UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I – CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAELA ALBANIZA OLIVEIRA SANTOS

ARQUITETURA LIMPA: PRINCÍPIOS E PRÁTICAS PARA UM DESENVOLVIMENTO DE SOFTWARE SUSTENTÁVEL

CAMPINA GRANDE
2024

RAFAELA ALBANIZA OLIVEIRA SANTOS

ARQUITETURA LIMPA: PRINCÍPIOS E PRÁTICAS PARA UM DESENVOLVIMENTO DE SOFTWARE SUSTENTÁVEL

Trabalho de Conclusão de Curso apresentado ao Departamento do Curso de Bacharelado em Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação.

Área de concentração: Engenharia de Software.

Orientador: Profª. Drª. Kézia de Vasconcelos Oliveira Dantas

CAMPINA GRANDE
2024

É expressamente proibido a comercialização deste documento, tanto na forma impressa como eletrônica. Sua reprodução total ou parcial é permitida exclusivamente para fins acadêmicos e científicos, desde que na reprodução figure a identificação do autor, título, instituição e ano do trabalho.

S237a Santos, Rafaela Albaniza Oliveira.
Arquitetura limpa [manuscrito] : princípios e práticas para um desenvolvimento de software sustentável / Rafaela Albaniza Oliveira Santos. - 2024.
48 p. : il. colorido.

Digitado.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia, 2024.
"Orientação : Profa. Dra. Kézia de Vasconcelos Oliveira Dantas, Coordenação do Curso de Computação - CCT. "

1. Arquitetura limpa. 2. Manutenibilidade. 3. Desacoplamento. 4. Refatoração. I. Título

21. ed. CDD 005.3

Elaborada por Bruno R. F. de Lima - CRB - 15/1021 BC/UEPB

RAFAELA ALBANIZA OLIVEIRA SANTOS

ARQUITETURA LIMPA: PRINCÍPIOS E PRÁTICAS PARA UM DESENVOLVIMENTO DE SOFTWARE SUSTENTÁVEL

Trabalho de Conclusão de Curso apresentado ao Departamento do Curso de Bacharelado em Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação.

Área de concentração: Engenharia de Software.

Aprovada em: 14/06/2024.

BANCA EXAMINADORA

Kézia de Vasconcelos Oliveira Dantas
Profa. Dra. Kézia de Vasconcelos O. Dantas (CCT/UEPB)
Orientador(a)

Sabrina de F. Souto.
Profa. Dra. Sabrina de Figueirêdo Souto (CCT/UEPB)
Examinador(a)

Diogo Florêncio de Lima.
Me. Diogo Florêncio de Lima (NUTES/UEPB)
Examinador(a)

AGRADECIMENTOS

Agradeço primeiramente a Deus por ser minha fortaleza e não me fazer desistir em tantos momentos que só Ele me entendeu.

Agradeço ao meu irmão, Rafael Oliveira Santos, que de forma inicial me apresentou o mundo da programação e sempre acreditou no meu potencial.

Agradeço ao meu pai, Jozenildo Silva de Oliveira, e à minha mãe, Maria das Neves Barbosa dos Santos, que me deram todos os suportes que precisei para garantir minha educação nas melhores condições.

Agradeço aos meus amigos que acreditaram no meu potencial e me ajudaram a trilhar minha carreira profissional.

Agradeço imensamente ao meu companheiro, parceiro, marido e incrível homem com quem divido minha vida, Rafael Silva Vasconcelos, que me ajudou a finalizar e não desistir do curso.

RESUMO

Este trabalho tem como objetivo investigar e analisar a importância da Arquitetura Limpa no desenvolvimento de software, discorrendo sobre seus princípios fundamentais e suas aplicações práticas, consideradas essenciais na garantia de uma melhor manutenibilidade e desacoplamento significativo. Para o desenvolvimento do trabalho, a metodologia adotada foi dividida em etapas, partindo da pesquisa bibliográfica e finalizando com uma aplicação prática com o estudo de caso de uso, mediante a refatoração de um sistema (aplicação mobile). Assim, o estudo abordou a arquitetura de software e seus padrões; a Arquitetura Limpa e seus princípios fundamentais, bem como as tecnologias utilizadas. A fim de apresentar uma aplicação prática, foi feito também um estudo de caso de uso, por meio do levantamento de requisitos, das tecnologias utilizadas e das aplicações iniciais da Arquitetura Limpa. Na sequência, os resultados obtidos com a refatoração do sistema foram debatidos, revelando a estrutura inicial até a reestruturação efetiva, embasada nos princípios fundamentais de Responsabilidade Única, Separação de Responsabilidades, padrão de *Design Factory*, *State* e o padrão Observer utilizando um pacote BloC do Flutter. Ao final, foi possível constatar a real importância de uma Arquitetura Limpa no desenvolvimento de software realizada por intermédio de um projeto bem arquitetado, como pôde ser observado no estudo de caso de uso por meio do processo de refatoração; e mostrar que a utilização dos princípios proporcionou melhorias significativas tanto na legibilidade do código quanto na modularidade do sistema, pois isso, certamente, resultará num sistema mais eficiente, coeso e de melhor manutenibilidade.

Palavras-chave: arquitetura limpa; manutenibilidade; desacoplamento; refatoração.

ABSTRACT

This work aims to investigate and analyze the importance of clean architecture in software development, discussing its fundamental principles and practical applications, considered essential in ensuring better maintainability and significant decoupling. For the development of the work, the methodology adopted was divided into stages, starting from bibliographical research and ending with a practical application with a use case study, through the refactoring of a system (mobile application). Thus, the study addressed software architecture and its standards, clean architecture and its fundamental principles, as well as the technologies used. In order to present a practical application, a use case study was also carried out, by gathering requirements, the technologies used and the initial applications of clean architecture. Afterwards, the results obtained with the system refactoring were discussed, revealing the initial structure until the effective restructuring, based on the fundamental principles of Single Responsibility, Separation of Responsibilities, Factory Design pattern, State and the Observer pattern using a BloC package from Flutter. In the end, it was possible to verify the real importance of a clean architecture in software development carried out through a well-architected project, as could be observed in the use case study through the refactoring process; and show that the use of the principles provided significant improvements in both the readability of the code and the modularity of the system, as this will certainly result in a more efficient, cohesive and better maintainable system.

Keywords: clean architecture; maintainability; decoupling; refactoring.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
GRASP	*General Responsibility Assignment Software Patterns*
SOA	*Service Oriented Architecture*
SoC	*Separation of Concerns*
SQL	*Structured Query Language*
UI	*User Interface*

SUMÁRIO

1 INTRODUÇÃO

Pesquisas apontam que 90% do tempo gasto pelos brasileiros no celular são em aplicativos móveis. Isso tem impulsionado as empresas a desenvolverem mais aplicativos para se relacionar melhor com o consumidor. Robson Cristovão, diretor comercial da Mouts de tecnologia, afirma que a procura por aplicativos personalizados cresceu significativamente no último ano (Consumidor Moderno, 2022).

Com avanço tecnológico e a crescente demanda por aplicativos e sistemas de software cada vez mais complexos, os desenvolvedores muitas vezes se veem presos em arquiteturas obsoletas e desorganizadas, já que prevalece maior recorrência de sistemas legados. A falta de uma abordagem de uma Arquitetura eficaz leva a problemas como acoplamento excessivo, dificuldade na implementação de novos recursos e testes ineficazes, resultando em ciclos de desenvolvimento prolongados e custos operacionais elevados. Essas estruturas, muitas vezes construídas de forma incremental e sem uma visão clara do design global, resultam em sistemas difíceis de manter, testar e ter uma escalabilidade.

Nesse contexto, é vantajoso adotar um padrão de arquitetura de software, como a Arquitetura Limpa, para garantir a reusabilidade do código (Martin, 2012). Existem outras alternativas que também promovem a modularidade e a manutenção do código, como a Arquitetura em Camadas (*Layered Architecture*), a Arquitetura Hexagonal (*Hexagonal Architecture*) e a Arquitetura Orientada a Serviços (*Service-Oriented Architecture - SOA*). Cada uma dessas arquiteturas oferece diferentes benefícios e pode ser escolhida com base nas necessidades específicas do projeto.

A Arquitetura Limpa surge como uma abordagem promissora para lidar com a complexidade inerente ao desenvolvimento de software, visto que prevalece uma busca por sistemas que sejam não apenas funcionais, mas também escaláveis, sustentáveis e facilmente mantidos. Essa Arquitetura tornou-se uma prioridade em ambientes de desenvolvimento, já que garante um desenvolvimento eficaz.

A Arquitetura Limpa é destacada por princípios de mais independência de UI (*User Interface*), independência de Banco de Dados e de fácil manutenção, isso porque existe a separação de camadas e comunicação entre elas desacopladas. A grande vantagem de utilizar essa arquitetura é a possibilidade de reusabilidade de

código, trazendo mais eficiência e rapidez nas manutenções e desenvolvimento de novas funcionalidades.

Dentre alguns trabalhos relacionados que utilizam dessa arquitetura foi analisado e estudado o artigo “Uma abordagem limpa para o desenvolvimento do flutter por meio do pacote de arquitetura *flutter clean*” (tradução própria). A proposta do trabalho é uma abordagem limpa para programação em Flutter utilizando a Arquitetura Limpa (Boukhary; Colmenares, 2019). Esse trabalho tem como objetivo principal o controle de gerenciamento de estados no *framework Flutter*, fazendo uso da Arquitetura Limpa, conseguindo, assim, obter sucesso nesse objetivo. No propósito de testar a independência e comunicação da Arquitetura, o autor faz uma troca de banco de dados de SQL para MongoDB e, nesse cenário, a aplicação funcionou perfeitamente sem nenhuma outra alteração além da camada onde é implementado o banco de dados. A realização da troca de banco de dados veio a validar que a arquitetura proposta de modo com que não seja alterado nada além da camada desejada que é a de banco de dados.

