

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
GOIANO - CAMPUS URUTAÍ
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ADRILEY SAMUEL RIBEIRO BARBOSA

**ANÁLISE COMPARATIVA ENTRE OS
PADRÕES MVC, MVP, MVVM E MVI NA
PLATAFORMA ANDROID**

Urutaí
2022

ADRILEY SAMUEL RIBEIRO BARBOSA

ANÁLISE COMPARATIVA ENTRE OS PADRÕES MVC, MVP, MVVM E MVI NA PLATAFORMA ANDROID

Trabalho de Curso apresentado ao curso de Bacharelado em Sistemas de Informação do Instituto Federal Goiano – Campus Urutaí, como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Dr. Júnio César de Lima

Urutaí
2022

Sistema desenvolvido pelo ICMC/USP
Dados Internacionais de Catalogação na Publicação (CIP)
Sistema Integrado de Bibliotecas - Instituto Federal Goiano

B238a Barbosa, Adriley Samuel Ribeiro
 Análise Comparativa Entre os Padrões MVC, MVP,
 MVVM E MVI na Plataforma Android / Adriley Samuel
 Ribeiro Barbosa; orientador JÚNIO CÉSAR LIMA. --
 Urutaí, 2022.
 54 p.

 TCC (Graduação em Sistemas de Informação) --
 Instituto Federal Goiano, Campus Urutaí, 2022.

 1. Android,. 2. Sann. 3. Software architecture.
 4. Mvc. 5. Mvp. I. LIMA, JÚNIO CÉSAR, orient. II.
 Título.

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR PRODUÇÕES TÉCNICO-CIENTÍFICAS NO REPOSITÓRIO INSTITUCIONAL DO IF GOIANO

Com base no disposto na Lei Federal nº 9.610, de 19 de fevereiro de 1998, AUTORIZO o Instituto Federal de Educação, Ciência e Tecnologia Goiano a disponibilizar gratuitamente o documento em formato digital no Repositório Institucional do IF Goiano (RIIF Goiano), sem ressarcimento de direitos autorais, conforme permissão assinada abaixo, para fins de leitura, download e impressão, a título de divulgação da produção técnico-científica no IF Goiano.

IDENTIFICAÇÃO DA PRODUÇÃO TÉCNICO-CIENTÍFICA

Tese (doutorado)

Dissertação (mestrado)

Monografia (especialização)

TCC (graduação)

Artigo científico

Capítulo de livro

Livro

Trabalho apresentado em evento

Produto técnico e educacional - Tipo:

Nome completo do autor:

Matrícula:

Título do trabalho:

RESTRIÇÕES DE ACESSO AO DOCUMENTO

Documento confidencial: Não Sim, justifique:

Informe a data que poderá ser disponibilizado no RIIF Goiano: / /

O documento está sujeito a registro de patente? Sim Não

O documento pode vir a ser publicado como livro? Sim Não

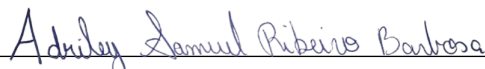
DECLARAÇÃO DE DISTRIBUIÇÃO NÃO-EXCLUSIVA

O(a) referido(a) autor(a) declara:

- Que o documento é seu trabalho original, detém os direitos autorais da produção técnico-científica e não infringe os direitos de qualquer outra pessoa ou entidade;
- Que obteve autorização de quaisquer materiais inclusos no documento do qual não detém os direitos de autoria, para conceder ao Instituto Federal de Educação, Ciência e Tecnologia Goiano os direitos requeridos e que este material cujos direitos autorais são de terceiros, estão claramente identificados e reconhecidos no texto ou conteúdo do documento entregue;
- Que cumpriu quaisquer obrigações exigidas por contrato ou acordo, caso o documento entregue seja baseado em trabalho financiado ou apoiado por outra instituição que não o Instituto Federal de Educação, Ciência e Tecnologia Goiano.


Local

/ /
Data



Assinatura do autor e/ou detentor dos direitos autorais

Ciente e de acordo:


Assinatura do(a) orientador(a)

ADRILEY SAMUEL RIBEIRO BARBOSA

**ANÁLISE COMPARATIVA ENTRE OS PADRÕES MVC,
MVP, MVVM E MVI NA PLATAFORMA ANDROID**

Monografia, defendida por Adriley Samuel Ribeiro Barbosa, apresentado ao Instituto Federal de Educação, Ciência e Tecnologia Goiano, como parte das exigências para a obtenção do título de Bacharel em Sistemas de Informação, aprovados pela banca examinadora.

COMISSÃO EXAMINADORA



Prof. Dr. Júnio César de Lima
Orientador



Profa. Dra. Mônica Sakuray Pais
Avaliadora



Profa. Dra. Nattane Luiza da Costa
Avaliadora

Urutaí (GO), 13 de julho de 2022.

AGRADECIMENTOS

Agradeço a Deus, pela força e perseverança para que os desafios fossem superados.

A todos colegas de classe e professores, que de alguma maneira, ajudaram ao longo de todos
esses anos.

E a todos que, direta ou indiretamente, contribuíram para a conclusão deste trabalho.

“Ninguém ignora tudo. Ninguém sabe tudo. Todos nós sabemos alguma coisa. Todos nós ignoramos alguma coisa. Por isso aprendemos sempre.”

Paulo Freire

RESUMO

O Android é um sistema operacional que está em crescente evolução. Por esse motivo pesquisas nesta plataforma tem sido de interesse da comunidade desde seu lançamento. Com a evolução das aplicações a dificuldade de desenvolver *software* de maneira eficiente e de qualidade aumentou exponencialmente. O padrão arquitetural *Model-View-ViewModel* (MVVM) é o mais popular para aplicações Android devido sua escalabilidade, porém não existe informações em abundância que mostrem que este padrão arquitetural apresente um melhor desempenho nas mais diversas situações de desenvolvimento de *software* quando comparado com os demais. O objetivo deste trabalho é comparar os padrões arquiteturais *Model-View-Controller* (MVC), *Model-View-Presenter* (MVP), MVVM e *Model-View-Interface* (MVI) com foco em desempenho. O modelo *Software Architecture Analysis Method* (SAAM) foi utilizado neste trabalho juntamente com dados coletados para definir qual é a arquitetura mais adequada para o desenvolvimento de aplicações Android. Os resultados mostraram que o MVVM apresenta uma melhor qualidade em características de escalabilidade e manutenção e tempos de performance satisfatórios quando comparados com os demais, sendo assim o melhor padrão para desenvolvimento de *software* no sistema operacional Android.

PALAVRAS-CHAVE: Android, SAAM, arquitetura de software, MVC, MVP, MVVM, MVI.

ABSTRACT

Android is an operating system that is constantly evolving. For this reason research on this platform has been of interest to the community since its launch. With the evolution of applications, the difficulty of developing software efficiently and with quality has increased exponentially. The Model-View-ViewModel (MVVM) architectural pattern is the most popular for Android applications due to its scalability, but there is not enough information to show that this architectural pattern presents a better performance in the most diverse software development situations when compared to the other ones. too much. The objective of this work is to purchase the Model-View-Controller (MVC), Model-View-Presenter (MVP), MVVM and Model-View-Interface (MVI) architectural patterns with a focus on performance. The Software Architecture Analysis Method (SAAM) model was used in this work together with collected data to define which is the most adequate architecture for the development of Android applications. The results showed that MVVM presents a better quality in terms of scalability and maintenance characteristics and satisfactory performance times when compared to the others, thus being the best standard for software development on the Android operating system.

KEY-WORDS: Android, SAAM, software architecture, MVC, MVP, MVVM, MVI.

Sumário

1	INTRODUÇÃO	11
2	REVISÃO LITERÁRIA	12
2.1	Arquitetura de <i>Software</i>	13
2.2	Qualidade de <i>Software</i>	13
2.3	<i>Software Architecture Analysis Method</i>	14
2.4	Padrões arquiteturais	15
2.4.1	MVC	15
2.4.2	MVP	17
2.4.3	MVVM	18
2.4.4	MVI	19
2.5	Arquitetura Android e seus Componentes	20
2.5.1	<i>Resources</i>	20
2.5.2	<i>Activities</i>	22
2.5.3	<i>Services</i>	24
2.5.4	<i>Intents/Filters</i>	25
2.5.5	<i>Broadcast Receivers</i>	26
2.5.6	<i>Content Providers</i>	27
3	MATERIAIS E MÉTODOS	27
3.1	Fase de Implementação	29
3.1.1	Casos de Uso	29
3.2	Fase de Avaliação	29
3.3	Implementação	32
3.3.1	Avaliação de Desempenho	32
3.3.2	Análise SAAM	33

4	RESULTADOS E DISCUSSÕES	34
4.1	Performance	34
4.1.1	Uso da CPU	35
4.1.2	Uso da Memória RAM	35
4.1.3	Tempo de execução	36
4.1.4	Uso de Memória RAM ao longo de cinco minutos	37
4.2	Capacidade de Manutenção e Testabilidade	38
4.2.1	Coesão	38
4.2.2	Acoplamento	39
5	CONCLUSÃO E TRABALHOS FUTUROS	41
	Referências	45
	Anexos	46
	Apêndice A: Aplicação Utilizada nos Testes	46
	Apêndice B: Resultados do Experimento	47
	Apêndice C: Estrutura de Arquivos do Projeto	48
	Apêndice D: Diagramas de Sequência	50

Lista de Figuras

1	Diagrama do <i>Model-View-Controller</i>	16
2	Diagrama do <i>Model-View-Presenter</i>	17
3	Diagrama do <i>Model-View-ViewModel</i>	18
4	Diagrama do <i>Model-View-Intent</i>	19
5	Chamada de funções no <i>Model-View-Intent</i>	20
6	Exemplo de estrutura a partir da raiz do projeto.	21
7	Ciclo de vida de uma <i>Activity</i>	23
8	Ciclo de vida de um <i>Service</i>	24
9	Comunicação entre um <i>broadcast receiver</i> e o sistema.	26
10	Comunicação entre <i>content provider</i> e aplicações Android.	27
11	Metodologia.	28
12	Aplicação utilizada nos testes - Casos de uso.	30
13	Metodologia para a coleta de dados de performance.	31
14	Quantidade de arquivos presentes em cada projeto.	32
15	Resultado uso de CPU em porcentagem.	35
16	Resultado uso de memória RAM.	36
17	Resultado dos tempos de execução.	37
18	Consumo de memória RAM em <i>megabytes</i> com o uso contínuo ao longo de 5 minutos.	38
19	Aplicação utilizada nos testes - Em execução padrão MVVM.	46
20	MVC - Estrutura de arquivos do projeto.	48
21	MVP - Estrutura de arquivos do projeto.	48
22	MVI - Estrutura de arquivos do projeto.	49
23	MVVM - Estrutura de arquivos do projeto.	49
24	MVC Diagrama de Sequência.	50

25	MVP Diagrama de Sequência.	51
26	MVI Diagrama de Sequência.	52
27	MVVM Diagrama de Sequência.	53

Lista de Tabelas

1	Tabela Coesão.	34
2	Tabela Acoplamento.	34
3	Tabela consumo de memória RAM em <i>megabytes</i> com o uso contínuo ao longo de 5 minutos.	38
4	MVC - Nível de Acoplamento	39
5	MVP - Nível de Acoplamento	40
6	MVVM - Nível de Acoplamento	40
7	MVI - Nível de Acoplamento	40
8	Resultados dos tempos de execução em milissegundos.	47
9	Resultados do uso de CPU em porcentagem.	47
10	Resultados do uso de memória RAM em <i>megabytes</i>	47

Lista de Abreviaturas

API *Application Programming Interface*

AVD *Android Virtual Device*

HTTP *Hypertext Transfer Protocol*

IEEE *Standard Glossary of Software Engineering Terminology*

MVC *Model-View-Controller*

MVI *Model-View-Intent*

MVP *Model-View-Presenter*

MVVM *Model-View-ViewModel*

RAM *Random Access Memory*

REST *Representational State Transfer*

SAAM *Software Architecture Analysis Method*

SOLID Acrônimo para cinco postulados de *design*

XML *eXtensible Markup Language*

1 INTRODUÇÃO

O Android (ANDROID, 2021) teve seu lançamento oficial em 2008 e com passar dos anos veio a se tornar o sistema operacional para dispositivos móbil de grande relevância mundial. Atualmente com mais de 2,8 milhões de aplicativos disponíveis (APPBRAIN, 2021), e com mais de 2,5 bilhões de usuários ativos (QUEIROZ, 2019) o Android conta com cerca de 72,8% dos dispositivos móbil da atualidade utilizando-o (STATCOUNTER, 2021), tornando-se o sistema operacional móbil mais utilizado do mundo.

Atualmente dentre as aplicações disponíveis na plataforma cerca de 13% são categorizadas como de baixa qualidade para a plataforma (APPBRAIN, 2021). Dentre os problemas presentes nessas aplicações muitos estão relacionados a incompatibilidade em diferentes dispositivos, interfaces fora dos padrões e principalmente problemas de performance.

Há diversos cenários que podem contribuir para um desempenho ruim em uma aplicação Android, dentre eles a escolha da arquitetura. Uma arquitetura inadequada para o contexto do problema pode resultar em alocação de recursos de maneira desorganizada ou inadequada, podendo gerar instabilidades no sistema, transições não suaves e consumo deliberado de recursos do sistema (GOOGLE, 2021).

A Google, empresa proprietária do Android não definiu uma arquitetura como padrão para o desenvolvimento de aplicações neste. Com essa liberdade dada aos desenvolvedores na plataforma surgiu-se diversos padrões de software para a criação de aplicações para o Android. Tal liberdade pode ser vista como algo paradoxal, existe a liberdade para escolher a arquitetura porém escolhas erradas podem gerar problemas de performance, testabilidade e aumento de código.

Adotar uma boa qualidade de software é de suma importância. Entre os principais benefícios de aderir a uma boa qualidade de software está a redução de custos gerados por falhas ou mal desempenho do software. Antecipar e prevenir possíveis falhas garante uma melhor resposta a situações críticas e diminuiu possíveis impactos posteriores (SINGH; GAUTAM, 2016).

Devido a necessidade de se estabelecer uma estrutura durante o desenvolvimento, inicialmente foram utilizados padrões de arquitetura que já eram utilizados na linguagem Java (ORACLE, 2021) no Android, como o *Model-View-Controller* (MVC) que tem como objetivo

separar a interface de usuário da lógica de negócio. A medida que novas necessidades específicas à plataforma surgiram, foram criadas versões customizadas deste para atendê-las. Veio assim o *Model-View-Presenter* (MVP) que tinha como intuito facilitar a execução de testes, e *Model-View-ViewModel* (MVVM) que tem como foco o menor acoplamento e a possibilidade de utilizar a mesma lógica de negócio em diferentes sistemas (NUNES, 2017). Posteriormente surgiu *Model-View-Interface* (MVI) que adicionou os conceitos de "intenção" e "estado" (SONI, 2018). Estes, sendo considerados como mais significativos para o desenvolvimento no Android .

