Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Estudo sobre o impacto do uso de padrões na qualidade de aplicações móveis

Maycon Muller da Silva Anjos

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Prof. Me. Euler Horta Marinho

Outubro, 2021
João Monlevade–MG

Maycon Muller da Silva Anjos

Estudo sobre o impacto do uso de padrões na qualidade de aplicações móveis

Orientador: Prof. Me. Euler Horta Marinho

Monografia apresentada ao curso de Sistemas de Informação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto

João Monlevade

Outubro de 2021

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

A599e Anjos, Maycon Muller da Silva .
Estudo sobre o impacto do uso de padrões na qualidade de aplicações móveis. [manuscrito] / Maycon Muller da Silva Anjos. - 2021.
71 f.: il.: color., gráf., tab..

Orientador: Prof. Me. Euler Horta Marinho.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Graduação em Sistemas de Informação .

1. Android (Recurso eletrônico). 2. Aplicativos móveis. 3. Garantia de qualidade. 4. Padrões de software. 5. Software de aplicação - Desenvolvimento. I. Marinho, Euler Horta. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004.41

Bibliotecário(a) Responsável: Flavia Reis - CRB6-2431

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTACAO E SISTEMAS

FOLHA DE APROVAÇÃO

Maycon Muller da Silva Anjos

Estudo sobre o impacto do uso de padrões na qualidade de aplicações móveis

Monografia apresentada ao Curso de Sistemas de Informação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação

Aprovada em 01 de setembro de 2021

Membros da banca

Mestre - Euler Horta Marinho - Orientador (Universidade Federal de Ouro Preto)
Doutor - Diego Zuquim Guimarães Garcia - (Universidade Federal de Ouro Preto)
Mestra Daniela Rodrigues Dias - (Doutoranda em Educação - Universidade Federal de Ouro Preto)

Euler Horta Marinho, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 13/10/2021

Documento assinado eletronicamente por **Euler Horta Marinho**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 13/10/2021, às 09:36, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0231768** e o código CRC **1167B551**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: - www.ufop.br

Este trabalho é dedicado aos meus pais Vicente e Claudia, e ao meu irmão Matheus que sempre me apoiaram e incentivaram em todas as minhas escolhas ao decorrer da minha vida e ainda forneceram todos os recursos possíveis para que eu pudesse chegar nesse momento tão especial.

Agradecimentos

Agradeço primeiramente a Deus por permitir que eu chegasse nesse momento tão importante da minha vida.

Aos meus pais Vicente e Claudia que sempre me apoiaram e incentivaram a nunca desistir dos meus objetivos e a sempre batalhar e correr atrás dos meus sonhos.

Agradeço ainda a todos os professores do curso, que contribuíram diretamente para meu crescimento pessoal e acadêmico no decorrer desses anos.

Agradeço ao meu orientador e Professor Euler, por ter me orientado de forma exemplar no desenvolvimento deste projeto, sempre abordando todos assuntos com muita propriedade e didática.

Por fim, agradeço a todos que contribuíram para meu crescimento pessoal, profissional e acadêmico que aqui não foram pontualmente citados.

"Science is more than a body of knowledge; it is a way of thinking."

— Carl Sagan (1934 – 1996),
in: The Demon-Haunted World: Science as a Candle in the Dark.

Resumo

Atualmente, existem diferentes formas de se desenvolver uma aplicação para um dispositivo móvel. Quando falamos de aplicações nativas na Plataforma Android, existe um conjunto de bibliotecas que auxilia os desenvolvedores a seguir as práticas de projeto recomendadas pela Google, o Android *Jetpack*. O presente trabalho consiste em uma análise do impacto da aplicabilidade de padrões no desenvolvimento de aplicações móveis, especificamente para a plataforma Android. Neste trabalho, foi desenvolvida uma aplicação para controle financeiro pessoal usando duas metodologias de desenvolvimento. A primeira, envolve uma versão da aplicação usando os componentes de arquitetura do *Jetpack* enquanto a segunda, sem utilizar quaisquer componentes do *Jetpack*. Com o objetivo de avaliar questões relacionadas a qualidade da aplicação, foi realizada uma análise comparativa entre as mesmas usando as métricas Chidamber e Kemerer (CK). Essa análise constatou que a aplicação desenvolvida usando o Android *Jetpack* apresentou características superiores no que tange a manutenibilidade.

Palavras-chave: Android. *Jetpack*. Métricas. Aplicações móveis.

Abstract

Currently, there are different ways to develop an mobile device application. When we talk about native applications on the Android Platform, there is a set of libraries that help developers to follow Google's recommended design practices, the Android *Jetpack*. In this work, an application for personal financial control was developed using two development methodologies. The first involves a version of the application using the *Jetpack* architecture components while the second does not use any *Jetpack* components. In order to evaluate issues related to the quality of the application, a comparative analysis was performed using the CK metrics. This analysis found that the application developed using Android *Jetpack* had superior characteristics in terms of maintainability.

Key-words: latex. abntex. text editoration.

Lista de ilustrações

Figura 1 – Padrões de Projeto . . . 20
Figura 2 – Padrão MVVM . . . 22
Figura 3 – Pilares do Android *Jetpack* . . . 23
Figura 4 – Vinculação de um componente de *layout* . . . 24
Figura 5 – Vinculação de um componente diretamente no *layout* . . . 25
Figura 6 – Estados e eventos que compõem o ciclo de vida da atividade do Android 26
Figura 7 – Criando um Lifecycle em Java . . . 26
Figura 8 – Interações do LiveData . . . 27
Figura 9 – Separação dos dados de Visualização. . . . 28
Figura 10 – Mediador Repository. . . . 29
Figura 11 – Relação do Room com os componentes . . . 31
Figura 12 – Componentes Básicos de Arquitetura do *Jetpack*. . . . 31
Figura 13 – Gráfico de utilização das dependências do Android *Jetpack* em repositórios 32
Figura 14 – Modelo de qualidade em uso. . . . 33
Figura 15 – Modelo de qualidade de produto. . . . 34
Figura 16 – Relacionamentos entre os atributos internos e externos de *software*. . . 36
Figura 17 – `Firebase Test Lab` . . . 39
Figura 18 – Tela principal da aplicação . . . 44
Figura 19 – Diagrama Entidade Relacionamento . . . 45
Figura 20 – Criação da tabela/entidade de receitas . . . 47
Figura 21 – Criação do `DAO` . . . 48
Figura 22 – Criação do RoomDatabase . . . 49
Figura 23 – Criação do *ViewModel* . . . 50
Figura 24 – Processo de inserção de uma nova Receita . . . 50
Figura 25 – Escopo dos testes . . . 51
Figura 26 – Dispositivos testados . . . 52
Figura 27 – Dicas de melhoria . . . 52
Figura 28 – Visão geral dos testes . . . 53
Figura 29 – Consumo de recursos durante o teste. . . . 53
Figura 30 – Tela de informações *Sliders* . . . 55
Figura 31 – Tela de direcionamento *Sliders* . . . 55
Figura 32 – Tela de cadastro e *login* . . . 56
Figura 33 – Tela principal Dashboard . . . 57
Figura 34 – Botão de Ação Flutuante `Floating Action Menu` . . . 58
Figura 35 – Cadastro de uma nova Receita . . . 59
Figura 36 – Cadastro de uma nova Despesa . . . 59

Figura 37 – Listar ou Remover uma Receita 60
Figura 38 – Listar ou Remover uma Despesa 61
Figura 39 – Editar/ Remover um item de lista 62
Figura 40 – Acoplamento entre classes de objetos 63
Figura 41 – Profundidade da Arvore de Herança 64
Figura 42 – Falta de Coesão em Métodos 64
Figura 43 – Número de filhos 65
Figura 44 – Resposta para uma classe 66
Figura 45 – Métodos ponderados por classe 66

Lista de tabelas

Tabela 1 – Métricas CK para cada uma das versões do aplicativo. 67

Lista de abreviaturas e siglas

Apps Aplicativos

API *Application Programming Interface*

CPU *Central Processing Unit*

CK Chidamber e Kemerer

DAO ***Data Access Objects***

ER Entidade Relacionamento

IDE *Integrated Development Environment*

IEC *International Electrotechnical Comission*

ISO *International Organization Standardization*

MVVM *Model-View-ViewModel*

NASA *National Aeronautics and Space Administration*

SGBD Sistema Gerenciador de Banco de Dados

SI Sistemas de Informação

SQL *Structured Query Language*

PC's *Personal Computers*

IU Interface de Usuário

WMC (*Weighted Methods per Class*

DIT *Depth Of Inheritance Tree*

NOC *Number of children*

CBO *Coupling Between Object Classes*

XML *Extensible Markup Language*

RFC *Response For a Class*

LCOM *Lack of Cohesion in Methods*

Sumário

1 INTRODUÇÃO . . . 16
1.1 O problema de pesquisa . . . 17
1.2 Objetivos . . . 17
1.2.1 Objetivos Específicos . . . 17
1.3 Metodologia . . . 18
1.4 Organização do trabalho . . . 18

2 REVISÃO BIBLIOGRÁFICA . . . 19
2.1 Sistemas de Informação . . . 19
2.2 Padrões de Software . . . 19
2.2.1 Padrões de Projeto . . . 20
2.2.2 Padrões Arquiteturais . . . 21
2.3 Aplicações Android . . . 22
2.3.1 Android Studio . . . 22
2.3.2 Android Jetpack . . . 23
2.3.3 Componentes de arquitetura do Android . . . 24
2.3.3.1 Data Binding Library . . . 24
2.3.3.2 Lifecycle . . . 25
2.3.3.3 LiveData . . . 26
2.3.3.4 ViewModel . . . 27
2.3.3.5 Repository . . . 29
2.3.3.6 Room . . . 29
2.3.3.7 Considerações . . . 30
2.4 Qualidade de Software . . . 32
2.4.1 ISO/IEC 25010 . . . 33
2.4.1.1 Manutenibilidade . . . 34
2.4.2 Medição e Métricas . . . 35
2.4.2.1 Métricas de produto . . . 35
2.5 Teste de software . . . 37
2.5.1 Testes Automatizados da Interface com o Usuário . . . 38
2.6 Considerações . . . 38

3 DESENVOLVIMENTO . . . 40
3.1 Levantamento e análise de requisitos . . . 40
3.1.1 Histórias de usuários . . . 41
3.1.1.1 Receitas . . . 41

3.1.1.2 Despesas . . . 42
3.1.1.3 Relatórios . . . 42
3.1.1.4 Autenticação . . . 43
3.2 Projeto da aplicação . . . 43
3.3 Modelagem de dados . . . 44
3.4 Desenvolvimento das versões da aplicação . . . 45
3.4.1 Primeira Aplicação . . . 46
3.4.2 Segunda Aplicação . . . 46
3.5 Testes . . . 51
3.6 Considerações . . . 52

4 RESULTADOS . . . 54
4.1 Apresentação da aplicação . . . 54
4.1.1 Funcionalidades da Aplicação . . . 54
4.2 Métricas / Análises . . . 62
4.3 Análises das Métricas CK . . . 62
4.3.1 Coupling Between Objects (*Coupling Between Object Classes* (CBO)) . . . 63
4.3.2 Depth of Inheritance Tree (*Depth Of Inheritance Tree* (DIT)) . . . 63
4.3.3 Lack of Cohesion of Methods (*Lack of Cohesion in Methods* (LCOM)) . . . 64
4.3.4 Number of Children (*Number of children* (NOC)) . . . 65
4.3.5 Response for Class (*Response For a Class* (RFC)) . . . 65
4.3.6 Weighted Method Complexity ((*Weighted Methods per Class* (WMC))) . . . 66
4.4 Considerações . . . 67

5 CONCLUSÃO . . . 68
5.1 Limitações e trabalhos futuros . . . 69

REFERÊNCIAS . . . 70

1 Introdução

À medida que projetar software para muitos é dito como um processo intangível e a garantia da qualidade da aplicação é fundamental, a solução deve se adequar ao máximo aos requisitos iniciais e propor mecanismos de abertura para contemplar possíveis requisitos futuros. A fim de atender essas particularidades, os projetistas de software utilizam diversos elementos descritos pela Engenharia de Software. Dentre esses elementos se encontram os padrões que tem a finalidade de resolver problemas específicos. Em conformidade com Sommerville (2011, p. 472), "os padrões de software são importantes para a garantia da qualidade, pois representam uma identificação das "melhores práticas". Durante o desenvolvimento de software, os padrões fornecem uma base sólida para a construção de software de boa qualidade".

> Os dados da Think Tank Pew Research Center [1], que estuda as principais questões de tendências e atitudes que moldam o mundo, o Brasil é o segundo país onde o mercado de aplicativos mais cresce. Atualmente, a Indonésia ocupa o primeiro lugar. Essa facilidade é uma das principais razões pelas quais o mercado de aplicativos tem ganhado tanto espaço e está se expandindo rapidamente. Um exemplo que pode ser usado para ilustrar esse sucesso são os aplicativos de finanças (DIGITAL HOUSE, 2021).

Segundo dos dados do Rank MyApp (2021) entre os anos de 2018 e 2019 houve um aumento de cinquenta por cento nas instalações e de duzentos e cinquenta por cento nas visitas aos aplicativos do segmento de finanças.

> O Google disponibilizou em 2018 o *Android Jetpack*, ele é caracterizado como uma coleção de componentes, bibliotecas, ferramentas e orientações para ajudar os desenvolvedores a criar Aplicativos (Apps) de alta qualidade com mais facilidade e com recursos que impulsionam a criação de aplicativos mais robustos de fácil manutenção e com menos códigos (GOOGLE, 2018).

Com o objetivo de evidenciar os principais aspectos inerentes à manutenibilidade de aplicações móveis que são influenciados pela utilização de padrões foi proposta uma análise comparativa entre duas aplicações do Android para controle financeiro, com o objetivo de avaliar o impacto da aplicação dos padrões arquiteturais e de projeto do Android *Jetpack* na qualidade das mesmas. Vale ressaltar que, cada uma das aplicações possui suas particularidades de desenvolvimento, uma foi concebida adotando os componentes, bibliotecas e orientações do *Android Jetpack* e outra, sem quaisquer metodologias de padronização para o seu desenvolvimento. Essa análise comparativa foi realizada através da utilização de algumas métricas de software relacionadas à manutenibilidade.

[1] <https://www.pewresearch.org/>Acesso em 27 set. 2021

1.1 O problema de pesquisa

Diante da crescente demanda pela utilização de dispositivos móveis, o desenvolvimento de aplicativos para esses dispositivos tende a acompanhar intimamente esse crescimento. Partindo do pressuposto de que, para que uma aplicação móvel se mantenha com qualidade e competitiva no mercado, ela deve ser constantemente atualizada. O estudo envolvendo a utilização de padrões de desenvolvimento, para melhorar questões relacionadas à qualidade da aplicação, no que tange a capacidade e flexibilidade da mesma em suportar futuras manutenções, se torna válido.

Segundo Sommerville (2011), a manutenção de software ocupa uma porção maior dos orçamentos de TI que o desenvolvimento. A manutenção é responsável por, aproximadamente, dois terços do orçamento, contra um terço para desenvolvimento. Ele ainda menciona que não há dúvida que o desenvolvimento de software para torná-lo mais manutenível é viável.

Com o aquecimento e competitividade do mercado de dispositivos móveis, a qualidade da aplicação se torna algo fundamental. Se o produto entregue for um produto de boa qualidade, um produto que possa incorporar novas funcionalidades de uma forma mais prática e rápida, isso virtualmente pode oferecer vantagem competitiva nesse nicho de mercado.

De acordo com Santa (2012), muitas pessoas utilizam anotações em cadernos, agendas ou em planilhas para tentar controlar seus gastos mensais para que os mesmos não ultrapassem os limites impostos do valor que se ganha mensalmente. Ela ainda menciona que existe um desafio enorme em poupar dinheiro para algumas pessoas, ou seja, controlar suas despesas e nesse contexto muitas pessoas não conseguem realizar esse controle.

Assim sendo, o desenvolvimento de um sistema financeiro móvel deve lidar com várias solicitações rotineiras de cadastro de transações, sendo uma transação caracterizada por uma receita e ou despesa. O mesmo deve ser abrangente a qualquer tipo de indivíduo que necessite de suporte para controlar seus ganhos e gastos.

1.2 Objetivos

O presente trabalho consiste em uma análise do impacto do uso de padrões na qualidade de aplicação móveis, especificamente para a plataforma *Android*.

1.2.1 Objetivos Específicos

Este trabalho possui os seguintes objetivos específicos:

- Levantar os requisitos das aplicações.

- Modelar e implementar um banco de dados para uma das aplicações.
- Desenvolver duas aplicações móveis na plataforma *Android* para gerenciamento de receitas e despesas pessoais.
- Utilizar o *Android Jetpack* em uma das aplicações.
- Criar protótipos com base nos requisitos levantados.
- Implementar testes automatizados para validação das aplicações desenvolvidas.
- Empregar métricas para equiparar a qualidade dos padrões aplicados quanto a manutenibilidade.

1.3 Metodologia

Este trabalho tem por finalidade evidenciar os principais aspectos relacionados à manutenibilidade de aplicações móveis que são influenciados pela utilização de padrões, de forma particular, adotando os principais componentes e orientações do Android *Jetpack*. Com isso, busca-se apoiar os principais envolvidos no processo de desenvolvimento e manutenção de aplicações.

Os passos para execução deste trabalho são assim definidos:

- Revisão da literatura sobre trabalhos que utilizam padrões do Android para tornar uma aplicação mais manutenível.
- Desenvolvimento de duas aplicações móveis, com e sem a utilização de padrões.
- Validação das aplicações acerca de suas funcionalidades e se as mesmas cumpriram todos os requisitos iniciais.
- Análise e discussão embasada na análise comparativa das duas aplicações desenvolvidas.
- Avaliação da manutenibilidade das aplicações através de métricas de software.

