Uso de TDD em aplicação Android MVVM e Clean Architecture

Gabriel Pellegrini Brollo

[1] Instituto Federal do Rio Grande do Sul (IFRS)
Veranópolis – RS – Brasil

gabi.p.brollo@gmail.com

Resumo. *Este artigo evidencia a aplicabilidade do TDD para desenvolvimento de aplicações Android usando MVVM e Clean Architecture, dessa forma, buscando alta qualidade do software desenvolvido. O TDD guia o desenvolvedor para uma solução simples e bem desenhada, aumentando a confiança e facilidade de manutenção do sistema, as arquiteturas usadas proporcionam grande testabilidade dos componentes, e compartilham princípios com o TDD. Foi desenvolvida uma aplicação Android que fará cálculo do IMC e indicará conteúdo para ganho e perda de peso. O desenvolvimento usando TDD aliado ao MVVM e Clean Architecture se mostrou bastante positivo na qualidade do código, na segurança e flexibilidade à mudanças e também no feedback rápido da solução desenvolvida, diminuindo grandes esforços em testes manuais para apenas alguns segundos de testes automatizados, que irão garantir o funcionamento correto da implementação.*

Abstract. *This article evidences the TDD applicability to develop Android apps using MVVM and Clean Architecture, that way, seeking for high quality in the developed software. The TDD guides the developer to a simple and well designed solution, increasing system reliability and maintainability, the used architectures provide great testability of the components, and share principles with TDD. An Android application was developed that will calculate the BMI and indicate content for weight gain and loss. The development using TDD allied to MVVM and Clean Architecture proved to be very positive in terms of code quality, security and flexibility to changes and also in the quick feedback of the developed solution, reducing great efforts in manual tests to just a few seconds of automated tests, which will ensure the correct functioning of the implementation.*

1. Introdução

Desde o lançamento do sistema operacional Android pelo Google em 2008, com o desenvolvimento de novos aplicativos, as funcionalidades disponíveis no sistema aumentaram radicalmente, da mesma forma, esses aplicativos cresceram em termos de complexidade, fazendo com que os desenvolvedores precisem se preocupar com técnicas para criar um sistema flexível à mudanças.

Esse problema de evolução saudável do código-fonte, não é restrito da plataforma Android, basicamente qualquer software, e principalmente os que recebem constantes modificações, podem ser inundados por débitos técnicos durante seu desenvolvimento.

Por esse motivo, os programadores precisam ter conhecimento dos princípios de programação e das arquiteturas de software, para que apliquem os conceitos adequados para cada caso de uso.

Atualmente o Google recomenda o uso de MVVM (Model View ViewModel) como arquitetura para desenvolvimento de aplicativos Android, que basicamente é composta por UI Layer (Camada de interface do usuário), Data Layer (Camada de dados) e opcionalmente Domain Layer (Camada de domínio), por ser a arquitetura recomendada, muitas bibliotecas são criadas para serem compatíveis e agilizar o desenvolvimento dos aplicativos. Para o código ficar ainda mais desacoplado, pode-se também usar alguns conceitos de Clean Architecture, que é bastante popular no desenvolvimento de aplicações orientadas à objetos, principalmente de grande porte.

Na análise de arquiteturas comuns do Android, feita por Nayab Akhtar e Sana Ghafoor, fica explícita a importância da arquitetura para o desenvolvimento:

> Foi observado que as aplicações desenvolvidas sem arquitetura não possibilitam manutenção, não podem ser futuramente evoluídas e isso pode também drasticamente aumentar o custo de desenvolvimento. Arquitetura de software adequada traz simplicidade, reusabilidade e testes eficientes para nossas aplicações. [AKHTAR 2021], tradução nossa. [1]

Este trabalho foi desenvolvido com foco no desenvolvimento Android, mas os conhecimentos apresentados podem ser usados em outras áreas de desenvolvimento de software, e até mesmo para pessoas fora dessa área que desejam ter uma noção de alto nível sobre o assunto.

