UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Vinícius Alba

Aplicação de Padrões de Projeto e Conceitos Arquiteturais em Aplicações Flutter

Uberlândia, Brasil

2024

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Vinícius Alba

Aplicação de Padrões de Projeto e Conceitos Arquiteturais em Aplicações Flutter

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Marcelo de Almeida Maia

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2024

Gustavo Vinícius Alba

Aplicação de Padrões de Projeto e Conceitos Arquiteturais em Aplicações Flutter

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 01 de abrir de 2024:

Marcelo de Almeida Maia
Orientador

Rodrigo Sanches Miani

Paulo Rodolfo da Silva Leite Coelho

Uberlândia, Brasil
2024

Agradecimentos

Agradeço primeiramente aos meus pais, por terem batalhado arduamente durante a minha infância e adolescência para que eu pudesse ter acesso a uma boa educação e, consequentemente, conseguisse ingressar em uma universidade federal.

Segundamente, agradeço às pessoas que pude ter o prazer de conhecer na faculdade, em especial aos membros da turma 63 de Ciência da Computação, que estiveram ao meu lado desde o início do curso em 2019, por terem sido a melhor turma de faculdade que eu poderia desejar e terem propiciado diversos momento incríveis em minha vida. Além disso, agradeço especialmente a Paulo, Geovanni e Pedro, com os quais tive a felicidade de dividir moradia após o retorno do presencial em 2022.

Agradeço também a todas as pessoas com quem tive o prazer de trabalhar junto desde que cheguei em Uberlândia, com as quais tive muitas trocas de experiências, aprendizados e momento memoráveis, e que sei que são pessoas que poderei levar para o resto da minha vida. Em especial, agradeço aos irmãos Aristeu e Harlisson, por terem me dado a primeira oportunidade de estágio quando estava ao fim do segundo período, o que fez com que eu pudesse me desenvolver muito como profissional e me encontrar na área de Engenharia de Software.

Por fim, agradeço a todos os professores da UFU com os quais pude aprender muito sobre a área de computação e que me permitiram, a cada semestre, ver o quão fascinante e abrangente a Computação é. Especialmente, agradeço ao meu orientador por me guiar durante a realização desse trabalho e proporcionar diversas discussões técnicas que agregaram muito para o meu conhecimento a cerca do assunto dessa monografia.

“Don’t you dare go hollow” - Laurentius, de Dark Souls

Resumo

No desenvolvimento dos primeiros softwares havia apenas uma preocupação primordial: que ele funcionasse. Porém, com o passar do tempo e os avanços e estudos na área de Engenharia de Software, tanto pelo mundo acadêmico quanto pelo mercado, passaram a ser desenvolvidas várias ferramentas, técnicas e modos de se fazer um bom software. O objetivo dessas, é o de facilitar a leitura do código e sua manutenção, de forma que o software consiga evoluir e atender a novas demandas exigidas. Esse trabalho tem como objetivo estudar a aplicação de algumas dessas técnicas em aplicações construídas com o *framework* Flutter, de forma a mostrar como tanto o projeto de código quanto a arquitetura podem trazer benefícios para o software sendo contruído. Assim, este trabalho apresenta um estudo feito dos princípios que regem a qualidade de um software e dos principais conceitos relacionados a arquitetura de software. Além disso, propõe-se a implementação de padrões de projeto em aplicações Flutter, mostrando o passo a passo e justificando as melhorias causadas no código. Ademais, apresenta-se a criação de duas propostas de arquitetura para aplicações Flutter baseadas nos estudos feitos em cima de *Domain Driven Design*, *Clean Architecture* e *Hexagonal Architecture*. Com isso, esse trabalho demonstra a aplicação de alguns padrões de projeto e traz uma visão geral sobre a aplicação de bons princípios de arquitetura de software em aplicações Flutter, mostrando o impacto que decisões arquiteturais podem trazer ao projeto.

Palavras-chave: Flutter, Dart, DDD, *Domain Driven Design*, *Clean Architecture*, Arquitetura Limpa, Arquitetura Hexagonal, *Design Patterns*, Padrões de Projeto.

Lista de ilustrações

Lista de abreviaturas e siglas

DDD	Domain Driven Design
YAGNI	You Aren't Gonna Need It
DRY	Don't Repeat Yourself
SRP	Single Responsibility Principle
OCP	Open-Closed Principle
LSP	Liskov Substitution Principle
ISP	Interface Segregation Principle
DIP	Dependency Inversion Principle
GUI	Graphic User Interface
HTTP	Hypertext Transfer Protocol
UI	User Interface
SO	Sistema Operacional
DTO	Data Transfer Object
API	Application Program Interface

Sumário

1 Introdução

No final da década de 1960, foram criados os primeiros computadores modernos, que objetivavam resolver problemas científicos e executar alguns poucos programas, sem que se houvesse uma preocupação central em torno do software em si. Com isso, em outubro de 1968, ocorreu uma conferência patrocinada pela Organização do Tratado do Atlântico Norte (OTAN), que produziu um relatório que afirmava a necessidade do uso de princípios práticos e teóricos para a construção de software, assim como em outros ramos da Engenharia. Foi nesse momento que cunhou-se o termo Engenharia de Software. Desde então, os avanços na área de desenvolvimento de software são notáveis, com a criação de bibliotecas e frameworks que permitem o reuso de código e abstraem os detalhes de baixo nível para interagir com computadores, além da criação de diversos padrões de projetos que podem ser seguidos para resolver problemas amplamente conhecidos (VALENTE, 2020a).

Por conta desses avanços que fizeram com que o desenvolvimento de software passase a operar em um nível mais abstrato, a barreira de entrada passou a ser menor, o que fez com que, segundo Martin (2017a) não fosse necessário ter-se muito conhecimento e habilidades técnicas de desenvolvimento de software para que se tenha um programa funcionando. Isso faz com que desenvolvedores em início de carreira ao redor do mundo trabalhem duramente para cumprir requisitos necessários para fazer suas aplicações funcionarem, de forma que o código produzido não é necessariamente o melhor, mas é um que funciona.

Agora, construir um software da forma certa é algo completamente diferente. Para isso, é necessário pensamento crítico e experiência que muitos desenvolvedores, ainda mais em início de carreira, ainda não tem. Quando o software é construído da forma certa, em geral, não é necessário uma grande quantidade de desenvolvedores para mantê-lo e continuar seu desenvolvimento. Com isso, alterações de código são feitas com menor esforço e de forma mais rápida, além de que bugs são reduzidos (MARTIN, 2017a).

Para a criação de projetos de software melhores que atinjam as qualidades descritas, têm-se diversos trabalhos e autores amplamente reconhecidos na área de Engenharia de Software e Arquitetura de Software, como Martin (2017a), Evans (2004), Cockburn (2005) e Gamma et al. (1994).

Apesar da existência de diversas arquiteturas de software desenvolvidas nas últimas duas décadas, segundo Lemos (2022a), muitas delas são muito semelhantes em sua essência, apesar de variarem nos detalhes. Todas buscam realizar a separação de interesses por meio da divisão do software em camadas. Assim, o mais importante no estudo de

Arquitetura de Software não é qual arquitetura que um projeto de software deve utilizar, mas sim os conceitos fundamentais que essas arquiteturas empregam para atingir o objetivo de minimizar os recursos humanos requeridos para construir e manter um sistema de software (MARTIN, 2017a).

Assim, o objetivo desse trabalho é explorar o uso de padrões de projeto e conceitos de arquitetura de software em aplicações Flutter. Esse trabalho visa descrever alguns padrões e princípios de desenvolvimento de software, assim como descrever alguns conceitos de arquitetura das principais obras existentes no ramo de Arquitetura e Engenharia de Software. Além disso, visto a não existência de muitos trabalhos sobre padrões de projeto e arquitetura no framework Flutter, esse trabalho busca demonstrar como a aplicação de padrões de projeto em aplicações Flutter pode melhorar a qualidade de código se comparado ao não uso deles em determinadas situações, assim como pode-se utilizar dos conceitos de arquitetura para construir uma aplicação Flutter melhor.

Dessa forma, em seu segundo capítulo, esse trabalho faz uma revisão da literatura nos temas de arquitetura de software e boas práticas e padrões de desenvolvimento de software. Em seu terceiro capítulo, esse trabalho mostra como alguns padrões de projeto podem ser aplicados em aplicações Flutter por meio de pequenas provas de conceito, nas quais é mostrada uma forma mais direta e simples de se resolver algum problema, além de mostrar todo o passo a passo para a aplicação do padrão de projeto para melhorar o trecho de código previamente desenvolvido. Ainda no que tange o desenvolvimento do trabalho, é mostrada a construção de uma arquitetura com base nos estudos feitos nas obras citadas na revisão bibliográfica, de forma que é criada uma aplicação simples de exemplo para aplicar de fato a arquitetura elaborada.

2 Revisão Bibliográfica

Nessa seção serão abordados os principais tópicos teóricos utilizados durante o desenvolvimento desse trabalhos, abrangendo desde definições mais simples de o que é arquitetura de software até a descrição de padrões de projeto e arquitetura construídas por diversos autores da área de Engenharia de Software.

2.1 Arquitetura de Software

Ao longo da literatura de engenharia de software, o termo "Arquitetura" é usado por diversos autores muitas vezes com significados diferentes. Segundo Valente (2020b), uma das definições mais comuns é que a arquitetura se importa com os detalhes de alto nível de um projeto de software, o que faz com que o foco deixe de ser na organização de classes e funções individuais, consideradas de detalhes de baixo nível, e passe a ser em componentes de maior tamanho, que são um agregado dessas classes e funções individuais.

Já outros autores discordam dessa definição. Martin (2017b) propõe que os pequenos detalhes de baixo nível suportam todas as decisões de alto nível, de forma que ambos são parte de um mesmo todo e definem a forma do sistema de um modo que não se pode ter um sem haver o outro. Ainda segundo o autor, algo muito importante de se considerar é o objetivo dessas decisões, sendo elas as de alto e baixo nível. O objetivo da arquitetura de software deve ser o de minimizar os recursos humanos requeridos para construir e manter um sistema de software, de forma que, em um sistema bem projetado e bem arquitetado, o esforço não deve aumentar com a adição de novas funcionalidades ao sistema.

Assim, sugere-se que a arquitetura de software tem um papel central no desenvolvimento de uma aplicação. Isso se deve ao fato de que uma implementação ruim de uma boa abstração causa pouco dano à base de código, enquanto que uma abstração inerentemente ruim, a falta de camadas ou a má organização dessas camadas, faz com que o software como um todo passe a perder o valor (KIEHL, 2021). Assim, por mais que o software funcione plenamente de início, se for impossível de mudá-lo devido a uma arquitetura ruim, esse software estará fadado ao fracasso. Já um software que não funciona bem, mas pode ser facilmente alterado, consegue se adaptar bem e ser corrigido com o tempo de forma a agregar valor a organização que o possui. Esse fato se encaixa muito bem no contexto atual de desenvolvimento de software e na grande adoção de práticas ágeis, em que os requisitos de software mudam com alta frequência e o software construído deve ser capaz de adequar a essas mudanças para que continue sendo útil para seus usuários.

Com isso, segundo Lemos (2022b), define-se que a Arquitetura de Software está relacionada à forma do sistema construído e que essa forma se caracteriza como a divisão e arranjo dos componentes de um sistema e o como eles se comunicam. O autor ainda estabelece que uma estratégia central no desenvolvimento da arquitetura de software é a de deixar opções em aberto pelo maior tempo possível, de forma a postergar ao máximo decisões relacionadas a detalhes de baixo nível, visto que esses detalhes tendem a mudar ao longo do tempo sem influenciar nas regras de negócio de alto nível. Assim, deve ser possível alterar detalhes de tecnologias específicas, como interface de usuário ou banco de dados, sem que isso modifique o núcleo da funcionalidade do sistema. Na realidade, pode ser possível que alguns detalhes de baixo nível nunca sejam realmente trocados em um projeto, porém ao separar esses detalhes das regras de negócio essenciais do software também tem-se o benefício de um código totalmente desacoplado, o que permitirá que esses trechos de código evoluam independentemente, além de permitir que o desenvolvedor trabalhe em uma parte do sistema sem se preocupar com as outras.

2.2 Princípios de Desenvolvimento de Software

Com a evolução da Engenharia de Software, diversos autores, tanto do meio acadêmico quanto do mercado, criaram diversos princípios de desenvolvimento, que visam resolver problemas comuns que podem ocorrem em diversos projetos. Um princípio muito conhecido é o representado pelo acrônimo YAGNI, “You Aren’t Gonna Need It”, que segundo Fowler (2015), descreve que não se deve desenvolver recursos que um software precisará no futuro, já que “você não vai precisar disso”. Esse princípio se alinha muito bem com a ideia de uma arquitetura adequada definida previamente, já que não se deve preocupar com detalhes de baixo nível até que realmente seja necessário, focando sempre nos detalhes de alto nível relacionados com o problema que o software busca resolver.

Outro princípio estabelecido é o DRY, cunhado por Hunt e Thomas (2000), que afirma que todo pedaço de conhecimento deve ter uma representação única, não ambígua e autoritativa dentro de um sistema, sendo DRY um acrônimo para “Don’t Repeat Yourself”. Assim, têm-se que esse princípio objetiva a não duplicação de código e busca sempre a modularização de código para que não se tenha problemas de informações serem atualizadas em parte do código, mas não em outras partes. Os próprios autores do princípio afirmam que não é uma questão de se o desenvolvedor irá lembrar de atualizar o código espalhado, mas sim uma questão de quando ele irá esquecer.

Muitos princípios são criados individualmente e, posteriormente, são agrupados ou expandidos por outros autores de forma a torná-los mais conhecidos. Isso foi o que aconteceu com os Princípios SOLID, um conjunto de cinco princípios criados ao longo do tempo e colocados sobre um acrônimo por Martin (2017b). Esses princípios declaram

como organizar funções e estruturas de dados em classes e como essas classes devem estar interconectadas. O objetivo do uso desses princípios é o de tolerar mudanças, tornar o código mais fácil de se entender e serem a base de componentes que podem ser usados em muitos sistemas de software.

Dentre os princípios SOLID, pode-se destacar dois que se relacionam fortemente com as definições de uma boa arquitetura apresentados anteriormente, são esse o *Open-Closed Principle(OCP)*, da letra O do acrônimo, e o *Dependency Inversion Principle(DIP)*, da letra D. O OCP declara que um sistema de software deve ser projetado de forma que o comportamento do sistema possa ser estendido pela adição de código novo, ao invés da alteração de código existente. Isso colabora com o objetivo da arquitetura, já que será possível adicionar novas funcionalidades de forma a não se ter um esforço muito grande. O DIP está fortemente relacionado com os objetivos de uma boa arquitetura, visto que esse princípio declara que o código que implementa as políticas de alto nível não deve depender de código que implementa os detalhes de baixo nível, e sim o contrário, os detalhes que devem depender das políticas. Com esse conceito em mente, tem-se também que a regra de negócio da aplicação nunca deve depender de detalhes técnicos, como banco de dados, interface de usuário, serviços de terceiros, dentre outros.

