



UNIVERSIDADE ESTADUAL DO TOCANTINS
CÂMPUS DE PALMAS
CURSO DE SISTEMAS DE INFORMAÇÃO

**ESTUDO DE CASO SOBRE ARQUITETURA LIMPA UTILIZANDO
FLUTTER**

VINÍCIUS VICCO PETRARCA DI ASSIS MIRANDA

Palmas - TO.

2022



UNIVERSIDADE ESTADUAL DO TOCANTINS
CÂMPUS DE PALMAS
CURSO DE SISTEMAS DE INFORMAÇÃO

**ESTUDO DE CASO SOBRE ARQUITETURA LIMPA UTILIZANDO
FLUTTER**

VINÍCIUS VICCO PETRARCA DI ASSIS MIRANDA

Projeto apresentado ao Curso de Sistemas de Informação da Universidade Estadual do Tocantins - UNITINS, como parte dos requisitos para a aprovação na disciplina de Trabalho de Conclusão de Curso, sob a orientação do Jocivan Suassone Alves.

Palmas - TO.

2022



CURSO DE SISTEMAS DE INFORMAÇÃO

**ESTUDO DE CASO SOBRE ARQUITETURA LIMPA UTILIZANDO
FLUTTER**

VINÍCIUS VICCO PETRARCA DI ASSIS MIRANDA

Jocivan Suassone Alves
Orientador

José Itamar Mendes de Souza Júnior
Examinador

Gleison Batista de Sousa
Examinador

Palmas - TO.
2022

**Dados Internacionais de Catalogação na Publicação
(CIP) Sistema de Bibliotecas da Universidade Estadual
do Tocantins**

M672e MIRANDA, VINICIUS VICCO PETRARCA DI ASSIS
MIRANDA
ESTUDO DE CASO SOBRE ARQUITETURA LIMPA
UTILIZANDO FLUTTER: ARQUITETURA LIMPA
UMA ABORDAGEM TEÓRICA . VINICIUS VICCO
PETRARCA DI ASSIS MIRANDA MIRANDA. -
Palmas, TO, 2022

Monografia Graduação - Universidade Estadual do
Tocantins – Câmpus Universitário de Palmas - Curso de
Sistemas de Informação, 2022.

Orientador: Jocivan Suassone Alves Alves
Coorientador: José Itamar Mendes de Souza Júnior
Souza

1. Arquitetura Limpa. 2. Desenvolvimento de
Software. 3. Flutter. 4. SOLID.

CDD 610.7

TODOS OS DIREITOS RESERVADOS – A reprodução total ou parcial, de qualquer forma ou por qualquer meio deste documento é autorizado desde que citada a fonte. A violação dos direitos do autor (Lei nº 9.610/98) é crime estabelecido pelo artigo 184 do Código Penal.

Elaborado pelo sistema de geração automática de ficha catalográfica da UNITINS com os dados fornecidos pelo(a) autor(a).

ATA DE DEFESA DE TRABALHO DE CONCLUSÃO DE CURSO DO CURSO DE SISTEMAS DE INFORMAÇÃO DA FUNDAÇÃO UNIVERSIDADE ESTADUAL DO TOCANTINS - UNITINS

Aos **16** dias do mês de **Dezembro** de **2022**, reuniu-se de forma remota, via plataforma do google meet, às **10:30 horas**, sob a Coordenação do Professor **Jocivan Suassone Alves**, a banca examinadora de Trabalho de Conclusão de Curso em Sistemas de Informação, composta pelos examinadores Professor **Jocivan Suassone Alves** (Orientador), Professor **José Itamar Mendes de Souza Júnior** e Professor **Gleison Batista de Sousa**, para avaliação da defesa do trabalho intitulado **“Estudo de Caso sobre Arquitetura Limpa Utilizando Flutter”** do acadêmico **Vinicius Vicco Petrarca Di Assis Miranda** como requisito para aprovação na disciplina Trabalho de Conclusão de Curso (TCC). Após exposição do trabalho realizado pelo acadêmico e arguição pelos Examinadores da banca, em conformidade com o disposto no Regulamento de Trabalho de Conclusão de Curso em Sistemas de Informação, a banca atribuiu a pontuação: **7,0**.


Sendo, portanto, o Acadêmico: (X) Aprovado () Reprovado

Assinam esta Ata:


Professor Orientador: Jocivan Suassone Alves

Examinador: José Itamar Mendes de Souza Júnior

Examinador: Gleison Batista de Sousa

Documento assinado digitalmente
 JOCIVAN SUASSONE ALVES
Data: 27/01/2023 10:07:14-0300
Verifique em <https://verificador.iti.br>

Documento assinado digitalmente
 JOSE ITAMAR MENDES DE SOUZA JUNIOR
Data: 26/01/2023 21:00:28-0300
Verifique em <https://verificador.iti.br>

Documento assinado digitalmente
 GLEISON BATISTA DE SOUSA
Data: 26/01/2023 20:24:28-0300
Verifique em <https://verificador.iti.br>

Prof. Jocivan Suassone Alves
Presidente da Banca Examinadora

Este trabalho é dedicado a minha esposa e meus pais.

Agradecimentos

Agradeço pelo grande apoio da minha esposa que me apoiou de todas as formas para conclusão de minha graduação.

Agradeço a minha família pelos impulsos necessários para conclusão desta etapa.

Agradeço o apoio fundamental para escrita deste trabalho sob a orientação do professor Jocivan Suassone Alves.

Agradeço a TI da Smilink, setor que trabalho no momento da escrita deste trabalho, exerceram papel fundamental no apoio de tarefas para que pudesse dedicar um tempo na escrita deste trabalho.

*Conheça todas as teorias,
domine todas as técnicas,
mas ao tocar uma alma humana,
seja apenas outra alma humana. (Carl Jung)*

Resumo

resumo: Esta monografia apresenta alguns princípios teóricos de design de código descritos no acrônimo *SOLID* e um conjunto de técnicas contidas na arquitetura limpa visando a compreensão do código limpo e a facilidade da modificação de seu comportamento. Um estudo realizado por ([KRASNER, 2021](#)) apresenta um resultado relevante sobre softwares de baixa qualidade e seu prejuízo de 2,08 trilhões de dólares em 2021. Para compreensão de alguns dilemas técnicos ([MARTIN, 2008](#)) revela por meio da matriz de Eisenhower a importância do que engenheiros de software devem adotar. Há também a apresentação de um estudo de caso sobre o pretexto de escolha do flutter entre outros frameworks.

