UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

CAIO HENRIQUE MARQUES BAL

PROPOSTA DE ARQUITETURA ORIENTADA A *PACKAGES* PARA APLICATIVOS *FLUTTER*

PONTA GROSSA
2022

CAIO HENRIQUE MARQUES BAL

PROPOSTA DE ARQUITETURA ORIENTADA A *PACKAGES* PARA APLICATIVOS *FLUTTER*

Proposition of a package-oriented architecture for flutter apps

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação da Universidade Tecnológica Federal do Paraná (UTFPR).
Orientador: Diego Roberto Antunes

PONTA GROSSA
2022

4.0 Internacional

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

CAIO HENRIQUE MARQUES BAL

PROPOSTA DE ARQUITETURA ORIENTADA A *PACKAGES* PARA APLICATIVOS *FLUTTER*

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 08/Novembro/2022

____

Diego Roberto Antunes
Doutorado
Universidade Tecnológica Federal do Paraná

____

Richard Duarte Ribeiro
Doutorado
Universidade Tecnológica Federal do Paraná

____

Vinícius Camargo Andrade
Mestrado
Universidade Tecnológica Federal do Paraná

PONTA GROSSA

2022

AGRADECIMENTOS

Primeiramente agradeço a meus pais Cristina e Celso por me manterem no caminho e me dar todo apoio necessário para chega onde estão. Também a meu irmão Celso Junior por ser o exemplo que sempre segui e me espelhei para moldar meu caráter. A todos meus amigos que me deram todo suporte necessário para manter minha cabeça dos melhores aos piores momentos. E finalmente a Ana L. Massa, por se manter ao meu lado durante todo desenvolvimento deste trabalho.

RESUMO

O crescimento da indústria de dispositivos móveis, combinado à maior exigência dos usuários quanto à qualidade e velocidade de desenvolvimento, torna necessária a utilização de arquiteturas de software que considerem cada aspecto do desenvolvimento móvel. A partir dessa necessidade, surgem arquiteturas como a de *microfrontends,* que atribui independência e modularidade às funcionalidades ao tratá-las como um aplicativo por si só. Entretanto, a arquitetura de *microfrontends* não possui uma formalização ou modelo de referência, implicando em diferentes formas de implementação pelas empresas que a utilizam e, consequentemente, inconsistências no produto final. Este trabalho apresenta uma proposta de Arquitetura Orientada a *Packages* como formalização da arquitetura de *microfrontends* para o ambiente de desenvolvimento *Flutter*. Para esta abordagem foi analisada a arquitetura de *microfrontends* implementada nos aplicativos das empresas Nubank, Banco Votorantim, Unico e IKEA a fim de estudar os elementos fundamentais comuns a cada implementação. A partir do estudo e padronização desses elementos, a arquitetura proposta estabelece critérios e formaliza a arquitetura de *microfrontends* que pode proporcionar modularização, escalabilidade do sistema e independência entre as funcionalidades. Por fim, foi desenvolvido um aplicativo que implementa a arquitetura proposta a fim de validar seu funcionamento.

Palavras-chave: *flutter*; arquitetura de *software*; arquitetura orientada a *packages*; desenvolvimento móvel; *microfrontends;*

ABSTRACT

The growth of the mobile device industry combined with the users' increased demand for quality and speed of development makes it necessary to use software architectures that consider every aspect of mobile development. From this need emerges architectures such as microfrontends, which attribute independence and modularity to functionality by treating it as an application. However, the microfrontend architecture has no formalization or theoretical referential, implying different forms of implementation by the companies that use it and, consequently, inconsistencies in the final product. This work proposes a Packages Oriented Architecture proposal as a formalization of the microfrontend architecture for the Flutter development environment. For this approach, the microfrontends architectures implemented in the applications of the companies Nubank, Banco Votorantim, Unico and IKEA were analysed to study the fundamental elements common to each implementation. From the study and standardization of these elements, the proposed architecture establishes criteria and formalizes the microfrontend architecture providing modularization, system scalability and independence among functionalities. Finally, an application that implements the proposed architecture was developed to validate its functioning.

Keywords: flutter; software architecture; *package*-oriented architecture; mobile development; microfrontends.

LISTA DE ILUSTRAÇÕES

Figura 1 - Um exemplo da arquitetura *Pipe-and-Filter*..19
Figura 2 - Arquitetura Cliente-Servidor 19
Figura 3 - Padrão de Arquitetura em camadas 20
Figura 4 - Arquitetura do *app* do banco Nubank 21
Figura 5 - Arquitetura do *app* do Banco Votorantim S.A. 22
Figura 6 - Arquitetura *microfrontends* do *app* da Unico......................................23
Figura 7 - Construção de interface a partir de *widget*..28
Figura 8 - Arquitetura Orientada a *Packages*..33
Figura 9 - Estrutura do aplicativo 38
Figura 10 - Duas telas do aplicativo desenvolvido, sendo: A) Tela de *login*; e B) Tela de *Dashboard* 39
Figura 11 - Calculadora...40
Figura 12 - Figura representativa da tela de Listas; sendo: A) a tela de Listas; e B) mostra a seleção de *card* e os itens visíveis 40
Figura 13 - Arquivos gerados pelo comando...41
Figura 14 - Estrutura da Base..42
Figura 15 - Código da função *main*...43
Figura 16 - Pasta *core* 44
Figura 17 - Captura da função *main* 44
Figura 18 - Pasta *lists* 45
Figura 19 - *Shared package dashboard lists* 46

LISTA DE TABELAS

Tabela 1 - Vantagens e Desvantagens da Arquitetura Orientada a *Packages*...35

LISTA DE ABREVIATURAS E SIGLAS

App	*Application*
API	*Application Programming Interface*
FGVcia	Fundação Getulio Vargas: Centro de Tecnologia de Informação Aplicada
Lib	*Lib*rary
SDK	*Software Development Kit*
SOA	*Service-oriented architecture*
TI	Tecnologia da Informação
UI	*User Interface*

SUMÁRIO

1 INTRODUÇÃO 12

1.1 Objetivos 13

1.1.1 Objetivos específicos 13

1.2 Organização do trabalho 14

2 DISPOSITIVOS MÓVEIS NO SÉCULO XXI 15

2.1 Desenvolvimento para dispositivos móveis 15

2.2 Arquitetura de *Software* 16

2.2.1 Estilos de Arquitetura 17

2.2.2 Exemplos de estilos arquiteturais 18

2.2.3 Modalidades arquiteturais 24

3 *FLUTTER* ENQUANTO KIT DE DESENVOLVIMENTO 27

3.1 Principais elementos 27

3.1.1 *Widget* 27

3.1.2 *Packages* 29

3.2 Benefícios do SDK *Flutter* 30

4 ARQUITETURA ORIENTADA A *PACKAGES* 32

4.1 Definição da Arquitetura Orientada a *Packages* 32

4.2 Organização da arquitetura 32

4.2.1 *Base* 33

4.2.2 *Core* 34

4.2.3 *Packages* 34

4.2.4 *Shared Packages* 35

4.3 Vantagens e desvantagens 35

4.3.1 Vantagens 36

4.3.2 Desvantagens 37

5 APLICATIVO DESENVOLVIDO 38

5.1 Apresentação do aplicativo 38

5.2 Desenvolvimento, estrutura e código 41

5.2.1 *Base* 42

5.2.2 *Core* 43

5.2.3 *Packages* 45

5.2.4 *Shared Packages* 46

6 CONSIDERAÇÕES FINAIS 48

6.1 Trabalhos futuros 49
REFERÊNCIAS 50

1 INTRODUÇÃO

A partir da década de 90, o crescimento da indústria de dispositivos móveis foi claro. A partir desse fenômeno, tornou-se possível o acesso à informação de qualquer lugar do mundo, para isso, se faz necessária a utilização e desenvolvimento de aplicações e aplicativos móveis (FIGUEIREDO; NAKAMURA, 2003, p.16).

A evolução do mercado de programação móvel, em meados de 2008, fomentou o surgimento de grandes empresas de plataformas de distribuição de aplicativos como a *App Store* da *Apple* ou a *Play Store* do *Google*. Estima-se que no ano de 2021, mais de 230 bilhões de aplicativos foram baixados no mundo, mais de 430 mil aplicativos baixados por minuto (DATA AI, 2022).

A criação de padrões de projetos direcionados ao desenvolvimento móvel tornou-se essencial, bem como, uma análise mais profunda de projeto, sob aspectos de funcionalidades e organização. Essa criação de novos padrões e análise de projeto, ocorrem com o intuito de manter a qualidade na vasta variedade de dispositivos encontrados no mercado, pois em conjunto com o crescimento anual do mercado, o usuário se torna mais exigente com a qualidade dos aplicativos desenvolvidos (CORRÊA, 2018, p. 23).

Para que seja feita a escolha dos métodos e plataformas para o projeto de um aplicativo, é inevitável estudar sobre arquiteturas de *software*, visto que, a arquitetura de um *software* é o que vai guiar o desenvolvimento e organizar as necessidades e demandas, além de refinar o projeto, tanto em tempo de desenvolvimento, quanto em qualidade.

Diante dos fatores apresentados, surgem plataformas e linguagens de desenvolvimento para suprir a demanda de mercado, contando com ambientes de desenvolvimento voltadas para a área de dispositivos móveis como, por exemplo, *Flutter*[1], *React Native*[2] e *Swift*[3].

Dentre as apresentadas, a plataforma *Flutter* funciona como um *kit* de desenvolvimento que visa criar aplicações móveis de forma rápida e que independem de plataformas, compilando seu código diretamente para código nativo, o que permite desenvolver para *Android* e *iOS* de forma simples (WINDMILL, 2020, p.6).

---

[1] Disponível em: https://flutter.dev/
[2] Disponível em: https://reactnative.dev/
[3] Disponível em: https://developer.apple.com/swift/

O desenvolvimento móvel necessita maneiras para organizar o processo e o projeto de criação de um aplicativo, tornando necessária a utilização de arquiteturas e padrões de *design* que sejam focados no desenvolvimento móvel e nas plataformas disponíveis. A partir dessa necessidade, conceitos como *microfrontends,* uma técnica de arquitetura de *software* onde cada funcionalidade de um aplicativo é dividida em uma parte independente do sistema, são adotados no mercado por grandes empresas como Nubank, Banco Votorantim, IKEA e Unico (FLUTTERANDO, 2016).

Porém, *microfrontends* consiste em um conceito de arquitetura de *software* que não possui formalização, o que causa inconsistências entre projetos e grandes variações nos produtos (GEERS, 2020, p.13). Dessa forma, este trabalho apresenta uma proposta de Arquitetura Orientada por *Packages* desenvolvida para a plataforma *Flutter,* que formaliza o padrão de *microfrontends.*

A *Flutter* proporciona um ambiente propício para formalização e viabilização desses padrões de *design* e, consequentemente, da arquitetura proposta, pois a presença de alguns de seus elementos essenciais, como *widgets* e *packages*. Estes elementos possibilitam a modularização e independência entre funcionalidades que são características presentes nos padrões de *microfrontends* para arquiteturas móveis.

1.1 Objetivos

Este trabalho tem como objetivo propor uma arquitetura de *software* que formalize o conceito de *microfrontends* utilizado no mercado, focada na plataforma de desenvolvimento *Flutter*, visando proporcionar uma modularização na forma de desenvolvimento de projetos.

1.1.1 Objetivos específicos

Para alcançar o objetivo geral foram definidos os seguintes objetivos específicos:

- Investigar arquiteturas de *software* existentes;
- Definir, por meio de revisão bibliográfica, aspectos importantes de arquiteturas de *software;*
- Investigar o conceito de *microfrontends* presentes no mercado;
- Definir a arquitetura proposta e seus componentes;

- Implementar um aplicativo a fim de validar a arquitetura proposta.

1.2 Organização do trabalho

Este trabalho está organizado da seguinte forma: o Capítulo 2 apresenta o referencial teórico do desenvolvimento de dispositivos móveis e arquiteturas de *software*. O Capítulo 3 apresenta a plataforma *Flutter* e suas principais características. No Capítulo 4 é apresentada a proposta de Arquitetura Orientada a *Packages.* Finalmente, o Capítulo 5 discorre sobre o desenvolvimento de um aplicativo pelo autor a partir da arquitetura proposta a fim de validar a solução.

2 DISPOSITIVOS MÓVEIS NO SÉCULO XXI

É notável que o desenvolvimento de tecnologias como comunicações móveis, comunicações por satélites e redes locais sem fio crescera significativamente desde a década de 1990. A maior acessibilidade nos dispositivos móveis trouxe diversas vantagens, como a possibilidade de manter o dispositivo à mão (independentemente da sua localização) e poder acessar diversas informações e serviços. De fato, os dispositivos móveis prometem uma visão de universalidade: acessível, ainda que virtualmente, por qualquer pessoa, em qualquer lugar, a qualquer hora, em qualquer dispositivo (FIGUEIREDO; NAKAMURA, 2003, p.16).