1.4 Organização do trabalho

O restante do trabalho é organizado da seguinte forma. O Capítulo 2 apresenta uma revisão bibliográfica dos principais tópicos e assuntos relacionados ao trabalho desenvolvido. O Capítulo 3 detalha os requisitos, tecnologias escolhidas, modelagem do sistema, dentre outros aspectos. Já o Capítulo 4 aborda a análise e os resultados da aplicação desenvolvida em cada uma das versões. Para finalizar o Capítulo 5 apresenta as considerações finais e propostas de trabalhos futuros.

2 Revisão bibliográfica

Este capítulo apresenta uma revisão bibliográfica acerca de conceitos, tecnologias e informações relacionadas ao desenvolvimento deste projeto.

2.1 Sistemas de Informação

Na visão de Laudon e Laudon (2011), os Sistemas de Informação (SI) podem ser tecnicamente definidos como um conjunto de componentes inter-relacionados que coletam, processam, armazenam e distribuem informações com o objetivo de dar apoio a uma organização nas atividades de tomada de decisões, coordenação e controle. No contexto deste trabalho, o objetivo do sistema de informação é servir de apoio a um usuário específico e não à uma organização como um todo.

Laudon e Laudon (2011) mencionam que *iPhones*, celulares Android, *BlackBerrys* e *tablets* de alta definição não são simplesmente itens de entretenimento, segundo ele uma parcela cada vez maior da computação empresarial está migrando de *Personal Computers* (PC's) e computadores *desktop* para os dispositivos móveis. Os autores enfatizam que, essas novas formas e ambientes de computação são comumente chamadas de "plataformas de computação e mídia emergentes".

2.2 Padrões de Software

No processo de concepção de um *software*, vários fatores devem ser levados em consideração. Por exemplo, não vale a pena resolver cada problema do projeto a partir do zero. Assim, para solucionar este problema, projetistas de software mais experientes reutilizam soluções que já funcionaram no passado. Dessa forma, essas soluções foram catalogas como padrões. Esses padrões resolvem problemas específicos de projetos e tornam os projetos orientados a objetos mais flexíveis e reutilizáveis (GAMMA, 2009).

Segundo (SOMMERVILLE, 2011), os padrões são muito importantes no processo de gerenciamento de qualidade de um software. Assim sendo, a seleção dos padrões que devem ser aplicados no processo de concepção de um software, é uma das partes mais importantes para garantir um software de qualidade. Padrões arquiteturais e de projeto podem ser empregados como ferramentas de apoio e suporte ao processo de manutenibilidade de um *software*.

Os padrões de projeto ainda colaboram para melhoria da documentação e manutenção de sistemas, pois fornecem uma especificação explícita de interações de classes e

objetos e o seu objetivo subjacente.(GAMMA, 2009)

2.2.1 Padrões de Projeto

Os padrões de projeto tornam mais fáceis a utilização de projetos e arquiteturas bem sucedidas, bem como, ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar opções que comprometam a reutilização.

Segundo Gamma (2009), um padrão tem quatro componentes principais: o **nome do padrão** que expressa sua própria essência de forma sucinta, **o problema**, onde é descrita a situação em que ele deve ser aplicado, **a solução** que descreve os elementos que compõe o padrão, seus relacionamentos, responsabilidades e colaborações e por último, mas não menos importante, **as consequências**, essas são as análises das vantagens e desvantagens da aplicação do padrão.

Quanto ao critério de finalidade dos padrões, os mesmos podem ser classificados como, padrões de criação, que se preocupam com o processo de criação de objetos, padrões estruturais, que lidam com a composição de classes ou objetos e, por fim, podem ser padrões comportamentais, que caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades. (GAMMA, 2009)

A Figura 1 ilustra os principais padrões de projeto mencionados por Gamma (2009) na sua obra, os mesmos, são classificados de acordo com sua finalidade e o seu escopo de utilização.

Figura 1 – Padrões de Projeto

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method (112)	Adapter (class) (140)	Interpreter (231) Template Method (301)
	Objeto	Abstract Factory (95) Builder (104) Prototype (121) Singleton (130)	Adapter (object) (140) Bridge (151) Composite (160) Decorator (170) Façade (179) Flyweight (187) Proxy (198)	Chain of Responsibility (212) Command (222) Iterator (244) Mediator (257) Memento (266) Observer (274) State (284) Strategy (292) Visitor (305)

Fonte: Gamma (2009, p. 26)

Um padrão de projeto pode ser implementado em qualquer linguagem de desenvolvimento, no âmbito de aplicações Android não é diferente, inúmeros padrões de projeto

podem ser utilizados, um dos mais constantemente utilizados pelo Android *Jetpack* são o *Observer*, *Mediator* e o *Singleton*, segue abaixo uma breve descrição sobre os mesmos:

- ***Singleton:*** O *Singleton* tem a função de garantir que uma classe possua apenas uma instância em toda aplicação e fornecer um ponto de acesso global para a mesma.
- ***Observer:*** O *Observer* define uma dependência de um para muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente.
- ***Mediator:*** O *Mediator* define um objeto para que encapsula a forma como um conjunto de objetos interagem. Ele possibilita o baixo acoplamento, evitando que objetos se refiram uns aos outros explicitamente.

2.2.2 Padrões Arquiteturais

Um padrão de arquitetura deve descrever uma organização de sistema bem-sucedida em sistemas anteriores. Deve incluir informações de quando o uso de um padrão é adequado, seus pontos fortes e fracos.(SOMMERVILLE, 2011)

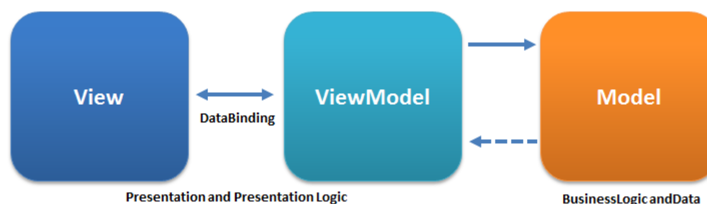
Ainda segundo Sommerville (2011), quando a manutenibilidade de *software* é um requisito crítico, a arquitetura do sistema deve ser projetada a partir de componentes autocontidos, que podem ser rapidamente alterados, ou seja, através de componentes pequenos e de baixa granularidade, com a finalidade de melhorar a manutenibilidade do artefato.

***Model-View-ViewModel* (MVVM)**
O MVVM é um padrão arquitetural que tem como objetivo separar a lógica de apresentação da lógica de negócio. Segundo Syromiatnikov e Weyns (2014), o MVVM possui três componentes com atribuições distintas:

1. ***View:*** A *View* tem a responsabilidade de renderizar a interface do usuário, ou seja, tem a função de definir o *layout* que será exibido na tela.
2. ***Model:*** O *Model* tem a função de manipular os dados.
3. ***ViewModel:*** O *ViewModel* lida com o estado de exibição e as interações do usuário, ou seja, ele atua como um intermediário entre a *View* e o *Model*, o mesmo tem a responsabilidade de manusear o *Model* para ser utilizado pela *View*. A ligação entre o *ViewModel* e a *View* e realizada através da sincronização por *observers*.

Afim de sintetizar o funcionamento do padrão, a Figura 2 demonstra como se dá a comunicação entre os componentes do MVVM.

Figura 2 – Padrão MVVM



Fonte: Wikimedia Commons (2021)

2.3 Aplicações Android

É inegável que o mercado de celulares vem crescendo cada vez mais. Hoje em dia, os usuários comuns estão buscando cada vez mais celulares com diversos recursos como câmeras, músicas, *Bluetooth*, jogos e outras funções. Para competir nesse nicho específico de mercado as empresas têm investido cada vez mais em tecnologia. O Android é a resposta do Google para ocupar esse espaço. Ele envolve uma plataforma para aplicativos móveis baseada em um sistema operacional *Linux*, uma interface visual rica, diversos aplicativos já instalados e um ambiente de desenvolvimento poderoso e flexível. (LECHETA, 2013)

Ademais, o sistema não é voltado apenas para aplicativos. Segundo Deitel, Deitel e Deitel (2015), os dispositivos Android estão presentes em robôs, motores a jato, satélites da *National Aeronautics and Space Administration* (NASA), console de jogos, geladeiras, televisores, equipamentos voltadas à saúde, relógios inteligentes, sistemas automotivos entre outros. Os autores enfatizam ainda que, o Android é uma plataforma de código aberto, o que estimula a rápida inovação, e que esse fato pode ser considerado uma vantagem para o sistema operacional, pois permite aos desenvolvedores verem o código fonte e contribuir para o Android, relatando erros e participando de grupos de discussão. Corroborando, Lecheta (2013) menciona que desenvolvedores de todo o mundo podem contribuir para o seu código fonte, adicionando funcionalidades e corrigindo falhas.

2.3.1 Android Studio

O Android Studio é o ambiente de desenvolvimento integrado do inglês *Integrated Development Environment* (IDE) oficial para o desenvolvimento de aplicativos Android. Além de apresentar um editor de código fonte, o mesmo possui inúmeras ferramentas e recursos avançados voltados para o aumento da produtividade na criação de aplicativos Android. É possível ter acesso à um emulador rápido com inúmeros recursos, um ambiente unificado que possibilita o desenvolvimento para todos dispositivos Android, bem como modelos de código e integração com o *GitHub*. (GOOGLE, 2021b)

2.3.2 Android Jetpack

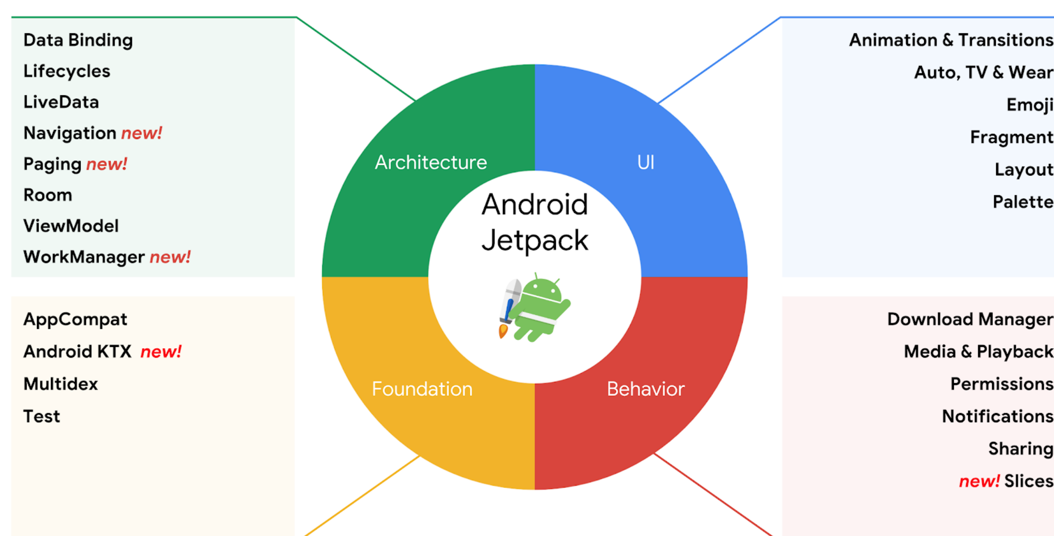
A Google disponibilizou em 2018 o Android *Jetpack*. Ele consiste em uma coleção de componentes, bibliotecas, ferramentas e orientações para ajudar os desenvolvedores a criar aplicativos de alta qualidade com mais facilidade. Os componentes do Android *Jetpack* são oferecidos em forma de bibliotecas e essas não fazem parte da plataforma subjacente do Android. Consequentemente, pode-se adotar cada componente em seu próprio ritmo, de acordo com as necessidades e no momento adequado (GOOGLE, 2018).

Além disso, Google (2018) enfatiza que os componentes do Android *Jetpack* foram criados para disponibilizar suas funcionalidades, independente da versão do Android. Dessa forma, o aplicativo desenvolvido utilizando o *Jetpack* pode ser executado em várias versões anteriores do Android.

O Android *Jetpack* foi construído com base em práticas de projeto modernas, como a separação de problemas, capacidade de testes e recursos que aumentam a produtividade e impulsionam a criação de aplicativos mais robustos, de alta qualidade, fácil manutenção e com menos código. (GOOGLE, 2018).

A Figura 3 ilustra os principais componentes do Android *Jetpack*. Eles englobam a biblioteca de suporte do Android, que inclui os componentes de arquitetura. Esses componentes foram desenvolvidos para trabalhar juntos, porém, não há obrigatoriedade de se utilizar todos de forma conjunta. Pode-se integrar as partes do Android Jetpack que resolvam um ou mais problemas do projeto e manter as partes do aplicativo que já funcionam bem.(GOOGLE, 2018)

Figura 3 – Pilares do Android *Jetpack*



Fonte: Google (2018)

2.3.3 Componentes de arquitetura do Android

O objetivo dos componentes de arquitetura é fornecer orientação sobre a arquitetura do aplicativo, com bibliotecas para tarefas comuns, como gerenciamento de ciclo de vida e persistência de dados. Os componentes de arquitetura ajudam a estruturar a aplicativo de maneira robusta, testável e sustentável com menos código clichê. As bibliotecas de componentes de arquitetura fazem parte do *Android Jetpack*.(GOOGLE, 2021b)

Nas seções subsequentes, são apresentados os principais componentes de arquitetura presentes no *Android Jetpack* que foram utilizados nesse projeto.

2.3.3.1 Data Binding Library

O `Data Binding Library` é uma biblioteca de apoio que permite vincular componentes de Interface de Usuário (IU) dos seus *layouts* a fontes de dados do aplicativo usando um formato declarativo, em vez de programático. Essa biblioteca gera automaticamente as classes necessárias para vincular as visualizações do *layout* aos seus objetos de dados.

A vinculação de componentes diretamente ao arquivo de *layout* permite remover muitas chamadas de *framework* da IU presentes nas classes Android e, dessa forma, as classes ficam mais simples e fáceis de manter. Ela também pode melhorar o desempenho do aplicativo e ajudar a evitar vazamentos de memória e exceções de ponteiro nulo, pois, não é necessário a criação de uma variável de *layout* na classe para referenciar um componente de *layout* específico.(GOOGLE, 2021b)

A Figura 4 ilustra como é feita a vinculação de um componente de *layout* sem a utilização do `Data Binding`. Neste caso, é necessário criar uma variável do tipo *TextView* dentro da classe para referenciar o componente de texto presente no arquivo de *layout*. Vale ressaltar que esse processo ocorre todas às vezes que for preciso referenciar um componente do *layout*.

Figura 4 – Vinculação de um componente de *layout*

```
TextView textView = findViewById(R.id.sample_text);
textView.setText(viewModel.getUserName());
```

Fonte: Google (2021b)

Por sua vez, a Figura 5 demonstra como é feita a vinculação do componente diretamente no arquivo de *layout*. Dessa forma, é possível acessar todos os componentes do arquivo de *layout* apenas com a criação de uma única variável do tipo *Binding*.

Figura 5 – Vinculação de um componente diretamente no *layout*

```
<TextView
        android:text="@{viewmodel.userName}" />
```

Fonte: Google (2021b)

2.3.3.2 Lifecycle

O `Lifecycle` faz parte dos componentes do *Jetpack* destinados a lidar com o reconhecimento do ciclo de vida do aplicativo. Esses tem por função a execução de determinadas ações, de acordo com uma mudança no *status* do ciclo de vida de outro componente como *Activitys* e *Fragments*. Dessa forma, eles auxiliam no processo de produzir códigos mais organizados, mais leves e mais fáceis de manter.(GOOGLE, 2021b)

Em linhas gerais, o `Lifecycle` é uma classe que contém informações sobre o estado do ciclo de vida de um componente e, quando acontece alguma alteração no ciclo de vida, seus eventos são chamados. Dessa forma, o `Lifecycle` permite que outros objetos observem o estado do componente em questão, sendo capaz de facilitar automaticamente a associação de comportamento dos componentes do sistema, diante das etapas do ciclo de vida do aplicativo.

O `Lifecycle` utiliza duas formas principais para rastrear o status do ciclo de vida do componente associado a ele, sendo eles :

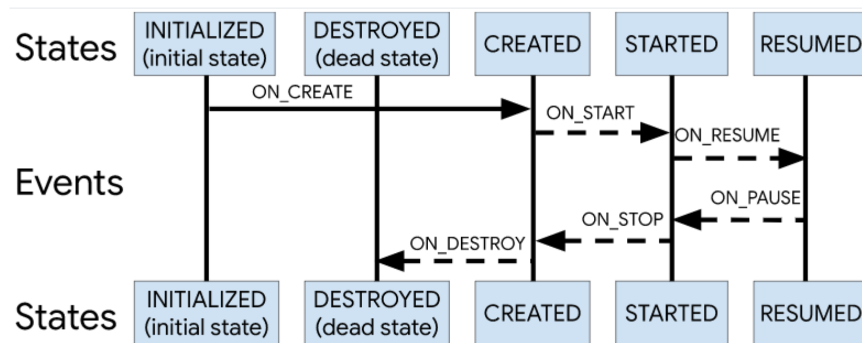
1. **Evento**: Eventos de ciclo de vida que são despachados do *framework* e da classe `Lifecycle`. Esses eventos são mapeados para os eventos de *callback* em *Activitys* e *Fragments*.

2. **Estado**: O estado atual do componente rastreado pelo objeto `Lifecycle`. A Figura 6 ilustra bem essas duas metodologias de rastreio realizadas pelo `Lifecycle`.

Via de regra, o `Lifecycle` fornece um mecanismo para que uma classe possa monitorar o status do ciclo de vida de um componente. Para realizar esse processo, a classe precisa adicionar um método chamado addObserver() da classe `Lifecycle` e, posteriormente, transmitir uma instância do seu observador. A Figura 7 ilustra o processo de mudança do comportamento do componente observado, de acordo com o ciclo de vida (ON_RESUME, ON_PAUSE) do aplicativo.

