

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Eduardo Pinheiro Barbosa

Desenvolvimento dirigido a domínio: um estudo de caso

Uberlândia/MG

2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Eduardo Pinheiro Barbosa

Desenvolvimento dirigido a domínio: um estudo de caso

Trabalho de Projeto de Conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção de grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Fabiano Azevedo Dorça

Universidade Federal de Uberlândia
Faculdade de Computação
Bacharelado em Ciência da Computação

Uberlândia/MG

2021

Eduardo Pinheiro Barbosa

Desenvolvimento dirigido a domínio: um estudo de caso

Trabalho de Projeto de Conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção de grau de Bacharel em Ciência da Computação.

Fabiano Azevedo Dorça

Orientador

Maria Adriana Vidigal de Lima

Professor

William Chaves de Souza Carvalho

Professor

Uberlândia/MG

2021

AGRADECIMENTOS

A Minha irmã que me deu muito apoio durante todo o meu período acadêmico.

A minha mãe, pelo amor, incentivo para a conclusão do curso.

Ao meu orientador, pelo suporte, correções e incentivos.

A Universidade Federal de Uberlândia e por todo seu corpo docente.

“A mente que se abre a uma nova ideia jamais voltará ao seu tamanho original”. Albert
Einstein

RESUMO

O termo *Domain Driven Design* (DDD) é uma abordagem de criação de software, fundamentada na elaboração de que cada aplicação que se pretende construir pertence a um domínio. Entre as diversas formas de criar softwares de fácil implementação, empregando variadas linguagens de programação, técnicas e abordagens de desenvolvimento, o DDD agrega conceitos, técnicas e princípios, do qual o destaque da programação está no Domínio. Percebe-se, no mercado de desenvolvimento de software a necessidade de atender aos requisitos do cliente e a essencialidade das constantes mudanças desses requisitos, sendo assim, se faz necessário aprofundar os conhecimentos sobre a linguagem e o funcionamento do *DDD*. O objetivo deste trabalho foi analisar os padrões existentes de aplicação do DDD a fim de analisar suas principais vantagens quando comparado com as outras abordagens já existentes. Foi criado um protótipo, visando a melhoria e organização de uma empresa de decoração de festa, de aplicação back-end utilizando as práticas do DDD e não houve implementação da parte web (front-end) da aplicação, desenvolvendo apenas a parte back-end. Concluiu-se que a aplicação conseguiu obter com êxito a implementação do DDD e teve os propósitos dessa abordagem implementados, como testes de unidades, desenvolvimento iterativo, separação de código em camadas e modelagem de domínio sendo colocados em prática.

Palavras chaves: *Domain Driven Design*; Engenharia de Software; Arquitetura de Software; Vantagens DDD; Desvantagens DDD;.

ABSTRACT

The term Domain Driven Design (DDD) is a software creation approach based on the elaboration that each application aspires to build, belongs to a domain. Between the many ways to create software that is easy to implement, use varied programming languages, development techniques and approaches, the DDD put together concepts, techniques and principles, of which the highlight of the program is in the Domain. Perceives, in the software development market the need to answer customer requirements and the essentiality of the constant changes of these requirements, therefore, it is necessary to deepen the knowledge about the language and how DDD works. The objective of this work was to analyze the existing patterns of DDD application in order to analyze its main advantages, when compared to other existing approaches. A prototype was created, directing at improving and organization of a party decoration company, of back-end application using the DDD practices and there was no implementation of the web (front-end) part of the application, developing only the back-end part. It was concluded that the application managed to obtain successfully implementing DDD and had the purposes of that approach implemented, like unit tests, iterative development, code separation in layers and domain modeling being put into practice.

Keywords: Domain Driven Design; Software Engineering; Software Architecture; DDD advantages;.

LISTA DE ILUSTRAÇÕES

Figura 1: Arquitetura de camadas	18
Figura 2: Mapa de navegação da linguagem do Design Dirigido por Modelos	19
Figura 3: Modelo Banco de Dados	23
Figura 4: Camadas e classes do DDD.....	25
Figura 5: Estruturação de pastas e classes da camada de API	26
Figura 6: Registrando injeção de dependência	26
Figura 7: Controller para consulta de todas as festas sobre um tema	27
Figura 8: Construtor teste controller.....	28
Figura 9: Teste controller	29
Figura 10: Estruturação de pastas e classes da camada de Aplicação	30
Figura 11: Exemplo do caso de uso para consultas de todas as festas	31
Figura 12: Teste unitário do caso de uso para consultas de todas as festas	32
Figura 13: Estruturação de pastas e classes da camada de Domínio	33
Figura 14: Modelo de classes	33
Figura 15: Validadores Flunt	34
Figura 16: Estruturação de pastas e classes da camada de Infraestrutura	35
Figura 17: Exemplo repositório Consulta informações do cliente	36
Figura 18: Exemplo data provider Cliente	37
Figura 19: Exemplo adapter consulta CPF	38

LISTA DE ABREVIATURAS E SIGLAS

DDD	<i>Domain Driven Design</i>
AS	Engenharia de Software
API	Application Programming Interface

SUMÁRIO

1 INTRODUÇÃO	11
1.2 OBJETIVOS.....	13
1.2.1 <i>Objetivo Geral</i>	13
1.2.2 <i>Ojetivos Específicos</i>	13
1.3 JUSTIFICATIVA.....	13
2 FUNDAMENTAÇÃO TEÓRICA	15
2.1 ENGENHARIA DE SOFTWARE.....	15
2.2 ARQUITETURA DE SOFTWARE.....	16
2.3 DDD - DOMAIN DRIVEN DESIGNER	16
3 PROTÓTIPO CRIADO.....	22
3.1 METODOLOGIA	22
3.2 DESENVOLVIMENTO	22
3.2.1 <i>Modelo de negócio</i>	22
3.2.2 <i>Ferramentas utilizadas na implementação</i>	23
3.2.3 <i>Aplicando conceitos do DDD</i>	24
3.2.3.1 <i>Camada API (Application Programming Interface)</i>	25
3.2.3.2 <i>Camada Aplicação</i>	29
3.2.3.3 <i>Camada Domínio</i>	32
3.2.3.4 <i>Camada Infraestrutura</i>	34
3.4 RESULTADOS OBTIDOS	39
4 CONCLUSÃO	41
REFERÊNCIAS.....	42

1 INTRODUÇÃO

Durante as décadas de 50 e 60 os sistemas de softwares eram bem triviais e, o desenvolvimento desses sistemas eram realizados de maneira manual, orientado em habilidades individuais e sem um tratamento específico (VICTORINO; BRÄSCHER, 2009).

Ao final desses períodos, observou-se a necessidade de dispositivos mais complexos e novos procedimentos e processos foram estruturados, originando os princípios do desenvolvimento do software (VICTORINO; BRÄSCHER, 2009).

Com a crescente evolução tecnológica observam-se significativas mudanças na maneira de interação dos usuários e seus acessos às informações, contribuindo assim para o aparecimento de novos paradigmas associados ao acesso à informação e na forma de aprendizado do usuário (MEIRELES; BONIFÁCIO, 2015).

O período contemporâneo em que estamos inseridos se evidencia pela sucessiva evolução da tecnologia e a crescente urgência em se desenvolver novos sistemas que respondam as necessidades de uma sociedade onde o novo velozmente se torna ultrapassado (VIEIRA; REIS, 2014).

É inexaurível o crescente avanço da tecnologia e a crescente exigência dos usuários para que os sistemas computacionais fiquem cada vez mais complexos, assim, a evolução e a elaboração de sistemas ficam cada vez mais relevantes (CORTÉS, 2013).

Partindo da necessidade em acelerar o processo de desenvolvimento de software vista ao imediato crescimento da demanda, criaram-se técnicas e metodologias que se destinam a diminuição de tempo e custo, aliadas a crescente produtividade, usabilidade e qualidade (VIEIRA; REIS, 2014).

Ligada a área da Ciência da Computação, a Engenharia de Software veio para apoiar esse processo, objetivando o desenvolvimento de produtos de software assim como uma relação custo benefício adequada (CORTÉS, 2013). Possui o centro nos aspectos da criação de softwares, englobando desde as etapas iniciais da especificação dos sistemas até sua conservação (LIMA, 2018).

A habitual Engenharia de Software vem aprimorando técnicas e ferramentas propícias para a criação de sistemas de computação em que o software desempenha precisamente aquilo que foi programado, objetivado e estruturado na criação do sistema (GIRARDI, 2004).

Uma das funções mais comuns no design de software é elaborar um protótipo do aplicativo de software. Ao projetar software de domínio específico, o designer tem a opção de usar o conhecimento sobre o domínio, além dos requisitos do usuário e princípios de design orientado a objetos (HRUBY, 2005).

Para a criação de um software confiável não pode haver informações confusas ou com erros e para isso é fundamental criar um modelo do domínio. Este modelo é definido como modelos incompletos da realidade, que tem como objetivo deliberar uma imperfeição em especial, onde o código das aplicações deve conduzir o modelo, levando informações claras sobre o domínio (MACEDO, 2009)

Werk (2017) define domínio como sendo o “assunto do qual a aplicação trata e da forma como ela deve se comportar, o que é usualmente chamado de lógica de negócio”, onde todo o âmbito de entendimento e funcionamento deve vir a reverter na lógica de uma aplicação, ou seja, seu domínio.

Inventado em 2004 por Eric Evans, o termo *Domain Driven Design* (DDD) é uma abordagem de criação de software, fundamentada na elaboração de que cada aplicação que se pretende construir pertence a um domínio (WERK, 2017).

Tal nomenclatura provém de uma metodologia de “modelo de domínio”, que se define como um agrupamento de ideias provenientes dos indivíduos que pertencem ao projeto, com termos e relações que representam a visão do domínio. Desta maneira, elabora-se uma linguagem única, exata para o desenvolvimento técnico (DA SILVA; FILHO; DA SILVA, 2018).

Entre as diversas formas de criar softwares de fácil implementação, empregando variadas linguagens de programação, técnicas e abordagens de desenvolvimento, o DDD agrega conceitos, técnicas e princípios, do qual o destaque da programação está no Domínio ou *Domain Model*. (FIRMINO; ZANELATO, 2014).

Com o aumento de métodos orientados para o projeto e desenvolvimento de software, o DDD quando executado em conjunto com essas técnicas, de forma moderada se torna altamente eficaz, podendo garantir desde a qualidade até a agilidade e objetividade do projeto (MATTOS; DOLL; ALMEIDA, 2010).