Palavras-chave: Flutter, Arquitetura Limpa, Software.

Abstract

abstract: This monograph presents some theoretical principles of code design described in the acronym *SOLID* and a set of techniques contained in clean architecture aimed at understanding clean code and facilitating the modification of its behavior. A study carried out by ([KRASNER, 2021](#)) presents a relevant result on low-quality software and its loss of 2.08 trillion dollars in 2021. To understand some technical dilemmas ([MARTIN, 2008](#)) reveals through the Eisenhower matrix the importance of what software engineers should adopt. There is also the presentation of a case study on the pretext of choosing flutter over other frameworks.

Palavras-chave: Flutter, Clean Architecture, Software.

Lista de ilustrações

Figura 1 – Representação da duplicação acidental.	19
Figura 2 – Representação do princípio LSP.	20
Figura 3 – Representação do princípio ISP	21
Figura 4 – A arquitetura limpa	23
Figura 5 – Proposta de arquitetura limpa aplicada ao flutter.	24
Figura 6 – Abstração de consumo de dados referente a camada de dados.	25

Lista de abreviaturas e siglas

API - Interface de programação de aplicativos.

DIP - Princípio aberto fechado.

DIP - Modelo de objeto de documento

ISP - Princípio de segregação de interface.

JSON - Objeto javascript de notação.

LSP - O Princípio da substituição de liskov.

OCP - Princípio aberto fechado.

POO - Programação orientada a objetos.

SDK - Kit de desenvolvimento de software.

SRP - Princípio da responsabilidade única.

TI - Tecnologia da informação.

UI - Interface de usuário.

XML - Linguagem de marcação extensiva.

Sumário

1	INTRODUÇÃO	13
2	PROBLEMA DE PESQUISA	14
3	JUSTIFICATIVA	15
4	OBJETIVO GERAL	16
4.1	Objetivos Específicos	16
5	METODOLOGIA	17
6	RESULTADOS	28
	REFERÊNCIAS	30

1 Introdução

Segundo a pesquisa feita por (STATISTA, 2022) existem dois sistemas operacionais principais mais utilizados, Android 71,47% e IOS 27,88% enquanto o restante fica com 0,65%, eventualmente desenvolver em ambos sistemas nativos iria requerer dois times de desenvolvimento. Flutter, um sdk desenvolvido pela google, oferece um código único para ambos sistemas operacionais, além de incluir a chamada de métodos nativos por meio de um recuso chamado *method channel*.

Durante a criação de um software é possível difundir o estudo em algumas etapas, sendo elas de requisitos, produção e implementação, contudo, este contexto se restringe a etapa de produção do software, apresentando clean architecture como uma filosofia de código com a finalidade de auxiliar na implementação, manutenção e vida útil do código. De acordo com o autor (MARTIN, 2008), programadores facilmente se encontram em um dilema, cumprir prazos que afetam a qualidade da construção do software ou desenvolver de forma eficiente podendo prejudicar alguns prazos, ainda mais com o mercado de software aquecido em que o custo de software se eleva assim como a necessidade de mão de obra qualificada para o desenvolvimento.

A arquitetura limpa é voltada para aplicação de um software mais eficiente e confiável, segundo (MARTIN, 2008) a finalidade de um software é funcionar sem interações humanas, sendo ele para gerar valor ou reduzir custos. Para (MARTIN, 2008) o comportamento deve ser escrito de forma que não gere a necessidade de contratação por conta de um desempenho inesperado, (MARTIN, 2008) ainda revela que muitas empresas passam a ter softwares insustentáveis por conta de decisões ruins dos desenvolvedores alinhadas ao pouco tempo disponibilizado pelos stakeholders.

Existem, no Brasil, 234 milhões de *smartphones* em uso segundo (FGVCIA, 2020). E, segundo (STATISTA, 2022), existem 3.80 bilhões de smartphones em uso no mundo, ou seja, 48.81% de toda a população mundial. Levando essa crescente adoção de smartphones em consideração, o desenvolvimento de soluções de software para estes dispositivos tornou-se um grande atrativo.

Segundo (FGVCIA, 2020), ficou claro que o volume total de transações realizadas em bancos por origem, em específico mobile banking atingiu 50% do total destas transações. Com este dado é possível afirmar que há um padrão no aumento de atendimentos mobile em transações.

2 Problema de Pesquisa

Ao desenvolver um projeto é preciso ter consciência de que ele está suscetível a mudanças de escopo. E, com o reconhecimento de que realizar alterações leva tempo e carece de atenção do programador para entender as funcionalidades de um código, surge como uma alternativa eficaz o uso da arquitetura limpa, que vem para minimizar o esforço de adicionar ou alterar comportamentos ao software.

Pensando nessas problemáticas, ([MARTIN, 2008](#)) a fim de documentar sua experiência sobre desenvolvimento de software, contando com sua experiência e contextualizando alguns cenários de dificuldades encontrados por desenvolvedores, como problemas em alterar comportamentos já existentes em softwares e maneiras de manter uma arquitetura limpa. Diante disso, o quão complexo pode ser para um programador modificar um código já existente ou desenvolver desde o início para uma arquitetura dentro dos critérios que o autor ressalta.

A arquitetura limpa proposta por ([MARTIN, 2008](#)) aborda que o código deve ser autoexplicativo, ou seja, ser bem escrito sem exatamente descartar a necessidade de uma documentação, mas partindo como algo auxiliar e não como pilar do processo de entendimento deste código. Portanto, o uso dessa abordagem pode trazer diversas vantagens, tanto para a própria rotina de trabalho do desenvolvedor como para outros profissionais que terão contato com o software desenvolvido.

Ainda de acordo com ([MARTIN, 2008](#)), um código mal escrito apresenta déficit financeiro para as empresas, trazendo mais desafios para o desenvolvimento prático e entrega final dos trabalhos. O tempo curto para o prazo de entrega e a necessidade de realizar mais contratações também são ações que podem acabar se tornando desafios cada vez mais comuns para algumas empresas.

3 Justificativa

A justificativa do desenvolvimento deste projeto é cativada pela ampla utilização de aplicativos móveis no Brasil. De acordo com a pesquisa feita pela Digital AdSpend, ([IAB, 2022](#)), em parceria com a Kantar Ibope Media, os mobiles foram responsáveis por 77% do consumo de publicidade digital em 2022.