Essas dificuldades que são comuns de encontrar no desenvolvimento de software e também a possibilidade de aplicar arquiteturas e técnicas que procuram melhorar a estrutura do software, nos motivam a experimentá-las juntamente com o TDD (Test Driven Development / Desenvolvimento orientado a teste), tendo visto que com isso será possível aproveitar o melhor de cada técnica envolvida.

Nesse cenário, objetiva-se provar a aplicabilidade de TDD no desenvolvimento Android MVVM com Clean Architecture, visando ganhos na qualidade de código, diminuindo o acoplamento e complexidade dos componentes, e também aumentando a segurança no momento de executar manutenções. Além do mais, procura-se analisar possíveis incompatibilidades da arquitetura com técnicas de TDD.

Levando em consideração a pirâmide de testes, não será tratado sobre testes de interface do usuário, devido ao tempo de execução desse tipo de testes.

Para tal propósito, será desenvolvido um protótipo de aplicativo que calcula o IMC de uma pessoa, e faz indicações de conteúdo baseado no resultado, usando TDD nos comportamentos da aplicação, regras de negócio e fluxos de dados, tendo como arquitetura o

[1] "It has been observed that the application developed without architecture are not maintainable cannot be further developed and it can also drastically increase the cost of the development. Proper software architecture brings simplicity, reusability and efficient testing in our applications."

MVVM e Clean Architecture.

A seguir apresentam-se fundamentos teóricos relacionados ao tema proposto. Continuando, serão aplicadas técnicas de TDD no desenvolvimento do protótipo que será analisado. Por fim conclui-se o artigo com os resultados e conhecimentos obtidos.

2. MVVM

Atualmente o Google recomenda o uso da arquitetura MVVM, e também que o desenvolvedor adapte-a para as necessidades de cada caso de uso. Além de recomendada, diversas bibliotecas são fornecidas pelo Google ou por projetos open source de terceiros, para facilitar e agilizar o desenvolvimento dos aplicativos que optam por essa arquitetura.

Essa arquitetura será facilitadora para que possamos aplicar o TDD:

> [...] o framework MVVM fornece uma camada View-Model separada para isolar os dados da interface do usuário. Com a característica de baixo acoplamento do MVVM, a capacidade de manutenção e testabilidade da aplicação são bastante aprimoradas. [ZHANG 2019], tradução nossa. [2]

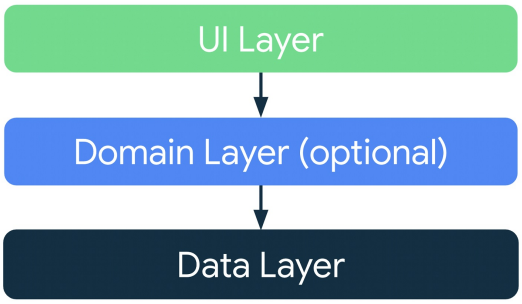

Figure 1. Diagrama das camadas do MVVM [GOOGLE 2022]

Conforme a documentação do Google, a arquitetura é dividida em 3 camadas (Figura 1), que são [GOOGLE 2022]:

1 - Camada de apresentação (UI Layer): é responsável por exibir componentes visuais e os dados do aplicativo na tela, e permitir a interação do usuário. Os principais componentes dessa camada são: Activity, Fragment e ViewModel.

2 - Camada de domínio (Domain Layer): é responsável por encapsular regras de negócio complexas e não complexas para promover a reusabilidade. Essa camada é opcional, pois nem todos aplicativos lidam com complexidade ou necessidade de reutilizar lógicas de negócio. Os principais componentes dessa camada são: UseCase e Interactor.

3 - Camada de dados (Data Layer): é responsável pelas regras da aplicação que definem como o aplicativo cria, armazena e altera os dados. Os principais componentes

[2] "[...] the MVVM framework provides a separate View-Model layer to isolate data from UI. With the low coupling characteristics of MVVM, the maintainability and testability of the application are greatly improved."

dessa camada são: Repository e DataSource.

