UNIVERSIDADE FEDERAL DE SANTA CATARINA
Giovani Lopes Schiar Junior

Criação de uma solução arquitetônica para organização de código em aplicações Android

Florianópolis
2019

Giovani Lopes Schiar Junior

Criação de uma solução arquitetônica para organização de código em aplicações Android

Trabalho de Conclusão do Curso de Graduação em Ciências da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Ricardo Pereira e Silva

Florianópolis

2019

Giovani Lopes Schiar Junior

CRIAÇÃO DE UMA SOLUÇÃO ARQUITETÔNICA PARA ORGANIZAÇÃO DE CÓDIGO EM APLICAÇÕES ANDROID

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de “Bacharel” e aprovado em sua forma final pelo curso de Ciências da Computação.

Florianópolis, 26 de junho de 2019.

_____

Prof. José Francisco Danilo De Guadalupe Correa Fletes, Dr.

Coordenador do curso

Banca Examinadora:

_____

Prof Ricardo Pereira e Silva, Dr.

Orientador

_____

Profª. Patrícia Vilain, Dra.

_____

Prof José Eduardo de Lucca, MSc.

Resumo

Com a demanda alta pelo desenvolvimento de aplicativos Android, é necessário encontrar as melhores formas de organizar e estruturar o código desses aplicativos, de forma a maximizar a reutilização de código e a manutenabilidade, e o meio ideal de definir isso é através do uso de uma arquitetura adequada. Os desenvolvedores do sistema operacional Android não definem uma arquitetura recomendada para a criação de aplicativos. Além disso, o modelo de execução do Android e as bibliotecas do Android SDK definem diretivas para o desenvolvimento de aplicativos que dificultam o baixo acoplamento e a alta coesão entre os componentes do aplicativo. Por esses fatores, definir uma arquitetura ideal é um grande desafio no desenvolvimento Android. Neste trabalho, foram estudados diversos padrões de arquitetura usados pela comunidade e foram analisadas suas vantagens e desvantagens no desenvolvimento Android. Também foram estudados os mais modernos componentes de arquitetura disponibilizados pela Google. Foi criado então um projeto *boilerplate* que apresenta uma arquitetura de software que visa auxiliar programadores no desenvolvimento de aplicações Android com características desejáveis de organização de código. Criou-se uma aplicação utilizando esse *boilerplate* criado. Para a avaliação do *boilerplate*, foram criadas duas aplicações utilizando o *boilerplate* como base. As aplicações foram submetidas a métricas de coesão e acoplamento. Foi também analisada uma outra aplicação já existente que não utilizou o *boilerplate*, para fins de comparação. Os resultados dão indícios de que o *boilerplate* se mostrou eficaz nas métricas estabelecidas e colaborou para o desenvolvimento de aplicações Android com uma boa arquitetura, baixo acoplamento e alta coesão.

Palavras-chave: Padrões de Arquitetura. Android. Boilerplate.

Abstract

With the high demand for the development of Android applications, it is necessary to find the best ways to organize and structure the code of these applications, in a way that maximizes code reuse and maintainability, and the most suitable way of defining that is by using an appropriate architecture. Developers of the Android operating system have not defined or recommended any architecture for the development of Android applications. Moreover, the Android execution model and the libraries of the Android SDK provide guidelines for the development of applications that hinder loose coupling and high cohesion between the application's components. Because of these factors, it has always been a big challenge to define the ideal architecture for Android development. In this work, several architecture patterns used by the community were studied and their advantages and disadvantages in Android development were analyzed. Also the most modern architecture components available by Google were studied. Therefore, a boilerplate project that presents a software architecture that intends to help programmers on Android application development was created to help them create their applications with desirable code organization features. An application that uses this boilerplate was also created. For the boilerplate evaluation, programmer users were asked to create an application based on the boilerplate. The applications were analyzed with cohesion and coupling metrics and the users were asked to respond to a questionnaire. Another existent application that was not based on the boilerplate was also analyzed for comparison. The results indicate that the boilerplate was effective on the metrics used and helped the development of Android applications with a good architecture, high cohesion, and low coupling.

Keywords: Architecture Patterns. Android. Boilerplate.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE GRÁFICOS

LISTA DE EQUAÇÕES

LISTA DE SIGLAS

SDK - *Software Development Kit*

MVC - *Model View Controller*

MVVM - *Model View View Model*

AOT - *Ahead Of Time*

JIT - *Just In Time*

NDK - *Native Development Kit*

APK - *Android Package*

UI - *User Interaction*

XML - *Extended Markup Language*

HTML - *Hypertext Markup Language*

MVP - *Model View Presenter*

MVI - *Model View Intent*

VIPER - *View Interactor Presenter Entity Router*

NPE - *Null Pointer Exception*

JVM - *Java Virtual Machine*

HTTP - *Hypertext Transfer Protocol*

API - *Application Programming Interface*

TCC - *Tight Class Cohesion*

NDC - *Number of Direct Connections*

NP - *Number of Possible Connections*

CF - *Coupling Factor*

TC - Total de Classes

LISTA DE TABELAS

SUMÁRIO

1. Introdução

Android tornou-se o sistema operacional dominante no mercado [1]. Logo, é natural a demanda e interesse por desenvolver aplicações para essa plataforma.

O desenvolvimento de aplicações nativas para Android tem sido feito tradicionalmente utilizando-se a linguagem Java. Existe uma outra linguagem criada recentemente, chamada Kotlin, que também pode ser utilizada [2]. Ambas atualmente são suportadas oficialmente pelo Google [3], empresa que adquiriu o sistema operacional Android em 2005 e o desenvolve desde então. No Android, a máquina virtual Java foi reescrita para ser utilizada como base para as aplicações da plataforma.

Uma aplicação Android utiliza um modelo de programação baseado em atividades [4], que são classes centrais onde, através delas, todos os recursos do Android podem facilmente ser utilizados, incluindo iniciar uma outra atividade, manipular componentes gráficos, gerenciamento de áudio, sensores, notificações, etc.

Um dos problemas desse modelo de programação do Android é que desenvolvedores têm a tendência de criar toda a lógica de suas aplicações em volta dessas atividades [5]. Esse comportamento resulta em aplicações onde as classes de atividades são muito grandes e complexas, enquanto as demais classes possuem pouco código. Não há uma arquitetura bem definida e recomendada oficialmente [4]. Cabe ao programador decidir a melhor abordagem e, historicamente, os desenvolvedores têm buscado utilizar a arquitetura Model-View-Controller [6], ou MVC, onde as classes das aplicações ficam divididas em Modelo, Visão e Controlador. Desde o início do desenvolvimento de aplicações para Android, os desenvolvedores têm enfrentado dificuldades em organizar o código e separar a lógica nesses três componentes [5] [7].

Nos últimos anos, muitos desenvolvedores começaram a perceber as desvantagens do MVC no Android e têm se questionado se o MVC é realmente a melhor arquitetura para aplicações Android. Muitos desses desenvolvedores têm proposto adaptar outras arquiteturas bem estabelecidas em outros contextos de

programação para o contexto do desenvolvimento Android [7], porém há sempre uma barreira a ser enfrentada, seja por limitações do framework Android, ou pelo alto acoplamento do acesso aos recursos do sistema às classes de atividade.

Recentemente, o Google lançou o Android Jetpack [8]: uma coleção de componentes com diversas finalidades que facilitam a programação Android. Entre eles, existem vários componentes arquiteturais. Apesar de ainda não existir uma arquitetura recomendada, esses componentes arquiteturais já permitem que, a partir deles, seja possível construir uma arquitetura de forma mais concisa [9]. Embora o Android Jetpack forneça a maioria das peças do quebra-cabeça de uma arquitetura ideal para Android, algumas peças devem ser preenchidas pelos desenvolvedores.

A proposta deste Trabalho de Conclusão de Curso é apresentar um código *boilerplate* de uma arquitetura simples, porém completa, voltada ao desenvolvimento de aplicações Android, e que visa combinar as vantagens dos componentes disponíveis no Android Jetpack com as vantagens de um padrão de arquitetura adequado para aplicações móveis. Uma aplicação Android exemplo explorando o uso da arquitetura proposta também será fornecida para auxiliar na compreensão da arquitetura.

Código *boilerplate*, no contexto deste trabalho, consiste em todos os arquivos necessários para compor a base de uma aplicação Android qualquer. As classes e arquivos desse *boilerplate* serão organizados de forma tão específica quanto necessário para que a arquitetura apresentada possa ser compreendida, e tão genérica quanto possível para que desenvolvedores possam usá-los no desenvolvimento de qualquer aplicação Android.

O código *boilerplate* proposto pretende explorar a arquitetura Model-View-ViewModel (Modelo, Visão, Modelo de Visão), ou MVVM, baseada em conceitos de programação reativa, em conjunto com os componentes arquiteturais contidos no Android Jetpack, que também seguem o paradigma da programação reativa. Esse *boilerplate* será criado na linguagem Kotlin, que, embora seja nova e menos conhecida do que a linguagem Java, é a linguagem de programação que a comunidade de desenvolvedores Android prefere utilizar para desenvolvimento Android atualmente [10], o que permite que o *boilerplate* tenha uma aceitação maior pela comunidade. Conceitos básicos da linguagem Kotlin estão presentes no

apêndice 1 para que leitores pouco familiarizados com a linguagem compreendam todas as nuances do código *boilerplate* proposto.

1.1 Objetivos

1.1.1 Objetivo geral

Propor um código *boilerplate* para aplicações Android que apresente uma arquitetura simples, porém completa, com o objetivo de facilitar a organização, manutenção e escalabilidade de aplicações para essa plataforma.

1.1.2 Objetivos específicos

- Projetar e desenvolver um código *boilerplate* que reflita a arquitetura definida de forma tão específica quanto necessário para que ela possa ser compreendida, e tão genérica quanto possível para que desenvolvedores possam usá-la no desenvolvimento de qualquer aplicação Android;
- Desenvolver um programa que utiliza esse *boilerplate* como forma de exemplificar o uso;
- Disponibilizar à comunidade desenvolvedora o *boilerplate* desenvolvido nessa pesquisa.

1.2 Metodologia

Para a execução deste projeto, a pesquisa desenvolvida terá caráter exploratório e se caracterizará como pesquisa bibliográfica. Será feita uma pesquisa sobre modelos arquiteturais existentes de programação Android utilizados na atualidade. Serão também estudadas as ferramentas recomendadas pelo Google para facilitar a programação Android; e então serão avaliadas soluções arquiteturais que utilizam esses componentes. Serão também explorados conceitos da programação reativa que possam ser utilizados para implementar a comunicação entre componentes da arquitetura.

Após isso, será construído um *boilerplate* utilizando as melhores práticas arquiteturais identificadas, e será desenvolvido um aplicativo que utilizará esse

boilerplate como forma de exemplificação. Como avaliação da arquitetura e do *boilerplate*, o *boilerplate* apresentado será utilizado por programadores externos no desenvolvimento de aplicações e métricas estabelecidas na literatura são usadas para avaliar a qualidade das aplicações. Por fim, a arquitetura, o boilerplate, e toda a documentação que os acompanha serão disponibilizados para a comunidade de desenvolvedores Android.

2. Android

O Android é um sistema operacional baseado no núcleo Linux. Foi projetado para ser executado principalmente em dispositivos com telas sensível ao toque como smartphones e tablets [11]. Possui também interface específica para TVs, carros e relógios de pulso. Atualmente, 74.69% dos usuários de smartphones no mundo utilizam o sistema operacional Android [1]. Dados de 2017 também indicam que o sistema operacional está presente em cerca de 37.93% de todos os dispositivos da atualidade, superando até mesmo o sistema operacional Windows [12].

2.1 Ecossistema

Ao contrário de muitas aplicações *desktop*, um aplicativo Android não se inicia por um método `main()` [13] [14]. As aplicações Android consistem em 4 tipos de componentes: atividades, serviços, provedores de conteúdo e receptores de transmissão [11]. É a partir deles que uma aplicação é construída, e é através deles que o sistema pode acessar a aplicação. Cada componente existe como uma entidade independente e desempenha funções específicas. O principal desses componentes é a atividade. A atividade é o componente mais visível ao usuário e que permite ao desenvolvedor controlar o que será exibido na tela, podendo então iniciar sua aplicação a partir dela.

O modelo de programação de um aplicativo Android é baseado principalmente nesse conceito de atividade; existe uma classe do sistema Android chamada *Activity* para esse fim. O programador precisa criar uma classe que herda de *Activity* e sobrescrever métodos específicos que correspondem a estágios específicos do ciclo de vida da atividade. Nesses métodos é feita a customização do comportamento da atividade. O sistema Android se encarrega de invocar os métodos específicos dessa classe conforme a atividade passa pelos seus estágios do ciclo de vida.

O desenvolvimento de aplicações nativas para Android utiliza primariamente *bytecode* Java. A partir desse *bytecode*, o compilador do Android compila os

arquivos para um formato interno da máquina virtual que roda no sistema, chamado DEX [13] [15]. ART e sua predecessora Dalvik são as máquinas virtuais que o Android suporta para executar aplicativos a partir do arquivo DEX. ART é mais recente, porém mantém compatibilidade com a Dalvik [13] [15]. Dalvik utiliza a tecnologia de compilação JIT, ou *just-in-time*, que compila e otimiza o código da aplicação em tempo de execução. Já a ART introduziu a compilação AOT, ou *ahead-of-time* ('a frente do tempo', compila e otimiza antes de executar). Na hora da instalação, ART utiliza uma ferramenta chamada *dex2oat*. Com essa ferramenta é possível compilar e otimizar o DEX antes de executar a aplicação. Dessa forma, a execução acaba por ser mais rápida que o JIT utilizado pela Dalvik, por não precisar realizar a compilação em tempo de execução [3] [15].