Como podemos ver na Figura 7, o objeto myLifecycleOwner implementa a interface **LifecycleOwner**. Essa é uma interface que possui um único método *getLifecycle*(). Como

Figura 6 – Estados e eventos que compõem o ciclo de vida da atividade do Android



Fonte: Google (2021b)

Figura 7 – Criando um Lifecycle em Java

```
public class MyObserver implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void connectListener() {
        ...
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void disconnectListener() {
        ...
    }
}

myLifecycleOwner.getLifecycle().addObserver(new MyObserver());
```

Fonte: Google (2021b)

a classe implementa a interface, esse método precisa ser implementado pela classe, assim indicando que a classe possui um `Lifecycle`. (GOOGLE, 2021b)

2.3.3.3 LiveData

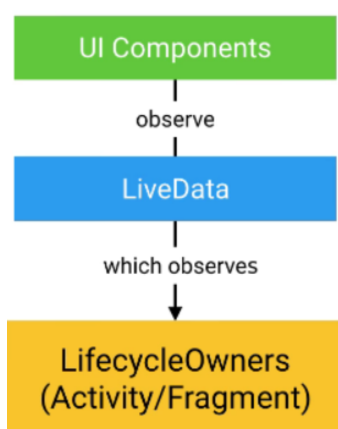
O `LiveData` é uma classe de suporte de dados que pode ser observada, ou seja, ela é uma classe armazenadora de dados que podem ser observados. Ela sempre armazena em cache a versão mais atualizada dos dados e avisa quem está observando sobre os dados quando estes são alterados. O `LiveData` não é um observador comum. Ele conta com o reconhecimento do ciclo de vida e, dessa forma, respeita o ciclo de vida de outros componentes do aplicativo.

o `LiveData` julga que um observador encontra-se em um estado ativo se o ciclo de vida do mesmo está no estado de *STARTED* ou *RESUMED*. Assim, ele transmite atualizações para os observadores ativos. Os observadores inativos, registrados para observar

objetos LiveData, não são notificados sobre as mudanças. Os componentes da IU apenas observam os dados relevantes. (GOOGLE, 2021b)

A Figura 8 ilustra essa vantagem do LiveData em relação aos observadores comuns. Ela demonstra a sequência lógica de ações, onde, os componentes de IU observam o LiveData, que, em contrapartida observa o ciclo de vida, seja de uma *Activity* ou um *Fragment* através do **LifecycleOwners**. Dessa forma, a interface é atualizada sempre que existir uma alteração de estado, em vez de ocorrer atualização toda vez que os dados da aplicação são modificados.

Figura 8 – Interações do LiveData



Fonte: Droidcon (2017)

2.3.3.4 ViewModel

O ViewModel faz parte da biblioteca de ciclo de vida do Android. Essa é uma classe que foi projetada para armazenar e gerenciar dados relacionados à IU sempre considerando o ciclo de vida da aplicativo. O ViewModel atua como um centro de comunicação entre o banco de dados do aplicativo e a IU. Sua principal função é sobreviver às mudanças de configuração.(GOOGLE, 2021b)

Se o sistema destruir ou recriar um controlador de IU, por ação de uma alteração de configuração, todos dados temporários relacionados à UI armazenados neste controlador são perdidos. Posteriormente, quando a atividade for criada a mesma terá de buscar as informações para o processamento. Essa abordagem é adequada apenas para pequenos volumes de dados, e não é recomendada para *app's* que lidam com volumes potencialmente grandes de dados. O objetivo do ViewModel é encapsular os dados de um controlador de IU para permitir que eles sobrevivam às mudanças de configuração

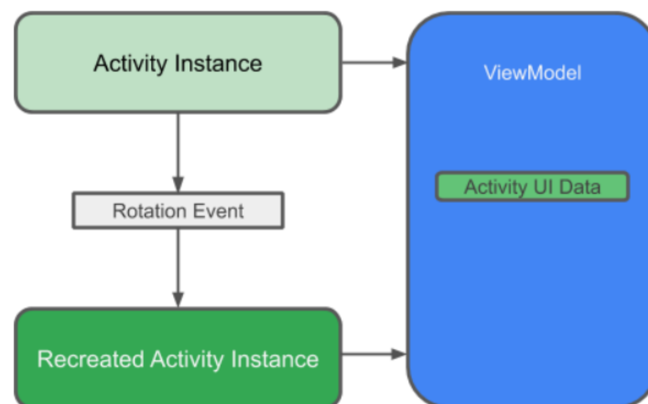
Ademais, a função dos controladores é exibir dados da IU, reagir às ações do usuário e lidar com a comunicação do sistema operacional. Exigir que os controladores sejam

responsáveis pelo carregamento dos dados de um banco acaba tornando a classe pesada, o que pode dificultar os testes e ainda, fazer com que uma classe lide sozinha com todo trabalho de um aplicativo, ao invés de delegar trabalho para outras classes. (GOOGLE, 2021b)

O `ViewModel` é uma forma de separar os dados da aplicativo e a IU implementada pelas classes *Activitys* e *Fragments*. Ele permite que a aplicativo siga o princípio de responsabilidade única, onde, os *Activityes* e *Fragments* são responsáveis por desenhar dados para a tela, enquanto o `ViewModel` tem o trabalho de manter e processar todos os dados para a IU. (GOOGLE, 2021a)

A Figura 9 retrata como a UI recorre ao `ViewModel` quando ocorre uma mudança de configuração. Neste exemplo, essa mudança é a rotação de tela. o `ViewModel` mantém as informações da IU e, quando a instância da *Activity* é recriada, ela busca os dados do `ViewModel` ao invés de recorrer ao banco propriamente dito. Isso ocorre devido ao sistema eficiente de gerenciamento de ciclo de vida, pois o **LifecycleOwner** fornece uma visão muito simples do ciclo de vida de uma *Activity* ou *Fragement* para o `ViewModel`.

Figura 9 – Separação dos dados de Visualização.



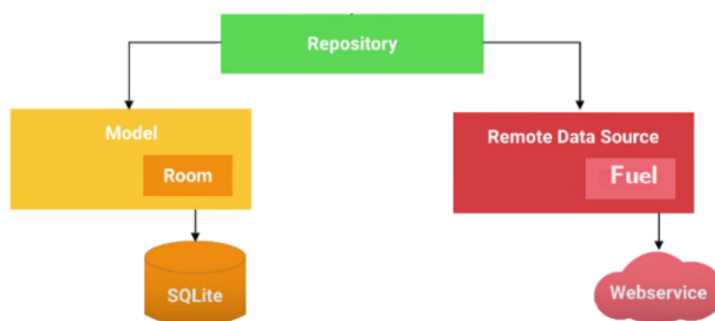
Fonte: Google (2021a)

2.3.3.5 Repository

Uma classe `Repository` abstrai o acesso a várias fontes de dados. Sua utilização é uma prática recomendada para a separação de código e arquitetura. O `Repository` manipula operações de dados e fornece uma *Application Programming Interface* (API) limpa para que o restante da aplicativo possa recuperar os dados de uma forma confiável e com facilidade. (GOOGLE, 2021b).

Segundo Google (2021a), o `Repository` implementa a lógica para decidir se deve buscar os dados da rede via *Web Services* ou usar os resultados armazenados em cache ou no banco de dados local. Via de regra, um repositório tem como objetivo servir como um intermediário entre diferentes fontes de dados. A Figura 10 ilustra o funcionamento básico de um `Repository`.

Figura 10 – Mediador Repository.



Fonte: Droidcon (2017)

O `Repository` ainda possibilita a utilização de vários *back-ends*, pois existe uma abstração das fontes de dados do restante da aplicativo, ou seja, a classe `ViewModel` responsável por requisitar os dados não sabe como os mesmos são buscados. Dessa forma, é possível fornecer ao modelo, a visualização de dados de várias implementações diferentes. (GOOGLE, 2021b)

2.3.3.6 Room

A `Room` é uma biblioteca de mapeamento de objetos que fornece persistência de dados locais com um uso mínimo de código clichê. Ela se trata de uma camada de banco de dados sobre um banco de dados *SQLite* é um mecanismo para permitir um acesso mais fluido ao banco de dados. Uma das vantagens de sua utilização é que a mesma valida cada consulta em relação ao seu esquema de dados, de forma que, as consultas em *Structured Query Language* (SQL) interrompidas resultem em erros em tempo de compilação, ao invés de falhas em tempo de execução. Outra vantagem, é o fato da `Room`

permitir observar mudanças no banco de dados, disponibilizando essas mudanças por meio de objetos `LiveData`. (GOOGLE, 2021b)

Segundo Google (2021a), por padrão, para evitar baixo desempenho da IU, a `Room` não permite a realização de consultas no *thread* principal. Quando as consultas do `Room` retornam *LiveData*, essas são executadas automaticamente de forma assíncrona em um *thread* de segundo plano.

Existem três componentes principais na `Room`:

1. **Banco de Dados:** É a base do banco de dados e serve como o principal ponto de acesso para a conexão com dados relacionais e persistentes do aplicativo. Essa classe de banco de dados `Room` deve ser abstrata e estender *RoomDatabase*. Habitualmente, uma única instância de um banco de dados `Room` é o suficiente para todo o aplicativo.

2. **Entidade:** Representa uma tabela dentro do banco de dados, ou seja, se trata dos modelos do aplicativo. Cada entidade corresponde a uma tabela que será criada no banco de dados.

3. ***Data Access Objects*** (DAO) : Sigla em inglês para **O**bjetos de **A**cesso a **D**ados, estes contém os métodos utilizados para acessar o banco de dados. A Room usa o DAO para criar uma API limpa para o código.

O DAO valida a consulta SQL em tempo de compilação e o associa a um método. Nele, são utilizadas notações mais elementares, para representar as operações de banco de dados mais comuns. Ao invés de passar todo banco de dados para o repositório, o DAO é passado para o construtor do repositório. Isso ocorre porque só precisamos acessar o DAO, pois ele contém todos os métodos de leitura / gravação do banco de dados. Não há necessidade de expor todo o banco de dados para o repositório. (GOOGLE, 2021b)

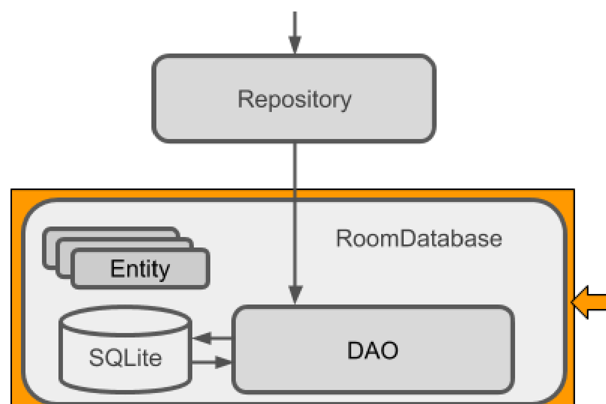
A Figura 11 ilustra como o `Room` envolve toda base de dados local, bem como acontece a interação com seus componentes internos.

2.3.3.7 Considerações

De maneira geral, esse conjunto de bibliotecas fornece os componentes básicos para criar aplicativos Android modulares, testáveis e robustos. Vale ressaltar que não é obrigatório utilizar todos os componentes, pode-se escolher o que se precisa, para solucionar um problema em questão. (DROIDCON, 2017)

A Figura 12 ilustra como se dá a interação entre alguns dos componentes citados nessa seção, desde uma interação do usuário na IU até a requisição de informações na base de dados. Vale ressaltar que cada componente depende apenas das informações fornecidas pelo componente que está no nível acima do seu. Dessa forma, essa é a sequência lógica de

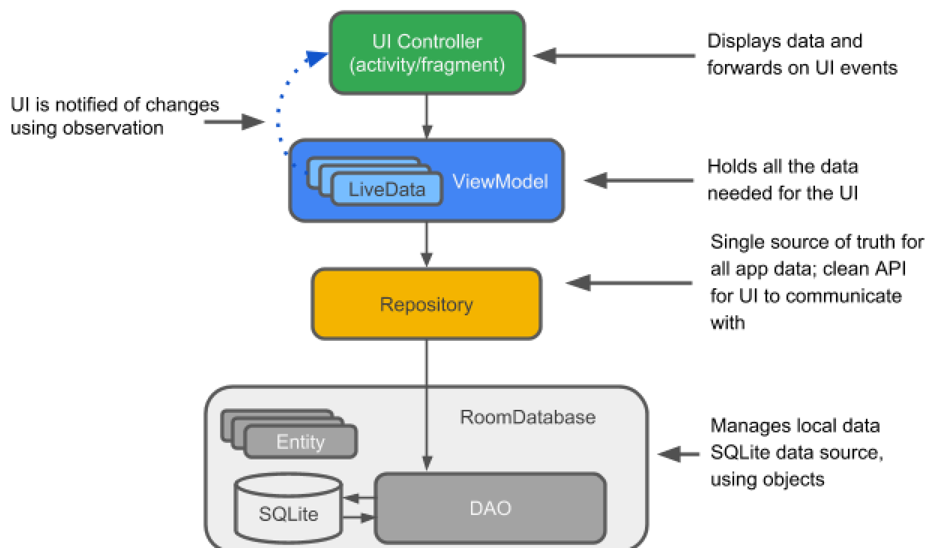
Figura 11 – Relação do Room com os componentes



Fonte: Google (2021b)

transmissão dos dados dentro de um aplicativo que utiliza esses componentes básicos de arquitetura fornecidos pelo *Android Jetpack*.

Figura 12 – Componentes Básicos de Arquitetura do *Jetpack*.

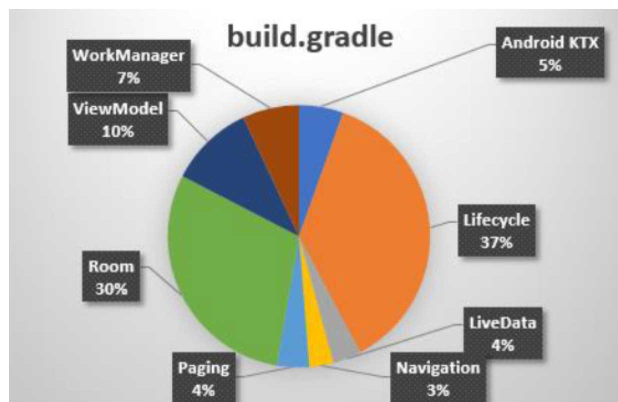


Fonte: Google (2021b)

Em 2018 foi realizada uma pesquisa por OLIVEIRA (2018) com o intuito de mensurar o uso do Android Jetpack no desenvolvimento de aplicativos Android de código aberto. Essa pesquisa teve como base a mineração de dados de 1606 repositórios do **`GitHub`** para realizar a análise da utilização do *Jetpack*. A Figura 13 ilustra como estava o cenário de utilização de alguns componentes do *Jetpack* na época de publicação da pesquisa. Segundo o autor, esses dados demonstram que a utilização do Android *Jetpack*

ainda estava baixa, uma vez que, do montante de 1606 repositórios analisados, 1504 não incorporaram quaisquer dependências do *Jetpack*.

Figura 13 – Gráfico de utilização das dependências do Android *Jetpack* em repositórios



Fonte: OLIVEIRA (2018)

A utilização dos componentes do *Jetpack*, fornece uma estratégia para diminuição da modularidade do aplicativo, ou seja, uma maior clareza da interdependência entre os componentes e da separação de interesses. Assim, a compreensibilidade da aplicativo pode ser aprimorada, bem como sua manutenibilidade. Segundo Pressman e Maxim (2016), a modularidade é o único atributo de *software* que possibilita que o programa seja intelectualmente gerenciável.

2.4 Qualidade de Software

No desenvolvimento de qualquer aplicação a qualidade é um fator determinante a ser levado em consideração. Em conformidade com Sommerville (2011), a qualidade de *software* é subjetiva e não é diretamente comparável à qualidade na indústria de manufatura. A qualidade de *software* não implica apenas se a funcionalidade do *software* foi corretamente implementada, mas também depende dos atributos não funcionais do sistema.

É impossível medir determinadas características de qualidade diretamente (por exemplo, manutenibilidade). Portando, a qualidade de um *software* não requer apenas que a funcionalidade de *software* seja corretamente implementada, ela também depende da forma como os requisitos não funcionais do sistema foram desenvolvidos. Dessa forma, a qualidade das aplicações pode ser avaliada de várias perspectivas. (SOMMERVILLE, 2011)

Corroborando, Pressman e Maxim (2016) enfatizam que a qualidade de *software* pode ser definida como uma gestão de qualidade efetiva aplicada de modo a criar um

produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam.

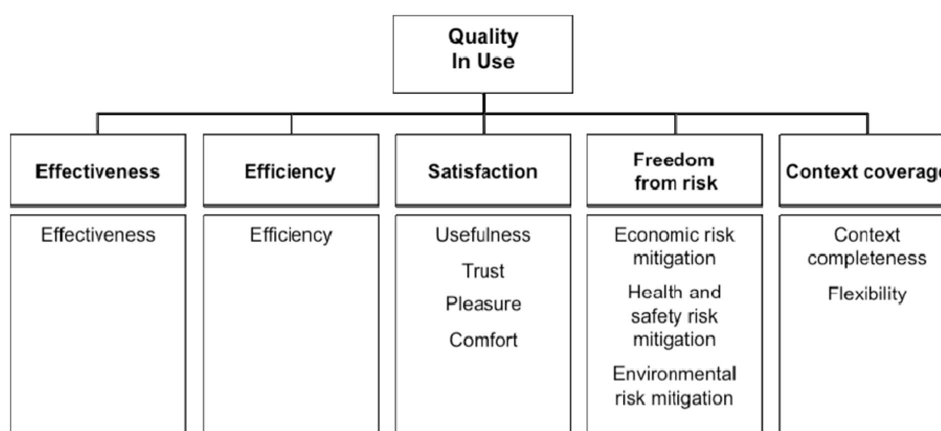
2.4.1 ISO/IEC 25010

As organizações normativas de influência internacional *International Organization Standardization* (ISO) [1] e a *International Electrotechnical Comission* (IEC) [2] disponibilizaram em 2011 a ISO/ IEC 25010. Ela é uma norma para definição de modelos para a avaliação da qualidade de *software*. Um dos principais problemas da Engenharia de *Software*, é a dificuldade de se medir a qualidade. O modelo de qualidade proposto pela ISO/IEC 25010 determina quais são as principais características de qualidade de *software* a serem levadas em consideração no processo de avaliação das propriedades de um produto de *software*. (ISO/IEC, 2010)

Segundo a ISO/IEC (2010), "A totalidade de características de um produto de *software* que lhe confere a capacidade de satisfazer necessidades explícitas e implícitas"

Modelo de Qualidade em uso:
O modelo de qualidade em uso se caracteriza por contar cinco características que confrontam os resultados da interação quando um produto é usado em um determinado contexto, onde, cada característica pode ser atribuída a diferentes atividades das partes interessadas, por exemplo, a interação de um operador ou a manutenção de um desenvolvedor. A Figura 14 define essas características relatadas no modelo de qualidade em uso.

