UNIVERSIDADE EDUARDO MONDLANE

FACULDADE DE CIÊNCIAS

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

ANÁLISE COMPARATIVA DE ARQUITECTURAS DE DESENVOLVIMENTO DE APLICAÇÕES ANDROID

ERCÍLIO MARQUES MANHIÇA

Maputo

2024

UNIVERSIDADE EDUARDO MONDLANE

FACULDADE DE CIÊNCIAS

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

ANÁLISE COMPARATIVA DE ARQUITECTURAS DE DESENVOLVIMENTO DE APLICAÇÕES ANDROID

ERCÍLIO MARQUES MANHIÇA

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Eduardo Mondlane, como parte dos requisitos para a obtenção do título de Mestre em Informática, especialização: Engenharia de Software.

Orientador: Prof. Doutor. José António Nhavoto

Co-Orientador: Mestre Osvaldo Cossa

Maputo

2024

UNIVERSIDADE EDUARDO MONDLANE

FACULDADE DE CIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ANÁLISE COMPARATIVA DE ARQUITECTURAS DE DESENVOLVIMENTO DE APLICAÇÕES ANDROID

ERCÍLIO MARQUES MANHIÇA

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Eduardo Mondlane, como parte dos requisitos para a obtenção do título de Mestre em Informática, especialização: Engenharia de Software.

Aprovado em 18 de abril de 2024, por:

Prof. Doutor. Emílio Mosse
(Arguente – DMI-UEM)

Prof. Doutor. Orlando Zacarias
(Presidente – DMI-UEM)

Prof. Doutor. José António Nhavoto
(Orientador – DMI-UEM)

Maputo

Maio/2024

Dedicatória

Este trabalho é dedicado aos meus familiares, em especial a minha mãe que tudo fez para garantir que tivesse a oportunidade de estudar.

AGRADECIMENTOS

Agradeço, em primeiro lugar, a Deus, fonte de toda sabedoria e força, por guiar meus passos ao longo desta jornada de aprendizado e pesquisa.

Em segundo lugar, gostaria de estender meus agradecimentos aos meus orientadores, Prof. Doutor José Nhavoto e Mestre Osvaldo Cossa, cuja orientação constante e conhecimento inestimável foram fundamentais. Seu feedback crítico e orientação moldaram e aprimoraram esta pesquisa de maneira significativa. Sou profundamente grato pela generosidade em compartilhar seu tempo e experiência comigo.

Quero expressar minha sincera gratidão aos membros do júri, que dedicaram seu tempo para avaliar este trabalho e fornecer comentários construtivos e sugestões valiosas. Suas contribuições desempenharam um papel vital na elevação da qualidade desta dissertação.

Não posso deixar de agradecer aos meus colegas, amigos e familiares que me apoiaram incansavelmente ao longo desta jornada. Seu incentivo, encorajamento e palavras de apoio foram âncoras em momentos de desafio. Agradeço também por compartilharem suas idéias, conhecimentos e experiências, enriquecendo assim esta pesquisa.

Por fim, gostaria de expressar minha profunda gratidão à minha família e à minha companheira. Sua amizade incondicional, compreensão e apoio foram a força motriz por trás desta conquista. Agradeço por estarem ao meu lado, acreditando em mim e me incentivando a buscar sempre o meu melhor.

Para todos aqueles mencionados e também àqueles que, de forma não especificada, contribuíram ao longo dessa jornada, saibam que cada um de vocês desempenhou um papel significativo neste trabalho. Sou profundamente grato por todas as contribuições, apoio e encorajamento que recebi ao longo deste percurso.

Epígrafe

"O sucesso é a soma de pequenos esforços repetidos dia após dia."
(Robert Collier, 1926)

RESUMO

Neste estudo, a pesquisa se concentrou em avaliar e comparar as diferentes arquitecturas de aplicações Android, com o objetivo de identificar as melhores práticas de desenvolvimento que garantam eficiência, modificabilidade e testabilidade. O contexto da crescente complexidade das aplicações móveis tornou essencial para os programadores adoptar abordagens arquitecturais eficazes.

Duas arquitecturas-chave, Model-View-ViewModel (MVVM) e Model-View-Intent (MVI), foram investigadas em detalhes. Cada uma dessas arquitecturas foi analisada em termos de vantagens e desvantagens, especialmente em relação a factores críticos, como desempenho, modificabilidade e testabilidade. A pesquisa se baseou em métricas abrangentes, incluindo o tempo de renderização de frames da tela, uso de memória e manutenibilidade de código, para avaliar objetivamente o desempenho das aplicações desenvolvidas com essas arquitecturas.

Além da análise teórica, a pesquisa também incluiu uma fase prática na qual uma aplicação Android representativa foi desenvolvida utilizando diferentes padrões arquitetônicos. Posteriormente, essa aplicação foi submetida à avaliação com base nas métricas definidas. Isso permitiu uma comparação quantitativa das diferentes arquitecturas, revelando insights importantes sobre como essas escolhas arquitectónicas impactam as aplicações móveis durante o desenvolvimento.

O estudo empregou abordagens como ATAM (Architecture Tradeoffs Analysis Method) e CBAM (Cost-Benefit Analysis Method) para avaliar aspectos técnicos e econômicos das arquitecturas, fornecendo uma visão completa dos desafios e oportunidades associados a cada uma delas.

Os resultados desta pesquisa contribuíram significativamente para o entendimento das melhores práticas na criação de aplicações Android eficientes, escaláveis e testáveis. A análise detalhada das métricas e a comparação das arquitecturas permitiram que os desenvolvedores tomassem decisões informadas ao escolher a arquitectura mais adequada para os requisitos específicos do projecto.

Palavras-chave: Android, MVVM, MVI, Arquitetura de Software, Desenvolvimento Android, Avaliação Econômica, ATAM, CBAM.

ABSTRACT

In this study, the research focused on evaluating and comparing different Android application architectures with the aim of identifying best development practices that ensure efficiency, modifiability, and testability. Given the increasing complexity of mobile applications, it has become essential for developers to adopt effective architectural approaches.

Two key architectures, Model-View-ViewModel (MVVM) and Model-View-Intent (MVI), were examined in detail. Each of these architectures was analyzed in terms of their advantages and disadvantages, particularly concerning critical factors such as performance, modifiability, and testability. The research relied on comprehensive metrics, including screen frame rendering time, memory usage, and code maintainability, to objectively assess the performance of applications developed using these architectures.

In addition to theoretical analysis, the research included a practical phase in which a representative Android application was developed using different architectural patterns. Subsequently, this application underwent evaluation based on the defined metrics, enabling a quantitative comparison of the various architectures and providing valuable insights into how these architectural choices impact mobile applications during development.

The study employed approaches such as ATAM (Architecture Tradeoff Analysis Method) and CBAM (Cost-Benefit Analysis Method) to assess the technical and economic aspects of the architectures, offering a comprehensive view of the challenges and opportunities associated with each.

The results of this research significantly contributed to understanding best practices in creating efficient, scalable, and testable Android applications. The detailed analysis of metrics and architecture comparisons empowered developers to make informed decisions when selecting the most suitable architecture for project-specific requirements.

Keywords: Android, MVVM, MVI, Software Architecture, Android Development, Economic Evaluation, ATAM, CBAM.

LISTA DE FIGURAS

Lista de Equações

LISTA DE TABELAS

LISTA DE ABREVIATURAS

Abreviatura	Significado
IDE	Integrated Development Environment
TIC	Tecnologias de Informação e Comunicação
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
MVI	Model-View-Intent
SAAM	Software Architecture Analysis Method
SI	Sistema de Informação
TIC	Tecnologias de Informação e Comunicação
ATAM	Architecture Tradeoff Analysis Method
CBAM	Cost-Benefit Analysis Method
VIPER	View-Interactor-Presenter-Entity-Router

Índice

1. INTRODUÇÃO

1.1 Contextualização

As Tecnologias de Informação e Comunicação (TIC) revolucionaram a forma como nos comunicamos, trabalhamos e interagimos com o mundo ao nosso redor. De acordo com Castelos (2011), "as TIC são ferramentas essenciais para a criação, armazenamento, processamento e disseminação da informação". Castelos (2011), enfatiza a importância dessas tecnologias na sociedade moderna, destacando seu papel fundamental na manipulação e compartilhamento das informações cruciais para o desenvolvimento humano e o progresso tecnológico.

As TIC englobam uma vasta gama de tecnologias interconectadas, das quais as tecnologias moveis - sistemas e infra-estruturas que permitem a comunicação e a conectividade sem fio (IBM, 2023), os dispositivos móveis, como *smartphones* e *tablets*, são parte integrante e revolucionária. Conforme apontado por Kaplan e Haenlein (2010), "os smartphones são considerados uma extensão digital de nossas vidas cotidianas, oferecendo novas maneiras de interacção e comunicação".

Hoje em dia, o uso de dispositivos móveis é mais popular do que nunca. De acordo com estatísticas recentes, existem mais de 4 bilhões de usuários de smartphones em todo o mundo, o que representa cerca de 50% da população global (Statista, 2023a). Desses dispositivos mais de 70% correm o sistema operativo Android (Statista, 2023b).

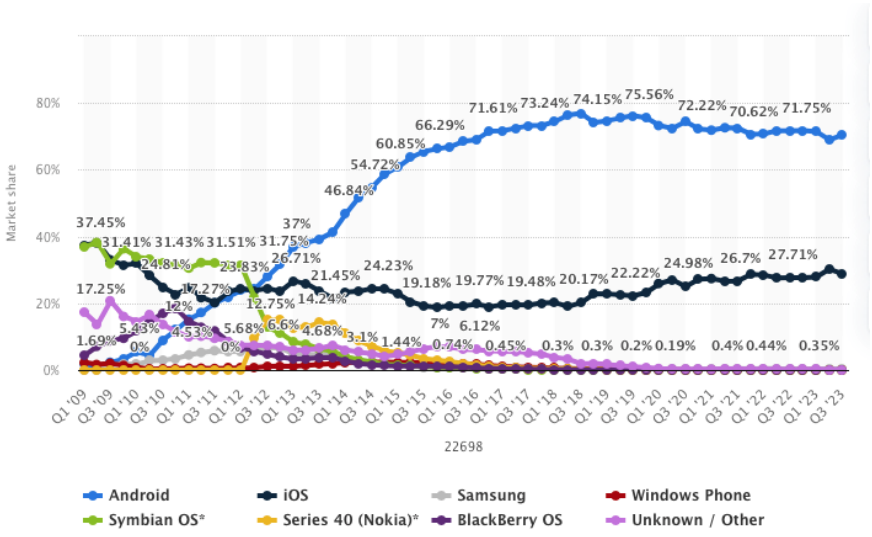


Figura 1. Distribuição de utilizadores de smartphones por sistema operativo (Statista, 2023a).

Nesse contexto, o desenvolvimento mobile desempenha um papel fundamental, permitindo que aplicativos e sistemas sejam acedidos e utilizados por meio de dispositivos móveis. Para impulsionar o desenvolvimento mobile e facilitar a criação de aplicativos eficientes, surgiram os Frameworks e as arquitecturas de desenvolvimento. Segundo Fontoura (2023), "Framework é uma estrutura ou conjunto de bibliotecas e ferramentas que fornece um modelo de desenvolvimento para a construção de aplicativos ou sistemas.

Juntamente com os Framework, as arquitecturas de desenvolvimento fornecem uma estrutura sólida para organizar o código do aplicativo. Conforme ressaltado por Martin (2017), "as arquitecturas de desenvolvimento definem a separação de responsabilidades entre os componentes do sistema, facilitando a manutenção, a escalabilidade e a colaboração entre os desenvolvedores". Algumas arquitecturas populares incluem o Model-View-Presenter (MVP), Model-View-ViewModel (MVVM) e o Model-View-Intent (MVI), VIPER(View-Interactor-Presenter-Entity-Router), MVC (Model-View-Controller), Clean Architecture, entre outras.

É neste contexto que a presente pesquisa se concentra, buscando aprofundar a compreensão de desenvolvimento mobile para Android. O objectivo é realizar uma análise comparativa entre as arquitecturas de aplicativos Android, com foco nas abordagens MVVM e MVI, a fim de identificar as melhores práticas para garantir eficiência, escalabilidade e segurança em diferentes tipos de aplicativos móveis. Para isso, serão explorados critérios como testabilidade, modificabilidade, desempenho, custo, benefício, e outros aspectos relevantes que contribuam para uma tomada de decisão informada no desenvolvimento de aplicações Android.

1.2 Problematização

Com o aumento exponencial do uso de dispositivos móveis e a ampla disseminação de aplicativos, surge a necessidade premente de identificar e aplicar as melhores práticas de desenvolvimento de aplicativos. Essa necessidade é crucial para assegurar o desempenho, escalabilidade, testabilidade e segurança dos aplicativos móveis (Bass et al., 2013).

Em meio à diversidade de arquitecturas disponíveis para o desenvolvimento de aplicativos Android e a evolução continua das tecnologias, desenvolvedores de aplicativos enfrentam dificuldades ao seleccionar a opção mais apropriada para seus projectos (Humeniuk, 2018). As peculiaridades e limitações de cada arquitectura tornam essa escolha complexa e decisiva. Eoin Woods, citado por Bass et al., (2013), destaca que a arquitectura de software compreende um conjunto de decisões de desenho cuja tomada inadequada pode resultar na interrupção do

projecto. Essa noção é reforçada por Bass et al., (2013), ao apontarem que, como arquitectos e engenheiros de software, frequentemente tomamos decisões de alto impacto económico baseadas em intuição, negligenciando a análise disciplinada de suas implicações financeiras. Tal abordagem pode ser aceitável para indivíduos com genialidade intuitiva, mas não se mostra eficaz para a maioria. A engenharia, ao contrário, preconiza a tomada de decisões sólidas e previsíveis, embasadas em critérios racionais. E é justamente para atingir esse fim que a aplicação de métodos é essencial.

De acordo com a literatura disponível, há uma tendência significativa de concentrar-se predominantemente nos aspectos técnicos ao avaliar arquitecturas e tomar decisões informadas sobre a escolha da arquitectura ideal. No entanto, é fundamental reconhecer que factores específicos de cada projecto, como a composição da equipe de desenvolvimento, os prazos e as restrições orçamentárias, exercem um impacto substancial na determinação da arquitectura mais adequada (Bass et al., 2013). Além disso, vale ressaltar a ausência de estudos que comparem os padrões MVVM e MVI, especialmente no contexto do novo paradigma de desenvolvimento de interfaces de usuário, o Jetpack Compose, que actualmente é a abordagem mais prevalente.

É precisamente nesse cenário que se insere o presente trabalho, que tem como objectivo propor uma abordagem actualizada e alinhada com o contexto de selecção de arquitecturas de desenvolvimento para aplicativos Android. Essa abordagem levará em consideração não apenas critérios técnicos, mas também elementos específicos do projecto que podem influenciar a tomada de decisão, com especial atenção para aspectos económicos. O trabalho busca preencher essa lacuna ao investigar como as diferentes arquitecturas se comportam em diversos cenários, incorporando aspectos técnicos e financeiros na análise.

Assim sendo, o trabalho se propõe a fornecer orientações abrangentes e personalizadas, adaptadas à realidade singular de cada projecto. Isso contribuirá para uma tomada de decisão mais bem fundamentada ao seleccionar a arquitectura de desenvolvimento mais apropriada para aplicativos Android, considerando não apenas os aspectos técnicos, mas também os económicos e contextuais.

1.3 Delimitação do Estudo

Este estudo tem como foco principal a análise comparativa de arquitecturas de aplicações Android, com o objectivo de identificar as melhores práticas de desenvolvimento para garantir eficiência e escalabilidade em diferentes tipos de aplicativos móveis. A pesquisa abordará os padrões arquitectónicos mais comumente utilizados, como Model-View-Intent (MVI) e Model-View-ViewModel (MVVM).

No entanto, é importante ressaltar que o estudo não abrangerá todas as possíveis arquitecturas e padrões existentes, dada a ampla variedade de abordagens disponíveis. Além disso, para comparação das arquitecturas será usado o Método de Análise de Custo-Benefício (Cost Benefit Analysis Method - CBAM) e o Método de Análise de Compensação Arquitecturais (Architecture Tradeoffs Analysis Method - ATAM). Além disso, o estudo se concentrará exclusivamente em aplicativos móveis desenvolvidos para a plataforma Android. Outras plataformas móveis, como iOS, não serão abordadas neste estudo.

Por fim, embora a pesquisa envolve uma fase prática de implementação de aplicativos móveis, ela não se aprofundará nas especificidades da codificação e programação de cada arquitectura. O objectivo principal é fornecer uma visão geral das arquitecturas, suas vantagens e desvantagens, e como elas podem ser aplicadas de forma eficiente, escalável e segura em diferentes tipos de aplicativos móveis.

1.4 Objectivos

1.4.1. Objectivo geral

Analisar comparativamente as arquitecturas de desenvolvimento de aplicativos Android.

1.4.2. Objectivos específicos

Para alcançar o objectivo geral enunciado, são definidos os seguintes objectivos específicos:

1. Identificar as principais arquitecturas de desenvolvimento de aplicativos Android utilizadas no mercado;
2. Identificar os principais métodos de avaliação de arquitecturas de software;
3. Realizar inquérito e experimentos para recolha de dados;
4. Analisar e interpretar os dados recolhidos;
5. Propor recomendações para o uso adequado de cada arquitectura em diferentes tipos de aplicativos móveis.

1.5 Motivação

A motivação deste estudo reside na necessidade de desenvolver aplicações móveis de alta qualidade que atendam às expectativas dos utilizadores. Com a crescente popularidade e dependência de dispositivos móveis, é fundamental que os desenvolvedores compreendam as melhores práticas de arquitectura e desenvolvimento para criar aplicativos eficientes, escaláveis e seguros.

Outrossim, constituiu forte motivação de realização desta pesquisa, o facto de o investigador ter feito parte da equipa de implementação de uma das maiores aplicações de finanças de África (O Mpesa) e pôde constatar junto com a equipa as principais áreas que o desenvolvimento Android carecia de estudos actualizados.

O estudo oferece contribuições significativas, tanto teóricas quanto práticas, centrando-se na exploração aprofundada e na aplicação prática de métodos de avaliação de arquitecturas. Do ponto de vista teórico, este trabalho proporciona uma análise rigorosa dessas arquitecturas, destacando suas características, vantagens e desvantagens com base em métricas e critérios de avaliação bem definidos.

Do ponto de vista prático, o estudo oferece orientações aos desenvolvedores Android sobre como implementar e aplicar esses métodos de avaliação em seus próprios projectos. Isso não apenas auxilia na tomada de decisões informadas sobre a escolha entre MVVM e MVI, mas também capacita os desenvolvedores a adoptar abordagens mais fundamentadas e eficazes na avaliação de outras arquitecturas. Portanto, este estudo não apenas contribui para a compreensão teórica aprofundada das arquitecturas, mas também enriquece as práticas de desenvolvimento, capacitando os profissionais a aplicar métodos de avaliação de arquitecturas de forma mais eficiente e precisa.

1.6 Estrutura do Trabalho

A estrutura deste trabalho é composta por sete (7) capítulos, cada um desempenhando um papel específico na abordagem do tema em questão:

1. **Introdução (Capítulo I):** Este capítulo contextualiza o trabalho, fornecendo um enquadramento dos principais conceitos a serem abordados.
2. **Fundamentação Teórica (Capítulo II):** Aqui, são apresentados conceitos fundamentais relacionados às Tecnologias de Informação e Comunicações (TIC),

Android, bem como métodos de análise de arquitecturas. Isso estabelece as bases teóricas necessárias para o desenvolvimento do trabalho.

3. **Metodologia (Capítulo III):** Este capítulo descreve a metodologia utilizada para a realização do trabalho. Detalha os métodos de colecta de informações e o processo de análise e interpretação dos dados colectados.
4. **Caso de Estudo (Capítulo IV):** Neste capítulo, é apresentado o projecto prático desenvolvido como parte deste trabalho.
5. **Análise dos Resultados (Capítulo V):** Aqui, são apresentados os resultados das avaliações das arquitecturas estudadas, oferecendo insights sobre o cenário actual de desenvolvimento em Moçambique.
6. **Discussão (Capítulo VI):** Este capítulo discute os resultados obtidos à luz da literatura existente, identificando pontos de convergência e divergência.
7. **Conclusões e Recomendações (Capítulo VII):** Por fim, o Capítulo VII apresenta as conclusões derivadas da análise comparativa dos temas abordados no trabalho, além de fornecer recomendações para trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

No Capítulo II, será apresentada uma abrangente revisão de literatura que serviu como alicerce fundamental para orientar a investigação em questão. Esta revisão engloba uma variedade de conceitos essenciais relacionados aos temas centrais do trabalho. Serão discutidas definições relevantes de sistemas de informação, da engenharia de software, metodologias de avaliação de arquitecturas, destacando as abordagens utilizadas para analisar e comparar diferentes arquitectura de software, discutiremos também os principais componentes e padrões de desenho relevantes para o desenvolvimento Android, com foco especial nas arquitecturas mais comuns. Além disso, abordaremos as bases necessárias para realizar avaliações económicas das arquitecturas, compreendendo como factores financeiros podem influenciar as escolhas de desenho.

2.1 Sistemas de Informação (SI)

Refere-se a um conjunto organizado de componentes inter-relacionados que colectam, processam, armazenam e distribuem informações para apoiar a tomada de decisões, o controle de operações e outras actividades em uma organização. Esses sistemas desempenham um papel fundamental na gestão e no funcionamento de uma variedade de organizações, ajudando a melhorar a eficiência, a eficácia e a competitividade (Laudon & Laudon, 2016).

Segundo Stair & Reynolds (2006), SI é um conjunto de componentes que interagem entre si, recolhendo, manipulando e disseminando dados e informações, a fim de atingir um determinado objectivo.

Entretanto para O' Brien (2004), SI corresponde a um conjunto organizado composto dos seguintes elementos: pessoa, *hardware*, *software*, redes de comunicação e recursos de dados que irão colectar transformar e disseminar as informações em uma organização.

Com base nessas definições, podemos entender sistemas de informação como estruturas organizadas de componentes interconectados que colectam, processam, armazenam e distribuem informações para auxiliar tomadas de decisão e operações em organizações. Esses sistemas envolvem pessoas, hardware, software, redes de comunicação e recursos de dados para colectar, transformar e disseminar informações, melhorando a eficiência e a competitividade das organizações.

Tecnologias Móveis

De acordo com a IBM (2023), referem-se aos sistemas e infra-estruturas que permitem a comunicação e a conectividade sem fio. Isso inclui redes de telefonia móvel (como 3G, 4G e 5G), redes Wi-Fi, Bluetooth, GPS (Sistema de Posicionamento Global) e outras tecnologias semelhantes.

Essas tecnologias permitem que **dispositivos móveis** - aparelhos electrónicos portáteis projectados para uso em movimento, se comuniquem entre si e com serviços baseados na nuvem, possibilitando funções como chamadas telefónicas, troca de mensagens, navegação na *web*, *streaming* de mídia, acesso a aplicativos, rastreamento de localização e muito mais (IBM, 2023).

2.2 Engenharia de Software

Pressman (2010), define a engenharia de software como o estabelecimento e uso de princípios fundamentais da engenharia com o objectivo de desenvolver software de forma económica, confiável e eficiente, que funcione em máquinas reais.

De acordo com Coursera (2023), Engenharia de Software é o campo da ciência da computação dedicado ao planejamento, desenvolvimento, teste e manutenção de aplicações de software, onde engenheiros de software aplicam princípios de engenharia e expertise em linguagens de programação para conceber soluções de software para os usuários finais.

Já para o Economic Times (2023), a engenharia de software é um estudo detalhado da engenharia voltado para o projecto, desenvolvimento e manutenção de software. A engenharia de software foi introduzida para lidar com os problemas de projectos de software de baixa qualidade. Problemas surgem quando um software geralmente excede prazos, orçamentos e níveis reduzidos de qualidade. Ela assegura que a aplicação seja construída de forma consistente, correcta, dentro do prazo, orçamento e conforme os requisitos. A demanda pela engenharia de software também surgiu para atender à imensa taxa de mudança nos requisitos do usuário e no ambiente em que a aplicação deve operar.

Destas definições, entende-se que engenharia de software é a aplicação de princípios de engenharia para desenvolver software de forma económica, confiável e eficiente. Envolve planejamento, desenvolvimento, teste e manutenção de aplicações de software, com foco na satisfação dos requisitos dos usuários. Ela aborda problemas de qualidade, prazos e orçamentos, adaptando-se às mudanças nas necessidades do usuário e no ambiente de operação.

Software de computador

Consiste em instruções detalhadas e pré-programadas que controlam e coordenam os componentes de *hardware* do computador em um sistema de informação (Laudon & Laudon, 2016).

2.3 Qualidade de Software

O esforço para melhorar a qualidade do software começou assim que este se integrou em todos os aspectos das nossas vidas. Na década de 1990, as principais corporações reconheceram que bilhões de dólares estavam sendo desperdiçados a cada ano em software que não cumpria com as características nem a funcionalidade prometida (Pressman, 2010). Apesar dos esforços, o código defeituoso ainda é o pesadelo da indústria de *software*, sendo responsável por até 45% do tempo de inactividade de sistemas baseados em computadores e custando às empresas americanas cerca de $100 bilhões de dólares ao ano em perdas de produtividade e reparos, conforme afirma a Standish Group citado por Pressman (2010). Isso não inclui o custo de perder clientes insatisfeitos.

Num nível um tanto pragmático, David Garvin citado por Pressman (2010), sugere que "a qualidade é um conceito complexo e multifacetado" que pode ser descrito a partir de cinco pontos de vista diferentes.

I. **Ponto de vista transcendental -** diz que a qualidade é algo que é reconhecido imediatamente, mas que não pode ser definido explicitamente.
II. **Ponto de vista do usuário** - concebe a qualidade em termos das metas específicas do usuário final. Se um produto as satisfaz, ele tem qualidade.
III. **Ponto de vista do fabricante** – a define em termos das especificações originais do produto. Se ele as cumpre, tem qualidade.
IV. **Ponto de vista do produto** - sugere que a qualidade está relacionada às características intrínsecas (funções e características) de um produto.
V. **Ponto de vista baseado no valor** - mede de acordo com o que um cliente está disposto a pagar por um produto. Na realidade, a qualidade engloba tudo isso e mais.

A qualidade do desenho refere-se às características que os arquitectos especificam para um produto. O tipo de materiais, tolerâncias e especificações de desempenho, tudo contribui para a qualidade do desenho. Se melhores materiais forem utilizados, tolerâncias mais rigorosas forem estabelecidas e maiores níveis de desempenho forem especificados, a qualidade do desenho de um produto aumenta se ele for fabricado de acordo com as especificações.

Ainda de acordo com Pressman (2010), no desenvolvimento de *software*, a qualidade do desenho inclui o grau em que o desenho atende às funções e características especificadas no modelo de requisitos. A qualidade de conformidade concentra-se no grau em que a implementação adere ao desenho e em que o sistema resultante atende às suas metas de requisitos e desempenho.

Em última análise, Glass citado por Pressman (2010), sustenta que a qualidade é importante, mas se o usuário não estiver satisfeito, nada mais importa. Opinião reforçada pelo DeMarco citado por Pressman (2010), ao dizer que "a qualidade de um produto está relacionada a quanto ele muda o mundo para melhor". Esse ponto de vista sobre a qualidade afirma que se um produto de software beneficia muito os usuários finais, eles estarão dispostos a tolerar problemas ocasionais de confiabilidade ou desempenho.

2.3.1 Atributos de Qualidade de *Software*

Segundo Bass et al. (2013), existem sete atributos essenciais de *software*, a saber: **Disponibilidade**, **Interoperabilidade**, **Modificabilidade**, **Desempenho**, **Segurança**, **Testabilidade** e **Usabilidade**. Neste trabalho, colocou-se a atenção nos atributos de Modificabilidade, Testabilidade e Desempenho.

Modificabilidade

Modificabilidade envolve a capacidade de realizar alterações, e o foco nesse aspecto está relacionado aos custos e riscos associados às mudanças.

De acordo com Bass et al. (2013), para planejar a modificabilidade, o arquitecto deve abordar quatro questões:

1. **O que pode ser alterado:** Qualquer aspecto do sistema, incluindo funções, plataforma, ambiente operacional, qualidades demonstradas e capacidade.
2. **Qual é a probabilidade da mudança:** Deve-se considerar quais mudanças são prováveis e merecem suporte, evitando planejar para todas as possibilidades.
3. **Quando a mudança é feita e quem a faz:** Mudanças podem ocorrer durante diferentes fases, e os responsáveis podem ser desenvolvedores, usuários finais ou administradores do sistema.
4. **Qual é o custo da mudança:** Isso envolve a introdução de mecanismos para suportar mudanças e o custo real das modificações usando esses mecanismos.

Segundo Lou (2016), existem quatro actividades de actualização comuns nas aplicações Android: novos recursos, melhoria de recursos, correcção de erros (bugs) e melhoria de qualidade.

a) **Novos recursos** são características que não estavam no aplicativo anteriormente, mas foram adicionadas na nova versão.
b) A **melhoria de recursos** indica que o aplicativo já tinha esse recurso, mas na nova versão eles fazem algumas melhorias. A fronteira entre a melhoria da função e novos recursos não é estrita. Às vezes, a melhoria das funções requer a adição de novos recursos.
c) A **correcção de bugs** se refere ao comportamento para corrigir erros ou travamentos no aplicativo.
d) A **melhoria da qualidade** é para melhorar a qualidade do software, como velocidade e segurança.

Para este trabalho o cenário seleccionado para modificação é a adição de novos recursos e correcção de bugs.

Critérios de modificação

Bachmann et al. Citado por Lou (2016), acreditam que o acoplamento e a coesão influenciarão a modificação do software. Esta afirmação é reforçada por Bass et al. (2013) ao afirmarem que tácticas para reduzir o custo de fazer uma mudança incluem tornar os módulos menores, aumentar a coesão e reduzir o acoplamento.

O acoplamento refere-se à força das associações entre diferentes módulos. A coesão indica a força das relações dentro de um módulo específico. Reduzir o acoplamento e aumentar a coesão melhora a modificabilidade. Componentes com alta coesão e baixo acoplamento têm efeitos de propagação fracos ao fazer alterações. São reconhecidos sete níveis de coesão e seis níveis de acoplamento. As tabelas 1 e 2 resumem os respectivos níveis.

Tabela 1. Níveis de Coesão

Categoria	Nível de coesão	Características
Alto nível	Funcional	Um componente executa uma única tarefa.
	Sequencial	Um item de dados é transferido sequencialmente entre tarefas dentro de um componente.
	Comunicacional	Todas as tarefas dentro de um componente compartilham os mesmos itens de dados de entrada ou saída.
	Processual	As tarefas dentro de um componente são conectadas por conectores de controle.

Níveis moderados	Temporal	As tarefas dentro de um componente são correlacionadas por relações temporais.
	Lógico	Tarefas que são agrupadas logicamente para realizar o mesmo tipo de funcionalidades.

Tabela 2. Níveis de Acoplamento

Categoria	Nível de acoplamento	Características
Alto nível	Contente	Um componente usa dados ou controle mantidos por outro componente.
	Comum	Os componentes compartilham itens de dados globais.
	Externo	Os componentes estão vinculados a entidades externas, como dispositivos ou dados externos.
Níveis moderados	controle	Controla o fluxo entre os componentes.
Níveis baixos	Dados estruturados	Os dados estruturados são transferidos entre componentes.
		Dados primitivos ou matrizes de dados primitivos são passados entre componentes.
		Os componentes se comunicam por meio de interfaces.

Nesta tese, para avaliar a modificabilidade da arquitectura, avaliou-se o nível de coesão e acoplamento de cada arquitectura de acordo com a descrição de níveis acima mencionados.

Desempenho

Trata-se de tempo e da capacidade do sistema de software de atender aos requisitos de tempo. Quando eventos ocorrem, como interrupções, mensagens, solicitações de usuários ou de outros sistemas, ou eventos de relógio marcando a passagem do tempo, o sistema ou algum elemento do sistema deve responder a eles a tempo. Caracterizar os eventos que podem ocorrer (e quando podem ocorrer) e a resposta baseada em tempo do sistema ou do elemento a esses eventos é a essência da discussão sobre desempenho (Bass et al., 2013).

Para Lou (2016), o desempenho é definido como "o grau em que um sistema ou componente realiza suas funções desenhadas dentro de limitações estabelecidas, como velocidade, precisão

ou uso de memória". Para melhorar o desempenho, de acordo com Bass et al. (2013) existem duas abordagens principais: a redução da demanda e a gestão mais eficiente dos recursos disponíveis.

A redução da demanda pode ter o efeito colateral de diminuir a fidelidade ou recusar o atendimento a algumas solicitações. Por outro lado, o gerenciamento mais eficaz dos recursos pode ser alcançado por meio de técnicas como agendamento, replicação ou simplesmente aumentando os recursos disponíveis.

Ao avaliar o desempenho de aplicativos Android, de acordo com a Android Developers (2023), um problema comum é a renderização lenta da interface do usuário, também conhecida como "*jank*". A renderização da interface do usuário envolve a geração de quadros dentro de um aplicativo e sua exibição na tela. Se um aplicativo sofre de renderização lenta da *interface* do utilizador, o sistema pode ser forçado a pular quadros, resultando em uma experiência perceptivelmente irregular (*stuttering*) para o usuário. Além disso, é importante notar que um aplicativo deve ser capaz de renderizar quadros em menos de 16 milissegundos para alcançar uma taxa de quadros de 60 quadros por segundo (fps), porque a maioria das pessoas não percebe benefícios além dessa taxa (Android Developers, 2023b).

Outro problema de desempenho da interface do usuário é a lentidão na resposta durante o tempo de inicialização do aplicativo. Os usuários normalmente esperam que os aplicativos sejam responsivos e tenham tempos de inicialização rápidos. Um tempo de inicialização lento pode resultar em uma experiência de usuário insatisfatória e até mesmo levar os usuários a desinstalar o aplicativo devido à sua qualidade decepcionante (Android Developers, 2023b).
O uso excessivo da CPU é um factor adicional que pode afectar o desempenho da interface do usuário de um aplicativo. Se a CPU estiver sobrecarregada, é um sinal de que o processador está sobrecarregado naquele momento. Isso pode afectar tanto o tempo de resposta quanto o tempo de carregamento do aplicativo, resultando em quadros congelados ao alternar entre telas ou problemas de responsividade durante a interacção com o aplicativo (PFLB, 2023).

Benchmarks **e Métricas**

De acordo com a Android Developers (2023), Benchmarking é uma maneira de inspeccionar e monitorar o desempenho do seu aplicativo. O desenvolvedor pode executar *benchmarks* regularmente para analisar e depurar problemas de desempenho e garantir que não introduza regressões nas alterações recentes.

O Android oferece duas bibliotecas e abordagens de benchmarking para analisar e testar diferentes tipos de situações no seu aplicativo: ***Macrobenchmark*** e ***Microbenchmark*** (Android Developers, 2023b).

a) Macrobenchmark

Essa abordagem é usada para testar e analisar interacções maiores e cenários mais amplos do seu aplicativo, como o tempo de inicialização do aplicativo, manipulações complexas da interface do usuário, rolagem de listas e navegação entre *layouts* diferentes (Android Developers, 2023).

Com o *Macrobenchmark*, é possível especificar o número de iterações e o tipo de métrica, que especifica o principal tipo de informação que será capturada. Para medir o tempo de inicialização do aplicativo, é usado o ***Startup Timing Metric***. Com essa métrica, o tempo até a exibição inicial é medido em milissegundos, o que descreve o tempo necessário para que um aplicativo seja renderizado na tela, incluindo a inicialização do processo e a criação da actividade (Android Developers, 2023b).

Outra métrica importante é ***Frame Timing Metric*** é usada para medir quão rápido um aplicativo pode produzir quadros, também conhecido como tempo de renderização. O *FrameTimingMetric* inclui duas medições específicas: ***frameDurationCpuMs*** e ***frameOverrunMs***. O *frame Duration pcms* mede o tempo necessário para produzir um quadro na CPU, enquanto o *frame Overruns* mede quanto tempo um quadro específico excede a duração desejada ou esperada. Números positivos da medição *frame Overruns* indicam um quadro perdido e "*jank*" (renderização lenta) ou "*stutter*" (irregular) visível. Números negativos indicam o tempo em que um quadro foi mais rápido do que o prazo esperado (Android Developers, 2023b).

b) Microbenchmark

Esta abordagem é destinada a testar e analisar operações menores e mais específicas em seu aplicativo, como o desempenho de funções ou métodos individuais.

Essas ferramentas de benchmarking são valiosas para assegurar que seu aplicativo mantenha um desempenho consistente e para identificar áreas de melhoria em termos de eficiência e velocidade. Elas desempenham um papel fundamental na manutenção da qualidade e da responsividade do seu aplicativo à medida que você realiza modificações e actualizações (Android Developers, 2023b).

Testabilidade

As estimativas da indústria de software indicam que entre 30 e 50 por cento (ou, em alguns casos, até mais) do custo de desenvolvimento de sistemas bem projectados é destinado aos testes. Se o arquitecto de software puder reduzir esse custo, o benefício é significativo (Bass et al., 2013).

Segundo Bass et al. (2013) testabilidade de *software* se refere à facilidade com que o *software* pode demonstrar suas falhas por meio de testes (tipicamente baseados na execução). Especificamente, a testabilidade se relaciona com a probabilidade, assumindo que o software tenha pelo menos uma falha, de que ele falhará em sua próxima execução de teste. De maneira intuitiva, um sistema é testável se "entregar" suas falhas facilmente. Se uma falha estiver presente em um sistema, queremos que ela falhe durante os testes o mais rápido possível.

O guia de desenvolvimento Android (Android Developers, 2021) categoriza quatro tipos de testes: **testes de unidade locais**, **testes de unidade instrumentados**, **testes de componentes** dentro do aplicativo e **testes entre aplicativos**. Para cada tipo de teste, existem bibliotecas de suporte específicas.

a) **Testes de unidade local** e os **Testes instrumentados** são ambos considerados Testes de Unidade, pois se concentram em testar partes específicas do aplicativo.
   Os testes instrumentais são executados em um dispositivo Android, seja físico ou emulado. O aplicativo é construído e instalado junto com um aplicativo de teste que injecta comandos e lê o estado. Geralmente, os testes instrumentados são testes de *interface* do usuário (UI), onde o aplicativo é iniciado e, em seguida, interage com ele.

   Já os testes locais são executados em sua máquina de desenvolvimento ou em um servidor, por isso também são chamados de testes do lado do hospedeiro. Eles geralmente são pequenos e rápidos, isolando o objecto de teste do restante do aplicativo. Conforme ilustra a figura 2.

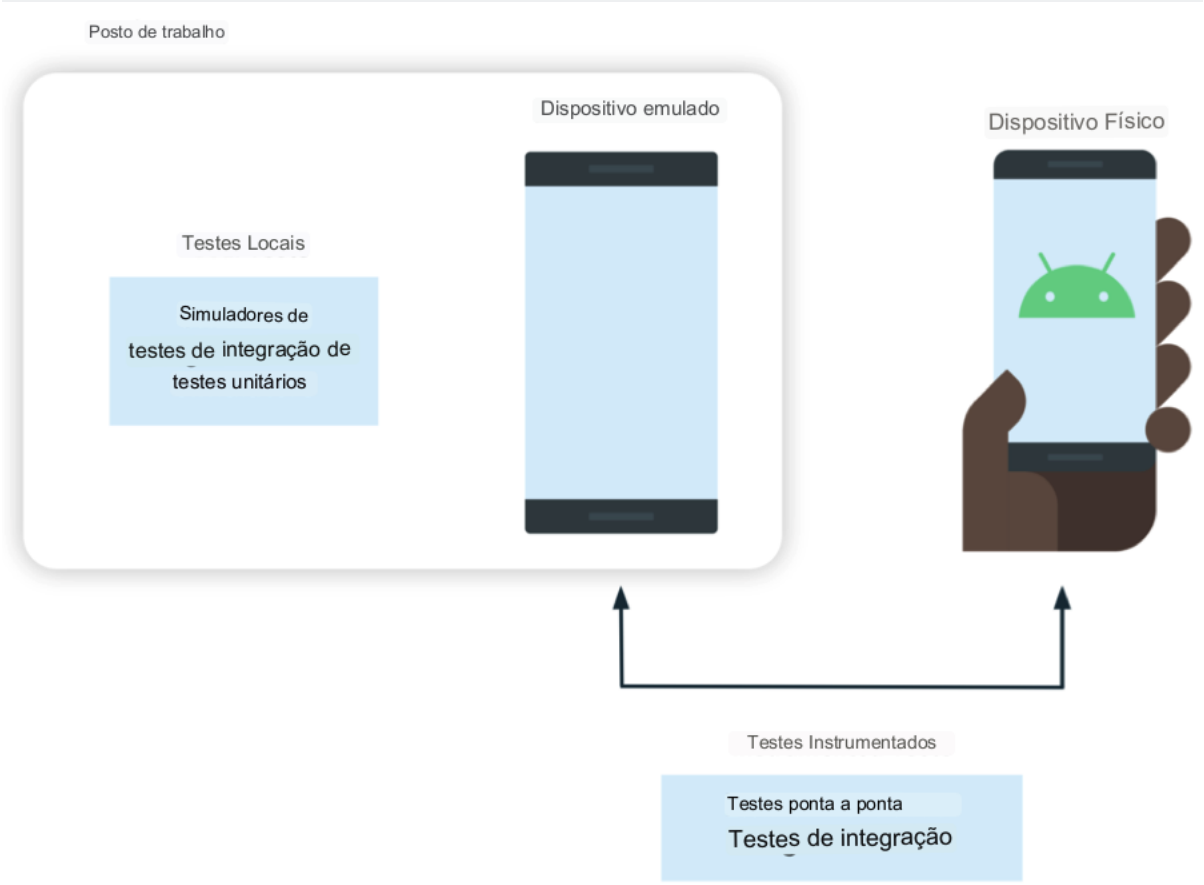


Figura 2. Testes unitários locais vs testes instrumentados (adaptado de Android Developers, 2021).

b) Testes de componentes dentro do aplicativo se concentram nas interacções entre o usuário e a *interface* do usuário. Esse tipo de teste verifica se a saída é a esperada quando o usuário interage com elementos da *interface*, como clicar em botões ou inserir dados.

c) Testes entre aplicativos se concentram em testar a interacção entre diferentes aplicativos. Por exemplo, alguns aplicativos permitem que os usuários adicionem eventos ao calendário do sistema, enquanto outros podem controlar o brilho da tela. Estes testes garantem que a integração entre aplicativos seja eficaz e livre de problemas.

De acordo com Lou, (2016), até o momento, diversos factores de testabilidade foram identificados, tais como observabilidade, controlabilidade, tamanho dos casos de teste, esforço para construir os casos de teste, tempo gasto para a execução dos casos de teste, facilidade na interpretação dos resultados dos testes e facilidade na identificação de erros.

Para esta tese, serão considerados três critérios principais para avaliar a testabilidade:

1. **Tamanho dos casos de teste**: A arquitectura com menos casos de teste é melhor.

2. **Tempo consumido para executar os casos de teste**: A aplicação com menos testes instrumentados consome menos tempo.
3. **Facilidade de depuração com pontos de interrupção**: Facilidade para os desenvolvedores depurar a aplicação com pontos de interrupção.

De acordo com Humeniuk (2018) a testabilidade de uma estrutura de código é uma métrica crucial para avaliar a capacidade do software de ser testado de forma eficaz. Uma das métricas mais utilizadas é a "cobertura de testes de código", que indica quais partes do código são cobertas por testes e quais não são. Essa métrica é geralmente apresentada como uma percentagem do código total que é testado.

Vantagens da Cobertura de Testes:

- **Avaliação da Estrutura de Classes:** A cobertura de testes permite avaliar como as classes estão estruturadas e como as responsabilidades estão distribuídas entre elas. Uma alta cobertura de testes sugere que o projecto possui baixo acoplamento e que as classes desempenham papéis específicos no sistema.
- **Compreensão do Sistema:** Uma boa cobertura de testes ajuda os desenvolvedores a compreender melhor o sistema, pois os testes funcionam como uma documentação viva do comportamento do código.
- **Facilidade de Manutenção:** Um código com alta cobertura de testes é mais fácil de manter, pois as mudanças podem ser feitas com mais confiança, sabendo que os testes ajudarão a identificar problemas.

Desafios da Avaliação de Arquitectura apenas com Cobertura de Testes:

- **Qualidade dos Testes:** Ter uma alta cobertura de testes não garante que os testes sejam de alta qualidade ou que validem o comportamento desejado do sistema. Os testes podem ser mal concebidos ou não capturar cenários críticos.
- **Foco nos Testes:** A ênfase excessiva na cobertura de testes pode desviar o foco do objectivo principal da arquitectura, que é implementar o comportamento correcto dos componentes.

2.3.2 Avaliação de Qualidade de *Software*

David Garvin citado por Pressman (2010) sugere que a qualidade deve ser considerada adoptando uma abordagem multidimensional que comece com a avaliação da conformidade e termine com uma visão transcendental (estética). Embora as oito dimensões de qualidade de

Garvin não tenham sido desenvolvidas especificamente para o *software*, elas se aplicam à qualidade deste:

a) **Qualidade do Desempenho** - O *software* entrega todo o conteúdo, funcionalidades e características especificadas no modelo de requisitos, proporcionando valor ao usuário final?

b) **Qualidade das Características** - O *software* possui características que surpreendem e agradam os utilizadores finais quando usado pela primeira vez?

c) **Confiabilidade** - O *software* fornece todas as características e capacidades sem falhas? Está disponível quando necessário? Entrega funcionalidade livre de erros?

d) **Conformidade** - O *software* está em conformidade com os padrões locais e externos relevantes para a aplicação? Está de acordo com o desenho de facto e convenções de código? Por exemplo, a *interface* do usuário está de acordo com as regras aceitas de desenho para selecção de menus ou entrada de dados?

e) **Durabilidade** - O *software* pode ser mantido (alterado) ou corrigido (depurado) sem gerar eventos colaterais inadvertidos? As mudanças causarão uma diminuição na taxa de erros ou na confiabilidade ao longo do tempo?

f) **Serviço** - Existe a possibilidade de o software receber manutenção (alterações) ou correcções (depuração) em um período de tempo razoavelmente curto? A equipe de suporte pode obter todas as informações necessárias para fazer alterações ou corrigir defeitos? Douglas Adams citado por Pressman (2010) faz um comentário irónico que parece relevante: "A diferença entre algo que pode dar errado e algo que possivelmente não dará errado é que, quando o último der errado, geralmente é impossível corrigi-lo ou repará-lo".

g) **Estética** - Não há dúvida de que todos temos uma visão diferente e altamente subjectiva do que é estético. Ainda assim, a maioria concordaria que uma entidade estética possui uma certa elegância, um fluxo único e uma "presença" evidente que é difícil de

quantificar, mas que é claramente percebida. O *software* estético possui essas características.

h) **Percepção** - Em certas situações, existem preconceitos que influenciarão a percepção da qualidade pelo usuário. Por exemplo, se um produto de *software* for lançado por um fornecedor que no passado demonstrou baixa qualidade, haverá desconfiança e a percepção da qualidade do produto será negativamente influenciada. Da mesma forma, se um vendedor tiver uma reputação excelente, será percebida boa qualidade, mesmo que esta não exista na realidade.