Qualquer linguagem que gera o *bytecode* compatível com a máquina virtual Java do Android é suportada para o desenvolvimento de aplicações Android. Java e Kotlin são as oficiais e que possuem suporte do Google [3]. Graças ao Android NDK, Native Development Kit, também é possível desenvolver aplicações Android utilizando código C e C++.

O arquivo final da compilação de um aplicativo Android é o arquivo APK, sigla para Android Package. As ferramentas do sistema Android compilam o código, juntam os arquivos DEX e todos os seus dados e recursos e criam o arquivo APK a partir deles. Esse arquivo contém tudo que é necessário para o Android instalar o aplicativo no dispositivo. Após instalado, cada aplicativo é executado sob o sistema operacional de forma independente.

O Android possui um core Linux multiusuário onde cada aplicativo se comporta como um usuário [3] [11]. Por padrão, cada aplicativo recebe um ID de usuário único do sistema. O sistema define permissões de acesso para todos os arquivos em um aplicativo de modo que somente o ID de usuário atribuído àquele aplicativo pode acessá-los.

Cada aplicativo possui sua própria instância da máquina virtual, fazendo com que os aplicativos executem de forma isolada no sistema.

Utilizando essa abordagem, o Android consegue reduzir ao máximo os privilégios de um aplicativo, fazendo com que cada aplicativo só possua permissão para acessar os recursos que ele precisa e nada mais.

2.2 Ciclo de vida da aplicação

O manual do Android traz diversas informações, reunidas e resumidas nesta seção [14][16].

Uma das principais diferenças de uma aplicação *desktop* tradicional e uma aplicação Android é que o modo de interação nem sempre inicia no mesmo lugar, fazendo com que o usuário possa iniciar a aplicação sob diferentes pontos de entrada [13]. Por exemplo, se o usuário abrir uma aplicação de email a partir da tela inicial de aplicativos, ele será levado a uma tela que mostra uma lista de emails. Em contrapartida, caso ele esteja utilizando uma aplicação de redes sociais que abre seu aplicativo de emails, ele pode ser levado diretamente a uma tela de dentro desse aplicativo para escrever um email. Ou seja, uma aplicação Android pode ser iniciada de diferentes pontos, e o usuário pode ver diferentes telas quando a aplicação é iniciada.

A classe `Activity` foi projetada para facilitar essa forma de interação que acessa diretamente diferentes telas da aplicação dependendo do contexto atual. Quando uma aplicação invoca outra aplicação, ela invoca uma determinada atividade dessa aplicação, e não o aplicativo como um todo, fazendo com que cada atividade seja um ponto diferente de interação com o usuário do aplicativo. Implementa-se uma atividade como uma subclasse da classe `Activity`.

Uma atividade provê uma janela onde ficarão os elementos de interface gráfica de uma tela da aplicação. Ela costuma ocupar a tela inteira do dispositivo, mas também pode ocupar parte da tela ou até flutuar sobre outras telas. Normalmente, cada atividade implementa uma tela da aplicação. Por exemplo, um aplicativo pode ter duas telas: uma de preferências e outra de selecionar uma foto.

Cada atividade pode também possuir um ou mais fragmentos, representados no Android pela classe `Fragment`, que são componentes menores de elementos de interface gráfica, e que permitem uma melhor organização dos elementos gráficos.

Fragmentos são uma espécie de subatividade, introduzidos no Android inicialmente para lidar com dispositivos com telas grandes como os tablets, onde é possível inserir mais conteúdo que dispositivos com telas menores. Eles

representam o comportamento de parte de uma atividade. Uma atividade pode possuir vários fragmentos combinados e cada fragmento age de forma independente dentro de uma atividade.

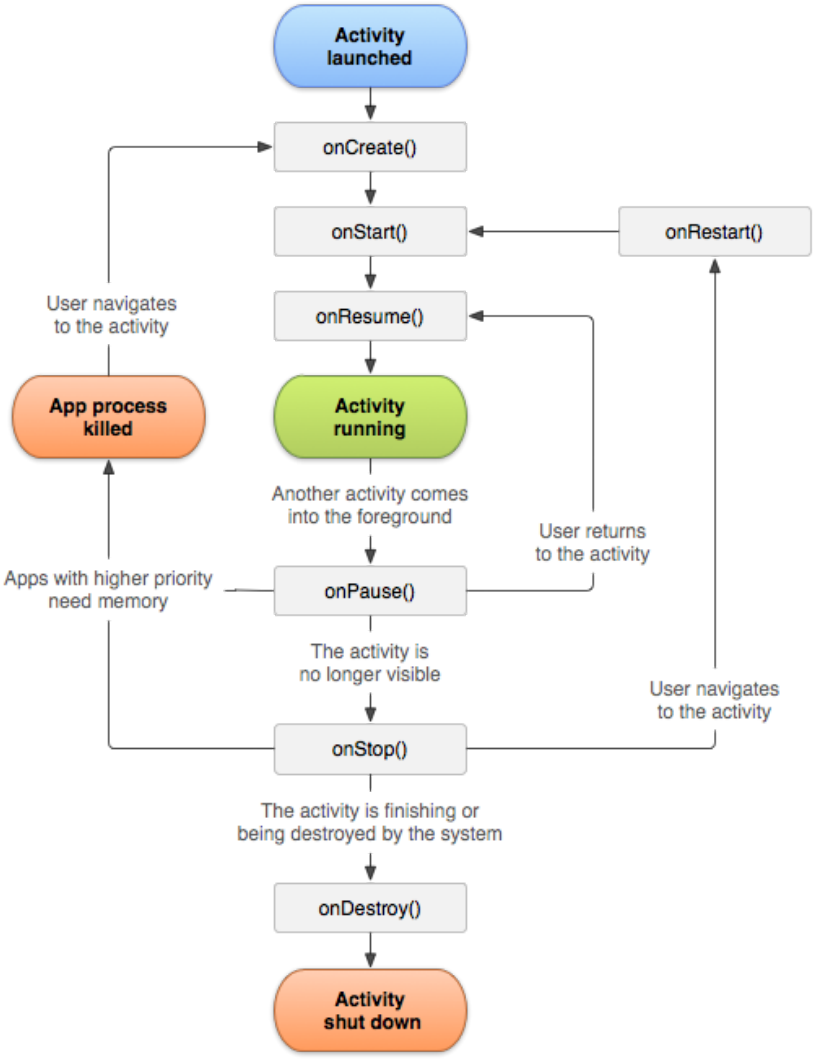
Fragmentos e atividades também podem ser chamados no Android de controladores de UI (do inglês *user interface*, ou interface do usuário). Ambos os componentes manipulam componentes gráficos na tela.

A maioria das aplicações Android possui múltiplas telas, o que geralmente significa múltiplos controladores de UI (múltiplas atividades, com possivelmente múltiplos fragmentos em cada uma). Em geral, existe uma atividade principal (representada por uma subclasse de `Activity`, que por convenção pode-se chamar de `MainActivity`), que é a primeira atividade a ser chamada quando o usuário abre o aplicativo da forma padrão, a partir do menu de aplicativos do dispositivo. A partir daí, a atividade pode chamar outras atividades para realizar diferentes ações. Por exemplo, uma aplicação de email pode ter como `MainActivity` uma tela que mostra todos os emails recebidos. Além do controlador de UI (atividade) que representa essa tela, a aplicação também pode ter outros controladores de UI que representam outras telas da aplicação, como escrever emails ou abrir um email em específico.

Enquanto o usuário percorre as diversas telas de sua aplicação, as instâncias de controladores de UI da aplicação passam por diversos estágios de seu ciclo de vida. Tanto a classe `Activity` quanto a classe `Fragment` proveem diversos métodos (*callbacks*) que permitem que o controlador de UI saiba quando cada mudança de estado ocorre; quando o sistema está criando, pausando ou continuando um controlador de UI, ou então destruindo o processo no qual o controlador de UI reside.

A figura 1 mostra os diferentes estágios no ciclo de vida de uma atividade. Para cada um desses estágios, existe um método da classe `Activity` que pode ser sobrescrito para customizar o comportamento da atividade.

Figura 1 - Os diferentes estados do ciclo de vida de uma atividade.



Fonte: Atividades | Android Developers [16].

Com os métodos de ciclo de vida é possível definir o comportamento de um controlador de UI quando um usuário entra ou reentra nele. Por exemplo, no caso de uma aplicação que transmite um vídeo via *streaming*, pode-se pausar o vídeo e terminar a conexão de rede quando o usuário muda para outra aplicação. Quando o usuário retornar para a aplicação, pode-se reiniciar a conexão de rede e permitir que o usuário continue assistindo o vídeo do mesmo ponto em que estava antes de mudar de aplicação. Em outras palavras, cada método de ciclo de vida permite dar um tratamento adequado a cada mudança de estado do controlador de UI.

2.3 Android SDK

O sistema operacional Android fornece ao desenvolvedor um framework para a criação de aplicativos para a plataforma chamado de Android SDK, onde SDK significa *Software Development Kit*, ou kit para desenvolvimento de software. Ao longo deste texto ele também será referido como “framework Android”, ou apenas “o framework”. Como visto anteriormente, uma aplicação Android pode consistir em uma combinação de quatro tipos de componentes: atividades, serviços, provedores de conteúdo e receptores de transmissão. Os aplicativos Android são então estruturados como uma combinação desses tipos de componentes atuando de forma independente. Por exemplo, uma aplicação pode ter uma atividade mostrando ao usuário seus elementos gráficos, ao mesmo tempo em que um serviço executa a compressão de uma imagem em segundo plano, enquanto um segundo serviço baixa novas imagens da internet.

O componente de serviço de uma aplicação Android consiste em rotinas que uma aplicação pode executar sem necessariamente estar sendo mostrada ao usuário no momento [17]. Por exemplo, uma aplicação de tocador de músicas pode continuar tocando uma música mesmo sem que a aplicação esteja visível para o usuário, ou seja, sem que exista qualquer atividade da aplicação em execução.

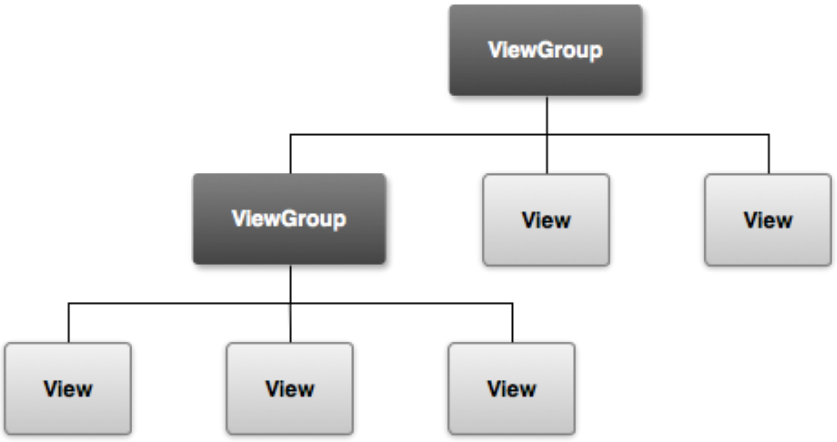
Provedores de conteúdo permitem que aplicações acessem dados públicos do sistema ou dados de outras aplicações [17]. Por exemplo, um aplicativo de mensagens pode utilizar um provedor de conteúdo para acessar a agenda do celular. Esses provedores encapsulam as informações necessárias para as aplicações acessarem recursos externos.

Os Receptores de transmissão são usados para tratar eventos de estado do sistema que um aplicativo pode receber, por exemplo, quando a bateria do celular está baixa, quando a tela for desligada ou quando houver uma captura de tela [17]. Também é possível gerar notificações através desse componente.

A interface gráfica do usuário é definida como uma hierarquia de objetos das classes `ViewGroup` e `View` [18]. Um `ViewGroup` define um contêiner para um ou

mais objetos `View`, enquanto cada objeto `View` representa um elemento gráfico na tela. A figura 2 ilustra essa hierarquia.

Figura 2 - A hierarquia de componentes gráficos (*Views*).



Fonte: Visão geral da IU | Android Developers [18].

Essa hierarquia de objetos View pode ser criada programaticamente nas atividades e fragmentos da aplicação, porém a forma mais fácil e efetiva de definir o layout da aplicação é através de arquivos XML [18]. Semelhante ao HTML em aplicações WEB, a interface gráfica do aplicativo pode ser codificada por meio de arquivos XML, onde nesse arquivo serão definidos a posição, o estilo e o layout dos elementos da tela. É possível definir arquivos XML que representem aspectos visuais de controladores de UI (tanto atividades quanto fragmentos). Também é possível declarar eventos que o controlador de UI atrelado a esse XML pode receber e tratá-los da maneira adequada. Uma atividade contém um XML inicial e também pode conter vários outros XML representando elementos gráficos específicos.

A troca de mensagens entre todos esses tipos de componentes do Android é realizada através de objetos *Intent* [4]. São objetos de mensagens utilizados para solicitar que algum componente realize uma ação específica.

É utilizando um *Intent* que, por exemplo, a aplicação pode iniciar outra atividade dela própria ou até mesmo uma atividade pertencente a outra aplicação [14]. Por exemplo, uma aplicação de galeria de fotos pode colocar dados a respeito

de uma foto em um *Intent* e solicitar ao sistema que inicie alguma atividade de outra aplicação responsável por postar a foto em alguma rede social.

Um dos principais elementos do ecossistema Android e que permeia por todos os componentes do framework é o objeto de contexto. O contexto é uma interface de informações globais do ambiente da aplicação. É uma classe abstrata cuja implementação é provida pelo sistema do Android [19]. Com ela é possível acessar recursos e classes específicos da aplicação, bem como realizar chamadas a operações a nível de aplicação, tal como iniciar atividades, receber e enviar *Intents*, etc.