Nesse sentido, é válido reconhecer que o uso de aplicativos é cada vez maior e, por isso, propor um trabalho sobre desenvolvimento mobile com flutter seguindo conceitos de uma arquitetura limpa, que surge como uma boa alternativa que pode esclarecer mais praticidade aos desenvolvedores com base na pesquisa realizada pela plataforma ([SENSORTOWER, 2022](#)), estão previstos que sejam gastos 171 bilhões de dólares neste mercado entre 2020 e 2024.

Uma pesquisa interessante apresentada por ([DATASCIENCE, 2022](#)) coletando dados do linkedin, é que em 2025 o Brasil irá atingir a faixa de nove milhões de empregos no setor de tecnologia, o que reforça a necessidade de profissionais no mercado, a maior faixa desta pesquisa é de quase sete milhões de desenvolvedores de software.

Dart que é incorporado ao SDK flutter apresenta boas características para este estudo tais como sintaxe similar a Java e C# que são popularmente conhecidas entre os desenvolvedores, o que pode tornar sua compreensão um pouco mais fácil. Segundo o site ([STACKOVERFLOW, 2021](#)) as duas linguagens estão entre as oito mais populares em uma pesquisa de opinião realizada em 2021, entretanto dart ainda se encontra na décima oitava posição mas vem tomando força se comparada a ao ano anterior da pesquisa onde se encontrava no vigésimo primeiro lugar.

Segundo ([MARTIN, 2008](#)), o autor revela ainda que os desenvolvedores modernos apresentam um excesso de confiança e trabalham demais, contudo, não dão o devido valor para um código limpo. E acreditam na ideia de que podem voltar para melhorar o código que já foi mal escrito.

Para ([KRASNER, 2021](#)), simplesmente não há bons desenvolvedores de software suficientes para criar todos os softwares novos e modificá-los de modo a atender o que os usuários precisam, O autor da pesquisa ([KRASNER, 2021](#)) ainda cita a ausência de programas educacionais para acompanhar a necessidade de desenvolvedores geradas pelo mercado.

4 Objetivo Geral

Demonstrar como clean architecture agrega valor ao código.

4.1 Objetivos Específicos

- Apontar altos custos operacionais pela falta de arquitetura limpa.
- Revelar dilemas técnicos e déficits de desenvolvedores.
- Enfatizar o valor estrutural de um software.
- Associar processos de design com estrutura presente no flutter.
- Apresentar paradigmas de estruturas de software.

5 Metodologia

Os procedimentos metodológicos deste trabalho se deram através de estudo de caso e revisão bibliográfica principalmente do livro arquitetura limpa escrito por (MARTIN, 2008), flutter e dart nas versões 3.3.0 e 2.18 entram como framework e linguagem para demonstração de conceitos específicos presentes no design de software. Este trabalho foi escrito e configurado em latex.

Um estudo de caso apresentado por (MARTIN, 2008) demonstra dados reais de um empresa que optou o anonimato, em um primeiro momento o autor revela uma tendência no crescimento no tamanho da equipe de engenharia do software, entretanto, revela um erro operacional necessitando de mais desenvolvedores para que cada release seja feita se aproximando de uma assíntota, em outras palavras, o custo do software vai drenar cada vez mais a lucratividade do modelo de negócio.

Para (MARTIN, 2008) um grande problema que os desenvolvedores insistem é na escrita de um código bagunçado que permite atender uma determinada demanda em pouco tempo, entretanto, o desenvolvimento de longo prazo e vida útil do software é gravemente afetado pois há excesso de confiança.

É comum que um *stakeholder* preze pela funcionalidade do software até porque este comportamento vai refletir a finalidade do software segundo (MARTIN, 2008) resume em gerar lucro ou reduzir um custo. (MARTIN, 2008) ressalta dois valores diferentes e cruciais importantes que devem ser mantidos pelo desenvolvedor. Os valores descritos por (MARTIN, 2008) são respectivamente comportamento e arquitetura e todo software os fornece, entretanto a maioria dos desenvolvedores focam no valor de menor importância, o comportamento.

O valor comportamento é descrito por (MARTIN, 2008) como resultado das especificações de software desenvolvidos em conjunto com o *stakeholder* onde os desenvolvedores escrevem o código fonte para que máquinas atendam essas especificações.

O valor arquitetura é ressaltado como valor mais importante por (MARTIN, 2008) pois é por meio dele que o comportamento pode ser alterado, revelando também que quanto mais melhor a arquitetura mais fácil é efetuar a modificação que atenda as expectativas do *stakeholder*.

Com a compreensão dos valores de um software, (MARTIN, 2008) apresenta a matriz de Eisenhower utilizando o valor comportamental como algo urgente mas nem sempre importante enquanto o valor arquitetural é citado como algo importante e nem sempre urgente. Ainda utilizando a matriz de Eisenhower (MARTIN, 2008) revela que

gerentes de negócios e desenvolvedores apresentam um déficit ao escolher o comportamento como algo importante e urgente ao apresentar um pensamento imediatista.

A necessidade de avaliar cautelosamente a arquitetura de um software não é uma tarefa simples e segundo (MARTIN, 2008) muitos gerentes de negócio não possuem a maturidade ou conhecimento sobre este valor o que leva a equipe de desenvolvimento a ter a obrigação de garantir e lutar por este valor.

Para atender de forma prática o valor da arquitetura de um software, (MARTIN, 2008) revela que um código deve seguir um conjunto de regras e instruções e usa o exemplo da programação estruturada que é um paradigma proposto por Edsger Wybe em 1968 com a finalidade de reduzir saltos gerados por declarações *go to* que eram prejudiciais para estrutura do programa por estruturas melhores como *if*, *else* e *do while*.

Outra estrutura conhecida pelos desenvolvedores é o paradigma da programação orientada a objetos onde (MARTIN, 2008) diz que por meio do uso disciplinado de ponteiros de função foi possível aplicar disciplina sobre a transferência indireta do controle, em outras palavras replicar comportamentos presentes em uma estrutura de código, sendo assim, a descoberta do polimorfismo.

O último paradigma referente a arquitetura de código que é apresentada por (MARTIN, 2008), é o paradigma funcional, que dispõe de alterar valores de variáveis sob uma disciplina restrita concluindo que sua finalidade é impor disciplina na atribuição.