Outro trabalho relacionado e interessante para abordar é “Implementando *Clean Architecture no ReactJS”* (Sczip, 2020). Onde é implementado um sistema de reconhecimento de imagens em que é enfatizado o princípio de inversão de dependência. O objetivo principal do autor é garantir que as camadas funcionem devidamente separadas e independentes, ou seja, o domínio deve funcionar da mesma forma independentemente do framework ou biblioteca que estiver utilizando. Dessa forma, evidencia que em um futuro poderá realizar troca sem precisar gastar muito tempo e trabalho.

É importante destacar que, inicialmente, a implementação da Arquitetura Limpa em um software pode causar uma complexidade maior para desenvolvedores menos experientes, causando muitas vezes, a desistência dessa implementação. Caso ocorra a implementação ainda não pode ser a garantia de que a Arquitetura vai se manter corretamente, visto que, à medida que o sistema cresce, a quantidade de dependências e a inter-relação entre os módulos pode se tornar difícil de gerenciar. Além disso, a criação de testes unitários e de integração pode ser complexa devido à necessidade de mockar e injetar dependências corretamente. (Michiura, 2020)

Diante do exposto, o presente trabalho requer o estudo e a apresentação de conceitos e demonstrações sobre os princípios de Arquitetura Limpa, bem como

explorar a importância dessa arquitetura para a criação de sistemas de software mais consolidados, flexíveis e com fácil manutenção.

Tal demonstração será representada por meio de um exemplo prático, onde vai ser refatorado um código-fonte, implementando, assim, a Arquitetura Limpa, utilizando-se de alguns princípios fundamentais no desenvolvimento, tais como Responsabilidade Única, Inversão de Dependência, Separação de Responsabilidades, Padrão de *Design Factory* e *State*. Nesse cenário, é importante apresentar essa reestruturação, visto que, para a Arquitetura Limpa ou qualquer outra arquitetura, é um processo longo e custoso em um desenvolvimento de software, principalmente se este é um software legado.

1.1 Objetivo geral

O objetivo deste trabalho é investigar e analisar a importância da Arquitetura Limpa no desenvolvimento de software, explanando a compreensão de seus princípios fundamentais e suas aplicações práticas, as quais serão fundamentais para garantir uma melhor manutenibilidade e desacoplamento significativo.

1.2 Objetivos específicos

- Realizar estudo sobre os conceitos, princípios, desvantagens e vantagens da Arquitetura Limpa.
- Aplicar os princípios GRASP, Responsabilidade Única, Inversão de dependência, Separação de responsabilidades, Padrão de *Design Factory* e *State* da Arquitetura Limpa utilizando um estudo de caso, sendo uma aplicação desenvolvida com o *framework Flutter.*
- Analisar o código reestruturado apresentando as melhorias proporcionadas ao código-fonte, com ênfase na manutenibilidade e no desacoplamento.

1.3 Estrutura do trabalho

O estudo será composto por seis capítulos. A seção introdutória apresenta o conceito e a importância da Arquitetura Limpa no desenvolvimento de software, bem como os objetivos geral e específicos deste trabalho.

Todo o processo de desenvolvimento e pesquisa é apresentado no segundo capítulo que trata sobre a metodologia do estudo.

O terceiro capítulo traz a fundamentação teórica, enfocando conceitos da Arquitetura Limpa e seus princípios.

O estudo de caso de uso é abordado no quarto capítulo, cujo enfoque expõe o levantamento de requisitos, as tecnologias utilizadas e aplicação iniciais da Arquitetura Limpa.

Os resultados e discussões expostos no quinto capítulo mostram a refatoração e implementação da Arquitetura Limpa, cuja abordagem apresenta uma análise das melhorias significativas com essas alterações.

Na última seção são feitas as considerações finais do trabalho, onde é apresentado de forma resumida como ocorreu a implementação da Arquitetura Limpa na aplicação, as melhorias adquiridas dessa refatoração e o planejamento de trabalhos futuros.

2 METODOLOGIA

A metodologia adotada para destacar a importância da Arquitetura Limpa, com uma demonstração de uma refatoração de um sistema neste trabalho foi dividida em cinco etapas.

- **1ª Etapa - pesquisa e apresentação sobre Arquitetura Limpa:** envolveu um processo de conhecimento acerca da Arquitetura Limpa. Nesse caso, foi feita uma revisão bibliográfica, por meio de pesquisas em artigos, sites, livros, a fim apresentar inicialmente uma visão geral da Arquitetura de Software e, por consequência, apontar como ocorreu o surgimento da Arquitetura Limpa.
- **2ª Etapa - pesquisa e apresentação sobre os princípios da Arquitetura Limpa**: trata-se da abordagem de outros pilares necessários para a implementação de uma Arquitetura Limpa. Com isso, foi necessário enfocar de forma mais aprofundada sobre a necessidade de alguns princípios inerentes ao tema, como a Independência de UI, a Independência de Banco de Dados, o Princípio de Responsabilidade Única para esse desenvolvimento.
- **3ª Etapa - análise inicial dos requisitos:** aqui se fez um levantamento de requisitos de uma aplicação já implementada, que será utilizada para o desenvolvimento da Arquitetura Limpa, destacando quais são suas funcionalidades. Essa análise ajudou a entender como funciona o aplicativo e, com isso, apontar as devidas separações de responsabilidades de forma correta e coerente.
- **4ª Etapa - refatoração de um aplicativo e implementação da Arquitetura Limpa:** nesta etapa foi definida uma refatoração de aplicativo para apresentar de forma prática como o Sistema fica mais simples e eficiente. Em primeira análise foi apresentada a visão inicial do aplicativo, como estava sendo realizada sua organização em comparação com a arquitetura desejada, apresentando de forma detalhada a estrutura que se deseja alcançar. Nessa primeira visão geral, foram enfatizados os pontos que seriam necessários modificar, pois estavam ferindo as regras cruciais

da Arquitetura Limpa. Em seguida, de forma detalhada de cada camada, as refatorações foram iniciadas, utilizando os princípios.

- **5ª Etapa - análise dos resultados:** nessa última etapa foi possível identificar as melhorias proporcionadas na refatoração do código, tais como: facilidade para compreensão de código, separações de funcionalidades bem definidas, comunicações corretas através de abstrações, dentre outras.

3 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão apresentados os conceitos principais da Arquitetura de Software, Arquitetura Limpa, conceitos e exemplos de princípios da Arquitetura Limpa e Padrões de *Design*.

3.1 Arquitetura de *software*

Com o avanço da tecnologia e desenvolvimento de sistemas mais complexos, tornou-se imprescindível um gerenciamento e organização eficiente. De forma gradativa, essa evolução da arquitetura foi se tornando evidente e necessária. Nas primeiras décadas (1950-60), os programas eram escritos de forma procedural, onde as instruções eram executadas em sequência.

Ainda na década de 1960 começaram a surgir estruturas de controle de fluxos, como loops e condicionais, que permitiam um melhor controle sobre o fluxo de execução dos programas. Na década de 1970, houve um avanço na criação de módulos reutilizáveis e sub-rotinas, o que permitiu a separação de código em partes mais gerenciáveis e reutilizáveis (Sommerville, 2015).

Os anos 1980 foram marcados pela ascensão da Programação Orientada a Objetos, um paradigma que enfatiza a organização de código em objetos, promovendo a reutilização e a modularidade. Durante os anos 1990 houve um foco crescente em padrões de projeto, que são soluções para problemas comuns de *design* de *software*. Além disso, a ideia de desenvolvimento baseado em componentes começou a ganhar destaque. Desde os anos 2000, tem-se visto um aumento na complexidade dos sistemas de *software*, levando ao desenvolvimento de arquiteturas mais complexas e escaláveis. Arquiteturas como a Arquitetura Orientada a Serviços (SOA), Microservices e Arquitetura Hexagonal ganharam popularidade (Sommerville, 2015).

Atualmente, a arquitetura de software é bastante variada, facilitando a decisão de desenvolvedores sobre o padrão e o estilo ideais para o sistema. Essa prática de arquitetura de *software* envolve considerações não apenas sobre a estrutura do sistema, mas também sobre requisitos não funcionais, como desempenho, escalabilidade, segurança e manutenção.

A arquitetura de *software* define o que é o sistema em termos de componentes computacionais e os relacionamentos entre estes componentes, os padrões que guiam a sua composição e restrições (Shaw; Garlan, 1996). A arquitetura de *software* é um conjunto de elementos arquiteturais (de dados, de processamento, de conexão) que possuem alguma organização. Os elementos e sua organização são definidos por decisões tomadas para satisfazer objetivos e restrições (Perry; Wolf, 1992).

Uma arquitetura deve, além de atender as demandas imediatas dos usuários, desenvolvedores e proprietários, atender também essas expectativas ao longo do tempo (Martin, 2017). Em 2017, Robert Martin propôs a Arquitetura Limpa, que tem como objetivo desenvolver aplicações coesas, com independência de tecnologias e ferramentas e garantindo a reusabilidade do código.

3.2 Padrões GRASP

Os padrões GRASP (*General Responsibility Assignment Software Patterns*) consistem em uma série de princípios baseados em conceitos para atribuição de responsabilidades a classes e objetos na construção de bons *softwares* usando programação orientada a objetos (Boas, 2019). Eles foram introduzidos por Craig Larman em seu livro "*Applying UML and Patterns"* e fornecem diretrizes para a atribuição de responsabilidades em um sistema orientado a objetos. Eles ajudam a identificar como atribuir responsabilidades a classes e objetos em um sistema de *software*.

Alguns dos padrões GRASP mais importantes:

- **Controlador (*Controller*)**: define um objeto responsável por receber e gerenciar solicitações do usuário, coordenando as operações necessárias para atendê-las.
- **Criação de Objetos (*Creator*)**: ajuda a determinar quais classes devem ser responsáveis por criar instâncias de outras classes.
- ***Expert* (Perito)**: determina qual classe possui a informação necessária para cumprir uma determinada responsabilidade.
- **Baixo acoplamento (*Low coupling*)**: visa manter as interações entre classes os mais flexíveis e independentes possíveis, minimizando o acoplamento entre elas.