3. Clean Architecture

Robert C. Martin (também conhecido como Uncle Bob) propõem uma arquitetura que tem o objetivo de promover a independência da camada de negócio em relação às outras camadas da aplicação [MARTIN 2017]. Fazendo com que a camada de negócio seja mantida intacta mesmo em casos de grandes alterações nas demais camadas, por exemplo: A troca de uma interface desktop para uma interface web, ou a troca de uma persistência em arquivos para um banco de dados relacional, não devem causar impactos nas regras de negócio.

Os círculos indicam diferentes áreas do software (Figura 2), quanto mais externo o círculo mais baixo nível, e quanto mais interno o círculo mais alto nível. As regras de negócio se dispõe nos círculos internos e precisam ser protegidos de alterações externas, com isso o mesmo pontua a regra de dependência: "As dependências do código-fonte devem apontar apenas para dentro, para políticas de nível superior." [MARTIN 2017]. Isso quer dizer que a camada de dados e a camada de interface irá depender da camada de negócio, que por si não depende de nenhuma outra, dessa forma, no código-fonte da camada de negócio não se deve fazer uso de nada que foi declarado nas outras camadas.

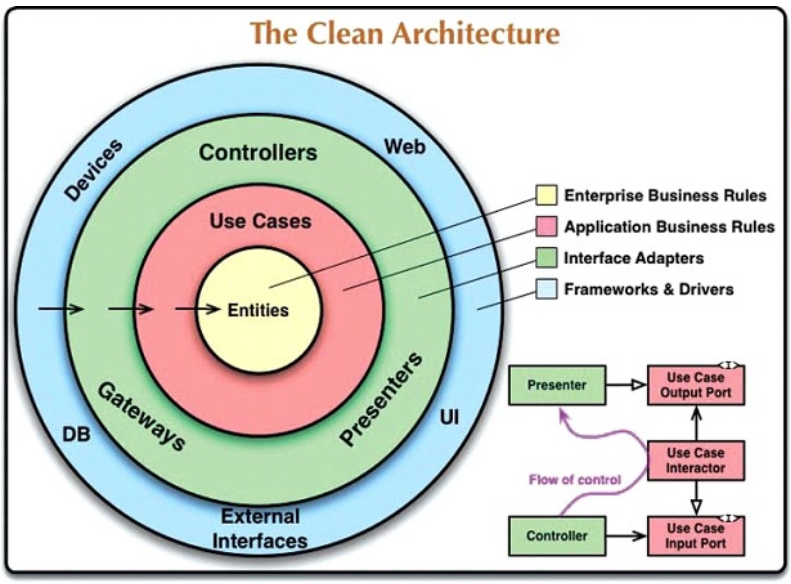

Figure 2. Diagrama da Clean Architecture [MARTIN 2017].

4. TDD

Test-Driven Development (Desenvolvimento orientado a testes) são técnicas que encorajam um desenvolvimento simplificado da solução em pequenos passos, o conjunto de testes passa confiança no momento de compreensão e manutenção do código-fonte ficará disponível após o desenvolvimento.

Muitas vezes os programadores não têm o entendimento claro e completo da solução necessária para resolver determinado problema, com isso o código vai sendo

desenvolvido e modificado até o momento que o software funciona. O código que funciona nem sempre é o código ideal, existe a necessidade de um código que funcione mas também esteja adequado para ser compreensível e estendido em um momento futuro.

Conforme Beck, o TDD existe para auxiliar no desenvolvimento de código limpo que funciona, dividindo o problema existente em pequenas partes, que são testes unitários, cada teste é focado em apenas uma micro necessidade do sistema [BECK 2003]. Esses testes primeiramente são usados para validar que a solução está funcionando durante seu desenvolvimento, após isso eles encorajam e possibilitam a refatoração do código com segurança buscando deixá-lo limpo. Terminando assim com uma solução que passou por um processo de iterações de criação e aperfeiçoamento até que se tem um código funcionando e limpo.

Robert C. Martin explica os passos que devem ser seguidos, esses que ficaram conhecidos como: “As 3 regras do TDD” [MARTIN 2021].

