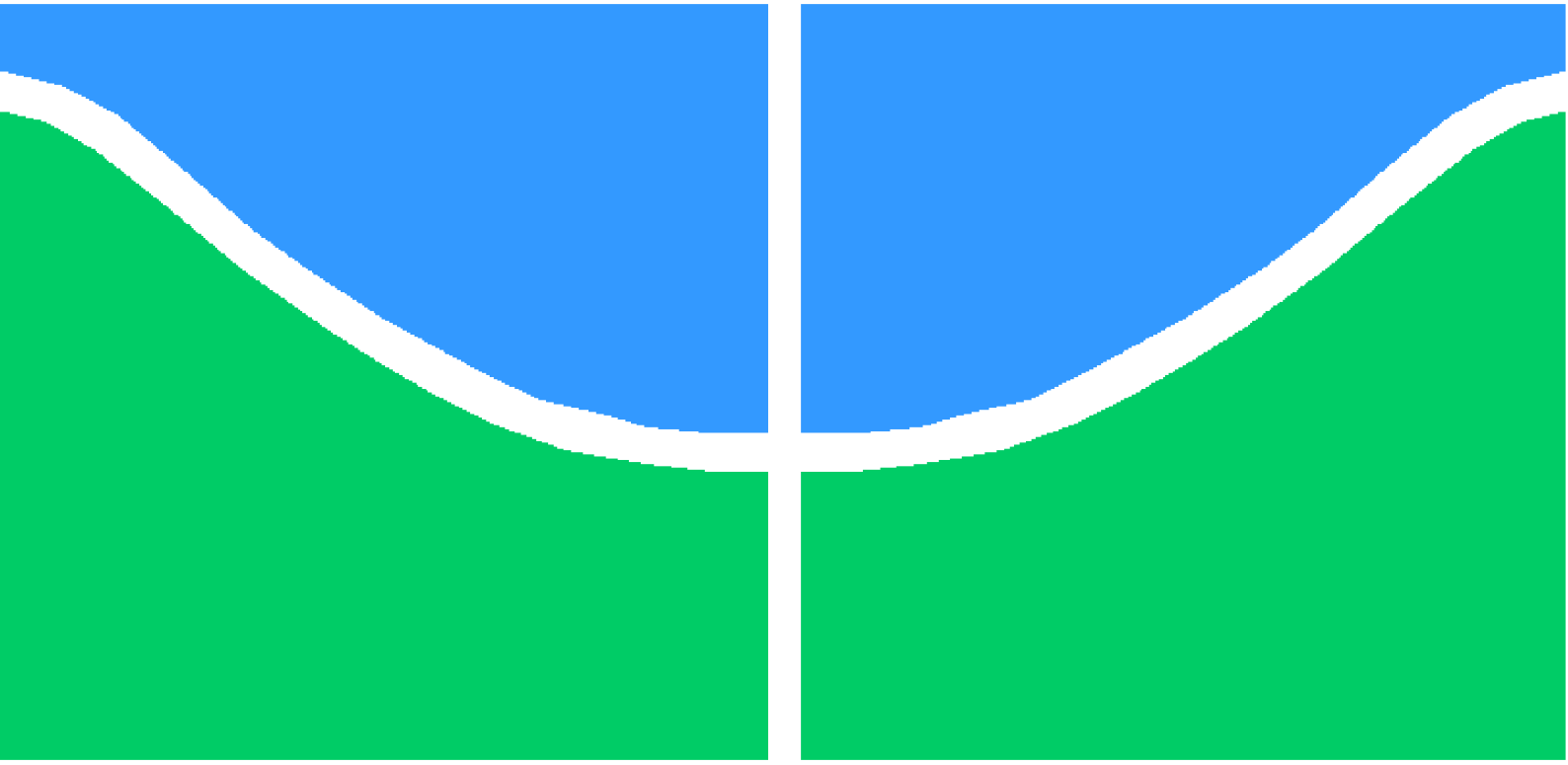


Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Arquitetura de Software: Um Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos

Autores: Gabriel Davi Silva Pereira e Danillo Gonçalves de Souza

Orientadora: Prof. Dra. Milene Serrano

Brasília, DF

2023

Gabriel Davi Silva Pereira e Danillo Gonçalves de Souza

Arquitetura de Software: Um Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dra. Milene Serrano

Brasília, DF

2023

Gabriel Davi Silva Pereira e Danillo Gonçalves de Souza

Arquitetura de Software: Um Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos/ Gabriel Davi Silva Pereira e Danillo Gonçalves de Souza. – Brasília, DF, 2023-

128 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dra. Milene Serrano

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2023.

1. Palavra-chave01. 2. Palavra-chave02. I. Prof. Dra. Milene Serrano. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Arquitetura de Software: Um Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos

CDU

Gabriel Davi Silva Pereira e Danillo Gonçalves de Souza

Arquitetura de Software: Um Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 17 de Fevereiro de 2023:

Prof. Dra. Milene Serrano
Orientador

Prof. Dr. Maurício Serrano
Examinador 1

M.Sc Rafael Fazzolino P. Barbosa
Examinador 2

Brasília, DF
2023

Dedicamos esse trabalho a todos aqueles que sempre acreditaram em nosso sucesso e potencial e que estiveram ao nosso lado ao longo de nossa jornada como acadêmicos.

Agradecimentos

Nossos agradecimentos são, primeiramente, à nossa família, pais e irmãos, que sempre nos deram apoio em todas as situações difíceis, nos incentivaram a seguir nossos sonhos e objetivos, e também proporcionaram uma educação de qualidade durante toda nossa vida que, por fim, resultou em nossa aprovação na renomada Universidade de Brasília.

Agradecemos também a nossos amigos e namoradas pelo incentivo que nos deram ao longo de nossa jornada acadêmica, e também pelo apoio demonstrado ao longo de todo o período de dedicação ao TCC.

Por fim, um agradecemos aos docentes da FGA, que nos proporcionaram o conhecimento necessário para nos tornarmos excelentes profissionais, em especial, à professora Milene Serrano, nossa orientadora, que nos convidou para sermos orientados por ela. A oportunidade única de ser orientado pela professora Milene, que possui bastante conhecimento na área de Arquitetura de *Software*, foi algo que acrescentou bastante na proposta desse trabalho.

Resumo

A partir do avanço das tecnologias dos *smartphones* e da popularidade dos mesmos, o desenvolvimento de aplicações híbridas para dispositivos móveis vem tornando-se cada dia mais comum. Apesar da relevância de aplicações *mobile*, a abrangência nos estudos sobre arquitetura ainda é escassa, quando comparado, por exemplo, a aplicações *web*. Nesse trabalho, foram realizados estudos sobre algumas das principais arquiteturas de software, aplicando-as no desenvolvimento de aplicativos móveis, sendo a construção orientada ao *Framework Cross-Platform* React Native. As arquiteturas escolhidas para condução dos estudos foram Arquitetura Limpa, Portas e Adaptadores e MVC. Ao longo dos estudos, procurou-se documentar os principais passos do desenvolvimento, baseando-se em uma metodologia híbrida, com práticas do Scrum e do Kanban; coletar métricas utilizando ferramentas de análise estática e, por fim, apresentar e analisar as métricas coletadas, conferindo informações, observadas no período, sobre cada arquitetura, bem como testes de integração com cobertura de 80%. Como intuito, e com foco nos comportamentos de cada arquitetura em vários aspectos (ex. reutilização e manutenibilidade), acredita-se que esse trabalho confere insumos e conhecimentos tanto aos autores, como também a outros interessados na pesquisa. Com viés prático, e centrado em desenvolvimento, os resultados reportados nesse trabalho evidenciam particularidades muito interessantes sobre cada arquitetura estudada.

Palavras-chaves: Arquitetura de *Software mobile*. Aplicações híbridas. *Framework Cross-Platform*. Padrão Arquitetural MVC. Arquitetura Portas e Adaptadores. Arquitetura Limpa.

Abstract

With the continuous evolution of smartphone technology and it's huge popularity, the development of hybrid applications for mobile devices is becoming universal. Although the popularity of mobile applications, studies about mobile application architecture are more scarce and simpler when compared, for example, with web applications architecture studies. The aim of is to study some of the most popular software architectures in the community and apply them in the development of mobile applications using the Cross-Platform Frameworks React Native. The architectures chosen were Clean Architecture, Ports and Adapters and MVC. During the studies, the application development will be documented using a hybrid methodology of Scrum and Kanban; metrics will be collected using static analysis tools and, by the end, the collected metrics of each architecture will be analyzed and presented. The main focus is to analyze different behaviors and characteristics of each architecture in multiple aspects (ex. reuse and maintainability).

Key-words: Software Architecture. Hybrid applications. Cross-Platform Framework. MCV Architectural Pattern. Ports and Adapters Architecture. Clean Architecture.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
BPMN	*Business Process Model and Notation*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
IDE	*Integrated Development Environment*
MVC	*Model View Controller*
PIBIC	Programa Institucional de Bolsas de Iniciação Científica
SDK	*Software Development Kit*
UI	*User Interface*

Sumário

1 Introdução

Este capítulo tem como intuito apresentar uma breve contextualização sobre o domínio e a área de atuação técnica desse trabalho, procurando introduzir conceitos relevantes para compreensão do trabalho realizado. Como domínio, o trabalho compreende o Desenvolvimento de Aplicativos Móveis. Como área de atuação, tem-se Engenharia de *Software*, mais precisamente, Arquitetura de *Software*. Será descrita ainda a justificativa, procurando acordar os principais motivos para realização do estudo proposto. Na sequência, têm-se Questão de Pesquisa e Objetivos, sendo esses apresentados como Objetivo Geral e Objetivos Específicos. Por fim, consta a Organização da Monografia em capítulos.

1.1 Contexto

O número de dispositivos móveis cresce de forma ininterrupta, diariamente. Diante desse cenário, a visão de negócio de diversas empresas precisou se adequar em atendimento ao público associado a esse crescimento. A utilização massiva de dispositivos móveis, em todo o mundo, moldou um novo mercado, o qual, atualmente, é significativo, bem estabelecido e indispensável para a maioria dos indivíduos (WENI, 2017).

No princípio, para atender o mercado de dispositivos móveis, era necessário desenvolver aplicações para cada tipo de Sistema Operacional (ex. Android e iOS). Porém, com o surgimento de ferramentas para o desenvolvimento multiplataforma, o processo de criação de aplicações móveis tornou-se simplificado (PALMIERI; SINGH; CICCHETTI, 2012). Existem inúmeros *Frameworks* para desenvolvimento multiplataforma que servem para o mesmo propósito, sendo esse: facilitar a criação de uma aplicação híbrida. Todavia, apesar das diversas facilidades que tais ferramentas oferecem ao programador, ainda sim, um aplicativo concebido sem planejamento adequado, em especial em termos arquiteturais, pode incorrer em insucessos (Software Engineering Institute, 2022).

Diante do exposto, pode-se entender Arquitetura de *Software* como algo que envolve estruturar e organizar componentes de sistemas e subsistemas para que interajam de forma harmoniosa entre si. Adicionalmente, confere insumos para definição/planejamento de uma estrutura adequada, que se enquadra a cada tipo de projeto (KRUCHTEN; OBBINK; STAFFORD, 2006). A arquitetura impacta todas as etapas do ciclo de vida de um *software*, desde a concepção, a partir do momento que decisões mais técnicas começam a ser tomadas, até projeto, implementação, testes e manutenções, sejam essas de cunho corretivo ou evolutivo. Portanto, estilos e padrões arquiteturais, quando bem aplicados, tendem a trazer benefícios a longo prazo, para o projeto como um todo (FOWLER, 2003).

Nesse sentido, estudos sobre Arquitetura de *Software*, em especial em cenários em crescente evolução, tal como no desenvolvimento de aplicativos móveis, são desejados e pertinentes, envolvendo uma gama de pesquisadores (LEE; SCHNEIDER; SCHELL, 2004) e iniciativas de mercado (GRØNLI et al., 2014) e (Android for Developers, 2022).

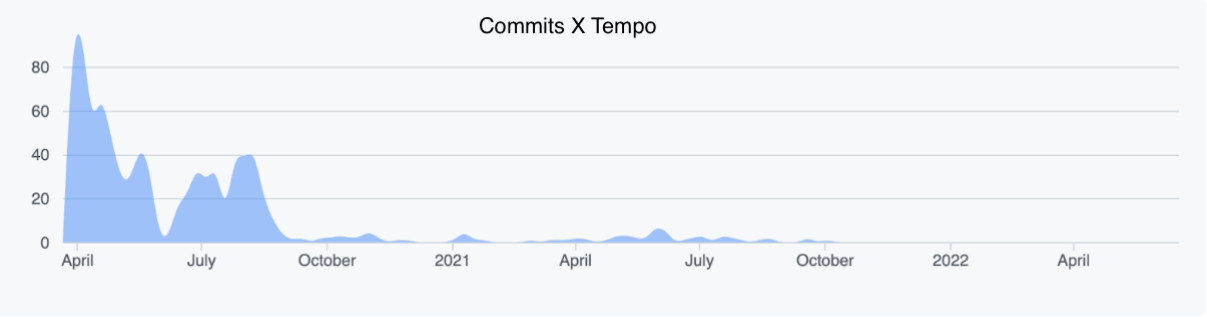
1.2 Justificativa

De forma geral, é muito mais simples entender os motivos de se utilizar uma arquitetura mais elaborada no desenvolvimento de um sistema quando observadas as consequências de sua ausência.

Um dos motivos mais comuns de se negligenciar a arquitetura de um *software* é a concepção equivocada de que a velocidade de entrega de um sistema é mais acelerada quando não há preocupação com o desenho de uma solução arquitetural (MARTIN, 2017). Ao aplicar essa abordagem, em um primeiro momento, novas funcionalidades podem ser entregues mais rapidamente. Todavia, em tempo de manutenção do *software* (BAKOTA et al., 2012), pode-se observar que há diminuição ou até estagnação no ritmo de desenvolvimento e, consequentemente, de entregas.

No intuito de conferir um exemplo mais concreto, que evidencia a negligência com o planejamento arquitetural, incorrendo, em tempo de manutenção e evolução do *software*, em uma baixa significativa em termos de funcionalidades entregues, a Figura 1 acorda um gráfico. Nesse gráfico, há uma correlação entre o número de *commits* entregues por uma equipe, em função do tempo, no contexto do projeto de um aplicativo, chamado Mia Ajuda (2020).

Figura 1 – *Commits* por tempo - Projeto Mia Ajuda



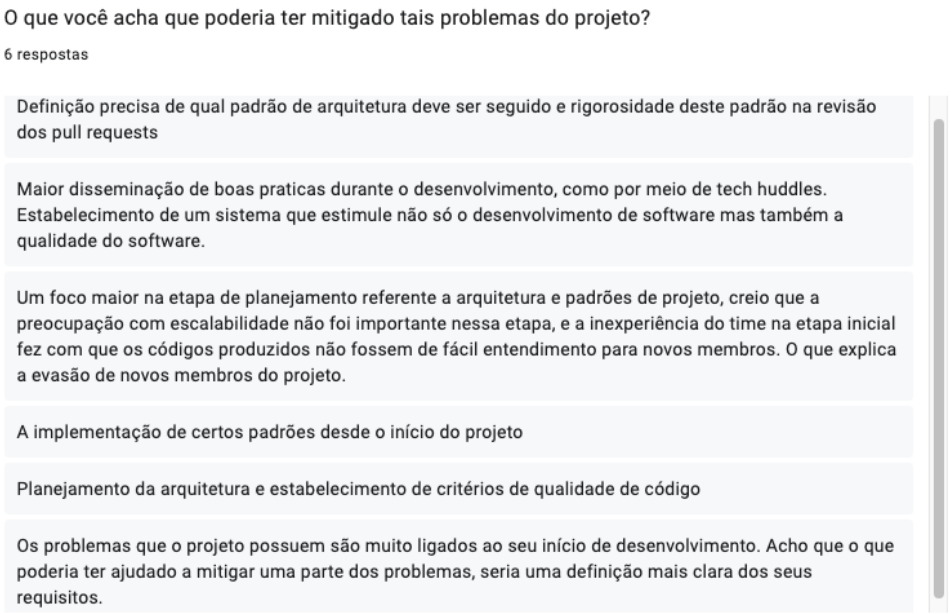
Fonte: *Github.*

Nesse projeto, alunos da Universidade de Brasília, graduandos em Engenharia de *Software* (predominantemente), desenvolveram um aplicativo, cujo intuito é aproximar pessoas que precisam de ajuda, com aquelas que podem e querem ajudar. O momento, no qual o projeto foi idealizado e desenvolvido, compreendeu os primeiros meses de

pandemia de Covid-19, em 2020. Portanto, havia a demanda do aplicativo ser desenvolvido rapidamente, permitindo conferir, o mais breve possível, o apoio a quem necessitava, ainda mais diante da situação de isolamento imposta na época. Tendo em mente essa urgência, a equipe acabou por negligenciar, principalmente, algumas práticas inerentes à Engenharia de Requisitos (ESPINDOLA; MAJDENBAUM; AUDY, 2004) e à área de Arquitetura de *Software*.

Conferindo uma breve análise dos dados revelados no gráfico, há uma evidente queda no número de *commits*, ao longo do tempo. Em um primeiro momento, muitos *commits* foram conferidos pela equipe, o que viabilizou a publicação da primeira versão do aplicativo Mia Ajuda, na PlayStore, em 2 meses e 9 dias. O ritmo caiu sensivelmente nos meses que sucedem o lançamento do aplicativo, em 31 de Maio de 2020. Há uma retomada, entre os meses de Julho e Agosto. Depois, já vislumbrando uma segunda versão, ou seja, em tempo de manutenção da primeira versão, e evolução da mesma para uma segunda versão, o número de *commits* manteve-se muito baixo, levando até à ausência de *commits*. A pergunta a ser feita é: O que ocorreu? De acordo com uma pesquisa realizada com os desenvolvedores que participaram ativamente na criação da aplicação, essa situação deve-se à vários fatores, dentre eles: a dificuldade na alteração do código fonte devido à ausência de padrões. Nesse caso, conforme os *feedbacks* coletados na pesquisa, se avaliado em termos de projeto, há carência de padrões de projeto, e até mesmo de uma análise qualitativa do código. Já em termos de arquitetura, não ocorreu, de forma criteriosa, um desenvolvimento orientado às boas práticas de um ou mais estilos arquiteturais. A seguir, nas Figuras 2, 3 e 4, podem ser vistos os resultados obtidos na pesquisa.

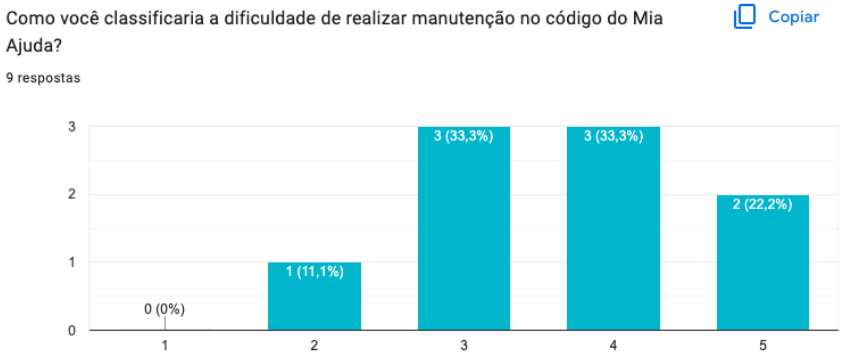
Figura 2 – Respostas de como mitigar problemas do Mia Ajuda.



Fonte: Autores.

Na Figura 3, cabe colocar que metade da equipe atuante no Projeto Mia Ajuda considera difícil realizar a manutenção no código do aplicativo. A escala utilizada na pesquisa varia de 0 a 5, onde o nível 0 representa nenhuma dificuldade, e o nível 5 representa dificuldade muito alta, passando por nível 1 (muito baixa dificuldade), nível 2 (baixa dificuldade), nível 3 (média dificuldade), e nível 4 (alta dificuldade). Como resultados, em uma amostra de nove participantes, tem-se: 33,3 % da equipe revelando dificuldade nível 4, e 22,2 % revelando dificuldade nível 5. Ainda há parte significativa da equipe que revela dificuldade nível 3, sendo 33,3 %. Apenas um membro da equipe (11,1%) revela dificuldade nível 2, não sendo mencionados o nível 1 pelos participantes.

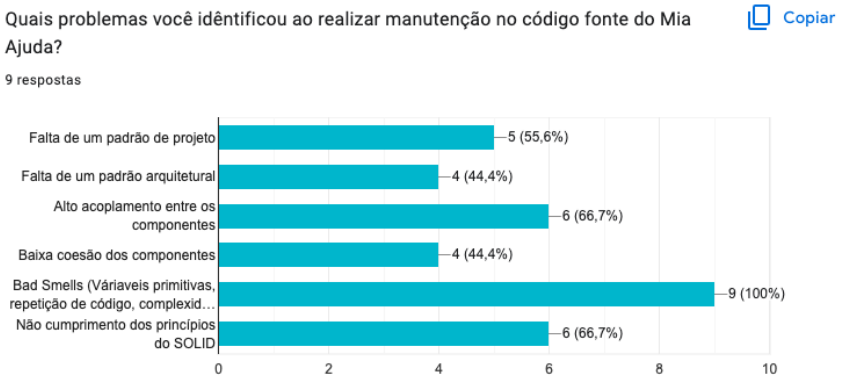
Figura 3 – Respostas com as principais dificuldades no código do Mia Ajuda.



Fonte: Autores.

Na Figura 4, são acordados alguns problemas identificados pelos participantes ao realizarem a manutenção do código do aplicativo Mia Ajuda. Ressalta-se que todos os problemas remetem à falta de planejamento em termos de projeto e arquitetura, com destaque para: identificação de *bad smells* (ex. repetição de código); falta de padrões de projeto (ex. Alto Acoplamento, Baixa Coesão e falta de princípios como SOLID), e falta de padronização e planejamento arquitetural.

Figura 4 – Respostas com os principais problemas no código do Mia Ajuda.



Fonte: Autores.

Como pode ser visto nas respostas, alterar algo, em uma arquitetura com muitos *bad smells* (GARCIA et al., 2009), não é uma tarefa fácil e, por vezes, de alto custo e esforço.

A implementação de novas funcionalidades afeta módulos do sistema que, em um primeiro momento, não pareciam ter correlação direta; *bugs* inesperados acabam por ocorrer frequentemente; há duplicação desnecessária de código, e até mesmo código morto, dentre outros vários fatores que tornam qualquer iniciativa, nesse código, praticamente, impossível. Mesmo aumentando o número de desenvolvedores, o que ocorreu no período, continuou inviável a evolução do aplicativo. Diante desse cenário complicado, a equipe de colaboradores perdeu sensivelmente o engajamento.

De acordo com Al-Badareen et al. (2011), há características que impactam diretamente na manutenção de *software*. Dentre as mais impactantes, têm-se Entendimento, Facilidade de Evolução, Modularização e Estruturação. No Projeto Mia Ajuda, há evidências claras de que o entendimento e a facilidade de evolução foram aspectos comprometidos. Além disso, considerando que modularização e estruturação compreendem a divisão do sistema em diferentes partes gerenciáveis, que se comunicam de forma adequada entre elas, esses aspectos também foram negligenciados no projeto. Tais colocações podem ser observadas consultando o repositório do Projeto Mia Ajuda, disponível aqui (github.com/mia-ajuda). Cabe ressaltar que, por definição, arquitetura de *software* tem correlação direta com esses aspectos. Por fim, conclui-se, com base no exposto, sobre a importância de se considerar um planejamento arquitetural adequado, sendo esse orientado às características supracitadas, visando menor esforço em tempo de manutenção e evolução do *software*.

1.3 Questão de Pesquisa

Essa pesquisa procura responder ao seguinte questionamento:

- Quais são os comportamentos mais observados quando as principais arquiteturas de *software* são aplicadas no desenvolvimento de aplicativos móveis? Entende-se por comportamentos quaisquer evidências correlacionadas aos tópicos: coesão, acoplamento, testabilidade, reutilização, manutenibilidade dentre outros aspectos envolvendo os componentes arquiteturais, e sugeridos como relevantes pela literatura especializada (POPE, 2010), (BROWN, 2014), (GRAÇA, 2017a), (BAILÉN, 2019), (FOWLER, 2003), (BOUKHARY; COLMENARES, 2019), (MARTINEZ, 2021) e (KRIGER, 2021). Análise estática de código e outros recursos serão utilizados para identificação de *code smells*, *bugs*, vulnerabilidades e outros insumos associados aos comportamentos. Além da realização de testes de integração para identificar o acoplamento e a testabilidade entre as camadas e/ou os componentes de cada arquitetura.

1.4 Objetivos

Como objetivo geral, esse trabalho compreende a realização de estudos sobre algumas das principais arquiteturas de *software*, aplicando-as no desenvolvimento de aplicativos móveis, de forma a identificar comportamentos específicos durante o ciclo de desenvolvimento capazes de conferir insumos em resposta à questão de pesquisa anteriormente apresentada.

Visando atingir o objetivo geral, procurou-se cumprir com os seguintes objetivos específicos:

- Realização de um levantamento bibliográfico, no intuito de acordar as arquiteturas mais aderentes ao desenvolvimento de aplicativos móveis;
- Identificação, também com base na literatura especializada, de princípios, métricas e parâmetros que permitam expor de forma mais clara sobre os comportamentos observados com a aplicação das arquiteturas;
- Especificação de cenários de uso, no domínio de desenvolvimento de aplicativos móveis, nos quais são aplicadas as arquiteturas;
- Implementação dos cenários de uso orientando-se pelas arquiteturas em estudo, e
- Apresentação e análise dos resultados obtidos ao longo dos estudos.

Cabe ressaltar ainda que os levantamentos bibliográficos, a implementação e a análise dos resultados obtidos são guiados, respectivamente, por metodologias específicas, sendo essas apresentadas em detalhes no Capítulo 4. Adicionalmente, ocorreu a necessidade de cenários de uso, de forma a delimitar o escopo de atuação desse trabalho.

1.5 Organização da Monografia

Esta monografia está estruturada da seguinte forma:

- Capítulo 2 - Referencial Teórico: Descrição dos principais embasamentos teóricos, os quais fundamentam o presente trabalho, com destaque para a área de Arquitetura de *Software* e as principais arquiteturas associadas ao domínio de desenvolvimento de aplicativos móveis;
- Capítulo 3 - Suporte Tecnológico: Apresentação das principais tecnologias, as quais possibilitaram a realização desse trabalho, com menção aos *frameworks* e plataformas de desenvolvimento, bem como à ferramenta de análise estática de código e testes;

- Capítulo 4 - Metodologia: Exibição do plano metodológico seguido pelos autores para a condução do trabalho como um todo, da classificação da pesquisa, seguida pelos levantamentos bibliográficos, às atividades de desenvolvimento e análise de resultados;
- Capítulo 5 - Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos: Exposição detalhada quando ao estudo desse trabalho em si, apresentando-o da concepção da ideia às provas de conceito e artefatos elaborados;
- Capítulo 6 - Análise de Resultados: Apresentação dos principais resultados obtidos até o momento, conferindo uma análise com base em pesquisa-ação, e
- Capítulo 7 - Conclusão: Fechamento do trabalho, retomando contexto, principais resultados obtidos, questão de pesquisa, e objetivos, bem como propondo ideias para trabalhos futuros.

2 Referencial Teórico

Nesse capítulo, serão apresentados os principais referenciais conceituais, os quais embasam esse Trabalho de Conclusão de Curso. Pretendeu-se, com esse trabalho, ter contribuições na área de Arquitetura de *Software*, mais especificamente, Arquitetura de *Software* para Aplicações *Mobile*.

Em um primeiro momento, tem-se a definição do termo Arquitetura, mais especificamente, Arquitetura de *Software*, uma vez que essa é a área de interesse desse estudo, no contexto da Engenharia de *Software*. Com base nessa visão introdutória da área, são conferidos insumos que auxiliam na compreensão da importância de uma Arquitetura de *Software* bem planejada e executada, e evidências, com base na literatura, que permitem perceber que uma dada arquitetura pode incorrer em situações indesejadas. Visando acordar formas de se mitigar essas situações, são apresentados princípios inerentes à uma boa arquitetura.

Entretanto, o presente trabalho centra suas contribuições no Desenvolvimento *Mobile*. Sendo assim, há colocações sobre o mercado *mobile*; as vantagens de se trabalhar orientado a *frameworks* que viabilizam o desenvolvimento de aplicações híbridas, bem como um olhar mais aprofundado sobre Arquitetura de *Software* e Desenvolvimento *Mobile*, procurando apresentar padrões arquiteturais mais aderentes a esse nicho de desenvolvimento.

Na sequência, há menção aos Testes de Integração, uma vez que os mesmos permitem perceber nuances sobre o funcionamento dos componentes arquiteturais de um software, ou seja, como os mesmos atuam de forma integrada. Essas nuances revelam aspectos interessantes sobre o comportamento da arquitetura, tal como a comunicação entre os módulos de um sistema.

Por fim, é conferido um resumo do capítulo, com os principais aspectos apresentados ao longo do mesmo.

2.1 Arquitetura e *Design* de *Software*

De forma a conferir uma visão geral sobre a área de atuação desse trabalho, seguem seções dedicadas à definição de Arquitetura de Software; situações indesejadas ao se negligenciar aspectos arquiteturais, e princípios que ajudam a mitigar tais situações.

2.1.1 Definição

De acordo com o dicionário da língua portuguesa (FERREIRA; ANJOS, 1988), define-se arquitetura como a arte de construir e decorar edifícios. Em seu sentido figurado, tem-se: forma e estrutura.

De fato, é muito mais simples o entendimento do conceito de arquitetura quando a mesma é aplicada em objetos palpáveis, que podem ser vistos e sentidos. Todavia, a complexidade da definição de arquitetura aumenta de forma significativa quando aplicada em sistemas de *software*.

Prédios possuem estruturas óbvias, com colunas e tijolos, que respeitam as leis da física e limitações da natureza. Entretanto, produtos de *software* são formados por componentes menores de *software*, que por sua vez são formados por componentes menores ainda, e assim por diante (MARTIN, 2017). Não bastasse essa inerente dificuldade, esses componentes e subcomponentes possuem um viés subjetivo, uma vez que são obtidos, mesmo em sua forma mais concreta, ou seja, codificados, orientando-se por paradigmas de programação.

Segundo Kristensen e Österbye (1994) e Daniels (2002), paradigmas de programação são modelos conceituais, que possuem abstrações, as quais viabilizam a representação de algo do mundo real em uma solução computacional. Abstrações, como o próprio termo sugere, são subjetivas, idealizações, representações, e não algo de estrutura sólida e concreta.

Apesar de possuírem naturezas distintas, é possível entender a arquitetura de um sistema como um grande "prédio", que possui micro compartimentos que compõem uma estrutura organizada. Analogias como essa são propostas e divulgadas pela comunidade de *software*, tal como consta em Arkusnexus (2020).

Há autores (FOWLER, 2003), inclusive, que acordam aspectos interessantes nessa analogia, mencionando que uma Arquitetura de *Software* vai além da simples comparação entre *software* e uma edificação, ou entre um arquiteto de *software* e um arquiteto de uma casa. Em ambos os casos, há envolvimento de muitos profissionais, visando viabilizar a construção de um *software* ou de uma casa. Entretanto, no papel do arquiteto de *software*, há a necessidade de "chefiar" vários envolvidos, incluindo vários profissionais da equipe técnica. Pode caber ao arquiteto de *software* o papel de conversar com o cliente, mas esse não é o principal foco de atuação desse profissional. No papel de um arquiteto de uma casa, pode-se ser atribuído o papel de lidar diretamente com o cliente. Em ambos os casos, há demanda de profissionais qualificados, especializados. Cabe, todavia, considerar essas inerentes diferenças de atuação.

Diante do exposto, é perceptível que definir Arquitetura de *Software* não é algo trivial. Cabem interpretações, abstrações, e muita margem para debates entre os especialis-

tas da área. Visando apresentar uma definição, em palavras mais simples, e que ajudam a compreender como a arquitetura é tratada no escopo desse trabalho, têm-se as colocações de Martin (2017). Basicamente, o autor define Arquitetura de *Software* como a forma como é disposto um sistema. Essa forma manifesta-se em como os componentes estão organizados, e na forma como os mesmos se comunicam entre si para formar o *software* como um todo.