- **Alta coesão (*High cohesion*)**: sugere que as responsabilidades de uma classe devem estar intimamente relacionadas e focadas em um único propósito.
- **Polimorfismo (*Polymorphism*)**: encoraja o uso de polimorfismo para permitir que objetos de diferentes classes sejam tratados de maneira uniforme.
- **Protegendo a aariação (*Protected variations*)**: propõe estratégias para proteger partes do sistema que podem mudar frequentemente de mudanças indesejadas em outras partes do sistema.
- **Indireção (*indirection*)**: introduz um intermediário entre duas classes para reduzir o acoplamento direto entre elas.

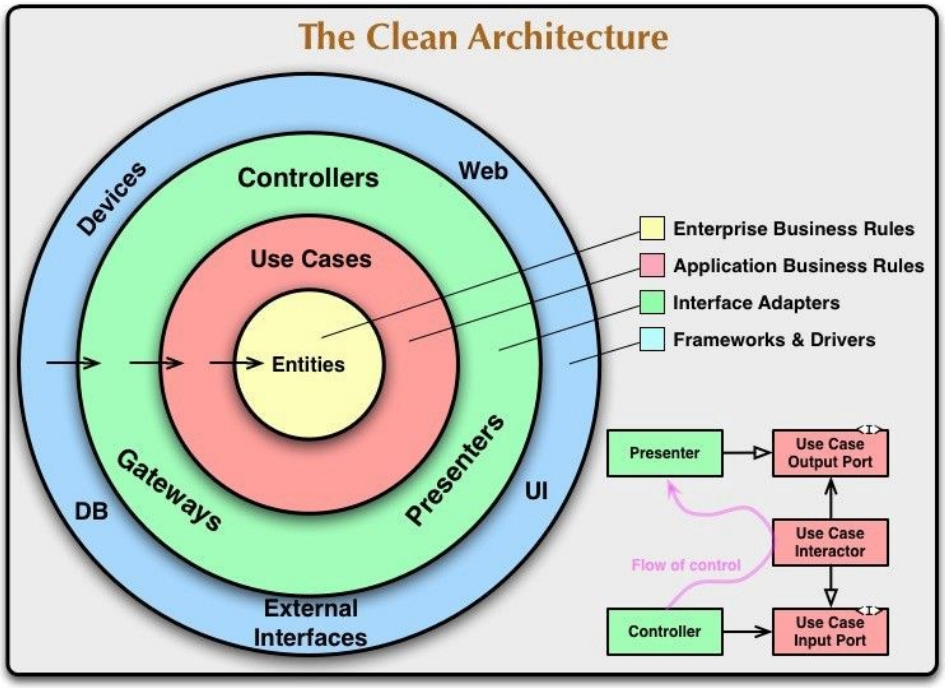
3.3 Arquitetura limpa

A Arquitetura Limpa foi criada em meados de 2012 por Robert Cecil Martin, mais conhecido na comunidade de TI como "*Uncle Bob*", e consolidado por Bob apenas em 2017. O objetivo era padronizar e organizar melhor o código que está sendo desenvolvido, para que pudesse garantir *softwares* fáceis de entender, manter e alterar no decorrer do tempo. Com esse objetivo, a Arquitetura Limpa poderia proporcionar ao arquiteto modelar as regras de negócio sem interferência externa, isso porque nessa arquitetura o core da aplicação não enxerga o "mundo externo", o que garante mais clareza, desacoplamento, manutenibilidade e escalabilidade.

A Arquitetura Limpa segue os princípios de SoC (Separation of Concerns), conhecido como Separação de Responsabilidades). Trata-se de um princípio que sugere que um sistema deve ser dividido em partes distintas, onde cada parte é responsável por uma única preocupação.

A figura 1 mostra a divisão da Arquitetura Limpa em camadas:

Figura 1 – Arquitetura Limpa



Fonte: Martin (2012).

- ***Entities* (entidades ou modelos de domínio):** é a camada mais interna e contém as classes que representam os conceitos de negócio do sistema. Elas são independentes de qualquer detalhe de implementação e descrevem os objetos e suas relações.
- ***Use Cases* (ou interatores):** contém os casos de uso ou interações que representam os cenários e funcionalidades da aplicação. Os *use cases* orquestram a interação entre as entidades para atender às necessidades de negócio.
- **Controllers (controladores ou adaptadores de interface do usuário):** são responsáveis pela comunicação com o mundo externo. Podem ser interfaces de linha de comando, APIs HTTP, interfaces gráficas, entre outros. Eles convertem as entradas e saídas externas em operações que podem ser consumidas pelas camadas internas.
- ***Gateways* (ou adaptadores de dados):** lidam com a comunicação com fontes de dados externas, como bancos de dados, APIs externas, serviços de terceiros etc. Os *gateways* traduzem as operações das camadas

internas em operações específicas para os diferentes tipos de fontes de dados.

A disposição das camadas segue uma hierarquia, com as camadas mais internas contendo a lógica de negócio e as camadas mais externas lidando com detalhes de implementação e interação com o ambiente externo.

Essa organização proporciona benefícios significativos, como a facilidade de testar as funcionalidades do sistema de forma isolada, a flexibilidade para substituir tecnologias ou componentes sem impactar o núcleo da aplicação, e a capacidade de manter um código mais organizado e compreensível ao longo do tempo.

Não seguir os princípios da Arquitetura Limpa pode levar a uma série de problemas em um projeto de *software*, conforme destaca Martin (2012):

- **Dificuldade na manutenção:** código não organizado e sem uma estrutura clara torna a manutenção mais difícil. Isso pode resultar em equipes gastando mais tempo e esforço para fazer alterações ou corrigir bugs.
- **Dificuldade de entendimento:** um código mal organizado pode ser difícil de compreender, especialmente para desenvolvedores que não estiveram envolvidos na fase inicial.
- **Acoplamento excessivo:** sem uma arquitetura bem definida, os componentes podem se tornar fortemente acoplados, o que dificulta a substituição ou modificação de partes do sistema sem afetar outras áreas.
- **Falta de reutilização:** componentes mal organizados e altamente acoplados tornam mais difícil a reutilização de código em diferentes partes do projeto ou em projetos futuros.
- **Dificuldades nos testes:** código mal estruturado e altamente acoplado pode ser difícil de testar de forma isolada. Isso pode resultar em testes mais complexos e menos eficazes.
- **Dificuldade em adotar novas tecnologias:** sem uma arquitetura bem definida, pode ser mais difícil adotar novas tecnologias ou *frameworks*, pois eles podem estar fortemente integrados ao código existente.
- **Risco de falhas de segurança:** uma arquitetura mal estruturada pode levar a vulnerabilidades de segurança, especialmente se as preocupações de segurança não forem separadas adequadamente.

- **Problemas de desempenho:** a falta de uma organização adequada pode resultar em código ineficiente, causando problemas de desempenho e escalabilidade.
- **Dificuldades na colaboração:** desenvolvedores podem ter dificuldade em trabalhar em conjunto de forma eficaz se o código não estiver bem organizado e documentado.
- **Código "espaguete":** sem uma Arquitetura Limpa, o código pode se tornar confuso e desorganizado, conhecido como "código espaguete". Isso dificulta a leitura e a manutenção.

Assim, para o objeto do presente trabalho, serão identificados e enfatizados dois problemas na aplicação, os quais serão refatorados no estudo de caso de uso: o primeiro se refere à dificuldade para manter o aplicativo, visto que se torna desgastante a resolução de bugs, alterações de código e implementação de novas features; e o segundo está relacionado ao acoplamento excessivo, que dificulta o desenvolvimento de uma aplicação, pois outras áreas são afetadas quando se realiza alguma implementação.

3.3.1 Regra de dependência

Um dos princípios mais fundamentais da Arquitetura Limpa é a regra de dependência, onde é definido que camadas mais internas não devem depender de camadas mais externas. O nome de um elemento declarado em uma camada externa não deve ser mencionado pelo código de uma camada interna. Isso inclui funções, classes, variáveis e qualquer outro elemento de código (Martin, 2012).

Esse princípio significa que o código no núcleo do sistema (geralmente o domínio ou a camada de negócios) não deve depender de detalhes de implementação ou tecnologias específicas nas camadas externas (como a interface do usuário ou a infraestrutura).

Na análise de um exemplo de sistema que gerencia pedidos online, têm-se três camadas principais:

- núcleo (entidades, regras de negócio e lógica central do sistema);
- casos de uso (interação entre camada de domínio e interface do usuário); e
- interface do usuário (inclui componentes como banco de dados, sistemas de arquivos, APIs externas etc.).

Ao supor que se tenha uma regra de negócio no núcleo, verificando se um pedido pode ser enviado e que isso dependa do estoque disponível para cada item no pedido, a camada de domínio tem uma função ***verificarDisponibilidade()***, que executa essa verificação.

Um exemplo a não seguir de como seria essa entidade, pode ser visto na figura 2:

Figura 2 – Entidade pedido

Fonte: Elaborada pela autora, 2023.

Esse exemplo não deve ser seguido, pois a função ***"verificarDisponibilidade()"*** que se encontra na camada de domínio está dependendo diretamente da infraestrutura para obter informações de estoque. Isso, claramente, viola o princípio de uma camada mais interna não depender de camadas externas.