1 - Você não tem permissão para escrever nenhum código de produção, a menos que seja para fazer um teste de unidade que está falhando, ter sucesso.

2 - Você não tem permissão para escrever mais testes de unidade do que o suficiente para falhar, e falhas de compilação são falhas.

3 - Você não tem permissão para escrever mais código de produção do que o suficiente para fazer o teste falhando ter sucesso.

Abaixo é possível ver o diagrama do ciclo de TDD considerando as regras acima:

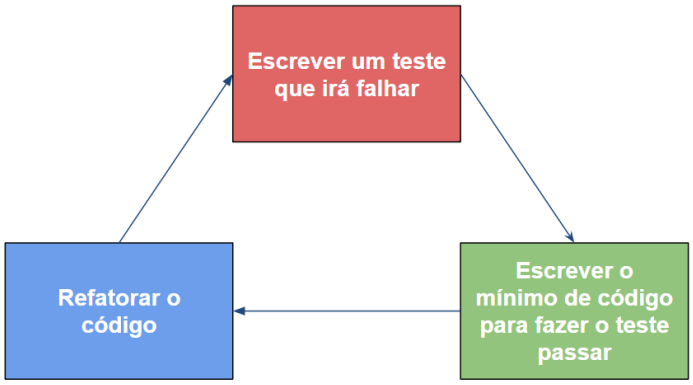

Figure 3. Diagrama ciclo TDD, elaborado pelo autor.

Dessa forma só se escreve código de produção, para fazer um teste que está falhando ter sucesso, de acordo com Beck, para fazer um teste falho passar para sucesso, pode-se usar 3 técnicas [BECK 2003]:

1 - fake-it: Retornar um valor constante que o teste está esperando, e progressivamente trocar essas constantes por variáveis até ter o código real.

2 - Implementação óbvia: Quando a solução é trivial, pode-se ir direto à implementação real.

3 - Triangulação: Prover variações de amostras, e ir generalizando a implementação para atender todos os casos [TARLINDER 2016].

Baseado nesses pequenos passos, ao final de todas as iterações, está pronta a solução funcional e bem desenvolvida, o conjunto de testes que foi escrito estará fornecendo segurança para futuras modificações no software e também servindo como documentação para o entendimento dos detalhes da solução.

5. Desenvolvimento

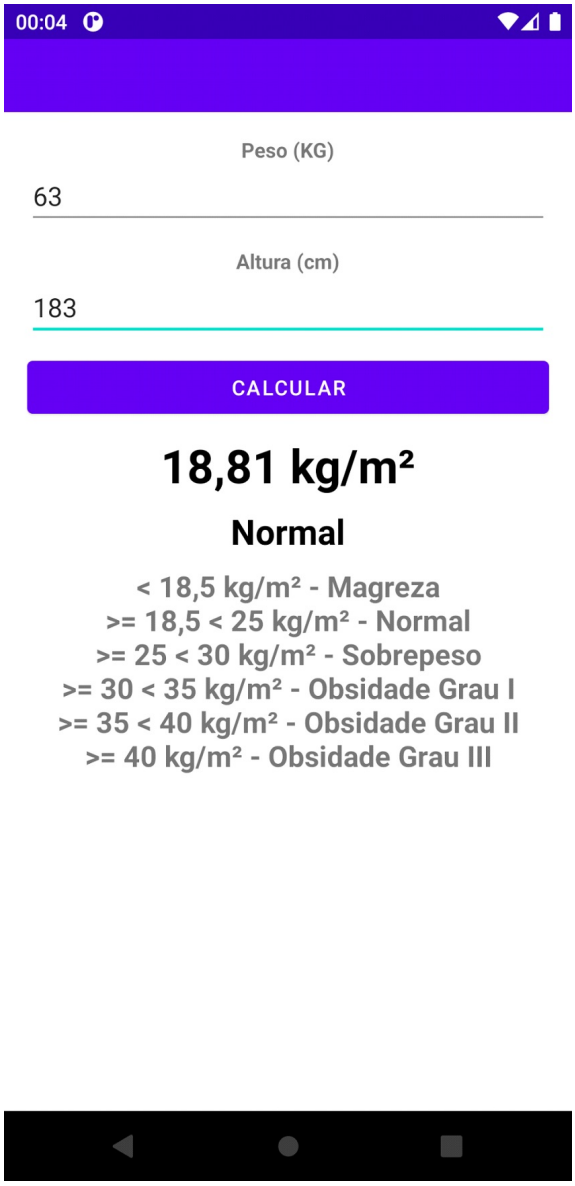

Figure 4. Tela de cálculo de IMC, elaborado pelo autor.