Por fim a atenção aos três paradigmas envolvem uma escrita melhor do valor estrutural de um software que segundo (MARTIN, 2008) resultam na disciplina adicional do desenvolvedor focando em grandes preocupações como função, separação de componentes e gerenciamento de dados.

Os princípios de design propostos são esclarecidos por meio do acrônimo *SOLID*, (MARTIN, 2008) diz que um software de nível médio deve tolerar mudanças, ser de fácil compreensão e que possam servir de base para outros softwares.

Para compreensão do primeiro princípio do *SOLID*, o SRP que tem como objetivo de tornar o código mais responsável por um único objetivo, (MARTIN, 2008) cita que um método deve atender a uma função única, ou seja não incluir mais que uma finalidade.

Ainda segundo (MARTIN, 2008) um módulo deve ser responsável por apenas um, e apenas um, ator e cita a coesão como forma de amarrar o código a sua responsabilidade.

Alguns exemplos relacionados ao uso incorreto do SRP são apresentados por (MARTIN, 2008) é o de duplicação accidental, onde um ator desempenha um papel que tem suas especificações para outro ator.

Em outras palavras, o princípio é rompido em decorrência de uma classe apresentar métodos que não pertencem necessariamente a um tipo de ator, por exemplo, segundo

(MARTIN, 2008), na figura 1 é possível acompanhar três atores de gestões diferentes, sendo financeiro, tecnologia e recursos humanos. Nesta representação (MARTIN, 2008) desenvolvimento trouxe o acoplamento de três atores, fazendo com que um ator prejudique as ações realizadas por outro.

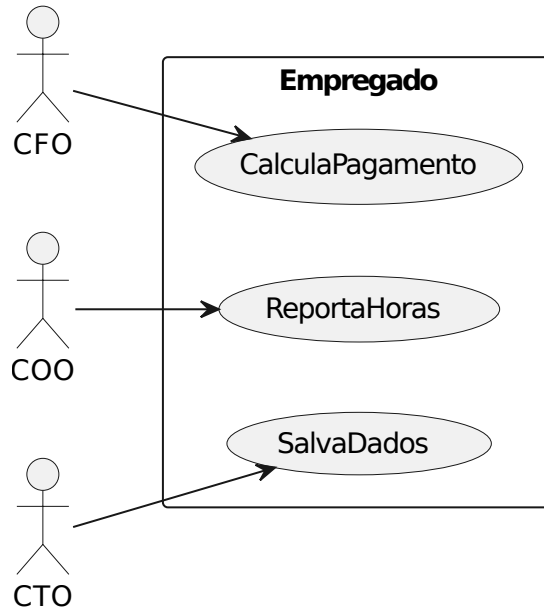


Figura 1 – Representação da duplicação acidental.

A forma de resolver este problema descrita por (MARTIN, 2008) está na visibilidade dos métodos para que cada método seja movido para classes diferentes fazendo com que uma classe não possa enxergar e alterar os métodos presentes uma na outra.

O segundo princípio de *SOLID*, é conhecido como princípio de aberto fechado. Na compreensão de (MARTIN, 2008), o autor cita este princípio como principal razão do estudo de arquitetura limpa e cita uma das falas de Berrtrand Meyer que diz, um artefato de software deve ser aberto para extensão mas fechado para modificação.

Ao tratar especificamente de abstrações, OCP, segundo (MARTIN, 2008) apresenta relacionamentos unidirecionais onde o intuito é proteger mudanças, entretanto as representar sem causar impacto sobre a interface. No dart, não existem interfaces, classes abstratas tem comportamento semelhante podendo criar métodos para serem posteriormente assinados, a implementação de uma classe abstrata permite a visualização destes métodos corresponder a sua assinatura.

Mudar o comportamento de um componente já existente pode oferecer riscos para um software dependendo de como ele foi implementado, observando isso, (MARTIN, 2008) demonstra que ao modificar um recurso visual há necessidade de proteger um controlador das mudanças exigidas na camada de apresentação, na camada de apresentação precaver-se das mudanças nas *views* e proteger as interfaces de mudanças não causando impacto em como os dados serão consumidos.

O princípio de substituição de liskov, LSP, foi apresentado por Barbara Liskov em 1988, o princípio é apontado por (MARTIN, 2008) como um guia para herança cujo objetivo é efetuar a substituição de classes de modo que não haja um forte acoplamento entre subtipos.

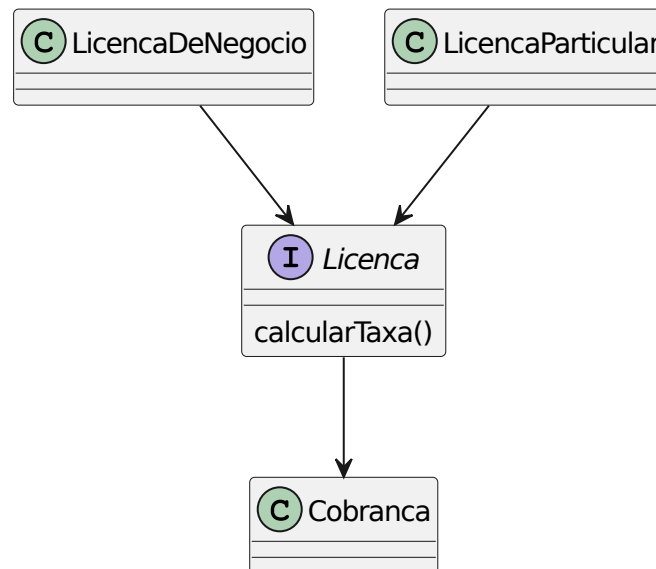


Figura 2 – Representação do princípio LSP.

Um exemplo abordado por (MARTIN, 2008) é o de duas classes, licença pessoal e licença de negócios, ambas herdam de uma classe chamada licença um método chamado calcular taxa que por sua vez herda a classe cobrança, representado pela figura 2

Ao princípio da orientação de objetos era possível associar LSP como uma forma de orientar o uso da herança, contudo, segundo (MARTIN, 2008) LSP se transformou em um princípio mais amplo de design de software para interfaces e implementações podendo elas assumirem diferentes formas. É dito ainda por (MARTIN, 2008) que várias classes presentes na linguagem ruby compartilham das mesmas assinaturas ou conjunto conjuntos de serviço da interface REST.