Figura 14 – Modelo de qualidade em uso.



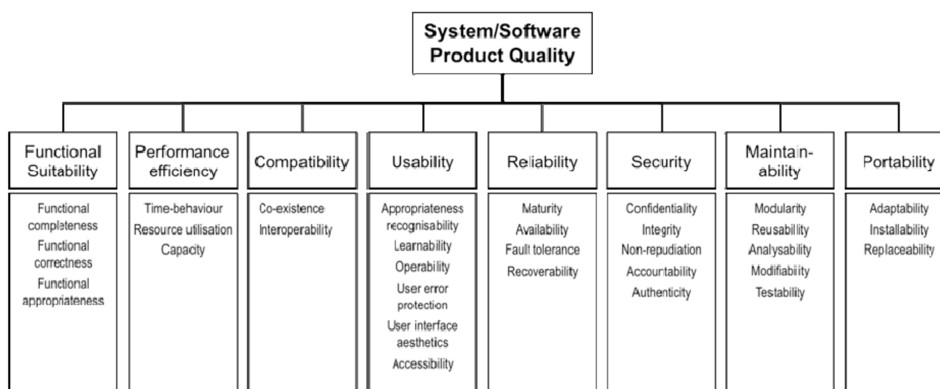
Fonte: ISO/IEC (2010)

[1] ISO é uma organização não-governamental, estabelecida em 1947, e que coordena o trabalho de órgãos de 127 países membros para promover a padronização de normas técnicas em âmbito mundial
[2] A IEC, fundada em 1906, conta com a participação de mais de 50 países e publica normas internacionais relacionadas com eletricidade, eletrônica e áreas relacionadas

Modelo de qualidade de produto:

O modelo de qualidade do produto define oito características para categorizar as propriedades de qualidade do produto de *software*. A Figura 15 demonstra as oito características da qualidade do produto de *software* que podem ser analisadas no processo de avaliação da qualidade.

Figura 15 – Modelo de qualidade de produto.



Fonte: ISO/IEC (2010)

O modelo de qualidade do produto se concentra no sistema de computador de destino que inclui o produto de *software* de destino, e o modelo de qualidade em uso se concentra em todo o sistema humano-computador que inclui o computador de destino, sistema e produto de *software* de destino. (ISO/IEC, 2010)

2.4.1.1 Manutenibilidade

A manutenibilidade é uma dentre as oito características da qualidade do produto descritas pela Norma ISO 25010 ISO/IEC (2010). Ela é definida como a capacidade (ou facilidade) do software ser modificado, agregando ao seu funcionamento melhorias, extensões de funcionalidades ou correções de erros e defeitos. A partir da manutenibilidade, ainda é possível vislumbrarmos algumas subcaracterísticas:

- Testabilidade: A testabilidade é dita como a capacidade de se testar o que foi modificado.
- Conformidade: Representa a relação dos requisitos funcionais e de desempenho que foram explicitamente declarados, a padrões de desenvolvimento claramente documentados. Essa subcaracterística ainda reflete a capacidade do produto de software de estar de acordo com normas, convenções e regulamentações relacionadas a manutenibilidade.

- Modificabilidade: Caracteriza-se pela facilidade com que o comportamento do software pode ser modificado.
- Analisabilidade: Demonstra a facilidade em se diagnosticar eventuais problemas e identificar as causas das deficiências ou falhas no software.

2.4.2 Medição e Métricas

A medição de *software* é uma metodologia utilizada para se quantificar alguns atributos de um produto ou processo de *software*. Em conformidade com Sommerville (2011), a medição preocupa-se com a derivação de um valor numérico ou perfil de um componente de *software*, sistema ou processo. Comparando esses valores entre si, juntamente com padrões aplicados em toda organização, é possível ser capaz de identificar conclusões sobre a qualidade do *software* ou avaliar a eficácia dos métodos de *software* utilizados.

A principal finalidade da medição é a possibilidade de usar a mesma para fazer julgamentos sobre a qualidade do *software*. Usando a medição de *software*, um sistema poderia, idealmente, ser avaliado usando uma variedade de métricas e, a partir dessa medição, deduzir um valor para a qualidade do sistema. Se o *software* atingir o limiar de qualidade requerido, então ele poderia ser aprovado sem revisão. (SOMMERVILLE, 2011)

Uma métrica de software é uma característica de um sistema de software, documentação de sistema ou processo de desenvolvimento que pode ser objetivamente medido. Alguns exemplos de métricas são: o tamanho de um produto em linhas de código, o número de defeitos relatados em um produto de software entregue, e o número de pessoas/dia requerido para desenvolver um componente de sistema. (SOMMERVILLE, 2011)

2.4.2.1 Métricas de produto

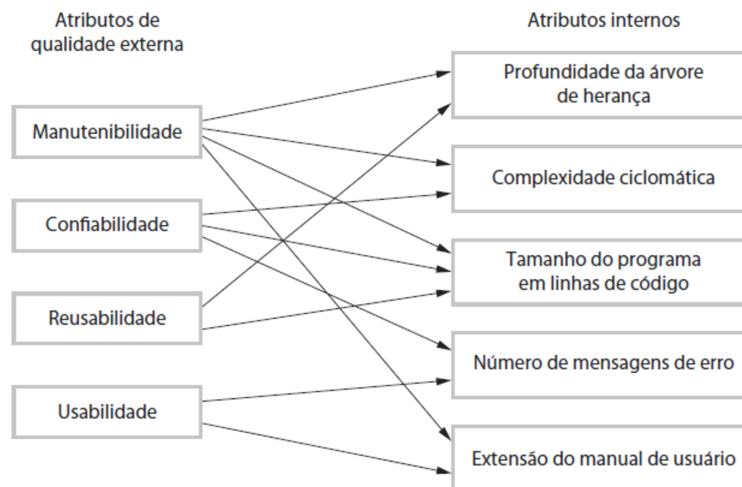
As métricas de produto, fazem parte de um grupo de métricas de previsão, este grupo ajuda a prever as características do *software*, ou seja, elas se preocupam com o *software* em si. Segundo Sommerville (2011), as métricas de produto são utilizadas para medir atributos internos de um sistema de *software*, ele ainda menciona que esse grupo de métricas, é o ideal para realizar um julgamento sobre características de qualidade como manutenibilidade, compreensibilidade e usabilidade que, por serem atributos de qualidade externa relacionados aos desenvolvedores e usuários, são fatores subjetivos, que não podem ser obviamente medidos.

A Figura 17 apresenta o relacionamento existente entre atributos de qualidade externa e os atributos internos de um *software*.

Ademais, características de software, como tamanho e complexidade ciclomática, não têm uma relação clara e consistente com atributos de qualidade tais como capacidade de compreensão e manutenibilidade. Os relacionamentos variam de acordo com os processos

de desenvolvimento e tecnologia usada, bem como o tipo de sistema que está sendo desenvolvido. (SOMMERVILLE, 2011)

Figura 16 – Relacionamentos entre os atributos internos e externos de *software*.



Fonte: Sommerville (2011, p. 469)

As métricas de produto se dividem em duas categorias:

1. **Métricas dinâmicas:** Essas são coletadas durante execução de um programa, elas ajudam a avaliar a eficiência e a confiabilidade de um programa, por exemplo, o tempo gasto para conclusão de uma computação.

2. **Métricas estáticas:** Ajudam a mensurar a complexidade e a facilidade de compreensão e manutenção de um sistema, por exemplo, o tamanho de código.

As métricas de Chidamber e Kemerer (1994), também chamadas de *suite* CK, abrangem seis métricas orientadas a objetos, pertencentes à classe de métricas estáticas. Segundo Sommerville (2011, p. 469), essas ainda são as métricas orientadas a objetos mais amplamente usadas.

A escolha das métricas CK para avaliação da manutenibilidade das aplicações foi determinada pelo fato de que estas métricas já estão consolidadas e fornecem uma avaliação sobre diversas características internas conhecidas, tais como coesão, acoplamento e complexidade da aplicação.

Segundo Sommerville (2011), as seis métricas de Chidamber e Kemerer (1994) são classificas como:

- **Métodos ponderados por classe** (WMC)
  É o número de métodos em cada classe, ponderados pela complexidade de cada

método. Portanto, um método simples pode ter uma complexidade de 1 e um método grande e complexo pode ter um valor muito superior. Quanto maior o valor para essa métrica, mais complexa a classe de objeto.

- **Profundidade da árvore de herança** (DIT)
  Representa o número de níveis discretos na árvore de herança em que as subclasses herdam atributos e operações (métodos) de superclasses. Quanto mais profunda a árvore de herança, mais complexo o projeto.

- **Número de filhos** (NOC)
  É uma medida do número de subclasses imediatas em uma classe. Ele mede a largura de uma hierarquia de classe, considerando que DIT mede sua profundidade. Um valor alto para NOC pode indicar um maior reúso.

- **Acoplamento entre classes de objetos** (CBO)
  Classes são acopladas quando métodos em uma classe usam métodos ou variáveis de instância definidas em uma classe diferente. CBO é uma medida de quanto acoplamento existe. Um valor alto para o CBO significa que as classes são altamente dependentes e, portanto, é mais provável que a mudança em uma classe afete outras classes do programa.

- **Resposta para uma classe** (RFC)
  RFC é a medida do número de métodos que poderiam ser executados em resposta a uma mensagem recebida por um objeto dessa classe. Mais uma vez, RFC está relacionada com a complexidade. Quanto maior o valor de RFC, mais complexa é a classe e, portanto, mais provável que inclua erros.

- **Falta de coesão em métodos** (LCOM)
  LCOM é calculada considerando os pares de métodos em uma classe. LCOM é a diferença entre o número de pares de métodos sem atributos compartilhados e o número de pares de métodos com atributos compartilhados.

A interpretação adequada dos valores das métricas é essencial para caracterizar, avaliar e melhorar o *design* de sistemas de *software*. Sem conhecer um limiar para uma determinada métrica, a comunidade de *software* não será capaz de aplicar métricas de *software* de uma forma efetiva (FERREIRA et al., 2012).

2.5 Teste de software

De acordo com Valente (2020), o *software* é uma das construções humanas mais complexas que existem. Assim sendo, os sistemas de *software* estão sujeitos aos mais variados tipos de erros e inconsistências. A incorporação de atividades de teste em projetos

de desenvolvimento de *software* é indispensável para que tais erros não cheguem aos usuários finais e causem prejuízos de valor incalculável.

O teste de *software* tem a finalidade de mostrar que o programa faz o que é proposto e para descobrir os defeitos do programa antes do uso. De forma geral, o processo de teste tem por finalidade alcançar dois objetivos principais. O primeiro objetivo, é garantir a validação do sistema, ou seja, demonstrar ao desenvolvedor e ao cliente que o software atende aos seus requisitos. Em contrapartida, o segundo caso preocupa-se em descobrir situações em que o software se comporta de maneira incorreta, indesejável ou de forma diferente das especificações. (SOMMERVILLE, 2011)

2.5.1 Testes Automatizados da Interface com o Usuário

De acordo com Sommerville (2011), em teste automatizados, os testes são codificados em um programa que é executado cada vez que o sistema em desenvolvimento é testado. Essa forma é geralmente mais rápida que o teste manual.

Os testes de *Interface com o Usuário* IU tem como objetivo testar todas as entradas possíveis para as interfaces do sistema, e assim, verificar se o funcionamento do sistema é o esperado para uma dada entrada. Via de regra, a realização desse teste de forma manual demandaria muito tempo e atenção por parte dos testador. Segundo Mainkar (2017), a abordagem de testes automatizados de IU garante que os testes sejam mais confiáveis e eficientes.

A Google disponibiliza uma ferramenta para execução de testes automatizados de IU, o *Firebase Test Lab*[3]. Ela fornece uma infraestrutura baseada em nuvem para testar aplicativos Android, permitindo que o aplicativo seja testado em diferentes dispositivos e configurações. Para se ter uma ideia melhor de como o aplicativo funcionará nas mãos de um usuário real, como resultado, a plataforma exibe relatórios de teste detalhados incluindo registros, capturas de tela e vídeos para corrigir qualquer problema e recuperar a estabilidade do aplicativo. (FIREBASE, GOOGLE, 2021)

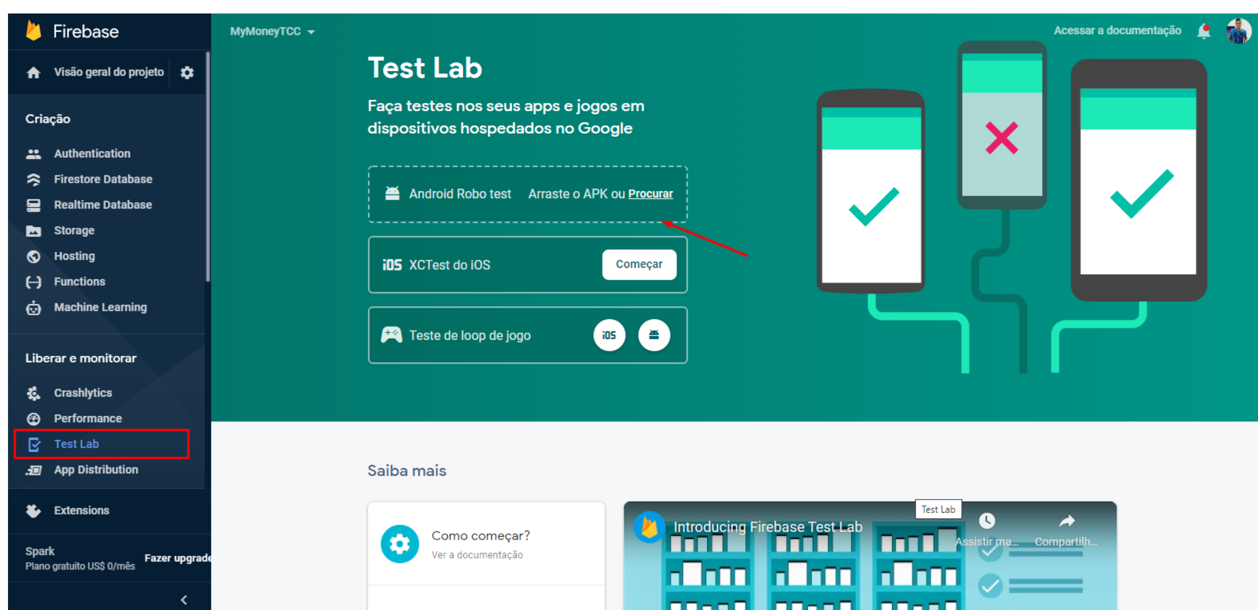
2.6 Considerações

Neste capítulo foram apresentados conceitos relacionados ao desenvolvimento deste trabalho, como a fundamentação do Android, plataforma utilizada no estudo, conceitos e características do Android *Jetpack*, uma revisão sobre as tecnologias relacionadas ao desenvolvimento, apresentação dos padrões correlatos, dentre outros conceitos.

No Capítulo 3 serão descritas as etapas do desenvolvimento, bem como apresentação dos requisitos e escolhas das tecnologias.

[3] <https://firebase.google.com/docs/test-lab> Acesso em: 27 de abril de 2021.

Figura 17 – Firebase Test Lab



Fonte: Firebase, Google (2021)

3 Desenvolvimento

Este capítulo apresenta as etapas realizadas durante o processo de desenvolvimento deste trabalho, explicitando abordagens e tecnologias utilizadas para especificação, projeto e desenvolvimento das funcionalidades.

3.1 Levantamento e análise de requisitos

No processo de levantamento de requisitos, foi realizada uma análise dos aplicativos já existentes que lidam com controle financeiro pessoal. Para cada um dos aplicativos analisados, foi levantada a possibilidade de melhoria em determinado aspecto ou funcionalidade. Com isso, espera-se que o aplicativo concebido englobe as funcionalidades principais de um gerenciador financeiro pessoal e possua algumas funcionalidades adicionais que possam ser observadas como diferenciais. Segue abaixo algumas das questões levantadas nesse processo:

- Qual tipo de usuário vai utilizar o aplicativo?
- Qual é a função principal de um gerenciador financeiro pessoal?
- Quais tipos de análises devem ser realizadas pelo aplicativo?
- Como estimular o usuário a poupar dinheiro?

O processo de documentação dos requisitos foi elaborado seguindo o padrão de histórias de usuários. Uma história de usuário deve descrever uma funcionalidade do sistema que será valiosa para um usuário ou comprador. Ademais, uma boa história de usuário deve ser independente, estimável, negociável, pequena, testável e possuir valor para os usuários ou clientes (COHN, 2004).