Existem vários tipos de contexto, como o contexto da aplicação, o contexto de uma atividade específica, ou o contexto de um serviço específico [19]. O uso dos objetos de contexto em aplicativos Android é um dos maiores desafios para a organização e estruturação de aplicativos Android [20], como será descrito na seção 2.4. Como é através desse objeto que se acessa todos os recursos do sistema, ele precisa ser conhecido por diversos componentes da aplicação, fazendo com que manter o baixo acoplamento entre os componentes seja um grande desafio no desenvolvimento Android [21].

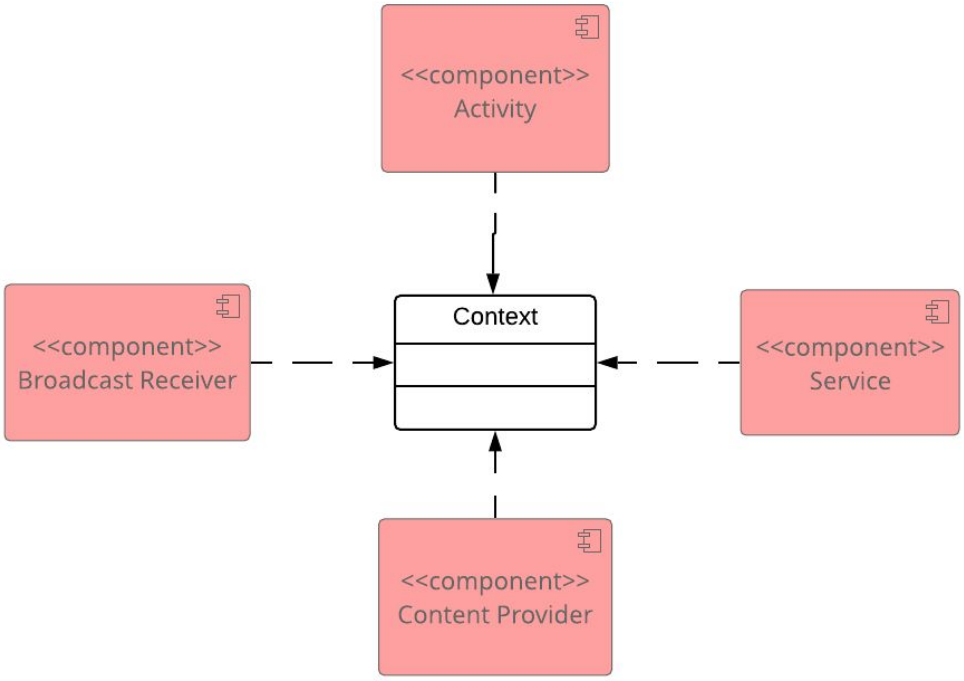
2.4 Desafios de Arquitetura

O sistema Android não recomenda uma arquitetura específica [4]. Muitos desenvolvedores, ao longo do tempo, tentaram adaptar os modelos de arquitetura existentes para o Android. O mais comum deles sempre foi o padrão MVC [6]. Porém, mesmo para esse padrão de arquitetura sempre houveram muitos questionamentos a respeito de como utilizá-lo. Por exemplo, os controladores de UI fazem parte da visão ou do controlador [21]? Controladores de UI podem conter elementos de interface gráfica, porém também podem ser o ponto de acesso a recursos do sistema operacional que estejam atrelados à lógica da aplicação.

Além disso, a forma como o Android trata contexto como sendo o meio para acessar recursos do sistema acaba dificultando a adesão do MVC; acaba fazendo com que todas as partes do MVC tenham ciência do contexto [20], o que resulta em

um grande acoplamento [7]. A figura abaixo ilustra os quatro componentes principais do Android SDK e suas dependências em relação a objetos de contexto.

Figura 3 - Os quatro principais componentes da arquitetura Android e sua dependência de *Context* mostrada.



Fonte: produção própria.

Como consequência dessa falta de padrão e desses desafios, muitos desenvolvedores, tanto iniciantes quanto experientes, têm a tendência de criar toda a lógica de suas aplicações em volta dos controladores de UI e das classes que possuem acesso a contextos [5]. Esse comportamento resulta em aplicações onde as classes dos controladores de UI são muito grandes e complexas e as demais classes possuem pouco código [21] [22].

Os desenvolvedores ao longo dos anos têm tentado utilizar variações do MVC na tentativa de contornar ao máximo esse problema do acoplamento gerado pelo contexto [7] [23]. A comunicação entre os componentes, principalmente entre controladores de UI, é outro grande desafio para a arquitetura. O Android força que qualquer objeto que seja passado de uma atividade a outra pelos objetos *Intent* tenha que implementar uma interface de serialização específica [17], o que faz com

que as atividades conheçam o conteúdo dos dados compartilhados pelos componentes da aplicação [21].

Por esses motivos, criar e manter a arquitetura de aplicativos Android é um grande desafio para desenvolvedores Android, especialmente os iniciantes [22]. Essa dificuldade é causada tanto pela falta de um direcionamento oficial por parte dos desenvolvedores do Android a respeito de como deve ser criada a arquitetura de um aplicativo Android, como também por todos os desafios impostos pelo ecossistema Android, como a organização do ciclo de vida da aplicação e das atividades, e a utilização do objeto de contexto em várias camadas da aplicação para realizar a comunicação com outros componentes do aplicativo e para utilizar recursos do sistema operacional. No próximo capítulo serão discutidos os principais padrões de arquitetura que desenvolvedores Android tentaram adaptar para o ecossistema Android, suas vantagens e limitações.

2.5 Testes unitários

Um dos grandes desafios dos padrões de arquitetura de desenvolvimento de softwares é permitir uma organização do código de forma que testes unitários possam ser criados para todos ou quase todos os componentes da aplicação [5]. Testes unitários são testes que avaliam cada unidade de lógica da aplicação separadamente e num ambiente controlado, permitindo a identificação de defeitos no código e auxiliando na manutenção da aplicação.

No desenvolvimento Android isto não é diferente [24]. Testes unitários podem ser criados no Android utilizando o framework JUnit. Testes JUnit são executados por uma máquina virtual Java no ambiente de desenvolvimento, ou seja, fora de um dispositivo Android real. Consequentemente, classes do framework Android (pacote `android`) não possuem funcionalidade real nesses testes unitários e seu comportamento precisa ser simulado. Existem, porém, bibliotecas que permitem a simulação do comportamento de classes do framework Android, permitindo a criação de testes unitários que as utilizem.

A criação de testes unitários pode ser tanto facilitada quanto dificultada pelo padrão de arquitetura escolhido para a aplicação [25]. Por isso, permitir uma alta testabilidade do software é mais um dos desafios desses padrões de arquitetura.

3. Padrões de arquitetura

Ao longo dos anos, foram criados os mais variados padrões de arquitetura para o desenvolvimento de softwares. Para cada ambiente onde eles foram utilizados, porém, pequenas adaptações precisaram ser feitas para que eles pudessem existir de forma coerente dentro do ambiente [25]. Muitas vezes essas adaptações eram consequência de limitações das próprias plataformas. No mundo da programação Android não foi diferente. O framework Android não implica em uma arquitetura específica, porém é necessário muitas adaptações específicas para se utilizar os padrões de arquitetura disponíveis, justamente pelo comportamento específico do ecossistema. É comum então que desenvolvedores, ao tentarem utilizar certo padrão arquitetural, precisem realizar ajustes no modelo da arquitetura, o que muitas vezes pode acabar tornando a arquitetura escolhida inviável para a plataforma. Neste capítulo serão discutidos os principais desses padrões, que problemas eles solucionam, e que problemas eles criam.

3.1 MVC

Inicialmente, a arquitetura que muitos programadores resolveram aplicar foi o MVC, de *Model-View-Controller*, ou Modelo-Visão-Controlador, como já foi descrito anteriormente. Esse padrão tem como objetivo separar as responsabilidades de qual componente se encarrega da lógica da aplicação e dos dados (Modelo), qual se encarrega de mostrar esses dados na interface gráfica para o usuário e coletar os eventos disparados pelo usuário na tela (Visão), e qual componente conecta os dois primeiros realizando a troca de mensagens, tratamento de eventos e o fluxo dos dados (Controlador). O MVC é um padrão de arquitetura muito bem difundido no universo Java, em diversos ambientes [26].

Arquitetos de software já realizaram muitas tentativas de adaptar o MVC para diferentes plataformas, e até mesmo em aplicações Java tradicionais há variação nas especificações das responsabilidades de cada um dos componentes [26] e de como funciona a comunicação entre eles. No Android não é diferente. Mesmo após anos de desenvolvimento de aplicativos Android, ainda há uma grande divergência

entre membros da comunidade Android a respeito de como conceitos como atividades e fragmentos se encaixam em uma arquitetura MVC [7] [22]. Pode-se ver uma atividade como um controlador que trata os eventos da visão, que no caso seriam os arquivos XML declarativos, e que ao mesmo tempo atualiza o modelo? Por outro lado, pode-se ver uma atividade como parte da visão, onde então o controlador seria um artefato isolado das classes do framework Android [21]?

Nenhuma dessas abordagens foi totalmente aceita pela comunidade até o momento, e muitos desenvolvedores então buscaram outras arquiteturas que pudessem se encaixar melhor nos conceitos já concebidos pelo framework Android [7] [22]. Independentemente da abordagem escolhida, o framework Android não fornece abstrações prontas para a criação dos componentes do MVC. Consequentemente, os componentes do MVC precisam ser criados pelo desenvolvedor, o que adiciona mais complexidade ao código da aplicação. Componentes do Android, como atividades e serviços, farão parte do MVC, mas precisarão ser inseridos dentro dos componentes do MVC criados. A necessidade dessa abordagem pode ser vista como uma das desvantagens do padrão MVC no desenvolvimento Android.

3.2 MVP

O MVP, de *Model-View-Presenter*, ou Modelo-Visão-Apresentador, é um padrão de arquitetura não muito distante do MVC clássico, porém introduz mudanças no papel do controlador, chamando-o de *presenter*, ou apresentador.

O *presenter* do MVP é responsável por definir toda a lógica de apresentação dos dados (daí o nome de apresentador), fazendo com que a visão se torne simplória, encarregada somente de realizar exatamente o que foi especificado no *presenter* sobre a disposição e características dos componentes a serem exibidos [25]. Nessa *view* simples do MVP, o tratamento de ações do usuário geralmente consiste apenas na chamada de uma função do *presenter*, delegando o tratamento dos eventos totalmente ao *presenter*.

Da mesma forma que ocorre com o MVC, no MVP não existe clareza de como organizar e separar o código em visão, modelo e apresentação no contexto do

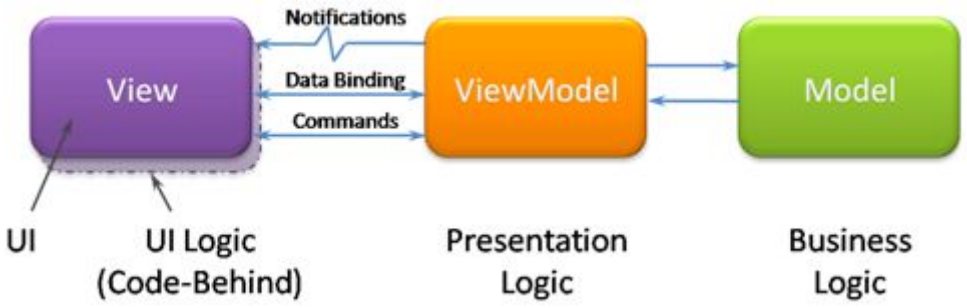
desenvolvimento Android [23]. Isso ocorre porque não existem componentes bem definidos dentro do framework Android que realizam exclusivamente o papel de visão, modelo ou apresentação. Porém, por causa das características do *presenter*, faz sentido ver as atividades da aplicação como *presenters*, onde a *view* seria as definições de layout nos arquivos XML e subclasses de View.

Realmente, a *Activity* é responsável pela exibição do conteúdo da tela e, por possuir um contexto, também é capaz de definir bem como os elementos serão mostrados. A comunidade tem ultimamente visto o MVP como um padrão aceitável para o ecossistema Android. Uma das vantagens do MVP é permitir uma separação entre a lógica de visão e os componentes gráficos do Android [25]. Dessa forma, é possível criar testes unitários que avaliem a lógica da visão sem precisar simular ou interagir com os componentes gráficos do Android. Por outro lado, existe ainda o risco de muito código ser colocado dentro das atividades, prejudicando a organização do código e dificultando a manutenção do aplicativo.

3.3 MVVM

O MVVM, de *Model-View-ViewModel*, ou Modelo-Visão-Modelo da Visão, é um padrão de arquitetura apresentado por John Gossman para aplicações da Microsoft que tem como objetivo automatizar as mudanças ocorridas no modelo para apresentá-las na visão [27]. O modelo e visão são semelhantes ao MVC e MVP. O *ViewModel* ou Modelo da Visão, diferentemente do *Presenter*, possui um elemento chamado *Binder*, ou vinculador, responsável por justamente estabelecer um vínculo entre a visão e a lógica de negócio automatizando as suas mudanças de estado. A figura 4 mostra ilustra esse comportamento.

Figura 4 - Demonstração do fluxo de dados da arquitetura MVVM.



Fonte: Entendendo o Pattern Model View ViewModel MVVM [28].

A principal diferença entre o *Presenter* e o *ViewModel* é que o *Presenter* possui referências para visão, enquanto que o *ViewModel* não possui. Ao invés disso, a visão diretamente vincula-se a propriedades do *ViewModel* e recebe notificações quando elas são atualizadas. Essas interações são ilustradas na figura acima.