2.1.2 Situações Indesejadas

Há situações que, quando notadas com certa frequência em um sistema, podem ser indícios fortes de que a arquitetura da solução não foi planejada de forma adequada (GRAÇA, 2017a). As principais situações em uma arquitetura inadequada são:

- Rigidez: Um *software* é dito rígido quando qualquer alteração incorre na necessidade de mais alterações em diferentes partes do sistema;
- Fragilidade: Quando frágil, um sistema pode quebrar em partes diferentes e imprevisíveis;
- Imobilidade: Há partes do sistema que poderiam ser reaproveitadas, mas o esforço para tornar isso possível é inviável;
- Viscosidade: Em um sistema viscoso, é muito mais simples desenvolver fora do padrão, sem a orientação de um planejamento de diretrizes, do que seguir um modelo. Um dos sinais de um *software* viscoso é o ambiente lento de desenvolvimento. O tempo de compilação é maior do que o necessário, o que leva desenvolvedores a implementar soluções de forma a "enganar" o compilador e diminuir a necessidade de recompilação;
- Repetição: Sem uma boa estruturação ou conhecimento de desenvolvimento, é comum código duplicado;
- Opacidade: O código foi escrito de forma confusa, sendo necessário adentrar (i.e. investigar de forma profunda a lógica) em classes e métodos a fim de entender o que está acontecendo, e
- Complexidade: Com o objetivo de evitar as seis situações mencionadas anteriormente, desenvolvedores procuram abstrações e padrões que tentam prever qualquer mudança que o *software* pode sofrer. Isso torna o código do sistema extremamente complexo. Já um *software* bem projetado tende a ser mais simples e fácil adaptá-lo.

2.1.3 Princípios

Tendo em mente a definição, a importância e as situações indesejadas, no contexto de Arquitetura de *software*, cabe a exposição de princípios. Esses princípios, se levados em consideração, mitigam falhas e acordam uma arquitetura mais adequada.

Vale ressaltar que, quando bem projetada, uma solução arquitetural pode ser bastante alinhada e endereçar adequadamente determinados problemas. Por isso, segundo Graça (2017c), não há uma arquitetura que atenda todo e qualquer propósito, assim como não há uma arquitetura de propósito único. Compete ao arquiteto de *software* entender o problema e outros insumos (ex. regras de negócio), e propor uma arquitetura mais adequada, por vezes adaptada a um dado cenário de uso. Nesse contexto, é importante conhecer diferentes alternativas arquiteturais, seus pontos negativos e positivos, bem como quais problemas buscam mitigar.

Orientando-se pelas colocações de Martin (2000), uma boa arquitetura, geralmente, é adaptável e independente de fatores externos, como serviços e bancos de dados. Observa-se o mínimo de esforço em sua adaptação quando há mudanças, por exemplo, nas regras de negócio. Essas boas práticas, via de regra, acompanham princípios essenciais de *Design* de *software*, conhecidos como SOLID (MARTIN, 2017).

SOLID é um acrônimo para descrever princípios que informam como organizar funções e classes e, como essas partes do sistema devem se comunicar. São eles:

- SRP (Princípio da Responsabilidade Única): Uma classe deve ter somente um motivo para mudar;
- OCP (Princípio do Aberto/Fechado): Uma classe deve ser aberta para extensão e fechada para modificações;
- LSP (Princípio de Substituição de Liskov): Para que partes do sistema possam ser intercambiáveis, elas devem seguir um contrato que possibilita elas serem substituídas umas pelas outras;
- ISP (Princípio da Segregação de Interface): Orienta arquitetos a não depender de coisas que não usam, e
- DIP (Princípio da Inversão de Dependência): Um código que implementa uma política de alto nível não pode depender de um código que implementa uma política de nível mais baixo, e sim de abstrações.

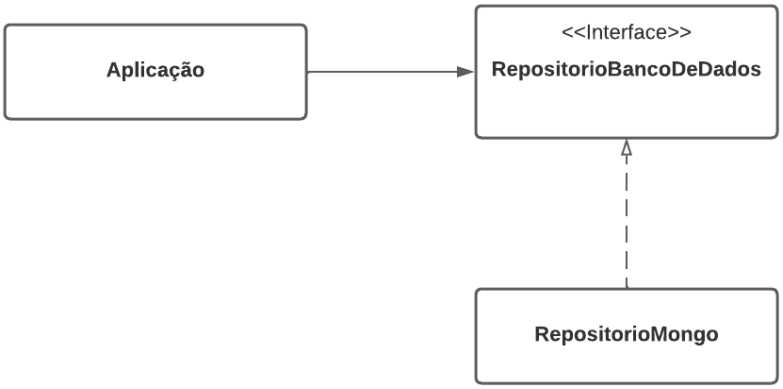
Um sistema de *software* bem projetado, segundo Shvets (2022), costuma manter o equilíbrio de tais princípios, mas nem sempre todos ao mesmo tempo. Entretanto, o cumprimento desses princípios não é uma tarefa fácil.

Junto aos princípios do SOLID, algumas boas práticas de estruturação de aplicações incluem o uso de Padrões de Projeto (SHVETS, 2022).

Padrões de Projeto constituem soluções catalogadas para situações comumente encontradas ao longo do projeto e do desenvolvimento de um *software*. Tais padrões são obtidos a partir de experiências prévias, estabelecendo como partes do *software* devem ser modeladas e implementadas. Em diferentes situações, essas técnicas ajudam a manter, até mesmo, alguns princípios descritos no SOLID. Por essa e outras razões, são soluções recorrentes em diversas implementações arquiteturais. Um catálogo de soluções, compreendendo a implementação de vários padrões de projeto, foi desenvolvido, por um dos autores dessa monografia, em um projeto PIBIC, resultando em um repositório para consulta por parte de interessados, disponível em: https://github.com/GabrielDVpereira/design-patterns-implementation

Visando um código mais desacoplado, o que auxilia em questões de manutenção, pode-se aplicar Injeção de Dependência (YANG; TEMPERO; MELTON, 2008). Injeta-se dependência em uma classe (ou ainda, algum módulo do projeto) pelo seu construtor, ao invés de instanciá-la(o) diretamente via classe. Essa prática dá-se por meio do uso de interfaces. A classe que espera a Injeção de Dependência em seu construtor está apontando para uma dada interface. Essa, por sua vez, será implementada pela classe que recebe a Injeção de Dependência. Percebe-se, nessa implementação, que há remoção no uso de classes concretas, optando por polimorfismo (princípio conhecido da Orientação a Objetos, conforme explicado em Larman (2012)), para tornar o código mais adaptável às mudanças, mais coeso e menos acoplado. Segue ainda um exemplo, na Figura 5, visando ilustrar a aplicação dessa boa prática no contexto de uma aplicação que utiliza um banco de dados.

Figura 5 – Exemplificação de Injeção de Dependência



Observe que o *RepositorioMongo* comunica-se com *Aplicação* por uma interface. Caso seja necessária a alteração do modelo do banco de dados, para Postgres, por exemplo, não haveria problemas com relação à classe principal, pois bastaria a nova classe imple-

mentar a interface *RepositorioBancoDeDados*. Com isso, a classe *Aplicação* não estaria ciente de tal alteração, facilitando a manutenção.

Princípios e boas práticas, aplicados no planejamento de uma arquitetura de *software*, tendem a colaborar quando alterações ocorrem, incluindo escolhas associadas à camada de persistência; aos aspectos tecnológicos e outros, bem como evoluções/ajustes nas regras de negócio e requisitos (MARTIN, 2017).

2.2 Desenvolvimento *Mobile*

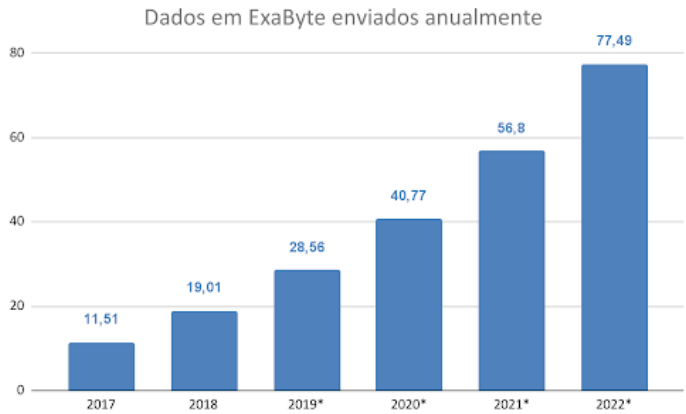
Nesse trabalho, há maior esforço em conferir contribuições para o contexto de aplicativos móveis. Dessa forma, nas próximas seções, constam, respectivamente: uma visão sobre o Mercado Mobile; e um descritivo sobre Framework para Aplicações Híbridas.

2.2.1 Mercado *Mobile*

Atualmente, a popularidade dos *smartphones* é inegável. Os *smartphones* estão presentes e fazem parte do dia a dia da maior parte da população mundial. De acordo com dados, divulgados de forma pública, pela Agência Nacional de Telecomunicações em 2015, o número de dispositivos móveis no Brasil ultrapassou a marca de 257 milhões de unidades, ou seja, a quantidade de aparelhos móveis superou a população de brasileiros, que na época era de 205 milhões (WENI, 2017).

A tendência de crescimento é acompanhada também fora do Brasil. De acordo com uma estimativa realizada pela empresa Cisco Systems, no ano de 2022, ocorreu um aumento no tráfego de dados vindo de dispositivos móveis de pelo menos 570% ExaBytes mensais em relação ao ano de 2017. O crescimento pode ser visto na Figura 6 (CLEMENT, 2020).

Figura 6 – Quantidade de tráfego de dados mensais estimados anualmente



FONTE: (CLEMENT, 2020)

Com o avanço da tecnologia dos dispositivos móveis e uma grande parcela da população utilizando-os, produtos que antes eram oferecidos de forma exclusiva para computadores e *notebooks* tiveram obrigatoriamente que ganhar uma versão para *mobile* também. Essa imposição de mercado criou um novo setor voltado exclusivamente para o desenvolvimento de aplicações móveis (WENI, 2017).

2.2.2 *Framework* para Aplicações Híbridas

Inicialmente, por se tratar de um novo mercado a ser explorado, as aplicações *mobile* eram desenvolvidas inteiramente de forma nativa, ou seja, todo e qualquer aplicativo deveria ser concebido a partir de um *software* proprietário para uma arquitetura de um sistema operacional que interpretava uma linguagem de programação específica. Sendo assim, aplicativos deveriam ser desenvolvidos duas vezes, considerando, principalmente, os dois sistemas operacionais dominantes no mercado para *smartphones* (iOS e Android), o que aumenta consideravelmente o custo e o esforço para criar e manter aplicações distintas (PALMIERI; SINGH; CICCHETTI, 2012).

Para mitigar os diversos problemas que os desenvolvedores possuíam com aplicações nativas, surgiram os *Frameworks* para aplicações *Cross-Platform* que possibilitaram a criação de aplicações híbridas. O desenvolvimento de aplicações híbridas trouxe, segundo os autores a seguir apresentados, diversos benefícios, tais como:

- Desenvolvimento de uma única aplicação: Anteriormente, para que uma aplicação fosse funcional em diversos sistemas operacionais, era necessário desenvolver diversos aplicativos nativos, cada qual especializado em cada tipo de arquitetura no mercado (PALMIERI; SINGH; CICCHETTI, 2012);

- Redução do custo: A utilização de *Frameworks-Cross* reduziu a quantidade de dinheiro gasto com o desenvolvimento e a manutenção da aplicação. Adicionalmente, diminuiu a necessidade de diversos times de desenvolvedores especializados em diversas linguagens de programação (PALMIERI; SINGH; CICCHETTI, 2012);

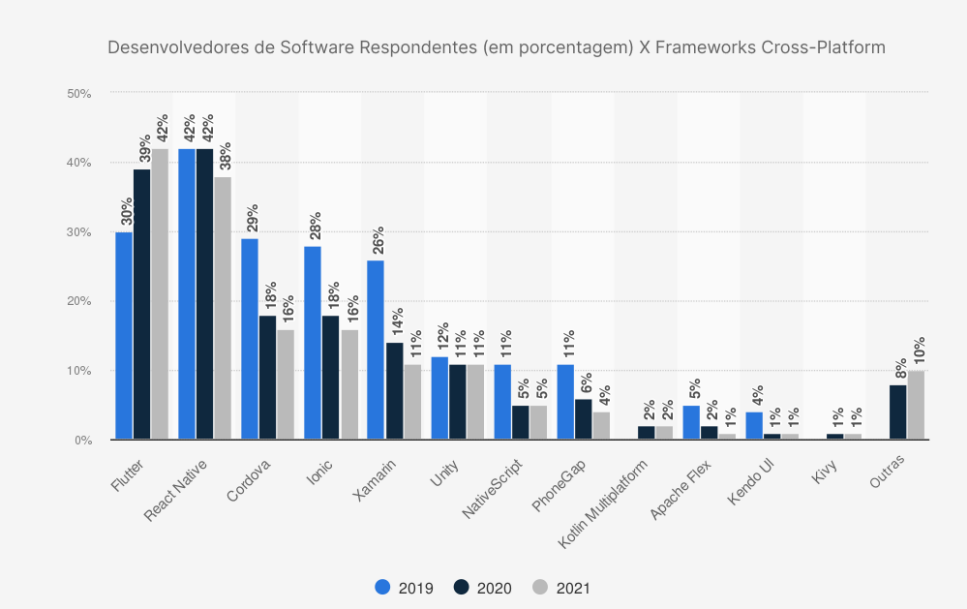
- Manutenção e implantação simples: Por se tratar de somente uma aplicação, é muito mais simples de identificar *bugs*, ajustá-los, e então enviar a correção para todos os dispositivos independentes do sistema operacional e suas versões (SRIVASTAVA, 2022);

- Desenvolvimento inicial veloz: Por se tratar de somente uma aplicação sendo desenvolvida, o esforço da equipe é reduzido entre 50% e 80%, pois todo o time está focado em um único objetivo (SRIVASTAVA, 2022), e

- Reutilização de código: Devido ao desenvolvimento de somente uma aplicação, o código pode ser 100% reaproveitado para todos os sistemas operacionais (SRIVASTAVA, 2022).

Visando acordar sobre a popularidade dos principais *Frameworks Cross-Platform* disponíveis no mercado, a empresa Jetbrains realizou uma pesquisa, de âmbito mundial, com uma amostragem de 31.743 desenvolvedores de *software*, entre os anos de 2019 e 2021, cujos dados estão apresentados na Figura 7 (JETBRAINS, 2021). Ressalta-se sobre a popularidade dos *frameworks* Fluter e React Native.

Figura 7 – *Frameworks Cross-Platform* mais usadas entre os anos de 2019 e 2021



FONTE: (VAILSHERY, 2022)

2.3 Arquitetura de *Software* para Desenvolvimento *Mobile*

Tendo como base a área de interesse desse trabalho, Arquitetura de *Software*, e o Desenvolvimento *Mobile*, cabe conferir uma visão geral do contexto de desenvolvimento de aplicativos móveis centrado nas principais arquiteturas de *software*, conforme as próximas seções, com destaque para: o padrão arquitetural MVC; a arquitetura Portas e Adaptadores, e a arquitetura Limpa.

2.3.1 Visão Geral

A evolução das capacidades de *hardware* em dispositivos móveis, seja em termos de memória, processamento ou outros, permitiu maior uso dos celulares, e portanto, maior uso de aplicativos em diversos setores: recreativos, corporativos, e estatais (MENDONÇA; BITTAR; DIAS, 2011). Em vários desses setores, há, inclusive, aumento de produtividade.

Todavia, ainda é possível perceber limitações em dispositivos móveis, em especial quanto à capacidade de processamento (SILVA; MACHARET; TEIXEIRA, 2008) e à disponibilidade de recursos (ex. tamanho de tela, bateria limitada, diferentes entradas e saídas) (RIBEIRO, 2006).

Diante do contexto apresentado, no qual constam vários desafios, justificam-se estudos exploratórios envolvendo arquitetura e desenvolvimento *mobile*, sendo essa a área de interesse foco do presente trabalho. Em alguns cenários de uso, para aplicativos que consomem uma grande quantidade de recursos computacionais, faz-se necessário, por exemplo, realizar processamentos em servidores, nos quais tais processos podem ser desencadeados através de chamadas do cliente (LA; KIM, 2012). Com uma arquitetura adequada, aplicativos de alta complexidade podem ser desenvolvidos com satisfatórios desempenho e consumo de recursos como memória, bateria, banda e outros (LA; KIM, 2012). Além de levar em consideração recursos de *hardware*, uma boa arquitetura deve permitir escalabilidade e facilidade de mudanças, conforme já discutido anteriormente, na seção 2.1 deste capítulo.

Todavia, essa modalidade de desenvolvimento é relativamente recente quando comparada a outras modalidades como, por exemplo, desenvolvimento *web*. Somado a esse fato, há evoluções constantes na área de Arquitetura de Software, acordando novas propostas arquiteturais. Nesse sentido, estudos neste tema, apesar de crescentes, ainda são, relativamente, reduzidos.

O estudo abordado por esse trabalho compreende apresentar alguns padrões arquiteturais, os quais são reconhecidos na comunidade de *software*, e aplicáveis ao desenvolvimento *mobile*.

2.3.2 Arquitetura MVC

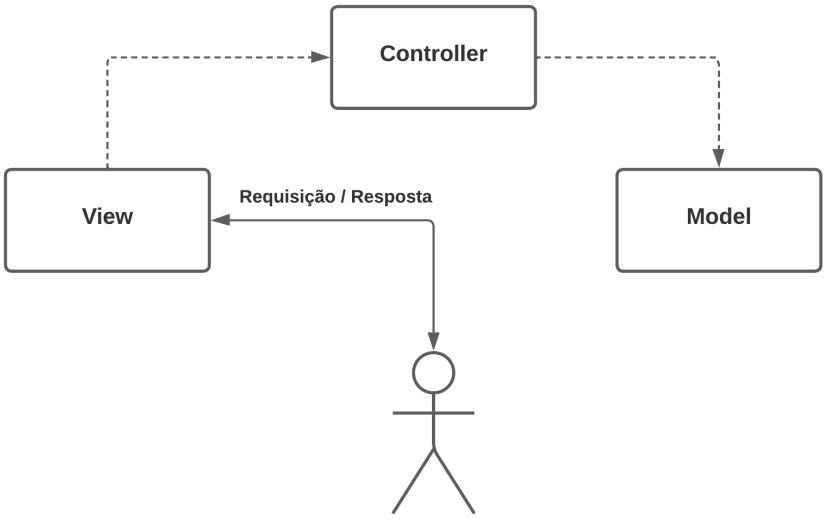
O padrão arquitetural *Model-View-Controller*, conhecido pela sigla MVC, surgiu em meados de 1970, e foi proposto pelo professor e cientista da computação, Trygve Reenskaug (REENSKAUG, 1979). A proposta surgiu com o intuito de separar a lógica de negócio da interface do usuário.

Como o próprio nome do padrão já indica, é uma arquitetura constituída por três camadas distintas, as quais se comunicam de forma harmoniosa entre si. Cada uma das camadas tem a seguinte representação:

- *Model*: Representa a lógica de negócio da aplicação, sendo a "camada", dentre as três, de mais baixo nível da aplicação. Processa dados recebidos e requisitados da aplicação (REENSKAUG, 1979);

- *View*: Representa a interface gráfica, sendo a única camada que se comunica de forma direta com o usuário (REENSKAUG, 1979), e
- *Controller*: Responsável por realizar o intermédio entre a *View* e a *Model*, sendo a camada que provê sentido aos dados que estão transitando na aplicação. Funciona como um estrutura intermediária, visando desacoplar as camadas *View* e *Model*, e possibilitando uma manutenção mais facilitada (REENSKAUG, 1979).

Figura 8 – Interações do MVC orientando-se por arquitetura do tipo restrita



Fonte: Autores.

Para uma melhor compreensão do padrão arquitetural MVC, será acordado um exemplo, orientando-se pelo exposto na Figura 8:

Considerando um usuário preenchendo um formulário *online*, após concluir o preenchimento, a requisição é encaminhada para uma *Controller*. A *Controller*, por sua vez, tem como objetivo identificar qual *Model* deve processar os dados enviados pelo usuário, via requisição. Assim que a *Model* for identificada, os dados são encaminhados para a mesma, e as informações são processadas. Os retornos, entre as camadas, orientando-se por uma Arquitetura do tipo Restrita (PATRIARCHA, 2018), ocorrem via padrões de projeto, tal como *Observer*, usando como base Orientação a Eventos, até o Cliente receber a resposta. Dessa forma, optou-se pela representação de dependências, das classes superiores para as classes inferiores, sem representar de forma explícita o sentido inverso, realizado via eventos e notificações.

Vantagens

O MVC está entre os padrões arquiteturais mais reconhecidos. O mesmo é bastante utilizado por conta dos benefícios que são proporcionados durante o ciclo de desenvolvimento, dentre eles:

- Alta coesão e baixo acoplamento: Por conta da alta coesão e do baixo acoplamento que o MVC proporciona, os desenvolvedores podem trabalhar, por exemplo, na lógica de negócio de uma *Model*, sem que haja a necessidade de alterar sequer uma única linha de código na *Controller* ou na *View* (GRAÇA, 2017a);
- Reaproveitamento de código: Com o MVC, também é possível reutilização de *software*, o que evita, por exemplo, a replicação de código (POPE, 2010). A mesma *Model* pode estar relacionada a distintas *Views*, e
- Manutenibilidade e escalabilidade: Outros benefícios do padrão, como consequência do baixo acoplamento e da alta coesão, a melhor manutenibilidade da arquitetura, bem como a alta escalabilidade que a mesma proporciona. Com o MVC, por exemplo, não importa o quão grande é a aplicação, há certo controle quanto à complexidade de se realizar alterações no código (FREEMAN; SANDERSON, 2011).

Em resumo, e com base na literatura mencionada, uma das principais vantagens do MVC consiste na facilidade de alterar código que já está pronto, permitindo a evolução do mesmo, mesmo envolvendo questões de escalabilidade da aplicação, sem que seja algo muito penoso para os desenvolvedores.

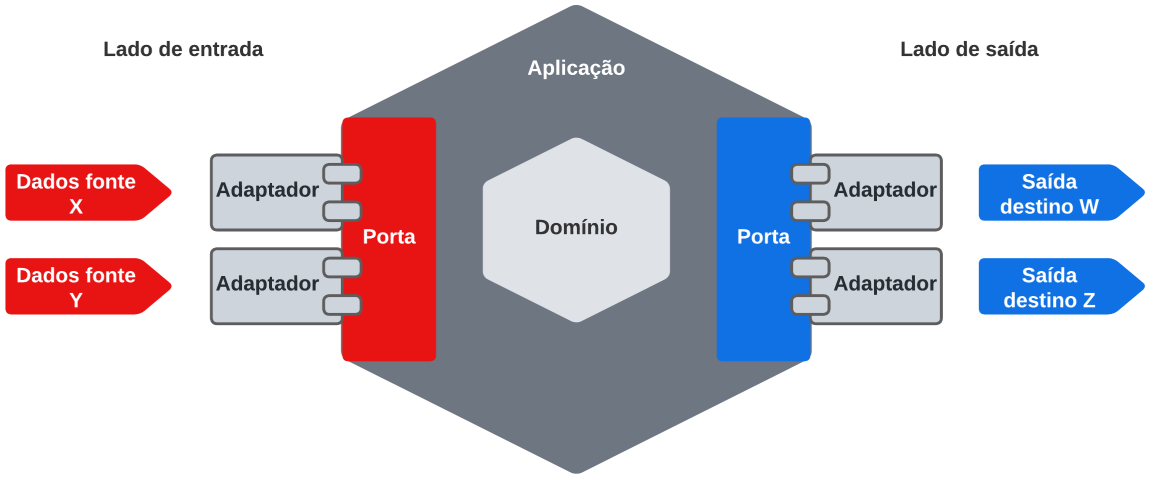
2.3.3 Arquitetura Portas e Adaptadores

O padrão arquitetural Portas e Adaptadores, também conhecido como arquitetura Hexagonal, foi proposto pelo cientista da computação Alistair Cockburn, e documentado no *blog* do mesmo (GRAÇA, 2017b), em 2005. Para Alistair Cockburn, o objetivo principal desse padrão arquitetural é:

> Permitir que uma aplicação seja igualmente desenvolvida para os usuários, programas, testes automatizados ou *scripts bash*, e para ser desenvolvida e testada de forma isolada dos seus eventuais dispositivos e bancos de dados de produção. (COCKBURN, 2005)

Sendo assim, a arquitetura Portas e Adaptadores deve ter sua lógica de negócio totalmente isolada dos componentes externos (BAILÉN, 2019). Para atingir esse objetivo, Cockburn propôs uma estrutura específica, conforme ilustrado na Figura 9.

Figura 9 – Representação da arquitetura Portas e Adaptadores



Fonte: Autores.

Como pode ser visto na Figura 9, a estrutura possui dois hexágonos sobrepondo-se, e um círculo ao redor. As abstrações compreendidas na estrutura representam as seguintes camadas:

- Domínio: Parte de mais baixo nível da aplicação, representando a lógica de negócio da mesma;
- Aplicação: Aplicação em si, podendo ser uma API, um *Frontend* ou qualquer tipo de aplicação, e
- Infraestrutura: Parte mais externa da arquitetura, responsável por receber ou enviar dados para a aplicação. Trata-se de uma camada inerente a qualquer tipo de tecnologia, fora do escopo da arquitetura.

Portas podem ser conceituadas como elementos de entrada e saída de dados na camada de aplicação. São utilizadas para limitar o tipo de dado que irá transitar naquela porta (GRAÇA, 2017b). Em uma analogia, as Portas podem ser vistas como entradas USB, permitindo que diversos tipos de dispositivos sejam conectados, contanto que os dispositivos possuam um adaptador de USB. Nesse sentido, há possibilidade de conexão com diversos serviços externos à aplicação, como por exemplo, conectar um banco de dados (MARTINEZ, 2021).

Os Adaptadores interagem com as Portas tanto para validar se o formato que está sendo mandado para a camada de aplicação está correto, chamados de Adaptadores de entrada; quanto para implementar o formato que a Porta exige que os dados sejam criados para os serviços externos à aplicação, chamados de Adaptadores de saída (GRAÇA, 2017b).

Na Figura 9, os Adaptadores de entrada e saída estão representados pelas cores vermelho e azul, respectivamente.

Vantagens

Segundo os autores mencionados a seguir, a arquitetura Portas e Adaptadores possui vantagens, dentre elas:

- Independência de componentes externos: Devido à forma como a arquitetura é proposta, a lógica do sistema pode funcionar independentemente do comportamento de agentes externos (BROWN, 2014);
- Simplicidade na realização de testes: Como consequência da arquitetura ser independente de agentes externos, é possível realizar testes de cada camada. Além disso, é possível realizar um *mock* de qualquer dependência externa (MARTINEZ, 2021). *Mocks* são objetos utilizados para "simular" interações de saída com dependências externas ao teste. Tais interações comportam-se como chamadas, realizadas pelo teste, visando mudança de estados (JUNG, 2019). Portanto, o uso de *mocks* facilita o controle e a inspeção de chamadas para dependências externas;
- Flexibilidade de Portas e Adaptadores: A arquitetura permite que a estrutura de uma Porta seja usada por diversos Adaptadores, com funções distintas e para serviços distintos (BROWN, 2014);
- Acoplamento baixo: Mudanças nas camadas mais externas não afetam as camadas internas. Camadas internas não possuem conhecimento da existência das camadas externas (BROWN, 2014), e
- Independência de tecnologias: Por ser uma arquitetura genérica a qualquer caso de uso, não importa a tecnologia usada, é possível realizar a troca de banco de dados, por exemplo, sem prejuízo ao projeto (BROWN, 2014).

Percebe-se, com base na literatura acordada anteriormente, que boa parte das vantagens da arquitetura Portas e Adaptadores advém do seu perfil generalista, independente de serviços externos. Assim, a aplicação pode se adaptar a diversos cenários de uso, mesmo que os mesmos não sejam relacionados.

2.3.4 Limpa

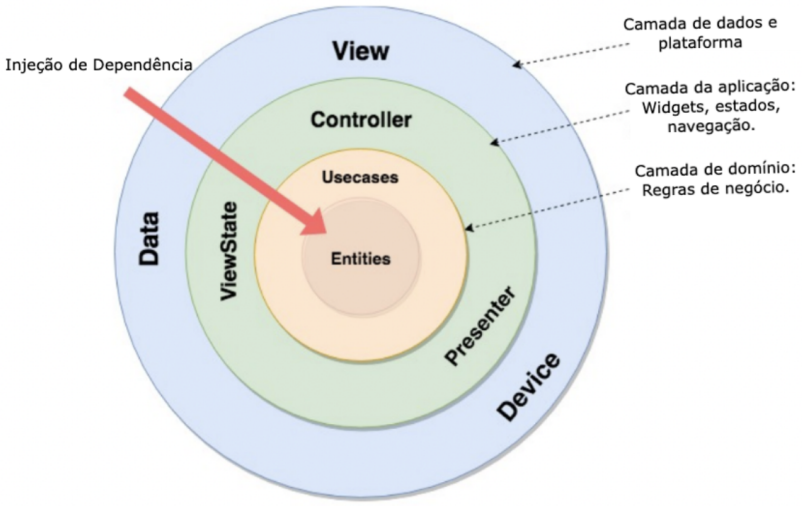
A arquitetura limpa foi introduzida por Robert C. Martin em seu livro (traduzido) "Arquitetura Limpa, O guia do Artesão para Estrutura e *Design* de *Software*", publicado em Março de 2019. Em sua publicação, "Uncle Bob", como é carinhosamente chamado por seus seguidores, reúne algumas práticas de *Design* de *software* que impulsionam a

criação de um projeto de *software* "limpo", isto é, de fácil entendimento, fácil modificação e escalável (MARTIN, 2017).

Conforme ilustrado na Figura 10, essa arquitetura baseia-se em camadas, as quais apresentam responsabilidades específicas na solução. São elas:

- Camada de domínio: Camada mais interna, na qual são reunidas as regras de negócio e as entidades que compõem a mesma. Essa é a camada que, via de regra, menos sofre alteração em uma aplicação. Isso ocorre, pois, em um projeto bem planejado, alterações nas regras de negócio são mínimas;
- Camada da aplicação: Camada intermediária, na qual se encontram *widgets*, estados e navegação, e
- Camada de dados ou infraestrutura: Camada responsável pela comunicação com o mundo exterior, na qual constam, por exemplo, *drivers* de banco de dados e chamadas a serviços externos.

Figura 10 – Representação das camadas na arquitetura Limpa



Fonte: Autores.

Pode-se notar diversos aspectos em comum entre a Arquitetura Limpa e a Arquitetura Portas e Adaptadores. Ambas trabalham com o conceito de camadas, as quais possuem responsabilidades próprias. Dentre os conceitos relevantes, e que regem as arquiteturas de natureza similar, cabe ressaltar sobre a dependência. Basicamente, camadas mais internas não podem ser dependentes das mais externas, ou seja, a dependência sempre aponta para a camada mais interna, sendo a central a mais independente entre elas. A comunicação entre as camadas dá-se por Injeção de Dependência, sendo essa uma boa prática de projeto, já descrita neste capítulo. Em resumo, a camada mais interna não está ciente de detalhes das camadas mais externas.