O princípio ISP completa algumas lacunas do LSP dependendo da linguagem de programação, segundo (MARTIN, 2008) a questão de implementação do ISP está ligada diretamente a tipagem da linguagem de programação e não a arquitetura. No dart é possível implementar este princípio por se tratar de uma linguagem tipada e orientada a objetos.

É valido ressaltar que de acordo com (MARTIN, 2008) escrever um código que depende de uma implementação específica pode gerar erros e recursos desnecessários resultando em problemas desconhecidos.

Na figura 3, possível acompanhar que cada classe herda seus respectivos métodos sem ferir outra implementação, ou seja, caso fosse necessário incluir um comportamento para um animal que voasse e tivesse a característica de peixe, utilizando dart seria possível

implementar mais de uma classe abstrata, contudo, desenvolver uma classe abstrata que reflita o comportamento esperado baseado na implementação que já existe pode ser mais vantajosa.

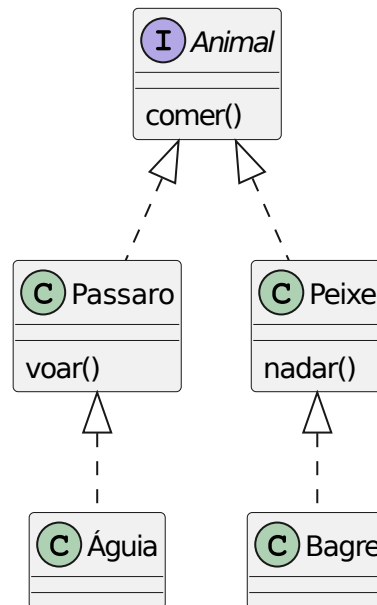


Figura 3 – Representação do princípio ISP

Durante o desenvolvimento de uma aplicação é normal se deparar com abstrações de um framework que às classes utilizam, isso existe para que segundo (MARTIN, 2008) o código atenda ao requisito de flexibilidade, o princípio da inversão de dependência refere-se as abstrações contidas na construção do código e não às classes que as implementam.

A utilização do DIP é focada em um conjunto de regras propostos por (MARTIN, 2008) que cita regras principais sintetizadas para um conjunto de práticas de programação relacionadas a DIP.

A primeira regra citada por (MARTIN, 2008) é que referir diretamente a classes concretas torna a substituição de código mais difícil, o ideal é referir-se às classes abstratas que estabelecem restrições severas sobre a criação de objetos durante a criação de objetos.

Na sequência, em um complemento a primeira regra (MARTIN, 2008) diz que vale ressaltar que existe uma diferença quanto ao uso deste critério ao utilizar de linguagens estaticamente tipadas e linguagens dinamicamente tipadas e que a herança é o relacionamento de código-fonte mais forte e rígido de todos, por fim (MARTIN, 2008) resalta também o cuidado com a dependência de objetos concretos.

Por fim, (MARTIN, 2008) aborda também a escrita de funções concretas como elemento prejudicial para criação do código limpo e resalta que a herança destes métodos não elimina as dependências e que na verdade acaba as herdando. Deste modo (MARTIN, 2008) sugere criar uma classe abstrata que contenha as funções, deste modo a dependência não faz juz a um tipo somente de implementação podendo várias implementações herdarem

o conceito da classe abstrata que foi desenvolvida.

Para (MARTIN, 2008) os princípios de *SOLID* são princípios de extrema importância para compreensão do conceito de arquitetura limpa, (MARTIN, 2008) também reforça que abordar os conceitos de uma arquitetura pode contribuir no entendimento completo de arquitetura limpa.

Uma arquitetura de sistema de software segundo (MARTIN, 2008) é a forma dada a este sistema por seus criadores, tal forma é a divisão de componentes, como se relacionam e se comportam diante da aplicação. De uma forma resumida (MARTIN, 2008) utiliza uma explicação simples para estratégia principal de uma arquitetura como a facilitação para atender ao máximo de opções possíveis pelo máximo de tempo possível.

Em outras palavras (MARTIN, 2008) afirma que o propósito primário de uma arquitetura é suportar o ciclo de vida do sistema, ou seja, facilitar seu entendimento, desenvolvimento, implantação e minimizar o custo da vida útil, isso tudo visando não maximizar a produtividade do programador.

O conceito geral de arquitetura limpa é estabelecido por (MARTIN, 2008) como um agrupamento de técnicas e decisões efetuadas por um desenvolvedor, para atender aos propósitos do parágrafo anterior. Embora arquiteturas possam variar (MARTIN, 2008) chega a citar algumas como por exemplo a arquitetura hexagonal proposta por Alistair Cockburn, a DCI proposta por James Coplien e Trygve Reenskaug e por fim a BCE proposta por Ivar Jacobson. Todas as arquiteturas conseguem produzir sistemas com algumas características.

A primeira característica importante mencionada por (MARTIN, 2008) é a de independência de frameworks, as arquiteturas não dependem de nenhum código presente em framework, isto é, sem nenhuma limitação ou adaptações geradas em decorrência dessa alteração.

Outra característica compartilhada pelas arquiteturas limpas mencionadas anteriormente é o da testabilidade, onde (MARTIN, 2008) menciona que uma UI possa ser testada sem banco de dados ou qualquer componente externo.

