integração com serviço Amazon S3 para armazenamento pela alta confiabilidade e segurança.
Regras de Negócio	• O usuário não pode alterar seu e-mail para um email já utilizado. • Para atualizar sua senha, o usuário deve informar sua senha antiga. • Para atualizar sua senha, o usuário deve confirmar sua nova senha.

Fonte: Autores (2021)

3.2.2 Agendamento de Serviços

O agendamento de serviços é subdividido pelo painel do prestador e o agendamento de serviços, sendo o primeiro responsável pela listagem de agendamentos feitos por dia e por prestador de serviço e por último a funcionalidade principal da aplicação.

Tabela 4 - Requisitos do painel do prestador

Painel do Prestador	
Requisitos Funcionais (RF)	• O prestador deve listar seus agendamentos de um dia

	específico sendo subdivididos no período da manhã e da tarde entre os horários de 8 horas até as 18 horas.
Requisitos Não Funcionais (RNF)	● A interface deve ter uma proposta simples e intuitiva para que qualquer prestador (barbeiro) possa utilizar e entender de maneira rápida. ● Os agendamentos do prestador devem ser armazenados em cache para melhoria do desempenho no momento de visualização.

Fonte: Autores (2021)

Tabela 5 - Requisitos do agendamento de serviços pelo prestador

Agendamento de serviços	
Requisitos Funcionais (RF)	● O usuário deve poder listar todos os prestadores de serviço (barbeiros) cadastrados. ● O usuário deve poder listar os dias de um mês com pelo menos um horário disponível de um prestador. ● O usuário deve poder listar os horários disponíveis em um dia específico de um prestador. ● O usuário deve poder realizar um novo agendamento com um prestador de escolha.
Requisitos Não Funcionais (RNF)	● A tela de agendamento deve ser intuitiva e com alta usabilidade para ser acessível a todos os usuários de diferentes perfis visando pontos como acessibilidade e desempenho. ● As informações do agendamento devem ser mantidas em segurança. ● A listagem de prestadores e dos agendamentos deve ser armazenada em cache visando as vantagens de desempenho e qualidade do uso.

Regras de Negócio	- Cada agendamento deve durar exatamente 1 hora. - Os agendamentos devem estar disponíveis entre 8h até as 18h (Primeiro às 8h, último às 17h) - O usuário não pode agendar em um horário já ocupado. - O usuário não pode agendar em um horário que já passou. - O usuário não pode agendar serviços consigo mesmo.

Fonte: Autores (2021)

3.2.3 Notificações de Agendamentos

Tabela 6 - Requisitos para sistema de notificações da aplicação.

Notificações	
Requisitos Funcionais (RF)	- O usuário prestador deve poder visualizar na aba de notificações os novos agendamentos. - O prestador deve poder visualizar novas e antigas notificações de agendamentos marcados. - As notificações devem poder ser marcadas com o status de lidas após serem visualizadas.
Requisitos Não Funcionais (RNF)	- Garantir que as notificações sejam enviadas em tempo real para a interface de ambos os tipos de usuário. - Disponibilizar na interface de maneira acessível e segura.

Fonte: Autores (2021)

3.3 IDENTIFICAR ESPAÇO DO PROBLEMA

A fim de definir o espaço do problema, norteado pela necessidade de cumprimento dos requisitos a serem implementados na solução em software, devemos definir, dentro do domínio de negócio da barbearia, os subdomínios que englobam as especificações gerais do modelo do domínio pontuado na seção anterior. Delimitando-os de acordo com suas funcionalidades.

3.3.1 Subdomínios

O primeiro passo é definir qual o núcleo do domínio, o qual é definido por ser a área da organização capaz dar o maior diferencial competitivo para ela e assim agregar mais valor de negócio. Em seguida, analisar qual o papel de cada subdomínio. Neste sentido, será analisado cada subdomínio que faz parte do espaço do problema nos tópicos abaixo.

3.3.1.1 Gerenciamento de contas

Todas as regras de cadastramento, manutenção e personalização da conta fazem parte deste subdomínio. Ele suporta o subdomínio de agendamento ao conter a base para a identificação de cada cliente e prestador de serviço e dentro dele há questões referentes a como as informações pessoais de cada cliente vão ser gerenciadas.

3.3.1.2 Agendamento

O subdomínio de Agendamento é o grande diferencial da organização por possibilitar a disponibilização da prestação de serviços de barbearia de uma forma conveniente e organizada para o seu cliente. Assim, este é o núcleo do domínio e por isso deve receber prioridade no seu entendimento e modelamento.

3.3.1.3 Notificações

Abrange qualquer regra visando notificar o cliente ou provedor do serviço de barbearia de um agendamento ou outras questões de negócio como a veiculação de promoções, novos serviços ou qualquer problema que possa acontecer com a prestação de serviços. É um subdomínio de suporte para o agendamento, pois, apesar de conter questões essenciais para o negócio, comporta modelos do domínio que tem uma função especializada e que podem ser tratados de forma separada ao núcleo do domínio.

3.4 DEFININDO O ESPAÇO DA SOLUÇÃO

Agora que foram identificadas as diferentes áreas dos problemas que precisam ser solucionados pelo software a ser desenvolvido e suas especificações, será definido uma solução para resolver os problemas identificados. O desenvolvimento por DDD não especifica exatamente como deveria ser esta solução, mas determina várias boas práticas que devem ser seguidas. Antes de descrever as boas práticas, será elucidado os termos utilizados no DDD para referir-se a uma solução específica.