Segundo Gamma et al. (1994), projetar software orientado a objetos é difícil, e projetar código orientado a objetos que seja reusável é mais difícil ainda. O código criado deve ser específico para resolver o problema em questão, mas ao mesmo tempo deve ser genérico o suficiente para resolver problemas e requerimento futuros. Assim, projetistas experientes tomaram como ação para resolver esse dilema em reusar soluções que foram bem sucedidas no passado e, com o tempo, surgiram os chamados padrões de projeto. No livro "Design Patterns: Elements of Reusable Object-Oriented Software", Gamma et al. (1994), descrevem alguns grupos de padrões de projeto para solucionar problemas comuns e melhorar o reuso de código em um projeto de software. Os padrões criacionais se preocupam no processo de criar um novo objeto para aumentar a flexibilidade e reuso. Os padrões estruturais explicam como agregar objetos e classes em estruturas maiores, ainda de forma a manter essas estruturas flexíveis e eficientes. Já o padrões comportamentais lidam com a comunicação e atribuição de responsabilidades entre objetos, fato esse que é de suma importância para a existência de uma boa arquitetura.

Outros dois conceitos de extrema importância no estudo de arquitetura de software são os conceitos de acoplamento e coesão em um projeto de software. Segundo Fowler (2001) se ao alterar um módulo de um programa requer alterar outro módulo, seja por questões de código duplicado ou pelo fato de um módulo acessar diretamente código de outro módulo, então o acoplamento existe entre esses módulos. Para a construção de um bom software, deseja-se ter um baixo acoplamento entre os componentes e módulos, de forma a ter uma independência maior entre eles e de forma que as políticas de alto nível

não dependam dos detalhes de baixo nível, o que está diretamente relacionado ao DIP. Já a coesão é, segundo Martin (2017b), a força que une um código a um determinado ator, o que se relaciona de forma próxima com o *Single Responsibility Principle(SPR)*, a letra S do acrônimo SOLID, que declara que um módulo deve ser responsável a um, e apenas um, ator, sendo "ator" aqui definido como um grupo de pessoas (normalmente usuários ou *stakeholders* de uma organização) que requerem mudanças em um software. Assim, a coesão mede o grau que um trecho de código (uma classe e/ou função) funciona para cumprir um único e bem definido propósito. Com isso, entende-se que um bom software deve sempre buscar um alto nível de coesão dentro de seus componentes e módulos em conjunto com um baixo acoplamento, de forma a atingir níveis satisfatórios de manutenibilidade, escalabilidade e confiabilidade.

2.2.1 Padrão de Projeto Observer

O padrão de projeto *Observer*, segundo Shvetz (2013), define uma relação de um para muitos entre objetos de forma que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizam automaticamente. Nesse padrão, é definido um objeto que é o mantenedor do modelo de dados. Outros objetos que desejem receber informação desse modelo de dados conforme ele mude, devem se inscrever nesse objeto, que irá transmitir para todos os ouvintes a nova informação assim que houver uma mudança no modelo de dados. Dessa forma, o padrão *Observer* define uma interface muito desacoplada que permite que múltiplos ouvintes sejam configurados em tempo de execução, com a possibilidade deles se inscreverem e desinscreverem a qualquer momento, como pode ser percebido na estrutura da Figura 1. Esse padrão consegue ser muito útil principalmente para casos de interface gráfica, visto que componentes visuais podem se inscrever em classes que controlam dados e responder a mudanças aos estados em tempo real, o que torna a experiência de quem usa a aplicação gráfica melhor, por conta da reatividade à ações do usuário que esse modelo permite.

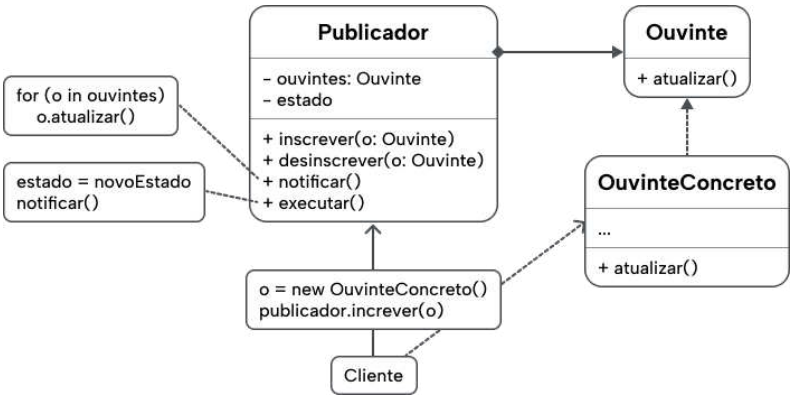

Figura 1 – Diagrama da Estrutura do Padrão Observer. Fonte: Do Autor.. Fonte: Do Autor.

2.2.2 Padrão de Projeto Builder

Segundo Gamma et al. (1994), o padrão de projeto *Builder* permite a separação da construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção pode criar diferentes representações. O padrão organiza a construção do objeto em um conjunto de passos, de modo que, para criar um objeto, é realizada a chamada apenas dos passos necessários(não necessariamente todos) no objeto *builder* e, ao final, é chamado um método que irá realizar de fato a construção do objeto desejado.

Para casos em que um mesmo processo de construção pode ter múltiplas implementações, é possível criar diferentes classes *builder* que implementam o mesmo conjunto de processos de construção. Para realizar o gerenciamento de diversas implementações de um mesmo processo de construção, o padrão de projeto especifica a existência de uma classe Diretor, que tem a responsabilidade de manejar quais os passos de construção a serem chamados para cada implementação, como pode ser visto na Figura 2.

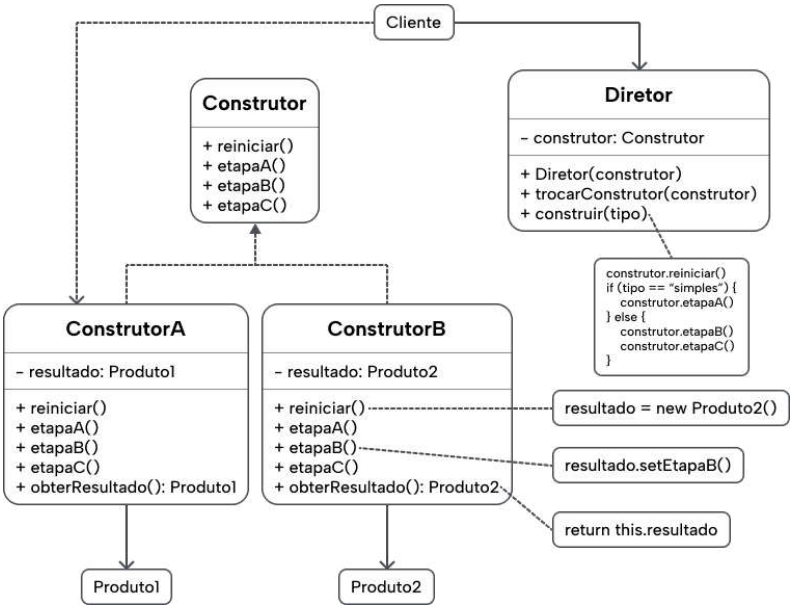

Figura 2 – Diagrama da Estrutura do Padrão Builder. Fonte: Do Autor.

2.2.3 Padrão de Projeto Factory

De acordo com Gamma et al. (1994), o *Factory* é um padrão criacional que provê uma interface para criação de objetos, mas permite que a subclasse decida de qual classe a ser instanciada, de forma que seu uso é útil quando é necessário que uma classe delegue a responsabilidade de criação de um objeto para uma de várias outras subclasses auxiliares com a necessidade de localizar o conhecimento de qual subclasse auxiliar que foi a delegada. A estrutura do padrão pode ser vista na Figura 3, em que Produto é a interface que define os métodos que os Produtos Concretos devem implementar e a classe

Criador é a classe abstrata que declara o método *Factory*, com as classes Criador Concreto implementando o método *Factory* e retornando o Produto Concreto.

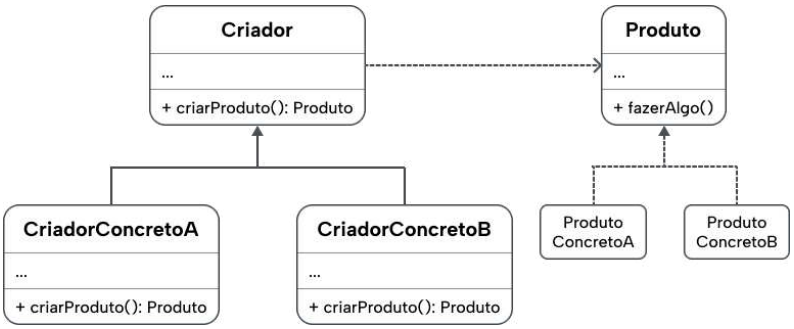

Figura 3 – Diagrama da Estrutura do Padrão Factory. Fonte: Do Autor.

2.2.4 Padrão de Projeto Repository

O padrão *Repository* foi introduzido por Evans (2004) e provê uma interface que abstrai o acesso a dados de uma aplicação, de forma que é possível adicionar, remover, atualizar e buscar dados de uma dada coleção sem que se tenha a preocupação com especificidades de onde os dados estão armazenados, já que esses detalhes de como manipular os dados na base de dados estarão presentes apenas na implementação dessa interface. Assim, esse padrão permite um menor acoplamento por prover uma interface de como obter ou manipular os objetos do domínio. Toda a lógica de como isso será persistido, seja em um banco de dados ou em um serviço externo, ficará como responsabilidade da implementação desse *Repository*, de forma com que isso será transparente para a camada que é responsável por executar a regra de negócio da aplicação, como visto no diagrama de Figura 4.

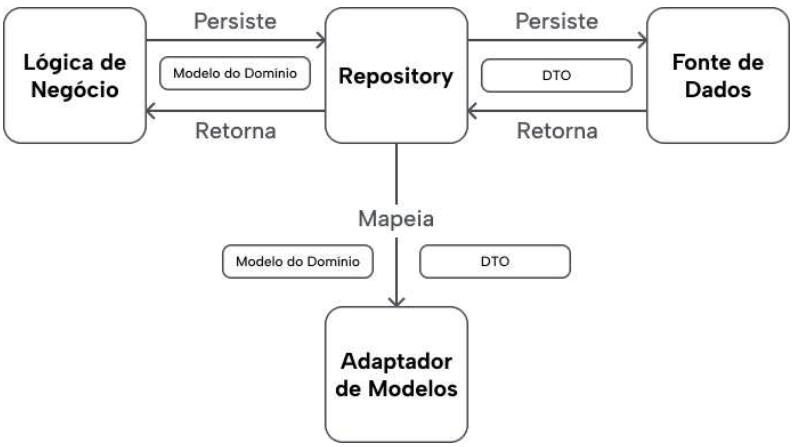

Figura 4 – Diagrama da Estrutura do Padrão Repository. Fonte: Do Autor.

2.3 Domain Driven Design

Segundo Fowler (2020), a ideia da necessidade de sistemas de software serem baseados em um modelo de domínio bem desenvolvido é algo presente na indústria há muito tempo, sendo um ponto chave dos trabalho das comunidades de bancos de dados e de linguagens orientadas a objetos entre os anos 1980 e 1990. Um dos trabalhos com maior destaque para resolver essa necessidade da indústria foi o *Domain Driven Design*, introduzido por Evans (2004) em seu livro "Domain-driven design: atacando as complexidades no coração do software", lançado em 2003.

O *Domain Driven Design(DDD)* se caracteriza como uma abordagem de desenvolvimento que focaliza na criação de um modelo de domínio que tenha um entendimento rico dos processos e regras de um domínio. Segundo Vernon (2013), um modelo de domínio é um modelo de software do domínio de negócio específico do qual se está trabalhando, de forma a normalmente ser representado por meio de um modelo de objeto que possui dados e comportamentos com significado literal e preciso do negócio em questão. Assim, para praticar o DDD é essencial criar um modelo de domínio único e cuidadosamente trabalhado nas necessidades do negócio, além de ter-se sempre a noção de que os modelos de domínio devem ser pequenos e muito focados, de forma a nunca buscar representar todo o domínio de negócio em um único e grande modelo de domínio.

Um aspecto de grande importância introduzido pelo DDD é a existência de uma linguagem universal no processo de desenvolvimento de software. Evans (2004) coloca em sua obra como ponto central a existência dos especialistas de domínio na criação de um projeto de software. Essas pessoas são caracterizadas por terem o conhecimento a cerca do que o projeto precisa e o conhecimento do segmento de negócio que a aplicação de software está envolvida. Assim, a linguagem universal proposta por Evans (2004) se coloca como a linguagem comum que deve ser compartilhada por toda equipe, tanto os desenvolvedores quanto os especialistas do domínio, de forma a diminuir o atrito na comunicação e a realização de traduções de termos técnicos entre as linguagens de desenvolvimento e de negócio.

De forma a materializar a linguagem universal e o modelo de domínio, o DDD introduz os chamados contextos delimitados e os mapas de contexto. Segundo Masotti (2022), como cada área de negócio possui conjuntos de termos diferentes no dia a dia de trabalho de acordo com o contexto inserido, a construção de um modelo de domínio unificado se torna uma tarefa complexa. Assim, os contextos delimitados colaboram no processo de desenvolvimento ao estabelecer limites e dividir os grandes modelos em contextos menores, com a criação de inter-relacionamento entre eles. Em conjunto com os contextos delimitados, têm-se os mapas de contextos, que atua como uma visão geral da modelagem desenvolvida, de forma a facilitar o entendimento dos contextos da aplicação. Esse documento deve abranger a relação entre os contextos delimitados de uma organi-

zação, de modo a colaborar com o entendimento da equipe sobre o domínio de negócio, as fronteiras entre os contextos e como essas fronteiras podem ser integradas.

Dentro do DDD, proposto por Evans (2004), a ideia de linguagem universal, modelo de domínio, contextos delimitados e mapas de contexto se unem sobre o design estratégico, o qual atua como primeira etapa do DDD e visa estabelecer limites e responsabilidades claras na construção da topologia de alto nível de um software, de forma que é muito importante o envolvimento de todos da equipe nesse processo de design estratégico, tanto dos desenvolvedores quanto dos especialistas de negócio, visto que esse processo é crucial para a criação do software. A próxima etapa no DDD é a chamada de design tático, que foca nos detalhes de implementação e tem como foco refinar o design estratégico por meio de padrões de abstração de médio e baixo nível, de forma a auxiliar na construção do código final.

O design tático colabora no momento de codificação do modelo de domínio, a partir das definições colocadas pelo design estratégico anteriormente, de forma que têm-se alguns componentes principais dentro dele: os Modelos de Domínio, os Serviços de Domínio e os Serviços de Aplicação. O primeiro busca representar o problema sendo resolvido, no qual são criados padrões para a representação dos objetos de domínio, como entidades, agregados e objetos de valor. Já os Serviços de Domínio se caracterizam como as estruturas que auxiliarão os Modelos de Domínio em situações mais complexas nas quais o modelos de domínio não conseguem realizar as suas ações, como no caso de acesso a dados em serviços externos, de forma que os serviços de domínio são responsáveis por orquestrar a relação entre os objetos de domínio. Enquanto isso, os Serviços de Aplicação são os responsáveis por orquestrar um fluxo de uso da aplicação, de forma que podem utilizar de vários serviços de domínio e modelos de domínio para realizar isso. O papel deles está mais ligado a como a aplicação de software deve funcionar do que como é a regra de negócio da aplicação, visto que esta já está contida nos serviços e modelos de domínio.

2.4 Hexagonal Architecture

A arquitetura hexagonal é uma proposta de arquitetura de software introduzida por Cockburn (2005), em uma tentativa de evitar que desenvolvedores caiam em problemas estruturais já conhecidos nos projetos de softwares orientados a objetos. O principal objetivo do autor é o de permitir que uma aplicação possa ser igualmente utilizada por usuários, programas, testes e *scripts* automatizados e que a aplicação possa ser desenvolvida e testada de forma isolada de seus dispositivos de tempo de execução e de bancos de dados.

