UNICESUMAR – UNIVERSIDADE CESUMAR
CENTRO DE CIÊNCIAS EXATAS TECNOLÓGICAS E AGRÁRIAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

VINICIUS AUGUSTO ZARAMELLO

KOTLIN E PADRÕES DE ARQUITETURA MODERNA

Maringá/2024

VINICIUS AUGUSTO ZARAMELLO

KOTLIN E PADRÕES DE ARQUITETURA MODERNA

Trabalho de Conclusão de Curso apresentado para obtenção do grau de bacharel em Engenharia de Software, do Centro Universitário Cesumar — Unicesumar

Orientador: Prof. Dr. João Choma

MARINGÁ/2024

VINICIUS AUGUSTO ZARAMELLO

KOTLIN E PADRÕES DE ARQUITETURA MODERNA

Trabalho de Conclusão de Curso apresentado para obtenção do grau de bacharel em Engenharia de Software, do Centro Universitário Cesumar — Unicesumar

Orientador: Prof. Dr. João Choma

BANCA EXAMINADORA

Prof. (Nome do orientador)

Afiliações

Prof. (Nome do professor avaliador)

Afiliações

Prof. (Nome do professor avaliador)

Afiliações

AGRADECIMENTOS

A Deus, por ter me dado força e perseverança durante todos esses anos, permitindo que eu chegasse ao fim deste curso.

Aos meus pais, Julio e Elizandra, que me apoiaram em todas as minhas escolhas e que, graças ao exemplo de esforço deles, me ajudaram a chegar até aqui.

A todos os membros da minha família, que acreditaram em mim e me apoiaram ao longo desses anos.

À Lilian Augusto Gimenes Ceron, pelo amor e apoio.

Agradeço a todos os meus amigos que fizeram parte dessa caminhada, que me motivaram e me deram o prazer de sua companhia durante esses anos. Em especial, agradeço a Caio Brioli Magalhães, Cauã Custodio Murata, Leonardo Rossi, Samuel Rigo e Natália Virdi Flausino.

Ao Professor João Choma, por sua orientação no meu trabalho de conclusão de curso.

Aos colegas de trabalho, Altieres de Matos, Diego Fernando do Espírito Santo, Andre Spineli Dolce, Tiago Hamati, Rubens Yamasaki e Marcio Lima, por me apoiarem durante essa caminhada, por me mostrarem as possibilidades da área que escolhi seguir e por contribuírem para minha formação profissional.

Aos Professores Aparecido Vilela Junior e Thiago Bússola, por cultivarem em mim a paixão pela tecnologia e pela programação.

A todos os professores, por me proporcionarem uma educação de qualidade ao longo de todos os anos da graduação, por sempre me apoiarem e incentivarem a seguir em frente e por me mostrarem que a depressão é, sim, um obstáculo, mas que, com perseverança, esse obstáculo pode ser superado, um passo de cada vez.

RESUMO

Com a crescente complexidade dos aplicativos móveis, a adoção de arquiteturas robustas e modulares tornou-se essencial para o desenvolvimento sustentável e escalável no ecossistema Android. Arquiteturas como Clean Architecture e MVVM emergem como soluções eficazes para organizar e desacoplar as responsabilidades entre camadas, facilitando a manutenção e o teste do sistema ao longo do tempo. A linguagem Kotlin, oficializada como padrão para Android, oferece ferramentas e recursos que se alinham a esses padrões arquiteturais, promovendo um desenvolvimento mais seguro e eficiente. O objetivo deste trabalho é analisar as vantagens e desvantagens das arquiteturas Clean Architecture e MVVM aplicadas ao desenvolvimento Android com Kotlin, por meio de um estudo de caso prático. Foi desenvolvido um aplicativo de formulário que utiliza essas arquiteturas para gerenciar a interface de usuário e a lógica de negócios de forma organizada e modular. Os resultados indicam que, embora demandem maior esforço inicial, essas arquiteturas melhoram a testabilidade e flexibilidade do sistema, sendo especialmente vantajosas em projetos de longa duração que exigem adaptabilidade.

ABSTRACT

With the increasing complexity of mobile applications, adopting robust and modular architectures has become essential for sustainable and scalable development within the Android ecosystem. Architectures such as Clean Architecture and MVVM emerge as effective solutions to organize and decouple responsibilities across layers, facilitating system maintenance and testing over time. Kotlin, officially recognized as the standard language for Android, provides tools and features that align with these architectural patterns, enabling safer and more efficient development. This study aims to analyze the advantages and disadvantages of Clean Architecture and MVVM architectures applied to Android development using Kotlin through a practical case study. A form application was developed using these architectures to manage the user interface and business logic in an organized and modular manner. The results indicate that, although requiring greater initial effort, these architectures improve system testability and flexibility, proving particularly advantageous in long-term projects requiring adaptability.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

Nos últimos anos, o desenvolvimento de software experimentou uma evolução significativa em suas arquiteturas, especialmente no contexto do desenvolvimento para dispositivos móveis. Com o aumento da complexidade dos aplicativos e a demanda por sistemas mais robustos e escaláveis, surgiram arquiteturas que priorizam uma organização clara e o desacoplamento das responsabilidades, garantindo modularidade e facilidade de manutenção a longo prazo. Esse cenário levou à adoção de arquiteturas como Clean Architecture e MVVM (Model-View-ViewModel), amplamente utilizadas no desenvolvimento de aplicativos Android devido à sua capacidade de estruturar sistemas de maneira modular e sustentável (BASS et al., 2012; MARTIN, 2017; RICHARDS; FORD, 2020).

Nesse contexto, a linguagem Kotlin emergiu como uma escolha preferencial para o desenvolvimento Android, consolidando-se oficialmente em 2017 como a linguagem de programação padrão na plataforma. Kotlin oferece uma sintaxe enxuta e expressiva, características como segurança de nulidade e suporte nativo a coroutines para operações assíncronas, proporcionando uma experiência de desenvolvimento mais eficiente e menos propensa a erros (JEMEROV; ISAKOVA, 2017). Essas vantagens fazem com que Kotlin forneça um suporte sólido para a implementação de arquiteturas modernas, facilitando a manutenção e escalabilidade dos projetos.

As arquiteturas modernas, como Clean Architecture e MVVM, ganham relevância ao promoverem a separação de responsabilidades e uma estrutura modular. A Clean Architecture, proposta por Robert C. Martin (MARTIN, 2017), organiza a aplicação em camadas, isolando a lógica de negócios dos detalhes de implementação e da interface. Já o MVVM se destaca pela sua capacidade de organizar a interação entre a interface de usuário e a lógica de negócios, gerenciando dados de forma reativa e independente. Juntas, essas arquiteturas oferecem um modelo robusto para enfrentar os desafios do desenvolvimento móvel, permitindo que equipes trabalhem em paralelo e mantenham a qualidade e a consistência do código ao longo do ciclo de vida do projeto.