As dimensões de qualidade de Garvin proporcionam uma visão "suave" da qualidade do software. Muitas dessas dimensões (embora nem todas) só podem ser consideradas de maneira subjectiva. Por essa razão, também é necessário um conjunto de factores “rígidos" de qualidade que se dividem em dois grandes grupos: 1) factores que podem ser medidos directamente (por exemplo, defeitos não descobertos durante os testes) e 2) factores que só podem ser medidos indirectamente (como a usabilidade ou a facilidade de manutenção). Em cada caso, medições devem ser feitas: o *software* deve ser comparado com algum dado para chegar a um indicador de qualidade.

2.4 Arquitectura de *Software*

Existem muitas definições de arquitectura de *software*, mas Bass et al. (2013), dão uma definição completa dizendo que arquitectura de software de um sistema é o conjunto de estruturas necessárias para raciocinar sobre o sistema, que compreendem elementos de software, relações entre eles e propriedades de ambos.

Bass et al., (2013), explicam as implicações dessa definição em cinco afirmações:

a) **Arquitectura é um Conjunto de Estruturas de *Software***

Uma estrutura é simplesmente um conjunto de elementos mantidos juntos por uma relação. Sistemas de *software* são compostos por muitas estruturas, e nenhuma estrutura única pode ser considerada como sendo a arquitectura.

Embora o *software* contenha um suprimento infinito de estruturas, nem todas são arquitecturais. Uma estrutura é arquitectural se ela suporta o raciocínio sobre o sistema e as propriedades do sistema. O raciocínio deve ser sobre um atributo do sistema que é importante para alguma parte interessada. Isso inclui funcionalidade alcançada pelo sistema, a

disponibilidade do sistema diante de falhas, a dificuldade de fazer mudanças específicas no sistema, a capacidade de resposta do sistema a solicitações do usuário e muitos outros.

b) Arquitectura é uma Abstracção

Porque a arquitectura consiste em estruturas e as estruturas consistem em elementos e relações, segue-se que uma arquitectura compreende elementos de software e como esses elementos se relacionam entre si. Isso significa que a arquitectura especificamente omite certas informações sobre elementos que não são úteis para raciocinar sobre o sistema - em particular, ela omite informações que não têm ramificações fora de um único elemento. Assim, uma arquitectura é, acima de tudo, uma abstracção de um sistema que selecciona certos detalhes e suprime outros. Em todos os sistemas modernos, os elementos interagem entre si por meio de interfaces que dividem os detalhes sobre um elemento em partes públicas e privadas. A arquitectura diz respeito ao lado público dessa divisão; detalhes privados dos elementos - detalhes relacionados apenas à implementação interna - não são arquitecturais. Além das interfaces, no entanto, a abstracção arquitectural nos permite observar o sistema em termos de seus elementos, como eles estão dispostos, como interagem, como são compostos, quais são suas propriedades que sustentam nosso raciocínio sobre o sistema, e assim por diante. Essa abstracção é essencial para domar a complexidade de um sistema - simplesmente não podemos e não queremos lidar com toda a complexidade o tempo todo.

c) Todo sistema de software possui uma arquitectura de *software*

Todo sistema pode ser mostrado como composto por elementos e relações entre eles para suportar algum tipo de raciocínio. No caso mais trivial, um sistema é ele próprio um único elemento - uma arquitectura desinteressante e provavelmente não útil, mas ainda assim uma arquitectura.

No entanto, mesmo que todo sistema possua uma arquitectura, não necessariamente isso significa que a arquitectura é conhecida por alguém. Talvez todas as pessoas que projectaram os sistemas já tenham ido embora, a documentação tenha desaparecido (ou nunca tenha sido produzida), o código-fonte tenha se perdido (ou nunca tenha sido entregue), e tudo o que temos seja o código binário em execução. Isso revela a diferença entre a arquitectura de um sistema e a representação dessa arquitectura. Como uma arquitectura pode existir independentemente de sua descrição ou especificação, isso destaca a importância da documentação da arquitectura.

d) A Arquitectura Inclui Comportamento

O comportamento de cada elemento faz parte da arquitectura, na medida em que esse comportamento pode ser usado para raciocinar sobre o sistema. Esse comportamento incorpora como os elementos interagem entre si, o que claramente faz parte da nossa definição de arquitectura.

Isso nos diz que desenhos de caixas e linhas que são apresentados como arquitecturas na verdade não são arquitecturas de forma alguma. Ao olhar para os nomes das caixas (banco de dados, interface gráfica do usuário, executável, etc.), um leitor pode imaginar a funcionalidade e o comportamento dos elementos correspondentes. Essa imagem mental se aproxima de uma arquitectura, mas ela surge da imaginação da mente do observador e depende de informações que não estão presentes. Isso não significa que o comportamento exacto e o desempenho de cada elemento precisem ser documentados em todas as circunstâncias - alguns aspectos do comportamento são detalhados demais e estão abaixo do nível de preocupação do arquitecto. No entanto, na medida em que o comportamento de um elemento influencia outro elemento ou influencia a aceitabilidade do sistema como um todo, esse comportamento deve ser considerado e documentado como parte da arquitectura de *software*.

e) Nem Todas as Arquitecturas São Boas Arquitecturas

A definição é indiferente quanto a se a arquitectura de um sistema é boa ou ruim. Uma arquitectura pode permitir ou impedir que um sistema alcance seus requisitos de comportamento, atributos de qualidade e ciclo de vida.

2.5 Avaliação de Arquitecturas de *Software*

Uma arquitectura de software bem elaborada é um dos principais factores que contribuem para o sucesso de um projecto de software (Lou, 2016). No ciclo de vida do desenvolvimento de software, encontrar problemas precocemente é fundamental, uma vez que a correcção se torna mais onerosa com o tempo.

Conforme destacado por Lou (2016), a fase de desenho da arquitectura é particularmente propícia para identificar problemas potenciais. A qualidade do software é significativamente influenciada pela arquitectura, tornando a avaliação da arquitectura um elemento crucial no desenvolvimento de software.

Essa perspectiva é reiterada por Bass et al. (2013), que ressaltam como a arquitectura desempenha um papel tão crucial no sucesso do sistema e do projecto de engenharia de software, justificando a necessidade de pausar e assegurar que a arquitectura planejada seja capaz de cumprir todas as expectativas.
Assim, a avaliação de arquitectura pode ser definida como o processo de determinar se uma arquitectura é adequada para o propósito a que se destina (Bass et al., 2013).

A avaliação de arquitectura desempenha um papel crucial ao garantir que a estrutura do sistema esteja alinhada com os objectivos de negócios e os requisitos de qualidade do software. Nesse sentido, pesquisadores têm desenvolvido uma variedade de métodos de avaliação, que podem ser categorizados em quatro principais abordagens: baseadas em cenários, baseadas em simulação, baseadas em experiência e modelagem matemática.
De acordo com Lou (2016), entre as diversas abordagens, o método de avaliação baseado em cenários é o mais consolidado. Recentemente, foram desenvolvidos diversos novos métodos de avaliação de arquitectura de software baseados em cenários por diferentes grupos académicos, muitos dos quais foram publicados em forma de livros ou teses de doutorado. Muitos desses métodos representam aprimoramentos dos já conhecidos **SAAM** (**Método de Análise de Arquitectura de Software**) ou **ATAM** (**Método de Análise de Compensação Arquitectural**), e alguns exemplos incluem o **CBAM** (**Método de Análise de Custo-Benefício**), **ALMA** (**Método de Análise de Modificabilidade em Nível de Arquitectura**) e **FAAM** (**Método de Análise de Arquitectura Familiar**) (Ionita et al., 2002). Destes, apenas 2 se destacam, o ATAM e CBAM.

2.5.1 Método de Análise de Compensação Arquitecturais (Architecture Tradeoffs Analysis Method - ATAM)

O ATAM é um método para avaliar projectos em nível de arquitectura que leva em consideração diversos atributos de qualidade, tais como modificabilidade, desempenho, confiabilidade e segurança, com o intuito de obter insights sobre se a implementação da arquitectura atenderá aos requisitos estabelecidos. O método identifica pontos de equilíbrio entre esses atributos, facilita a comunicação entre as partes interessadas (como usuários, desenvolvedores, clientes e mantenedores) a partir da perspectiva de cada atributo, esclarece e aprimora requisitos, além de oferecer uma estrutura para um processo contínuo e simultâneo de desenho e análise do sistema (Kazman et al., 1998).

Essa abordagem é reforçada por Ionita et al. (2002), que descrevem o ATAM como um método de arquitectura baseado em cenários, cujo foco é avaliar atributos de qualidade como modificabilidade, portabilidade, extensibilidade e integrabilidade. Além da avaliação desses atributos, segundo os autores, o ATAM explora as interacções entre eles e suas interdependências, destacando mecanismos de compensação e oportunidades entre várias qualidades.

Etapas de Avaliação ATAM

O método é dividido em quatro principais áreas de actividades: colecta de cenários e requisitos, visões arquitectónicas e realização de cenários, construção de modelos e análises, e compensações (Kazman et al., 1998). O método funciona como ilustrado na figura 3:

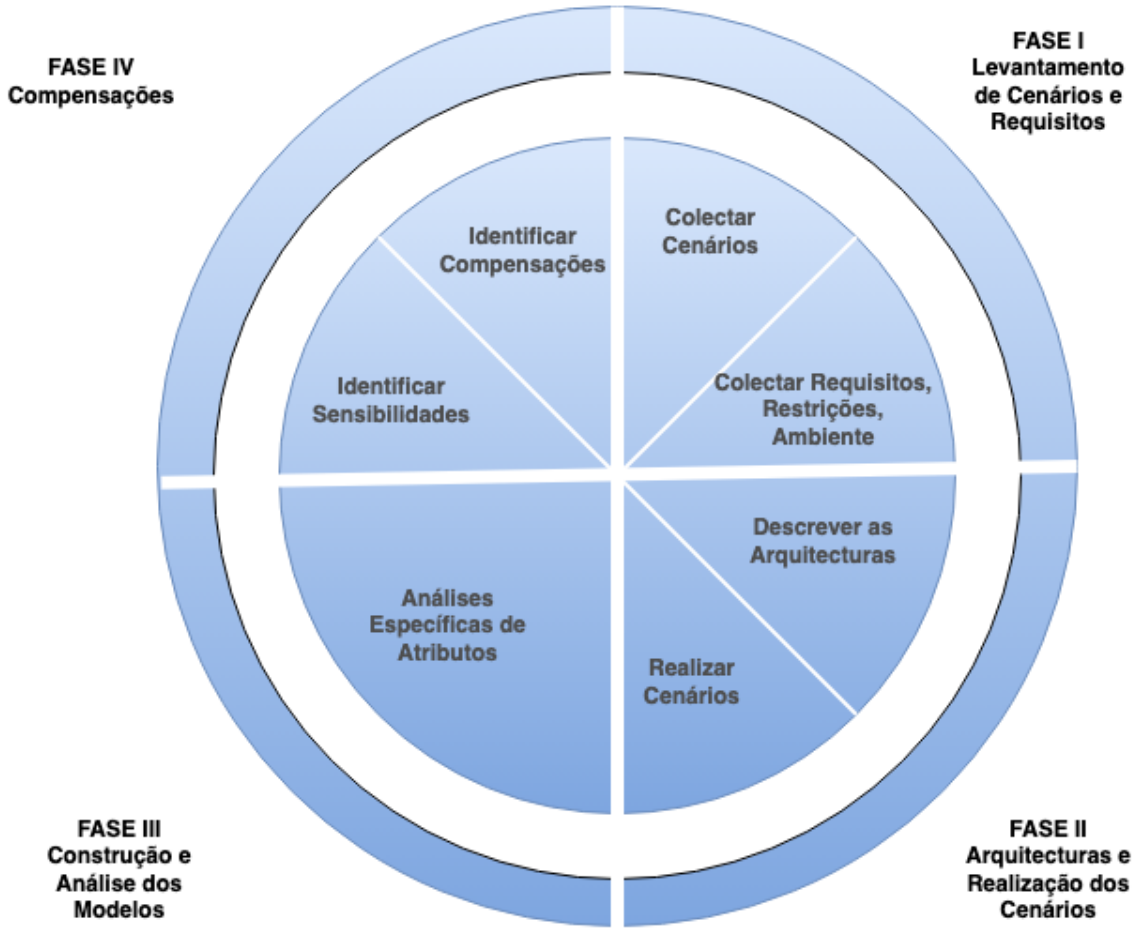


Figura 3. Fases do Método de Análise de Compensação de Arquitectura (Adaptado de Kazman et al., 1998).

a) Colectar Cenários

Os primeiros passos do método - a licitação de cenários de uso do sistema e a colecta de requisitos, restrições e detalhes ambientais de um grupo representativo de partes interessadas - são intercambiáveis. Em determinados momentos, os requisitos existem antes mesmo de

qualquer análise arquitectural ser iniciada. Em outros casos, são os cenários que orientam a definição dos requisitos. O processo de análise pode ser iniciado com qualquer um desses passos de forma vantajosa. A licitação de cenários tem como objectivo operacionalizar tanto os requisitos funcionais quanto os de qualidade, além de facilitar a comunicação entre as partes interessadas e desenvolver uma visão compartilhada das actividades importantes que o sistema deve suportar.

Lou (2016) resumiu esta etapa ao afirmar que nela são identificados os cenários de uso do sistema que são comuns e relevantes. É explorada a natureza dos problemas que o sistema irá abordar e como ele irá executar acções para resolver esses problemas. Essa etapa é executada com base no contexto específico em que o sistema irá operar no mundo real.

b) Colectar Requisitos/Restrições/Ambiente

Na etapa de colecta de requisitos, é essencial identificar os requisitos, restrições e ambiente com base nos atributos do sistema. Os requisitos podem variar de valores específicos a cenários hipotéticos. O ambiente e as restrições são registrados, uma vez que afectam análises posteriores. Os requisitos funcionam como critérios de avaliação e representam as características mais significativas do sistema, acordadas por todas as partes interessadas. No ATAM, esses requisitos são conhecidos como "requisitos baseados em atributos", priorizando diferentes qualidades de software em projectos distintos (Kazman et al., 1998; Lou, 2016).

c) Descrever Visões Arquitecturais

De acordo com Lou (2016), nesta etapa, são apresentadas várias arquitecturas concorrentes. Cada descrição de arquitectura deve explicar o componente central e os atributos que são relevantes para avaliar as qualidades.

Kazman et al. (1998) explicam que requisitos, cenários e princípios de projecto de engenharia influenciam a criação de arquitecturas, estabelecendo limites para as opções de design. Os sistemas existentes, interoperabilidade e experiências passadas também restringem as possíveis arquitecturas. As arquitecturas candidatas devem descrever elementos e propriedades relevantes para atributos de qualidade, como confiabilidade, desempenho e segurança, frequentemente representados em diferentes visões arquitectónicas.

d) Análises Específicas de Atributos

Após a definição dos requisitos e da arquitectura inicial, a análise dos atributos de qualidade ocorre separadamente para cada atributo e cada arquitectura, sem interacções entre eles. Isso

permite que especialistas em atributos individuais contribuam com suas experiências (Kazman et al., 1998).

De acordo com Lou (2016), experimentos empíricos são realizados para tornar os resultados claros e convincentes. O método ATAM não prescreve um método de análise específico para cada atributo, tornando o processo flexível, mas potencialmente complicado, pois os arquitectos precisam definir seus próprios métodos para cada qualidade a ser analisada.

e) Identificar Sensibilidades

Nesta etapa, é determinada a sensibilidade das análises individuais dos atributos em relação a elementos arquitectónicos específicos. Isso envolve a variação de um ou mais atributos da arquitectura; os modelos são então ajustados para reflectir essas mudanças de desenho, e os resultados são avaliados. Quaisquer valores modelados que sejam significativamente afectados por uma mudança na arquitectura são considerados pontos de sensibilidade (Kazman et al., 1998).

Em outras palavras, após a conclusão da etapa IV, analisamos as interacções entre os atributos de qualidade (na etapa II) e os componentes da arquitectura (na etapa III). A partir dessa etapa, ganhamos a compreensão de que tipo de mudanças na arquitectura impactarão os atributos de qualidade (Lou, 2016).

f) Identificar Compensações

O próximo passo do método envolve a análise crítica dos modelos arquitecturais construídos, identificando os pontos de compensação arquitectural, focando na interacção entre análises de atributos. Os pontos de sensibilidade arquitectural são determinados, identificando elementos sensíveis a vários atributos. Por exemplo, o número de servidores em uma arquitectura cliente-servidor é um ponto de compensação, afectando desempenho, disponibilidade e segurança. Esses pontos são áreas de possíveis compensações arquitecturais (Kazman et al., 1998).

Compensações são, de acordo com Lou (2016) uma representação dos benefícios e prejuízos associados à escolha de uma arquitectura. Elas também descrevem as relações entre os atributos de qualidade e os componentes da arquitectura. No entanto, ao contrário da Etapa V, essa relação é de um para muitos.

Após completar as etapas mencionadas, é possível comparar os resultados das análises com os requisitos. Segundo (Kazman et al., 1998) se as análises indicarem que o comportamento

previsto do sistema está suficientemente próximo dos requisitos, os arquitectos podem avançar para um nível mais detalhado de desenho ou para a implementação. Contudo, na prática, é vantajoso continuar monitorando a arquitectura com modelos analíticos para apoiar o desenvolvimento, a implantação e até mesmo a manutenção. O desenho nunca é finalizado no ciclo de vida de um sistema, e a análise também não deve ser interrompida. Caso a análise revele algum problema, é elaborado um plano de acção para modificar a arquitectura, os modelos ou os requisitos. Esse plano de acção se baseia nas análises específicas de atributos e na identificação de pontos de compensação. Isso culmina em uma nova iteração do método.

2.5.2 Método de Análise de Custo-Benefício (*Cost Benefit Analysis Method* - CBAM)

CBAM dá continuidade ao ponto em que o ATAM finaliza, constituindo um método focado na arquitectura para analisar as implicações de custos, benefícios e cronograma das decisões arquitectónicas conforme ilustrado na figura 4. Além disso, o CBAM avalia o grau de incerteza associado a essas avaliações, a fim de fornecer uma base sólida para um processo de tomada de decisão informado em relação à arquitectura (Ionita et al., 2002).

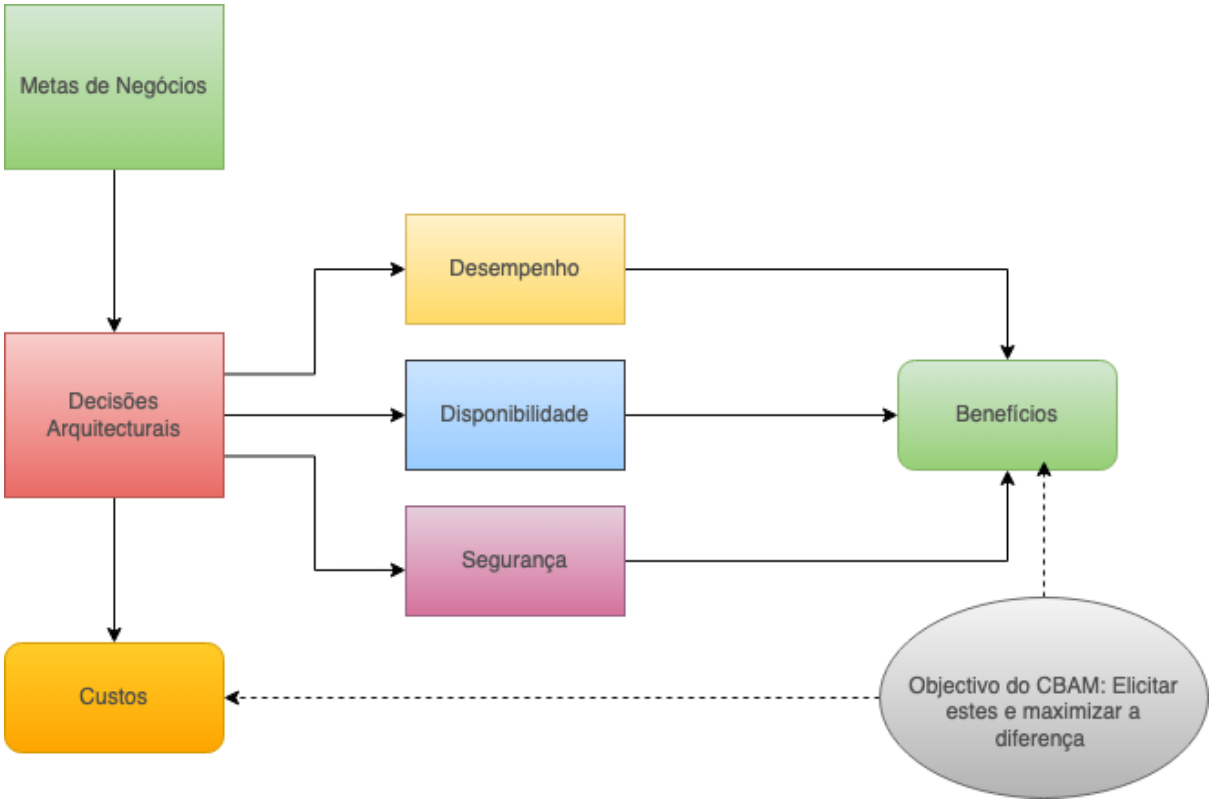


Figura 4. Contexto do processo do CBAM (adaptado do Ionita et al., 2002).

Conforme destacado por Ionita et al. (2002), o CBAM se diferencia dos métodos anteriores ao estabelecer uma conexão entre dois domínios no desenvolvimento de software: o processo de arquitectura e a economia da organização. O CBAM incorpora os custos (juntamente com os

orçamentos ou recursos financeiros implícitos) como atributos de qualidade, que devem ser considerados em meio às compensações quando se planeja um sistema de software. Enquanto SAAM e ATAM consideram principalmente atributos de qualidade arquitectónica, como modificabilidade, desempenho, disponibilidade e usabilidade nas decisões de desenho, o CBAM reconhece que custos, benefícios e riscos são igualmente importantes. Esses factores devem ser considerados durante o processo de tomada de decisões arquitecturais.

Etapas de Avaliação CBAM

O CBAM consiste em duas fases. A primeira fase é chamada de triagem, seguida por uma segunda fase chamada de exame detalhado. A primeira fase é às vezes necessária no caso de haver muitas estratégias arquitectónicas a serem discutidas e apenas algumas devem ser escolhidas para um exame detalhado posterior. Caso contrário, o processo de avaliação começa directamente na segunda fase (Ionita et al., 2002). Para ambas as fases do CBAM, são prescritas seis etapas principais conforme mostra a figura 5:

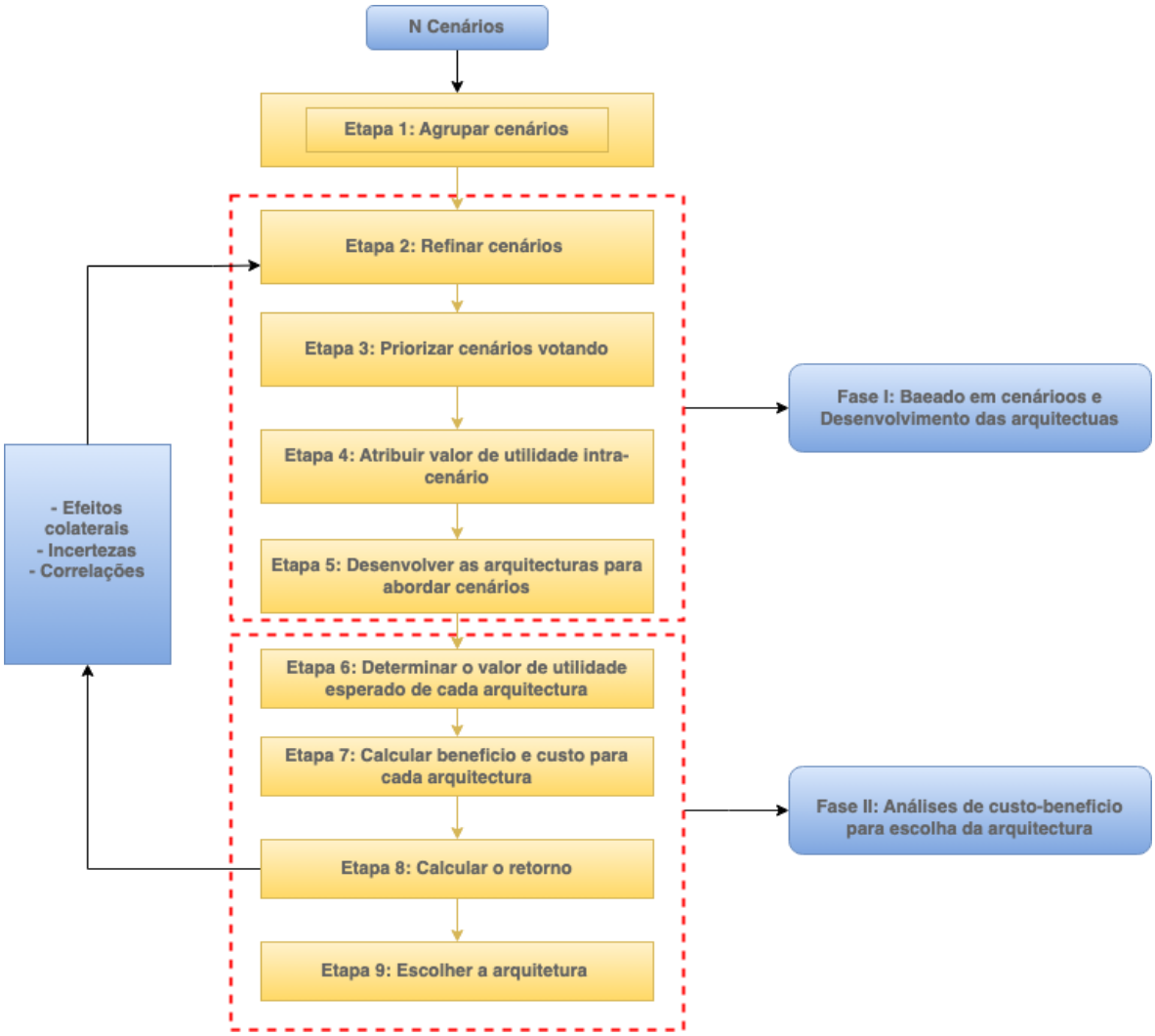


Figura 5. Etapas do método de avaliação CBAM (adaptado do Ionita et al., 2002).

a) Agrupar Cenários

Na primeira etapa, são escolhidos os cenários que mais preocupam as partes interessadas do sistema. Para cada um desses cenários, são propostas diferentes estratégias arquitectónicas que abordam os cenários específicos (Ionita et al., 2002).

b) Refinar Cenários

Na segunda etapa, refinam-se os cenários seleccionados na etapa 1, direccionando a atenção para as medidas de estímulo e resposta. Obtém-se o nível de resposta desejado do atributo de qualidade para cada cenário, abrangendo os piores casos, o cenário actual, e o melhor cenário possível (Bass et al., 2013).

c) Priorizar Cenários

Na terceira etapa, priorizam-se os cenários refinados com base nos votos dos stakeholders. Atribui-se um peso de 10 ao cenário mais bem avaliado; e aos demais cenários um peso relativo ao mais bem avaliado. Isso se torna o peso usado no cálculo do benefício geral de uma estratégia (Bass et al., 2013).

d) Atribuir Valor de Utilidade

Na quarta etapa, define-se a utilidade para cada nível de resposta do atributo de qualidade (pior caso, actual, desejado, melhor caso) para os cenários da etapa 3. Esses valores de utilidade podem ser organizados de forma conveniente em uma tabela, com uma linha para cada cenário e uma coluna para cada um dos quatro níveis de resposta do atributo de qualidade (Bass et al., 2013).

e) Desenvolver as Estratégias Arquitectónicas

Nesta fase, mapeia-se estratégias arquitectónicas para cenários e determina-se os níveis esperados de resposta do atributo de qualidade. Para cada estratégia arquitectural em consideração, determina-se os níveis esperados de resposta do atributo de qualidade que resultarão para cada cenário (Bass et al., 2013).

f) Determinar a Utilidade de Resposta das Estratégias Arquitectónicas

Calcule a utilidade dos níveis de resposta esperados para os atributos de qualidade através de interpolação. Utilizando os valores de utilidade previamente obtidos, estime a utilidade para o nível de resposta esperado de cada atributo de qualidade relacionado à estratégia arquitectural em análise (Bass et al., 2013).

g) Calcular o Custo e Benefício das Estratégias Arquitectónicas

Calcula-se o benefício global obtido a partir de uma estratégia arquitectural. Isso envolve subtrair o valor de utilidade do nível "actual" do nível “desejado” e normalizá-lo com base nos votos obtidos na etapa 3. Some o benefício de cada estratégia arquitectural em todos os cenários e para todos os atributos de qualidade relevantes (Bass et al., 2013).

h) Calcular o Retorno de Investimento das Estratégias Arquitectónicas

Em seguida, escolhe-se as estratégias arquitectónicas com base no Valor Benefício-Custo (VFC), considerando as restrições de custo e cronograma. Determine as implicações de custo e cronograma de cada estratégia arquitectural e calcule o valor de VFC para cada uma delas como uma relação entre benefício e custo. Classifique as estratégias arquitectónicas de acordo com o valor de VFC e seleccione as melhores até que o orçamento ou o cronograma não permitam mais escolhas (Bass et al., 2013).

i) Escolher a das Estratégias Arquitectónicas

Por fim, valide os resultados com base na intuição. Avalie se as estratégias arquitectónicas seleccionadas parecem estar alinhadas com os objectivos de negócios da organização. Se não estiverem, considere quaisquer questões que possam ter sido negligenciadas durante esta análise. Se surgirem problemas significativos, considere realizar outra iteração destas etapas (Bass et al., 2013).

Segundo Ionita et al., (2002), até o momento, nenhum método incorporou a perspectiva económica na avaliação de atributos de qualidade de software e na análise de compensações, excepto o CBAM. Além disso, como o CBAM é construído sobre métodos gerais de avaliação de arquitectura como SAAM e ATAM, ele herda os benefícios desses métodos em relação à eficiência.

Na Tabela 3, estão resumidas as principais características de cada método. Para cada uma das técnicas apresentadas, serão destacados pontos comuns, fraquezas ou pontos fortes, bem como outros aspectos relevantes.

Tabela 3. Resumo dos métodos de avaliação de arquitecturas

Método	Qualidade Avaliada	Métricas e Suporte de Ferramentas	Descrição do Processo	Pontos Fortes	Pontos Fracos	Tipos de Sistemas Aplicáveis
ATAM	Modificabilidade	Pontos de Sensibilidade; Pontos de Compensação suportados pela Ferramenta ATA	Boa	Geração de cenários com base em requisitos; Aplicável a propriedades estáticas e dinâmicas; Árvore de utilidade de qualidade.	Requer conhecimento técnico detalhado	Todos
SAAM	Modificabilidade	Classificação de cenários (directos vs. indirectos)	Razoável	Identificar potenciais áreas de alta complexidade; aberto para qualquer descrição arquitectural.	Não é uma métrica de qualidade clara. Não é suportado por técnicas para executar as etapas.	Todos
CBAM	Custos, Benefícios e Implicações de Cronograma	Tempo e Custos	Razoável	Fornecer medidas de negócios para mudanças específicas no sistema; Explicitar a incerteza associada às estimativas.	Identificar e negociar custos e benefícios pode ser feito pelos participantes de forma aberta.	Todos

Para este trabalho, foram seleccionados os métodos de avaliação ATAM e CBAM, A escolha é justificada com base nas seguintes razões:

1. **Abordagem Compreensiva**: O ATAM é um método amplamente reconhecido para avaliar a qualidade técnica e características de uma arquitectura de software. Este se concentra em identificar Tradeoffs técnicos, características de qualidade e preocupações de arquitectura. O CBAM, por outro lado, oferece uma perspectiva económica, considerando custos e benefícios associados à arquitectura. Combinar esses dois métodos proporciona uma análise abrangente que aborda tanto os aspectos técnicos quanto os económicos das arquitecturas.
2. **Balanceamento entre Qualidade Técnica e Viabilidade Económica**: As arquitecturas de *software* não são apenas sobre desenho técnico, mas também sobre a viabilidade económica. Ambos os métodos, ATAM e CBAM, ajudam a equilibrar esses dois aspectos cruciais. O ATAM ajuda a garantir que as arquitecturas atendam aos padrões de qualidade técnica, enquanto o CBAM ajuda a avaliar se a implementação e manutenção dessas arquitecturas são economicamente viáveis.
3. **Identificação de Tradeoffs**: O ATAM é especialmente eficaz na identificação de Tradeoffs de desenho, o que é fundamental na escolha entre diferentes abordagens arquitectónicas. Ele ajuda a identificar compromissos entre características de qualidade, como desempenho, testabilidade e modificabilidade. O CBAM pode complementar essa análise ao avaliar os custos associados a esses Tradeoffs e os benefícios potenciais decorrentes da escolha da arquitectura.
4. **Tomada de Decisões Embasadas em Evidências**: A combinação de ATAM e CBAM fornece uma base sólida para a tomada de decisões embasadas em evidências. Isso significa que as decisões relacionadas à escolha de uma arquitectura ou aperfeiçoamento de uma arquitectura existente podem ser fundamentadas tanto em critérios técnicos quanto económicos.

Portanto, a escolha de utilizar o ATAM e o CBAM para analisar as arquitecturas Android MVI e MVVM mostra-se adequada, pois permite uma análise holística e equilibrada, considerando tanto os aspectos técnicos quanto os económicos dessas arquitecturas.

2.6 Análise Económica de Arquitecturas

Até o momento, a maioria dos métodos de avaliação de arquitectura tem se concentrado nas relações entre decisões arquitectónicas e atributos de qualidade. No entanto, esses métodos costumam negligenciar as implicações económicas das decisões arquitectónicas. Além dos custos de construção, os custos a longo prazo de manutenção e actualização são frequentemente subestimados. É essencial considerar não apenas os custos, mas também os benefícios que as decisões arquitectónicas podem trazer. Portanto, é necessário desenvolver modelos económicos de software que abordem custos, benefícios, riscos e implicações de cronograma (Bass et al., 2013).

2.6.1 Contexto de Tomada de Decisão

Os objectivos comerciais desempenham um papel fundamental nos requisitos das arquitecturas. Uma vez que as principais decisões arquitectónicas têm implicações técnicas e económicas, os objectivos comerciais por trás de um sistema de software devem ser usados para orientar directamente essas decisões. A implicação económica mais imediata de uma decisão de objectivo comercial em uma arquitectura é como ela afecta o custo de implementação do sistema. Os atributos de qualidade alcançados pelas decisões arquitectónicas têm implicações económicas adicionais devido aos benefícios (que chamamos de utilidade) que podem ser derivados dessas decisões; por exemplo, tornar o sistema mais rápido, seguro ou mais fácil de manter e actualizar. É essa interacção entre os custos e os benefícios das decisões arquitectónicas que orienta (e limita) o arquitecto (Bass et al., 2013). A Figura 6 ilustra essa interacção.

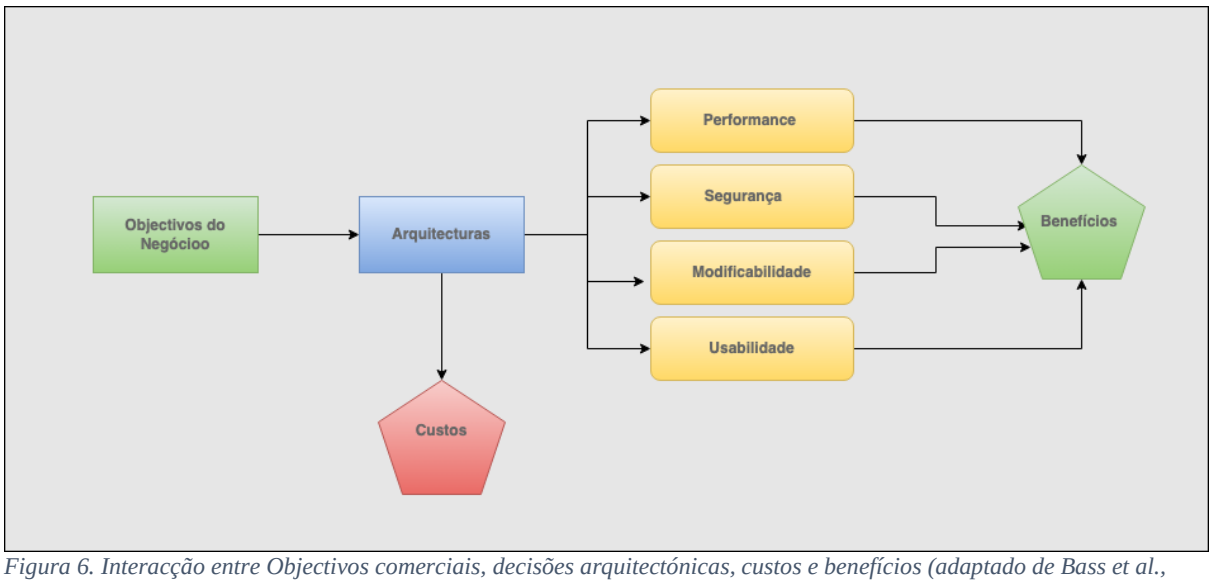


Figura 6. Interacção entre Objectivos comerciais, decisões arquitectónicas, custos e benefícios (adaptado de Bass et al., 2013).

Bass et al. (2013) destacam que compreender os custos e benefícios de decisões específicas permite uma selecção fundamentada entre alternativas concorrentes. A análise económica não toma decisões, mas fornece um quadro para avaliar o valor em relação ao custo (VFC) de investimentos arquitectónicos. Ela ajuda os stakeholders a tomar decisões racionais que atendam às suas necessidades e aversão ao risco. A análise económica se aplica às decisões arquitectónicas que estabelecem uma estratégia arquitectónica ampla, ajudando a avaliar sua viabilidade e a seleccionar a melhor estratégia entre opções concorrentes.

2.6.2 A Base para as Análises Económicas

Nesta secção vamos descrever as ideias-chave que formam a base das análises económicas. Começamos considerando uma colecção de cenários gerados como parte da obtenção de requisitos, uma avaliação arquitectónica ou especificamente para a análise económica. Examinamos como esses cenários diferem nos valores de suas respostas projetadas e, em seguida, atribuímos utilidade a esses valores. A utilidade é baseada na importância de cada cenário sendo considerado em relação ao valor de sua resposta prevista. Munidos de nossos cenários, em seguida, consideramos as estratégias arquitectónicas que levam às diversas respostas projectadas. Cada estratégia tem um custo e afecta múltiplos atributos de qualidade. Ou seja, uma estratégia arquitectónica poderia ser implementada para alcançar alguma resposta projectada, mas enquanto alcança essa resposta, ela também afecta outros atributos de qualidade. A utilidade desses "efeitos colaterais" deve ser considerada ao avaliar a utilidade geral de uma estratégia. É essa utilidade geral que combinamos com o custo do projecto de uma estratégia arquitectónica para calcular uma medida final de VFC (Valor em Relação ao Custo) (Bass et al., 2013).

Curvas de Utilidade-Resposta

A análise económica utiliza cenários de atributos de qualidade como uma maneira concreta de expressar e representar atributos de qualidade específicos. Variamos os valores das respostas e questionamos qual é a utilidade de cada resposta. Isso leva ao conceito de curva de utilidade-resposta (Bass et al., 2013).

O par estímulo-resposta de cada cenário oferece alguma utilidade (valor) aos stakeholders, e a utilidade de diferentes valores possíveis para a resposta pode ser comparada. Esse conceito de utilidade tem raízes que remontam ao século XVIII e é uma técnica para tornar conceitos muito diferentes comparáveis. Para nos auxiliar na tomada de decisões arquitectónicas importantes,

podemos desejar comparar o valor de alto desempenho com o valor de alta modificabilidade, com o valor de alta usabilidade e assim por diante (Bass et al., 2013).

O conceito de utilidade nos permite fazer isso. Embora às vezes seja necessário um pouco de incentivo para que eles o façam, os stakeholders podem expressar suas necessidades usando medidas concretas de resposta, como "99,999% disponível". No entanto, isso deixa em aberto a questão de quanto eles valorizam atributos de qualidade um pouco menos exigentes, como "99,99% disponível". Isso seria quase tão bom? Se sim, então o custo menor de alcançar esse valor menor pode torná-lo a opção preferida, especialmente se alcançar o valor mais alto resultasse em impactos negativos em outro atributo de qualidade, como o desempenho (Bass et al., 2013).

Capturar a utilidade de respostas alternativas de um cenário permite que o arquitecto faça compensações envolvendo esse atributo de qualidade. Podemos representar cada relação entre um conjunto de medidas de utilidade e um conjunto correspondente de medidas de resposta como um gráfico de curva de utilidade-resposta (Bass et al., 2013).

Alguns exemplos de curvas de utilidade-resposta são mostrados na Figura 7. Em cada um, os pontos rotulados como a, b ou c representam diferentes valores de resposta. A curva de utilidade-resposta, assim, mostra a utilidade como uma função do valor da resposta.

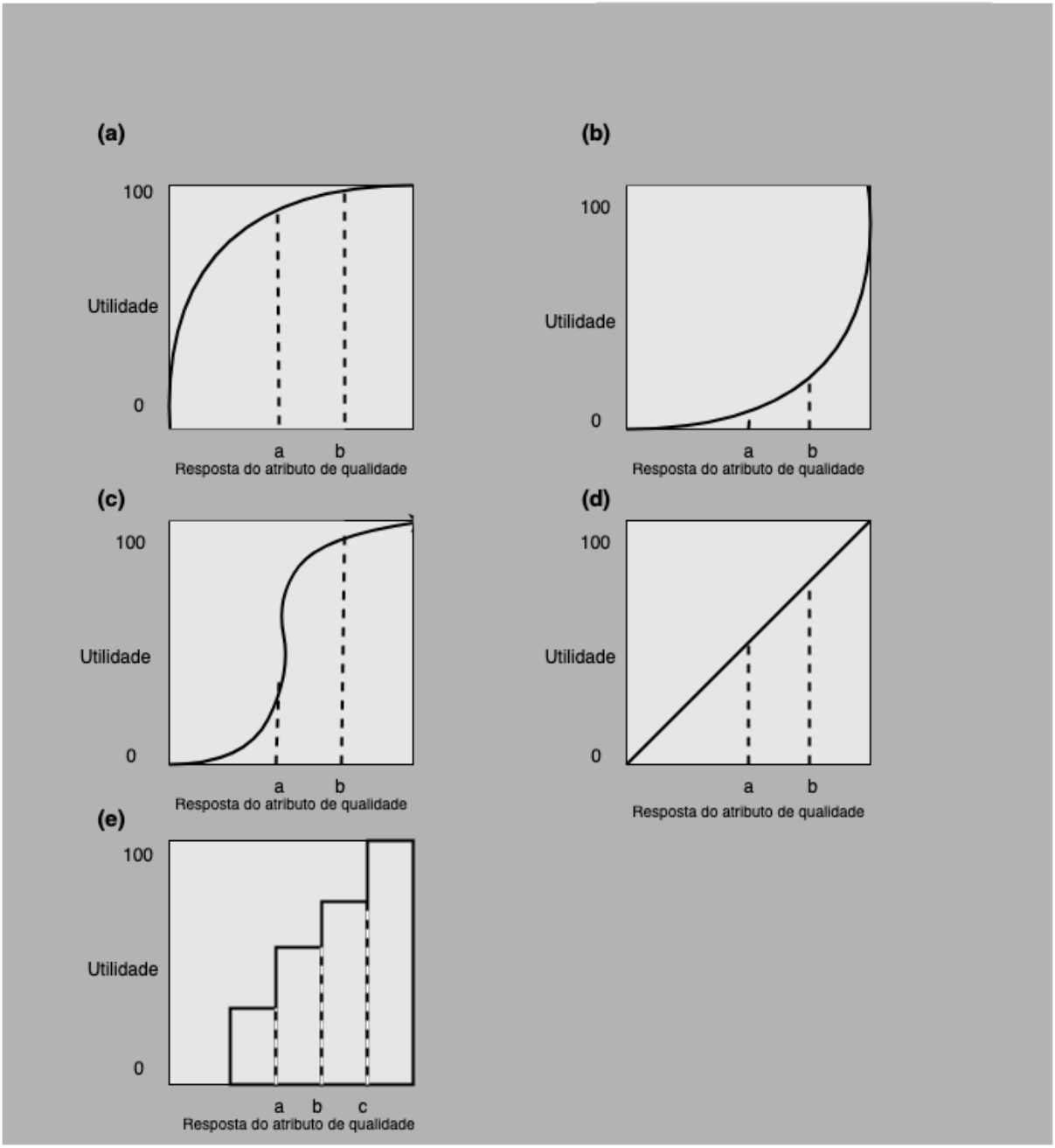


Figura 7. Alguns exemplos de curvas de utilidade-resposta (adaptado de Bass et al., 2013).