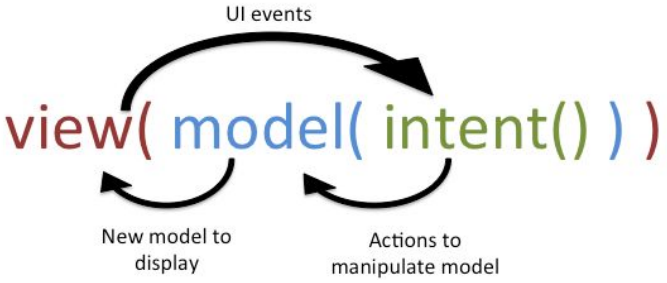
Uma das vantagens do MVVM é a separação total entre a lógica do negócio e a apresentação da interface gráfica, permitindo que a visão seja modificada sem que o modelo seja afetado, considerando que a interface com o ViewModel continue a mesma. Outra vantagem desse padrão é permitir a criação simplificada de testes unitários do modelo e do ViewModel sem usar a visão [29].

3.4 MVI

No padrão de arquitetura MVI, de *Model-View-Intent*, ou Modelo-Visão-Intenção, a principal característica é uma mudança no fluxo de dados da aplicação. Ele ocorre de forma unidirecional, ao contrário do MVC e do MVP, nos quais o fluxo de dados ocorre de forma bidirecional. O modelo e a visão se comportam de forma semelhante ao modelo e visão do MVC. A intenção conceitualmente representa a vontade do usuário ou da aplicação [30]. É através dela que são captados os eventos de mudança dos objetos do modelo da aplicação, isso é, o estado, que se traduzem em ações. E assim, essa mudança de estado

através de ações se reflete em uma mudança na visão, uma atualização do que está sendo mostrado na tela dos elementos.

Figura 5 - Demonstração prática da atualização da *view* através de eventos de *UI* recebidos pelo *intent* assim atualizando o *model*.



Fonte: Model-View-Intent on Android [30].

O fluxo de dados no MVI começa com as ações do usuário, que são enviadas da *view* para o *intent* através de eventos de interface do usuário. Esse *intent* é o elemento que representa uma intenção do usuário. Como resposta à ação do usuário, o *intent* realiza ações, onde cada ação disparada pelo *intent* irá modificar um ou mais objetos do modelo, alterando assim o estado da aplicação. Quando o modelo é alterado, há um mecanismo que altera automaticamente a visão, mostrando na tela a alteração visível dos elementos gráficos em tempo real.

3.5 VIPER

O VIPER, de *View-Interactor-Presenter-Entity-Router*, ou Visão-Interagente-Apresentador-Entidade-Roteador, é um padrão de arquitetura apresentado em 2014 pela Mutual Mobile e baseado no padrão *Clean Architecture* de Robert C. Martin [31] [32]. De acordo com o *Clean Architecture*, a aplicação é dividida em casos de uso, onde cada caso de uso geralmente representa uma tela da aplicação. No VIPER, cada caso de uso possui seus próprios componentes de visão, interagente, apresentador, entidades e roteador.

Roteadores são usados para definir a navegação da aplicação, e são responsáveis por iniciar o caso de uso e realizar a transição para algum outro caso de uso. A visão define aspectos específicos da interface gráfica do caso de uso e

não deve possuir nenhuma lógica, como ocorre também no MVP. A lógica de apresentação é toda gerenciada pelo apresentador, que se comunica diretamente com a visão para informá-la o que deve ser apresentado. As entidades do caso de uso são os artefatos básicos do domínio da aplicação, que costumam ser as classes base do modelo da aplicação. O interagente, por fim, contém a lógica e as regras de negócio do caso de uso, sendo responsável por interagir com o apresentador e manusear as entidades [31].

Uma das vantagens do VIPER é a separação lógica da aplicação em casos de uso, ou telas, e a quebra do código de cada caso de uso em componentes pequenos e com responsabilidades bem definidas. Umas das principais desvantagens, porém, está na dificuldade em se organizar código que é compartilhado entre diferentes casos de uso. Geralmente código compartilhado acaba sendo colocado fora dos componentes dos VIPER, sujeito a má organização e estruturação. Além disso, dependendo do domínio da aplicação, pode ser difícil organizar o código da aplicação em componentes tão específicos.

4. Android Jetpack

Visando facilitar o desenvolvimento de aplicações Android e incentivar os desenvolvedores a utilizar boas práticas de programação, recentemente o Google lançou o Android Jetpack [8], que consiste em um conjunto de componentes de software que é externo ao Android SDK, complementando-o, com o objetivo de utilizar as novas funcionalidades da plataforma sem perder a compatibilidade com versões antigas do sistema. Entre os componentes contidos no Android Jetpack estão os Componentes de Arquitetura, criados com o intuito de auxiliar desenvolvedores a definir uma arquitetura para suas aplicações. Ao longo deste capítulo serão descritas as principais características desses Componentes de Arquitetura, bem como suas vantagens e desvantagens. Por fim, serão analisados os aspectos mais importantes do Android Jetpack e dos padrões de arquitetura vistos no capítulo 3, de forma a permitir que uma arquitetura ideal para o desenvolvimento Android seja elaborada.

4.1 Componentes de Arquitetura

Os componentes de arquitetura do Android Jetpack fornecem ferramentas para auxiliar na organização e estruturação do código da aplicação. Além disso, esses Componentes de Arquitetura também tratam internamente diversos aspectos intrínsecos do ecossistema Android [8]. Ao realizar esse tratamento internamente, esses componentes permitem que os desenvolvedores não precisem se preocupar em fazer com que a arquitetura da aplicação tenha que lidar com eles. A arquitetura da aplicação pode então ser focada nos aspectos mais inerentes da lógica própria da aplicação.

Por ser criado e recomendado pelo próprio Google, os componentes do Android Jetpack se mostram bastante promissores, e mostram um caminho de convergência para a questão dos padrões de arquitetura para o desenvolvimento Android [9].

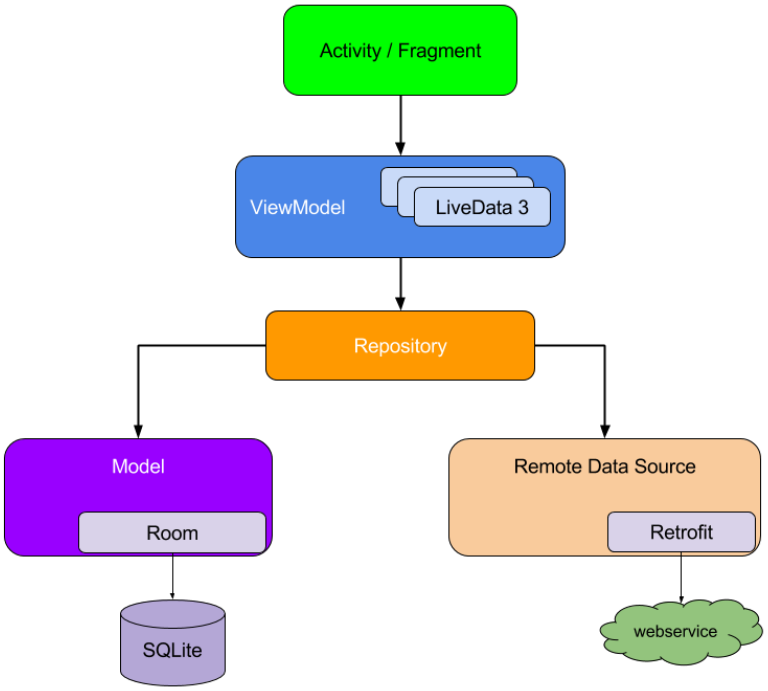
Conceitualmente, esses Componentes de Arquitetura do Jetpack se aproximam bastante dos conceitos do padrão MVVM como visto anteriormente. De

fato, pode-se ver os Componentes de Arquitetura do Jetpack como uma versão modificada do MVVM.

4.2 Implementação

Na implementação de uma aplicação utilizando os Componentes de Arquitetura do Android Jetpack, os componentes da arquitetura da aplicação ficam separados conforme ilustrado na figura 6.

Figura 6 - Diagrama da forma recomendada pela Google de utilização do Android Jetpack na arquitetura de aplicações.



Fonte: Guia para a arquitetura do app | Android Developers [33].

Os controladores de UI da aplicação (objetos *Activity* e *Fragment*) se comunicam com o componente *ViewModel*, informando-o a respeito das ações do usuário. Esse componente *ViewModel* funciona de forma idêntica ao componente *ViewModel* do padrão de arquitetura MVVM, por isso, pode-se dizer que os Componentes de Arquitetura do Android Jetpack seguem uma versão modificada da

arquitetura MVVM. Isso significa que a *View* recebe notificações quando as propriedades do *ViewModel* são modificadas, e, portanto, o *ViewModel* não se comunica diretamente com a *View*. No *ViewModel* do Android Jetpack, as funções do *Binder* do padrão MVVM são implementadas por uma classe chamada *LiveData*. Cada instância de *LiveData* encapsula um dado relevante para a View e pode ser observada pela View através do padrão *Observer*. Dessa forma, quando as propriedades do *ViewModel* são modificadas, as propriedades da *View* também o são, de forma reativa [34].

4.3 Vantagens e Desvantagens

A utilização dos Componentes de Arquitetura do Android Jetpack para a criação da arquitetura de uma aplicação Android possui algumas vantagens e desvantagens em comparação com os padrões de arquitetura descritos no capítulo 3.

Uma das principais vantagens do uso dos Componentes de Arquitetura do Android Jetpack é que eles possuem integração automática com componentes do próprio framework Android [35].

Ao invés de o desenvolvedor criar toda a funcionalidade do componente ViewModel, os Componentes de Arquitetura já fornecem uma classe ViewModel pronta para ser utilizada. Desenvolvedores podem então criar subclasses de ViewModel para implementar seus objetos ViewModel específicos.

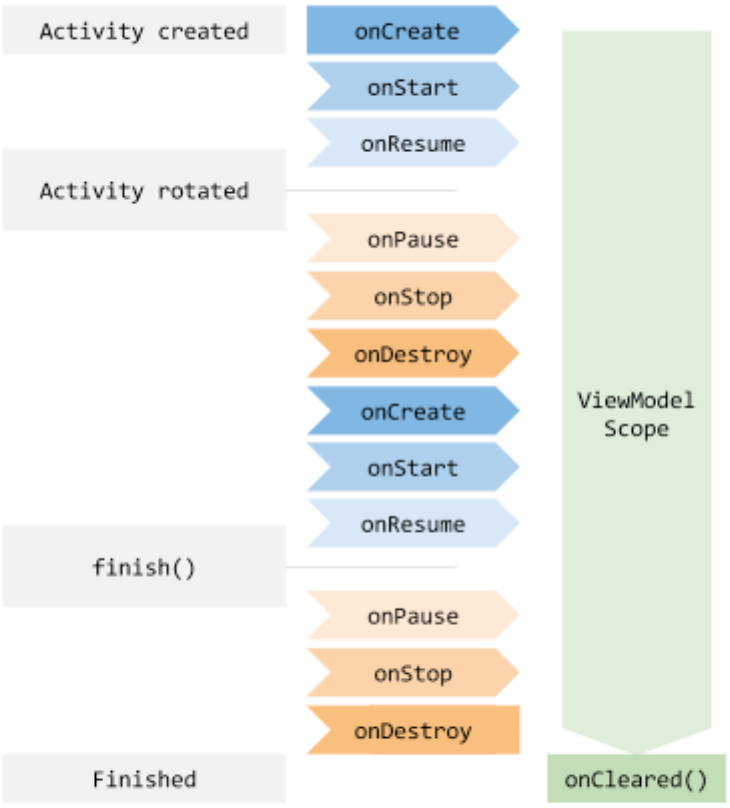
Também não é necessário implementar o componente Binder do MVVM, pois os Componentes de Arquitetura também fornecem a classe LiveData para cumprir esse papel.

Dessa forma, aderir à versão modificada do MVVM conforme especificado pelos Componentes de Arquitetura do Android Jetpack é mais simples do que implementar todos os arcabouços da arquitetura manualmente, como ocorre nos outros padrões de arquitetura descritos no capítulo 3.

Outra vantagem dos Componentes de Arquitetura é que eles fornecem uma forma simplificada de lidar com aspectos do framework Android, como contexto e ciclo de vida de atividades. A figura 7 mostra que o escopo de um ViewModel não

muda com os diversos ciclos de vida dos controladores de UI, representado aqui como atividade.

Figura 7 - Os diferentes ciclos de vida de uma atividade e o escopo de um *ViewModel*.



Fonte: ViewModel Overview | Android Developers [36].

Dessa forma, utilizando o ViewModel dos Componentes de Arquitetura, os desenvolvedores não precisam lidar com o ciclo de vida dos controladores de UI, ao contrário do que acontece quando se utiliza outro padrão de arquitetura. Não precisar lidar com o ciclo de vida desses componentes do framework Android reduz a complexidade da comunicação entre a View e o ViewModel [35].

Há também desvantagens na utilização dos Componentes de Arquitetura do Android Jetpack em relação aos padrões de arquitetura descritos no capítulo 3. Embora esses componentes possam ser usados para definir a arquitetura de uma aplicação Android, o Android Jetpack apenas fornece esses componentes como peças separadas da arquitetura, ao invés de definir exatamente como essas peças se juntam para formar a arquitetura. Não há detalhes na documentação sobre como

implementar o componente Repository da arquitetura, e não há clareza a respeito de como implementar a comunicação entre ViewModel e Repository e vice-versa.

Além disso, a documentação do Android Jetpack incentiva o vínculo de propriedades do modelo diretamente na View, criando um acoplamento implícito entre os objetos do modelo e a View [36], o que fere conceitos de baixo acoplamento e alta coesão. Por fim, alguns componentes gráficos do framework Android, como a classe RecyclerView, possuem uma forma diferente de se comunicarem com a lógica do negócio [37]. Para utilizar esses componentes gráficos na arquitetura é necessário implementar uma integração customizada entre esses componentes e o ViewModel.