A implementação prática realizada neste trabalho permitiu observar que ambas as arquiteturas trazem benefícios específicos ao desenvolvimento de

aplicativos móveis. A Clean Architecture mostrou-se eficaz ao garantir flexibilidade e manutenibilidade ao código, com sua estrutura modular que facilita futuras adaptações. O MVVM, por sua vez, destacou-se no gerenciamento reativo da interface, simplificando a interação do usuário com o aplicativo e oferecendo um desempenho eficiente. No entanto, também foram observadas algumas limitações, como a complexidade inicial para configurar essas arquiteturas, o que pode ser desafiador para projetos menores ou equipes menos experientes.

A estrutura do trabalho é organizada da seguinte maneira: na Seção 2 são apresentados os objetivos e justificativas do estudo; a Seção 3 aborda a fundamentação teórica, onde são apresentados os conceitos básicos e as principais características das arquiteturas estudadas. Na Seção 4 são discutidos os trabalhos relacionados, que fornecem uma visão comparativa do tema abordado. Na Seção 5 é apresentada a metodologia, detalhando o método de pesquisa adotado, incluindo os critérios de comparação e as ferramentas utilizadas. A Seção 6 apresenta o desenvolvimento, exemplificando o fluxo de desenvolvimento das aplicações e a organização modular do código. A Seção 7 exibe os resultados da implementação prática e a análise comparativa, enquanto a Seção 8 traz as considerações finais e possíveis direções para trabalhos futuros.

2. OBJETIVO GERAL

O objetivo geral deste trabalho é apresentar, explicar e implementar as arquiteturas Clean Architecture e MVVM no desenvolvimento Android com Kotlin, com ênfase em seus pontos fortes e limitações.

2.1 Objetivos específicos

Para alcançar o objetivo geral, são definidos os seguintes objetivos específicos:

1. Investigar na literatura o uso das arquiteturas Clean Architecture e MVVM no desenvolvimento Android;
2. Analisar as vantagens e desvantagens de implementar essas arquiteturas utilizando Kotlin;

3. Implementar um estudo de caso que demonstre a aplicação prática de Clean Architecture e MVVM no desenvolvimento de um formulário Android;
4. Avaliar os resultados obtidos com a implementação, destacando modularidade, testabilidade e eficiência no desenvolvimento;

3. FUNDAMENTAÇÃO TEÓRICA

Para o desenvolvimento de aplicações móveis modernas e escaláveis, é essencial a adoção de práticas e padrões arquitetônicos bem definidos. Esses padrões fornecem diretrizes que ajudam na organização modular do código, na separação de responsabilidades e na manutenção de um sistema sustentável ao longo do tempo (BASS et al., 2012; MARTIN, 2017; FOWLER, 2004).

No contexto de desenvolvimento Android, a utilização de linguagens otimizadas para a plataforma, como Kotlin, e arquiteturas bem estruturadas, como MVVM e Clean Architecture, tem se mostrado essencial para a criação de aplicações robustas e eficientes. Com uma arquitetura sólida é possível atender tanto requisitos funcionais quanto não funcionais, como desempenho e segurança, necessários para garantir uma experiência satisfatória ao usuário final e facilitar a manutenção do sistema (CAYER; KLEIN, 2021; RICHARDS; FORD, 2020; LEE; RICHARDS, 2019).

3.1 Arquitetura de Software

A arquitetura de software desempenha um papel crítico na garantia da sustentabilidade, flexibilidade e qualidade de um sistema ao longo do seu ciclo de vida, promovendo a criação de sistemas robustos e preparados para o futuro. Define a estrutura do sistema, organizando seus componentes e a forma como interagem entre si. Segundo Bass, Clements e Kazman, “arquitetura de software é a estrutura ou estruturas do sistema, que compreendem componentes de software, as propriedades visíveis externamente desses componentes e as relações entre eles” (BASS et al., 2012). Assim, a arquitetura orienta as decisões de projeto e garante que o sistema satisfaça requisitos funcionais e não funcionais, como desempenho e segurança.

Nos últimos anos, a arquitetura de software ganhou destaque como um campo fundamental na engenharia de software, à medida que os profissionais perceberam sua importância para o desenvolvimento de sistemas robustos e de fácil manutenção. Como apontam Richards e Ford, uma arquitetura bem projetada fornece um entendimento compartilhado entre desenvolvedores e stakeholders (RICHARDS; FORD, 2020). Esse entendimento é essencial para garantir a modularização do sistema, reduzindo a complexidade e facilitando a comunicação entre as partes envolvidas.

Olhando para testabilidade Feathers (2004) enfatiza que sistemas com componentes desacoplados são menos suscetíveis a bugs e mais fáceis de manter e estender (FEATHERS, 2004). Desta forma, a dissociação entre módulos promove uma estrutura modular e flexível, permitindo que alterações num módulo não afetem o funcionamento dos restantes e simplificando a integração de novos recursos.

Além disso, uma arquitetura de software bem projetada permite a evolução contínua do sistema. As arquiteturas bem definidas permitem que sistemas complexos evoluam de maneira controlada, reduzindo o risco de se tornare inflexíveis (GARLAN; SHAW, 1994). Isto permite acompanhar as exigências do mercado e as necessidades dos utilizadores sem que o sistema se torne obsoleto ou excessivamente complexo.

3.2 Kotlin

Kotlin emergiu como uma das linguagens mais influentes no desenvolvimento Android devido a sua combinação de produtividade, segurança e concisão. Criada pela JetBrains em 2011 e oficialmente adotada pelo Google como linguagem preferencial para Android em 2017, Kotlin apresenta uma sintaxe mais enxuta e funcionalidades modernas que facilitam o desenvolvimento de código limpo e sustentável. Como destacam Jemerov e Isakova, “Kotlin foi projetada para ser interoperável com Java e, ao mesmo tempo, corrigir alguns de seus pontos fracos, como a verbosidade e a ausência de recursos de segurança para nulidade” (JEMEROV; ISAKOVA, 2017). Essa interoperabilidade permite que os desenvolvedores utilizem Kotlin em projetos já existentes, integrando-se perfeitamente a ambientes Java, enquanto agrega recursos de ponta.