A ideia central da arquitetura hexagonal, também conhecida com arquitetura *port and adapters*, é a de separar o código interno, que conterá as regras de negócio da aplicação,

do código externo, que fará a comunicação com vias externas, como bancos de dados e interfaces de usuário. Essa separação se dá por meio da criação de portas na camada de aplicação para que se comunique com agentes externos, como pode ser visto na Figura 5. A palavra "porta" nesse contexto é similar à ideia de portas de um sistema operacional, em que qualquer dispositivo que adere ao protocolo de uma porta pode se *plugar* a ela. Esses dispositivos que aderem às portas são chamados de adaptadores. Por exemplo, uma aplicação que precisa se comunicar com um meio externo irá criar uma porta para isso, que pode ser implementada por um adaptador que se comunica com um banco de dados SQL para trazer os resultados. Se esse banco de dados for trocado por um armazenamento em arquivo ou um banco de dados NoSQL, criaria-se ter um novo adaptador que iria se *plugar* com a aplicação, mantendo o contrato previamente estabelecido pela porta.

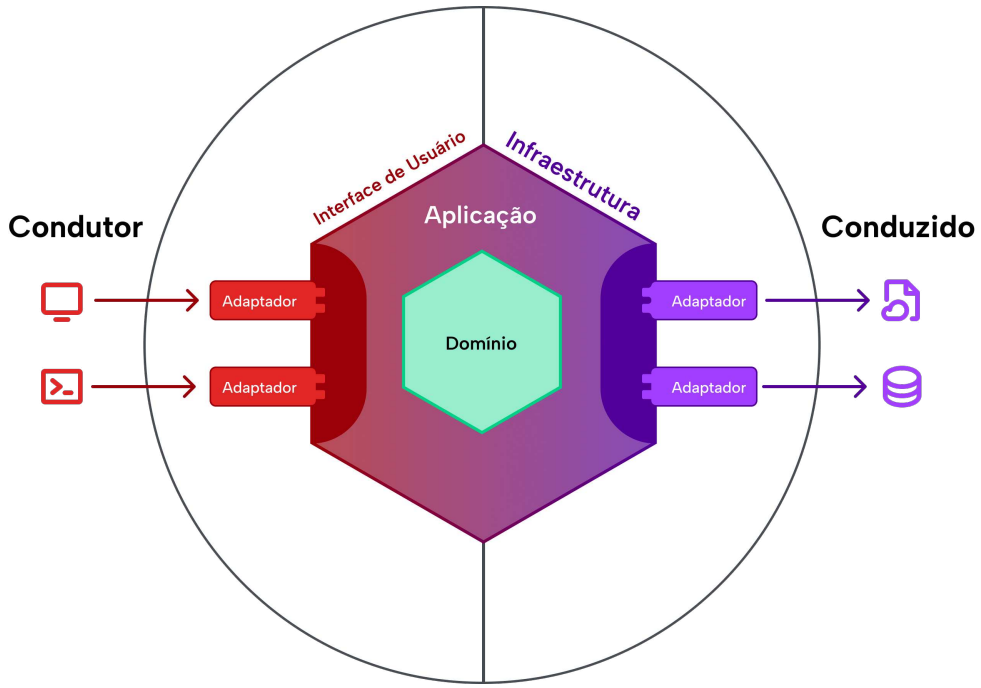

Figura 5 – Diagrama representativo da Arquitetura Hexagonal. Fonte: Do Autor.

Ainda nessa ideia da criação de adaptadores, Cockburn (2005) faz a distinção entre adaptadores primário e secundários. Um adaptador primário se caracteriza por conduzir a aplicação, atuando como entrada de dados nela, em muitos casos sendo o meio pelo qual o usuário acessa a aplicação, como por uma interface GUI ou por chamadas HTTP. Já o adaptador secundário, colocado como lado conduzido da arquitetura, age ao prover para aplicação dados que ela pode precisar para responder a uma interação feita por um adaptador primário, de forma a atuar na saída de dados, normalmente acessando um banco de dados ou interagindo com algum serviço externo para solicitar algum dado. É importante destacar ainda que a arquitetura hexagonal foca em isolar as regras de negócio da aplicação do mundo externo, mas não estabelece nada sobre como devem ser conduzidas as regras de negócio na camada de aplicação. Assim, pode-se utilizar de outras técnicas

para gerenciamento do domínio da aplicação, como por exemplo o *Domain Driven Design*.

2.5 Arquitetura Limpa

Após anos na indústria de desenvolvimento de software, Martin (2012) percebeu o surgimento de muitas arquiteturas de sistemas sendo desenvolvidas, a maioria delas variando pouco nos detalhes de implementação, mas com diversas semelhanças em suas essências, com foco na independência de agentes externos, testabilidade, manutenibilidade, alta coesão e baixo acoplamento.

Assim, Martin (2012) definiu um modelo geral de como se ter uma arquitetura mais limpa, que não fosse "suja" de detalhes externos que não interessam ao cerne da aplicação e à regra de negócio em questão. Esse modelo é representado por algumas camadas, que podem ser vistas na Figura 6, de forma que existe uma regra de dependência, na qual as dependências de código devem apontar sempre para dentro. Dessa forma, qualquer função, classe, variável ou entidade de software que reside em uma camada interna não deve, em hipótese alguma, saber da existência de uma entidade pertencente a uma camada mais externa.

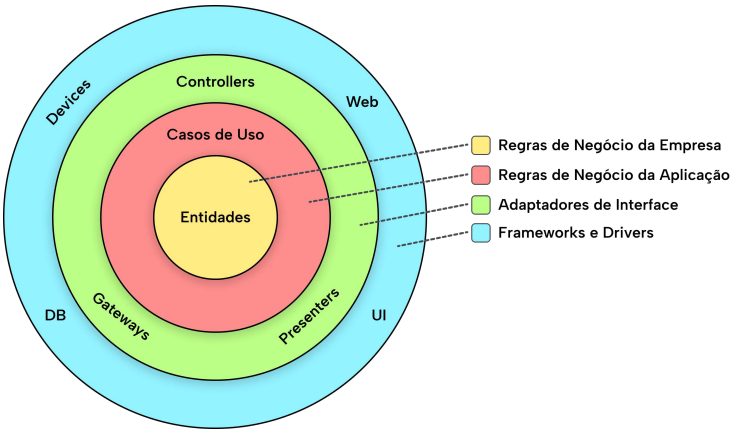

Figura 6 – Diagrama representativo da Arquitetura Limpa. Fonte: Do Autor.

Na camada mais central da arquitetura limpa se encontram as entidades, que são responsáveis por representar as regras de negócio da organização, que podem ser objetos com métodos ou apenas um conjunto de estruturas de dados e funções. As entidades representam as regras gerais de mais alto nível e são as menos passíveis de mudar em caso de algo externo alterar.

Na próxima camada, tem-se os casos de uso, que representam as regras de negócio da aplicação de software e orquestram o fluxo de manipulação das entidades, de forma a fazer as entidades executarem suas regras de negócio da organização de modo a atingir os objetivos dos casos de uso. Qualquer mudança externa também não deve alterar essas

camada, porém mudanças nas entidades ou na forma que a aplicação opera irão afetar o software presente nessa camada.

A camada seguinte é a de adaptadores de interface. Essa camada é responsável por ter um conjunto de adaptadores que irão converter os dados da forma mais conveniente em relação aos casos de uso e entidades para a forma mais conveniente para agentes externos, como bancos de dados e a web, e vice-versa. Os modelos nessa camada são apenas estruturas de dados representacionais que irão ser passadas pelas estruturas do software, como funções e classes.

Na camada mais externa proposta por Martin (2012), tem-se a camada de *Frameworks* e *Drivers*, que é composta pela implementação de *frameworks* e ferramentas como bancos de dados, web, etc. Nessa camada se encontram todos os detalhes da aplicação e, no máximo, se encontra código que irá servir como ponte para se comunicar com a camada mais interna.

Assim, essa divisão em camadas se mostra útil ao isolar o domínio da aplicação de tudo que é externo a ela, como a interface gráfica e bancos de dados, presentes apenas na camada mais externa de *Frameworks* e *Drivers*. Dessa forma, como no caso da Arquitetura Hexagonal, é possível de utilizar técnicas vindas do *Domain Driven Design* para estruturar o núcleo da aplicação.

2.6 Flutter e Dart

Para a implementação dos padrões de projeto e dos estudos em arquitetura desse trabalho, será usada a linguagem de programação Dart, em conjunto com o *framework* Flutter para construção da interface gráfica e execução da aplicação em si. A linguagem Dart é uma linguagem de programação projetada por Lars Bak e Kasper Lund e desenvolvida dentro da Google, apresentada inicialmente em 2011 na conferência GOTO e com lançamento da sua versão 1.0 em novembro de 2013. A linguagem se caracteriza por ser orientada a objetos, baseada em classes, com sintaxe inspirada na linguagem C e por possuir *garbage collector*, de forma a ser muito semelhante à Java, além de possuir uma tipagem forte e dinâmica, com inferência de tipos. Ademais, a linguagem é projetada de forma a ser apropriada para desenvolvimento de aplicações cliente, priorizando tanto desenvolvimento (pela funcionalidade de *hot reload*, que permite com que um projeto não tenha que ser compilado novamente após alguma alteração de código) quanto experiências em produção de alta qualidade em diversas plataformas (*mobile*, *web* e *desktop*).

A linguagem Dart também se caracteriza pela funcionalidade de *null safety*, disponibilizada na sua versão 2.12 e presente em outras linguagens como Kotlin, Swift e C#, que faz com que toda variável criada por padrão não possa ser nula. Para que se possa atribuir um valor nulo a uma variável, é necessário explicitar a possibilidade do valor nulo,

como mostrado na Figura 7. Um outro fator diferencial da linguagem é o fato dela estar fortemente ligada ao *framework* Flutter, também da Google, para criação de interfaces gráficas. Essa ligação se materializa por a linguagem dar suporte e evoluir de acordo com as necessidades do *framework*.

```dart
// Com null safety, nenhuma dessas variáveis pode ser nula
var i = 42;
String name = getFileName();
final b = Foo();

// Explicitando uma variável como nula
int? aNullableInt = null;
```

Figura 7 – Exemplo de Null Safety em Dart. Fonte: Do Autor.

Sobre o Flutter em si, ele foi apresentado inicialmente em 2015 e lançado em versão estável em dezembro de 2018. Ele se caracteriza como um *framework* para construção de interfaces gráficas que utiliza da linguagem Dart para construir aplicações compiladas nativamente para diversas plataformas, como *mobile*, *web*, *desktop* e sistemas embarcados a partir de uma única base de código, com destaque para o fato das aplicações rodarem a 60 quadros por segundo e com o desenvolvedor conseguindo controlar cada pixel na tela.

Muitos *frameworks* para criação de interface trabalham de forma imperativa, com a construção sendo realizada por meio de objeto criados e modificações sendo feitas de forma explícita pela chamada de métodos desses objetos. O Flutter se diferencia por sua UI ser feita de forma declarativa. Os componentes de visualização, representados por objetos e comumente chamados de *Widgets*, são imutáveis e são apenas esquemáticas leves de como a tela deve ser. Para que seja feita alguma alteração na tela, algum *widget* deve acionar uma reconstrução dele mesmo, que fará com que no próximo quadro de renderização a tela seja recriada com base no novo estado. Sobre estados, a interface no Flutter se caracteriza por ser uma função do estado atual dos componentes, de forma que o estado é caracterizado como qualquer dado que seja necessário para reconstruir a interface em um dado momento do tempo. Na Figura 8, tem-se um exemplo de como funciona uma interface com Flutter. É criado um objeto que estende de *StatefulWidget*, que é um *widget* que possui um estado mutável, e possui dentro de si uma variável do tipo inteiro que guarda um valor de contador. A interface é construída dentro do método *build*, que é chamado sempre que há um acionamento para reconstrução da tela. Esse acionamento é realizado por meio da chamada do método *setState*, que é herdado da classe *State*. Assim, pela construção da tela do exemplo da Figura 8, sempre que o usuário tocar ou clicar no valor do contador sendo mostrado ao centro do tela, irá ocorrer o incremento do valor da variável do contador e a tela será reconstruída de forma a ser exibido o novo valor do contador no próximo quadro de renderização da aplicação.

```dart
class Page extends StatefulWidget {
  const Page({Key? key}) : super(key: key);

  @override
  State<Page> createState() => _PageState();
}

class _PageState extends State<Page> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        setState(() {
          count += 1;
        });
      },
      child: Center(
        child: Text(count.toString()),
      ),
    );
  }
}
```

Figura 8 – Exemplo de interface declarativa. Fonte: Do Autor.

3 Desenvolvimento

Neste capítulo serão abordados alguns problemas comuns no desenvolvimento de aplicações Flutter, de forma que para cada problema será apresentada uma ou mais resoluções de forma a aplicar padrões de projeto e de arquitetura amplamente conhecidos e já documentados. Por meio desses padrões, busca-se código mais padronizado, legível, manutenível e que haja uma menor chance de erros serem cometidos pelos desenvolvedores enquanto constroem softwares complexos.

3.1 O Problema da Representação de Estados

Por o framework Flutter ser focado na criação de aplicações cliente, ou seja, que são executadas no dispositivo do usuário, existem muitos casos em que é necessário que haja a comunicação com um servidor para execução de regras de negócio e armazenamento de dados. Essa comunicação é feita por meio de requisições à rede internet e ocorrem de maneira assíncrona, com a resposta dessas requisições demorando algum tempo para acontecer. Além disso, essas requisições podem se completar de duas formas: com sucesso e com erro. Em casos de sucesso, costuma-se ter como o resultado a informação que se deseja obter do servidor, ou então uma confirmação de que o envio de dados foi realizado com sucesso. Em casos de erros, pode-se ter mensagens ou códigos de erro de forma que o usuário possa entender melhor qual o erro que ocorreu ao realizar uma ação dentro da aplicação, já que o erro pode ser causado por motivos diversos, desde um valor inválido informado pelo usuário até erros de comunicação de baixo nível na rede. Esse mesmo princípio pode se aplicar para comunicações que não sejam em rede, como em acessos a bancos de dados no dispositivo do usuário, chamadas de sistema operacional, comunicação por *Bluetooth*, dentre outras.

Com isso em vista, têm-se que qualquer tipo de comunicação assíncrona pode ser representada por meio de um diagrama de estados, em que a partir de um estado inicial, transiciona-se para um estado de carregamento a partir de uma ação do usuário e, após a realização da ação assíncrona, pode-se transicionar para um estado de sucesso ou para um estado de erro, como descrito na Figura 9.

Para exemplificar esse mecanismo de realizar uma ação assíncrona a como pode-se melhorá-la, irá ser criada uma tela em que, ao apertar de um botão, será exibido um componente de carregamento e depois de um tempo o conteúdo mudará para um texto de sucesso ou de erro. Para realizar essa simulação de requisição assíncrona, terá-se a função *getInformationFromNetwork*, exibida na Figura 10, que irá simular uma ação assíncrona aguardando 2 segundos e, de forma aleatória, jogará uma exceção personalizada com uma

Figura 9 – Diagrama dos estados possíveis em uma ação assíncrona. Fonte: Do Autor.

mensagem de erro ou retornará uma classe de sucesso, que conterá uma *String* com a informação fictícia que teria sido obtida.

```dart
Future<SuccessModel> getInformationFromNetwork() async {
  await Future.delayed(const Duration(seconds: 2));
  final isError = Random().nextInt(10) % 2 == 0;
  if (isError) {
    throw ErrorException("Ocorreu um erro ao buscar os dados");
  }
  return SuccessModel("Informação de sucesso!");
}
```

Figura 10 – Função para simular uma ação assíncrona. Fonte: Do Autor.