Um determinado contexto limitado, que geralmente é implementado como um projeto na maioria das linguagens, é uma solução para um certo espaço de problema. Nele está contido um grupo de entidades e outros objetos relacionados, e o significado de cada nome nestas estruturas tem o sentido delimitado pelo seu contexto. Por exemplo, é possível ter em um contexto de loja de produtos uma entidade chamada cliente relacionada a pessoa que já fez uma compra nesta loja, enquanto, em um contexto de navegadores web, a entidade cliente poderia ser qualquer navegador participando de uma requisição HTTP. Assim, o contexto é um limite linguístico explícito que contém o modelo (VERNON, 2013) e representa a solução em software dos problemas no domínio da organização.

Uma primeira boa prática a ser implementada é tentar criar para cada subdomínio uma solução específica para ele, ou seja, criar um contexto limitado contido neste subdomínio. Fazer isto ajuda a evitar termos possivelmente ambíguos vindos de áreas diferentes e facilita o entendimento do objetivo do contexto, pois separa as responsabilidades de cada contexto por subdomínio de atuação. No entanto, é preciso ponderar sobre a viabilidade desta separação em relação a tecnologia que vai ser utilizada na implementação e na complexidade que vai ser gerada na integração de vários contextos. Esta análise deve levar em consideração não só o momento atual do projeto, mas também se o modelo de negócio do projeto tende cada vez mais crescer, pois isto poderia ser um fator que justificaria a separação em vários contextos para a manutenção dos projetos no longo prazo.

A ideia agora é tentar modelar duas possíveis situações e avaliar qual seria a mais viável segundo considerações de crescimento da complexidade com o tempo e das restrições impostas pela tecnologia utilizada. Neste sentido, duas situações de modelamento estratégico vão ser desenvolvidas e depois haverá uma ponderação de qual será utilizada e o porquê. Uma delas usa de forma literal, a boa prática proposta e assim cria, para cada subdomínio identificado, um contexto limitado próprio, e a outra é modelada na hipótese mais simples de que todo o domínio será atendido por um único contexto. Com isto, nesta e na próxima seção, estas duas situações serão trabalhadas e ao final um veredicto será dado sobre qual será o modelo escolhido.

3.4.1 Um contexto por subdomínio

Cada contexto limitado foi projetado dentro de um respectivo subdomínio, alinhando-se com os princípios de modelamento estratégico do DDD (VERNON, 2013), mapeando-os em uma relação de um para um. Fazer isto coloca cada espaço de solução, o contexto limitado, para lidar com o seu próprio bem definido espaço do problema, o subdomínio subjacente, facilitando

a identificação do objetivo a que cada contexto se propõe. Com este intuito, uma descrição foi feita para relacionar a solução que cada contexto representa para sua respectiva área problema.

3.4.1.1 Contexto de gerenciamento de contas

Este contexto delimita modelos que permitem o registro das atividades de cada prestador e cliente. Assim, a informação de cada conta fornece dados essenciais para que o agendamento aconteça de forma coerente e gera a possibilidade de um atendimento especializado visando as necessidades individuais de cada cliente.

3.4.1.2 Contexto de Agendamento

Neste sistema, o contexto envolve termos comuns aos processos de agendamento. Ele delimita o modelo de conexão entre o cliente e o barbeiro para a prestação dos serviços de barbearia. Esta simples delimitação permite um fácil reconhecimento dos objetivos deste contexto e facilita a criação de uma linguagem clara e objetiva. Como este é o contexto limitado que está localizado no núcleo do domínio, ele é o que deve receber priorização de recursos para seu desenvolvimento a fim de gerar mais valor verdadeiro para o negócio.

3.4.1.3 Contexto de notificações

Este é o contexto que engloba os modelos de comunicação com os usuários do sistema de agendamento. O contexto preocupa-se em solucionar as regras de gerar avisos aos prestadores de serviço da barbearia sobre algum agendamento vinculado a eles e a veiculação sobre alguma nova promoção ou serviço de barbearia. Tudo isto é definido como uma série de serviços de notificação que são fornecidos ao contexto de agendamento e de gerenciamento de contas.

3.4.2 Um contexto para todo o domínio

O contexto agora soluciona o problema de cada um dos subdomínios identificados evitando conflitos de linguagem entre as áreas, o que é facilitado pela quantidade pequena de entidades empregadas para descrever o modelo de negócio e a própria natureza distinta de cada subdomínio entre si. Assim, este contexto integra em uma única aplicação, separada em diversos módulos, todas as funcionalidades descritas nos contextos acima e, como ele é único, não há a necessidade de integração com outros contextos, ou seja, cada funcionalidade de um