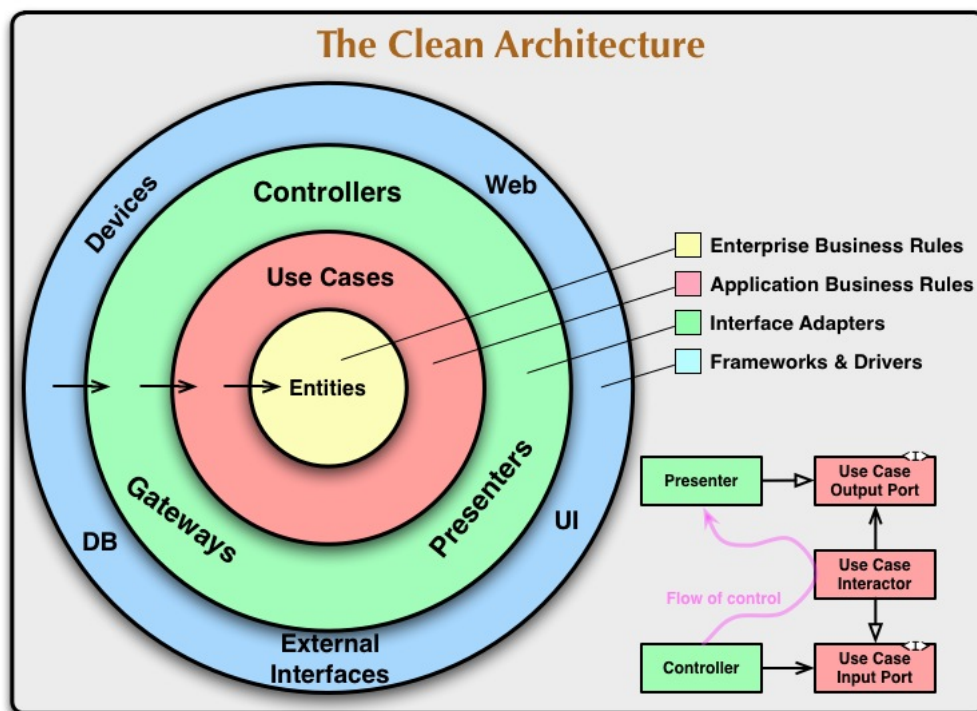


Figura 4 – A arquitetura limpa
(MARTIN, 2008)

Observando a figura 5, é possível compreender a arquitetura limpa de (MARTIN, 2008), as camadas possuem uma dependência entre si, entretanto, quanto mais fora do centro, mais próximo a configurações e estímulos externos.

A importância de separar o código em camadas é citado por (MARTIN, 2008) como algo fundamental a ser seguida como regra de dependência, ainda segundo (MARTIN, 2008) as dependências de código devem apontar somente para dentro, na direção das políticas de nível mais alto. Deste modo um elemento interno não sabe sobre os elementos do círculo externo.

Utilizando a figura 5 ao centro, as entidades, contém as regras de negócio. Para (MARTIN, 2008) as entidades podem ser classes com um conjunto de métodos, estruturas de dados e funções. Essas regras de negócio podem ser alteradas de acordo com a necessidade da empresa.

Em sequência existem os casos de uso, estes, segundo (MARTIN, 2008), são responsáveis por orquestrar o fluxo de entidades e regras de negócio, neles uma alteração não pode afetar uma entidade assim como essa camada não deve ser afetada por conta de modificações externas a ela. Entretanto, segundo (MARTIN, 2008) mudanças na operação da aplicação podem levar a alterações no caso de uso.

Em alguns casos um software deve lidar não somente com informações presentes no seu ciclo de vida passando a depender de dados externos como um BD ou uma

API, independentemente do dado que irá receber a camada de adaptadores de interface citada por (MARTIN, 2008) tem o papel de converter estes dados para um formato mais conveniente para algum agente externo, nessa camada existe a conversão de um dado para sua persistência em um banco ou similar.

A camada mais externa e mais distante da regra de negócios é definida como *frameworks e drivers*, segundo (MARTIN, 2008) não há muita programação para lidar com esta camada e sim configurações para estabelecer conexões ao círculo interno, nesta camada existem bancos de dados, frameworks e drivers que auxiliam no processo de conexão com informações externas.

Flutter é um framework e funciona com a atuação do dart para criação de modelos conceituais e suas implementações, como mencionado anteriormente muito similar ao java, sendo orientado a objetos e permitindo que limites arquiteturais sejam impostos e respeitados.

Para o desenvolvimento de uma aplicação com os conceitos abordados por (MARTIN, 2008) a construção de um modelo arquitetural semelhante ao proposto na figura 5 é citado por (MATT, 2019) em um modelo pragmático.

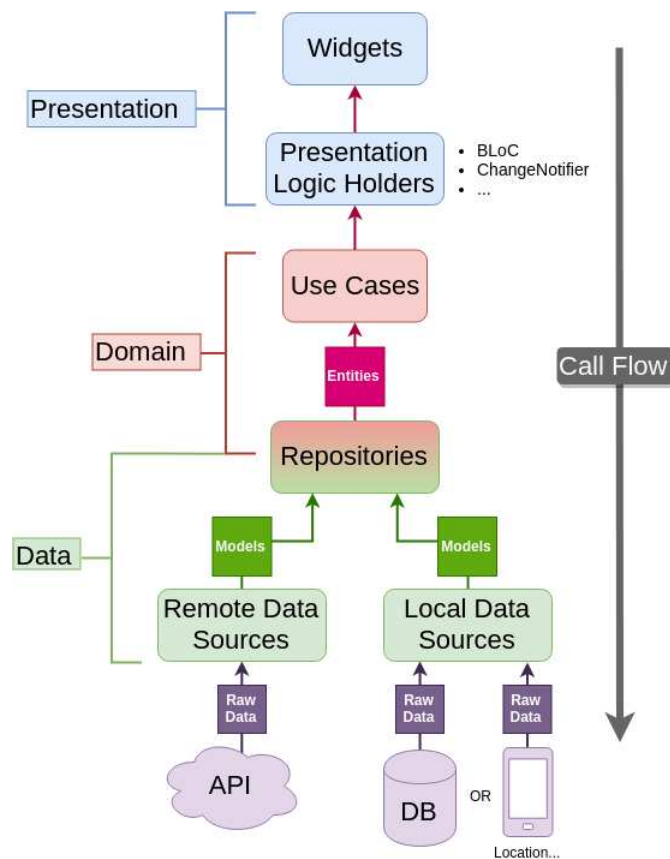


Figura 5 – Proposta de arquitetura limpa aplicada ao flutter.
(MATT, 2019)

Diante da proposta, (MATT, 2019) traça uma linha partindo da camada de

apresentação até a camada de dados, isso porque o flutter gera aplicações mobile com intuito da criação visual atendendo expectativas de clientes finais em vários negócios.

A camada de dados refere-se ao consumo externo de dados da aplicação proposta por (MATT, 2019), onde dados persistidos em bancos locais ou remotos podem ser solicitados através de uma implementação cujo a abstração é a mesma para uma API, mudando em alguns casos apenas a forma que é utilizada.

É possível construir uma classe abstrata cujo objetivo é consumir um tipo de dado específico, por exemplo informações do usuário para isso, o aplicativo deve aplicar um conjunto de estruturas atendendo essa arquitetura sem quebrar ou atravessar a camada de apresentação para a camada de dados sem passar pela camada de domínio.