3.1.1 Exemplo Adhoc para Representação de Estados

Como um exemplo *adhoc*, utilizando o mecanismo mais básico para gerenciamento de estado efêmero do Flutter, o *setState*, pode-se criar dentro de um *StatefulWidget* três variáveis, uma variável booleana para representar o carregamento da requisição, uma *String nullable* com o conteúdo da mensagem de erro e uma variável *nullable* com o objeto de sucesso, como exemplificado na Figura 11.

```dart
class AdhocPage extends StatefulWidget {
  const AdhocPage({Key? key}) : super(key: key);

  @override
  State<AdhocPage> createState() => _AdhocPageState();
}

class _AdhocPageState extends State<AdhocPage> {
  bool isLoading = false;
  String? errorMessage;
  SuccessModel? successData;
  // ...
}
```

Figura 11 – Variáveis para representar os estados da página. Fonte: Do Autor.

Em seguida, cria-se a parte visual que tratará por exibir diferentes componentes de acordo com os valores contidos nas variáveis descritas anteriormente. Todo esse trecho de código está presente dentro do método *build* da classe *_AdhocPageState*, como visto na Figura 12.

```dart
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("Abordagem Adhoc"),
      ),
      body: SizedBox(
        width: MediaQuery.of(context).size.width,
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            if (isLoading)
              const CircularProgressIndicator()
            else if (errorMessage != null)
              FailureWidget(message: errorMessage!)
            else if (successData != null)
              SuccessWidget(data: successData!)
            else
              const Text("Aperte o botão para fazer a requisição"),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          getInformation();
        },
      ),
    );
  }
```

Figura 12 – Método *build* para construção da tela da abordagem *adhoc*. Fonte: Do Autor.

Por último, nessa classe têm-se a função *getInformation*, que será responsável por alterar as variáveis e fazer a chamada à função que simulará a requisição, visto na Figura 13. Sobre essa abordagem, embora seja simples e direta, pode-se apontar alguns problemas.

```dart
Future<void> getInformation() async {
    try {
      setState(() {
        isLoading = true;
        errorMessage = null;
        successData = null;
      });
      final result = await getInformationFromNetwork();
      setState(() {
        successData = result;
        isLoading = false;
      });
    } on ErrorException catch (exception) {
      setState(() {
        errorMessage = exception.message;
        isLoading = false;
      });
    }
  }
```

Figura 13 – Função que realizará a alteração dos estados de acordo com o andamento da ação assíncrona. Fonte: Do Autor.

Um primeiro problema é a quebra do Princípio da Responsabilidade Única, visto que a classe responsável pela exibição da interface gráfica é também responsável por gerenciar o estado da aplicação.

Além disso, a forma apresentada permite a representação de estados inválidos. Por exemplo, é possível de ter a variável *isLoading* com o valor verdadeiro ao mesmo tempo que a variável *successData* não é nula ou então é possível ter ambas as variáveis *successData* e *errorMessage* não nulas ao mesmo tempo. Esses casos e outros são considerados representações inválidas, visto que não estão previstas no diagrama da Figura 9, além de ferirem o princípio DRY que afirma que todo pedaço de conhecimento deve ter uma representação única, não ambígua e autoritativa dentro de um sistema.

Outro possível problema é que a forma apresentada não força o desenvolvedor a tratar, na parte da interface visual, todos os estados possíveis, de forma que o desenvolvedor pode se esquecer de criar os componentes a serem exibidos para algum dos estados, ou então, caso seja adicionado um estado novo no futuro da aplicação, também é possível que os desenvolvedores não adicionem a tratativa na parte de renderização de tela.

3.1.2 Aplicação do Observer Pattern

Para resolver o primeiro problema de quebra do Princípio de Responsabilidade Única, pode-se criar uma classe separada para fazer o gerenciamento do estado da aplicação, de forma que a classe da interface gráfica será responsável apenas por exibir o conteúdo de acordo com o estado atual presente nela. Porém, o método *setState* só está presente dentro do *StatefulWidget* criado, não podendo ser acessado por outras classes. Um meio de resolver esse problema, pode ser feito pela implementação do padrão de projeto *Observer*. Assim, como visto na Figura 14, pode-se criar uma classe *Notifier* que conterá dentro dela uma lista de *callbacks* e os métodos *addListener*, *removeListener* e *notifyListeners* responsáveis por adicionar um ouvinte, remover um ouvinte e notificar todos os ouvintes, respectivamente.

```
1  class Notifier {
2    List<Function()> listeners = [];
3
4    addListener(Function() newListener) {
5      listeners.add(newListener);
6    }
7
8    removeListener(Function() listenerToRemove) {
9      listeners.remove(listenerToRemove);
10   }
11
12   notifyListeners() {
13     for (final listener in listeners) {
14       listener();
15     }
16   }
17 }
```

Figura 14 – Classe *Notifier* que implementa o *Observer Pattern*. Fonte: Do Autor.

Para auxiliar no quesito de exibição na tela, pode-se criar uma classe *NotifierBuilder*, que será um *StatefulWidget* que receberá uma instância da classe *Notifier* e na sua inicialização chamará o método *addListener* passando uma função anônima que chama a função *setState*, que causará a reconstrução da tela sempre que o método *notifyListeners* for chamado na classe *Notifier*. O código para a classe *NotifierBuilder*, que nada mais é que um ouvinte inscrito na classe publicadora *Notifier*, pode ser visto na Figura 15.

```dart
class NotifierBuilder extends StatefulWidget {
  final Notifier controller;
  final Widget Function() builder;

  const NotifierBuilder({
    Key? key,
    required this.controller,
    required this.builder,
  }) : super(key: key);

  @override
  State<NotifierBuilder> createState() => _NotifierBuilderState();
}

class _NotifierBuilderState extends State<NotifierBuilder> {
  @override
  void initState() {
    super.initState();
    widget.controller.addListener(listener);
  }

  void listener() {
    setState(() {});
  }

  @override
  Widget build(BuildContext context) {
    return widget.builder();
  }

  @override
  void dispose() {
    widget.controller.removeListener(listener);
    super.dispose();
  }
}
```

Figura 15 – Classe *NotifierBuilder*, um ouvinte do *Notifier*. Fonte: Do Autor.

Com isso, pode-se criar uma classe que será a nossa controladora de estado e estenderá da classe *Notifier*. Nessa classe *Controller*, têm-se as mesmas variáveis de carregamento, erro e sucesso que antes estavam contidas na classe relativa a UI. Ademais, também está presente nessa classe a função *getInformation*, porém ao invés de usar a função *setState* para acionar a atualização do estado na tela no próximo *frame*, irá ser chamada a função *notifyListeners*, que irá avisar a todos os ouvintes que houve uma alteração e fará com que o ouvinte *NotifierBuilder* chame a função *setState* em seu ouvinte. Além disso, como pode ser visto na Figura 16, as variáveis foram criadas de forma privada e, portanto, foi criado um *getter* para que seja possível que os componentes visuais utilizem os valores das variáveis, mas não os modifiquem.

Com isso, basta reescrever a parte da interface gráfica para passar a utilizar essa classe *Controller*, com o resultado sendo o mostrado na Figura 17.

```dart
class Controller extends Notifier {
  bool _isLoading = false;
  String? _errorMessage;
  SuccessModel? _successData;

  Future<void> getInformation() async {
    try {
      _isLoading = true;
      _errorMessage = null;
      _successData = null;
      notifyListeners();

      final result = await getInformationFromNetwork();

      _successData = result;
      _isLoading = false;
      notifyListeners();
    } on ErrorException catch (exception) {
      _errorMessage = exception.message;
      _isLoading = false;
      notifyListeners();
    }
  }

  bool get isLoading => _isLoading;
  String? get errorMessage => _errorMessage;
  SuccessModel? get successData => _successData;
}
```

Figura 16 – Classe *Controller*, responsável por gerenciar o estado da aplicação. Fonte: Do Autor.

Assim, com a implementação do padrão de projeto *Observer* tem-se que todo código referente a alteração de estado e obtenção dos dados pela ação assíncrona ficam separados da parte do código de construção da interface. Porém, ainda têm-se o problema de que pode-se haver a representação de estados inválidos, como descrito anteriormente, o qual será tratado a seguir.

3.1.3 Aplicação de Polimorfismo para Representação de Estados Válidos

Para resolver a questão dos estados inválidos, pode-se utilizar de polimorfismo para que seja criada uma classe base que represente o estado geral e subclasses que irão representar os reais estados da aplicação: inicial, carregamento, erro e sucesso, como visto na Figura 18. Assim, os dados de sucesso só estão disponíveis dentro da classe de estado de sucesso e o mesmo é válido para o caso de erro, de forma que esses dados não pode ser acessados de forma errônea quando a aplicação estiver em outro estado.

Com essas classes novas para representar o estado, pode-se refatorar a classe *Controller* para que não seja mais possível haver a representação inválida de múltiplos estados ao mesmo tempo, como era possível anteriormente, além de o código ficar mais enxuto e semântico pela substituição das três variáveis por uma que explicitamente representa o estado da aplicação. Essas alterações podem ser vistas na Figura 19.

```dart
class WithNotifierPage extends StatefulWidget {
  // ...
}

class _WithNotifierPageState extends State<WithNotifierPage> {
  final controller = Controller();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("Abordagem com Observer Pattern"),
      ),
      body: SizedBox(
        width: MediaQuery.of(context).size.width,
        child: Center(
          child: NotifierBuilder(
            controller: controller,
            builder: () {
              if (controller.isLoading) {
                return const CircularProgressIndicator();
              } else if (controller.errorMessage != null) {
                return FailureWidget(message: controller.errorMessage!);
              } else if (controller.successData != null) {
                return SuccessWidget(data: controller.successData!);
              }
              return const Text("Aperte o botão para fazer a requisição");
            },
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          controller.getInformation();
        },
      ),
    );
  }
}
```

Figura 17 – Código da UI utilizando o *Controller* criado. Fonte: Do Autor.

Com essa alteração feita, poode-se agora alterar a classe da interface para utilizar esses novos estados, fazendo a verificação se a variável *state* do *Controller* é dos subtipos de estado inicial, de carregamento, de erro ou de sucesso, como visto na Figura 20.

3.1.4 Aplicação de Classes Seladas para Match Exaustivo

Um problema apontado que ainda não foi resolvido é o da possibilidade de o desenvolvedor não tratar todos os estados da aplicação. Isso pode ser resolvido por meio de mecanismos introduzidos pela linguagem Dart em sua versão 3. Nessa versão foram introduzidas classes seladas e várias mecânicas de *pattern matching* na linguagem, que podem ser usadas nas classes de estado apresentadas anteriormente para que, por meio do *pattern matching*, o compilador da linguagem avise sobre estados não tratados. As classes seladas funcionam de forma que elas só podem ser estendidas ou implementadas dentro da mesma biblioteca(e uma biblioteca no Dart é tudo que está dentro de um único arquivo). Dessa forma, o compilador consegue fazer um *match* exaustivo e saber em tempo de compilação todas as subclasses que estendem a classe selada e consegue bloquear a compilação. Na Figura 21 é mostrada a alteração feita na classe de estado, que deixa de ser abstrata e passa a ser selada. Além disso, às subclasses foi adicionada a palavra reservada *final*, também introduzida na versão 3 da linguagem, para indicar

```dart
abstract class InfoState {}

class InfoInitialState extends InfoState {}

class InfoLoadingState extends InfoState {}

class InfoSuccessState extends InfoState {
  final SuccessModel successData;
  InfoSuccessState(this.successData);
}

class InfoErrorState extends InfoState {
  final String message;
  InfoErrorState(this.message);
}
```

Figura 18 – Representação de código utilizando polimorfismo. Fonte: Do Autor.

```dart
class Controller extends Notifier {
  InfoState _state = InfoInitialState();

  Future<void> getInformation() async {
    try {
      _state = InfoLoadingState();
      notifyListeners();

      final result = await getInformationFromNetwork();

      _state = InfoSuccessState(result);
      notifyListeners();
    } on ErrorException catch (exception) {
      _state = InfoErrorState(exception.message);
      notifyListeners();
    }
  }

  InfoState get state => _state;
}
```

Figura 19 – Nova versão do *Controller* utilizando as classes de estado criadas. Fonte: Do Autor.

que elas não podem ser estendidas ou implementadas fora da biblioteca, mas podem ser construídas.

Com essa alteração feita nas classes de estados, pode-se alterar a interface da aplicação para que seja feito o *pattern matching* em cima dos possíveis estados da aplicação utilizando a estrutura de *switch-case*, como visto na Figura 22, de forma que caso sejam adicionados novos estados no futuro o compilador já irá avisar que não está sendo feito o *match* exaustivo daquela classe, eliminando portanto a chance da aplicação chegar com um *bug* de não ter tratativa de um estado para os usuários.

```dart
class WithStatePage extends StatefulWidget {
  // ...
}

class _WithStatePageState extends State<WithStatePage> {
  final controller = Controller();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("Abordagem com Observer Pattern e State Pattern"),
      ),
      body: SizedBox(
        width: MediaQuery.of(context).size.width,
        child: Center(
          child: NotifierBuilder(
            controller: controller,
            builder: () {
              final state = controller.state;
              if (state is InfoLoadingState) {
                return const CircularProgressIndicator();
              } else if (state is InfoErrorState) {
                return FailureWidget(message: state.message);
              } else if (state is InfoSuccessState) {
                return SuccessWidget(data: state.successData);
              }
              return const Text("Aperte o botão para fazer a requisição");
            },
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          controller.getInformation();
        },
      ),
    );
  }
}
```

Figura 20 – UI utilizando as classes de estado criadas. Fonte: Do Autor.

```dart
sealed class InfoState {}

final class InfoInitialState extends InfoState {}

final class InfoLoadingState extends InfoState {}

final class InfoSuccessState extends InfoState {
  final SuccessModel successData;
  InfoSuccessState(this.successData);
}

final class InfoErrorState extends InfoState {
  final String message;
  InfoErrorState(this.message);
}
```

Figura 21 – Alteração nas classes de estados, introduzido o conceito de classes seladas. Fonte: Do Autor.

3.2 Melhorias para Formulários e suas Validações

Em aplicações executadas no lado do cliente, é muito comum a presença de formulários, seja para criação de conta, login, criação de recursos, edição de dados, etc. Para isso, é comum a presença de campos de formulários nessas aplicações, que são componentes de interface que permitem ao usuário digitar valores e registrá-los, seja por um

```dart
class SealedPage extends StatefulWidget {
  // ...
}

class _SealedPageState extends State<SealedPage> {
  final controller = Controller();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("Abordagem com Classes Seladas"),
      ),
      body: SizedBox(
        width: MediaQuery.of(context).size.width,
        child: Center(
          child: NotifierBuilder(
            controller: controller,
            builder: () {
              return switch (controller.state) {
                InfoInitialState() => const Text("Aperte o botão para fazer a requisição"),
                InfoLoadingState() => const CircularProgressIndicator(),
                InfoErrorState(message: final msg) => FailureWidget(message: msg),
                InfoSuccessState(successData: final data) => SuccessWidget(data: data),
              };
            },
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          controller.getInformation();
        },
      ),
    );
  }
}
```

Figura 22 – UI utilizando *pattern matching* para exibição dos componentes de acordo com o estado. Fonte: Do Autor.

envio desses dados à algum servidor ou pela escrita em um banco de dados no próprio dispositivo.