A segurança contra null pointers é um dos principais diferenciais da linguagem. Em Kotlin, a nulidade é tratada nativamente, reduzindo a ocorrência do erro comum “NullPointerException” que é recorrente em Java. Essa funcionalidade permite que os desenvolvedores trabalhem de maneira mais segura, evitando falhas que podem comprometer a confiabilidade do aplicativo. Como observa Bloch, “o tratamento de nulidade nativo de Kotlin é uma inovação que não apenas melhora a segurança, mas também simplifica a codificação” (BLOCH, 2018).

Outro recurso essencial de Kotlin são as coroutines, que facilitam a programação assíncrona e concorrente. Em Android, onde operações como chamadas de rede e manipulação de banco de dados exigem processamento fora da linha principal para evitar bloqueios na interface do usuário, as coroutines se destacam pela simplicidade e pelo controle. Ao invés de depender de callbacks complexos, as coroutines fornecem uma maneira elegante de lidar com tarefas assíncronas (FAYYAZ; MADINA, 2020).

Além disso, a sintaxe enxuta de Kotlin promove o desenvolvimento de código mais conciso, o que aumenta a produtividade e reduz a probabilidade de erros. Recursos como funções de extensão, lambdas e operadores simplificados permitem que os desenvolvedores escrevam mais com menos(WAMPLER; SEIDEL, 2019).

Dessa forma, Kotlin se solidifica como uma linguagem moderna e eficiente que não apenas atende às demandas atuais do desenvolvimento Android, mas também projeta o ecossistema para o futuro.

3.3 Padrões arquiteturais

Os padrões de arquitetura de software representam planos estruturais compostos por um conjunto de regras e diretrizes desenvolvidas para organizar e modularizar sistemas, permitindo escalabilidade e qualidade. Esses padrões evoluíram a partir dos erros e aprendizados obtidos ao longo de décadas de desenvolvimento, surgindo como soluções generalistas para problemas recorrentes e oferecendo bases sólidas para criação de software robusto (MARTIN, 2010).

No mercado atual, sete padrões principais são amplamente utilizados, sendo eles: MVC, MVP, MVVM, CLEAN, VIPER, REDUX e MVI. No desenvolvimento de

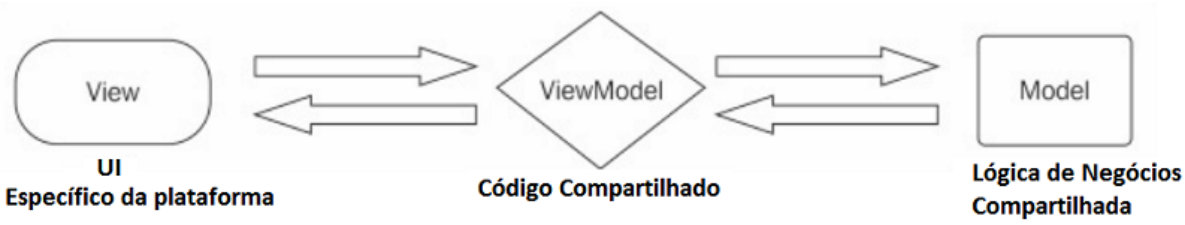
sistemas móveis, especialmente Android, destacam-se os padrões MVVM, VIPER e MVI, adaptados para atender às particularidades de dispositivos móveis, como restrições de recursos e mudanças frequentes na interface de usuário. Os padrões mais utilizados no desenvolvimento de aplicativos Android incluem MVC, MVP, MVVM e MVI, enquanto os demais são adaptações do desenvolvimento web para o ecossistema Android (VARADI, 2018).

Os padrões arquiteturais mencionados oferecem uma estrutura modular e organizada ao projeto, cada um com suas especificidades e aplicabilidades. Assim, eles garantem que o software seja fácil de manter e escalar ao longo do tempo, além de simplificar a integração de novas funcionalidades e o teste individualizado de componentes, promovendo uma base sólida para a construção de soluções tecnológicas modernas e eficientes. Nos próximos tópicos, os padrões mais relevantes para o desenvolvimento Android serão abordados em detalhes.

3.3.1 MVVM

Criado em 2005 por Ken Cooper e Ted Peters, o padrão Model-View-ViewModel (MVVM) surgiu com o objetivo de simplificar a programação orientada a eventos para interfaces de usuário, especialmente em cenários complexos onde a separação de responsabilidades é crucial (SUN; CHEN; YU, 2017). Esse padrão traz uma divisão clara entre a lógica de negócios e a interface gráfica, facilitando a organização do código e promovendo manutenibilidade e testabilidade (SANTANA et al., 2015).

Figura 1: Diagrama do Model-View-ViewModel.



Fonte: (MACORATTI, 2011)

O padrão MVVM é composto por três elementos principais: (Macoratti, 2011)

- **Model**: Representa a lógica de negócios e manipula o acesso aos dados necessários para a aplicação, sendo responsável por qualquer manipulação de dados, validação ou transformação necessária.
- **View**: Compreende a interface de usuário e todo o código relacionado à exibição visual dos dados. A camada View exibe as informações ao usuário e envia suas ações para o ViewModel.
- **ViewModel**: Atua como a ponte entre a View e o Model, orquestrando a comunicação e mantendo o estado da interface de forma reativa. Essa camada gerencia a lógica de apresentação, expõe propriedades que a View pode observar e desencadeia operações no Model conforme necessário.

Essa estrutura modularizada é vantajosa para testes unitários, já que as classes ViewModel são independentes da interface de usuário, facilitando o desenvolvimento e a manutenção de testes automatizados. Além disso, a interoperabilidade oferecida pelo MVVM permite que as aplicações sejam adaptadas para diferentes plataformas com ajustes mínimos, o que é especialmente útil no desenvolvimento multiplataforma (MUNAWAR; WAHYU, 2020).

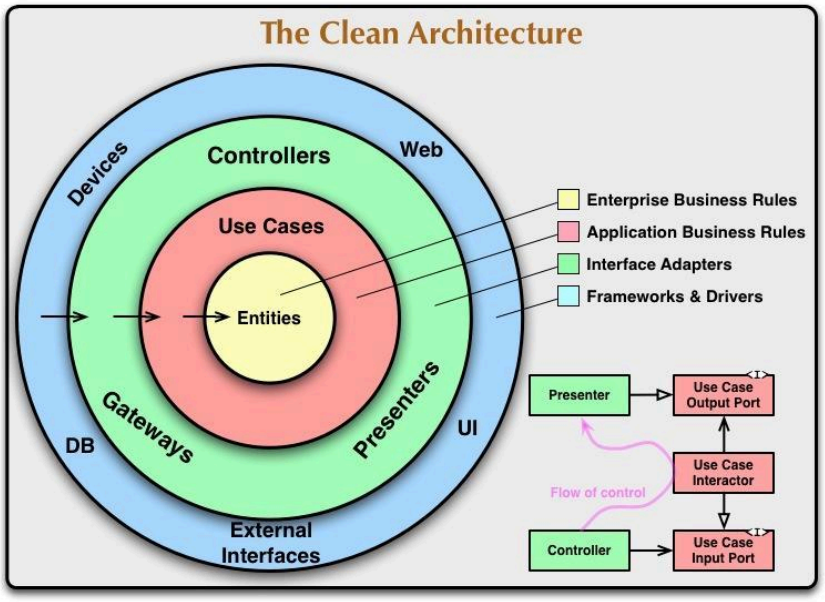
O MVVM tem sido amplamente adotado no desenvolvimento Android devido a essa separação de responsabilidades e ao suporte nativo a data binding, que permite que a View se atualize automaticamente conforme mudanças no ViewModel. A implementação reativa permite uma experiência fluida ao usuário, ao mesmo tempo em que simplifica a manutenção e o crescimento do código em longo prazo (Zhu et al., 2019).