Observa-se o aumento do uso de dispositivos móveis no mundo inteiro, principalmente o número crescente de celulares no Brasil. A 32ª edição da Pesquisa Anual do FGVcia sobre o Mercado Brasileiro de TI e Uso nas Empresas publicada em 2021, apontou que o país possui 234 milhões de celulares inteligentes, conhecidos como *smartphones* (FGV, 2020).

As aplicações para dispositivos móveis também expandiram; apenas a *Apple Store* conta com 34 milhões de desenvolvedores registrados (APPLE, 2022). O desenvolvimento de aplicações para dispositivos móveis é um modelo de negócios em constante expansão e, por isso, esta seção buscou a fundamentação teórica acerca do desenvolvimento em dispositivos móveis e a arquitetura de *software* necessária (FIGUEIREDO; NAKAMURA, 2003, p.16).

2.1 Desenvolvimento para dispositivos móveis

O desenvolvimento de aplicativos móveis é uma nova tendência no mercado da indústria de *software* e parte importante do desenvolvimento econômico de um país, além de ensino e pesquisa voltados a fornecer funções simples e intuitivas. O desenvolvimento de aplicações *mobile* é um processo de criação de aplicativos que podem ser instalados em dispositivos portáteis durante sua fabricação ou baixados posteriormente (SANTOS, 2016, p. 491).

Portanto, para se adaptar a essa nova tendência global, é necessário desenvolver aplicativos nativos para as plataformas móveis, ou desenvolver aplicativos híbridos, considerando as dificuldades que existem no desenvolvimento puramente *mobile* (CORRÊA, 2018, p. 23).

Devido à complexidade do processo de desenvolvimento de aplicações móveis, é importante que o processo de testes seja alterado para acomodar as mudanças no ambiente. Isto pode ser feito por meio do uso de novos métodos de teste, e uma das tarefas mais desafiadoras, é a de garantir que as aplicações sejam testadas adequadamente (SANTOS; CORREIA, 2015, p. 1).

Algumas das principais dificuldades encontradas no desenvolvimento para dispositivos móveis são:

- Pluralidade de dispositivos e plataformas no mercado;
- Método de testar as aplicações para garantir que funcionem nas plataformas;
- Quais modificações devem ser feitas para acomodar as diferenças entre as plataformas;

Uma solução potencial para avaliar a qualidade de aplicações móveis e realizar testes é utilizar uma *device farm* ou "fazenda de dispositivos", que são disponibilizados por plataformas focadas em testes como *AWS Device Farm*[4], *Firebase Test Labs*[5] *ou Azure App Center*[6]. Elas permitem implantar a aplicação em um grande conjunto de dispositivos e, então, em conjunto com o nó da nuvem, lançar testes de ponta a ponta, simulando o comportamento de qualquer usuário (LASO, 2022 p. 3).

No entanto, ao iniciar o desenvolvimento de um aplicativo, o escopo tecnológico do projeto precisa ser analisado sob diferentes aspectos, para assim, determinar pontos técnicos, por exemplo, a melhor plataforma e método de desenvolvimento (CORRÊA, 2018, p. 23).

Assim, a escolha do método de desenvolvimento e estrutura do projeto, bem como seus benefícios, são essenciais para compreender o funcionamento e desenvolvimento de aplicações *mobile*. E nesse contexto se faz necessário o estudo da arquitetura de *software*.

2.2 Arquitetura de *Software*

A Arquitetura de *Software* foi mencionada pela primeira vez no relatório *Software Engineering Tecnhiques*, descrito como técnicas de comum acordo entre os

[4] Disponível em: https://aws.amazon.com/pt/device-farm/
[5] Disponível em: https://firebase.google.com/products/test-lab
[6] Disponível em: https://azure.microsoft.com/en-us/products/app-center/

engenheiros do que seriam boas práticas para desenvolver um *software* suave, bonito e com bom código (RANDELL; BUXTON, 1970, p. 9).

Desde então, não há precisão sobre a definição de Arquitetura de *Software*. Como efeito, o *website "The Software Engineering Institute's"* contém mais de 150 definições elaboradas pelos mais diversos autores (CLEMENTS *et al.,* 2003, p. 3).

Para os autores Perry e Wolf (1992, p. 43), a Arquitetura de *Software* se preocupa com a seleção dos elementos arquitetônicos, suas interações e restrições a esses elementos, e as interações necessárias para fornecer uma estrutura satisfatória. Portanto, a Arquitetura de *Software* passa a ser entendida como uma evolução natural das abstrações de projetos, na busca de novas formas de construir sistemas de *software* maiores e mais complexos (SHAW; DELINE; ZELESNIK, 1996, p. 2). Ela pode ser também, a estrutura dos componentes de um programa ou sistema, suas relações, e diretrizes que governam a evolução ao longo do tempo (GARLAN; PERRY, 1995, p. 269).

A Arquitetura de *Software* apresenta uma visão de um sistema de *software* com conectores e componentes, estes que encapsulam algum conjunto coerente de funcionalidades, enquanto os conectores percebem a interação em tempo de execução entre os componentes, o sistema do projeto atinge certas qualidades com base em sua composição de componentes e conectores (ALBIN, 2003, p. 27).

Para esse trabalho, considera-se que Arquitetura de *Software* é a disposição ou organização que se dá em um projeto de desenvolvimento de *software,* considerando estruturação de componentes do projeto, organização e estrutura, ligação entre *packages* e gerenciamento das fases de desenvolvimento assim como padrões internos do projeto.

2.2.1 Estilos de Arquitetura

O estilo de arquitetura é como um modelo para projeto *software* servindo como estrutura guia, detalhando componentes e módulos e as ligações entre eles (PRESSMAN, 2005, p. 256). De forma geral, todo *software* possui um estilo de arquitetura, mesmo que ela não tenha sido definida (SOMMERVILLE, 2011, p. 151).

Um estilo arquitetural expressa o esquema de organização estrutural de um sistema, bem como fornece o conjunto de componentes, suas responsabilidades e formas de interação entre eles, além de estabelecer padrões de uso (BASS; CLEMENTS; KAZMAN, 2003, p. 24). Cada estilo de arquitetura trabalha com

diferentes tipos de atributos. Portanto, para identificar a arquitetura a partir do estilo aplicado, é necessário ter conhecimento de quais propriedades são mais relevantes para a solução e confrontá-las com as propriedades atendidas pelo estilo aplicado (SOMMERVILLE, 2011, p. 152).

Ressalta-se que a arquitetura não se define apenas pela adoção de um ou vários estilos, pode-se dizer que os estilos de arquitetura são etapas úteis que visam definir a arquitetura de *software*, isto é, cada um constitui um conjunto de decisões iniciais de projeto e fornece a base para as próximas etapas e, em decorrência disto, a escolha do estilo deve ser guiada pelas propriedades gerais e específicas exigidas pela aplicação (BASS; CLEMENTS; KAZMAN, 2003, p. 29).

Não é possível representar todas as informações relevantes da arquitetura de um sistema em um único modelo de arquitetura, pois cada modelo mostra uma visão ou perspectiva do sistema. O modelo pode indicar como um sistema é decomposto em módulos, como os processos de tempo de execução interagem ou as maneiras pelas quais os componentes do sistema são distribuídos em uma rede. Todos são úteis em momentos diferentes, portanto, para *design* e documentação, muitas vezes se precisa apresentar várias visualizações da arquitetura do *software* (SOMMERVILLE, 2011, p. 153).

2.2.2 Exemplos de estilos arquiteturais

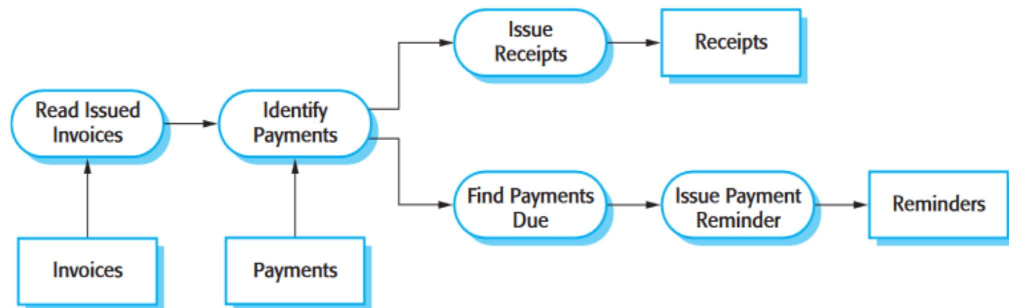
Cada projeto precisa de padrões de arquiteturas diferentes para solucionar e organizar todo o processo de desenvolvimento. Um projeto não precisa aplicar apenas uma arquitetura. Mapear os requisitos e necessidades de um projeto é o que irá guiar qual ou quais estilos entram neste processo (GRÜNBACHER, 2004, p. 236).

Estilos como o *Pipe-and-Filter* são utilizados em sistemas com alto fluxo de dados. Ele envolve interações assíncronas entre componentes que manipulam dados através de conectores. exemplos de projetos que implementam esse padrão com frequência são chamados de compiladores (GALSTER, 2010, p. 350). Esse estilo é adequado para projetos de sistema que requerem estágios de processamento. Cada estágio é implementado como um filtro que recebe e produz dados na saída após realizar algumas transformações (BASS; CLEMENTS; KAZMAN, 2003, p. 23).

A Figura 1 exemplifica esse tipo de arquitetura por meio da representação de uma organização que emite faturas para seus clientes e, uma vez por semana, confirma os pagamentos. Para as faturas que foram pagas, um recibo é emitido, já as

faturas que não foram pagas dentro do prazo de pagamento permitido, um lembrete é emitido.

Figura 1 - Um exemplo da arquitetura *Pipe-and-Filter*

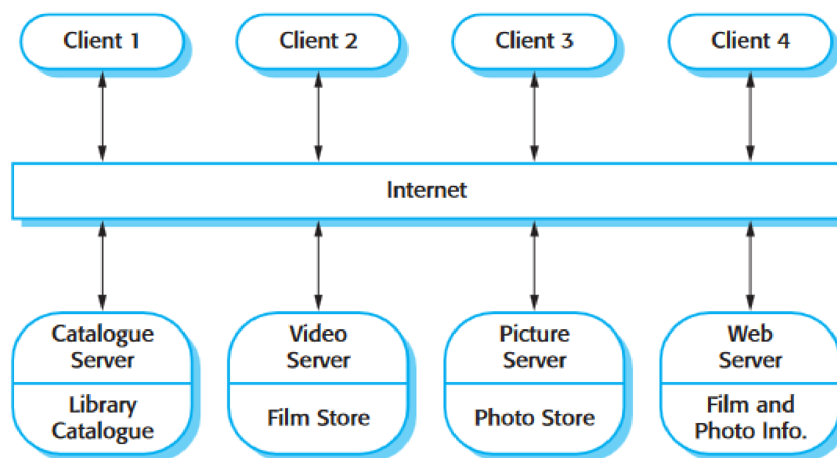


Fonte: SOMMERVILLE, 2011, p. 163

Para sistemas hierárquicos o padrão Cliente-Servidor é muito utilizado, ele envolve a interação síncrona entre clientes que fazem requisições por meio de protocolos para o servidor, o qual é o único elemento que um cliente pode se comunicar, sem ter acesso a outros clientes. Aplicações *web* recorrentemente usam esta arquitetura (OLUWATOSIN, 2014, p. 67).

A Figura 2 exemplifica o conceito da arquitetura Cliente-Servidor em um sistema em que os clientes distintos fazem requisições utilizando o protocolo de Internet para servidores distintos.