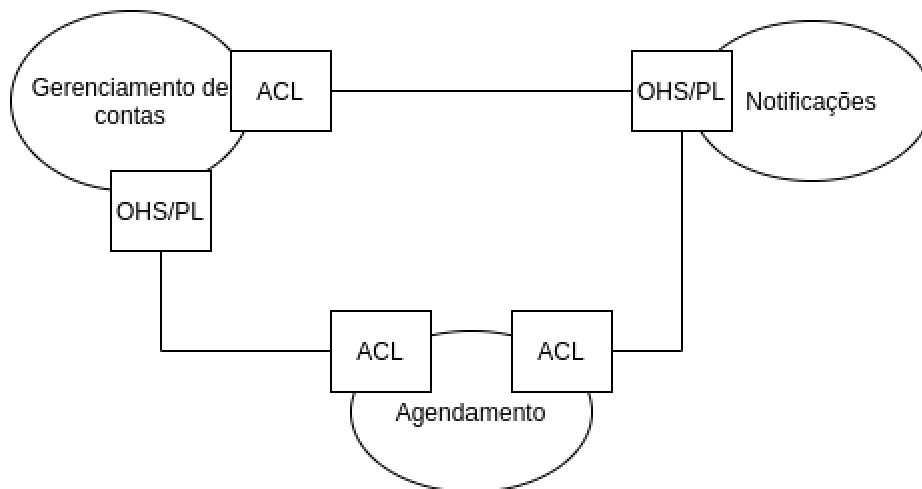
módulo é exposta ao outro por meio da importação de pacotes responsáveis por fazer esta conexão.

3.5 MAPA DE CONTEXTO

Como a segunda situação é composta por apenas um contexto, o seu mapa de contexto não precisa ser desenhado já que são necessários pelo menos dois contextos para haver uma relação de integração. Por isto, nesta seção, será abordado apenas a situação com vários contextos.

A partir da análise de subdomínio realizada e da delimitação de um contexto como uma solução especializada para cada subdomínio, agora é possível desenhar o mapa de contextos para elucidar as relações de integração de cada contexto. Este mapa é mostrado abaixo na Figura 9. Os Símbolos U e D determinam qual contexto depende da forma como os serviços vão ser ofertados e qual determina o formato dessa oferta de serviços, respectivamente.

Figura 9 - Mapa de Contexto



Fonte: Autores (2021)

3.5.1 Relações do contexto de notificações

O contexto de notificações se integra com os outros dois por meio de comunicações baseadas no estilo arquitetural REST proposto por Roy Thomas Fielding em sua tese de Phd. Este estilo será implementado utilizando os protocolos HTTP e HTTPS (FIELDING, 2000). Assim, o contexto disponibiliza serviços para que o contexto de agendamento avise um prestador de serviço de um agendamento feito em seu nome e para que o contexto de gerenciamento de contas contate clientes alvo de promoções e novos serviços. Estas comunicações utilizam representações em JSON para trocar dados das entidades definidas em

seus modelos. Como exemplo, o contexto de agendamento utiliza o método HTTP Post, com os dados do agendamento no corpo da requisição, para informar o contexto de notificações de um novo agendamento válido realizado e este cria uma nova notificação para o prestador de serviço, que fica acessível a ele em outro *endpoint*, avisando-o do agendamento.

3.5.2 Relações do contexto de gerenciamento de contas

O gerenciamento de contas estabelece uma camada anticorrupção que traduz os modelos recebidos por JSON do contexto de notificações para utilizar internamente. O motivo principal dessa integração é o contexto de gerenciamento de contas contatar o de notificações sobre eventuais promoções ou novos serviços de acordo com o perfil alvo.

A integração com o agendamento é semelhante, porém, neste caso, o contexto de agendamento consome os serviços ofertados pelo contexto de gerenciamento de contas, através do protocolo HTTP, para receber as informações dos usuários envolvidos em um agendamento.

3.5.3 Relações do contexto de Agendamento

Este contexto define camadas anticorrupção que traduzem as respostas baseadas nos modelos dos outros contextos para modelos no seu próprio contexto. Estes modelos externos são recebidos através de respostas no formato JSON de requisições HTTP durante o consumo dos serviços ofertados pelos outros dois contextos como foi ilustrado nos subtópicos.

3.6 COMPARAÇÃO DAS POSSIBILIDADES DO MODELAMENTO ESTRATÉGICO

A complexidade da integração da primeira situação é evidente por todas as estruturas que vão ser criadas em cada contexto para suportar esta integração. No caso da situação com um contexto, você não tem uma separação de responsabilidades a nível de aplicações, mas consegue ter a nível de módulos o que ainda pode dar margem a conflitos de linguagem sobre as regras de negócio que envolvem vários subdomínios, porém, pelo tamanho do modelo previsto, estes conflitos conseguem ser evitados sem grande malabarismo.

É importante levar em consideração também o fato de que este trabalho está sendo desenvolvido por apenas duas pessoas que formariam uma equipe para os três contextos. Uma boa prática do DDD é colocar uma equipe por contexto para facilitar a comunicação da equipe em relação ao modelo através de uma única linguagem ubíqua (VERNON, 2013). Assim, colocar uma mesma equipe para se comunicar e desenvolver três linguagens ubíquas diferentes ao mesmo tempo é possivelmente contra produtivo.

Este projeto, como um todo, não tem planos para grandes expansões ao longo do tempo e, com os poucos modelos que emprega, já dá um bom suporte até para regras de negócio um pouco mais exigentes sobre o domínio. Por essas considerações, decidiu-se que não justificaria aumentar a complexidade do projeto em estruturas de integrações só para separar as responsabilidades e assim a situação escolhida, para este trabalho, é desenvolver ela utilizando um único contexto.