O presente trabalho tem como objetivo demonstrar através dos quesitos aqui analisados, qual é a melhor arquitetura para o desenvolvimento de aplicações na plataforma Android. Dessa forma, será estudado a estrutura dos padrões MVC, MVP, MVVM e MVI. Além da arquitetura Android e seus componentes para posteriormente aplicar os padrões para a obtenção dos dados e efetuar uma análise comparativa destes.

O estudo dos padrões, bem como da arquitetura Android e seus componentes será utilizado para a estruturação de uma aplicação para mensuração dos dados de performance gerados, estes referentes a cada padrão e a comparação entre estes. Com os dados obtidos a partir deste trabalho os desenvolvedores poderão ter o conhecimento melhor acerca dos padrões arquiteturais e escolher melhor de acordo com o contexto do problema a ser solucionado.

Este trabalho está estruturado em 5 seções. A atual descreve a introdução, categorizando a seção 1. A seção 2 faz uma revisão em relação a conceitos importantes no contexto de desenvolvimento de software e arquitetura Android. A seção 3 descreve a metodologia utilizada na análise dos padrões arquiteturais. A seção 4 apresenta os resultados dos testes efetuados e discorre acerca destes. A seção 5 aborda as considerações finais e trabalhos futuros.

2 REVISÃO LITERÁRIA

Nesta seção é feita uma síntese de conceitos pertinentes a desenvolvimento de software relacionados ao tema do trabalho proposto. A princípio é concebida uma percepção acerca do conceito de arquitetura de software, para posteriormente discorrer sobre qualidade de software é análise de qualidade de software. Também é abordado definições importantes dentro da arquitetura Android, explanando acerca de como o Android organiza recursos, gerencia eventos de interface de usuário, lida com processamento em segundo plano e gerenciamento de dados.

2.1 Arquitetura de *Software*

A arquitetura de software de um programa ou sistema de computação é a estrutura ou estruturas do sistema, que compreende os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles (GARLAN, 2012).

Ao longo da última década, o projeto arquitetônico emergiu como um importante sub-campo da engenharia de software. Os profissionais perceberam que ter um bom projeto de arquitetura é um fator crítico para o sucesso de desenvolvimento de sistemas complexos (DAVID, 2010).

Uma boa arquitetura de software fornece uma abstração intelectualmente apreensível de um sistema complexo. Esta abstração oferece vários benefícios para a análise do comportamento do sistema antes deste ser construído, assim como reutilização de elementos e decisões arquitetônicas individuais, decisões de projeto que afetam o desenvolvimento, comunicação entre partes interessadas e gestão de riscos.

Uma arquitetura inapropriada pode resultar em estruturas de implementação que afetam negativamente as propriedades do ciclo de vida do sistema, assim como compreensibilidade, testabilidade e extensibilidade. Uma arquitetura ineficaz pode resultar em impactos negativos durante o desenvolvimento do software e dificuldades em sua manutenção (GARCIA et al., 2014).

Um projeto de arquitetura de software deve estar em conformidade com os principais requisitos relacionados à funcionalidade do software, bem como satisfazer requisitos não funcionais, como: desempenho, confiabilidade, portabilidade, escalabilidade e interoperabilidade.

2.2 Qualidade de *Software*

O *Standard Glossary of Software Engineering Terminology* (IEEE) define qualidade de software como a composição de características e atributos para o *software* atender aos requisitos das partes interessadas (LAND, 1997). No requisito de engenharia de *software*, a qualidade do *software* também é considerada um requisito não funcional. A qualidade do *software* pode ser dividida em duas categorias: requisitos de desenvolvimento e requisito operacional. Os requisitos de desenvolvimento são da perspectiva do desenvolvedor e que deve ser fácil compreensão para as atividades de desenvolvimento, por exemplo manutenção e compreensibilidade. Em-

bora os requisitos operacionais sejam mais importantes para usuários, por exemplo usabilidade e desempenho.

A qualidade do *software* é influenciada por vários fatores, dos quais a arquitetura de *software* é o mais importante. Medvidovic e Taylor afirmam que a boa qualidade do *software* está diretamente relacionada ao bom design da arquitetura de *software* (MEDVIDOVIC; TAYLOR, 2010). Chen descreve a qualidade de *software* como requisitos arquitetônicos significativos, o que implica que o arquitetura tem um alto impacto na qualidade do software (CHEN; BABAR; NUSEIBEH, 2012).

2.3 *Software Architecture Analysis Method*

O *Software Architecture Analysis Method* (SAAM) é um método usado em arquitetura de *software* para avaliar a qualidade de um sistema. As avaliações de arquitetura podem ser realizadas em um ou mais estágios do processo de desenvolvimento de software. A avaliação tem como finalidade comparar e identificar pontos fortes e fracos em arquiteturas diferentes, e possíveis alternativas dentre as possibilidades de arquitetura para o sistema durante os estágios iniciais do *design* (MATTSSON; GRAHN; MÅRTENSSON, 2006).

O modelo SAAM se concentra em métodos de avaliação de arquitetura de software que abordam um ou mais atributos referentes a qualidade de software. De acordo com o padrão IEEE 610.12-1990 (CHANDRASEKAR; RAJESH; RAJESH, 2014) são citados quatro atributos de qualidade de software, sendo eles:

- Capacidade de manutenção: Facilidade com que um sistema ou componente pode ser modificado para corrigir falhas, melhorar o desempenho ou outros atributos. A capacidade de manutenção incorpora aspectos como legibilidade e compreensão do código-fonte. A capacidade de manutenção também se preocupa com a testabilidade até certo ponto, pois o sistema deve ser revalidado durante a manutenção;
- Desempenho: Definido como o grau em que um sistema ou componente realiza suas funções designadas dentro de determinadas restrições, como velocidade, precisão ou uso de memória. Existem muitos aspectos de desempenho, por exemplo, latência, rendimento e capacidade;

- Testabilidade: A testabilidade é o quanto que um sistema ou componente facilita o estabelecimento de critérios de teste e a realização destes para determinar se esses critérios foram atendidos. Pode-se ser interpretado como o esforço necessário para validar o sistema em relação aos requisitos. Um sistema com alta testabilidade pode ser validado rapidamente.
- Portabilidade: Facilidade com que um sistema ou componente pode ser transferido de um *hardware* ou *software* para outro ambiente;

O modelo SAAM propõem uma abordagem assertiva para a avaliação da qualidade de *software*. Soluções como SAAM propõem métricas para solução problemas recorrentes no contexto de desenvolvimento de *software*, essas recomendações devem se necessário ser adaptadas de acordo com o contexto e utilizadas no domínio em questão do problema (MEIAPPANE; CHITHRA; VENKATAESAN, 2013).

2.4 Padrões arquiteturais

Os padrões de arquitetura são planos com um conjunto de regras a seguir. Esses padrões evoluíram por meio dos erros cometidos durante a codificação ao longo dos anos. Soluções generalistas para problemas recorrentes criadas com o propósito de tornar menos árdua a tarefa de criação de software escalável e de qualidade. Padrões trazem diversos benefícios quando são aplicados adequadamente ao contexto em questão, uma boa avaliação garante que todos os aspectos sejam mensurados cuidadosamente de forma a melhor suprir as necessidades específicas do problema (MARTIN, 2010).

Atualmente existem sete padrões que são comumente utilizados no mercado, sendo eles: MVC, MVP, MVVM, CLEAN, VIPER, REDUX e MVI. Dentre os padrões citados o MVVM, VIPER e MVI são padrões utilizados em sistemas móbile. Os demais também são utilizados para o desenvolvimento Android, porém estes são adaptações ao ecossistema Android. Dentre os citados os mais utilizados no desenvolvimento de aplicações Android são o MVC, MVP, MVVM e o MVI (VARADI, 2018), estes serão abordados nas próximas subseções.

2.4.1 MVC

Model-View-Controller (MVC) é um padrão de projeto de *software* criado na década de 1970 por Trygve Reenskaug, este focado no reuso de código e na separação de conceitos

em três camadas interconectadas onde cada é responsável por uma característica específica. Uma camada responsável pela apresentação dos dados, sendo esta a que o usuário interage com o sistema, uma para a modelagem dos dados e outra para a manipulação desses dados (IVANOVICH et al., 2019).

MVC é um padrão de apresentação da interface do usuário que se concentra em separar a interface de usuário (*View*) de sua camada de negócios (*Model*). Para isso, o MVC define três componentes, veja Figura 2.4.1. O MVC foi projetado para ser utilizado em aplicações focadas na interação com o usuário, aplicações voltadas para interfaces gráficas. No MVC o *Controller* é baseado em comportamentos e pode ser compartilhado entre múltiplas *Views*, tendo menor acoplamento, rápido reaproveitamento de código e uma melhor separação de interesses.

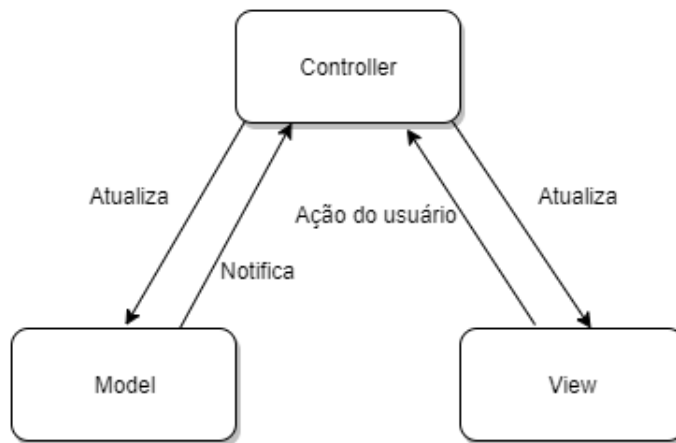


Figura 1: Diagrama do *Model-View-Controller*. Fonte: (KEARNEY, 2019).

O *Controller* é a ponte entre as camadas *View* e *Model*. Encarregado de toda a parte lógica da aplicação este gerencia o comportamento dos dados através de regras de negócios, lógica e funções. Quando uma requisição é feita uma função correspondente processa e retorna as informações requisitadas. O *Model* só tem conhecimento de suas operações e das requisições, garantindo um melhor encapsulamento e uma modularização eficiente.

A *View* é a parte de representação de dados, sendo esta qualquer saída de dados, como uma tabela, diagrama ou uma simples lista. Todos os dados solicitados no *Model* são exibidos na *View*, e esta também provém interações com o usuário que são repassadas para o *Controller*. O *Controller* é responsável pela mediação da entrada e saída dos dados, gerenciando a *View* e o *Model* de acordo com a demanda do sistema. Cada ação do usuário resulta em chamada do *Controller* para a *View* ou *Model* de acordo com a necessidade.

O modelo MVC tem diversas vantagens que o tornam um padrão de desenvolvimento de *software* eficaz. Por sua ideologia simples de compreender este acaba por ser um padrão fácil de manter, testar e atualizar, principalmente em sistemas compostos. A aplicação se torna escalável, facilita o reuso de código, melhor nível de sustentabilidade e graças a sua arquitetura modular permite que partes distintas do sistema sejam criadas simultaneamente durante sua criação.

A medida que o tamanho e a complexidade do projeto crescem o MVC não é considerada uma boa opção. A complexidade do projeto cresce, a quantidade de arquivos e pastas continuará aumentando também e esse aumento pode dificultar a manutenção em sistemas de grande porte, o que acaba por tornar o MVC um padrão de projeto mais interessante em sistemas de pequeno e médio porte (DAOUDI et al., 2019).

2.4.2 MVP

Model-View-Presenter (MVP) criado em 1996 pela Taligent e foi projetado com o objetivo de facilitar o teste de unidade automatizado e melhorar a separação de interesses (POTEL, 1996). MVP também é um padrão de apresentação da interface do usuário e é considerado por muitos uma evolução dos conceitos do MVC. O MVP tem como base três camadas e separa as responsabilidades em quatro componentes, veja Figura 2.4.2.

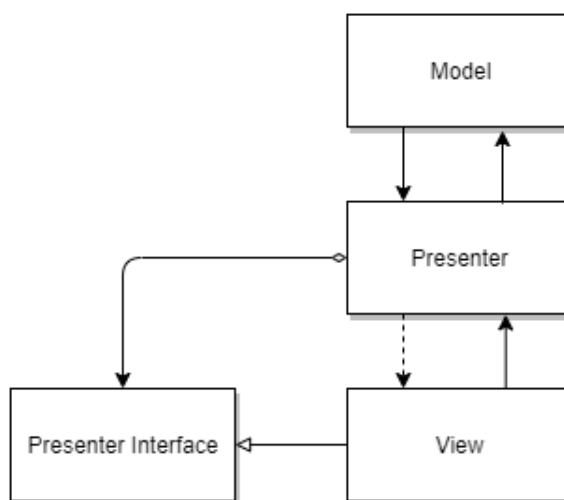


Figura 2: Diagrama do *Model-View-Presenter*. Fonte: (CHEREDNICHENKO, 2021).

View é responsável por apresentar os elementos da interface de usuário. É uma interface passiva que exibe os dados entregues pelo *Model* e capta as ações do usuário e as envia para

o *Presenter*. *Presenter* responde as ações da interface de usuário controlando a interação entre a *View* e o *Model*. *Presenter (Interface)* é a interface que define os eventos que o *Presenter* poderá responder servindo unicamente para desacoplar a *View* do *Presenter*. Além disso, o *Model* é responsável pelos comportamentos de negócio e gerenciamento de estado (CESAR, 2019).

A interface do *Presenter* garante que seja fácil realizar testes na *View*. A separação de interesses garante um gerenciamento eficaz dos dados e da interface de usuário, bem como esta interação com os dados. Implementações MVP tem uma melhor modularização do sistema facilitando sua manutenção no geral (POTEL, 1996).

2.4.3 MVVM

Model-View-ViewModel (MVVM) foi criado em 2005 por Ken Cooper e Ted Peters (SUN; CHEN; YU, 2017) tendo como objetivo simplificar a programação orientada a eventos de interface de usuário (SANTANA et al., 2015).

O padrão MVVM é composto por três elementos conforme mostra a Figura 2.4.3. *Model* é a lógica de negócios que impulsiona a aplicação e quaisquer objetos de negócios. *View* é a interface do usuário, ela inclui qualquer código relacionado a interface do usuário. *ViewModel* a camada que age como integradora em aplicações MVVM. A camada *ViewModel* coordena as operações entre a *View* e a camada *Model*. Uma camada *ViewModel* irá conter propriedades que a *View* vai obter ou definir, e funções para cada operação que pode ser feita pelo usuário em cada *View*. A camada *ViewModel* também evocará operações sobre a camada *Model*, se necessário.

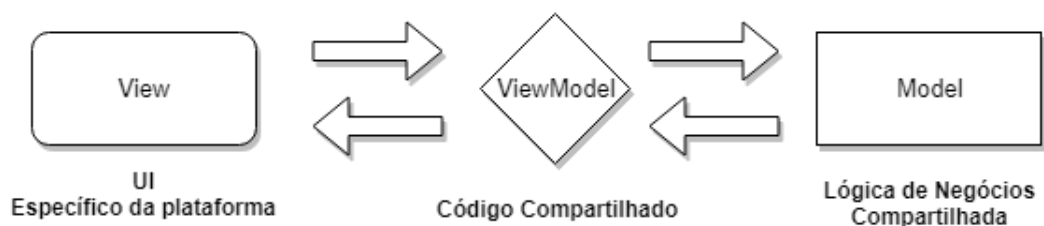


Figura 3: Diagrama do *Model-View-ViewModel*. Fonte: (MACORATTI, 2011).