Figura 2 - Arquitetura Cliente-Servidor

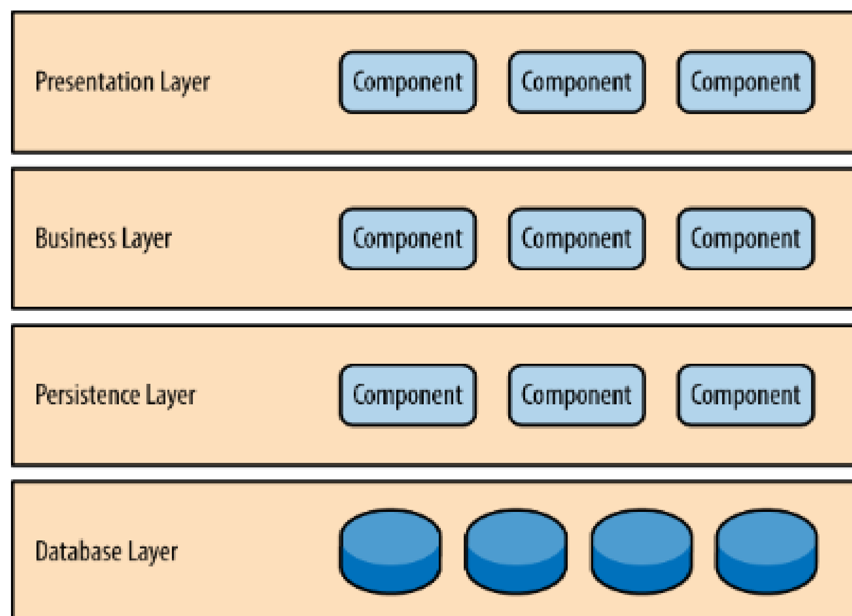


Fonte: SOMMERVILLE, 2011, p.162

A Arquitetura em Camadas, também chamada de arquitetura *N-Tier* e pode ser considerada como a arquitetura mais comum. Sua organização se dá pela separação do sistema em camadas horizontais, tais como, camada lógica, camada de regra de negócio e camada de apresentação. Cada camada tem sua função específica e não precisa se preocupar com a função ou dados de outras camadas. Requisições devem ser feitas apenas para camadas diretamente ligadas (RICHARDS, 2015, p. 2).

A Figura 3 demonstra um sistema que possui o estilo de Arquitetura em Camadas sendo: apresentação, regra de negócio, persistência e banco de dados.

Figura 3 - Padrão de Arquitetura em camadas



Fonte: RICHARDS, 2015, p. 2

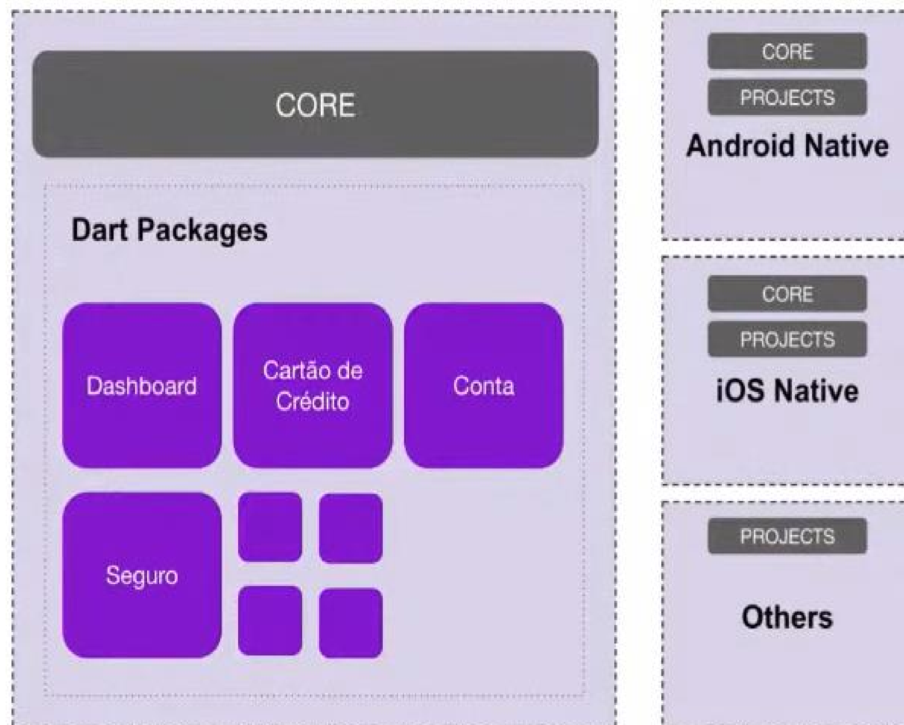
A arquitetura de *Microfrontends*, *Microapps* ou orientada a módulos, surgiu em 2016 como um método alternativo de organização e estrutura, em que o *frontend* (cliente) é seccionado a fim de atribuir maior efetividade e independência entre os times de desenvolvimento, além de isolar os riscos de falhas em áreas menores e fornecer uma maior previsibilidade (GEERS, 2020, p. 12). Ela é mais utilizada quando o aplicativo tem funcionalidades bem definidas e precisa de alta escalabilidade, por exemplo, aplicativos de banco que apresentam funcionalidades de carteira, poupança, pix, cartão de crédito, entre outros (FLUTTERANDO, 2022).

Apesar de ser utilizada no mercado, a arquitetura de *Microfrontends* não possui formalização, sofrendo mudanças de implementação conforme a empresa que a utiliza (GEERS, 2020, p. 15). Alguns exemplos de empresas que utilizam suas

próprias formas de aplicação da arquitetura são: Nubank, Banco Votorantim (BV), Unico e IKEA (FLUTTERANDO, 2022).

O Nubank é uma das maiores *fintechs* brasileiras, e um dos líderes em inovação tecnológica no desenvolvimento móvel (MARQUES, 2018, p. 22). A empresa utiliza uma arquitetura de *microfrontends* representada na Figura 4.

Figura 4 - Arquitetura do *app* do banco Nubank



Fonte: Flutterando (2022)

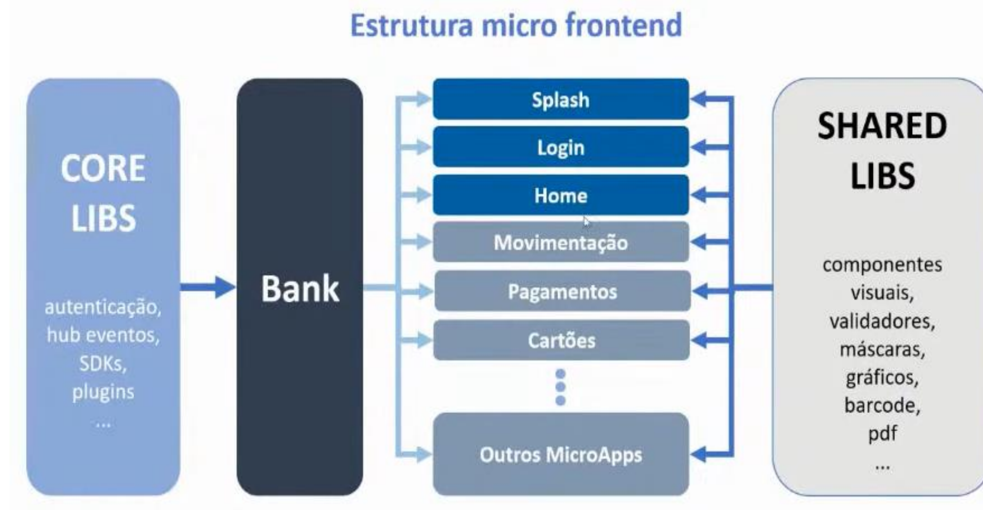
Cada projeto de aplicativo é separado em componente *Core* (1) que faz a gestão de rotas ou navegação entre funcionalidades, estado e dados do *app,* e *Dart packages*, (2) que correspondem as funcionalidades disponíveis para o usuário. A arquitetura utilizada pelo banco também implementa projetos separados para *Android Native, iOS Native* e projetos extras que são utilizados apenas em alguns casos dentro do aplicativo (FLUTTERANDO, 2022).

Dentro da arquitetura aplicada, os *Dart packages* são as funcionalidades, que devem ser independentes, desacopladas e não possuir comunicação ou navegação direta entre elas, isso é, utilizam o componente *Core* para intermediar e validar a navegação entre funcionalidades como, por exemplo, Pix, Poupança, Conta ou Cartão de Crédito.

Por sua vez, o BV (Banco Votorantim S.A.) é um dos maiores bancos privados do Brasil, inserido em atividades como metalurgia, siderurgia, química, petroquímica

e tecnologia (MINELLA, 2007, p. 111). Sua utilização de *microfrontends* é demonstrada na Figura 5.

Figura 5 - Arquitetura do *app* do Banco Votorantim S.A.

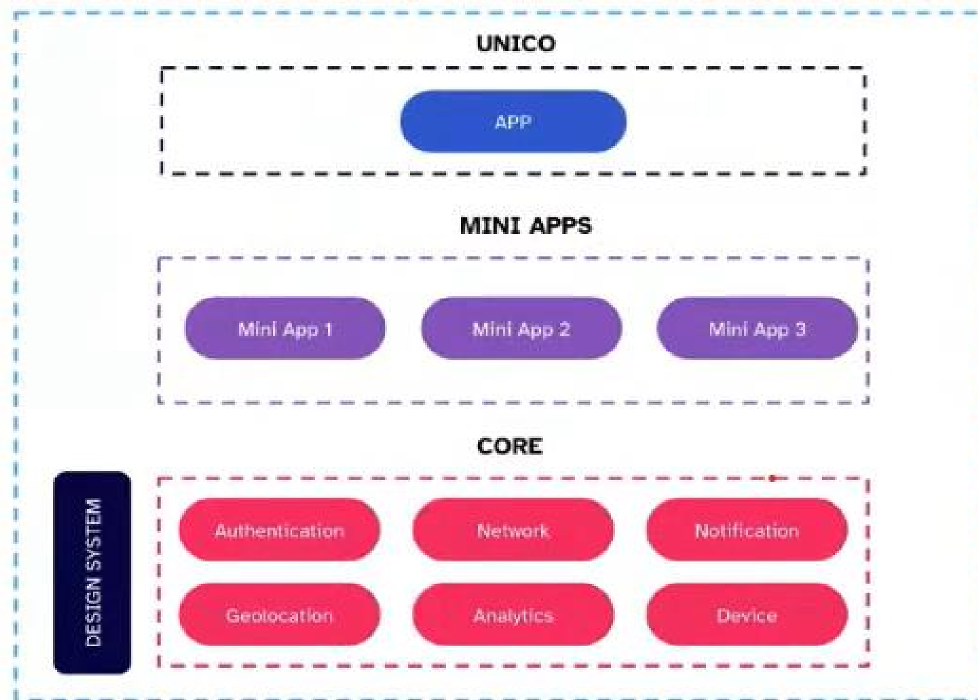


Fonte: Flutterando (2022)

Na implementação do BV, a arquitetura *microfrontends* é utilizada separando o aplicativo em um *Core*, assim como no Nubank, que vai gerenciar todos os eventos, estados, autenticações e dados da aplicação. É implementado um componente *Bank* que contém as funcionalidades independentes como *Splash, Login, Home,* entre outras. E por fim, um único pacote de *Shared Libs* contendo partes do aplicativo que são compartilhadas entre as funcionalidades como os componentes visuais, validades e máscaras, por exemplo. Para o BV, os *MicroApps* seguem o mesmo conceito de desacoplamento apresentado no Nubank. Todas as funcionalidades são independentes e sem comunicação e navegação direta, utilizando o componente *Bank* como intermédio da navegação.

A Unico é uma *startup* e *IDTech* que proporciona identificação digital para pessoas e empresas e tem grande participação no mercado de tecnologia (FLUTTERANDO, 2022). Assim como o Nubank e BV, ela também tem sua própria estrutura para *microfrontends*, observada na Figura 6.

Figura 6 - Arquitetura *microfrontends* do *app* da Unico



Fonte: Flutterando (2022)

A Unico faz a separação da aplicação utilizando 3 tipos de componentes distintos. (1) Um *app* que tem funções de inicialização do aplicativo, assim como manter seu estado, chamado de *Core* no Nubank e BV; (2) componentes *Mini App* que são as funcionalidades dentro da aplicação; e (3) componentes *Core* que contém dados que são utilizados dentro de cada *Mini App* (FLUTTERANDO, 2022).

A Unico, assim como as aplicações apresentadas previamente, também desacopla os *MiniApps*, impossibilitando a comunicação e navegação direta entre as funcionalidade e utilizando o componente *App* para fazer o intermédio de navegação.

Por fim, a IKEA é uma empresa sueca e uma das maiores vendedoras de móveis e decoração do mundo (IKEA, 2022). Sua arquitetura de *microfrontends* foi descrita por Gustaf Nilsson Kotte (líder técnico da equipe de desenvolvimento da empresa), como uma arquitetura pensada para que times de 10 a 12 pessoas trabalhem em funcionalidades diferentes e independentes sem perder eficiência (CONVERSATIONS ABOUT SOFTWARE ENGINEERING, 2018).

Kotte ainda explica que o sistema é dividido por funcionalidades, assim como os sistemas descritos anteriormente, com um componente central que guarda todos dados utilizados e os distribui e um componente responsável exclusivamente pela

comunicação dos dados do *frontend* com o *backend* (CONVERSATIONS ABOUT SOFTWARE ENGINEERING, 2018).

Analisando as empresas apresentadas que utilizam arquiteturas de *microfrontends*, é possível identificar fatores comuns a todas as implementações, tais como: um componente de inicialização, separação das funcionalidades da aplicação em componentes independentes, utilização de centralização de dados.