3.3.2 Clean Architecture

Proposta por Robert C. Martin, conhecido como Uncle Bob, a Clean Architecture visa garantir a independência da camada de negócios em relação às demais camadas de uma aplicação, promovendo a modularidade e facilitando a manutenção e evolução do sistema (MARTIN, 2017). Seu objetivo é que mudanças em camadas externas, como a interface de usuário ou o mecanismo de persistência de dados, não afetem as regras de negócio. Essa estrutura é especialmente eficaz

em projetos que necessitam de alta adaptabilidade e fácil expansão para novas plataformas, interfaces ou tecnologias de armazenamento.

Figura 2 - Diagrama da Clean Architecture.



Fonte: (MARTIN 2017)

A Clean Architecture é estruturada em círculos concêntricos, que representam diferentes áreas e níveis do software. Quanto mais próximo do núcleo, maior a importância e o nível de abstração das regras de negócio. Os níveis externos referem-se a implementações de baixo nível, como a interface de usuário e detalhes de persistência, que servem ao núcleo sem comprometer sua lógica (MARTIN, 2017).

O princípio central da Clean Architecture é a regra de dependência, que estabelece que as dependências de código devem sempre apontar para dentro, em direção às políticas e camadas de maior nível. Dessa forma, as camadas externas, como a interface e os dados, dependem das camadas de negócio, mas o contrário não ocorre. No código-fonte da camada de negócios, evita-se o uso de qualquer elemento declarado nas outras camadas, protegendo a lógica de negócios de mudanças externas e promovendo um alto grau de independência. (Martin, 2017)

A Clean Architecture tem sido amplamente adotada por sua capacidade de proporcionar sistemas escaláveis e manuteníveis, alinhando-se às necessidades contemporâneas de desenvolvimento ágil e de longa duração (MARTIN, 2017).

4. TRABALHOS RELACIONADOS

Barbosa (2022) apresenta uma análise comparativa entre os padrões arquiteturais MVC, MVP, MVVM e MVI, com foco na plataforma Android. O autor avalia cada padrão em relação à modularidade, facilidade de manutenção e desempenho, demonstrando como esses padrões influenciam na organização de código e na experiência de desenvolvimento. Esse trabalho foi desenvolvido no Instituto Federal Goiano e utiliza estudos práticos para examinar os impactos de cada padrão, oferecendo insights sobre suas vantagens e desvantagens no desenvolvimento de aplicações móveis.

Brollo (2023) propõe uma implementação de aplicação Android utilizando MVVM e Clean Architecture, além de adotar práticas de TDD (Desenvolvimento Orientado a Testes) para melhorar a confiabilidade e a manutenibilidade do código. Desenvolvido no Instituto Federal do Rio Grande do Sul, o estudo explora o impacto do uso conjunto dessas práticas e arquiteturas, discutindo a testabilidade e a modularidade alcançadas ao adotar Clean Architecture com Kotlin em um ambiente Android. A pesquisa enfatiza a importância do TDD para garantir a robustez das funcionalidades ao longo do ciclo de vida do desenvolvimento.

5. METODOLOGIA

A fim de alcançar os objetivos propostos, este trabalho adota uma metodologia exploratória e aplicada. Primeiramente, é realizada uma revisão bibliográfica para fundamentar o estudo, investigando o uso de Clean Architecture e MVVM na literatura sobre desenvolvimento Android com Kotlin. Em seguida, aplica-se um estudo de caso com a implementação de um aplicativo Android que utiliza essas arquiteturas. A análise prática avalia modularidade, testabilidade e

manutenibilidade, fundamentando a discussão sobre os benefícios e desafios dessas abordagens no desenvolvimento Android.

Quadro 1 – Classificação da pesquisa

Pesquisa	Classificação
Natureza	Aplicada
Objetivos	Descritivo e explicativo
Abordagem	Qualitativa e quantitativa
Fonte de Informação	Pesquisa bibliográfica e estudo de caso
Procedimentos técnicos	Revisão de literatura e implementação prática

Fonte: elaborado pelo autor

5.1 Quanto à natureza e aos objetivos da pesquisa

A pesquisa possui uma natureza aplicada, pois visa fornecer contribuições práticas para o desenvolvimento Android, analisando o uso das arquiteturas Clean Architecture e MVVM com Kotlin. Ao investigar as melhores práticas e resultados dessas implementações, espera-se contribuir para a formação de desenvolvedores que desejam adotar uma estrutura modular e escalável no desenvolvimento Android (BASS et al., 2012; MARTIN, 2017).

Quanto aos objetivos, a pesquisa é descritiva e explicativa. Seu objetivo é explorar os princípios dessas arquiteturas e os benefícios e desafios encontrados na prática, fornecendo uma visão completa sobre a aplicação de Clean Architecture e MVVM no desenvolvimento de software móvel com Kotlin (FEATHERS, 2004; RICHARDS; FORD, 2020).

5.2 Quanto à abordagem da pesquisa

Esta pesquisa adota uma abordagem mista, combinando elementos qualitativos e quantitativos para uma análise abrangente. A abordagem qualitativa abrange a revisão bibliográfica, enquanto a quantitativa foca na mensuração de aspectos como modularidade, testabilidade e desempenho no estudo de caso prático. A análise dos dados do projeto permite observar as diferenças e

características das arquiteturas MVVM e Clean Architecture no contexto Android, complementada pela rastreabilidade das informações levantadas na literatura para validar as observações práticas (LEEDY; ORMROD, 2016; CRESWELL; CLARK, 2018).