Vantagens

Há benefícios, acordados pela literatura, sobre a arquitetura Limpa, dentre eles (BOUKHARY; COLMENARES, 2019):

- Independência de *Framework*: Concentrar as regras de negócio na camada mais central permite ter independência em relação aos *frameworks*. Sendo assim, qualquer alteração de *framework*s não afeta qualquer funcionalidade que constitui a parte *core* da aplicação;
- Testabilidade: há facilidade na realização de testes em uma aplicação que se orienta por uma arquitetura Limpa, uma vez que seus módulos, separados em camadas, permitem testes isolados de cada parte da aplicação, bem como a simulação de camadas externas para o teste. Novamente, assim como na arquitetura Portas e Adaptadores, há facilidade na criação de *mocks* para testes, nas camadas que estão sendo testadas, e
- Isolamento de serviços externos: Serviços externos à aplicação, que se comunicam com o exterior, encontram-se na camada mais externa. Sendo assim, as regras de negócio ficam isoladas, e inertes à qualquer mudança desse meio externo. Tal isolamento permite facilidades na alteração de qualquer serviço, sem impactar outros componentes.

Conforme a literatura mencionada anteriormente, percebe-se que uma característica bastante evidente na arquitetura Limpa é a possibilidade de conferir mudanças, evitando efeitos colaterais, em especial, nas regras de negócio. Trocar módulos, bibliotecas e *frameworks* são bem menos custosos, devido à natureza de divisão de responsabilidades de cada camada. Dessa forma, essa arquitetura isola partes da aplicação mais estáveis daquelas mais passíveis de mudança, o que melhora a estruturação dos componentes arquiteturais, e diminui riscos de "quebra" em módulos não correlacionados.

2.4 Testes de Integração

Conforme exposto anteriormente, esse trabalho compreende revelar e reportar sobre os principais comportamentos dos componentes arquiteturais de um *software mobile*, considerando diferentes arquiteturas. Entretanto, essa revelação precisa ser apoiada em insumos mais concretos, visando mostrar essa parte comportamental de forma clara e objetiva. Nesse sentido, optou-se pelo uso de Testes de *Software*, mais especificamente, de Testes de Integração.

O processo de realização de testes de *software* baseia-se em identificar se o produto final realiza aquilo que o mesmo se propôs a fazer. Os testes, ao serem executados seguindo

boas práticas e técnicas, trazem diversos benefícios, tanto ao produto quanto ao ciclo de vida do projeto, como por exemplo: a mitigação de *bugs*, redução do tempo de manutenção e de custos (IBM, 2022). A realização de testes é de extrema importância no desenvolvimento de qualquer *software*. Os testes têm como propósito garantir que o produto cumpra com todos os requisitos pré estabelecidos, sua robustez e metrificar a qualidade do sistema (HAMILTON, 2022).

Para testar uma aplicação por completo, podem ser realizados diversos tipos de testes para inúmeros casos de uso. Mesmo um software relativamente simples pode possuir uma imensa variedade de testes, compreendendo desde testes unitários, os quais testam cada função do código fonte; até testes de usabilidade com o cliente final, sem necessidade alguma do conhecimento do código original (IBM, 2022).

Dentre os inúmeros tipos de testes existentes, os que se destacam na avaliação do comportamento entre os componentes de um sistema são os testes de integração. Os testes de integração garantem a boa operabilidade do funcionamento dos componentes de um software de forma integrada (IBM, 2022).

O testes de integração tem como objetivo identificar se a comunicação entre os módulos de um sistema está ocorrendo como deveria (KRIGER, 2021). Esses testes procuram garantir se os dados estão transitando normalmente entre as camadas de uma arquitetura de um sistema (KRIGER, 2021). No escopo dos testes de integração, existem quatro técnicas distintas de testes, que serão descritas de forma breve a seguir:

- *Bottom-Up*: O teste é iniciado entre camadas de mais baixo nível a camadas de mais alto nível de abstração (KRIGER, 2021). Trata-se de uma excelente abordagem para verificar de antemão comportamentos específicos na camada de baixo nível (KRIGER, 2021);

- *Top-Down*: O teste inicia-se por camadas de alto nível, progredindo para as camadas de mais baixo nível de abstração do sistema. Um "esqueleto" dos módulos é criado e então integrado, aos poucos, com componentes do sistema (KRIGER, 2021). A utilização de testes *Top-Down* permite que as principais funções do código sejam previamente testadas;

- Híbrida: A abordagem híbrida, ou *Sandwich*, é um misto entre as técnicas *Bottom-Up* e *Top-Down* (AWATI, 2022). Utiliza-se da técnica de *Top-Down* para testar as principais funções do *software*, e então, para os módulos restantes, utiliza-se o *Bottom-Up* (KRIGER, 2021). Essa é uma abordagem recomendada para produtos de *softwares* grandes e complexos (AWATI, 2022), e

- *Big Bang*: A técnica *Big Bang* consiste em testar, previamente, cada modulo e, então, integra-los de uma só vez (KRIGER, 2021). Essa técnica pode causar certos

problemas durante sua implementação, pois, por não serem realizados de forma incremental como as outras técnicas, podem surgir diversos erros, de difícil solução, entre as interfaces dos módulos (KRIGER, 2021). A abordagem *Big Bang* é mais bem aproveitada em produtos de software *softwares* mais simples (AWATI, 2022).

No presente trabalho, faz o uso da técnica *Top-Down*, uma vez que são testado primeiro as funcionalidades principais e, então se necessário, é testado o resto dos componentes de forma unitária.

2.5 Resumo do Capítulo

O capítulo apresentou conceitos teóricos primordiais a respeito de *Arquitetura de Software*, Desenvolvimento *Mobile* e testes de integração que fundamentam os principais pontos de pesquisa deste trabalho. Inicialmente, foram apresentados aspectos gerais sobre arquitetura, bem como seu conceito, situações que demonstram quando a arquitetura de um sistema não foi projetada de forma adequada, e princípios relevantes que constituem uma arquitetura apropriada.

Adicionalmente, foram apresentados o atual contexto que concerne o desenvolvimento *mobile*, e os principais *frameworks* de desenvolvimento híbrido disponíveis no mercado.

O capítulo segue com maior detalhamento sobre algumas das principais Arquiteturas consolidadas no mercado: MVC, Arquitetura Portas e Adaptadores e Arquitetura Limpa, apresentando breves introduções e as principais vantagens de cada. Ao longo do capítulo, não foram acordadas desvantagens. Isso ocorre, pois, na literatura, esse tipo de apontamento, normalmente, é colocado de forma superficial e generalista, com menção, principalmente, dos fatores, sendo: necessidade de maior conhecimento e capacitação técnica para implementação dessas arquiteturais, em especial, Portas e Adaptadores e Limpa; e uso excessivo de classes adicionais, o que incorre em maior dedicação para conquista de um código realmente generalista e desacoplado das tecnologias. A intenção desse estudo é justamente conferir insumos mais concretos sobre os comportamentos dessas arquiteturas, no desenvolvimento de aplicativos móveis, com base, por exemplo, em métricas de análise estática de código e realização de testes de integração, conforme acordado no próximo capítulo.

Por fim, também foi conferido um breve detalhamento sobre testes de integração.

3 Suporte Tecnológico

Este capítulo tem como intuito apresentar as principais ferramentas e tecnologias utilizadas para a realização desse trabalho. Em um primeiro momento, será descrito o *framework* React Native, utilizado para desenvolvimento de aplicações híbridas, ou seja, que sejam funcionais em diferentes sistemas operacionais. Na sequência, é abordada a plataforma de desenvolvimento Firebase, oferecida pelo Google, e que confere recursos complementares para o desenvolvimento de aplicativos móveis, aplicações web e outros. O SonarQube, ferramenta para coletar métricas do código, será coberto também, uma vez que o mesmo foi utilizado para revelar aspectos comportamentais observáveis, no contexto de aplicativos móveis, ao se aplicar arquiteturas de *software* conhecidas em determinados cenários de uso. O Expo, que facilita o desenvolvimento de aplicações em React Native. Há ainda o Jest que é um *framework* para realização de testes em JavaScript. Por fim, tem-se o resumo do capítulo.

3.1 React Native

O React Native é um *framework open-source* desenvolvido em JavaScript, que possui suporte ao TypeScript, um transpilado do JavaScript que adiciona tipagem estática à linguagem, e foi concebido pelo Facebook e utilizado para a criação de aplicações híbridas (PATERSKA, 2021). Atualmente, o *framework* em JavaScript vem, aos poucos, perdendo relevância para o Flutter, porém ainda possui uma grande popularidade na comunidade mundial de *software*, como pode ser visto na Figura 7, mostrada anteriormente, nessa monografia. Devido ao React Native ter liderado em termos de popularidade por anos, muitas das aplicações utilizadas atualmente foram desenvolvidas com o *framework*.

Esse *framework* interpreta código escrito em JavaScript e o converte para código nativo, de forma assíncrona, ou seja, todo e qualquer componente renderizado em tela é do próprio sistema e não do React Native. Esse comportamento de se comunicar com o código nativo de cada sistema operacional ocorre orientando-se pelo padrão de projeto Bridge. Tal padrão confere a flexibilidade que o *framework* possui (PATERSKA, 2021). O Bridge é um padrão de projeto estrutural, que divide a lógica da aplicação em duas hierarquias principais: Abstração e Implementação. Abstração é a hierarquia de mais alto nível, a qual mantém uma referência para um objeto da Implementação. É responsabilidade da Abstração delegar trabalho para a hierarquia de Implementação. Já a hierarquia de Implementação pode conter níveis mais genéricos, porém deve ser preparada, especializada, para se comunicar com a hierarquia de alto nível (SHVETS, 2022).

Além das diversas vantagens que *Frameworks* híbridos possuem, o React Native,

por conta de sua arquitetura ser construída com o padrão de projeto Bridge, é bastante reconhecido em termos de desempenho. Isso ocorre, pois todos os componentes renderizados em tela são do próprio sistema operacional. Entretanto, não possui desempenho melhor do que aplicações 100% nativas (PATERSKA, 2021).

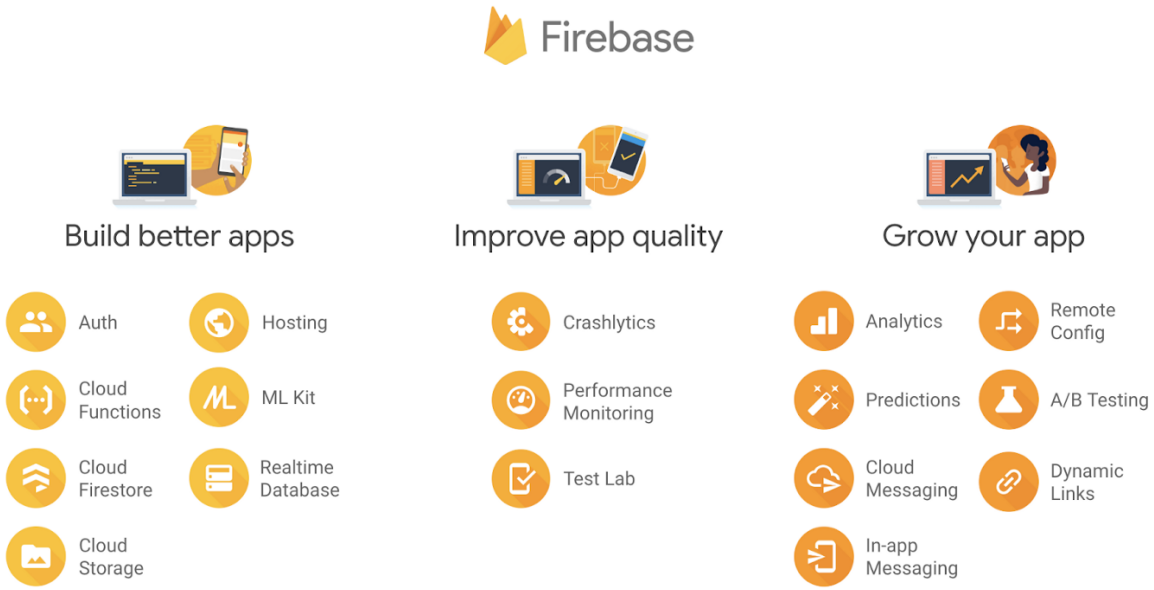
Outra vantagem do React Native em relação aos outros *Frameworks* do mercado é o tamanho de sua comunidade. A comunidade do React Native está em constante evolução, sempre há novos *updates* com novas *features*, correções de *bugs* e inúmeros *plugins* em JavaScript, que podem ser usados para facilitar o desenvolvimento. Adicionalmente, cabe ressaltar que o Facebook participa de forma direta na evolução do *Framework*(PATERSKA, 2021).

3.2 Firebase

Firebase é uma plataforma de desenvolvimento de aplicações, mantida pelo Google, e que disponibiliza diversas ferramentas em atendimento ao desenvolvimento de soluções em diversas áreas, tais como: aplicativos móveis, *websites* e jogos (FIREBASE, 2022).

A vasta possibilidade de criação que a plataforma oferece popularizou o Firebase como uma das ferramentas mais utilizadas para desenvolvimento *mobile* do mercado. Cerca de 3 milhões de aplicativos utilizaram o Firebase como solução de desenvolvimento em 2021 (Google I/O, 2021).

Figura 11 – Ferramentas disponíveis para uso com Firebase.



FONTE: (STEVENSON, 2018)

A Figura 11 mostra algumas das principais ferramentas de desenvolvimento ofere-

cidas pela plataforma. Recursos como Firestore oferecem serviços de banco de dados não relacionais, os quais possuem funcionalidades como monitoramento em tempo real, algo que torna o desenvolvimento de aplicativos com funcionalidade de conversa, por exemplo, muito mais simples.

Além disso, dentre os principais recursos disponíveis, vale destacar o Firebase Auth, que oferece serviços de autenticação, inclusive de *logins* sociais. Há ainda o Analytics, sendo esse um serviço útil para coletar métricas. Com ele, pode-se monitorar ações dos usuários, o que possibilita a realização de um estudo detalhado de como eles interagem com a aplicação.

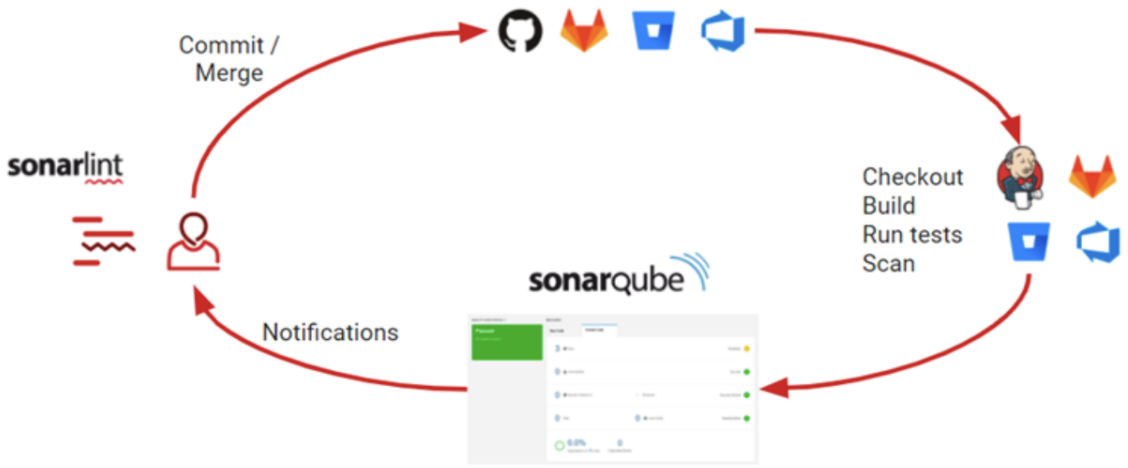
São diversos os serviços oferecidos pelo Firebase. Todos possuem um limite de uso gratuito. Quando tal limite é atingido, os valores a serem pagos para uso dos serviços variam de acordo com o uso dos serviços em demanda. Esses recursos estão disponíveis via SDK para diversas linguagens e *frameworks*, tais como: NodeJs, Python, Flutter, e outros.

3.3 SonarQube

O SonarQube é uma ferramenta de análise estática de código amplamente utilizada por desenvolvedores para coleta de métricas de qualidade. Seu uso possibilita a detecção de *bugs*, vulnerabilidades e *code smells* de forma eficiente (SonarQube, 2022).

Essa ferramenta possui mecanismos de integração contínua, que torna possível sua inclusão e *workflows*. Com isso, há a possibilidade de checagens periódicas automatizadas, em *branches* e *pull requests*.

Figura 12 – Exemplo de integração contínua utilizando-se o SonarQube.



FONTE: (SonarQube, 2022)

A Figura 12 é uma demonstração de um *workflow* de trabalho utilizando a ferramenta. Basicamente, tem-se uma subdivisão em três etapas:

1. Desenvolvedores criam seus códigos e integram com a *branch* de desenvolvimento. Dentro do processo, pode-se usar alguma ferramenta como o SonarLint, que consiste em uma extensão para IDE, a qual promove *feedback* de qualidade de código durante o processo de desenvolvimento (SonarQube, 2022);

2. A ferramenta de integração contínua de escolha do desenvolvedor executa os passos, principalmente: processar a *build*, executar testes unitários e escanear código, utilizando-se do SonarQube, e

3. A ferramenta agrupa as métricas obtidas e as envia para um servidor do SonarQube, onde ficam disponíveis para consulta por parte dos desenvolvedores.

A coleta de métricas de qualidade é uma prática relevante para que se avalie periodicamente a saúde do código de um dado projeto. Com isso, ajustes de erros e prevenção contra problemas de vulnerabilidades tendem a ser tornar descomplicados e de fácil desenvolvimento.

3.4 Expo

O Expo é uma ferramenta criada para facilitar o desenvolvimento de aplicações em React Native. Esse *bundle* permite com que seja possível executar aplicações em React Native sem a necessidade de instalar ferramentas de compilação de código nativo (XCode e Android Studio). É uma forma mais enxuta, rápida e simples de configurar o ambiente de desenvolvimento (FERNANDES, 2018).

3.5 Jest

O Jest é um *framework* com foco na realização de testes em JavaScript. Esse *framework* permite escrever testes de forma rápida e sem a necessidade de realizar inúmeras configurações (FACEBOOK, 2022b). O Jest possui diversas ferramentas, o que torna o *framework* bastante adequado, e uma documentação bem completa sobre cada uma de suas funções (FACEBOOK, 2022b).

3.6 Redux

O Redux é uma biblioteca de manipulação de estados da interface de usuários disponível para *frameworks* construídos em JavaScript (ABRAMOV, 2021). Um exemplo é o próprio React Native. A biblioteca é capaz de tornar aplicações, que utilizam a mesma, mais veloz por conta das diversas otimizações para prevenir renderizações desnecessárias,

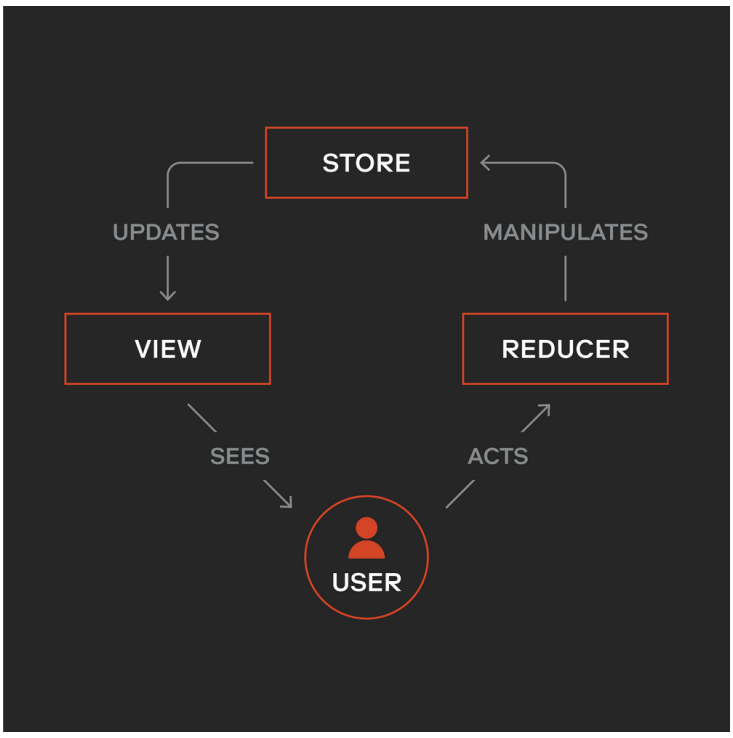
além de permitir agrupar toda a lógica de manipulação de estados em um local somente (ABRAMOV, 2021).

A arquitetura do Redux é extremamente parecida com o MVC (BAUMGARTNER, 2021). A biblioteca funciona a partir das seguintes estruturas:

- *Reducer*: É chamado por uma ação realizada pelo usuário, equivalente à *Controller*, manipulando os dados para serem salvos na *Store* (BAUMGARTNER, 2021);
- *Store*: Camada equivalente à *Model*, armazenando todos os dados que manipulam a aplicação, e permanecendo a lógica de negócio. Só é possível ser acessada essa camada a partir da camada *Reducer* (BAUMGARTNER, 2021), e
- *View*: Interface de usuário em si, onde tudo é manipulado pelos dados da *Store*.

Como pode ser observado na Figura 13, há similaridade entre a arquitetura MVC e a arquitetura utilizada em aplicativos que implementam o Redux como a gerência de estado. Assim, o *Reducer* é comparável a uma *Controller*; a *Store* à *Model*, e a *View* equivalente à *View* no modelo MVC.

Figura 13 – Arquitetura do Redux.



FONTE: (BAUMGARTNER, 2021)

3.7 Codecov

Codecov é uma ferramenta de coleta de métricas de testes, tais como: Porcentagem total de código testado em um repositório e histórico da porcentagem dos testes realizados. Ele ajuda a analisar, de forma mais intuitiva e visual, alguns parâmetros como cobertura de testes por linhas de código, arquivos e assim por diante, permitindo as empresas entregarem um código de qualidade (CODECOV, 2022).

3.8 Resumo do Capítulo

Esse capítulo introduziu as principais ferramentas e tecnologias que foram utilizadas para viabilizar o trabalho, em termos de desenvolvimento bem como de coleta de métricas/dados visando responder a questão de pesquisa. Há detalhamentos sobre o *framework* React Native, o Firebase, o SonarQube, o Expo, o Jest e, por fim, Redux. No intuito de conferir um resumo sobre os principais apoios tecnológicos, segue a Tabela 1 e Tabela 2.

Tabela 1 – Principais tecnologias adotadas no trabalho
parte 1

Nome	**Descrição**	**Link**
React Native	*Framework* para criação de aplicações híbridas.	React Native ou https://reactnative.dev/
Firebase	Ferramentas e serviços para o desenvolvimento de aplicações em nuvem oferecido pelo Google.	Firebase ou https://firebase.google.com/
SonarQube	Ferramenta de análise estática de código. Fornece uma analise completa da qualidade do código fonte desenvolvido, cobertura de testes e avisa sobre possíveis ameaças.	Sonarqube ou https://www.sonarqube.org/
Jest	*Framework* para realizar testes em código JavaScript.	Jest ou https://jestjs.io/pt-BR/

Tabela 2 – Principais tecnologias adotadas no trabalho
parte 2

Nome	**Descrição**	**Link**
Redux	Biblioteca para manipular estados em *frameworks* JavaScript.	Redux ou https://react-redux.js.org/
Codecov	Ferramenta de coleta de cobertura de testes.	Codedov ou https://docs.codecov.com/docs/
LaTex	Linguagem de marcação, assim como o HTML, porém voltado para escrita de artigos.	LaTex ou https://www.latex-project.org/
Overleaf	Ferramenta *online* de edição de texto no formato LaTex.	Overleaf ou https://pt.overleaf.com/
Slack	Ferramenta de comunicação com foco em facilitar a comunicação empresarial, separando assuntos distintos em canais.	Slack ou https://slack.com/
Telegram	Ferramenta de comunicação com foco na velocidade de entrega das mensagens.	Telegram ou https://telegram.org/
Draw.io	Ferramenta gratuita para modelagem de diagramas colaborativos.	Draw.io ou https://drawio-app.com/
Figma	Ferramenta para a modelagem de protótipos colaborativos.	Figma ou https://www.figma.com/
Trello	Ferramenta visual para o gerenciamento de times e projetos.	Trello ou https://trello.com/

Por fim, cabe mencionar que serão utilizadas, adicionalmente, ferramentas para edição de texto (i.e. Latex e Overleaf); comunicação entre os membros envolvidos no projeto (i.e. Slack e Telegram), elaboração de artefatos de modelagem (i.e. Figma e Draw.io), e

condução de atividades gerenciais (i.e. Trello).

4 Metodologia

Neste capítulo, são detalhados aspectos metodológicos, os quais guiam as principais atividades do trabalho. Sendo assim, ocorre menção à classificação da pesquisa, considerando abordagem, natureza, objetivos e procedimentos. São apresentados, na sequência, o fluxo de atividades desse projeto, bem como as respectivas metodologias adotadas para viabilizar investigação junto à literatura especializada, desenvolvimento e análise de resultados. Depois, cronograma, conferindo detalhamento temporal às atividades compreendidas no TCC. Por fim, tem-se o resumo do capítulo.

4.1 Classificação da Pesquisa

Uma pesquisa, em termos metodológicos, pode ser classificada em relação à abordagem, à natureza, aos objetivos e aos procedimentos (GERHARDT; SILVEIRA, 2009). Sendo assim, os tópicos a seguir buscam detalhar cada aspecto de acordo com tal classificação.

4.1.1 Abordagem

A abordagem de pesquisa realizada por esse trabalho pode ser classificada como qualitativa e quantitativa. Determinadas métricas, tais como número de *bugs* identificados em uma dada implementação de arquitetura em um aplicativo móvel, correspondem a informações concretas, quantificáveis, e que conferiram insumos para que os comportamentos das arquiteturas fossem documentados, visando futuras análises (GERHARDT; SILVEIRA, 2009). Entretanto, há aspectos comportamentais mais subjetivos, ou seja, não facilmente quantificados, sendo tratados em termos qualitativos. Dentre esses aspectos, pode-se mencionar a experiência de desenvolvimento das aplicações para os autores e a facilidade para a realização dos testes de integração.

4.1.2 Natureza

A natureza desse projeto pode ser identificada como aplicada. Desta forma, buscou-se realizar implementações que aplicassem as arquiteturas de *software* em cenários de uso orientados ao desenvolvimento de aplicativos móveis (GERHARDT; SILVEIRA, 2009). Esse viés aplicado permitiu, dentre outras contribuições, expor os comportamentos de cada arquitetura, conferindo insumos que poderão ser observados e analisados pelos interessados nos estudos dessa pesquisa.

4.1.3 Objetivos

Quanto aos objetivos, a pesquisa pode ser classificada como exploratória. Sendo assim, há investigação, junto à literatura, visando maior conhecimento do domínio, além do desenvolvimento de provas de conceito, em cenários de uso, no intuito de explorar diferentes arquiteturas no contexto de desenvolvimento de aplicativos móveis, promovendo maior familiaridade com o problema abordado nesse trabalho (GERHARDT; SILVEIRA, 2009).

4.1.4 Procedimentos

Quanto aos procedimentos, pode-se mencionar: pesquisa bibliográfica, partindo para o levantamento de referências teóricas e práticas, já realizadas e publicadas na comunidade especializada (GERHARDT; SILVEIRA, 2009); e pesquisa-ação (KRAFTA et al., 2009), usando como base ciclos envolvendo as seguintes etapas: exploratória, planejamento, ação e avaliação. A pesquisa bibliográfica e a pesquisa-ação orientam-se, respectivamente, pela Metodologia Investigativa e pela Metodologia de Análise de Resultados, conforme exposto nas seções 4.3 e 4.5 adiante.

4.2 Fluxo de Atividades

Conforme ilustrado na Figura 14, são atividades e subprocessos relativos à primeira fase de elaboração do TCC:

- Definir o tema: Definir, com auxilio dos orientadores, um tema de estudo na área de Engenharia de *Software* para desenvolver a pesquisa. Dada a relevância da área, identificada com estudos preliminares na literatura, optou-se pelo tema "Arquitetura de *Software*: Um Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos". Status: Realizado;
- Levantamento bibliográfico: Subprocesso que visou a realização da pesquisa bibliográfica no tema definido, usando como orientação a Metodologia Investigativa, descrita na seção 4.3. Status: Realizado;
- Fechar escopo: Delimitar os principais aspectos contemplados no trabalho, nos quais se pretendeu contribuir de alguma forma. Nesse sentido, foram definidas arquiteturas específicas, sugeridas na literatura como pertinentes para o desenvolvimento de aplicativos móveis híbridos, sendo as mesmas objetos de estudo desse trabalho. Status: Realizado;
- Estabelecer referencial teórico: Definir os conceitos e demais detalhes teóricos envolvendo arquitetura de *software*, desenvolvimento *mobile* híbrido, arquitetura MVC,

arquitetura de Portas e Adaptadores, arquitetura Limpa, e Testes de Integração conforme consta no Referencial Teórico 2. Status: Realizado;

- Estabelecer suporte tecnológico: Definir as principais ferramentas e tecnologias para desenvolvimento da solução, conforme apresentado em Suporte Tecnológico 3. Status: Realizado;

- Especificar metodologia: Definição e apresentação dos detalhamentos metodológicos da pesquisa, conforme especificado no presente capítulo. Status: Realizado;

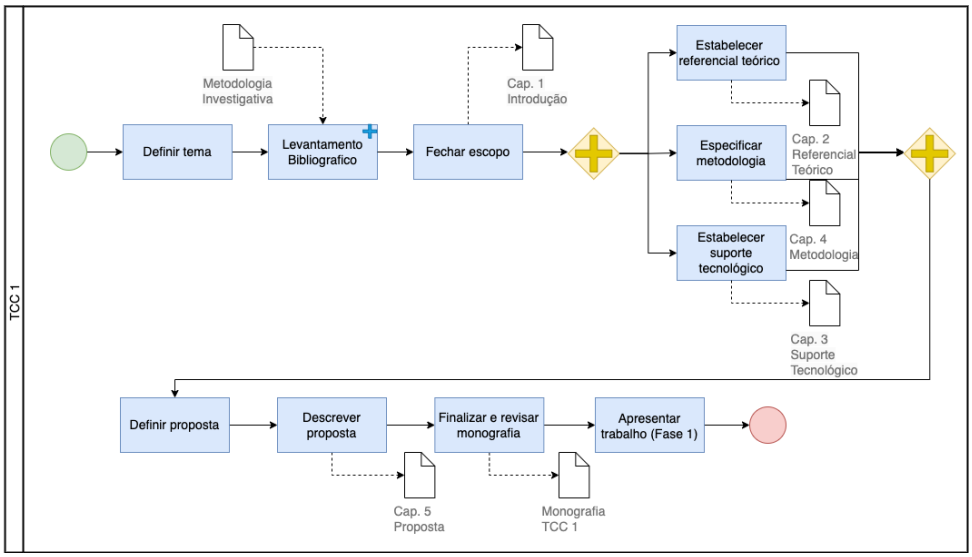
- Definir proposta: Abordar de forma clara os referenciais teórico e tecnológico de modo a estabelecer uma proposta interessante e agregadora em termos de contribuições. Status: Realizado;

- Descrever proposta: Detalhar mais claramente sobre a proposta, revelando quais arquiteturas de *software* pretendiam ser tratadas no escopo do trabalho; quais possíveis cenários de uso seriam pertinentes ao conhecimento exploratório do desenvolvimento de aplicativos móveis híbridos, dentre outros detalhamentos, os quais constam especificados em Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos 5. Status: Realizado;

- Finalizar e revisar monografia: Ajustes finais quanto à escrita e ao conteúdo entregues na primeira fase do trabalho. Status: Realizado, e

- Apresentar trabalho (Fase 1): Submissão e apresentação do trabalho aos membros da banca. Status: Realizado.