Dentre as características comuns, o componente de inicialização é visto como *Core* tanto no Nubank quanto no BV enquanto na Unico, é chamado de App e para IKEA, é utilizado o componente de comunicação entre funcionalidades para inicialização. É nele que o aplicativo é iniciado e são mantidas rotas de navegação entre funcionalidades do sistema (FLUTTERANDO, 2022).

Também em todas as implementações citadas de *microfrontends* as funcionalidades do sistema são separadas e independentes, vistas no Nubank divididas em *Dart Packages*, no BV como *MicroApps* e na Unico em *Mini Apps.* As funcionalidades são o conteúdo principal do aplicativo, no caso dos bancos podem ser perfis de usuário, carteiras, poupanças ou pix enquanto no aplicativo da Unico a geração de identidade digital ou validação de uma identidade digital (FLUTTERANDO, 2022).

Por fim, a centralização de dados é feita utilizando um componente separando dados e funções do sistema, deixando essas informações disponíveis para os pacotes de funcionalidades utilizarem aumentando a reutilização de código e facilitando manutenibilidade de código. O componente de centralização de dados é apresentado como *Shared Libs* no BV, *Core* na Unico, *Other* no Nubank e não foi dado um nome para o componente pela IKEA (FLUTTERANDO, 2022).

Entretanto, pela falta de formalização, ainda são apresentadas diferenças significativas nas implementações de cada empresa, o que ocasionam inconsistências nos sistemas e variações no produto final. Tais diferenças podem ser percebidas na falta de nomenclatura padrão para os componentes, forma de centralização e utilização de componentes e navegação entre funcionalidades.

2.2.3 Modalidades arquiteturais

Tendo em vista que estilos de arquitetura podem ser abrangentes e solucionar problemas diferentes, os autores Bass, Clements e Kazman em "*Software Architecture*

in Practice” categorizam os estilos em 3 modalidades: a) Componente-e-conector; b) Alocação; e c) Módulo; (BASS; CLEMENTS; KAZMAN, 2003, p. 24).

a) Componente-e-conector

Os elementos são componentes de tempo de execução (que são as principais unidades de computação) e conectores (que são os veículos de comunicação entre os componentes) (RINGERT, 2014, p. 13).

Os componentes são elementos do modelo que fornecem e encapsulam os serviços fornecidos que são, por exemplo, a computação ou o armazenamento de dados. Os componentes fornecem e requerem esses serviços por meio de interfaces. As estruturas de componentes-e-conectores ajudam a responder perguntas como: “Quais são os principais componentes de execução e como eles interagem?” (RINGERT, 2014, p. 14).

b) Alocação

As estruturas de alocação descrevem o mapeamento de estruturas de *software* para os ambientes do sistema, também mostram a relação entre os elementos do *software* e os elementos em um ou mais ambientes externos, nos quais o *software* é criado e executado (BASS; CLEMENTS; KAZMAN, 2003, p. 36).

c) Módulo

Os elementos do projeto são vistos como módulos, que são unidades de implementação. Cada módulo possui uma responsabilidade funcional e pode ser independente de outros módulos. Há menos ênfase em como o *software* resultante se manifesta em tempo de execução (BASS; CLEMENTS; KAZMAN, 2003, p. 36).

A característica modular pode ser vista como ou a atribuição de responsabilidade às partes de um sistema, ou a separação de partes em subsistemas. Essa forma arquitetural traz vantagens como a diminuição no tempo de desenvolvimento, possibilitando a divisão de times em módulos diferentes, maior flexibilidade, mudanças em módulos de forma independente e, por fim, compreensibilidade, pois o sistema pode ser estudado um módulo por vez (PARNAS, 1972, p. 1054).

Contemporaneamente, a utilização de arquiteturas “Módulos” tem aumentado com a introdução do conceito de *Plug and Produce*, que definido é como arquiteturas que tentam facilitar a instalação ou remoção de um recurso ou módulo de um sistema em produção sem qualquer mudança na configuração ou reprogramação, pela indústria 4.0 (HOMAY *et al.,* 2019, p. 1165).

A partir da compreensão da modalidade Módulo, a próxima seção aborda arquiteturas modulares nas quais o presente estudo se baseia. Porém, antes de abordar a arquitetura proposta, é necessária a apresentação do ambiente de desenvolvimento *Flutter* para maior compreensão do estudo apresentado.

3 *FLUTTER* ENQUANTO KIT DE DESENVOLVIMENTO

Flutter é um *software development kit* (SDK), isto é, um pacote de desenvolvimento de código aberto, mantido e construído em 2015 pela empresa Google que tem como foco capacitar desenvolvedores a programarem aplicativos *mobile* (WINDMILL, 2020, p. 4).

A linguagem implementada pelo SDK é denomina-se *Dart*, e é desenvolvida pelo Google em 2011, trata-se de uma linguagem de programação usada para desenvolver aplicações *web, desktop, server side* e *mobile* (BIESSEK, 2019, p. 7). A linguagem *Dart a*dota padrões orientados a objetos, e é *client-optimized,* ou seja, focada no desenvolvimento *frontend* (MIOLA, 2020, p. 20).

O *kit Flutter* apresenta todas as necessidades básicas para o desenvolvimento de uma aplicação como uma *engine* de renderização, componentes de interface, estrutura de testes, ferramentas de rotas, entre outras, além da facilidade na criação de projetos podendo utilizar um único comando *“flutter create”* (WINDMILL, 2020, p. 8).

Alguns dos principais elementos utilizados no *Flutter* para o desenvolvimento de aplicativos são as *widgets* e *packages*. Portanto, para se efetuar a análise proposta neste trabalho, cabe o estudo destes elementos, eis que delinearão pressupostos, requisitos e eventuais elementos acidentais deste.

3.1 Principais elementos

3.1.1 Widget

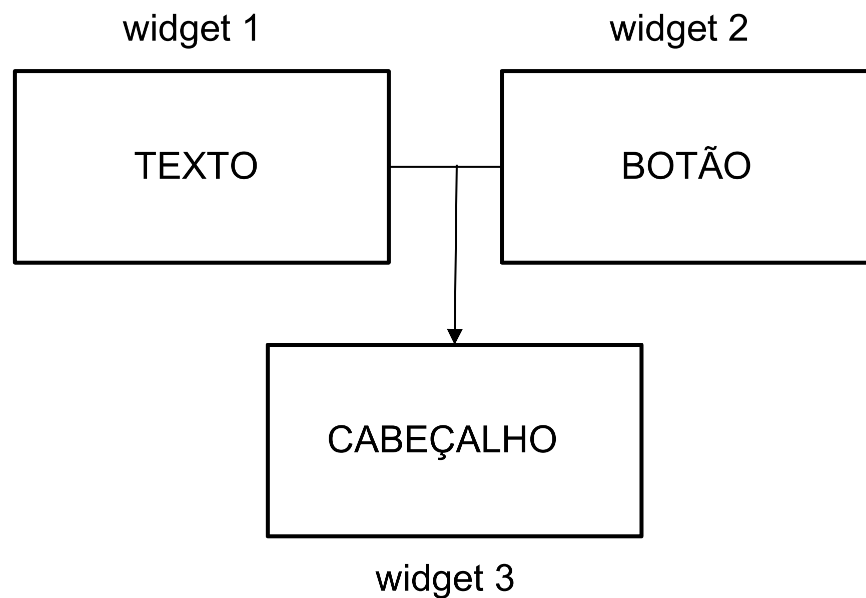
Segundo o autor Payne em seu livro *“Beginning App Development with Flutter-Apress”* publicado em 2019, *widgets* são como peças de Lego, pequenos componentes que quando agrupados podem formar novas estruturas. O autor exemplifica o desenvolvimento de aplicativos como a construção de uma personagem de quadrinhos em tamanho real com peças de Lego. Componentizando cada pequena parte como pés, mãos, cabeça, braços e juntando suas partes posteriormente (PAYNE, 2019, p. 20).

Widgets podem ser entendidas como a representação das partes da aplicação. Várias *widgets* juntas compõem uma interface para o usuário, da mesma forma que diferentes peças formam um quebra-cabeça. A intenção das *widgets* é

montar uma aplicação modular e escalável com o mínimo de código, sem impor limitações.

Para a construção de uma interface de usuário *mobile* no *Flutter*, são usadas *widgets*, as quais são classes *Dart*, descritas como interface que podem se agregar formando novas *widgets* (WINDMILL, 2020, p. 23). A Figura 7 exemplifica a criação de uma nova *widget* “Cabeçalho" sendo criada a partir das *widgets* de "Botão” e "Campo de Texto”.

Figura 7 - Construção de interface a partir de *widget*



Fonte: Autoria própria (2022).

Esse agrupamento de *widgets,* representado na Figura 7, cria uma árvore de *widgets* e, por meio desta, o *framework* realiza uma interpretação para gerar a interface em tela. Quando o estado de uma *widget* é alterado, o *framework* compara essa nova árvore de *widgets* com a anterior e modifica apenas a *widget* necessária (WINDMILL, 2020, p. 57).

Widgets podem ser separadas na forma que lidam com o estado da aplicação, tendo ou não a necessidade de uma nova renderização da tela a partir da mudança de um dado (PAYNE, 2019, p. 35). Estado é informação que pode ser lida de forma síncrona quando a *widget* é criada e pode mudar durante seu ciclo de vida (FERRERO, 2022, p. 16).

A interface do usuário quase nunca é estática, ela muda com frequência. Embora imutáveis por definição, as *widgets* não devem ser definitivas, visto que, se trata da interface do usuário que certamente mudará durante o ciclo de vida de um

aplicativo e, nesse contexto, o *Flutter* fornece dois tipos de *widgets*: *stateless e stateful* (BIESSEK, 2019, p. 122)

Stateless widgets não dependem dos dados, mas em apresentar informação para a interface gráfica. Cabe ao *framework* decidir quando e como renderizá-lo, caso precise ser atualizado, a *widget* será destruída e reconstruída, perdendo os dados antigos e renderizando apenas os novos (WINDMILL, 2020, p. 60).

Stateful widgets dependem dos dados (que não podem ser perdidos caso sofra uma modificação) e da apresentação de informação na tela. Como por exemplo: em um aplicativo de compras, o contador de itens do carrinho precisa manter a quantidade de itens adicionados e atualizar o contador caso algum item seja adicionado ou removido, não podendo perder a quantidade de itens mesmo que outra parte do aplicativo necessite de uma renderização (WINDMILL, 2020, p. 61).

3.1.2 Packages

Packages são entendidos como ferramentas escritas por outros programadores que podem ser usadas no seu código para deixar o desenvolvimento mais rápido e produtivo (PAYNE, 2019, p. 20). Um *package Dart* é um diretório que contém um arquivo *pubspec*. Podem ocorrer dependências dentro do *pubspec*, bibliotecas *Dart*, aplicativos, recursos, testes, imagens e exemplos. *Packages* podem ser acessados e gerenciados pela plataforma *Pub* no *site* https://pub.dev/. A plataforma gerencia *packages* enviados por desenvolvedores do mundo todo (WINDMILL, 2020, p. 318).

Em suma, entende-se que *packages* são aplicações que podem ser adicionadas a um aplicativo para facilitar o desenvolvimento, criar funcionalidades ou resolver problemas sem que seja necessário o desenvolvimento de todas as partes por um mesmo programador, o que possibilita a reutilização e componentização. Por exemplo, um aplicativo que use a câmera do celular, o programador não precisaria desenvolver o código para acesso da câmera ou tratamento da imagem, e sim utilizar um *package* disponível na plataforma *Pub* que já tenha implementado tais funcionalidades.

Todo *package* possui um arquivo *pubspec.yaml* que contém configurações como, por exemplo, suas dependências, importações e em que plataforma de *packages* ele será publicado. É utilizado o comando "*flutter create –template=package*" para criar *packages* em um projeto.

3.2 Benefícios do SDK *Flutter*

O *Flutter* compila direto para código nativo e usa a *engine Skia* para renderizar componentes em tela. O aplicativo não perde performance em comparação com outros *apps* desenvolvidos diretamente em linguagem nativa, pois é como se ele estivesse executando na linguagem nativa sem ser convertido do *Dart* (WINDMILL, 2020, p. 10).

O *Flutter* permite que desenvolvedores escrevam aplicativos *Android, iOS, web e desktop* com uma única base de código, mantendo as performances e aspectos em cada plataforma. Uma linguagem altamente produtiva, como o *Dart* acelera o processo de codificação e torna a estrutura agradável (MIOLA, 2020, p. 28). A plataforma dá liberdade ao desenvolvedor para criar seus próprios botões e funcionalidades, não forçando que o desenvolvimento fique preso à padrões *Android* ou *iOS* (BANDEIRA, 2022).