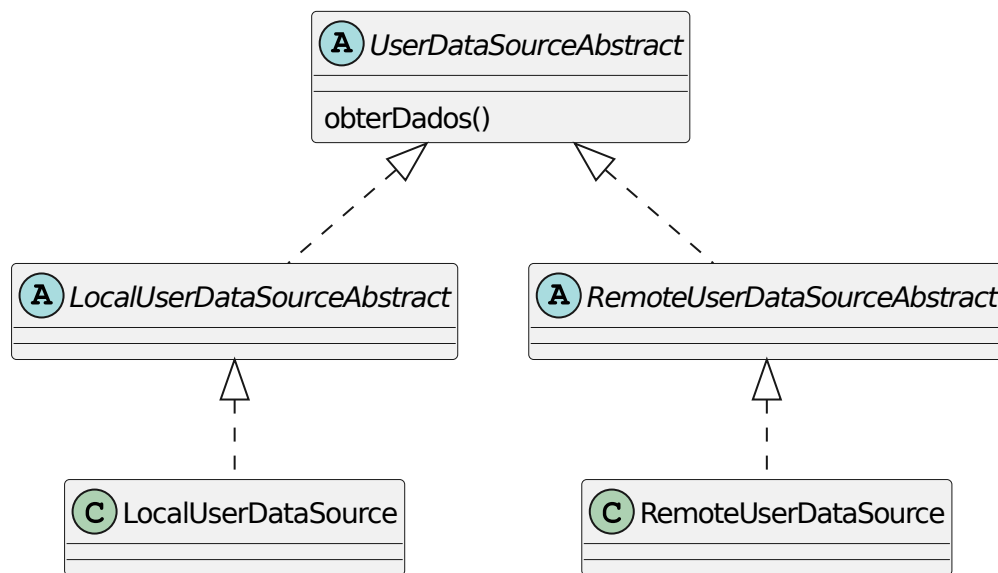


Figura 6 – Abstração de consumo de dados referente a camada de dados.

Baseado na figura 6 é possível consumir os dados com um método que existe na abstração que será respectivamente passado as duas classes abstratas que o implementam, entretanto às classes seguem com sua implementação utilizando o princípio de DIP. Em resumo a forma de consumir o dado já é esperada em um formato de dados específica, seja ele um JSON ou um XML, desta forma, o método já possui uma assinatura, entretanto, a forma que obtem esse método pode ser restrito a classe que o implementa, por exemplo, se houver a necessidade de efetuar a busca de um dado remoto a classe deve implementar da abstração remota, pois através dessa abstração podemos definir testes unidades com o tipo além de incorporar métodos restritos a este tipo de consumo.

Seguindo o fluxo de dados externos para conversão em entidades conhecidas pela aplicação, na figura 5 o repositório é apresentado como um *middleware* entre a camada de domínio e a camada de dados, sua função é direcionar os dados para camada de domínio, isto é, convertê-los em um entidade conhecida para a aplicação, além disso, o repositório não deve permitir com que exceções passem para camada superior sendo o ideal a criação

de uma entidade de erro e oferecendo a dualidade de dados entre a entidade satisfatória e uma entidade genérica para tratativa de erros.

Na camada de domínio, os modelos são utilizados pelos casos de uso são utilizados para administração do fluxo além de conter as entidades, ou seja, durante uma chamada um componente efetua a requisição de um caso de uso que contém uma entidade seja para consumo ou envio no seu fluxo de chamadas.

Por fim, a camada de apresentação proposta por (MATT, 2019) demonstra algumas possibilidades adicionais fornecidas pelo flutter chamadas de gerenciados de estado. A função destes está por sua vez associada ao método *build* que tem como objetivo refazer a tela de acordo com um conjunto de informações nela presente. O framework tolera uma vasta gama de gerenciadores de estado, contudo, a forma deste gerenciador de estado ser aplicado deve estar somente relacionado a camada de apresentação e não com o consumo de dados ou conversão de entidades regra imposta pelo princípio OCP.

Flutter é um framework que trabalha seus componentes com a orientação a objetos logo a sua componentização é feita como uma pilha de programação e ao empilhar objetos para apresentação para o usuário, elementos adotam um comportamento similar a DOM, onde elementos podem conter múltiplos filhos e um design flexível.

Flutter tem se demonstrado como uma boa alternativa para desenvolvimento híbrido e na baixa complexidade de implementação de código segundo (FREIRE, 2019). Neste estudo de caso o nubank, uma fintech totalmente digital desenvolvida no Brasil e com foco em mobile first, apresentou algumas características importantes para essa pesquisa.

O desenvolvimento do negócio apresentado por (FREIRE, 2019) listou uma breve linha temporal onde haviam adotado Kotlin e Swifts para atender as exigências de negócio, conforme o crescimento da empresa algumas soluções tiveram que ser aplicadas o que demandava mais complexidade.

Sobre o caso apresentado por (FREIRE, 2019) em 2016 a política de tecnologia era focar no desenvolvimento e criação de um time ágil, autônomo e multifuncional. Para (FREIRE, 2019) escrever uma feature duas vezes era visto como um desperdício e então aprender uma única plataforma híbrida poderia reduzir a barreira de entrada para que desenvolvedores backend pudessem contribuir com frontend mobile.

Ainda segundo (FREIRE, 2019) react native era uma alternativa estabelecida e mantida pro grandes empresas, logo escrever um código em react native e graphql era uma alternativa viável uma vez que a empresa alcançou segundo (FREIRE, 2019) treze milhões de usuários.

Com a necessidade, (FREIRE, 2019) começou a procurar outra alternativa visando cinco pontos principais sendo eles respectivamente, experiência dos desenvolvedores, vi-

abilidade a longo prazo, a não necessidade de se especializar na plataforma, custo de abstração incremental e o risco de abstração não linear.

Após análise (FREIRE, 2019) pôde chegar em um teste de usabilidade utilizando flutter onde engenheiros do software tinham que adicionar uma feature para que os usuários pudessem através de botões com valores pré definidos depositar dinheiro em sua conta.

Ainda analisando a possibilidade de uso do flutter em 2019 (FREIRE, 2019) ainda ressalta que nos critérios mais importantes react native ganhava com suporte da comunidade, restrições da loja, conhecimento da equipe e testes. Contudo a relação de suporte a terceiros com seus *breaking changes* e desempenho (FREIRE, 2019) apresenta estes dois deficits como divisor de águas.