A principal fonte de informação utilizada é a pesquisa bibliográfica, realizada em bases científicas como Google Scholar e ACM Digital Library, para fundamentar os conceitos das arquiteturas aplicadas. Também foi conduzido um estudo de caso com um aplicativo Android desenvolvido em Kotlin, a fim de observar a implementação dos princípios das arquiteturas de forma prática (LEEDY; ORMROD, 2016).

5.3 Quanto aos procedimentos técnicos

Para alcançar os objetivos específicos, a metodologia desdobra-se em etapas organizadas de acordo com os seguintes procedimentos técnicos:

1. **Revisão Bibliográfica:** Foi realizada uma pesquisa detalhada sobre os conceitos e práticas das arquiteturas MVVM e Clean Architecture, utilizando fontes acadêmicas e artigos técnicos que discutem a aplicabilidade dessas arquiteturas no desenvolvimento Android.
2. **Implementação Prática (Estudo de Caso):** Um aplicativo Android foi desenvolvido em Kotlin utilizando Clean Architecture e MVVM. A implementação foi planejada para validar a organização em camadas e o desacoplamento de responsabilidades. Essa abordagem teve como foco a observação prática e a rastreabilidade das práticas descritas na literatura.
3. **Análise dos Resultados:** Os resultados obtidos com a implementação foram avaliados com base nos critérios de modularidade, testabilidade e manutenibilidade. A análise baseou-se tanto na observação prática quanto na comparação com os dados obtidos na fundamentação teórica, discutindo as vantagens e desvantagens de cada arquitetura.

5.3.1 Desenvolvimento do Estudo de Caso

No estudo de caso, foram implementadas funcionalidades básicas comuns a aplicativos Android, como um formulário para adicionar dados e uma lista para exibi-los. Esses recursos foram projetados para aplicar as arquiteturas MVVM e Clean Architecture, garantindo a separação de responsabilidades e o desacoplamento entre as camadas.

5.3.2 Análise dos Resultados

A análise dos dados obtidos foi conduzida com o intuito de verificar a eficácia das arquiteturas na organização e manutenção do código. Modularidade, testabilidade e flexibilidade foram mensuradas e comparadas com os conceitos presentes na literatura, garantindo a rastreabilidade entre teoria e prática. Essa análise oferece uma visão prática das contribuições das arquiteturas Clean Architecture e MVVM no desenvolvimento de aplicativos Android com Kotlin.

6. Apresentação de resultados

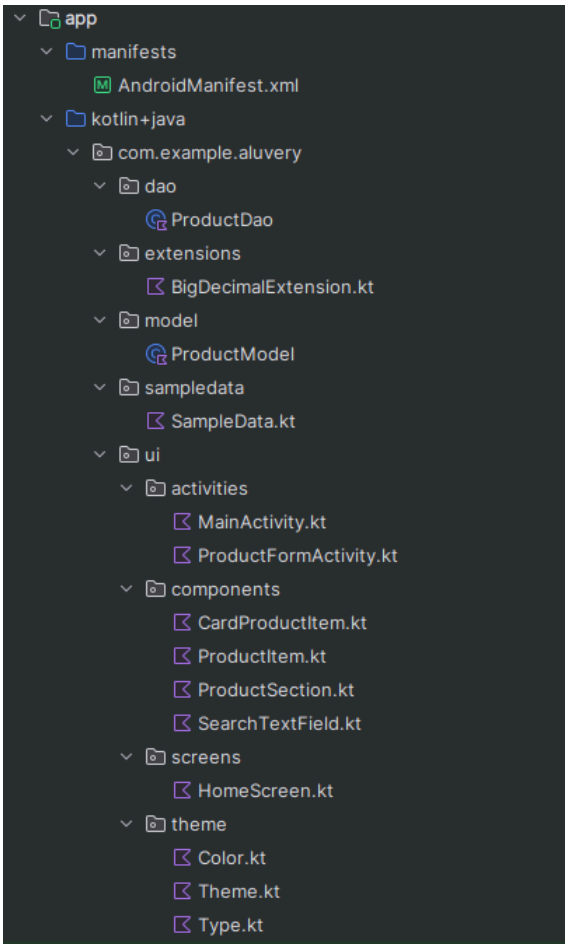
Nesta seção, são apresentados os resultados obtidos com a implementação das arquiteturas Clean Architecture e MVVM em uma aplicação Android desenvolvida com Kotlin. O aplicativo consiste em um formulário que armazena os dados localmente em uma lista e os exibe na interface do usuário. As análises cobrem aspectos de modularidade, testabilidade, desempenho e flexibilidade do sistema, ilustradas pela estrutura de pastas do projeto com base nas pesquisas feitas.

6.1 Resultados para MVVM

A implementação da arquitetura MVVM (Model-View-ViewModel) neste projeto foi essencial para estruturar a camada de apresentação de forma clara e reativa, destacando as vantagens de organizar a interface de usuário (UI) e a lógica de apresentação de forma independente. A organização da árvore de pacotes reflete essa separação, com as pastas dedicadas a View e ViewModel, garantindo que as responsabilidades de cada camada estejam claramente definidas e isoladas. Essa abordagem, também observada nos estudos de Barbosa (2022) e Brollo (2023),

facilita a manutenção e o teste das funcionalidades, além de promover uma estrutura modular que permite uma escalabilidade eficiente.

Figura 3 - Árvore de pacotes Mvvm



Fonte: Elaborado pelo autor

6.1.1 Estrutura do Projeto para MVVM

A estrutura do projeto, organizada conforme os princípios MVVM, apresenta uma distribuição de pastas que facilita a compreensão e o acesso rápido aos componentes de cada camada. No pacote ui/view, estão concentrados os componentes de interface, como MainActivity, ProductFormActivity, e os diversos componentes visuais, incluindo ProductItem, CardProductItem, e ProductSection. Esses componentes são responsáveis apenas por exibir os dados e receber as interações do usuário, sem necessidade de se conectar diretamente com a lógica de negócios ou os dados.

No pacote ui/viewmodel, os ViewModels, como ProductViewModel, atuam como intermediários entre a interface e a camada de dados. Essa estrutura permite que a View observe o estado do ViewModel e seja automaticamente atualizada sempre que houver alterações, eliminando a necessidade de atualizar manualmente os componentes visuais. Segundo Brollo (2023), essa independência entre View e ViewModel é uma das principais vantagens do MVVM, pois reduz o acoplamento e promove uma atualização dinâmica da interface com base no estado dos dados.

A estrutura do projeto em MVVM também favorece a legibilidade e a organização do código. Barbosa (2022) destaca que a disposição das camadas auxilia na clareza do fluxo de dados, facilitando o entendimento da lógica por parte de novos desenvolvedores, além de proporcionar uma modularização eficaz, como observada neste projeto.