Figura 14 – Fluxo de atividades TCC (primeira etapa)

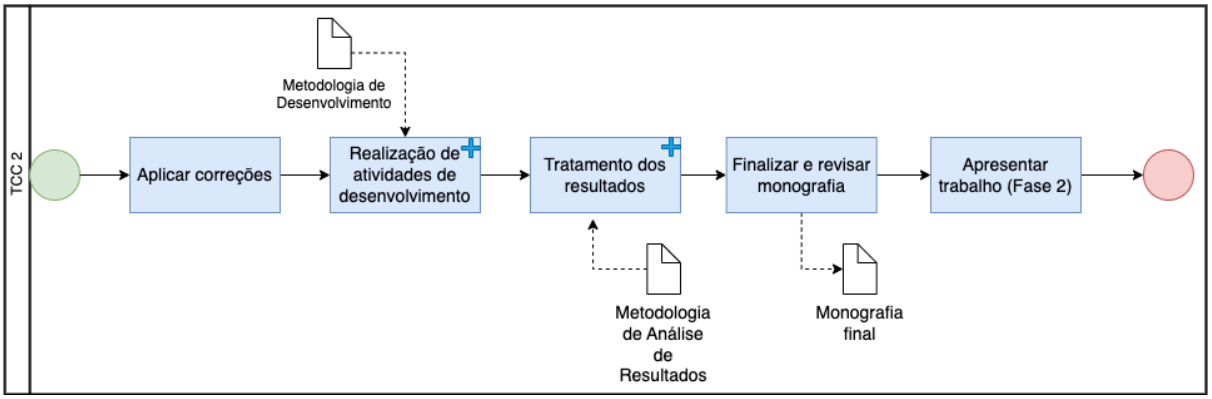


Fonte: Autores.

Conforme ilustrado na Figura 15, as atividades e os subprocessos relativos à segunda fase do TCC foram planejados da seguinte forma:

- Aplicar correções: Antes de dar continuidade ao trabalho, foram aplicadas as devidas correções na monografia, conforme apontado pela banca. Status: Realizado;
- Realização de atividades de desenvolvimento: Esse subprocesso foi organizado em três partes, sendo realizadas de forma paralela. Com base nos cenários de uso estabelecidos, foram aplicadas (i) a arquitetura Portas e Adaptadores, (ii) a Arquitetura Limpa e (iii) o padrão arquitetural MVC. As atividades de desenvolvimento foram realizadas com base na Metodologia de Desenvolvimento, conforme descrito na seção 4.4. Status: Realizado.
- Tratamento dos Resultados: Subprocesso que compreende as atividades estabelecidas nas etapas de pesquisa-ação, conforme consta detalhado na Metodologia de Análise de Resultados, seção 4.5. Status: Realizado;
- Finalizar e revisar monografia: Ajustes finais quanto à escrita e ao conteúdo entregues na segunda fase do trabalho. Status: Realizado, e
- Apresentar trabalho (Fase 2): Submissão e apresentação do trabalho final aos membros da banca. Status: Em Andamento.

Figura 15 – Fluxo de atividades TCC (segunda etapa)



Fonte: Autores.

4.3 Levantamento Bibliográfico

O levantamento bibliográfico foi desenvolvido a partir de conteúdos já disponíveis, incluindo livros e artigos científicos. Esses materiais foram usados como base para a construção do conhecimento compartilhado por essa monografia.

4.3.1 *Strings* de busca

Uma vez que o tema foi acordado, foram definidas *Strings* de busca para que fossem realizadas pesquisas de forma mais objetivas nas principais bases de dados. Ao decorrer da evolução do levantamento bibliográfico, têm-se refinamentos contínuos nas *Strings* de busca utilizadas. A Tabela 3 confere as principais *Strings* de busca, utilizadas na base Google Scholar, bem como a quantidade de artigos retornados. A base de dados Google Scholar foi escolhida, uma vez que o objetivo era o retorno de uma literatura mais abrangente, não apenas de artigos, mas também de livros de autores da área de interesse desse trabalho e materiais conhecidos e divulgados por empresas de renome.

Tabela 3 – *Strings* de Busca

String	**Base de Dados**	**Quantidade**
'Software architecture'	Google Scholar	4.100.000
'Hybrid Mobile Development'	Google Scholar	2.580.000
'Mvc Architecture'	Google Scholar	43.800
'Clean Architecture'	Google Scholar	1.070.000
'Ports and Adapters Architecture'	Google Scholar	15.300
'MVC' AND 'Hybrid Mobile Development'	Google Scholar	7.710
'Ports and Adapters' AND 'Hybrid Mobile Development'	Google Scholar	12.600
'Clean Architecture' AND 'Hybrid Mobile Development'	Google Scholar	110.000

Ocorreu certa dificuldade em encontrar exemplos práticos, capazes de mostrar as arquiteturas de forma mais aplicada, revelando aspectos mais concretos. Por isso, optou-se por especializar as *Strings* de busca, procurando avaliar cada arquitetura no domínio de interesse, sendo esse de desenvolvimento *mobile* híbrido. Percebeu-se, com essa estratégia, uma redução significativa no quantitativo de materiais retornados. De toda forma, ainda consta, conforme os números apresentados, uma grande quantidade de materiais retornados. Dessa forma, foram estabelecidos critérios de seleção, vistos a seguir, para filtragem dos materiais mais aderentes ao presente trabalho.

4.3.2 Critérios de Seleção

Uma vez com os materiais de referência em mãos, tornou-se necessário refinar para que fossem selecionados somente os principais artigos, livros e outros. Para tal, foi utilizada a leitura exploratória (GIL et al., 2002). Por meio disso, foram considerados os seguintes critérios de seleção:

- Definição de Arquitetura de *Software*;
- Tratar Arquitetura MVC;
- Tratar Arquitetura Portas e Adaptadores;

- Tratar Arquitetura Limpa;
- Tratar desenvolvimento *mobile*;
- Tratar Arquitetura Portas e Adaptadores no âmbito de desenvolvimento *mobile*, e
- Tratar Arquitetura Limpa no âmbito de desenvolvimento *mobile*.

Com base nisso, alguns dos principais artigos foram selecionados:

- *A. Hexagonal architecture* (GRAÇA, 2017b);
- *H. Mvc and its alternatives* (GRAÇA, 2017a);
- *C. Clean Architecture: A Craftsman's Guide to Software Structure and Design* (MARTIN, 2017);
- *What is hexagonal architecture* (BROWN, 2014);
- *A clean approach to flutter development through the flutter clean architecture package* (BOUKHARY; COLMENARES, 2019), e
- *Comparison of cross-platform mobile development tools* (PALMIERI; SINGH; CICCHETTI, 2012).

Recursos adicionais interessantes, devido ao uso da ferramenta Google Drive, foram manter o rastro das referências selecionadas como base para o trabalho, além de conhecimento sobre o número de citações de cada fonte.

4.4 Metodologia de Desenvolvimento

Toda e qualquer atividade de desenvolvimento do projeto orientou-se pelos princípios ágeis, sendo guiada por uma metodologia híbrida, ou seja, combinando práticas do Scrum (SCHWABER; SUTHERLAND, 2020) e do Kanban (TOTVS, 2021). De acordo com o Guia do Scrum, o mesmo é um *framework*, que pode ser utilizado por qualquer indivíduo, para realização de problemas complexos, a partir de iterações adaptativas (SCHWABER; SUTHERLAND, 2020).

Para o desenvolvimento das três aplicações propostas, foi utilizada a metodologia de desenvolvimento ágil híbrida, orientando-se por Scrum e Kanban.

O Scrum funciona a partir da definição de um *backlog*, sendo esse um nível de especificação de projeto envolvendo diferentes granularidades (ex. Tema, Épico, *Feature*, História de Usuário, Tarefa e Critério de Aceitação). O que é especificado no *backlog* deve

agregar valor ao produto a ser desenvolvido. A partir dessa especificação, são iniciadas diversas iterações do *framework*, conhecidas como *sprint*. As *sprints* são períodos curtos de desenvolvimento, com data de início e fim bem definidos, baseando-se nas definições compreendidas no *backlog* (SCHWABER; SUTHERLAND, 2020).

Como citado anteriormente, também foi utilizado o Kanban para o desenvolvimento desse trabalho. O Kanban é um método de gestão de atividades de forma visual, que tem o intuito de indicar as tarefas que estão sendo realizadas e limitar a quantidade das mesmas por integrante. Trata-se de um processo que permitiu melhorias de forma gradual (TOTVS, 2021). Diante do exposto, e orientando-se pelo Scrum e pelo Kanban, a Metodologia de Desenvolvimento desse projeto envolveu, basicamente:

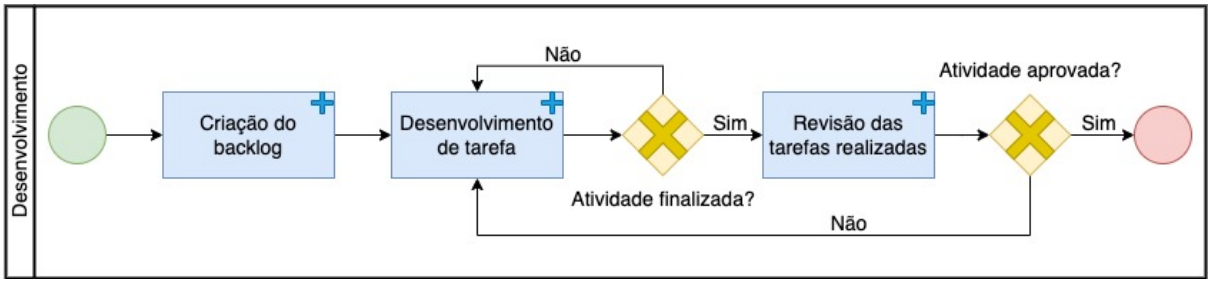
- Criação do *backlog*: Foram definidas diversas tarefas, antes do desenvolvimento, para que fossem cumpridas até o final do trabalho. Essas tarefas foram especificadas com status *ToDo*, em um primeiro momento;

- Desenvolvimento de tarefa: Tarefas selecionadas do *backlog* pelos integrantes foram desenvolvidas até o final de cada *sprint*. Ao entrar em desenvolvimento de fato, o *status* foi modificado para *Doing*. A cada tarefa finalizada, a mesma recebeu o status *Done*, permitindo dar continuidade ao desenvolvimento com outra tarefa, bem como progredir para a revisão da tarefa finalizada, e

- Revisão das tarefas realizadas: Enviar tarefas realizadas (i.e. status *Done*) para avaliação e aprovação dos envolvidos no projeto (autores e orientadora). Caso a tarefa não fosse aprovada, a mesma retornava ao desenvolvimento, tendo seu status alterado para *Doing*. Uma vez aprovada, o status passava a ser *Reviewed*;

A Figura 16 ilustra os detalhes compreendidos no subprocesso Realização das Atividades de Desenvolvimento, o qual resume a Metodologia de Desenvolvimento.

Figura 16 – Fluxo de desenvolvimento



Fonte: Autores.

4.5 Metodologia de Análise de Resultados

Com o intuito de analisar os resultados obtidos ao longo do trabalho, foi utilizada a metodologia de análise de resultados conhecida como pesquisa-ação. Esse método baseia-se em realizar uma ação, ou resolução, a partir da investigação de uma problemática. Os pesquisadores, no método pesquisa-ação, estão envolvidos de forma cooperativa, ou participativa (GERHARDT; SILVEIRA, 2009).

Pesquisa-ação envolve protocolos, cujas etapas podem ser dispostas conforme demanda de cada pesquisa. Para fins dessa pesquisa, o fluxo de pesquisa-ação foi orientado, conforme consta a seguir:

- Etapa Exploratória: O foco dessa etapa inicial foi obter informações significativas visando a elaboração da pesquisa (KRAFTA et al., 2009). Nesse sentido, foi realizada uma Pesquisa Bibliográfica, conforme consta descrito na Metodologia Investigativa, cujos resultados estão documentados, principalmente, nos capítulos Referencial Teórico 2 e Suporte Tecnológico 3. Adicionalmente, ainda como parte dessa etapa exploratória, foram acordadas novas informações à medida que as implementações dos cenários de uso se tornaram viáveis para consulta e apreciação por parte dos envolvidos e de terceiros. A escolha da amostra desses terceiros foi orientada à persona, especificamente elaborada para esse projeto, e detalhada na Figura 17.

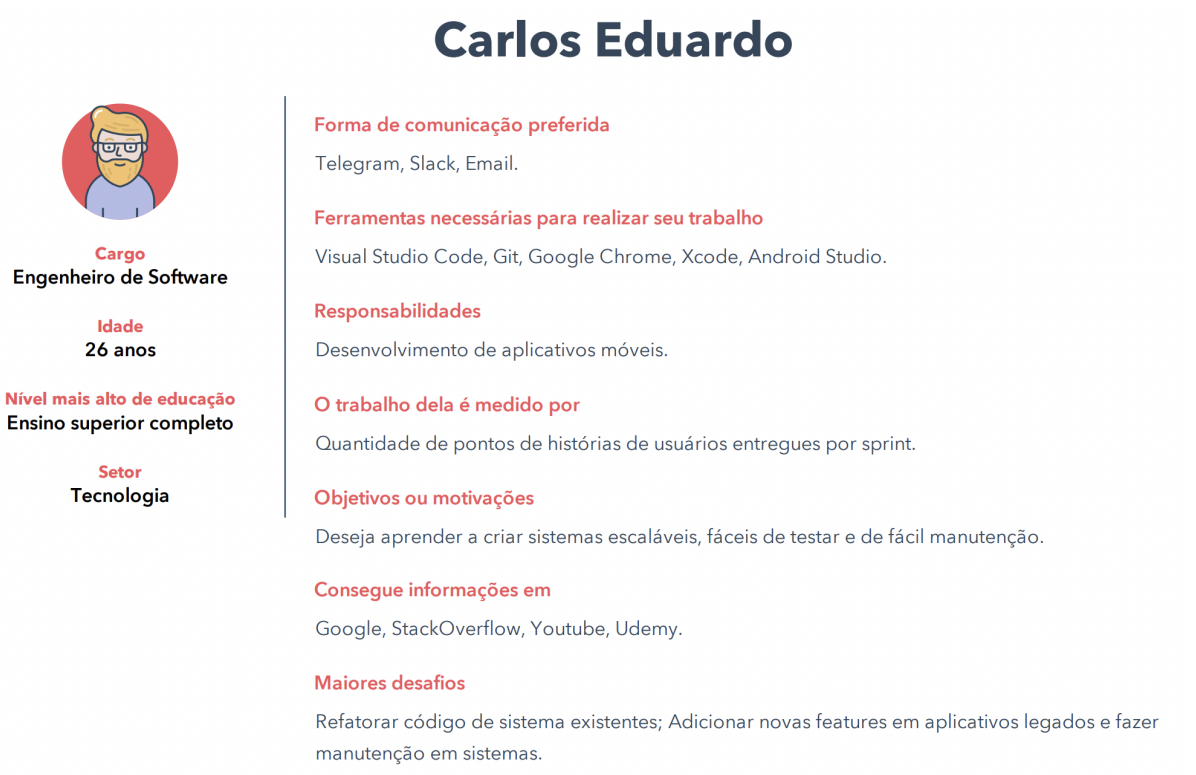
- Etapa Planejamento: O foco dessa etapa foi conferir continuidade à Etapa Exploratória, definindo de forma mais clara quais dados eram pertinentes para a pesquisa, sendo necessária a realização de uma filtragem, com base nos interesses de pesquisa (KRAFTA et al., 2009). O escopo dessa etapa ainda permaneceu em definição, em um primeiro momento. Sabia-se de antemão sobre o perfil do público alvo, conforme descrito na *persona*. Entretanto, quais métricas quantitativas seriam aferidas, quais critérios qualitativos seriam consultados e como tais insumos seriam obtidos ainda permaneceram por um tempo em processo de fechamento. De toda forma, no momento, cabe ressaltar que o SonarQube, por exemplo, permitiu acordar informações relevantes em termos quantitativos, tais como: número de *bugs* e erros, e ainda complexidade ciclomática. Adicionalmente, permitiu levantar vulnerabilidades e *code smells*. Esses insumos, junto à literatura, levam à indicação de tendências de cunho qualitativo, tais como: Alta/Média/Baixa Manutenibilidade. Por exemplo, a complexidade ciclomática pode ser entendida como uma métrica capaz de medir a complexidade de um dado código fonte. Quanto maior o seu valor, maior a dificuldade de se entender, modificar e, consequentemente, testar o código fonte (AJALA et al., 2016). Tais aspectos incorrem em comprometimentos em termos de Manutenibilidade. Testes de integração também foram utilizados afim de metrificar o quão simples foi implementar testes em cada arquitetura. Com base no exposto, apesar das múltiplas

alternativas de análise, que podiam ser planejadas, as tecnologias e os demais apoios estabelecidos para essa pesquisa, Suportes Tecnológicos 3, possibilitaram adequada análise quanti e qualitativa, conforme Análise de Resultados 6;

- Etapa Ação: O foco dessa etapa foi a criação de uma base de informações, as quais foram exploradas e planejadas anteriormente, em atendimento aos interesses da pesquisa, permitindo, então, coletar dados para futura avaliação (KRAFTA et al., 2009), e

- Etapa Avaliação: O foco dessa última etapa foi analisar os dados obtidos/coletados. Como ocorreram mais ciclos de pesquisa-ação, nessa etapa, coube avaliar se ocorreram melhorias e/ou fragilidades, quando comparados os dados da iteração em foco com os dados de iterações anteriores (KRAFTA et al., 2009). Quando possível, e em caso de fragilidades, novos ciclos envolvendo exploração, planejamento e ação foram realizados e/ou previstos. Ao final de cada ciclo, ocorreu a necessidade de avaliação dos resultados.

Figura 17 – *Persona*



Fonte: Autores.

4.6 Cronograma

Baseado nos fluxos propostos anteriormente, foram desenvolvidos os seguintes cronogramas para cumprimento, respectivamente, da primeira fase (Tabela 4) e da segunda fase (Tabela 5) do TCC.

Tabela 4 – Cronograma para o TCC (primeira etapa)

	Jun/2022	**Jul/2022**	**Ago/2022**	**Set/2022**
Definir o tema	X			
Levantamento bibliográfico	X			
Fechar escopo	X			
Estabelecer referencial teórico		X		
Estabelecer suporte tecnológico		X		
Especificar metodologia		X		
Definir proposta			X	
Descrever proposta			X	
Finalizar e revisar monografia				X
Apresentar trabalho (Fase 1)				X

Tabela 5 – Cronograma para o TCC (segunda etapa)

	Out/2022	**Nov/2022**	**Dez/2022**	**Jan/2023**	**Fev/2023**
Aplicar correções	X				
Realização de atividades de desenvolvimento	X	X	X		
Tratamento dos resultados			X	X	
Finalizar e revisar monografia				X	X
Apresentar trabalho (Fase 2)					X

4.7 Resumo do Capítulo

O capítulo teve o intuito de expor os detalhamentos metodológicos seguidos para a realização do trabalho.

Primeiramente, a pesquisa foi classificada, resultando em: abordagem híbrida (quanti e qualitativa), natureza aplicada, objetivos exploratórios e procedimentos Pesquisa Bibliográfica e Pesquisa-ação. Logo após, foi apresentado o fluxo de trabalho com as principais atividades e subprocessos, com destaque aos subprocessos que se orientaram por

metodologias específicas, sendo: Metodologia Investigativa, com Pesquisa Bibliográfica; Metodologia de Desenvolvimento, com a combinação de princípios do Scrum e do Kanban, e Metodologia de Análise de Resultados, com Pesquisa-ação. Há definição de *persona*, para melhor compreensão do perfil de interessado no tema da pesquisa, bem como apresentação dos cronogramas que conferem uma visão temporal quanto a realização das atividades e subprocessos.

5 Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos

Este capítulo tem como intuito apresentar, de forma mais detalhada, o foco principal desse trabalho, sendo: Estudo Orientado ao Desenvolvimento de Aplicativos Móveis Híbridos.

Desta forma, e visando apresentar sobre as aplicações que foram desenvolvidas em cada arquitetura (no caso, MVC, Portas e Adaptadores e Limpa), consta um descritivo sobre as funcionalidades comuns às aplicações, sendo focadas nas *features* de *chat* e mapa.

Adicionalmente, é apresentada uma prova de conceito, mais extensa, com o desenvolvimento de um aplicativo, chamado FindDev. A ideia é conferir uma visão mais clara, concreta e objetiva sobre vários desafios transpostos ao longo da realização desse trabalho. Em um primeiro momento, como uma das *features* relevantes abordadas por esse estudo envolveu o uso de geolocalização e a listagem de informações em um mapa, ocorreu a implementação, no FindDev, que abrangeu tais funcionalidades.

Para que os suportes tecnológicos, estabelecidos nesse trabalho, fossem instalados, configurados, integrados e conhecidos pelos autores, o aplicativo foi desenvolvido usando React Native como *framework* de desenvolvimento; Firebase, conferindo apoio para o uso de banco de dados NoSQL, bem como para o armazenamento dos dados do aplicativo via Firestore; GitHub, usando a API para obtenção de dados dos usuários, tais como: *emails*, repositórios, e principais tecnologias utilizadas pelos mesmos, e Expo, auxiliando no desenvolvimento *mobile* híbrido em React Native, particularmente, contribuindo na implementação da interface do aplicativo, dentre outros aspectos.

Na sequência, são conferidos comentários gerais, procurando destacar princípios da Arquitetura Portas e Adaptadores (ou Hexagonal), sendo esse o padrão arquitetural utilizado na implementação do aplicativo FindDev.

Ressalta-se ainda que, similarmente ao que consta especificado nesse capítulo, para o caso da Arquitetura de Portas e Adaptadores, com foco na *feature* mapa, ocorreram implementações de outros aplicativos móveis nas demais arquiteturas em estudo, para as *features* mapa e *chat*, além da implementação de aplicativo móvel, na Arquitetura de Portas e adaptadores, para a *feature chat*. Esse escopo de desenvolvimento mais abrangente e completo, bem como as respectivas análises de resultados usando pesquisa-ação, foram atuações realizadas na segunda etapa desse trabalho. Por fim, tem-se o resumo do capítulo.

5.1 Contexto

Conforme acordado anteriormente, ao longo dos últimos anos, tem ocorrido um crescimento considerável na quantidade de usuários de *smartphones*. Esse aumento no número de usuários deve-se, dentre outros fatores, às constantes evoluções das tecnologias bem como aos custos mais reduzidos dos *smartphones*. Além, é claro, da necessidade e presença de *software* em vários domínios, impactando no cotidiano das pessoas de forma ampla. Com a maior demanda pelo uso de dispositivos móveis, em termos globais, novos mercados começaram a surgir. Com eles, aumentou também a relevância e o interesse no desenvolvimento de aplicativos móveis.

Entretanto, desenvolver aplicativos móveis não é uma tarefa fácil, tampouco rápida e padronizada. No passado então, essa tarefa era ainda mais árdua, uma vez que o desenvolvimento dos primeiros aplicativos móveis demandava, exclusivamente, a implementação via código nativo, ou seja, para cada sistema operacional, era necessário desenvolver e manter um novo aplicativo específico. Tal realidade incorria, principalmente, em custos mais altos, inviabilizando alguns casos.

Mais atualmente, e visando facilitar o desenvolvimento de aplicativos móveis, surgiram os primeiros *Frameworks Cross-Platform*, permitindo a implementação de aplicações híbridas. A principal premissa desses *frameworks* é simplificar e diminuir custos, considerando o desenvolvimento multiplataforma, ou seja, os aplicativos são desenvolvidos de forma única, mas podem servir a diferentes Sistemas Operacionais.

Utilizando *frameworks* para criação de aplicativos multiplataformas, os times de desenvolvimento tendem a se concentrar na qualidade final do produto que está sendo entregue, uma vez que a equipe se preocupa com apenas o desenvolvimento de um único aplicativo, e não mais de um aplicativo para cada Sistema Operacional. Há ainda outras vantagens, tais como reutilização de *software* facilitada e manutenção simplificada, além da possibilidade de se aplicar padrões de projeto e arquiteturais que permitem viabilizar mais adequadamente questões como escalabilidade.

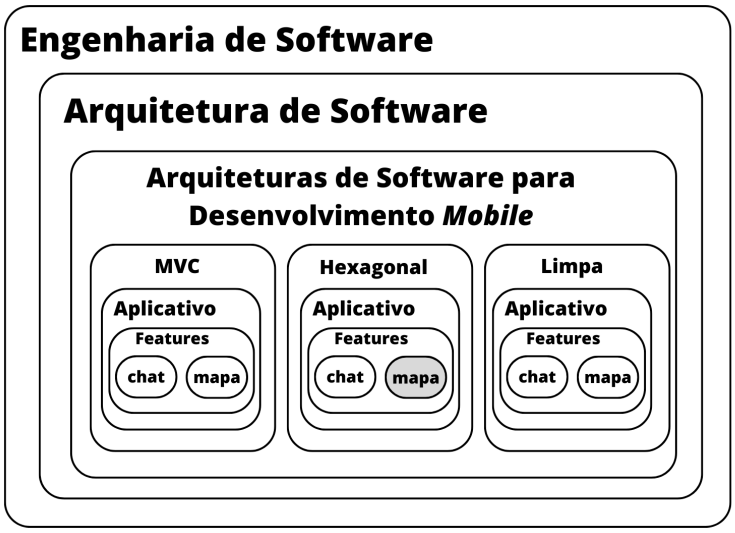
Contudo, mesmo com essas conquistas e esses facilitadores, o mercado *mobile* ainda é visto como algo recente, pouco menos de 20 anos. Sendo assim, há muito o que estudar, padronizar, evoluir, e pesquisar, em especial, se comparado à solidez que se tem em relação ao desenvolvimento *web*, por exemplo. Mesmo nesse último, ainda cabem constantes adequações e contribuições.

Diante do exposto, o foco desse trabalho compreendeu expor/revelar, dentre os padrões arquiteturais difundidos na comunidade de software (i.e. MVC, Portas e Adaptadores e Limpa), como aplicações móveis, construídas com *frameworks Cross-platform*, mais especificamente o React Native, se comportam. Procurou-se revelar esses comportamentos, para as arquiteturas mencionadas, usando métricas de análise de código estático, bem como

relatos dos desenvolvedores, sendo esses autores desse projeto. Assim, foram utilizados ferramentais específicos, dentre eles: SonarQube, serviços do Firebase (ex. Firestore), Jest e Codecov. Em termos de metodologia de análise de resultados, conforme consta na seção 4.5, foram utilizados ciclos de pesquisa-ação, com ênfase nas etapas: Exploratória, Planejamento, Ação e Avaliação.

Na Figura 18, consta um esquemático, que orienta sobre o domínio e o escopo de atuação desse estudo. Tem-se que a mesma está, na Engenharia de Software, centrada na área de Arquitetura de Software. Mais especificamente, há um olhar mais concentrado em Arquiteturas de Software para Desenvolvimento *Mobile*, sendo os padrões escolhidos: MVC, Portas e Adaptadores e Limpa. Além disso, dado o amplo escopo de desenvolvimento dos aplicativos móveis, optou-se por contribuir no desenvolvimento de aplicativos, cujas *features* de relevância são *chat* e mapa. Tais *features* foram identificadas, segundo a literatura especializada (GONZALEZ, 2019) e (LEDWON, 2018), como sendo gargalos no desenvolvimento de aplicativos para dispositivos móveis, uma vez que viabilizam demandas como: (i) comunicação via *chat*, com mensagens sendo trocadas em tempo real, notificações, sincronismo e *cache*, e (ii) alto esforço em termos de processamento, com inúmeras informações podendo ser exibidas e manipuladas nos mapas.

Figura 18 – Escopo de Atuação do Trabalho



Fonte: Autores.

5.2 Visão Geral das Aplicações

No desenvolvimento das aplicações, que foram objetos de estudo e análise nesse trabalho, tem-se foco nas mesmas funcionalidades, até para limitar o escopo. A intenção foi que o usuário final não conseguisse distinguir uma aplicação de outra, apesar de cada aplicação ser desenvolvida em uma arquitetura específica. Padronizando ainda mais, foi

utilizado o mesmo suporte tecnológico, no caso, React Native, Firebase e Expo. Por fim, as aplicações desenvolvidas contam com as *features* de *chat* e mapa. No total, portanto, foram seis cenários de uso, combinando Arquitetura e *Feature*: MVC/*Chat*, MVC/Mapa, Portas e Adaptadores/*Chat*, Portas e Adaptadores/Mapa, Limpa/*Chat* e Limpa/Mapa.

Conforme já colocado, a *feature* de *chat* foi escolhida por envolver desafios, tais como: necessidade de manter a segurança e a confiabilidade da aplicação, e relevância das aplicações de comunicações, permitindo as pessoas estarem sempre conectadas(GONZALEZ, 2019). Reforçando essa relevância, podem ser mencionadas as aplicações Telegram e Whatsapp, sendo utilizadas por centenas de milhares de indivíduos diariamente.

Dentre os pré-requisitos de uma aplicação de *chat* estão:

- Mensagens em tempo real: prioritárias no desenvolvimento de uma aplicação de comunicação. Assim que uma mensagem for enviada por um usuário qualquer, na posição de remetente, a mesma deve ser recebida por um segundo usuário, na posição de destinatário, de forma quase que instantânea. Portanto, o fluxo de mensagens deve fluir de forma rápida (GONZALEZ, 2019)), e

- Sincronismo de mensagem: prioritário nesse contexto. Ao disponibilizar um *chat* aos seus usuários, a aplicação deve garantir que nenhuma mensagem seja perdida durante o envio ou o recebimento. O usuário deve ter a garantia de que suas mensagens foram recebidas com sucesso, bem como na ordem em que foram enviadas (LEDWON, 2018).

Para fins de prova de conceito, foi escolhida a Arquitetura Portas e Adaptadores, ou Hexagonal, bem como o desenvolvimento de um aplicativo móvel, focado na *feature* de mapa. Questões como visão geral do aplicativo, arquitetura, principais tecnologias exploradas, bem como breves comentários são conferidos nas próximas seções.

5.3 Prova de Conceito Preliminar

No intuito de demonstrar a viabilidade técnica da proposta, foi provida uma prova de conceito que consistiu no desenvolvimento de um aplicativo, o FindDev. O aplicativo foi implementado usando o React Native e demais suportes tecnológicos acordados no Capítulo 3 - Suporte Tecnológico. Portanto, permitiu avaliar vários desafios, os quais acabaram por ocorrer ao longo do trabalho, sendo os principais: instalações de ferramentas, configurações de ambiente, integrações das tecnologias, além de dificuldades ao se implementar as *features*. Seguem seções, nas quais constam outros detalhes dessa prova de conceito.

5.3.1 Aplicativo FindDev

FindDev é um aplicativo que possibilita desenvolvedores acharem outros desenvolvedores próximos, bastando somente informar os nomes de usuário da plataforma GitHub.

O resultado do desenvolvimento do aplicativo pode ser conferido através do repositório, disponível na plataforma Github através do *link* https://github.com/TCC-Gabriel-Danillo/TCC-POC.

As funcionalidades do sistema desenvolvido relacionam-se, principalmente, com a necessidade de Geolocalização. A seguir, constam as principais funcionalidades do aplicativo, apresentadas na Tabela 6, que descrevem visões parciais do *backlog* do aplicativo.

Tabela 6 – *Backlog* (Parte I)

Feature	**Histórias de Usuário**
Mapa ->Geolocalização	Eu, como usuário, desejo entrar no aplicativo, com o meu usuário do GitHub.
	Eu, como usuário, desejo que a minha posição seja registrada no mapa.
	Eu, como usuário, desejo que outros desenvolvedores visualizem a minha posição no mapa.

Com base na implementação das Histórias de Usuário que constam na Tabela 6, uma vez que o usuário entra no aplicativo, sua posição é registrada para que outros usuários possam visualizá-la em uma mapa. A Figura 19 ilustra a tela de boas vindas, na qual se permite entrar com o usuário de GitHub. O fornecimento do nome de usuário na plataforma Github, juntamente com a sua posição, fornece todas as informações necessárias para que se obtenha os dados do usuário, tais como: *email* e tecnologias utilizadas.

Tabela 7 – *Backlog* (Parte II)

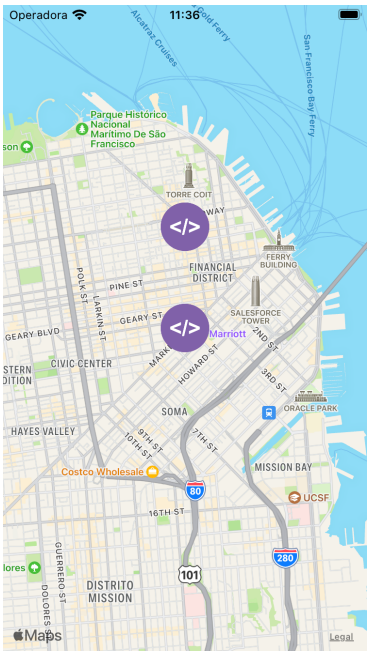
Feature	**Histórias de Usuário**
Mapa ->Geolocalização	Eu, como usuário, desejo ver informações sobre outros usuários.