Assim, além da coleta desses dados pela aplicação cliente, é muito importante haver também a validação desses dados na própria aplicação por dois principais motivos.

O primeiro deles se relaciona com a experiência do usuário. Em caso do usuário digitar algo que seja inválido, é importante avisá-lo com boas mensagens de erros que sejam as mais descritivas possível, além de mostrar essas mensagens de erro de forma reativa, ou seja, conforme ele digita nos campos, os mesmos são processados para que ele já saiba o quanto antes se o valor está incorreto, visto que pode ser uma experiência desagradável para um usuário de uma aplicação só ser avisado sobre erros no formulário somente após preencher diversos campos de formulário. Outro motivo que denota a importância de se haver a validação na própria aplicação é de garantir a consistência de dados do formulário a ser preenchido, de forma que não sejam armazenados dados inválidos. Por exemplo, pode-se garantir que o valor em um campo que deve ser um CPF, seja realmente um CPF válido de acordo com o algoritmo de geração de CPF, ou que um campo que é uma data realmente seja uma data válida, e não uma data que aponte para um mês maior que 12, ou um dia que não existe em um determinado mês.

3.2.1 Estruturas Padrões para Lidar com Formulários

Para lidar com validações de campos de formulário, o Flutter por si só já oferece mecanismos prontos para lidar com isso, por meio do *widget* *TextFormField*, que possui uma propriedade *validator* que permite a execução de uma função anônima que será executada a cada vez que houver uma modificação no campo. Essa função deve retornar *null* em caso do campo estar válido ou uma *String* com a mensagem de error em caso do campo estar inválido. Para que já seja executada a validação conforme o usuário digitar no campo, o framework disponibiliza a propriedade *autovalidateMode*, que pode ser atribuída com o valor *AutovalidateMode.onUserInteraction*. Ainda, para que seja possível obter o valor do campo posteriormente, é necessário criar uma instância de *TextEditingController* para cada campo e adicionar no atributo *controller* de cada *TextFormField*. Ao final de um formulário, é comum ter-se um botão para fazer o registros dos dados e, para que possa-se prosseguir com a ação desejada, o formulário deve estar válido. Para isso, o framework disponibiliza uma classe *Form* que deve envolver os componentes de formulário, de forma que deve ser dada a essa instância uma *key*, do tipo *GlobalKey<FormState>*, que possibilitará a verificação da validade do formulário, como visto na Figura 23.

```dart
class _FormExamplePageState extends State<FormExamplePage> {
  final formKey = GlobalKey<FormState>();
  final controller = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("Exemplo de Formulário")),
      body: Form(
        key: formKey,
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            TextFormField(
              controller: controller,
              autovalidateMode: AutovalidateMode.onUserInteraction,
              validator: (value) {
                if (value?.isEmpty ?? false) return "Preencha o campo";
                return null;
              },
            ),
            const SizedBox(height: 24),
            CustomButton(
              text: "Registrar",
              onTap: () {
                if (formKey.currentState?.validate() ?? false) {
                  // Formulário válido
                }
              },
            ),
          ],
        ),
      ),
    );
  }
}
```

Figura 23 – Exemplo de criação de formulário com Flutter. Fonte: Do Autor.

Essa abordagem, porém, pode apresentar alguns problemas. Para demonstrar esses problemas, e também como resolvê-los por meio da aplicação de padrões de projeto, foi criado um exemplo de formulário com três campos: um para e-mail, que não é obrigatório,

mas caso seja preenchido deve ser válido, outro para senha, que é obrigatório e deve ter pelo menos 12 caracteres e pelo menos uma letra e um número, e um campo de confirmação de senha, que é obrigatório e deve verificar se seu conteúdo está idêntico ao campo de senha. Além disso, a tela terá um botão, que só deve estar habilitado caso o formulário esteja válido, como visto nas Figuras 24 e 25.

Figura 24 – Exemplos de formulários inválidos. Fonte: Do Autor.

Figura 25 – Exemplos de formulários válidos. Fonte: Do Autor.

A partir do exemplo apresentado na Figura 23, para criar esse formulário de demonstração é necessário criar as funções de validação para cada um dos campos para que seja feita a validação de cada um dos campos, como visto na Figura 26, e adicionar essa função de validação em cada campo, assim como o *TextEditingController* de cada campo, como exemplificado na Figura 27. Além disso, para que o botão reaja à validação do formulário, é necessário criar uma variável booleana e uma função que irá verificar se a validação está correta e chamar o método *setState* para atualizar a tela com o novo valor da variável, como na Figura 28. Essa função deve ser chamada sempre que houver uma alteração em cada formulário, como é possível visualizar na Figura 27.

A partir dessa demonstração inicial, pode-se notar alguns problemas. O primeiro deles é o fato da classe responsável pela tela interface gráfica da aplicação ter que se

```dart
String? validatePassword(String? value) {
  List<String> errors = [];
  if (value == null || value.isEmpty) {
    errors.add("O campo é obrigatório.");
  }
  if ((value?.length ?? 0) < 12) {
    errors.add("A senha deve conter ao menos 12 caracteres!");
  }
  if (!RegExp(r"^(?!.*\s)(?=.*\d)(?=.*[a-zA-Z]).+$").hasMatch(value ?? "")) {
    errors.add("A senha deve ter números e letras!");
  }
  return errors.isEmpty ? null : errors.join("\n");
}

String? validateEquality(String? value1, String value2) {
  if (value1 != value2) {
    return "Valores devem ser iguais";
  }
  return null;
}

String? validateEmail(String? value) {
  if (value == null || value.isEmpty) {
    return null;
  }
  if (!EmailValidator.validate(value)) {
    return "E-mail inválido";
  }
  return null;
}
```

Figura 26 – Funções para validação dos campos da demonstração. Fonte: Do Autor.

```dart
TextFormField(
  decoration: const InputDecoration(
    label: Text("E-mail"),
    border: OutlineInputBorder(),
  ),
  controller: emailController,
  autovalidateMode: AutovalidateMode.onUserInteraction,
  validator: (value) {
    return validateEmail(value);
  },
  onChanged: (value) {
    checkFields();
  },
),
```

Figura 27 – Widget TextFormField para o campo de e-mail. Fonte: Do Autor.

preocupar também com a validação do formulário, o que sobrecarrega as responsabilidades da classe e fere o Single Responsibility Principle, do SOLID. Um segundo problema é o fato de ter-se que criar explicitamente um *TextEditingController* para cada um dos campos do formulário, o que fará com que tenham muitas variáveis a serem controladas em casos de formulários com muitos campos. Outro porém é a necessidade de chamar a função *checkFields* em cada um dos campos, o que pode ser facilmente esquecido pelo desenvolvedor em algum campo, o que acarretará em um bug na aplicação. Além disso, há o fato de ter-se que chamar as funções de validação duas vezes, o que também pode ser esquecido durante o desenvolvimento e fere o princípio DRY. Um último ponto que é problemático para a escalabilidade da aplicação é o fato das funções de validação serem específicas para o formulário em questão, de forma que é possível que elas não sejam reaproveitáveis em sua totalidade para outros formulários da aplicação. Por exemplo,

```dart
class _AdhocFormPageState extends State<AdhocFormPage> {
  // ...
  bool isValid = false;

  void checkFields() {
    if (validateEmail(emailController.text) != null ||
        validatePassword(passwordController.text) != null ||
        validateEquality(passwordController.text, confirmPasswordController.text) != null) {
      isValid = false;
    } else {
      isValid = true;
    }
    setState(() {});
  }
  // ...
}
```

Figura 28 – Função para verificação da validade. Fonte: Do Autor.

para o campo e-mail do caso demonstrado não há a validação se o campo é obrigatório. Porém, se em outra parte da aplicação ocorrer um campo e-mail que seja obrigatório teria-se que: ou criar uma nova função, com trecho de código semelhante e, portanto, com duplicação de parte de base de código, ou adicionar um parâmetro booleano na função já existente, o que poderá causar bug em partes da aplicação que já estão desenvolvidas.

3.2.2 Aplicação do Padrão Builder em Validadores

Para resolver o último problema apontado, sobre a especificidade e não reaproveitamento das funções de validação, pode-se aplicar o padrão de projeto Builder, de forma que poderá ser criada a uma classe apenas para lidar com validação de forma mais simplificada. Seguindo o padrão, qualquer parte do código que precisar de fazer uma validação irá especificar, por meio de encadeamento de métodos, quais aspectos do valor de entrada que deseja-se validar e, ao final, chamar um método *build* que irá retornar *null* caso a entrada esteja válida ou uma *String* com a mensagem do que está inválido na dada entrada. A implementação feita, vista na Figura 29 consiste em criar uma classe que recebe em seu construtor a variável do tipo *String nullable* que será feita a validação. Ainda dentro dessa classe, há uma lista de *String* com os erros que podem ser registrados para a entrada dada e também uma variável booleana que indica a obrigatoriedade do campo. Assim, a classe terá uma série de métodos para pequenas validações que serão compostas por quem utilizar a classe, de forma que cada método irá retornar sua própria instância e em caso da entrada ser inválida irá adicionar uma mensagem de erro na lista de erros. Ao final, o método *build* irá retornar *null* caso a lista de erros esteja vazia ou uma *String* que concatenará todos os erros.

3.2.3 Criação de uma Estrutura para Gerenciamento de Formulários

Já para resolver os outros problemas apontados, pode-se criar uma nova estrutura separada para lidar apenas com a validação do formulário em si, de forma a isolar toda a lógica de validação da parte da interface gráfica e eliminar as duplicações de código

```dart
class Validator {
  final String? valueToTest;
  final List<String> _errors = [];
  bool _isRequired = false;

  Validator(this.valueToTest);

  bool _checkRequired() {
    return !_isRequired && (valueToTest?.isEmpty ?? false);
  }
  Validator isPasswordValid([String? message]) {
    if (_checkRequired()) return this;
    if (!RegExp(r"^(?!.*\s)(?=.*\d)(?=.*[a-zA-Z]).+$").hasMatch(valueToTest!)) _errors.add("A senha deve ter números e letras!");
    return this;
  }
  Validator isEmailValid([String? message]) {
    if (_checkRequired()) return this;
    if (!EmailValidator.validate(valueToTest!)) _errors.add(message ?? "E-mail inválido");
    return this;
  }
  Validator isEqualTo(String toCompare, [String? message]) {
    if (_checkRequired()) return this;
    if (valueToTest != toCompare) _errors.add(message ?? "Valores devem ser iguais");
    return this;
  }
  Validator isRequired([String? message]) {
    _isRequired = true;
    if (valueToTest?.isEmpty ?? false) _errors.add(message ?? "O campo é obrigatório.");
    return this;
  }
  Validator maxLength(int max, [String? message]) {
    if (_checkRequired()) return this;
    if (valueToTest!.length > max) _errors.add(message ?? "O tamanho máximo é $max!");
    return this;
  }
  Validator minLength(int min) {
    if (_checkRequired()) return this;
    if (valueToTest!.length <= min) _errors.add("O tamanho mínimo é $min!");
    return this;
  }
  String? build([String join = '\n']) => _errors.isEmpty ? null : _errors.join(join);
}
```

Figura 29 – Classe Validator com aplicação do *Builder Pattern*. Fonte: Do Autor.

apontadas anteriormente. Além disso, essa nova classe, chamada de *FormManager*, seguirá o padrão de projeto *Observer*, de forma que ela automaticamente irá verificar as alterações nos formulários e indicar aos seus ouvintes se o formulário está em um estado válido ou não.

A implementação da classe *FormManager*, vista na Figura 30 terá como estrutura de dados um *Map<String, FormObject>*, nomeada como *FormsMap*, na qual as chaves serão uma *String* com um identificador para o campo e o valor armazenado será uma instância da classe *FormObject*, que as seguintes propriedades:

- *label*, que será o texto a ser exibido na tela para cada campo do formulário;
- *validator*, que será o método *build* da classe *Validator* criada anteriormente;
- *controller*, que será uma instância de *TextEditingController* a ser criada pelo próprio *FormManager*;
- *onChange*, que será uma função a ser chamada a cada vez que houver modificação em um campo do formulário, também gerenciada pela própria classe *FormManager*;

Além dessa estrutura de dados, a classe *FormManager* terá o método *setForms*, que irá receber um *Map<String, FormObjectParameters>*, em que o *FormObjectParameters* é possui apenas os atributos *label* e *validator* da classe *FormObject*. Esse método irá passar por todas as entradas do mapa passado e irá adicionar uma nova entrada na estrutura de *FormsMap*, de forma a criar o *TextEditingController* para cada campo e definir a função anônima a ser chamada quando o campo for alterado. Essa função anônima chamará a função privada *_checkIsValid* que passará por toda a estrutura de dados *FormsMap* e checará se o valor atual do campo, provido pelo atributo *controller*, está válido. Caso todos estejam válidos, atribuirá o valor verdadeiro para a variável *_isValid*, caso contrário, atribuirá o valor falso. Após a chamada desse método privado, será chamado o método herdado da classe *Notifier*, *notifyListeners*, que irá avisar a todos os ouvintes sobre o novo estado da classe.

```dart
typedef FormsMap = Map<String, FormObject>;

class FormManager extends Notifier {
  final FormsMap _forms = {};
  bool _isValid = false;

  void setForms(Map<String, FormObjectParameters> input) {
    for (final i in input.entries) {
      _forms.addAll({
        i.key: FormObject(
          label: i.value.label,
          validator: i.value.validator,
          controller: TextEditingController(),
          onChange: (_) {
            _checkIsValid();
            notifyListeners();
          },
        ),
      });
    }
  }

  void _checkIsValid() {
    for (final item in _forms.entries) {
      if (item.value.validator(item.value.controller.text) != null) {
        _isValid = false;
        return;
      }
    }
    _isValid = true;
  }

  Map<String, FormObject> get forms => _forms;
  bool get isValid => _isValid;
}
```

Figura 30 – Classe *FormManager*, responsável por encapsular todo o registro e validação dos formulários. Fonte: Do Autor.

Assim, como o *FormManager* criado, pode-se criar uma nova instância dele na classe da UI e, na inicialização da tela, por meio do método *initState*, definir quais os campos do formulário e quais serão as validações feitas sobre eles, como demonstrado na Figura 31. Essa forma de criar formulários, além de ser menos passível de erros por descrever-se as validações apenas uma vez, é muito mais semântica, já que deixa explícito

o que está sendo validado em cada campo.

```dart
class _FormProposalPageState extends State<FormProposalPage> {
  final formManager = FormManager();

  @override
  void initState() {
    super.initState();
    formManager.setForms({
      "email": FormObjectParameters(
        label: "E-mail",
        validator: (v) => Validator(v).isEmailValid().build(),
      ),
      "password": FormObjectParameters(
        label: "Senha",
        validator: (v) => Validator(v).minLength(12).isPasswordValid().isRequired().build(),
      ),
      "confirm_password": FormObjectParameters(
        label: "Confirmar Senha",
        validator: (v) => Validator(v).isEqualTo(formManager.forms["password"]!.controller.text, "As senhas devem ser iguais").isRequired().build(),
      ),
    });
  }

  // ...
}
```

Figura 31 – Inicialização dos formulários na tela. Fonte: Do Autor.