6.1.2 Modularidade em MVVM

A modularidade promovida pela arquitetura MVVM é evidente na divisão das responsabilidades de cada camada causando facilidade na manutenção, a separação de responsabilidades permite que ajustes na interface possam ser feitos diretamente na camada de View, sem afetar a lógica de apresentação gerida pelo ViewModel. Modificações na exibição de produtos ou no comportamento dos componentes visuais, por exemplo, podem ser realizadas de maneira independente, otimizando o tempo de manutenção e reduzindo o risco de erros em outras partes do sistema e a reatividade e observação de dados que permite que a View observe diretamente o ViewModel, a interface é atualizada de forma reativa sempre que o estado do ViewModel é alterado, oferecendo uma experiência de usuário mais fluida e responsiva. Segundo Barbosa (2022), esse padrão torna o desenvolvimento de interfaces mais previsível, pois garante que a UI exiba sempre o estado atual dos dados.

6.1.3 Testabilidade em MVVM

A estrutura MVVM proporciona uma alta testabilidade na camada de apresentação, uma vantagem essencial em projetos onde a validação do estado da interface e a lógica de apresentação são necessárias. A validação da Lógica de Apresentação ocorre com os ViewModels isolados, é possível realizar testes unitários para verificar se o estado dos dados é gerido corretamente e se a interface

exibe as informações esperadas. Isso permite uma validação segura e independente da lógica de negócios, seguindo as boas práticas sugeridas por Brollo (2023). Já o Teste da Reatividade da Interface é a capacidade de observar mudanças no ViewModel facilita a validação do comportamento da interface em resposta a diferentes estados, sem precisar mockar a camada de dados diretamente. A abordagem MVVM se mostrou eficaz na execução de testes sobre as interações entre a interface e o ViewModel, assegurando que o usuário veja sempre dados atualizados.

6.1.4 Desempenho e Eficiência em MVVM

Dada a simplicidade do aplicativo, o padrão MVVM mostrou-se eficiente em termos de desempenho e na atualização da interface:

1. **Atualização Eficiente da Interface:** Como a View é atualizada automaticamente sempre que o estado no ViewModel muda, há uma otimização no processamento, reduzindo a necessidade de operações manuais para controlar a UI. A abordagem reativa foi eficaz para garantir um fluxo de dados contínuo e evitar sobrecargas na thread principal, alinhada às recomendações de Brollo (2023) sobre eficiência em aplicativos Android.
2. **Simplicidade e Responsividade:** Em um aplicativo sem operações complexas, a estrutura MVVM oferece uma implementação leve e eficiente. Como o foco está na reatividade e na atualização automática da interface, o MVVM atende bem ao propósito do projeto, garantindo uma execução ágil e sem travamentos.

6.1.5 Flexibilidade para Alterações em MVVM

A estrutura modular do MVVM proporciona flexibilidade para que o sistema seja facilmente adaptado a futuras mudanças. A expansão com Novas Funcionalidades de UI, a divisão em camadas permite que novas funcionalidades sejam adicionadas à interface sem impacto na lógica de apresentação. Por exemplo, se novos campos ou validações precisarem ser incluídos no formulário, isso pode ser feito apenas ajustando a View e o ViewModel, o que favorece a manutenção e a evolução do sistema e a facilidade para Integrações Futuras, a camada de ViewModel pode ser ajustada para incluir APIs externas ou interações com um banco de dados, caso o projeto evolua para isso. Essa separação entre ViewModel e

View não apenas simplifica o desenvolvimento, como também facilita futuras integrações sem que a camada de interface precise ser redesenhada.

6.2 Resultados para Clean Architecture

A implementação da Clean Architecture neste projeto organizou o aplicativo Android em camadas distintas e bem definidas para maximizar a modularidade e a escalabilidade. A árvore de pacotes reflete essa organização, com uma separação clara entre dados, domínio e apresentação, cada qual com suas respectivas responsabilidades. “A Clean Architecture propõe que cada camada seja isolada, para que mudanças em camadas externas não afetem a lógica central do sistema”, conforme mencionado por Martin (2017), que destaca a importância de organizar dependências de maneira que apontem apenas para o centro, onde estão as políticas de negócio.

6.2.1 Estrutura do Projeto para Clean Architecture

A estrutura do projeto foi elaborada com base nos princípios da Clean Architecture, promovendo o isolamento entre as diferentes responsabilidades do sistema. No pacote data, são armazenadas as implementações que tratam do acesso e da manipulação dos dados:

1. **Local:** Contém o ProductDao, responsável pela manipulação do armazenamento local.
2. **Remote:** Voltado a comunicação externa entre API ou outros tipos de serviços.
3. **Repository:** Implementa a interface de repositório definida na camada de domínio e coordena o acesso aos dados locais e remotos, seguindo o princípio de desacoplamento da Clean Architecture.

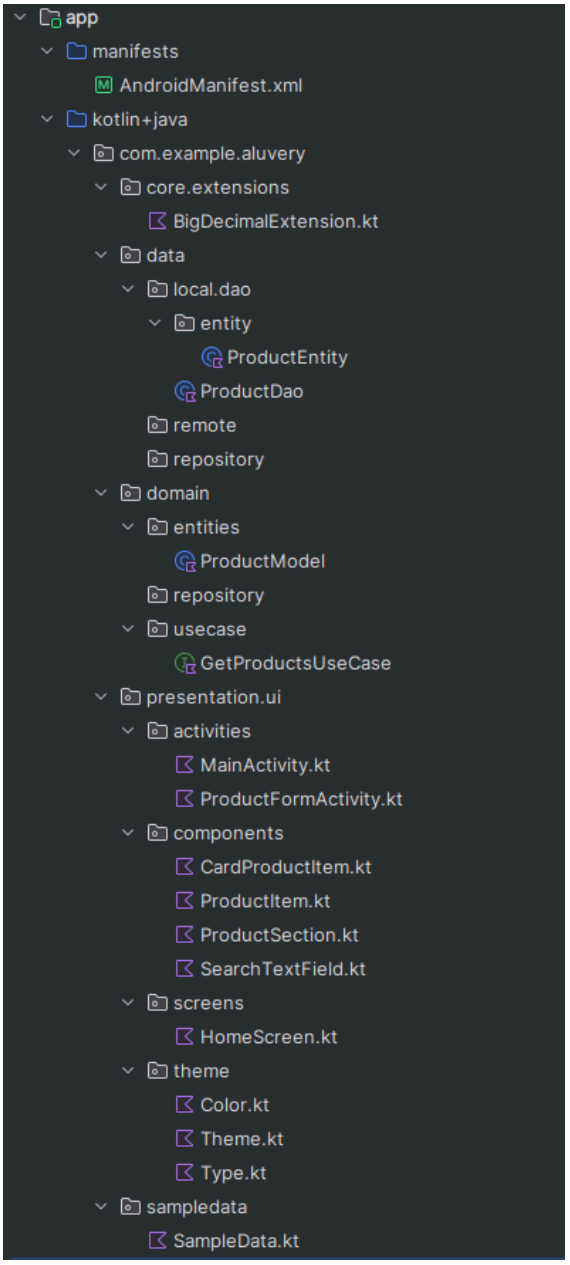
A camada domain abriga as regras de negócios e inclui o Product (modelo de domínio) e o GetProductsUseCase (caso de uso). Esses elementos definem a lógica central do sistema de forma independente, interagindo com a camada de dados por meio de interfaces, o que facilita mudanças e testes na lógica sem afetar diretamente a camada de apresentação.