Não necessitando de estar unido as atuais tecnologias, o DDD foca em dominar às regras de negócio e a maneira em que elas estarão refletidas no código e no modelo de domínio (CUKIER, 2010).

As técnicas de DDD são utilizadas com êxito em diversos projetos. Um exemplo são Hu e Peng (2007), que demonstram por meio de uma aplicação o uso de DDD no

domínio do mercado cambial. Eles utilizam quatro conceitos principais: dinheiro; moeda; taxa de câmbio e *markup* (conceito do valor do custo para a formação do preço de venda). Deixando explícito que estes conceitos no modelo de domínio aumentam a facilidade na manutenção e entendimento do código.

Outro exemplo para a aplicação do uso de DDD é o da empresa Norueguesa de exploração de gás e petróleo, a Statoil. Este modelo foi dividido em vários submodelos, pois o domínio desta aplicação é muito extenso e com muitas regras de negócio. O contato entre os especialistas e os desenvolvedores gerou a criação de um modelo bem definido a partir dos modelos abstrusos do domínio (LANDRE et al.; 2006).

Diante do exposto, observa-se que o uso de métodos se tornou necessário para o desenvolvimento de sistemas, onde o DDD emprega uma fácil definição em função dos benefícios que são oferecido às empresas de desenvolvimento.

1.2 OBJETIVOS

1.2.1 OBJETIVO GERAL

Analisar os padrões existentes de aplicação do *Domain Driven Design* (DDD) a fim de analisar suas principais vantagens quando comparado com as outras abordagens já existentes.

1.2.2 OBJETIVOS ESPECÍFICOS

- Compreender a linguagem da abordagem DDD
- Realizar o estudo do DDD em um protótipo aplicando as abordagens do mesmo.

1.3 JUSTIFICATIVA

Parte-se da premissa que o principal objetivo do DDD é manter em conformidade as características dos softwares de necessidade complexas, sendo assim quando não controlada esta complexidade eleva o custo de desenvolvimento, evolução e de manutenção diminuindo a qualidade e a performance.

O DDD é um mecanismo que pode assessorar a comunicação, no que envolve os desenvolvedores e os profissionais especialistas no domínio do sistema, gerando mais facilidade ao processo de levantamento de requisitos essenciais para a criação do sistema.

É importante salientar que DDD não se trata de um framework nem de uma metodologia, e sim de abordagem para a modelagem de software, isto é, uma sequência de ideias e princípios que podem acrescentar valor ao produto final, ou mesmo simplificar, uma vez que forem aplicados no processo de desenvolvimento de software. A ideia basilar do DDD é a de que o seu código fonte não é o mais importante em um software, nem a arquitetura utilizada, nem a tecnologia sobre a qual foi desenvolvido, e sim a dificuldade que o mesmo se sugere a esclarecer, ou em outras palavras, a regra de negócio (DE SOUZA, 2016).

Diante do exposto, percebe-se, no mercado de desenvolvimento de software a necessidade de atender aos requisitos do cliente e a essencialidade das constantes mudanças desses requisitos.

Sendo assim, se faz necessário aprofundar os conhecimentos sobre a linguagem e o funcionamento do *Domain Driven Designer*.

O presente trabalho está estruturado em capítulos e, além desta introdução teórica que traz os objetivos e a justificativa do trabalho, ele será dividido da seguinte forma:

- Capítulo 2: apresenta a fundamentação teórica sobre o tema, explorando a Engenharia de Software, Arquiteturas de Software e o *Domain Driven Design* (DDD);
- Capítulo 3: descreve o desenvolvimento do trabalho, e
- Capítulo 4: possui as considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 ENGENHARIA DE SOFTWARE

Para acompanhar o crescente avanço tecnológico, a maneira de criação de tecnologias de software alterou o método como as soluções são produzidas, acrescentando assim novos desafios no sistema de ensino da Engenharia de Software (ES) (MEIRELES; BONIFÁCIO, 2015).

Devido à sua amplitude de conhecimentos, a área ES é frequentemente abordada por meio de disciplinas no decorrer do curso de Ciência da Computação (ALVES; BENITTI, 2006). Ela é uma disciplina que envolve todos os aspectos da produção de software, a contar dos estágios iniciais de especificação até a manutenção do sistema (VICTORINO; BRÄSCHER, 2009).

A Engenharia de Software representa uma das matérias de maior relevância na área da Computação e, tem como propósito, execução de teoria, conhecimento e prática para o desenvolvimento eficiente e competente de sistemas de software que respondam aos requisitos dos usuários (PORTELA, VASCONCELOS, OLIVEIRA; 2017).

É uma das áreas com maior relevância, sendo indispensável para a formação de engenheiros de software, onde sua finalidade é corroborar para o desenvolvimento profissional (SOUZA; FRANÇA, 2016).

A aprendizagem da ES deriva de várias áreas de conhecimento, fundamentando-se em uma atuação interdisciplinar e com conhecimentos prévios adquiridos através de outras disciplinas. Outro ponto significativo é o seu aspecto tecnicista, que estabelece comportamentos éticos, trabalho em equipe e experiências práticas em aspectos técnicos no desenvolvimento de soluções em software (ALVES; BENITTI, 2006).

De modo geral, as atividades ligadas a ES que apresentam funções associadas à modelagem de dados ou da informação são execuções de levantamento e análise de requisitos, onde no seu princípio não há interesse em uso de linguagens documentárias ou definições consensuais e seu foco é a automação dos processos através do desenvolvimento de software (VICTORINO; BRÄSCHER, 2009).

A ES se cria por meio de um conjunto de fases, onde cada uma dessas pode abranger métodos, ferramentas e procedimentos, das quais as formas de organização são referidas como modelo de Engenharia de Software, integrando três fases: definição, desenvolvimento e manutenção (AZEVEDO JUNIOR; CAMPOS, 2008).

2.2 ARQUITETURA DE SOFTWARE

Incorporado na Engenharia de Software, a Arquitetura de software procura assessorar o gerenciamento da complexidade do desenvolvimento de sistemas, propondo o ensinamento das estruturas desses sistemas e suas correlações e forma a organizar o entendimento, com intenção de alcançar as qualidades do desenvolvimento (DOMINGOS, 2004).

A expressão “arquitetura de software” foi apresentada pela primeira vez no fim da década de 60 por Edsger Dijkstra, entretanto só obteve relevância na década de 90 e, desde então, várias definições surgiram, porém todas com ideias em comum e que se completam (NHIMI, 2016).

Amui (2015) refere-se à arquitetura de software como uma “organização geral do software e aos modos pelos quais disponibiliza integridade conceitual para um sistema”.

De acordo com Perry e Wolf (1992), a Arquitetura de Software é um agrupamento de elementos arquiteturais que contêm alguma organização e, dentre esses constituintes, estão os elementos de processamento, que empregam ou modificam a informação; os componentes de dados, que possuem a mensagem a ser usada e modificada e os componentes de conexão, que conectam elementos de qualquer tipo entre si.

É imprescindível para a Engenharia de Software está voltada para as demandas estruturais e tecnológicas do desenvolvimento (DOMINGOS, 2004). Como seus objetivos, podem-se citar a diminuição dos riscos associados ao projeto e as funções que o rodeiam e, evidenciam-se entre os benefícios obtidos a contenção de risco correlacionado ao desenvolvimento e manutenção; a construção de aplicações flexíveis e de qualidade e a criação de valor aos negócios do cliente (NHIMI, 2016).

2.3 DDD - DOMAIN DRIVEN DESIGN

A criação de um software não é algo simples, uma vez que se deve adquirir um bom modelo de domínio para monitorar a complexidade do software que será desenvolvido. Tal modelo deve abranger elementos relevantes significativos para a construção do domínio, de forma que os desenvolvedores venham a conseguir um aproveitamento suficiente (SANTOS, 2015).

Na incessante busca de assegurar a agilidade nas técnicas para o desenvolvimento de um software surgiram várias técnicas e métodos criados para as inúmeras fases ligadas ao desenvolvimento do projeto. Entre elas, apresentando uma abordagem voltada para o domínio do problema nomeada *Domain Driven Design* (DDD) (MATTOS; DOLL; ALMEIDA, 2010).

Agregado na Engenharia de Software, essa abordagem de desenvolvimento de sistemas engloba técnicas e métodos que salientam a associação com o cliente, onde o sucesso e/ou a aceitabilidade de um software estão intimamente ligados ao relacionamento das metodologias utilizadas pelo DDD (DA SILVA; FILHO; DA SILVA, 2018).

O DDD é executado como uma maneira de guiar métodos de desenvolvimento de modo ágil (EVANS, 2003) oferecendo uma alternativa de representar o mundo autêntico na arquitetura, focando e reconhecendo o domínio central, que são por sua vez características relevantes para se obter uma qualidade superior da arquitetura de software (HIPPCHEN et.al., 2018).

Mattos; Doll; Almeida (2010) ratificam que a primeira aparição do termo *Domain Driven Desing* foi em 2004, no livro de Eric Evans, com título “*Domain Driven Desing: atacando as complexidades no coração do software*”, onde o autor conceitua DDD como uma forma de pensar e um agrupamento de princípios que irão cooperar na criação de sistemas potentes pelos desenvolvedores.

Evans é dono de uma empresa especializada em assessoria para desenvolvimento de software, e seu livro é fundamentado em suas experiências no decorrer de mais de 20 anos trabalhando na área de desenvolvimento de software (CUKIER, 2010).

Uma das distinções do DDD em comparação com outras filosofias de projeto é o panorama dado a modelagem, onde a modelagem de domínio e a implementação de aplicação não são consideradas como atividades diferentes, mas sim que se apresenta em outros métodos, sendo um modelo apresentado em códigos (MACEDO, 2009).

Alguns domínios compreendem o mundo físico, outros podem ser atingíveis, a carga de informações pode ser grande, porém, alguns modelos são considerados dispositivos redutores dessa sobrecarga de informações (SANTOS, 2015).

Na figura 1, pode-se observar as quatro camadas essenciais seguindo os princípios do DDD, sendo elas:

1 – Camada de interface de Usuário: Camada que exibe informações do sistema e executa comandos do usuário (CUKIER, 2010; MATTOS; DOLL; ALMEIDA, 2010).

2 – Camada de aplicação: Não possui a logicidade do negócio, limitando apenas por ser encarregada por vincular a Interface de Usuário com as camadas inferiores (CUKIER, 2010; MATTOS; DOLL; ALMEIDA, 2010).