Para terminar o uso das novas estruturas criadas, basta passar as informações do *FormManager* para os componentes da interface. Nisso, o *TextFormField* terá que receber os parâmetros *label*, *controller*, *validator* e *onChange* em cada campo. Além disso, para customizações visuais nos campos de formulário, como adição de ícones e customização de fonte, teria-se que repetir grandes trechos de código. Para evitar essa duplicação desnecessária, pode-se criar um componente customizado para o formulário, como visto na Figura 32.

```dart
class CustomTextField extends StatelessWidget {
  final FormObject form;

  const CustomTextField({
    Key? key,
    required this.form,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return TextFormField(
      decoration: InputDecoration(
        border: const OutlineInputBorder(),
        label: Text(form.label),
      ),
      controller: form.controller,
      onChanged: form.onChange,
      autovalidateMode: AutovalidateMode.onUserInteraction,
      validator: form.validator,
    );
  }
}
```

Figura 32 – Criação do componente personalizado para um formulário. Fonte: Do Autor.

Assim, o uso de todas essas classes será como na Figura 33, em que um *NotifierBuilder* notificará os componentes sobre as alterações do *FormManager* e será listado de forma direta quais os campos existentes para aquele formulário, além de o widget *Cus-*

tomButton já saber sempre se o estado do formulário é válido ou não para bloquear ou não o clique sobre o botão.

```
NotifierBuilder(
  controller: formManager,
  builder: () {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        CustomTextField(form: formManager.forms["email"]!),
        const SizedBox(height: 12),
        CustomTextField(form: formManager.forms["password"]!),
        const SizedBox(height: 12),
        CustomTextField(form: formManager.forms["confirm_password"]!),
        const SizedBox(height: 24),
        CustomButton(
          text: "Entrar",
          width: double.infinity,
          isDisabled: !formManager.isValid,
          onTap: () {},
        ),
      ],
    );
  },
)
```

Figura 33 – Uso dos formulários com reatividade no código da interface gráfica. Fonte: Do Autor.

3.3 Apresentação de Componentes de Plataformas Específicas

Em termos de identidade visual, cada sistema operacional possui seu próprio *Design System*, que corresponde a um conjunto de componentes prontos que seguem um padrão estabelecido por uma organização. Assim, têm-se que a *Google* possui o *Material Design*, a *Apple* em seus SOs possui o *Human Interface Guidelines*, a *Microsoft* com o *Windows* possui a *Fluent UI*, dentre outros. Com isso, no momento de desenvolver uma aplicação cliente utilizando o Flutter, têm-se duas opções no que tange à construção de interfaces gráficas. Pode-se optar por criar um *Design System* próprio para a aplicação, de forma que ele seja totalmente customizável e tenha um mesmo comportamento em todas as plataformas em que a aplicação irá rodar. Essa escolha colabora com a aplicação ter uma identidade visual mais forte, porém num momento inicial os usuário podem ficar perdidos pelo fato da interface não ser muito semelhante à interface do sistema operacional que usam, necessitando de um tempo para se acostumarem.

Uma outra opção é de fazer com que o layout da aplicação seja único para cada plataforma, seja em sua totalidade, seja em componentes específicos que o *Design System* do SO já possui um componente que realiza a função desejada. Para aplicações nativas, essa tarefa se torna trivial, visto que um desenvolvedor *Android*, que irá criar uma aplicação em Kotlin e possui acesso apenas ao *Material Design* da Google para a criação dos componentes. O mesmo é válido para desenvolvedores *iOS*, que já possuirão todas as ferramentas fornecidas pela Apple e conseguirão apenas criar aplicações para o ecossistema da Apple.

Porém, para desenvolvedores que utilizam de *frameworks cross-platform*, como o Flutter, para adotar essa segunda abordagem seria necessário em todo componente que se deseje ter a aparência da plataforma em que o app está rodando, fazer verificações sobre qual plaforma a aplicação está sendo utilizada, o que levará a duplicação em diversas partes da base de código. Assim, de forma a seguir o princípio DRY, essa seção desse trabalho propõe em utilizar o padrão de projeto *Factory* para resolver essa problemática. Será desenvolvida uma prova de conceito simples que mostrará um componente *Dialog* ao apertar de uma botão, de forma que ele seguirá o *Material Design* quando a aplicação estiver no *Android* e seguirá o *Cupertino Design System* quando estiver rodando no *iOS*, como mostrado na Figura 34

Figura 34 – Exemplo da visualização dos Dialog no iOS e Android, respectivamente. Fonte: Do Autor.

3.3.1 Forma Direta de Apresentar Componentes Nativos

Para a exibição de *Dialogs*, o Flutter disponibiliza a função assíncrona *showDialog*, que recebe o *BuildContext* e uma função que retornará um widget a ser exibido. Esse widget será o *CupertinoAlertDialog* para o *iOS* e um *AlertDialog* para o *Android*, ambos providos pelo próprio Flutter, de forma que será necessário chamar métodos estáticos da classe *Platform*, da biblioteca *dart:io* para verificar em qual SO a aplicação está rodando, como mostrado na Figura 35.

Apesar de bem direta, a forma apresentada acarretará em problemas futuros para o projeto, visto que haverá sempre a duplicação de código ao chamar as classes *AlertDialog* e *CupertinoAlertDialog* em vários lugares na base de código, de forma que caso o Flutter

```dart
ElevatedButton(
  child: const Text("Aperte para abrir o Dialog"),
  onPressed: () {
    showDialog(
      context: context,
      builder: (context) {
        if (Platform.isIOS) {
          return CupertinoAlertDialog(
            title: const Text("Confirmação Necessária"),
            content: const Text("Deseja confirmar a operação?"),
            actions: [
              TextButton(
                onPressed: () => Navigator.pop(context),
                child: const Text("Sim"),
              ),
              TextButton(
                onPressed: () => Navigator.pop(context),
                child: const Text("Não"),
              ),
            ],
          );
        } else {
          return AlertDialog(
            title: const Text("Confirmação Necessária"),
            content: const Text("Deseja confirmar a operação?"),
            actions: [
              TextButton(
                onPressed: () => Navigator.pop(context),
                child: const Text("Sim"),
              ),
              TextButton(
                onPressed: () => Navigator.pop(context),
                child: const Text("Não"),
              ),
            ],
          );
        }
      },
    );
  },
)
```

Figura 35 – Exemplo direto do uso de diferentes Dialogs de acordo com o SO no qual a aplicação está rodando. Fonte: Do Autor.

mude os parâmetros dessas classes em versões futuras, ou até mesmo descontinue a classe em favor de outra, seria necessário fazer alterações em vários arquivos do projeto. Além disso, têm-se o fato de sempre ser necessária as verificações de plataforma em todo lugar que for necessário exibir o *Dialog*, de forma que se em um futuro a aplicação passar a dar suporte a uma nova plataforma, será necessário ir em todas as checagens e adicionar uma nova condição para essa nova plataforma para que seja exibido um novo componente.

3.3.2 Uso do Padrão Factory para Exibir Componentes Nativos

Diante dos problemas apresentados, é possível introduzir o padrão de projeto *Factory*, no qual a partir de uma classe abstrata pode-se definir qual será a interface necessária para a criação de um *Dialog*, com a aplicação instanciando a devida implementação do *Dialog* de acordo com o sistema operacional no qual a aplicação está sendo executada.Para implementar esse padrão, é necessário primeiramente criar a interface que todas as implementações de *Dialog* devem seguir. Essa interface receberá o título, conteúdo e as ações

possíveis para os botões do *Dialog*, como visto na Figura 36.

```dart
abstract interface class BaseDialog<T> {
  Future<T?> show(
    BuildContext context, {
    required String title,
    required String content,
    required List<DialogAction> actions,
  });
}

class DialogAction {
  String text;
  void Function() onPressed;

  DialogAction({
    required this.text,
    required this.onPressed,
  });
}
```

Figura 36 – Interface para criação de Dialogs. Fonte: Do Autor.

Em seguida, deve-se criar as implementações dos *Dialogs* a partir dessa interface criada, com cada uma sendo responsável por exibir o componente correto para aquela plataforma, como visto na Figura 37.

Com as implementações da interface *BaseDialog* feita, deve-se agora realizar a configuração de qual *Dialog* será utilizado, de forma que isso deve ser feito na inicialização da aplicação, na função *main*, com a atribuição da variável global *dialog*, como pode ser visto na Figura 38

O uso do padrão pode ser visualizado na Figura 39, no qual é chamada diretamente o método *show* na variável *dialog*, em que são passados argumentos como título, conteúdo e ações, definidos na interface *BaseDialog*

Com essa implementação utilizando o padrão *Factory*, quando for necessário adicionar uma nova plataforma, bastará criar a nova implementação da interface *BaseDialog* e adicionar uma única nova condição na inicialização da aplicação. Dessa forma, qualquer parte da interface gráfica poderá exibir *Dialogs* sem ter que se preocupar com qual a plataforma na qual a aplicação está sendo executada, de forma a focar apenas no conteúdo que o *Dialog* deve receber. É importante destacar que esse padrão pode ser aplicado

```dart
class IosDialog<T> implements BaseDialog {
  @override
  Future<T?> show(BuildContext context,
      {required String title, required String content, required List<DialogAction> actions}) async {
    return showDialog(
      context: context,
      builder: (context) => CupertinoAlertDialog(
        title: Text(title),
        content: Text(content),
        actions: actions
            .map<Widget>((e) => CupertinoButton(onPressed: e.onPressed, child: Text(e.text)))
            .toList(),
      ),
    );
  }
}

class AndroidDialog<T> implements BaseDialog {
  @override
  Future<T?> show(BuildContext context,
      {required String title, required String content, required List<DialogAction> actions}) async {
    return showDialog(
      context: context,
      builder: (context) => AlertDialog(
        title: Text(title),
        content: Text(content),
        actions: actions
            .map<Widget>((e) => TextButton(onPressed: e.onPressed, child: Text(e.text)))
            .toList(),
      ),
    );
  }
}
```

Figura 37 – Implementações dos *Dialogs* com base na interface *BaseDialog*. Fonte: Do Autor.

para qualquer parte da interface, desde partes que chamem componentes de interface comuns nos sistemas operacionais, como *overlays* de carregamento, *date pickers*, *dialogs*, até componentes mais específicos da aplicação em si, como *cards*, campos de texto, botões, etc.

3.4 Aplicação de Conceitos de Arquitetura de Software

Para a abordagem prática dos estudos sobre arquitetura realizados no Capítulo 2 desse trabalho, foi construída uma aplicação cliente utilizando o framework Flutter. Essa aplicação consiste em um gerenciador de tarefas simples, no qual o usuário poderá criar novas tarefas, editá-las e marcá-las como feitas ou não feitas. Uma tarefa terá título, data, horário e uma descrição opcional. As telas de protótipo da aplicação podem ser vistas na Figura 40, que consistem de uma tela para visualização das tarefas, de forma que elas são separadas em feitas e não feitas, e também uma tela para criação e edição de uma tarefa. Toda a prototipação do projeto foi baseada na aplicação *FlutterPad*, criada em um evento promovido pela comunidade *Flutterando* em dezembro de 2023.

Do ponto de vista de ferramentas utilizadas para construção da aplicação, além

```dart
late final BaseDialog dialog;

void main() {
  dialog = _configureDialog();
  runApp(const MyApp());
}

BaseDialog _configureDialog() {
  if (Platform.isIOS) {
    return IosDialog();
  } else {
    return AndroidDialog();
  }
}
```

Figura 38 – Inicialização do *Dialog* de acordo com o Sistema Operacional. Fonte: Do Autor.

```dart
ElevatedButton(
  child: const Text("Aperte para abrir o Dialog"),
  onPressed: () {
    dialog.show(
      context,
      title: "Confirmação Necessária",
      content: "Deseja confirmar a operação?",
      actions: [
        DialogAction(text: "Sim", onPressed: () => Navigator.pop(context)),
        DialogAction(text: "Não", onPressed: () => Navigator.pop(context)),
      ],
    );
  },
)
```

Figura 39 – Uso do padrão *Factory*. Fonte: Do Autor.

do Flutter para construção da interface gráfica, destaca-se o uso das seguintes bibliotecas de Dart:

- *Isar*[1] para banco de dados local da aplicação;
- *Dio*[2] para realizar requisições HTTP na aplicação;
- *JSON Rest Server*[3] para simular uma API REST para comunicação com um servidor;
- *GetIt*[4] como provedor de instâncias de classes, auxiliando na inversão de controle e injeção de dependências;

[1] Disponível em https://pub.dev/packages/isar
[2] Disponível em https://pub.dev/packages/dio
[3] Disponível em https://pub.dev/packages/json_rest_server
[4] Disponível em https://pub.dev/packages/get_it

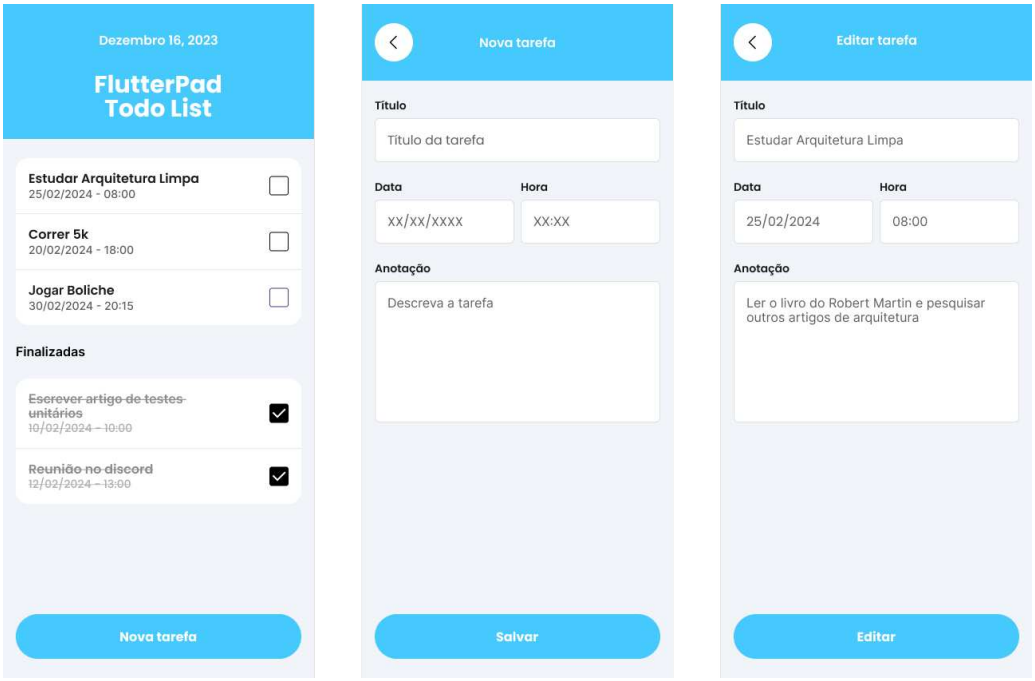

Figura 40 – Protótipo da Aplicação FlutterPad. Fonte: Comunidade Flutterando.

- *Internet Connection Checker*[5] e *Connectivity Plus*[6] para verificação de acesso à internet;
- *Equatable*[7] para facilitar a comparação de classes por seu conteúdo, sem ter que sobreescrever métodos de igualdade.

Todo o código utilizado nesse trabalho está disponível em https://tcc.alba.dev e na Figura 41 é possível ver a estrutura de pastas do repositório. Nas pastas "flutterpad", "flutterpad_simplified" e "poc", que são projetos Flutter, o código fonte escrito em Dart se localiza na pasta "lib", as demais pastas sendo as configurações nativas geradas na criação do projeto para cada uma das plaformas e o arquivo "pubspec.yaml" contendo informações do projeto, como por exemplo as dependências utilizadas. Na pasta "backend" há os arquivos "config.yaml", que define as configuraçòes da API REST a ser executada pelo pacote *JSON Rest Server*, e o arquivo "database.json", que armazena os dados da API em formato JSON e já possui uma carga inicial com algumas tarefas. Na pasta "text", há os arquivos utilizados para a escrita desse texto, em que há os arquivo ".tex" e ".bib", e as imagens utilizadas ao longo do trabalho dentro da pasta "figures". Além disso, há a pasta ".vscode" com arquivos de configuração específicos da IDE Visual Studio Code, utilizada para a realização do trabalho.