4.4 Programação Reativa

O paradigma de programação reativa é um paradigma declarativo que descreve relações de causa e efeito entre instâncias, componentes, ou quaisquer elementos de uma aplicação [38]. Nos últimos anos, muitos desenvolvedores têm buscado trazer esses conceitos para o desenvolvimento de aplicações Android [39]. Entre os principais objetivos desse movimento estão facilitar a estruturação do código e permitir uma representação mais intuitiva e legível da lógica da comunicação entre diferentes componentes da aplicação.

De acordo com esse paradigma, um procedimento pode ser escrito como uma sequência de causas e efeitos [38]. A forma como o código é escrito faz com que a relação entre os componentes e o fluxo dos dados fiquem claros, porém não expõe como os tratamentos dos dados ocorre internamente, nem especifica como a informação trafega entre os componentes. A expressão `a := b + c`, por exemplo, que, na programação imperativa, representa apenas a atribuição de uma soma, na programação reativa representa um vínculo implícito entre as variáveis `a`, `b` e `c`. Mais especificamente, representa que toda modificação no valor de `b` ou `c` automaticamente atualiza o valor de `a` com o novo resultado da soma.

O padrão de arquitetura MVVM se baseia um pouco nesses conceitos da programação reativa. Ao vincular um campo de texto na View a uma string no ViewModel, o padrão MVVM tenta representar uma relação de causa e efeito entre

esses componentes, fazendo com o que o texto que aparece no campo de texto da View seja automaticamente atualizado quando a string do ViewModel é modificada.

Ao fornecer componentes de arquitetura baseados em conceitos da programação reativa, como os do padrão MVVM, os desenvolvedores do Android Jetpack estão incentivando os desenvolvedores Android a seguir a mentalidade desse paradigma em suas aplicações. Diversos outros frameworks de desenvolvimento de aplicações modernos têm explorado esses mesmos conceitos da programação reativa, demonstrando um crescimento no interesse por padrões de arquitetura que se baseiam neles [39].

4.5 Considerações

Ao longo dos capítulos 3 e 4 foram descritos diversos padrões de arquitetura usados no desenvolvimento de aplicações Android. Com base nas vantagens e desvantagens de cada abordagem, pode-se identificar uma arquitetura simples, porém completa, que possa auxiliar no desenvolvimento de novas aplicações Android, suprindo as limitações da plataforma e de sua documentação, ao mesmo tempo que traz os principais benefícios que os desenvolvedores Android esperam de uma arquitetura.

5. Desenvolvimento

Neste capítulo é descrita em detalhes a arquitetura proposta para o desenvolvimento de aplicações Android. A proposta deste Trabalho de Conclusão de Curso é propor uma arquitetura simples, porém completa, voltada ao desenvolvimento de aplicações Android, e que visa combinar as vantagens dos componentes disponíveis no Android Jetpack com as vantagens de um padrão de arquitetura adequado para aplicações móveis. Essa arquitetura foi proposta na forma de um *boilerplate* que demonstra como organizar o código da aplicação seguindo a arquitetura. O *boilerplate* é também acompanhado de diagramas que ilustram os componentes da arquitetura. Uma aplicação Android exemplo explorando o uso da arquitetura proposta também é fornecida para auxiliar na compreensão da arquitetura.

Para esse *boilerplate*, foi adotado o princípio de aplicativos de atividade única (*Single Activity*), conforme recomendação dos desenvolvedores do sistema Android [8]. Ou seja, o aplicativo será composto por uma única atividade e diversos fragmentos. Com esse princípio é possível utilizar um componente chamado *Navigation*.

5.1 Boilerplate

Boilerplate vem de uma expressão em inglês, originalmente do ramo jornalístico, onde *boilerplate* significa chapa de ebulição [51]. É muito utilizado nas esteiras para a preparação de textos para serem impressos em jornais em massa. No ramo da programação, esse termo ficou conhecido como qualquer trecho de código que pode ser usado como modelo inicial para construção de aplicações.

Código *boilerplate*, no contexto deste trabalho, consiste em todos os arquivos, trechos de código e estrutura de diretório necessários para compor a base de uma aplicação Android qualquer. Ele fornece, além da base de um projeto Android, um código reutilizável de forma que o desenvolvedor não precise construir a aplicação do zero. Esse *boilerplate* responde diversas dúvidas quanto a utilização e aplicação dos conceitos de arquitetura discorridos nos capítulos anteriores.

O código desenvolvido foi bem documentado explicando como ele pode ser utilizado para construir a base de qualquer aplicação dessa plataforma. Ele foi construído de tal forma que seja possível demonstrar o fluxo de dados de forma clara e concisa.

Para o *boilerplate*, foram utilizadas as mais recentes recomendações do Google para a arquitetura de aplicações Android [35], como a utilização dos Componentes de Arquitetura do Android Jetpack. Além de serem a recomendação do Google, esses componentes são uma escolha interessante por possuírem uma integração natural com o framework Android, o que reduz o esforço dos desenvolvedores em utilizá-los no ambiente Android. A arquitetura proposta também será baseada no padrão de arquitetura MVVM, que se mostrou uma arquitetura que se encaixa muito bem com os componentes recomendados contidos no Android Jetpack, combinados com conceitos importantes da programação reativa; tudo isso codificado na linguagem de programação Kotlin. É possível o desenvolvedor utilizar Java partindo do código *boilerplate* desenvolvido, graças à interoperabilidade entre ambas as linguagens.

A arquitetura proposta, embora se baseie nos Componentes de Arquitetura do Android Jetpack, em alguns pontos fará um uso desses componentes de forma ligeiramente diferente da utilizada na documentação do Android Jetpack. Esses casos serão enfatizados no texto.

As classes e arquivos desse *boilerplate* foram organizados de forma tão específica quanto necessário para que a arquitetura apresentada possa ser compreendida, e tão genérica quanto possível para que desenvolvedores possam usá-los no desenvolvimento de qualquer aplicação Android.

5.2 Arquitetura

5.2.1 Arquitetura proposta

A arquitetura desenvolvida é composta principalmente por 3 pacotes: modelo (*Model*), visão (*View*) e modelo da visão (*ViewModel*).

O pacote modelo possui os seguintes componentes:

- repositório (*Repository*);
- fontes de dados (*DataSources*);
- lógica de negócio (*BusinessLogic*);
- e entidades (*Entities*).

A visão foi dividida entre componentes declarativos (arquivos XML no diretório `res`) e componentes em Kotlin. Os componentes Kotlin são os seguintes:

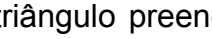
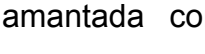
- fragmentos (*Fragment*);
- atividade principal (*MainActivity*);
- auxiliares de *data binding* (*BindAdapter*);
- e dados da visão (*ViewData*).

A figura 8 mostra um diagrama completo da arquitetura.

Cada um dos pacotes e componentes do diagrama são descritos em detalhes nas seções a seguir. As caixas no diagrama possuem significado de acordo com a sua borda:

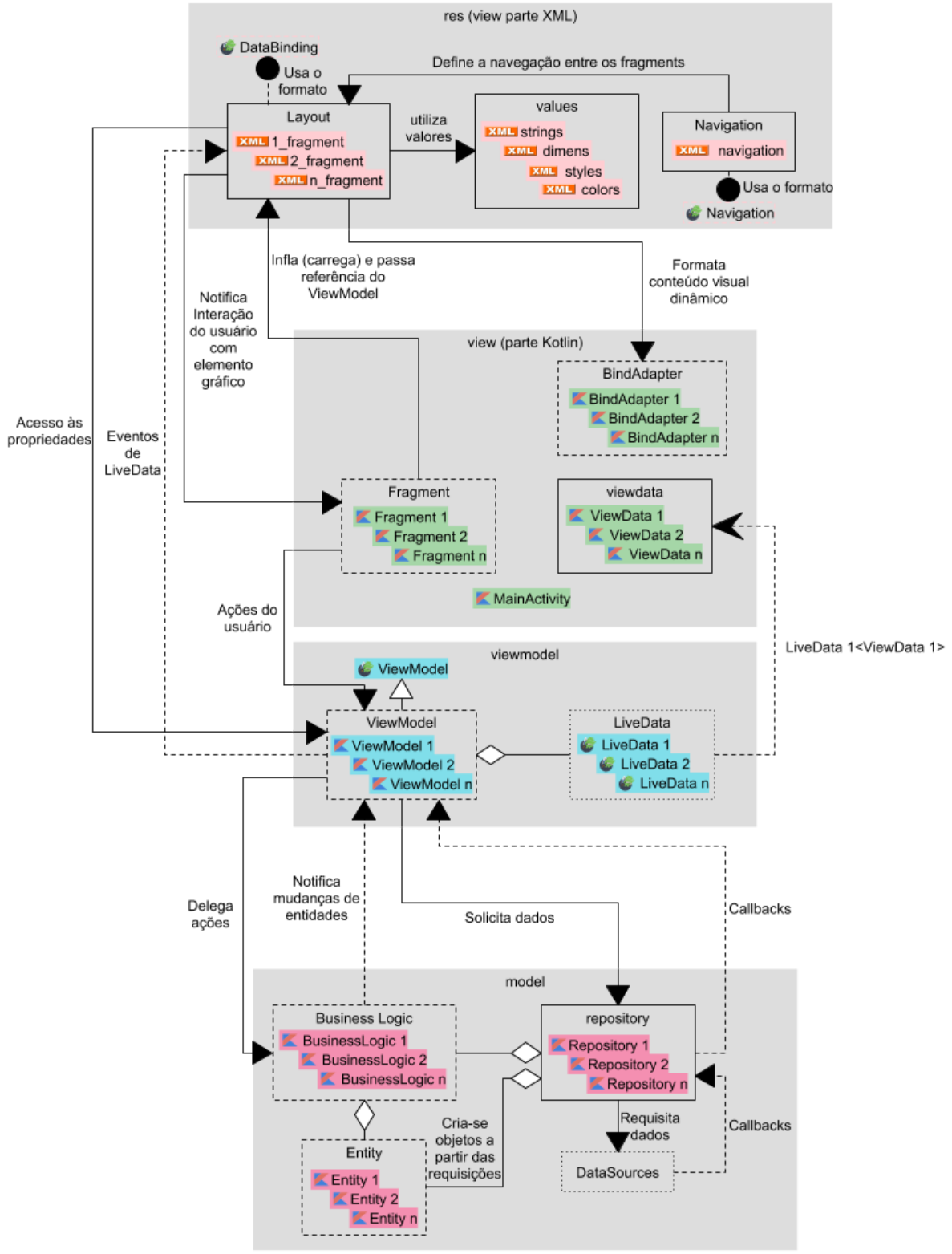
- Borda contínua/sem borda com fundo cinza: diretórios/pacotes para código Kotlin, e diretórios para arquivos XML;
- Borda tracejada: coleções de classes para código Kotlin, coleção de arquivos XML para arquivos XML;
- Borda pontilhada: dados internos das classes expostos.

As arestas entre elementos também possuem significado de acordo com o formato da aresta e ponta:

- Ponta triângulo preenchido com aresta contínua (): referência de um arquivo para o outro, como chamadas de métodos ou acesso a propriedades;
- Ponta triângulo preenchido com aresta tracejada (): retorno de callbacks;
- Seta diamantada com aresta contínua (): composição entre classes ou coleções de classes. A classe ou coleção que origina a seta compõe a classe ou coleção que recebe a seta;

- Ponta pontiaguda com aresta tracejada ( ): a classe ou coleção que origina a seta está parametrizando a outra;
- Ponta triângulo não preenchido com aresta contínua (): de uma coleção para uma classe significa que todas as classes da coleção herdam da classe.

Figura 8 - Diagrama completo do *boilerplate* produzido.



Fonte: produção própria.

Basicamente, o fluxo de dados na aplicação é o seguinte:

1. A *View* recebe ações do usuário e envia comandos ao *ViewModel*;
2. *ViewModel* solicita dados ao *Repository* e/ou delega ações à *BusinessLogic*;
3. *Repository* busca dados das fontes de dados (banco de dados, *webservices*, etc), cria entidades e retorna elas ao *ViewModel* em formato de *callbacks*;
4. *ViewModel* atualiza suas estruturas internas, chamadas de *LiveData*, com as novas informações a serem mostradas para o usuário;
5. *View* recebe notificações de que os dados do *ViewModel* mudaram, e assim atualiza a tela.

No diagrama, todos os lugares onde a arquitetura utiliza algum componente dos Componentes de Arquitetura do Android Jetpack foram destacados com o símbolo do Android Jetpack. Todas as classes de *ViewModel* herdam da classe `ViewModel` do Android Jetpack. As classes de *ViewModel* possuem como atributos `LiveData` que são classes do Android Jetpack que encapsulam objetos de *ViewData*. Na parte XML, os layouts contam com referências diretas ao ViewModel graças ao *DataBinding*, e o arquivo `navigation` é utilizado pelo *Navigation* para definir a navegação.

Esse fluxo conta com o auxílio de alguns Componentes de Arquitetura presentes no Android Jetpack e, de forma geral, implementa o padrão MVVM. As setas contínuas da figura mostram referências diretas. Os controladores de UI, por exemplo, representados na figura pelo componente *View*, possuem uma referência aos objetos *ViewModel* com os quais eles precisam se conectar. Cada *ViewModel* também possui uma referência a um ou mais objetos *Repository*, para os quais ele pede todos os dados do modelo que ele precisa repassar para a *View*. Cada *Repository*, por sua vez, manuseia os dados do modelo e acessa os diferentes *Data Sources* da aplicação (webservices, banco de dados, etc) para buscar, criar, ou modificar os dados do modelo armazenados neles.

O fluxo de retorno dos dados do modelo para a *View* ocorre de forma implícita e indireta, usufruindo de conceitos da programação reativa em alguns

pontos e utilizando o padrão Observer. Esse fluxo de retorno dos dados é mostrado através das setas pontilhadas na figura.