Esta arquitetura em si é otimizada para testes de unidade, bem como para o desenvolvimento multiplataforma. As classes *ViewModel* de um aplicativo não têm dependências sobre a camada de interface do usuário, isto garante interoperabilidade entre sistemas distintos com alterações mínimas na estrutura do código (MUNAWAR; WAHYU, 2020).

2.4.4 MVI

Model-View-Intent (MVI) foi criado em 2014 por André Staltz (SONI, 2018), sendo amplamente aceito pela comunidade de desenvolvedores devido a sua premissa. Em meados de 2017 esse padrão de arquitetura recebeu cada vez mais atenção entre os desenvolvedores Android (DABROWSKI, 2019). É semelhante a outros padrões comumente conhecidos como MVP ou MVVM, mas introduz dois novos conceitos: a intenção e o estado.

O padrão MVI é separado em três camadas: *Model*, *View*, *Intent*. A camada *Intent* é responsável por capturar a entrada do usuário (ou seja, eventos de interface de usuário, como eventos de clique) e a traduz em algo que será passado como um parâmetro para o *Model*, pode ser valores, ações ou comandos. O *Model* recebe o retorno do *Intent* como entrada para manipular seus respectivos dados, onde o retorno então é um novo *Model*, este com seu estado alterado. O *View* pega o *Model* alterado e em seguida exibe este para a interface de usuário.

A Figura 2.4.4 mostra o diagrama correspondente a arquitetura, onde o a mesma segue um ciclo de vida natureza unidirecional e cíclico. Quando o usuário interage com o sistema é chamada uma *Intent*, esta designa o *Model* para executar a operação e o mesmo atualiza a *View* após a solicitação ser concluída e então é retornado o novo estado do sistema ao usuário.

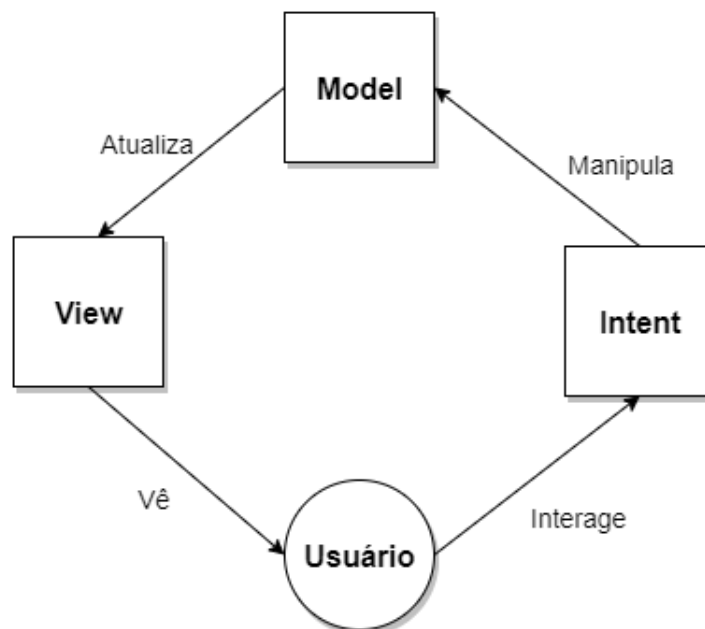


Figura 4: Diagrama do *Model-View-Intent*. Fonte: (GALOS, 2021).

O *Intent* é um evento enviado ao *Model* para realizar uma tarefa específica. Ele pode ser

acionado pelo usuário ou por outras partes do seu aplicativo. Como resultado disso, um novo estado é definido no *Model* que por sua vez atualiza a interface do usuário. Na arquitetura MVI, o *View* escuta o estado. Cada vez que o estado muda, a *View* é notificada. Na Figura 2.4.4 é mostrado como ocorre a chamada dos métodos, esta em cascata, onde o retorno de um método dispara a chamada do próximo subsequente.

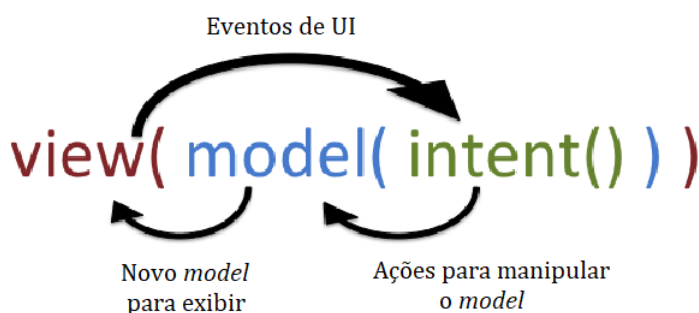


Figura 5: Chamada de funções no *Model-View-Intent*. Fonte: (MUNAWAR, 2014).

2.5 Arquitetura Android e seus Componentes

A arquitetura do sistema operacional Android em sua essência é completamente modular, isso garante uma melhor separação de interesses, fazendo com que o mínimo de recurso seja utilizado para a execução de cada tarefa, e que cada parte seja responsável por um determinado conjunto de encargos correlatos. Por mais modularizada que seja a arquitetura, a mesma é altamente acoplável, os mais diversos componentes interagem em conjunto para a execução de uma tarefa, todos requisitados de acordo com a demanda do sistema, dada a situação.

Nas subseções subsequentes serão abordados os principais componentes da arquitetura Android e conceitos relacionados a estes, discorrendo acerca dos componentes que compõem a estrutura do sistema e seu funcionamento de maneira geral.

2.5.1 Resources

Resources são os arquivos adicionais utilizados pelo código da aplicação. Além de codificar para o aplicativo existem muitos outros itens que são utilizados para construir um aplicativo Android. Diversos *resources* são utilizados no código, como conteúdo estático, *bitmaps*, cores, definições de *layout*, textos na de interface de usuário e instruções de animação (TIWARI, 2018). Na Figura 2.5.1 é mostrado a estrutura de arquivos de uma aplicação Android. *Resources* são sempre mantidos separadamente em vários subdiretórios no diretório "res/" do projeto.

```
MyProject/  
  app/  
    manifest/  
      AndroidManifest.xml  
  java/  
    MainActivity.java  
  res/  
    drawable/  
      icon.png  
    layout/  
      activity_main.xml  
      info.xml  
    values/  
      strings.xml
```

Figura 6: Exemplo de estrutura a partir da raiz do projeto. Fonte: (MOHTASHIM, 2015).

Diferentes arquivos podem ser utilizados como *resources* durante a construção de uma aplicação Android, os utilizados com mais frequência são:

- *anim*: Arquivos XML (*eXtensible Markup Language*) que definem animações. Eles são salvos na pasta "res/anim" e são acessados pela classe "R.anim" pelas demais classes.
- *color*: Arquivos XML que definem uma lista de estado de cores. Eles são salvos em "res/color" e acessados na classe "R.color".
- *drawable*: Arquivos de imagem com o formato PNG, JPG, GIF ou arquivos XML que são compilados em *bitmaps*, listas de estado, formas, *drawable* de animação. Eles são salvos em "res/drawable/" e acessados a partir da classe "R.drawable".
- *layout*: Arquivos XML que definem um *layout* de interface do usuário. Eles são salvos em "res/layout/" e acessados a partir da classes "R.layout".
- *menu*: Arquivos XML que definem menus de aplicativos, como menu de opções, menu de contexto ou sub-menus. Eles são salvos em "res/menu" e acessados a partir da classe "R.menu".
- *raw*: Arquivos arbitrários para salvar em sua forma bruta. é necessário chamar "Resources.openRawResource()" com o número identificador do recurso, que é "R.raw.<nome do arquivo>" para abrir esses arquivos brutos. Exemplo: "R.raw.passwordsList".

- *values*: Arquivos XML que contêm valores simples, como *string*, inteiros e cores. Algumas convenções de nome de arquivo para recursos:
 - arrays.xml para matrizes de recurso e acessados pela classe "R.array".
 - integers.xml para número inteiros de recursos e acessados através da classe "R.integer".
 - bools.xml para *booleano* e acessado pela classe "R.bool".
 - colors.xml para valores de cores e acessados através da classe "R.color".
 - dimens.xml para valores de dimensão e acessados através da classe "R.dimen".
 - string.xml para valores de dimensão e acessados através da classe "R.string".
 - styles.xml para estilos e acessados através da classe "R.style".
- xml: Arquivos arbitrários que podem ser lidos em tempo de execução chamando "Resources.getXML()".

2.5.2 Activities

Uma *activity* é um componente que fornece uma tela com a qual os usuários podem interagir. Ela representa uma única tela com uma interface de usuário. Diferentes *activities* são agrupadas em um único aplicativo Android e embora as *activities* colaborem entre si cada *activity* é independente (GATTAL, 2018).

O sistema Android inicia sua execução com uma *activity* começando com uma chamada no método "onCreate()". Há uma sequência de métodos de retorno de chamada que inicia uma *activity* e uma sequência de métodos de retorno de chamada que interrompem uma *activity*, conforme mostrado no diagrama de ciclo de vida da *activity* na Figura 2.5.2.

- onCreate(): Este é o primeiro método e é chamado quando a *activity* é instanciada no sistema pela primeira vez.
- onStart(): Este retorno de chamada é chamado quando a *activity* se torna visível para o usuário.
- onResume(): Isso é chamado quando o usuário começa a interagir com o aplicativo.
- onPause(): Chamado quando a *activity* atual está sendo pausada e a *activity* anterior está sendo retomada. A *activity* pausada não recebe entrada do usuário e não pode executar nenhum código.

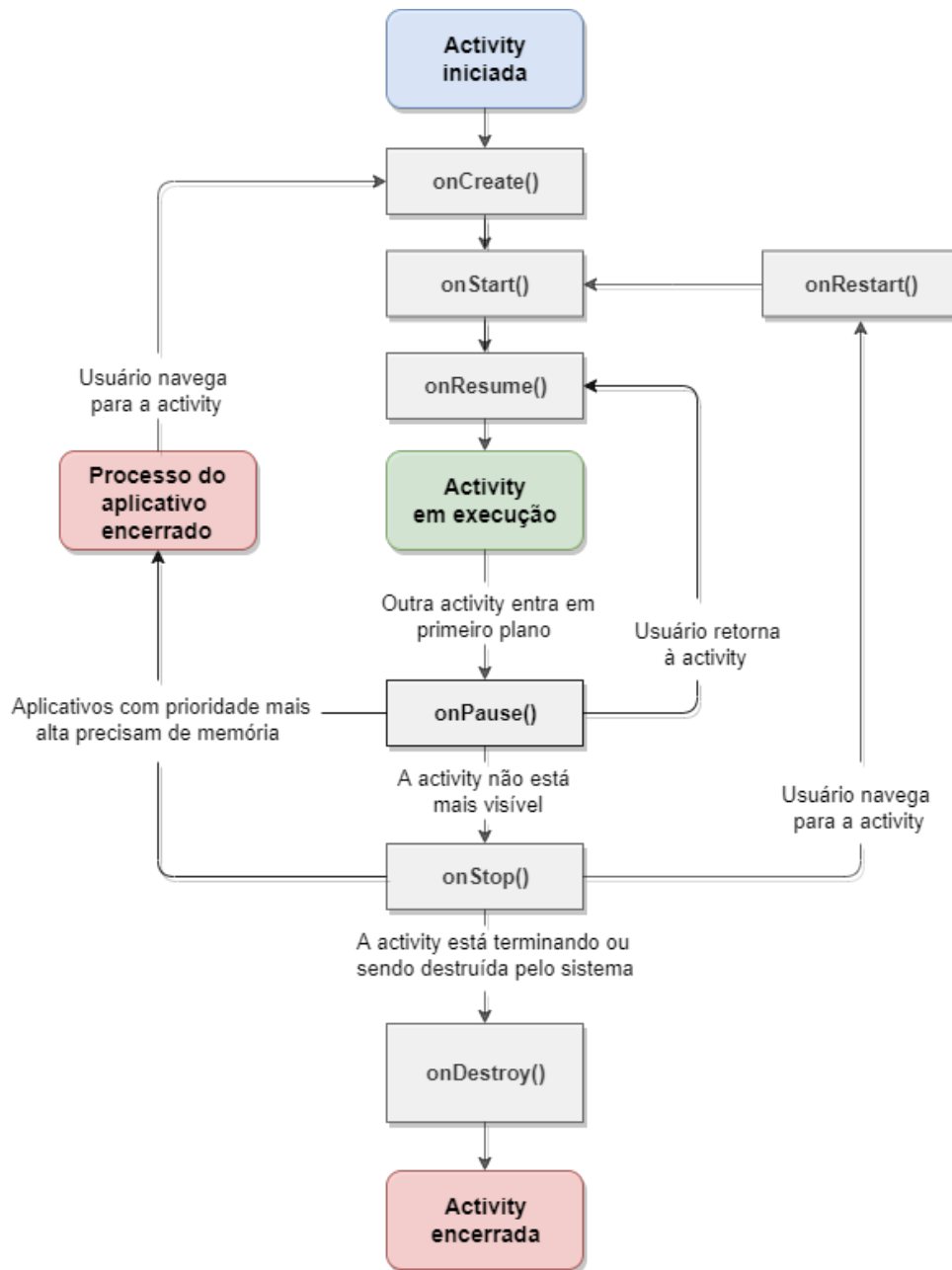


Figura 7: Ciclo de vida de uma *Activity*. Fonte: (ARACHCHI, 2018).

- `onStop()`: Este método é chamado quando a *activity* não está mais visível.
- `onDestroy()`: Este método é chamado antes que a *activity* seja destruída pelo sistema.
- `onRestart()`: Este método é chamado quando a *activity* é reiniciada após interrompê-la.

2.5.3 Services

Um *service* é um componente executado em segundo plano para realizar operações de longa duração, como reproduzir música, lidar com transações de rede ou receber dados em um servidor remoto. O *service* não possui interface de usuário e segue executando em segundo plano mesmo se o aplicativo for encerrado (MOHTASHIM, 2016). Um *service* tem seu próprio ciclo de vida assim como uma *activity* e pode essencialmente assumir dois estados, conforme representado na Figura 2.5.3.