Tal linguagem combinada com a alta performance e produtividade facilita a testagem, trazendo a necessidade de que os testes sejam escritos apenas uma vez, disponibilizando 3 tipos de testes.

O Teste Unitário (1), que pode ser descrito como testes menores e mais rápidos que focam em conferir a funcionalidade de uma classe ou função sem estarem ligados a *widgets* (WINDMILL, 2020, p. 293); Testes de *Widget* (2) que são utilizados para verificar se uma única *widget* é utilizada e interage com outras *widgets* da forma esperada; e Testes de Integração (3) que emulam a utilização do usuário em tela e, portanto, precisam de um sistema funcionando com interface. Estes testes são utilizados para verificar se todas as *widgets* e serviços disponíveis no aplicativo funcionam e estão integrados de forma correta, além de prover *feedback* de performance (WINDMILL, 2020, p. 295).

A característica híbrida também é uma vantagem, pois possibilita que um aplicativo seja desenvolvido independente do sistema operacional, sem que seja necessário o conhecimento de múltiplas linguagens ou, no caso de uma empresa, múltiplos times para cada sistema operacional (WINDMILL, 2020, p. 22).

Por fim, uma de suas principais vantagens é a facilidade e velocidade de desenvolvimento. Eric Seidel, gerente de projeto do *Flutter* no Google, no evento *Dart Developer Summit* realizado em 2016, apresentou um aplicativo desenvolvido

utilizando o *Flutter,* o tempo de desenvolvimento observado foi 3 vezes mais rápido que com plataformas nativas para *Android e iOS* (KEYNOTE, 2016).

Dessa forma, é possível perceber que a plataforma escolhida possui alta aceitabilidade no meio, além de estar aberta para o desenvolvimento de padrões de projetos, assim como, o desenvolvimento deste trabalho.

4 ARQUITETURA ORIENTADA A *PACKAGES*

Como apresentado previamente, uma Arquitetura de *Software* pode ser entendida como a abstração de projetos, assim como, a forma de organização e construção de um sistema considerando as necessidades e requisitos apresentados (SHAW; DELINE; ZELESNIK, 1996, p. 2).

Também foi apresentada a definição de arquitetura, que é utilizada neste trabalho como a disposição ou organização em um projeto de desenvolvimento de *software,* considerando a estruturação de componentes do projeto, a organização e estrutura, ligação entre *packages* e o gerenciamento das fases de desenvolvimento, assim como, os padrões internos do projeto.

Neste capítulo, é abordada a proposta da Arquitetura Orientada a *Packages*, demonstrando seus princípios, em que ela se baseia, a organização do sistema, como a arquitetura pode diminuir o número de testes executados em um aplicativo e suas vantagens e desvantagens.

4.1 Definição da Arquitetura Orientada a *Packages*

A Arquitetura Orientada a *Packages* pode ser definida como a formalização de uma arquitetura de *microfrontends* utilizando o ambiente de desenvolvimento *Flutter*. Cada funcionalidade do sistema é tratada como um aplicativo independente e desacoplado que pode ser inserido ou removido do sistema, sem quebrar o funcionamento da aplicação como um todo. Esse desacoplamento das funcionalidades faz com que a Arquitetura Orientada a *Packages* seja modular, escalável e proporcione uma divisão clara de cada parte da aplicação, assim como *packages* independentes.

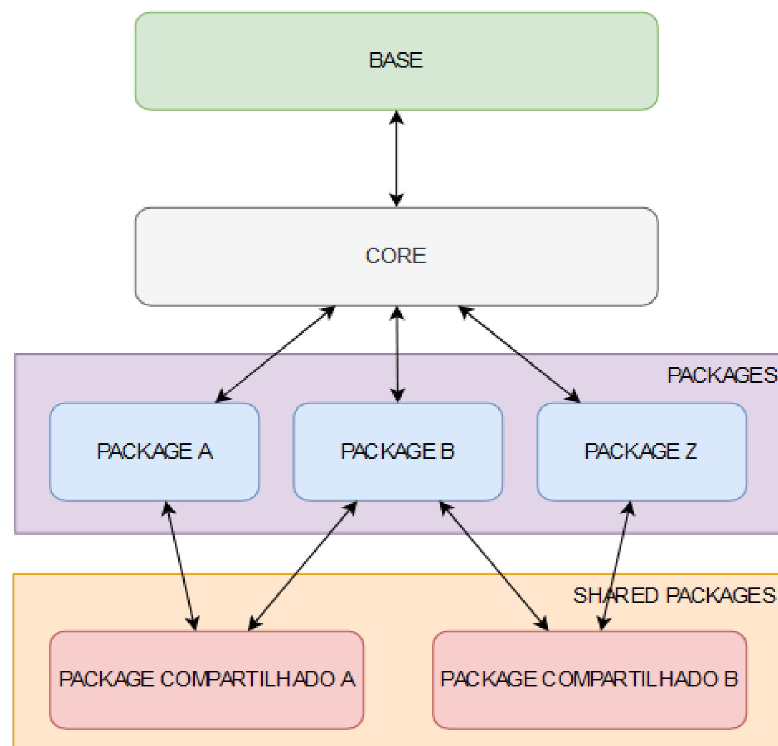
4.2 Organização da arquitetura

Arquitetura Orientada a *Packages* pode ser descrita a partir de como se organizam os componentes dentro de um sistema. Cada componente do sistema é uma camada independente e desempenha funções definidas como, por exemplo, manter a lista de possíveis páginas do sistema, ou tratar as requisições de mudança de página.

A Figura 8 apresenta a organização dessa arquitetura e a divisão dos componentes mencionados em: *Base* (4.2.1); *Core* (4.2.2); *Packages* (4.2.3); e, *Shared Packages* (4.2.4).

Pode-se fazer uma comparação da Arquitetura em Camadas com a Arquitetura Orientada a *Packages,* de forma que cada componente pode ser visto como uma camada, porém, elas se diferenciam na função que a camada tem sobre a aplicação, nos protocolos de comunicação e na forma de comunicação entre camadas. Cada componente da Arquitetura Orientada a *Packages* tem funcionalidade própria e poderia ser vista como um aplicativo em si.

Figura 8 - Arquitetura Orientada a *Packages*



Fonte: Autoria própria (2022).

4.2.1 *Base*

A *Base* na arquitetura irá conter todos os processos que precisam ser feitos previamente à execução da aplicação como, por exemplo, a determinação de variáveis de ambiente e importações de *packages* necessários para a comunicação com a camada de *Core,* importação de todos os *Packages* como dependências para que haja navegação, e o mais importante, a definição das rotas da aplicação, que são quais as páginas ou telas do aplicativo.

O componente *Base* tende a ser o mais simples na implementação e um dos mais importantes do sistema, pois serve como uma porta de entrada, é o ambiente em que a aplicação irá ser apresentada e irá pegar os dados iniciais, caso eles existam. Porém, ela não irá tratá-los ou manter nenhum estado da aplicação, pois seu foco é iniciar o aplicativo e manter a lista de rotas.

4.2.2 *Core*

O componente *Core* funciona como a espinha dorsal da aplicação e um intermediário entre a inicialização feita no componente *Base* e as funcionalidades que estão nos *packages*. Neste componente, é criada a lógica de navegação do aplicativo, a geração das rotas que serão listadas no componente *Base* e os contratos que todos os *packages* terão que cumprir para serem executados de forma correta.

Como dito anteriormente, *packages* não podem ter ligações diretas entre si, então para que um *package* consiga enviar e receber dados de outro *package*, ele irá transferir o pedido de navegação para o *Core,* que por sua vez, utilizando a lista de rotas da *Base*, irá direcionar a comunicação para o *package,* caso ele esteja presente na lista de rotas.

O *Core* também é o responsável por definir o contrato que todos os *packages* devem seguir para serem vistos como válidos, seja por meio de uma classe que poderá ser implementada dentro do *package* ou por um algoritmo de validação que o *package* deve passar.

O conceito de separar uma aplicação maior em aplicações menores não é novo, e é observado em arquiteturas de *microfrontends* ou Arquiteturas Orientadas A Serviços.

4.2.3 *Packages*

Os *packages* são a principal característica desta arquitetura, eles são *packages Dart* do *Flutter* e atuam na abstração de uma aplicação ou necessidade que o sistema prevê e devem sempre ser independentes.

Um *package* implementado é uma funcionalidade que o sistema tem de forma independente. Isso quer dizer que um sistema que possua *login*, câmera, *dashboard* e listagem de itens, deve possuir no mínimo 4 *packages* que representam cada uma das funcionalidades, cada um desses *packages* pode ser reutilizado em outro sistema

que precise da mesma funcionalidade. Um *package* na arquitetura proposta é criado utilizando o comando de criação de *packages* no *Flutter* "*flutter create --template=package*".

Eles não podem ter conexão direta com outros *packages.* Para sair de uma funcionalidade e entrar em outra, o sistema deve fazer um pedido de navegação ao componente *Core* que irá redirecionar a aplicação para a funcionalidade desejada.

4.2.4 *Shared Packages*

Shared Packages seguem a mesma linha de raciocínio dos *packages*, são *packages Flutter* independentes, porém, com a diferença que não são a implementação de uma função do sistema, mas a implementação de características compartilhadas entre *packages*. Isso quer dizer que um *Shared Package* pode conter, por exemplo, *Widgets* que são utilizadas em diferentes funcionalidades, como botões ou caixas de texto e dados de usuário, que podem estar presentes em uma funcionalidade de *dashboard* ou em um perfil de usuário.

Estes componentes são utilizados para aumentar o reuso de código no sistema e diminuir consideravelmente o código que é incluído em *packages* com pouca ou nenhuma modificação, transferindo o que é compartilhado entre um ou mais *packages* para um *package* separado e comum entre eles, o que basta para que haja a importação desse *shared package* por um *package*.

4.3 Vantagens e desvantagens

Para apresentar a lista de vantagens e desvantagens da utilização da Arquitetura Orientada a *Packages*, foi desenvolvida a Tabela 1, seus itens serão descritos na próxima seção.

Tabela 1 - Vantagens e Desvantagens da Arquitetura Orientada a *Packages*

VANTAGENS	DESVANTAGENS
Alta escalabilidade, modularidade e independência.	Complexidade de implementação.
Divisão bem definida de partes do sistema.	Arquitetura não generalista (pensada para *Flutter*).
Fácil manutenibilidade de *packages* externos.	Perda do estado entre *packages.*

Fonte: Autoria própria (2022).

4.3.1 Vantagens

Assim como a Arquitetura Orientada a Serviços, a Arquitetura Orientada a *Packages*, apresenta algumas de suas vantagens como a escalabilidade, modularidade e independência (ALSHUQAYRAN; ALI; EVANS, 2016, p. 48).

A escalabilidade é a possibilidade do sistema de crescer, atendendo demandas sem perder qualidade, já a modularidade é a capacidade do sistema de reutilizar funcionalidades, por fim, independência é a característica que implica em uma parte do sistema que não depende de dados de outra parte (ALSHUQAYRAN; ALI; EVANS, 2016, p. 48). Tais características são complementares, pois a independência dos *packages* possibilita que apenas cumprindo o contrato montado na camada *Core,* um *package* possa ser validado e acoplado ao sistema mantendo a característica de escalar.

Cada funcionalidade do aplicativo fica bem definida dentro de um *package,* assim como em arquiteturas de *microfrontends*. Esta divisão do aplicativo possibilita que diferentes times de desenvolvimento, trabalhem de forma independente em funcionalidades distintas (FLUTTERANDO, 2022). A possibilidade de fácil divisão de times de desenvolvimento é uma grande vantagem para empresas, pois diminui o escopo necessário para compreensão do sistema em cada funcionalidade (FLUTTERANDO, 2021).

A modularização também possibilita que testes do aplicativo sejam realizados nos *packages* e suas dependências, não sendo necessária uma bateria única de testes para todo aplicativo.

A utilização dos *shared packages* soluciona um problema que ocorre em arquiteturas de *microfrontends* que não utilizam componentes compartilhados, o versionamento de *packages* externos.

Dart packages externos presentes na plataforma *Pub*[7] importados para o projeto podem dispor de diferentes versões. Caso mais de um *package* necessite de *packages* externos, estes devem ser importados em um *shared package*, que é utilizado como dependência nos *packages*, mantendo assim uma única versão do *package* importado. Tal característica é útil para que não existam diferentes versões

[7] Disponível em: https://pub.dev/

de mesmos *packages* externos ou que fiquem defasados ou com *bugs* por falta de atualizações.

4.3.2 Desvantagens

Com necessidade de protocolos rígidos de comunicação entre *packages* e implementações separadas de cada funcionalidade presente no sistema, a Arquitetura Orientada a *Packages* não se mostra adequada para projetos menores devido à grande complexidade de implantação.