Com base na implementação da História de Usuário que consta na Tabela 7, após informar o nome de usuário da plataforma Github, o usuário é redirecionado para um mapa, no qual poderá acessar informações de outros usuários. A Figura 20 ilustra a tela correspondente a essa funcionalidade.

Figura 19 – Página inicial do aplicativo FindDev

Fonte: Autores.

Figura 20 – Página do mapa do aplicativo FindDev - Visualização de Usuários



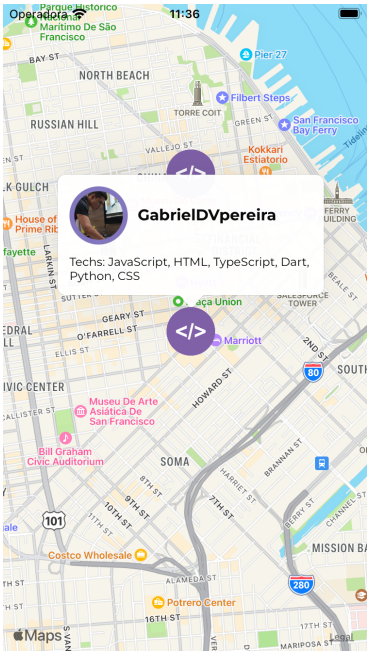
Fonte: Autores.

Com base na implementação da História de Usuário que consta na Tabela 8, na página do mapa, ao pressionar em um ícone no mapa, é possível visualizar detalhes sobre um desenvolvedor. Dentre os detalhes, destacam-se: foto, nome no Github, principais tecnologias e, se disponível, *email*. A Figura 21 ilustra a tela correspondente a essa funcionalidade.

Tabela 8 – *Backlog* (Parte III)

Feature	**Histórias de Usuário**
Mapa ->Geolocalização	Eu, como usuário, desejo visualizar detalhes sobre um desenvolvedor específico, via pop up.

Figura 21 – Página do mapa do aplicativo FindDev - Visualização de Detalhes do Desenvolvedor



Fonte: Autores.

5.3.1.1 Arquitetura

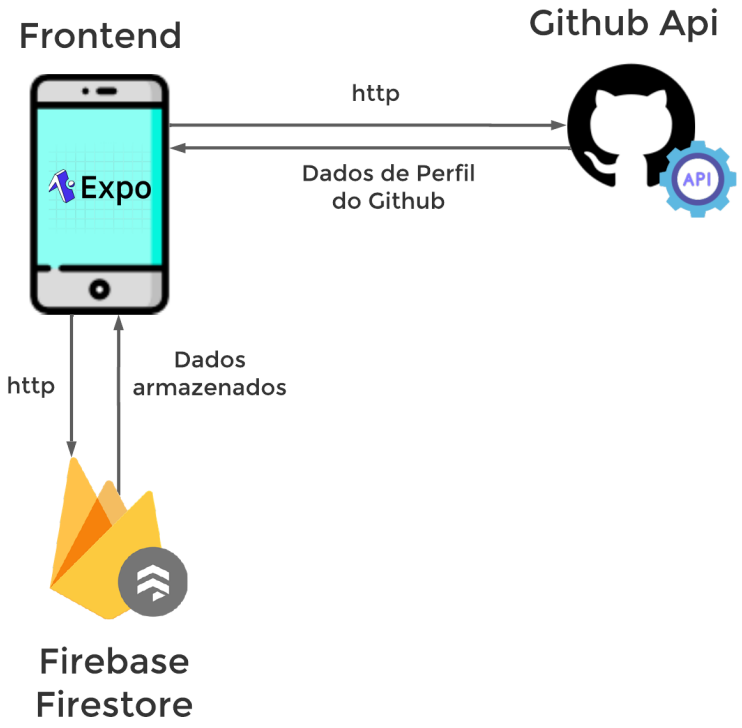
O sistema desenvolvido é composto por três componentes principais, como mostrado na Figura 22. São eles: *Frontend*, Github Api e Firebase Firestore.

O *Frontend* consiste no aplicativo desenvolvido em React Native, que se comunica com a API do GitHub para que se obtenha os dados do usuário, tais como: *email*, principais tecnologias nas quais ele desenvolve, dentre outras informações.

O Firebase Firestore é usado para armazenamento de dados. Nele, todos os dados obtidos pelo Github e a posição do usuário serão armazenados no banco de dados. Utilizou-se a ferramenta de banco de dados NoSQL do Firebase, com o Firestore para armazenamento.

A API do GitHub facilitou algumas necessidades de implementação. No caso, foi útil para se obter dados, a partir do nome do usuário na plataforma. Trata-se de uma API, cuja política de uso é de cunho aberto, pública. Com isso, foram obtidos vários dados disponíveis, dentre eles: repositórios dos desenvolvedores, *email*, tecnologias apreciadas, dentre outros. A documentação da API, consultada pelos autores desse trabalho ao longo da implementação do aplicativo FindDev, está disponível no *link*: https://docs.github.com/en/rest.

Figura 22 – Modelagem da arquitetura FindDev



Fonte: Autores.

5.3.1.2 Expo

O Expo, como já explicado em tópicos anteriores, consiste na ferramenta que auxilia no desenvolvimento *mobile* híbrido, utilizando React Native (FERNANDES, 2018). Orientando-se pelo Expo, foi desenvolvida a interface da aplicação. Adicionalmente, também usando recursos específicos desse suporte tecnológico, tal como o de geolocalização, foram obtidas a latitude e a longitude correspondentes à posição atual do usuário. Tal recurso viabilizou a implementação de geolocalização, inerente à *feature* mapa.

Ressalta-se que o Expo conta com adequada documentação, disponível *online*, de acesso aberto, e que contém várias orientações para lidar com aspectos muito relevantes no desenvolvimento *mobile*, dentre elas: bateria, brilho, acelerômetro, câmera e outros. No caso da geolocalização, a documentação utilizada pelos autores, para viabilizar a implementação, encontra-se disponível no *link*: https://docs.expo.dev/versions/latest/sdk/location/.

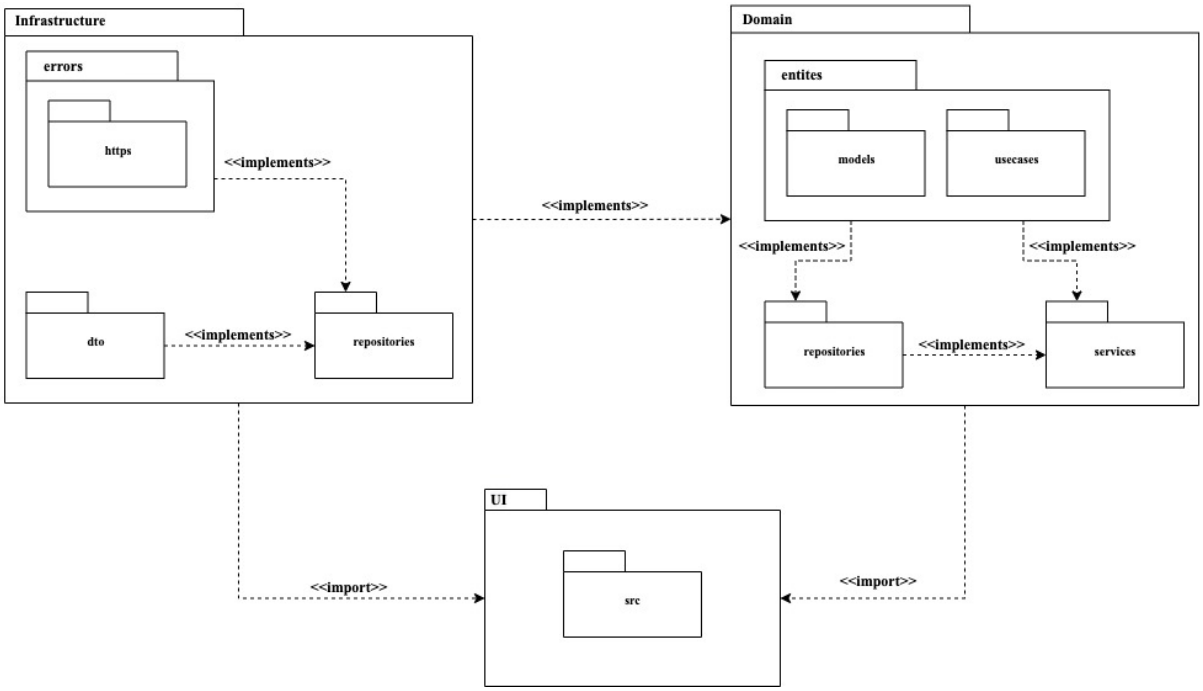
5.3.2 Estrutura de Pastas do Aplicativo

O código do aplicativo está organizado em pastas, orientando-se por uma estrutura em pacotes. O diretório principal, que corresponde ao código como um todo, é o *src*. Constam ainda três subdiretórios, cada qual representando uma camada da arquitetura Portas e Adaptadores. A seguir, é conferida uma breve explicação sobre cada subdiretório:

- *Domain*: Representa a camada de domínio da aplicação. Essa camada é responsável pela lógica de negócio do *app* FindDev. Nela, também consta definido o formato dos dados que serão utilizados, ou seja, as portas. Cada um dos arquivos de interface representa uma porta. Ainda nesse diretório, encontra-se a pasta *entities*, que apresenta as interfaces para modelos, bem como os casos de usos, que representam portas primárias da aplicação. A pasta *repositores* são declarações de portas secundárias, ou seja, comunicação com o meio externo. Por fim, na pasta *service*, encontra-se a implementação das portas primárias;
- *Infrastructure*: Representa a camada responsável por realizar toda a lógica agnóstica da aplicação. Essa camada é incumbida por estabelecer a conexão com o Firebase e qualquer outro serviço externo que possa vir a existir, ou seja, ela implementa as portas secundárias presentes no camada de *Domain*, e
- UI: Representa a camada de interface, que interage com o usuário. Ela faz uso dos adaptadores presentes em *Domain* e *Infrastructure*, para relacionar toda a lógica do negócio com a parte visual do sistema.

Para representar os diretórios de forma visual, foi especificado o Diagrama de pacotes, ilustrado na Figura 23. Lembrando que, com base em (BROWN, 2014), as camadas internas da arquitetura Portas e Adaptadores não possuem conhecimento da existência das camadas externas. Considera-se, na modelagem, a camada mais externa, a *Infrastructure*. Já a mais interna, a *Domain*.

Figura 23 – Diagrama de pacotes



Fonte: Autores.

5.3.2.1 Comentários Gerais

O resultado final do aplicativo FindDev demonstra os principais aspectos acordados na literatura sobre a Arquitetura Portas e Adaptadores (i.e. Hexagonal), sendo o desacoplamento de código e a clara divisão de responsabilidades entre as camadas arquiteturais (BAILÉN, 2019). Cada camada é desacoplada das demais, comunicando-se por relações de dependência e interfaces padronizadas. Além disso, cada camada tem responsabilidades bem definidas, o que tende a facilitar questões de manutenção e evolução do aplicativo FindDev.

O Firebase, por exemplo, responsável pelo armazenamento de dados, poderia ser facilmente substituído por outro sistema de gerenciamento, devido ao uso de uma porta que descreve como a interface de banco de dados do sistema deve se comportar no aplicativo.

Conforme exposto, percebe-se a facilidade de alteração de dependências no aplicativo, sem o comprometimento dos principais componentes arquiteturais. Novamente, tal facilidade traz benefícios para a manutenção evolutiva do FindDev.

Análises complementares e mais aprofundadas, usando métricas e ferramentas como SonarQube, Analytics e Jest, foram tratadas no escopo do TCC (segunda etapa), incluindo para o caso do FindDev. De toda forma, com a prova de conceito provida, percebe-se que vários desafios e demandas do projeto foram transpostos, dentre eles: (i) instalações de várias tecnologias; (ii) configurações do ambiente de desenvolvimento; (iii) integração das tecnologias envolvidas, e (iv) leitura, conhecimento e familiarização dos conceitos, arquitetura Portas e Adaptadores e *feature* mapa, dentre outros.

5.4 Aplicações Desenvolvidas

Seguindo adiante e dando ênfase em detalhar o escopo prático do projeto, tem-se foco nas apresentações das aplicações desenvolvidas, bem como na descrição de cada arquitetura inerente ao presente estudo. É importante ressaltar que, para cada aplicação desenvolvida, foram utilizadas métricas coletadas pela ferramenta SonarQube (2022), sendo apresentadas com intuito de demonstrar/evidenciar que foram seguidas boas práticas de desenvolvimento.

A realização de testes de integração foram essenciais para expor fragilidades e pontos fortes de cada arquitetura (RICHARDSON; WOLF, 1996). Por isso, para cada aplicativo desenvolvido, estabeleceu-se critério de cobertura de testes, que sempre deve estar acima de 80 %.

Nesse aspecto, como já mencionado anteriormente, foram conferidos, para cada arquitetura, dois tipos de aplicativos, sendo: (i) um envolvendo mapa/geolocalização como funcionalidade principal, e (ii) outro envolvendo conversa em tempo real. Dessa forma,

foram planejados e implementados: o FindDev, semelhante à prova de conceito apresentada, mas com incremento de novas funcionalidades como autenticação com o Github e filtro de usuários por distância, e o DevChat, um aplicativo de conversa entre desenvolvedores, envolvendo uma tecnologia de desenvolvimento como tópico central de cada conversa.

O desenvolvimento de cada aplicativo foi guiado pela metodologia híbrida apresentada pela sessão 4.4, sendo assim, houve a combinação do uso do Scrum com o Kanban.

5.4.1 Especificações gerais

Para as duas modalidades de aplicativo, a arquitetura geral que descreve os componentes principais e a comunicação entre os mesmos é representada pela Figura 22. Com isso, continua-se utilizado o Firebase e seu módulo Firetore como banco de dados não relacional; o Expo como ferramenta de desenvolvimento para aplicativos móveis, aproveitando-se de módulos como geolocalização e integração com autenticação com provedores (para este estudo, o Github Auth), e o GithubApi para obtenção dos dados sobre tecnologias utilizadas pelos desenvolvedores. A seguir, para cada aplicativo, pode-se observar seu *backlog*, bem como telas que compõem cada aplicativo.

5.4.1.1 FindDev

A aplicação FindDev, que fora apresentada como prova de conceito na primeira etapa do TCC, na seção 5.3.1, teve seu desenvolvimento continuado para estudos mais aprofundados do comportamento das arquiteturas implementadas.

O aplicativo em si, recapitulando de forma breve sobre explicações conferidas na seção 5.3.1, consiste no conceito simples de renderizar um mapa com a localização de diversos desenvolvedores. A partir da localização dos desenvolvedores, é possível identificar as tecnologias mais utilizadas pelos mesmo, e também navegar diretamente para a página do Github do usuário. O aplicativo foi separado em três épicos distintos, sendo eles: autenticação, geolocalização e informações. Na Tabela 9, a seguir, há as principais histórias de usuário modeladas para o épico de autenticação.

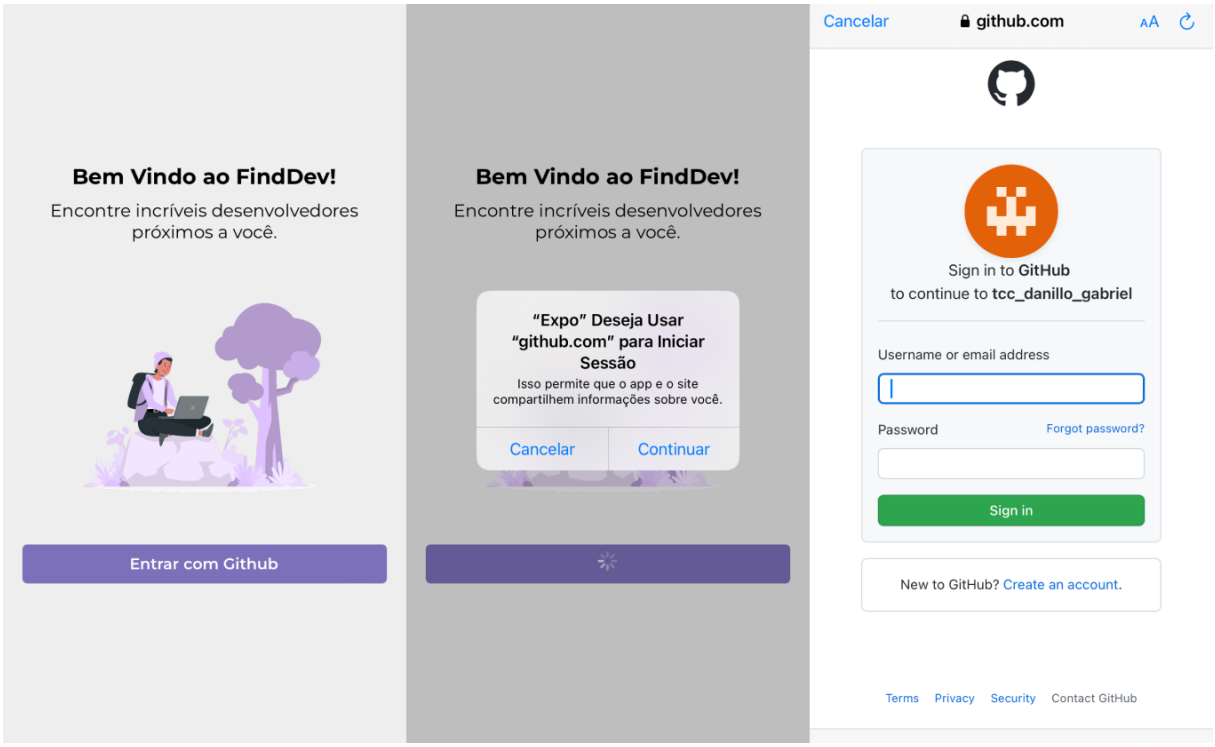
A autenticação de usuários a partir do Github, Figura 24, representou a alteração mais impactante para a aplicação. Sendo assim, é possível limitar a forma como os usuários interagem com o aplicativo. Toda a parte de segurança e autenticação ficou como responsabilidade do Github. Além disso, a partir da autenticação de usuários, foi possível realizar a persistência de dados do mesmo. Dessa forma, o usuário não necessita solicitar, novamente, uma nova autenticação para o Github.

O épico de Geolocalização manteve-se quase que inteiramente idêntico à versão da prova de conceito. Na versão mais recente do FindDev, Tabela 10, foi adicionado um filtro para renderização de usuários em um raio 10km. Na versão mais antiga da aplicação, todos

Tabela 9 – *Backlog FindDev* (Parte I)

Feature	**Histórias de Usuário**
Autenticação ->Github	Eu, como usuário, desejo ser capaz de logar no aplicativo utilizando a conta do Github, para que eu possa ver minhas informações do git, como tecnologias, repositórios, nome de usuário e etc, e, também, de outros usuários.
	Eu, como usuário, desejo ser capaz de realizar cadastro na aplicação a partir do Github, para que eu possa ver minhas informações do git, como tecnologias, repositórios, nome de usuário e etc, e, também, de outros usuários.
	Eu, como usuário, desejo ser capaz de realizar logout na aplicação.

Figura 24 – Fluxo de *login* com autenticação.



Fonte: Autores.

os usuários eram requisitados de uma vez só do banco. Essa mudança foi realizada para que, caso haja uma grande quantidade de usuários a serem renderizados no mapa, não ocorra perda alguma de desempenho para o usuário ao utilizar a aplicação. Entretanto, a estilização no *app* manteve-se idêntica à Figura 20.

Por fim, o épico de Informações, Tabela 11, não incluiu nova *feature* na aplicação. Todos os comportamentos listados mantiveram-se idênticos à versão inicial apresentada na prova de conceito. As *features* podem ser vistas na Figura 21.

Tabela 10 – *Backlog FindDev* (Parte II)

Feature	**Histórias de Usuário**
Geolocalização	Eu, como usuário, desejo que minha localização atual atualizada a todo momento que eu logar na aplicação, para que outros usuários sejam capazes de ver minha última localização atual.
	Eu, como usuário, desejo que minha localização final seja salva assim que realizar *logoff* da aplicação, para que outros usuários sejam capazes de ver minha localização mesmo quando estiver sem rede.
	Eu, como usuário, desejo visualizar desenvolvedores em um raio de 10km, para que possa ter uma melhor experiência de usuário e o *app* seja mais fluido.

Tabela 11 – *Backlog FindDev* (Parte III)

Feature	**Histórias de Usuário**
Informações	Eu, como usuário, desejo visualizar detalhes sobre um desenvolvedor específico, via *pop-up*.
	Eu, como usuário, desejo ser redirecionado para a página do Github do desenvolvedor ao pressionar o *pop-up*, para que possa ver mais detalhes sobre um usuário.

5.4.1.2 DevChat

O aplicativo DevChat é um aplicativo focado em trocas de experiências entre desenvolvedores. Nele, usuários são capazes de iniciar conversas sobre temas de seu interesse com outros usuários, de acordo com as tecnologias mais utilizadas na plataforma Github. A seguir, constam as principais funcionalidades do aplicativo, apresentadas na Tabela 12, que descrevem visões parciais do *backlog* do aplicativo:

Tabela 12 – *Backlog DevChat* (Parte I)

Feature	**Histórias de Usuário**
Autenticação ->Github	Eu, como usuário, desejo ser capaz de logar no aplicativo utilizando a conta do Github, para que eu possa ver minhas informações do git, como tecnologias, repositórios, nome de usuário e etc, e, também, de outros usuários.
	Eu, como usuário, desejo ser capaz de realizar cadastro na aplicação a partir do Github, para que eu possa ver minhas informações do git, como tecnologias, repositórios, nome de usuário e etc, e, também, de outros usuários.
	Eu, como usuário, desejo ser capaz de realizar logout na aplicação.

Com base na implementação das histórias de usuário presentes na Tabela 12, é possível obter as principais informações sobre os desenvolvedores, tais como: tecnologia, foto de perfil, nome de usuário e assim por diante. Com isso, é possível categorizar os usuários de acordo com sua área de interesse, orientando-se pelas tecnologias utilizadas na plataforma Github. A Figura 25 ilustra a tela de boas vindas, que permite ao usuário se autenticar utilizando seu Github.

Figura 25 – Página de boas vindas do aplicativo DevChat.

Fonte: Autores.

Tabela 13 – *Backlog DevChat* (Parte II)

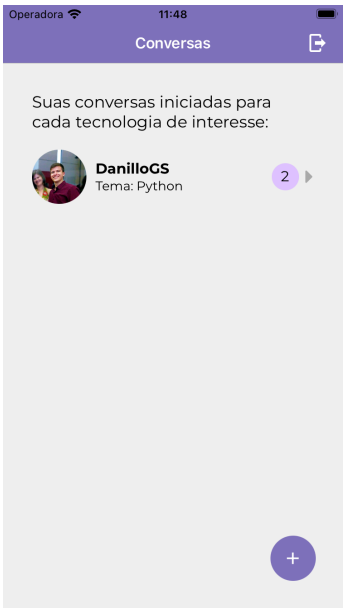
Feature	**Histórias de Usuário**
Conversa ->Página inicial	Eu, como usuário, desejo acessar a página inicial após o login e ver minhas conversas iniciadas.
	Eu, como usuário, desejo ver um botão de iniciar uma nova conversa e, ao pressioná-lo, desejo ver uma lista de pessoas por tecnologias, para que possa iniciar uma nova conversa.

Como ilustrado na Tabela 13, a implementação das histórias de usuário relacionadas à página principal corresponde ao módulo de conversa. Para isso, foi utilizada a ferramenta Firebase, que possui suporte a banco de dados, auxiliando em tais funcionalidades. Com isso, uma lista de conversas iniciadas é mostrada na tela inicial, bem como um botão flutuante em tela que possibilita o fluxo para iniciar uma nova conversa, como ilustrado nas Figuras 26 e 27.

Figura 26 – Página inicial, quando não há conversas.

Fonte: Autores.

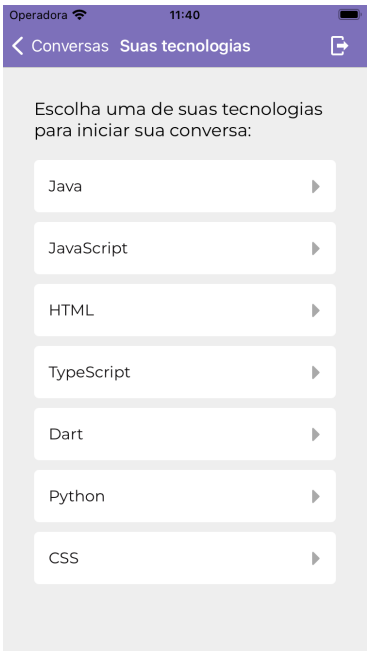
Figura 27 – Página inicial, quando há conversas.



Fonte: Autores.

Quando pressionado o botão de adicionar uma nova conversa, é apresentada uma lista de tecnologias de interesse do usuário autenticado, obtida via API do Github, como mostrado na Figura 28.

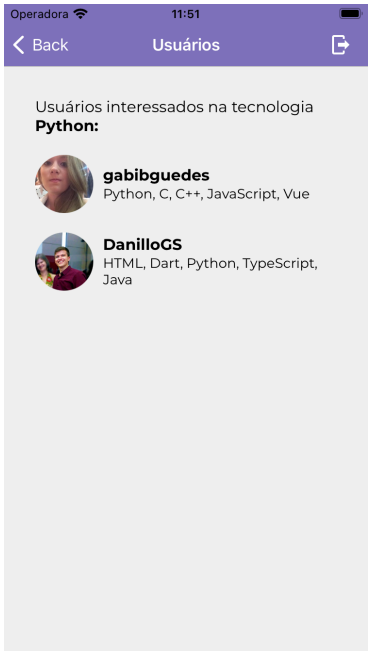
Figura 28 – Lista de tecnologias parte 1



Fonte: Autores.

Ao selecionar uma das tecnologias, são apresentados os usuários que também possuem interesse em tal tecnologia, como mostrado na Figura 29.

Figura 29 – Lista de tecnologias parte 2



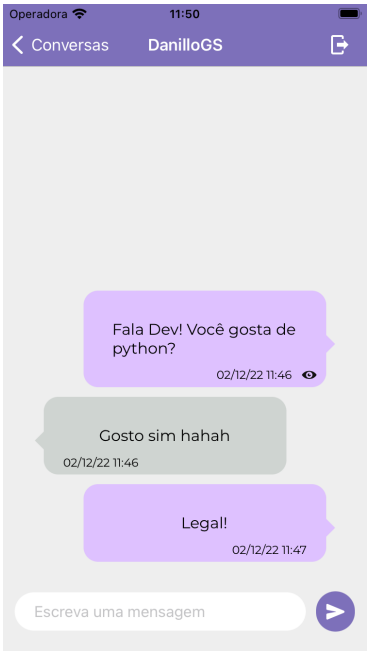
Fonte: Autores.

Tabela 14 – *Backlog DevChat* (Parte III)

Feature	**Histórias de Usuário**
Conversa ->Mensagens	Eu, como usuário, desejo ser capaz de iniciar uma nova conversa com um usuário.
	Eu, como usuário, desejo ser capaz de enviar e receber mensagens em tempo real.
	Eu, como usuário, quero que minhas mensagens sejam marcadas como lidas no momento que o usuário a qual estou conversando visualizá-la.

Por fim, ao iniciar uma nova conversa, e considerando as histórias de usuário especificadas na Tabela 14, uma página de mensagens é apresentada, como ilustrada na Figura 30, onde usuários podem interagir em tempo real.

Figura 30 – Página de mensagens.



Fonte: Autores.

A arquitetura geral da aplicação é extremamente similar ao usado pela prova de conceito, representada na Figura 22. Há utilização da ferramenta Firebase para banco de dados, e GithubApi para promover a autenticação e a obtenção dos dados dos usuários.

5.4.2 Arquiteturas Específicas

A arquitetura específica de cada aplicativo é apresentada em um diagrama de pacotes, o qual procura ilustrar de forma mais adequada, e para cada arquitetura, sua organização em um nível mais macro de abstração. Vale ressaltar que, para a mesma arquitetura, cada aplicativo apresenta de uma forma geral o mesmo esquema organizacional. Sendo assim, possuindo diagramas semelhantes. Por isso, a seguir, é apresentado um esquema de pacotes para cada arquitetura, e não para cada aplicativo.

5.4.2.1 Portas e Adaptadores

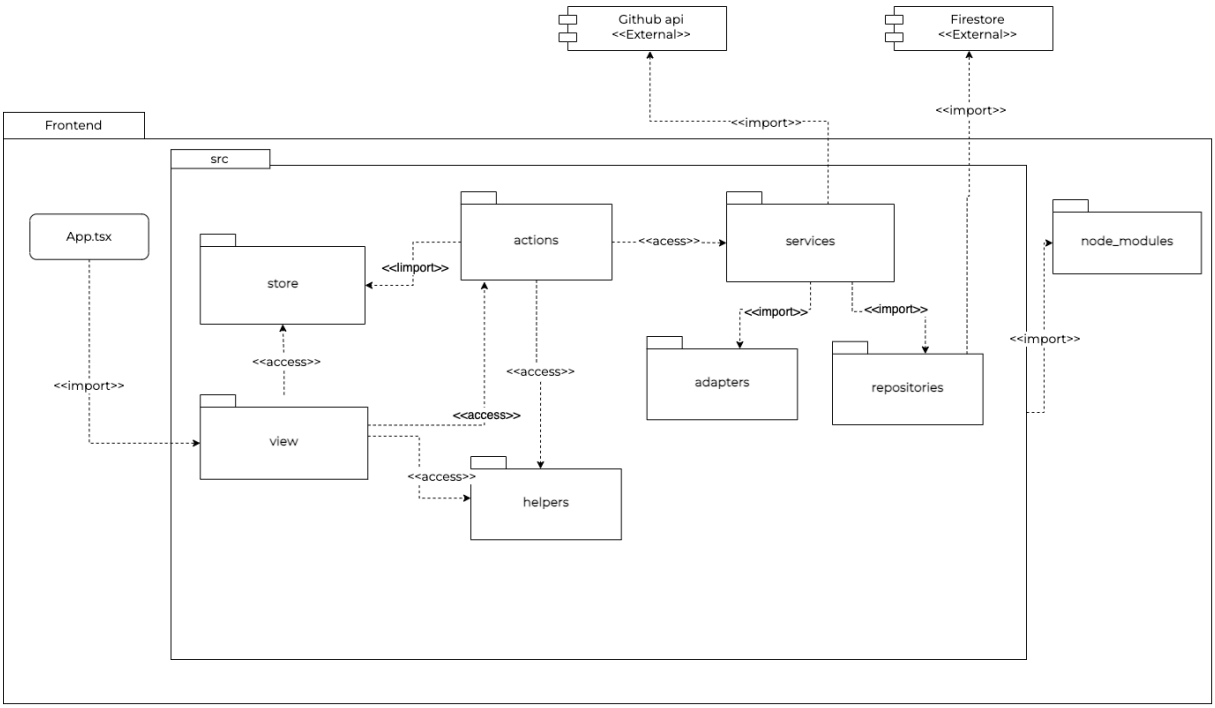
A implementação geral dos pacotes da aplicação, na arquitetura Portas e Adaptadores, manteve-se a mesma lógica seguida na prova de conceito, executada anteriormente, como pode ser visto no diagrama de pacotes da Figura 23.

5.4.2.2 MVC

A seguir, na Figura 31, está representado o diagrama de pacotes das aplicações desenvolvidas, seguindo os princípios da arquitetura MVC utilizando o Redux. Como dito

anteriormente na seção 3.6, o Redux permite utilizar um padrão de desenvolvimento que possui um comportamento análogo ao MVC (BAUMGARTNER, 2021).