A curva de utilidade-resposta representa como a utilidade derivada de uma resposta específica varia à medida que a resposta varia. Conforme visto na Figura 9, a utilidade pode variar de forma não linear, linear ou até mesmo como uma função de etapas. Por exemplo, o gráfico (c) retrata um aumento acentuado na utilidade ao longo de uma pequena mudança no nível de resposta de um atributo de qualidade. No gráfico (a), uma mudança moderada no nível de resposta resulta apenas em uma mudança muito pequena na utilidade para o usuário (Bass et al., 2013).

Ponderando os Cenários

Diferentes cenários terão diferentes importâncias para os stakeholders; a fim de fazer uma escolha das estratégias arquitectónicas que melhor atenda aos desejos dos stakeholders, devemos ponderar os cenários. Não adianta gastar muito esforço optimizando um cenário específico no qual os stakeholders têm pouco interesse real.

Um método de ponderação dos cenários é priorizá-los e usar o ranking de prioridade como peso. Assim, para N cenários, o de mais alta prioridade recebe um peso de 1, o próximo de mais alta prioridade recebe um peso de (N-1) IN e assim por diante. Isso transforma o problema da ponderação dos cenários em um problema de atribuição de prioridades. Os stakeholders podem determinar as prioridades por meio de várias formas de votação. Um método simples é fazer com que cada stakeholder atribua prioridades aos cenários (de 1 a N), e a prioridade total do cenário é a soma das prioridades que ele recebe de todos os stakeholders. Essa votação pode ser pública ou secreta.

Efeitos Colaterais

Para Bass et al., (2013), cada estratégia arquitectónica afecta não apenas os atributos de qualidade que foram seleccionados para serem alcançados, mas também outros atributos de qualidade. Como se pode imaginar, esses efeitos colaterais em outros atributos de qualidade frequentemente são negativos. Se esses efeitos forem muito negativos, devemos garantir que haja um cenário para o atributo de efeito colateral e determinar sua curva de utilidade-resposta, para que possamos adicionar sua utilidade à equação de tomada de decisão. Calculamos o benefício de aplicar uma estratégia arquitectónica somando seus benefícios para todos os atributos de qualidade relevantes; para alguns atributos de qualidade, o benefício de uma estratégia pode ser negativo.

Determinação de Custos

Segundo Bass et al., (2013), uma das limitações no campo da arquitectura de software é a escassez de modelos de custo para várias estratégias arquitectónicas. Existem muitos modelos de custo de software, mas eles são baseados em características gerais do sistema, como tamanho ou pontos de função. Esses modelos são inadequados para responder à pergunta de quanto custa, por exemplo, usar um padrão de publicação-assinatura em uma parte específica da arquitectura. Portanto, diante deste cenário os arquitectos frequentemente recorrem a técnicas de estimativa. Onde um número absoluto para o custo não é necessário para classificar as estratégias de arquitectura candidatas. Frequentemente pode-se dizer algo como "Suponha

que a estratégia A custe $x. Parece que a estratégia B custará $2x e a estratégia C custará $0.5x." Isso é extremamente útil. Uma segunda abordagem é usar estimativas muito grosseiras. Ou, se você não tem confiança nesse grau de certeza, pode dizer algo como "A estratégia A custará muito, a estratégia B não deve custar muito, e a estratégia C provavelmente está em algum lugar no meio."

Determinação de Benefício e Normalização

O benefício global de uma estratégia arquitectónica em diferentes cenários de atributos de qualidade é a soma da utilidade associada a cada um deles, ponderada pela importância do cenário. Para cada estratégia arquitectónica i, o seu benefício Bi ao longo de j cenários (cada um com peso Wj) é

Equação 1. Fórmula de cálculo de benefício global para cada estratégia arquitectónica

$$B_i = \Sigma_j \, (b_{i,j} \times W_j)$$

Ao observarmos a equação 2, cada valor bi, j é calculado como a variação na utilidade (em comparação com a estratégia arquitectónica actualmente em vigor, ou aquela que está em competição com a que está sendo considerada) gerada pela estratégia arquitectónica em relação a esse cenário específico:

Equação 2. Fórmula de cálculo de benefício ao longo dos cenários para cada estratégia arquitectónica

$$b_{i,j} = U_{expected} - U_{current}$$

Isto é, a utilidade do valor esperado da estratégia arquitectónica menos a utilidade do sistema actual em relação a este cenário.

Calculando Valor pelo Custo (VFC)

O VFC para cada estratégia arquitectónica é a razão do benefício total, Bi, para o custo, Ci, de sua implementação:

Equação 3. Fórmula para o cálculo do valor pelo custo para cada estratégia arquitectónica

$$VFC = B_i \, / \, C_i$$

O custo Ci é estimado usando um modelo adequado para o sistema e o ambiente em desenvolvimento, como um modelo de custo que estima o custo de implementação medindo a complexidade de interacção de uma arquitectura. Você pode usar essa pontuação VFC para classificar as estratégias arquitectónicas em consideração.

Considere as curvas (a) e (b) na Figura 7, A curva (a) se aplaina à medida que a resposta do atributo de qualidade melhora. Nesse caso, é provável que um ponto seja alcançado em que o VFC diminui à medida que a resposta do atributo de qualidade melhora; gastar mais dinheiro não resultará em um aumento significativo na utilidade. Por outro lado, a curva (b) mostra que uma pequena melhoria na resposta do atributo de qualidade pode gerar um aumento muito significativo na utilidade. Nessa situação, uma estratégia arquitectónica cujo VFC é baixo pode ter uma classificação significativamente mais alta com uma melhoria modesta em sua resposta de atributo de qualidade.
Com os conceitos abordados nesta secção, podemos determinar que a sua aplicação prática é reflectida no método de avaliação CBAM.

2.7 Plataforma Android

Segundo O termo Speckmann, (2008)"Android" tem suas raízes na palavra grega "andr-", que denota "homem ou masculino", combinada com o sufixo "-eides", que sugere "similar ou da mesma espécie".

O Android é uma estrutura de software projectada para dispositivos móveis, consistindo em um conjunto de programas de sistema e aplicativos que juntos formam um sistema completo. Essa plataforma de software serve como base para o funcionamento de aplicativos, de maneira semelhante a uma plataforma real de trabalho.

O sistema Android é baseado em Linux, usando padrões de desenho de arquitectura de pilha de software. Conforme ilustrado na Figura 10, a arquitectura do Android é composta por quatro camadas: *kernel* do Linux, Bibliotecas e Android *Runtime*, *Framework* de Aplicativos e Aplicativos, conforme ilustrado na figura 8. Cada camada possui encapsulamento inferior, enquanto fornece uma interface de chamada para a camada superior (Ma et al., 2014).

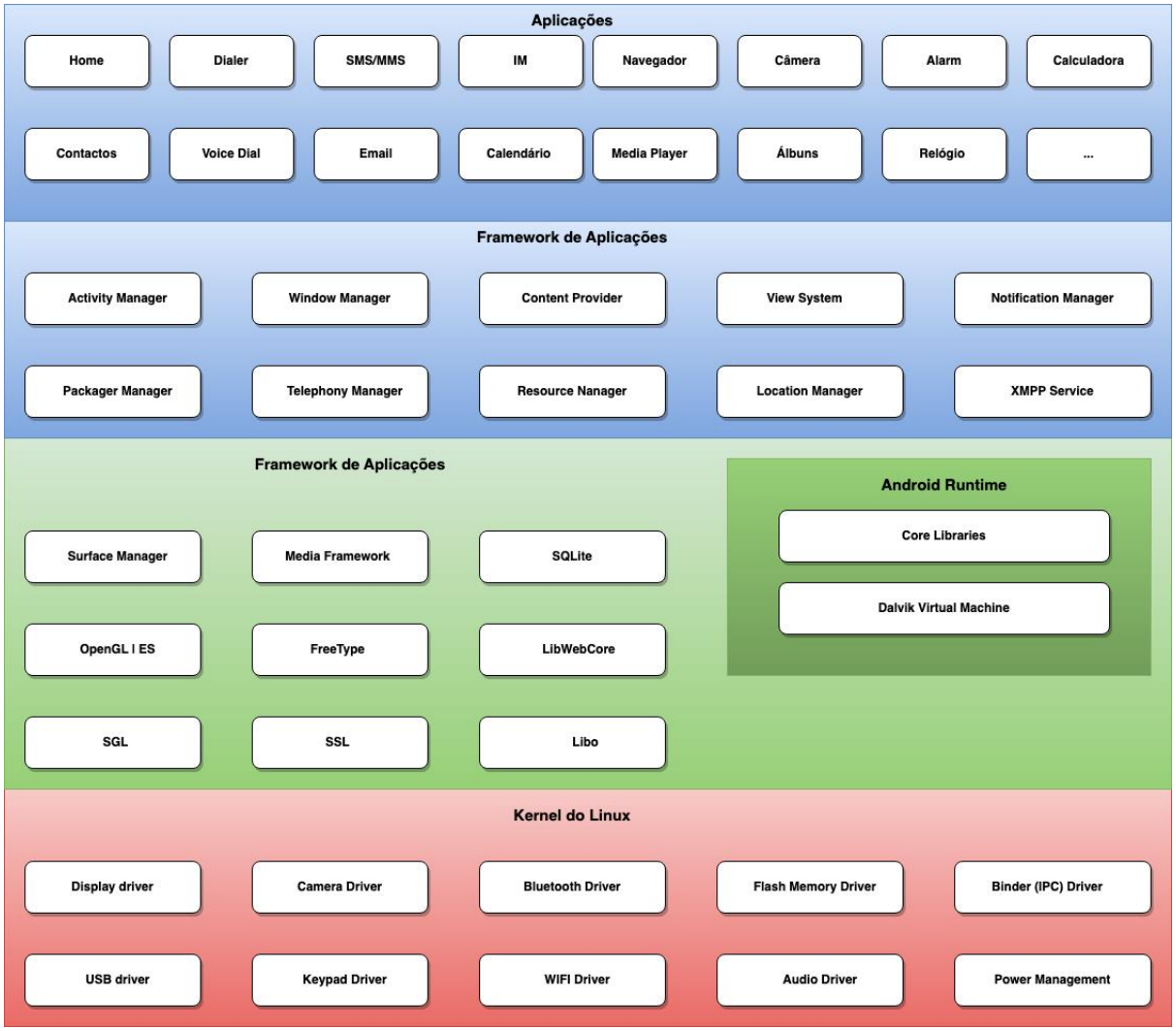


Figura 8. Arquitectura do Sistema Android (Adaptado do Ma et al., 2014).

a) Camada de Aplicativos

É a camada mais alta da arquitectura do Android, e essa camada é usada para os aplicativos instalados (ou seja, telefone, e-mail, etc.). A maioria dos aplicativos são aplicativos nativos, como *Google Maps*, câmara, navegador, SMS, contactos e calendários. O utilizador final utiliza o *framework* de aplicativos para operar esses aplicativos (Sharma & Kaur, 2014).

b) Estrutura de Aplicativos

Esta camada disponibiliza diversos conjuntos de pacotes de serviços de aplicativos. Nela estão inclusas as classes e serviços necessários para os aplicativos Android desenvolvidos. Os desenvolvedores têm a capacidade de reutilizar e expandir os componentes já presentes na Interface de Programação de Aplicativos. Os gerenciadores existentes nesta camada possibilitam que o aplicativo aceda dados (Sharma & Kaur, 2014). Os gerenciadores presentes são os seguintes:

- **Gerenciador de Actividades**: Controla e gere adequadamente todas as actividades e lida com o ciclo de vida das aplicações.
- **Gerenciador de Recursos**: Fornece acesso a recursos não relacionados ao código, como gráficos, etc.
- **Gerenciador de Notificações**: Permite que todas as aplicações mostrem alertas personalizados na barra de status.
- **Gerenciador de Localização**: Quando o usuário entra ou sai de uma localização geográfica específica, ele acciona alertas.
- **Gerenciador de Pacotes**: Os dados sobre os pacotes instalados no dispositivo são obtidos por meio do uso do gerenciador de pacotes.
- **Gerenciador de Telefonia**: As configurações de conexão de rede e todas as informações sobre serviços no dispositivo são manipuladas pelo gerenciador de telefonia.
- **Gerenciador de Janelas**: Para criar visualizações e layouts, o gerenciador de janelas é usado.
- **Provedor de Conteúdo**: Permite que aplicativos acedam dados de outros aplicativos, ou seja, compartilhamento de dados. Por exemplo: se tivermos um aplicativo de anotações em nosso celular e quisermos procurar uma localização específica cujo endereço acabamos de anotar, podemos considerar usar o aplicativo de mapas directamente do nosso aplicativo de anotações, em vez de alternar entre aplicativos.

c) Android *Runtime* e Bibliotecas

Nesta camada, encontra-se um componente central chamado Máquina Virtual Dalvik (DVM), onde cada processo é executado em uma instância separada na VM, ou seja, na Máquina Virtual. Além disso, nesta camada, estão presentes várias bibliotecas como SSL, SQLite ou libc (Sharma & Kaur, 2014).

- **Android Runtime**: é responsável por executar todos os aplicativos Android. O Android possui sua própria Máquina Virtual, ou seja, a DVM (*Dalvik Virtual Machine*), que é usada para executar os aplicativos Android conforme ilustra a figura 9. A DVM executa arquivos no formato (.dex). Devido à propriedade da DVM, os usuários podem executar vários aplicativos ao mesmo tempo.

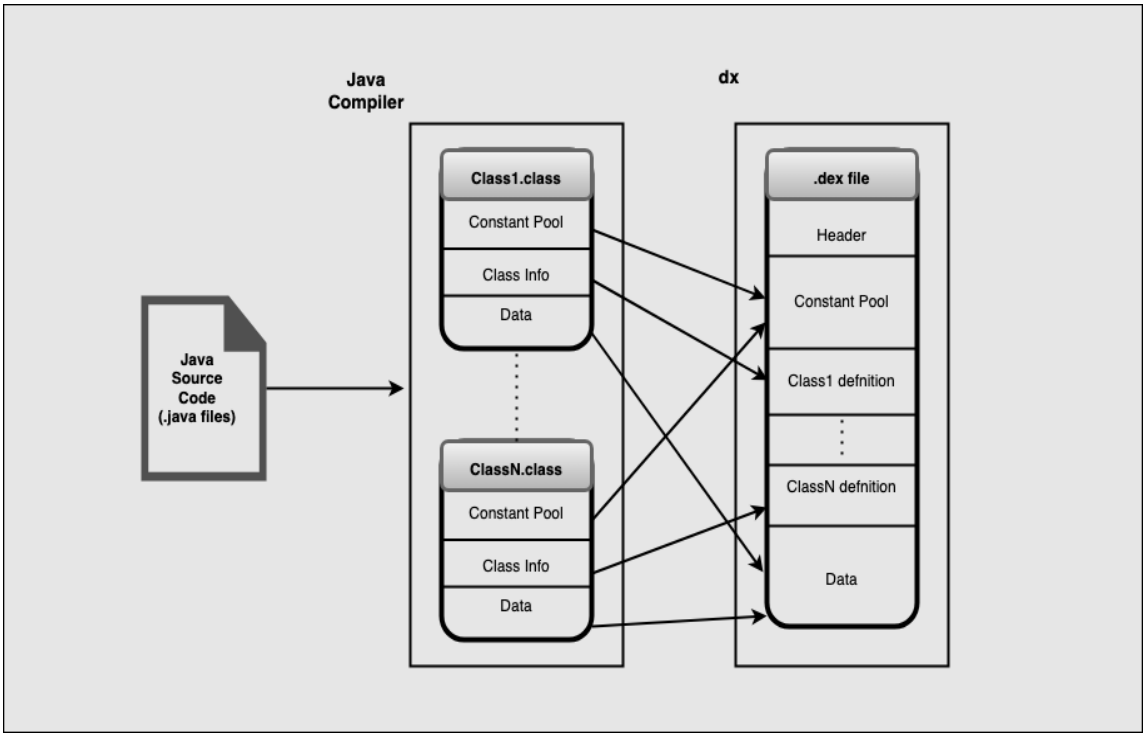


Figura 9. Processo da Máquina Virtual Dalvik (Sharma & Kaur, 2014).

- **Bibliotecas**: O Android possui suas próprias bibliotecas. Essas bibliotecas são desenvolvidas em C/C++ e não podem ser acedidas directamente. Através da ajuda da segunda camada, que é o *framework* de aplicativos, é possível aceder a essas bibliotecas. Há uma variedade de bibliotecas disponíveis, como bibliotecas web para aceder a navegadores da web e bibliotecas para formatos de áudio e vídeo, entre outras.

d) Kernel do Linux

Esta camada constitui o cerne da arquitectura do Android. Ela provê um sistema operacional estável e comprovado para a plataforma móvel, fornecendo serviços como gerenciamento de memória, de energia e segurança, entre outros. Além disso, ela facilita a conexão do software com o hardware, optimizando a comunicação entre esses dois componentes. Ao longo de muitos anos, diversos desenvolvedores têm modificado o *kernel* do Linux, mantendo-o em constante evolução e submetendo-o a múltiplas alterações em busca de melhorias por meio de testes, implementação, análise, correcção e aprimoramento. Essa camada também oferece várias características de segurança fundamentais para o sistema Android, entre elas:

- Um modelo de permissões definido pelo usuário;
- Isolamento de processos;
- Procedimentos disponíveis para IPC (Comunicação Inter processual) segura;
- Capacidade de eliminar compartilhamento desnecessário e inseguro do *kernel*.

Conforme apontado por Ma et al., (2014), no sistema operacional Android, cada aplicativo é isolado de todos os demais, operando em seu próprio espaço de memória, impedindo que qualquer pessoa aceda os dados de outros aplicativos. Para viabilizar a comunicação entre aplicativos, o Android utiliza o método de passagem de mensagens. Devido à natureza aberta de seu sistema operacional, o Android proporciona uma interface de usuário envolvente e interactiva. Na próxima secção, exploraremos mais detalhadamente essa estrutura.

A. Intent

O Android utiliza o conceito de "*Intent*" para transferir mensagens entre aplicações, a fim de proporcionar comunicação. Um *Intent* é uma mensagem que contém informações sobre um destinatário e dados. Além disso, o *Intent* também é empregado para estabelecer comunicação entre aplicativos e o sistema operacional. Esse tipo de *Intent* é conhecido como "*Intent* de Transmissão" e é enviado para múltiplas aplicações simultaneamente.

Os *Intents* são amplamente categorizados em duas categorias: *Intents* Explícitos e *Intents* Implícitos. O propósito principal de um *Intent* explícito é encontrar o destinatário pretendido pelo nome. Por outro lado, um *Intent* implícito deixa essa decisão para a plataforma Android determinar qual aplicativo deve receber o *Intent* (Ma et al., 2014).

B. Componente

As aplicações Android são escritas na linguagem Java e Kotlin. Elas são compiladas em códigos de bytes, que são convertidos em um arquivo executável Dalvik (.dex) que, por sua vez, é compilado em um arquivo de pacote Android (arquivo APK), o qual pode ser instalado nos dispositivos Android. Os *Intents* são distribuídos entre os componentes. No Android, existem quatro tipos de componentes. Esses componentes são declarados em um arquivo de manifesto se o desenvolvedor os utiliza (Ma et al., 2014). Esses componentes interagem para criar aplicativos funcionais e completos no ecossistema Android. Cada componente possui um papel específico no ciclo de vida e na funcionalidade geral de um aplicativo.

1. Actividades (*Activities*)

Actividade é um processo visível que é apresentado na tela, ou seja, funciona em primeiro plano na tela do dispositivo e interage com o usuário por meio de interfaces de usuário. A actividade é introduzida pelo *Intent* e cada actividade é uma unidade independente que pode ser iniciada e encerrada (Ma et al., 2014).

2. **Serviços (*Services*)**

São componentes que executam tarefas em segundo plano, ou seja, funcionam sem uma Interface de Usuário na tela. Eles são frequentemente usados para realizar operações longas, como baixar arquivos, reproduzir música ou realizar sincronizações de dados (Ma et al., 2014).

3. Broadcast Receivers

É um componente responsável por receber o *Broadcast Intent* do sistema operacional. Por exemplo, se um aplicativo deseja identificar um evento de inicialização do sistema operacional, ele precisa receber o *Broadcast Intent* com a ajuda do Broadcast Receiver (Ma et al., 2014).

4. Provedores de Conteúdo (*Content Providers*)

Gerenciam e disponibilizam dados para outras partes do aplicativo ou para outros aplicativos. Eles permitem compartilhar dados estruturados entre diferentes componentes e aplicativos. Por exemplo, esse componente gere a lista de contactos e o registro de chamadas telefónicas (Ma et al., 2014).

Outros componentes de grande relevância no ecossistema Android são os fragmentos (*Fragments*), que constituem partes da interface do usuário ou comportamento de um aplicativo, podendo ser incorporados em uma Actividade. Sua introdução ocorreu pela primeira vez no Android 3.0 (Honeycomb). De acordo com a definição, um fragmento é semelhante a uma actividade, com a diferença fundamental de que uma actividade preenche toda a tela, enquanto um fragmento ocupa apenas uma porção dela. A presença de fragmentos confere maior flexibilidade à *interface* do usuário do aplicativo (Lou, 2016).

De acordo com a mesma fonte Lou (2016), os Recursos também desempenham um papel crucial. Esses recursos são arquivos e conteúdo estático adicionais que o código do aplicativo utiliza. Eles abrangem diversos tipos, como *layouts*, imagens *bitmap*, dimensões, traduções, entre outros. No âmbito desta tese, o recurso mais significativo é o *layout*, que exerce uma influência substancial sobre a aparência estática da *interface* do aplicativo. O arquivo de recurso de *layout* é formatado em XML e organiza a disposição e o visual de diversos elementos (botões, textos, caixas de selecção, etc.). No entanto, para esta tese, será adoptado um novo paradigma de desenvolvimento de interfaces de usuário: o **Jetpack Compose**.

Jetpack Compose é um moderno conjunto de ferramentas recomendada pelo Android para criar interface do usuário, desenvolvido pela Google, que simplifica e acelera o processo de construção de interfaces de usuário para aplicativos Android. Ele permite que os

desenvolvedores criem componentes de interface do usuário utilizando uma sintaxe declarativa, na qual você descreve como a interface do usuário deve parecer e se comportar, em vez de escrever uma sequência de comandos imperativos (Android Developers, 2021).

Com o Jetpack Compose, elementos da interface do usuário são criados como funções, facilitando a criação e o gerenciamento de *layouts* complexos. Além disso, ele oferece recursos como actualizações automáticas da *interface* do usuário quando os dados mudam, melhorias nos testes da interface do usuário e uma integração mais eficiente com o ecossistema Android já existente (Android Developers, 2021).

O Jetpack Compose tem como objectivo aprimorar a experiência de desenvolvimento ao reduzir o código repetitivo, aumentar a produtividade dos desenvolvedores e melhorar a qualidade geral das interfaces dos aplicativos Android. Ele foi projectado para funcionar de maneira integrada com o código e as bibliotecas Android já existentes, permitindo que os desenvolvedores o adoptem gradualmente em seus projectos (Android Developers, 2021).

2.8 Arquitecturas de Desenvolvimento de Aplicações Nativas Android

As arquitecturas de software são essencialmente um método de organizar o código, evitando o acúmulo de tudo em um único arquivo ou directório. A estruturação adequada é crucial para o desenvolvimento de aplicativos, levando em consideração diversos factores, como a natureza da aplicação e a plataforma em que ela será executada (Android Developers, 2023a).

Existem várias arquitecturas de aplicativos Android, sendo as pioneiras o Modelo-Visão-Controlador (MVC) e o Modelo-Visão-Apresentador (MVP), que desempenharam papéis significativos na evolução das arquitecturas actuais. Essas arquitecturas iniciais dividem o aplicativo em três componentes fundamentais: o Modelo, responsável pela lógica de negócios e pelos dados; a Visão, encarregada de exibir informações; e o Controlador (ou apresentador), que gere as interacções e as actualizações da Visão (Android Developers, 2023a).

No entanto, para este trabalho, nosso foco estará nas arquitecturas mais recentes, MVVM (*Model-View-ViewModel*) e MVI (*Model-View-Intent*), que herdam muitos dos princípios do MVC e do MVP. Estas arquitecturas representam uma evolução natural e oferecem benefícios adicionais, como maior eficiência e manutenibilidade. Portanto, examinaremos em detalhes o funcionamento e as vantagens dessas arquitecturas modernas, considerando as raízes sólidas estabelecidas pelo MVC e pelo MVP.

2.8.1 Arquitectura Modelo-Visão-ModeloVisualização (*Model-View-ViewModel* MVVM)

Segundo (Maharjan, 2018), o MVVM foi originalmente introduzido pela Microsoft como um padrão natural para plataformas XAML. No contexto da plataforma Android, porém, ele representa uma abordagem mais recente. Essa arquitectura tem como propósito abstrair o estado e o comportamento de uma *View*, permitindo o desenvolvimento independente da interface do usuário e do modelo. Essa abstracção é alcançada ao introduzir o *ViewModel* como um mediador, cuja função é expor os dados do modelo e gerenciar a lógica do aplicativo durante a exibição da *View*.

De acordo com a Digital Ocean, (2022), como m,mostrado na Figura 10, a arquitectura MVVM possui três componentes: Modelo, View e ViewModel.

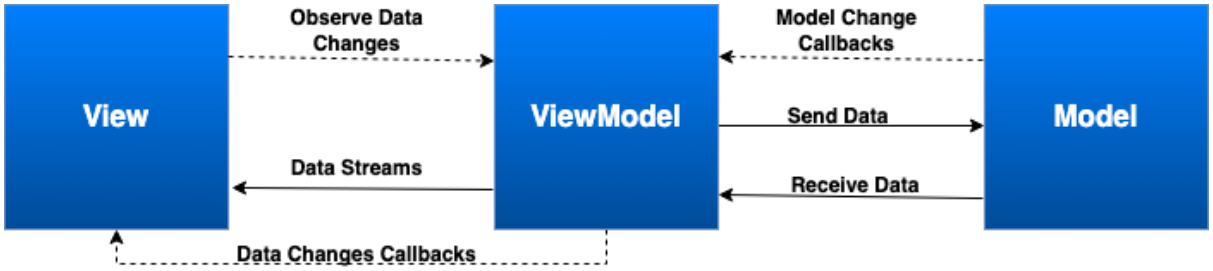


Figura 10. Interacção Model View ViewModel (Digital Ocean, 2022).

a) *O **View,*** representa a interface do usuário da aplicação, sem qualquer lógica da aplicação. Observa o *ViewModel*.

b) O **Modelo**, contém os dados da aplicação. Não pode se comunicar directamente com a Visualização. Geralmente, é recomendado expor os dados para o *ViewModel* por meio de *Observables*.

c) *O **ViewModel**,* age como um elo entre o Modelo e a *View*. É responsável por transformar os dados do Modelo. Fornece fluxos de dados para a visualização. Também utiliza ganchos ou retornos de chamada para actualizar a *View*. Ele solicitará os dados do Modelo.

As interacções entre esses componentes são explicadas na Figura 12. A conexão entre o *ViewModel* e a *View* é mais complexa do que na arquitectura MVP. Existem dois tipos de conexões: conexão tradicional e conexão de ligação de dados. As conexões tradicionais são semelhantes nas arquitecturas MVC e MVP, onde um componente *ViewModel* altera a *View* no código Java (Lou, 2016).

Implementação do MVVM no Android

A arquitectura MVVM é semelhante à arquitectura MVP. Ela é uma evolução com a biblioteca de vinculação de dados na arquitectura MVP. A diferença principal é a dependência do arquivo XML da View para a ViewModel/e classe Model.

Existem duas maneiras de implementar o MVVM no Android:

1. *Data Binding*
2. RXJava

Nesta tese, usaremos apenas o *Data Binding (Vinculação de Dados)*. A Biblioteca *Data Binding* foi introduzida pelo Google para vincular dados directamente no layout XML (Digital Ocean, 2022).

A ligação de dados (*Data Binding*) é um novo mecanismo introduzido no MVVM. Ele permite que a View seja directamente vinculada às propriedades e operações do ViewModel. Com a ligação de dados, o componente ViewModel não precisa notificar a View sobre as alterações por meio de código, e a View sabe que os dados foram carregados e exibe os dados por si só (Lou, 2016).

A sua implementação de acordo com a Digital Ocean (2022) segue os seguintes passos:

I. **Crie as classes de modelo**: Crie as classes que representam os dados e a lógica de negócios do seu aplicativo.

II. **Crie as classes de *ViewModel***: Crie classes *ViewModel* para cada tela ou componente do seu aplicativo. O *ViewModel* deve conter os dados que a *View* precisa exibir e também os métodos para lidar com as acções do usuário.

III. **Configure o *Databinding***: Use o *Databinding* para conectar a *View* ao *ViewModel*. Isso permite que os dados sejam automaticamente actualizados na View sempre que houver uma alteração no *ViewModel*.

IV. **Conecte a View ao ViewModel**: Na camada da *View*, conecte a *View* ao *ViewModel* usando o *Databinding*. Isso pode ser feito no arquivo de *layout* XML da *View*.

V. **Implemente a lógica de apresentação**: No *ViewModel*, implemente a lógica de apresentação necessária para formatar os dados do modelo de maneira adequada para a exibição. Isso pode incluir formatação de datas, cálculos ou qualquer outra transformação de dados.

VI. **Lide com acções do usuário**: No *ViewModel*, implemente métodos para lidar com as acções do usuário, como cliques em botões ou selecções em listas. Esses métodos podem actualizar os dados do modelo ou executar outras acções relacionadas.

VII. **Teste**: Como o *ViewModel* contém a lógica de apresentação, ele pode ser testado independentemente da interface do usuário. Crie testes de unidade para verificar se a lógica de apresentação está funcionando correctamente.

A figura 11, ilustra o diagrama UML que representa visualmente como os componentes da arquitectura MVVM interagem em um aplicativo Android, proporcionando uma clara separação de preocupações e um fluxo eficiente de dados entre as camadas.

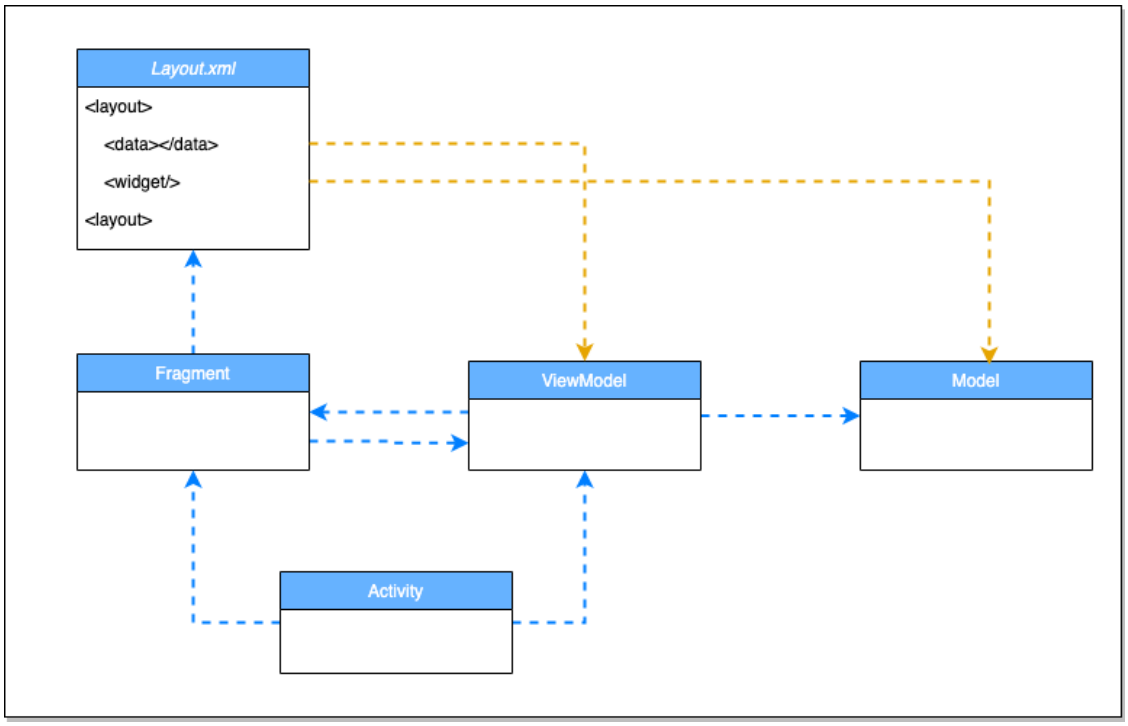


Figura 11. Diagrama de classes do MVVM (Adaptado de Lou, 2016).

As interacções de acordo com a Digital Ocean (2022) ocorrem da seguinte maneira:

- **Da View para o ViewModel**: A *View* observa objectos *LiveData* fornecidos pelo *ViewModel*. Quando o *LiveData* sofre alterações, o ViewModel atualiza o *LiveData*, e as mudanças são automaticamente reflectidas na *View*.

- **Do ViewModel para a View**: O *ViewModel* actualiza os objectos *LiveData*, e essas alterações são reflectidas automaticamente na View. O *ViewModel* não possui referências directas à *View*, garantindo uma separação de preocupações.

- **Do ViewModel para o Modelo**: O *ViewModel* interage com o Modelo por meio do Repositório, solicitando e recuperando dados conforme necessário. O *ViewModel* também aplica as transformações necessárias nos dados antes de expô-los à *View*.

- **Do Repositório para o Modelo**: O Repositório abstrai a fonte de dados (local ou remota) e busca dados das várias fontes conforme necessárias pelo *ViewModel*.

2.8.2 Arquitectura Modelo-Visão-Intenção (*Model-View-Intent* MVI)

Model-View-Intent (MVI) foi criado em 2014 por André Staltz, sendo amplamente aceito pela comunidade de desenvolvedores devido a sua premissa (Samuel & Barbosa, 2022). De acordo com Chauhan et al., (2021) MVI segue um fluxo unidireccional de dados, ela descreve uma aplicação como um conjunto de três componentes:

a) ***Model***: O estado da aplicação.
b) ***View***: A representação da aplicação voltada para o usuário.
c) ***Intent***: Uma acção contendo cálculos necessários para responder a interacções do usuário e actualizar o estado da aplicação.

Implementação do MVI no Android

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, acções 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 12 mostra o diagrama correspondente a arquitectura, onde a mesma segue um ciclo de vida natureza unidireccional e cíclico.

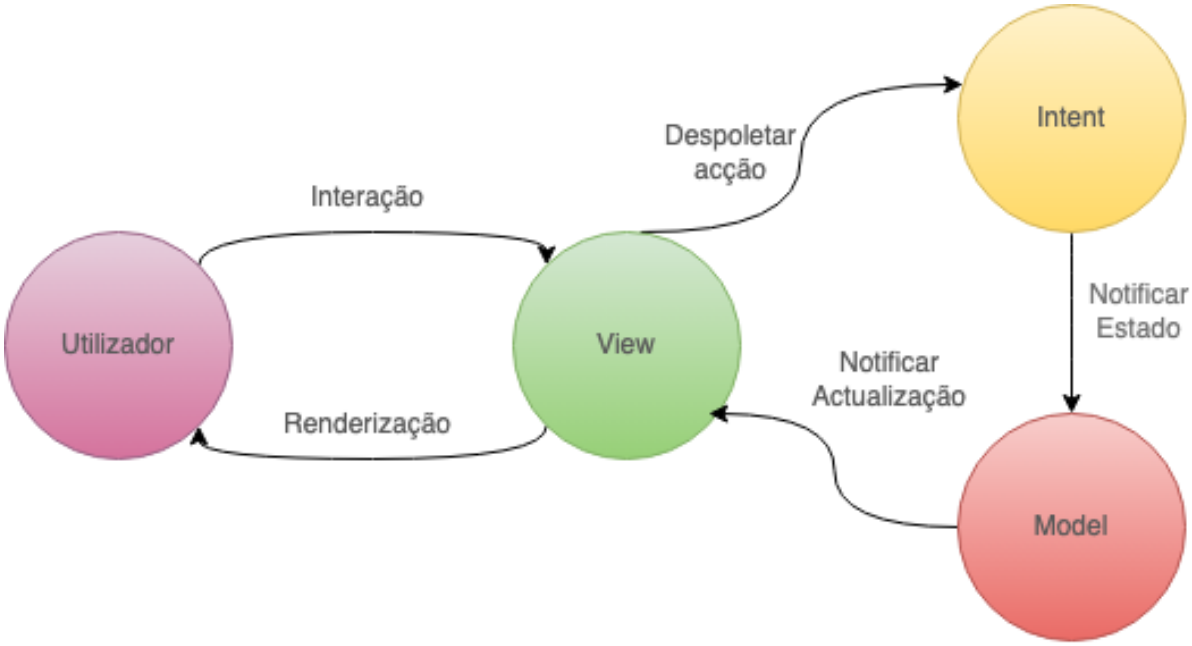


Figura 12. Fluxo Unidireccional de Dados no MVI (adaptado de Chauhan et al., 2021).

O ciclo ocorre da seguinte forma: Inicialmente, o usuário interage com a camada de visualização, o que gera acções correspondentes. Essas acções são então mapeadas pela aplicação para intenções, que têm o propósito de alterar o estado da aplicação. Posteriormente, o modelo notifica a camada de visualização com o estado recém-actualizado, levando a uma actualização da *interface* do usuário para reflectir esse novo estado. Em resposta a essa actualização, o usuário interage com a *interface* do usuário actualizada, desencadeando a produção de novas acções e reiniciando o ciclo.

2.9 Trabalhos Relacionados

No âmbito da análise comparativa das arquitecturas de desenvolvimento para Android, uma série de estudos tem sido realizada, sendo alguns deles notáveis devido à sua relevância actual e abrangência.

Em 2018, Humeniuk conduziu uma pesquisa intitulada "***Android Architecture Comparison*: MVP vs. VIPER**" (Comparação de Arquitecturas Android: MVP vs. VIPER). O objectivo central desta tese era responder à pergunta fundamental: "Qual arquitectura é mais adequada para o desenvolvimento de diferentes tipos de projectos?". Para abordar essa questão, o autor empregou um método de comparação similar ao utilizado na avaliação de algoritmos. Inicialmente, um conjunto de casos de uso foi implementado utilizando ambas as arquitecturas. Em seguida, um conjunto de métricas foi escolhido, abrangendo aspectos como modificabilidade, testabilidade, consumo de memória e tempo necessário para executar um

caso de uso. Posteriormente, todas essas métricas foram aplicadas às arquitecturas e os dados colectados sob as mesmas condições e configurações.

Ao final da análise, o autor concluiu que ambas as arquitecturas se mostraram viáveis. As métricas evidenciaram que cada arquitectura se destacou em um determinado aspecto, mas apresentou desempenho inferior em outros. Isso ressalta a ausência de uma solução absolutamente ideal para todas as situações, destacando a importância de seleccionar a arquitectura com base nas necessidades e restrições específicas do projecto. Tais limitações podem abranger custo ou tempo de desenvolvimento, desempenho da aplicação e qualidade do código.

Em 2016, Lou conduziu um estudo intitulado "***A Comparison of Android Native App Architecture MVC, MVP, and MVVM***" (Uma Comparação das Arquitecturas de Aplicativos Nativos Android MVC, MVP e MVVM). O objectivo central dessa pesquisa era realizar uma análise abrangente para determinar se as arquitecturas MVP (*Model-View-Presenter*) e MVVM (*Model-View-ViewModel*) são superiores à arquitectura MVC (*Model-View-Controller*) em termos de qualidade. Para responder a essa indagação, empregou-se o Método de Análise de Troca de Arquitectura (*Architecture Tradeoffs Analysis Method*). Posteriormente, três critérios foram estabelecidos: testabilidade, modificabilidade e desempenho.

Após a análise e a realização dos experimentos, o autor chegou a duas conclusões fundamentais. Em primeiro lugar, as arquitecturas MVP e MVVM superam a arquitectura MVC em termos de qualidades como testabilidade, modificabilidade e desempenho. A camada de apresentação (*Presenter*) desempenha um papel positivo nessas características. Em segundo lugar, é importante ressaltar que o MVVM nem sempre é superior ao MVP. O mecanismo de vinculação de dados (*Data Binding*) no MVVM contribui positivamente para a testabilidade, porém tem um impacto negativo na modificabilidade. Consequentemente, embora o MVVM apresente uma melhor testabilidade, sua capacidade de modificação é inferior em comparação ao MVP. E por isso o autor recomenda a migrar da arquitectura MVC para as arquitecturas MVP/MVVM para obter melhores qualidades de software. No entanto, quando se trata de escolher entre as arquitecturas MVP e MVVM, isso depende dos requisitos.

Em 2022, Samuel & Barbosa (2022), realizaram um estudo intitulado "**Análise Comparativa entre os Padrões MVC, MVP, MVVM e MVI na Plataforma Android**", cujo objectivo era comparar os padrões arquitecturais *Model-View-Controller* (MVC), *Model-View-Presenter*

(MVP), *Model-View-ViewModel* (MVVM) e *Model-View-Intent* (MVI) com foco em desempenho. Para essa análise, foi empregado o modelo *Software Architecture Analysis Method* (SAMM), juntamente com dados colectados, a fim de determinar qual arquitectura é mais adequada para o desenvolvimento de aplicações Android.

Os autores concluíram que os padrões MVVM, MVP e MVI superaram o MVC ao dividir as aplicações em componentes modulares e de propósito específico. Observaram também que o MVVM apresenta um desempenho superior em comparação com os outros padrões. Em aplicações de grande porte, o MVVM emergiu como o padrão de projecto de software mais ideal, enquanto o MVC, por oferecer uma implementação mais compreensível e simplificada, mostrou-se mais adequado para projectos de menor escala e abordagem simples.

3. METODOLOGIA

Nesta seção, detalharemos a abordagem metodológica que será empregada para a execução da Análise de Custo-Benefício (CBAM) e Método de Análise de Compensação Arquitecturais (ATAM), descritos na secção 2.4.3. Isso inclui o processo de desenvolvimento, a classificação da pesquisa, as técnicas de recolha de dados e a análise de dados.

3.1 Classificação da Pesquisa

a) **Quanto à Abordagem** - Este estudo adoptou uma abordagem de pesquisa quantitativa, centrada nas métricas técnicas e económicas obtidas durante a implementação do método CBAM.

b) **Quanto à Natureza** - Em relação à natureza da pesquisa, este estudo é uma pesquisa aplicada. Seu objectivo é gerar conhecimento com aplicabilidade prática, com o intuito de resolver problemas específicos (Marconi & Lakatos, 2003).

c) **Quanto aos Objectivos da Pesquisa** - foi realizada uma investigação exploratória. A abordagem exploratória permitiu uma compreensão mais profunda do problema e das soluções propostas. Conforme Marconi e Marconi e Lakatos (2003), esse tipo de pesquisa visa a familiarizar-se com o problema, tornando-o mais explícito ou construindo hipóteses.

d) **Quanto ao Método** - Em relação ao método, a pesquisa adoptou uma abordagem indutiva. Partindo da análise de um caso particular, buscou-se chegar a conclusões mais amplas. Conforme Marconi e Marconi e Lakatos (2003), o método de indução é um processo mental que, a partir de dados particulares devidamente comprovados, infere uma verdade geral ou universal, que não é contida nas partes examinadas.

e) **Quanto aos procedimentos** - A pesquisa é de cunho bibliográfico, baseou-se em fontes bibliográficas. Além disso, será foi experimental, permitindo a realização de testes. Conforme Marconi e Lakatos (2003), o propósito desses estudos é frequentemente demonstrar a viabilidade de uma determinada técnica ou programa como uma solução potencial e viável para programas práticos específicos.

3.2 Técnicas De Recolha De Dados

Para colectar dados de forma abrangente e precisa, as seguintes técnicas foram empregadas:

3.2.1 Pesquisa bibliográfica

Considerada mãe de toda pesquisa, fundamenta-se em fontes bibliográficas; ou seja, os dados são obtidos a partir de fontes escritas, portanto, de uma modalidade específica de documentos,

que são obras escritas, impressas em editoras, comercializadas em livrarias e classificadas em biblioteca (Silveira, 2009), A utilização desta técnica permitiu ao autor entender com alguma profundidade conceitos imprescindíveis para elaboração do presente estudo.

3.2.2 Pesquisa documental

foram analisados documentos contemporâneos e retrospectivos, considerados cientificamente autênticos (Silveira, 2009). A análise documental auxiliou na compreensão de conceitos técnicos e estáticos por meio de documentos técnicos de fontes confiáveis, como o Android Developers e a Digital Ocean.

3.2.3 Colecta Automatizada de Métricas Técnicas

Durante o processo de desenvolvimento dos aplicativos, incorporou-se ferramentas de análise estática de código e ferramentas de medição de desempenho para colectar métricas técnicas automaticamente. Isso incluiu métricas como linhas de código, tempo de resposta, uso de memória, entre outras. As ferramentas utilizadas para medir o desempenho de cada arquitectura são o *Android Studio Profiler* e o *Macrobenchmark*.

3.2.4 Inquérito por questionário

É um instrumento de colecta de dados constituído por uma série ordenada de perguntas que devem ser respondidas por escrito pelo informante, sem a presença do pesquisador. Objectiva levantar opiniões, crenças, sentimentos, interesses, expectativas, situações vivenciadas. A linguagem utilizada no questionário deve ser simples e directa, para que quem vá responder compreenda com clareza o que está sendo perguntado (Silveira, 2009).

Esta técnica foi utilizada para inquirir por meio de um questionário de perguntas semifechadas, aos membros da comunidade de desenvolvedores de sistemas em Moçambique, com o objectivo de entender o contexto local sobre o uso de arquitecturas de desenvolvimento de software como descrito a seguir.

Processo de Recolha de Dados e Selecção de Participantes:

Amostragem: O processo de recolha de dados deste estudo envolveu uma abordagem de amostragem que visava obter informações abrangentes sobre o uso de arquitecturas de desenvolvimento de software no contexto local de Moçambique. Para atingir esse objectivo, a amostra consistiu em membros da comunidade de desenvolvedores de sistemas em Moçambique. A escolha dessa amostra teve como base a relevância dos participantes para o

tópico da pesquisa, uma vez que estes possuem experiência prática no desenvolvimento de software e estão imersos no contexto local.