Segundo Cohn (2004), uma história de usuário possui uma sintaxe básica para refletir a real finalidade da funcionalidade do sistema. Um exemplo de história de usuário é demonstrado à seguir:

- **Como um...** (Quem) (Ator)
- **Eu quero...** (O que) (Funcionalidade Requisitada)
- **De modo que...** (Por que) (Valor do negócio ou benefício a ser obtido)

Os requisitos levantados nessa etapa do projeto foram documentados utilizando o *SprintGround*, [1]. Este se trata de uma aplicação para que os desenvolvedores de *software* possam organizar o trabalho e acompanhar as etapas do desenvolvimento.

O montante de requisitos levantados nessa etapa são apresentados no formato de histórias de usuários na Subseção 3.1.1. Ademais, segue abaixo um dos requisitos levantados relacionados a uma despesa:

- **Como um** usuário do aplicativo,
- **Eu quero** cadastrar uma despesa no aplicativo,
- **De modo que** eu possa selecionar o valor da despesa bem como sua data e sua categoria.

3.1.1 Histórias de usuários

Segue abaixo as historias de usuário geradas durante o levantamento de requisitos da aplicação de controle financeiro.

3.1.1.1 Receitas

Cadastrar Receita:
Como um usuário do aplicativo,
Eu quero cadastrar uma receita no aplicativo,
De modo que eu possa selecionar o valor da receita bem como sua data e sua categoria.

Listar Receitas:
Como um usuário do aplicativo,
Eu quero poder visualizar todas as receitas por mim cadastradas no aplicativo,
De modo que tenha uma lista de todas receitas cadastradas no aplicativo bem como seja possível visualizar nessa lista, os principais atributos cadastrados em uma determinada receita;

Editar Receitas:
Como um usuário do aplicativo,
Eu quero editar os dados de uma receita em questão ,
De modo que possa alterar todos os dados de uma receita por mim cadastrada para corrigir possíveis falhas ou erros ocorridos no processo de criação da mesma;

[1] <https://www.sprintground.com/>Acesso em: 10 de Mar. 2021.

Deletar Receitas:
Como um usuário do aplicativo,
Eu quero remover todos os dados de uma receita em questão,
De modo que essa receita não apareça mais em minha lista de receitas bem como seu valor não seja aplicado em meu balanço mensal;

3.1.1.2 Despesas

Cadastrar uma Despesa:
Como um usuário do aplicativo,
Eu quero cadastrar uma despesa no aplicativo,
De modo que eu possa selecionar o valor da despesa bem como sua data e sua categoria;

Listar Despesas:
Como um usuário do aplicativo,
Eu quero poder visualizar todas as despesas por mim cadastradas no aplicativo,
De modo que tenha uma lista de todas despesas cadastradas no aplicativo bem como seja possível visualizar nessa lista, os principais atributos cadastrados em uma determinada despesa;

Editar Despesas:
Como um usuário do aplicativo,
Eu quero editar os dados de uma despesa em questão,
De modo que possa alterar todos os dados de uma despesa por mim cadastrada para corrigir possíveis falhas ou erros ocorridos no processo de criação da mesma;

Deletar Despesas:
Como um usuário do aplicativo,
Eu quero remover todos os dados de uma despesa em questão,
De modo que essa despesa não apareça mais em minha lista de despesas bem como seu valor não seja aplicado em meu balanço;

3.1.1.3 Relatórios

Painel Principal *Dashboard*:
Como um usuário do aplicativo,
Eu quero poder visualizar a relação entre o meu total de receitas e o total de despesas cadastradas no aplicativo,
De modo que eu possa mensurar o valor monetário que ainda se encontra disponível para uso;

Painel Principal *Dashboard*:
Como um usuário do aplicativo,
Eu quero poder visualizar a de forma gráfica quais são as despesas que mais consomem valor monetário,
De modo que eu possa analisar onde (em qual categoria) estou gastando mais dinheiro dentro de uma determinado mês;

Dados Monetários:
Como um usuário do aplicativo,
Eu quero poder visualizar a cotação atualizada de moedas,
De modo que eu possa visualizar em tempo real os valores das principais moedas internacionais.

3.1.1.4 Autenticação

Privacidade dos dados:
Como um usuário do aplicativo,
Eu quero poder realizar todo processo de autenticação no sistema,
De modo que eu precise digitar todos meus dados cadastrais no aplicativo após ter realizado um *logoff* e assim, minhas informações sejam cobertas por uma camada de segurança;

3.2 Projeto da aplicação

No processo de desenvolvimento da aplicação, foram desenvolvidos protótipos de tela, para auxílio no processo de validação dos requisitos inicias. Segundo Sommerville (2011), os protótipos permitem aos usuários ver quão bem o sistema dá suporte a seu trabalho. Os usuários ainda podem obter novas ideias de requisitos e encontrar pontos fortes e fracos do software.

Vale a pena ressaltar que todo processo de prototipação da aplicação foi realizada usando o Adobe XD.[2] O Adobe XD se trata de uma ferramenta de design de experiência do usuário onde é possível criar vários protótipos de tela de forma responsiva, bem como, é possível visualizar todas a telas do aplicativo ao mesmo tempo e simular como os elementos visuais se comportarão quando o usuário tocar na tela.

A figura Figura 18 ilustra um protótipo da tela principal da aplicação desenvolvida.

[2] <https://www.adobe.com/br/products/xd.html> Acesso em: 30 Abr 2021.

Figura 18 – Tela principal da aplicação

Fonte: Elaborado pelo autor (2021)

3.3 Modelagem de dados

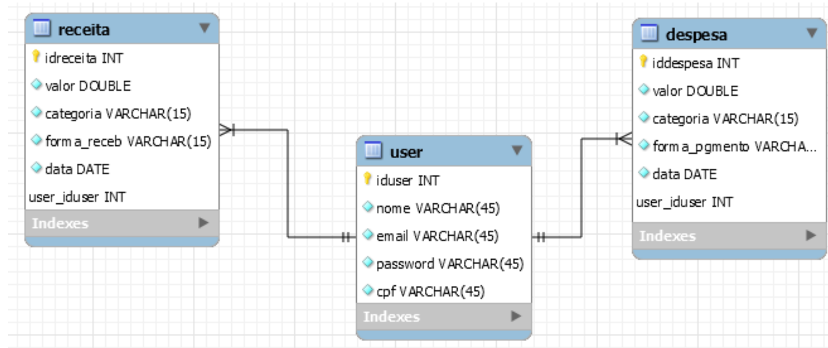
A implementação do banco de dados local foi realizada usando o Sistema Gerenciador de Banco de Dados (SGBD) `SQLite`[3]. O mesmo foi escolhido devido à sua facilidade de integração com a plataforma Android e ao conhecimento prévio da execução de tal ferramenta.

Vale ressaltar que, na aplicação que utiliza os componentes do `Jetpack`, todo processo foi realizado usando a `Room` que fornece uma camada de abstração sobre o `SQLite`. Ele funciona como um intermediário entre a aplicação e a execução do SQL puro. Em contrapartida, a outra versão da aplicação não contou com um sistema de banco de dados específico. Na mesma, foi usada a serialização de arquivos para persistência dos dados.

Para criar o diagrama Entidade Relacionamento (ER) foi utilizado o MySQL Workbench, [4]. O diagrama é apresentado na Figura 19.

[3] <https://www.sqlite.org/index.html> Acesso em: 15 Jun. 2021.
[4] <https://www.mysql.com/products/workbench/> Acesso em: 15 Jun. 2021.

Figura 19 – Diagrama Entidade Relacionamento



Fonte: Elaborado pelo autor (2021)

3.4 Desenvolvimento das versões da aplicação

Foram desenvolvidas duas versões do aplicativo de controle financeiro pessoal. Para cada uma das aplicações, uma metodologia de desenvolvimento foi utilizada. A primeira versão da aplicação não seguiu um padrão arquitetural específico, apenas padrões de projetos já fornecidos pela plataforma do Android. Abaixo temos alguns padrões de projeto fornecidos pelo Android que foram usados nas aplicações.

1. *Builder*: O *Builder* foi amplamente utilizado nas duas aplicações através do componente do Android denominado *Builder*. Este simplifica a criação de objetos da mesma classe com propriedades diferentes. Um exemplo de utilização do padrão *Builder* no Android é o `AlertDialog.Builder`. Essa se trata de uma classe de alertas do Android.

2. *Singleton*: O *Singleton* foi utilizado na segunda aplicação para criar uma única instância do banco de dados local. Como o Android trabalha gerenciando várias *threads*, o padrão só possui realmente uma efetividade quando é aplicada a palavra reservada ***synchronized***. A mesma tem a função de permitir que apenas um processo por vez acesse o trecho de código. A Figura 22, mostra esse processo. O *Singleton* também é utilizado no componente `ViewHolder`. Esse é o objeto que faz referência a cada item de *layout* que representa as listas de receitas e despesas do aplicativo.

3. *Adapter*: O *Adapter* foi amplamente utilizado nas duas aplicações para criar listas interativas de objetos (receitas e despesas), utilizando o componente `RecyclerView.Adapter`.

4. *Abstract Factory*: O *Abstract Factory* é usado pelo componente `Intent`. Este é utilizado para criar objetos que realizam a interação entre as telas do aplicativo.

3.4.1 Primeira Aplicação

Como mencionado anteriormente, a primeira versão da aplicação não seguiu um padrão arquitetural específico, bem como, a persistência dos dados foi baseada na serialização de objetos. Esse processo consiste em converter os valores de uma instância em uma sequência de *streams* de *bytes*, ou seja, é um processo que permite transformar o estado de um objeto em uma sequência de *bytes*. Assim, depois que o objeto for serializado ele pode ser persistido em um arquivo de dados.

3.4.2 Segunda Aplicação

A segunda aplicação seguiu o padrão arquitetural MVVM. Este se trata de um padrão de arquitetura recomendado pela Google para trabalhar com os componentes de arquitetura do Android *Jetpack*. Dessa forma, o desenvolvimento dessa aplicação foi baseado no Room Android, disponibilizado como um *codelab* pela Google.[5] Assim sendo, essa aplicação utilizou o padrão MVVM combinado aos principais componentes do *Jetpack*, desde a persistência dos dados até a interação do usuário com a *view*.

Os componentes do MVVM utilizados na aplicação consistem em :

- *Model*: Esse componente da arquitetura é responsável pela manipulação dos dados que são utilizados no aplicativo, sejam dados locais ou em repositórios externos.
- *View*: A *view* é responsável por renderizar as informações para o usuário. Para isso, utiliza-se *activities*, *fragments*, arquivos no formato *Extensible Markup Language* (XML) e *dialogs*.
- *ViewModel*: O *ViewModel* é utilizado para sempre manter o estado da *view* atualizada e condizente com os dados do *model*, diante das interações do usuário com a *view*.

Etapas do desenvolvimento da segunda versão do aplicativo.

1. O primeiro passo na criação do aplicativo, foi o processo de importação de todas as dependências relacionadas aos componentes do `Android Jetpack`, dependências relacionadas aos componentes da `Room`, `Lifecycle`, IU, dentre outros.
2. Posteriormente, foram criadas as entidades do aplicativo. As entidades são as tabelas no banco de dados. Elas são identificadas pela assinatura `@Entity` no início da classe. Algumas anotações podem ser sinalizadas dentro da classe tais como `@PrimaryKey`, que indica qual argumento é a chave primária da tabela e `@NonNull`, que indica que o argumento em questão não pode ser nulo. Ademais, vale ressaltar que cada campo

[5] <https://developer.android.com/codelabs/android-room-with-a-view/?authuser=2#0> Acesso em: 15 Jun. 2021.

armazenado no banco precisa ser público ou possuir um método *getter*. A Figura 20 expõe um trecho da definição da entidade de Receitas do aplicativo.

Figura 20 – Criação da tabela/entidade de receitas

```
@Entity(tableName = "receita_table")  //Cada @Entityclasse representa uma tabela SQLite
public class Receita {

    @PrimaryKey(autoGenerate = true) //id criada automaticamente
    @NonNull //não pode ser nula
    @ColumnInfo(name = "id") //nome da coluna
    private  int id;

    @NonNull
    private  double  valor;

    @NonNull
    private  String data;

    @NonNull
    private  String categoria;

    private   String descricao;

    private   String tipoRecebimento;
```

Fonte: Elaborado pelo autor (2021)

3. Dando prosseguimento ao desenvolvimento, foi criado o DAO para validar o código SQL em tempo de compilação, associando o código SQL a um método da interface. Assim sendo, todas as consultas realizadas no banco tem que ser anotadas nessa classe. A Figura 21 ilustra um trecho do DAO criado para ilustrar esse processo. A assinatura *onConflict = OnConflictStrategy.IGNORE* no método `@Insert` indica que uma nova receita deve ser ignorada se ela for exatamente igual a outra já existente na base de dados.
   Como podemos notar na Figura 21, o método `getAlphabetizedReceitas()` retorna uma lista de objetos `LiveData`. Essa é a classe que nos permite observar as alterações nos dados e, de acordo com uma mudança, atualizar a IU. Todo esse processo de atualização é realizado pela `Room`. Ela gera todo o código necessário para atualizar o `LiveData` quando o banco de dados é atualizado.

4. Posteriormente, foi criado o banco de dados propriamente dito, o componente do `Jetpack` denominado `Room`. Ele usa o DAO para fazer consultas na base de dados. Em nosso caso, seguindo os padrões normalmente recomendados, foi desenvolvida apenas uma instância do banco de dados para todo aplicativo. Esta nada mais é que uma classe abstrata que estende a classe `RoomDatabase`. A mesma deve conter a anotação `@Database`. Como mencionado anteriormente, o banco implementa o

Figura 21 – Criação do `DAO`

```
@Dao
public interface MyDAO {

    // permite a inserção de várias receitas iguais
    // usando uma estratégia de conflito
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    void insert(Receita receita);

    @Query("DELETE FROM receita_table")
    void deleteAll();

    @Query("SELECT * FROM receita_table ORDER BY valor ASC")
    LiveData<List<Receita>> getAlphabetizedReceitas();

    @Query("SELECT * FROM receita_table")
    List<Receita> getArrayAllReceitas();

    @Query("SELECT sum(valor) FROM receita_table")
    double valorTotalReceitas();
```

Fonte: Elaborado pelo autor (2021)

padrão de projeto `Singleton` para evitar que várias intâncias sejam abertas ao mesmo tempo. Como podemos ver na Figura 22, o método *getDatabase*() devolve o `Singleton`.

5. O repositório não faz parte dos componentes de arquitetura do `Jetpack`, mas sua utilização é uma prática recomendada para separação do código e arquitetura do aplicativo. Ele é o componente que irá gerenciar todas as consultas ao repositório local ou remoto. Ao invés de passar todo banco de dados para o repositório, é passado apenas o DAO. Assim sendo, só é necessário acessar o DAO, que irá conter todos os métodos de leitura e escrita na base de dados.

6. A próxima etapa do desenvolvimento foi a criação do `ViewModel` para manter a interface do usuário sempre atualizada com os dados reais do aplicativo. Dessa forma, os dados da IU são separados das suas classes de `Activity` ou `Fragment`. Assim, é seguido o princípio de responsabilidade única, ou seja, os componentes de IU só são responsáveis por montar e apresentar as telas para o usuário, deixando o processo de cuidar e manter os dados a cargo do `ViewModel`. O repositório e IU são totalmente separados pelo `ViewModel`. Não existem chamadas ao banco de dados pelo `ViewModel`. Todo processo é realizado no repositório e, assim, os métodos de pesquisa no banco devem retornar uma referência para o repositório. A Figura 23 ilustra um trecho do `ViewModel` implementado.

Figura 22 – Criação do RoomDatabase

```
@Database(entities = {Receita.class, Despesa.class , Usuario.class}, version = 1, exportSchema = false)
public abstract  class MyRoomDatabase extends RoomDatabase {

    public abstract MyDAO myDAO();

    private static volatile MyRoomDatabase INSTANCE;
    private static final int NUMBER_OF_THREADS = 4;
    static final ExecutorService databaseWriteExecutor =
            Executors.newFixedThreadPool(NUMBER_OF_THREADS);

    static MyRoomDatabase getDatabase(final Context context) {
        if (INSTANCE == null) {
            synchronized (MyRoomDatabase.class) {
                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                            MyRoomDatabase.class,  name: "my_database")
                            .build();
                }
            }
        }
        return INSTANCE;
    }
```

Fonte: Elaborado pelo autor (2021)

7. Para buscar os dados de câmbio via *Web Service*, foi utilizada a API do `AwesomeApi`[6]. A mesma disponibiliza um *endpoint* de forma gratuita de cotação de moedas. As informações são atualizadas a cada 30 segundos e são disponibilizadas as cotações de mais 150 moedas diferentes. Vale ressaltar que, no decorrer do desenvolvimento, era utilizada a API da `Exchange Rates`[7]. Entretanto, ela sofreu alterações em seus termos de uso e passou a oferecer um serviço pago, o que levou a migração do serviço para o `AwesomeApi`.

8. Posteriormente, foram criados todos componentes visuais do aplicativo, os aquivos XML, *RecyclerViews* e as *Activities* para todas as regras de interação com o usuário existentes.

9. Para autenticação do aplicativo, foi utilizado o serviço de autenticação do Firebase[8] utilizando e-mail e senha.

Foi elaborado um diagrama para modelar a estrutura da aplicação e a comunicação entre os componentes na funcionalidade de inserção de uma nova receita. A Figura 24 ilustra esse diagrama e mostra como ocorre a comunicação entre os componentes. Como podemos notar a *view* responsável por exibir a lista de receitas está sincronizada com o `ViewModel`

[6] <https://docs.awesomeapi.com.br/api-de-moedas> Acesso em: 15 Jun. 2021.
[7] <https://exchangeratesapi.io/> Acesso em: 15 Jun. 2021.
[8] <https://firebase.google.com/> Acesso em: 15 Jun. 2021.