O aplicativo protótipo desenvolvido (Figura 4), tem a funcionalidade de calcular o IMC de uma pessoa, e de acordo com o resultado, fazer indicação de receitas tanto para

ganho como perda de peso, usando uma API pública.

Os componentes de tela e layouts, não foram testados neste trabalho, devido a necessidade de execução em um dispositivo Android físico ou virtual, pois isso deixa o feedback do teste muito demorado, ficando impraticável a dinâmica de desenvolvimento com o TDD (Item 4). Para o presente trabalho, foi considerado suficiente a aplicação de testes nas regras de negócio.

Após criação do projeto no Android Studio, foram configuradas as dependências necessárias, e a versão da linguagem Kotlin (1.6.21). Foi utilizado o jUnit 4 para criação dos testes, e o Koin como framework de injeção de dependências, por ser desenvolvido em Kotlin e ter total compatibilidade com componentes da arquitetura MVVM.

Devido a não aplicação de TDD para interface, todas telas foram desenvolvidas por primeiro, usando XML e ViewBinding. Toda atualização da tela se dá pela observação de uma variável viewState localizada no ViewModel, conforme o valor dessa variável for atualizado, os dados serão refletidos na tela, dessa forma, a interface depende do controlador, aderindo às regras de dependências do Clean Architecture.

Usando Clean Architecture, o desenvolvimento das regras de negócio é independente de outras camadas, em vista disso, este trabalho procura se aproveitar ao máximo dessa independência para facilitar a criação dos testes para as regras de negócio com a aplicação do TDD.

Como início do desenvolvimento, foi pesquisado a tabela de faixas de IMC, e criado um enum com o IMC mínimo e máximo de cada faixa, essas informações fazem parte da funcionalidade principal do aplicativo, e serão acessadas globalmente. Tendo a tabela de faixas, foi criada uma função que ao receber um IMC por parâmetro, retorna em qual faixa de IMC o valor está presente, assim definindo o que foi nomeado de classe de IMC, que são: Magreza, Normal, Sobrepeso, Obesidade Grau I, Obesidade Grau II, Obesidade Grau III.

REQUISITO	REGRA	TESTE
Informar peso e altura	Na tela o aplicativo terá dois campos númericos para informar o peso e altura	
Cálculo IMC	Ao clicar no botão calcular, o sistema irá usar o peso e altura informados para calcular o IMC e mostrar em qual classificação o valor está posicionado. Classificações de IMC: Magreza, Normal, Sobrepeso, Obesidade Grau I, Obesidade Grau II e Obesidade Grau III.	Garantir que todas as faixas de IMC estão sendo calculadas corretamente, e em caso de IMC negativo, deve retornar uma exceção.
Indicar Receita	Ao clicar no resultado do IMC, o usuário será direcionado para uma tela de indicação de receitas, usuários abaixo do peso receberão indicação de receitas para ganhar peso usuário acima do peso receberão receitar para perder peso.	Garantir que a indicação de receitas será adequada ao IMC do usuário, e para casos de classificação normal, não será necessário buscar indicação de receitas.

Figure 5. Tabela de regras de negócio, elaborado pelo autor.