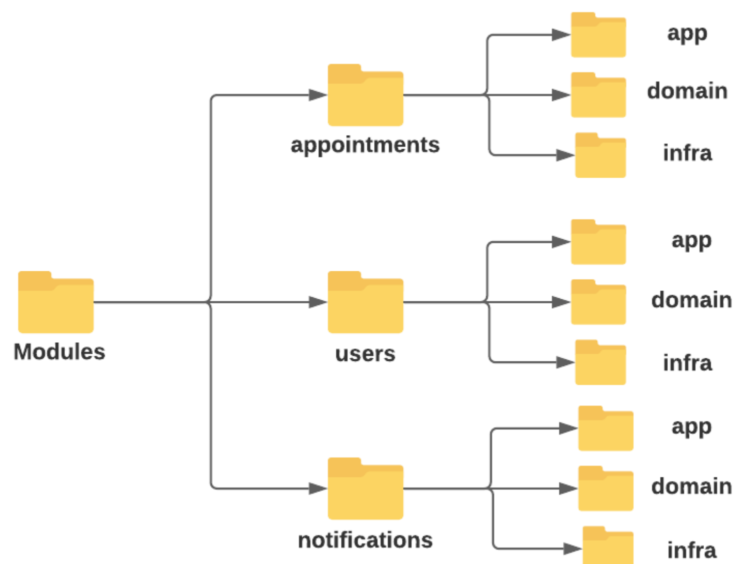
3.7 DEFINIÇÃO DA ARQUITETURA

A flexibilidade que o DDD oferece na estruturação da arquitetura, seguindo seus princípios de foco no domínio, possibilita a elaboração de maneira única e adaptada para diferentes cenários. Desta maneira foi realizado para este trabalho a organização em diretórios definidos como *Modules* e *Shared*.

3.7.1 Diretório *Modules*

A pasta *modules* fica responsável pela estruturação do código por diferentes domínios, sendo eles no momento: *users*, *appointments* e *notifications*. Neste local ficam determinados arquivos como DTO (*Data Transfer Object*), *Entities*, *Repositories* (incluído pastas como *fake/mock* para implementação de testes automatizados), *Provider* (específicos para o domínio) e a pasta de Infraestrutura específica para os usuários. Os módulos indicados são criados de acordo com o domínio do negócio (identificado na seção 3.2) e subdivididos em três diferentes diretórios: *App* (*Application*), *Domain* e *Infra* (*Infrastructure*).

Figura 10 - Organização de diretórios em módulos.

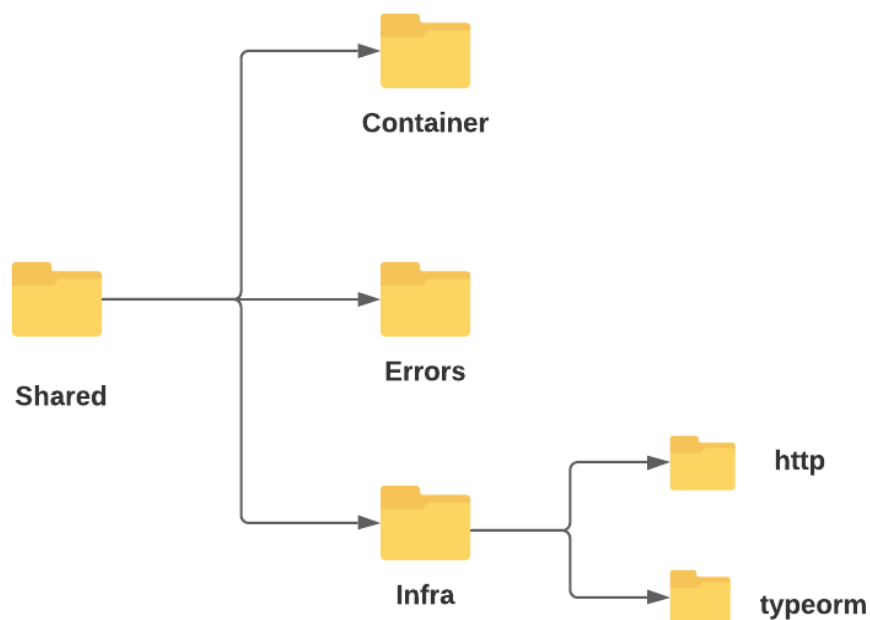


Fonte: Autores (2021)

3.7.2 Diretório *Shared*

A pasta *Shared* fica encarregada com os arquivos que são compartilhados entre módulos ou até entre todos. Neste local ficará armazenado arquivos como *Errors*, *Shared Database*, *Shared Routes*, *Shared Middlewares* e *Shared Providers* (*Storage Provider, Email Provider*). Este sistema é formado por três diferentes subpastas chamadas *Container* (Provedores de serviço de e-mail, storage, cache), *Errors* (Classe para resposta de erros para ser utilizado dentro de todos os módulos) e *Infra*. Sendo este último responsável pelas rotas de todos os módulos, middlewares e integração com banco de dados. O manuseio de informações com um banco de dados relacional (PostgreSQL) foi realizado utilizando ORM (*Object Relational Mapping*). Foi nomeado especificamente como http, referenciado de protocolo http, para aumentar a maleabilidade do sistema caso, futuramente, seja substituído ou adicionado outro protocolo como, por exemplo, GRPC (*Google Remote Procedure Call*). Segundo Cruz (2020), isso demonstra a vantagem principal do DDD que possibilita a criação de código dentro do DDD, com componentes bem definidos e alto desacoplamento que possam ser substituídos ou adaptados causando menos impacto no sistema em geral.