Os *ViewModel* fornecem *callbacks* que são chamados pelos *Repository* quando as informações solicitadas se tornam disponíveis. Os *ViewModel*, por sua vez, atualizam suas propriedades internas, e de forma reativa a *View* é informada automaticamente dessas mudanças, podendo assim atualizar a interface gráfica.

As seções seguintes irão tratar com mais detalhes cada parte do diagrama acima.

5.2.2 View

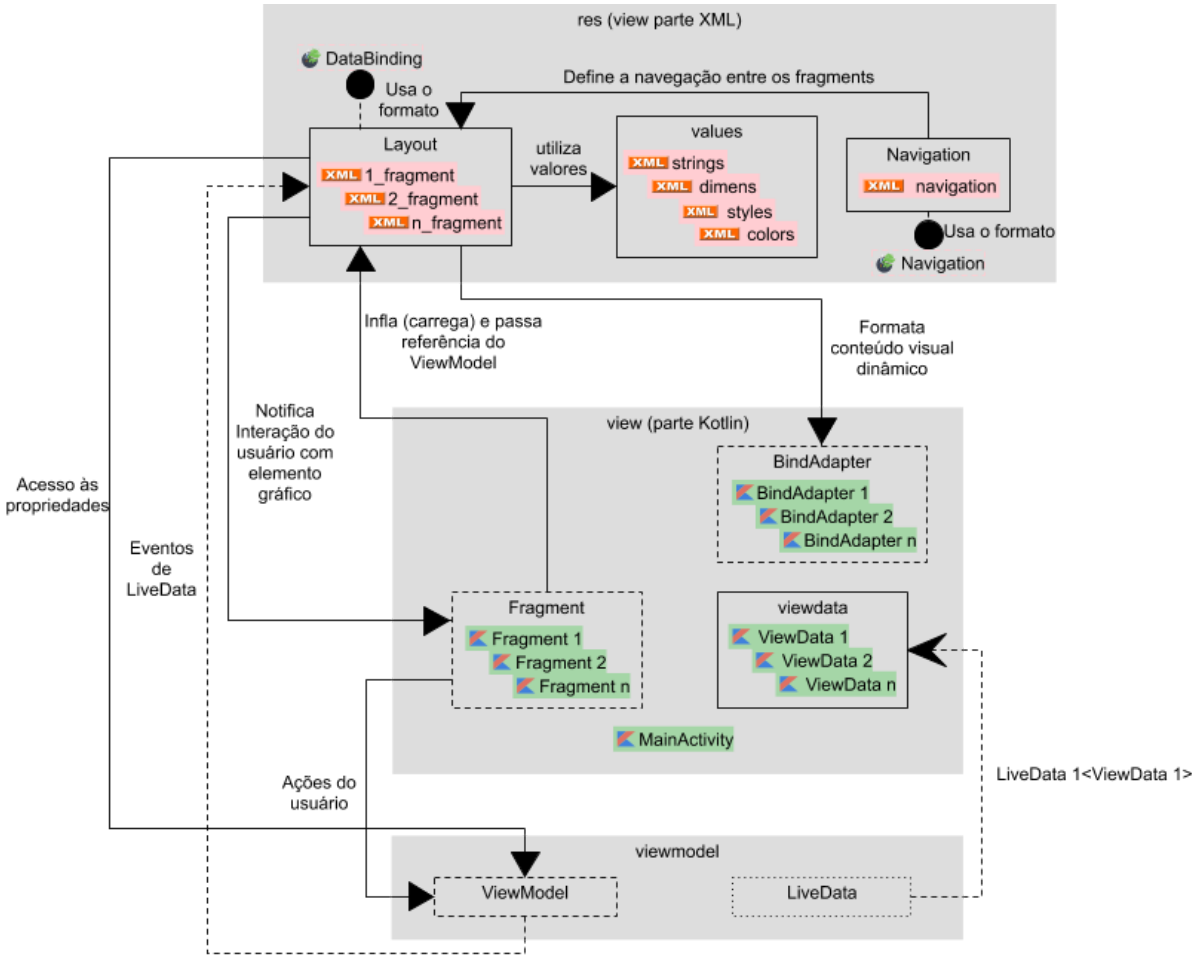
A figura 9 mostra as partes que compõem a *View*. As comunicações externas e internas serão discutidas ao longo desta seção.

Nota-se no diagrama um pacote (`view`) e um diretório (`res`). No Android, o componente *View* consiste em duas partes distintas: classes Kotlin e arquivos declarativos no formato XML.

As classes Kotlin são primariamente controladores de UI. Além disso, as classes Kotlin da View também incluem *BindAdapters* e o pacote que contém classes *ViewData*. Todas essas partes da *View* do *boilerplate* irão ser discutidas adiante.

Os arquivos XML estão localizados em uma estrutura de diretórios localizados na pasta `res` do projeto Android. Contém arquivos de layouts onde são declarados elementos gráficos. Os arquivos no diretório v*alues* são arquivos que definem constantes do programa utilizadas nos layouts, enquanto o arquivo `navigation` que define a navegação entre os fragmentos. Há ainda outros arquivos, discutidos nas seções posteriores.

Figura 9 - Diagrama da *View* do *boilerplate*, com os pacotes em que se comunica simplificados.



Fonte: produção própria.

5.2.2.1 View parte XML (diretório de recursos (`res`))

Essa parte da `View` contém uma estrutura de diretórios onde estão localizados os mais diversos recursos XML. Segue uma breve descrição deles:

- s*trings*: são os textos que compõem a aplicação. O Android conta com muitos recursos de internacionalização e localização, disponíveis para a configuração desses textos;
- s*tyles*: são os estilos da aplicação. Com eles é possível definir estilos padrão contendo um conjunto de cores, dimensões, e atributos de layout diversos;

- n*avigation*: é onde é definida a navegação da aplicação segundo o componente *Navigation* do Android Jetpack;
- d*imens*: são definidas nesse documento as dimensões padrão dos componentes declarados em arquivos de layout;
- l*ayout*: são arquivos onde são declarados todos os componentes gráficos estáticos da aplicação;
- d*rawable* e m*ipmap*: são definidas as imagens e formas da aplicação. Para as imagens, são definidas pastas separadas de acordo com os tamanhos de tela padronizados de dispositivos Android.

5.2.2.1.1 Data Binding

Para aumentar a expressividade dos documentos XML da View, e dessa forma auxiliar na manutenção e testabilidade da aplicação, o componente *View* utiliza vínculos de dados (*data binding*). É uma forma de vincular as propriedades do *ViewModel* (os objetos *ViewData*, que contém os dados necessários para a *View* renderizar na tela) diretamente aos componentes gráficos nos layouts. Ao vincular os dados do *ViewModel* diretamente ao layout em XML, permite-se simplificar o código Kotlin da *View*, pois não é necessária a manipulação de objetos *ViewData* por parte dos fragmentos e atividades. Nesse processo, o XML tem acesso direto ao conteúdo do *ViewModel*.

A utilização do *data binding* nos layouts para acessar diretamente os objetos *ViewData* é bastante simples, por ser realizada através do paradigma de programação reativa. Internamente, porém, em tempo de compilação, o *DataBinding* gera automaticamente várias classes que gerenciam os observadores dos objetos *LiveData* em que estão encapsulados os *ViewData* e entregam as mudanças diretamente ao layout.

5.2.2.1.2 Navigation

Navigation é um componente presente no Android Jetpack que tem como objetivo organizar e configurar a navegação entre os fragmentos. É um documento escrito em XML. Com esse documento é possível definir a transição de entrada e saída de cada fragmento.

5.2.2.2 Pacote view

O pacote `view` é a parte onde vão todas as classes que manipulam elementos de interface gráfica de forma programática.

5.2.2.2.1 BindingAdapters

Os *bind adapters* são classes Kotlin que customizam o processo de pegar um valor num objeto ViewData e passá-lo para o componente gráfico no layout. Essas classes possuem o propósito de manipular esses objetos *ViewData* e prepará-los para serem fornecidos aos layouts através do *data binding*. Se uma propriedade de algum *ViewData* precisar ser manipulada antes de ser repassada para o layout, isso deve ser feito nas classes de *bind adapters*, ao invés de ser feito diretamente no XML do layout. Por exemplo, manipular um elemento de texto para colocar um rótulo no valor de um atributo ViewData. Essas classes possuem métodos que recebem como parâmetro o elemento gráfico que vai receber o valor e o ViewData a ser manipulado. Cada método então realiza a manipulação e repassa o valor para o elemento gráfico.

Esses métodos então podem realizar manipulações com esse elemento gráfico, como alterar o texto, forma e posição, utilizando os atributos do objeto de *ViewData* do parâmetro. Com isso pode-se criar conteúdo dinâmico e personalizado a elementos gráficos em tempo de execução. Um exemplo de *bind adapter* está contido no *boilerplate* produzido e discutido com mais detalhes na seção de implementação.

5.2.2.2.2 ViewData

Os objetos que fornecem dados a serem exibidos na tela e são encapsulados por *LiveData* são chamados de dados da visão (*ViewData*). Esses objetos devem ser compostos por dados básicos da plataforma, como strings, números e booleanos, e também podem ser agrupamentos desses dados básicos. Nesse *boilerplate* a recomendação é que os *ViewData* possuam apenas as informações suficientes para que os layouts e *BindAdapters* os utilizem para a exibição correta

dos dados. É necessário que as informações nos *ViewData* já estejam formatadas, de forma a não necessitar de lógica extra para serem exibidas.

5.2.2.3 Comunicação entre View e ViewModel

A comunicação entre classes dos pacotes `view` e `viewmodel` consiste no seguinte fluxo:

1. Um fragmento estabelece o vínculo com uma ou várias classes de *ViewModel* pedindo referências a elas ao *ViewModelProviders*, que é um singleton do Android Jetpack que cria e provê instâncias do *ViewModel*;
2. O fragmento infla (carrega) seu layout correspondente e passa a referência do *ViewModel* a esse layout;
3. O fragmento requisita operações do *ViewModel* que podem representar a vontade do usuário, vinda de eventos do layout ou então busca dos dados;
4. *ViewModel*, atuando como fachada, delega as operações ao pacote modelo, que responde com as entidades novas.
5. Com as entidades novas, o ViewModel atualiza seus atributos *LiveData* com objetos *ViewData,* alimentados por dados das entidades;
6. Com o auxílio do código gerado pelo Data Binding, o layout recebe automaticamente os novos objetos ViewData, assim atualizando reativamente a interface gráfica.

Os dados que o *ViewModel* expõe para a *View* ficam encapsulados em objetos do tipo *LiveData*. Esses *LiveData* são componentes observáveis contidos no *ViewModel*. Através deles, a *View* é capaz de observar suas mudanças e sempre exibir os dados mais atualizados. Esses componentes é que implementam características da programação reativa. Sua utilização se assemelha ao uso do padrão de projeto *Observer*.

5.2.3 ViewModel

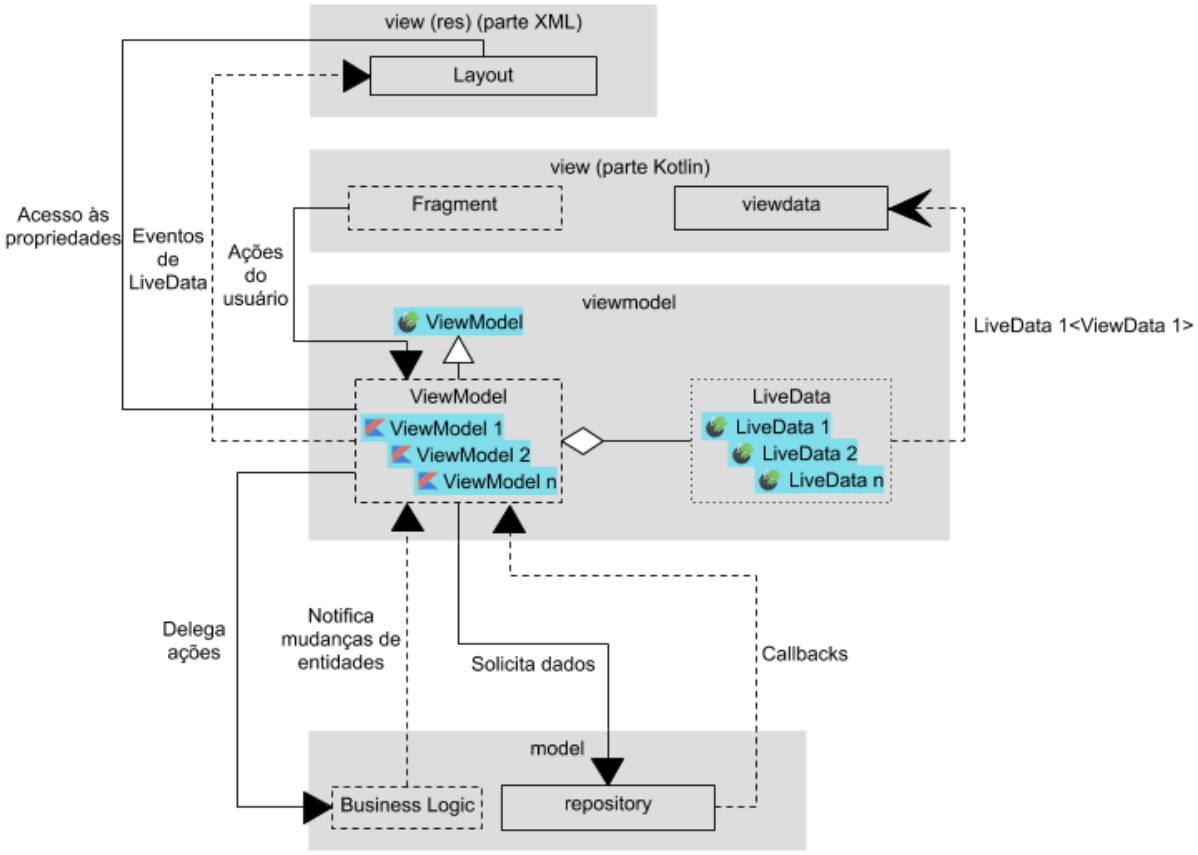
Para o compartilhamento de informações de modelo entre diferentes telas da aplicação, a documentação do Android Jetpack incentiva que essas diferentes telas compartilhem o mesmo *ViewModel* [36]. Acredita-se, porém, que essa abordagem

pode trazer desvantagens em termos de alto acoplamento entre as telas da aplicação e baixa coesão do *ViewModel*. Na arquitetura proposta neste trabalho, recomenda-se que seja criado um *ViewModel* para cada tipo de dado no domínio da aplicação, da mesma forma como podem existir diferentes rotas de *webservices* ou diferentes tabelas no banco de dados representando cada tipo de dado no domínio da aplicação. Essa separação visa reduzir o acoplamento entre telas da aplicação e aumentar a coesão de *ViewModels*.