3 – Camada de domínio: onde todo o centro do DDD se encontra, deve se manter o mais isolado possível de todas as outras camadas (CUKIER, 2010; SANTOS, 2015)

4 – Camada de Infra estrutura: oferece recursos técnicos que darão suporte às camadas superiores, atuando como uma biblioteca e concedendo informações entre elas (CUKIER, 2010; MATTOS; DOLL; ALMEIDA, 2010).

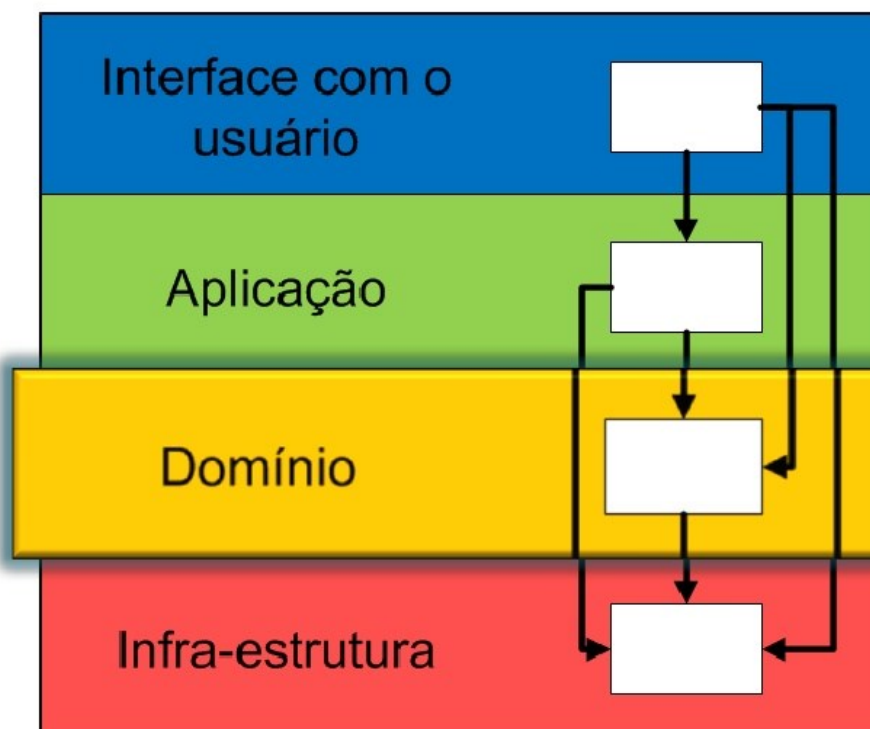


Figura 1: Arquitetura de camadas
Fonte: GOOGLE Imagens

Cada uma dessas camadas deve desempenhar apenas a função a qual se confere, uma vez que o isolamento da camada domínio é uma premissa para o DDD, sendo esta camada o centro do DDD (MATTOS; DOLL; ALMEIDA, 2010).

Para que o software consiga servir otimamente a um domínio específico, é indispensável que previamente que se estipule uma Linguagem Ubíqua, estabelecendo as definições que integram as conversas cotidianas entre os especialistas e os desenvolvedores, sendo compreendido por todos, sem ambiguidades (COSTA; HILD, s/d; CUKIER, 2010).

Evans (2003) define a Linguagem Ubíqua como sendo uma linguagem comum com termos claramente definidos, que integram o domínio do negócio e que são aplicados por todos os envolvidos no processo do desenvolvimento do sistema.

Para que a Linguagem Ubíqua tenha positividade no desenvolvimento do software, ela deve representar os desejos dos especialistas e desenvolvedores do domínio, sendo assim, o seu uso gera elaboração do domínio do modelo, minimizando a necessidade de análise e modelos de projetos e possibilitando a concentração da equipe de projeto nas tarefas primordiais durante o desenvolvimento de uma funcionalidade estabelecida (SANTOS, 2015).

Na figura 2, podem-se observar os padrões em DDD para criar a camada de domínio, também conhecidos como blocos de construção.



Figura 2: Mapa de navegação da linguagem do Design Dirigido por Modelos
Fonte: SANTOS, 2015

1 - Entidades: Classes de objetos que carecem de uma identidade, a definição desse objeto deve ser simples, estabelecida pela sua identidade e não seus atributos. Enquadra configurações especiais de modelagem e de design (CUKIER, 2010; MATTOS; DOLL; ALMEIDA, 2010; SANTOS, 2015). A identidade não é estabelecida pela união dos atributos ou pelo estado do objeto, mas sim por uma ininterruptão temporal, sendo que identidades mal acordadas podem gerar uma incongruência de dados (MACEDO, 2009).

2 - Objetos de valor: Ao contrário do item anterior, os objetos de valor não possuem uma identidade ou descrevem alguma característica, sendo que na maioria das vezes são imutáveis e possuem um curto ciclo de vida (MATTOS; DOLL; ALMEIDA, 2010; SANTOS, 2015).

3 - Serviços: São as classes que contém lógica de negócio, porém não pertencem a uma Entidade ou a um Objeto de valor (COSTA; HILD, s/d), devem apresentar algum significado para o domínio (MACEDO, 2009).

4 - Módulos: No decorrer da evolução do modelo, cresce o número de conceitos com o qual o desenvolvedor irá lidar, assim como o número de classes e suas associações, gerando assim a inevitabilidade de agrupar as classes em módulos, que nada mais são do que blocos lógicos onde o conjunto de classes da aplicação é particionado (MACEDO, 2009).

5 - Agregados: Este bloco exerce a manutenção da integridade do modelo (CUKIER, 2010); podem ser definidos como conjuntos de Entidades ou Objetos de Valores que são agrupados em uma classe única (MATTOS; DOLL; ALMEIDA, 2010).

6- Fábricas: São empregadas para agrupar a informação necessária para a elaboração de objetos complexos ou agregados, gerando uma interface que reproduz os desejos do cliente e uma visão subjetiva do objeto a ser construído (MATTOS; DOLL; ALMEIDA, 2010; SANTOS, 2015).

7 - Repositórios: Têm como objetivo agrupar toda a lógica necessária para atingir a referência de um objeto, onde estes não necessitam lidar com a infraestrutura para apanhar informações essenciais de outros objetos de domínio (COSTA; HILD, s/d). São classes responsáveis por controlar o ciclo de vida dos outros objetos (Entidades, Objetos de valor, Agregados); concentram operações de criação, alteração e remoção de objetos (CUKIER, 2010).

Um conceito importante para o DDD é o Bounded Context que são delimitadores ou agrupamentos de domínios de um determinado negócio, isso significa que as entidades desse contexto devem ser construídas expressando as intenções do contexto para os quais a entidade venha a pertencer.

A maior complexidade vista no desenvolvimento de um sistema é compreender o domínio do problema a ser solucionado e suas regras. Partindo do estudo do domínio, cria-se o modelo, que pode ser definido como uma estrutura de conhecimento facilitado, onde acomoda conhecimentos válidos e evidencia o problema que será introduzido. O

modelo de domínio não atua somente como um diagrama simples, mas sim dá conceito que o diagrama objetiva transferir (COSTA; HILD, s/d).

Uma vez que o desenvolvimento permanece evidenciado no modelo de domínio, a etapa principal deve ser o isolamento dos conceitos que correspondem o entendimento desse domínio, sendo o modo preferível para atingir tal isolamento a utilização de uma arquitetura multicamadas (MACEDO, 2009).

A viabilidade de isolar responsabilidades distintas em camadas diferenciadas é uma das vantagens de se usar tal arquitetura. O projeto se beneficia especialmente do isolamento da camada de domínio, possibilitando ao desenvolvedor se centralizar na definição do domínio e em seu relacionamento. O isolamento dessa camada sustenta a totalidade do projeto (MACEDO, 2009).

A diferença prevalecente entre o DDD e as demais abordagens de desenvolvimento é o mérito referido a modelagem de domínio, uma vez que esta modelagem e a implementação são tratadas como atividades semelhantes que não progridem mutuamente (COSTA; HILD, s/d).

3 PROTÓTIPO CRIADO

3.1 METODOLOGIA

Em um primeiro momento foi realizada uma revisão de literatura, baseada em consultas em livros, dissertações e artigos científicos selecionados através de busca em livros e sites da web, utilizando as plataformas de pesquisa Google Acadêmico, Scielo e Periódicos da CAPES.

Após, foi identificada a demanda de negócio de uma empresa que realiza decorações de festas, e, através desta necessidade foi criado um protótipo de aplicação back-end utilizando as práticas do DDD.

Neste protótipo não houve implementação da parte web (front-end) da aplicação, desenvolvendo apenas a parte back-end.

3.2 DESENVOLVIMENTO

3.2.1 Modelo de negócio

Protótipo criado visando a melhoria e organização de uma empresa de decoração de festa, de pequeno porte e familiar. Atualmente todo o controle é feito de forma manual anotado em agendas, o que pode gerar confusão na hora de alugar materiais e decorações de festas, uma vez que a empresa presta serviço em mais de um local no mesmo dia.

Foi desenvolvido no protótipo os tópicos referentes a (figura 3):

- 1) Festa: Contempla os itens referentes a identificação do evento, data do evento, buffet, quantidade de convidados e forma de pagamento.
- 2) Tema: decoração escolhida.
- 3) Peças decorativas: materiais que serão usados na decoração da festa.
- 4) Cliente: dados do cliente como nome, CPF, endereço.
- 5) Salão: identificação do endereço da festa.
- 6) Telefone: contato do cliente e do responsável pelo local da festa.