Figura 11 - Diretório *Shared* e suas subdivisões.



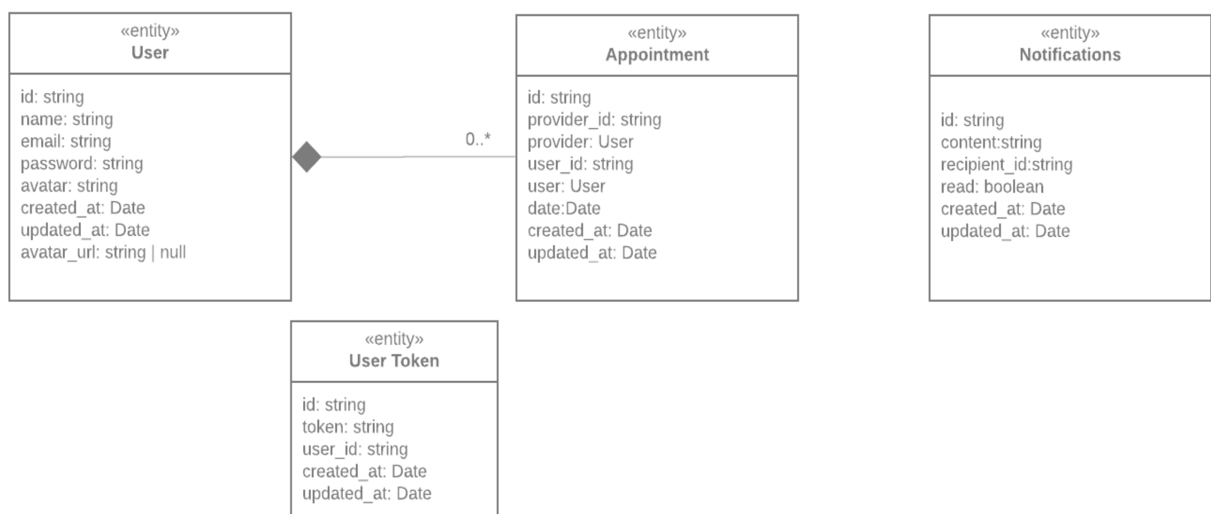
Fonte: Autores (2021)

3.8 APLICAÇÃO DE PADRÕES DO DDD

3.8.1 Entidades

Todas as entidades quando criadas recebem um identificador único que é atribuído pelo sistema, conhecido como *id.* Essa identificação é utilizada como referência para tal Entidade quando requisitada, seja esse pedido uma visualização, atualização ou remoção da aplicação (COSTA; HILD, 2019). Mediante análise do domínio foi possível definir quais classes necessitavam de uma entidade própria, sendo elas: *Appointment* (Agendamento), *User* (Usuário), *User Token* e *Notification*(Notificação).

Figura 12 - Diagrama de Classes (UML) do sistema de agendamento



Fonte: Autores (2021)

3.8.2 Repositórios

A implementação dos repositórios foi feita nas camadas de domínio e infraestrutura, sendo o primeiro criado com a finalidade de criar uma abstração e a segunda sendo de maneira geral com funcionalidades comuns para entidades, como por exemplo, adicionar, atualizar ou excluir um objeto do banco de dados relacionado. Foram utilizados ao todo quatro diferentes arquivos repositórios de implementação (infraestrutura), sendo eles *UsersRepository*, *UserTokenRepository*, *AppointmentRepository* e *NotificationRepository*. Os demais são os repositórios interface contidos na camada de domínio como *IUsersRepository*, *IUserTokenRepository*, *IAppointmentRepository* e *INotificationRepository* e os repositórios falsos (fakes/mocks) criados para testes automatizados da aplicação.

Dentro dos repositórios da camada de infraestrutura a implementação foi feita com injeção de dependências, ou seja, quando os repositórios da camada de domínio são referenciados, os repositórios de infraestrutura são acionados recebendo as informações e comandos do domínio.

Tabela 7 - Lista de arquivos do tipo Repositório

UsersRepository.ts	UserTokensRepository.ts
IUsersRepository.ts	IUserTokensRepository.ts
FakeUsersRepository.ts	FakeUserTokensRepository.ts
AppointmentRepository.ts	NotificationRepository.ts
IAppointmentRepository.ts	INotificationRepository.ts
FakeAppointmentRepository.ts	FakeNotificationRepository.ts

Fonte: Autores (2021)

3.8.3 Serviços

Os serviços servem grande propósito ao domínio, isto é, devem representar operações baseadas em regras de negócio. O resultado é que os serviços pertencem à camada de domínio e sua função é coordenar as operações que envolvem outros objetos sendo assim a função de manipular os repositórios e fazer a comunicação com a camada de aplicação. Para isso foi desenvolvido serviços base, contendo atividades e funcionalidades comuns do sistema de agendamento em cada entidade. Os serviços dentro de *user* foram definidos como autenticação do usuário, criação de usuário, visualização de perfil do usuário, atualização do perfil (informações e avatar) e recuperação de senha. Dentro de *appointments* foram definidas as regras de negócio necessárias para realização e visualização dos agendamentos de forma que evite eventuais erros e falhas na sua utilização. Entre eles estão a criação de agendamento, de maneira que não ocorra conflitos de data e horário, e a listagem da disponibilidade de horários por dia para a entidade usuário configurada como prestador de serviço.