Selecção de Participantes: Os participantes foram seleccionados com base na acessibilidade e disponibilidade para participar do estudo. Dada a natureza da pesquisa, que procurava entender as práticas e critérios relacionados à selecção de arquitecturas de software em projectos em Moçambique, os membros da comunidade de desenvolvedores de sistemas eram uma escolha adequada. Como observado nos resultados, os participantes eram representativos da área, com diferentes graus de experiência na indústria de desenvolvimento de software, trabalhando em plataformas Web, Desktop e Android e actuando em sectores como telecomunicações, tecnologia da informação e saúde entre outros desempenhando diversas funções no ciclo de vida de desenvolvimento de sistemas.

Questionário: Os dados foram recolhidos por meio de um questionário de perguntas semifechadas. As perguntas permitiram a obtenção de informações quantitativas sobre a utilização de arquitecturas específicas, métodos de avaliação, motivos para não avaliar ou não usar uma arquitectura específica, entre outros, permitiu a obtenção de dados numéricos que podem ser facilmente analisados e quantificados. Isso proporcionou uma visão clara das preferências e tendências dos participantes em relação às arquitecturas de desenvolvimento de software.

O uso dessas técnicas de inquérito contribuiu para a obtenção de uma visão completa das práticas e critérios em relação às arquitecturas de software na comunidade de desenvolvedores de sistemas em Moçambique. Os participantes tiveram a oportunidade de compartilhar suas experiências e conhecimentos, o que enriqueceu os resultados da pesquisa.

Limitações da Amostragem: É importante reconhecer que a amostra, neste caso, era composta principalmente por membros da comunidade de desenvolvedores de sistemas em Moçambique e pode não representar completamente todos os contextos de desenvolvimento de software no país. As conclusões e achados do estudo são aplicáveis a essa amostra específica e podem não ser generalizáveis para contextos diferentes. Portanto, qualquer generalização para outros grupos de desenvolvedores ou contextos deve ser feita com cuidado, considerando as potenciais diferenças.

3.3 Processo de Desenvolvimento de Aplicativos

O processo de desenvolvimento adoptará uma abordagem ágil, enfatizando iterações contínuas e refinamento das soluções. Para atender a esse objectivo, serão implementadas as arquitecturas MVI e MVVM em dois aplicativos Android distintos. Esse enfoque permitirá uma avaliação mais abrangente das arquitecturas em termos técnicos e económicos.

3.3.1 Ferramentas de Desenvolvimento

i. Linguagens de programação

Para a implementação dos aplicativos, a linguagem de programação escolhida será o Kotlin. A escolha do Kotlin é respaldada por sua natureza concisa, segura e moderna, que oferece facilidades para desenvolvimento Android eficiente.

ii. Plataforma - Os aplicativos serão desenvolvidos na plataforma Android, aproveitando sua ampla base de usuários e ferramentas robustas de desenvolvimento.

iii. Framework - Será utilizado o *framework* Jetpack Compose versão 1.4.3 para a criação das *interfaces* de usuário. Essa escolha baseia-se na capacidade do Jetpack Compose de oferecer uma experiência de desenvolvimento declarativa e flexível para a criação de interfaces modernas e interactivas.

iv. Ambiente Integrado de Desenvolvimento (IDE) - A *IDE* seleccionada para o desenvolvimento será o Android Studio. O Android Studio oferece um ambiente completo e optimizado para o desenvolvimento de aplicativos Android, com recursos de depuração, análise e integração com as ferramentas de desenvolvimento.

v. Sistema de Controlo de Versões (SCV) e Repositório Remoto de Código - O sistema de controle de versões adoptado será o Git, com integração a plataforma de hospedagem de repositórios, o GitHub. Isso possibilitará o controle colaborativo do código-fonte e o acompanhamento das versões.

vi. Sistema de Gestão de Base de Dados (SGBD) - Para o armazenamento de dados, será utilizado um Sistema de Gestão de Base de Dados (SGBD) adequado à aplicação, neste caso o ROOM 2.5.2. O Room é uma biblioteca que faz parte da arquitectura de componentes do Android e é usada para criar e gerenciar um banco de dados SQLite de maneira mais eficiente e orientada a objectos.

Essas ferramentas de desenvolvimento proporcionarão um ambiente sólido e moderno para a implementação dos aplicativos, permitindo a exploração das métricas técnicas e económicas conforme planejado na metodologia.

3.4 Análise de dados e testes

No contexto em que ambas as arquitecturas têm a *ViewModel* como componente principal, a avaliação vai além da mera cobertura de testes, que é semelhante em ambas. O objectivo é determinar qual das arquitecturas facilita uma implementação mais natural e eficaz do comportamento correcto dos componentes.

Para alcançar esse objectivo, foram desenvolvidos 7 testes unitários e 7 testes instrumentados para os casos de uso apresentados na seção 5.2.1, que se concentram na ViewModel, a peça central de ambas as arquitecturas. Esses testes servem como um meio de avaliar a usabilidade e a facilidade de implementação das funcionalidades correctas em ambas as arquitecturas.

Esses testes são projectados para garantir que a ViewModel se comporte correctamente ao interagir com tarefas, incluindo a inserção, actualização e exclusão de tarefas. Eles também verificam se a ViewModel reflecte as mudanças esperadas nos dados, bem como valida os fluxos de tela, contribuindo para uma análise abrangente da eficácia das arquitecturas em proporcionar um comportamento preciso e eficiente dos componentes.

4. CASO DE ESTUDO

Neste capítulo, proporciona-se uma visão detalhada do caso de estudo, que é o desenvolvimento de uma aplicação intitulada "ToDo". Exploraremos seus objectivos, estrutura e contexto em que será desenvolvido, focando em fornecer uma compreensão completa do projecto em si.

4.1 Objetivos e Contexto

O projecto "ToDo" tem como objectivo principal desenvolver um aplicativo móvel para gerenciamento de tarefas em dispositivos Android. Este aplicativo tem como objectivo ser simples o suficiente para que se possa entender rapidamente, mas complexo o suficiente para destacar decisões de desenho difíceis e cenários de teste.

A aplicação permitirá que os usuários criem, visualizem, editem e excluam tarefas, uma tarefa possui um título e descrição, e pode estar concluída ou activa. Tarefas concluídas podem ser excluídas de uma só vez, proporcionando uma solução conveniente e eficiente para o gerenciamento de suas actividades diárias.

Este projecto visa demonstrar a aplicação prática das arquitecturas Modelo-Visão-Apresentador (MVP) e Modelo-Visão-ModeloVisualização (MVVM) no contexto de desenvolvimento de aplicativos Android. A criação de duas versões do aplicativo, cada uma seguindo uma arquitectura específica, permitirá a comparação das abordagens em termos de usabilidade, escalabilidade e eficiência do código.

4.2 Organização do documento de requisitos

Para que se tenha uma interpretação clara e correcta deste documento, é importante que se tenha conhecimento de algumas convenções e termos específicos usados durante a sua elaboração, os mesmos são descritos a seguir:

a) Identificação de Requisitos

Foi usada a denotação RF para representação dos requisitos funcionais do sistema e NF para os requisitos não funcionais. Exemplo: RF01 – primeiro requisito funcional; NF01 – primeiro requisito não funcional.

b) Prioridade de Requisitos

Para estabelecer a prioridade dos requisitos foram adoptadas as denominações “Alta”, “Média” e “Baixa”.

- **Alta** - dado a requisitos que são a base da regra do negócio, ou seja, sem esses requisitos implementados o sistema é incapaz de atender às expectativas básicas.
- **Média** - requisitos que dão suporte às principais funções optimizando as operações, e sem estas o sistema funciona, porém com um desempenho não satisfatório.
- **Baixa** - é o requisito que não compromete as funcionalidades básicas do sistema, isto é, o sistema pode funcionar de forma satisfatória sem ele. Geralmente colocado para próxima versão do sistema caso não haja tempo suficiente para a sua implementação

4.3 Estrutura e Funcionalidades do Aplicativo

O aplicativo oferecerá as seguintes funcionalidades essenciais aos usuários (tabela 4):

Tabela 4. Requisitos da aplicação ToDo

Código	Nome	Descrição	Beneficiário	Prioridade
RF001	Adição de novas tarefas	O sistema deve permitir criar tarefas, inserindo um título, uma descrição e sua prioridade.	Utilizador	Alta
RF002	Visualização de tarefa	O sistema deve permitir a visualização da lista completa das tarefas e também detalhes de uma tarefa específica	Utilizador	Alta
RF003	Edição de tarefas	O sistema deve permitir a edição de detalhes da tarefa	Utilizador	Alta
RF004	Filtragem das tarefas	O sistema deve permitir filtrar as tarefas por categoria	Utilizador	Média
RF005	Exclusão de tarefas	O sistema deve permitir a eliminação de tarefas	Utilizador	Alta
RF006	Pesquisa de tarefas	O sistema deve permitir fazer a pesquisa de tarefas	Utilizador	Média
RNF001	Mudança de tema	O sistema deve permitir mudar de tema (tema claro e escuro)	Utilizador	Baixa

4.4 Interface de Utilizador

A interface de utilizador do aplicativo será projectada com foco na simplicidade e usabilidade. Os elementos visuais serão cuidadosamente seleccionados para garantir uma experiência intuitiva e eficaz para os usuários conforme ilustra a figura 13 e 14.

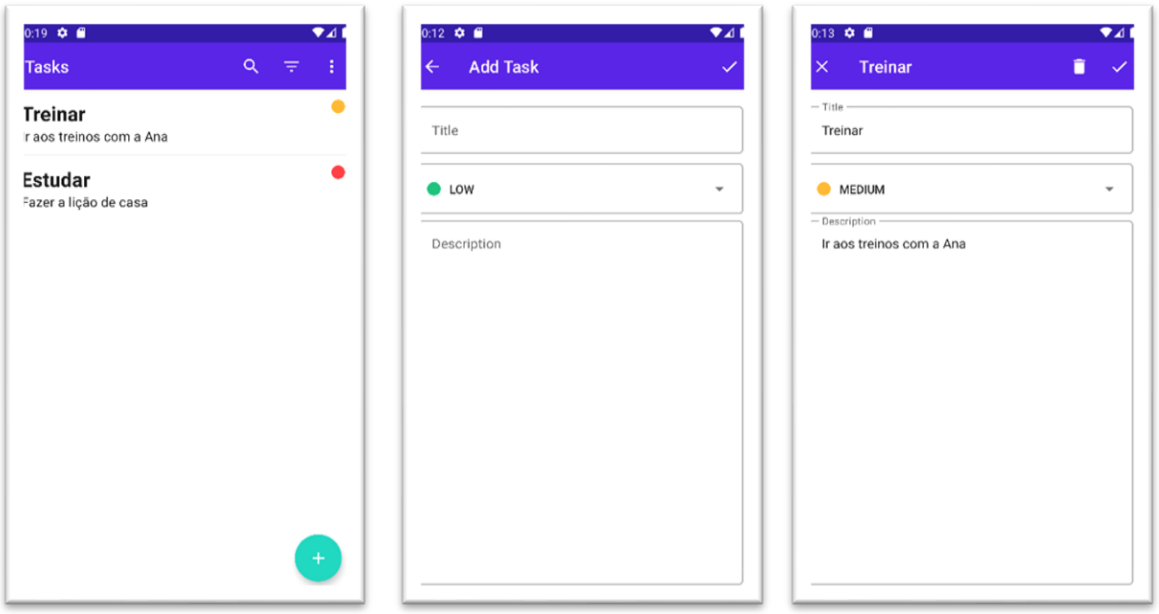


Figura 13. Interface de utilizador no modo claro.

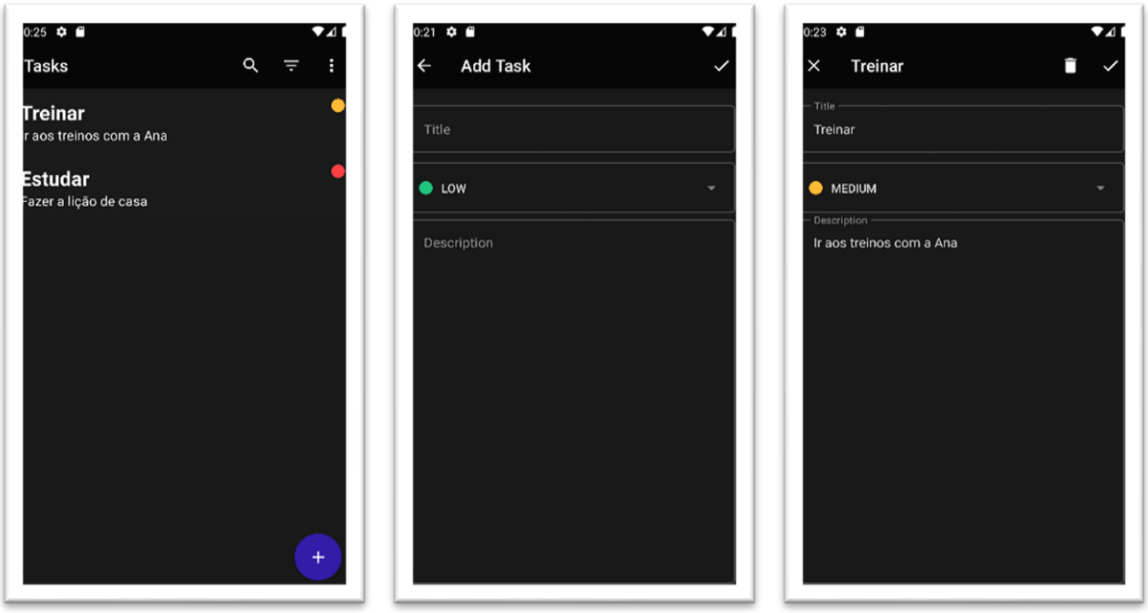


Figura 14. Interface de utilizador no modo escuro.

5. APRESENTAÇÃO DE RESULTADOS E DISCUSSÃO

Nesta seção serão apresentados os resultados obtidos através do inquérito e do experimento, que incluiu a aplicação do ATAM e CBAM às diferentes arquiteturas de desenvolvimento de aplicações nativas Android: Model-View-Intent (MVI) e Model-View-ViewModel (MVVM). irá se analisar os resultados deste estudo à luz de pesquisas anteriores conduzidas por outros investigadores. Além disso, exploraremos as implicações práticas e teóricas dos resultados, discutiremos a generalização dos achados para outros contextos ou populações, destacaremos as limitações e pontos fortes do trabalho e, por fim, apontaremos as direções para futuras investigações.

5.1 Resultados do Inquérito

O inquérito (Apêndice C) teve como objectivo entender as práticas e critérios relacionados à selecção de arquitecturas de software em projectos. O inquérito teve 36 participantes distribuídos conforme a figura 15.

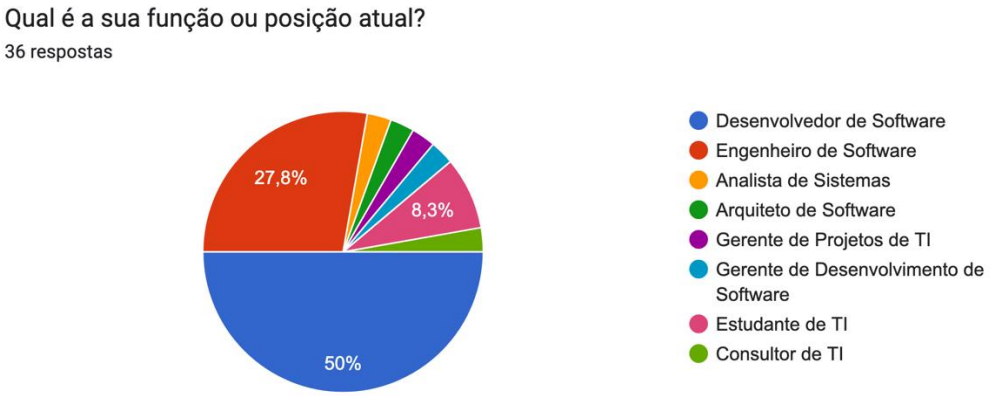


Figura 15. Distribuição dos participantes por função

1. Plataforma alvo

Web, Desktop e Android são as plataformas mais utilizadas pelos participantes conforme mostra a figura 16.

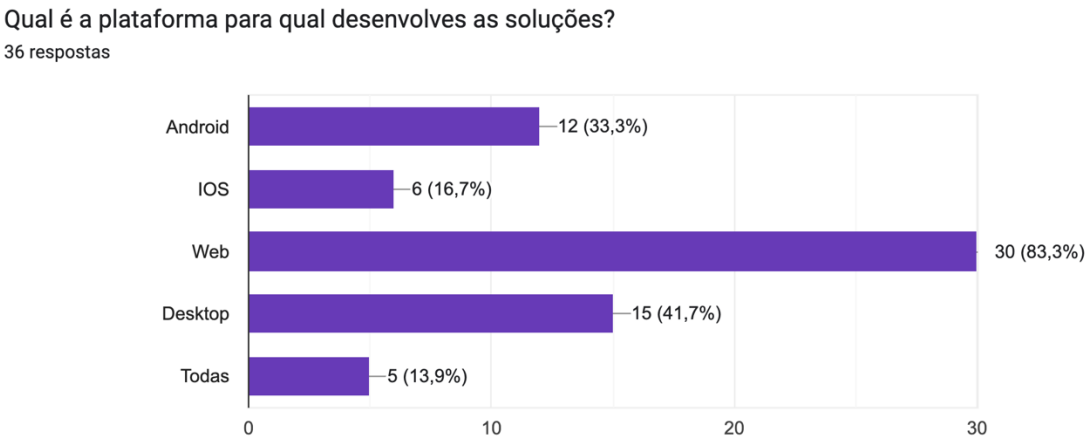


Figura 16. As plataformas mais utilizadas.

2. Tempo de Experiência na Área

A maioria dos profissionais tem mais de 5 anos de experiência na área de desenvolvimento de software conforme ilustra a figura 17.

Figura 17. Tempo de experiência dos participantes.

3. Indústria ou Área de Actuação

Telecomunicações, Tecnologia da Informação (TI) e Saúde são os sectores mais frequentes de actuação dos participantes conforme ilustra a figura 18.

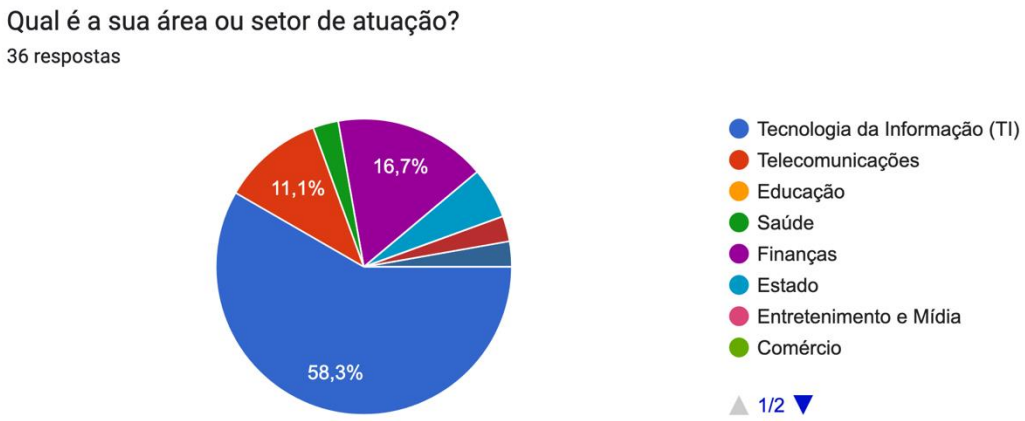


Figura 18. Distribuição dos participantes dor área de atuação.

4. Uso de Arquitecturas

A maioria dos profissionais utiliza arquitecturas específicas em seus projectos conforme mostra a figura 19, com MVC, Clean Architecture e MVVM sendo as mais comuns de acordo com a figura 20. Onde a experiência prévia e os requisitos técnicos do projecto mostram-se sendo as que mais influenciam na escolha da arquitectura como ilustra a figura 21.

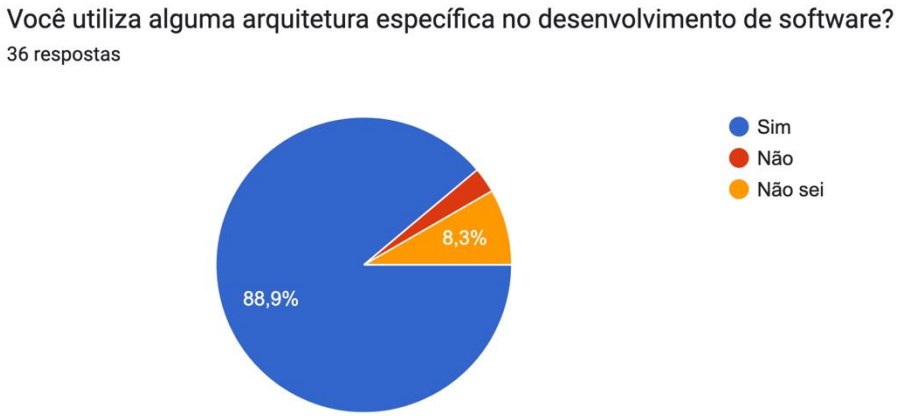


Figura 19. Utilização de arquitecturas de software no desenvolvimento de aplicações

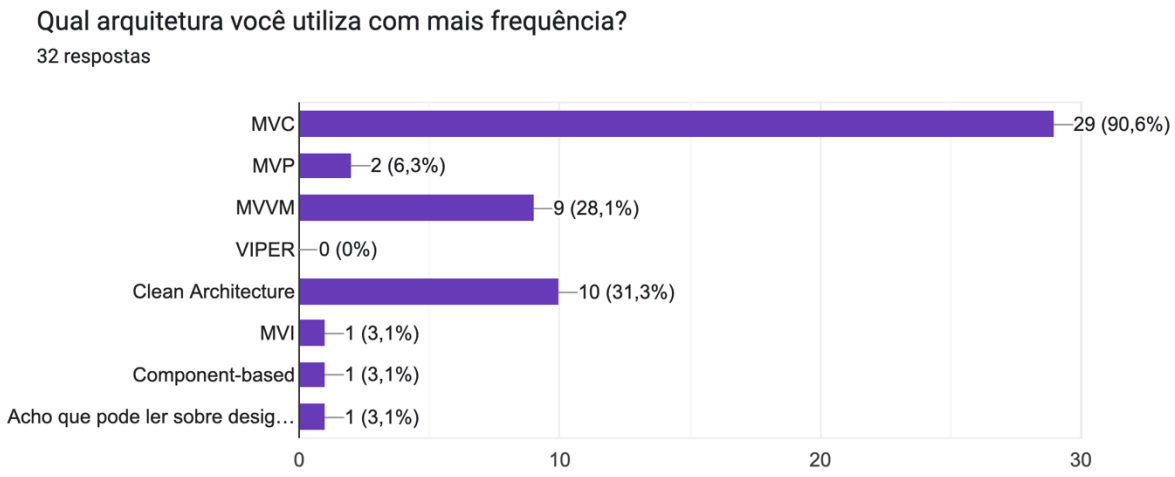


Figura 20. Arquitecturas mais utilizadas

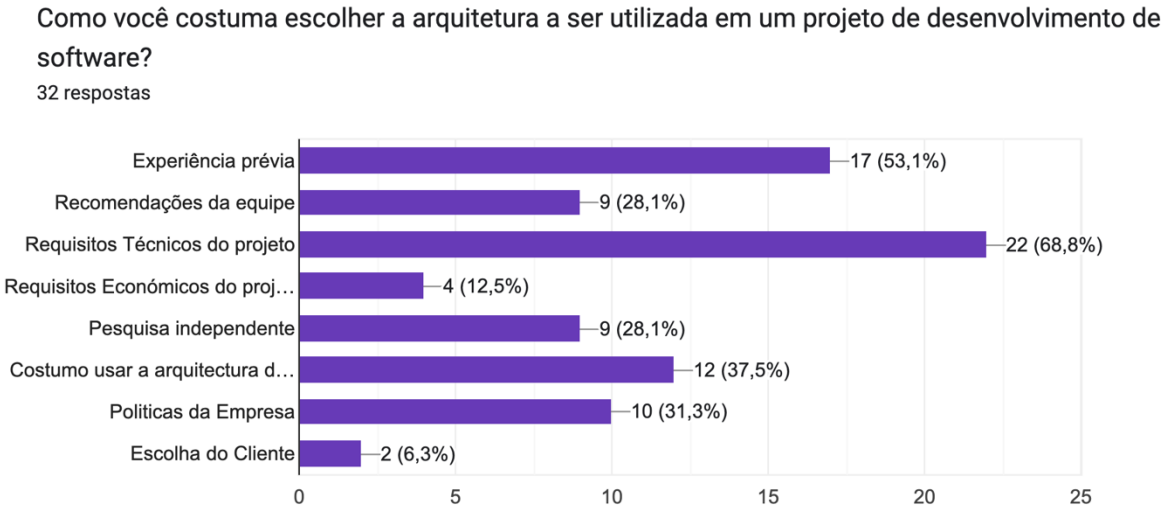


Figura 21. Critérios utilizados na escolha de arquitecturas

5. Avaliação de Arquitecturas

A avaliação formal de arquitecturas é realizada por alguns profissionais (figura 22), sendo SAAM e CBAM os métodos mais comuns (figura 23), onde os aspectos mais considerados na avaliação são a escalabilidade e desempenho da arquitectura (figura 24).

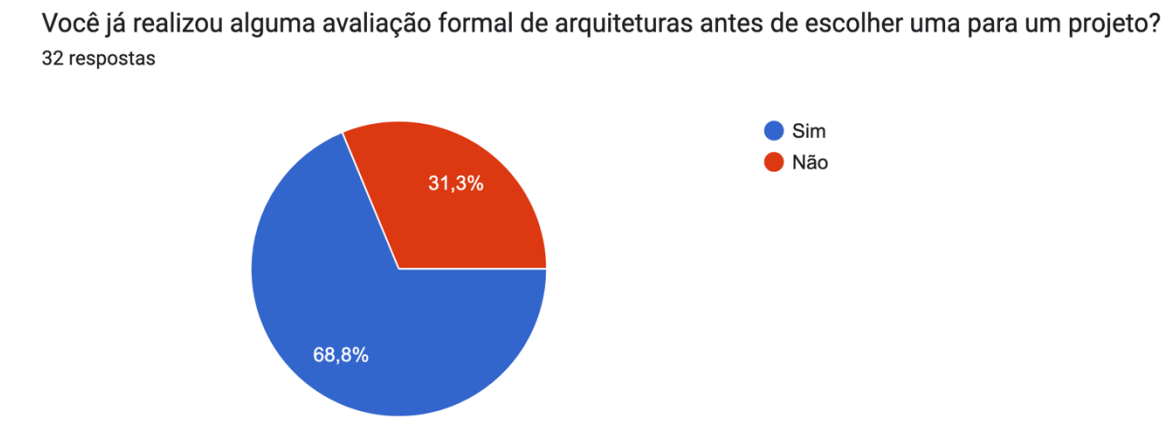


Figura 22. Realização de avaliação formal das arquitecturas

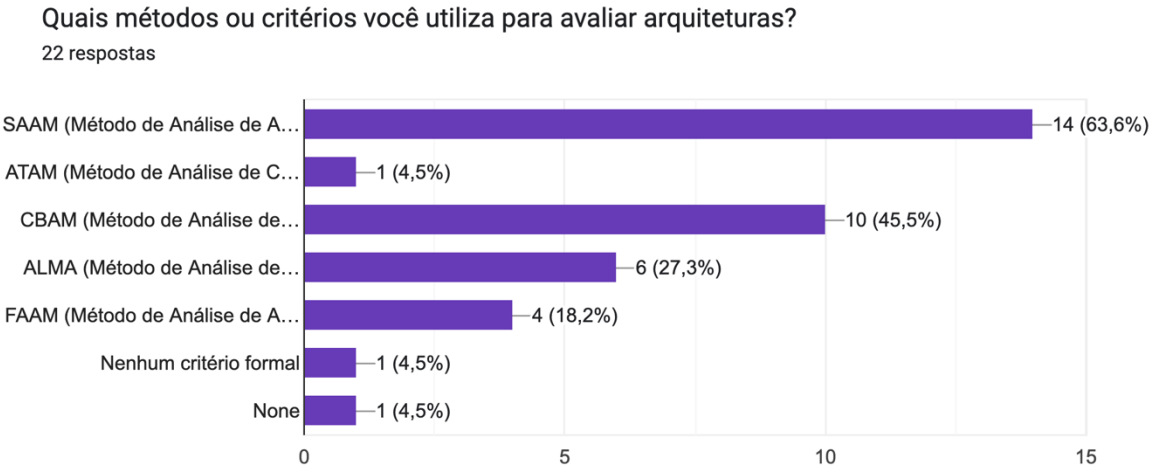


Figura 23. Métodos de avaliação de arquitecturas mais utilizados

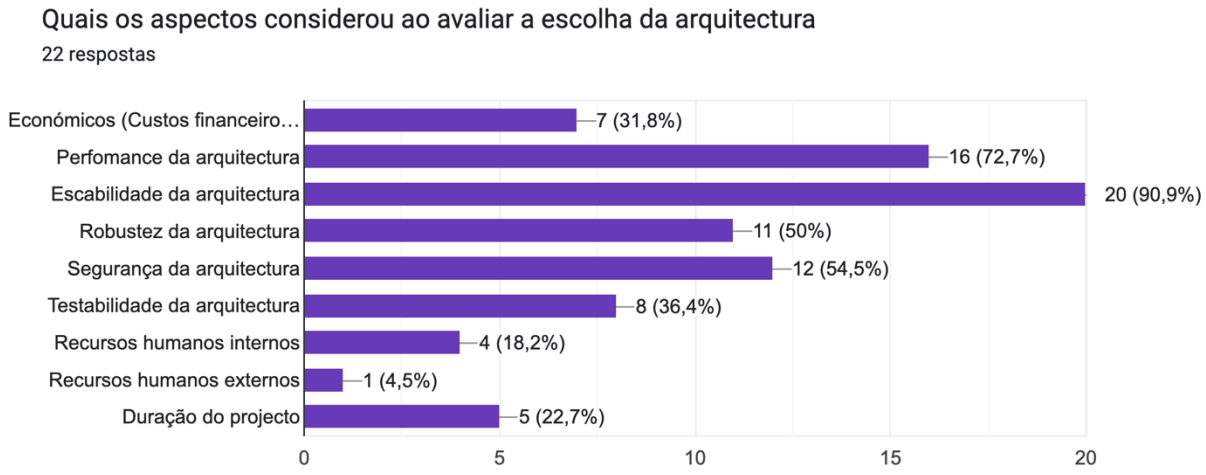


Figura 24. Aspectos mais considerados durante a avaliação das arquitecturas

6. Motivos para Não Avaliar ou Não Usar uma Arquitectura Específica

A falta de tempo aliada a falta de conhecimento sobre a importância da avaliação de arquitecturas é uma barreira comum para avaliação de arquitecturas (figura 25). Enquanto que a complexidade excessiva e falta de conhecimento impedem a utilização de arquitectura adequada ou especifica nos projectos (figura 26).

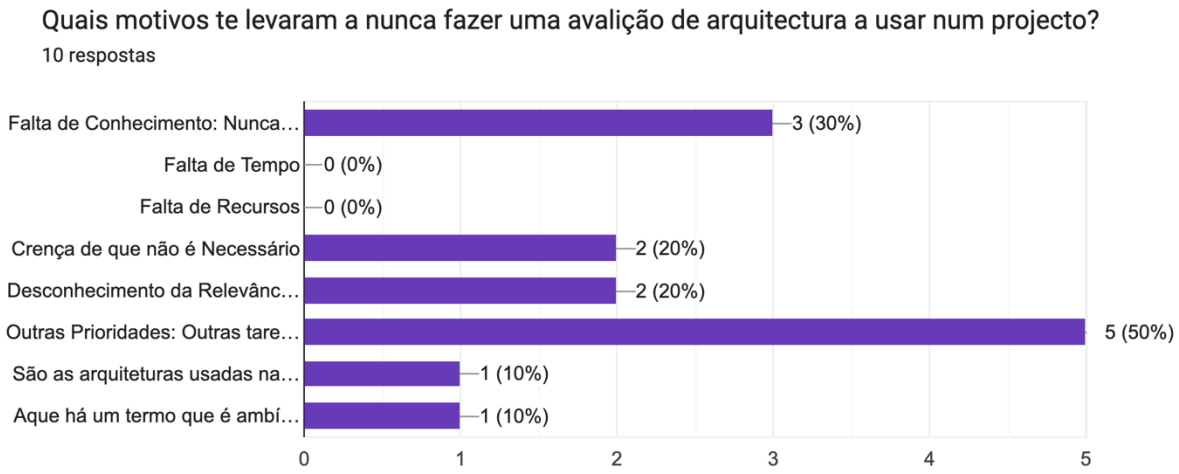


Figura 25. Motivos para nunca ter feito avaliação de arquitecturas

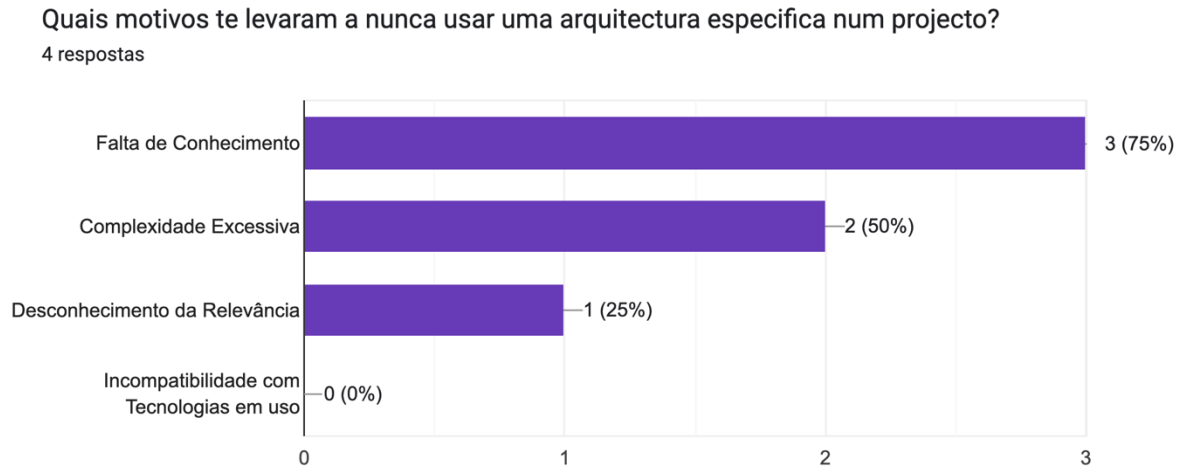


Figura 26. Motivos pra nunca ter utilizado uma arquitectura especifica

5.2 Resultados do ATAM

Nesta seção, serão discutidos os resultados da análise arquitectural realizada utilizando o ATAM para as arquitecturas MVI e MVVM. Serão apresentados os principais achados relacionados aos atributos de qualidade, Tradeoffs identificados e insights obtidos durante o processo de avaliação.

5.2.1. Descrição do Experimento

A implementação do experimento consistiu em um conjunto de casos de uso que executam operações básicas no aplicativo Android. Cada caso de uso foi projectado para representar diferentes tipos de cenários de uso. Esses cenários consistem em diferentes números de acções simples. Esta secção também contém tabelas com as medições obtidas no experimento realizado num emulador com as características descritas na tabela 5.

Os códigos-fonte detalhados dos experimentos, incluindo os casos de uso para as estruturas MVVM e MVI, estão disponíveis nos Apêndices A e B, respectivamente, como resultado da implementação deste experimento.

Tabela 5. Características do emulador utilizado para testes

Propriedades	Descrição
Nome do dispositivo	TECNO MOBILE LIMITÉE TECNO KB7
Versão do sistema operacional	27
Número de série	039213395T030572
RAM	0,9 GB
Processador	Processeur ARMv7 rév. 3 (v71)
Fabricante	TECNO MOBILE LIMITED
Modelo	TECNO KB7

Casos de Uso da Avaliação

Os casos de uso seleccionados para esta avaliação estão no domínio da aplicação apresentada no capítulo 4, que é uma aplicação de registo de tarefas, a seguir segue a descrição dos mesmos:

a) Criação de tarefa

Tabela 6. Caso de uso de criação de tarefa

UC001	
Caso de uso	Criar tarefa
Actor principal	Utilizador
Pré-Condições	Acesso a aplicação

Pós-Condições	Os dados da tarefa são armazenados na base de dados local do dispositivo.
Cenário de Sucesso Principal	1. Abrir a tela de criação de tarefa; 2. Digitar o título; 3. Digitar a descrição; 4. Seleccionar a prioridade; 5. Clicar botão guardar;
Fluxos Alternativos	O sistema informa que os dados não foram preenchidos

b) Editar Tarefa

Tabela 7. Caso de uso de edição de tarefa

UC002	
Caso de uso	Editar tarefa
Actor principal	Utilizador
Pré-Condições	Acesso a aplicação; Existência da tarefa que se pretende editar;
Pós-Condições	Os dados da tarefa são actualizados e armazenados na base de dados local do dispositivo.
Cenário de Sucesso Principal	1. Abrir a tela de listagem de tarefas; 2. Seleccionar uma tarefa existente; 3. Editar o título (opcional); 4. Editar a descrição (opcional); 5. Editar a prioridade (opcional); 6. Clicar botão guardar;
Fluxos Alternativos	O sistema informa que os dados não foram preenchidos

c) Apagar Tarefa

Tabela 8. Caso de uso de eliminação de tarefa

UC003	
Caso de uso	Apagar tarefa
Actor principal	Utilizador
Pré-Condições	Acesso a aplicação; Existência da tarefa que se pretende apagar;
Pós-Condições	Os dados da tarefa são eliminados da base de dados local do dispositivo.

Cenário de Sucesso Principal	1. Abrir a tela de listagem de tarefas; 2. Seleccionar uma tarefa existente; 3. Clicar no botão apagar;
Fluxos Alternativos	Não é aplicável

d) Pesquisar Tarefa

Tabela 9. Caso de uso de pesquisa de tarefas

UC004	
Caso de uso	Pesquisar tarefa
Actor principal	Utilizador
Pré-Condições	Acesso a aplicação
Pós-Condições	Não é aplicável
Cenário de Sucesso Principal	1. Abrir a tela de listagem de tarefas; 2. Clicar no ícone de pesquisa; 3. Introduzir a chave de pesquisa; 4. Clicar no botão de pesquisa;
Fluxos Alternativos	Não é aplicável

e) Filtrar Tarefas por Prioridade

Tabela 10. Caso de uso de filtragem de tarefas

UC005	
Caso de uso	Filtrar tarefas por prioridade
Actor principal	Utilizador
Pré-Condições	Acesso a aplicação
Pós-Condições	Não é aplicável
Cenário de Sucesso Principal	1. Abrir a tela de listagem de tarefas; 2. Clicar no ícone de filtro; 3. Seleccionar a prioridade que deseja;
Fluxos Alternativos	Não é aplicável

f) Visualização de Tarefas

Tabela 11. Caso de uso de visualização de tarefas

UC006	
Caso de uso	Visualização de tarefas
Actor principal	Utilizador
Pré-Condições	Acesso a aplicação
Pós-Condições	Não é aplicável
Cenário de Sucesso Principal	1. Abrir a tela de listagem de tarefas; 2. Clicar numa tarefa específica
Fluxos Alternativos	Não é aplicável

5.2.2. Desempenho

O desempenho será medido usando o *Benchmark* com as métricas ***Startup Timing*** Para medir o tempo de inicialização do aplicativo e ***Frame Timing*** para medir quão rápido o aplicativo pode produzir quadros, também conhecido como tempo de renderização.

O teste de inicialização foi realizado usando os três modos: *COLD, WARM e HOT*. No modo *COLD*, o dispositivo está em um cenário de pior caso, onde nenhum dos processos necessários para a inicialização do aplicativo foi iniciado antecipadamente. O modo *WARM* representa um caso de inicialização médio, onde os recursos necessários para iniciar o aplicativo ainda estavam na memória. O modo *HOT* indica o cenário de melhor caso, onde o aplicativo já estava em execução, mas não em primeiro plano.

Os outros testes de *benchmark* que mediram o *Frame Timing Metric* foram executados apenas no modo *WARM*, uma vez que alterá-lo não seria relevante para a métrica de tempo de renderização de quadros. Todos os testes foram feitos correndo 5 iterações para cada teste.

a) O Teste de Inicialização

Os resultados são representados por 3 valores, o tempo mínimo, médio e máximo em milissegundos.

Tabela 12. Resultados de testes de inicialização do aplicativo ToDo

Cenário/Tempo(ms)	Min	Média	Max
MVVM COLD	3088	3088	3088
MVI COLD	2673,4	2681,2	2689
MVVM WARM	1289,8	1289,8	1289,8

MVI WARM	1010,6	1296,1	1673
MVVM HOT	759,7	759,7	759,7
MVI HOT	266,2	266,2	627

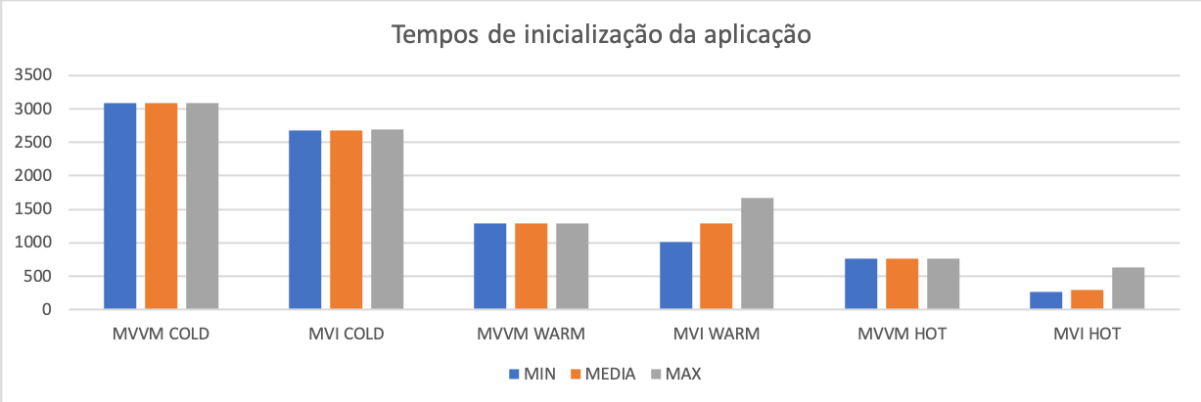


Figura 27. Resultados de testes de inicialização da aplicação.

A tabela 12 mostra os tempos de inicialização da aplicação nos 3 modos, onde no cenário *COLD*, o modelo MVVM apresenta um tempo de inicialização constante de 3088 ms. No mesmo cenário *COLD*, o modelo MVI tem variações nos tempos de inicialização, com um mínimo de 2673,4 ms, uma média de 2681,2 ms e um máximo de 2689 ms. Portanto, neste cenário, o MVI é mais rápido na inicialização em comparação com o MVVM, especialmente em seu valor mínimo. No cenário *WARM*, o modelo MVVM tem um tempo de inicialização constante de 1289,8 ms. No mesmo cenário *WARM*, o modelo MVI apresenta variações nos tempos de inicialização, com um mínimo de 1010,6 ms, uma média de 1296,1 ms e um máximo de 1673 ms. Nesse caso, o MVVM tem um tempo de inicialização mais consistente e ligeiramente mais rápido em comparação com o MVI. No cenário HOT, o modelo MVVM tem um tempo de inicialização constante de 759,7 ms. No mesmo cenário *HOT*, o modelo MVI apresenta variações nos tempos de inicialização, com um mínimo de 266,2 ms, uma média de 297,9 ms e um máximo de 627 ms. Nesse caso, o MVVM também tem um tempo de inicialização mais consistente, mas significativamente maior em comparação com o MVI.

Portanto, ao considerar os diferentes cenários, pode-se observar que o desempenho do MVVM e do MVI varia dependendo das condições de inicialização. O MVVM é mais consistente no tempo de inicialização, enquanto o MVI pode ser mais rápido em cenários de inicialização no modo *COLD*.

b) Testes de tempo de renderização

Para o teste a seguir foram considerados todos os casos de uso excepto o caso de caso de pesquisa (UC004), devido a limitações de implementação de certos comportamentos.

Os resultados são representados por 4 valores em milissegundos:

- **P50 (Percentil 50)**: Este valor indica o tempo que leva para que 50% das medições estejam abaixo desse valor. Em outras palavras, metade das medições foi concluída em menos tempo do que o P50.
- **P90 (Percentil 90)**: Este valor indica o tempo que leva para que 90% das medições estejam abaixo desse valor. Apenas 10% das medições excederam o P90.
- **P95 (Percentil 95)**: Este valor indica o tempo que leva para que 95% das medições estejam abaixo desse valor. Apenas 5% das medições excederam o P95.
- **P99 (Percentil 99)**: Este valor indica o tempo que leva para que 99% das medições estejam abaixo desse valor. Apenas 1% das medições excederam o P99.

Esses percentis são úteis para entender a distribuição dos tempos de processamento. Valores mais baixos nos percentis indicam um desempenho mais rápido, enquanto valores mais altos indicam um desempenho mais lento (Android Developers, 2023a).