Na camada **presentation**, estão organizados os componentes de UI, que incluem telas e ViewModels:

1. **UI:** Contém os componentes visuais e as telas, como HomeScreen e ProductFormScreen, responsáveis por exibir os dados e interagir com o usuário.
2. **ViewModel:** O ProductViewModel serve como ponte entre a camada de domínio e a UI, gerenciando o estado da interface e organizando as comunicações de dados.

Essa estrutura modularizada proporciona uma clara organização dos elementos de cada camada e facilita a evolução do projeto com mudanças independentes.

Figura 4 - Árvore de pacotes de Clean Architecture



Fonte: Elaborado pelo autor

Descrição: A árvore do projeto exibe a organização modular, evidenciando o desacoplamento entre a camada de dados, a camada de domínio e a camada de apresentação.

6.2.2 Modularidade em Clean Architecture

A Clean Architecture confere uma estrutura altamente modular, essencial para projetos que buscam manter a flexibilidade e a clareza de responsabilidades. A independência entre as camadas permite que a lógica de negócios e o acesso aos dados sejam mantidos isolados da interface de usuário, de modo que **modificações em uma camada não impactem diretamente as demais** (Martin, 2017). Esse modelo favorece o reuso de componentes de negócios, permitindo que a camada de domínio seja facilmente reutilizada em outros contextos ou até mesmo em outros projetos, uma característica que aumenta a sustentabilidade do projeto a longo prazo.

6.2.3 Testabilidade em Clean Architecture

A divisão das responsabilidades e o desacoplamento entre camadas proporcionados pela Clean Architecture aumentam a testabilidade do aplicativo:

1. **Testes de Casos de Uso:** A camada de domínio, ao conter os casos de uso de forma independente, permite que a lógica de negócios seja testada isoladamente, garantindo que o comportamento do sistema atenda aos requisitos sem a influência da interface ou da implementação de dados.
2. **Isolamento Completo das Camadas:** Cada camada pode ser validada separadamente. Isso facilita a identificação de erros e mantém a confiabilidade do sistema ao longo do desenvolvimento.

6.2.4 Desempenho e Eficiência em Clean Architecture

Apesar da simplicidade do aplicativo, a estrutura modular da Clean Architecture assegura que o desempenho seja eficiente e controlado. O fluxo de dados entre as camadas segue um caminho claro e direto, evitando sobrecargas desnecessárias e garantindo que as operações de dados e a lógica de negócios sejam processadas separadamente da interface, o que resulta em uma experiência de usuário estável e sem travamentos (Brown et al., 2012).

6.2.5 Flexibilidade para Alterações em Clean Architecture

A arquitetura implementada oferece flexibilidade para futuras alterações e expansões. Novas funcionalidades, como validações adicionais ou integração com APIs, podem ser adicionadas sem interferir na estrutura existente, uma prática que reduz os custos de manutenção e facilita expansões a longo prazo (Martin, 2017). A camada de dados pode ser facilmente estendida para incluir APIs ou bancos de dados externos, mantendo a lógica de negócios e a interface de usuário protegidas de modificações nas fontes de dados, o que permite que o projeto se adapte a novos requisitos de maneira rápida e organizada.

6.3 Ameaças à Validade do Estudo

Este estudo apresenta algumas limitações que podem influenciar a validade e a generalização dos resultados. Primeiramente, a aplicação desenvolvida foi relativamente simples, consistindo em um formulário com armazenamento de dados local. Esse contexto limitado pode não capturar completamente os desafios de implementação e as vantagens das arquiteturas MVVM e Clean Architecture em sistemas de maior complexidade ou com demandas de performance específicas, como aplicativos que dependem de bancos de dados remotos ou integração com múltiplas APIs.

Além disso, a análise comparativa foi baseada em uma única implementação de cada arquitetura, o que significa que os resultados podem variar conforme a experiência dos desenvolvedores e o contexto específico do projeto. A ausência de testes em um ambiente de produção ou com usuários finais também é um fator que limita a avaliação da usabilidade e da escalabilidade das arquiteturas analisadas.

Outro ponto a ser considerado é a possível influência de vieses ao longo do desenvolvimento e da análise, já que o estudo foi conduzido sem avaliação independente ou revisão por outros profissionais da área. Para reduzir essas ameaças, futuras pesquisas podem incluir análises com projetos de complexidade variada, além de avaliações por múltiplos desenvolvedores e testes com feedback de usuários.

7. Conclusão

Este trabalho apresentou uma análise das arquiteturas MVVM e Clean Architecture aplicadas ao desenvolvimento Android com a linguagem Kotlin, destacando suas vantagens e desvantagens com base em uma implementação prática e fundamentação teórica. A adoção de MVVM se mostrou vantajosa para o gerenciamento da interface de usuário, promovendo reatividade e facilitando a separação entre a lógica de negócios e a camada de apresentação. Essa estrutura permite que as Views observem automaticamente as mudanças no estado dos dados, proporcionando uma experiência fluida ao usuário e simplificando a lógica de UI. Por outro lado, uma das desvantagens do MVVM é a complexidade inicial na configuração do data binding e na sincronização dos dados entre o ViewModel e a View, o que pode exigir uma curva de aprendizado maior, especialmente em projetos com equipes menos experientes.

A Clean Architecture, por sua vez, provou ser essencial para promover uma estrutura modular, com camadas independentes que tornam o código mais flexível e testável. A separação de camadas — em dados, domínio e apresentação — permite que alterações em uma camada tenham mínimo impacto sobre as demais, facilitando a manutenção e a escalabilidade do sistema. Essa modularidade traz benefícios evidentes em projetos de longo prazo, que precisam evoluir continuamente para se adaptar às demandas dos usuários. No entanto, a organização em várias camadas pode ser excessivamente complexa para sistemas mais simples, pois exige um esforço inicial considerável de configuração e organização.