Figura 31 – Diagrama de pacotes para arquitetura MVC.



Fonte: Autores.

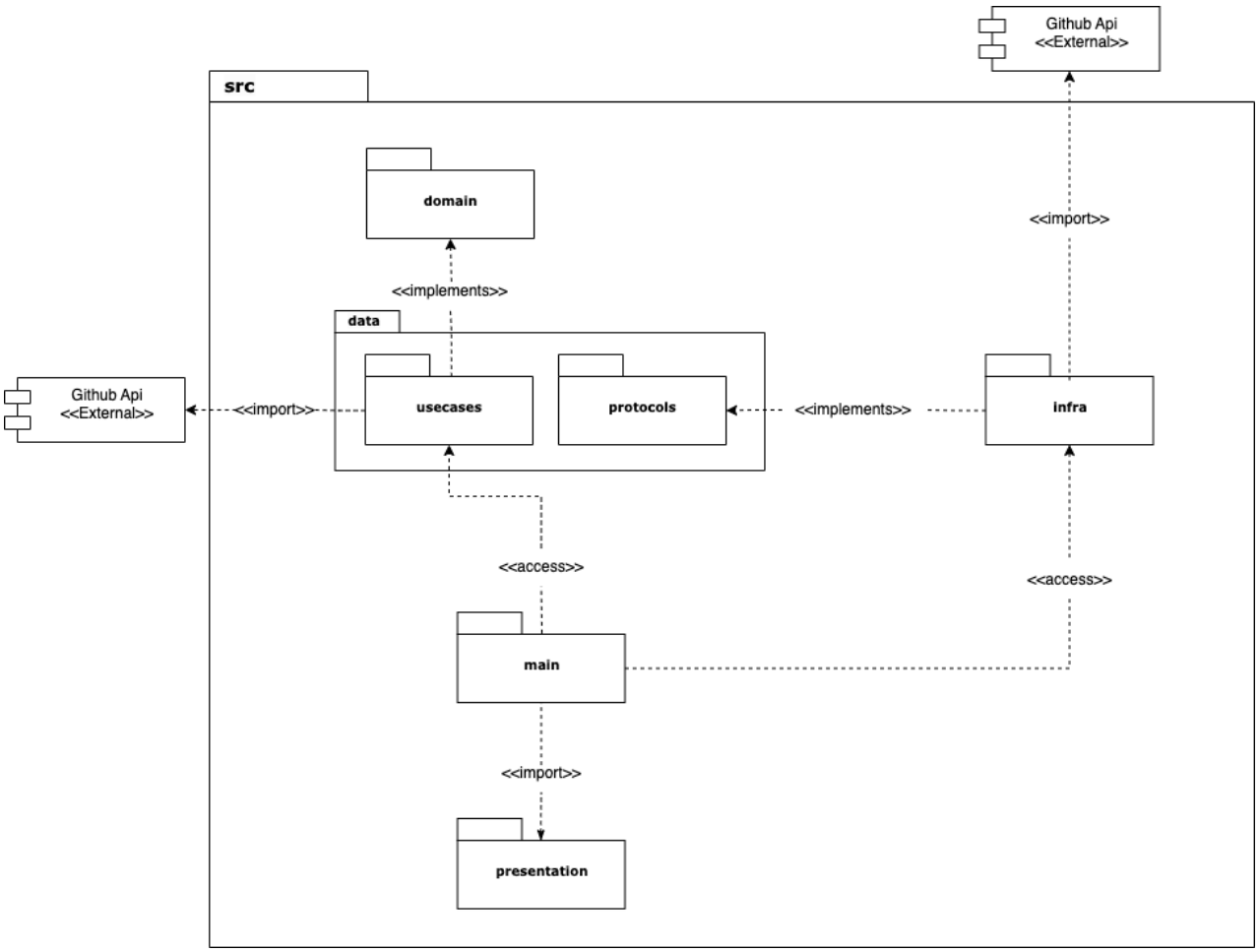
Os principais diretórios desse modelo podem ser descritos da seguinte forma:

- *Store*: Onde é mantido o estado da aplicação;
- *Action*: De onde partem as solicitações de mudanças de estados e onde é mantida a lógica de negócio;
- *View*: Interface para o usuário. Nela, partem as chamadas para a *Action* que alteram a *Store*. Uma vez quem uma *Store* é atualizada, a *View* atualiza para mostrar seus valores atualizados para o usuário;
- *Helpers*: Algumas funções para auxiliar o desenvolvimento, como isolamento de lógica ou funções que são comumente utilizadas dentro da aplicação;
- *Services*: São, geralmente, classes que são utilizadas para comunicação externa (como comunicação com banco de dados ou APIs externas);
- *Repositories*: Métodos que comunicam diretamente com banco de dados. São invocados pelos *Services* e
- *Adapters*: Métodos que adaptam o uso de bibliotecas externas para interfaces definidas pela aplicação. Assim, desacoplam o sistema de pacotes externos. Também são invocados pelos *Services*.

5.4.2.3 Arquitetura Limpa

A Figura 32 apresenta o esquema de pacotes utilizado para representar a Arquitetura Limpa. Nele, assim como na arquitetura de Portas e Adaptadores, é possível observar uma certa granularidade de suas camadas, que são subdivididas de acordo com o papel que irão realizar dentro do sistema.

Figura 32 – Diagrama de pacotes para arquitetura Limpa.



Fonte: Autores.

- *Domain*: Onde são implementadas as regras de negócio da aplicação. Não exite qualquer implementação de código nessa camada, apenas definição de interfaces e abstrações que irão garantir a robustez da arquitetura;
- *Data*: Essa camada implementa as interfaces declaradas na camada de *Domain*. Então, se a camada de domínio declara um uma interface que representa um caso de uso, a camada de *Data* possui a classe que implementa essa interface. Além disso, por vezes, para implementar os casos de uso, as classes em *Data* precisam consultar serviços, bancos de dados e assim por diante. Para isso, ela define abstrações de dependências que precisa, e tais dependências são implementadas pela camada de *Infrastructure* ou apenas *Infra*;

- *Infra*: A camada de *Infra* implementa as dependências que a camada de *Data* necessita. Por vezes, essa camada é a ponte de qualquer comunicação com dependências externas e a aplicação.

- *Presentation*: A camada de *Presentation* representa a parte visual do sistema. Ela também utiliza os casos de uso que foram declarados pela *Domain* e implementados por *Data*. Todavia ela possui somente referência das interfaces produzidas por *Domain*. A utilização dessas funções dá-se graças à injeção de dependência criada pela camada de *Main*, e

- *Main*: Essa camada é responsável em unir os componentes de outras camadas. Ela possibilita a comunicação entre a camada de *Data* e as implementações da camada de *Infra* por meio da injeção de dependência. Ela fornece a implementação de *Data* para a camada de *Presentation* por meio, também, da injeção de dependência. Logo, nota-se que é uma camada que possui uma relação com todas as camadas dessa arquitetura.

5.5 Resumo do Capítulo

Neste capítulo, foram conferidos detalhes sobre os estudos orientados ao desenvolvimento de aplicativos móveis híbridos. Para tanto, foram retomados aspectos de contextualização e apresentação do escopo geral do projeto. Adicionalmente, foi especificada a prova de conceito. Na prova de conceito, procurou-se explorar demandas e desafios, vistos como importantes para a plena concretização do estudo. Na sequência, constam as aplicações desenvolvidas, com foco em especificações gerais bem como nas modelagens de cunho organizacional das arquiteturas específicas: Portas e Adaptadores, MVC e Limpa.

6 Análise de Resultados

Este capítulo apresenta a análise dos resultados obtidos, baseando-se nas experiências de desenvolvimento, sob o ponto de vista dos autores, bem como nos testes de integração realizados nas aplicações implementadas. Cada uma das aplicações desenvolvidas, FindDev e DevChat, tiveram seus requisitos detalhadamente documentados no Capítulo 5. Doravante, o intuito é documentar cada etapa que foi seguida, considerando o protocolo de pesquisa-ação estabelecido no Capítulo 4. Tal protocolo compreende as fases: Exploratória, Planejamento, Ação e Avaliação. Por fim, tem-se o Resumo do Capítulo.

6.1 Pesquisa-Ação

Conforme descrito no Capítulo 4, acerca da metodologia, o presente trabalho está de acordo com as etapas propostas no protocolo de pesquisa-ação. A princípio, ocorreu a etapa exploratória, na qual levantou-se o referencial teórico. Tal referencial conferiu embasamento para os estudos realizados, além da definição do público alvo através da criação de persona. Na sequência, ocorreu a etapa de planejamento, na qual elicitou-se os requisitos das aplicações FindDev e DevChat; especificou-se a diagramação de pacotes de cada arquitetura, bem como detalhes acerca da realização dos testes. A próxima etapa, focada na ação, consistiu na construção das aplicações em si, orientando-se pelo planejado na fase anterior. Por fim, há a etapa de avaliação, que consistiu em documentar as impressões que os autores tiveram durante o desenvolvimento das aplicações na fase anterior.

6.1.1 Etapa Exploratória

Nesse momento, foi acordado o escopo do trabalho como um todo. A definição das referências, Teóricas e Tecnológicas, foi baseada em pesquisas na literatura, bem como em sites especializados na área de Arquitetura de Software e aplicações *mobile*. Adicionalmente, apoiou-se na experiência dos autores com tecnologias bem aderentes à proposta do trabalho. Cabe ressaltar que ambos os autores dessa monografia possuem experiência profissional na área, atuando em empresas como a IBM e em uma *startup* internacional. Além disso, nessa etapa, identificou-se o público alvo, ou seja, possíveis interessados nos resultados pretendidos com esse trabalho. O perfil desse interessado foi especificado usando o conceito de Persona, sendo essa apresentada na seção 4.5 (Figura 17).

Basicamente, o perfil compreende alguém que: atua no desenvolvimento de aplica-

tivos móveis; faz uso de tecnologias emergentes (ex. Visual Studio Code, Xcode, Android Studio); tem a iniciativa de buscar informações usando recursos *online*; deseja aprender mais sobre sistemas escaláveis, testáveis e manuteníveis, e sabe o quão desafiador é refatorar código de sistemas existentes, adicionando, removendo ou refinando *features* em sistemas legados.

6.1.2 Etapa Planejamento

O planejamento ocorreu via o detalhamento e a priorização dos requisitos, sendo elicitados com base nas *features* Mapa e *Chat*; a modelagem desses requisitos em *Backlogs*, e as especificações arquiteturais usando Diagramações UML, mais especificamente, Diagramas de Pacotes. Ressalta-se ainda esses detalhamentos encontram-se acordados no Capítulo 5. Adicionalmente, definiu-se que seriam utilizadas, além de métricas de qualidade, também testes de integração, com cobertura de testes acima de 80%.

6.1.3 Etapa Ação

Como a etapa ação compreende a implementação de fato de cada aplicação, FindDev e DevChat, optou-se por apresentar os resultados dessa etapa em seções dedicadas, conforme consta a seguir.

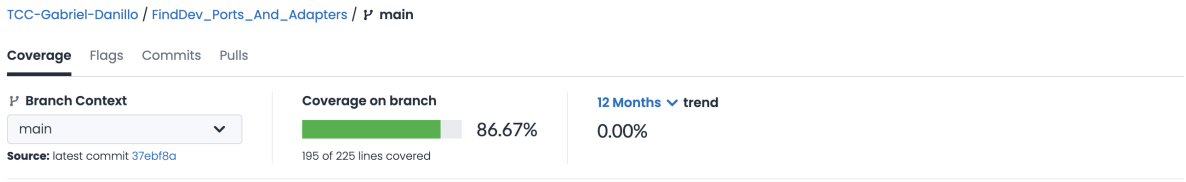
6.1.3.1 FindDev Portas e Adaptadores

A aplicação FindDev implementada, utilizando a arquitetura Portas e Adaptadores, pode ser vista no *Github* pelo *link*:
https://github.com/TCC-Gabriel-Danillo/FindDev_Ports_And_Adapters.

Como pode ser observado no repositório, o desenvolvimento dessa aplicação ocorreu orientando-se pela Metodologia de Desenvolvimento, estabelecida no Capítulo 4, mais especificamente na seção 4.4. Fez-se uso, portanto, de práticas ágeis, com foco nas *features* e histórias de usuário estabelecidas via *backlog*, no descritivo dessa aplicação apresentado no Capítulo 5. Como exemplo, pode-se mencionar a implementação da "*feature* Geolocalização: localização de usuários atualiza conforme movimento do mapa".

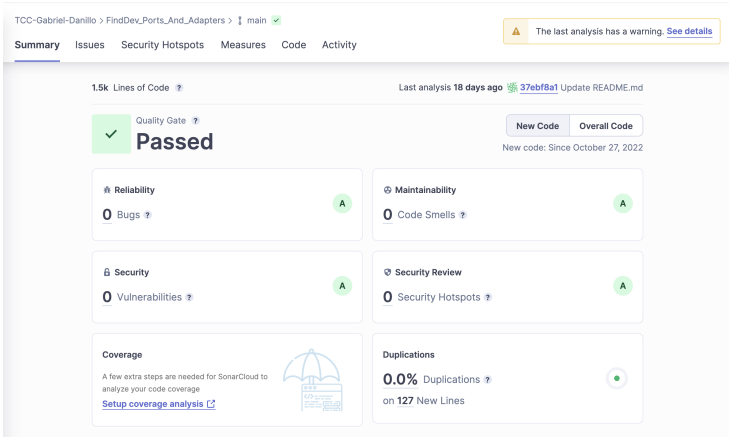
Na sequência, a Figura 33 bem como a Figura 34 demonstram que a meta de cobertura de testes foi alcançada, registrando acima de 80%, mais precisamente 86,67%. Acredita-se que essa cobertura foi obtida muito devido ao uso de boas práticas de programação, orientando-se o mais de acordo possível pela Arquitetura de Portas e Adaptadores, fazendo uso dos referenciais teóricos descritos no Capítulo 2 (seção 2.3.3). Outras percepções são apresentadas adiante, sob o ponto de vista dos autores, atuando como desenvolvedores da aplicação FindDev, em sua versão centrada na Arquitetura Portas e Adaptadores.

Figura 33 – Cobertura de código coletada pela ferramenta Codecov para a aplicação FindDev Portas e Adaptadores.



Fonte: Codecov (2022).

Figura 34 – Métricas coletadas pela ferramenta Sonar para a aplicação FindDev Portas e Adaptadores.



Fonte: SonarQube (2022).

Dentre os diversos desafios para a implementação da Arquitetura Portas e Adaptadores, o primeiro identificado pelos desenvolvedores foi a questão da granularidade. Entende-se por alta granularidade, a necessidade de se criar, por exemplo, serviços - ou seja, um nível de abstração superior, de maior granularidade em termos de componente arquitetural - para cada nova *feature* do *software*. A alta granularidade, também responsável pela elogiada robustez da Arquitetura Portas e Adaptadores, resulta em algumas dificuldades, em especial, na criação de novas *features* no *software*. Nesse caso, há sempre a necessidade de criar um novo serviço para a aplicação. Como consequência, cria-se uma quantidade bastante razoável de arquivos, visando garantir que os princípios da arquitetura sejam respeitados. Ao criar diversos arquivos, que dependem uns dos outros, aumenta sensivelmente as chances do desenvolvedor se perder/se confundir, diante das várias alterações que devem ser realizadas ao longo desse processo.

Observam-se vantagens também, tal como o fato de ser uma arquitetura robusta, permissiva para a realização de testes e escalável. Entretanto, para aplicações simples, como foi o caso do FindDev, na visão dos autores, não é uma arquitetura recomendável. Reporta-se, corroborando com esse ponto de vista, a existência inerente de complexidade para se manter a integridade e a lógica das camadas, seguindo os princípios da arquitetura.

Tal complexidade demanda um esforço muito custoso, não se justificando para o caso de aplicações pequenas.

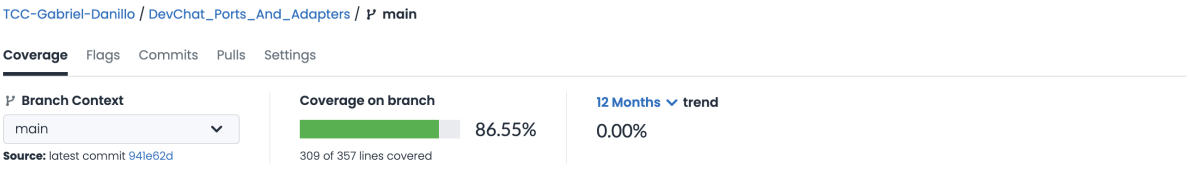
6.1.3.2 DevChat Portas e Adaptadores

A aplicação DevChat, implementada utilizando a arquitetura Portas e Adaptadores, pode ser vista no *Github* pelo *link*: https://github.com/TCC-Gabriel-Danillo/DevChat_Ports_And_Adapters.

Novamente, via repositório, observa-se que o desenvolvimento dessa aplicação ocorreu orientando-se pela Metodologia de Desenvolvimento, estabelecida no Capítulo 4, mais especificamente na seção 4.4, bem como pelo *backlog*, apresentado no descritivo dessa aplicação, no Capítulo 5. Como exemplo, pode-se mencionar a implementação das *features*: "*feature* - Conversa ->Página inicial: *add message input component* " e "*feature* - Conversa ->Página inicial: *add message ballon component*".

Na sequência, a Figura 35 bem como a Figura 36 demonstram que a meta de cobertura de testes foi alcançada, registrando acima de 80%, mais precisamente 86,55%. Novamente, orientou-se pelas boas práticas de programação, em concordância com a Arquitetura de Portas e Adaptadores, fazendo uso dos referenciais teóricos descritos no Capítulo 2 (seção 2.3.3). Outras percepções são apresentadas adiante, sob o ponto de vista dos autores, atuando como desenvolvedores da aplicação DevChat, em sua versão centrada na Arquitetura Portas e Adaptadores.

Figura 35 – Cobertura de código coletada pela ferramenta Codecov para a aplicação DevChat Portas e Adaptadores.

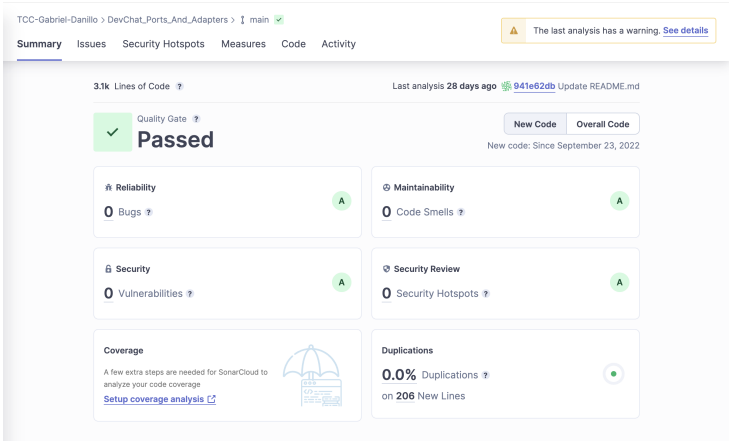


Fonte: Codecov (2022).

Durante todo o processo de desenvolvimento da aplicação DevChat, foram observados desafios. Dentre os principais, cabem ser mencionados:

- Inicialmente, configurar o projeto para que o mesmo pudesse utilizar somente uma fonte de bibliotecas. Para isso, houve necessidade de adaptar o compilador de código para reconhecer os arquivos presentes em pastas, fora do escopo de onde a configuração do projeto em React foi inicializado;
- Manter independência entre as camadas, de forma que as mesmas não tenham dependência de detalhes de implementação das outras, foi desafiador, principalmente para o módulo de mensagem da aplicação. A funcionalidade de tempo real é algo

Figura 36 – Métricas coletadas pela ferramenta Sonar para a aplicação DevChat Portas e Adaptadores.



Fonte: SonarQube (2022).

intrinsecamente atrelado ao Firebase. Todavia, foi possível alcançar esse objetivo, respeitando esse princípio da Arquitetura Portas e Adaptadores, utilizando-se *callbacks*. Esses foram acionados a cada alteração no banco, e passados via métodos de cada camada, como ilustrado na Figura 37, e

- Entender qual era a responsabilidade de cada camada, como por exemplo em qual camada era o ideal de tratar os erros ou adaptar os dados para a comunicação. Via de regra, todas as adaptações e tratativas de erros eram criadas dentro dos serviços, pois os mesmos representavam o "meio de campo" entre os repositórios e os componentes.

Figura 37 – Repositório do banco de dados em tempo real.

```
import { Unsubscribe } from '@firebase/util';
import { parseFirebaseSnapshot, parseCollection, getRefFromArgs } from '@infrastructure/helpers';
import {
    getFirestore,
    Firestore,
    onSnapshot,
} from 'firebase/firestore';
import { QueryOptions, RealtimeDatabaseRepository, VoidCallback } from "../../domain/repositories"

export class FirebaseRealtimeDatabaseRepository  implements RealtimeDatabaseRepository {
    private unsubscribeFunction?: Unsubscribe

    private readonly firestore: Firestore = getFirestore()
    private readonly collections: string[]

    constructor(...collections: string[]){
        this.collections = collections
    }

    watch<T>(cb: VoidCallback<T>, args: QueryOptions): void {
        const collection = parseCollection(this.collections, this.firestore)
        const q = getRefFromArgs(collection, args);
        this.unsubscribeFunction = onSnapshot(q, (querySnapShot) => {
            const docs = parseFirebaseSnapshot<T>(querySnapShot)
            cb(docs)
        });
    }

    unwatch(): void{
        this.unsubscribeFunction?.()
    }
}
```

Fonte: Autores.

6.1.3.3 FindDev MVC

A aplicação FindDev implementada utilizando a MVC pode ser vista no *Github* pelo *link*:
https://github.com/TCC-Gabriel-Danillo/FindDev_MVC.

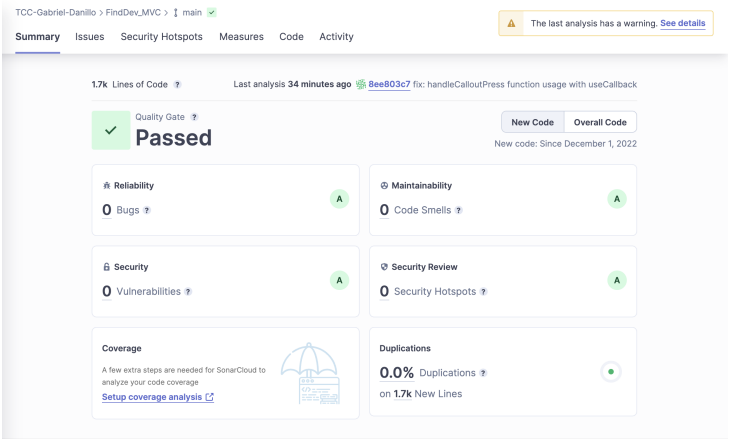
Novamente, via repositório, observa-se que o desenvolvimento dessa aplicação ocorreu orientando-se pela Metodologia de Desenvolvimento, estabelecida no Capítulo 4, mais especificamente na seção 4.4, bem como pelo *backlog*, apresentado no descritivo dessa aplicação, no Capítulo 5. Constam ainda evidências do uso de Mocks ("test: add auth service stub "), e do uso do arcabouço tecnológico estabelecido no Capítulo 3, com Redux ("Setup Redux #1").

Na sequência, a Figura 38 bem como a Figura 39 demonstram que a meta de cobertura de testes foi alcançada, registrando acima de 80%, mais precisamente 82,86%. Novamente, orientou-se pelas boas práticas de programação, em concordância com o Padrão Arquitetural MVC, fazendo uso dos referenciais teóricos descritos no Capítulo 2 (seção 2.3.2). Outras percepções são apresentadas adiante, sob o ponto de vista dos autores, atuando como desenvolvedores da aplicação Find-Dev, em sua versão centrada no Padrão Arquitetural MVC.

Figura 38 – Cobertura de código coletada pela ferramenta Codecov para a aplicação FindDev MVC.

Fonte: Codecov (2022).

Figura 39 – Métricas coletadas pela ferramenta Sonar para a aplicação FindDev MVC.



Fonte: SonarQube (2022).

Durante o processo de desenvolvimento da aplicação Find-Dev orientando-se pelo Padrão Arquitetural MVC, percebeu-se certa dificuldade na realização dos testes de integração, no intuito de superar a meta de cobertura. Para tal, foi necessária a realização da simulação de alguns pacotes, principalmente a biblioteca de gerência de estado (Redux). Dessa forma, foi possível simular o estado das *models* da aplicação durante os testes. A Figura 40 ilustra como ocorreu tal simulação.

Figura 40 – *Mock* para gerência de estado(Redux) para aplicação FindDev MVC

```
import React, { PropsWithChildren } from 'react'
import { render } from '@testing-library/react-native'
import type { RenderOptions } from '@testing-library/react-native'
import type { PreloadedState } from '@reduxjs/toolkit'
import { Provider } from 'react-redux'

import { setupStore } from '_/store'
import type { AppStore, RootState } from '_/store'
import { AuthServiceStub } from '../mocks/authServiceStub'
import { UserServiceStub } from '../mocks/userServiceStub'
interface ExtendedRenderOptions extends Omit<RenderOptions, 'queries'> {
    preloadedState?: PreloadedState<RootState>
    store?: AppStore
}

const authService = new AuthServiceStub()
const userService = new UserServiceStub()

export function renderWithProviders(
    ui: React.ReactElement,
    {
        preloadedState = {},
        store = setupStore({
            preloadedState,
            authService,
            userService
        }),
        ...renderOptions
    }: ExtendedRenderOptions = {}
) {
    function Wrapper({ children }: PropsWithChildren<{}>): JSX.Element {
        return <Provider store={store}>{children}</Provider>
    }
    return { store, ...render(ui, { wrapper: Wrapper, ...renderOptions }) }
}
```

Fonte: Autores.

6.1.3.4 DevChat MVC

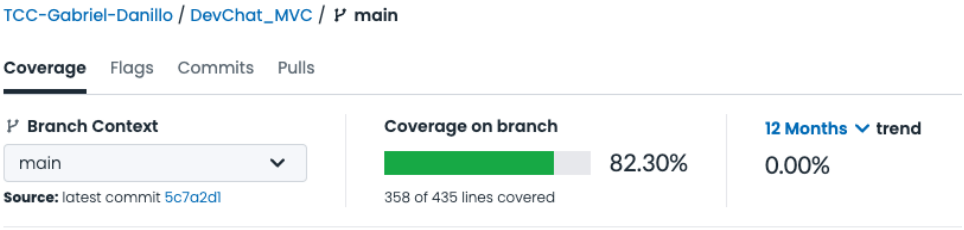
A aplicação DevChat, implementada utilizando MVC, pode ser vista no Github pelo *link*: https://github.com/TCC-Gabriel-Danillo/DevChat_MVC.

Novamente, via repositório, observa-se que o desenvolvimento dessa aplicação ocorreu orientando-se pela Metodologia de Desenvolvimento, estabelecida no Capítulo 4, mais especificamente na seção 4.4, bem como pelo *backlog*, apresentado no descritivo dessa aplicação, no Capítulo 5. Adicionalmente, têm-se evidências sobre os Testes de Integração ("Testes de integracao (#10)"), bem como sobre o uso do Sonar ("add: Adicionado métricas do sonar no README"), sendo esse um dos suportes tecnológicos estabelecidos no Capítulo 3 para uso nesse TCC.

As Figuras 41 e 42 são dedicadas aos testes. Na Figura 42, constam as métricas coletadas pelo Sonar, as quais demonstram que boas práticas de programação foram aplicadas no código. Já, na Figura 41, está ilustrada a cobertura de testes total da aplicação nesse cenário de uso. Tem-se, novamente, cobertura acima de 80%, mais precisamente, 82,30%.

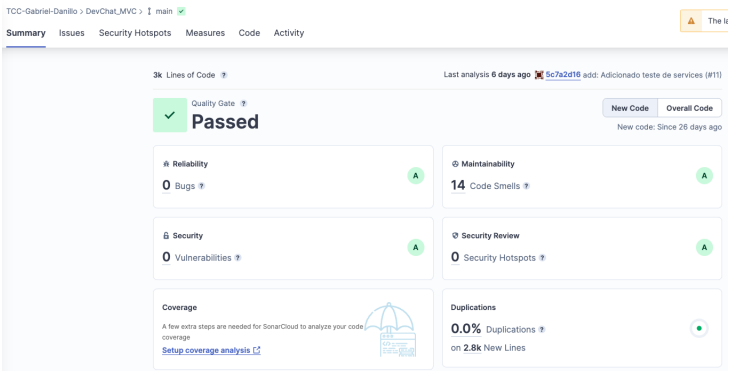
A seguir, na Figura 42, está ilustrada a cobertura coletada pelo Sonar que demonstram que boas práticas de programação foram aplicadas no código. Já, na Figura 41 está ilustrado a cobertura de teste total da aplicação.

Figura 41 – Cobertura de código coletada pela ferramenta Codecov para a aplicação DevChat MVC.



Fonte: Codecov (2022).

Figura 42 – Métricas coletadas pela ferramenta Sonar para a aplicação DevChat MVC.



Fonte: SonarQube (2022).

Dentre as maiores dificuldades encontradas na implementação da aplicação DevChat, centrada no Padrão Arquitetural MVC, a realização de testes de integração foi a de maior destaque. Nesse contexto, registra-se que, além de precisar utilizar o mesmo *mock* da Figura 40, ocorreu, adicionalmente, a necessidade de realizar configurações, de forma explícita, no Jest. O intuito dessas configurações foi ignorar certos arquivos, que não faziam sentido para os testes de integração, e que estavam interferindo, negativamente, na média de cobertura de testes, reduzindo-a.

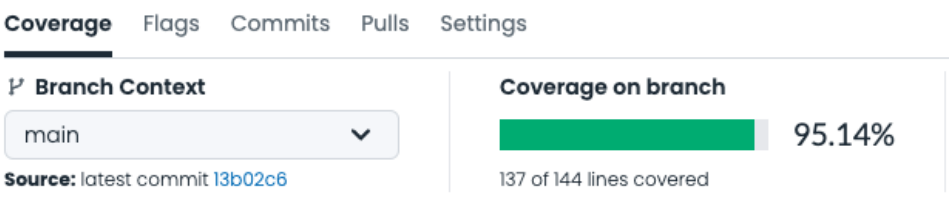
6.1.3.5 FindDev Arquitetura Limpa

A aplicação FindDev, implementada utilizando a Arquitetura Limpa, pode ser vista no Github pelo link: https://github.com/TCC-Gabriel-Danillo/FindDev_Clean_Arch.

Novamente, via repositório, observa-se que o desenvolvimento dessa aplicação ocorreu orientando-se pela Metodologia de Desenvolvimento, estabelecida no Capítulo 4, mais especificamente na seção 4.4, bem como pelo *backlog*, apresentado no descritivo dessa aplicação, no Capítulo 5. Constam ainda evidências do uso teórico dos conceitos abordados no Capítulo 2 e ferramentas abordadas no Capítulo 3, como teste de integração ("Testes integração (#11)") e o uso do Firebase ("Feature infrastucture (#2)").

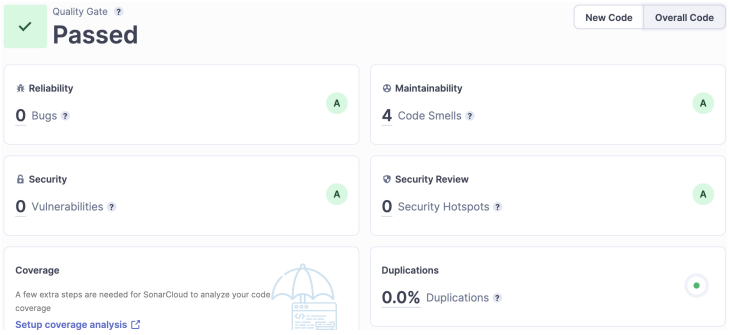
A seguir, na Figura 43, está ilustrado a cobertura de teste total da aplicação. Já, na Figura 44, estão as métricas coletadas pelo Sonar que demonstram que boas práticas de programação foram aplicadas no código.