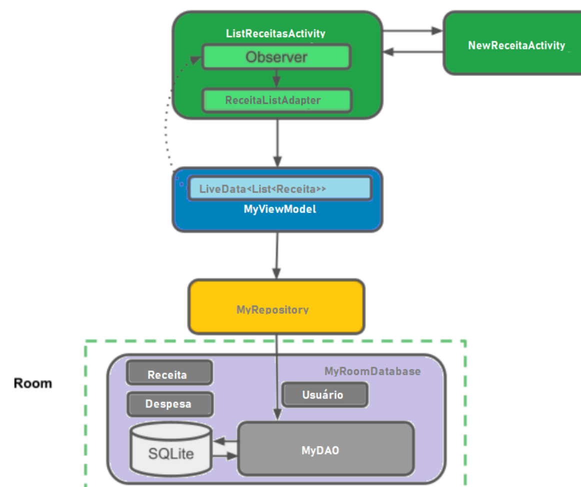
Figura 23 – Criação do *ViewModel*

```
public List<Despesa> getArrayAllDespesas() throws ExecutionException, InterruptedException {
    return mRepository.getArrayAllDespesas();
}

public Double valorTotalReceitas () throws ExecutionException, InterruptedException{
    return mRepository.valorTotalReceitas();
}

public Double valorTotalDespesas () throws ExecutionException, InterruptedException{
    return mRepository.valorTotalDespesas();
}

public void insert(Receita receita) { mRepository.insert(receita); }
public void insert(Despesa despesa) { mRepository.insert(despesa); }

public void update(Receita receita) { mRepository.update(receita); }
public void updateD(Despesa despesa) { mRepository.updateD(despesa); }

public void delete(Receita receita) { mRepository.delete(receita); }
public void deleteD(Despesa despesa) { mRepository.deleteD(despesa); }
```

Fonte: Elaborado pelo autor (2021)

por meio de um *observer*. Consequentemente, o `ViewModel` realiza as requisições necessárias no repositório para obter os dados atualizados. Esse processo só é possível graças a lista de receitas do tipo `LiveData`, que é a classe armazenadora de dados observáveis.

Figura 24 – Processo de inserção de uma nova Receita

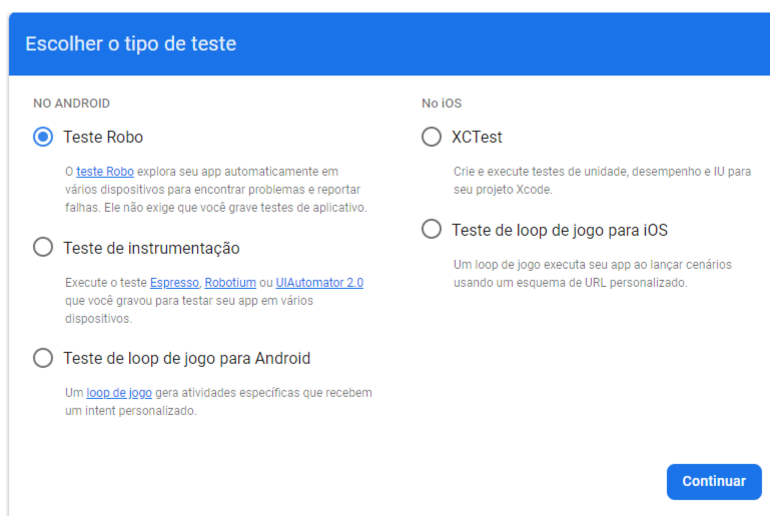


Fonte: Elaborado pelo autor (2021) adaptado de Google (2021a)

3.5 Testes

Todas as duas versões do aplicativo foram testadas usando os testes de interface com o usuário IU. Para a realização dos testes, foi usada a plataforma do Firebase, especificamente o serviço do Firebase Test Lab[9]. Todas as versões do aplicativo possuem a mesma IU. Porém, as versões foram testadas e validadas individualmente. Para realizar o teste, basta adicionar o arquivo `.apk` gerado pela aplicação e selecionar o dispositivo no qual se deseja realizar os testes. A Figura 25 ilustra o escopo dos testes que foram desenvolvidos na plataforma. A Figura 26 ilustra a quantidade de dispositivos disponíveis para a realização dos testes e os que foram utilizados para testar as versões, vale ressaltar que a versão gratuita da plataforma disponibiliza a realização de cinco testes diários em dispositivos físicos e dez em dispositivos virtuais.

Figura 25 – Escopo dos testes

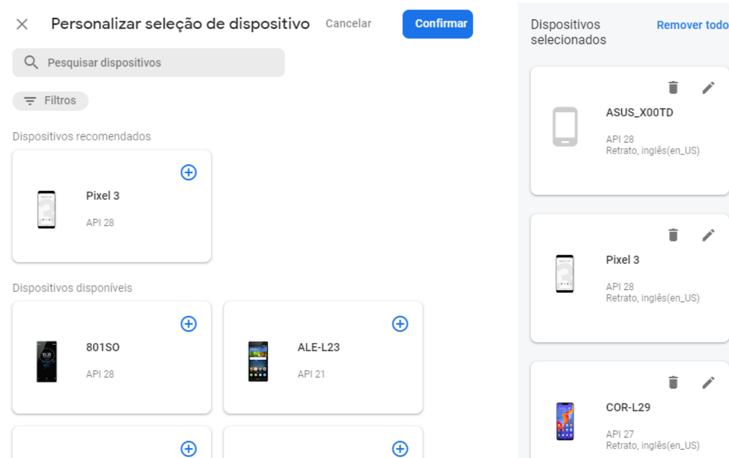


Fonte: Firebase, Google (2021)

Na processo de validação da aplicação várias sugestões de melhoria foram informadas pelo *framework* de testes do Firebase. A Figura 27 ilustra uma das sugestões levantadas nos testes que posteriormente foram aplicadas na aplicação, já a Figura 28 demostra uma visão geral dos resultados coletados em um dos testes de validação, seguida pela Figura 29 que demostra os recursos consumidos pela aplicação durante um dos testes, onde podemos ver a utilização dos recursos de *Central Processing Unit* (CPU) memória e rede.

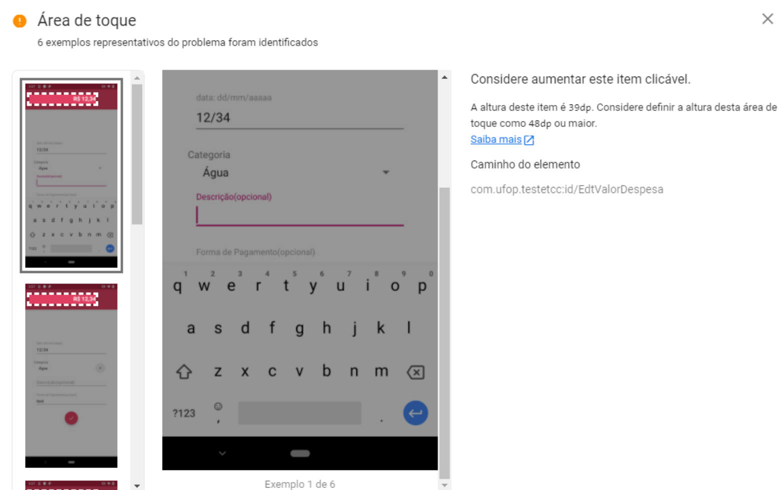
[9] <https://firebase.google.com/docs/test-lab> Acesso em: 15 Jun. 2021

Figura 26 – Dispositivos testados



Fonte: Firebase, Google (2021)

Figura 27 – Dicas de melhoria



Fonte: Firebase, Google (2021)

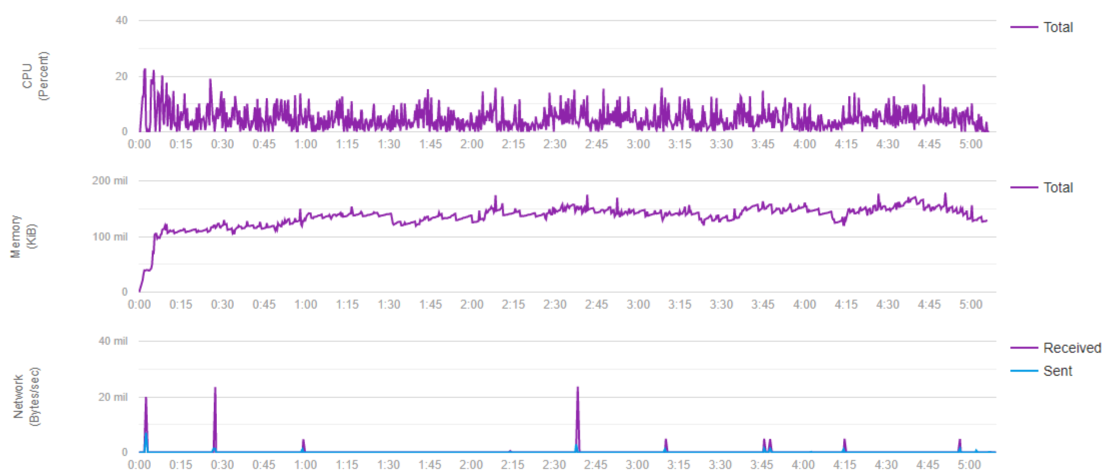
3.6 Considerações

Este capítulo preocupou-se em apresentar as etapas do desenvolvimento do trabalho, onde foram descritas as tecnologias escolhidas para o desenvolvimento, a fase de levantamento de requisitos, modelagem do banco de dados local da aplicação, estruturação dos principais componentes utilizados no *back-end* e a etapa de testes de IU, bem como, foram apresentadas todas ferramentas de suporte para realização das tarefas supracitadas.

Figura 28 – Visão geral dos testes

Fonte: Firebase, Google (2021)

Figura 29 – Consumo de recursos durante o teste.



Fonte: Firebase, Google (2021)

4 Resultados

4.1 Apresentação da aplicação

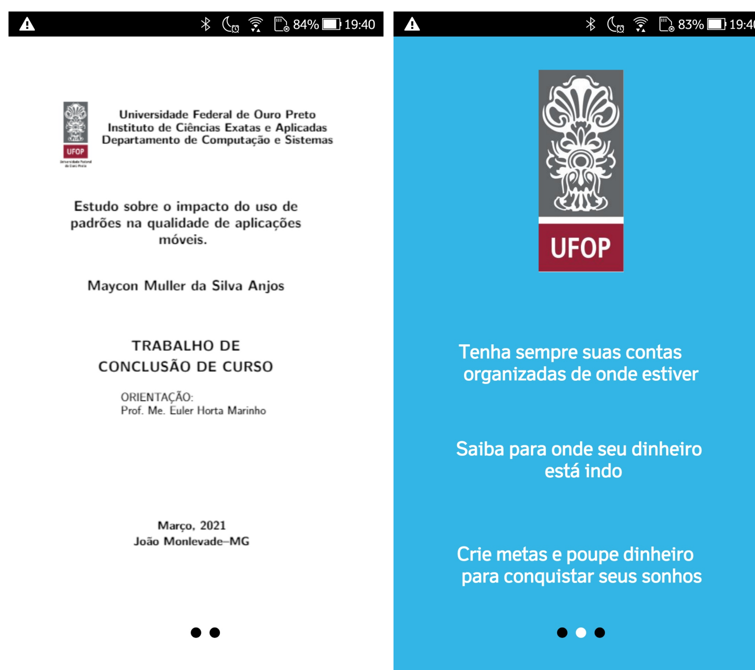
A aplicação desenvolvida consiste em um gerenciador financeiro pessoal. Ela apresenta funcionalidades que permitem inserir receitas e despesas, visualizar o saldo em valor na carteira, bem como, navegar pelos meses e visualizar quais são as receitas e despesas no referido mês. Ainda, é possível visualizar de forma gráfica o percentual gasto em cada categoria de despesas no mês em questão. Na Subseção 4.1.1, são apresentadas todas telas e funcionalidades da aplicação. Vale ressaltar que, as duas aplicações utilizaram as mesmas interfaces (IU). Dessa forma, nessa etapa não se faz necessário discernir uma aplicação da outra.

4.1.1 Funcionalidades da Aplicação

No processo de início de execução da aplicação as primeiras interfaces que são exibidas para o usuário sãos os chamados *Sliders*. Esses têm a função de fornecer uma prévia sobre as informações e funcionalidades que estão contidas no aplicativo. Após a exibição dos mesmos, é apresentada a tela de cadastro da aplicação. Todavia, se o usuário já possuir uma conta, ele pode clicar no texto "Já tenho conta" e efetuar o login na aplicação. A Figura 30 ilustra a exibição dos dois primeiros *sliders*. De forma similar, a Figura 31 exibe a tela de direcionamento para o *login* ou cadastro na aplicação.

As próximas exibições criadas foram as telas de cadastro e *login* na aplicação. Após realizar o cadastro ou *login*, o usuário é redirecionado à página principal da aplicação. Vale ressaltar que, se o usuário estiver com uma sessão ativa no sistema, ele é automaticamente redirecionado para a tela principal do aplicativo. A Figura 32 ilustra as telas de cadastro e *login* do aplicativo.

Figura 30 – Tela de informações *Sliders*

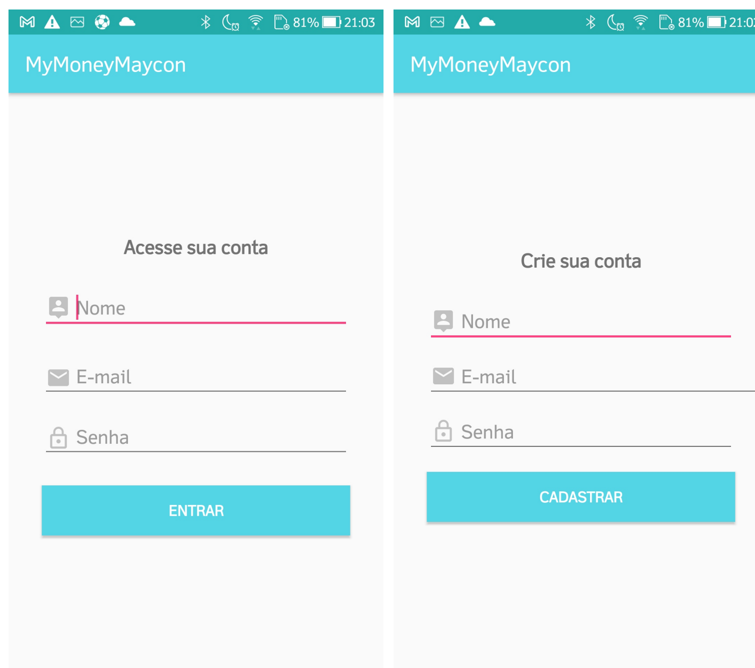


Fonte: Elaborado pelo autor (2021)

Figura 31 – Tela de direcionamento *Sliders*

Fonte: Elaborado pelo autor (2021)

Figura 32 – Tela de cadastro e *login*

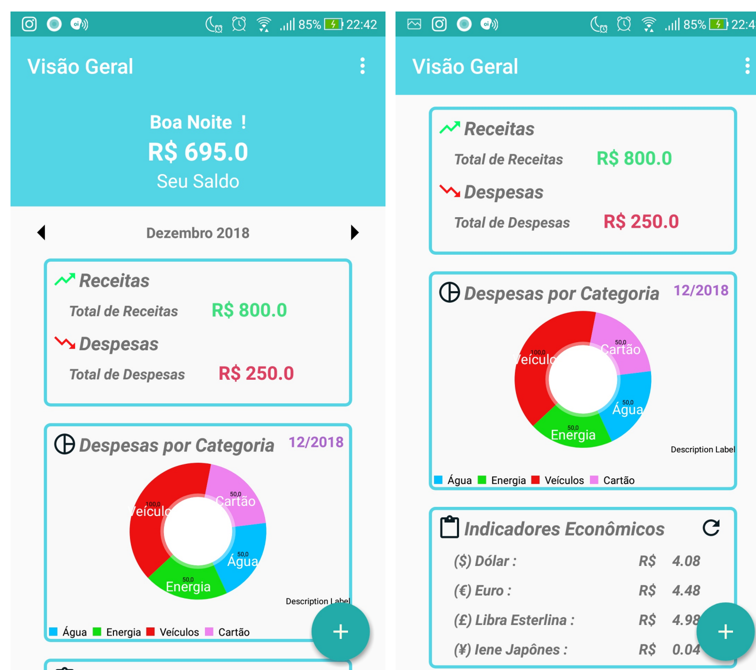


Fonte: Elaborado pelo autor (2021)

Como dito anteriormente, após a realização de todo processo de autenticação na aplicação, o usuário é redirecionado para a tela principal (*Dashboard*). Nesta, é possível visualizar o balanço geral, uma mensagem de boas vindas e um gráfico das categorias das despesas onde mais se tem destinado dinheiro. Vale ressaltar que todos os dados do *Dashboard* são atualizados de acordo com a seleção do mês no `CalendarView` pelo usuário. Ainda é possível visualizar um *layout* com as principais cotações de moedas em tempo real. Esses são os dados coletados através da API da `AwesomeApi`[1], que apresenta os dados atualizados a cada 30 segundos. A Figura 33 ilustra a tela principal da aplicação e suas principais funcionalidades.

[1] <https://docs.awesomeapi.com.br/api-de-moedas> Acesso em: 30 Jun. 2021.

Figura 33 – Tela principal Dashboard



Fonte: Elaborado pelo autor (2021)

Como podemos notar, a tela principal de *Dashboard* possui um botão no canto inferior direito da tela. Esse é um botão de ação flutuante que é utilizado para criar uma nova receita ou despesa. A Figura 34 ilustra a ação do botão. Quando o mesmo é acionado é aberto um menu com a opção de escolher entre uma nova receita ou despesa.