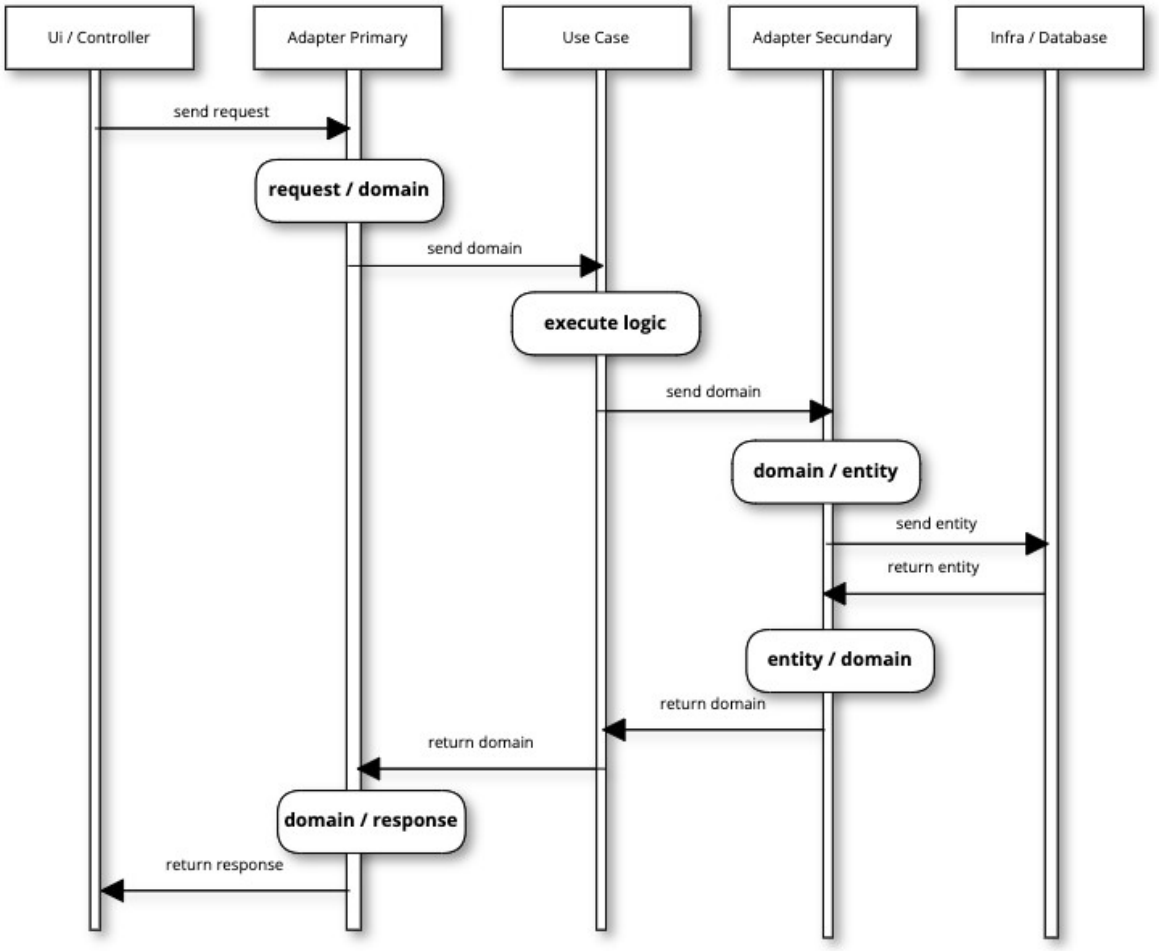
A forma correta seria a função ***"verificarDisponibilidade()",*** ou seja, receber um objeto como argumento, como mostra a figura 3:

Figura 3 – Entidade Pedido

Fonte: Elaborada pela autora, 2023.

O ***RepositorioEstoque*****,** visto na figura como um parâmetro, é uma abstração que a camada de domínio entende e pode ser implementada de várias maneiras, como, por exemplo, usando um banco de dados, uma API, etc. Com isso, a camada de domínio não depende diretamente da infraestrutura, mas sim de uma abstração que pode ser fornecida por qualquer camada externa, cumprindo o fluxo correto de requisição, conforme pode ser visto na figura 4:

Figura 4 – Fluxo Arquitetura Limpa



Fonte: Barbosa Filho (2023).

Na figura acima é apresentado um fluxo mais claro para enfatizar que nenhuma camada interna pode saber sobre camadas externas. Inicialmente, têm-se as entradas de dados na UI, conversão de seus valores na camada *Adapter Primary*, execução de caso de uso, conversão de dados para entidade no *Adapter Secundary* e finalizando com a chamada externa ao banco de dados. Isso permite que a camada de domínio permaneça independente de detalhes de implementação,

facilitando os testes e tornando o código mais flexível e adaptável a mudanças nas camadas externas.

Para proporcionar um melhor desenvolvimento, a Arquitetura Limpa é potencializada implementando em conjunto com padrões de *design*. A Arquitetura Limpa propõe uma estrutura em camadas, como a de entidades, de caso de uso, de interface do usuário etc. Os padrões de *design* podem ser aplicados em cada uma dessas camadas para resolver problemas específicos de *design* dentro delas, mantendo a separação de preocupações e a clareza arquitetural.

3.3.2 Padrões de design (design patterns)

O termo "d*esign patterns*" (padrões de *design*) foi popularizado pelo livro "*Design Patterns: Elements of Reusable Object-Oriented Software*" ou "Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos" (Gamma; Helm; Johnson; Vlissides, 1994), também conhecidos como "*gang of four*" (gangue dos quatro). O livro foi publicado em 1994 e é amplamente considerado um marco na Engenharia de Software (Refactoring Guru, [s.d.]).

Os padrões de *design* são ferramentas valiosas para implementar os princípios da Arquitetura Limpa. Eles são projetados para promover a reutilização de código, facilitar a manutenção e melhorar a comunicação entre os membros da equipe. O que pode garantir uma Arquitetura Limpa e organizada.

Desde a publicação do livro dos *gang of four*, muitos outros padrões de projeto foram identificados e documentados pela comunidade de desenvolvedores. Os *design patterns* são agora uma parte fundamental do conhecimento em Engenharia de Software e são vastamente utilizados em muitas linguagens de programação e ambientes de desenvolvimento.

3.3.2.1 Categorias de design patterns

Inicialmente, é necessário identificar o problema que será solucionado, visto que *design patterns* são soluções generalizadas e reutilizáveis para problemas comuns no *design* de *software*. Após a identificação do problema, deve-se escolher o padrão adequado.

Os padrões são organizados em categorias, como **padrões de criação**, **padrões estruturais** e **padrões comportamentais.** Utiliza-se com mais detalhes na refatoração de aplicativos deste estudo, dois padrões importantes para desenvolvimento: padrão factory e padrão state. Com isso, é dado seguimento para a implementação do padrão no código, porém é necessário garantir que esteja cumprindo todos os requisitos definidos pelo padrão. Apresenta-se a seguir os principais *design patterns* e suas categorias.

3.3.2.2 Padrões de criação

Os padrões de criação são aqueles que focam na forma como os objetos são criados, garantindo assim que uma classe seja instanciada da forma correta. Eles ajudam a tornar um sistema independentemente de como seus objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto um padrão de criação de objeto delegará a instanciação para outro objeto.

Alguns dos padrões de criação são listados abaixo:

- ***Singleton*:** garante que um objeto terá apenas uma única instância, isto é, que uma classe irá gerar apenas um objeto e que este estará disponível de forma única para todo o escopo de uma aplicação.
- ***Factory method*:** define uma interface para criar objetos, mas permite que as subclasses decidam que classe instanciar. Também é conhecido como construtor virtual, possibilitando adiar a criação do objeto a subclasses.
- ***Abstract factory*:** deve ser aplicado quando se deseja isolar a aplicação da implementação da classe concreta, que poderia ser um componente e ou *framework* específico no qual a aplicação conheceria apenas uma interface e a implementação concreta seria conhecida apenas em tempo de execução ou compilação.

3.3.2.3 Padrões estruturais

Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. Os de classes utilizam a herança para compor interfaces ou implementações, e os de objeto, ao invés de compor

interfaces ou implementações, descrevem maneiras de compor objetos para obter novas funcionalidades. A flexibilidade obtida pela composição de objetos provém da capacidade de mudar a composição em tempo de execução, o que não é possível com a composição estática (herança de classes).

São exemplos de alguns dos padrões estruturais:

- ***Adapter*:** permite que a interface de uma classe seja usada como outra interface. Atua como um intermediário entre dois objetos, convertendo a interface de um objeto em outra interface que um cliente espera encontrar.
- ***Composite*:** compartilha a mesma interface para um grupo de objetos. É útil quando se quer tratar objetos individuais e composições de objetos de maneira uniforme.
- ***Decorator*:** anexa responsabilidades adicionais a um objeto de forma dinâmica. É uma alternativa flexível à herança para estender funcionalidades.
- ***Facade*:** funciona como uma fachada ou uma “cara amigável” para o sistema subjacente, oferecendo um conjunto de métodos simplificados que podem ser chamados pelos clientes em vez de lidar diretamente com a complexidade do sistema.

3.3.2.4 Padrões comportamentais

Os padrões comportamentais de classes utilizam a herança para distribuir o comportamento entre classes, e os padrões de comportamento de objeto utilizam a composição de objetos em contrapartida à herança. São descritos como grupos de objetos que cooperam para a execução de uma tarefa que não poderia ser executada por um objeto sozinho. Descreve- se a seguir alguns desses padrões comportamentais:

- ***Observer*:** define uma dependência entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.
- ***Strategy*:** define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Isso permite que o algoritmo varie independentemente dos clientes que o utilizam.

- ***State*:** permite que um objeto altere seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado de classe.

3.3.3 SOLID

O SOLID é a base da Arquitetura Limpa, é um conjunto de princípios de *design* de *software*, não um padrão de *design* específico. Ele é um acrônimo, que representa cinco princípios fundamentais que ajudam a criar um código mais legível, sustentável e flexível (Dias, 2019). A partir desses princípios, é possível garantir a aplicação de uma Arquitetura Limpa. Cada letra do SOLID representa um desses princípios:

3.3.3.1 "S" – Single Responsibility Principle (princípio da responsabilidade única)

De acordo com esse princípio, uma classe deve ter apenas um motivo para mudar. Isso significa que uma classe deve ter uma única responsabilidade e deve fazer apenas uma coisa. Martin (2012) afirma que "se uma classe tem mais de uma responsabilidade, as responsabilidades se tornam acopladas". Para o autor, alterações em uma responsabilidade pode acarretar prejuízos ou inibições na "capacidade da classe de cumprir as outras". *Logo,* "esse tipo de acoplamento leva a projetos frágeis que estragam de maneiras inesperadas quando alterados".

3.3.3.2 "O" – Open/Closed Principle (princípio aberto/fechado)

Nesse princípio, as entidades de *software* (como classes, módulos, funções) devem estar abertas para extensão, mas fechadas para modificação. Isso significa que você pode estender o comportamento de uma classe sem modificar seu código fonte.