Tabela 8 - Lista de serviços da aplicação, seus métodos, parâmetros e *endpoints* utilizados.

Serviço	Tipo de parâmetro / Método	Parâmetros	Endpoint
Create Appointment Service	Body Params / POST	provider_id, date	/appointments
List Provider Appointments Service	Query Params / GET	year, month, day	/appointments/me
List Provider Month Availability Service	Query Params / GET	year, month	/providers/{provider_id}/month-availability
List Provider Day Availability Service	Query Params / GET	year, month, day	/providers/{provider_id}/day-availability
Create User Service	Body Params / POST	name, email, password	/users
Update User Avatar Service	Multipart-Form-Data /PATCH	*.png, *jpg, *jpeg	/users/avatar

List Providers Service	Route Params / GET	—	/providers
Authenticate User Service	Body Params / POST	email, password	/sessions
Reset Password Service	Body Params / POST	password, token	/password/reset
Send Forgot Password Email Service	Body Params / POST	email	/password/forgot
Show Profile Service	Route Params / GET	—	/profile
Update Profile Service.ts	Body Params / PUT	name, email,old_passowrd, password, password_confirmation	/profile

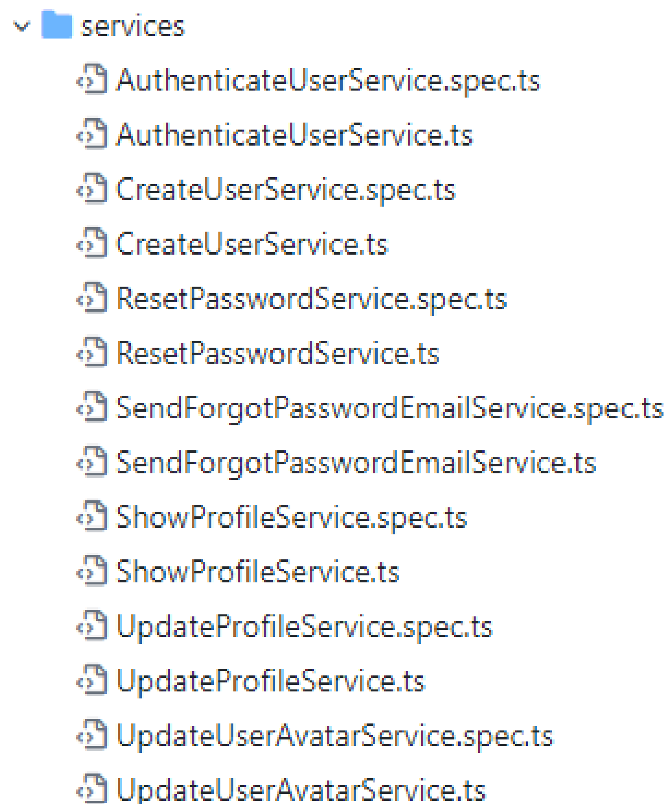
Fonte: Autores (2021)

3.8.4 Testes Unitários

Para validar as regras de negócio do sistema foram realizados testes de unidade com o intuito de avaliar todos os componentes de serviço, caracterizando uma parte extremamente importante para o desenvolvimento de software. Todos os serviços foram testados utilizando repositórios *fakes* (falsos) criados particularmente para realizar tal implementação, localizados dentro do diretório *service*, os testes foram nomeados com a extensão "spec.ts" para cada serviço da aplicação. De acordo com Macedo (2009), essa abordagem de testes executados ao

longo do andamento do projeto, aliado à integração contínua, ajudam a apontar os problemas mais cedo.

Figura 13 - Arquivo de serviços com seus respectivos testes unitários.



Fonte: Autores (2021)

Os testes realizados tiveram entradas (*inputs*) equivalentes a cada serviço especificado esperando como resultado as saídas(*outputs*) de acordo com as regras de negócio de cada funcionalidade. O serviço de criação de agendamento (*CreateAppointmentService*), por exemplo, recebe um objeto *appointment* (agendamento) com as propriedades *date* (data), *provider_id* (*id* do *provider*) e *user_id* (id pro usuário) esperando como saída aprovada:

- Possibilitar criação de novo agendamento.
- Impedir criação de dois agendamentos no mesmo horário.
- Impedir criação de agendamentos em dias que já passaram.
- Impedir criação de agendamento de provider (barbeiro) para outro *provider*.

- Impedir criação de agendamentos fora do horário especificado do negócio, de 8 horas às 18 horas, sendo neste caso o último agendamento às 17 horas.

Seguindo pela mesma lógica de testes unitários em todos os serviços é possível avaliar todas as funcionalidades para que seja oferecido de maneira segura para toda a aplicação, obedecendo as suas regras de negócio e entregando um software de alta qualidade.