Assim, a arquitetura proposta é mais adequada para projetos de desenvolvimento de maior complexidade, que ganharão tempo com a reusabilidade constante de *packages* e com a divisão bem definida de cada parte do aplicativo.

A Arquitetura Orientada a *Packages* é uma arquitetura pensada e implementada visando a plataforma de desenvolvimento *Flutter,* portanto, ela ainda não possui validação fora do escopo apresentado no *kit* e, por isso, mudanças na organização e padrões podem ser necessárias para que ela sirva para outras linguagens.

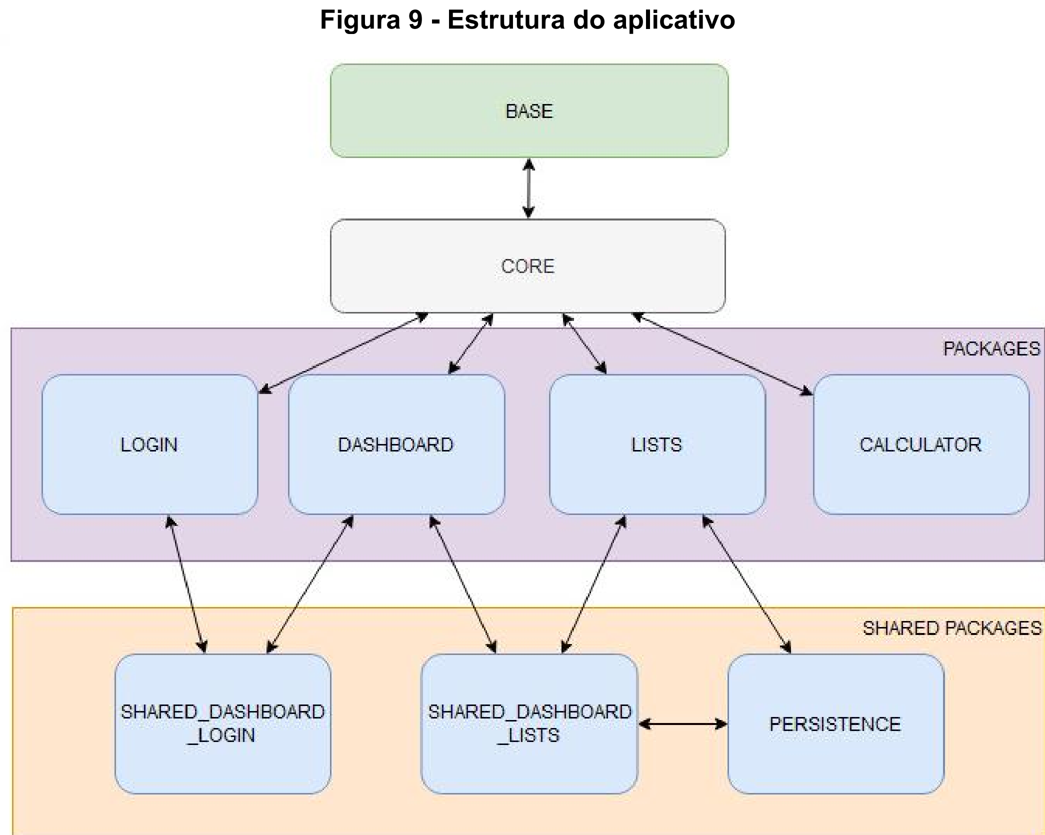
A principal desvantagem da utilização de *Packages* independentes que não possuem comunicação direta, é a necessidade de remontagem de tela em caso de atualização de dados que são compartilhados entre funcionalidades. Caso um *package* A, contendo um saldo de conta corrente faça uma requisição de navegação para um *package* B que modifique esse saldo, ao fazer a navegação de volta para o *package* A, a tela deve ser remontada ou os dados estarão desatualizados.

Diante das vantagens e desvantagens apresentadas, percebe-se que a arquitetura proposta deve ser utilizada em sistemas *Flutter* com maior pluralidade de funcionalidades, com múltiplos times de desenvolvimento ou que possuam escalabilidade, modularidade e independência. Caso a arquitetura seja aplicada em sistemas que não apresentem tais características, ela será deficiente.

5 APLICATIVO DESENVOLVIDO

Para a validação da arquitetura proposta, foi desenvolvido um aplicativo de listas seguindo as especificações do capítulo anterior. A Figura 9 apresenta a estrutura do aplicativo. Cada componente da Figura 9 corresponde a um componente de mesma cor na Figura 8.

Figura 9 - Estrutura do aplicativo



Fonte: Autoria própria (2022)

Este capítulo discorre sobre as funcionalidades implementadas e o processo de desenvolvimento do aplicativo. O código completo do aplicativo pode ser encontrado no repositório GitHub no link https://github.com/CaioBal/*package*oriented.

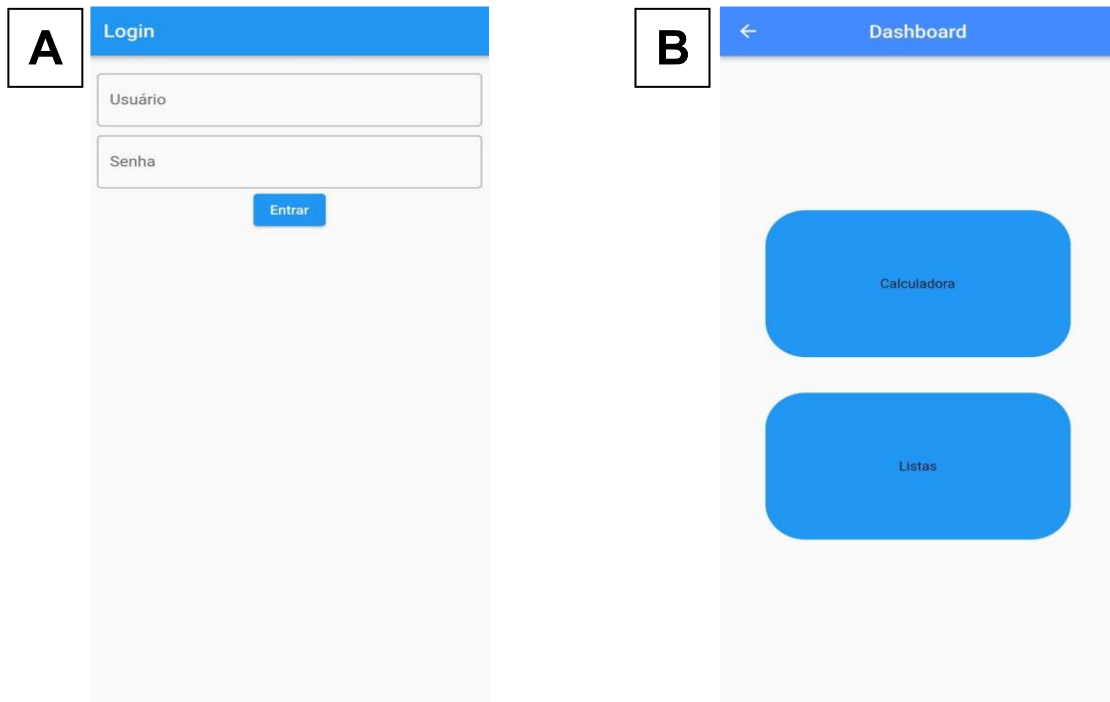
5.1 Apresentação do aplicativo

O aplicativo apresenta 4 *Microapps* ou funcionalidades no total, sendo elas: *Login*, *Dashboard*, Calculadora e Listas.

Ao iniciar o aplicativo, o usuário se encontra na tela de *Login*, representada na Figura 10A. Esta possui duas entradas de dados (para usuário e senha) e um botão que ao ser acionado inicia uma validação e avança para a próxima tela. A segunda

tela do aplicativo é o *Dashboard*, Figura 10B. Nela são dispostos dois *cards,* que ao serem acionados levam para funcionalidades diferentes. O primeiro *card*, nomeado como “Calculadora”, leva o usuário para uma calculadora. O segundo *card,* de nome “Listas”, leva para as listas do usuário.

Figura 10 – Duas telas do aplicativo desenvolvido, sendo: A) Tela de *login*; e B) Tela de *Dashboard*



Fonte: Autoria própria (2022).

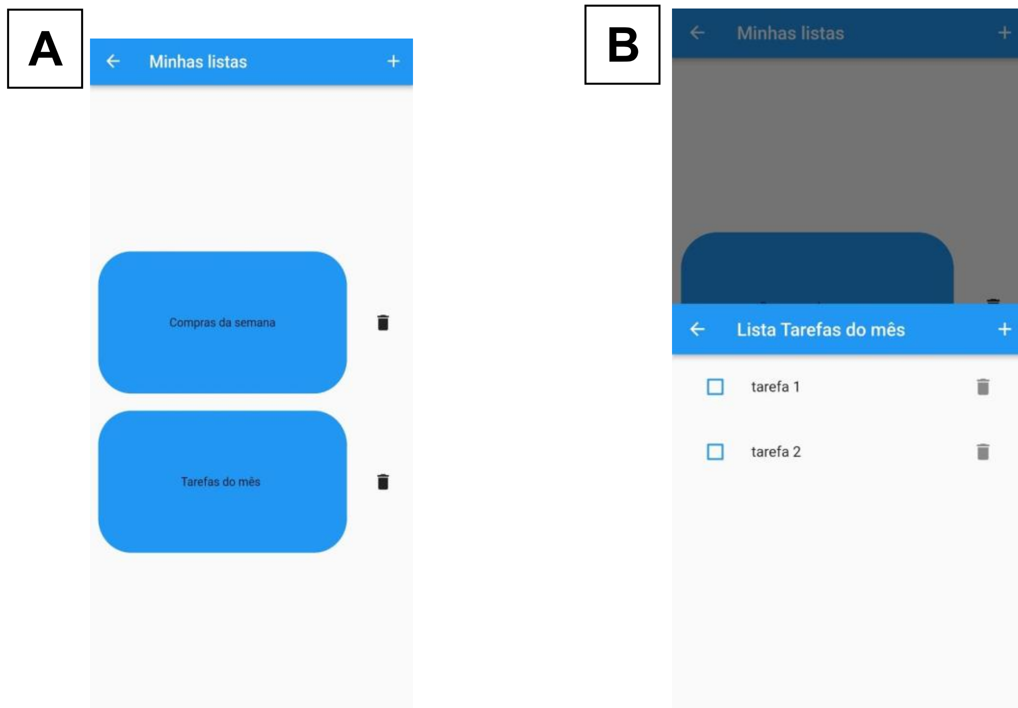
A calculadora, observada na Figura 11, apresenta as operações matemáticas básicas como soma, subtração, multiplicação e divisão. A funcionalidade foi adicionada para facilitar, caso o usuário necessite fazer contas com algum item das listas.

Figura 11 - Calculadora

Fonte: Autoria própria (2022)

As listas, Figura 12A, podem ser adicionadas utilizando o ícone em forma de “+” no topo direito da tela e removidas no ícone de lixeira encontrado a direta do *card*. Cada uma das listas é independente e não dividem itens entre si. A Figura 12B mostra que ao selecionar um *card*, seus itens ficam visíveis. Os itens podem ser adicionados à lista utilizando o ícone em forma de “+” e removidos no ícone de lixeira.

Figura 12 - Figura representativa da tela de Listas; sendo: A) a tela de Listas; e B) mostra a seleção de *card* e os itens visíveis



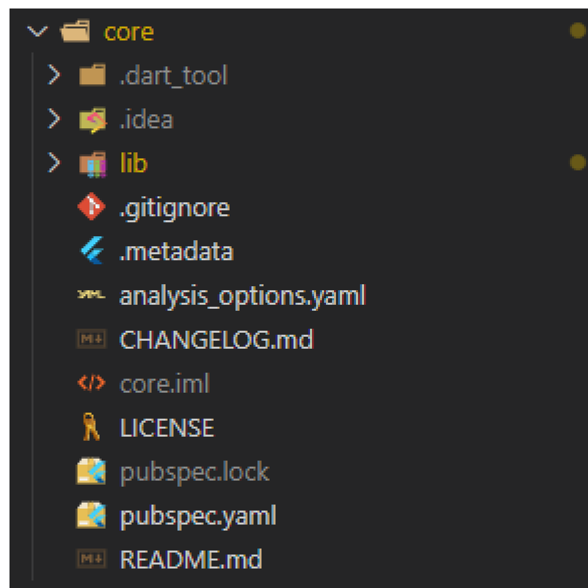
Fonte: Autoria própria (2022)

5.2 Desenvolvimento, estrutura e código

Esta seção aborda a estrutura das pastas do sistema, demonstrando como a organização do aplicativo implementa a Arquitetura Orientada a *Packages* e como cada parte do desenvolvimento do aplicativo foi executada.