A primeira regra de negócio a ser desenvolvida usando TDD foi o cálculo do IMC, sendo assim o primeiro teste foi criado, buscando simular um caso de magreza, nele foi feito uso de uma classe UseCase, com um método que recebe por parâmetro a altura (m) e peso (kg), até esse momento, a classe e o método citados ainda não existiam, estavam somente sendo utilizados no teste. Tendo em vista os passos do TDD, um teste foi criado e estava falhando, no caso, por motivo de falha de compilação, dessa forma, o próximo passo foi a criação da classe UseCase com a implementação do método, que inicialmente

retornava um valor fixo indicando magreza (17), mesmo não sendo a implementação completa do cálculo de IMC, dessa forma o primeiro teste estava tendo sucesso.

Para continuar o desenvolvimento do cálculo, foi criado outro teste, agora validando IMCs na faixa normal, o teste começou falhando, claramente porque a implementação sempre retornava magreza. Para ter sucesso neste teste, devido a ser uma implementação trivial, foi implementado o cálculo de IMC por completo, dividindo o peso pela altura ao quadrado (kg/m²). Com o cálculo completo, foi rápido criar os testes para todas outras faixas de IMC, e também para valores inválidos como negativos, no qual é validado para o teste receber uma exceção, dessa forma foi feita a triangulação de todos casos que podem acontecer no sistema, ou seja, nessa regra de negócio foi possível aplicar as 3 técnicas do TDD (Item 4).

Após todos testes do cálculo terem passado com sucesso, foram feitos alguns testes de mutação, para validar que os testes escritos estavam realmente testando o que se esperava, para isso, foi comentado ou alterado parte do código de produção em pontos que fariam o cálculo funcionar de forma inesperada, a cada mutação os testes foram executados e sempre algum deles falhava, demonstrando que caso a funcionalidade do cálculo fosse alterada de forma errônea, os testes avisariam o desenvolvedor, dando segurança em futuras manutenções. Nesse momento o desenvolvimento da regra de negócio estava finalizada com uma suíte de testes garantindo seu correto funcionamento.

A interface onde o usuário informa peso e altura, passou a fazer uso da funcionalidade do cálculo. Após usar o aplicativo, foi preferido alterar a unidade de medida da altura de metros para centímetros, simulando uma alteração na regra de negócio (Figura 6), para continuar aplicando TDD e contemplar essa necessidade.

Com a alteração da unidade de medida da altura, também foi alterado o tipo de dado de Double para Int, logo, os testes foram atualizados para trabalhar com a nova unidade de medida e novo tipo de dado, desta maneira, os testes passaram a falhar por erro de compilação, pois o método do cálculo ainda estava no antigo formato. O próximo passo foi fazer o teste compilar, alterando o tipo de dado dos parâmetros, após isso os testes foram executados, mas continuaram falhando, pois nossa implementação do cálculo ainda estava trabalhando com metros e não centímetros.

Neste momento o TDD estava indicando fazer o teste em falha passar com sucesso, então para isso, dentro do método do cálculo foi feita a conversão da unidade de medida da altura, após isso, todos os testes passaram com sucesso, provando que a funcionalidade estava de acordo com o comportamento desejado.

Figure 6. Modificações no código para aceitar a alteração na regra de negócio, elaborado pelo autor.

Após o cálculo de IMC estar completo, foi iniciada a funcionalidade de indicar receitas de acordo com o resultado do IMC. Novamente o layout da tela foi desenvolvido anteriormente, para continuar com foco no TDD da regra de negócio.

O primeiro teste para o UseCase responsável por buscar as receitas foi iniciado, imaginando como gostaria que fosse a interface do UseCase para o resto da aplicação, logo foi criada a classe para ele, onde no construtor é injetada uma instância da interface Repository, trazendo os conceitos de inversão de dependência do Clean Architecture, ou seja, a nossa camada de domínio não irá depender da camada que realmente vai buscar as receitas (camada de dados), mas sim a camada de dados vai precisar implementar a interface de Repository, que está localizada na camada de domínio (Figura 7). Com isso podemos desenvolver e testar nosso UseCase sem necessidade da camada de dados, fazendo uso de Mocks.

Mock é uma técnica bastante útil, os objetos Mock podem ser considerados uma técnica padrão no teste de software, permitindo testar um componente isolado, simulando o comportamento de suas dependências [SPADINI, D; ANICHE, M; BRUNTINK, M, BACCHELLI, A. 2018].