4 ANÁLISE DOS RESULTADOS

Mediante ao protótipo de sistema de agendamento desenvolvido seguindo a metodologia Domain Driven Design, foi possível alcançar os resultados desejados propostos como objetivo deste trabalho. Em primeiro lugar, foi fundamental para a metodologia DDD definir, compartilhar e discutir o modelo de domínio, incluindo seus subdomínios e contextos limitados. Neste trabalho não foi feita a elaboração com especialistas de domínio em relação a agendamentos em barbearias, por isso é importante esclarecer que em um cenário real é imprescindível que haja colaboração entre as equipes de desenvolvimento e especialistas de domínio. Criar a linguagem ubíqua (e onipresente) faz parte disso, para que cada membro da equipe conheça e entenda o domínio.

O plano para aplicar o DDD foi definido seguindo a modelagem estratégica e a modelagem tática, porém para ambas as partes foi analisado um melhor aproveitamento da metodologia DDD em sistemas com complexidade maior que a usada como caso neste trabalho. Durante o desenvolvimento, isso ficou ainda mais claro pela pequena quantidade de Módulos, Entidades, Repositórios e Serviços da aplicação mesmo todos cumprindo muito bem seu propósito. Os Módulos tornaram o sistema mais organizado, separando as classes em vista das necessidades do domínio; Entidades permitiram que os objetos do sistema tivessem uma estrutura definida com identificação única, podendo assim serem monitorados durante toda a vida útil do sistema; os Repositórios fizeram com que os módulos fossem isolados das outras camadas, excluindo quaisquer dependências de tecnologias de persistência de dados. Dessa forma, é possível promover uma ideia de desacoplamento em que pode ser utilizado outra tecnologia para acesso ao banco de dados, ou até mesmo utilizar dois bancos de dados sem a necessidade de alterar a camada em si.

A arquitetura escolhida teve como base a divisão por camadas de Evans (2003) e obteve como resultado um código bem dividido e de fácil compreensão, tornando assim um conjunto de arquivos mais acessíveis para manutenção e descomplicado para novas implementações. A maior dificuldade do trabalho foi entender como o DDD deve ser aplicado tanto nas fases iniciais de definição do domínio como dentro do código. Apesar do livro de Evans (2003) possuir alguns exemplos práticos, de início pode ser muito complicado para aplicar algo explicado muito em teoria. Já o livro escrito por Vernon (2013), Implementando Domain Drive Design, o método é exposto de maneira mais prática facilitando mais a compreensão de como aplicar e como utilizar seus paradigmas.

5 CONSIDERAÇÕES FINAIS

O DDD serviu como uma importante ferramenta conceitual tanto para poder organizar as ideias do domínio que formavam o software quanto para alinhar elas com as partes do software que davam suporte para o domínio. Utilizar essa abordagem aumentou a complexidade do código e por isso as considerações da complexidade do modelo são importantes para não gerar uma solução muito grande em um problema pequeno e dificultar sua implementação, porém ela deu uma melhor visualização e um foco no desenvolvimento dos conceitos fundamentais no projeto, as regras de negócio e a forma dos objetos do domínio, e facilitou a escolha de uma arquitetura que simplificava a interação entre esses objetos e o desacoplamento de componentes do software.

Há muitas coisas não abordadas sobre o DDD nesse trabalho e que poderiam receber um grande destaque, como o padrão ES e o CQRS discutidos na seção de trabalhos relacionados, porém, como foi feita a consideração de evitar um aumento de complexidade para um problema tão simples quanto o que esse trabalho queria resolver, essas boas práticas não foram implementadas.

Como tínhamos pouco tempo para desenvolver esse trabalho e estávamos bem ocupados, decidimos não incluir uma parte importante no desenvolvimento por DDD que envolve discutir e formar a linguagem ubíqua por interações frequentes com os conhecedores do domínio. Partimos do ponto em que essas informações já tinham sido coletadas e modelamos com base nessas ideias, porém o DDD funciona também como um processo iterativo em que essa etapa de interação se repete continuamente de forma a refinar o modelo de software construído pelos desenvolvedores e com uma participação ativa dos experts do domínio no sentido de validar o funcionamento de acordo com as regras do negócio.

O trabalho focou primariamente em mostrar o poder de organização conceitual que o DDD dá com os seus princípios e conceitos básicos de modelagem que a partir de ideias simples consegue se encaixar bem para diversos casos que exigem uma modelagem mais bem delimitada e contextualizada. Assim é possível modelar softwares grandes e complexos de forma escalável e fazendo com que o time por trás do projeto consiga ainda manter uma boa comunicação, facilitando a implementação de novas funcionalidades, a correção de bugs e a manutenção do software no longo prazo.

REFERÊNCIAS

DELGADO, Fábio. DDD uma abordagem prática. 2018. Disponível em: https://medium.com/@fabio.delgado2/domain-driven-design-75b15393067d. Acesso em: 03 maio 2021.

KAMAKURA, Felipe. O que é design de software? 2013. Disponível em: https://kamaondev.wordpress.com/2013/09/09/o-que-e-design-de-software/. Acesso em: 03 maio 2021.

EVANS, Eric. Domain-Driven Design: tackling complexity in the heart of software. Boston: Addison-Wesley Professional, 2003. 560 p.

Millett, S. (2015). Patterns, Principles and Practices of Domain-Driven Design. John Wiley & Sons. 800p.

WILLIANS, Wesley. O que é DDD – Domain Driven Design. 2019. Disponível em: https://fullcycle.com.br/domain-driven-design/. Acesso em: 03 maio 2021.