Para que a estrutura principal de arquivos do *Flutter* fosse criada, foi utilizado o comando “*flutter create base*”. Este comando gera a estrutura necessária para que uma aplicação possa funcionar, além de uma função *main* que é por onde o aplicativo se inicia. Os outros componentes presentes no aplicativo foram gerados utilizando o comando de criação de *Dart packages,* isto é, “*flutter create --template=package package_name*”. A Figura 13 apresenta os arquivos gerados pelo comando, dentre os arquivos, a pasta *lib*, e o arquivo *pubspec.yaml* vão ser modificados durante o desenvolvimento.

Figura 13 - Arquivos gerados pelo comando



Fonte Autoria própria (2022)

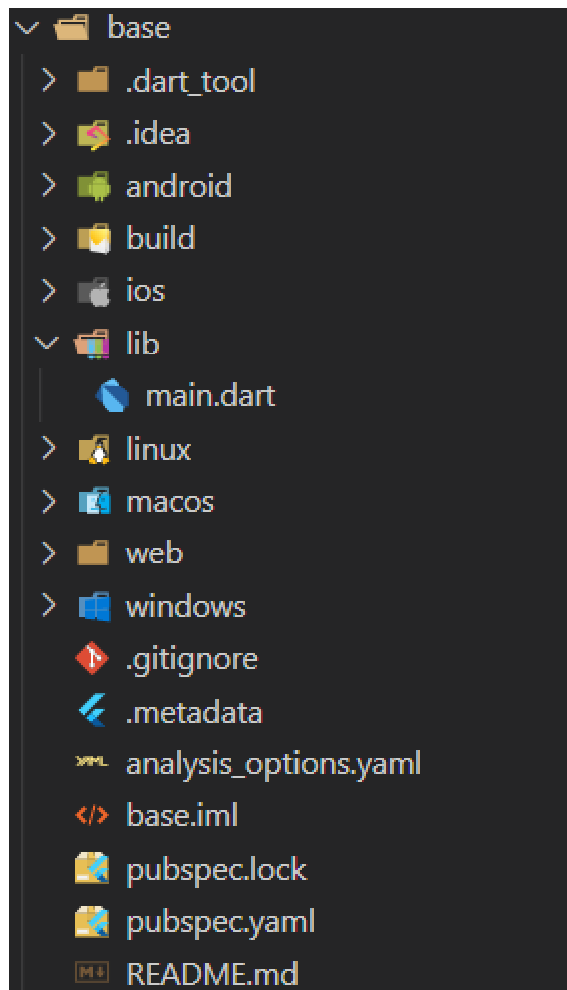
A pasta *lib* mantém a implementação do código de cada funcionalidade. Já o arquivo *pubspec.yaml* é responsável por manter informações, como o nome de um *package,* suas dependências e importações. Ambos serão explicados separadamente posteriormente.

O sistema, assim como a arquitetura, está dividido em 4 partes: Base (5.2.1); *Core* (5.2.2); *Packages* (5.2.3); e *Shared Packages* (5.2.4);

5.2.1 *Base*

O *package base* implementa o componente descrito na arquitetura como *Base*. Sua estrutura pode ser vista na Figura 14. No arquivo *pubspec.yaml* são importados outros *packages* como dependências, por exemplo, pois como foi descrito na arquitetura, é o componente *Base* que irá tratar a lista de navegação de uma aplicação, portanto, é necessário que a *Base* consiga acessar tais *packages*.

Figura 14 - Estrutura da Base



Fonte: Autoria própria (2022)

A pasta *lib* contém um único arquivo *main.dart*, e na Figura 15 é possível observar parte do código desse arquivo. A classe *MyApp* é responsável por fazer as configurações do aplicativo e iniciá-lo. Ela estende as classes *StatelessWidget* e Base, que será apresentada no item seguinte, o *Core.* Das linhas 17 a 32 do código, são feitas configurações iniciais do aplicativo como, por exemplo, título, escolha de temas e cores, geração de rotas e definição inicial das rotas. A partir da linha 33 são

instanciadas variáveis da classe pai, *Base*, referentes ao contrato do aplicativo e as rotas que são usadas pelo aplicativo que faz navegação por rotas nomeadas.

Figura 15 - Código da função *main*

```
13  void main() {
14    runApp(MyApp());
15  }
16
17  class MyApp extends StatelessWidget with Base {
18    @override
19    Widget build(BuildContext context) {
20      super.registerRouters();
21
22      return MaterialApp(
23        title: 'Flutter TCC',
24        theme: ThemeData(
25          primarySwatch: Colors.blue,
26        ),
27        debugShowCheckedModeBanner: false,
28        navigatorKey: navigatorKey,
29        onGenerateRoute: super.generateRoute,
30        initialRoute: '/login',
31      );
32    }
33
34    @override
35    List<AppContract> get appContracts =>
36        [LoginRouter(), DashboardRouter(), CalculatorRouter(), ListsRouter()];
37
38    @override
39    Map<String, Widget Function(BuildContext context, Object? args)>
40        get baseRoutes => {};
41  }
42
```

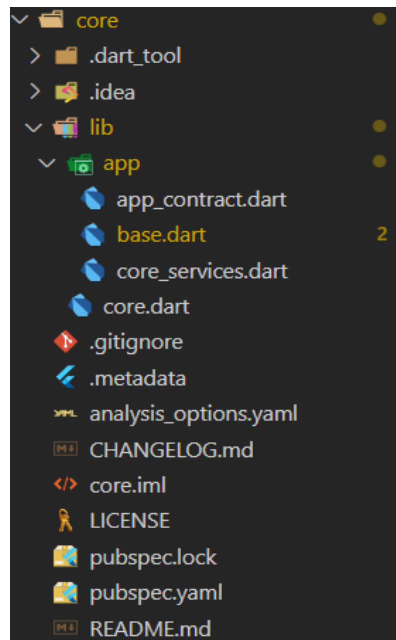
Fonte: Autoria própria (2022)

5.2.2 *Core*

O *package Core* implementa o componente descrito na arquitetura como *Core*. Sua estrutura pode ser vista na Figura 16. No aplicativo, o *Core* não possui nenhuma dependência de outros *packages* portanto o arquivo *pubspec.yaml* não foi modificado.

Já a pasta *lib* contém uma pasta *app* e um arquivo *core.dart* que faz a exportação das funções implementadas no *package*. Dentro da pasta *app*, o arquivo *core_services.dart* cria a variável chamada *navigatorKey,* que é responsável pela navegação no aplicativo e será utilizada no *package base.*

Figura 16 - Pasta *core*



Fonte: Autoria própria (2022)

Ainda na pasta *app, o* arquivo *app_contract.dart* contém a criação de uma classe abstrata *AppContract* que será o contrato que todos *packages* devem implementar para serem validados no sistema, nesta aplicação, ela exige um nome para o *package* e sua rota. Em "*base.dart*" está a criação da classe abstrata *Base*, observada na Figura 17, que é implementada na função *main* do *package* base.

Figura 17 - Captura da função *main*

```
5   abstract class Base {
6     List<AppContract> get appContracts;
7
8     Map<String, Widget Function(BuildContext context, Object? args)>
9         get baseRoutes;
10
11    final Map<String, Widget Function(BuildContext context, Object? args)>
12        routes = {};
13
14    void registerRouters() {
15      if (baseRoutes?.isNotEmpty ?? false) routes.addAll(baseRoutes);
16      if (appContracts?.isNotEmpty ?? false) {
17        for (AppContract appContract in appContracts) {
18          routes.addAll(appContract.routes);
19        }
20      }
21    }
22
23    Route<dynamic>? generateRoute(RouteSettings settings) {
24      var routerName = settings.name;
25      var routerArgs = settings.arguments;
26
27      var navigateTo = routes[routerName];
28      if (navigateTo == null) return null;
29
30      return MaterialPageRoute(
31        builder: (context) => navigateTo.call(context, routerArgs),
32      );
33    }
34  }
```

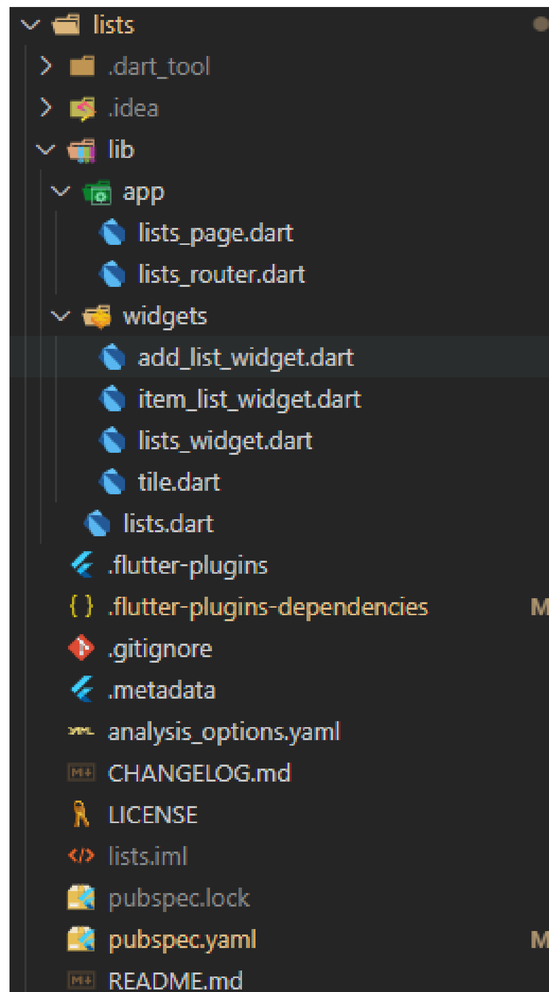
Fonte: Autoria própria (2022)

Das Linhas 6 a 12, são criadas as variáveis que devem ser instanciadas pela classe que a implementa, elas são responsáveis por validar o contrato do *package* e adicioná-lo na lista de rotas. A partir da linha 14, são criadas as funções de registro de rotas e de geração das rotas registradas.

5.2.3 *Packages*

Os *packages* são cada uma das funcionalidades do aplicativo e na Figura 18 é possível observar como é a estrutura do *package lists*. Dentro de cada *package* o *pubspec.yaml* terá a importação de *packages* externos e *shared packages* que serão utilizados. Além disso é importado como dependência o *core* que será responsável pelas requisições de navegação da funcionalidade.

Figura 18 - Pasta *lists*



Fonte: Autoria própria (2022)

Na pasta *lib*, todos os *packages* possuem um arquivo *.dart* de exportação das funções implementadas e uma pasta *app* com os arquivos *router*, que é responsável

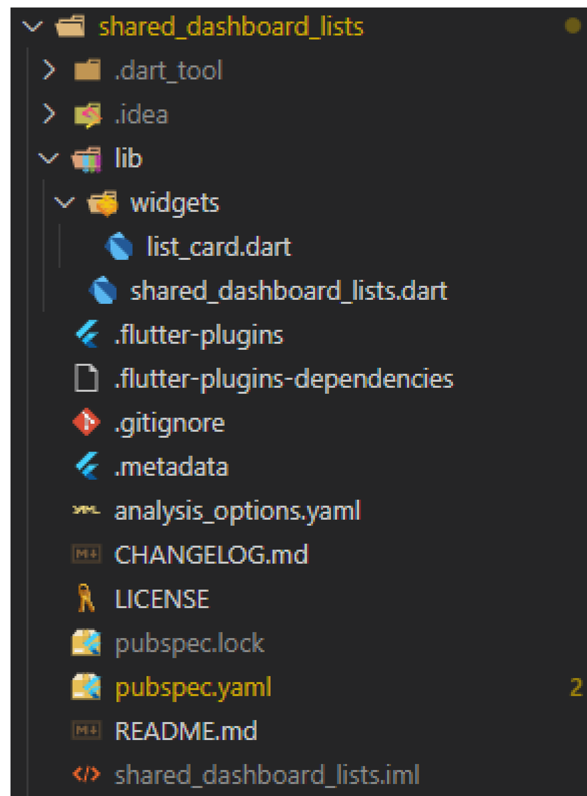
por implementar o contrato especificado no *Core* ,e *page*, que contém a estrutura da página da funcionalidade.

Fora os arquivos que são comuns a todos *packages*, a *lib* pode conter *widgets* implementadas apenas em uma funcionalidade ou arquivos do tipo *controller* para manter regras de negócio.

5.2.4 *Shared Packages*

Os *shared packages* são *packages* que implementam características comuns a dois ou mais *packages,* como *widgets*, por exemplo. A Figura 19 mostra como é a estrutura do *shared_dashboard_lists.*

Figura 19 - *Shared package dashboard lists*



Fonte: Autoria própria (2022)

Caso haja algum *package* externo que é utilizado em mais de um *package,* ele deve ser mantido em um *shared package*. Para isso, é feita a importação de quaisquer *packages* externos no arquivo *pubspec.yaml.*

Na *lib*, diferente de um *package* que implementa uma funcionalidade por completo e possui os arquivos *page* e *router*, o *shared package* implementa apenas as *widgets* que serão importadas nos *packages* que a utilizam.