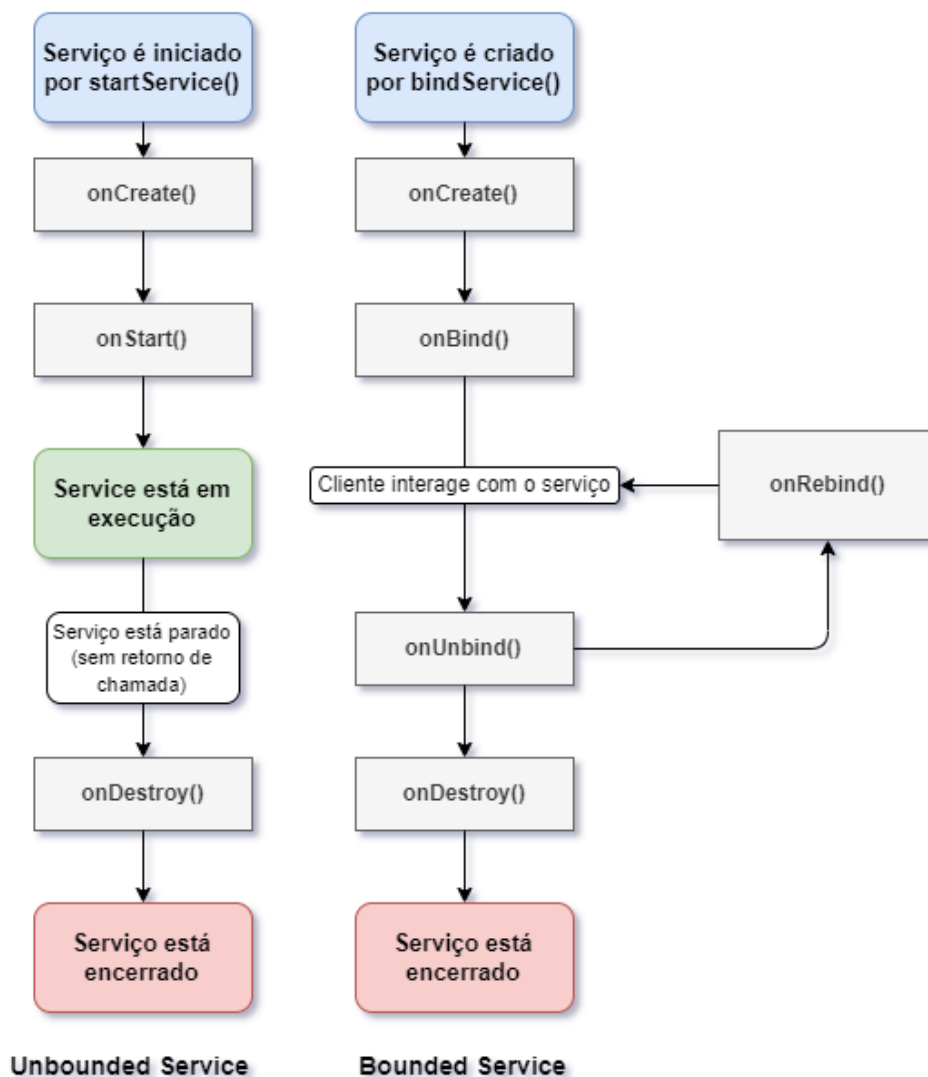


Figura 8: Ciclo de vida de um *Service*. Fonte: (MOHTASHIM, 2016).

De acordo com a Figura 2.5.3, quando um *service* é iniciado como *Unbounded*, um componente do aplicativo inicia o mesmo chamando o método "startService()". Depois de iniciado,

um *service* pode ser executado em segundo plano indefinidamente, mesmo se o componente que o iniciou for destruído. *Bounded* e quando um *service* é vinculado à um componente do aplicativo, e este se vincula a ele chamando "bindService()". Um *service* vinculado oferece uma interface cliente-servidor que permite aos componentes interagir com o *service*, enviar solicitações, obter resultados e até mesmo efetuar comunicação entre processos. Uma descrição mais detalhada correspondente a cada método mostrado na Figura 2.5.3:

- onStart(): O sistema chama esse método quando outro componente, como uma *activity*, solicita que o *service* seja iniciado.
- onBind(): O sistema chama esse método quando outro componente deseja vincular-se ao *service*.
- onUnbind(): O sistema chama esse método quando todos os clientes se desconectam de uma determinada interface publicada pelo *service*.
- onRebind(): O sistema chama este método quando novos clientes se conectam ao *service*, após ter sido previamente notificado de que todos haviam se desconectado em seu "onUnbind()".
- onCreate(): O sistema chama esse método quando o serviço é criado pela primeira vez usando "startService()" ou "bindService()".
- onDestroy(): O sistema chama esse método quando o *service* não é mais usado e está sendo destruído.

2.5.4 Intents/Filters

Uma *intent* é uma descrição abstrata de uma operação a ser executada. *Filters* e *intents* representam o mesmo conceito porém *intents* estão relacionadas a tarefas que podem ser executadas enquanto *filters* são *intents* para recepção destas requisições. *Intents* podem ser usadas com chamadas no método "startActivity()" para iniciar uma *activity*, "broadcastIntent()" para enviá-la a qualquer componente *broadcast receiver* interessado e "startService()" ou "bindService()" para se comunicar com um *service* em segundo plano.

Um objeto *intent* é uma estrutura de dados passiva que contém uma descrição abstrata de uma operação a ser executada. Por exemplo em uma requisição que tenha uma *activity*

que precise iniciar um cliente de *e-mail* e enviar um *e-mail* utilizando o dispositivo Android, a *activity* enviaria uma *intent* para o sistema junto com o informações e assim o mesmo designaria o aplicativo ou componente do sistema mais apropriado para a requisição em questão.

2.5.5 Broadcast Receivers

Os *broadcast receivers* simplesmente respondem às transmissões de mensagens de outros aplicativos ou do próprio sistema, e essas mensagens são enviadas através de *intents* por todo o sistema (GARG, 2017). Os aplicativos também podem iniciar transmissões para permitir que outros aplicativos saibam que alguns dados foram baixados para o dispositivo e estão disponíveis para uso, portanto, este é o receptor de transmissão que interceptará essa comunicação e iniciará a ação apropriada. A Figura 2.5.5 mostra como o *broadcast receiver* interage com o sistema Android. Uma *Intent* correspondente ao *broadcast receiver* é registrada, esta tem o papel de acionar o *broadcast receiver* quando uma mensagem correspondente ocorre no sistema.



Figura 9: Comunicação entre um *broadcast receiver* e o sistema. Fonte: (GARG, 2017).

Existem duas etapas importantes para fazer o *broadcast receiver* funcionar para as *intents* de transmissão do sistema: a primeira é criar o *broadcast receiver* e a segunda o registro do *broadcast receiver* no "AndroidManifest.xml". *Intens* são altamente personalizáveis e adaptáveis as mais diversas necessidades e caso necessário é possível que o desenvolvedor crie suas próprias *intents*.

2.5.6 Content Providers

Um *content provider* fornece dados de um aplicativo para outros, mediante solicitação. Solicitações são tratadas pelo *content provider* e repassadas ao seu requisitor (SHAHRIAR; HADDAD, 2014). Um *content provider* pode usar diferentes maneiras de armazenar seus dados, onde os dados podem ser armazenados em um banco de dados, em arquivos ou mesmo em uma rede. A Figura 2.5.6 mostra um diagrama exemplificando como o *content provider* faz o intermédio entre a camada de negócio e a camada de dados adicionando uma camada extra para intermediar a troca de informações.

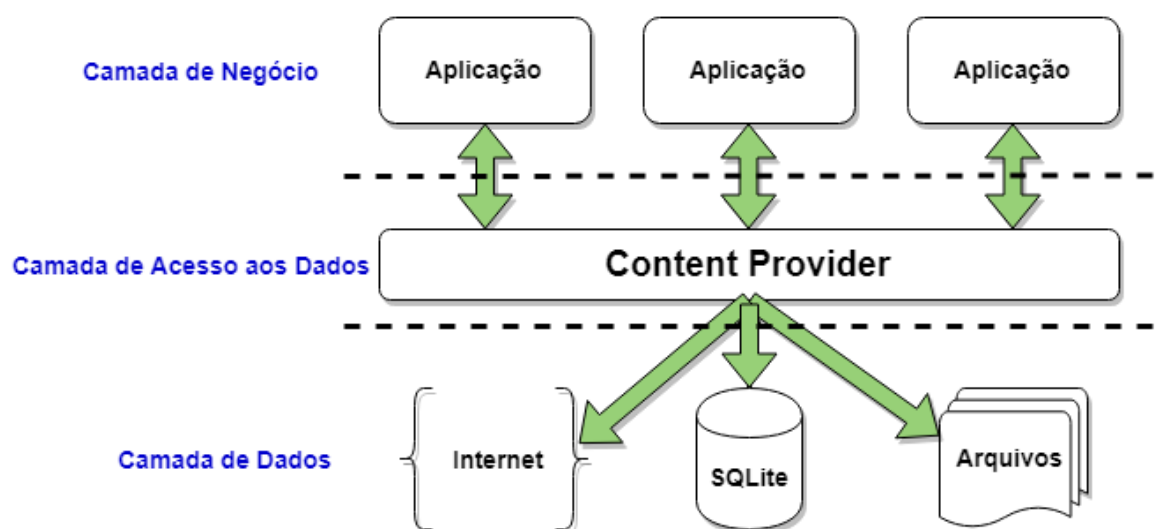


Figura 10: Comunicação entre *content provider* e aplicações Android. Fonte: (GARG, 2017).

Os *content providers* permitem que se centralize o conteúdo em um local de forma que diferentes aplicativos possam acessá-los conforme a necessidade. Um *content provider* se comporta de maneira muito semelhante a um banco de dados onde se pode consultá-lo, editar seu conteúdo, bem como adicionar ou excluir conteúdo usando os métodos inserir, alterar, remover e buscar. Na maioria dos casos, esses dados são armazenados em um banco de dados SQLite (HIPPI, 2021).

3 MATERIAIS E MÉTODOS

Inicialmente foi feito um estudo acerca da arquitetura do sistema operacional Android e os padrões MVC, MVP, MVVM e MVI. Após a contextualização inicial tem-se duas grandes fases para o desenvolvimento deste trabalho, sendo (i) fase de implementação e (ii) fase de

avaliação. Na fase de implementação foi estruturada uma aplicação para aplicação dos padrões, aqui nomeada de *Data Handler App*, a mesma aplicação com uma implementação correspondente em cada padrão. Na fase de avaliação o aplicativo que foi construído será utilizado para a coleta dos dados relacionados a performance, e avaliação das arquiteturas utilizando o modelo SAAM. Na Figura 3 uma visão geral da metodologia utilizada.

Para versionamento do código foi utilizado o sistema de controle de versões GIT (CONSERVANCY, 2021) e os projetos foram hospedados no serviço em nuvem GitHub (GITHUB, 2021). Todas as implementações aqui utilizadas encontram-se em um repositório público disponível no site GitHub (BARBOSA, 2021).

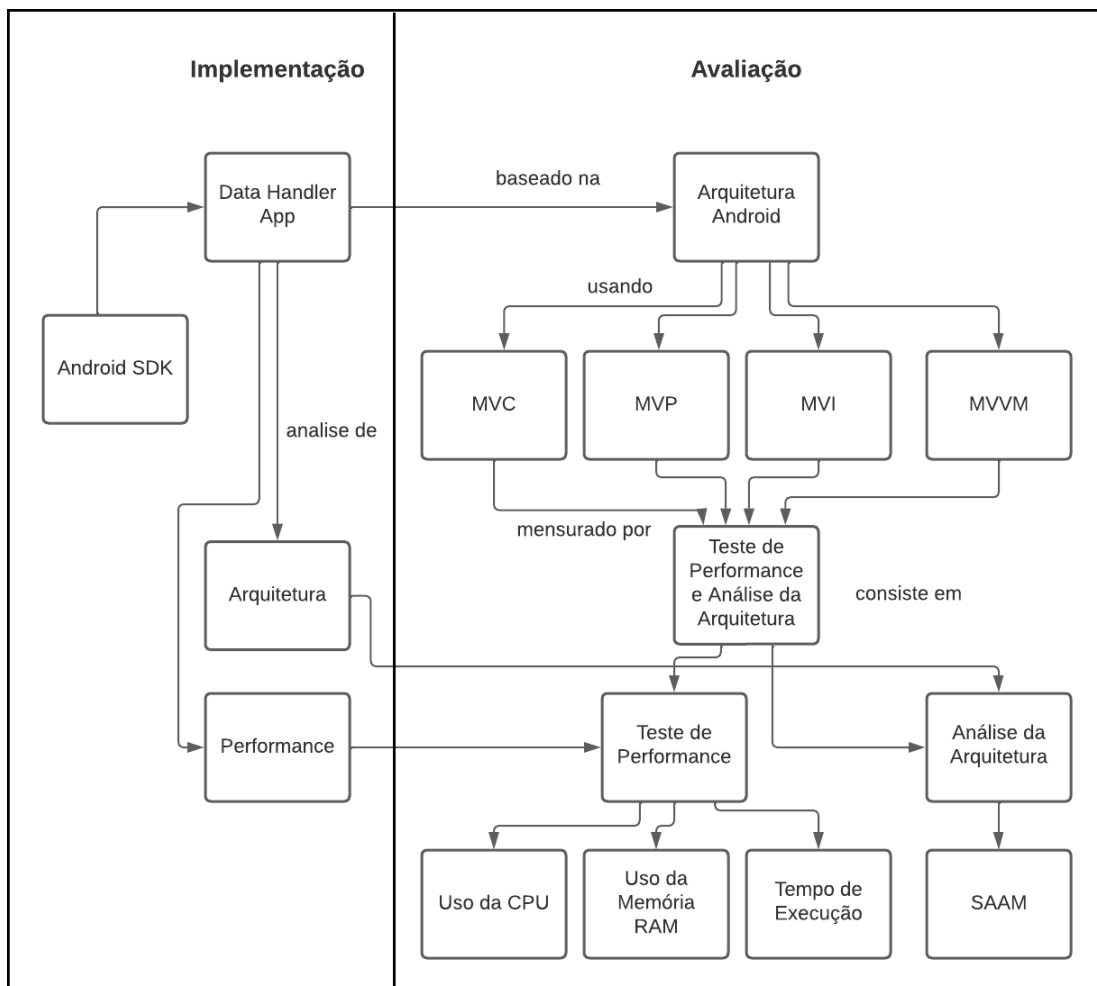


Figura 11: Metodologia. Fonte: elaborada pelo autor.

3.1 Fase de Implementação

Esta etapa tem como objetivo analisar as necessidades da aplicação, para projetar um sistema capaz de servir como meio para obtenção dos dados de performance, para posteriormente serem avaliados através das métricas estabelecidas. A aplicação utilizada para os testes conta com uma interface voltada para a manipulação de dados advindos de uma fonte local e duas fontes remotas. O uso está relacionado a manuseio de informações através da interação do usuário com as funcionalidades presentes na interface. A interface conta com botões, cada um com sua respectiva descrição indicando a devida utilidade do mesmo.

3.1.1 Casos de Uso

Os requisitos funcionais do aplicativo são modelados na forma de casos de uso conforme mostrado na Figura 3.1.1. O diagrama de casos de uso ilustra a interação entre o ator e as funcionalidades do sistema. As funcionalidades do aplicativo tem como base recursos para manipulação de dados, sendo estes: operações básicas (consultar, inserir, atualizar e excluir) em um banco de dados local, operações básicas em um banco de dados remoto e consulta em uma API (*Application Programming Interface*) esta seguindo o padrão REST (*Representational State Transfer*). Os dados manipulados são palavras e caracteres em outros idiomas.