Domain-Driven Design, do início ao código. 2018. Disponível em: https://medium.com/cwi-software/domain-driven-design-do-início-ao-código-569b23cb3d47. Acesso em: 10 mar. 2021.

CRIULANSCY, Pierre. Domain-Driven Design for javascript developers: a 10 minutes introduction to grasp the whys and hows of ddd for your project. A 10 minutes introduction to grasp the whys and hows of DDD for your project. 2019. Disponível em: https://medium.com/spotlight-on-javascript/domain-driven-design-for-javascript-developers-9fc3f681931a. Acesso em: 10 abr. 2021.

Macedo, O. A. C. (2009). Diretrizes para desenvolvimento de linhas de produtos de software com base em Domain-Driven Design e métodos ágeis. PhD thesis, Universidade de Sao Paulo.

SOUSA, Julio Martins de. Levantamento de Requisitos: o ponto de partida do projeto de software. O ponto de partida do projeto de software. 2018. Disponível em: https://blog.cedrotech.com/levantamento-de-requisitos-o-ponto-de-partida-do-projeto-de-software/. Acesso em: 17 abr. 2021.

OUKES, Peter et al. Domain-Driven Design applied to land administration system development: lessons from the netherlands. **Land Use Policy**. [S. L.], p. 105379. 01 maio 2021. Disponível em: https://www.sciencedirect.com/science/article/pii/S0264837721001022. Acesso em: 12 maio 2021.

BASED, Md. Abdul; HASAN, Md. Mahabub Al; ISLAM, Md. Mazidul. Authentication of Voters using the Domain Driven Design (DDD) Architecture for Electronic Voting Systems. In: INTERNATIONAL CONFERENCE ON COMPUTER NETWORKS AND COMMUNICATION TECHNOLOGY, 1., 2016, Xiamen. **Proceedings of the International Conference on Computer Networks and Communication Technology (CNCT 2016)**. [S.L.]: Atlantis Press, 2016. p. 786-790. Disponível em: https://doi.org/10.2991/cnct-16.2017.109. Acesso em: 11 maio 2021.

SHENGLIN, Li et al. Application of DDD Theory in Analysis and Design of Equipment Maintenance System. In: 2019 IEEE SYMPOSIUM SERIES ON COMPUTATIONAL INTELLIGENCE (SSCI), 1., 2019, Xiamen. 2019 **IEEE Symposium Series on**

Computational Intelligence (SSCI). [S.L.]: Ieee, 2019. p. 3275-3280. Disponível em: https://ieeexplore.ieee.org/document/9003028. Acesso em: 11 maio 2021.

ULUDAğ, Ömer et al. Supporting Large-Scale Agile Development with Domain-Driven Design. In: AGILE PROCESSES IN SOFTWARE ENGINEERING AND EXTREME PROGRAMMING, 1., 2018, Cham. **Lecture Notes in Business Information Processing**. Cham: Springer International Publishing, 2018. p. 232-247. Disponível em: https://link.springer.com/chapter/10.1007/978-3-319-91602-6_16. Acesso em: 07 maio 2021.

BRITO, Bruno. **Domain-Driven Design - Conceitos básicos**. 2019. Disponível em: https://www.brunobrito.net.br/domain-driven-design/. Acesso em: 07 maio 2021.

VICENTE, Gabriel. **ANÁLISE DO BENEFÍCIO DA ABORDAGEM DDD NA DIMINUIÇÃO DE FALHAS DE COMUNICAÇÃO NO DESENVOLVIMENTO DE PROJETOS DE SOFTWARE**. 2018. 1 v. TCC (Graduação) - Curso de Ciência da Computação, Faculdade Anhanguera de Taubaté, Taubaté, 2018. Disponível em: https://monografias.brasilescola.uol.com.br/computacao/analise-beneficio-abordagem-ddd-na-diminuicao-falhas-comunicacao-desenvolvimento-projetos-software.htm#índice 15. Acesso em: 07 maio 2021.

VERNON, Vaughn. **Implementing Domain-Driven Design**. 2. ed. Westford: Courier, 2013. 925 p.

PIRES, Eduardo. DDD não é arquitetura em camadas. 2016. Disponível em: https://www.eduardopires.net.br/2016/08/ddd-nao-e-arquitetura-em-camadas/. Acesso em: 17 abr. 2021.

MARTIN, Robert C.. Clean Architecture: A Craftsman's Guide to Software Structure and Design. London: Pearson, 2017. 432 p.

FOWLER, Martin. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley Professional, 2002. 560 p.

ALFF, Chico. O que são Requisitos Funcionais e Não Funcionais? 2020. Disponível em: https://analisederequisitos.com.br/requisitos-funcionais-e-nao-funcionais/. Acesso em: 17 mar. 2021.

BENINCASA, Carlos Vinícius de Araujo. PLANO DE NEGÓCIOS BARBEARIA – BARBER CONCEPT. 2018. 51 f. TCC - Curso de Administração, Faculdade de Tecnologia e Ciências Sociais Aplicadas – Fatecs, Brasília, 2018.

COSTA, Adriano; HILD, Tony Alexander. Estudo da modelagem de software Domain-Driven Design aplicado na refatoração de um sistema informatizado para auxílio de Agentes Comunitários de Saúde. 2019. 21 f. TCC (Doutorado) - Curso de Ciência da Computação, Universidade Estadual do Centro-Oeste - Unicentro, Guarapuava, 2019.