Figura 43 – Cobertura de código coletada pela ferramenta Codecov para a aplicação FindDev Arquitetura Limpa.



Fonte: Codecov (2022).

Figura 44 – Métricas coletadas pela ferramenta Sonar para a aplicação FindDev Arquitetura Limpa.



Fonte: SonarQube (2022).

Ao desenvolver o FindDev, utilizando Arquitetura Limpa, não foram notados grandes desafios. Por conta de ser a última aplicação desenvolvida, foi possível aproveitar grande parte da lógica e da codificação. Além disso, o FindDev é, tecnicamente, uma aplicação mais simples que o DevChat, e possui quase que, somente, a metade de linhas de código. A maior "complicação" sentida foi estudar para entender os conceitos da arquitetura para o início do desenvolvimento. Após o entendimento da arquitetura, o desenvolvimento fluiu como o planejado.

6.1.3.6 DevChat Arquitetura Limpa

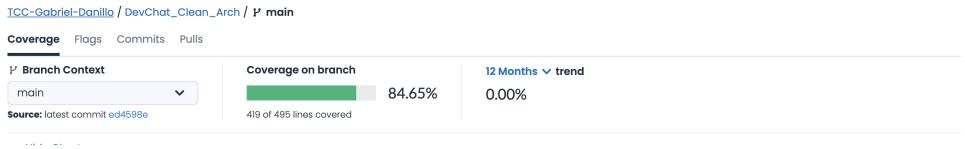
A aplicação DevChat, implementada utilizando a Arquitetura Limpa, pode ser vista no *Github* pelo link: https://github.com/TCC-Gabriel-Danillo/DevChat_Clean_Arch.

Novamente, via repositório, observa-se que o desenvolvimento dessa aplicação ocorreu orientando-se pela Metodologia de Desenvolvimento, estabelecida no Capítulo 4, mais especificamente na seção 4.4, bem como pelo *backlog*, apresentado no descritivo dessa aplicação, no Capítulo 5. Constam ainda evidências do uso teórico dos conceitos abordados no Capítulo 2 bem como uso de ferramentas citadas no Capítulo 3, como o uso do padrão de projeto *Factory* ("feat(main/factories): add conversationServiceFactory") e o uso da ferramenta Jest para realizar os testes de integração ("test: adding test config").

Por se tratar de uma aplicação que já foi desenvolvida em outros dois momentos, notou-se que não houve demais complicações durante a etapa de desenvolvimento, a não ser a curva de aprendizado para entender de forma clara a responsabilidade de cada camada apresentada na Arquitetura Limpa.

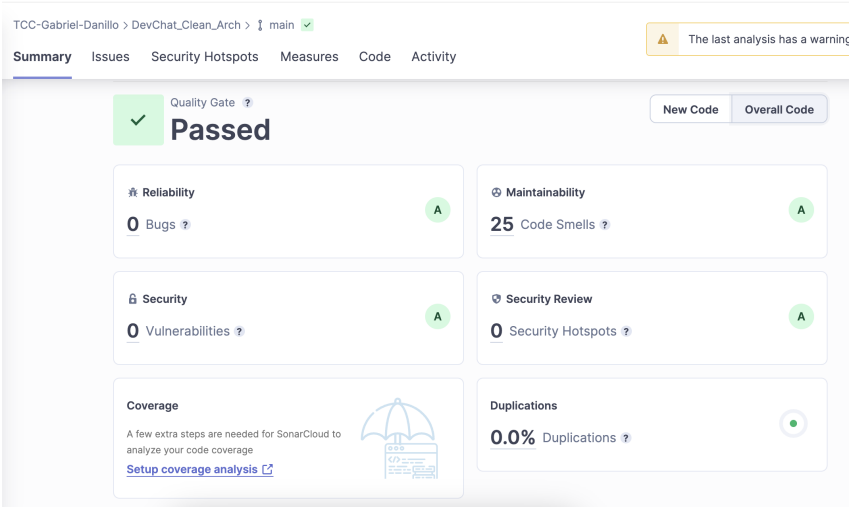
A seguir, na Figura 46, estão as métricas coletadas pelo Sonar que demonstram que boas práticas de programação foram aplicadas no código. Já, na Figura 45, está ilustrada a cobertura de teste total da aplicação.

Figura 45 – Cobertura de código coletada pela ferramenta Codecov para a aplicação DevChat Arquitetura Limpa.



Fonte: Codecov (2022).

Figura 46 – Métricas coletadas pela ferramenta Sonar para a aplicação DevChat Arquitetura Limpa.



Fonte: SonarQube (2022).

6.1.4 Etapa Avaliação

A etapa de avaliação visa relatar as impressões de desenvolvimento de cada arquitetura para o aplicativo FindDev e DevChat, os desafios de implementações, e discutir sobre o desenvolvimento dos testes.

6.1.4.1 FindDev Portas e Adaptadores

No FindDev, por se tratar de uma aplicação já desenvolvida em momentos anteriores, experiências distintas foram percebidas durante o ciclo de desenvolvimento da aplicação. Após a primeira etapa desse TCC, com os novos requisitos sendo elicitados para o aplicativo FindDev, ocorreu a necessidade de manutenção.

Nesse sentido, antes de continuar o desenvolvimento do aplicativo FindDev na segunda etapa do TCC, percebeu-se sobre a quantidade de arquivos necessária para implementar a arquitetura Portas e Adaptadores da maneira correta. Tal complexidade pode ser vista no diagrama de pacotes, ilustrado na Figura 23.

Essa inerente necessidade de se ter componentes arquiteturais de alta granularidade, e em grande quantidade, deve-se aos princípios da arquitetura de Portas e Adaptadores.

Para que seja possível deixar a camada de Interface com comportamento isolado, foi necessária a criação das classes no *repositories*. Tais classes representam a conexão com serviços externos do FindDev, e a criação do diretório dto. Esse diretório representa o formato de dado que cada um dos serviços externos retornam. Toda e qualquer classe criada em *repositories* da Interface possui a capacidade de ter sua lógica interna em qualquer tipo de tecnologia.

Para que os serviços da Interface sejam utilizados, foram criados os *services*, *repositories* e *entities* na camada de Domínio. Tanto as pastas *repositories* e *entities* possuem as portas da arquitetura. Tais estruturas são responsáveis por moldar o formato de dados e classes que irão transitar entre as camadas da arquitetura. O diretório *services* é responsável por utilizar os métodos criados na Interface e manipular os mesmos.

Já no diretório ui, a camada de Aplicação interage com os dados recebidos da Interface e modelados pelo Domínio. Além disso, a camada de Aplicação possui um comportamento bem específico para que possa interagir com outras camadas. Porém, esse caso será mais bem tratado mais adiante.

Retomando sobre a alta granularidade, a mesma ocorreu por conta da interação entre as camadas, especificação dos dados e definição de comportamentos. Cada uma das camadas na Arquitetura Portas e Adaptadores possui funções bem definidas, e os dados que transitam entre as mesmas possuem um formato previamente especificado (GRAÇA, 2017b).

Na camada de Aplicação, para que os dados da Interface e do Domínio pudessem ser utilizados, foram criados contextos, estruturas (*Providers* e *Consumers*) que permitem com que dados da aplicação possam transitar livremente entre todos os seus componentes filhos (FACEBOOK, 2022a). A utilização de contextos possibilitou que comportamentos comuns fossem compartilhados por toda a aplicação, sendo os principais: serviço de autenticação do usuário com Github e Firebase e conexão com o banco de dados Firestore.

A utilização de contextos garante que os princípios da arquitetura sejam seguidos, pois, dessa forma, a camada de Aplicação torna-se totalmente independente de qualquer lógica ou tecnologia utilizada em outras camadas. Os contextos recebem os objetos de classes de camadas externas, e funcionam como serviços exclusivos da camada de Aplicação.

Entretanto, apesar dos contextos facilitarem a implementação da arquitetura, o FindDev é uma aplicação bem simples, utilizada apenas para fins de estudos relacionados à arquitetura de software. Diante dessa particularidade, e como pode ser visto na Figura 47, já existem três contextos distintos, apenas para realizar funções básicas da aplicação. Então, caso o aplicativo fosse maior, para cada nova função que fosse surgindo para o *app*, seria necessário criar um novo contexto. O acúmulo de diversos *providers* pode gerar um problema de desempenho na aplicação, pois os dados que são manipulados para um

Figura 47 – *Providers* de *contexts* na raiz do projeto

Fonte: Autores.

contexto são salvos para todos os filhos do mesmo. Caso haja um acúmulo grande de contextos na raiz do projeto, isso pode afetar o *app* como um todo.

Percebeu-se, também, que devido à especificação dos dados que irão transitar entre as camadas e a independência lógica de cada uma delas, é extremamente simples de realizar manutenção no aplicativo (GRAÇA, 2017b). A tipagem de dados exigida pela estrutura da arquitetura combinada com o TypeScript a torna bastante robusta.

Por fim, foram realizados testes de integração com a abordagem de testes *Top Down*, a qual consiste em testar primeiro as principais *features* de um *sofware* (AWATI, 2022). As etapas de testes foram separadas de acordo com a quantidade de páginas da aplicação, ou seja, constavam dois arquivos de testes no total, cada um para testar as *features* disponíveis na página. Tanto os testes da página inicial, quanto os testes da página de mapa, consistiram em verificar a integração entre os seguintes serviços:

- Autenticação de usuário com o Github;
- Persistência dos dados do usuário, e
- Envio de dados cadastrados para o Firestore.

Para que fosse possível realizar os testes dos serviços, foi necessário criar diversos *stubs*, e realizar uma injeção dos mesmos nos *contexts*, citados anteriormente. *Stubs* são *mocks* mais simples que apenas mudam o comportamento de componente e não o componente por completo (JUNG, 2019). Durante a realização dos testes de integração, a etapa mais penosa foi realizar o *mock* de bibliotecas de terceiros. Porém, uma forma que os desenvolvedores encontraram de contornar tal desafio, sem comprometer os princípios e comportamentos da arquitetura, foi criar uma função à parte, e então passar como dependência para o serviço. Repassar para o *service* uma função customizada de um serviço externo reduziu drasticamente a complexidade dos testes, não sendo mais necessária a criação de um *mock*, mas sim de um *stub* simples.

6.1.4.2 DevChat Portas e Adaptadores

Inicialmente, pode-se notar a escassez de referências disponíveis que utilizam a o React Native como tecnologia de desenvolvimento na Arquitetura Portas e Adaptadores. Por isso, constantemente fez-se pertinente certa "dedução", de acordo com os conceitos da arquitetura, de como seria a correta aplicação nessa tecnologia.

Algumas regras de negócio, como a autenticação, devem estar na camada de domínio, quando é utilizada a Arquitetura de Portas e Adaptadores (GRAÇA, 2017b). Todavia, em algumas funcionalidades, como autenticação por provedores (Google, Github e etc), essa ação é feita pelo uso de *hooks* (REACT, 2022b), algo que está muito associado ao React. Portanto, tal lógica ficaria na camada de aplicação. Por isso, a lógica de autenticação foi implementada, em sua maioria, através de criação de *hooks* customizados (REACT, 2022a). Esse foi usado como um *Adapter*, para isolar a dependência do uso da biblioteca externa de autenticação, que dá origem ao *hook* de autenticação (nota-se que é uma prática muito eficaz para reduzir o acoplamento de código, no qual *hooks* customizados podem agir como serviços dentro de aplicação).

Como já citado no relato do aplicativo FindDev 6.1.4.1, o uso de gerência de estado, utilizando a ferramenta ContextApi, ajudou a manter o desacoplamento de código ao possibilitar o compartilhamento de funções dentro dos componentes do projeto, agindo como uma injeção de dependência, nesse sentido, as funções implementadas pelas camadas de domínio e infraestrutura puderam ser utilizadas nos arquivos da camada de aplicação de forma a não quebrar os princípios de arquitetura, como a própria boa prática de injeção de dependência (MARTIN, 2000).

Como consta na Figura 48, as funções são passadas para o *provider* como *Props*, no qual ficarão acessíveis para o restante da aplicação.

Todavia, vale reforçar, como já citado no relato anterior 6.1.4.1, que há um possível prejuízo em desempenho, uma vez que, à medida que o projeto se desenvolve, mais contextos podem ser adicionados, causando renderização excessiva na árvore de componentes.

Figura 48 – Uso de ContextApi para uso de injeção de dependência.

```
export function AuthProviderComposer({ children }: Props){
    const authPromptService = useAuthPrompt()
    const gitAuthHttp = new HttpRepositoryImp(GITHUB_URL.AUTH_BASE_URL)
    const gitApiHttp = new HttpRepositoryImp(GITHUB_URL.API_BASE_URL)
    const userDbRepository = new FirebaseDatabaseRepository(FIREBASE_COLLECTION.USERS)
    const authService = new AuthService(gitAuthHttp, gitApiHttp, userDbRepository)
    const localStorageRepository = new LocalStorage()

    return(
      <AuthContextProvider
        authPromptService={authPromptService}
        authService={authService}
        localStorageRepository={localStorageRepository}
      >
        {children}
      </AuthContextProvider>
    )
}
```

Fonte: Autores.

No que diz respeito aos aspectos da arquitetura, percebe-se uma alta granularidade dos arquivos. Desta forma, funcionalidades simples geram diversos arquivos. Por exemplo, para desenvolvimento do módulo de mensagem, que comunica com um banco de dados, primeiramente definiu-se:

- Entidade de mensagem, na camada de domínio;
- Caso de uso de mensagem, na camada de domínio;
- Implementação do caso de uso, também na camada de domínio;
- Uso de uma abstração de banco de dados, como repositório, onde a interface está presente na camada de domínio, e sua implementação na camada de infraestrutura;
- Um contexto para compartilhar as funções implementadas anteriormente entre os componentes, e
- Arquivos de interface como páginas e componentes reutilizáveis.

Nota-se que a granularidade natural da arquitetura gera e altera diversos arquivos. Isso deve-se, principalmente, à natureza em camadas da arquitetura.

Inicialmente, devido à quantidade de camadas, há uma impressão de que se está passando a execução de um serviço de uma dada funcionalidade de camada para camada. Essa percepção muitas vezes é compreendida pela expressão popular “brincar de batata quente”. Tal comportamento pode ser ilustrado pela funcionalidade de marcar uma mensagem como lida, na qual uma mensagem tem seu atributo “*read*” atualizado no banco de dados. A ação acontece através das seguintes etapas:

- Uma função presente no contexto, chamada "*markAsRead*", é acionada, passando os dados da mensagem que deve ser atualizada;
- O *context* chama o método da classe *MessageService*, presente na camada de domínio, "*updateMessage*", que altera do dado de "*read*" da mensagem, e
- O serviço aciona o repositório, presente na camada de infraestrutura, que altera o valor da mensagem no banco.

Observa-se que a informação da mensagem transita entre as camadas da aplicação, até chegar no banco de dados. Essa dinâmica, inicialmente, causa uma certa confusão a respeito da responsabilidade de cada camada, como em qual camada deve ocorrer o tratamento de erros, e assim por diante.

Pode-se notar também, como ilustrado na Figura 49, que o acesso aos arquivos de diversos diretórios requer o acesso a diversos caminhos. Todavia, como mostrado na Figura 50, isso pode ser mitigado com o uso de *alias* para caminhos de arquivos, disponível no Typescript, combinado com o uso de compiladores de código como o Babel (WIERUCH, 2022).

Figura 49 – Caminho de arquivos para acesso de funções entre camadas.

```
import { AuthContextProvider } from '../context';
import {
  FirebaseDatabaseRepository,
  HttpRepositoryImp,
  LocalStorage
} from "../../../infrastructure/repositories"
import { AuthService } from "../../../domain/services"
import { FIREBASE_COLLECTION, GITHUB_URL } from "..//constants";
import { useAuthPrompt } from "../hooks";
```

Fonte: Autores.

Por fim, na etapa de testes da aplicação, inicialmente, ocorreu certa dificuldade para encontrar referências detalhadas de como configurar o ambiente de testes para o sistema. Todavia, uma vez configurada, nota-se o motivo pelo qual a arquitetura tem a testabilidade como ponto forte. O uso de camadas através da injeção de dependência simplifica o teste de integração, bastando-se criar *mocks* que simulassem as funções de outras camada e, então, injetando-as na camada a qual se deseja testar. Todavia, criar um *mock* para o uso de *hooks* de terceiros, a princípio, não foi algo intuitivo. Para tal, foram utilizados os *hooks* customizados, que permitiram a criação de *mocks* de forma mais simplificada, somente simulando o *hook* de bibliotecas externas.

Figura 50 – Caminho dos arquivos com o uso de *Alias*.

Fonte: Autores.

6.1.4.3 FindDev MVC

Para o desenvolvimento do aplicativo FindDev utilizando a arquitetura MVC, foi utilizada a biblioteca de gerência de estado Redux, amplamente conhecida na comunidade de desenvolvimento. O motivo de tal escolha deveu-se à arquitetura de desenvolvimento de tal biblioteca. Como citado anteriormente no Capítulo 3, essa arquitetura possui diversas semelhanças com o modelo proposto pelo MVC. Sendo assim, aplicar tal arquitetura foi extremamente ágil, visto a ampla referência de desenvolvimento, e a comunidade ativa de desenvolvedores que se interessam pela biblioteca.

Como já exposto na seção 2.1, arquitetura de software não é uma ciência exata (GRAÇA, 2017c), por isso não há somente uma maneira de desenvolver na Arquitetura MVC. Desta forma, apesar do desenvolvimento com a biblioteca Redux não ser amplamente difundido como Arquitetura MVC, pode-se considerá-la uma adaptação de tal arquitetura, aplicada para o React (BAUMGARTNER, 2021).

Em um primeiro momento, durante o desenvolvimento do aplicativo, pode-se reparar na facilidade e na fluidez de como ocorreu o processo de desenvolvimento. Qualquer situação de dificuldade foi resolvida com facilidade através de buscas rápidas em fóruns de desenvolvimento.

Além disso, pode-se reparar como a lógica que altera o estado da aplicação é bem separada do trecho de código que realiza as mudanças de estado, diferente do uso de Contextos, utilizados no desenvolvimento dos aplicativos anteriores. A Figura 51 mostra o trecho de código para a *action* de usuários, responsável por buscar uma lista de usuários no banco de dados, e com isso, desencadear uma mudança de estado através do método de *dispatch*, que é responsável por solicitar uma mudança no estado da aplicação. Assim, a Figura 52 mostra um *Slice* de usuários, parte do código que fica responsável por administrar o estado da aplicação, tal como: guardar e compartilhar seus valores com o restante do

sistema, bem como alterá-los quando uma ação da *action* de usuários for disparada. Uma mudança de estado permite que um usuário, através da *View*, possa ver alterações em tela, como é o caso da listagem de usuários.

Figura 51 – *Action* de usuário, responsável por solicitar uma mudança no valor do estado de usuários no mapa.

```
import { Dispatch } from "@reduxjs/toolkit"
import { alertError } from "_/helpers";
import { addUsers, usersLoading, usersLoaded } from "_/store/usersStore"
import { AppThunk, LatLng } from "_/types"

const defaultDistanceInM = 5 * 1000;

export const getUsersAction = (location: LatLng): AppThunk => {
    return async (dispatch: Dispatch, getState, { userService }) => {
        try {
            const { auth: { user: authUser } } = getState()
            dispatch(usersLoading())
            const users = await userService.listUsersByDistance(location, defaultDistanceInM)
            const usersWithoutAuthed = users.filter(user => user.id !== authUser?.id)
            dispatch(addUsers(usersWithoutAuthed))

        } catch (err) {
            console.error(err)
            alertError("Algo deu errado ao recuperar usuários.")
        } finally {
            dispatch(usersLoaded())
        }
    }
}
```

Fonte: Autores.

Diante do exposto, as *Views* da aplicação foram em sua maioria isoladas de qualquer lógica que envolvesse regras de negócio, restando somente a função de chamar as *actions* que iniciam o processo de mudança de estado. Logo, o fluxo para uma mudança de estado, geralmente, segue a seguinte estrutura:

- *View* mostra a interface com os estados atualizados para o usuário, e chama as *actions*, quando necessárias, para executar alguma regra de negócio;
- *Actions* recebem as chamadas da *View*, realizam alguma lógica, solicitam mudanças no estado através dos métodos de *dispatch*, e,
- *Slices* guardam os estados, e recebem as solicitações de mudança de estado. Tais solicitações são processadas pelos *reducers*, que dentro dos *slices*, alteram o valor do estado para que a *View* possa ser atualizada.

Apesar da biblioteca não ser orientada a camadas como a de Portas e Adaptadores, isso não significa que ocorra sempre um acoplamento alto entre bibliotecas externas com o restante da aplicação. Ainda sim, foi possível aplicar alguns padrões de projeto como *Adapters* e *Repositories* em conjunto com a injeção de dependência, tirando proveito da inversão de controle. Todavia, ainda sim, para aplicar a injeção de dependência, percebe-se

Figura 52 – *Slice* de usuário, reponsável por aplicar a alteração e guardar os valores de estado para os usuários.

```
1  import { createSlice, PayloadAction } from '@reduxjs/toolkit'
2  import { UserState } from './types'
3  import { User } from '_/types'
4
5  const initialState: UserState = {
6      users: [],
7      isLoading: false
8  }
9
10 export const usersSlice = createSlice({
11     name: 'users',
12     initialState,
13     reducers: {
14         usersLoading: (state) => {
15             state.isLoading = true
16         },
17         addUsers: (state, action: PayloadAction<User[]>)  => {
18             state.users = action.payload
19         },
20         usersLoaded: (state) => {
21             state.isLoading = false
22         }
23     },
24 })
25
26 export const { addUsers, usersLoaded, usersLoading } = usersSlice.actions
27 export const usersReducer =  usersSlice.reducer
28
```

Fonte: Autores.

o acoplamento à biblioteca de gerência de estado, visto que para isso é preciso passar as dependências como argumentos durante a configuração da biblioteca. A Figura 53 mostra como exemplo um *adapter* criado para que os métodos de geolocalização não dependessem de módulos externos. Adiante, a Figura 54 representa uma parte do processo de configuração do Redux, no qual pode ser observada a injeção de dependência através do atributo "*extraArgument*".

Figura 53 – *Adapter* para biblioteca de geolocalização.

```
import { Position, LatLng } from "_/types";
import { LocationAdapter, Bounds } from "./types";
import * as Location from 'expo-location';
import { generateGeoHash, generateHashBounds } from "_/helpers";

export class LocationAdapterImp implements LocationAdapter {
    async requestPermission() {
        const { status } = await Location.requestForegroundPermissionsAsync();
        return status === Location.PermissionStatus.GRANTED
    }

    async getCurrentPosition(): Promise<Position> {
        const location = await Location.getCurrentPositionAsync({});
        const { latitude, longitude } = location.coords
        return {
            location: {
                latitude,
                longitude
            },
            geohash: generateGeoHash(latitude, longitude)
        }
    }

    generateGeoHashBounds(location: LatLng, distanceInM: number): Bounds {
        const bounds = generateHashBounds(location, distanceInM)
        return bounds
    }
}
```

Fonte: Autores.

Figura 54 – Objeto de configuração do Redux. Dependências extras são passadas para "*extraArgument*" para serem injetadas nas *actions*

```
type StoreOptions = {
  preloadedState?: PreloadedState<RootState>,
  authService: AuthService,
  userService: UserService
}
export const setupStore = (options: StoreOptions) => {
  const { authService, userService, preloadedState } = options
  return configureStore({
    reducer: persistedReducer,
    preloadedState,
    middleware(getDefaultMiddleware) {
      return getDefaultMiddleware<MiddlewareOptions>({
        serializableCheck: false,
        thunk: {
          extraArgument: {
            authService,
            userService
          }
        }
      })
    },
  })
}
```

Fonte: Autores.

Por fim, durante a etapa de testes, sentiu-se um pouco mais de resistência de escrever os testes sem depender de módulos, principalmente com o Redux. Sendo necessário, como

já citado, simular seu comportamento durante os testes para que se pudesse criar os testes de integração corretamente.

6.1.4.4 DevChat MVC

Para o desenvolvimento do DevChat com a arquitetura MVC, também, foi utilizada a biblioteca Redux como base. O Redux, como explicado anteriormente no Capítulo 3, possui um comportamento pautado na arquitetura MVC (BAUMGARTNER, 2021). Por isso, foi uma biblioteca escolhida pelos desenvolvedores para auxiliar no desenvolvimento. A princípio, o uso do Redux foi uma experiência nova para o desenvolvimento, resultando em certa dificuldade. Por isso, foi necessário, antes do inicio do desenvolvimento, um breve período de adaptação e aprendizagem dos princípios da biblioteca.

Assim que uma base sólida de conhecimento foi estabelecida, a construção da aplicação foi extremamente simples e rápida. Existem diversos conteúdos de qualidade na internet que auxiliaram na construção da aplicação em MVC com Redux. O desenvolvimento da aplicação foi muito simples, devido à alta coesão e ao baixo acoplamento entre as camadas do MVC (GRAÇA, 2017a). Por exemplo, toda e qualquer lógica construída na camada da Model não impactou de forma direta a camada View.

Além disso, muito da lógica de serviços construídos para a arquitetura anterior, Portas e Adaptadores, puderam ser adaptados e reutilizados para o MVC. Todos os serviços existentes, anteriormente utilizados em contextos do React Native, foram reaproveitados e injetados para serem utilizados diretamente nas *actions*, estrutura fundamental para garantir os princípios do MVC com Redux. Cada *action*, como explicado anteriormente na seção 6.1.4.3, realiza uma chamada lógica para mudanças de estados da aplicação. Na Figura 55, há um trecho de código das *actions* de autenticação.

Figura 55 – *Actions* de autenticação

```
export const authenticateGithub = (credentials: AuthCredentialType): AppThunk => {
  return async (dispatch: Dispatch, _, { authService }) => {
    dispatch(authIsLoading());
    const user = await authService.authenticateGithub(credentials);
    dispatch(authedUser(user));
    dispatch(authIsLoaded());
  };
};

export const logoutAction = (): AppThunk => {
  return async (dispatch) => {
    dispatch(removeUser());
  };
};
```

Fonte: Autores.

Em havendo mais tarefas para serem realizadas, bastaria, apenas, criar uma nova

action e utilizar o *authService*. O serviço injetado na *store* global, em uma estrutura semelhante à Figura 54, é compartilhado com todas as *actions* disponíveis. Tal injeção realizada foi capaz de tornar o código de todos os serviços disponíveis reutilizável para todas as *actions*, e fechado apenas para o escopo das *actions*, característica bastante vantajosa na Arquitetura MVC (POPE, 2010).

Uma das facilidades que o MVC possibilitou, especificamente para a aplicação DevChat, foi na implementação das tarefas relacionadas ao serviço *messageService*. Por exemplo, durante o desenvolvimento da mesma *feature* na Arquitetura Portas e Adaptadores, foi necessário utilizar um contexto dinâmico, aninhado com outro contexto genérico de mensagens, para renderizar corretamente as mensagens na página. Sendo assim, foi implementada uma estrutura extremamente complexa. Na Figura 56, há um trecho do contexto dinâmico. Já no MVC, foi possível implementar o padrão de projeto *Builder* no *messageService*, sendo esse um padrão utilizado para criação de objetos complexos (SHVETS, 2022), para cada nova conversa que fosse aberta. Na Figura 57, consta um trecho da *action* que constrói a lógica da mensagem.

Figura 56 – Contexto dinâmico no DevChat com arquitetura Portas e Adaptadores

```
function MessageProviderComposer({ children, conversation }: Props) {
  const messageDatabaseRepository = new FirebaseDatabaseRepository(
    FIREBASE_COLLECTION.CONVERSATIONS,
    conversation.id,
    FIREBASE_COLLECTION.MESSAGES
  );
  const messageRealtimeDatabaseRepository =
    new FirebaseRealtimeDatabaseRepository(
      FIREBASE_COLLECTION.CONVERSATIONS,
      conversation.id,
      FIREBASE_COLLECTION.MESSAGES
    );
  const userDatabaseRepository = new FirebaseDatabaseRepository(
    FIREBASE_COLLECTION.USERS
  );

  const messageService = new MessageService(
    messageDatabaseRepository,
    messageRealtimeDatabaseRepository,
    userDatabaseRepository
  );

  return (
    <MessageContextProvider
      messageService={messageService}
      conversation={conversation}
    >
      {children}
    </MessageContextProvider>
  );
}
```

Fonte: Autores.

Apesar do MVC ter possibilitado uma construção acelerada da aplicação, a realização de testes de integração no DevChat foi bastante trabalhosa, e a qualidade dos testes foi inferior em relação às demais arquiteturas estudadas. Como foi dito na seção 6.1.3.4, algumas estruturas tiveram de ser ignoradas no teste para que fosse possível alcançar uma

Figura 57 – *Action* que constrói a lógica das mensagens

```
export const listenMessages = (currentConversation: Conversation): AppThunk => {
  return async (dispatch: AppDispatch, _1, { messageService }) => {
    dispatch(initNewConversation(currentConversation));
    dispatch(loadingMessages());

    messageService.listenMessages((messages: Message[]) => {
      dispatch(loadedMessages());
      dispatch(setMessages(messages));
      dispatch(updateConversation(messages, currentConversation));
    });
  };
};

const initNewConversation = (conversation: Conversation): AppThunk => {
  return async (_1, _2, { messageService }) => {
    const conversationId = conversation.id;
    messageService.setCollectionMessageDB(CONVERSATIONS, conversationId, MESSAGES);
    messageService.setCollectionMessageDBRealTime(CONVERSATIONS, conversationId, MESSAGES);
  };
};
```

Fonte: Autores.

cobertura de testes considerável, em atendimento ao pré estabelecido como requisito nesse TCC, ou seja, pelo menos 80%.

6.1.4.5 FindDev Arquitetura Limpa

Para o desenvolvimento da aplicação FindDev foi necessário um esforço maior de estudos sobre a Arquitetura Limpa quando comparado as outras arquitetura estudadas anteriormente. A arquitetura proposta por Robert C. Martin, apesar de ser bastante efetiva no que se propõem a fazer, não é tão difundida e conhecida quanto as outras estudadas nesse trabalho. Por conta disso, o período para juntar boas referências e conteúdos de qualidade para a execução do projeto foi ligeiramente mais extenso.

A Arquitetura Limpa divide em camadas as diferentes responsabilidades da aplicação, a fim aumentar a escalabilidade do projeto (MARTIN, 2017). Tal comportamento de granular bem as distintas obrigações do código assemelhou-se bastante com a Arquitetura de Portas e Adaptadores, e produziu resultados e abordagens semelhantes, porém, com uma maior granularidade.

Como mencionado anteriormente na seção 5.4.2.3, o diretório *Data* é onde a implementação dos casos de uso do sistema ocorre, ou seja, é onde estão as classes com os serviços que o FindDev realiza. Assim como na arquitetura anterior, foi possível reaproveitar o código das funcionalidades base do FindDev: Autenticação do usuário com o Github, fornecimento de coordenadas e gestão de usuários da aplicação. Além disso, vale lembrar que para a implementação de cada um dos casos foram criadas interfaces no diretório *Domain*, todos os serviços da aplicação devem ser previamente especificados nesse diretório com o intuito de garantir uma robustez do código.

Assim como na Arquitetura de Portas e Adaptadores, para que fosse possível

utilizar os serviços implementados no diretório *Data*, foi necessária a criação de contextos no diretório *Presentation*, camada responsável pela parte visual da aplicação. Os serviços são injetados em tais contextos e os contextos transmitem os comportamentos dos casos de uso implementados para a aplicação em geral. Na Figura 58, há a injeção dos serviços na raiz do projeto. Para cada novo serviço implementado, é necessário implementar um novo contexto.

Figura 58 – Contextos com injeção de serviços

```
export const Main = () => {
  const gitApi = new HttpsClient(GITHUB_URL.API_BASE_URL);
  const gitAuth = new HttpsClient(GITHUB_URL.AUTH_BASE_URL);

  const location = new LocationService();

  const userDatabase = new Database("users");
  const localStorage = new LocalStorage();
  const authService = new AuthService(gitApi, gitAuth);
  const userService = new UsersService(location, userDatabase);

  const [usingFonts] = useCustomFonts();

  if (usingFonts)
    return (
      <LocationContextProvider locationService={location}>
        <AuthContextProvider authService={authService} localStorage={localStorage}>
          <UserContextProvider localStorage={localStorage} userService={userService}>
            <Routes />
          </UserContextProvider>
        </AuthContextProvider>
      </LocationContextProvider>
    );
  return <></>;
};
```

Fonte: Autores.