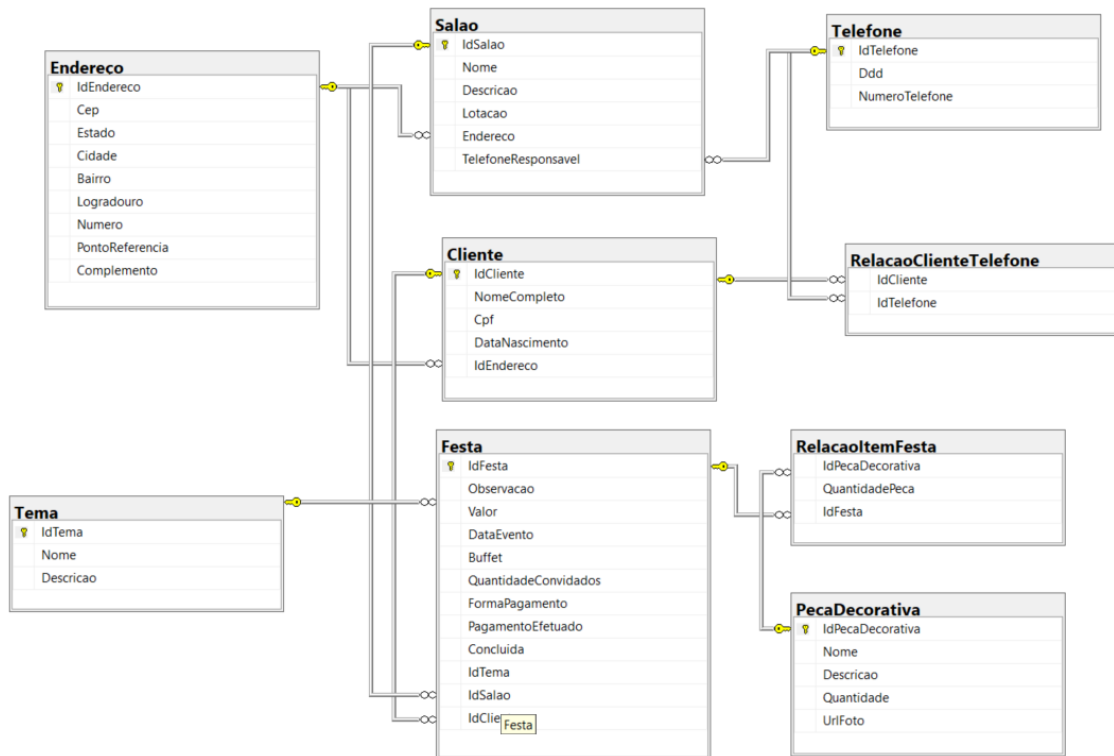


Figura 3: Modelo Banco de Dados
Fonte: O autor

3.2.2 Ferramentas utilizadas na implementação

- **.NET Core**: O .NET é uma plataforma de desenvolvimento de software livre gratuita para a criação de diversos tipos de aplicativos que compartilha a funcionalidade entre diferentes aplicativos usando bibliotecas de classes (MICROSOFT, 2020).
- **Linguagem C#**: É uma linguagem de programação moderna, orientada a objeto e de tipo seguro. Foi desenvolvida pela Microsoft como uma parte do framework .Net. O C# oferece muitos recursos para os desenvolvedores, permitindo que criem muitos tipos de aplicativos seguros e robustos que são executados no ecossistema do .NET. (MICROSOFT, 2021).
- **Visual Studio**: O ambiente de desenvolvimento integrado do Visual Studio é um painel de inicialização criativo que possui diversas ferramentas, utilizadas para editar, depurar e compilar o código e, em seguida, publicar um aplicativo. Um IDE (ambiente de desenvolvimento integrado) é um programa com muitos

recursos que pode ser usado por muitos aspectos do desenvolvimento de software (MICROSOFT, 2019).

- Dapper: É um mapeador de objetos simples para .NET, sendo um ORM que em termos de velocidade e é virtualmente tão rápido quanto usar um leitor de dados ADO.NET bruto. Um ORM é um Mapeador Relacional de Objeto, que é responsável pelo mapeamento entre banco de dados e linguagem de programação (DAPPER, s/d).
- Flunt: É uma biblioteca do .NET Core que implementa um padrão de notificações em sua aplicação para concentrar erros e mutação em determinadas ações e entidades (BALTA.IO, s/d).

3.2.3 Aplicando conceitos do DDD

No protótipo foi implementado a arquitetura em camadas proposta pelo DDD, aplicando todos esses conceito a um único bounded context, utilizando da estrutura de classes representada pela figura 4 que será explicada com mais detalhes nas próximas sessões desse capítulo.

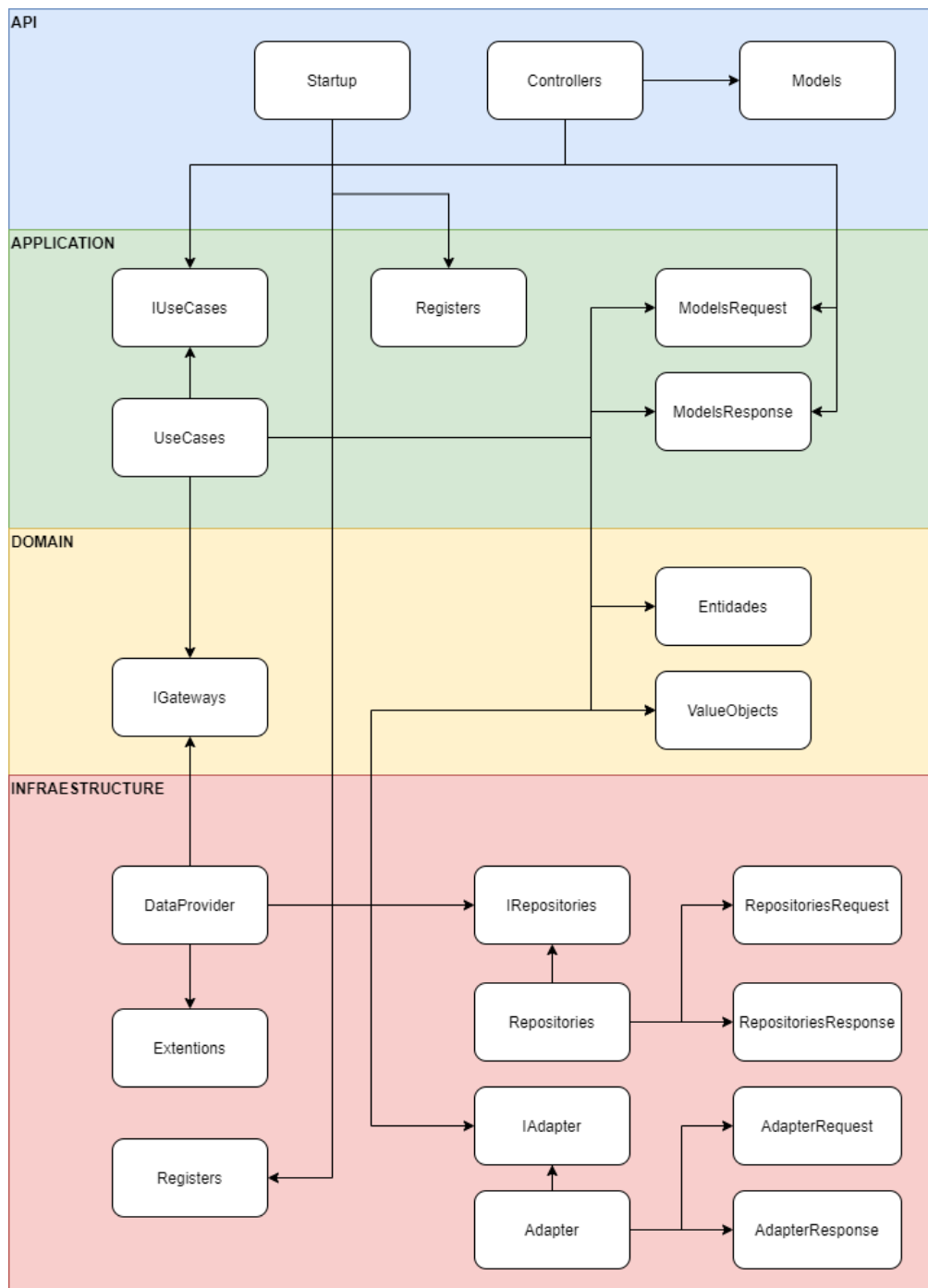


Figura 4: Camadas e classes do DDD
Fonte: O autor

3.2.3.1 Camada API (Application Programming Interface)

Nessa camada da aplicação foi desenvolvida toda a parte de interface com os sistemas externos que utilizam o protótipo. Nessa camada temos os arquivos `Program.cs` e `Startup.cs` que são as classes responsáveis por iniciar o nosso sistema e também temos uma pasta chamada `Controllers` que dentro dessa temos todas as controllers do que podem

ser utilizadas no nosso sistema. Atendendo a seguinte estrutura de pastas e classes (Figura 5):

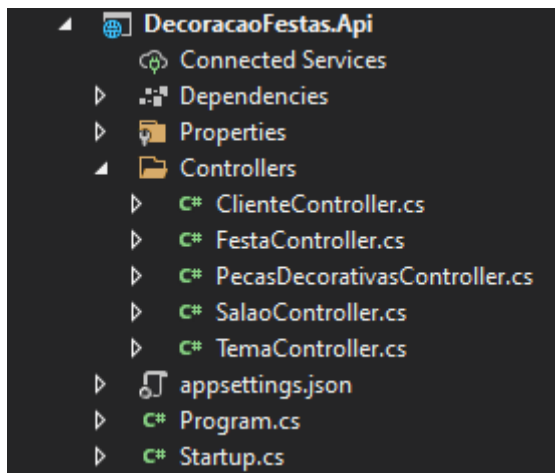


Figura 5: Estruturação de pastas e classes da camada de API
Fonte: O autor

É nessa camada que foi definida toda a parte de configurações de injeção de dependência do .NET Core localizada na classe Startup, como se vê na figura 6:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.RegisterApplicationServices();
    services.RegisterInfraServices();
    services.RegisterDataBase(Configuration);

    services.AddSwaggerGen();
}
```

Figura 6: Registrando injeção de dependência
Fonte: O autor

Também na mesma camada foi desenvolvido as controllers da API. Como o exemplo de uma consulta de todas as festas que foram realizadas com o mesmo tema. Nessa classe em específico podemos observar toda a injeção de dependência das classes de interfaces que foram implementadas na camada de aplicação do protótipo, toda a padronização de logs do sistema e retornos esperados pelo cliente, conforme a figura 7.