3.2 Fase de Avaliação

Na fase de avaliação os aplicativos MVC, MVP, MVI e MVVM que foram construídos serão testados em relação ao seu desempenho seguindo as seguintes etapas experimentais:

1. Execute os aplicativos MVC, MVP, MVI e MVVM no dispositivo Android.
2. Durante a execução será monitorado o uso de CPU, uso de memória e o tempo de execução.
3. Os dados serão coletados através da ferramenta *Profiler* presente no Android Studio.
4. As medições serão feitas considerando dois cenários:
 - (a) Uso de CPU, consumo de memória RAM e tempo de execução em uma requisição à uma API REST.

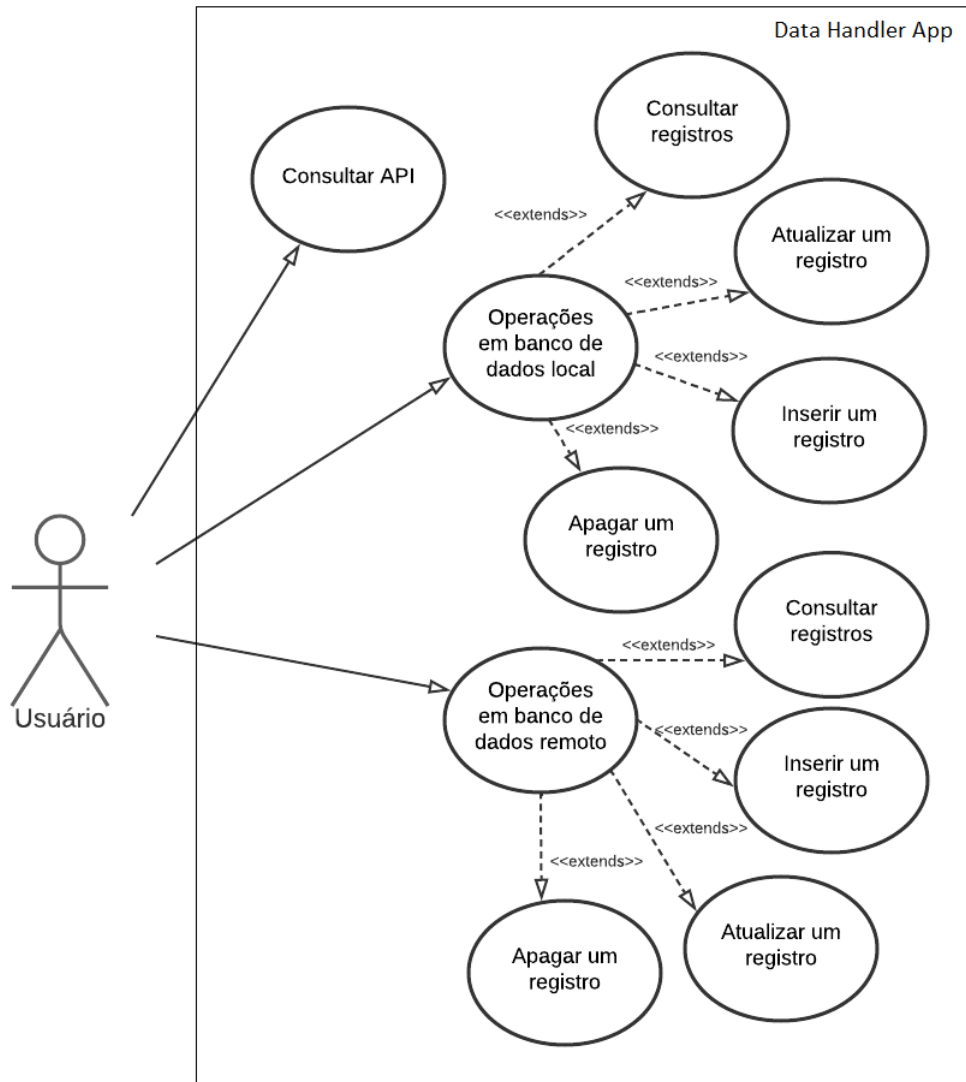


Figura 12: Aplicação utilizada nos testes - Casos de uso. Fonte: elaborada pelo autor.

- (b) Consumo de memória RAM com o uso contínuo da aplicação ao longo de cinco minutos.
5. Cada cenário será realizados cinco vezes, e será feita a mínima, média e máxima dos resultados.

Os experimentos são realizados por meio de medição de desempenho. Os dados referentes ao desempenho geral da aplicação são obtidos através da ferramenta *Profiler* presente no Android Studio, estes sendo uso de CPU em porcentagem, uso de memória em *megabytes* (mb) e o tempo de execução em milissegundos (ms). O cenário que faz uma requisição à uma API REST, esta requisição é feita à um *endpoint* que retorna 20 mil registros, estes sendo caracteres e palavras em chinês e japonês. A escolha dessa API deve-se ao fato do grande volume de da-

dos retornados, a escolha indeferiu as questões aqui analisadas ficando apenas a importância da quantidade de registros retornados, ideal para os testes que aqui foram submetidos. Na Figura 3.2 uma visão geral sobre as etapas com mais detalhes sendo executadas de maneira sequencial.

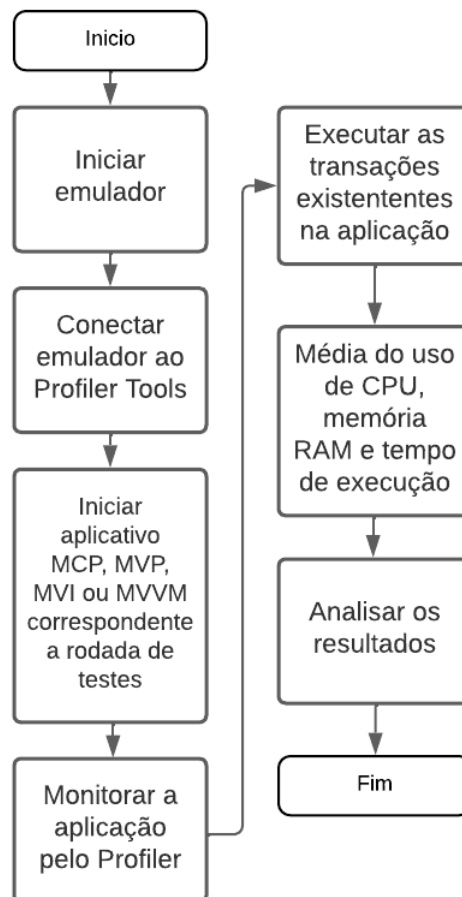


Figura 13: Metodologia para a coleta de dados de performance. Fonte: elaborada pelo autor.

Para os experimentos foi utilizado um dispositivo Android virtual criado pela ferramenta *Device Manager* presente no Android Studio. As especificações do dispositivo Android usado no experimento são as seguintes:

- Sistema Operacional: Android 10.0
- Memória RAM: 2GB
- Processador: 4 núcleos a 2.8GHz
- Armazenamento Interno: 2GB

Os dados obtidos na forma de uso de CPU, uso de memória e valores de tempo de execução em cada cenário serão comparados entre as aplicações submetidas a teste. Quanto menor o valor de uso de CPU e uso de memória, melhor, assim como tempo de execução. Com base nesses dados pode-se ter um comparativo entre as arquiteturas MVC, MVP, MVI e MVVM em termos de performance.

3.3 Implementação

O aplicativo é implementado no Android Studio IDE com a linguagem de programação Java majoritariamente e Kotlin. A aplicação implementada em Kotlin é a correspondente ao MVI, trata-se de um requisito para a arquitetura e infere apenas nas questões de implementação, não interfere no resultado da avaliação visto que o código Kotlin é transcrito para Java antes de ser compilado.

A quantidade de arquivos presentes em cada projeto mostrado na Figura 3.3. No Apêndice A apresentada uma visão do aplicativo utilizado nos testes. Já no Apêndice C é mostrada uma visão da estrutura de arquivos referente a cada projeto.

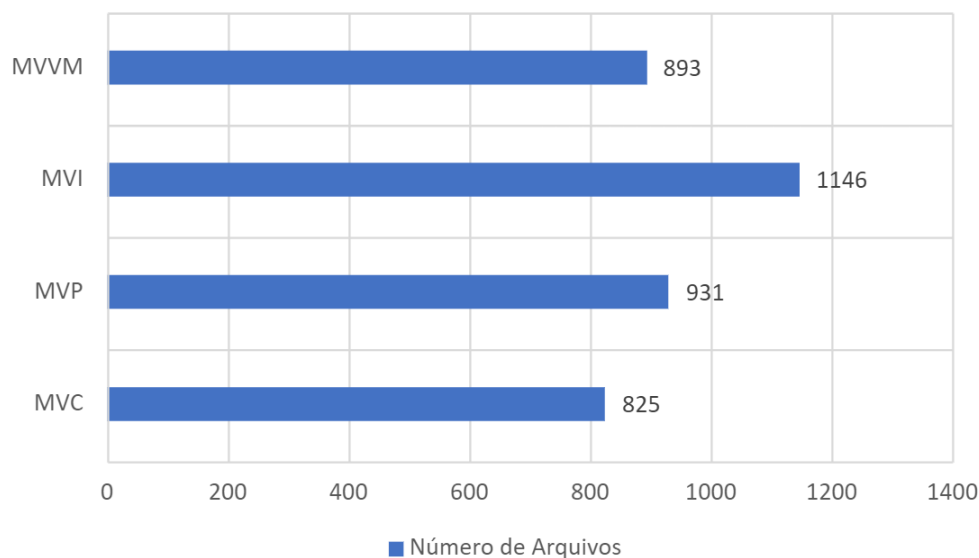


Figura 14: Quantidade de arquivos presentes em cada projeto. Fonte: elaborada pelo autor.

3.3.1 Avaliação de Desempenho

Os aplicativos foram avaliados levando em consideração os seguintes quesitos: uso de CPU, uso de memória RAM e tempo de execução. A avaliação aborda dois cenários, o pri-

meiro cenário leva em conta todos os quesitos de observação estabelecidos, onde é feita uma requisição à uma API REST que retorna um grande volume de dados (20 mil registros), e o segundo cenário avalia consumo de memória RAM em uso contínuo da aplicação ao longo de um período de cinco minutos. Os valores são o valor mínimo, médio e máximo de um conjunto de cinco amostras.

3.3.2 Análise SAAM

O modelo SAAM tem como pilar quatro princípios a serem utilizados durante uma avaliação: capacidade de manutenção; desempenho, testabilidade e portabilidade. A portabilidade não entra no âmbito desta avaliação pois trata-se de aplicações focadas em uma única plataforma alvo. Resta então o desempenho, a capacidade de modificação e a testabilidade.

Acoplamento refere-se à força das associações entre diferentes módulos. A coesão indica as relações dentro de um módulo específico. Redução do acoplamento e o reforço da coesão melhoram a modificabilidade e a testabilidade da aplicação. Sete níveis de coesão e seis níveis de acoplamento são reconhecidos, o artigo (ZHAO; ZOU, 2011) resume os níveis mostrados na Tabela 1 e na Tabela 2.

Para avaliar a capacidade de modificação e a testabilidade de cada arquitetura foi avaliado o nível de coesão e acoplamento de cada arquitetura de acordo com a descrição fornecida na Tabela 1 e na Tabela 2.

Há três categorias de coesão de acordo com a Tabela 1, são elas: baixo, moderado e alto. O nível de coesão é associado à uma categoria de acordo com suas características, componentes que tem como característica a execução de uma única tarefa são considerados "Nível alto", já tarefas que são executadas por um conjunto de componentes são consideradas "Nível baixo". O ideal é que quando mais elevado o grau de coesão melhor, ou seja "Nível alto".

Assim como na Tabela 1 a Tabela 2 segue a mesma abordagem de separação em três níveis, sendo eles: baixo, moderado e alto. Ao contrário da abordagem anterior em que quanto maior o grau melhor, nesta quanto menor o nível melhor. Um exemplo de componente considerado com um nível de acoplamento alto é quando este depende de informações de outro componente para funcionar. Já componentes de nível baixo apenas repassam suas informações à terceiros, sem que haja uma dependência direta com outro componente para seu funcionamento.

Para a análise foi avaliado os componentes em cada arquitetura com base nas descrições estabelecidas em cada característica e seu respectivo nível, e assim classificado o nível de coesão e acoplamento em cada componente.

Tabela 1: Tabela Coesão. Fonte: (ZHAO; ZOU, 2011).

Categoria	Nível de Coesão	Características
Nível alto	Funcional	Um componente executa uma única tarefa.
Nível moderado	Sequencial	Um item é transferido entre tarefas dentro de um componente.
Nível moderado	Comunicacional	Tarefas dentro de um componente compartilham dados.
Nível moderado	Procedural	Tarefas são conectadas por conectores de controle.
Nível baixo	Temporal	Tarefas são correlacionadas por relações temporárias.
Nível baixo	Lógico	Tarefas são agrupadas para executar o mesmo tipo de funcionalidade.

Tabela 2: Tabela Acoplamento. Fonte: (ZHAO; ZOU, 2011).

Categoria	Nível de Coesão	Características
Nível alto	Conteúdo	Um componente usa dados/controle de outro componente.
Nível alto	Comum	Os componentes compartilham itens de dados globais.
Nível alto	Externo	Os componentes estão vinculados a entidades externas.
Nível moderado	Controle	O controle controla o fluxo entre os componentes.
Nível baixo	Dados Estruturados	Os dados estruturados são transferidos entre os componentes.
Nível baixo	Dados	Dados primitivos ou matrizes são passados entre os componentes.
Nível baixo	Mensagem	Os componentes se comunicam por meio de interfaces.

4 RESULTADOS E DISCUSSÕES

Para a análise de performance foram mensuradas questões referentes a tempo de execução, consumo de memória RAM e consumo de CPU. Para características mais abstratas a serem avaliadas foi utilizado o modelo SAAM, foi levado em consideração os seguintes pontos: capacidade de modificação e testabilidade.

4.1 Performance

Os resultados de performance são avaliados levando como base a mínima, média e máxima de um conjunto de cinco amostras em cada arquitetura. O uso da CPU é medido em porcentagem, o tempo de execução em milissegundos (ms) e o consumo de memória RAM em *megabytes*.

4.1.1 Uso da CPU

Os resultados da medição do uso da CPU podem ser observados na Figura 4.1.1. O MVI tem o maior uso de CPU com uma máxima de 39%, seguido pelo MVC com 38%. O menor uso é equivalente nas arquiteturas MVI e MVP com 30%, a menor máxima é do MVVM com 36%. Os valores da média são bem próximos em todas as arquiteturas. O MVVM tem um melhor desempenho no quesito de uso de CPU, dado que sua máxima e média estão entre os menores valores do conjunto de amostras seguido pelo MVP. O MVI se mostrou como a arquitetura com o maior uso de CPU seguido pelo MVC.

O número de chamadas necessárias para executar uma transação é bem maior no MVI em comparação as demais arquiteturas, o que pode ser a causa para o maior pico de consumo de CPU nessa arquitetura quando comparado com as demais. O MVVM tem sua máxima de consumo de CPU menor entre as demais arquiteturas, o número de chamadas efetuado é bem inferior quando comparado ao MVI assim como mostrado nos diagramas de sequência presentes no Apêndice D.