Por conta de todos os componentes do sistema serem totalmente desacoplados uns dos outros, a realização de testes de integração foi muito simples de ser realizada. Foi necessário, para a realização dos testes, a criação de pouquíssimos *Mocks*, tais esses bem simples, e *Stubs* dos serviços. Assim que os *Stubs* criados são injetados nos contextos do aplicativo, é possível simular o funcionamento da aplicação e a comunicação entre os serviços e camadas distintas. Não foi complicado obter uma porcentagem razoável de testes utilizando a Arquitetura Limpa.

Ao desenvolver a aplicação, pode-se perceber que existem bastante semelhanças entre as implementações da Arquitetura Limpa e Arquitetura Portas e Adaptadores, discutida na seção 6.1.4.1. Desde a alta granularidade e tipagem de dados que transitam entre as camadas, até a utilização de contextos na raiz do projeto, que podem causar problemas de desempenho, caso existam muitos contextos acumulados.

A Arquitetura Limpa proporciona uma grande robustez devido à especificação de seus casos de uso e o isolamento das camadas. Isso fornece uma segurança a mais para o desenvolvedor durante futuras manutenções. Além disso, possibilita uma maior

facilidade para realização de testes, tanto de integração quanto unitários, na aplicação. Apesar dos benefícios que a Arquitetura Limpa proporciona, a mesma pode não ser a melhor opção para aplicações muito simples, como o caso do FindDev, por conta de ser uma arquitetura mais complexa, com menos exemplos disponíveis; e bastante pragmática de ser implementada a principio.

6.1.4.6 DevChat Arquitetura Limpa

No desenvolvimento do aplicativo DevChat para a Arquitetura Limpa, pode-se notar a semelhança entre os desafios encontrados durante o desenvolvimento do mesmo aplicativo para a arquitetura de Portas e Adaptadores. Isso se deve ao fato de ambas serem arquiteturas mais orientadas a domínio, possuindo alta granularidade entre as camadas.

Durante o desenvolvimento, o primeiro desafio foi entender a responsabilidade de cada camada para que, então, fosse possível criar os códigos em sua respectiva camada. Todavia, é possível notar a escassez de referências de exemplos de implementações em React Native nessa arquitetura. Para tal, foi feito o uso de algumas referências de implementações em outras tecnologias, visto que a responsabilidade de cada camada é intrínseca a qualquer ferramenta.

Uma vez compreendendo sobre as camadas e suas responsabilidades, a proposta dessa arquitetura fica bastante clara: levar independência entre cada camada e possibilitar o desenvolvimento de um sistema resistente a qualquer alteração. Cada camada implementa uma responsabilidade e declara dependências que devem ser implementadas em outra camada mais acima. Desta forma, seguindo a Figura 32 evidenciada na seção 5.4.2.3, no DevChat têm-se:

- Camada de *Domain*: Diretório, onde foram definidas as regras de negócio da aplicação para enviar mensagem, autenticação, iniciar conversa e assim por diante;
- *Data*: Implementação da lógica dos serviços da aplicação baseado nos casos de uso. Nessa camada, foram implementados os serviços de gestão de usuários, conversas e mensagens. Além disso, também foram definidas abstrações de estruturas que tais serviços necessitam para o funcionamento pleno.
- *Infra*: Basicamente, é a camada que implementa a comunicação com os serviços externos da aplicação, no caso do DevChat, o Firebase e o *Axios*. Então, por exemplo, se para enviar uma mensagem, a classe implementada em *Data* precisa salvar um documento de mensagem no banco de dados do Firebase, ela declara a interface para tal comunicação e a camada de *Infra* provê a implementação da classe para essa abstração, comunicando-se diretamente com o SDK do Firebase para salvar uma nova mensagem.

- *Presentation*: A camada de *Presentation* possui bastante afinidade com a tecnologia principal desse trabalho, o React Native. Tudo relacionado ao visual da aplicação foi criado nessa camada, e

- *Main*: Nessa camada, foi implementada a injeção dos serviços dos casos de uso, das classes da camada de *Infra* e, por fim, a pilha de navegação entre todas as telas da aplicação. Essa camada é a que junta todas as demais para que a aplicação funcione como o esperado.

Para esse projeto, o uso de algumas boas práticas ajudaram a manter o código bem estruturado e pouco rígido, ou seja de fácil alteração. Foi utilizado o padrão de projeto *Factory* para gerar os casos de usos que eram injetados na camada de *Main* para a camada de *Presentation*, como mostrado na Figura 59. A Figura 60 mostra o uso de tal *Factory*, no qual observa-se que a função *ConvesationProviderComposer* não possui informação dos detalhes de implementação do caso de uso, somente de sua interface. Isso possibilitaria a troca do caso de uso, caso necessário, contanto que o mesmo implemente a mesma interface.

Figura 59 – *Factory* para o caso de uso de conversa.

```
export const makeConversationService = (): ConversationUseCase => {
    const conversationDatabaseRepository = new FirebaseDatabaseRepository(
        FIREBASE_COLLECTION.CONVERSATIONS
    )
    const userDatabaseRepository = new FirebaseDatabaseRepository(
        FIREBASE_COLLECTION.USERS
    )
    const conversationRealTimeDatabaseRepository = new FirebaseRealtimeDatabaseRepository(
        FIREBASE_COLLECTION.CONVERSATIONS
    )

    return new ConversationService(
        conversationDatabaseRepository,
        userDatabaseRepository,
        conversationRealTimeDatabaseRepository
    )
}
```

Fonte: Autores.

Figura 60 – Uso da *Factory* para o caso de uso de conversa.

```
export function ConvesationProviderComposer({ children }: Props){
    const conversationService = makeConversationService()

    return(
        <ConversationContextProvider conversationService={conversationService}>
            {children}
        </ConversationContextProvider>
    )
}
```

Fonte: Autores.

Adicionalmente, foi utilizado o padrão de projeto *Adapter* para que se fosse gerada uma inversão de dependência com relação às bibliotecas externas. Essa prática é muito comum na comunicação entre a camada de *Data* e *Infra*, uma vez que a primeira camada define protocolos que necessitam de comunicação externa, e a segunda as implementa. A seguir, a Figura 61 mostra uma interface que serve de protocolo para a camada de *Data*. A Figura 62 mostra a implementação dessa interface na camada de *Infra*, utilizando a biblioteca *Axios* para realizar requisições Http. Vale notar que *Data* não possui conhecimento de detalhes de implementação, tampouco do uso do *Axios* como biblioteca para fazer requisição, graças ao uso do padrão *Adapter* e da injeção de dependência.

Figura 61 – *Adapter* para *Http*.

```
export interface HttpClient {
    get: <T>(endpoint: string, config?: Record<string, any>, params?: any) => Promise<T | undefined>;
    post: <T>(endpoint: string, data: Record<string, any>, config?: any) => Promise<T | undefined>;
}
```

Fonte: Autores.

Figura 62 – Implementação de *Adapter* para *Http* com *Axios*.

```
export class AxiosHttpClient implements HttpClient {
    api: AxiosInstance;
    constructor(baseURL: string){
        this.api = axios.create({
            baseURL,
            headers
        })
    }
   async post <T>(endpoint: string, data: Record<string, any>, config?: any): Promise<T | undefined>  {
        try {
            const response = await this.api.post(endpoint, data, { ...config });
            return response.data as T;
        } catch(error){
            if (error instanceof AxiosError) throw this.handleAxiosError(error)
            throw new ServerError()
        }
    }
     async get <T>(endpoint: string, config?: any, params?: Record<string, any>): Promise<T | undefined> {
        try {
            const response = await this.api.get(endpoint, { ...config, params });
            return response.data as T;
        } catch(error){
            if (error instanceof AxiosError) throw this.handleAxiosError(error)
            throw new ServerError()
        }
    }

    handleAxiosError(error: AxiosError<any, any>) {
        const { status, message } = error
        const errorFromRes = ErrorFromRes(status, message)
        if (errorFromRes) return errorFromRes
        return new ServerError(message)
      }
}
```

Fonte: Autores.

Devido à experiência adquirida pelo desenvolvimento dos outros *Apps* e à natureza dessa arquitetura, que facilita o desenvolvimento de testes com base em suas camadas, os testes foram bastantes fáceis de se realizar. Cada camada, por declarar suas dependências como interface, possibilita a criação de *mocks* que implementam tais interfaces e que são injetadas na respectiva camada durante os testes.

Por fim, como já citado, essa arquitetura possui um aspecto bastante resistente, permitindo mudanças de grau elevado como alteração de bibliotecas e refatorações, sem

que haja grandes riscos de se introduzir erros. Todavia, é de certa maneira complexa para iniciantes na programação com React Native, devido à sua curva de aprendizado e à escassez de referências de desenvolvimento.

6.2 Resumo do Capítulo

Este Capítulo apresentou os resultados obtidos ao longo das etapas da pesquisa ação, descritas no Capítulo 4. A Etapa de Exploratória definiu o conteúdo no qual o atual trabalho encontra-se baseado. Em seguida, a Etapa de Planejamento aborda os processos realizados para o início do desenvolvimento das aplicações. Seguido, então, pela Etapa de Ação que compreendeu o desenvolvimento das aplicações FindDev e DevChat. Por fim, a Etapa de Avaliação, documentou as impressões de desenvolvimento de cada um dos desenvolvedores.

7 Conclusão

Esse capítulo tem como objetivo apresentar as conclusões finais sobre o que foi desenvolvido nesse trabalho. Inicialmente, será retomado o contexto geral que justifica a realização desse trabalho. Depois, há exposição quanto ao status do trabalho, conferindo se os objetivos específicos foram atendidos e resposta à questão de pesquisa. Adiante, têm-se as contribuições que esse projeto faz à comunidade, bem como suas principais fragilidades. Por fim, com base nessas fragilidades, são tecidas possibilidades de trabalhos futuros.

7.1 Contexto Geral

A arquitetura de *Software* é um aspecto de extrema relevância para desenvolvimento de sistemas, pois ela permite criar soluções que atendam problemas em curto e longo prazos, de uma forma eficiente e escalável (GRAÇA, 2017c). Aplicações que possuem um código com uma estrutura frágil, viscosa e com muitas duplicações tendem ao insucesso (MARTIN, 2017).

Nesse aspecto, com a importância de uma bom design de *software* aliado à popularidade dos dispositivos móveis e tecnologias de desenvolvimento de aplicativos híbridos, surge o interesse de se estudar mais profundamente o comportamento de algumas arquiteturas relevantes para a comunidade, quando aplicadas à tecnologias mais emergentes, tal como o React Native. A ideia foi conferir uma visão mais prática e concreta sobre o uso dessas arquiteturas no desenvolvimento de aplicativos móveis, com relatos sobre as percepções de especialistas e testes de integração.

7.2 Status

Nas próximas seções, retoma-se os objetivos específicos acordados no Capítulo 1, buscando reportar se os mesmos foram cumpridos ou não. Com base no cumprimento desses objetivos, procura-se responder à questão de pesquisa, conferindo insumos adicionais, sendo esses adquiridos com o desenvolvimento desse projeto.

7.2.1 Objetivos

Foi especificados como objetivos do trabalho:

- Realização de um levantamento bibliográfico, no intuito de acordar as arquiteturas mais aderentes ao desenvolvimento de aplicativos móveis. *Status*: Cumprido,

apresentado no Capítulo 2;

- Identificação, também com base na literatura especializada, de princípios, métricas e parâmetros que permitam expor de forma mais clara sobre os comportamentos observados com a aplicação das arquiteturas. *Status*: Cumprido, apresentado no Capítulo 2;
- Especificação de cenários de uso, no domínio de desenvolvimento de aplicativos móveis, nos quais são aplicadas as arquiteturas. *Status*: Cumprido, apresentado no Capítulo 5;
- Implementação dos os cenários de uso orientando-se pelas arquiteturas em estudo. *Status*: Cumprido, apresentado no Capítulo 6 e
- Apresentação e análise dos resultados obtidos ao longo dos estudos *Status*: Cumprido, apresentado em no Capítulo 6.

Ao se atingir os objetivos específicos, atingiu-se também o objetivo geral deste trabalho, que foi a realização de estudos sobre algumas das principais arquiteturas de software, aplicando-as no desenvolvimento de aplicativos móveis.

7.2.2 Questão de pesquisa

A questão de pesquisa levantada no Capítulo 1 trouxe um questionamento a respeito dos comportamentos observados ao se aplicar as arquiteturas MVC, Portas e Adaptadores e Arquitetura Limpa para o desenvolvimento de aplicativos móveis.

Desta forma, uma análise mais detalhada sobre a questão pode ser observada no Capítulo 6, que apresenta os resultados obtidos durante a pesquisa.

Todavia, de forma geral, pôde-se observar a subjetividade da arquitetura, que para cada caso, foi adaptada para se adequar à tecnologia em uso de forma que sua essência fosse preservada. Logo, alguns comportamentos como o uso de *Hooks*, caminho de arquivos através de *alias* para acesso de funções em cada camada, e a injeção de dependência utilizando-se de uma biblioteca de gerência de estados foram os principais aspectos observados em todas as arquiteturas, sendo utilizados como ferramentas para potencializar seus usos.

7.3 Contribuições e Fragilidades

Entender o caso de uso de uma arquitetura e como a mesma pode se comportar de acordo com a tecnologia em uso são aspectos de extrema relevância, que diferenciam desenvolvedores mais experientes daqueles mais leigos (MARTIN, 2017) (GRAÇA, 2017c).

Uma boa arquitetura significa um sistema menos frágil, fácil de alterar e manter (GRAÇA, 2017c). Por isso, os resultados coletados por esse trabalho relevantes para desenvolvedores terem um contato inicial com algumas das principais arquiteturas de *Software*, quando aplicadas à tecnologia React Native de forma, visando ampliarem seus conhecimentos em arquiteturas de interesse no mercado de *software*.

Todavia, algumas fragilidades podem ser apontadas. Por conta do escopo do trabalho, focado em diversos aplicativos para diversas arquiteturas, fez com o que não fosse possível aprofundar a análise em nenhuma arquitetura estudada. Também, o uso das mesma ferramentas para todos os aplicativos fez com que fosse possível reaproveitar diversos conhecimentos entre os desenvolvimentos, o que facilitou a etapa de ação do trabalho, mas que pode ter prejudicado a simulação de um cenário real de desenvolvimento, no qual requisitos mudam de forma constante.

7.4 Trabalhos Futuros

Considerando as fragilidades apresentadas na seção anterior, alguns trabalhos futuros podem realizados, tais como:

- Em um trabalho focado em uma única arquitetura, o desenvolvedor teria, a princípio, o objetivo de criar um aplicativo completo e testar, assim como feito por esse trabalho, utilizando diversas tecnologias disponíveis na comunidade, como por exemplo: Axios, MongoDB, Firebase, Redux, dentre outros. Em um primeiro momento, o trabalho poderia focar no desenvolvimento do aplicativo. Depois, seria interessante alterar as ferramentas em uso, visando realizar uma análise sobre o processo de alteração de requisitos;

- Utilizar os aplicativos desenvolvidos por esse trabalho, porém alterar as ferramentas em uso para uma análise de como se comporta a etapa de manutenção para as arquiteturas tratadas;

- Aplicar uma das arquiteturas em um aplicativo mais robusto, com a funcionalidade de vídeo *Streaming*. A escolha de tal funcionalidade deve-se à complexidade associada a mesma, compreendendo, por exemplo: uso de *caching*; codificação de vídeo e áudio; gerência de conteúdo; uso métricas como número de visualização, *likes*, e assim por diante.

Referências

ABRAMOV, D. Why use redux. 2021. Disponível em: <https://react-redux.js.org/introduction/why-use-react-redux>. Acesso em: janeiro. 2023. Citado 2 vezes nas páginas 54 e 55.

AJALA, V. et al. Análise da complexidade ciclomática como apoio ao processo de desenvolvimento do pensamento algorítmico. In: *XXXVI Congresso da Sociedade Brasileira de Computação*. Santo Angelo, RS, Brasil: DesafIE - 5º Workshop de Desafios da Computação aplicada à Educação, 2016. Anais. Citado na página 66.

AL-BADAREEN, A. B. et al. The impact of software quality on maintenance process. *International journal of computers*, v. 5, n. 2, p. 1–8, 2011. Citado na página 29.

Android for Developers. Android for developers. 2022. Disponível em: <https://developer.android.com/>. Acesso em: agosto. 2022. Citado na página 26.

ARKUSNEXUS, P. *Software Architecture, a Building Construction Analogy*. 2020. Disponível em: <https://www.arkusnexus.com/blog/software-architecture-construction-analogy>. Acesso em: julho. 2022. Citado na página 34.

AWATI, R. Integration testing or integration and testing (it). 2022. Disponível em: <https://www.techtarget.com/searchsoftwarequality/definition/integration-testing#:~:text=Common%20approaches%20to%20integration%20testing,them%20all%20as%20one%20unit.> Acesso em: outubro. 2022. Citado 3 vezes nas páginas 48, 49 e 103.

BAILÉN, R. Understanding hexagonal architecture with a spring boot implementation. 2019. Disponível em: <https://betterprogramming.pub/hexagonal-architecture-with-spring-boot-74e93030eba3>. Acesso em: julho. 2022. Citado 3 vezes nas páginas 29, 43 e 80.

BAKOTA, T. et al. A cost model based on software maintainability. p. 316–325, 2012. Disponível em: <https://ieeexplore.ieee.org/document/6405288>. Acesso em: julho. 2022. Citado na página 26.

BAUMGARTNER, R. M. How react and redux brought back mvc and everyone loved it. 2021. Disponível em: <https://rangle.io/blog/how-react-and-redux-brought-back-mvc-and-everyone-loved-it>. Acesso em: janeiro. 2023. Citado 4 vezes nas páginas 55, 88, 107 e 111.

BOUKHARY, S.; COLMENARES, E. A clean approach to flutter development through the flutter clean architecture package. In: *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. Las Vegas, NV, USA: IEEE, 2019. p. 1115–1120. Citado 3 vezes nas páginas 29, 47 e 64.

BROWN, P. What is hexagonal architecture? 2014. Disponível em: <http://culttt.com/2014/12/31/hexagonal-architecture/>. Acesso em: julho. 2022. Citado 4 vezes nas páginas 29, 45, 64 e 79.

CLEMENT, J. Global mobile data traffic from 2017 to 2022. 2020. Disponível em: <https://www.statista.com/statistics/271405/global-mobile-data-traffic-forecast/>. Acesso em: julho. 2022. Citado na página 38.

COCKBURN, A. Hexagonal architecture. 2005. Disponível em: <https://alistair.cockburn.us/hexagonal-architecture/>. Acesso em: julho. 2022. Citado na página 43.

CODECOV. *Codecov Docs*. 2022. Disponível em: <https://about.codecov.io/>. Acesso em: Janeiro. 2023. Citado 7 vezes nas páginas 56, 93, 94, 96, 98, 99 e 101.

DANIELS, J. Modeling with a sense of purpose. *IEEE software*, IEEE, v. 19, n. 1, p. 8–10, 2002. Citado na página 34.

ESPINDOLA, R. S. de; MAJDENBAUM, A.; AUDY, J. L. N. Uma análise crítica dos desafios para engenharia de requisitos em manutenção de software. p. 226–238, 2004. Citado na página 27.

FACEBOOK. Getting started. 2022. Disponível em: <https://reactjs.org/>. Acesso em: novembro. 2022. Citado na página 102.

FACEBOOK. What is jest. 2022. Disponível em: <https://jestjs.io/pt-BR/>. Acesso em: novembro. 2022. Citado na página 54.

FERNANDES, D. Expo: o que é, para que serve e quando utilizar? 2018. Disponível em: <https://blog.rocketseat.com.br/expo-react-native/>. Acesso em: agosto. 2022. Citado 2 vezes nas páginas 54 e 78.

FERREIRA, A. de H.; ANJOS, M. dos. *Dicionário Aurélio básico da língua portuguesa*. Rio de Janeiro, RJ: Editora Nova Fronteira, 1988. Citado na página 34.

FIREBASE. *Firebase Documentation*. 2022. Disponível em: <https://firebase.google.com/>. Acesso em: julho. 2022. Citado na página 52.

FOWLER, M. *Building Architect*. 2003. Disponível em: <https://martinfowler.com/bliki/BuildingArchitect.html>. Acesso em: julho. 2022. Citado 3 vezes nas páginas 25, 29 e 34.

FREEMAN, A.; SANDERSON, S. *Pro ASP.NET MVC 3 framework*. 3. ed. New York, NY: APRESS, 2011. Citado na página 43.

GARCIA, J. et al. Identifying architectural bad smells. *2009 13th European Conference on Software Maintenance and Reengineering*, p. 255–258, 2009. Citado na página 29.

GERHARDT, T. E.; SILVEIRA, D. T. *Métodos de pesquisa*. Porto Alegre: Editora UFRGS, 2009. Citado 3 vezes nas páginas 59, 60 e 66.

GIL, A. C. et al. *Como elaborar projetos de pesquisa*. São Paulo: Atlas São Paulo, 2002. v. 4. Citado na página 63.

GONZALEZ, D. The biggest challenges of building a custom chat app and how to overcome them. 2019. Disponível em: <https://www.itproportal.com/features/the-biggest-challenges-of-building-a-custom-chat-app-and-how-to-overcome-them-checklist-included/>. Acesso em: agosto. 2022. Citado 2 vezes nas páginas 73 e 74.

Google I/O. Google i/o. 2021. Disponível em: <https://io.google/2021/?lng=en>. Acesso em: julho. 2022. Citado na página 52.

GRAÇA, H. Mvc and its alternatives. 2017. Disponível em: <https://herbertograca.com/2017/08/17/mvc-and-its-variants/>. Acesso em: julho. 2022. Citado 5 vezes nas páginas 29, 35, 43, 64 e 111.

GRAÇA, H. Ports adapters architecture. 2017. Disponível em: <https://herbertograca.com/2017/09/14/ports-adapters-architecture/>. Acesso em: julho. 2022. Citado 6 vezes nas páginas 43, 44, 64, 102, 103 e 104.

GRAÇA, H. Software architecture premises. 2017. Disponível em: <https://herbertograca.com/2017/07/05/software-architecture-premises/>. Acesso em: Junho. 2022. Citado 5 vezes nas páginas 36, 107, 119, 120 e 121.

GRØNLI, T.-M. et al. Mobile application platform heterogeneity: Android vs windows phone vs ios vs firefox os. In: *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*. Victoria, BC, Canada: IEEE, 2014. p. 635–641. Citado na página 26.

HAMILTON, T. What is software testing? definition. 2022. Disponível em: <https://www.guru99.com/software-testing-introduction-importance.html>. Acesso em: outubro. 2022. Citado na página 48.

IBM. What is software testing. 2022. Disponível em: <https://www.ibm.com/topics/software-testing#:~:text=Software%20testing%20is%20the%20process,development%20costs%20and%20improving%20performance.> Acesso em: outubro. 2022. Citado na página 48.

JETBRAINS. O estado do ecossistema de desenvolvedores de 2021. 2021. Disponível em: <https://www.jetbrains.com/pt-br/lp/devecosystem-2021/>. Acesso em: julho. 2022. Citado na página 40.

JUNG, J. How to test software, part i: mocking, stubbing, and contract testing. 2019. Disponível em: <https://circleci.com/blog/how-to-test-software-part-i-mocking-stubbing-and-contract-testing/>. Acesso em: julho. 2022. Citado 2 vezes nas páginas 45 e 104.

KRAFTA, L. et al. O método da pesquisa-ação: um estudo em uma empresa de coleta e análise de dados. *Revista Quanti Quali*, 2009. Disponível em: <https://posgraduacao.faccat.br/moodle/pluginfile.php/1725/mod_resource/content/0/09pesquisa_acao_2009_1.pdf>. Acesso em: agosto. 2022. Citado 3 vezes nas páginas 60, 66 e 67.

KRIGER, D. O que é teste de integração e quais são os tipos de teste? 2021. Disponível em: <https://kenzie.com.br/blog/teste-de-integracao/>. Acesso em: outubro. 2022. Citado 3 vezes nas páginas 29, 48 e 49.

KRISTENSEN, B. B.; ÖSTERBYE, K. Conceptual modeling and programming languages. *ACM Sigplan Notices*, ACM New York, NY, USA, v. 29, n. 9, p. 81–90, 1994. Citado na página 34.

KRUCHTEN, P.; OBBINK, H.; STAFFORD, J. The past, present, and future for software architecture. *IEEE Software*, Institute of Electrical and Electronics Engineers (IEEE), v. 23, n. 2, p. 22–30, 2006. Citado na página 25.

LA, H.-J.; KIM, S.-D. Balanced mvc architecture for high efficiency mobile applications. *KSII Transactions on Internet and Information Systems (TIIS)*, Korean Society for Internet Information, v. 6, n. 5, p. 1421–1444, 2012. Citado na página 41.

LARMAN, C. *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. [S.l.]: Pearson Education India, 2012. Citado na página 37.

LEDWON, P. Challenges of building a reliable realtime chat service. 2018. Disponível em: <https://www.infoq.com/articles/challenges-realtime-chat-service-pusher/>. Acesso em: agosto. 2022. Citado 2 vezes nas páginas 73 e 74.

LEE, V.; SCHNEIDER, H.; SCHELL, R. *Mobile Applications*: Architecture, design, and development. Philadelphia, PA: Prentice Hall, 2004. Citado na página 26.

MARTIN, R. C. Design principles and design patterns. *Object Mentor*, v. 1, n. 34, p. 597, 2000. Citado 2 vezes nas páginas 36 e 104.

MARTIN, R. C. *Clean architecture*: A craftsman's guide to software structure and design. 1. ed. Philadelphia, PA: Prentice Hall, 2017. Citado 10 vezes nas páginas 26, 34, 35, 36, 38, 46, 64, 113, 119 e 120.

MARTINEZ, P. Hexagonal architecture, there are always two sides to every story. 2021. Disponível em: <https://medium.com/ssense-tech/hexagonal-architecture-there-are-always-two-sides-to-every-story-bc0780ed7d9c>. Acesso em: julho. 2022. Citado 3 vezes nas páginas 29, 44 e 45.

MENDONÇA, V. R. L. de; BITTAR, T. J.; DIAS, M. de S. Um estudo dos sistemas operacionais android e ios para o desenvolvimento de aplicativos. Departamento de Ciência da Computação. Universidade Federal de Goiás, 2011. Citado na página 40.

Mia Ajuda. *Mia Ajuda*. 2020. Disponível em: <https://miaajuda.netlify.app/>. Acesso em: julho. 2022. Citado na página 26.

PALMIERI, M.; SINGH, I.; CICCHETTI, A. Comparison of cross-platform mobile development tools. In: *2012 16th International Conference on Intelligence in Next Generation Networks*. Berlin, Germany: IEEE, 2012. Citado 3 vezes nas páginas 25, 39 e 64.

PATERSKA, P. What is react native and when to use it for your app? 2021. Disponível em: <https://www.elpassion.com/blog/what-is-react-native-and-when-to-use-it>. Acesso em: agosto. 2022. Citado 2 vezes nas páginas 51 e 52.

PATRIARCHA, O. C. Arquitetura de sistemas como diferencial estratÉgico em uma empresa de desenvolvimento de software. Instituto Federal de Ciência e Tecnologia de São Paulo, Câmpus São Paulo, p. 167, 2018. Monografia, Pós-Graduação Lato Sensu em Gestão da Tecnologia da Informação. Disponível em: <https://spo.ifsp.edu.br/images/phocadownload/DOCUMENTOS_MENU_LATERAL_FIXO/POS_GRADUA%C3%87%C3%83O/ESPECIALIZA%C3%87%C3%83O/Gest%C3%A3o_da_Tecnologia_da_Informa%C3%A7%C3%A3o_____/PRODUCAO/2018/Arquitetura_de_Sistemas_como_Diferencial_Estrat%C3%A9gico_em_uma_Empresa_de_Desenvolvimento_de_Software.pdf>. Acesso em: Setembro. 2022. Citado na página 42.

POPE, K. *Zend Framework 1.8 Web Application Development*. Birmingham, England: Packt Publishing, 2010. Citado 3 vezes nas páginas 29, 43 e 112.

REACT. *Custom Hooks*. 2022. Disponível em: <https://reactjs.org/docs/hooks-custom.html>. Acesso em: Novembro. 2022. Citado na página 104.

REACT. *React Hooks*. 2022. Disponível em: <https://reactjs.org/docs/hooks-intro.html>. Acesso em: Novembro. 2022. Citado na página 104.

REENSKAUG, T. Models - views - controllers. 1979. Disponível em: <http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf>. Acesso em: julho. 2022. Citado 2 vezes nas páginas 41 e 42.

RIBEIRO, D. F. Estudo de interface humano-máquina em dispositivos móveis. Departamento de Informática e Estatística. Universidade Federal de Santa Catarina, p. 1–6, 2006. Citado na página 41.

RICHARDSON, D. J.; WOLF, A. L. Software testing at the architectural level. In: *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*. [S.l.: s.n.], 1996. p. 68–71. Citado na página 80.

SCHWABER, K.; SUTHERLAND, J. The scrum guide: The definitive guide to scrum: The rules of the game. 2020. Disponível em: <https://scrumguides.org/>. Acesso em: agosto. 2022. Citado 2 vezes nas páginas 64 e 65.

SHVETS, A. Padrões de projeto. 2022. Disponível em: <https://refactoring.guru/pt-br/design-patterns/>. Acesso em: julho. 2022. Citado 4 vezes nas páginas 36, 37, 51 e 112.

SILVA, T. H.; MACHARET, D. G.; TEIXEIRA, C. F. Análise do desempenho de algoritmos criptográficos em dispositivos móveis. *28 Congresso da SBC*, Belém, PA, p. 1–17, 2008. Anais. Citado na página 41.

Software Engineering Institute. Software architecture. 2022. Disponível em: <https://www.sei.cmu.edu/our-work/software-architecture/>. Acesso em: julho. 2022. Citado na página 25.

SonarQube. *SonarQube Documentation*. 2022. Disponível em: <https://docs.sonarqube.org/latest/>. Acesso em: julho. 2022. Citado 9 vezes nas páginas 53, 54, 80, 93, 95, 96, 98, 100 e 101.

SRIVASTAVA, S. Top 10 best cross-platform app development frameworks. 2022. Disponível em: <https://appinventiv.com/blog/cross-platform-app-frameworks/>. Acesso em: julho. 2022. Citado 2 vezes nas páginas 39 e 40.

STEVENSON, D. What is firebase? the complete story, abridged. 2018. Disponível em: <https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0>. Acesso em: julho. 2022. Citado na página 52.

TOTVS. Kanban: conceito, como funciona, vantagens e implementação. 2021. Disponível em: <https://www.totvs.com/blog/negocios/kanban/>. Acesso em: julho. 2022. Citado 2 vezes nas páginas 64 e 65.

VAILSHERY, L. S. Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021. 2022. Disponível em: <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/>. Acesso em: julho. 2022. Citado na página 40.

WENI. Por que os dispositivos móveis estão dominando o mercado? 2017. Disponível em: <https://weni.ai/blog/por-que-os-dispositivos-moveis-estao-dominando-o-mercado-chatbot/>. Acesso em: julho. 2022. Citado 3 vezes nas páginas 25, 38 e 39.

WIERUCH, R. *Babel Module Resolver with TypeScript*. 2022. Disponível em: <https://www.robinwieruch.de/babel-module-resolver-typescript/>. Acesso em: Novembro. 2022. Citado na página 106.

YANG, H. Y.; TEMPERO, E.; MELTON, H. An empirical study into use of dependency injection in java. In: *19th Australian Conference on Software Engineering (aswec 2008)*. Perth, WA, Austrália: IEEE, 2008. p. 239–247. Citado na página 37.