Vale destacar que essa aplicação foi desenvolvida com o intuito de conseguir fun-

[5] Disponível em https://pub.dev/packages/internet_connection_checker
[6] Disponível em https://pub.dev/packages/connectivity_plus
[7] Disponível em https://pub.dev/packages/equatable

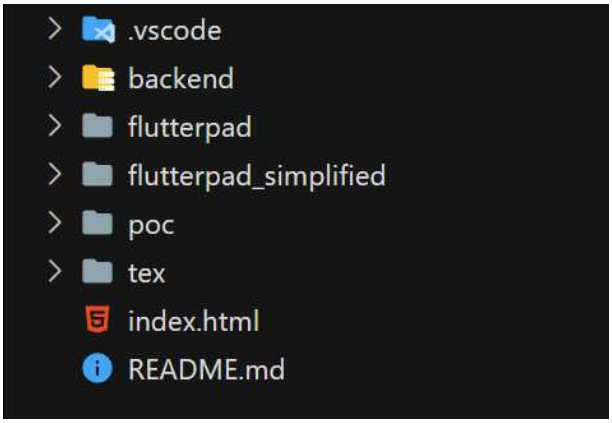

Figura 41 – Estrutura de arquivos do repositório do projeto. Fonte: Do Autor.

cionar de forma totalmente offline quando o dispositivo do usuário não tiver conexão com a internet, mas a partir do momento que houver o restabelecimento da conexão, a aplicação deve conseguir se sincronizar com a API sendo utilizada e obter dados mais recentes e enviar dados que ainda estão pendentes de serem enviados. Mais detalhes sobre essa sincronização estarão presentes no momento que for discutida a camada de infraestrutura da aplicação.

3.4.1 Uma Aplicação Clássica da Arquitetura Limpa

Com os estudos realizados em cima dos conceitos de *Domain Driven Design*, Arquitetura Hexagonal e Arquitetura Limpa, percebe-se que no quesito de arquitetura de software, os conceitos das três obras são muito semelhantes, como todos focando na não interferência de detalhes externos como interface gráfica e acesso a dados na parte do domínio e regras de negócio da aplicação, como todas buscando o desacomplamento entre as camadas e utilizando de diversos princípios de desenvolvimento de software, como os princípios do acrônimo SOLID, padrões de projeto e outros discutidos no capítulo 2 desse trabalho.

Apesar de em sua obra “Clean Architecture: A Craftsman’s Guide to Software Structure and Design” Robert Martin não definir uma estrutura exata para uma arquitetura limpa, mas sim definir aspectos que tornam uma arquitetura de um projeto de software mais simpla, é possível se inspirar nos mais de 30 capítulos da obra para formar de fato uma arquitetura para um projeto Flutter. Assim, a arquitetura montada para o aplicativo *FlutterPad* se inspira nos conceitos de DDD, Arquitetura Hexagonal e fortemente nos conceitos tornam uma arquitetura, de fato, limpa, segundo a visão de Robert Martin. Na Figura 42 pode-se visualizar um diagrama que representa a arquitetura elaborada, que contém 3 camadas: a camada de domínio, a camada de apresentação e a camada de infraestrutura, em que cada uma tem suas responsabilidades e objetivos e serão discutidas durante essa seção. Além disso, na figura é possível visualizar o fluxo das chamadas

a partir das interações do usuário que se inicia na camada de apresentação (equivalente ao lado condutor da arquitetura hexagonal) até chegar na camada de infraestrutura(que é equivalente ao lado conduzido da arquitetura hexagonal), assim como também o fluxo de dados que se mostra na direção oposta ao fluxo de chamadas.

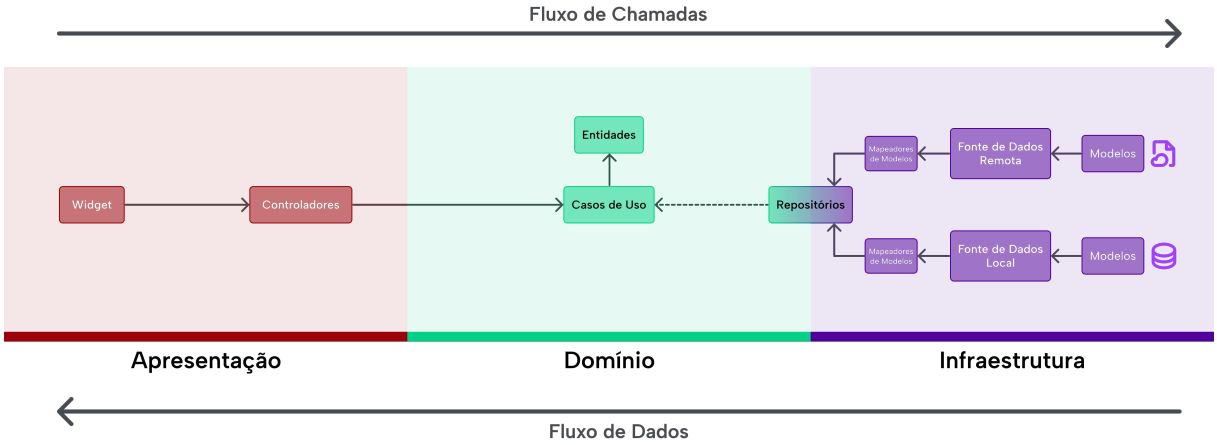

Figura 42 – Diagrama da Arquitetura Desenvolvida. Fonte: Do Autor.

Na arquitetura elaborada, a camada de domínio, como seu próprio nome diz, é responsável por tratar tudo que é relativo ao domínio da aplicação e possui 3 principais componentes: as entidades, os casos de uso e as interfaces dos repositórios. As entidades, como colocado nas obras citadas anteriormente, são responsáveis por representar a estrutura e o comportamento das regras de negócio nas quais a aplicação de software trabalha e devem possuir alguma forma de identidade que represente unicamente aquela instância. Para a aplicação *FlutterPad*, foi criada uma entidade para representação de uma tarefa, como pode ser visto na Figura 43. Essa entidade estende da classe *Equatable* para facilitar a comparação por meio dos seus atributos em vez do endereço de memória da instância da classe. Além disso, para esse projeto, optou-se por todos os seus atributos serem colocados como a palavra-chave *final*, o que garante a imutabilidade da classe e faz com que seus atributos não possam ser alterados após a instanciação da classe. Como comportamento, por se tratar de um exemplo simples, foram criado 2 métodos, um para marcar a tarefa como concluída, e que retornará uma nova instância da classe, e o método *copyWith*, que trabalha de forma a retornar uma nova instância da classe com novos atributos passados no método.

Ainda na camada de domínio, têm-se os casos de uso. Inspirados nos casos de uso descritos na Arquitetura Limpa e nos *Application Services* descritos no DDD, os casos de uso são os responsáveis por descrever o modo como um sistema automatizado é usado, de forma a resolver uma intenção completa ou um fluxo de uso exigido pelo usuário da aplicação. Na aplicação desenvolvida foi criada uma classe abstrata para representar os casos de uso, em que define-se um tipo genérico para entrada de dados e um para a saída, além da presença de uma função que executará o caso de uso, como visto na Figura

```dart
class TaskEntity extends Equatable {
  final String id;
  final String text;
  final String? description;
  final DateTime date;
  final bool completed;

  const TaskEntity({
    required this.id,
    required this.text,
    required this.description,
    required this.date,
    required this.completed,
  });

  TaskEntity markTaskCompletion(bool completed) {
    return copyWith(completed: completed);
  }

  TaskEntity copyWith({
    String? id,
    String? text,
    String? description,
    DateTime? date,
    bool? completed,
  }) {
    return TaskEntity(
      id: id ?? this.id,
      text: text ?? this.text,
      description: description ?? this.description,
      date: date ?? this.date,
      completed: completed ?? this.completed,
    );
  }
}
```

Figura 43 – Código para classe TaskEntity. Fonte: Do Autor.

44. Como exemplo de caso de uso desenvolvido, pode-se colocar o caso de uso utlizado para marcar uma tarefa como completa ou não, como mostrado na Figura 45. Essa classe estenderá da classe de *Usecase* criada anteriormente e receberá como entrada um DTO que conterá a tarefa que será manipulado e se ela deve ser marcada como realizada ou não.

```dart
abstract base class Usecase<InputParam, OutputParam> {
  Future<OutputParam> call(InputParam input);
}

```

Figura 44 – Classe genérica para representação de casos de uso. Fonte: Do Autor.

Por último na camada de domínio, têm-se os repositórios, que inspirados no padrão de projeto *Repository* do DDD, servem como uma abstração para o acesso a dados que os casos de uso necessitarão, como visto na Figura 45, que recebe uma instância de *TaskRepository* em seu construtor, e que é usado para chamar o método responsável por atualizar a tarefa após ela ser marcada como realizada ou não. Os repositórios criados nessa camada se caracterizam por ser apenas a interface que fará o acesso aos dados que os casos de uso necessitarão, de forma que a implementação desses repositórios deve estar em outra camada que não seja a de domínio, visto que essa camada não tem como responsabilidade lidar com persistência e acesso a dados. Ainda, vale salientar o fato dos

```dart
final class MarkTasksCompletionUsecase extends Usecase<TaskCompletionParams, void> {
  final TasksRepository _repository;

  MarkTasksCompletionUsecase(this._repository);

  @override
  Future<void> call(TaskCompletionParams input) async {
    TaskEntity task = input.task;
    task = task.markTaskCompletion(input.completion);
    await _repository.updateTask(task);
  }
}

typedef TaskCompletionParams = ({
  TaskEntity task,
  bool completion,
});
```

Figura 45 – Caso de uso para marcação de realização de tarefas. Fonte: Do Autor.

casos de uso especificarem uma interface a ser passada em seu construtor, o que atende o *Dependency Inversion Principle* do SOLID de Robert Martin e permite a Inversão de Controle sobre as dependências da aplicação, de forma que, por exemplo, na criação de testes unitários para os casos de uso é possível criar uma outra implementação que simule o acesso a dados, ao invés de depender de reais acessos a bancos de dados ou serviços externos para testar os casos de uso. Na Figura 46 pode-se ver os métodos da interface utilizados na aplicação. Outro ponto de destaque é que, caso a aplicação tenha outros modelos de domínio, deve-se criar outros repositórios para acesso e persistência de dados, ao invés de utilizar o repositório já criado, o que colaborará para a manutenção e legibilidade da aplicação por aplicar o *Interface Segregation Principle* do SOLID.

```dart
abstract interface class TasksRepository {
  Future<void> createTask(TaskEntity task);
  Future<void> updateTask(TaskEntity task);
  Future<List<TaskEntity>> getPendingTasks();
  Future<List<TaskEntity>> getCompletedTasksInLastMonth();
  Future<void> synchronizeTasks();
}

```

Figura 46 – Interface do Repositório de Tarefas. Fonte: Do Autor.

A camada de infraestrutura, têm-se como principal responsabilidade a realização de acesso a dados, seja por meio de escritas e leituras à bancos de dados locais do dispositivo do usuário, seja por meio de requisições na internet por meio de protocolos como HTTP ou gRPC, ou até mesmo comunicação por *Bluetooth* com outros dispositivos. Como visto na Figura 42, a camada de infraestrutura tem como seus componentes: repositório, modelos, fontes de dados e mapeadores.

Os repositórios nessa camada são as implementações concretas das interfaces dos repositórios existentes na camada de domínio. Essas implementações têm como objetivo escolher a fonte de dados apropriada para obter o dado exigido pela interface. A escolha da fonte de dados pode ser feita por meio de verificações de o dispositivo do usuário tem acesso a conexão de internet, para então saber se deve obter o dado de um banco de

dados local do dispositivo ou se deve consultar algum tipo de API disponível online. A depender da lógica da aplicação, também é possível que não seja necessário fazer esse tipo de escolha e o repositório tenha apenas o papel de chamar diretamente uma das fontes de dados para realizar a operação, atuando como um intermediário da operação definida da interface que o caso de uso necessita.

Na arquitetura criada para essa aplicação, os modelos estão muito relacionados com as fontes de dados, de forma que o modelo é a representação de uma entidade para aquela fonte de dados, visto que as representações, por conta dos mecanismo de persistência ou de requisições na web, podem diferir do modo como a aplicação representa a regra de negócio por meio das entidades. Assim, como se deseja que as entidades não sejam "sujas" pelos mecanismos relacionados a persistência de dados, há a criação dos modelos. Além disso, para que se possa enviar os dados dos modelos para a camada de domínio, há a presença dos mapeadores, que são classes que conterão métodos que conseguem converter um modelo numa entidade e vice-versa. Na Figura 47, tem-se a classe que representa uma tarefa no banco de dados local, que na aplicação desenvolvida utiliza a biblioteca *Isar*. Pode-se notar que essa classe possui uma série de adições em relação à entidade de tarefa mostrada na Figura 43, como por exemplo a anotação *@collection* presente no topo da classe, a presença de um identificador novo exigido pela biblioteca e outros 2 atribuitos *syncDate* e *createdRemotelly*, utilizados para sincronização de dados na aplicação.

```dart
@collection
class IsarTaskModel {
  Id id = Isar.autoIncrement;

  String taskId;
  String text;
  String? description;
  DateTime date;
  bool completion;
  DateTime? syncDate;
  bool createdRemotelly = false;

  IsarTaskModel({
    required this.taskId,
    required this.text,
    required this.description,
    required this.date,
    required this.completion,
    this.syncDate,
    this.createdRemotelly = false,
  });
}
```

Figura 47 – Classe que representa uma tarefa no banco de dados. Fonte: Do Autor.

No quesito das fontes de dados, foram criadas duas fontes de dados para essa aplicação, a *TasksLocalDatasource*, que utiliza da bilioteca *Isar* e a *TasksRemoteDatasource*,

que utiliza da biblioteca *Dio* para comunicar com a API criada pelo pacote *JSON Rest Server*. Na Figura 48, tem-se um exemplo de uso da biblioteca *Isar* para salvar uma instância de *IsarTaskModel* a partir de uma dada *TaskEntity*, de forma com que os detalhes relativos à persistência de dados e sincrozação do modelo do domínio estão contidos apenas na camada de infraestrutura, sem nenhum conhecimento desses detalhes pela camada de domínio.

```dart
@override
Future<void> saveTask(TaskEntity task, bool synchorized, {String? newTaskId}) async {
  final collection = _isar.collection<IsarTaskModel>();
  final taskModel = IsarTaskAdapter.fromEntity(task);
  final existingTaskOnDb = await collection.filter().taskIdEqualTo(task.id).findFirst();
  if (existingTaskOnDb != null) {
    taskModel.id = existingTaskOnDb.id;
    taskModel.createdRemotelly = existingTaskOnDb.createdRemotelly;
  }
  if (synchorized == false) {
    taskModel.syncDate = null;
  } else {
    taskModel.syncDate = clock.now();
    taskModel.createdRemotelly = true;
  }
  if (newTaskId != null) {
    taskModel.taskId = newTaskId;
  }
  await _isar.writeTxn(() async {
    await collection.put(taskModel);
  });
}
```

Figura 48 – Método do *TasksLocalDatasource* que salva uma tarefa no banco de dados. Fonte: Do Autor.