No entendimento de Martin (2017), "para que os sistemas de software sejam fáceis de mudar, eles devem ser projetados de modo a permitirem que o comportamento desses sistemas mude pela adição de um novo código em vez da alteração do código existente".

3.3.3.3 “L” – Liskov Substitution Principle (princípio da substituição de Liskov)

Com base nesse princípio, os objetos de uma subclasse devem poder ser substituídos por objetos da classe base sem alterar a correção do programa. Assim, “para construir sistemas de software com partes intercambiáveis, essas partes devem aderir a um contrato que permita que essas partes sejam substituídas umas pelas outras” (Martin, 2017).

3.3.3.4 “I” – Interface segregation principle (princípio da segregação de interfaces)

Uma classe não deve ser forçada a implementar interfaces que ela não utiliza. Em vez disso, as interfaces devem ser divididas em interfaces menores e mais específicas. “O ISP aconselha os projetistas de software a não dependerem de coisas que não usam” (Martin, 2017).

3.3.3.5 “D” – Dependency inversion principle (princípio da inversão de dependência)

Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Além disso, abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações. “O código que implementa a diretiva de alto nível não deve depender do código que implementa detalhes de baixo nível” (Martin, 2017).

3.3.4 Tecnologias utilizadas

Dada a importância das tecnologias utilizadas para a implementação da Arquitetura Limpa, convém discorrer sobre a arquitetura do Flutter adotada para o estudo do caso de uso abordado mais adiante.

3.3.4.1 Framework flutter

O Flutter é considerado uma estrutura de desenvolvimento de código aberto, desenvolvido pela Google. Nessa estrutura é possível desenvolver interfaces para apps, sites e portais para diversas plataformas. Ou seja, é possível criar projetos

para mobile, desktop e web em único código, tornando um processo de desenvolvimento mais simples e rápido. Ele é baseado na linguagem de programação Dart (Flutter).

A arquitetura do Flutter é baseada em três conceitos principais: Widgets, Renderização, Gerenciamento de Estado e Hot Reload (Flutter).

3.3.4.2 Arquitetura do Flutter

- **Widgets:** são os blocos de construção fundamentais de qualquer aplicativo Flutter e representam tudo, desde botões simples até *layouts* complexos (Flutter).
- **Renderização:** quando ocorrem alterações na interface do usuário (como alterações de estado), o Flutter recria apenas os *widgets* afetados, minimizando a quantidade de trabalho necessário para atualizar a interface do usuário (Flutter).
- **Gestão de estado:** o Flutter oferece várias opções para gerenciar o estado do aplicativo. Isso inclui o uso de *StatefulWidget* para *widgets* que precisam de estado mutável e o uso de gerenciadores de estado como *Provider*, Bloc, MobX, entre outros (Flutter).
- **Hot reload:** o Flutter possui uma funcionalidade chamada *Hot Reload* que permite aos desenvolvedores ver as alterações feitas no código imediatamente refletidas no aplicativo em execução (Flutter).

A fim de complementar a revisão teórica exposta, o próximo capítulo trará um estudo de caso de uso e uma visão geral da aplicação utilizada para implementação da Arquitetura Limpa.

4 ESTUDO DE CASO DE USO

Para a implementação e validação da Arquitetura Limpa, foi realizada uma refatoração de um sistema de receitas alimentícias (aplicação desenvolvida durante curso Cod3r - Udemy, e aprimorada para fins profissionais). Nela foram listadas receitas e detalhes com seu passo a passo, e ainda será possível realizar algumas funcionalidades, como filtrar por categoria desejada. Para tal, foi necessário iniciar esse processo com um levantamento de requisitos, bem como identificar quais tecnologias seriam utilizadas nesse desenvolvimento.

4.1 Levantamento de requisitos

O passo inicial para levantar os requisitos apontou as principais funcionalidades identificadas no sistema, conforme relatado a seguir:

- **Visualizar categorias**: na tela inicial do aplicativo foram descritas todas as categorias alimentícias disponíveis para usuário.
- **Visualizar detalhes de receitas pertencentes a categoria específica:** nessa tela de detalhes da receita foram listados os ingredientes necessários para a receitas, o passo a passo de como preparar a receita e também se existia a possibilidade de favoritar a receita.
- **Favoritar receita**: na tela de detalhes das receitas foi possível realizar a ação de favoritar uma receita, o que poderia favorecer ao usuário, mais tarde, o acesso a essa lista de favoritos e imediatamente localizar a receita desejada.
- **Configurar filtro**: nessa funcionalidade foi possível filtrar receitas de acordo com algumas opções como “sem glúten”, “sem lactose”, “vegano” e “vegetariano” na listagem de receita.
- **Visualizar lista de receitas favoritas**: na tela inicial do aplicativo foi criada uma opção para visualizar as receitas que foram favoritadas, possibilitando navegar para detalhes dessa receita.

O quadro 1 expõe o relacionamento entre os requisitos e as telas do sistema, enfatizando a separação de responsabilidade entre eles.

Quadro 1 - Relacionamento entre requisitos e telas referentes no sistema

Requisito	Tela do requisito referente
Detalhes de receitas	Tela da receita
Favoritar receita	Tela da receita
Filtro de listagem	Tela de Filtro
Lista de receitas favoritas	Tela Favoritos

Fonte: Elaborado pela autora, 2024.

4.1.1 Tecnologias utilizadas

O sistema desenvolvido teve como a tecnologia escolhida o *Framework Flutter*. Trata-se de uma ferramenta que utiliza o conceito de *widgets*, que são objetos em Flutter e representam partes da UI (imagens, botões etc.) para construir interfaces de usuário, o que permite uma rápida iteração e customização de elementos visuais.

4.1.2 Aplicação inicial

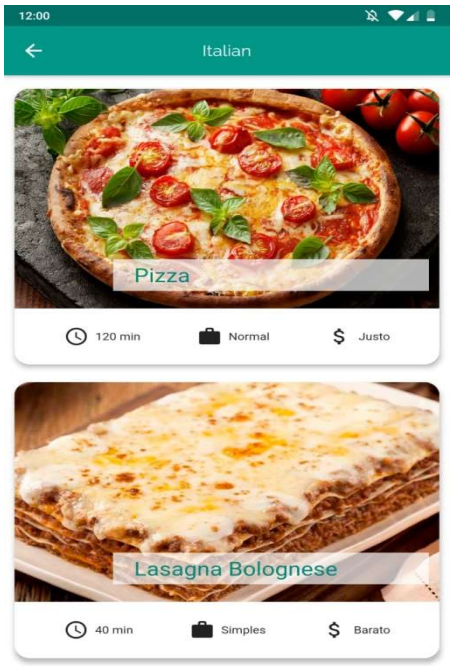
O aplicativo em Flutter consiste em listagem de receitas alimentícias, considerando sua categoria especificada, com diferencial de filtrar alimentos de acordo com o desejado: Sem Glúten, Vegano, Vegetariano e Sem Lactose. A aplicação mostra o passo a passo da receita, ingredientes, possibilidade de configurar para listar receitas com filtro desejado e as receitas favoritas.

A figura 5 tem a possibilidade de visualizar duas abas: ***Categories*** **e** ***Favorites***. A aba *Categories* mostra de forma categorizada as receitas presentes no aplicativo e a aba *Favorites* lista todas as categorias favoritas (figura 10). Já a figura 6 mostra a listagem de receitas da categoria anteriormente. Essa listagem apresenta também o tempo de preparo, dificuldade e custo.

Figura 5 – Listagem de categorias

Fonte: Elaborada pela autora, 2023.

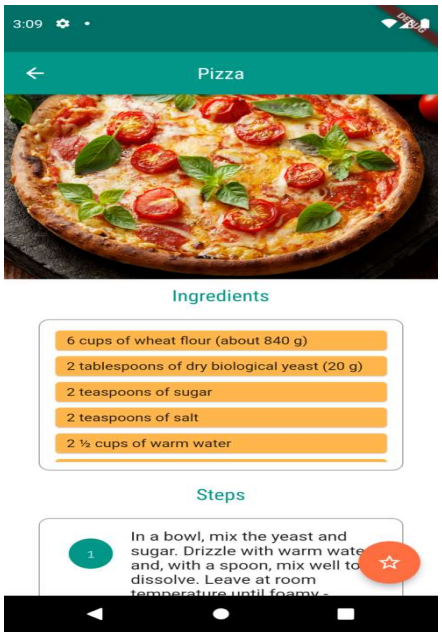
Figura 6 – Listagem de receitas



Fonte: Elaborada pela autora, 2023.

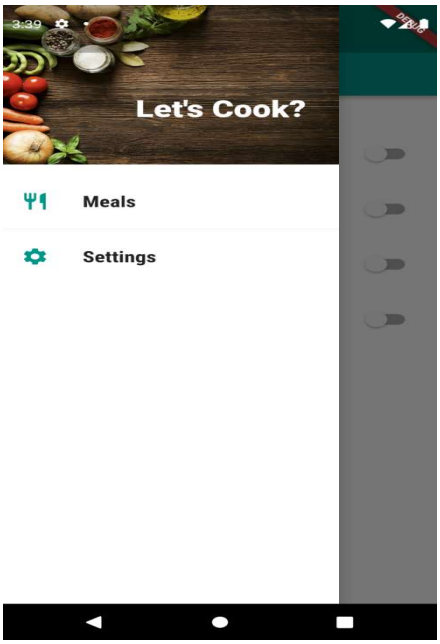
A figura 7 apresenta os detalhes da receita, trazendo consigo a listagem de ingredientes, o passo a passo da receita e o *FloatingButton* para favoritar a receita. Na figura 8 tem-se um componente Drawer, que oferece duas opções: a primeira “Meal”, que redireciona para a tela inicial de categorias; e a opção “Settings”, que apresenta o filtro da listagem (figura 9).

Figura 7 – Tela detalhes da receita



Fonte: Elaborada pela autora, 2023.