Tabela 13. Resultados de testes de tempo de renderização dos casos de uso

Cenário/Tempo(ms)		P50	P90	P95	P99
UC001	MVVM	32,7	94,3	172,2	326,5
	MVI	57,2	203,9	267,1	544,4
UC002	MVVM	32,2	99,5	159,5	244,8
	MVI	47,1	180,3	239,4	340,6
UC003	MVVM	29,2	95,5	148,6	256
	MVI	47,2	148,8	222,9	334
UC005	MVVM	42,1	119,7	207,8	300,1
	MVI	56,3	174,9	244,8	466,3
UC006	MVVM	44,3	166,4	246,9	380,4
	MVI	59	311,3	395,4	656,6

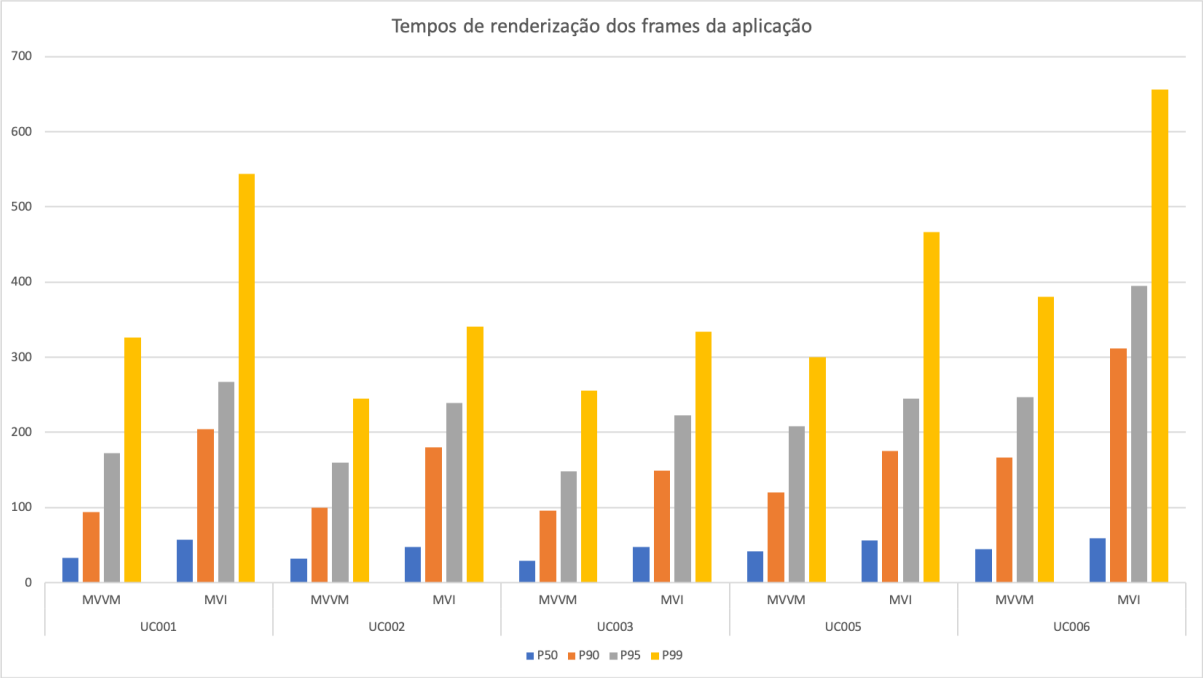


Figura 28. Resultados de testes de tempos de renderização de frames para os casos de uso.

A Tabela 13 mostra que em todos os casos de uso, o MVVM tem tempos mais baixos no percentil (P50, P90, P95 e P99) em comparação com o MVI. Isso sugere que o MVVM tende a ter um desempenho mais rápido na manipulação de quadros ou eventos em todos os cenários. O P50 do MVVM é menor do que o do MVI, o que indica que a maioria das medições do MVVM é mais rápida. Os percentis superiores (P90, P95 e P99) mostram que o MVVM também tem uma vantagem em termos de desempenho nas medições mais lentas. Por exemplo, o P99 do MVVM é muito menor do que o do MVI em todos os casos, o que indica que mesmo nas situações mais lentas, o MVVM tende a ser mais rápido.

5.2.3. Modificabilidade

As métricas de modificabilidade são colectadas para cada funcionalidade. Essas métricas incluem o número de classes que precisam ser criadas para implementar a funcionalidade, o número de classes que precisam ser modificadas para implementar a funcionalidade e o número de linhas de código que precisam ser adicionadas ou modificadas para implementar a funcionalidade.

a) Classes necessárias para implementação do projecto

Para esta análise foram consideradas apenas classes e ficheiros kotlin com relevância para o projecto e fins de comparação, por isso, por exemplo não temos os ficheiros *gradles*, pois estes são constantes para ambas as arquitecturas.

Arquitectura MVI

Tabela 14. Classes e linhas de código necessárias para implementação do projecto na arquitectura MVI

Ficheiro	Total de Linhas	Linhas de código (LOC)	Percentagem de LOC	Linhas Comentadas (LC)	Percentagem de LC	Linhas em Branco (LB)	Percentagem de LB
Action.kt	7	2	29%	4	57%	1	14%
ActionLabels.kt	14	12	86%	0	0%	2	14%
AppBenchmark.kt	216	155	72%	13	6%	48	22%
BaseStore.kt	36	20	56%	10	28%	6	17%
Color.kt	39	30	77%	0	0%	9	23%
Constants.kt	22	16	73%	0	0%	6	27%
DatabaseModule.kt	69	61	88%	0	0%	8	12%
DataStoreRepository.kt	53	46	87%	0	0%	7	13%
Dimensions.kt	12	8	67%	0	0%	4	33%
DisplayAlertDialog.kt	56	53	95%	0	0%	3	5%
EmptyContent.kt	47	44	94%	0	0%	3	6%
FakeDataStoreRepository.kt	19	15	79%	0	0%	4	21%
FakeDataStoreRepository.kt	19	15	79%	0	0%	4	21%
FakeToDoTaskRepository.kt	74	58	78%	4	5%	12	16%
FakeToDoTaskRepository.kt	57	45	79%	0	0%	12	21%
IDataStoreRepository.kt	11	8	73%	0	0%	3	27%
InstrumentedTests.kt	255	221	87%	0	0%	34	13%
IToDoRepository.kt	23	14	61%	0	0%	9	39%

ListAppBar.kt	273	249	91%	0	0%	24	9%
ListComposable.kt	30	27	90%	0	0%	3	10%
ListContent.kt	334	312	93%	0	0%	22	7%
ListScreen.kt	159	148	93%	0	0%	11	7%
MainActivity.kt	30	27	90%	0	0%	3	10%
Middleware.kt	21	8	38%	12	57%	1	5%
Navigation.kt	31	29	94%	0	0%	2	6%
Priority.kt	14	12	86%	0	0%	2	14%
PriorityDropDown.kt	129	123	95%	0	0%	6	5%
PriorityItem.kt	37	34	92%	0	0%	3	8%
Reducer.kt	16	4	25%	10	63%	2	13%
RequestState.kt	10	7	70%	0	0%	3	30%
Screens.kt	17	14	82%	0	0%	3	18%
SearchAppBarState.kt	7	6	86%	0	0%	1	14%
Shape.kt	11	9	82%	0	0%	2	18%
SharedViewModel.kt	184	154	84%	0	0%	30	16%
SharedViewModelTest.kt	179	156	87%	0	0%	23	13%
State.kt	7	2	29%	4	57%	1	14%
Store.kt	13	6	46%	3	23%	4	31%
TaskAppBar.kt	220	182	83%	17	8%	21	10%
TaskComposable.kt	50	44	88%	0	0%	6	12%
TaskContent.kt	91	71	78%	12	13%	8	9%
TaskMiddleware.kt	408	297	73%	63	15%	48	12%
TaskReducer.kt	97	71	73%	5	5%	21	22%
TasksActions.kt	58	44	76%	0	0%	14	24%
TaskScreen.kt	73	64	88%	0	0%	9	12%
TasksStore.kt	33	21	64%	10	30%	2	6%

TaskViewState.kt	33	30	91%	0	0%	3	9%
Theme.kt	44	30	68%	8	18%	6	14%
ToDoApplication.kt	7	5	71%	0	0%	2	29%
ToDoDao.kt	57	32	56%	14	25%	11	19%
ToDoDatabase.kt	10	8	80%	0	0%	2	20%
ToDoRepository.kt	40	31	78%	0	0%	9	23%
ToDoTask.kt	14	12	86%	0	0%	2	14%
Type.kt	28	13	46%	13	46%	2	7%
Total:	3794	3105	82%	202	5%	487	13%

Arquitectura MVVM

Tabela 15. Classes e linhas de código necessárias para implementação do projecto na arquitectura MVVM

Ficheiro	Total de Linhas	Linhas de código (LOC)	Percentagem de LOC	Linhas Comentadas (LC)	Percentagem de LC	Linhas em Branco (LB)	Percentagem de LB
Action.kt	14	12	86%	0	0%	2	14%
AppBenchmark.kt	215	154	72%	13	6%	48	22%
Color.kt	39	30	77%	0	0%	9	23%
Constants.kt	22	16	73%	0	0%	6	27%
DatabaseModule.kt	44	38	86%	0	0%	6	14%
DataStoreRepository.kt	53	46	87%	0	0%	7	13%
Dimensions.kt	12	8	67%	0	0%	4	33%
DisplayAlertDialog.kt	56	53	95%	0	0%	3	5%
EmptyContent.kt	47	44	94%	0	0%	3	6%
FakeDataStoreRepository.kt	19	15	79%	0	0%	4	21%
FakeDataStoreRepository.kt	19	15	79%	0	0%	4	21%

FakeToDoTaskRepository.kt	57	45	79%	0	0%	12	21%
FakeToDoTaskRepository.kt	74	58	78%	4	5%	12	16%
IDataStoreRepository.kt	11	8	73%	0	0%	3	27%
InstrumentedTests.kt	247	212	86%	0	0%	35	14%
IToDoRepository.kt	23	14	61%	0	0%	9	39%
ListAppBar.kt	270	247	91%	0	0%	23	9%
ListComposable.kt	47	42	89%	0	0%	5	11%
ListContent.kt	316	297	94%	0	0%	19	6%
ListScreen.kt	165	149	90%	0	0%	16	10%
MainActivity.kt	30	27	90%	0	0%	3	10%
Navigation.kt	31	29	94%	0	0%	2	6%
Priority.kt	14	12	86%	0	0%	2	14%
PriorityDropDown.kt	130	124	95%	0	0%	6	5%
PriorityItem.kt	38	35	92%	0	0%	3	8%
RequestState.kt	10	7	70%	0	0%	3	30%
Screens.kt	17	14	82%	0	0%	3	18%
SearchAppBarState.kt	7	6	86%	0	0%	1	14%
Shape.kt	11	9	82%	0	0%	2	18%
SharedViewModel.kt	248	217	88%	0	0%	31	13%
SharedViewModelTest.kt	137	116	85%	0	0%	21	15%
TaskAppBar.kt	191	172	90%	0	0%	19	10%
TaskComposable.kt	48	42	88%	0	0%	6	13%
TaskContent.kt	86	79	92%	0	0%	7	8%
TaskScreen.kt	69	61	88%	0	0%	8	12%
Theme.kt	44	30	68%	8	18%	6	14%
ToDoApplication.kt	7	5	71%	0	0%	2	29%
ToDoDao.kt	58	33	57%	14	24%	11	19%

ToDoDatabase.kt	10	8	80%	0	0%	2	20%
ToDoRepository.kt	40	31	78%	0	0%	9	23%
ToDoTask.kt	14	12	86%	0	0%	2	14%
Type.kt	28	13	46%	13	46%	2	7%
Total:	3018	2585	86%	52	2%	381	12%

Das tabelas 14 e 15, pode constatar-se que:

Linhas de Código (LOC): A arquitectura MVI possui um total de 3105 linhas de código, enquanto a MVVM possui 2585 linhas de código. Isso significa que a MVI têm mais linhas de código em comparação com a MVVM.

Percentagem de LOC: A MVVM possui uma percentagem ligeiramente maior de linhas de código (86%) em comparação com a MVI (82%). Isso pode indicar que a MVVM tem uma proporção um pouco maior de código em relação ao total de linhas.

Linhas Comentadas (LC): A MVI tem 202 linhas comentadas, enquanto a MVVM tem apenas 52. Isso sugere que a MVVM pode ter uma documentação mais escassa ou que a MVI está mais bem documentada.

Percentagem de LC: A MVI tem uma percentagem de linhas comentadas de 5%, enquanto a MVVM tem apenas 2%. Isso indica que a MVI tem uma documentação mais extensa em comparação com a MVVM.

Linhas em Branco (LB): Ambas as arquitecturas têm uma percentagem quase semelhante de linhas em branco, com 13% e 12% cada. Isso sugere que a formatação do código é semelhante em ambas as arquitecturas.

Com base nos dados fornecidos, a MVVM parece ter uma base de código um pouco menor que a MVI. No entanto, é importante ressaltar que esses números por si só não são suficientes para determinar a qualidade ou eficácia de uma arquitectura.

b) Classes que precisam ser criadas para implementar cada caso de uso

Tabela 16. Número de classes que precisam ser criadas para implementar cada caso de uso.

Funcionalidade/Arquitectura	MVVM	MVI
Estrutura base	2	9
Camada de domínio	7	11
UC001	3	3
UC002	3	3
UC003	4	4
UC004	1	1
UC005	3	3
UC006	3	3

A Tabela 16 mostra o número de classes que precisam ser adicionadas para implementar cada recurso específico em ambas as arquitecturas.

c) Classes que precisam ser modificadas para implementar cada caso de uso

Tabela 17. Número de classes que precisam ser criadas para implementar cada caso de uso.

Funcionalidade/Arquitectura	MVVM	MVI
UC001	2	7
UC002	2	7
UC003	2	7
UC004	2	7
UC005	2	7
UC006	2	7

A Tabela 17 mostra o número de classes que precisam ser adicionadas para editar cada recurso específico em ambas as arquitecturas.

d) Linhas de código necessárias para implementar cada caso de uso

Tabela 18. Número de linhas de código necessários para implementar ou editar cada caso de uso.

Funcionalidade/Arquitectura	MVVM	MVI
Camada de infraestrutura	51	119
Camada de domínio	226	864
UC001	425	689
UC002	425	625
UC003	531	613
UC004	798	896
UC005	980	1315
UC006	415	1176

A Tabela 18 mostra o número de classes que precisam ser adicionadas para implementar cada recurso específico em ambas as arquitecturas.

e) Coesão do código

Para avaliar o nível de coesão, é essencial examinar cada componente da arquitectura. No contexto do MVI (Model-View-Intent), a coesão é efectivamente alcançada pela subdivisão dos componentes em três partes principais: Modelo (Model), Visualização (View) e Intenção (*Intent*). Cada um desses componentes desempenha uma função distintamente definida. No geral, o MVI promove uma coesão de alto grau no nível funcional, uma vez que cada componente possui responsabilidades claramente delineadas e bem definidas.

Similarmente, o MVVM (Model-View-ViewModel) também se destaca na promoção da coesão. Cada componente desempenha um papel bem definido e não interfere nas responsabilidades dos demais. Consequentemente, tanto o MVI quanto o MVVM exibem um alto grau de coesão a nível funcional.

f) Acoplamento do código

O nível de acoplamento descreve as interacções entre diferentes componentes que são mais complexas. Como discutido na seção 2.2.1.1, os dois cenários de modificação mais comuns são adicionar novos recursos e corrigir erros. Para tornar isso mais claro, a figura mostra um diagrama de sequência de eventos que representa as interacções necessárias entre diferentes componentes para realizar a maioria dos casos de uso.

1. Diagrama de Sequência de eventos para MVVM

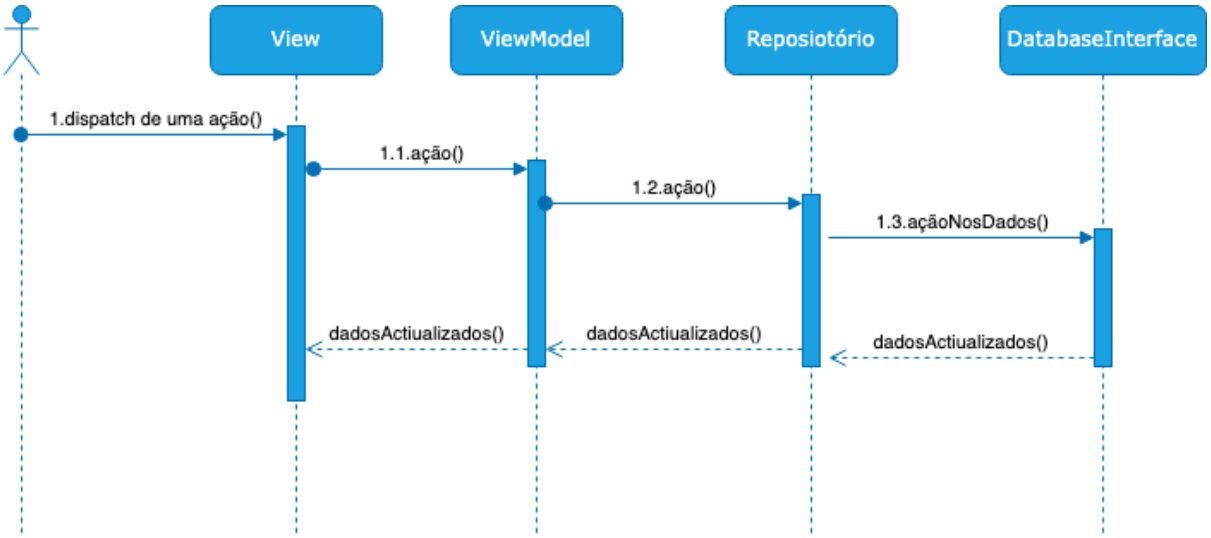


Figura 29. Diagrama de sequência de eventos para MVVM.

A figura 29 mostra as relações existentes entre os diferentes componentes na arquitectura MVVM, que é resumida na tabela 19.

Tabela 19. Resumo das relações e nível de acoplamento dos componentes na arquitectura MVVM.

Fonte	Destino	Conexão	Nível de Acoplamento
View	ViewModel	Envia acções pela interface do utilizador	Mensagem
ViewModel	Repositório	Enviar parâmetros para alteração de dados	Dados estruturados
Repositório	DatabaseInterface	Enviar parâmetros para alteração de dados	Dados estruturados
DatabaseInterface	Repositório	Envia resposta para apresentação de dados	Dados estruturados

Repositório	ViewModel	Envia resposta para apresentação de dados	Dados estruturados
ViewModel	View	Envia dados	Mensagem e Controle

Na arquitectura MVVM, o nível de acoplamento varia de Mensagem (comunicação entre View e ViewModel) a Dados Estruturados (comunicação entre outros componentes). A maior parte da comunicação é baseada em dados estruturados, o que indica que há uma forte relação de dados entre esses componentes, entretanto possui poucas conexões, o que pode representar menos complexidade na manipulação e propagação de dados.

2. Diagrama de Sequência de eventos para MVI

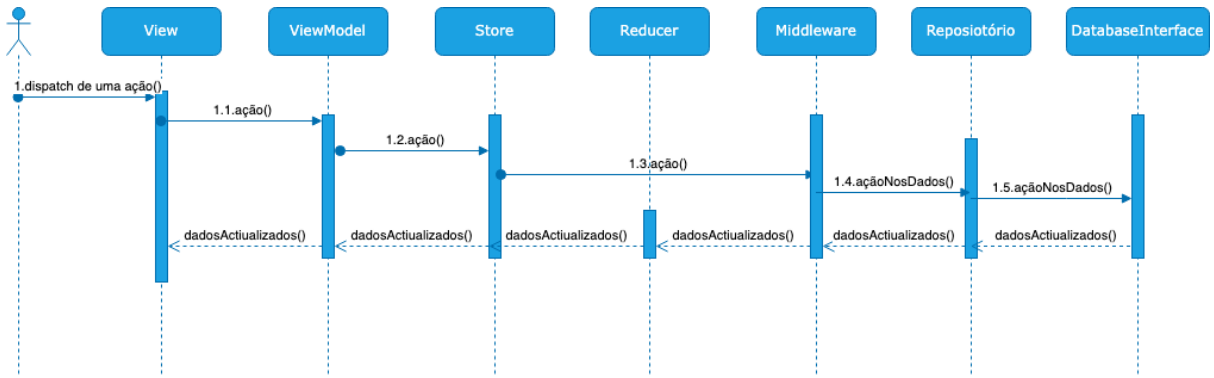


Figura 30. Diagrama de sequência de eventos para MVI.

A figura 30 mostra as relações existentes entre os diferentes componentes na arquitectura MVI, que é resumida na tabela 20.

Tabela 20. Resumo das relações e nível de acoplamento dos componentes da arquitectura MVI

Fonte	Destino	Conexão	Nível de Acoplamento
View	ViewModel	Envia acções pela interface do utilizador	Mensagem
ViewModel	Store	Enviar parâmetros para alteração de dados	Dados estruturados

Store	Middleware	Enviar parâmetros para alteração de dados	Dados estruturados
Middleware	Repositório	Enviar parâmetros para alteração de dados	Dados estruturados
Repositório	DatabaseInterface	Enviar parâmetros para alteração de dados	Dados estruturados
DatabaseInterface	Repositório	Envia resposta para apresentação de dados	Dados estruturados
Repositório	Middleware	Envia resposta para apresentação de dados	Dados estruturados
Middleware	Reducer	Envia resposta para apresentação de dados	Dados estruturados
Reducer	Store	Envia resposta para apresentação de dados	Dados estruturados
Store	ViewModel	Envia resposta para apresentação de dados	Dados estruturados
ViewModel	View	Envia dados	Mensagem e Controle

Na arquitetura MVI, o nível de acoplamento varia desde a comunicação por meio de Mensagens (entre View e ViewModel) até a troca de Dados Estruturados (entre outros componentes). No entanto, ao introduzir componentes adicionais, como *Store*, *Middleware* e *Reducer*, e distribuir responsabilidades de maneira mais específica entre eles, é possível alcançar um acoplamento mais reduzido em termos de dados estruturados. Isso resulta em uma relação mais flexível entre os componentes, facilitando a manutenção e a extensão do sistema, uma vez que as mudanças em um componente têm menos impacto nos outros.

Em termos de acoplamento, com os dados apresentados podemos notar que:

- Ambas as arquitecturas têm uma comunicação principal entre View e ViewModel usando Mensagens e Controle.
- Tanto MVVM quanto MVI têm uma forte relação de dados estruturados entre os componentes, especialmente nas camadas intermediárias (Repositório, DatabaseInterface, Middleware, Store, Reducer).
- MVI introduz componentes adicionais (Store, Middleware, Reducer) que aumentam a complexidade mas reduzem o nível de acoplamento em relação à MVVM.

Em resumo, a arquitectura MVI tende a ter um nível de acoplamento de dados estruturados um menor devido à introdução de componentes intermediários, enquanto a MVVM é mais alto o nível de acoplamento.

5.2.4. Testabilidade

1) Testes na Arquitectura MVVM

Na arquitectura MVVM, os testes se concentram principalmente na classe ***Shared ViewModel***, onde foram implementados e executados com sucesso 7 casos de teste, tanto para testes unitários como para testes instrumentados.

Para viabilizar esses testes, foi necessário criar dois repositórios de dados falsos, nomeados como **FakeToDoTaskRepository** e **FakeDataStoreRepository**. Esses repositórios fictícios desempenharam um papel fundamental na agilização dos testes, garantindo que eles pudessem ser executados de forma rápida e eficiente.

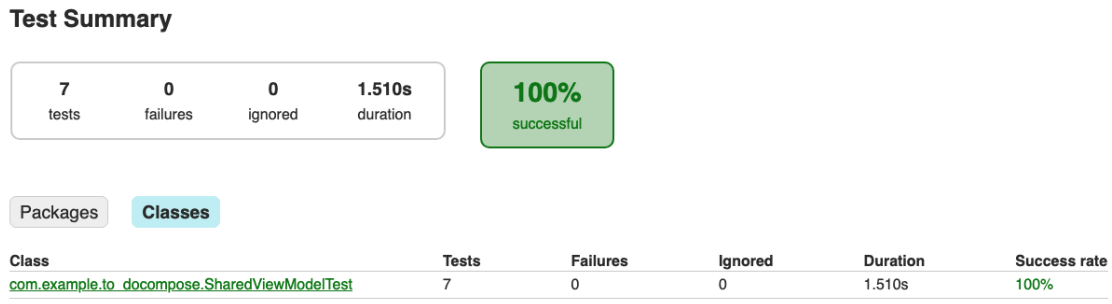


Figura 31. Resultados de testes unitários da arquitectura MVVM.

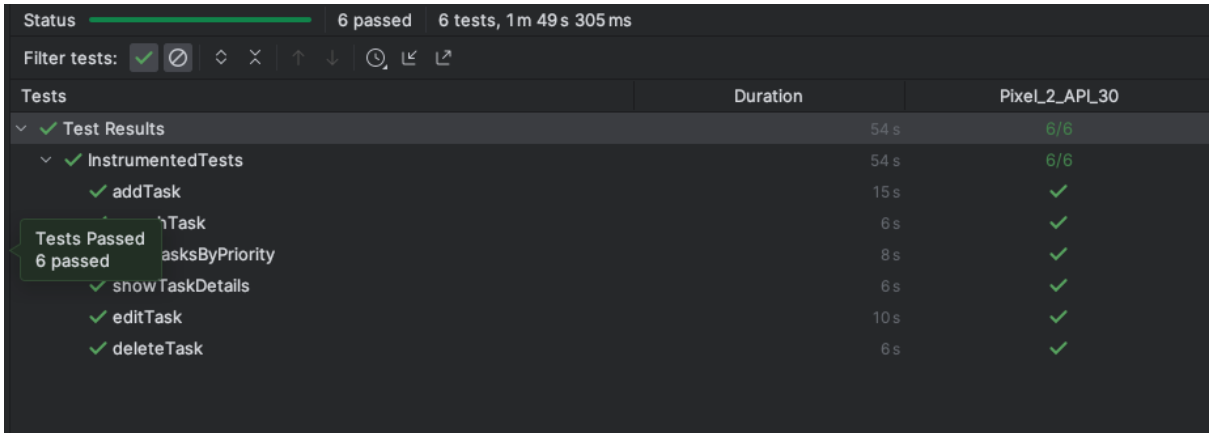


Figura 32. Resultados de testes instrumentados da arquitectura MVVM.

2) Testes na Arquitectura MVI

Na arquitectura MVI, de igual modo, os testes se concentram principalmente na classe **SharedViewModel**, onde foram implementados e executados com sucesso 7 casos de teste. Esses testes foram concluídos com êxito em 0.255 segundos, conforme indicado na Figura 18. Para viabilizar esses testes, foi necessário criar dois repositórios de dados falsos, nomeados como **FakeToDoTaskRepository** e **FakeDataStoreRepository**. Esses repositórios fictícios desempenharam um papel fundamental na agilização dos testes, garantindo que eles pudessem ser executados de forma rápida e eficiente.

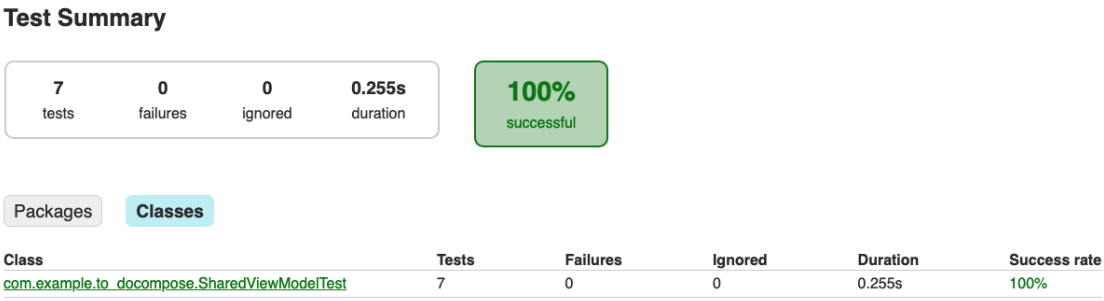


Figura 33. Resultados de testes da arquitectura MVI.

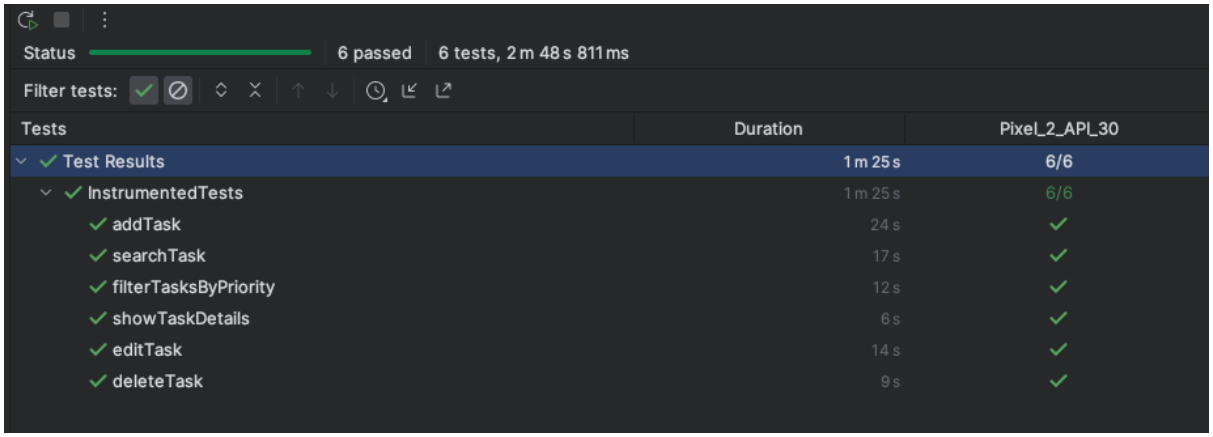


Figura 34. Resultados de testes instrumentados da arquitectura MVI.

Para ambas as arquitecturas foram realizados testes unitários e instrumentados, conforme mostram os resultados acima (figuras 21,32,33 e 34), com isto, pode-se se concluir que:

1. **Tamanho dos casos de teste**: neste critério ambas arquitecturas apresentam praticamente o mesmo tamanho.
2. **Tempo consumido para executar os casos de teste**: para este critério os testes da arquitectura MVVM executaram em menos tempo, somando o total de 1 minuto e 49 segundos e 305 milissegundos, enquanto no que os do MVI somaram o total de 2 minutos e 48 segundos e 811 milissegundos.
3. **Facilidade de depuração com pontos de interrupção**: Nas duas arquitecturas, o processo é o mesmo. Com a ajuda de uma biblioteca de terceiros, os resultados são fornecidos directamente após a execução do caso de teste. No Android Studio, os casos de teste aprovados são exibidos em verde, enquanto os casos que falharam aparecem em vermelho com as razões.
4. **Facilidade de escrever os testes**: A arquitetura MVI demonstrou uma maior facilidade na elaboração de testes em comparação com a arquitetura MVVM. Essa observação pode ser atribuída à natureza modular e desacoplada do MVI, que favorece a criação de testes específicos para componentes individuais ou cenários particulares.

Desta forma, quanto à testabilidade, pelo critério do tempo consumido na execução dos casos de testes, a arquitectura MVVM ganha, entretanto quando a facilidade de teste dos componentes o MVI sai-se melhor.

5.3 Resultados do CBAM

Nesta seção, serão apresentados os resultados da análise económica realizada utilizando o CBAM para as arquitecturas MVI e MVVM. Serão discutidos os custos e benefícios associados a cada arquitectura, bem como o retorno sobre o investimento (ROI) estimado.

5.3.1 Seleccionar Cenários de Interesse e suas Estratégias Arquitecturais Associadas

Um conjunto dos cenários brutos colectados na avaliação ATAM serviram de base para esta análise conforme a tabela 21.

Tabela 21. Cenários colectados em ordem de prioridade.

Cenário	Descrição do Cenário
1	Reduzir o tempo médio de inicialização da aplicação
2	Reduzir o tempo de renderização das telas de visualização de detalhes da tarefa de 99% das vezes
3	Reduzir o tempo de renderização das telas na adição de uma tarefa.
4	Reduzir o tempo de renderização na Filtragem de tarefas por prioridade
5	Reduzir a complexidade para escalabilidade da aplicação

5.3.2 Avaliar Benefícios dos Atributos de Qualidade

Os cenários foram refinados, dando atenção especial à especificação precisa de suas medidas estímulo-resposta. Os objectivos de resposta para o pior caso, caso actual, caso desejado e melhor caso de cada cenário foram obtidos e registrados, conforme mostrado na Tabela 22.

Tabela 22. Objectivos de resposta para os cenários refinados.

Cenário	Pior Caso	Caso Desejado	Melhor Caso
1	>3100 ms	<3088 ms	<2500 ms

2	> 344 ms	<340 ms	<244 ms
3	>545 ms	<544 ms	<326 ms
4	>467 ms	<467 ms	<300 ms
5	>5 classes	<=4 classes	<3 classes

5.3.3 Priorizar Cenários

Os votos alocados para o conjunto completo de cenários foram limitados a 100, conforme mostrado na tabela 23.

Tabela 23. Cenários com os respectivos votos.

Cenário	Votos	Pior Caso	Caso Desejado	Melhor Caso
1	25	>3100 ms	<3088 ms	<2500ms
2	20	> 344 ms	<340 ms	<244 ms
3	25	>545 ms	<544 ms	<326 ms
4	10	>467 ms	<467 ms	<300 ms
5	20	>5 classes	<=4 classes	<3 classes

5.3.4 Atribuir Utilidade

Neste passo, a utilidade de cada cenário foi determinada, pela utilidade de cada cenário para o objectivo final da aplicação. Uma pontuação de utilidade de 0 representa nenhuma utilidade; uma pontuação de 100 representava a utilidade máxima possível. Os resultados desse processo estão apresentados na Tabela 24.

Tabela 24. Grau de utilidade de cada cenário.

Cenário	Votos	Pior Caso	Caso Desejado	Melhor Caso
1	25	10	80	100
2	20	10	75	100
3	25	0	85	100
4	10	15	65	100
5	20	20	75	100

5.3.5 Desenvolver Estratégias Arquitectónicas para Cenários e Determinar Seus Níveis de Resposta Esperados de Atributos de Qualidade

O conjunto de estratégias arquitectónicas, juntamente com a determinação dos cenários que abordam, é mostrado na Tabela 25. Para cada par estratégico arquitectónica/cenário, são mostrados os níveis de resposta esperados com relação a esse cenário (juntamente com a resposta actual, para fins de comparação).

Tabela 25. Estratégias arquitectónicas implementadas para cada cenário.

Estratégias Arquitectónicas	Descrição	Cenários	Resposta Actual	Resposta Esperada
1	Model -View-ViewModel (MVVM)	1	3088 ms	<3088 ms
		2	244,8ms	<340 ms
		3	326,5ms	<544 ms
		4	300,1ms	<467 ms
		5	3 a 4 classes	<=4 classes
2	Model-View-Intent (MVI)	1	2681,2ms	<3088 ms
		2	340,6ms	<340 ms

		3	544,4 ms	<544 ms
		4	466,3ms	<467 ms
		5	3 a 4 classes	<=4 classes

5.3.6 Calcular o Benefício Total Obtido de uma Estratégia Arquitectónica

Com base nas informações colectadas, conforme representado na Tabela 26, o benefício total de cada estratégia arquitectónica pode agora ser calculado, seguindo a equação 1:

Tabela 26. Estratégias arquitectónicas com os pesos dos seus resultados actuais e esperados.

Estratégias Arquitectónicas	Descrição	Cenários	Resposta Actual	Resposta Esperada
1	Model -View-ViewModel (MVVM)	1	45	80
		2	75	75
		3	85	85
		4	65	65
		5	50	75
2	Model-View-Intent (MVI)	1	80	80
		2	45	75
		3	40	85
		4	30	65
		5	50	75

Tabela 27. Benefício das estratégias arquitectónicas.

Estratégias Arquitectónicas	Cenários	Peso dos Cenários	Benefício Bruto da Estratégia Arquitectónica	Benefício Normalizado da Estratégia Arquitectónica	Benefício Total da Estratégia Arquitectónica
1	1	25	-35	-875	1200
	2	20	30	600	
	3	25	45	1125	
	4	10	35	350	
	5	20	0	0	
2	1	25	35	875	-1200
	2	20	-30	-600	
	3	25	-45	-1125	
	4	10	-35	-350	
	5	20	0	0	

5.3.7 Escolher Estratégias Arquitectónicas com Base no VFC sujeito a Restrições de Custo

Para completar a análise, estimou-se o custo de cada estratégia arquitectónica. As estimativas foram baseadas na experiência com a aplicação, e um retorno sobre o investimento para cada estratégia arquitectónica foi calculado. Usando o VFC, foi possível classificar cada estratégia. Isso é mostrado na Tabela 28.

Tabela 28. Valor pelo custo (VFC) das estratégias arquitectónicas.

Estratégia	custo	Benéfico total da estratégia	VFC da estratégia	Classificação da estratégia
1	3018	1200	0.397614	1
2	3794	-1200	-0.31629	2

Com isto concluímos a avaliação pelo método CBAM, cuja conclusão é apresentada na tabela 28, onde o modelo MVVM tem maior valor por custo.

5.4 Comparação e Selecção da Arquitectura

Nesta seção, serão comparadas as arquitecturas MVI e MVVM com base nos resultados obtidos no ATAM e CBAM. A selecção final da arquitectura será fundamentada em considerações técnicas e económicas, levando em conta os atributos de qualidade e o valor económico agregado.

5.4.1 Comparação das Arquitecturas

Com base nos dados recolhidos do experimento, pode se fazer algumas considerações gerais quanto aos:

Custos

a) Custo de Desenvolvimento

O custo de desenvolvimento incluiria o tempo e os recursos necessários para implementar cada arquitectura. Neste caso, a arquitectura MVI pode exigir um pouco mais de linhas de código e, portanto, mais tempo de desenvolvimento em comparação com a MVVM. No entanto, esses custos podem ser mitigados pela maior facilidade de teste e pela documentação mais extensa da MVI.

b) Custo de Manutenção

A manutenção de software é um custo contínuo ao longo do ciclo de vida de um aplicativo. A arquitectura MVI introduz componentes adicionais, como Store, Middleware e Reducer, o que pode aumentar a complexidade e o custo de manutenção em comparação com a MVVM. No

entanto, a MVI também pode ser mais testável, o que pode facilitar a manutenção.

Benefícios

a) Desempenho

A arquitectura MVI mostra ter um desempenho ligeiramente melhor em cenários de inicialização COLD, enquanto a MVVM é mais consistente em cenários WARM e HOT. Os benefícios de desempenho podem variar dependendo dos requisitos específicos do seu aplicativo e das preferências do usuário.

b) Testabilidade

A MVVM demonstrou ser mais rápida na execução de testes unitários em comparação com a MVI. Isso pode resultar em economia de tempo durante o desenvolvimento e teste do aplicativo, o que é um benefício tangível. Entretanto o MVI pode ser mais testável devido à sua estrutura mais modular e à introdução de componentes intermediários, como Store e Middleware. Isso pode facilitar a criação de testes unitários mais abrangentes.

c) Modificabilidade

A MVVM mostra ter uma base de código ligeiramente menor e requer menos linhas de código para implementar e editar casos de uso novos. Isso pode simplificar o processo de desenvolvimento e manutenção.

d) Coesão e Acoplamento

Ambas as arquitecturas exibem níveis de coesão adequados, com componentes bem definidos. Quanto ao acoplamento, a MVI introduz componentes adicionais, o que pode influenciar o acoplamento em comparação com a MVVM. A complexidade resultante pode ser tanto uma desvantagem quanto uma vantagem, dependendo da experiência da equipe de desenvolvimento. Em alguns cenários, um maior acoplamento pode ser benéfico, especialmente quando os componentes necessitam de extensa comunicação e compartilhamento de dados.

e) Retorno sobre o Investimento (ROI)

O ROI depende dos custos e benefícios específicos do seu projecto e das prioridades da sua equipe. Se a prioridade for o desempenho em cenários COLD, a MVI pode oferecer um ROI positivo. Se a prioridade for a facilidade de desenvolvimento e manutenção, a MVVM pode ser mais eficaz em termos de ROI.

5.4.2 Selecção da Arquitectura

A escolha final da arquitectura para o projecto desta tese é a MVVM. Isso se baseia em uma análise de custos e benefícios, onde consideramos o custo de desenvolvimento, o custo de manutenção, o desempenho, a testabilidade, a modificabilidade, a coesão, o acoplamento e o retorno sobre o investimento (ROI).

Os resultados da análise de custos e benefícios, apoiados pelo CBAM, indicaram que a arquitectura MVVM é mais adequada para este projecto. Embora a MVI tenha vantagens de desempenho em cenários *COLD* e seja altamente testável, a MVVM oferece um equilíbrio mais favorável entre custos e benefícios. Ela tem custos de desenvolvimento e manutenção potencialmente menores, é eficaz em cenários *WARM* e *HOT*, oferece uma boa testabilidade e uma base de código mais enxuta.

Recomendações para escolha de arquitectura:

a) Projectos Maiores:

Na esfera de projectos mais extensos, a arquitetura MVI destaca-se como vantajosa para manter o código organizado e modular. Essa abordagem promove uma clara separação de preocupações, facilitando a escalabilidade e manutenção do código ao longo do tempo. Por outro lado, MVVM é preferível quando o objetivo é manter a *codebase* enxuta, fornecendo uma estrutura elegante para projectos de grande escala.

b) Projectos Médios:

Ao lidar com projectos de tamanho intermediário, é imperativo avaliar requisitos específicos. A adopção de MVI é aconselhável para equipes experientes, aproveitando seus benefícios de modularidade e organização. Por outro lado, MVVM pode ser preferível quando a prioridade é a simplicidade e facilidade de implementação.

c) Projectos Menores:

Para projectos de menor porte, a escolha entre MVI e MVVM deve ser estrategicamente considerada. MVVM é uma escolha segura, oferecendo uma estrutura que é fácil de entender e implementar rapidamente. No entanto, MVI pode ser considerada se a equipe estiver confortável lidando com a complexidade adicional, proporcionando potencial para expansão futura.

5.5 Contexto Teórico e Objectivos

O presente estudo foi realizado com o objectivo de proporcionar uma abordagem mais actualizada e alinhada ao contexto na selecção de arquitecturas de desenvolvimento para aplicativos Android. Para isso, considerou-se não apenas critérios técnicos, mas também factores financeiros que podem influenciar a decisão. A presente pesquisa buscou preencher a lacuna existente na literatura e na comunidade de desenvolvedores ao explorar como as arquitecturas se comportam em diferentes contextos, levando em consideração elementos relacionados à parte técnica e financeira.

5.6 Comparação com Pesquisas Anteriores

Comparando os resultados desta tese com pesquisas anteriores, nota-se que há uma tendência consistente em reconhecer a importância de seleccionar a arquitectura com base nas necessidades e restrições específicas do projecto. Os estudos anteriores, como os de Humeniuk (2018), Lou (2016) e Samuel & Barbosa (2022), também concluíram que não existe uma solução absolutamente ideal que se aplique a todos os cenários. Cada arquitectura tem seus pontos fortes e fracos, e a escolha depende dos requisitos específicos do projecto. Estas conclusões são confirmadas pelos participantes do inquérito, onde vê-se que por exemplo desenvolvedores de aplicações web usam na sua maioria a arquitectura web, e os desenvolvedores Android MVVM ou MVI.

Em particular, no estudo de Samuel & Barbosa (2022) observaram que o MVVM possui um desempenho superior em comparação aos outros padrões e é ideal para projectos maiores, indo em parte de acordo com as verificações do presente estudo, que chegou à mesma conclusão em relação ao desempenho, entretanto quanto a adequação para projectos maiores, para o presente estudo, a escolha vai depender de quais métricas o dono do projecto vai considerar mais, uma vez que cada uma dessas arquictecturas apresenta vantagens ideais para projectos maiores.

5.7 Valor e Implicações Práticas e Teóricas

Os resultados deste estudo têm implicações práticas significativas para desenvolvedores e equipes de projecto de aplicativos Android. Esta abordagem, que inclui a avaliação de custos e benefícios, pode ajudar as equipes a tomar decisões mais informadas sobre qual arquitectura

adoptar. Destacamos a importância de considerar não apenas os aspectos técnicos, mas também as implicações financeiras, o que pode resultar em um ROI mais favorável.

Além disso, teoricamente, a presente pesquisa contribui para uma compreensão mais holística da selecção de arquitecturas de desenvolvimento, incorporando aspectos económicos à análise, pois de acordo com os resultados do inquérito notou-se que a maioria dos desenvolvedores que realizam avaliações de arquitecturas não consideram aspectos económicos, em parte isto é influenciado por falta de conhecimento. Nesse sentido além de estar a contribuir com o presente relatório apresentando o passo a passo de realização de uma avaliação de arquitectura espera-se que este trabalho possa inspirar pesquisas futuras que explorem ainda mais a intersecção entre tecnologia e finanças no contexto do desenvolvimento de software.

5.8 Generalização dos Resultados

É importante notar que os resultados deste estudo podem ser generalizados para uma variedade de contextos de desenvolvimento de aplicativos, não apenas Android. No entanto, a aplicabilidade exacta das descobertas pode variar dependendo das características específicas de cada projecto, como aspectos técnicos (tecnologia em uso, Frameworks, etc), tamanho da equipe, orçamento disponível e metas de desempenho.

5.9 Limitações e Pontos Fortes

Uma limitação deste estudo é que a análise de custos e benefícios foi realizada de forma não tão exaustiva quanto poderia ter sido, foram considerados poucos casos de uso. Uma pesquisa futura poderia se aprofundar ainda mais nessa análise, levando em consideração projectos de desenvolvimento de aplicativos mais reais com mais casos de uso.

Por outro lado, um ponto forte deste estudo é a consideração de factores financeiros na selecção de arquitecturas, e a utilização das novas tecnologias de desenvolvimento de aplicativos Android (Jetpack Compose), preenchendo uma lacuna importante na literatura existente.

6. CONCLUSÃO

Este estudo teve como objectivo principal realizar uma análise comparativa das arquitecturas de aplicativos Android, com o propósito de identificar as melhores práticas de desenvolvimento. Para atingir esse objectivo, definimos objectivos específicos que abrangeram desde a identificação das principais arquitecturas até a proposição de recomendações para seu uso adequado em diferentes tipos de aplicativos móveis.

Durante a condução desta pesquisa, identificou-se e analisou-se detalhadamente as principais arquitecturas de aplicativos Android, nomeadamente MVVM e MVI. Explorou-se métodos de avaliação que incluíram a análise de custo-benefício (CBAM) e o método de análise de compensação arquitectural (ATAM), permitindo uma compreensão abrangente dos aspectos técnicos e económicos dessas arquitecturas. Realizou-se experimentos que envolveram a colecta automatizada de métricas técnicas, fornecendo dados valiosos sobre o desempenho e a eficiência de cada arquitectura.

Os resultados desta pesquisa enfatizam a importância de uma abordagem abrangente na selecção de arquitecturas de aplicativos Android. Não existe uma arquitectura universalmente ideal; cada uma possui suas forças e fraquezas. A escolha deve ser baseada nas necessidades e restrições específicas de cada projecto.

Além disso, destacou-se a relevância de considerar as implicações financeiras ao seleccionar uma arquitectura. A análise de custo-benefício pode ajudar a tomar decisões mais informadas e resultar em um ROI mais favorável ao longo do ciclo de vida do aplicativo.