Nas duas camadas apresentadas até então, tem-se a presença apenas de código Dart e o uso de algumas bibliotecas da linguagem para ações específicas como acesso a banco de dados e requisições na internet, de forma que em ambas as camadas não a presença do Flutter. Isso ocorre pois na arquitetura elaborada, a sua presença se restringe a camada de apresentação, que tem como responsabilidade criar a interface gráfica que o usuário irá utilizar, de forma que interagirá com a regra de negócio da aplicação por meio de chamadas aos casos de uso presentes na camada de domínio. Assim, caso deseje-se utilizar outro framework para construção de interface gráfica, ou até mesmo deseje-se alterar para que a aplicação rode apenas no console da máquina do usuário, será necessário apenas alterar código na camada de apresentação, visto que a regra de negócio está presente apenas na camada de domínio e o acesso a dados está restrito à camada de infraestrutura.

Dessa forma, na camada de apresentação têm-se como componentes: os *stores*(que também podem ser chamados de controladores) e os componentes visuais, comumente chamados no Flutter de *Widgets*. Os stores tem como objetivo fazer as chamadas assíncronas aos casos de uso e fazer o papel de gerenciar o estado da aplicação, utilizando de técnicas de reatividade como apresentado na Seção 3.1. Já os *widgets* são os componentes do Flutter que serão responsáveis por realizar a criação das telas e páginas.

Além da organização de cada uma das camadas, vale destacar um trecho de código posterior a inicialização da aplicação que é o responsável por inicializar todas as instâncias das classes usadas no projeto, desde os controladores na camada de apresentação até às

fontes de dados na camada de infraestrutura. Para isso, foi utilizado a biblioteca *GetIt*, que age no projeto como localizador de serviços e antes da aplicação Flutter ser inicializada, faz o registro de todas as instâncias, o que pode ser visto na Figura 49.O registro das instâncias pode ser como *factory*, em que cada chamada para obter a instância da classe retorna uma nova instância, o que é útil em casos em que a classe tem apenas o papel de fazer uma operação e retornar um dado, como nos casos de uso, repositórios e fontes de dados. Além disso, o registro pode ser feito por meio de *lazy singletons*, no qual a biblioteca irá criar a instância daquela classe desejada apenas quando ela for chamada pela primeira vez e manterá essa instância em memória enquanto a aplicação estiver rodando, o que é útil para classes que carregam o estado da aplicação e podem ser chamadas em diversos pontos da aplicação, como é o exemplo dos controladores. Essa parte do sistema a ser chamada em sua inicialização é colocada por Martin (2017b) no capítulo 26, no qual ele descreve o componente principal como o detalhe final de uma aplicação, de forma a ser o componente mais sujo de todos os componentes e que deve montar todas as condições inicias e configurações da aplicação.

```dart
final GetIt locator = GetIt.instance;

void registerDependencies() {
  // External Packages
  locator.registerFactory(() => InternetConnectionChecker());
  locator.registerFactory(() => Connectivity());
  locator.registerFactory(() => Dio());
  locator.registerFactory(() => IsarDatasource.isar);
  locator.registerFactory(() => const Uuid());
  // Services
  locator.registerFactory<NetworkChecker>(() => MixedNetworkChecker(locator.get(), locator.get()));
  // Datasources
  locator.registerFactory<TasksRemoteDatasource>(() => RestTasksRemoteDatasource(locator.get()));
  locator.registerFactory<TasksLocalDatasource>(() => IsarTasksLocalDatasource(locator.get()));
  // Repositories
  locator.registerFactory<TasksRepository>(() => TasksRepositoryImpl(locator.get(), locator.get(), locator.get()));
  // Usecases
  locator.registerFactory(() => GetLastCompletedTasksUsecase(locator.get()));
  locator.registerFactory(() => GetPendingTasksUsecase(locator.get()));
  locator.registerFactory(() => MarkTasksCompletionUsecase(locator.get()));
  locator.registerLazySingleton(() => SynchronizeTasksUsecase(locator.get()));
  locator.registerLazySingleton(() => CreateTaskUsecase(locator.get(), locator.get()));
  locator.registerLazySingleton(() => EditTaskUsecase(locator.get()));
  // Stores
  locator.registerLazySingleton(() => GetTasksStore(locator.get(), locator.get()));
  locator.registerLazySingleton(() => MarkTaskCompletionStore(locator.get()));
  locator.registerLazySingleton(() => SynchronizeStore(locator.get()));
  locator.registerLazySingleton(() => CreateEditStore(locator.get(), locator.get()));
}
```

Figura 49 – Função para fazer o registro de instâncias antes da inicialização da aplicação. Fonte: Do Autor.

3.4.2 Uma Versão Simplificada da Arquitetura Limpa

Com a criação do projeto na Seção 3.4, percebeu-se que em uma aplicação cliente, grande parte das *Application Business Rules*, descritas na obra Arquitetura Limpa, podem estar atreladas à tela da aplicação, principalmente em casos de aplicações mais simples que tem como objetivo apenas resgatar um dado de um servidor por meio de uma API para mostrar para o usuário e enviar dados de formulário para esse mesmo servidor para que se tenha o registro, de forma que se há mais controle sobre os dados em questão quando

eles estão do lado do servidor. Em casos como esses, as entidades criadas na aplicação cliente facilmente espelham como o dado deve ser representado em tela.

Com isso em mente, foi desenvolvida uma segunda versão da aplicação *FlutterPad*, que se diferencia da anterior pelo fato de, em sua arquitetura, a camada de domínio ter se unido com a de apresentação, como pode ser visto no Figura 50. Nessa nova versão, o papel dos casos de uso foram unificados dentro do controladores, que já acessam diretamente os repositórios e manipulam as entidades.

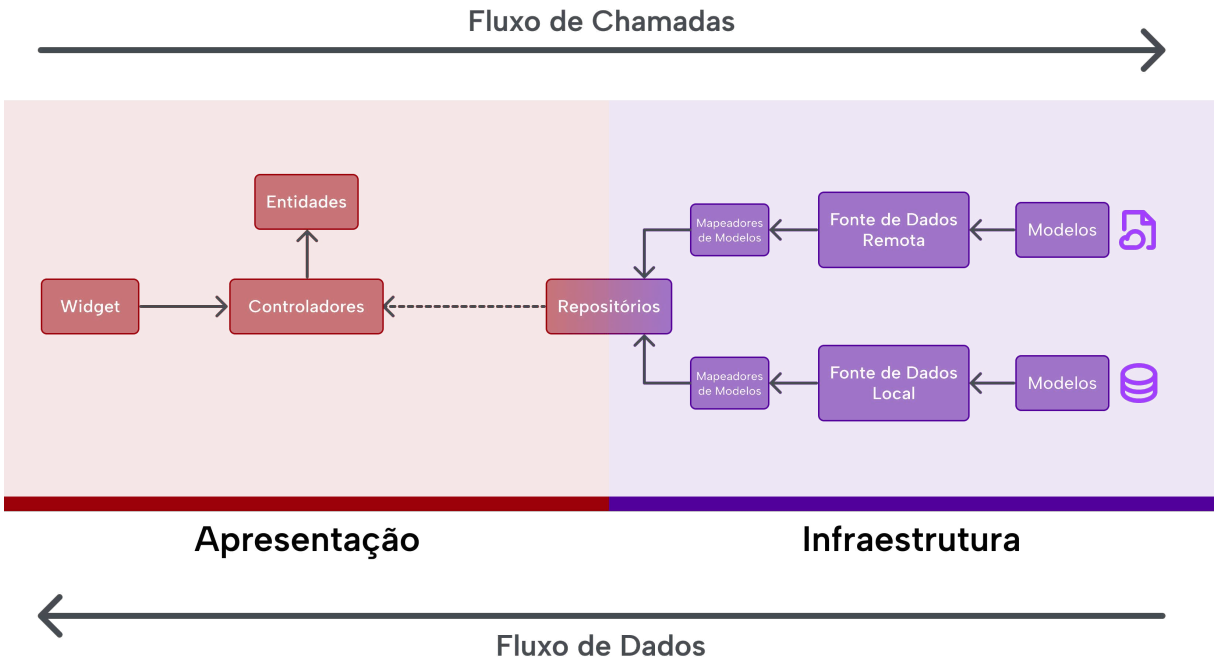

Figura 50 – Diagrama da Arquitetura Simplificada. Fonte: Do Autor.

Como vantagem dessa arquitetura simplificada, pode-se notar uma maior velocidade no momento da construção do código, visto que simplificou-se o projeto ao retirar uma camada e o componente de caso de uso, o que permitirá com que o desenvolvedor tenha um maior foco na construção da interface e na experiência do usuário que irá construir aquela aplicação, visto que, em casos de aplicações mais simples que não possuem uma forte regra de negócio do lado do cliente, o tempo gasto na criação de código dos casos de uso, que por muitas vezes apenas são um intermediário para o acesso ao repositório, irá aumentar o tempo de desenvolvimento do projeto e aumentar o tamanho da base de código de forma desnecessária. Ainda considerando o caso de aplicações simples, na parte de infraestrutura, caso seja definido que a aplicação irá sempre consumir apenas uma base de dados, pode-se remover os componentes de fonte de dados, de forma que o próprio repositório obtenha os dados, chame o mapeador para converter para a entidade e retorne esse dado para o controlador que o chamou.

4 Conclusão

Nesse trabalho foram apresentadas várias técnicas e abordagens que colaboram para uma melhor construção de um software. Quanto à aplicação dos padrões de projeto, mostra-se casos positivos nas implementações dos padrões *Observer*, *Builder*, *Factory* e *Repository* em aplicações Flutter, visto que esses padrões, no geral, ajudaram a tornar o código mais coeso e mais desacoplado.

Quanto aos estudos sobre arquitetura de software realizados, percebe-se que o mais importante não é a arquitetura utilizada em um projeto de software, e sim a aplicação de bons princípios que irão levar a uma independência entre as camadas, a não interferência de ferramentas externas na regra de negócio da aplicação e uma alta coesão e baixo acoplamento na base de código. Além disso, outro fator muito importante de ser considerado no momento de definição da arquitetura de uma aplicação cliente é qual o impacto que a regra de negócio irá ter nessa aplicação. Caso seja uma aplicação muito simples, que sirva somente como interface para o usuário acessar os dados contidos em um servidor, é esperado que uma arquitetura muito complexa venha a causar lentidão no processo de desenvolvimento, visto que as regras de negócio não estarão na aplicação cliente, e sim na aplicação que está rodando no lado do servidor.

Assim, é mais plausível que tenha-se uma arquitetura mais robusta do lado do servidor e talvez uma mais simplificada do lado do cliente. Agora, caso a aplicação cliente seja o principal foco do problema a ser resolvido de forma computacional, de forma que essa aplicação cliente deve executar uma série de regras, guardar informações em um banco local, comunicar com um ou mais servidores, tratar sobre sincronização de dados, fazer comunicação com outros dispositivos, como impressoras e fones de ouvidos, dentre outras ações mais complexas, então é muito provável que essa aplicação cliente precise ter uma arquitetura mais complexa, de forma que essa aplicação poderá ser desenvolvida e evoluída de uma forma melhor ao longo dos anos.

No quesito de trabalhos futuros, existem alguns trabalhos que podem se originar a partir deste. Uma primeira opção seria a aplicação de outros padrões de projeto em aplicações Flutter, visto que nesse trabalho foram explorados apenas quatro padrões e, considerando que apenas o catálogo de padrões fornecido por Gamma et al. (1994) possui mais de 20 padrões documentados, com certeza há mais aplicações deles possíveis de se realizar.

Outra opção de trabalho pode ser o desenvolvimento de uma aplicação mais complexa, com mais funcionalidades e mais regras de negócio do lado do cliente, de forma a realizar uma validação mais precisa das duas propostas elaboradas, de forma que pode

ser feita até algum tipo de análise quantitativa entre as propostas.

Uma terceira possibilidade, já mais próxima da área de Sistemas Distribuídos, é buscar melhorar a funcionalidade de sincronização de tarefas feita para a aplicação *FlutterPad*, visto que a implementação foi feita de forma bem simples, com o envio de tarefas pendentes quando é realizada alguma ação como abrir o aplicativo, marcar uma tarefa como feita ou criar/editar uma tarefa. Para a construção de um modelo mais elaborado, provavelmente seria necessário uma aplicação do lado do servidor que esteja pronta para lidar com a sincronização, de forma a prover a lista de tarefas para a aplicação cliente de forma contínua, por meio de algum mecanismo de *websocket* e que também consiga lidar com potenciais conflitos que podem acontecer pelo uso da aplicação cliente ao mesmo tempo em múltiplos dispositivos.

Referências

COCKBURN, A. **Hexagonal architecture**. 2005. <https://alistair.cockburn.us/hexagonal-architecture/>. [Online; acessado em 28 de outubro de 2023]. Citado 3 vezes nas páginas 10, 19 e 20.

EVANS, E. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. [S.l.]: Addison-Wesley, 2004. Citado 4 vezes nas páginas 10, 17, 18 e 19.

FOWLER, M. **Reducing Coupling**. 2001. <https://www.martinfowler.com/ieeeSoftware/coupling.pdf>. [Online; acessado em 28 de outubro de 2023]. Citado na página 14.

_____. **Yagni**. 2015. <https://martinfowler.com/bliki/Yagni.html>. [Online; acessado em 28 de outubro de 2023]. Citado na página 13.

_____. **Domain Driven Design**. 2020. <https://martinfowler.com/bliki/DomainDrivenDesign.html>. [Online; acessado em 28 de outubro de 2023]. Citado na página 18.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. M. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.]: Addison-Wesley Professional, 1994. Citado 4 vezes nas páginas 10, 14, 16 e 57.

HUNT, A.; THOMAS, D. **The Pragmatic programmer : from journeyman to master**. [S.l.]: Addison-Wesley, 2000. Citado na página 13.

KIEHL, C. **Software development topics I've changed my mind on after 6 years in the industry**. 2021. <https://chriskiehl.com/article/thoughts-after-6-years>. [Online; acessado em 28 de outubro de 2023]. Citado na página 12.

LEMOS, O. **Arquitetura Limpa na Prática**. 2022. Disponível em: <https://pay.hotmart.com/O59619511K>. Citado na página 10.

_____. **Arquitetura Limpa na Prática**. 2022. Disponível em: <https://pay.hotmart.com/O59619511K>. Citado na página 13.

MARTIN, R. **The Clean Architecture**. 2012. <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>. [Online; acessado em 28 de outubro de 2023]. Citado 2 vezes nas páginas 21 e 22.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. [S.l.]: Prentice Hall, 2017. (Robert C. Martin Series). Citado 2 vezes nas páginas 10 e 11.

_____. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. [S.l.]: Prentice Hall, 2017. (Robert C. Martin Series). Citado 4 vezes nas páginas 12, 13, 15 e 55.

MASOTTI, D. **Domain-Driven Design: guia básico sobre DDD**. 2022. <https://www.zup.com.br/blog/domain-driven-design-ddd>. [Online; acessado em 28 de outubro de 2023]. Citado na página 18.

SHVETZ, A. **Design Partterns Explained Simply**. [S.l.: s.n.], 2013. Citado na página 15.

VALENTE, M. T. **Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade**. [S.l.]: Editora: Independente, 2020. Citado na página 10.

_____. **Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade**. [S.l.]: Editora: Independente, 2020. Citado na página 12.

VERNON, V. **Implementing Domain-Driven Design**. [S.l.]: Addison-Wesley, 2013. Citado na página 18.