Esse é, então, um ponto onde a arquitetura proposta difere um pouco do uso padrão dos Componentes de Arquitetura do Android Jetpack. Ao incentivar a criação de um *ViewModel* para cada tipo de dado no domínio da aplicação, a arquitetura proposta neste trabalho aceita que esse *ViewModel* seja compartilhado apenas entre os fragmentos que exibem aquele tipo de dado do domínio.

A figura 10 mostra uma parte do diagrama geral da arquitetura com enfoque no pacote `viewmodel` e suas comunicações. As partes presentes nos Componentes de Arquitetura estão explicitadas com o símbolo do Android Jetpack ().

Figura 10 - Diagrama do *ViewModel* do *boilerplate*, com os pacotes em que se comunica simplificados.



Fonte: produção própria.

5.2.3.1 Classes ViewModel

O componente *ViewModel* possui o mesmo papel do *ViewModel* presente na arquitetura MVVM, e utiliza o próprio componente *ViewModel* disponível nos Componentes de Arquitetura do Android Jetpack. Cada classe de *ViewModel* herda da super classe `ViewModel` presente no Android Jetpack. Esses componentes são cientes dos ciclos de vida dos componentes de visão, como os controladores de UI, e, portanto, permanecem sempre disponíveis independentemente do ciclo de vida dos componentes de visão.

O framework Android gerencia os ciclos de vida dos controladores de UI. O framework pode decidir destruir ou recriar um controlador de UI em resposta a

certas ações do usuário ou eventos do dispositivo que são externos e, portanto, fora do controle do desenvolvedor.

Se o sistema destruir ou recriar um controlador de UI, qualquer dado relacionado à UI está perdido. Por exemplo, uma aplicação possui uma lista de usuários em um dos seus fragmentos. Quando a atividade é recriada por alguma mudança de configuração, o novo fragmento terá que fazer de novo a requisição dessa lista de usuários. Para dados simples, os fragmentos podem usar o método `onSaveInstanceState()` para guardar e restaurar os dados, porém essa abordagem é aconselhada somente para poucas quantidades de dados que podem ser serializados e então desserializados, mas não para listas potencialmente grandes, como uma lista de usuários ou imagens.

Outro problema é que os controladores de UI frequentemente precisam realizar chamadas assíncronas, as quais podem levar algum tempo para retornar. Os controladores de UI precisam gerenciar essas chamadas para garantir que o sistema limpe os recursos após a sua destruição, para evitar potenciais vazamentos de memória (*memory leaks*). Gerenciar essas chamadas assíncronas ao mesmo tempo em que os controladores de UI passam por seus diferentes estágios de ciclo de vida é uma tarefa que requer cuidado. Além disso, no cenário onde o controlador de UI é recriado por alguma mudança de configuração, ocorre desperdício de recursos, visto que os dados carregados anteriormente são perdidos e o objeto pode ter que refazer chamadas que já fez antes.

Controladores de UI, como atividades e fragmentos, são primariamente feitos para mostrar dados de UI, reagir a ações do usuário ou tratar operações de comunicação com o sistema, como requisições de permissões. Fazer com que os controladores de UI também sejam responsáveis por carregar dados de fontes de dados adiciona complexidade à classe. Designar responsabilidades excessivas a controladores de UI poderá resultar em uma única classe que tenta tratar toda a lógica da aplicação sozinha, ao invés de delegar esse trabalho a outras classes [23]. Além disso, essa abordagem também dificulta a criação de testes unitários [24].

Os Componentes de Arquitetura do Android Jetpack, então, proveem o componente de arquitetura *ViewModel*, que terá a responsabilidade de gerenciar os dados que irão ser exibidos, removendo esse papel dos controladores de UI.

Objetos de *ViewModel* retêm automaticamente os dados durante as diversas mudanças de configuração dos controladores de UI.

Um *ViewModel* só é destruído quando a atividade ou fragmento que o usa é terminado (passa pelo estado `onDestroy`). O framework Android chama o método `onCleared()`, e então o objeto de *ViewModel* pode limpar seus recursos.

O componente de arquitetura *ViewModel* é o que compõe o VM do MVVM. Seu propósito é:

- Transformar ações do usuário em mensagens para os repositórios e classes da lógica de negócio;
- Formatar *ViewData* com os dados do modelo para expô-los encapsulados como *LiveData*.

Um *ViewModel* nunca poderá receber referências a classes de `view`, ou quaisquer classes que podem conter alguma referência ao contexto da atividade ou componente de visão. O papel de lidar com os componentes de visão é exclusivamente dos *BindAdapters*, layouts ou eventuais objetos de formatação do *View*.

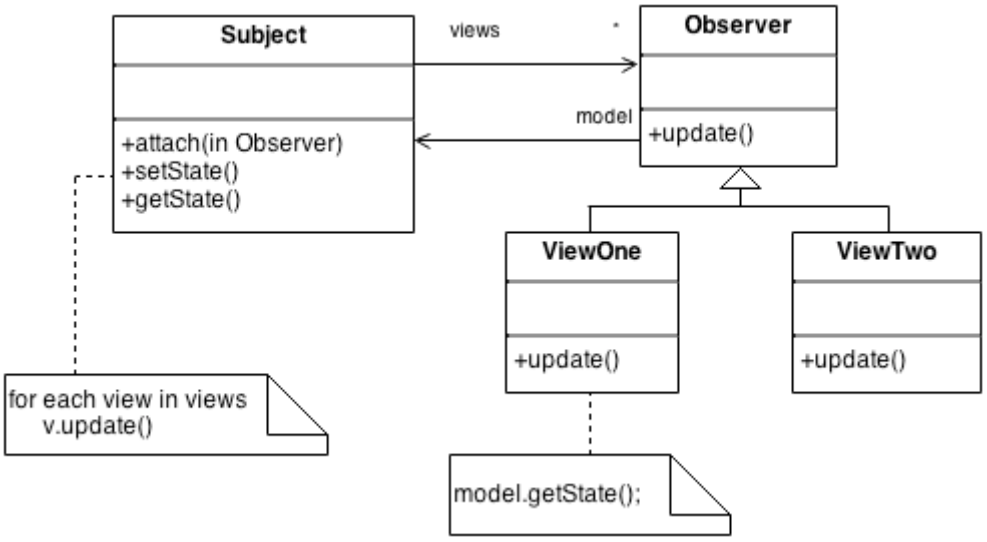
Para o isolamento de *ViewModels*, os repositórios são passados pelo construtor do *ViewModel*, e o mesmo só recebe uma referência a uma interface que é implementada pelo repositório. Portanto, o *ViewModel* não tem a responsabilidade de criar o repositório; isso se chama injeção de dependência. Durante a realização de testes unitários é possível passar para o *ViewModel* um objeto falso que simula o comportamento de um repositório, tornando o *ViewModel* testável de forma isolada.

Os atributos de um *ViewModel* são constituídos de *ViewData* encapsulados por *LiveData*. *LiveData* é um dos Componentes de Arquitetura do Android Jetpack.

5.2.3.2 LiveData

O envio de dados entre *View* e *ViewModel* de forma reativa é realizado pelas objetos *LiveData*. É uma classe que implementa o padrão *Observer*. A figura 11 mostra um diagrama UML do padrão *Observer*:

Figura 11 - Diagrama UML de classe do padrão *Observer*.



Fonte: Observer Design Pattern [46].

De forma a estender o padrão Observer tradicional, o funcionamento do *LiveData* é ciente do ciclo de vida dos controladores de UI. Para isso, sempre que um novo observador é adicionado ao *LiveData*, é passado também como parâmetro um controlador de UI. Eventos de alteração nos dados observados só são enviados aos observadores se esse controlador de UI encontra-se ativo. Além disso, se o controlador de UI é destruído, o vínculo se encerra e os observadores são removidos.

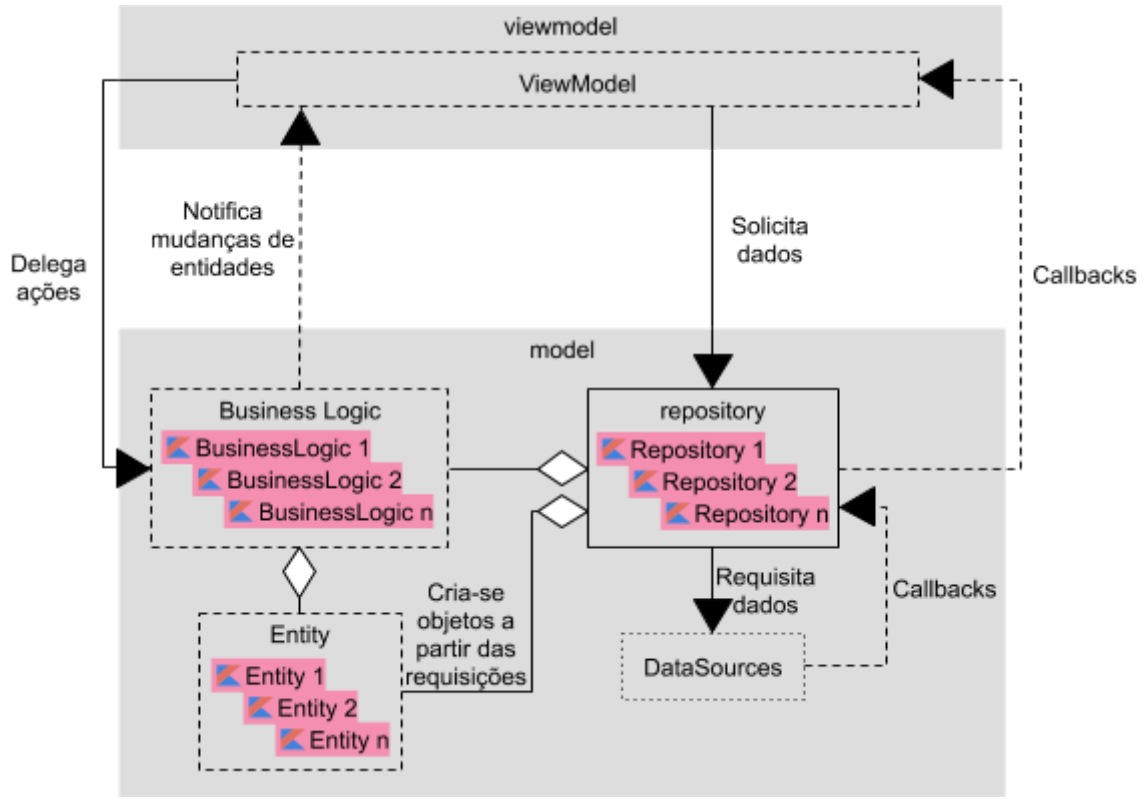
No método `onCreateView()` dos fragmentos são buscadas as referências para os *ViewModels* utilizados pelo fragmento. Esses *ViewModels* são então repassados para o componente de DataBinding, que fica responsável por expor o *ViewModel* ao layout associado ao fragmento. Assim, o layout, possuindo a referência do *ViewModel*, poderá vincular seus componentes gráficos aos ViewData de forma reativa. Internamente, são adicionados observadores aos *LiveDatas* que englobam esses *ViewData*.

5.2.4 Model

A figura 12 acima mostra o pacote modelo e suas conexões em evidência. O modelo é responsável pelas entidades, que consistem nas classes básicas de

dados do domínio da aplicação. O modelo também é responsável pelos repositórios, que realizam as requisições de dados, e a lógica de negócio da aplicação.

Figura 12 - Diagrama do *Model* do *boilerplate*, com os pacotes em que se comunica simplificados.



Fonte: produção própria.

5.2.4.1 Entidade

Para as entidades (objetos *Entity* no diagrama) da aplicação, utiliza-se o recurso de classes de dados (*Data Classes*) disponível no Kotlin.

Essas entidades representam os objetos básicos do domínio da aplicação e seus atributos. Elas não possuem funcionalidade nenhuma, além de definir o formato dos objetos do domínio com seus respectivos atributos.

5.2.4.2 Repositório

Para as requisições de dados, utiliza-se o padrão de repositório (*Repository*). Cada tipo de dado do domínio da aplicação, geralmente representado por uma rota num *webservice* ou uma tabela num banco de dados, possui seu próprio *Repository*. Ou seja, para cada *ViewModel* da aplicação, o ideal é que exista um *Repository* fornecendo a ele as entidades do modelo. Seu propósito é fazer requisições para as fontes de dados (*Data Sources*), que podem ser requisições HTTP para alguma API, uma consulta no banco de dados, ou qualquer recurso que forneça os dados necessários para criar as entidades. Geralmente, a comunicação entre os repositórios e as fontes de dados é realizada através de chamadas assíncronas. O repositório passa para o *Data Source* uma referência de um método (*callback*) para receber os dados, e assim montar a entidade e passar para o *ViewModel*. Classes de repositório também poderão possuir instâncias de classes de lógica de negócio para realizar a lógica da aplicação.