Figura 34 – Botão de Ação Flutuante `Floating Action Menu`

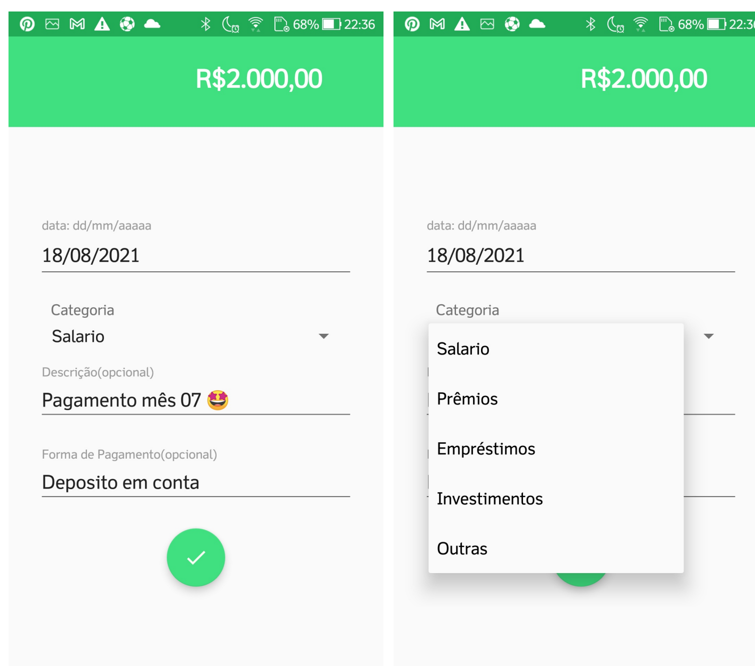
Fonte: Elaborado pelo autor (2021)

Ao selecionar uma das opções no botão de ação, um novo *layout* é exibido para o usuário para criar uma nova receita ou despesa. A Figura 35 ilustra o processo de criação de uma nova receita. Isso exige o preenchimento dos campos pelo usuário, alguns obrigatórios e outros opcionais. O *layout* possui uma máscara para o campo valor em reais e outra para o formato correto da data, que vem previamente preenchida com a data atual do sistema.

A segunda imagem da Figura 35 ilustra que para a seleção da categoria, foi criado um *Spinner*. Esse processo também foi utilizado como estratégia para limitar a quantidade de dados de entrada que devem ser fornecidos pelo usuário, facilitando assim, o processo de manipulação dos mesmos.

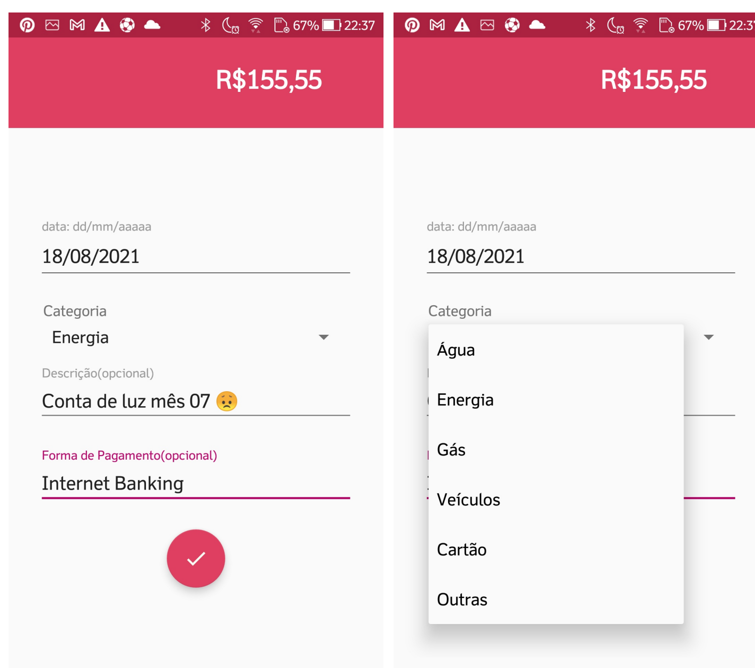
A Figura 36 ilustra todo esse processo mencionado anteriormente, agora para inserção de uma nova despesa no aplicativo.

Figura 35 – Cadastro de uma nova Receita



Fonte: Elaborado pelo autor (2021)

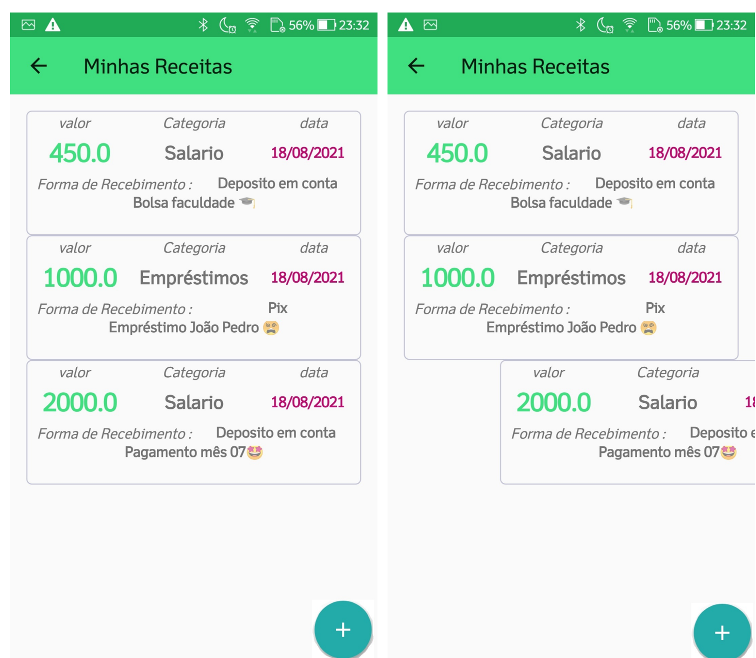
Figura 36 – Cadastro de uma nova Despesa



Fonte: Elaborado pelo autor (2021)

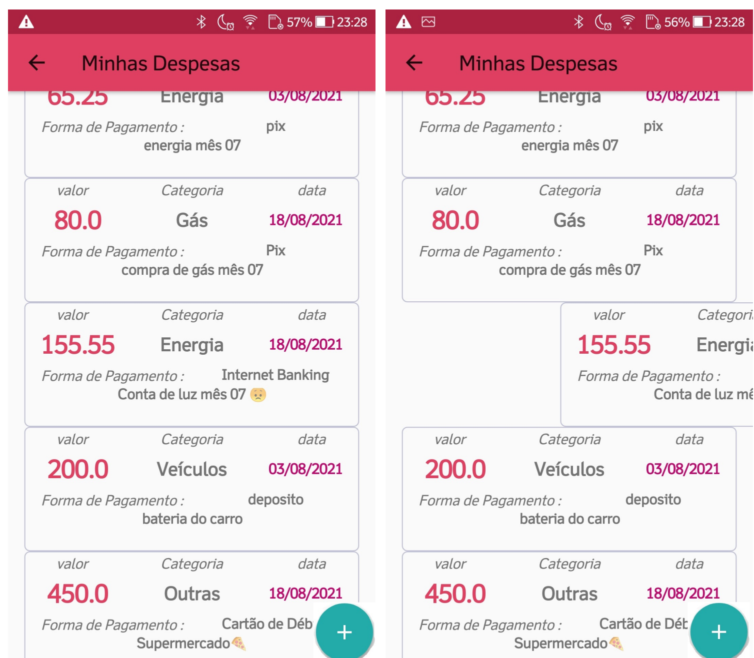
Para listar as receitas e despesas, foram criados dois novos *layouts*. Nestes, são exibidas as listas com as receitas e despesas criadas de acordo com o mês selecionado na tela inicial. Por meio dessa lista, é possível deletar uma receita ou despesa, apenas arrastando um item de visualização para esquerda ou direita. Nesse caso, foi implementada uma estratégia de exclusão usando o método `onSwiped` do Android. A Figura 37 ilustra o processo para a lista de receitas. Já a Figura 39 demostra como ocorre o processo para a lista de despesas.

Figura 37 – Listar ou Remover uma Receita



Fonte: Elaborado pelo autor (2021)

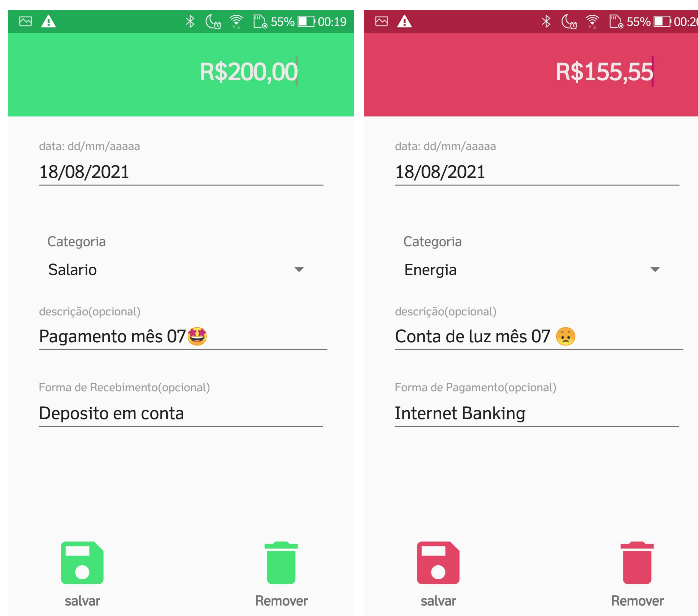
Figura 38 – Listar ou Remover uma Despesa



Fonte: Elaborado pelo autor (2021)

Por último, foi desenvolvida a funcionalidade de editar uma receita ou despesa. Essa ação é disparada com o toque do usuário sobre uma receita ou despesa selecionada nas respectivas listas. Com o evento do toque, um novo *layout* é acionado para realizar a edição ou até mesmo a exclusão de um item. Vale a pena ressaltar que todos os dados referentes ao item são previamente carregados na *Activity*, para facilitar o processo de edição ou exclusão pelo usuário. A Figura 39 demostra esse processo relatado para uma receita e despesa.

Figura 39 – Editar/ Remover um item de lista



Fonte: Elaborado pelo autor (2021)

4.2 Métricas / Análises

No processo de análise das versões desenvolvidas da aplicação, foram utilizadas as métricas de Chidamber e Kemerer CK. Essas serviram como uma base para mensurar as características da qualidade de software que estão atreladas à manutenibilidade. Para a obtenção dos dados paras as seis métricas CK, foi utilizado o plugin *MétricsReloaded*[2]. O mesmo se trata de uma ferramente amplamente conhecida, para a obtenção de métricas, mediante a análise do código fonte da aplicação. Assim sendo, o *plugin* foi integrado ao Android Studio para calcular as seis métricas CK.

Ademais, vale a pena ressaltar que os resultados advindos das métricas de Chidamber e Kemerer CK, são obtidos através de uma análise individual, para cada uma das classes ou interfaces da aplicação. A partir do conjunto de valores é calculada uma média.

Consoante com as informações apresentadas, a seção 4.3, apresenta os resultados obtidos de forma individual, para cada uma das seis métricas CK.

4.3 Análises das Métricas CK

Essa seção visa apresentar de forma detalhada os valores obtidos para as métricas CK para as duas versões da aplicação. Com isso, busca-se, examinar os referidos resultados

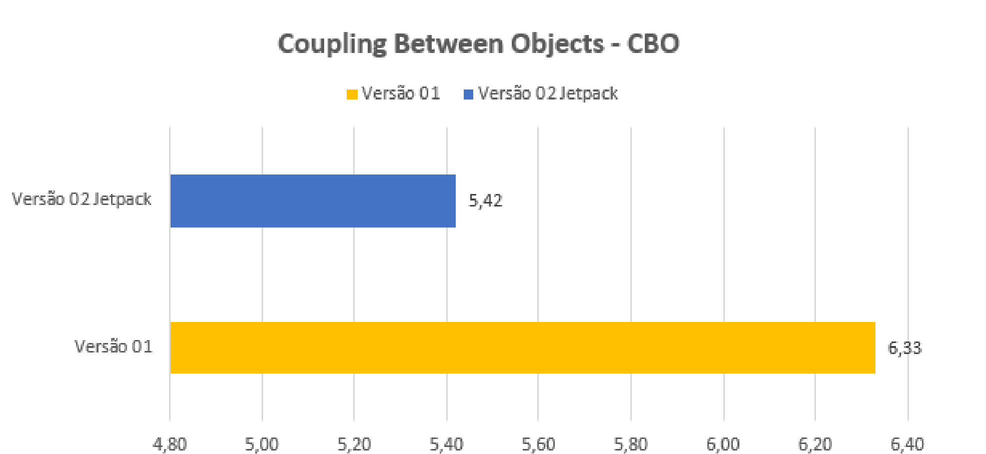
[2] <https://plugins.jetbrains.com/plugin/93-metricsreloaded> Acesso em 30 Mai. 2021

de forma individual para justificar os valores auferidos.

4.3.1 Coupling Between Objects (CBO)

O acoplamento entre classes de objetos, tem por finalidade mensurar o grau de dependência entre as classes da aplicação. Um exemplo para ilustrar esse processo, é quando alteramos um método público de uma classe "X" e o mesmo é usado por uma classe "Y", você é obrigado a alterar esse método também em "Y". Corroborando, Sommerville (2011) menciona que um valor alto para o CBO significa que as aplicações são altamente dependentes. A Figura 40 ilustra o resultado para a métrica CBO para cada uma das versões da aplicação.

Figura 40 – Acoplamento entre classes de objetos



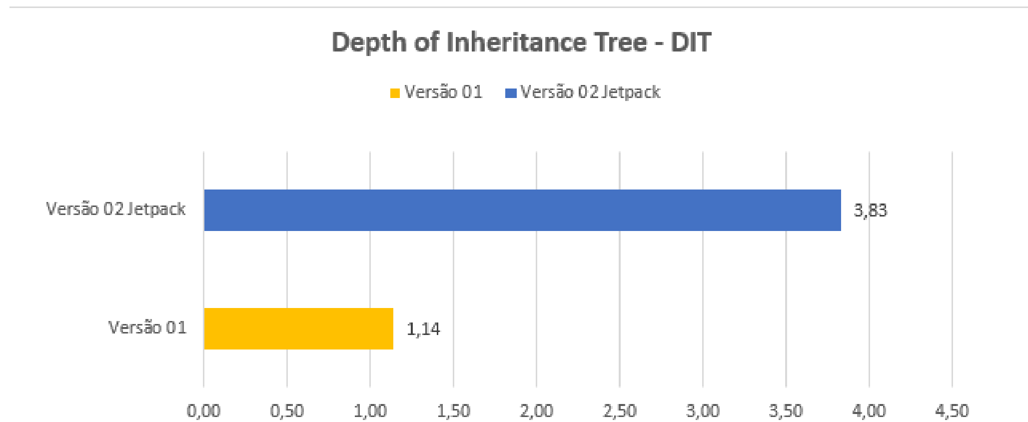
Fonte: Elaborado pelo autor (2021)

Nessa primeira análise, é perceptível as diferenças de valores nas versões. É apresentada uma significativa vantagem para a aplicação desenvolvida usando o *Jetpack*, provavelmente por seu desenvolvimento abordar componentes com alto grau de modularização e, portanto, coesos.

4.3.2 Depth of Inheritance Tree (DIT)

Essa métrica basicamente demostra o quão complexo é o projeto. Ela reflete o número de níveis existentes na árvore de herança. Consequentemente, quando maior o valor dessa métrica mais profunda é a árvore de herança é mais incompreensível o projeto se torna. A demonstra os resultados obtidos para a DIT para cada uma das versões da aplicação.

Figura 41 – Profundidade da Arvore de Herança



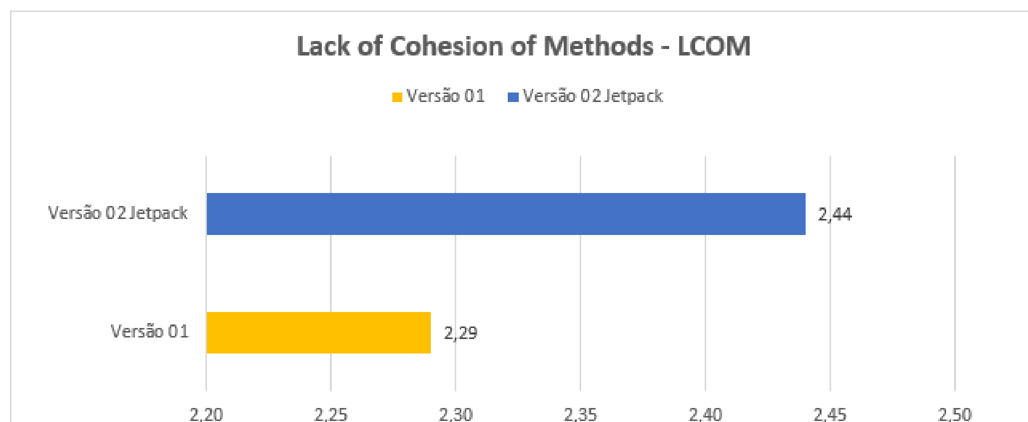
Fonte: Elaborado pelo autor (2021)

Contrastando com a métrica anteriormente apresentada, a métrica DIT apresentou um valor um tanto quanto maior para a aplicação desenvolvida usando o *Jetpack* com o valor de 3.83, enquanto a versão 01 obteve o valor de 1.14.

4.3.3 Lack of Cohesion of Methods (LCOM)

Segundo Sommerville (2011), a métrica de falta de coesão em métodos LCOM é calculada considerando os pares de métodos em uma classe. Ela transmite a diferença entre o número de pares de métodos sem atributos compartilhados e o número de pares de métodos com atributos compartilhados. A Figura 42 apresenta os valores coletados para a métrica LCOM em ambas as aplicações.