Em resumo, este estudo oferece uma abordagem técnica e financeira para a selecção de arquitecturas de desenvolvimento de aplicativos Android, considerando tanto os aspectos técnicos quanto os financeiros.

Direções para Futuras Pesquisas

Para futuras investigações, sugere-se a exploração de tópicos como migração de projectos entre as arquitecturas MVVM e MVI e também migração de projectos feitos em XML para Jetpack Compose. Com a crescente adopção do Jetpack Compose, estudos proprondo modelos ou *frameworks* de migração de aplicações XML para Jetpack Compose podem ser de grande valor para a comunidade de desenvolvimento Android.

Além disso, a análise de custos e benefícios pode ser estendida para considerar não apenas os custos de desenvolvimento, mas também os custos de manutenção e operação ao longo do ciclo de vida do aplicativo. Isso proporciona uma visão ainda mais completa das implicações financeiras das arquitecturas.

7. REFERÊNCIAS

Anderson, M., & Cohn, S. (2019). Mobile Technology and Home Broadband 2019. Acedido em 01/07/2023, em: https://www.pewresearch.org/internet/2019/06/13/mobile-technology-and-home-broadband-2019/.

Android Developers. (2021). *Jetpack Compose: Introdução*. Acedido em 21/08/2023, em: https://developer.android.com/jetpack/compose.

Android Developers. (2023a). *APP QUALITY*. Acedido em 21/08/2023, em: https://developer.android.com/topic/performance/benchmarking/benchmarking-overview.

Android Developers. (2023b). *Slow rendering*. Acedido em 21/08/2023, em: https://developer.android.com/topic/performance/vitals/render.

Bass, L., Clements, P., & Kazman, R. (2013). *Software architecture in practice*. Software Architecture in Practice, Third Edition. Boston: Addison-Wesley.

Castells, M. (2011). *A sociedade em rede*. Acedido em 19/06/2023, em: https://globalizacaoeintegracaoregionalufabc.files.wordpress.com/2014/10/castells-m-a-sociedade-em-rede.pdf.

Chauhan, K., Kumar, S., Sethia, D., & Alam, M. N. (2021). Performance analysis of kotlin coroutines on android in a model-view-intent architecture pattern. *2nd International Conference for Emerging Technology (INCET), May 21-23, 2021, Belgaum, India.*

Coursera. (2023. What Does a Software Engineer Do? Acedido em 07/11/2023, em: https://www.coursera.org/articles/software-engineer.

Digital Ocean. (2022, August 3). *Android MVVM Desenho Pattern*. Acedido em 22/08/2023, em: https://www.digitalocean.com/community/tutorials/android-mvvm-desenho-pattern

Economic Times. (2023). What is 'Software Engineering'. Acedido em 07/11/2023, Em: https://economictimes.indiatimes.com/definition/software-engineering

eMarketer. (2021). US Time Spent with Mobile. Acedido em 19/06/2023, em: https://www.emarketer.com/Content/Us-Time-Spent-with-Mobile-2021.

Fontoura, A. (2023, March 27). *Framework*. Acedido em 01/07/2023, em: https://fm2s.com.br/blog/framework-funcao-e-importancia.

Humeniuk, V. (2018). *Android Architecture Comparison: MVP vs. VIPER*. Linnaeus University, Faculty of Technology, Department of computer science and media technology (CM). Acedido em 02/06/2023, em:https://dokumen.tips/documents/android-architecture-comparison-mvp-vs-1291671fulltext01pdf-smartphone-was.html?page=4.

IBM, (2023). Mobile Tecnology. Acedido em 05/10/2023, em: https://www.ibm.com/topics/mobile-technology

Ionita, M., Hammer, D., & ObbinkHenk. (2002). Scenario-Based Software Architecture Evaluation Methods: An Overview. Eindhoven: *Department of Mathematics and Computing Science, Technical University Eindhoven*.

Kaplan, A. M., & Haenlein, M. (2010). Users of the world, unite! The challenges and opportunities of Social Media. Indiana: *Business Horizons*.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., & Carriere, J. (1998). The Architecture Tradeoff Analysis Method. Pittsburgh: *Software Engineering Institute Carnegie Mellon University Pittsburgh*.

Laudon, K. C., & Laudon, J. P. (2016). Management Information Systems: Managing the Digital Firm (14th Edition). Pearson.

Lo, P. (2010). *Software Architecture Analysis Method (SAAM)*. Acedido em 09/07/2023, em: http://www.peter-lo.com/Teaching/U08182/L07A.pdf

Lou, T. (2016). *A comparison of Android Native App Architecture MVC, MVP and MVVM*. Eindhoven: Eindhoven University of Technology.

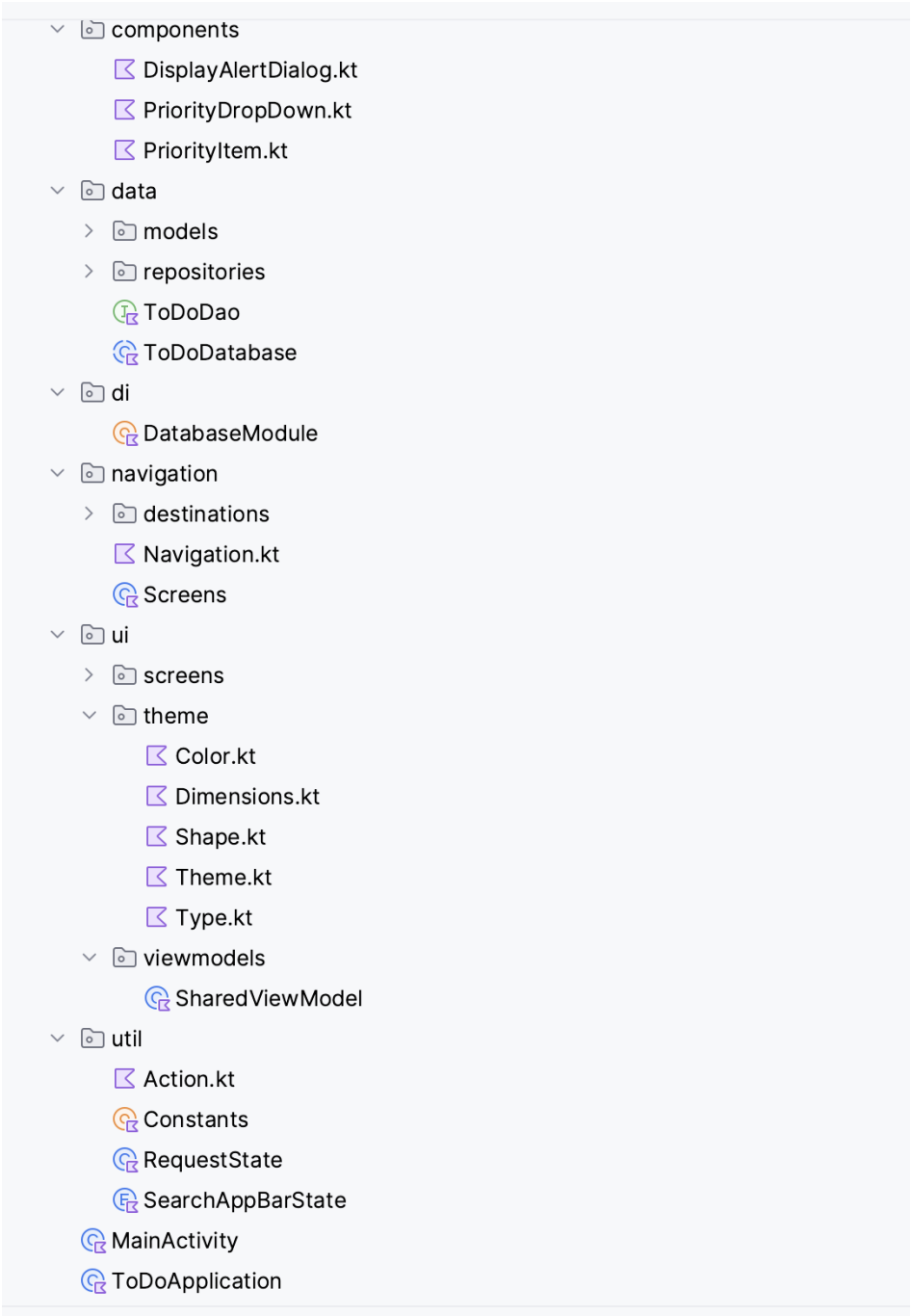
Ma, L., Gu, L., & Wang, J. (2014). Research and development of mobile application for android platform. Nanquim: *International Journal of Multimedia and Ubiquitous Engineering*.
Maharjan, B. (2018). *Puzzle game using Android MVVM Architecture*. Acedido em 21/07/2023, em: https://www.theseus.fi/handle/10024/145871
Marconi, M. de A., & Lakatos, E. M. (2003). *Fundamentos de Metodologia Científica*. 5ª edição, São Paulo: Editora Atlas.
Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Desenho*. New Jersey: Prentice Hall.
O'Brien, J. (2004), Sistemas de informação e as decisões gerenciais na era da Internet, 2. ed. Saraiva, São Paulo.
PFLB. (2023). *Android apps performance and load testing: tips and tricks*. Acedido em 26/07/2023, em: https://pflb.us/blog/android-apps-performance-and-load-testing/.
Pressman, R. S. (2010). *Ingeniería del software: un enfoque práctico*. 5ª edição, Nova Iorque: McGraw-Hill.
Samuel, A., Barbosa, R. (2022). *Análise Comparativa entre os padrões MVC, MVP, MVVM E MVI na plataforma Android*. Goiano: Instituto Federal De Educação, Ciência E Tecnologia.
Sharma, S., & Kaur, P. (2014). *Recent advances in engineering and computational sciences (RAECS). Chandigarh: University Institute of Engineering and Technology*.
Silveira, T. E. G.; D. T. (2009). *MÉTODOS DE PESQUISA. Rio Grande do Sul:* UFRGS Editora.
Speckmann, B. (2008). *The Android mobile platform*. Michigan: Eastern Michigan University.
Stair, R. Reynolds, G. (2006), Princípios de sistemas de informação: uma abordagem gerencial.
Pioneira Thomson Learning, São Paulo.
Statista. (2023a). Number of smartphone users worldwide from 2013 to 2028. Acedido em 19/06/2023, em: https://www.statista.com/forecasts/1143723/smartphone-users-in-the-world
Statista. (2023b). Global Market Share held by Mobile Operating Systems from 2009 to 2023, by Quarter. Acedido em 05/10/2023, em: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

8. APÊNDICES

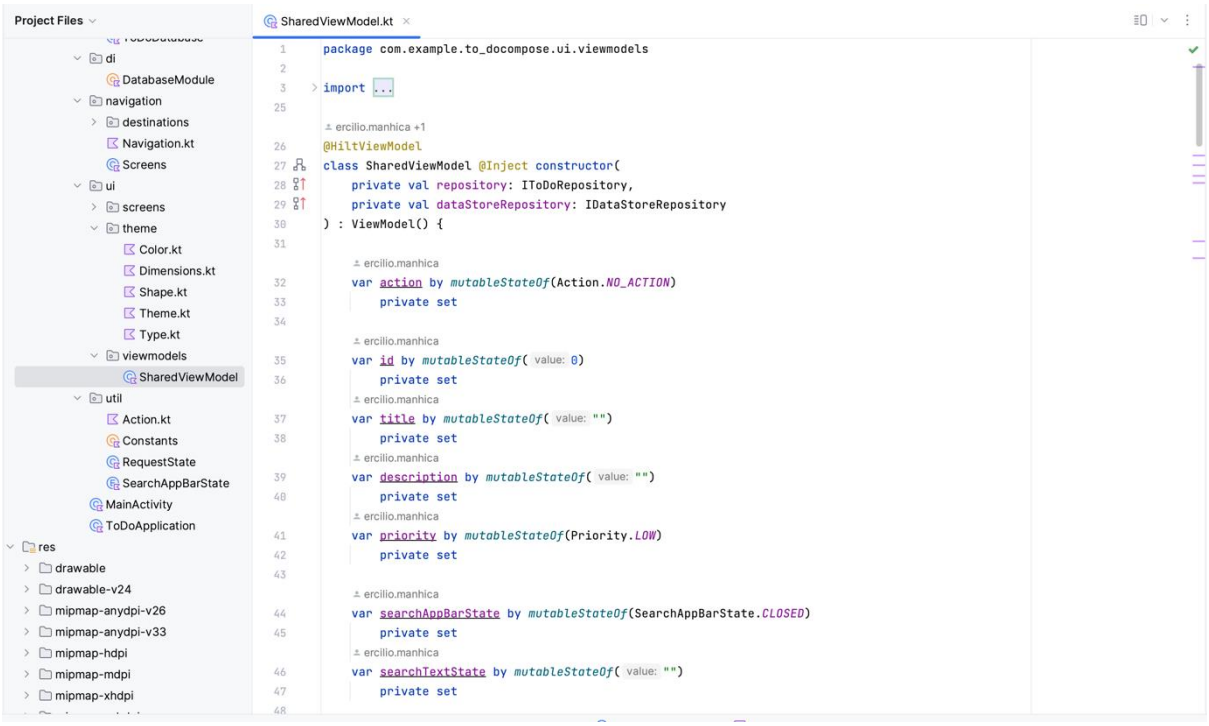
a) A: Códigos MVVM

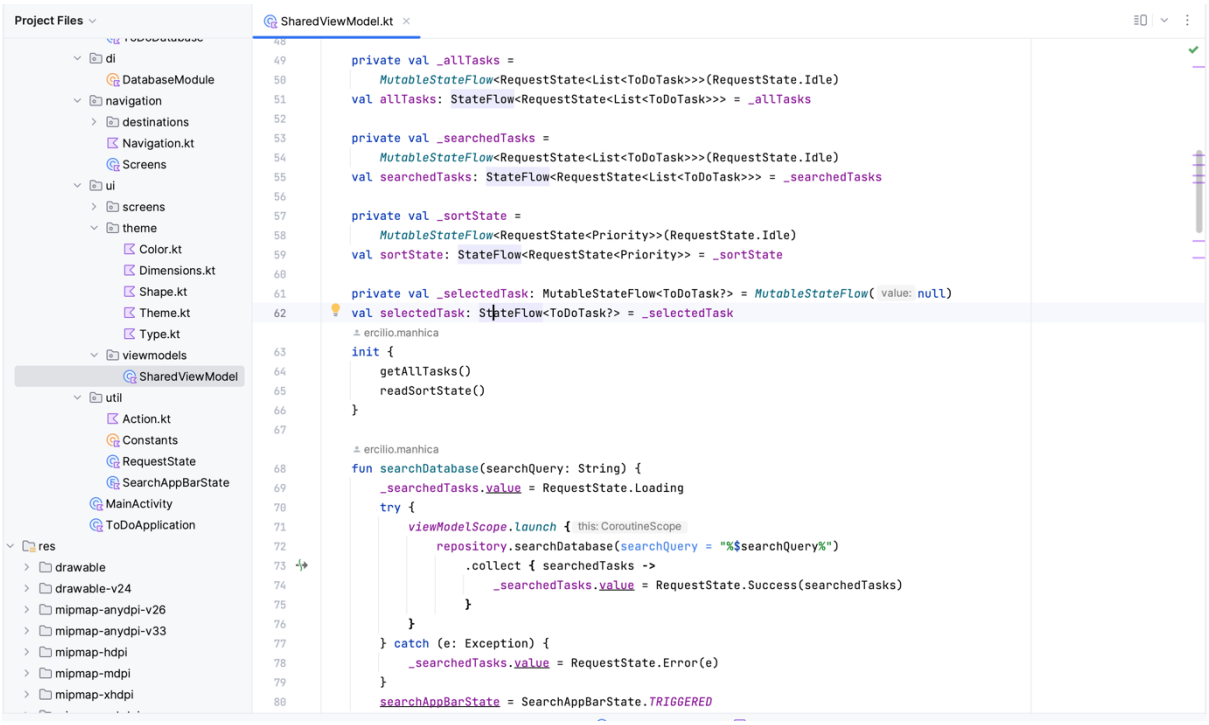
Este apêndice contém capturas de tela de códigos e trechos de código relacionados à estrutura de arquitectura MVVM (Model-View-ViewModel) usada neste estudo. Todo o código usado neste projecto poderá ser encontrado no GitHub https://github.com/ErcilioMarques/ToDoApp.

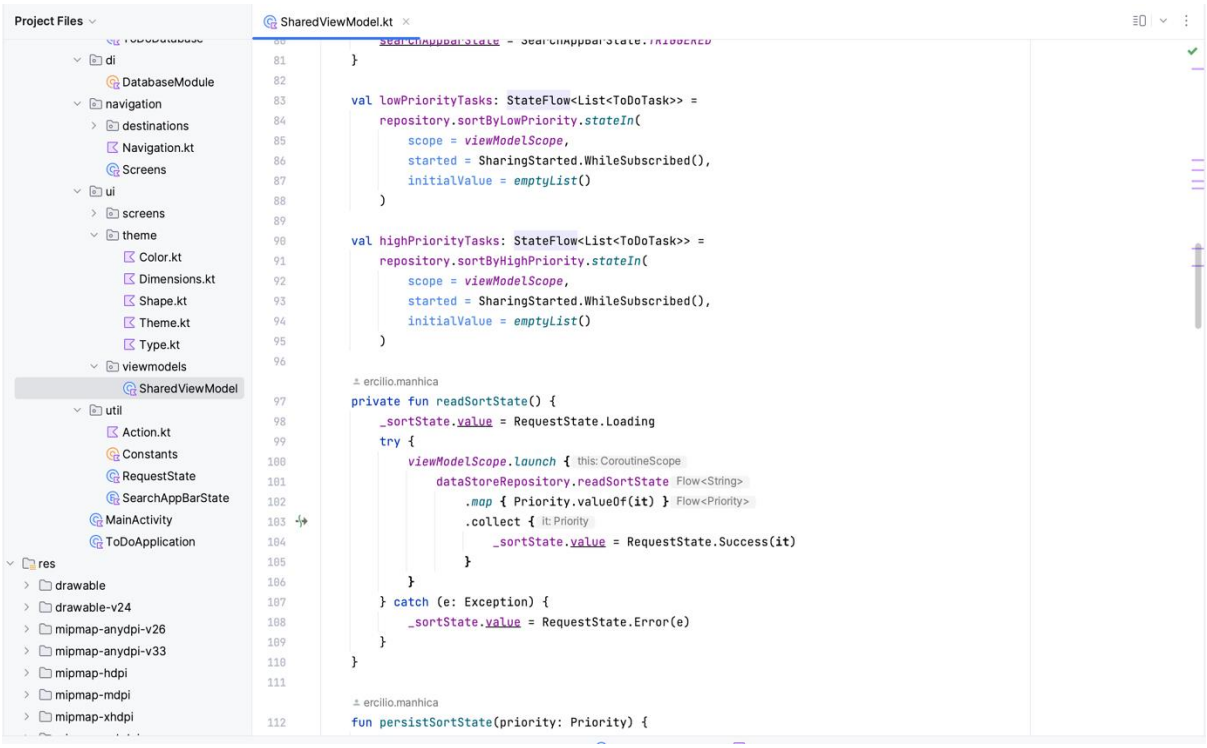
1. **Estrutura de Código-Fonte MVVM**:



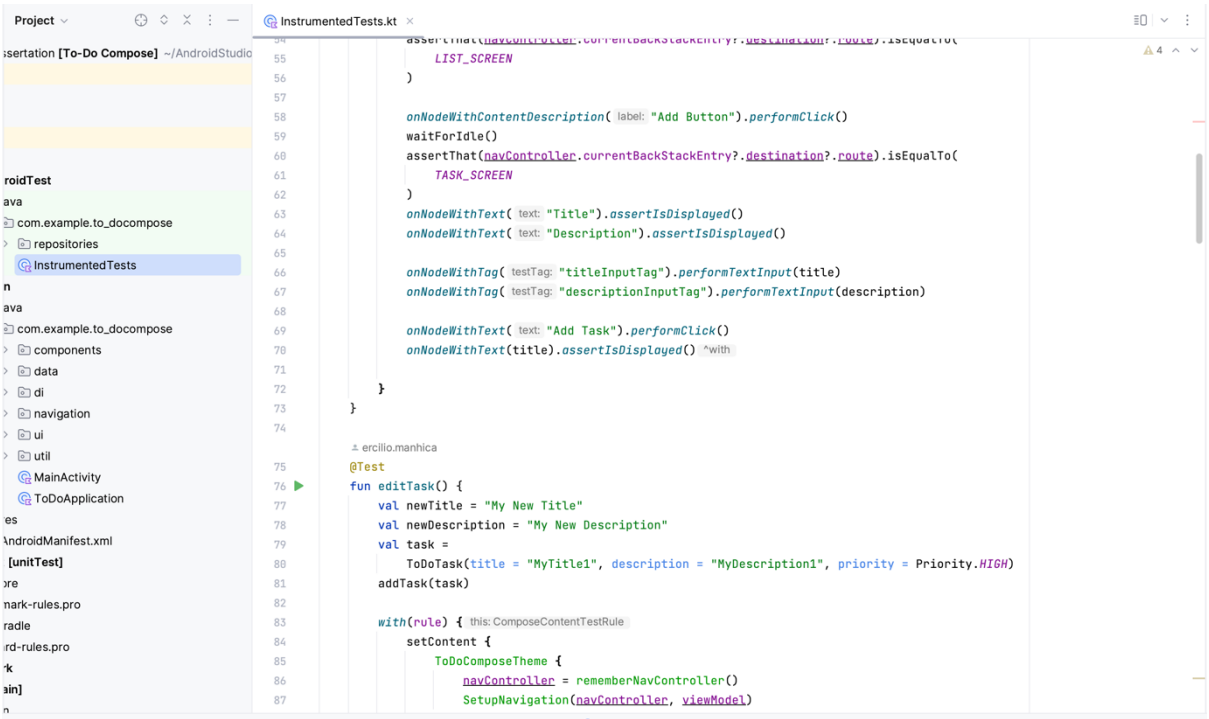
2. **Código-Fonte do ViewModel MVVM**:

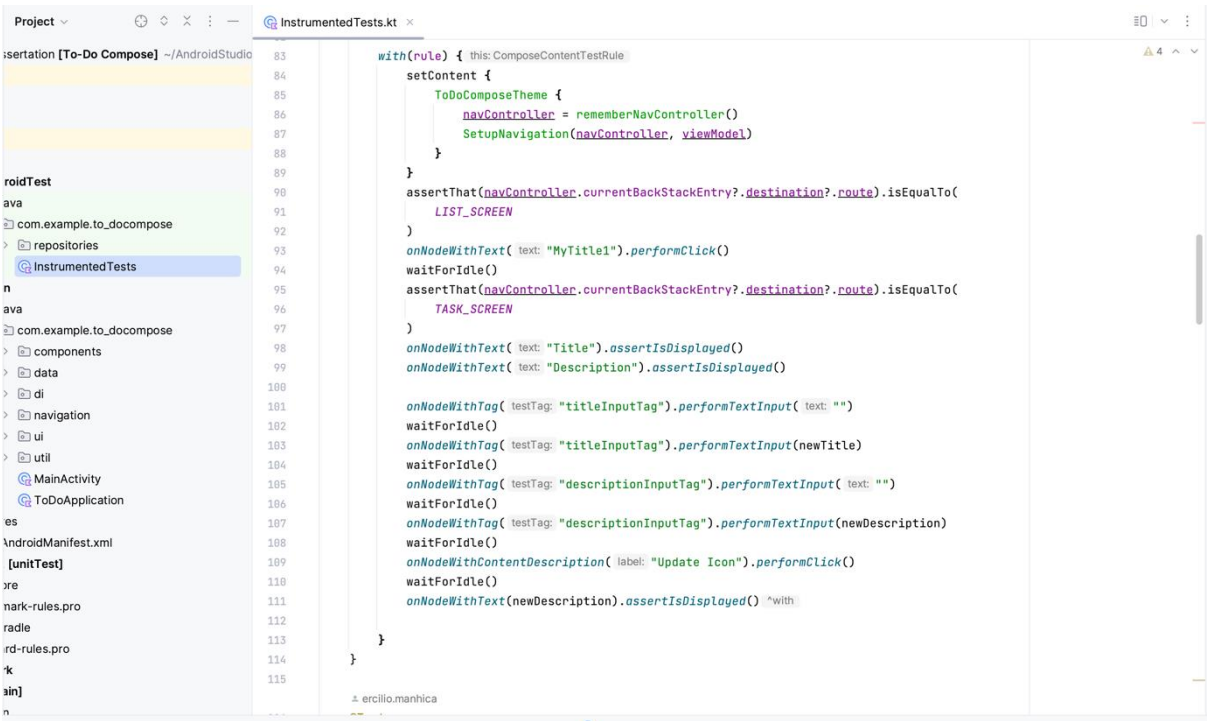


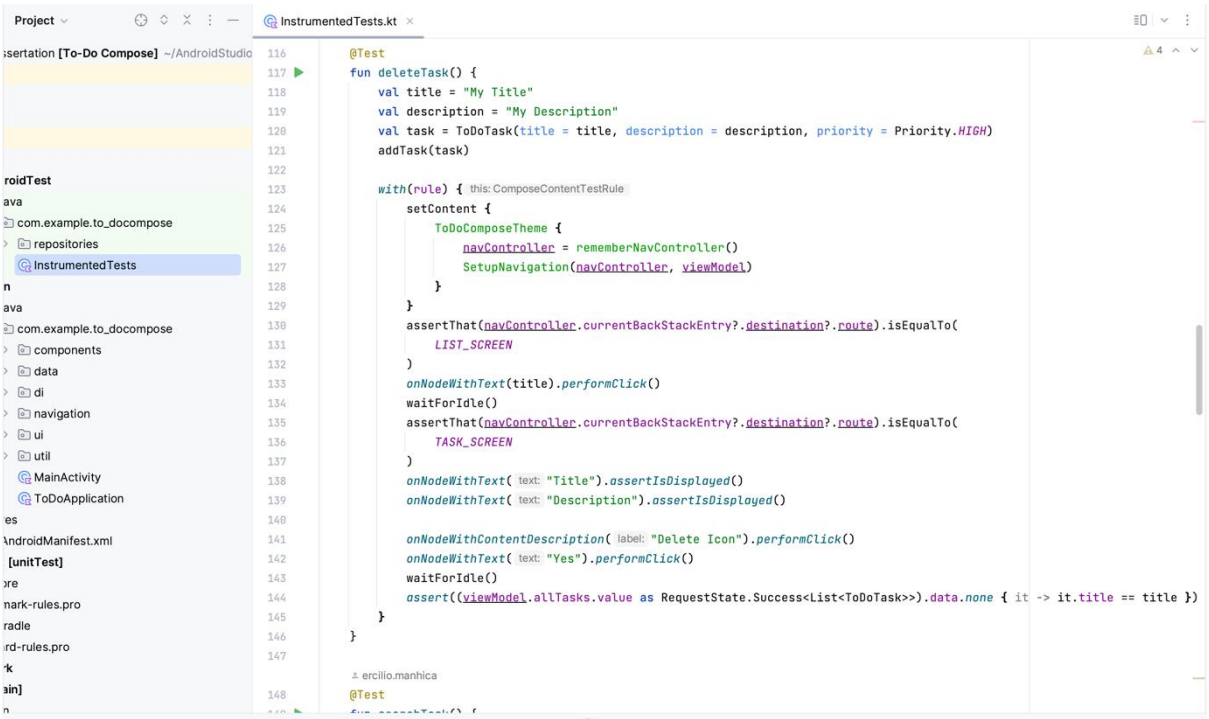


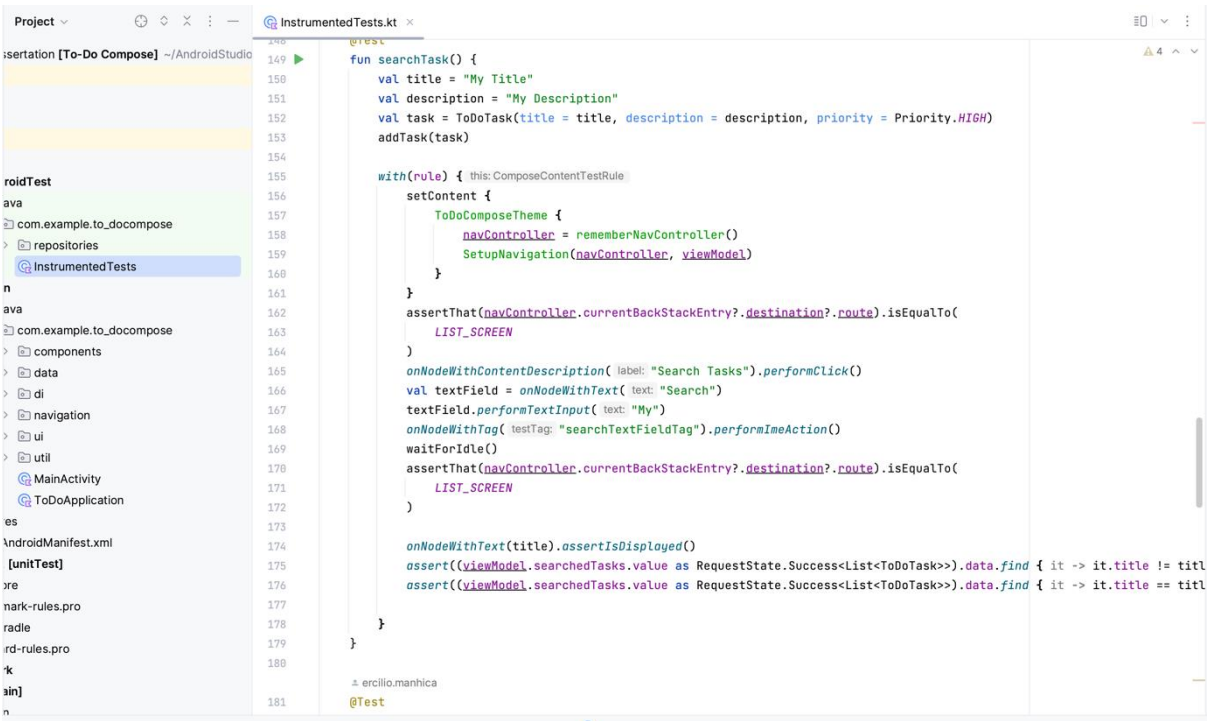


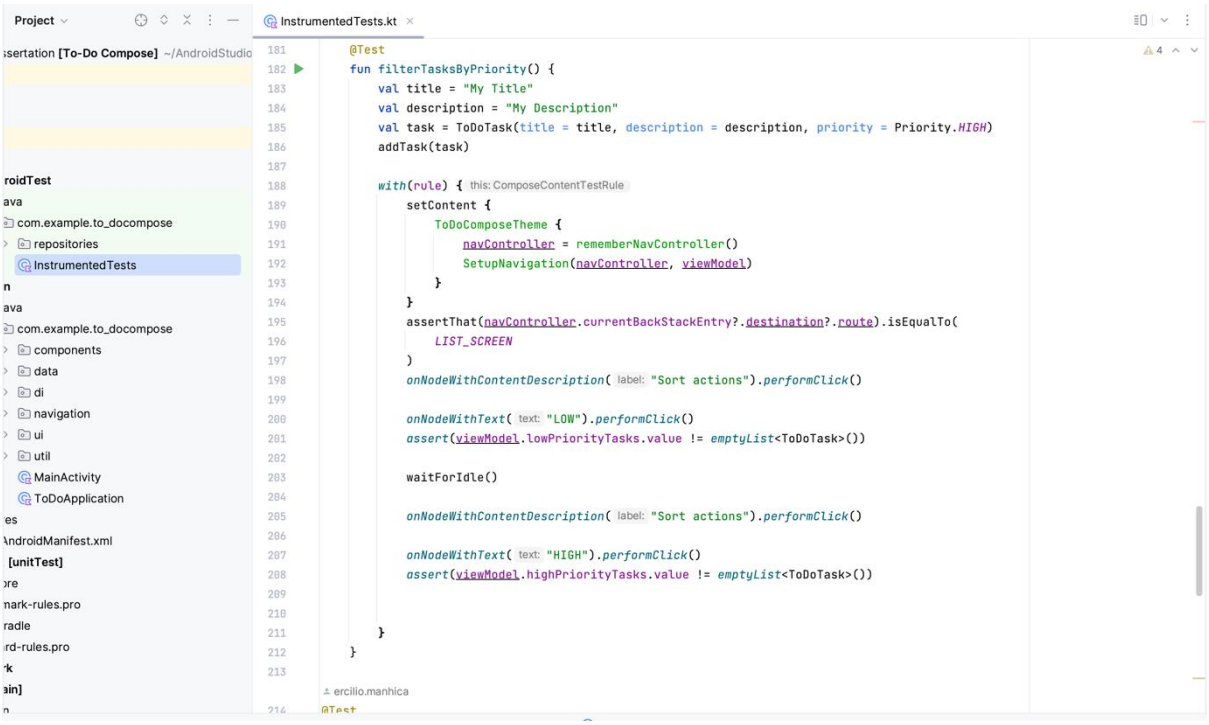
3. Testes Instrumentais MVVM

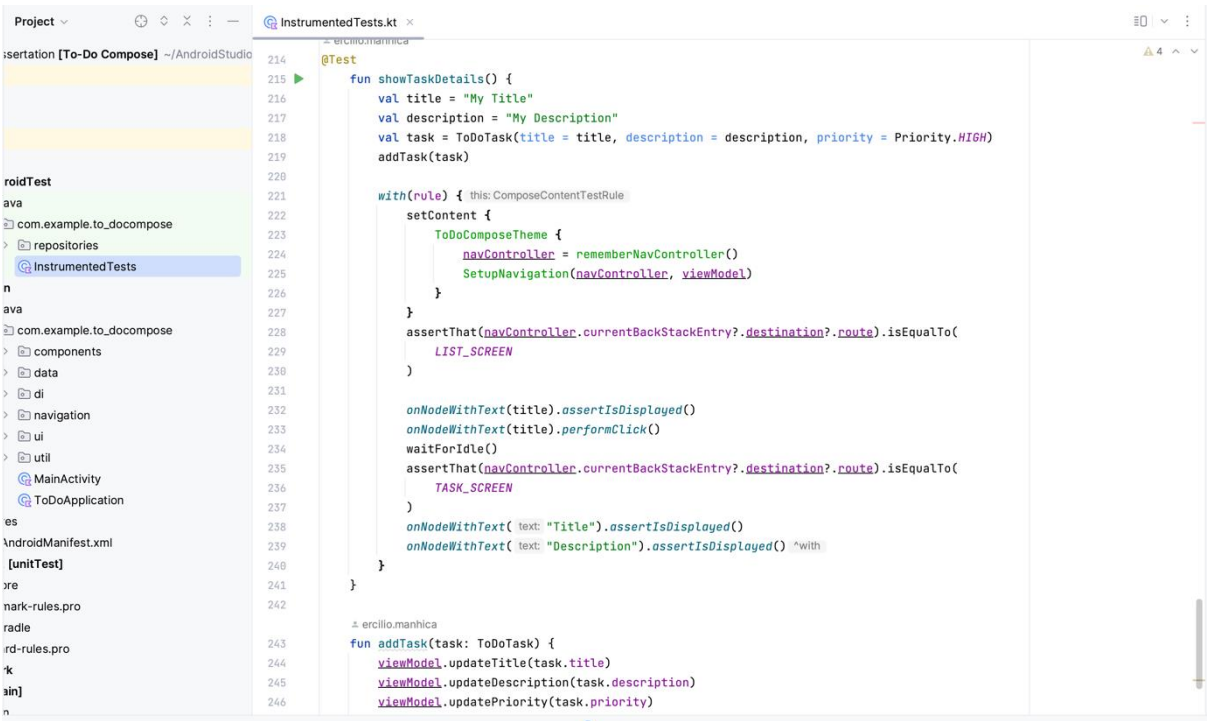

Project
InstrumentedTests.kt
sertation [To-Do Compose] ~/AndroidStudio
roidTest
ava
com.example.to_docompose
repositories
InstrumentedTests
n
ava
com.example.to_docompose
components
data
di
navigation
ui
util
MainActivity
ToDoApplication
es
AndroidManifest.xml
[unitTest]
ore
mark-rules.pro
radle
rd-rules.pro
k
ain]
LIST_SCREEN
)
onNodeWithContentDescription(label: "Add Button").performClick()
waitForIdle()
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
TASK_SCREEN
)
onNodeWithText(text: "Title").assertIsDisplayed()
onNodeWithText(text: "Description").assertIsDisplayed()
onNodeWithTag(testTag: "titleInputTag").performTextInput(title)
onNodeWithTag(testTag: "descriptionInputTag").performTextInput(description)
onNodeWithText(text: "Add Task").performClick()
onNodeWithText(title).assertIsDisplayed() ^with
}
}
ercilio.manhica
@Test
fun editTask() {
val newTitle = "My New Title"
val newDescription = "My New Description"
val task =
ToDoTask(title = "MyTitle1", description = "MyDescription1", priority = Priority.HIGH)
addTask(task)
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)


Project
InstrumentedTests.kt
sertation [To-Do Compose] ~/AndroidStudio
roidTest
ava
com.example.to_docompose
repositories
InstrumentedTests
n
ava
com.example.to_docompose
components
data
di
navigation
ui
util
MainActivity
ToDoApplication
es
AndroidManifest.xml
[unitTest]
ore
mark-rules.pro
radle
rd-rules.pro
k
ain]
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)
}
}
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithText(text: "MyTitle1").performClick()
waitForIdle()
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
TASK_SCREEN
)
onNodeWithText(text: "Title").assertIsDisplayed()
onNodeWithText(text: "Description").assertIsDisplayed()
onNodeWithTag(testTag: "titleInputTag").performTextInput(text: "")
waitForIdle()
onNodeWithTag(testTag: "titleInputTag").performTextInput(newTitle)
waitForIdle()
onNodeWithTag(testTag: "descriptionInputTag").performTextInput(text: "")
waitForIdle()
onNodeWithTag(testTag: "descriptionInputTag").performTextInput(newDescription)
waitForIdle()
onNodeWithContentDescription(label: "Update Icon").performClick()
waitForIdle()
onNodeWithText(newDescription).assertIsDisplayed() ^with
}
}
ercilio.manhica

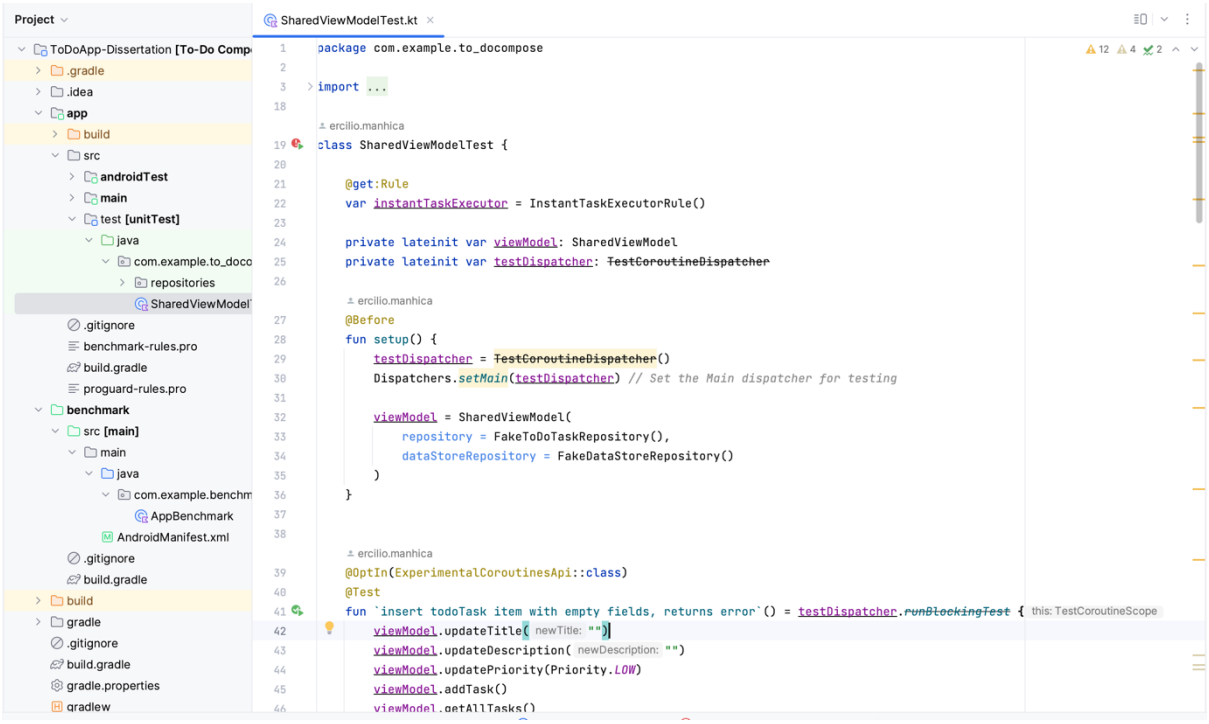

Project
InstrumentedTests.kt
sertation [To-Do Compose] ~/AndroidStudio
roidTest
ava
com.example.to_docompose
repositories
InstrumentedTests
n
ava
com.example.to_docompose
components
data
di
navigation
ui
util
MainActivity
ToDoApplication
es
AndroidManifest.xml
[unitTest]
ore
mark-rules.pro
radle
rd-rules.pro
k
ain]
@Test
fun deleteTask() {
val title = "My Title"
val description = "My Description"
val task = ToDoTask(title = title, description = description, priority = Priority.HIGH)
addTask(task)
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)
}
}
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithText(title).performClick()
waitForIdle()
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
TASK_SCREEN
)
onNodeWithText(text: "Title").assertIsDisplayed()
onNodeWithText(text: "Description").assertIsDisplayed()
onNodeWithContentDescription(label: "Delete Icon").performClick()
onNodeWithText(text: "Yes").performClick()
waitForIdle()
assert((viewModel.allTasks.value as RequestState.Success<List<ToDoTask>>).data.none { it -> it.title == title })
}
}
ercilio.manhica
@Test

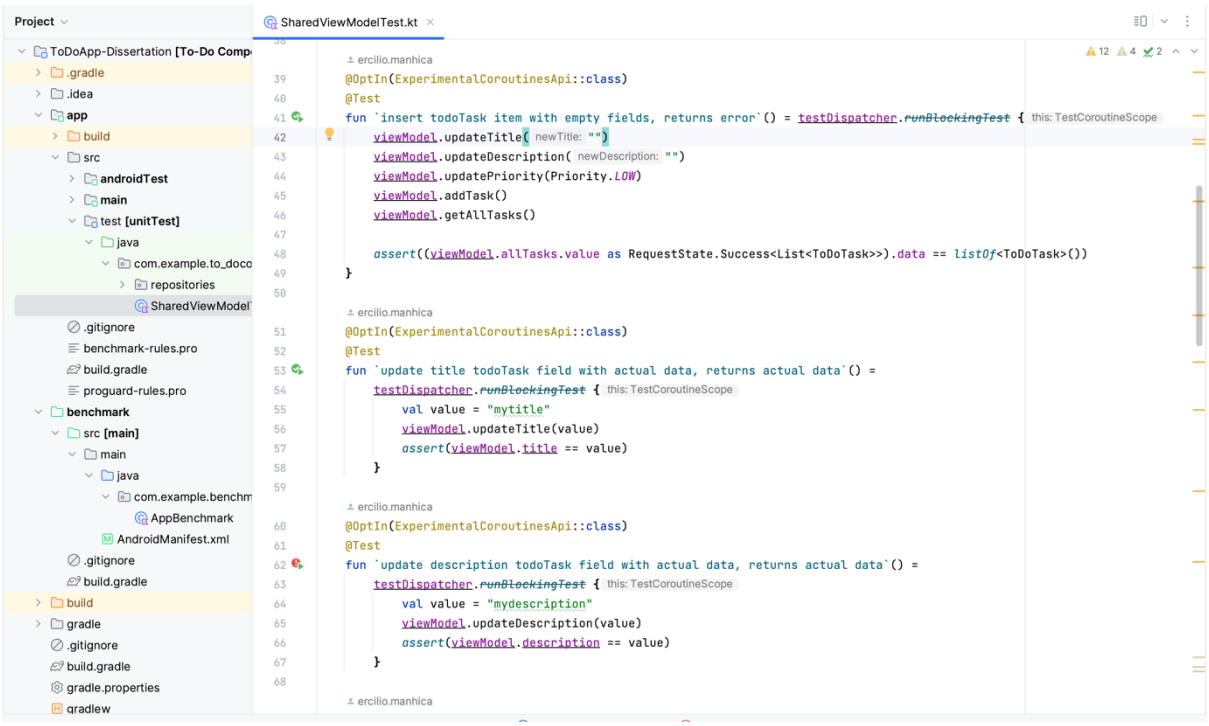

Project
InstrumentedTests.kt
sertation [To-Do Compose] ~/AndroidStudio
roidTest
ava
com.example.to_docompose
repositories
InstrumentedTests
n
ava
com.example.to_docompose
components
data
di
navigation
ui
util
MainActivity
ToDoApplication
es
AndroidManifest.xml
[unitTest]
ore
mark-rules.pro
radle
rd-rules.pro
k
ain]
fun searchTask() {
val title = "My Title"
val description = "My Description"
val task = ToDoTask(title = title, description = description, priority = Priority.HIGH)
addTask(task)
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)
}
}
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithContentDescription(label: "Search Tasks").performClick()
val textField = onNodeWithText(text: "Search")
textField.performTextInput(text: "My")
onNodeWithTag(testTag: "searchTextFieldTag").performImeAction()
waitForIdle()
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithText(title).assertIsDisplayed()
assert((viewModel.searchedTasks.value as RequestState.Success<List<ToDoTask>>).data.find { it -> it.title != titl
assert((viewModel.searchedTasks.value as RequestState.Success<List<ToDoTask>>).data.find { it -> it.title == titl
}
}
ercilio.manhica
@Test