```

[Route("festa")]
[ApiController]
public class FestaController : ControllerBase
{
    private readonly ILogger<FestaController> logger;
    private readonly IConsultarFestaUseCase consultarFestaUseCase;
    private readonly IAtualizarFestaUseCase atualizarFestaUseCase;
    private readonly ICadastrarFestaUseCase cadastrarFestaUseCase;

    public FestaController(ILogger<FestaController> logger,
        IConsultarFestaUseCase consultarFestaUseCase,
        IAtualizarFestaUseCase atualizarFestaUseCase,
        ICadastrarFestaUseCase cadastrarFestaUseCase)
    {
        this.logger = logger;
        this.consultarFestaUseCase = consultarFestaUseCase;
        this.atualizarFestaUseCase = atualizarFestaUseCase;
        this.cadastrarFestaUseCase = cadastrarFestaUseCase;
    }

    [HttpGet("/tema/{nome_tema}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status400BadRequest)]
    [ProducesResponseType(StatusCodes.Status500InternalServerError)]
    public async Task<IActionResult> ConsultarFestasPorTema(
        [FromRoute(Name = "nome_tema")] [BindRequired] string nomeTema)
    {
        try
        {
            logger.LogDebug("Mensagem recebida ConsultarFestasPorTema, nome do tema: {nomeTema}", nomeTema);

            var festas = await consultarFestaUseCase.ConsultarFestasPorTema(nomeTema);

            if (festas.IsSuccess)
            {
                logger.LogInformation("Requisição concluída com sucesso!");
                return Ok(festas.Value);
            }
            else
            {
                logger.LogError("Ocorreu um erro ao processar a requisição. Erro: {@0}", festas.Error);
                return BadRequest("Ocorreu uma falha ao processar a requisição");
            }
        }
        catch (Exception ex)
        {
            logger.LogError("Ocorreu um erro ao processar a requisição. Erro: {@0}", ex);
            return StatusCode(StatusCodes.Status500InternalServerError, "Ocorreu um erro ao processar a requisição.");
        }
    }
}

```

Figura 7: Controller para consulta de todas as festas sobre um tema
 Fonte: O autor

Na figura 7 temos todas as interfaces dos casos de uso sendo definidas e posteriormente recebidas via injeção de dependência no construtor da classe. Também foi definido um método ConsultarFestaPorTema onde esse método consulta o caso de uso consultarFestaUseCase e trata o retorno para apresentar para o cliente.

Ainda sobre o exemplo da figura 7 foram desenvolvidos os testes unitários, no qual foi utilizado a biblioteca Moq para conseguir fazer o setup de cada teste de unidade. Na figura 8 temos um exemplo de uma classe de testes unitários, onde estão sendo criados diversos mocks para os casos de usos que são utilizados na classe FestaController. Na figura 9 temos um caso de teste que implementa o método

ConsultarFestasPorTema_FluxoSucesso_OkObjectResult, nesse método utilizamos os mocks gerados para configurar um retorno para o caso de uso utilizado, realiza a chamada para a controller e valida se o retorno está de acordo com o esperado para esse caso de teste.

```
17     public class FestaControllerTest
18     {
19         private readonly Mock<ILogger<FestaController>> loggerMock;
20         private readonly Mock<IConsultarFestaUseCase> consultarFestaUseCaseMock;
21         private readonly Mock<IAtualizarFestaUseCase> atualizarFestaUseCaseMock;
22         private readonly Mock<ICadastrarFestaUseCase> cadastrarFestaUseCaseMock;
23         private readonly FestaController festaController;
24
25         public FestaControllerTest()
26         {
27             this.loggerMock = new Mock<ILogger<FestaController>>();
28             this.consultarFestaUseCaseMock = new Mock<IConsultarFestaUseCase>();
29             this.atualizarFestaUseCaseMock = new Mock<IAtualizarFestaUseCase>();
30             this.cadastrarFestaUseCaseMock = new Mock<ICadastrarFestaUseCase>();
31
32             festaController = new FestaController(loggerMock.Object,
33                 consultarFestaUseCaseMock.Object,
34                 atualizarFestaUseCaseMock.Object,
35                 cadastrarFestaUseCaseMock.Object);
36         }
37     }
```

Figura 8: Construtor teste controller

Fonte: O autor

```

38 [Fact]
39 public void ConsultarFestasPorTema_FluxoSucesso_OkObjectResult()
40 {
41     //Arrange
42     var listaRetorno = new List<FestaResult>();
43     listaRetorno.Add(new FestaResult
44     {
45         IdFesta = 1,
46         Observacao = "Teste Observacao",
47         Valor = 500,
48         DataEvento = "01-06-2021",
49         Buffet = "",
50         QuantidadeConvidados = 50,
51         FormaPagamento = "Debito",
52         PagamentoEfetuado = false,
53         Concluida = false,
54         IdSalao = 2,
55         IdCliente = 3,
56         IdTema = 4
57     });
58
59     consultarFestaUseCaseMock.Setup(f => f.ConsultarFestasPorTema(It.IsAny<string>()))
60         .ReturnsAsync(Result.Success<List<FestaResult>>(listaRetorno));
61
62     //Act
63     var resultado = festaController.ConsultarFestasPorTema("Nome Tema Teste");
64     resultado.Wait();
65
66     //Assert
67     var tipoResultado = Assert.IsType<OkObjectResult>(resultado.Result);
68     var valor = (List<FestaResult>)tipoResultado.Value;
69     Assert.Equal(1, valor.FirstOrDefault().IdFesta);
70     Assert.Equal("Teste Observacao", valor.FirstOrDefault().Observacao);
71     Assert.Equal(500, valor.FirstOrDefault().Valor);
72     Assert.Equal("01-06-2021", valor.FirstOrDefault().DataEvento);
73     Assert.Equal("", valor.FirstOrDefault().Buffet);
74     Assert.Equal(50, valor.FirstOrDefault().QuantidadeConvidados);
75     Assert.Equal("Debito", valor.FirstOrDefault().FormaPagamento);
76     Assert.False(valor.FirstOrDefault().PagamentoEfetuado);
77     Assert.False(valor.FirstOrDefault().Concluida);
78     Assert.Equal(2, valor.FirstOrDefault().IdSalao);
79     Assert.Equal(3, valor.FirstOrDefault().IdCliente);
80     Assert.Equal(4, valor.FirstOrDefault().IdTema);
81 }
82 }

```

Figura 9: Teste controller

Fonte: O autor

3.2.3.2 Camada Aplicação

A camada de aplicação é responsável por receber todas as requisições da camada de API e repassa-las para os casos de uso corretos. Ela é quem dá o acesso a todas as camadas inferiores do sistema e foi implementada com a estrutura de pastas e fontes (Figura 10). Nessa estrutura podemos notar que temos as interfaces de casos de uso, os próprios casos de uso que implementam essas interfaces, os modelos utilizados (objetos de requisição e de resposta para cada caso de uso) e também as extensões que estão os

métodos para transformações de objetos (objetos de domínio sendo transformados em objetos de aplicação).

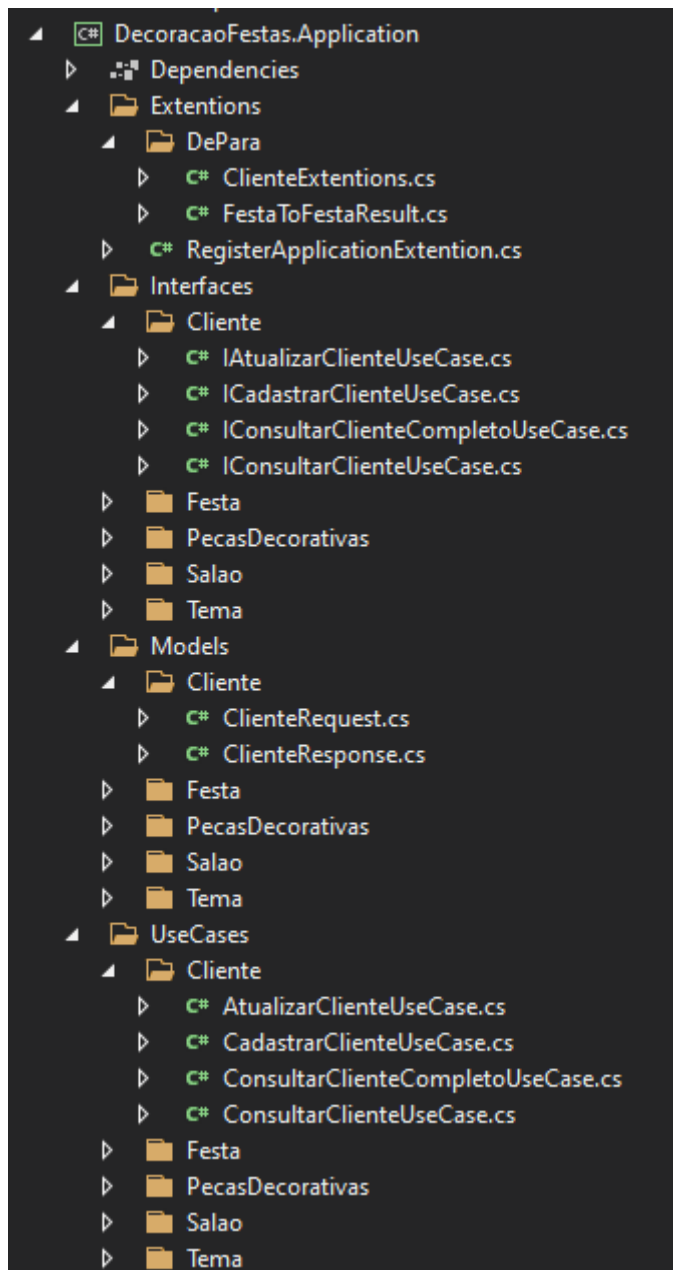


Figura 10: Estruturação de pastas e classes da camada de Aplicação
Fonte: O autor

Utilizando o mesmo exemplo dado na seção 4.2.2.1 temos a classe ConsultarFestaUseCase.cs nessa classe foi codificado todo o caso de uso para consultar todas as festas realizadas com um tema em específico. Na figura 11 podemos ver essa codificação e observar que temos a injeção de dependência das classes de interface da camada de domínio que nesse caso tem o nome de gateway.

No método ConsultarFestasPorTema temos a orquestração dos recursos da camada de infraestrutura pois as classes que implementam esses gateways estão na camada de infra. Nesse exemplo primeiro é consultado o tema e após essa consulta é feita uma validação desse tema retornado e somente depois de ter certeza de que é um tema valido é realizado a consulta de todas as festas que foram realizadas com esse tema e retornado para a controller solicitante.