Figura 42 – Falta de Coesão em Métodos



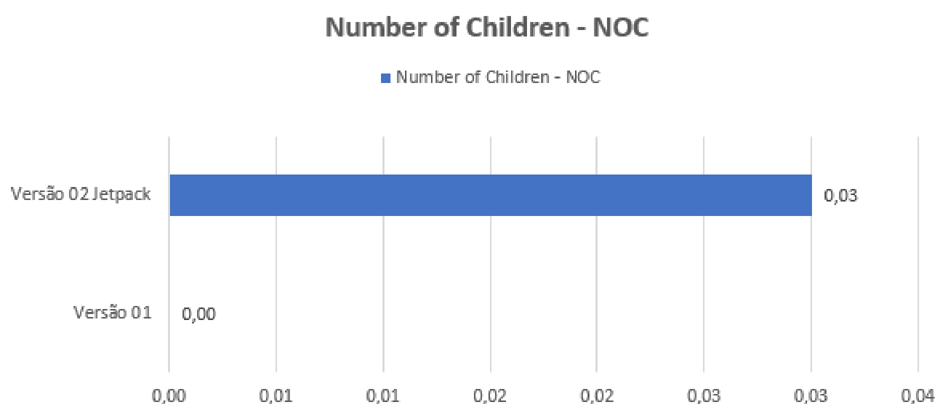
Fonte: Elaborado pelo autor (2021)

A aplicação desenvolvida com os componentes de arquitetura do *Jetpack* também obteve um valor maior que a versão 01 para a métrica LCOM. Os valores foram respectivamente iguais à 2,44 e 2,29 para as duas versões. Ademais, a diferença não foi tão discrepante quanto na métrica analisada anteriormente.

4.3.4 Number of Children (NOC)

Segundo Sommerville (2011), essa métrica indica a largura de uma hierarquia de classe. Ele menciona que um valor alto para NOC possivelmente indica maior reúso. A Figura 43 demonstra os resultados obtidos para a referida métrica para ambas as aplicações.

Figura 43 – Número de filhos



Fonte: Elaborado pelo autor (2021)

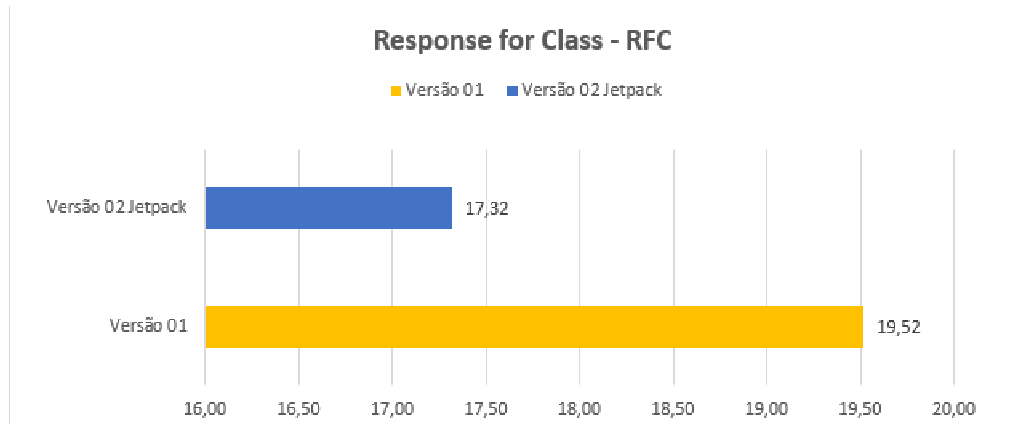
Os valores obtidos para a métrica NOC para aplicação desenvolvida nos padrões do *Jetpack* foi de 0,03. Neste caso, foi contabilizado o valor um, apenas para uma classe de toda aplicação. O valor da métrica para a primeira aplicação foi zero. O *plugin* utilizado não computa o processo de utilização das bibliotecas do Android durante o desenvolvimento, o que ocasionou a geração de valores tão baixos para ambas as versões.

4.3.5 Response for Class (RFC)

Para Sommerville (2011), a métrica RFC, Resposta para uma classe, tem relação com a medida do número de métodos que podem ser executados em resposta à uma mensagem recebida por um objeto da classe. Ele ainda menciona que, quanto maiores os valores para esse métrica, mais complexa é a classe. A Figura 44 ilustra os valores obtidos para essa métrica, para as duas versões desenvolvidas.

Dessa vez, a aplicação construída usando o *Jetpack* obteve valores menores em comparação à primeira aplicação desenvolvida. Esta aplicação apresentou um valor de 17,32 contra 19,52 para aplicação sem o uso dos componentes de arquitetura do *Jetpack*.

Figura 44 – Resposta para uma classe

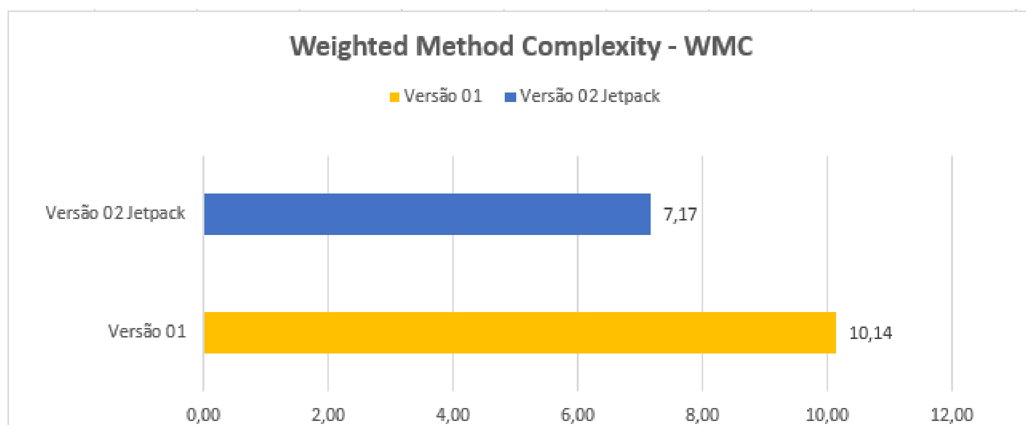


Fonte: Elaborado pelo autor (2021)

4.3.6 Weighted Method Complexity (WMC))

Como exposto por Sommerville (2011), o resultado da métrica de Métodos ponderados por classe WMC evidencia o número de métodos em cada classe, ponderando pela complexidade de cada método, segundo ele quanto maior essa métrica, mais complexa é a classe de objeto. A Figura 45 ilustra os resultados obtidos para essa métrica em ambas aplicações.

Figura 45 – Métodos ponderados por classe



Fonte: Elaborado pelo autor (2021)

Mais uma vez, a aplicação que empregou os componentes de arquitetura do *Jetpack* apresentou um menor valor para essa métrica, a mesma obteve o valor de 7,17, já a versão 01 atingiu o valor de 10,14.

4.4 Considerações

Neste capitulo foi apresentado as duas versões da aplicação de controle finaneceiro, bem como, as principais funcionalidades das versões. Posteriormente, foram denotados os resultados obtidos em cada uma das seis métricas CK, para cada uma das versões. Resumindo, a versão desenvolvida usando os componentes de arquitetura do *Jetpack* alcançou melhores resultados para as seguintes métricas: *Coupling Between Objects* (CBO), *Number of Children* (NOC), *Response for Class* (RFC) e *Weighted Method Complexity* (WMC). Por sua vez, a versão do aplicativo desenvolvida sem os padrões de arquitetura do *Jetpack* obteve um melhor resultado para as métricas : *Depth of Inheritance Tree* (DIT) e *Lack of Cohesion of Methods* (LCOM). A Tabela 1, demonstra todos os valores das métricas CK gerados para as duas versões da aplicação de controle financeiro pessoal.

Tabela 1 – Métricas CK para cada uma das versões do aplicativo.

Versões da Aplicação	**Métricas CK**					
	CBO	DIT	LCOM	NOC	RFC	WMC
Versão 01	6.33	1.14	2.29	0.0	19.52	10.14
Versão 02 (*Jetpack*)	5.42	3.83	2.44	0.03	17.32	7.17

Fonte – Elaborado pelo autor (2021).

5 Considerações Finais

Este trabalho apresentou o desenvolvimento de uma aplicação de controle financeiro pessoal, que serviu como estudo de caso para a avaliação do uso dos componentes de arquitetura do *Jetpack*, como um padrão para desenvolvimento de aplicações nativas em Android. Foram desenvolvidas duas versões da aplicação, uma compreendendo os principais componentes de arquitetura do *Jetpack* e outra sem utilizar quaisquer componentes de arquitetura do *Jetpack*. Com a finalidade de analisar e mensurar a qualidade das versões desenvolvidas, foram utilizadas as métricas de Chidamber e Kemerer CK para guiar as análises. A partir das mesmas, foi possível aferir a superioridade da aplicação desenvolvida usando os componentes de arquitetura do *Jetpack*.

Essa superioridade foi constatada pela compreensão da arquitetura da aplicação e, ainda, pelo discernimento da função de cada componente do *Jetpack* dentro aplicação. Os valores auferidos para quatro métricas de Chidamber e Kemerer CK foram favoráveis para a aplicação desenvolvida usando o *Jetpack*.

Para as métricas *Depth of Inheritance Tree* (DIT) e *Lack of Cohesion of Methods* (LCOM) os valores não foram favoráveis para a aplicação que utiliza o *Jetpack*. Entretanto, o maior valor obtido para a métrica DIT, possivelmente, evidencia uma melhor estratégia de reúso. Para a métrica LCOM, a diferença entre os valores das duas aplicações não foi considerável. Vale a pena ressaltar que não existem valores limiares universais para a interpretação dos valores obtidos pelas métricas.

A interpretação adequada dos valores das métricas é essencial para caracterizar, avaliar e melhorar o *design* de sistemas de *software*. Sem conhecer um limiar para uma determinada métrica, a comunidade de *software* não será capaz de aplicar métricas de *software* de uma forma efetiva (FERREIRA et al., 2012).

Ademais, este estudo descreveu toda fundamentação teórica relacionada ao assunto, desde o processo de levantamento de requisitos até a concepção e validação das versões da aplicação pelos testes. Os resultados iniciais sugerem a eficácia do uso de padrões no desenvolvimento de aplicações nativas do Android.

Apesar de não ser o foco principal deste estudo, a aplicação desenvolvida pode ser usada por qualquer pessoa que esteja em busca de controlar seus ganhos e gastos. Vale a pena ressaltar que, o foco deste estudo foi em fornecer uma análise qualitativa advinda da comparação das versões desenvolvidas. Dessa forma, a aplicação não possui um usuário ou público-alvo específico.

Certamente, a realização deste trabalho contribuiu no processo de aprendizado e

desenvolvimento de aplicações nativas em Android e, ainda, expôs as diferentes perspectivas de desenvolvimento para o onipresente domínio de aplicações móveis.

5.1 Limitações e trabalhos futuros

A principal limitação encontrada, foi a falta de experiência no desenvolvimento de aplicações nativas do Android, visto que todo o aprendizado das ferramentas e tecnologias foram adquiridos ao decorrer do desenvolvimento do projeto. Isso, de certa forma, limitou a construção de uma aplicação com mais funcionalidades.

No processo de desenvolvimento deste trabalho, vários pontos para a execução de trabalhos futuros foram identificados. Dentre eles vale destacar que o desenvolvimento de aplicações usando a linguagem de programação Kotlin [1] é um ponto a se considerar, visto que o *Jetpack* possui um conjunto de extensões para se trabalhar com Kotlin denominado Android KTX[2]. Com as extensões do KTX, o Jetpack, a Plataforma Android e outras *APIs* podem fazer uso de uma linguagem Kotlin concisa e idiomática. (GOOGLE, 2021b)

A utilização do Jetpack Compose[3] também pode ser explorada em trabalhos futuros. Este se trata de um *kit* de ferramentas do Android para criar IU nativas com mais rapidez e menos código.

A utilização de outros componentes do Android *Jetpack*, tais como :

`WorkManager`[4] : Esse se trata de um componente do *Jetpack* para facilitar a programação de tarefas assíncronas (em segundo plano) que precisam ser executadas mesmo se a aplicação fechar ou o dispositivo for reiniciado. (GOOGLE, 2021b)

`Paging 2`[5]: O *Paging 2* é uma estratégia para trabalhar com dados sob demanda, ou seja, carregar e exibir para o usuário pequenos blocos de dados por vez, o que reduz o uso de largura de banda da rede e dos recursos do sistema. (GOOGLE, 2021b)

`Test`[6]: O *Test* é um componente do *Jetpack* voltado para execução de testes, com ele é possível verificar a precisão, o comportamento funcional e a usabilidade do aplicativo antes de lançá-lo publicamente. (GOOGLE, 2021b)

A avaliação de outras métricas para mensurar a qualidade de uma aplicação Android também é algo válido para trabalhos futuros, visto que, as métricas CK são métricas voltadas para linguagens orientadas a objetos e não englobam a linguagem de marcação XML utilizada pelas aplicações nativas do Android.

[1] <https://developer.android.com/kotlin?hl=pt-br> Acesso em 12 Ago. 2021.
[2] <https://developer.android.com/kotlin/ktx?hl=pt-br> Acesso em 12 Ago. 2021.
[3] <https://developer.android.com/jetpack/compose?hl=pt-br> Acesso em 12 Ago. 2021.
[4] <https://developer.android.com/reference/androidx/work/WorkManager> Acesso em 12 Ago. 2021.
[5] <https://developer.android.com/topic/libraries/architecture/paging> Acesso em 12 Ago. 2021.
[6] <https://developer.android.com/training/testing> Acesso em 12 Ago. 2021.

Referências

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, IEEE, v. 20, n. 6, p. 476–493, 1994. Citado na página 36.

COHN, M. *User stories applied: For agile software development*. [S.l.]: Addison-Wesley Professional, 2004. Citado na página 40.

DEITEL, H.; DEITEL, P.; DEITEL, A. *Android: Como Programar*. [S.l.]: Bookman Editora, 2015. Citado na página 22.

DIGITAL HOUSE. *DigitalHouse*. 2021. Disponível em: <https://www.digitalhouse.com/br/blog/crescimento-do-mercado-de-aplicativos-brasil>. Acesso em: 25 set. 2021. Citado na página 16.

DROIDCON. *Pro Android dev*. 2017. Disponível em: <https://proandroiddev.com/android-architecture-components-cb1ea88d3835>. Acesso em: 09 Jun. 2021. Citado 3 vezes nas páginas 27, 29 e 30.

FERREIRA, K. A. et al. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, Elsevier, v. 85, n. 2, p. 244–257, 2012. Citado 2 vezes nas páginas 37 e 68.

FIREBASE, GOOGLE. *Firebase Test Lab*. 2021. Disponível em: <https://firebase.google.com/docs/test-lab>. Acesso em: 23 Jun. 2021. Citado 5 vezes nas páginas 38, 39, 51, 52 e 53.

GAMMA, E. *Padrões de projetos: soluções reutilizáveis*. [S.l.]: Bookman editora, 2009. Citado 2 vezes nas páginas 19 e 20.

GOOGLE. *Blog Google Developers*. 2018. Disponível em: <https://developers-br.googleblog.com/2018/05/use-o-android-jetpack-para-acelerar-o.html>. Acesso em: 31 mai. 2021. Citado 2 vezes nas páginas 16 e 23.

GOOGLE. *Developer CodeLabs Android*. 2021. Disponível em: <https://developer.android.com/codelabs/android-room-with-a-view/>. Acesso em: 10 Jun. 2021. Citado 4 vezes nas páginas 28, 29, 30 e 50.

GOOGLE. *Documentation for app developers*. 2021. Disponível em: <https://developer.android.com/docs>. Acesso em: 31 mai. 2021. Citado 10 vezes nas páginas 22, 24, 25, 26, 27, 28, 29, 30, 31 e 69.

ISO/IEC. ISO/IEC 25010-systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—system and software quality models. 2010. Citado 2 vezes nas páginas 33 e 34.

LAUDON, K. C.; LAUDON, J. P. *Sistemas de informação gerenciais. 9ª Edição*. [S.l.]: São Paulo: Pearson, 2011. Citado na página 19.

LECHETA, R. R. *Google Android-3ª Edição: Aprenda a criar aplicações para dispositivos móveis com o Android SDK*. [S.l.]: Novatec Editora, 2013. Citado na página 22.

MAINKAR, P. *Expert Android Programming: Master skills to build enterprise grade Android applications*. [S.l.]: Packt Publishing Ltd, 2017. Citado na página 38.

OLIVEIRA, R. M. S. D. *Estudo sobre a utilização do Android Jetpack no desenvolvimento de aplicativos Android*. Tese (Doutorado) — UNIVERSIDADE FEDERAL DE PERNAMBUCO, 2018. Citado 2 vezes nas páginas 31 e 32.

PRESSMAN, R.; MAXIM, B. *Engenharia de Software-8ª Edição*. [S.l.]: McGraw Hill Brasil, 2016. Citado na página 32.

RANK MYAPP. *Rank MyApp*. 2021. Disponível em: <https://www.rankmyapp.com/pt-br/?utm_source=blog&utm_medium=DH&utm_campaign=home-rankmyapp-pt>. Acesso em: 25 set. 2021. Citado na página 16.

SANTA, J. A. d. U. campus. Sistema gerenciador financeiro pessoal móvel. 2012. Citado na página 17.

SOMMERVILLE, I. *Software engineering 9th ed*. [S.l.]: Addison-Wesley/Pearson, 2011. Citado 13 vezes nas páginas 16, 17, 19, 21, 32, 35, 36, 38, 43, 63, 64, 65 e 66.

SYROMIATNIKOV, A.; WEYNS, D. A journey through the land of model-view-design patterns. In: IEEE. *2014 IEEE/IFIP Conference on Software Architecture*. [S.l.], 2014. p. 21–30. Citado na página 21.

VALENTE, M. T. *Engenharia de Software Moderna*. [S.l.]: Independente, 2020. Citado na página 37.

WIKIMEDIA COMMONS. *Repositório de mídia livre*. 2021. Disponível em: <https://commons.wikimedia.org/wiki/File:MVVMPattern.png>. Acesso em: 23 Jun. 2021. Citado na página 22.