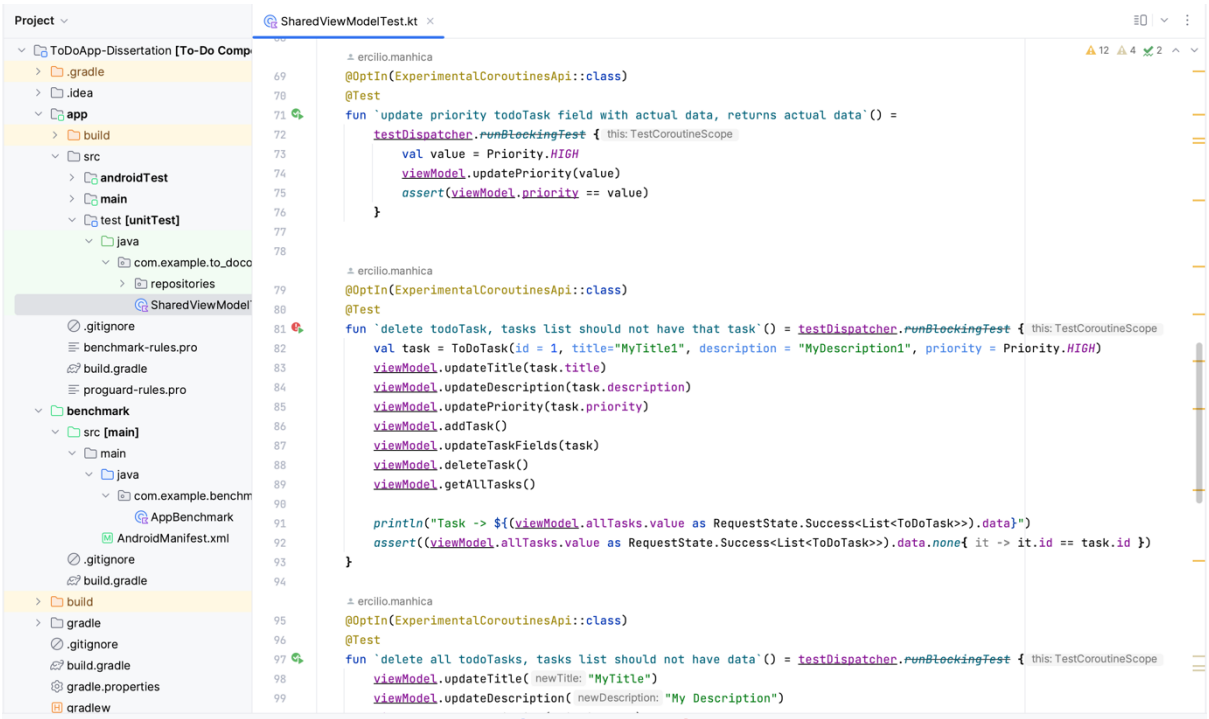

Project
InstrumentedTests.kt
sertation [To-Do Compose] ~/AndroidStudio
roidTest
ava
com.example.to_docompose
repositories
InstrumentedTests
n
ava
com.example.to_docompose
components
data
di
navigation
ui
util
MainActivity
ToDoApplication
es
AndroidManifest.xml
[unitTest]
ore
mark-rules.pro
radle
rd-rules.pro
k
ain]
@Test
fun filterTasksByPriority() {
val title = "My Title"
val description = "My Description"
val task = ToDoTask(title = title, description = description, priority = Priority.HIGH)
addTask(task)
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)
}
}
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithContentDescription(label: "Sort actions").performClick()
onNodeWithText(text: "LOW").performClick()
assert(viewModel.lowPriorityTasks.value != emptyList<ToDoTask>())
waitForIdle()
onNodeWithContentDescription(label: "Sort actions").performClick()
onNodeWithText(text: "HIGH").performClick()
assert(viewModel.highPriorityTasks.value != emptyList<ToDoTask>())
}
}
ercilio.manhica
@Test

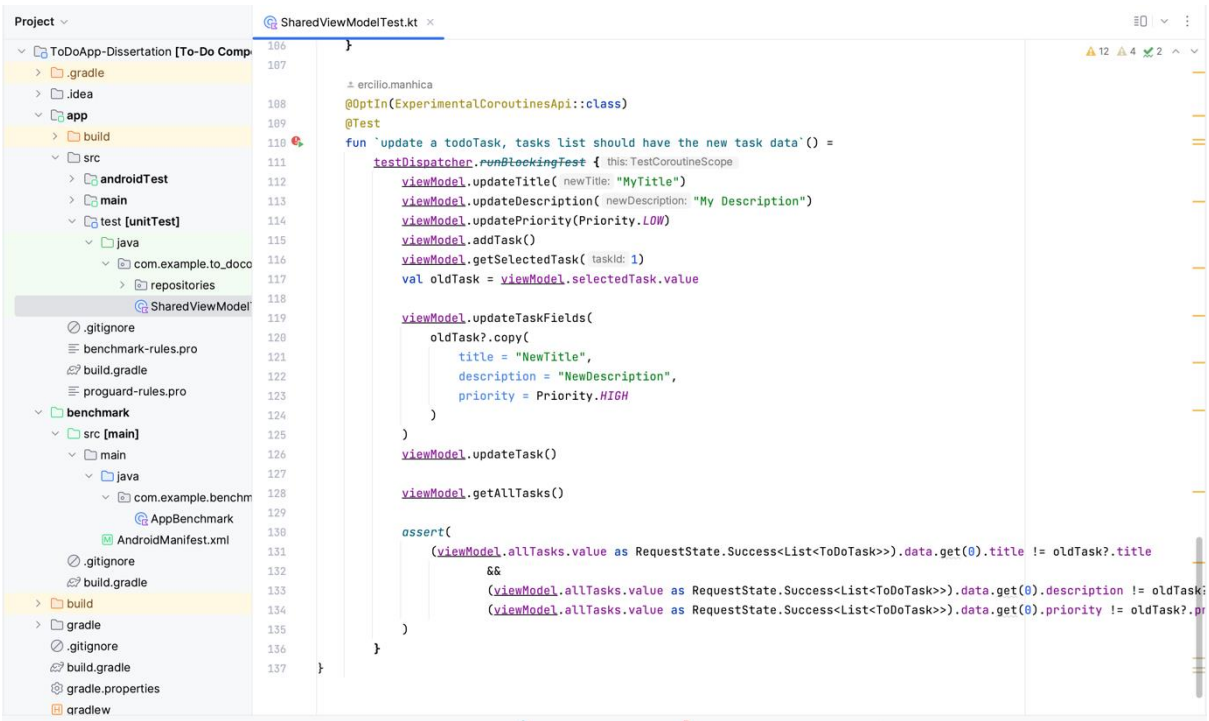

Project
InstrumentedTests.kt
sertation [To-Do Compose] ~/AndroidStudio
roidTest
ava
com.example.to_docompose
repositories
InstrumentedTests
n
ava
com.example.to_docompose
components
data
di
navigation
ui
util
MainActivity
ToDoApplication
es
AndroidManifest.xml
[unitTest]
ore
mark-rules.pro
radle
rd-rules.pro
k
ain]
@Test
fun showTaskDetails() {
val title = "My Title"
val description = "My Description"
val task = ToDoTask(title = title, description = description, priority = Priority.HIGH)
addTask(task)
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)
}
}
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithText(title).assertIsDisplayed()
onNodeWithText(title).performClick()
waitForIdle()
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
TASK_SCREEN
)
onNodeWithText(text: "Title").assertIsDisplayed()
onNodeWithText(text: "Description").assertIsDisplayed() ^with
}
}
ercilio.manhica
fun addTask(task: ToDoTask) {
viewModel.updateTitle(task.title)
viewModel.updateDescription(task.description)
viewModel.updatePriority(task.priority)

4. Testes Unitários para MVVM:





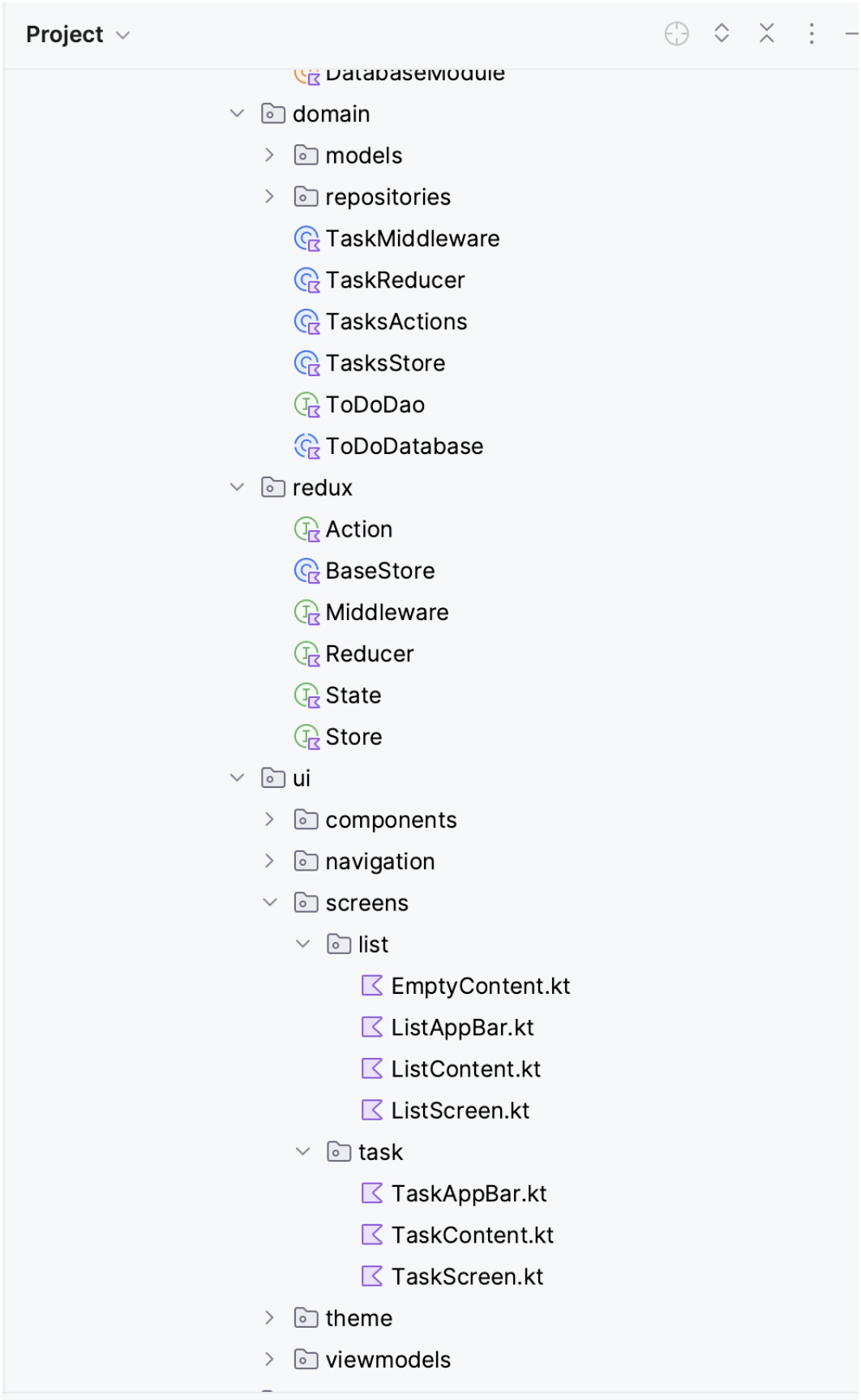




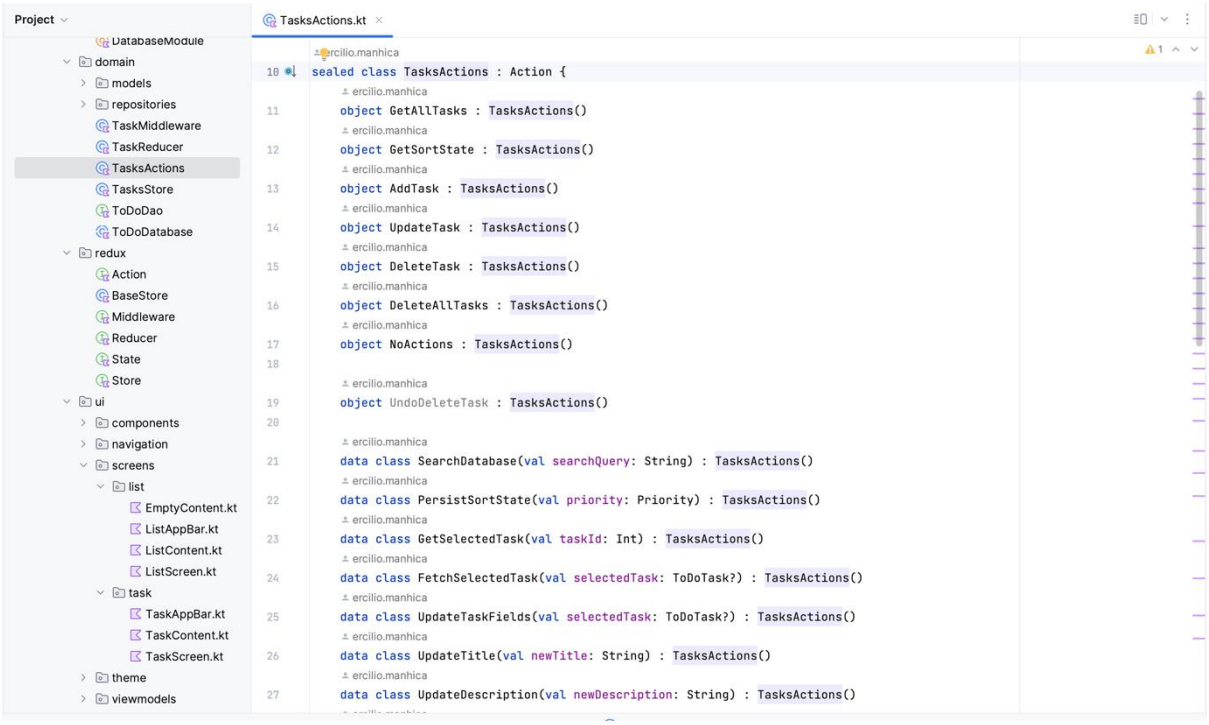
b) B: Códigos MVI

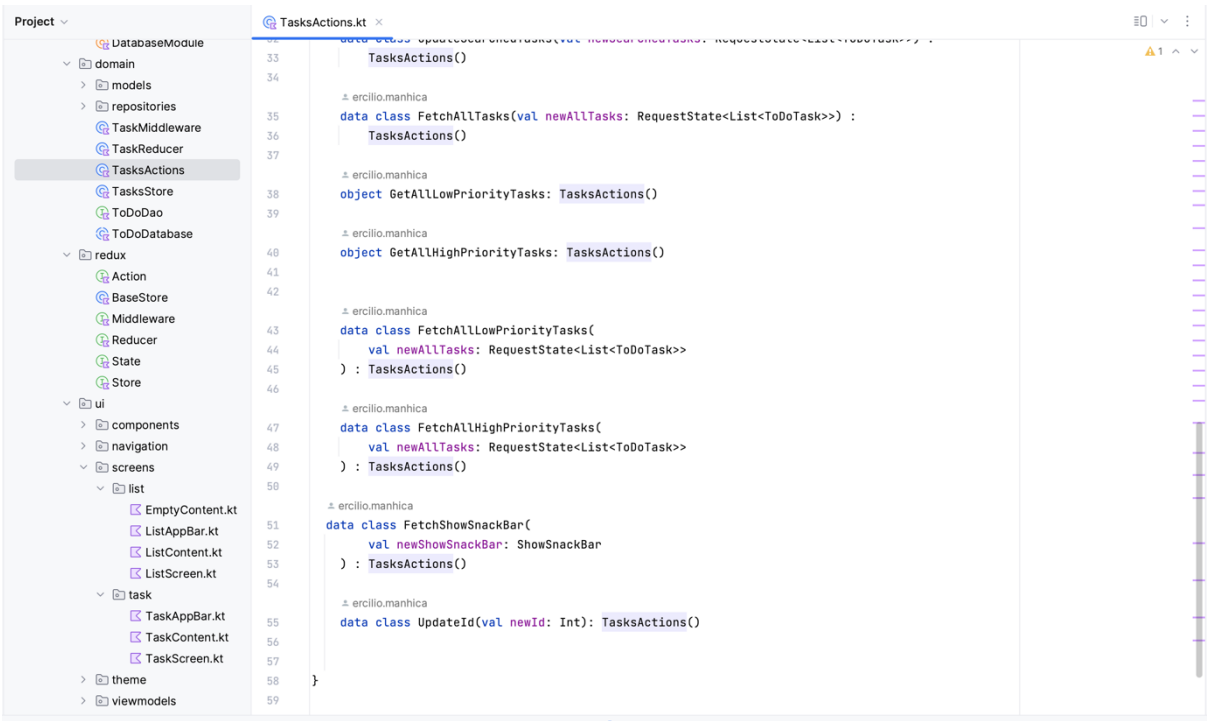
Este apêndice contém capturas de tela de códigos e trechos de código relacionados à estrutura de arquitectura MVI (Model-View-Intent) usada neste estudo.

1. **Estrutura Código-Fonte MVI**:

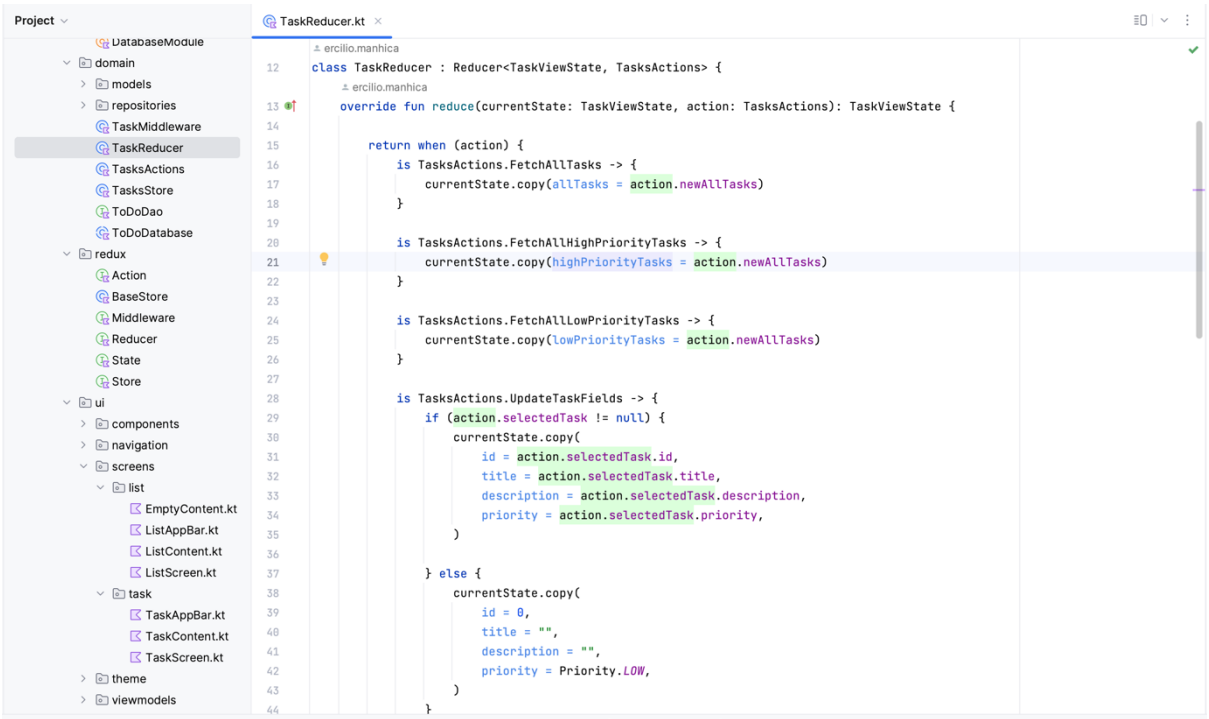


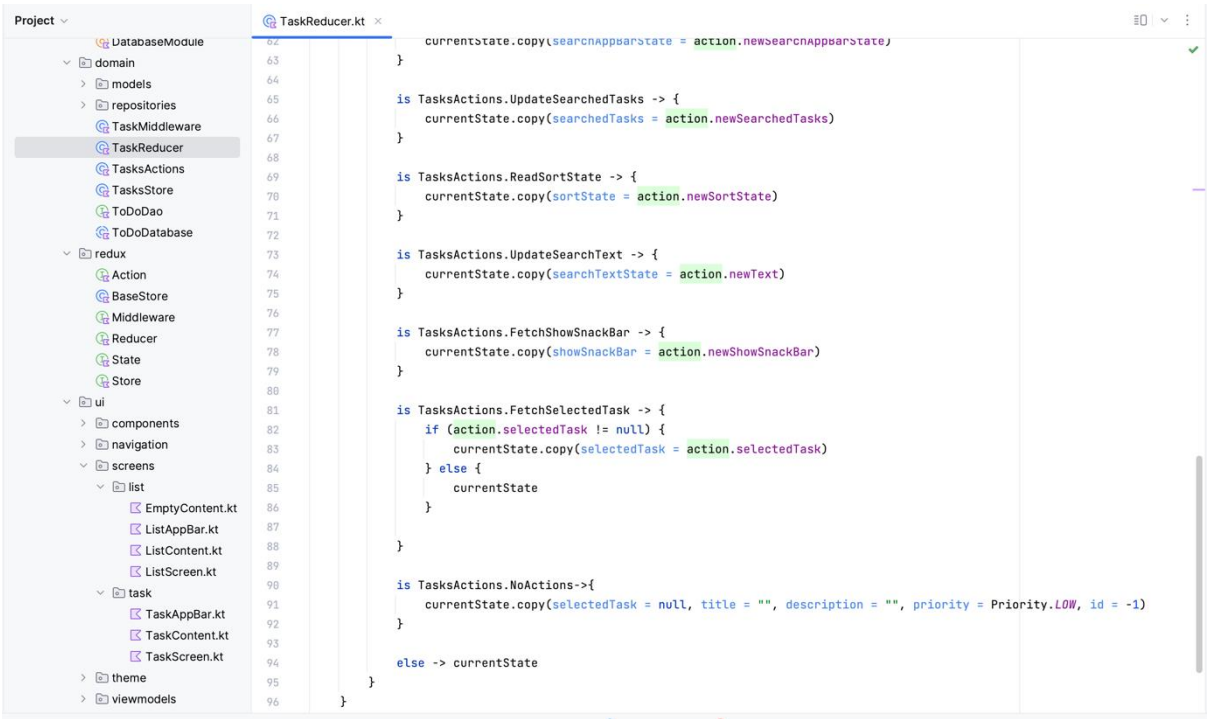
2. **Intent MVI**:


Project
TasksActions.kt
DatabaseModule
domain
models
repositories
TaskMiddleware
TaskReducer
TasksActions
TasksStore
ToDoDao
ToDoDatabase
redux
Action
BaseStore
Middleware
Reducer
State
Store
ui
components
navigation
screens
list
EmptyContent.kt
ListAppBar.kt
ListContent.kt
ListScreen.kt
task
TaskAppBar.kt
TaskContent.kt
TaskScreen.kt
theme
viewmodels
ercilio.manhica
sealed class TasksActions : Action {
object GetAllTasks : TasksActions()
object GetSortState : TasksActions()
object AddTask : TasksActions()
object UpdateTask : TasksActions()
object DeleteTask : TasksActions()
object DeleteAllTasks : TasksActions()
object NoActions : TasksActions()
object UndoDeleteTask : TasksActions()
data class SearchDatabase(val searchQuery: String) : TasksActions()
data class PersistSortState(val priority: Priority) : TasksActions()
data class GetSelectedTask(val taskId: Int) : TasksActions()
data class FetchSelectedTask(val selectedTask: ToDoTask?) : TasksActions()
data class UpdateTaskFields(val selectedTask: ToDoTask?) : TasksActions()
data class UpdateTitle(val newTitle: String) : TasksActions()
data class UpdateDescription(val newDescription: String) : TasksActions()

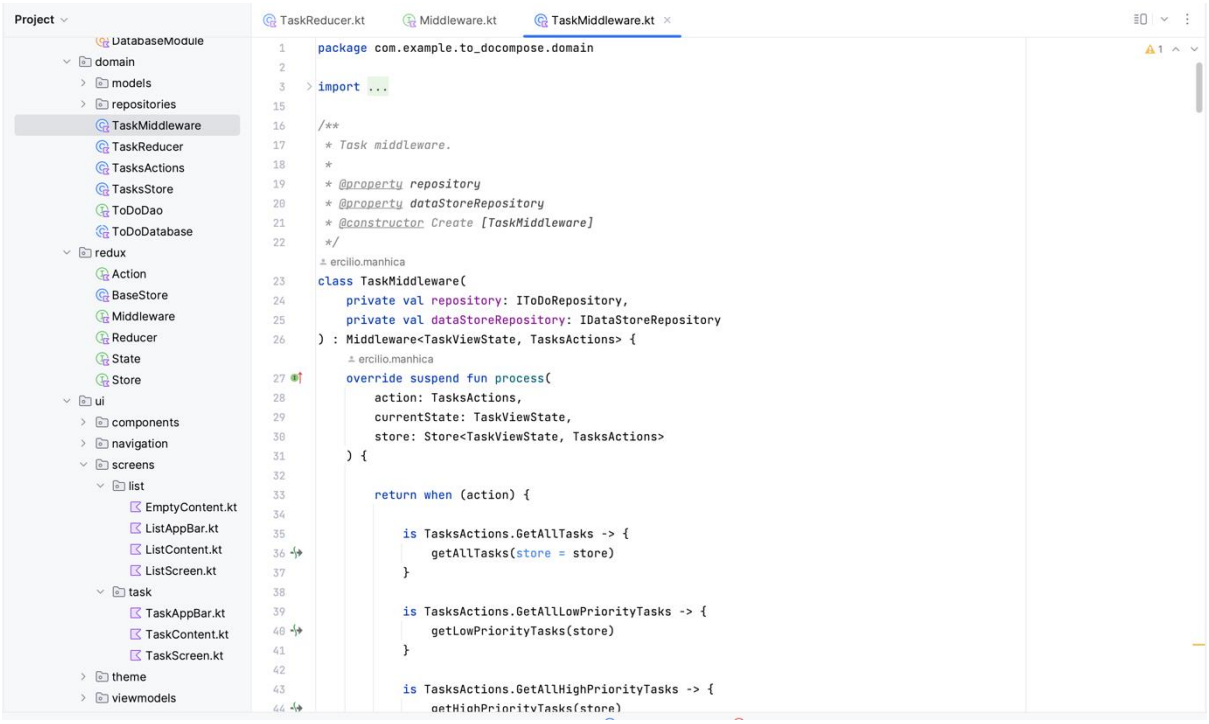

Project
TasksActions.kt
TasksActions()
data class FetchAllTasks(val newAllTasks: RequestState<List<ToDoTask>>) :
TasksActions()
object GetAllLowPriorityTasks: TasksActions()
object GetAllHighPriorityTasks: TasksActions()
data class FetchAllLowPriorityTasks(
val newAllTasks: RequestState<List<ToDoTask>>
) : TasksActions()
data class FetchAllHighPriorityTasks(
val newAllTasks: RequestState<List<ToDoTask>>
) : TasksActions()
data class FetchShowSnackBar(
val newShowSnackBar: ShowSnackBar
) : TasksActions()
data class UpdateId(val newId: Int): TasksActions()
}

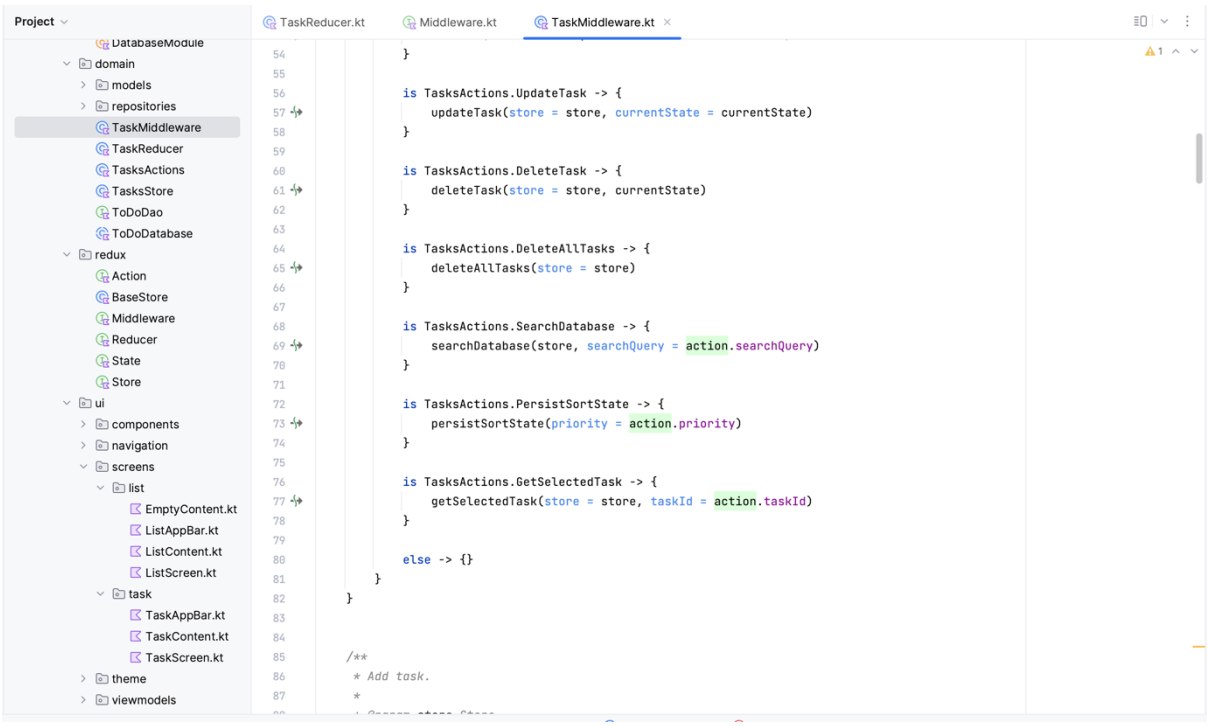
3. **Reducer MVI**:

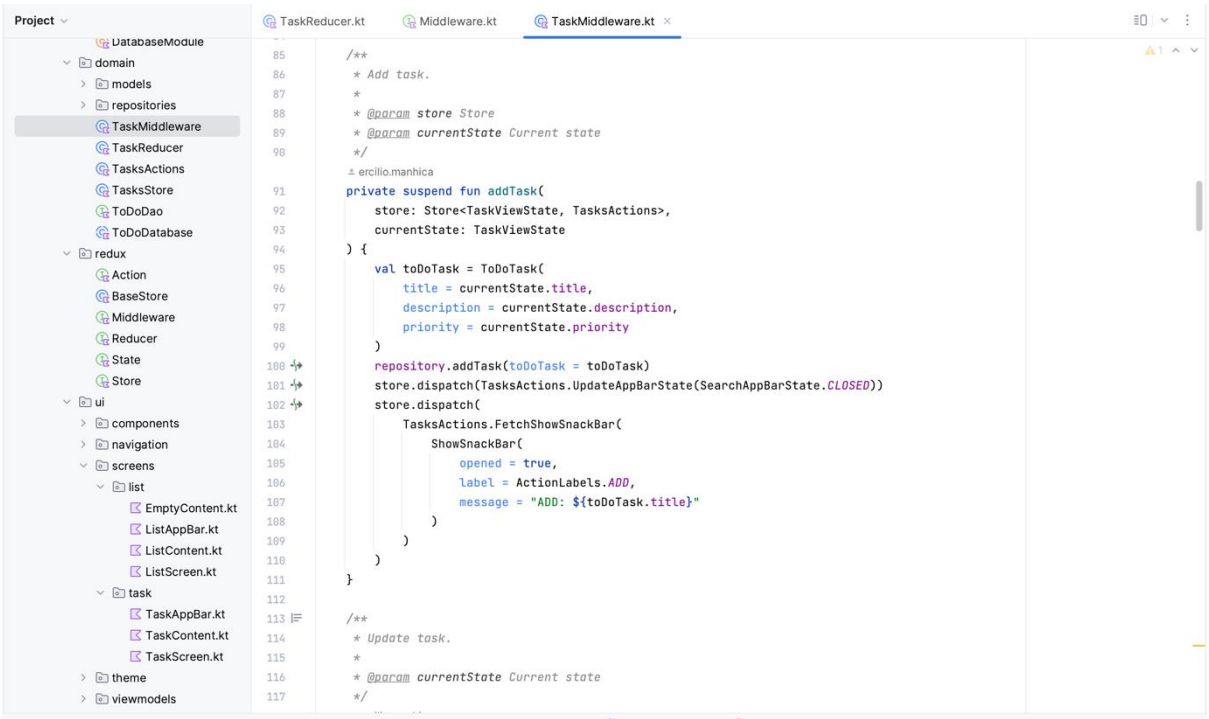


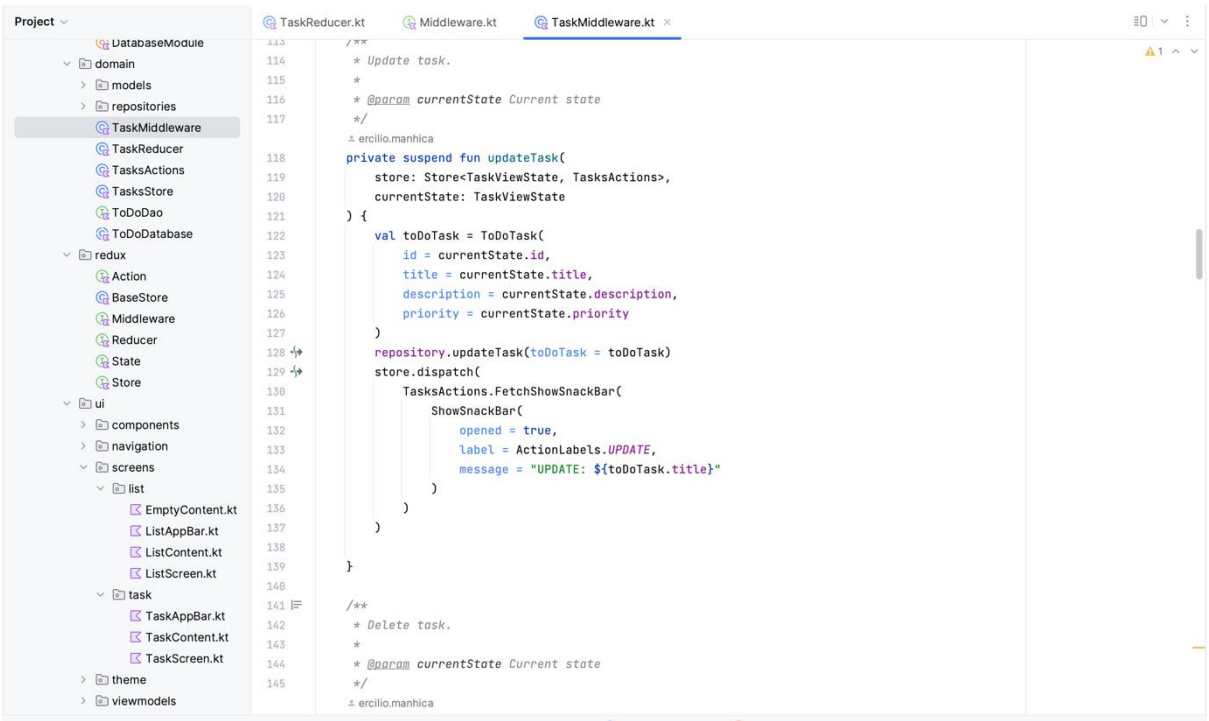


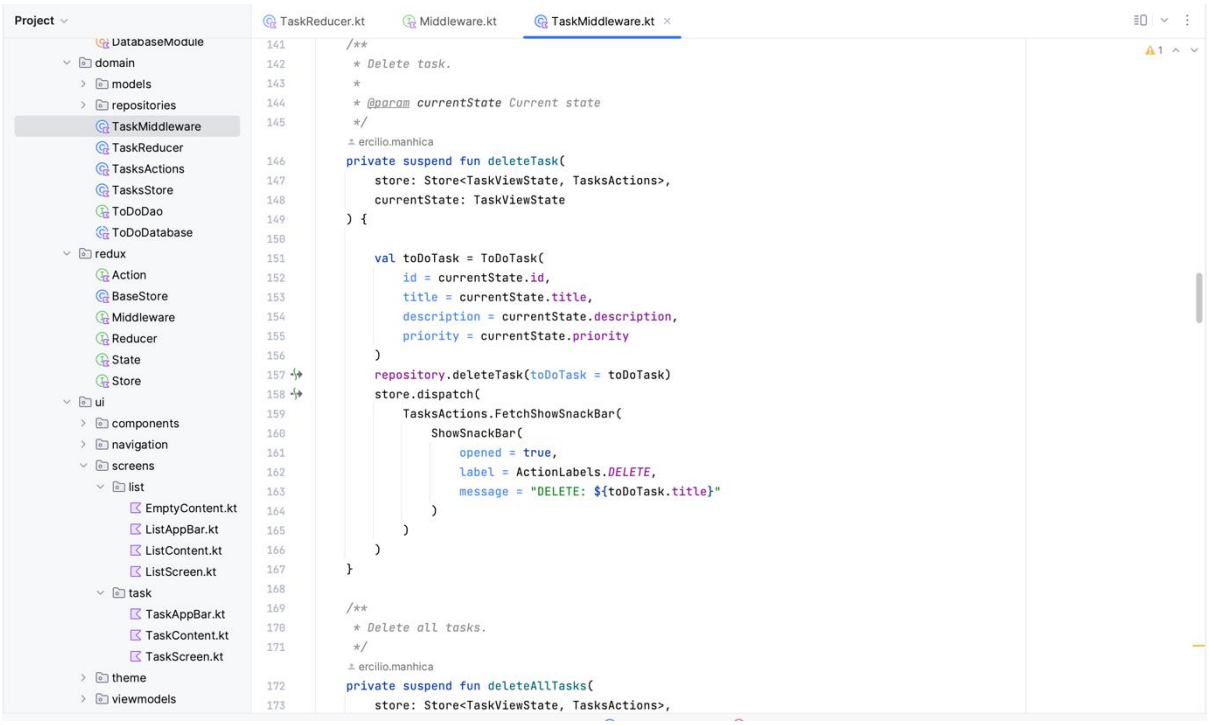
4. **Middleware MVI**:

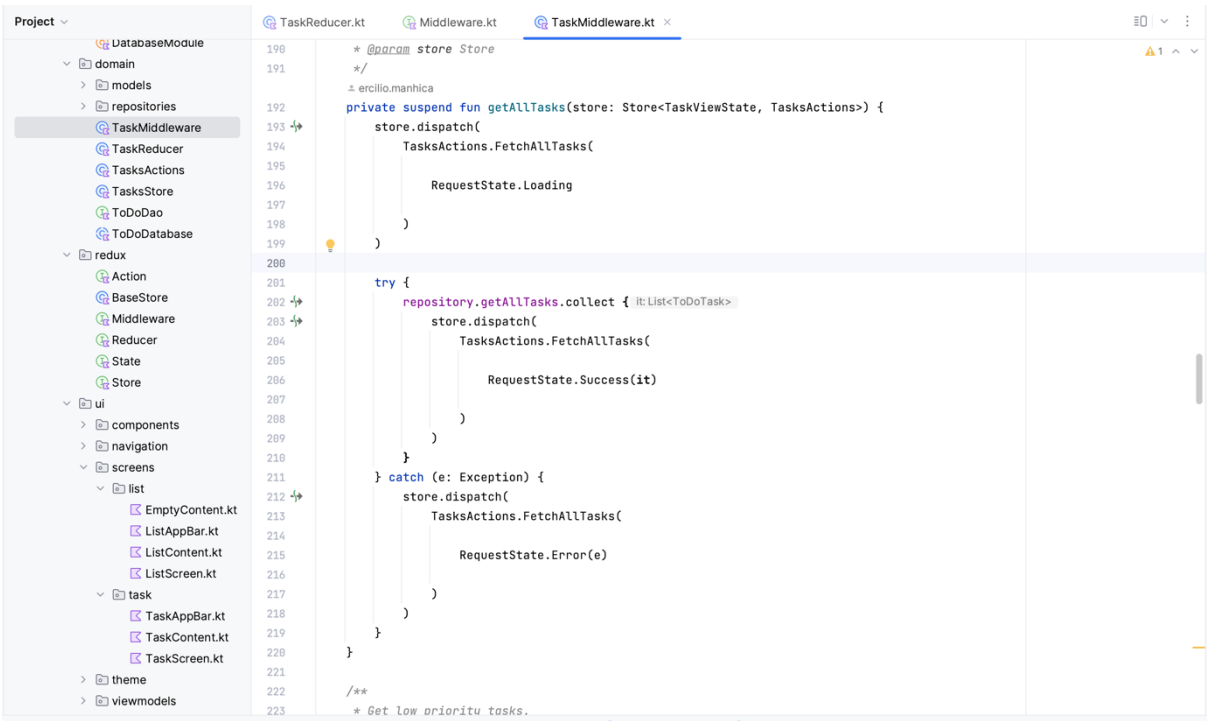


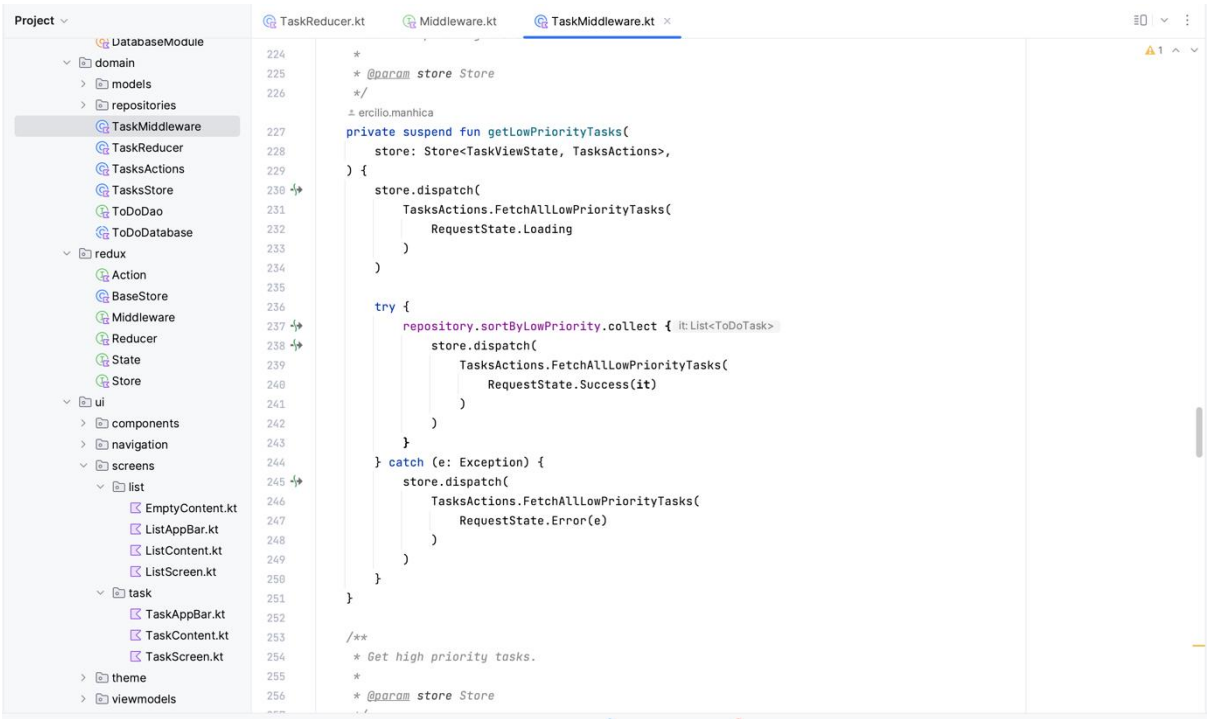


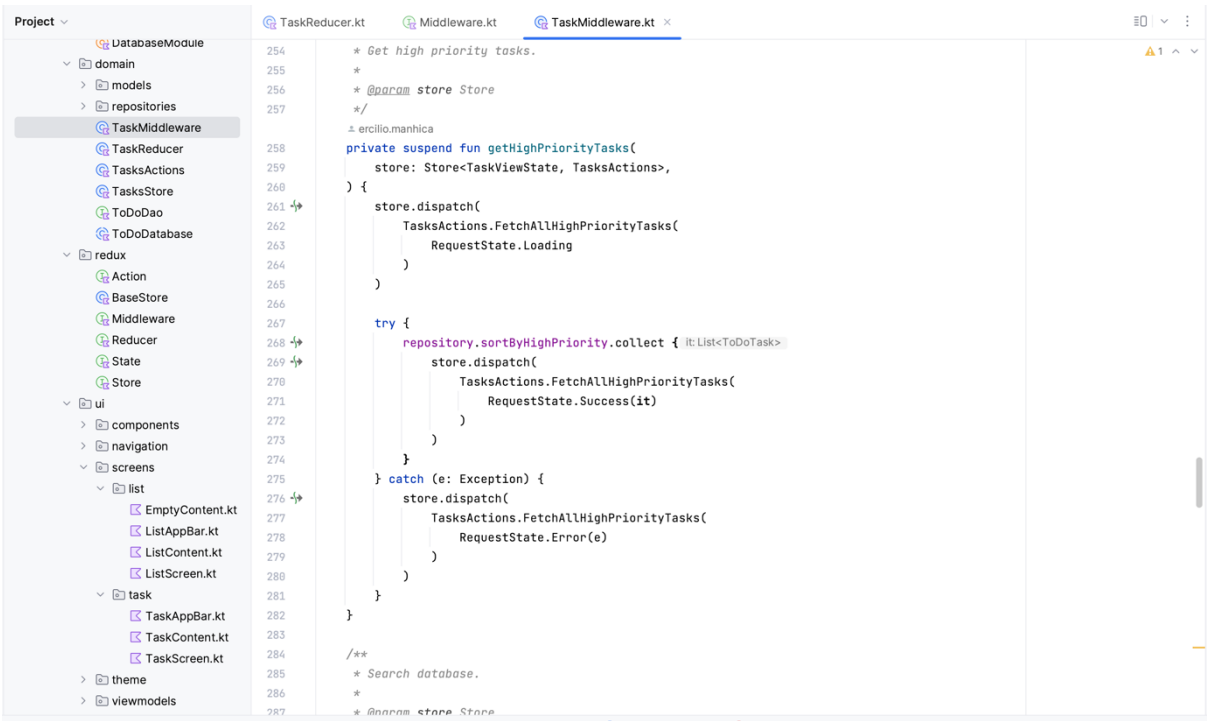

Project
TaskReducer.kt
Middleware.kt
TaskMiddleware.kt
DatabaseModule
domain
models
repositories
TaskMiddleware
TaskReducer
TasksActions
TasksStore
ToDoDao
ToDoDatabase
redux
Action
BaseStore
Middleware
Reducer
State
Store
ui
components
navigation
screens
list
EmptyContent.kt
ListAppBar.kt
ListContent.kt
ListScreen.kt
task
TaskAppBar.kt
TaskContent.kt
TaskScreen.kt
theme
viewmodels
/**
* Add task.
*
* @param store Store
* @param currentState Current state
*/
ercilio.manhica
private suspend fun addTask(
store: Store<TaskViewState, TasksActions>,
currentState: TaskViewState
) {
val toDoTask = ToDoTask(
title = currentState.title,
description = currentState.description,
priority = currentState.priority
)
repository.addTask(toDoTask = toDoTask)
store.dispatch(TasksActions.UpdateAppBarState(SearchAppBarState.CLOSED))
store.dispatch(
TasksActions.FetchShowSnackBar(
ShowSnackBar(
opened = true,
label = ActionLabels.ADD,
message = "ADD: ${toDoTask.title}"
)
)
)
}
/**
* Update task.
*
* @param currentState Current state
*/