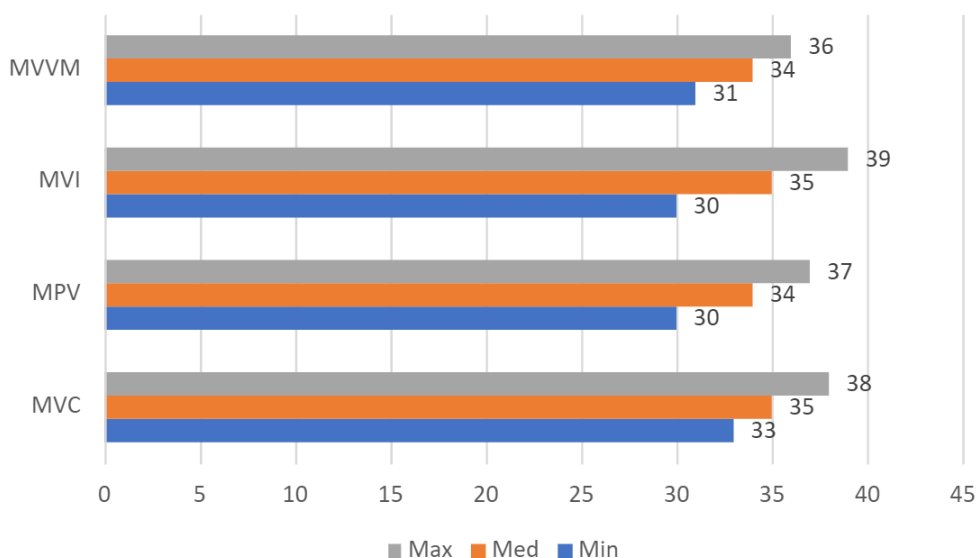


Figura 15: Resultado uso de CPU em porcentagem. Fonte: elaborada pelo autor.

4.1.2 Uso da Memória RAM

Os resultados do uso de memória RAM podem ser observados na Figura 4.1.2. O MVC tem menor uso de memória RAM com a mínima de 71,5 mb, seguido pelo MVVM com 72,3 mb. O MVVM tem a menor média com 73,3 mb, e a menor máxima com 74,8 mb. O

MVP tem a maior máxima com 83,9 mb seguido pelo MVI com 83,1 mb. O MVI teve valores de uso de memória RAM acima das demais arquiteturas. Na medição do uso de memória RAM assim como no uso de CPU o MVVM segue com melhor resultado quando comparado as demais arquiteturas, e diferentemente dos resultados do uso de CPU o MVP não teve um bom desempenho geral apresentando a maior máxima de uso de memória RAM.

Todas as arquiteturas apresentaram um consumo de memória RAM próximo em suas médias e máximas, com exceção do MVVM que teve valores inferiores que as demais em sua média e máxima. O fato do MVVM ter um menor consumo de memória RAM pode estar relacionado ao uso do *data-binding* e bibliotecas como *LiveData* que evita possíveis vazamentos de memória, em sua implementação padrão.

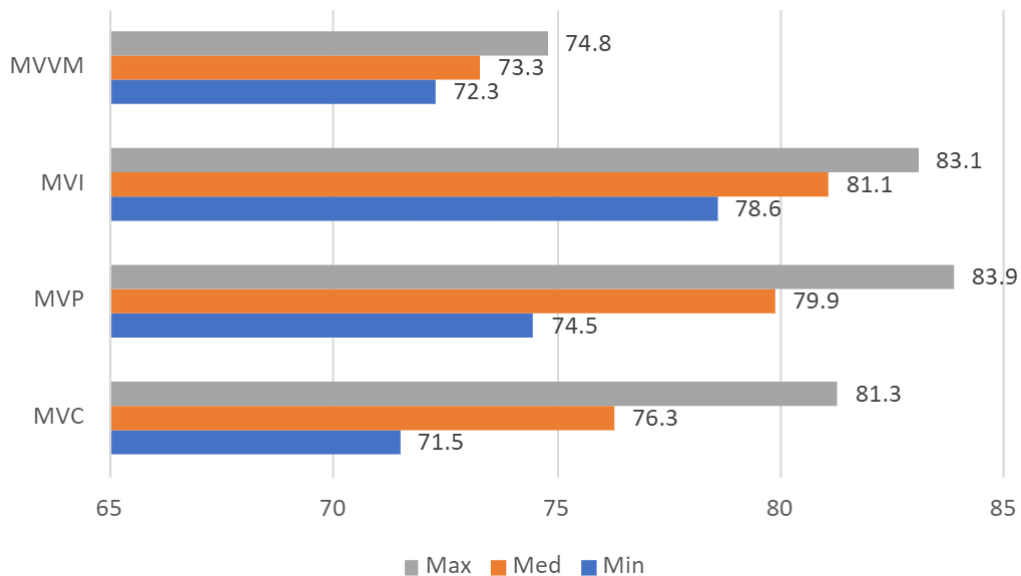


Figura 16: Resultado uso de memória RAM em *megabytes*. Fonte: elaborada pelo autor.

4.1.3 Tempo de execução

Os resultados das medições de tempo de execução são mostrados na Figura 4.1.3. O MVVM tem em seus resultados números inferiores aos demais, conta com a menor mínima, média e máxima. O MVI tem os piores resultados entre os demais, enquanto o MVP e o MVC tem resultados bem próximos, mais de maneira geral o MVP se mostrou superior ao MVC. O MVVM tem sua mínima em 1245 ms, sendo este o menor valor enquanto o MVI 2207 ms, na média MVVM com 1284 ms enquanto o MVI com 2234 ms, e na máxima a diferença de valores segue grande com MVVM com 1326 ms de tempo de execução e o MVI com 2309 ms.

O MVVM quando comparado ao MVP tem uma leve vantagem, o MVM conta com uma média de 1284 ms enquanto o MVP com 1425 ms. Os valores do MVP não se distanciam muito dos resultados obtidos no MVC.

O fato MVVM ter valores bem inferiores aos demais pode estar relacionado ao uso do *data-binding*, além do número de chamadas efetuadas para a execução de uma transação (veja Apêndice D) ser menor que as demais arquiteturas. Os valores superiores do MVI podem estar relacionados ao número maior de chamadas durante a execução de uma transação, o que pode resultar em um *delay* na resposta do sistema.

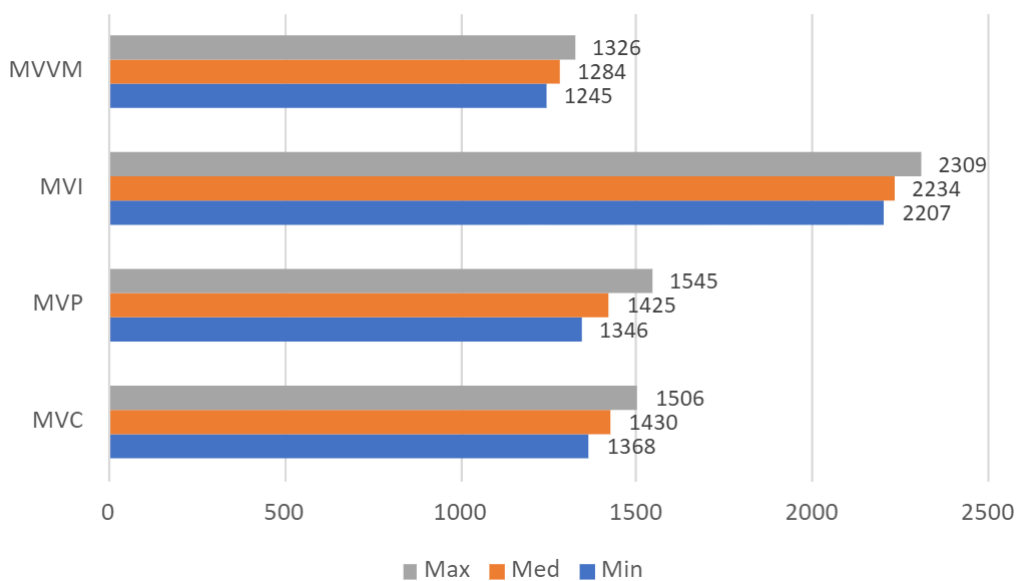


Figura 17: Resultado dos tempos de execução em milissegundos. Fonte: elaborada pelo autor.

4.1.4 Uso de Memória RAM ao longo de cinco minutos

Os resultados das medições de consumo de memória RAM ao longo de cinco minutos podem ser vistos na Figura 4.1.4 e com mais detalhes na Tabela 3. O MVI tem o maior uso contínuo de memória RAM ao longo do tempo, o MVP e o MVP seguem bem próximos, porém o MVC teve um maior consumo com o decorrer das medições, o pico inicial de consumo de memória RAM do MVC é o maior entre os demais e o MVVM tem o menor consumo no decorrer das medições.

O fato do MVI ter mostrado um consumo de memória RAM acima das demais arquiteturas está relacionado a quantidade de chamadas em cadeia que a aplicação tem que executar

com o uso contínuo da aplicação. Nas demais arquiteturas os resultados são bem próximos com exceção do MVVM que apresenta um melhor gerenciamento de memória.

Tabela 3: Tabela consumo de memória RAM em *megabytes* com o uso contínuo ao longo de 5 minutos.

	0	1	2	3	4	5
MVC	76,6	97,5	98,1	99,2	101,1	101,8
MVP	74,8	101,3	97,7	107,9	102,4	101,2
MVI	74,7	107,2	108,5	109,3	104,1	106,3
MVVM	71,7	97,8	95,4	97,9	98,2	98,7

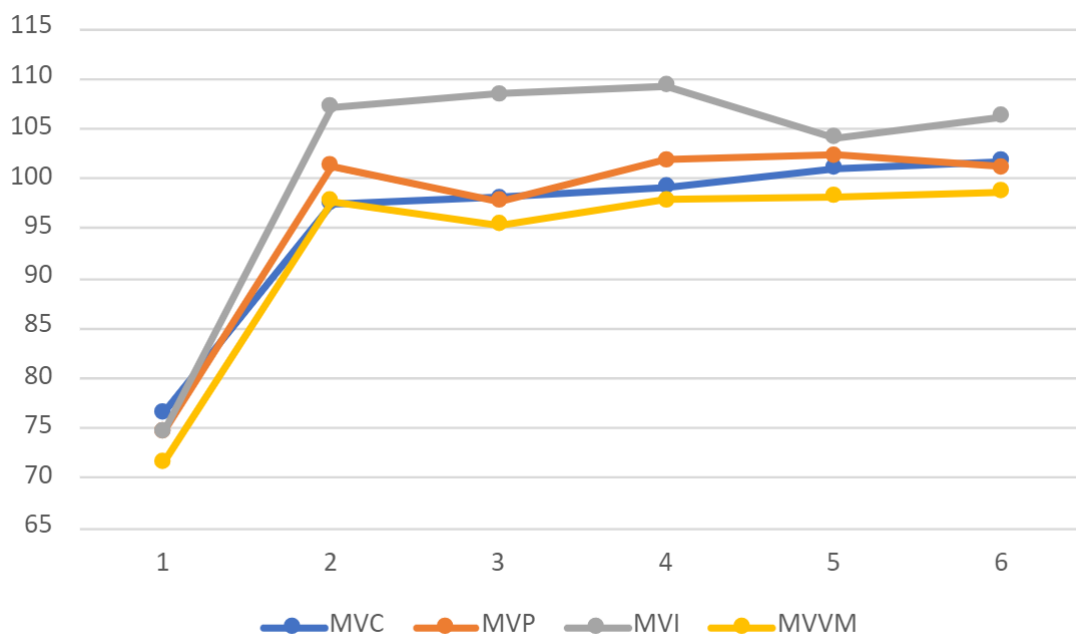


Figura 18: Consumo de memória RAM com o uso contínuo ao longo de 5 minutos. Fonte: elaborada pelo autor.

4.2 Capacidade de Manutenção e Testabilidade

Dois pontos foram avaliados com base no modelo SAAM, capacidade de manutenção e testabilidade. Para avaliar cada ponto foi analisado o nível de coesão e acoplamento em cada arquitetura individualmente, levando em consideração os níveis de acoplamento e coesão especificados na seção anterior.

4.2.1 Coesão

Para avaliar o nível de coesão de cada arquitetura é necessário analisar cada componente de cada arquitetura separadamente. O componente *Model* é igual nas quatro arquiteturas. O

componente *View* nas quatro arquiteturas tem uma abordagem semelhante, na qual segue-se o ciclo de vida básico de uma *activity*, no qual este é "Procedural" de acordo com a tabela de coesão mencionada anteriormente. Por estes motivos os componentes *Model* e *View* não estão incluídos nessa discussão. Apenas o terceiro componente, *Controller* no MVC, *Presenter* no MVP, *ViewModel* no MVVM e *Intent* no MVI serão discutidos.

No MVC o *Controller* é responsável pela vinculação dos dados na *Activity/Fragment*. No MVP, o *Presenter* assume as responsabilidades do *Controller*. Ou seja, o *Controller* no MVC e o *Presenter* no MVP fazem as mesmas tarefas. Portanto esses componentes têm o nível de coesão semelhante. No MVVM, o nível de coesão para o componente *ViewModel* é o mesmo do *Presenter* no MVP, assim como no *Intent* no MVI. No geral, as quatro arquiteturas compartilham o mesmo nível de coesão.

4.2.2 Acoplamento

No MVC a conexão entre *Model*, *View* e *Controller* estão no nível "Dados Estruturados". A conexão entre o *Controller* e *Model*, *Controller* e *View* estão no nível "Controle". Mais detalhes na Tabela 4.

Tabela 4: MVC - Nível de Acoplamento

Componente	Destino	Conexão	Acoplamento
<i>Model</i>	<i>View</i>	<i>Model</i> envia os dados para <i>View</i>	<i>Dados Estruturados</i>
<i>Model</i>	<i>Controller</i>	<i>Model</i> envia os dados para <i>Controller</i>	<i>Dados Estruturados</i>
<i>View</i>	<i>Model</i>	<i>View</i> envia os dados para <i>Model</i>	<i>Dados Estruturados</i>
<i>Controller</i>	<i>Model</i>	<i>Controller</i> controla o fluxo de dados	<i>Controle</i>
<i>Controller</i>	<i>View</i>	<i>Controller</i> controla o fluxo de dados	<i>Controle</i>

No MVP o acoplamento do componente *View* e *Presenter* está no nível "Mensagem", o que significa que a comunicação é via interface. O componente *Presenter* passa parâmetros para o *Model* para realizar tarefas. Após a execução das tarefas o *Model* retornará a resposta que pode incluir dados ou uma mensagem de erro. Assim, as conexões entre o *Model* e *Presenter* estão em nível "Dados Estruturados". As interações dos componentes estão resumidas na Tabela 5.

No MVVM o acoplamento é um pouco diferente por causa do *data binding*, ou seja, o arquivo xml do *layout* fica responsável por gerenciar os eventos, estes serão disparados no componente *ViewModel*, a partir deste ponto segue igual as demais arquiteturas. As interações dos componentes estão resumidas na Tabela 6.