Ambas as arquiteturas mostraram-se eficazes para garantir uma testabilidade elevada. A independência entre as camadas e o desacoplamento promovido pela Clean Architecture permitiram testes unitários isolados na camada de domínio e nas classes de ViewModel. O uso de MVVM facilitou a criação de testes para validar a interação entre a lógica e a interface, confirmando a eficácia de ambas as arquiteturas na verificação e manutenção da qualidade do código. No entanto, a criação de uma infraestrutura de testes eficaz demanda planejamento e atenção, especialmente à medida que o sistema cresce em complexidade, o que pode aumentar o tempo e os recursos necessários para a implementação.

Em termos de desempenho e eficiência, a aplicação dos princípios de Clean Architecture e MVVM, em conjunto com os recursos do Kotlin, como as coroutines, resultou em uma execução ágil, mesmo em um aplicativo simples como um formulário. A separação de responsabilidades permitiu que operações intensivas fossem otimizadas sem afetar a interface, o que é vantajoso para garantir um desempenho contínuo. Contudo, em sistemas mais complexos, o custo de gerenciar dependências entre as camadas pode impactar o desempenho se não for otimizado adequadamente, especialmente em dispositivos com recursos limitados.

Por fim, as arquiteturas analisadas demonstraram grande flexibilidade para adaptações e expansão, uma característica fundamental em um ambiente de desenvolvimento em constante evolução. A estrutura modular facilita a adaptação a novas tecnologias e requisitos sem comprometer a integridade do sistema. Entretanto, essa flexibilidade também pode tornar o código mais fragmentado e complexo para novos desenvolvedores, o que pode dificultar a curva de aprendizado e a entrada de novos membros na equipe.

Em conclusão, Clean Architecture e MVVM, juntamente com Kotlin, oferecem uma base sólida para o desenvolvimento de aplicações Android de fácil manutenção, escaláveis e com alto nível de qualidade. No entanto, a complexidade adicional e o esforço inicial exigidos pelas arquiteturas devem ser considerados, especialmente em projetos menores ou com equipes menos experientes. A escolha por essas arquiteturas depende diretamente das necessidades de cada projeto e dos recursos disponíveis. Para trabalhos futuros, recomenda-se que a análise de arquiteturas seja estendida a projetos mais complexos, onde a integração com APIs, armazenamento remoto e funcionalidades avançadas permitam uma avaliação mais robusta das arquiteturas em cenários reais. Além disso, um estudo envolvendo desenvolvedores de diferentes níveis de experiência e feedback de usuários pode oferecer uma perspectiva mais abrangente sobre as vantagens e limitações do MVVM e da Clean Architecture no desenvolvimento Android.

8. Trabalhos Futuros

Este estudo abre oportunidades para futuras pesquisas e aprimoramentos, que podem contribuir para expandir o conhecimento e a aplicabilidade de Clean Architecture e MVVM. A seguir, destacam-se duas sugestões de trabalhos futuros:

1. **Estudo Comparativo com Outras Arquiteturas de Estado e Dados**

Com o surgimento de novas abordagens, estudos futuros poderiam ampliar esta pesquisa ao comparar Clean Architecture e MVVM com outras arquiteturas e padrões de gerenciamento de estado, como MVI (Model-View-Intent) e Redux. Um estudo comparativo ajudaria a avaliar as vantagens e limitações de cada abordagem em diferentes cenários, especialmente em projetos que exigem maior controle de estado e manipulação de dados complexos.

2. **Implementação com Integração a APIs e Banco de Dados**

Este trabalho focou em um aplicativo de formulário com armazenamento local temporário. Em futuras pesquisas, recomenda-se expandir a análise para incluir integrações com APIs e bancos de dados locais e remotos, o que permitiria uma análise mais aprofundada do desempenho, escalabilidade e segurança dessas arquiteturas em aplicações com maior fluxo de dados e interações complexas.

Essas possibilidades de pesquisa futura contribuem para o aprimoramento das práticas de desenvolvimento Android, aprofundando o conhecimento sobre Clean Architecture e MVVM e fornecendo diretrizes mais detalhadas para a criação de aplicativos escaláveis e modulares.

REFERÊNCIAS BIBLIOGRÁFICAS

BASS, L.; CLEMENTS, P.; KAZMAN, R. Software Architecture in Practice. Addison-Wesley, 2012.

MARTIN, R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017.

RICHARDS, M.; FORD, N. Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media, 2020.

JEMEROV, S.; ISAKOVA, S. Kotlin in Action. Manning Publications, 2017.

FAYYAZ, U.; MADINA, A. Programming Android with Kotlin: Achieving Structured Concurrency with Coroutines. Packt Publishing, 2020.

GARCIA, J.; POPESCU, D.; MEDVIDOVIC, N.; GORLIM, P. Enhancing Architectural Modularity through Crosscutting Concern Scopes. ACM Transactions on Software Engineering and Methodology, 2014.

FEATHERS, M. Working Effectively with Legacy Code. Prentice Hall, 2004.

BARBOSA, Adriley Samuel Ribeiro. Análise comparativa entre os padrões MVC, MVP, MVVM e MVI na plataforma Android. 2022. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Instituto Federal Goiano, Campus Urutaí, Urutaí, 2022.

BROLLO, Gabriel Pellegrini. Uso de TDD em aplicação Android MVVM e Clean Architecture. 2023. Trabalho de Conclusão de Curso – Instituto Federal do Rio Grande do Sul, Campus Veranópolis, Veranópolis, 2023.

LEEDY, P. D.; ORMROD, J. E. *Practical Research: Planning and Design*. Pearson, 2016.

CRESWELL, J. W.; CLARK, V. L. P. *Designing and Conducting Mixed Methods Research*. Sage Publications, 2018.

MACORATTI, J. (2011). MVVM – Model-View-ViewModel no Desenvolvimento de Aplicações WPF. Blog Macoratti.

ZHU, L., et al. (2019). Android Programming with MVVM and Data Binding: Developing a Modern Android Application with MVVM.

FOWLER, M. (2004). Patterns of Enterprise Application Architecture. Addison-Wesley Professional

BROWN, W., MALVEAU, R., MCCORMICK, H. W., & MOWBRAY, T. J. (2012). AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Addison-Wesley Professional.

CAYER, A., & Klein, J. (2021). Effective Kotlin: Best Practices for Kotlin Developers. Pragmatic Bookshelf.

LEE, T., & Richards, D. (2019). "Adopting Kotlin for Android Development: Benefits and Challenges." Journal of Systems and Software, 152, 59–73.