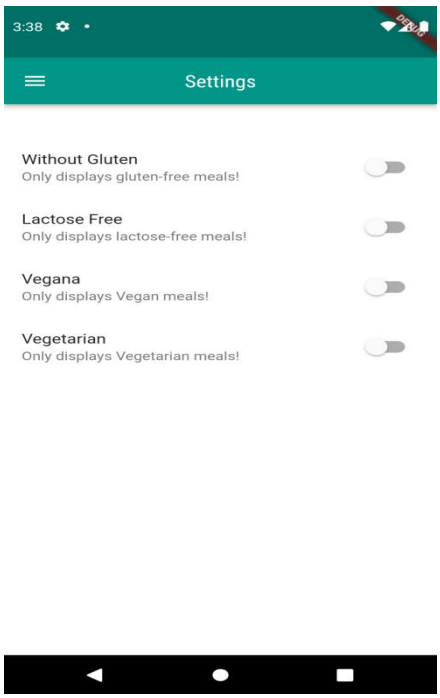
Figura 8 – *Drawer* para acesso ao filtro



Fonte: Elaborada pela autora, 2023.

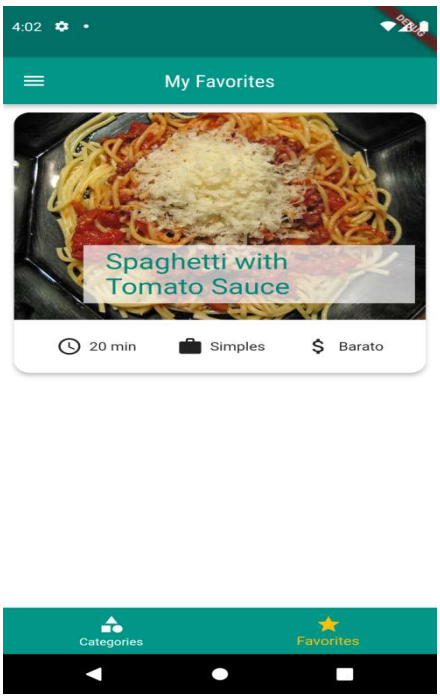
A figura 9 apresenta o filtro da listagem de receitas, enquanto a figura 10 mostra a listagem de receitas favoritas.

Figura 9 – *Settings*



Fonte: Elaborada pela autora, 2023.

Figura 10 – Tela de favoritos



Fonte: Elaborada pela autora, 2023.

5 RESULTADOS E DISCUSSÕES

A fim de garantir o melhor entendimento sobre Arquitetura Limpa e seus princípios, foi utilizada uma refatoração de um sistema como exemplo. Esse sistema é uma aplicação mobile apresentada no Estudo de Caso. Sendo assim, foi apresentada a estrutura inicial, na qual se identificou que não estão sendo utilizados princípios da Arquitetura Limpa; em seguida foi iniciada a reestruturação, com base nos princípios fundamentais como Responsabilidade Única, Separação de Responsabilidades, Padrão de *Design Factory*, *State* e o padrão Observer utilizando um pacote BloC do Flutter.

Nessa refatoração foram desenvolvidos alguns casos de usos, separadas as regras de negócios de desenvolvimento de telas e componentes, e melhoradas as divisões de camadas e suas devidas comunicações, proporcionando um código coeso.

5.1 Visão estrutural inicial da aplicação

Um dos pontos principais para garantir uma estrutura correta da Arquitetura Limpa é a estrutura de pastas. A figura 11 mostra a estrutura inicial:

Figura 11 – Estrutura inicial de pastas

Fonte: Elaborada pela autora, 2023.

É possível observar que a estrutura utiliza de algumas camadas parecidas com Arquitetura Limpa, porém o princípio não foi aplicado corretamente. Na camada “data” não existem definições de contratos, portanto, faltam implementações de casos de uso e ainda ausência de invocações de casos de uso (camada

Presentation). Consequentemente, a UI (*Interface* do Usuário) realiza requisições e conversões de dados, o que fere drasticamente o princípio de Separação de Responsabilidades, proporcionando um problema evidente: dificuldade de manter o aplicativo. Essa dificuldade para manter o aplicativo decorre porque, em uma aplicação, a UI deve ser separada da lógica de negócio e da manipulação de dados.

A figura 12 mostra a camada UI, na qual apresenta as categorias acessando a constante *DUMMY_CATEGORIES* (listagem de categorias), não seguindo, assim, o princípio de responsabilidade única, mediante o qual uma classe deve ter apenas um motivo para mudar, ou seja, deve ter somente uma responsabilidade ou propósito. Nesse caso, fez-se necessária a criação de casos de uso e a camada de *controllers* para realização da comunicação entre a camada de UI e dados.

Figura 12 – Camada UI (*CategoriesScreen*)

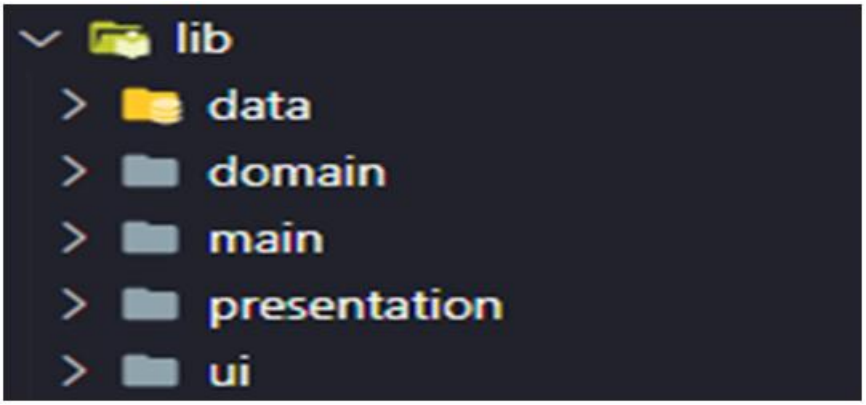
```dart
import 'package:flutter/material.dart';

import '../components/category_item.dart';
import '../data/dummy_data.dart';

class CategoriesScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return GridView(
      padding: const EdgeInsets.all(25),
      gridDelegate: SliverGridDelegateWithMaxCrossAxisExtent(
        maxCrossAxisExtent: 200,
        childAspectRatio: 3 / 2,
        crossAxisSpacing: 20,
        mainAxisSpacing: 20,
      ),
      children: DUMMY_CATEGORIES.map((cat) {
        return CategoryItem(cat);
      }).toList(),
    );
  }
}
```

Fonte: Elaborada pela autora, 2023.

A camada UI também está separada em *screens* e *components*. Essa ausência de separação de camadas e suas respectivas responsabilidades prejudica a manutenção do sistema, isso porque, ao refatorar, prejudica o sistema como um

todo, tornando um código extremamente acoplado. A ideia foi refatorar a estrutura de pastas para essa forma, conforme apresentado na figura 13:

Figura 13 – Estrutura pastas



Fonte: Elaborada pela autora, 2023.

Cada pasta traz consigo a representação das camadas de dados da Arquitetura Limpa:

- **Data:** representa a camada de dados da Arquitetura Limpa, sendo dependente da camada de domínio. Contém as implementações das regras de negócio que são declaradas no *domain*.
- ***Domain*:** representa a camada mais interna da aplicação, contém as regras de negócio.
- ***Main*:** camada onde são realizadas as invocações e injeções de dependências das classes, definitivamente é uma camada que conhece todas as demais.
- ***Presentation*:** lida com a comunicação entre a camada de interface do usuário e a camada de aplicação.
- **UI:** contém os componentes e interfaces visuais que são vistas no sistema, é aqui que propriamente são criadas as telas.

Nesse formato de estrutura, desconsiderou-se a camada infra, já que o sistema não realizará chamadas externas para obtenção de dados, serão dados mockados. A camada Infra contém as implementações referentes ao protocolo HTTP e ao cache, também é único local em que se terá acesso a dependências externas.

5.2 Camada de domínio

Essa camada, representada pela figura 14, armazena a lógica de negócios ou domínio central da aplicação. Inclui entidades, objetos de valor, interfaces de repositório e serviços do domínio.

Figura 14 – Camada domínio

Fonte: Elaborada pela autora, 2023.

Aqui também foram definidas as duas entidades (figuras 15 e 16) que serão usadas pelo caso de uso de refeições e categorias: a *CategoryEntity* e a *MealEntity*.

Figura 15 – *CategoryEntity*

```dart
import 'package:equatable/equatable.dart';
import 'package:flutter/material.dart';

class CategoryEntity extends Equatable {
  final String id;
  final String title;
  final Color color;
  final String? image;
  const CategoryEntity({required this.id, required this.title, this.color = Colors.orange, this.image});

  @override
  List<Object?> get props => [id, title, color, image];
}
```

Fonte: Elaborada pela autora, 2023.

Figura 16 – *MealEntity*

```dart
import 'package:equatable/equatable.dart';

class MealEntity extends Equatable {
  final String id;
  final List<String> categories;
  final String title;
  final String imageUrl;
  final List<String> ingredients;
  final List<String> steps;
  final int duration;
  final bool isGlutenFree;
  final bool isLactoseFree;
  final bool isVegan;
  final bool isVegetarian;
  final Complexity complexity;
  final Cost cost;

  const MealEntity({
    required this.id,
    required this.categories,
    required this.title,
    required this.imageUrl,
    required this.ingredients,
    required this.steps,
    required this.duration,
    required this.isGlutenFree,
    required this.isLactoseFree,
    required this.isVegan,
    required this.isVegetarian,
    required this.complexity,
    required this.cost,
  });

  @override
  List<Object?> get props => [
        id,
        categories,
        title,
        imageUrl,
        ingredients,
        steps,
        duration,
        isGlutenFree,
        isLactoseFree,
        isVegan,
        isVegetarian,
        complexity,
        cost,
      ];
}
```

Fonte: Elaborada pela autora, 2023.

Já na pasta de usecases (figura 17), foram definidos os contratos do caso de uso refeição e categorias, para serem implementados na camada de dados.