Para (FREIRE, 2019) a experiência de desenvolvimento no flutter e suporte a reloads instantâneos como *hot reload*, documentação oficial e uma API muito estável fez com que (FREIRE, 2019) focasse no flutter como tecnologia principal.

6 Resultados

No dilema apresentado por (MARTIN, 2008), o autor propõe na conclusão de seu estudo de caso sobre a empresa que teve seu lucro equiparado aos custos de release que o melhor caminho é adotar medidas que levem a sério a qualidade de arquitetura de um software.

Segundo (MARTIN, 2008) o autor conclui que o princípio da SRP demonstra-se responsável pelos limites de criação arquitetural e também atuando como fechamento comum dos componentes.

OCP demonstra grande contribuição para o desenvolvimento da arquitetura de sistemas, segundo (MARTIN, 2008), este princípio consiste em fazer com que o sistema tenha facilidade em implementar uma mudança sem que o impacto seja compartilhado por mais entidades.

A utilização do princípio LSP deve ser estendido ao nível da arquitetura, para (MARTIN, 2008) a violação da capacidade de substituição pode contaminar a arquitetura do sistema adicionando uma quantidade indesejada de mecanismos extras na tentativa de atender situações específicas.

A atuação do princípio de segregação de interface para (MARTIN, 2008) é tratar o acoplamento de itens para que não haja dependência em sua estrutura, diante disso, o desenvolvimento de classes visando atender apenas uma especificação resulta em uma estrutura complexa para alterar.

O uso do princípio DIP revela como tratar arquiteturas em alto nível de forma a garantir sua integridade mantendo um nível de acoplamento baixo, segundo (MARTIN, 2008) Ele será o princípio organizador mais visível em qualquer diagrama de arquitetura.

Para (MARTIN, 2008) Uma arquitetura deve-se basear em algo fácil de modificação com a maior quantidade possível de abstrações tornando o código capaz de suportar múltiplas opções resultando em uma vida útil maior ao código. Dito isso, (MARTIN, 2008) reforça também que um software difícil de desenvolver terá uma vida curta portanto o papel fundamental da arquitetura é facilitar o desenvolvimento desse sistema pelas equipes de desenvolvedores e assim conclui o papel de uma arquitetura de software.

É possível inferir que para (MARTIN, 2008), nenhuma arquitetura limpa possa ser considerada caso não atenda as características principais de independência de framework, testabilidade, independência de UI, independência de bancos de dados e qualquer outra dependência externa. Quando em desacordo com a afirmação de (MARTIN, 2008) uma arquitetura não pode ser chamada de limpa pois carece de conceitos que levam um código

a uma estrutura mais relaxada levando a gastos desnecessários e horas para correções de problemas encontrados no código.

Para (MARTIN, 2008) o papel de um arquiteto deve ser deixar a maior quantidade opções abertas para sua equipe. Priorizar a arquitetura é algo reforçado por (MARTIN, 2008) como fundamental para qualquer desenvolvedor, seguir boas práticas e conceitos e não fazer justificativas como tempo tem papel fundamental para reduzir custos e priorizar a qualidade do software.

Baseado nessas informações é possível desenvolver uma arquitetura limpa voltada para o flutter que respeitem os princípios impostos pelo *SOLID* e que esteja de acordo com a figura 5 e suas camadas. O que facilita seu desenvolvimento pois impõe um conjunto de regras e camadas a serem respeitadas.

O estudo de caso apresentado por (FREIRE, 2019), concluí que a sensação de utilização do flutter é baseada no grande potencial enxergado em 2019, onde há reconhecimento de outro framework e suas boas propriedades mas ressalta sua visão para pontos positivos presentes no flutter.

Diante disso é possível afirmar que para uma grande fintech, flutter se mostrou útil, e apresentou as necessidades propostas por (MARTIN, 2008) onde um comportamento deve ser alterado com facilidade além de deixar a maior quantidade de opções abertas possíveis que foi o foco do (FREIRE, 2019) para basear a sua escolha.

Referências

DATASCIENCE, M. *Brazil is estimated to have a capacity of 9m new technology jobs*. 2022. <<https://msit.powerbi.com/view?r=eyJrIjoizWMYnJA0YzAtZGY4Zi00MTI1LTk4MjQtNWl1NTA5NDY1MzRjIiwidCI6IjcyZjk4OGJmLT>>. Accessed: 2022-11-03.

FGVCIA, M. F. S. Pesquisa anual do fgvcia. *Uso de TI nas Empresas*, v. 31, 2020.

FREIRE, A. *Porquê nós achamos que Flutter vai nos ajudar a escalar o desenvolvimento mobile no Nubank*. 2019. <<https://medium.com/flutter-comunidade-br/porqu%C3%AA-n%C3%B3s-achamos-que-flutter-vai-nos-ajudar-a-escalar-o-desenvolvimento-mobile-no-nubank-95d07b4554d7>>.

IAB. *Digital AdSpend 1H 2022*. 2022. <<https://iabbrasil.com.br/wp-content/uploads/2022/09/Digital-AdSpend-2022-1H.pdf>>. Accessed: 2022-11-01.

KRASNER, H. The cost of poor software quality in the us: a 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2021.

MARTIN, R. C. *The T_EX Book*. 1. ed. [S.l.]: Pearson Education, 2008.

MATT, R. *TDD Clean Architecture Course*. 2019. <<https://resocoder.com/2019/08/27/flutter-tdd-clean-architecture-course-1-explanation-project-structure/>>. Accessed: 2022-11-01.

SENSORTOWER. *Mobile Market Forecast 2022 - 2026*. 2022. <https://go.sensortower.com/mobile-market-forecast-2022-2026.html?utm_source=resource-library&utm_medium=resource-library&utm_content=mobile-market-forecast-2022-2026&campaign=7016f000001ukZW>. Accessed: 2022-11-01.

STACKOVERFLOW. *StackOverFlow - Most Popular Technologies 2*. 2021. <<https://insights.stackoverflow.com/survey/2021/#technology-most-popular-technologies>>.

STATISTA. *Market share of mobile operating systems in Brazil from January 2019 to August 2022*. 2022. <<https://www.statista.com/statistics/262167/market-share-held-by-mobile-operating-systems-in-brazil/>>.