```
public class ConsultarFestaUseCase : IConsultarFestaUseCase
{
    private readonly ILogger<ConsultarFestaUseCase> logger;
    private readonly ITemaGateway temaGateway;
    private readonly IFestaGateway festaGateway;

    public ConsultarFestaUseCase(ILogger<ConsultarFestaUseCase> logger,
        ITemaGateway temaGateway,
        IFestaGateway festaGateway)
    {
        this.logger = logger;
        this.temaGateway = temaGateway;
        this.festaGateway = festaGateway;
    }

    public async Task<Result<List<FestaResult>>> ConsultarFestasPorTema(string nomeTema)
    {
        logger.LogDebug("Requisição recebida no usecase {0}", GetType().Name);

        var tema = await temaGateway.BuscarTema(nomeTema);

        if (tema.IsSuccess)
        {
            var festas = await festaGateway.BuscarFestasPorTema(tema.Value.idTema);
            var resultado = festas.Value.ToListFestaResult();

            if (!festas.IsSuccess)
            {
                return Result.Failure<List<FestaResult>>(festas.Error);
            }
            return Result.Success(resultado);
        }
        else
        {
            return Result.Failure<List<FestaResult>>(tema.Error);
        }
    }
}
```

Figura 11: Exemplo do caso de uso para consultas de todas as festas
Fonte: O autor

Para o mesmo método acima foi implementado os testes unitários (Figura 12). Onde é definido todos os mocks necessários para esse caso de uso, no teste `ConsultarFestasPorTema_FluxoSucesso_ResultSuccessFestaResult` podemos ver que foi utilizado do mock para configurar um retorno sempre que o gateway for consultado. Sendo assim garantimos que testaremos somente o método `ConsultarFestaPorTema` do caso de uso `consultarFestaUseCase`.

```
public class ConsultarFestaUseCaseTest
{
    private readonly Mock<ILogger<ConsultarFestaUseCase>> loggerMock;
    private readonly Mock<ITemaGateway> temaGateway;
    private readonly Mock<IFestaGateway> festaGateway;

    private readonly ConsultarFestaUseCase consultarFestaUseCase;

    public ConsultarFestaUseCaseTest()
    {
        loggerMock = new Mock<ILogger<ConsultarFestaUseCase>>();
        temaGateway = new Mock<ITemaGateway>();
        festaGateway = new Mock<IFestaGateway>();

        consultarFestaUseCase = new ConsultarFestaUseCase(loggerMock.Object, temaGateway.Object, festaGateway.Object);
    }

    [Fact]
    public async void ConsultarFestasPorTema_FluxoSucesso_ResultSuccessFestaResult()
    {
        //Arrange
        temaGateway.Setup(f => f.BuscarTema(It.IsAny<string>()))
            .ReturnsAsync(Result.Success<Tema>(new Tema { idTema = 1,
                Descricao = "Teste Descricao",
                Nome = "Teste Nome"}));

        festaGateway.Setup(f => f.BuscarFestasPorTema(1))
            .ReturnsAsync(Result.Success<List<Festa>>(RetornaListaFestaValida()));

        //Act
        var resultado = await consultarFestaUseCase.ConsultarFestasPorTema("Teste Nome");

        //Assert
        Assert.True(resultado.IsSuccess);
    }
}
```

Figura 12: Teste unitário do caso de uso para consultas de todas as festas
Fonte: O autor

3.2.3.3 Camada Domínio

Essa é uma camada fundamental para o DDD essa camada foi implementada com a estrutura de pastas e classes da Figura 13, onde temos as entidades, objetos de valor e também as interfaces que serão implementadas pela camada de infraestrutura.

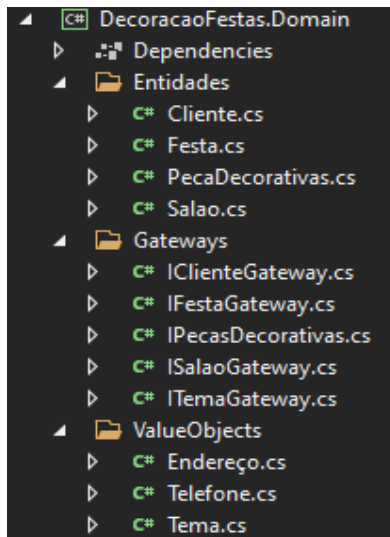


Figura 13: Estruturação de pastas e classes da camada de Domínio
Fonte: O autor

Após um estudo prévio a respeito do domínio foi possível identificar o que é objeto de valor e o que é entidade. Um exemplo de Entidade é a classe Cliente pois pode ser considerado um objeto mutável, é possível identificar fazendo algumas perguntas como: “Se um cliente mudar de endereço ou telefone isso afetará o contato com ele?”. Já os objetos de valores em caso de alteração eles podem ser descartados pois o novo objeto acaba sobrescrevendo o antigo. Um exemplo disso é o Telefone, quando alteramos o código de discagem direta ele assume um valor totalmente novo. Com esse estudo podemos observar na figura 14 o que são objetos de valor e o que é entidade no protótipo.

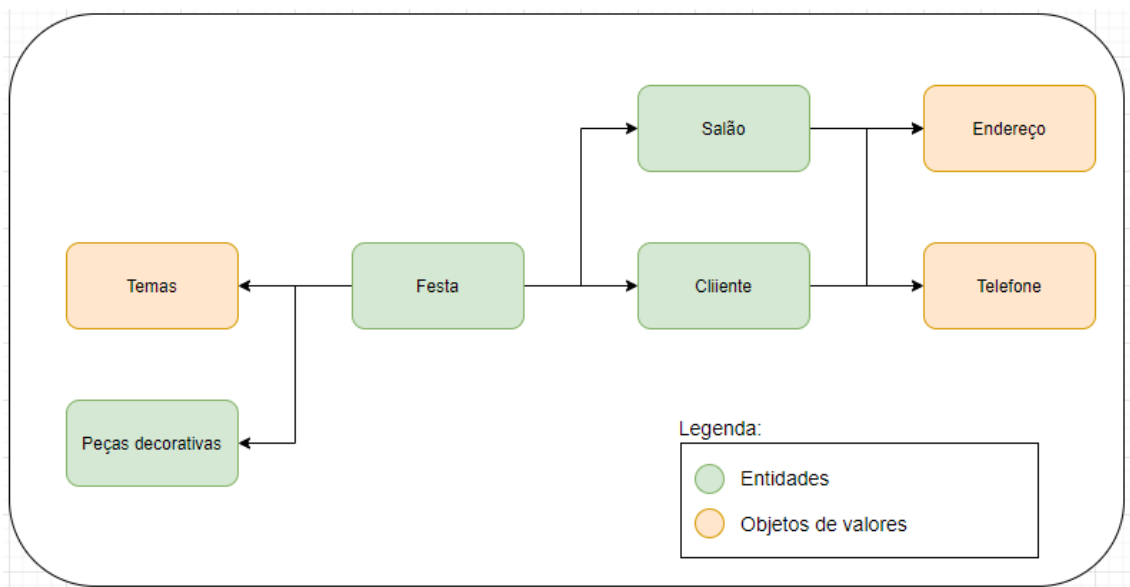


Figura 14: Modelo de classes
Fonte: O autor

Tanto para as entidades quanto para os objetos de valores é necessário garantir que estão validos antes se utilizar um objeto instanciado, para isso foi utilizado um pacote chamado Flunt, esse pacote possibilita a utilização de notificações em caso de um objeto invalido. Como se vê no exemplo da imagem 15

```
public class PecaDecorativas : Notifiable<Notification>
{
    3 references
    public string Nome { get; private set; }
    3 references
    public string Descricao { get; private set; }
    2 references
    public int Quantidade { get; private set; }
    3 references
    public string UrlImagem { get; private set; }

    0 references
    public PecaDecorativas(string nome, string descricao, int quantidade, string urlImagem)
    {
        Nome = nome;
        Descricao = descricao;
        Quantidade = quantidade;
        UrlImagem = urlImagem;

        if (Nome.Length <= 150)
            AddNotification("NomeLengthMaximo", "O nome da peça decorativa precisa ser menor que 150");

        if (Nome.Length > 5)
            AddNotification("NomeLengthMinimo", "O nome da peça decorativa precisa ser maior que 5");

        if (Descricao.Length <= 300)
            AddNotification("DescricaoLengthMaximo", "A descrição da peça decorativa precisa ser menor que 300");

        if (Descricao.Length > 10)
            AddNotification("DescricaoLengthMinimo", "A descrição da peça decorativa precisa ser maior que 10");

        if (UrlImagem.Length <= 300)
            AddNotification("UrlImagemLengthMaximo", "A URL da imagem da peça decorativa precisa ser menor que 300");

        if (UrlImagem.Length > 10)
            AddNotification("UrlImagemLengthMinimo", "A URL da imagem da peça decorativa precisa ser maior que 10");

        if (Quantidade <= 0)
            AddNotification("QuantidadePositiva", "A quantidade da peça decorativa precisa ter um valor positivo e maior que zero");
    }
}
```

Figura 15: Validadores Flunt

Fonte: O autor

3.2.3.4 Camada Infraestrutura

Essa camada é responsável por comunicar com recursos externos a aplicação que são necessários para execução, como banco de dados e API's externas. No Protótipo temos uma camada de infra desenvolvida com a estrutura de pastas e classes (imagem 16)

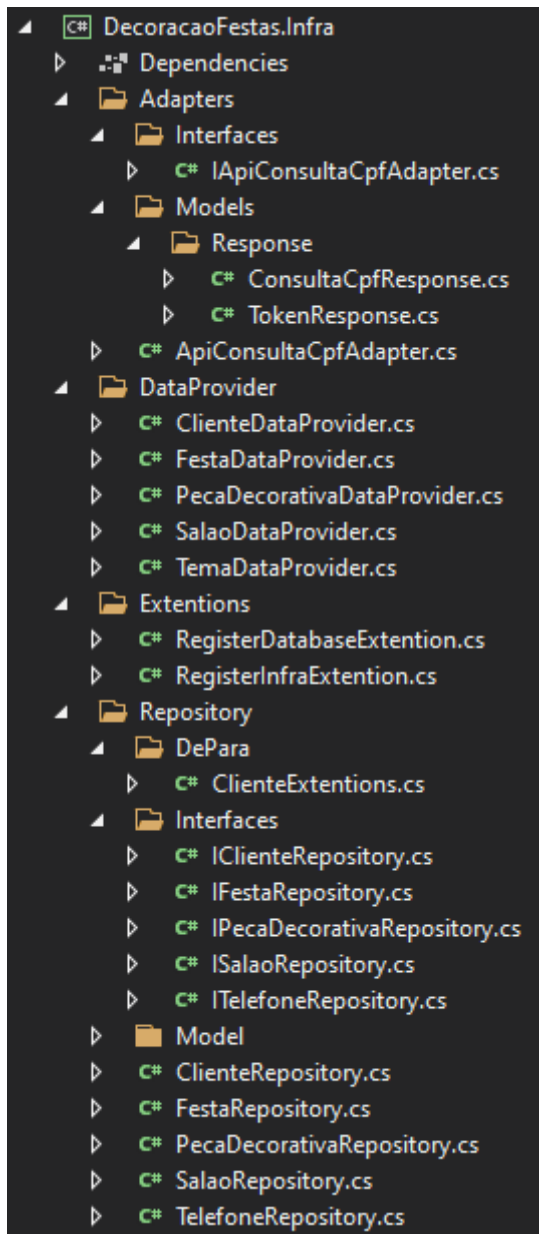


Figura 16: Estruturação de pastas e classes da camada de Infraestrutura
Fonte: O autor

Com essa estrutura podemos ver que temos os repositores, as classes responsáveis por acessar o banco de dados como representado pela imagem 17. Nessas classes temos o método `ConsultaCompletaCliente` que utilizad de um código SQL para execução do comando de consulta no banco de dados. Após essa consulta temos o objeto da consulta sendo transformado em um objeto de domínio para então fazer o retorno da função.