Project
TaskReducer.kt
Middleware.kt
TaskMiddleware.kt
DatabaseModule
domain
models
repositories
TaskMiddleware
TaskReducer
TasksActions
TasksStore
ToDoDao
ToDoDatabase
redux
Action
BaseStore
Middleware
Reducer
State
Store
ui
components
navigation
screens
list
EmptyContent.kt
ListAppBar.kt
ListContent.kt
ListScreen.kt
task
TaskAppBar.kt
TaskContent.kt
TaskScreen.kt
theme
viewmodels
/**
* Update task.
*
* @param currentState Current state
*/
ercilio.manhica
private suspend fun updateTask(
store: Store<TaskViewState, TasksActions>,
currentState: TaskViewState
) {
val toDoTask = ToDoTask(
id = currentState.id,
title = currentState.title,
description = currentState.description,
priority = currentState.priority
)
repository.updateTask(toDoTask = toDoTask)
store.dispatch(
TasksActions.FetchShowSnackBar(
ShowSnackBar(
opened = true,
label = ActionLabels.UPDATE,
message = "UPDATE: ${toDoTask.title}"
)
)
)
}
/**
* Delete task.
*
* @param currentState Current state
*/
ercilio.manhica

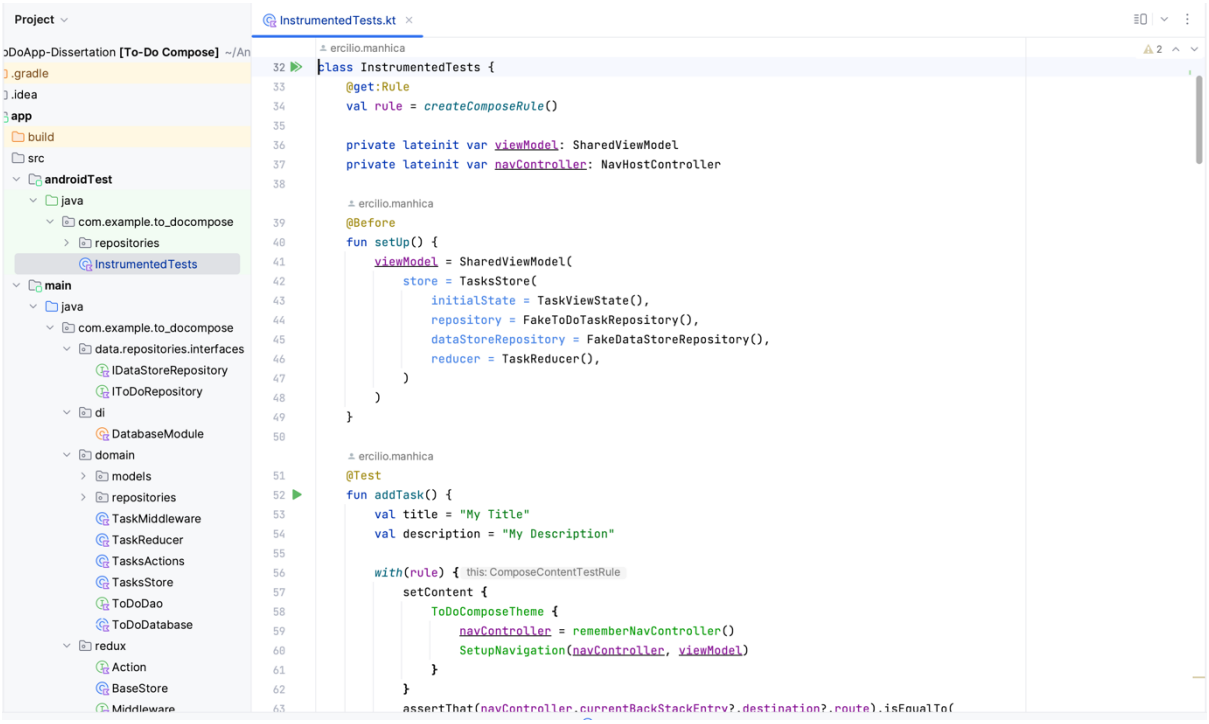

Project
TaskReducer.kt
Middleware.kt
TaskMiddleware.kt
DatabaseModule
domain
models
repositories
TaskMiddleware
TaskReducer
TasksActions
TasksStore
ToDoDao
ToDoDatabase
redux
Action
BaseStore
Middleware
Reducer
State
Store
ui
components
navigation
screens
list
EmptyContent.kt
ListAppBar.kt
ListContent.kt
ListScreen.kt
task
TaskAppBar.kt
TaskContent.kt
TaskScreen.kt
theme
viewmodels
/**
* Delete task.
*
* @param currentState Current state
*/
ercilio.manhica
private suspend fun deleteTask(
store: Store<TaskViewState, TasksActions>,
currentState: TaskViewState
) {
val toDoTask = ToDoTask(
id = currentState.id,
title = currentState.title,
description = currentState.description,
priority = currentState.priority
)
repository.deleteTask(toDoTask = toDoTask)
store.dispatch(
TasksActions.FetchShowSnackBar(
ShowSnackBar(
opened = true,
label = ActionLabels.DELETE,
message = "DELETE: ${toDoTask.title}"
)
)
)
}
/**
* Delete all tasks.
*/
ercilio.manhica
private suspend fun deleteAllTasks(
store: Store<TaskViewState, TasksActions>,

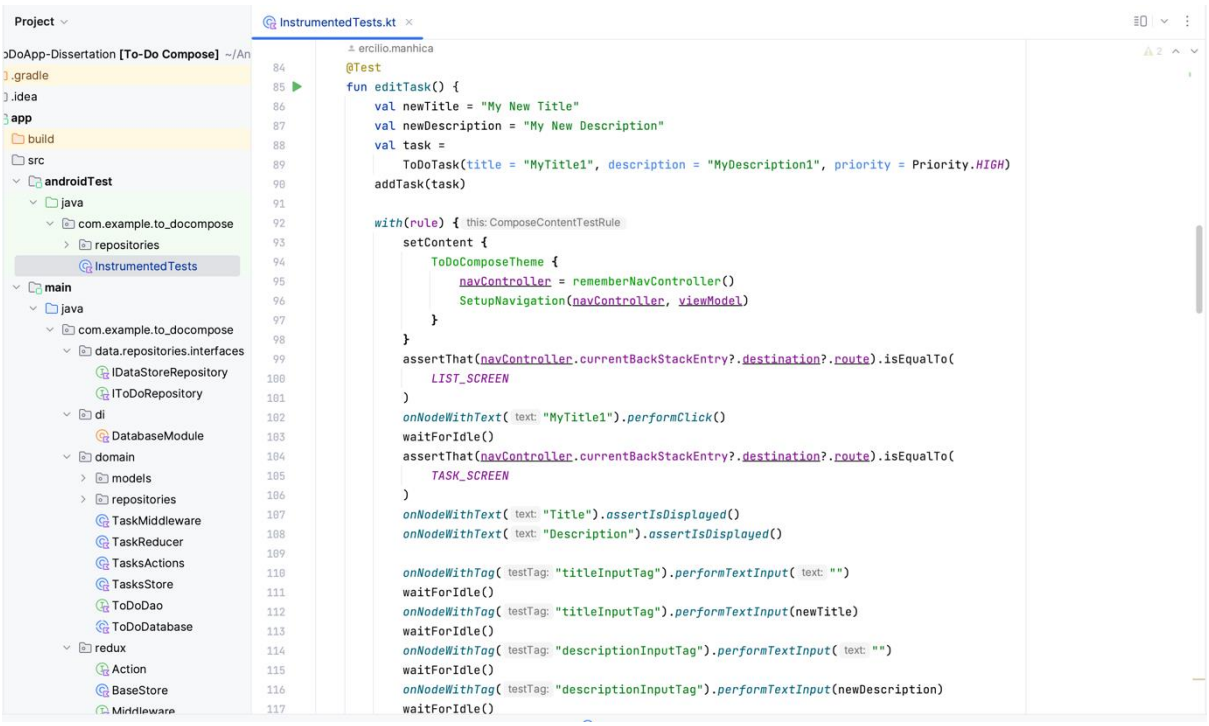

Project
TaskReducer.kt
Middleware.kt
TaskMiddleware.kt
DatabaseModule
domain
models
repositories
TaskMiddleware
TaskReducer
TasksActions
TasksStore
ToDoDao
ToDoDatabase
redux
Action
BaseStore
Middleware
Reducer
State
Store
ui
components
navigation
screens
list
EmptyContent.kt
ListAppBar.kt
ListContent.kt
ListScreen.kt
task
TaskAppBar.kt
TaskContent.kt
TaskScreen.kt
theme
viewmodels
* @param store Store
*/
ercilio.manhica
private suspend fun getAllTasks(store: Store<TaskViewState, TasksActions>) {
store.dispatch(
TasksActions.FetchAllTasks(
RequestState.Loading
)
)
try {
repository.getAllTasks.collect { it: List<ToDoTask>
store.dispatch(
TasksActions.FetchAllTasks(
RequestState.Success(it)
)
)
}
} catch (e: Exception) {
store.dispatch(
TasksActions.FetchAllTasks(
RequestState.Error(e)
)
)
}
}
/**
* Get low priority tasks.

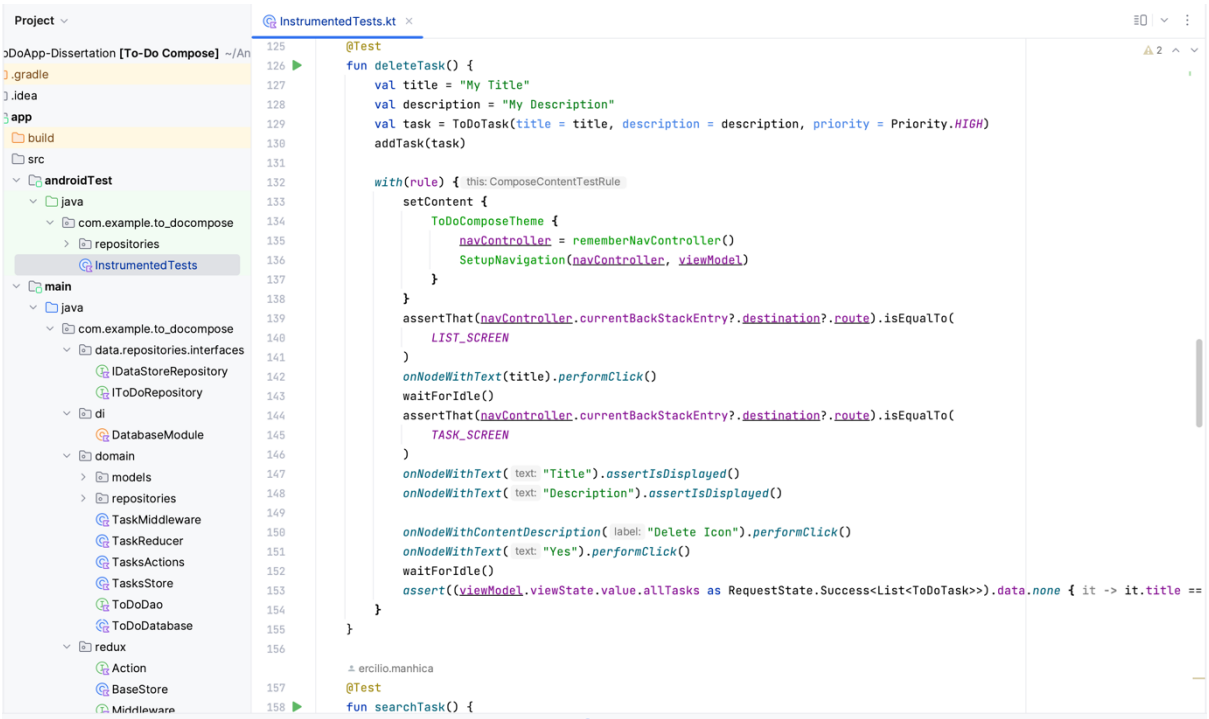

Project
TaskReducer.kt
Middleware.kt
TaskMiddleware.kt
DatabaseModule
domain
models
repositories
TaskMiddleware
TaskReducer
TasksActions
TasksStore
ToDoDao
ToDoDatabase
redux
Action
BaseStore
Middleware
Reducer
State
Store
ui
components
navigation
screens
list
EmptyContent.kt
ListAppBar.kt
ListContent.kt
ListScreen.kt
task
TaskAppBar.kt
TaskContent.kt
TaskScreen.kt
theme
viewmodels
* @param store Store
*/
ercilio.manhica
private suspend fun getLowPriorityTasks(
store: Store<TaskViewState, TasksActions>,
) {
store.dispatch(
TasksActions.FetchAllLowPriorityTasks(
RequestState.Loading
)
)
try {
repository.sortByLowPriority.collect { it: List<ToDoTask>
store.dispatch(
TasksActions.FetchAllLowPriorityTasks(
RequestState.Success(it)
)
)
}
} catch (e: Exception) {
store.dispatch(
TasksActions.FetchAllLowPriorityTasks(
RequestState.Error(e)
)
)
}
}
/**
* Get high priority tasks.
*
* @param store Store

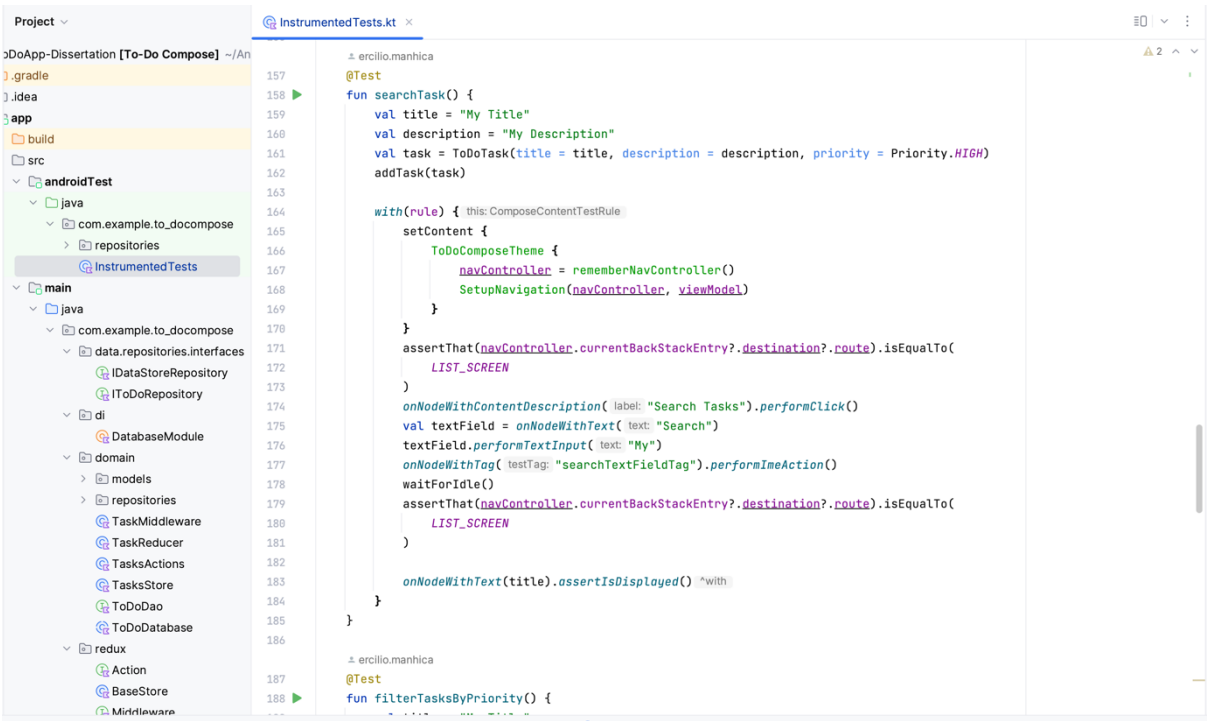

Project
TaskReducer.kt
Middleware.kt
TaskMiddleware.kt
DatabaseModule
domain
models
repositories
TaskMiddleware
TaskReducer
TasksActions
TasksStore
ToDoDao
ToDoDatabase
redux
Action
BaseStore
Middleware
Reducer
State
Store
ui
components
navigation
screens
list
EmptyContent.kt
ListAppBar.kt
ListContent.kt
ListScreen.kt
task
TaskAppBar.kt
TaskContent.kt
TaskScreen.kt
theme
viewmodels
* Get high priority tasks.
*
* @param store Store
*/
ercilio.manhica
private suspend fun getHighPriorityTasks(
store: Store<TaskViewState, TasksActions>,
) {
store.dispatch(
TasksActions.FetchAllHighPriorityTasks(
RequestState.Loading
)
)
try {
repository.sortByHighPriority.collect { it: List<ToDoTask>
store.dispatch(
TasksActions.FetchAllHighPriorityTasks(
RequestState.Success(it)
)
)
}
} catch (e: Exception) {
store.dispatch(
TasksActions.FetchAllHighPriorityTasks(
RequestState.Error(e)
)
)
}
}
/**
* Search database.
*

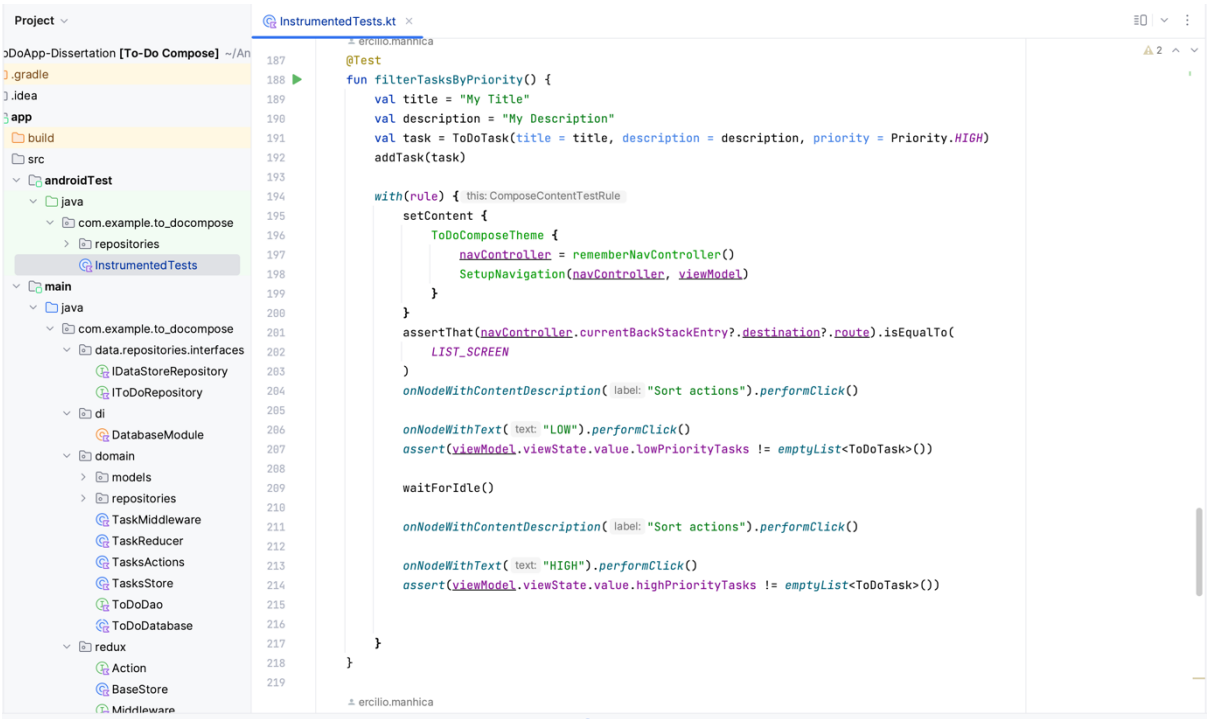
5. **Testes Instrumentais para MVI**:

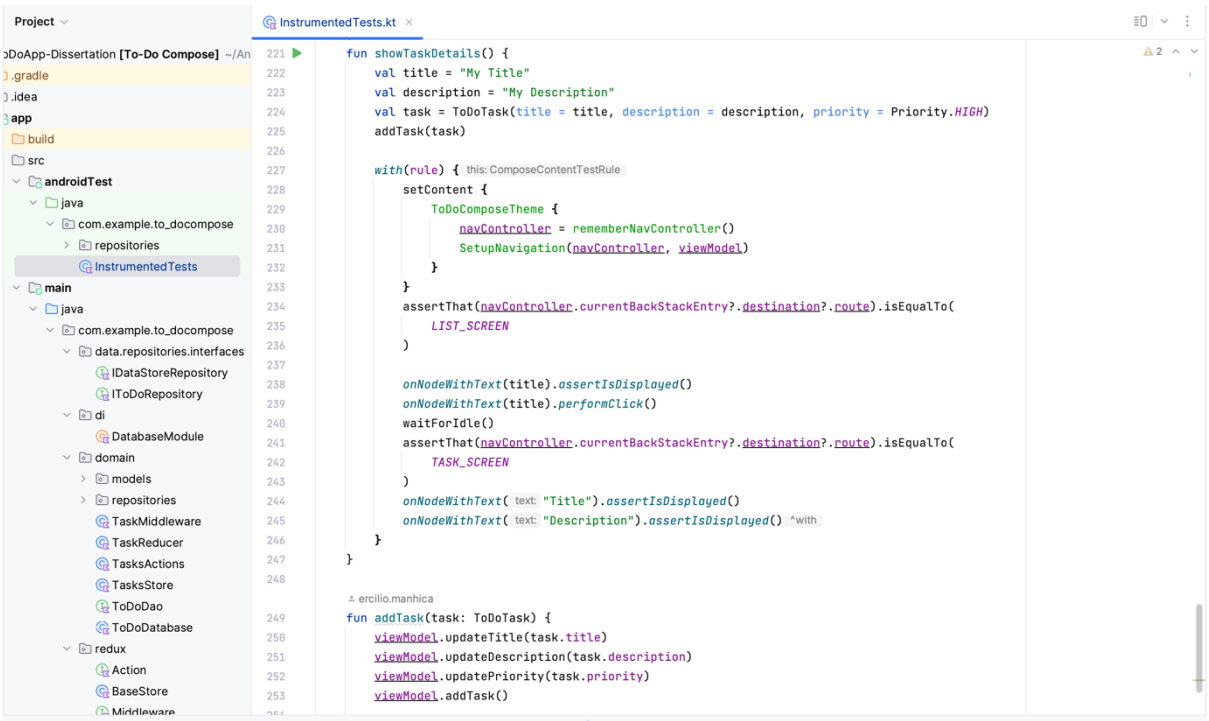




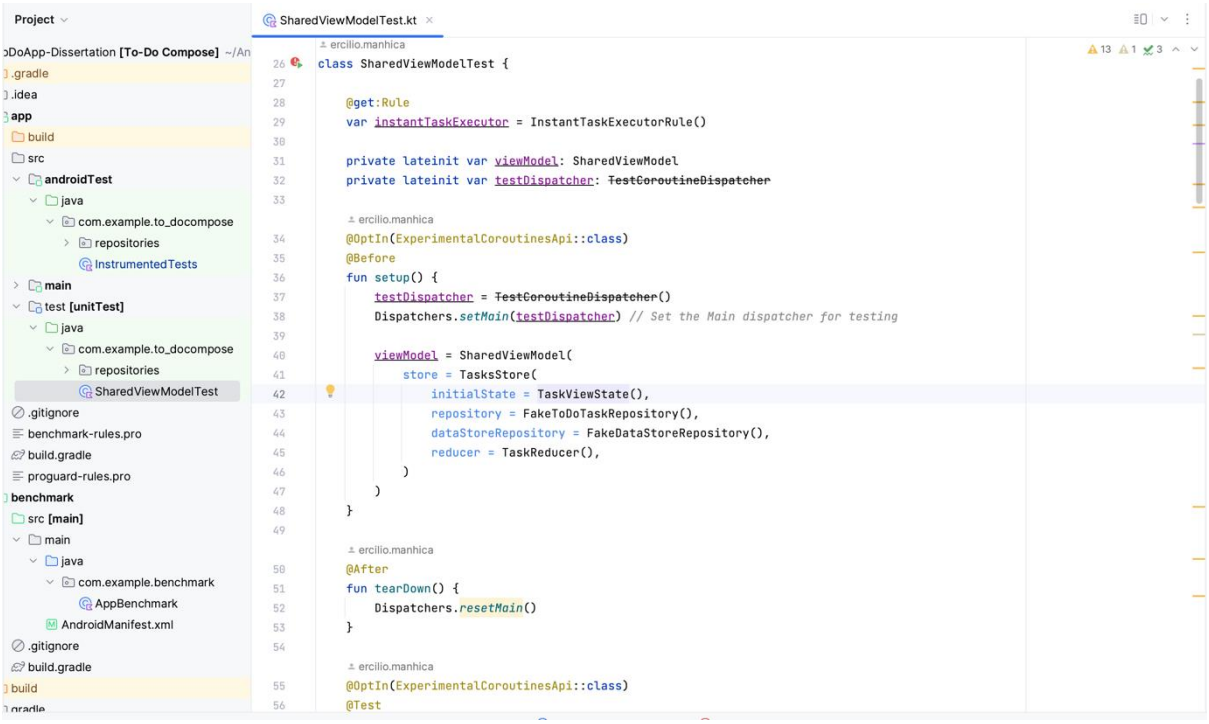

Project
InstrumentedTests.kt
oDoApp-Dissertation [To-Do Compose] ~/An
.gradle
.idea
app
build
src
androidTest
java
com.example.to_docompose
repositories
InstrumentedTests
main
java
com.example.to_docompose
data.repositories.interfaces
IDataStoreRepository
IToDoRepository
di
DatabaseModule
domain
models
repositories
TaskMiddleware
TaskReducer
TasksActions
TasksStore
ToDoDao
ToDoDatabase
redux
Action
BaseStore
Middleware
@Test
fun deleteTask() {
val title = "My Title"
val description = "My Description"
val task = ToDoTask(title = title, description = description, priority = Priority.HIGH)
addTask(task)
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)
}
}
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithText(title).performClick()
waitForIdle()
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
TASK_SCREEN
)
onNodeWithText(text: "Title").assertIsDisplayed()
onNodeWithText(text: "Description").assertIsDisplayed()
onNodeWithContentDescription(label: "Delete Icon").performClick()
onNodeWithText(text: "Yes").performClick()
waitForIdle()
assert((viewModel.viewState.value.allTasks as RequestState.Success<List<ToDoTask>>).data.none { it -> it.title ==
}
}
ercilio.manhica
@Test
fun searchTask() {

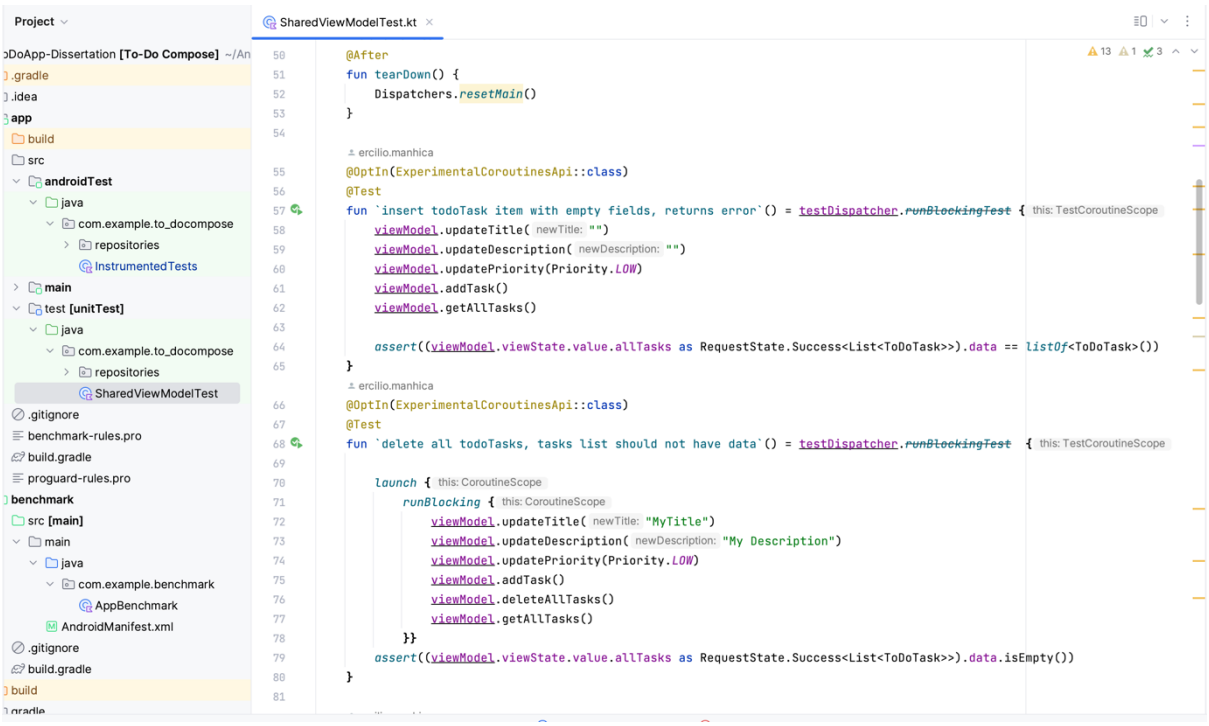

Project
InstrumentedTests.kt
oDoApp-Dissertation [To-Do Compose] ~/An
.gradle
.idea
app
build
src
androidTest
java
com.example.to_docompose
repositories
InstrumentedTests
main
java
com.example.to_docompose
data.repositories.interfaces
IDataStoreRepository
IToDoRepository
di
DatabaseModule
domain
models
repositories
TaskMiddleware
TaskReducer
TasksActions
TasksStore
ToDoDao
ToDoDatabase
redux
Action
BaseStore
Middleware
ercilio.manhica
@Test
fun searchTask() {
val title = "My Title"
val description = "My Description"
val task = ToDoTask(title = title, description = description, priority = Priority.HIGH)
addTask(task)
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)
}
}
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithContentDescription(label: "Search Tasks").performClick()
val textField = onNodeWithText(text: "Search")
textField.performTextInput(text: "My")
onNodeWithTag(testTag: "searchTextFieldTag").performImeAction()
waitForIdle()
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithText(title).assertIsDisplayed() ^with
}
}
ercilio.manhica
@Test
fun filterTasksByPriority() {

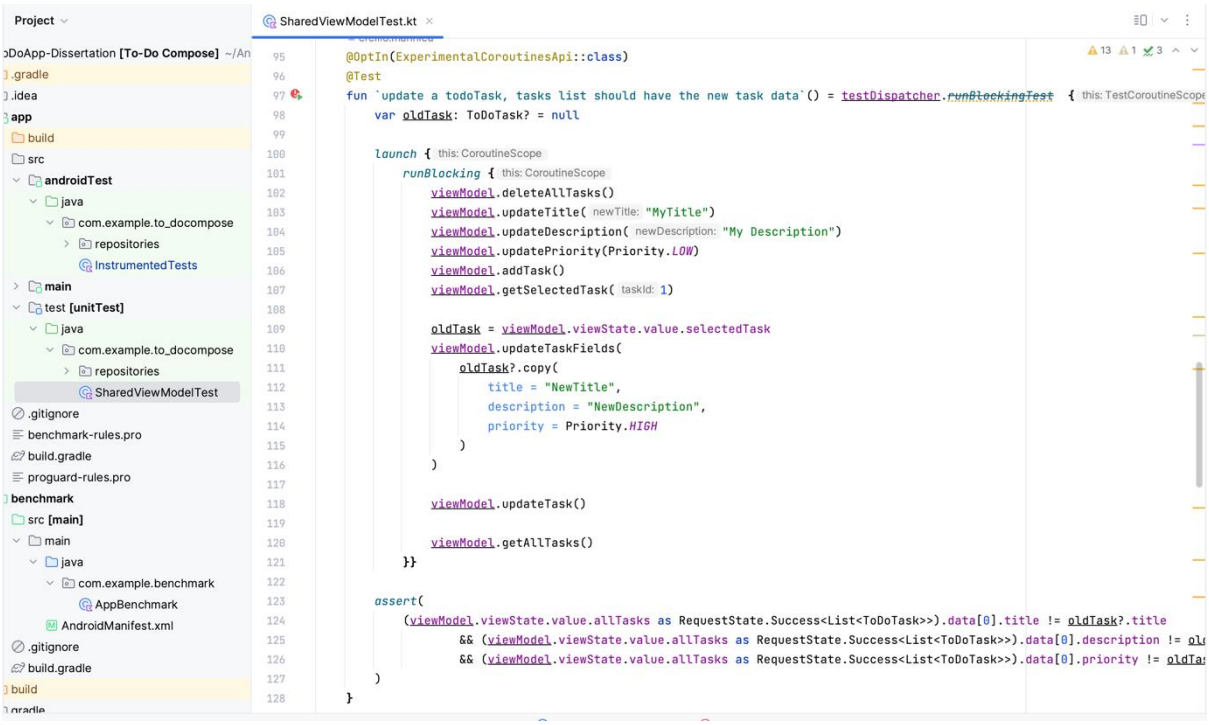

Project
InstrumentedTests.kt
ercilio.manhica
@Test
fun filterTasksByPriority() {
val title = "My Title"
val description = "My Description"
val task = ToDoTask(title = title, description = description, priority = Priority.HIGH)
addTask(task)
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)
}
}
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithContentDescription(label: "Sort actions").performClick()
onNodeWithText(text: "LOW").performClick()
assert(viewModel.viewState.value.lowPriorityTasks != emptyList<ToDoTask>())
waitForIdle()
onNodeWithContentDescription(label: "Sort actions").performClick()
onNodeWithText(text: "HIGH").performClick()
assert(viewModel.viewState.value.highPriorityTasks != emptyList<ToDoTask>())
}
}
ercilio.manhica

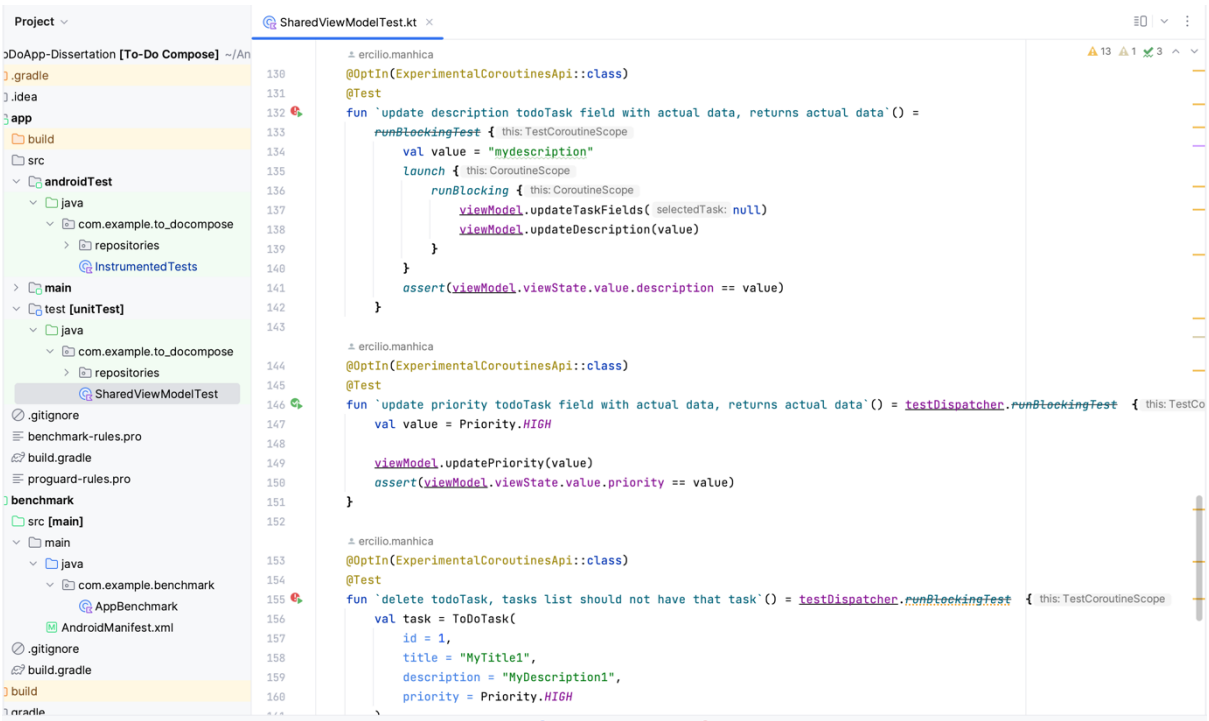

Project
InstrumentedTests.kt
fun showTaskDetails() {
val title = "My Title"
val description = "My Description"
val task = ToDoTask(title = title, description = description, priority = Priority.HIGH)
addTask(task)
with(rule) { this: ComposeContentTestRule
setContent {
ToDoComposeTheme {
navController = rememberNavController()
SetupNavigation(navController, viewModel)
}
}
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
LIST_SCREEN
)
onNodeWithText(title).assertIsDisplayed()
onNodeWithText(title).performClick()
waitForIdle()
assertThat(navController.currentBackStackEntry?.destination?.route).isEqualTo(
TASK_SCREEN
)
onNodeWithText(text: "Title").assertIsDisplayed()
onNodeWithText(text: "Description").assertIsDisplayed() ^with
}
}
ercilio.manhica
fun addTask(task: ToDoTask) {
viewModel.updateTitle(task.title)
viewModel.updateDescription(task.description)
viewModel.updatePriority(task.priority)
viewModel.addTask()

6. **Testes Unitários para MVI**:






Project
SharedViewModelTest.kt
@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun `update a todoTask, tasks list should have the new task data`() = testDispatcher.runBlockingTest { this: TestCoroutineScope
var oldTask: ToDoTask? = null
launch { this: CoroutineScope
runBlocking { this: CoroutineScope
viewModel.deleteAllTasks()
viewModel.updateTitle(newTitle: "MyTitle")
viewModel.updateDescription(newDescription: "My Description")
viewModel.updatePriority(Priority.LOW)
viewModel.addTask()
viewModel.getSelectedTask(taskId: 1)
oldTask = viewModel.viewState.value.selectedTask
viewModel.updateTaskFields(
oldTask?.copy(
title = "NewTitle",
description = "NewDescription",
priority = Priority.HIGH
)
)
viewModel.updateTask()
viewModel.getAllTasks()
}}
assert(
(viewModel.viewState.value.allTasks as RequestState.Success<List<ToDoTask>>).data[0].title != oldTask?.title
&& (viewModel.viewState.value.allTasks as RequestState.Success<List<ToDoTask>>).data[0].description != old
&& (viewModel.viewState.value.allTasks as RequestState.Success<List<ToDoTask>>).data[0].priority != oldTas
)
}


Project
SharedViewModelTest.kt
ercilio.manhica
@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun `update description todoTask field with actual data, returns actual data`() =
runBlockingTest { this: TestCoroutineScope
val value = "mydescription"
launch { this: CoroutineScope
runBlocking { this: CoroutineScope
viewModel.updateTaskFields(selectedTask: null)
viewModel.updateDescription(value)
}
}
assert(viewModel.viewState.value.description == value)
}
ercilio.manhica
@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun `update priority todoTask field with actual data, returns actual data`() = testDispatcher.runBlockingTest { this: TestCo
val value = Priority.HIGH
viewModel.updatePriority(value)
assert(viewModel.viewState.value.priority == value)
}
ercilio.manhica
@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun `delete todoTask, tasks list should not have that task`() = testDispatcher.runBlockingTest { this: TestCoroutineScope
val task = ToDoTask(
id = 1,
title = "MyTitle1",
description = "MyDescription1",
priority = Priority.HIGH

c) Formulário Sobre Utilização de Arquitecturas

Arquitecturas de Desenvolvimento

Prezado(a) Participante,

Este questionário faz parte de uma pesquisa acadêmica conduzida como parte de uma dissertação de mestrado em Engenharia de Software, com o objetivo de entender melhor as práticas e decisões relacionadas ao uso de arquiteturas no desenvolvimento de software. Sua participação é fundamental para o sucesso deste estudo.

As informações coletadas neste questionário serão tratadas de forma confidencial e utilizadas apenas para fins acadêmicos. Não haverá divulgação de informações pessoais ou identificáveis. Agradecemos antecipadamente por dedicar seu tempo para responder às seguintes perguntas.

* Indica uma pergunta obrigatória

1. Qual é a sua função ou posição atual? *

Marcar apenas uma oval.

- ◯ Desenvolvedor de Software
- ◯ Engenheiro de Software
- ◯ Analista de Sistemas
- ◯ Arquiteto de Software
- ◯ Gerente de Projetos de TI
- ◯ Gerente de Desenvolvimento de Software
- ◯ Estudante de TI
- ◯ Outra: _____

2. Qual é a plataforma para qual desenvolves as soluções? *

Marcar tudo o que for aplicável.

- ☐ Android
- ☐ IOS
- ☐ Web
- ☐ Desktop
- ☐ Todas

3. Quanto tempo de experiência possui na área de desenvolvimento de software? *

Marcar apenas uma oval.

- ◯ Menos de 1 ano
- ◯ 1-3 anos
- ◯ 3-5 anos
- ◯ Mais de 5 anos

4. Qual é a sua área ou setor de atuação? *

Marcar apenas uma oval.

- ◯ Tecnologia da Informação (TI)
- ◯ Telecomunicações
- ◯ Educação
- ◯ Saúde
- ◯ Finanças
- ◯ Estado
- ◯ Entretenimento e Mídia
- ◯ Comércio
- ◯ Outra: _____

5. Você utiliza alguma arquitetura específica no desenvolvimento de software? *

Marcar apenas uma oval.

- ◯ Sim
- ◯ Não *Avançar para a pergunta 12*
- ◯ Não sei *Avançar para a pergunta 12*

Arquitecturas Utilizadas

6. Qual arquitetura você utiliza com mais frequência? *

Marcar tudo o que for aplicável.

- [ ] MVC
- [ ] MVP
- [ ] MVVM
- [ ] VIPER
- [ ] Clean Architecture
- [ ] MVI
- [ ] Outra: _____

7. Como você costuma escolher a arquitetura a ser utilizada em um projeto de desenvolvimento de software? *

Marcar tudo o que for aplicável.

- [ ] Experiência prévia
- [ ] Recomendações da equipe
- [ ] Requisitos Técnicos do projeto
- [ ] Requisitos Económicos do projeto
- [ ] Pesquisa independente
- [ ] Costumo usar a arquitectura do Framework
- [ ] Politicas da Empresa
- [ ] Escolha do Cliente
- [ ] Outra: _____

8. Você já realizou alguma avaliação formal de arquiteturas antes de escolher uma para um projeto? *

Marcar apenas uma oval.

- ◯ Sim
- ◯ Não *Avançar para a pergunta 11*

Métodos de Avaliação de Arquitecturas

9. Quais métodos ou critérios você utiliza para avaliar arquiteturas? *

Marcar tudo o que for aplicável.

- [ ] SAAM (Método de Análise de Arquitetura de Software)
- [ ] ATAM (Método de Análise de Compensação Arquitetural)
- [ ] CBAM (Método de Análise de Custo-Benefício)
- [ ] ALMA (Método de Análise de Modificabilidade em Nível de Arquitetura)
- [ ] FAAM (Método de Análise de Arquitetura Familiar)
- [ ] Outra: _____

10. Quais os aspectos considerou ao avaliar a escolha da arquitectura *

Marcar tudo o que for aplicável.

- [ ] Económicos (Custos financeiros do projecto)
- [ ] Perfomance da arquitectura
- [ ] Escabilidade da arquitectura
- [ ] Robustez da arquitectura
- [ ] Segurança da arquitectura
- [ ] Testabilidade da arquitectura
- [ ] Recursos humanos internos
- [ ] Recursos humanos externos
- [ ] Duração do projecto
- [ ] Outra: _____

Avaliação de Arquitecturas

11. Quais motivos te levaram a nunca fazer uma avalição de arquitectura a usar num projecto? *

Marcar tudo o que for aplicável.

- [ ] Falta de Conhecimento: Nunca tive conhecimento sobre a importância
- [ ] Falta de Tempo
- [ ] Falta de Recursos
- [ ] Crença de que não é Necessário
- [ ] Desconhecimento da Relevância Econômica
- [ ] Outras Prioridades: Outras tarefas ou prioridades eram mais urgentes.
- [ ] Outra: _____

Arquitecturas Utilizadas

12. Quais motivos te levaram a nunca usar uma arquitectura especifica num projecto? *

Marcar tudo o que for aplicável.

- [ ] Falta de Conhecimento
- [ ] Complexidade Excessiva
- [ ] Desconhecimento da Relevância
- [ ] Incompatibilidade com Tecnologias em uso
- [ ] Outra: _____