Tabela 5: MVP - Nível de Acoplamento

Componente	Destino	Conexão	Acoplamento
<i>View</i>	<i>Presenter</i>	<i>Interface</i>	<i>Mensagem</i>
<i>Presenter</i>	<i>View</i>	<i>Interface</i>	<i>Mensagem</i>
<i>Presenter</i>	<i>Model</i>	Parâmetros: dados primitivos/objetos	<i>Dados Estruturados</i>
<i>Model</i>	<i>Presenter</i>	Retorno: dados primitivos/objetos	<i>Dados Estruturados</i>

Tabela 6: MVVM - Nível de Acoplamento

Componente	Destino	Conexão	Acoplamento
<i>View</i>	<i>View-Model</i>	<i>View-Model</i> passa os comandos através de uma <i>interface</i>	<i>Mensagem/Controle</i>
<i>View-Model</i>	<i>View</i>	<i>Interface</i>	<i>Mensagem</i>
<i>View-Model</i>	<i>Model</i>	Parâmetros: dados primitivos/objetos	<i>Dados Estruturados</i>
<i>Model</i>	<i>View-Model</i>	Retorno: dados primitivos/objetos	<i>Dados Estruturados</i>
<i>Model</i>	<i>View</i>	Fluxo de dados	<i>Dados Estruturados</i>

No MVI o acoplamento do componente *View* e *Intent* é nível *Message level* de maneira bidirecional. O componente *Intent* passa parâmetros para o *Model* para realizar tarefas. Após a execução das tarefas o *Model* retornará a resposta que pode incluir dados ou uma mensagem de erro. Assim, as conexões entre o *Model* e *Intent* estão em nível "Dados Estruturados", e da mesma forma *Model* e *View*. As interações dos componentes estão resumidas na Tabela 7.

Tabela 7: MVI - Nível de Acoplamento

Componente	Destino	Conexão	Acoplamento
<i>View</i>	<i>Intent</i>	Envia evento	<i>Mensagem</i>
<i>Intent</i>	<i>View</i>	Envia evento	<i>Mensagem</i>
<i>Intent</i>	<i>Model</i>	Parâmetros: dados primitivos/objetos	<i>Dados Estruturados</i>
<i>Model</i>	<i>Intent</i>	Retorno: dados primitivos/objetos	<i>Dados Estruturados</i>
<i>Model</i>	<i>View</i>	Fluxo de dados	<i>Dados Estruturados</i>

Os componentes MVC tem um nível de acoplamento maior. Os níveis de acoplamento são principalmente nível "Dados Estruturados" e "Controle", que são os mais altos quando comparados com as demais arquiteturas. A arquitetura MVP tem o nível de acoplamento principalmente no nível "Mensagem" e "Dados Estruturados". A arquitetura MVVM é semelhante à arquitetura MVP, mais tem uma conexão a mais de nível "Dados Estruturados", logo, tem um acoplamento mais alto quando comparado ao MVP. O MVI tem um nível de acoplamento próximo ao MVVM com uma conexão a mais de nível "Mensagem", assim sendo, este tem um acoplamento maior que o MVVM. Para mais detalhes sobre a estrutura de execução das aplicações veja o Apêndice D.

Os níveis de acoplamento para as quatro arquiteturas são: MVC > MVI > MVVM > MVP. Por modificabilidade e testabilidade: MVC < MVI < MVVM < MVP. No MVC os com-

ponentes são mais acoplados quando comparados com as demais arquiteturas e o MVP é o que tem o menor nível de acoplamento. O alto acoplamento dificulta a modificação dos componentes, uma alteração efetuada em uma parte do sistema ressoa pelas demais, sendo necessário que partes adjacentes sejam alteradas. Além das dificuldades de alteração, alto acoplamento dificulta a testabilidade do sistema, visto que torna-se complexo isolar partes específicas do sistema para testá-las. Assim sendo MVC uma arquitetura com sua manutenção sendo mais trabalhosa quando comparada com as demais arquiteturas e MVP a mais simples. Mais detalhes acerca dos resultados veja o Apêndice B e D.

5 CONCLUSÃO E TRABALHOS FUTUROS

No Android como em qualquer outro sistema a arquitetura de *software* é um conceito importante. Uma arquitetura consistente garante uma alta escalabilidade, bom desempenho e fácil manutenção. Diferentes padrões oferecem as mais diversas abordagens para lidar com sistêmicas diversas. Neste trabalho foi efetuada uma pesquisa assertiva em busca de informações sobre os padrões arquiteturais mais comumente utilizados no mercado.

O padrão MVC se mostrou melhor para projetos menores onde é possível trabalhar com uma estrutura mais simples e reaproveitar rapidamente os comportamentos dos *Controllers* entre múltiplas *Views*. Para projetos maiores o MVP se mostrou a solução ideal juntamente com o MVVM devido ao seu desacoplamento entre a camada de dados e a interface de usuário, sua capacidade de manutenção e expansão do sistema. A utilização da modularização de maneira eficiente ao separar os interesses garante a facilidade na criação de testes e permite uma evolução estrutural do aplicativo sem precedentes.

O MVVM tem o melhor tempo de execução de todos e valores medianos nos demais quesitos analisados. MVC traz uma implementação mais inteligível, ideal para projetos de menor escala e de abordagem simples, porém ineficiente em aplicações complexas e propensas a expandir. O MVP e o MVVM tem resultados próximos porém o MVP tem um consumo maior de memória RAM quando comparado ao MVVM e os demais. O MVI teve seus tempos de execução e consumo de recursos maior que os demais.

O MVVM, MPV e MVI fazem um trabalho melhor do que o MVC ao dividir seu aplicativo em componentes modulares e de propósito único. O MVVM tem uma performance superior comparado aos demais. Por mais que o MVP e o MVVM tenham resultados próximos o MVP

tem sua implementação mais simples em uma comparação direta com o MVVM. O MVI apresenta a implementação mais complexa enquanto o MVC tem a estrutura mais simples. Para aplicações de pequeno porte o MVC é o padrão de projeto recomendado dada a simplicidade de sua implementação, já em aplicações de grande porte o MVVM o padrão de projeto de *software* ideal para desenvolvimento de aplicações Android.

Durante o desenvolvimento deste trabalho foi constatado alguns pontos que podem ser melhorados em trabalhos futuros e novas métricas para resultados mais assertivos em certos aspectos, sendo estas: aumentar o número de amostras, levantar mais cenários para uma noção mais ampla acerca dos resultados em cada arquitetura em contextos diferentes, implementações mais complexas e maior variedade de dispositivos Android durante os testes.

Referências

- ANDROID. *Android*. 2021. Disponível em: <https://www.android.com/intl/pt-BR_br/>.
- APPBRAIN. *Number of Android apps on Google Play*. 2021. Disponível em: <<https://appbrain.com/stats/number-of-android-apps>>.
- ARACHCHI, T. W. *Android activity lifecycle*. 2018. Disponível em: <<https://medium.com/@thinuwanwickramaarachchi/android-activity-lifecycle-b8126ed3e985>>.
- BARBOSA, A. S. R. *Código fonte*. 2021. Disponível em: <<https://github.com/adrileysrb/trabalho-de-conclusao>>.
- CESAR, A. F. Uma análise comparativa entre os padrões mvp e mvvm na plataforma android. Universidade Federal da Paraíba, 2019.
- CHANDRASEKAR, A.; RAJESH, S.; RAJESH, P. A research study on software quality attributes. *International Journal of Scientific and Research Publications*, Citeseer, v. 4, n. 1, p. 14–19, 2014.
- CHEN, L.; BABAR, M. A.; NUSEIBEH, B. Characterizing architecturally significant requirements. *IEEE software*, IEEE, v. 30, n. 2, p. 38–45, 2012.
- CHEREDNICHENKO, S. *MVC vs MVP vs MVVM vs MVI. Choosing an Architecture for Android App*. 2021. Disponível em: <mobindustry.net/blog/mvc-vs-mvp-vs-mvvm-vs-mvi-choosing-an-architecture-for-android-app/>.
- CONSERVANCY, S. F. *Git –distributed-is-the-new-centralized*. 2021. Disponível em: <<https://git-scm.com/>>.
- DABROWSKI, M. *Medium - Build Your own MVI Framework*. 2019. Disponível em: <<https://medium.com/appnroll-publication/build-your-own-mvi-framework-a76d72c6e8e7>>.
- DAOUDI, A. et al. An exploratory study of mvc-based architectural patterns in android apps. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. [S.l.: s.n.], 2019. p. 1711–1720.
- DAVID, G. Software architecture: a roadmap. In: *Proceedings of the Conference on the Future of Software Engineering*. [S.l.: s.n.], 2010. p. 91–101.
- GALOS, M. *Modern Android Architecture with MVI design pattern*. 2021. Disponível em: <<https://amsterdamstandard.com/en/post/modern-android-architecture-with-mvi-design-pattern>>.
- GARCIA, J. et al. Identifying architectural bad smells. In: *IEEE. 2009 13th European Conference on Software Maintenance and Reengineering*. [S.l.], 2014. p. 255–258.
- GARG, S. *Android Complete Guide*. 2017. Disponível em: <https://www.tutorialspoint.com/android/android_resources.html>.
- GARLAN, D. Software architecture. Carnegie Mellon University, 2012.

- GATTAL, A. *Understand Android Basics Part 1: Application, Activity and Lifecycle*. 2018. Disponível em: <<https://medium.com/@Abderraouf/understand-android-basics-part-1-application-activity-and-lifecycle-b559bb1e40e>>.
- GITHUB. *Where the world builds software*. 2021. Disponível em: <<https://github.com/>>.
- GOOGLE. *Visão geral da avaliação do desempenho de apps*. 2021. Disponível em: <<https://developer.android.com/studio/profile/measuring-performance?hl=pt-br>>.
- HIPP, D. R. *What Is SQLite?* 2021. Disponível em: <<https://sqlite.org/index.html>>.
- IVANOVICH, B. V. et al. Using mvc pattern in the software development to simulate production of high cylindrical steel ingots. *Journal of Crystal Growth*, v. 526, p. 125240, 2019. ISSN 0022-0248. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S0022024819304555>>.
- KEARNEY, A. *The MVC Schema*. 2019. Disponível em: <<https://medium.com/@adamkearney124/model-view-controller-f2bdf1ee999>>.
- LAND, S. K. Results of the ieee survey of software engineering standards users. In: *IEEE. Proceedings of IEEE International Symposium on Software Engineering Standards*. [S.l.], 1997. p. 242–270.
- MACORATTI, J. C. *Compreendendo o padrão MVVM : Model-View-ViewModel (revisão)*. 2011. Disponível em: <http://www.macoratti.net/16/09/net_mvvm1.htm>.
- MARTIN, R. C. Design principles and design patterns. *Object Mentor*, v. 1, n. 34, p. 597, 2010.
- MATTSSON, M.; GRAHN, H.; MÅRTENSSON, F. Software architecture evaluation methods for performance, maintainability, testability, and portability. In: *CITeseer. Second International Conference on the Quality of Software Architectures*. [S.l.], 2006.
- MEDVIDOVIC, N.; TAYLOR, R. N. Software architecture: foundations, theory, and practice. In: *IEEE. 2010 ACM/IEEE 32nd International Conference on Software Engineering*. [S.l.], 2010. v. 2, p. 471–472.
- MEIAPPANE, A.; CHITHRA, B.; VENKATAESAN, P. Evaluation of software architecture quality attribute for an internet banking system. *arXiv preprint arXiv:1312.2342*, 2013.
- MOHTASHIM, M. *Android Resources Organizing and Accessing*. 2015. Disponível em: <https://www.tutorialspoint.com/android/android_services.htm>.
- MOHTASHIM, M. *Android - Services*. 2016. Disponível em: <https://www.tutorialspoint.com/android/android_services.htm>.
- MUNAWAR, B. W. G.; WAHYU, U. Performance comparison of native android application on mvp and mvvm. 2020.
- MUNAWAR, P. *Android MVI-Reactive Architecture Pattern*. 2014. Disponível em: <<https://medium.com/code-yoga/mvi-model-view-intent-pattern-in-android-98c143d1ee7c>>.
- NUNES, F. *Android MVC x MVP x MVVM qual Pattern utilizar — Parte 1*. 2017. Disponível em: <<https://medium.com/@FilipeFNunes/android-mvc-x-mvp-x-mvvm-qual-pattern-utilizar-parte-1-3defc5c89afd>>.

ORACLE. *Java é a Linguagem das Possibilidades*. 2021. Disponível em: <<https://www.oracle.com/br/java/technologies/>>.

POTEL, M. *Mvp: Model-view-presenter the taligent programming model for c++ and java*. *Taligent Inc*, v. 20, 1996.

QUEIROZ, A. *IO19: Google anuncia que atualmente existem 2,5 bilhões de dispositivos Android ativos*. 2019. Disponível em: <<https://www.tudocelular.com/mercado/noticias/n141261/io19-google-atualmente-2-5-bilhoes-android-ativos.html>>.

SANTANA, V. J. et al. *Sistemas de informação escaláveis utilizando programação orientada a eventos e nosql*. In: *2015 10th Iberian Conference on Information Systems and Technologies, CISTI 2015*. [S.l.: s.n.], 2015.

SHAHRIAR, H.; HADDAD, H. M. *Content provider leakage vulnerability detection in android applications*. In: *Proceedings of the 7th International Conference on Security of Information and Networks*. [S.l.: s.n.], 2014. p. 359–366.

SINGH, B.; GAUTAM, S. *The impact of software development process on software quality: a review*. In: *IEEE. 2016 8th international conference on computational intelligence and communication networks (CICN)*. [S.l.], 2016. p. 666–672.

SONI, S. *Padrão MVI (Model-View-Intent) no Android*. 2018. Disponível em: <<https://medium.com/code-yoga/mvi-model-view-intent-pattern-in-android-98c143d1ee7c>>.

STATCOUNTER. *Mobile Operating System Market Share Worldwide*. 2021. Disponível em: <<https://gs.statcounter.com/os-market-share/mobile/worldwide>>.

SUN, W.; CHEN, H.; YU, W. *The exploration and practice of mvvm pattern on android platform*. In: *ATLANTIS PRESS. 2016 4th International Conference on Machinery, Materials and Information Technology Applications*. [S.l.], 2017.

TIWARI, V. *Application Resources Overview in Android*. 2018. Disponível em: <<https://medium.com/@vikasviki/application-resources-overview-in-android-2986dea099ed>>.

VARADI, G. *MVC/MVP/MVVM/CLEAN/VIPER/REDUX — building abstractions for the sake of building abstractions (and because they're pretty and popular)*. 2018. Disponível em: <<https://proandroiddev.com/mvc-mvp-mvvm-clean-viper-redux-mvi-prnsaaspfruicc-building-abstractions-for-the-sake-of-building-18>>.

ZHAO, F. K. X.; ZOU, Y. *Improving the modifiability of the architecture of business applications*. 11th International Conference on Quality Software, Madrid, 2011, 2011.