```

4 references
public class ClienteRepository : IClienteRepository
{
    5 references
    public IDbConnection conexao { get; set; }
    1 reference
    public ILogger<ClienteRepository> logger { get; set; }

    0 references
    public ClienteRepository(IDbConnection conexao, ILogger<ClienteRepository> logger)
    {
        this.conexao = conexao;
        this.logger = logger;
    }

    2 references
    public async Task<Result<Cliente, string>> ConsultarCompletaCliente(string cpf)
    {
        var query = @"select cli.IdCliente, cli.NomeCompleto, cli.Cpf, cli.DataNascimento
                        ,ende.IdEndereco, ende.Cep, ende.Estado, ende.Cidade, ende.Bairro, ende
                        ,tel.IdTelefone, tel.Ddd, tel.NumeroTelefone
                        from [GerenciadorFestas].[dbo].[Cliente] cli
                        inner join [GerenciadorFestas].[dbo].[Endereço] ende
                        on ende.IdEndereco = cli.IdEndereco
                        inner join [GerenciadorFestas].[dbo].[RelacaoClienteTelefone] rel
                        on rel.IdCliente = cli.IdCliente
                        inner join [GerenciadorFestas].[dbo].[Telefone] tel
                        on tel.IdTelefone = rel.IdTelefone
                        where Cpf = '11258397676'";

        conexao.Open();

        var comando = new SqlCommand(query, (SqlConnection)conexao);
        var resultado = comando.ExecuteReader();
        var ListaRetorno = new List<ClienteCompletoResult>();

        while (resultado.Read())
        {
            var retorno = new ClienteCompletoResult();

            retorno.IdCliente = resultado.GetInt64(0);
            retorno.NomeCompleto = resultado.GetString(1);
            retorno.Cpf = resultado.GetString(2);
            retorno.DataNascimento = resultado.GetDateTime(3);
            retorno.IdEndereco = resultado.GetInt64(4);
            retorno.Cep = resultado.GetString(5);
            retorno.Estado = resultado.GetString(6);
            retorno.Cidade = resultado.GetString(7);
            retorno.Bairro = resultado.GetString(8);
            retorno.Logradouro = resultado.GetString(9);
            retorno.Numero = resultado.GetString(10);
            retorno.PontoReferencia = resultado.GetString(11);
            retorno.Complemento = resultado.GetString(12);
            retorno.IdTelefone = resultado.GetInt64(13);
            retorno.Ddd = resultado.GetInt32(14);
            retorno.NumeroTelefone = resultado.GetInt64(15);

            ListaRetorno.Add(retorno);
        }

        conexao.Close();
        conexao.Dispose();

        return ListaRetorno.ToClienteDomain();
    }
}

```

Figura 17: Exemplo repositor Consulta informações do cliente

Fonte: O autor

Também temos os data providers que são os serviços que sabem todo o fluxo e locais dentro da infraestrutura para devolver ou consolidar uma informação, o data provider é responsável por conhecer os bancos de dados e aplicações externas. Como na figura 18:

```
public class ClienteDataProvider : IClienteGateway
{
    private readonly IApiConsultaCpfAdapter apiConsultaCpfAdapter;
    private readonly IClienteRepository clienteRepository;

    public ClienteDataProvider(IApiConsultaCpfAdapter apiConsultaCpfAdapter,
        IClienteRepository clienteRepository)
    {
        this.apiConsultaCpfAdapter = apiConsultaCpfAdapter;
        this.clienteRepository = clienteRepository;
    }

    public Task<Result<bool, string>> AtualizarCliente(Cliente cliente)
    {
        return clienteRepository.AtualizarCliente(cliente);
    }

    public Task<Result<bool, string>> CadastrarCliente(Cliente cliente)
    {
        return clienteRepository.CadastrarCliente(cliente);
    }

    public Task<Result<Cliente, string>> ConsultarCliente(string cpf)
    {
        return clienteRepository.ConsultarCliente(cpf);
    }

    public Task<Result<Cliente, string>> ConsultarCompletaCliente(string cpf)
    {
        return clienteRepository.ConsultarCompletaCliente(cpf);
    }

    public Task<Result<bool, string>> ConsultarSituacaoCpf(string cpf)
    {
        return apiConsultaCpfAdapter.ConsultarSituacaoCpf(cpf);
    }
}
```

Figura 18: Exemplo data provider Cliente
Fonte: O autor

Ainda na camada de infraestrutura temos também os adapters, que são responsáveis por garantir a comunicação com sistemas externos, como por exemplo na imagem 19 temos a implementação da comunicação com a API de Consulta do CPF.

Primeiramente o método ConsultarSituacaoCpf obtém um token válido para a comunicação HTTP, tendo esse token é criada uma nova requisição de consulta de CPF utilizando o protocolo HTTP e executada essa requisição. Tendo realizado essa requisição é tratado o retorno e disponibilizado para o requisitante.

```
public class ApiConsultaCpfAdapter : IApiConsultaCpfAdapter
{
    private readonly IConfiguration config;
    private readonly ILogger<ApiConsultaCpfAdapter> logger;

    public ApiConsultaCpfAdapter(IConfiguration config, ILogger<ApiConsultaCpfAdapter> logger)
    {
        this.config = config;
        this.logger = logger;
    }

    public Result<bool, string> ConsultarSituacaoCpf(string cpf)
    {
        try
        {
            var token = BuscarToken();

            var client = new RestClient(config.GetSection("UrlCpf").Value + cpf);
            client.Timeout = -1;
            var request = new RestRequest(Method.GET);
            request.AddHeader("Authorization", "Bearer " + token);
            IRestResponse response = client.Execute(request);