Figura 17 – *MealsCaseUses*

```dart
import 'package:meals/domain/entities/meal_entity.dart';

abstract class MealsCaseUses {
  Future<List<MealEntity>> getMeals();
}

```

Fonte: Elaborada pela autora, 2023.

Observa-se que o caso de uso MealsCaseUses não teve valor como entrada e retornou uma lista da entidade definida (MealEntity). O caso de uso CategoriesCasesUses (figura 18) também não teve valor como entrada e retornou uma lista de entidade (CategoriesCaseUse).

Figura 18 – *CategoriesCaseUses*

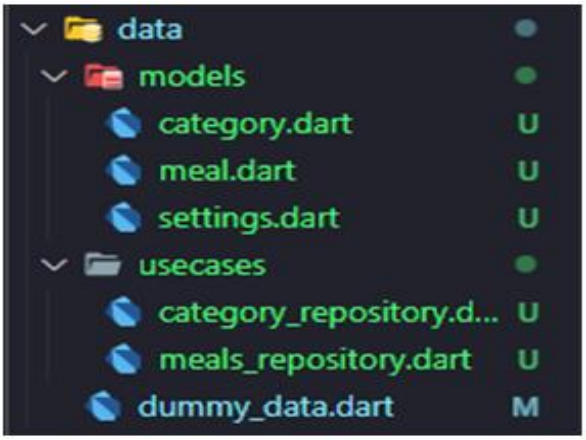
```dart
import 'package:meals/domain/entities/category_entity.dart';

abstract class CategoriesCaseUses {
  Future<List<CategoryEntity>> getCategories();
}
```

Fonte: Elaborada pela autora, 2023.

5.3 Camada de dados

Essa camada representa os dados, sendo totalmente dependente da camada de domínio. Nela foram implementadas as regras de negócio definidas no domínio. A organização das pastas é mostrada na figura 19.

Figura 19 – Camada data



Fonte: Elaborada pela autora, 2023.

Dentro da pasta de usecases, têm-se as implementações concretas dos casos de usos que foram definidos anteriormente na camada de domínio. Para o caso de uso de refeição foi feita uma implementação de MealsRepository e para caso de uso de categorias foi realizada a implementação de CategoryRepository (figura 20).

Figura 20 – *CategoryRepository*

```dart
import 'package:meals/data/dummy_data.dart';
import 'package:meals/domain/entities/category_entity.dart';
import 'package:meals/domain/usecases/categories.dart';

class CategoryRepository extends CategoriesCaseUses {
  @override
  List<CategoryEntity> getCategories() {
    return DUMMY_CATEGORIES.map((e) => e.toEntity()).toList();
  }
}
```

Fonte: Elaborada pela autora, 2023.

Conforme observado, o *repository* acessou a lista constante ***DUMMY_CATEGORY*** no formato Model, foi realizada a conversão para Entidade, retornando uma lista de entidade Categoria. Com a ausência de requisições HTTP,

descartou a necessidade de verificação se ocorreu falha ou sucesso na requisição. Sendo realizado o mesmo processo no *repository* de refeições.

Figura 21 – *MealsRepository*

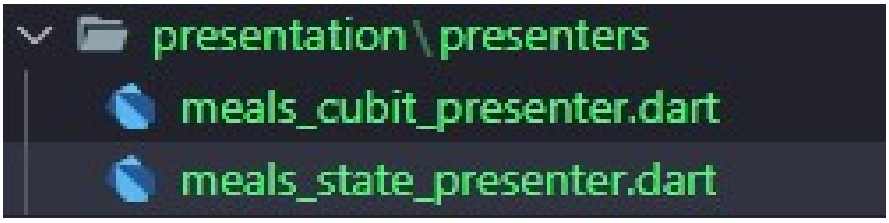
```dart
import 'package:meals/data/dummy_data.dart';
import 'package:meals/domain/entities/meal_entity.dart';
import 'package:meals/domain/usecases/meals.dart';

class MealsRepository extends MealsCaseUses {
  @override
  List<MealEntity> getMeals() {
    return DUMMY_MEALS.map((e) => e.toEntity()).toList();
  }
}
```

Fonte: Elaborada pela autora, 2023.

5.4 Camada presentation

É responsável por lidar com a interação do sistema com o usuário. Essa camada lida com a entrada e saída do sistema, como receber dados do usuário, apresentar informações e gerenciar a interface do usuário. A camada *presentation* é responsável por invocar os casos de uso (figura 22).

Figura 22 – Camada *presentation*



Fonte: Elaborada pela autora, 2023.

Para a implementação de gerenciamento de estado e controle comportamental da interface, utilizou-se o pacote Bloc do Flutter, bem como a implementação Cubit, que tem a mesma finalidade de gerenciar estados. Onde a

diferença marcante das implementações Cubit e Bloc é que primeira não invoca eventos, sendo assim os métodos são públicos e mais fácil de utilizar (figura 23).

Figura 23 – *MealsCubit*

```dart
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:meals/domain/usecases/categories.dart';
import 'package:meals/domain/usecases/meals.dart';
import 'package:meals/presentation/presenters/meals_state_presenter.dart';
import 'package:meals/ui/pages/meals_presenter.dart';

class MealsCubit extends Cubit<MealsCubitState> implements MealsPresenter
{
  final MealsCaseUses getMealsUseCase;
  final CategoriesCaseUses getCategoriesUseCase;

  MealsCubit({
    required this.getMealsUseCase,
    required this.getCategoriesUseCase,
  }) : super(MealsCubitStateInitial());

  Future<void> getMeals() async {
    emit(MealsCubitStateLoading());
    try {
      final result = await getMealsUseCase.getMeals();
      emit(MealsCubitStateLoaded(meals: result));
    } catch (e) {
      emit(MealsCubitStateError());
    }
  }

  Future<void> getCategories() async {
    emit(MealsCubitStateLoading());
    try {
      final result = await getCategoriesUseCase.getCategories();
      emit(CategoriesCubitStateLoaded(categories: result));
    } catch (e) {
      emit(MealsCubitStateError());
    }
  }
}
```

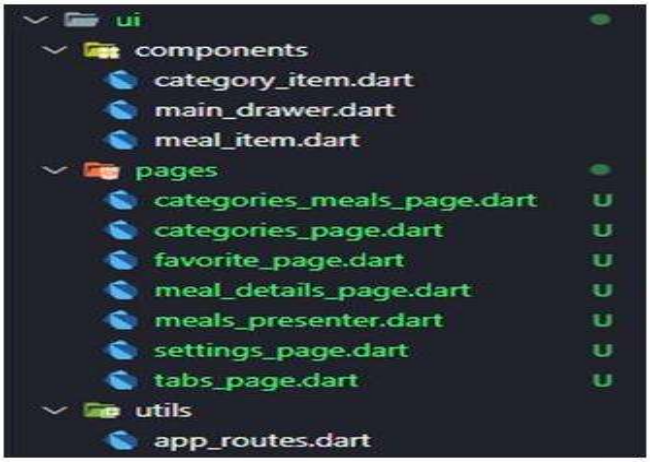
Fonte: Elaborada pela autora, 2023.

Nota-se que, ao utilizar uma interface MealsPresenter para implementação, que está definida na camada de UI, ficou garantida a inversão de dependência.

5.5 Camada de interface de usuário

Trata-se da camada para utilização dos componentes do Flutter. Como dito anteriormente, a camada UI estava separada em duas pastas: *screens* e *components*. Então, para realizar essa correção, moveu-se a pasta *components*, *pages* e *utils* para a camada UI (figura 24).

Figura 24 – Camada UI



Fonte: Elaborada pela autora, 2023.

A camada UI teve como rota inicial TabsPage, onde fez uma navegação entre duas telas: *CategoriesPage* e *FavoritePage.* Essa camada realizou invocações de casos de uso. A figura 25 mostra a invocação para o caso de uso de getCategories.

Figura 25 – *CategoriesPage*

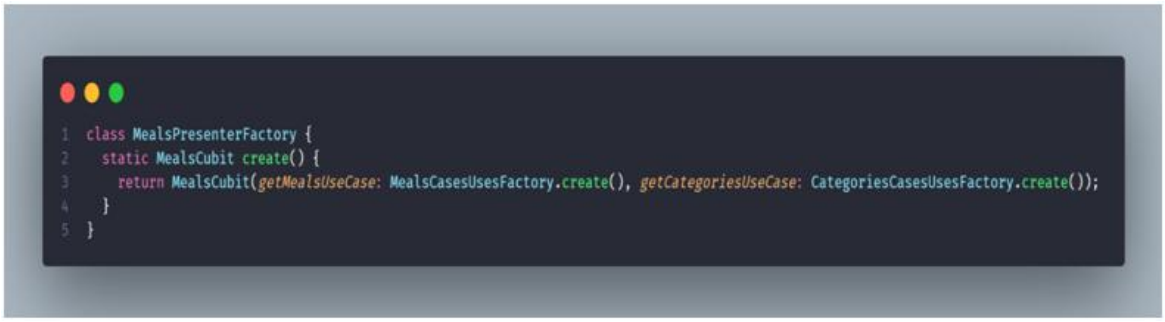
```dart
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:meals/main/factories/meals_presenter_factory.dart';
import 'package:meals/presentation/presenters/meals_cubit_presenter.dart';
import 'package:meals/presentation/presenters/meals_state_presenter.dart';

import '../components/category_item.dart';

class CategoriesPage extends StatefulWidget {
  @override
  State<CategoriesPage> createState() => _CategoriesPageState();
}

class _CategoriesPageState extends State<CategoriesPage> {
  final MealsCubit presenter = MealsPresenterFactory.create();

  @override
  void initState() {
    presenter.getCategories();
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return BlocBuilder(
      bloc: presenter,
      builder: (context, state) {
        if (state is CategoriesCubitStateLoaded) {
          return GridView(
            padding: const EdgeInsets.all(25),
            gridDelegate: SliverGridDelegateWithMaxCrossAxisExtent(
              maxCrossAxisExtent: 200,
              childAspectRatio: 3 / 2,
              crossAxisSpacing: 20,
              mainAxisSpacing: 20,
            ),
            children: state.categories.map((cat) {
              return CategoryItem(cat);
            }).toList(),
          );
        }
        return Container();
      },
    );
  }
}
```

Fonte: Elaborada pela autora, 2023.