Na aplicação desenvolvida, também é implementado um *shared package* chamado *Persistence*, que é responsável pelas funções ligadas a bancos de dados que serão utilizadas pelas listas. Para a persistência de dados foi utilizando o *package* externo “*Object Box”.*

6 CONSIDERAÇÕES FINAIS

Com a crescente demanda de mercado por aplicações móveis, se mostra necessária a utilização de arquiteturas de *software,* como a de *microfrontends*, voltadas para desenvolvimento de aplicativos, buscando manter a qualidade do *software*, produtividade do time de desenvolvimento e a consistência do produto entregue. Porém, com a falta de formalização para tal arquitetura, os produtos existentes apresentam inconsistências de projeto e de desenvolvimento (GEERS, 2020, p.15). Desta forma, este trabalho apresentou uma proposta de Arquitetura Orientada a *Packages* que buscou formalizar a arquitetura de *microfrontends* para desenvolvimento móvel utilizando a plataforma *Flutter*.

Para o desenvolvimento da formalização proposta, foi realizada uma análise sobre arquiteturas de *software* levantando quais aspectos do projeto de desenvolvimento devem ser integrados em uma arquitetura. Além disso, foram estudadas as implementações dos aplicativos das empresas Nubank, Banco Votorantim, Unico e IKEA que fazem a utilização de versões da arquitetura de *microfrontends.* Por fim, foi apresentada a plataforma *Flutter*, em que a formalização foi desenvolvida, para que houvesse maior compreensão do ambiente onde a arquitetura proposta será aplicada.

A partir dos estudos apresentados, foi proposta a formalização da Arquitetura Orientada a *Packages* dividindo o sistema em 4 componentes: *Base; Core; Packages* e *Shared Packages* (módulos compartilhados). Estes componentes são responsáveis, respectivamente, por: iniciar o aplicativo; manter as rotas de navegação entre funcionalidades; gerenciar os contratos entre funcionalidades e aplicativo; desenvolver as funcionalidades da aplicação; desenvolver e manter *Widgets* e *Dart Packages* comuns entre funcionalidades.

Por fim, foi desenvolvido um aplicativo de criação de listas e gerenciamento de itens que implementa a Arquitetura Orientada a *Packages* visando sua validação. O aplicativo apresenta as funcionalidades de: *Login; Dashboard;* Calculadora e Listas. Como Shared Packages, foram desenvolvidos módulos: “persistence”, esse módulo é responsável pela persistência dos dados das listas; “shared_dashboard_login”, utilizados pelos módulos *Dashboard* e *Login* de forma independente; e “shared_dashboard_lists”, utilizado pelo *Dashboard* e *Lists*.

Neste contexto, é possível concluir que a arquitetura proposta possui desvantagens quando aplicada em projetos com poucas funcionalidades, isso porque é necessária a criação de componentes de estrutura para gerenciamento de rotas, inicialização do aplicativo e contratos entre funcionalidades.

Além disso, é apresentada uma complexidade prévia ao desenvolvimento da aplicação, pois a exigência de funcionalidades independentes é um fator que deve ser considerado na etapa de projeto do sistema. Essa exigência vai manter as características da arquitetura de modularidade e escalabilidade.

A Arquitetura Orientada a *Packages,* portanto, é mais bem utilizada em projetos de aplicações com grande número de funcionalidades, que necessitem de uma divisão bem definida de partes do sistema e que sejam escaláveis.

6.1 Trabalhos futuros

Para trabalhos futuros, a arquitetura proposta deve ser comparada com arquiteturas já presentes na literatura, para que seja feita uma análise do tempo de desenvolvimento e qualidade de *software*.

Como mostrado na seção 4.3.2, a arquitetura apresenta perda de estado ao navegar entre *packages*. Dessa forma, mostra-se possível o desenvolvimento de uma solução para a perda do estado, mantendo os dados atualizados independente do *package* que os utilizam. Além disso, conforme apresentado na mesma seção, a arquitetura proposta é desenvolvida na plataforma *Flutter*. Portanto, cabe um estudo da arquitetura para sua adaptação e utilização em outros ambientes ou com outras linguagens de programação a fim de abranger o escopo de aplicação da arquitetura.

REFERÊNCIAS

ALBIN, S. T. **The art of *software* architecture: design methods and techniques**. John Wiley & Sons, 2003.

ALSHUQAYRAN, N.; ALI, N.; EVANS, R. A systematic mapping study in microservice architecture. In: **2016 IEEE 9th International Conference on Service-Oriented Computing and Applications** (SOCA). IEEE, 2016. p. 44-51.

AMAZON. **O que é Arquitetura orientada a serviços?**. 2022. Disponível em: https://aws.amazon.com/pt/what-is/service-oriented-architecture/. Acesso em: 15 out. 2022.

APPLE. Desenvolvimento para a App Store. In: **Desenvolvimento para a App Store**. [S. l.], 2022. Disponível em: https://www.apple.com/br/app-store/developing-for-the-app-store/. Acesso em: 25 set. 2022.

BANDEIRA, K. **Flutter: ferramenta facilita a vida de quem quer desenvolver aplicativos para smartphones**. Folha de Pernambuco, 2022. Disponível em: https://www.folhape.com.br/colunistas/tecnologia-e-games/flutter-ferramenta-facilita-a-vida-de-quem-quer-desenvolver-aplicativos-para-smartphones/29632/. Acesso em: 15 out 2022.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice**. Addison-Wesley Professional, 2003.

BIESSEK, A. **Flutter for Beginners: An introductory guide to building cross-platform mobile applications with Flutter and Dart 2**. Packt Publishing Ltd, 2019.

CLEMENTS, P., GARLAN, D., LITTLE, R., NORD, R., & STAFFORD, J. Documenting software architectures: views and beyond. In: **25th International Conference on Software Engineering, 2003.** Proceedings. IEEE, 2003. p. 740-741.

CONVERSATIONS ABOUT SOFTWARE ENGINEERING: **Micro Frontends with Gustaf Nilsson Kotte.** Entrevistado: Gustaf Nilsson Kotte. Entrevistador: Stefan Tilkov. 2018. Disponível em: https://www.case-podcast.org/22-micro-frontends-with-gustaf-nilsson-kotte. Acesso em: 25 set. 2022.

CORRÊA, L. C. **Desafios, agilidade e simplicidade**: uma abordagem para desenvolvimento mobile. 2018.

DAGNE, L. **Flutter for cross-platform App and SDK development**. 2019.

DATA AI. 2022. Disponível em: https://www.data.ai/pt/ Acesso em: 20 nov. 2022.

DMITRY, N.; MANFRED, S. S. On micro-services architecture. **International Journal of Open Information Technologies**, v. 2, n. 9, p. 24-27, 2014.

FERRERO, A. L. **Development of a large-scale flutter app**. 2022.

FGV (Fundação Getúlio Vargas). **Brasil tem 424 milhões de dispositivos digitais em uso**, revela a 31ª Pesquisa Anual do FGVcia. 8 jun. 2020. Disponível em: https://portal.fgv.br/noticias/brasil-tem-424-milhoes-dispositivos-digitais-uso-revela-31a-pesquisa-anual-fgvcia. Acesso em: 25 set. 2022.

FIGUEIREDO, C. M. S.; NAKAMURA, E. **Computação móvel**: Novas oportunidades e novos desafios. T&C Amazônia, v. 1, n. 2, p. 21, 2003.

FLUTTERANDO: **Flutter Brasil**. Direção: Flutterando. Intérprete: Thiago Suzano. Online: Flutterando, 2016. Disponível em: https://www.youtube.com/watch?v=OSex7G3wXg4&ab_channel=*Flutter*ando. Acesso em: 15 out. 2022.

FLUTTERANDO: **Flutter Brasil** 2022. Direção: Flutterando. Intérprete: Thiago Suzano. Online: *Flutter*ando, 2016. Disponível em: https://www.youtube.com/watch?v=qhg7BxK9Si4. Acesso em: 15 out. 2022.

FRISTER, D.; OBERWEIS, A.; GORANOV, A. Automated Testing of Mobile Applications Using a Robotic Arm. In: **2020 International Conference on Computational Science and Computational Intelligence** (CSCI). IEEE, 2020. p. 1729-1735.

GALSTER, M.; EBERLEIN, A.; MOUSSAVI, M. **Systematic selection of *software* architecture styles**. Iet Software, v. 4, n. 5, p. 349-360, 2010.

GARLAN, D.; PERRY, D. E. **Introduction to the special issue on *software* architecture**. IEEE Trans. Software Eng., v. 21, n. 4, p. 269-274, 1995.

GRÜNBACHER, P.; EGYED, A.; MEDVIDOVIC, N. **Reconciling software requirements and architectures with intermediate models**. Software & Systems Modeling, v. 3, n. 3, p. 235-253, 2004.

GEERS, M. **Micro frontends in action**. Simon and Schuster, 2020.

HOMAY, A.; *et al.,* A survey: Microservices architecture in advanced manufacturing systems. In: **2019 IEEE 17th International Conference on Industrial Informatics** (INDIN). IEEE, 2019. p. 1165-1168.

IKEA. **Sobre a IKEA.** 2022. Disponível em: https://www.ikea.com/pt/pt/this-is-ikea/about-us/ Acesso em: 22 nov. 2022.

JAMIL, M A.; *et al.,* Software testing techniques: A literature review. In: **2016 6th international conference on information and communication technology for the Muslim world** (ICT4M). IEEE, 2016. p. 177-182.

KEYNOTE: *Flutter* (Dart Developer Summit 2016). Direção: Eric Seidel. Munique, Alemanha: Google Germany GmbH, 2016. Disponível em:

https://www.youtube.com/watch?v=Mx-AllVZ1VY&t=996s&ab_channel=GoogleDevelopers. Acesso em: 28 set. 2022.

LASO, S.; *et al.,* Perses: A framework for the continuous evaluation of the QoS of distributed mobile applications. **Pervasive and Mobile Computing**, v. 84, p. 101627, 2022.
MARQUES, F. F. **NUBANK: O mercado de Fintechs no Brasil**. 2018.

MCILROY, M. D.; *et al.,* Mass-produced *software* components. In: **Proceedings of the 1st international conference on software engineering**, Garmisch Pattenkirchen, Germany. 1968. p. 88-98.

MCLEOD J. R. R.; EVERETT, G. D. **Software Testing**: **Testing Across the Entire Software Development Life Cycle**. John Wiley & Sons, 2007.

MINELLA, A. C. **Maiores bancos privados no Brasil: um perfil econômico e sociopolítico**. Sociologias, p. 100-125, 2007.

MIOLA, A. **Flutter Complete Reference**: Create Beautiful, Fast and Native Apps for Any Device. Independently published, 2020.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of *software* testing**. John Wiley & Sons, 2011.

OLUWATOSIN, H. S. **Client-server model**. IOSR Journal of Computer Engineering, v. 16, n. 1, p. 67-71, 2014.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. In: **Pioneers and their contributions to software engineering.** Springer, Berlin, Heidelberg, 1972. p. 479-1054.

PAYNE, R. **Beginning App Development with Flutter**: Create Cross-Platform Mobile Apps. Apress, 2019.

PERRY, D. E.; WOLF, A. L. **Foundations for the study of software architecture**. ACM SIGSOFT *Software* engineering notes, v. 17, n. 4, p. 40-52, 1992.

PRESSMAN, R. S. **Software engineering**: a practitioner's approach. Palgrave macmillan, 2005.

RANDELL, B.; BUXTON, J. N. **Software Engineering Techniques**: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27th-31st October 1969. 1970.

RICHARDS, M. ***Software* architecture patterns.** 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Incorporated, 2015.

RINGERT, J. O. **Analysis and synthesis of interactive component and connector systems**. Shaker, 2014.

SANTOS, A.; *et al.,* Investigating the adoption of agile practices in mobile application development. In: **Proceedings of the 18th International Conference on Enterprise Information Systems**, Itália. 2016.

SHAW, M.; DELINE, R.; ZELESNIK, G. **Abstractions and Implementations for Architectural Connections**. 3rd ICCDS, 1996.

SOMMERVILLE, I. **Software engineering** 9th Edition. ISBN-10, v. 137035152, p. 18, 2011.

VALIPOUR, M. H.; *et al.,* A brief survey of *software* architecture concepts and service-oriented architecture. In: **2009 2nd IEEE International Conference on Computer Science and Information Technology**. IEEE, 2009. p. 34-38.

WINDMILL, E. **Flutter in action**. Simon and Schuster, 2020.