            if (response.StatusCode == HttpStatusCode.OK)
            {
                var cpfResponse = JsonConvert.DeserializeObject<ConsultaCpfResponse>(response.Content);
                return Result.Success<bool, string>(cpfResponse.PossueRestricao);
            }
            else
            {
                logger.LogError("Ocorreu um erro na consulta de CPF, resposta={@0}", response);
                return Result.Failure<bool, string>("Ocorreu um erro ao consultar o serviço de CPF");
            }
        }
        catch (Exception ex)
        {
            logger.LogError("Ocorreu uma exceção na consulta de CPF, exceção={@0}", ex);
            return Result.Failure<bool, string>("Ocorreu um erro ao consultar o serviço de CPF");
        }
    }
}
```

Figura 19: Exemplo adapter consulta CPF.

Fonte: O autor

3.4 RESULTADOS OBTIDOS

Dado o que todos os cenários apresentados na sessão 4.2, podemos identificar que o protótipo criado, utilizando das tecnologias apresentadas na sessão 3.2.2, obteve sucesso na implementação do DDD e na abordagem implementada, apresentando vantagens e desvantagens na utilização do DDD, sendo elas:

Vantagens:

- Utilização de entidades com domínio rico, onde temos informações do domínio centralizadas, isoladas e independentes. Garantindo que todas as regras dessa entidade estão contidas nela mesma.
- Código menos acoplado e mais coeso, um exemplo é caso seja necessário trocar uma base de dados a única camada alterada é de infraestrutura as outras continuam iguais.
- Realizar testes de unidade se torna muito mais fácil, pois temos diversos módulos do sistema que são injetados com utilização de classes de interface, sendo assim fica fácil utilizar ferramentas para setup de código.
- Com a utilização de linguagem ubíqua torna-se mais fácil o entendimento do negócio por parte da equipe de desenvolvimento.

Desvantagens:

- Notamos que o DDD não é algo trivial e que precisa ser estudado, para conseguir aplicar todos os seus conceitos isso causa uma curva de aprendizagem para o desenvolvedor.
- É notável que o DDD tem uma complexidade de código maior, portanto deverá ser indicado para projetos com um domínio rico e uma complexidade grande de negócio.

E de acordo com o apresentado podemos notar que o DDD não é somente vantagens, ele traz consigo algumas desvantagens que também terão que ser levadas em consideração ao optar por essa abordagem.

Durante o desenvolvimento foram levadas essas desvantagens em consideração porém conseguimos entender que as vantagens se sobrepuseram, pois conseguimos atingir riqueza de domínio, apresentando um código fonte desacoplado, testável e de fácil de fácil manutenção.

4 CONCLUSÃO

Conclui-se que no protótipo desenvolvido utilizando da abordagem de DDD conseguiu alcançar os resultados esperados. A aplicação conseguiu obter com êxito a implementação do DDD e teve os propósitos dessa abordagem implementados, como testes de unidades, desenvolvimento iterativo, separação de código em camadas e modelagem de domínio sendo colocados em prática.

Também podemos observar que o DDD não é sempre o indicado, pois se o projeto que está sendo implementado não tem um domínio rico e uma complexidade grande é possível aplicar outras técnicas mais simples e que terão uma entrega mais rápida.

Neste trabalho foi realizado a implementação do modelo de negócio voltado para decoração de festas, que pode auxiliar o nodo de negócio a conseguir expor de forma mais simples e pratica para os seus clientes. E a partir desses resultados discutidos mostra-se a importância de trabalhos futuros, outros escopos e domínios poderiam ser abordados para avaliação mais detalhada do DDD em comparação com outras abordagens de desenvolvimento.

REFERÊNCIAS

ALVES, A. G.; BENITTI, F. B. V. Processo de desenvolvimento integrando disciplinas de engenharia de software. In: **XIV Workshop sobre Educação em Computação–Anais do XXVI Congresso da Sociedade Brasileira de Computação–Anais do Campo Grande, 2006.** Disponível em <https://www.researchgate.net/profile/Fabiane_Benitti/publication/237804009_Processo_de_Desenvolvimento_Integrando_Disciplinas_de_Engenharia_de_Software/links/02e7e5314c7705a63f000000.pdf>. Acesso em abril de 2021.

Amui, S. F. **Processos de desenvolvimento de software.** Rio de Janeiro: SESES, 2015. Disponível em <<https://docplayer.com.br/182889494-Processos-de-desenvolvimento-de-software-autor-saulo-franca-amui.html>>. Acesso em abril de 2021.

AZEVEDO JUNIOR, D. P. de; CAMPOS, R. de. Definição de requisitos de software baseada numa arquitetura de modelagem de negócios. **Prod.**, São Paulo, v. 18, n. 1, p. 26-46, 2008. Available from <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0103-65132008000100003&lng=en&nrm=iso>. <https://doi.org/10.1590/S0103-65132008000100003>. Acesso em março de 2021.

BALTA.IO. **Flunt.** s/d. Disponível em <<https://balta.io/blog/flunt>>. Acesso em maio de 2021.

CORTÉS, M. I. **Informática: Fundamentos de Engenharia de Software.** UAB/UECE. Fortaleza/Ceará. 2013. Disponível em <<file:///C:/Users/user/Downloads/Fundamentos%20de%20Engenharia%20de%20Software.pdf>>. Acesso em março de 2021.

COSTA, A.; HILD, T. A. Estudo da modelagem de software Domain-Driven Design aplicado na faturação de um sistema informatizado para auxílio de Agentes Comunitários de Saúde. s/d. Disponível em <https://www2.unicentro.br/decomp/files/2019/03/TCC_Final_Adriano_Costa.pdf>. Acesso em março de 2021.

DA SILVA, T. A.; FILHO, C. de C. C.; DA SILVA, C. T. M. Investigação ontológica da obra de arte digital: linguagem ubíqua, modelo de domínio e programação voltada para as artes visuais. **V Simpósio Internacional de Inovação em Mídias Interativas – SIIMI.** p. 1-388–416. UFG. Goiânia. 2018. Disponível em <https://files.cercomp.ufg.br/weby/up/777/o/16_-_Teofilo_Augusto.pdf>. Acesso em março de 2021.

DAPPER. **Dapper Tutorial.** s/d. Disponível em <<https://dapper-tutorial.net/dapper>>. Acesso em maio de 2021.

CUKIER, D. DDD – Introdução a Domain Driven Design. Daniel Cukier. **Agile And Art**. Jul. 2010. Disponível em <<http://www.agileandart.com/2010/07/16/ddd-introducao-a-domain-driven-design/>>. Acesso em março de 2021.

DE SOUZA, M. V. R. **Abordagens de testes: Uma comparação entre as metodologias tradicional e ágil**. 2016. Dissertação apresentada como requisito para obtenção do título de Bacharel em Sistemas de Informação da Faculdade Federal de Pernambuco. Recife. 2016. Disponível em: <https://www.cin.ufpe.br/~tg/2016-2/mvrs.pdf>. Acesso em maio de 2021.

DOMINGOS, M. **Uma arquitetura de referência para sistemas de informação e portais de serviços de governo eletrônico**. 2004. Dissertação apresentada ao Programa de Pós Graduação em Engenharia de Produção da Universidade Federal de Santa Catarina como requisito parcial para a obtenção do grau de Mestre em Engenharia de Produção. Florianópolis. 2004. Disponível em <<https://core.ac.uk/download/pdf/30369146.pdf>>. Acesso em maio de 2021.

EVANS, E. **Domain Driven Design: Tackling Complexity in the Heart of Software**. Ed. Addilson-Wesley, 2003.

FIRMINO, G. L.B.; ZANELATO, A. P. A. Desenvolvimento de camadas utilizando DDD. **ETIC-Encontro de Iniciação Científica**, v. 10, n. 10, 2014. Disponível em <<http://intertemas.toledoprudente.edu.br/index.php/ETIC/article/view/3978>>. Acesso em março de 2021.

GIRARDI, R. Engenharia de Software baseada em Agentes. In: **Procedimentos do IV Congresso Brasileiro de Ciência da Computação (CBComp 2004)**. Itajaí/SC, 2004. Disponível em <http://www.niece.ufrgs.br/eventos/CBCOMP/2004/mini_curso/MiniCurso%2006.pdf>. Acesso em março de 2021.

HIPPCHEN, B.; GIESSLER, P.; STEINEGGER, R. H.; SCHNEIDER, M.; ABECK, S. Designing Microservice-Based Applications by Using a Domain-Driven Design Approach. **International Journal on Advances in Software**, v. 10, n. 3 e 4, pág. 432-445, 2018. Disponível em <<https://pdfs.semanticscholar.org/6ee2/67ac6cc4a7986d3b6a74c6a59360f1cbe8e8.pdf>>. Acesso em abril de 2021.

HRUBY, Pavel. Ontology-based domain-driven design. In: **OOPSLA Workshop on Best Practices for Model-Driven Software Development, San Diego, CA, USA**. 2005. p. 38. Disponível em <<https://s23m.com/oopsla2005/hruby.pdf>>. Acesso em abril de 2021.

HU, Ying; PENG, Sam. So we thought we knew money. In: **Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion**. 2007. p. 971-975. Disponível em <<https://dl.acm.org/doi/abs/10.1145/1297846.1297965>>. Acesso em maio de 2021.

LANDRE, E.; WESENBERG, H.; RONNEBERG, H. Architectural improvement by use of strategic level domain-driven design. In: **Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications**. 2006. p. 809-814. Disponível em: <https://dl.acm.org/doi/abs/10.1145/1176617.1176728>>. Acesso em março de 2021.

LIMA, T. de. **Uma análise de soft skills na visão dos profissionais da engenharia de software**. 2018. Dissertação apresentada ao programa de Pós-Graduação em Engenharia de Software da Universidade do Vale dos Sinos (UNISINOS) como requisito parcial para obtenção do título de especialista. São Leopoldo, RS. 2018. Disponível em <http://www.repositorio.jesuita.org.br/bitstream/handle/UNISINOS/7555/Tiago%20de%20Lima_.pdf?sequence=1>. Acesso em abril de 2021.

MACEDO, O. A. C. **Diretrizes para desenvolvimento de linhas de produtos de software com base em Domain-Driven Design e métodos ágeis**. 2009. Tese de Doutorado. Universidade de São Paulo. 2009. Acesso em <<https://www.teses.usp.br/teses/disponiveis/55/55134/tde-24032009-192923/en.php>>. Acesso em abril de 2021.

MATTOS, K. M.; DOLL, L. M.; ALMEIDA, J. DOMAIN-DRIVEN DESIGN E TEST-DRIVEN DEVELOPMENT. **5º Encontro de Engenharia e Tecnologia dos Campos Gerais**. 2010. Disponível em <http://www.5eetcg.uepg.br/Anais/artigospdf/50021_vf1.pdf>. Acesso em abril de 2021.

MEIRELES, M. C.; BONIFÁCIO, B. Uso de métodos ágeis e aprendizagem baseada em problema no ensino de engenharia de software: Um relato de experiência. **Brazilian Symposium on Computers in Education (SBIE)**. Vol. 16, n. 1, p. 180, 2015. Disponível em <<https://www.br-ie.org/pub/index.php/sbie/article/view/5148>>. Acesso em abril de 2021.

MICROSOFT. **Documentação do C#**. 2021. Disponível em <<https://docs.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/>>. Acesso em maio de 2021.

MICROSOFT. **Documentação do IDE do Visual Studio**. 2019. Disponível em <<https://docs.microsoft.com/pt-br/visualstudio/get-started/visual-studio-ide?view=vs-2019>>. Acesso em maio de 2021.

MICROSOFT. **Introdução ao .NET**. 2020. Disponível em <<https://docs.microsoft.com/pt-br/dotnet/core/introduction>>. Acesso em maio de 2021.

NHIMI, F. T. L. R. Princípios e Práticas em Arquitetura de Software. **Instituto de Gestão em Tecnologia da Informação**. 2016. Disponível em <<http://www.machado.mg.gov.br/files/concursos/1cf11cf161fe4eb688dfeec880d6b4d9.pdf>>. Acesso em maio de 2021.

PERRY, D. E.; WOLF, A. L. Foundations for the Study of Software Architecture. **ACM SIGSOFT Software Engineering Notes**, v. 17, n. 4, p. 40, 1992. Disponível em

<<https://www.ics.uci.edu/~taylor/classes/221/Wolf-Perry-1992.pdf>>. Acesso em abril de 2021.

PORTELA, C.; VASCONCELOS, A.; OLIVEIRA, S. R. B. Um Modelo Iterativo para o Ensino de Engenharia de Software Baseado em Abordagens Focadas no Aluno. In: **Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)**. 2017. disponível em <<https://www.br-ie.org/pub/index.php/sbie/article/view/7559>>. Acesso em abril de 2021.

SANTOS, E. C. S. **Integração da abordagem Domain-Driven Design e de técnica Behaviour-Driven Development no desenvolvimento de aplicações web**. 2015. Dissertação apresentada para obtenção do título de Mestre em Ciência da Computação. 2015. Disponível em <<https://repositorio.ufscar.br/handle/ufscar/7588>>. Acesso em abril de 2021.

SOUZA, M.; FRANÇA, C. O Sucesso dos Jogos para Ensino de Disciplinas de Engenharia de Software sob a Ótica de uma Teoria Motivacional. In: **Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)**. 2016. p. 450. Disponível em <<https://www.br-ie.org/pub/index.php/sbie/article/view/6726>>. Acesso em abril de 2021.

VICTORINO, M.; BRÄSCHER, M. Organização da informação e do conhecimento, engenharia de software e arquitetura orientada a serviços: uma abordagem holística para o desenvolvimento de sistemas de informação computadorizados. **DataGramaZero-Revista de Ciência da Informação**, v. 10, n. 3, p. 119, 2009. Disponível em <https://brapci.inf.br/_repositorio/2010/08/pdf_739cdbe8fd_0011617.pdf>. Acesso em abril de 2021.

VIEIRA, R. S.; REIS, U. Abordagem dirigida ao domínio aplicado na arquitetura de sistemas web. **Revista de Sistemas e Computação-RSC**, v. 4, n. 2, 2015. Disponível em <<https://amazon-c.unifacs.br/index.php/rsc/article/view/3090>>. Acesso em abril de 2021.