Foi realizada a invocação e, caso obtivesse sucesso, seria consumida a listagem de categorias na tela de *CategoriesPage,* como mostrado na figura 25. Também está apresentado na mesma figura o uso do padrão Observer, no qual foi usada a ferramenta Bloc.

Nota-se, ainda, que foi utilizado o padrão de *design Factory* (fábrica), onde as funções foram responsáveis por instanciar as classes e injetar suas dependências. A figura 26 mostra a classe responsável por instanciar o caso de uso de *Meals.*

Figura 26 – *Factory* do caso de uso



Fonte: Elaborada pela autora, 2023.

A figura 27 traz a tela *Settings*, que utilizou o padrão de *design state* (comportamental) para realizar as mudanças de estado. O “setState()” em Flutter é um método utilizado para atualizar a interface do usuário (UI). Ele notifica o framework Flutter que o estado do *widget* mudou e que uma reconstrução da UI é necessária. Nesse caso, é atualizado o valor do *switch*, caso o usuário selecione as opções.

Figura 27 – Tela *settings*

Fonte: Elaborada pela autora, 2023.

Figura 28 – Código da tela *settings*

```dart
class _SettingsPageState extends State<SettingsPage> {
  late Settings settings;

  @override
  void initState() {
    super.initState();
    settings = widget.settings;
  }

  Widget _createSwitch(
    String title,
    String subtitle,
    bool value,
    Function(bool) onChanged,
  ) {
    return SwitchListTile.adaptive(
      value: value,
      subtitle: Text(subtitle),
      title: Text(title),
      onChanged: (value) {
        onChanged(value);
        widget.onSettingsChanged(settings);
      },
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.teal,
        title: Text('Settings'),
        centerTitle: true,
      ),
      drawer: MainDrawer(),
      body: Column(
        children: [
          Container(
            padding: EdgeInsets.all(20),
          ),
          Expanded(
            child: ListView(
              children: [
                _createSwitch(
                  'Without Gluten',
                  'Only displays gluten-free meals!',
                  settings.isGlutenFree,
                  (value) => setState(() => settings.isGlutenFree = value),
                ),
                _createSwitch(
                  'Lactose Free',
                  'Only displays lactose-free meals!',
                  settings.isLactoseFree,
                  (value) => setState(() => settings.isLactoseFree = value),
                ),
                _createSwitch(
                  'Vegana',
                  'Only displays Vegan meals!',
                  settings.isVegan,
                  (value) => setState(() => settings.isVegan = value),
                ),
                _createSwitch(
                  'Vegetarian',
                  'Only displays Vegetarian meals!',
                  settings.isVegetarian,
                  (value) => setState(() => settings.isVegetarian = value),
                ),
              ],
            ),
          )
        ],
      ),
    );
  }
}
```

Fonte: Elaborada pela autora, 2023.

6 CONCLUSÃO

Este trabalho apresentou a importância de uma Arquitetura Limpa em desenvolvimento de software, enfatizando essa relevância na criação de sistemas robustos, escaláveis e de fácil manutenção. Foi utilizada uma aplicação mobile como estudo de caso, demonstrando claramente os benefícios práticos da aplicação dos princípios de Arquitetura Limpa através da refatoração.

Nesse processo de refatoração, ficou evidenciado como a reestruturação da aplicação utilizando os princípios da Arquitetura Limpa proporcionou melhorias significativas na legibilidade do código e na modularidade do sistema. A separação de responsabilidade, implementação de interfaces e a garantia de responsabilidade única foram aspectos significativos que contribuíram para uma Arquitetura Limpa e coesa.

A etapa inicial dessa reestruturação foi analisar como a aplicação mobile já estava implementada, quais princípios não estavam sendo utilizados, quais poderiam ser utilizados e quais estavam sendo utilizados, porém de forma incorreta. Com esses dados mapeados, foi possível inicialmente separar as responsabilidades e definições de camadas corretas, aplicando assim os princípios necessários. Com as camadas definidas, teve início a realização das comunicações entre elas de forma correta. Essas comunicações são que realmente fecham o ciclo de reestruturação do aplicativo. O resultado da refatoração é de modo claro separar responsabilidades de Bancos de Dados, separar regras de negócios da UI, gerando um desacoplamento significativo e assim garantir um melhor desempenho na aplicação.

Reestruturação para Arquitetura Limpa ou qualquer outra arquitetura é um processo longo e custoso em um desenvolvimento de software, principalmente se é um software legado. Um software legado é aquele desenvolvido há algum tempo e que continua em uso, mas que, devido à sua idade, pode ser difícil de manter, adaptar ou entender. Por muitas vezes não é prioridade a realização dessa reestruturação por quem desenvolve esse software, quando se fala de uma empresa fornecedora de sistemas ou de um único desenvolvedor autônomo. Essa falta de prioridade pode ter diversos motivos como: considerar prioridades, novas funcionalidades e falta de mão de obra.

Diante desse contexto, e a fim de complementar este estudo em trabalhos futuros, sugere-se a necessidade de implementação de novas funcionalidades (cadastro de novas receitas por usuários, compartilhamento de receitas, cadastro e login de usuários) e, principalmente, o desenvolvimento de um banco de dados para o sistema realizar o consumo e assim ser possível visualizar as tratativas de retornos e suas devidas comunicações entre camadas.

Por isso, é importante abordar sobre a Arquitetura Limpa e enfatizar a necessidade de um projeto bem arquitetado, antes de começar o desenvolvimento da aplicação, pois, somente assim, será possível proporcionar um sistema fácil de manter, coeso e eficiente.

REFERÊNCIAS

BARBOSA FILHO, Carlos. **Arquitetura hexagonal:** entenda e pratique. LinkedIn, 2023. Disponível em: https://www.linkedin.com/pulse/arquitetura-hexagonal-entenda-e-pratique-carlos-barbosa-filho/. Acesso em: 4 jul. 2024.

BOAS, Leandro Vilas. **Padrões GRASP - padrões de atribuir responsabilidades**: medium. Disponivel em: https://medium.com/@leandrovboas/padr%C3%B5es-grasp-padr%C3%B5es-de-atribuir-responsabilidades-1ae4351eb204. Acesso em: 20 jun. 2024.

BOUKHARY, S.; COLMENARES, E. A clean approach to flutter development through the flutter clean architecture package. In: **International Conference on Computational Science and Computational Intelligence (CSCI).** [S.l.: s.n.], 2019, p. 1115-1120.

BUENO, Carlos Eduardo de Oliveira. **Desenvolvimento de um aplicativo utilizando o framework Flutter e arquitetura limpa**. Disponível em: https://repositorio.pucgoias.edu.br/jspui/bitstream/123456789/1861/1/TCC%202%20-%20CARLOS.pdf . Acesso em: 21 jan. 2024.

CONSUMIDOR Moderno. **Mais de 90% do tempo no celular é gasto em apps.** Disponível em: https://consumidormoderno.com.br/2022/10/13/apps-melhores-aplicativos/#:~:text=Categorias%20de%20apps%20mais%20utilizados%20e%20as%20que%20mais%20crescem&text=Os%20downloads%20de%20aplicativos%20financeiros,175%25%20nos%20%C3%BAltimos%204%20anos. Acesso em: 16 dez. 2023.

DIAS, Ricardo. **Os princípios SOLID**: Medium, 2019. Disponível em: https://medium.com/contexto-delimitado/os-princ%C3%ADpios-solid-34b136f507bb#:~:text=A%20origem%20do%20SOLID&text=A%20hist%C3%B3ria%20de%20sua%20origem,de%20catalogar%20os%20mais%20importantes. Acesso em: 22 out. 2023.

FLUTTER. **Documentação**. Disponível em: https://docs.flutter.dev/. Acesso em: 11 maio 2024.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de projeto – soluções reutilizáveis de software orientado a objetos**. 12. ed. Porto Alegre: Bookman, 1994.

MARTIN, Robert C. **Clean Architecture: a craftsman's guide to software structure and design**. USA: Prentice Hall Press, 2017.

MARTIN, Robert C. **The clean architecture**. Blog do Robert C. Martin, [S.l.], 13 ago. 2012. Disponível em: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Acesso em: 10 maio 2024.

MICHIURA, Fábio. **Clean architecture com MVVM: o que é, vantagens e como utilizar em aplicações Android**: Objective. Disponível em: https://www.objective.com.br/insights/clean-architecture-com-mvvm/#:~:text=Desvantagens%20de%20utilizar%20a%20Clean%20Architecture&text=Pela%20arquitetura%20exigir%20o%20acr%C3%A9scimo,para%20projetos%20de%20baixa%20complexidade. Acesso em: 20 jun. 2024.

P. JUNIOR, Alessandro Rodrigo F.; PASSERINI, Jefferson Antonio Ribeiro. **Arquitetura de software aplicada ao front-end.** Disponível em: https://www.fef.br/upload_arquivos/geral/arq_63fdcc434a344.pdf. Acesso em: 21 jan. 2024.

PERRY, D.E.; WOLF, A. L.; **Foundations for the study of software architecture.** SIGSOFT Software Engineering, out.1996.

REFACTORING Guru. **Design patterns**. Disponível em: https://refactoring.guru/pt-br/design-patterns. Acesso em: 4 jul. 2024.

SCZIP, João Guilherme Berti. **Implementando clean architecture no reactjs**: Medium. Disponível em: https://joaogbsczip.medium.com/implementando-clean-architecture-no-reactjs-af17fb70ca6. Acesso em: 02 maio 2024.

SHAW, M.; GARLAN, D. **Software architecture**. Perspectives on an Emerging Discipline, Prentice Hall, 1996.

SOMMERVILLE, Ian. **Engenharia de software**. São Paulo: Pearson, 2016.

VERGILIO, Silvia Regina. **Arquitetura de software**. Disponível em: https://www.inf.ufpr.br/andrey/ci163/IntroduzArquiteturaAl.pdf. Acesso em: 07 out. 2023.