Anexos

Apêndice A: Aplicação Utilizada nos Testes

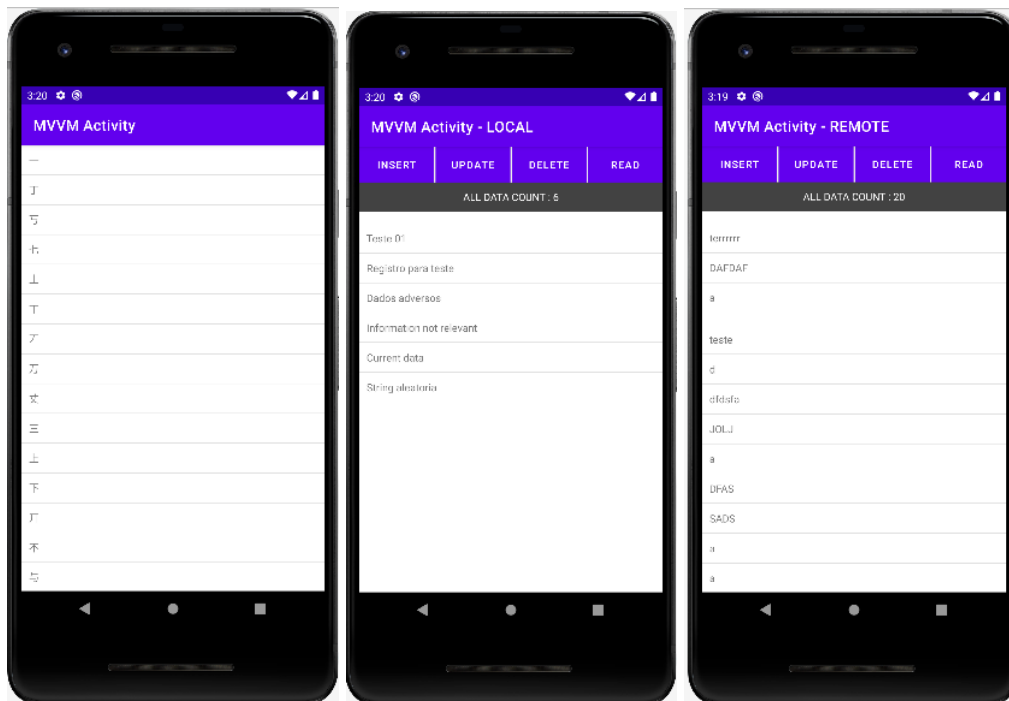


Figura 19: Aplicação utilizada nos testes - Em execução padrão MVVM.
Fonte: elaborada pelo autor.

Apêndice B: Resultados do Experimento

Resultados do experimento em cada arquitetura. Tempo de execução, consumo de memória RAM e CPU, e uso contínuo de memória RAM ao longo de cinco minutos.

Tabela 8: Resultados dos tempos de execução em milissegundos.

MVC	1368	1505	1448	1400	1431
MVP	1420	1346	1545	1399	1417
MVI	2.223	2207	2226	2309	2207
MVVM	1326	1297	1302	1245	1250

Tabela 9: Resultados do uso de CPU em porcentagem.

MVC	38	34	33	37	33
MVP	37	30	36	35	31
MVI	30	39	35	37	35
MVVM	31	35	33	35	36

Tabela 10: Resultados do uso de memória RAM em *megabytes*.

MVC	75,3	71,5	81,3	79,3	74,5
MVP	74,4	79,8	78,2	83,9	83,2
MVI	80,7	80,4	78,6	83,1	82,8
MVVM	72,3	71,8	74,8	73,2	74,4

Apêndice C: Estrutura de Arquivos do Projeto

Estrutura de arquivos dos projetos de cada aplicação:

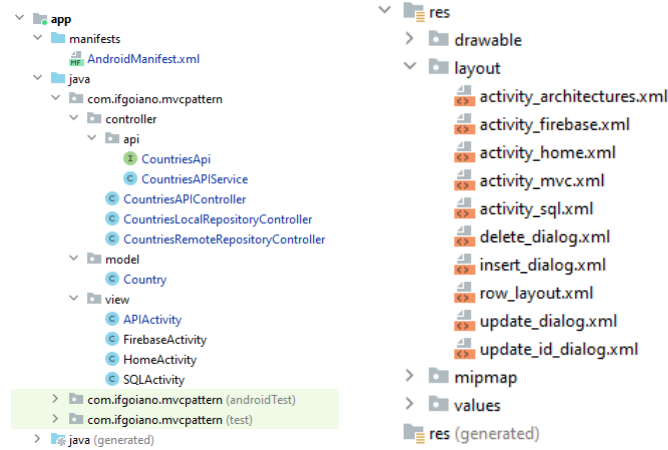


Figura 20: MVC - Estrutura de arquivos do projeto. Fonte: elaborada pelo autor.

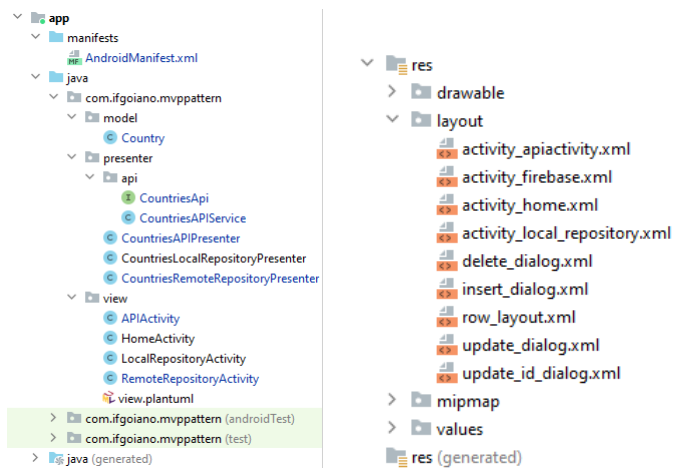


Figura 21: MVP - Estrutura de arquivos do projeto. Fonte: elaborada pelo autor.

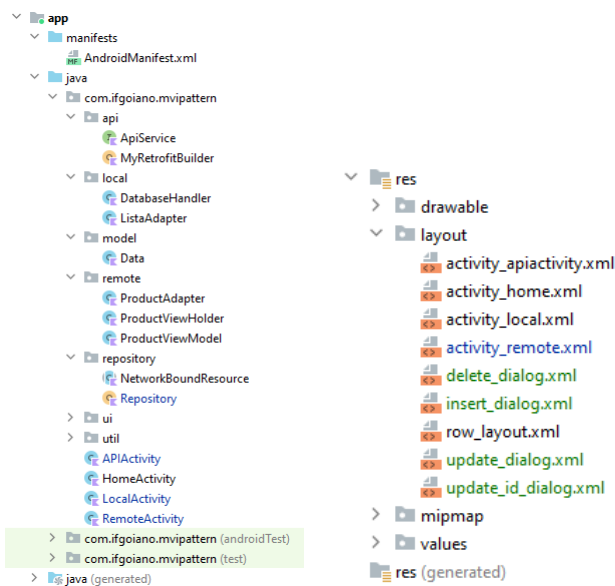


Figura 22: MVI - Estrutura de arquivos do projeto. Fonte: elaborada pelo autor.

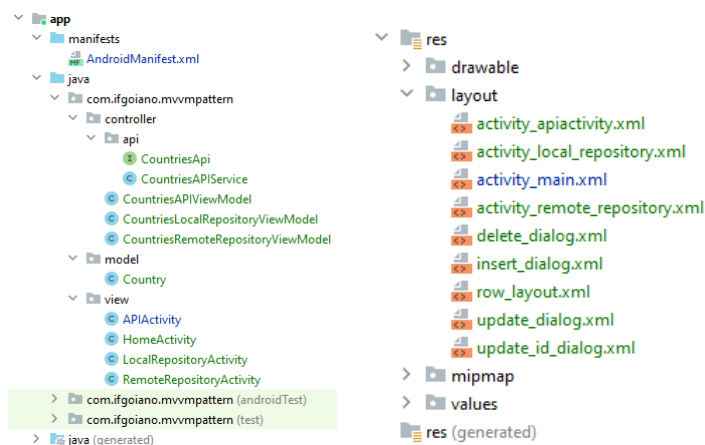


Figura 23: MVVM - Estrutura de arquivos do projeto. Fonte: elaborada pelo autor.

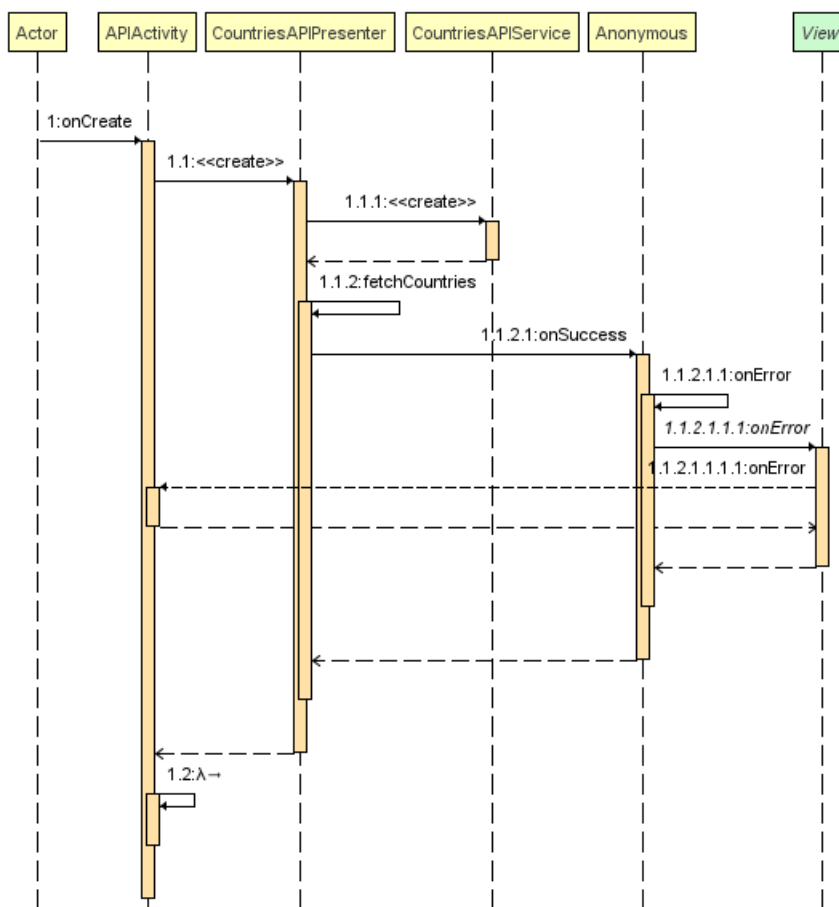


Figura 25: MVP Diagrama de Sequência. Fonte: elaborada pelo autor.

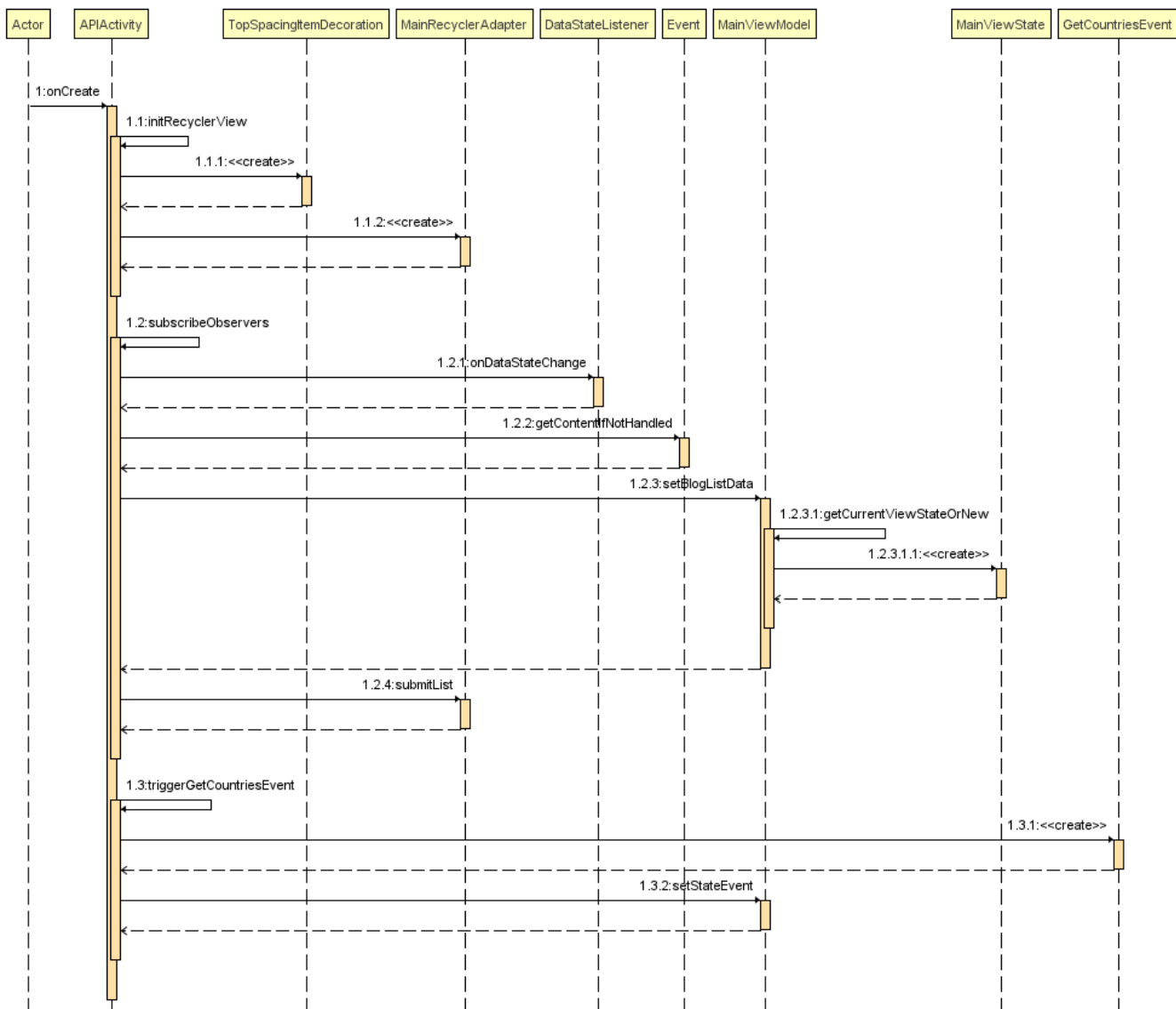


Figura 26: MVI Diagrama de Sequência. Fonte: elaborada pelo autor.

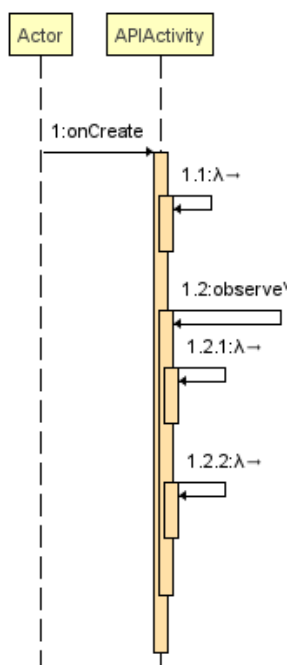


Figura 27: MVVM Diagrama de Sequência. Fonte: elaborada pelo autor.