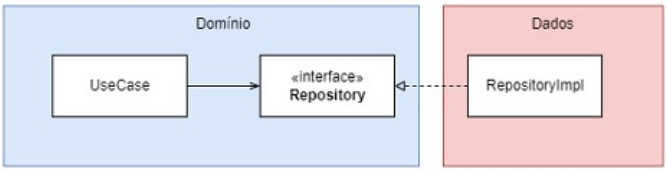

Figure 7. Diagrama de dependências entre UseCase e Repository, elaborado pelo autor.

O primeiro teste ficou responsável por garantir que quando forem buscadas as receitas, a lista de todas receitas deve ser retornada. Para ter sucesso neste teste, foi trivial fazer a chamada da busca de receitas presente no Repository, que no teste foi "mockado" para devolver uma lista de receitas com sucesso.

O segundo teste foi para validar que quando algum erro acontecer na busca das receitas, uma mensagem de erro será retornada. O teste já rodou a primeira vez com sucesso, sendo que também o Repository foi "mockado" retornando a mensagem de erro.

Com esses testes obtendo sucesso, foi analisado novamente o problema, procurando observar quais outras regras deveriam estar presentes nessa solução. Com isso, foi percebido que quando o IMC estiver na faixa normal, não se deve buscar receitas, essa regra de negócio tinha sido implementada na tela, mas precisava fazer parte do UseCase, sendo que em outros componentes do aplicativo poderia ser necessária a mesma validação, fazendo com que o código fosse duplicado, tendo-a no UseCase a reusabilidade da regra de negócio é aprimorada.

Sendo assim, o terceiro teste garante que não sejam buscadas receitas para o IMC normal, para isso é verificado se o Repository não foi invocado pelo UseCase, e também que seja retornada uma informação específica para a tela tratar de forma diferente. Inicialmente o teste ficou em falha, pois a implementação até o momento estava chamando o Repository em todas faixas de IMC. Então foi implementada uma validação no UseCase, que quando recebido por parâmetro um IMC normal, é diretamente retornado uma

informação de peso normal, sem fazer a busca de receitas, com isso o teste passou de falha para sucesso.

Em seguida, procurando por mais testes possíveis para o UseCase, foi percebido que ele deveria decidir de acordo com IMC, qual o tipo de receita a ser buscado. Então foram criados testes para cada classe de IMC, e validado se o tipo de receita que o UseCase passava para o Repository buscar estava correto.

A partir de então, a camada de domínio está completamente desenvolvida, com duas classes de testes (uma para o cálculo e outra para a busca de receitas) que validam todas funcionalidades e fluxos necessários.

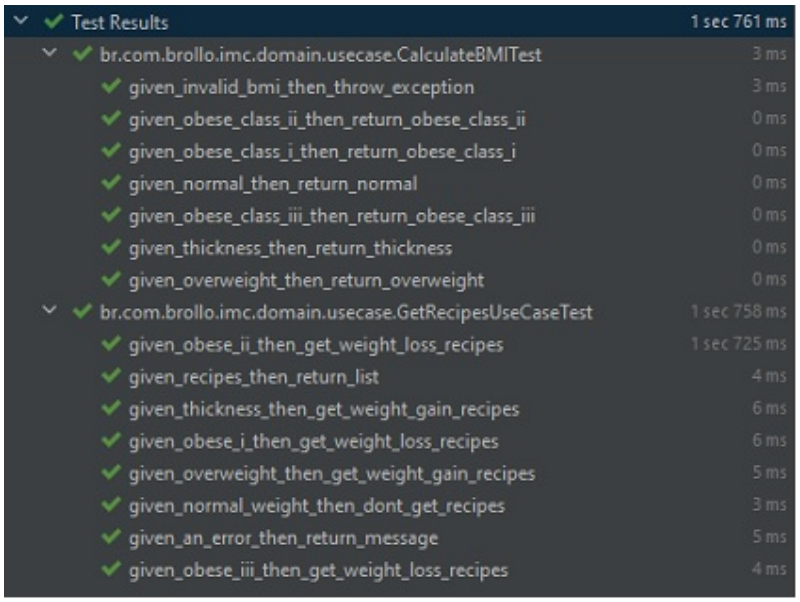

Figure 8. Execução dos testes da camada de domínio, elaborado pelo autor.

Testes (Figura 8) que podem ser executados em menos de 2 segundos no momento que for desejado, assegurando o correto funcionamento das regras de negócio, o que deixa bastante prático em momentos de manutenção no código fonte, pois é possível rodar todos testes rapidamente e ter certeza que nada foi impactado pela alteração executada.

6. Conclusão

O presente trabalho teve como objetivo demonstrar a aplicabilidade de TDD no desenvolvimento de um aplicativo protótipo Android com arquitetura MVVM e Clean Architecture.

Durante o processo de TDD foi possível perceber que até mesmo sem ter as regras da aplicação totalmente claras e definidas inicialmente, de forma incremental o TDD possibilitou a quebra de cada problema em partes menores, favorecendo o entendimento da necessidade para desenvolver uma solução simplificada.

Além de diminuir a complexidade cognitiva no desenvolvimento de uma solução, o TDD também contribuiu na análise das responsabilidades de cada camada da arquitetura, e na definição das APIs dos componentes, aumentando a coesão e legibilidade do

código, aprimorando a qualidade do software e a flexibilidade à mudanças, e possibilitando a refatoração contínua com segurança.

A arquitetura MVVM e Clean Architecture permitem uma separação maior das responsabilidades, com isso a aplicação do TDD é radicalmente facilitada, pois é possível criar testes para pequenos componentes do software e validar cada uma das possibilidades, sem depender de outros componentes que não são o foco do teste em questão.

O uso do componente UseCase para encapsular as regras de negócios pode se mostrar desnecessário nos casos que o problema é muito simples, como quando o UseCase somente chama uma função do Repository e não tem nenhuma lógica própria. Mas por consistência com outras regras de negócio complexas que foram encapsuladas em UseCases ou por deixar previamente um local específico para adição de lógica no futuro, o desenvolvedor pode optar por manter o uso do UseCase. Até mesmo o Google define essa camada como opcional, por isso é responsabilidade do desenvolvedor analisar a real necessidade e preferência para existência do UseCase. Como trabalho futuro pode-se estudar onde fica o ponto de equilíbrio para esse componente.

Um problema complexo pode gerar uma grande quantidade de testes a serem escritos, além do custo em tempo para escrita dos testes, existe um acoplamento do código de teste com o código de produção, podendo fazer com que alterações no código de produção impactem em necessidade de atualização dos testes.

Por fim conclui-se que o desenvolvimento usando TDD aliado ao MVVM e Clean Architecture se mostrou bastante positivo na qualidade do código, na segurança e flexibilidade à mudanças e também no feedback rápido da solução desenvolvida, diminuindo grandes esforços em testes manuais para apenas alguns segundos de testes automatizados, que irão garantir o funcionamento correto da implementação.

References

AKHTAR, N; GHAFOOR, S. (2021). Analysis of architectural patterns for android development. ResearchGate, Acesso em: 02 jun. de 2022.

BECK, K. (2003). *Test-Driven Development: by exemple*. Addison-Wesley Professional, 1º edition.

GOOGLE (2022). Guide to app architecture. Android Developer, Acesso em 02 jun. de 2022.

MARTIN, R. (2017). *Clean Architecture*. Pearson Education, 1º edition.

MARTIN, R. (2021). The cycles of tdd. Clean Coder Blog, Acesso em: 02 jun. de 2022.

SPADINI, D; ANICHE, M; BRUNTINK, M, BACCHELLI, A. (2018). Mock objects for testing java systems. Springer Nature, Acesso em: 28 jun. de 2022.

TARLINDER, A. (2016). *Developer Testing: Building Quality into Software*. Addison-Wesley Professional, 1º edition.

ZHANG, H. (2019). Design and implementation of social event application based on android. Theseus, Acesso em: 02 jun. de 2022.