5.2.4.3 Lógica de negócio

A lógica de negócio (*Business Logic*) da aplicação se concentra neste componente. As classes de lógica de negócio são responsáveis por processar os dados e produzir resultados que alterem os objetos do modelo, o que gera notificações para o *ViewModel*.

As classes da lógica do negócio são acessadas diretamente pelos repositórios, porém, também podem receber mensagens do *ViewModel* solicitando ações.

Por conta de não ter conhecimento de classes de visão ou de qualquer outra classe do pacote `android`, pode-se dizer que esse componente não tem ciência de que é um componente de uma aplicação Android. Consequentemente, é um componente independente de plataforma. Essa característica permite que essa lógica seja facilmente convertida para outra linguagem de programação ou outra plataforma suportada pela linguagem Kotlin sem modificações significativas no código.

5.2.5 Testes unitários

Um teste unitário consiste em aplicar dados de entrada (input) e testar se o retorno ou saída de um método (output) segue o esperado. A arquitetura proposta foi planejada de forma que seja possível criar testes unitários para quase todos os componentes da aplicação.

É comum que os métodos de uma classe dependam de classes de outros componentes da aplicação ou de componentes externos à aplicação, o que pode dificultar a criação de um teste unitário que avalie apenas o comportamento dos métodos da classe de forma isolada. Uma prática comum para resolver esse problema é criar instâncias falsas dos componentes externos, as quais possuem um comportamento controlado e definido pelo ambiente de teste. Essas instâncias são chamadas geralmente pela comunidade de *mocks*. Para o *boilerplate* proposto neste trabalho, foi utilizada uma biblioteca externa, chamada *Mockito*, para facilitar a criação de *mocks*.

Essa biblioteca possui muitos recursos para lidar com objetos *mock*. O mais simples deles é um método chamado `mock<T>()`. É passada uma classe como argumento (`T`) e o *Mockito* cria uma subclasse de `T`, sobrescrevendo todos os métodos de `T`, deixando todos os métodos vazios. Torna-se uma classe sem lógica, apenas com o propósito de substituir um objeto para um teste unitário.

Todo o código de testes unitários é feito para não ser executado em dispositivos Android. Portanto, não há código do framework Android disponível para ser utilizado nos testes. Métodos do componente *View,* por exemplo, não podem ser testados com testes unitários *JUnit* sem que sejam criados *mocks* para todas as classes usadas ou referenciadas do framework Android. Essa prática não é recomendada [24]. Recomenda-se que as classes do componente *View* possuam o mínimo de lógica possível e não sejam testadas com testes unitários. Classes de modelo são livres de código do framework Android, logo, é possível utilizá-las sem dificuldades em testes unitários.

Existem alguns desafios acerca de aplicar testes unitários a classes pertencentes ao componente ViewModel. Os atributos *LiveData* são classes pertencentes ao pacote `androidx` do Android Jetpack, que não faz parte do

framework Android (pacote `android`). Um detalhe em particular, porém, dificulta a criação de testes unitários com *LiveData*. O mecanismo de atualização do estado de um *LiveData* só pode ser feito na *thread* da interface gráfica. Para fazer isso, o *LiveData* acessa internamente classes do framework Android, como *Handler* e *Looper*, que permitem um controle maior sobre as *threads* do sistema Android.

Tecnicamente, do jeito que *LiveData* foi construído, não seria possível testar classes de *ViewModel* num ambiente *JUnit* sem que existam mocks do framework Android inteiro. Porém, para esse caso específico, o pacote `androidx` provê uma solução simples que torna possível utilizar *LiveData* em testes unitários. Para isso, adiciona-se uma regra no arquivo de teste chamada *InstantTaskRule*. Esta regra troca o uso do *Handler* e o *Looper* do framework Android utilizados pelos componentes de arquitetura presentes no ViewModel, possibilitando que classes do componente *ViewModel* sejam totalmente testáveis.

5.3 Implementação

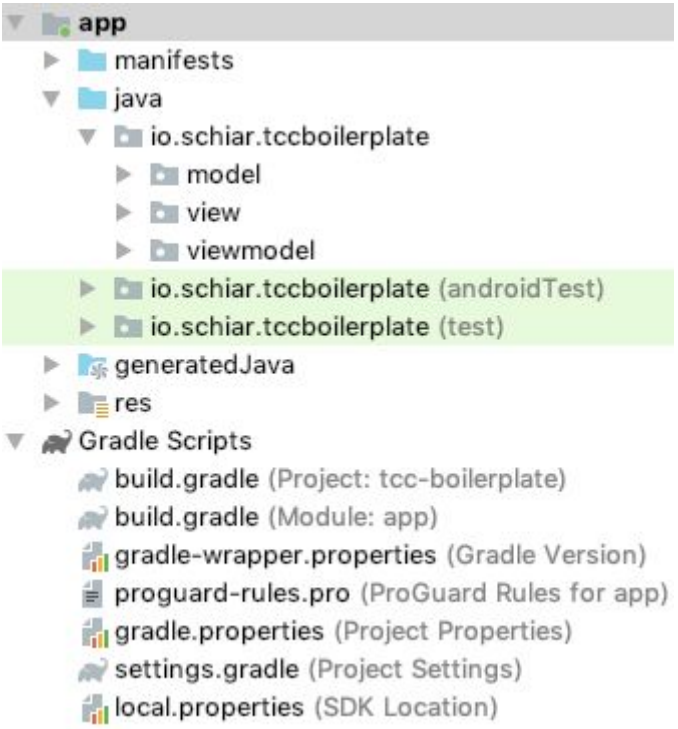
Nesta seção é apresentada a implementação do *boilerplate* seguindo a arquitetura proposta. O objetivo do código é ilustrar os componentes da arquitetura, e não demonstrar funcionalidade. Porém, esse *boilerplate* pode ser compilado e executado como uma aplicação Android sem muita funcionalidade, o que permite que programadores desenvolvam suas aplicações a partir dele.

5.3.1 Estrutura geral

A organização do código de projetos Android é semelhante à organização de projetos Java, onde cada pacote precisa estar em uma pasta do sistema. A figura 13 mostra a estrutura.

O formato de visualização do Android Studio simplifica e abstrai os diretórios do projeto. Basicamente, todo o código do boilerplate está dentro do diretório java. Apesar do nome, o código foi desenvolvido em Kotlin.

Figura 13 - Estrutura de diretórios do projeto Android do *boilerplate*.



Fonte: Android Studio.

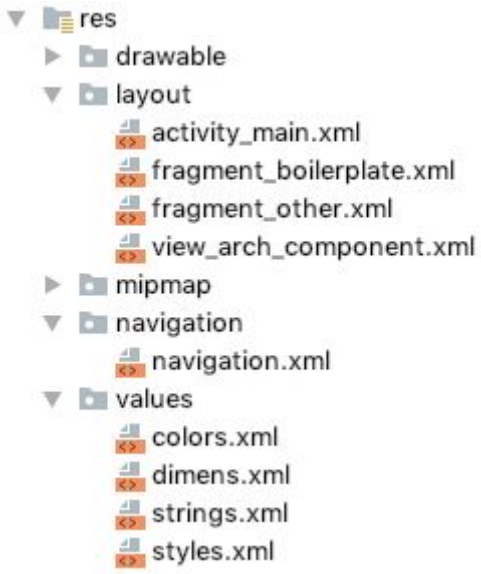
5.3.2 View

A *View* é dividida em componentes programáticos em Kotlin e componentes declarativos em XML. O conteúdo programático fica dentro do diretório `view` e o conteúdo declarativo fica dentro do diretório `res` como ilustrado na figura 14 e 15.

Figura 14 - Estrutura de diretórios do pacote `view` do *boilerplate*.

Fonte: Android Studio.

Figura 15 - Estrutura de diretórios diretório `res` chamado de parte XML da View.



Fonte: Android Studio.

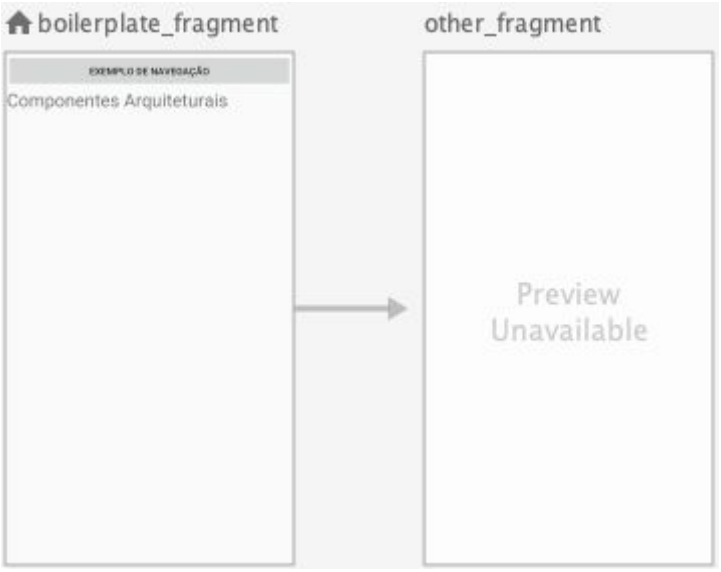
As pastas `drawable` e `mipmap` foram ocultadas para simplificação. Elas servem para colocar os ícones e imagens da aplicação. `drawable`, em especial, poderá guardar estilização adicional de componentes de visão. Existe também, dentro de `drawable`, uma pasta para cada tamanho padronizado dos dispositivos Android.

5.3.2.1 Navigation

Essa pasta guarda todos os arquivos usados para definir a navegação dos fragmentos da aplicação. O *boilerplate* produzido utiliza apenas um arquivo de `navigation`, chamado *navigation.xml*.

Abrindo-se o componente de *Navigation* no Android Studio, é possível observar um diagrama com telas da aplicação e suas interfaces gráficas. Esse editor provido pelo Android Studio interpreta o arquivo XML e mostra as telas da aplicação e como elas estão conectadas. A figura 16 mostra os dois fragmentos contidos no *boilerplate* proposto e a navegação definida partindo de `boilerplate_fragment` para `other_fragment`. Também é possível definir toda a navegação dos fragmentos através deste editor.

Figura 16 - Diagrama do navigation, mostrando os dois fragmentos que existem no boilerplate desenvolvido e a navegação entre eles.



Fonte: Android Studio.

No *boilerplate* proposto, foi definido um exemplo simples de navegação, apenas para fins de demonstração desse componente. Esse *Navigation* possui então a declaração dos fragmentos. Além disso, dentro da declaração do fragmento `boilerplate_fragment`, existe também um componente de ação (*action*) que define a navegação entre os dois componentes.

Código 1 - Trecho de código XML extraído do arquivo *navigation.xml* mostrando a declaração de um fragmento e a navegação para outro fragmento.

```
<fragment
    android:id="@+id/boilerplate_fragment"
    android:label="@string/title"
    android:name="io.schiar.tccboilerplate.view.BoilerplateFragment"
    tools:layout="@layout/fragment_boilerplate"
>
    <action
        android:id="@+id/boilerplate_to_other"
        app:destination="@id/other_fragment"
        app:enterAnim="@anim/nav_default_enter_anim"
        app:exitAnim="@anim/nav_default_exit_anim"
        app:popEnterAnim="@anim/nav_default_pop_enter_anim"
        app:popExitAnim="@anim/nav_default_pop_exit_anim"
```

```
    />
</fragment>
```

Fonte: produção própria.

Os fragmentos são declarados em *tags* `<fragment>` com identificação (*id*), título que a barra de tarefas irá exibir quando estiver com esse fragmento sendo exibido na tela (*label*), nome da classe do fragmento (*name*) e o layout associado ao fragmento (*layout*). Os filhos dessa *tag* contém os componentes de ação, representados por `<action>`, com identificação (*id*), o destino (*destination*) e os diversos atributos de transição.

5.3.2.2 Data Binding

Com os *Data Bindings*, os componentes declarativos de layout ganham a habilidade de poder referenciar o *ViewModel* diretamente criando uma ligação implícita direta entre os componentes mais declarativos e simples da *View* e o *ViewModel* da aplicação.

Código 2 - Trecho de código XML extraído do arquivo *fragment_other.xml* exemplificando o data binding.

```
<layout ...>
    <data>
        <variable
          name="viewModel"
          type="io.schiar.tccboilerplate.viewmodel.OtherViewModel"
        />
    </data>
    <FrameLayout ...>
        <TextView ... android:text="@{viewModel.helloWorld}" />
    </FrameLayout>
</layout>
```

Fonte: produção própria.

Todo o conteúdo de um arquivo de layout fica dentro de tags `<layout>`. A estrutura de um documento de layout com *DataBinding* possui duas seções distintas. Uma delas se chama `<data>`, onde são declaradas as variáveis que serão

recebidas e utilizadas pelo *DataBinding*. Cada variável é declarada com nome (`name`) e tipo (`type`). É no tipo que é colocado o nome absoluto de uma classe do *ViewModel* que será utilizada. A outra seção, chamada no *boilerplate* apresentado de `FrameLayout`, define o layout do fragmento, onde `FrameLayout` é o nome do componente de layout.

O Android provê diversos tipos de layout para se utilizar, e é com esse layout que são definidas as regras de como os filhos do layout devem se dispor na tela. Dentro do componente de layout são definidos os componentes gráficos. Nesse exemplo é utilizado um `TextView`, que simplesmente exibe um texto na interface gráfica. São definidas diversas propriedades, sendo algumas omitidas com reticências para simplificação.

A propriedade exibida é a `android:text`. Essa propriedade contém como valor um *LiveData* que consta no *ViewModel*. Quando o fragmento for carregado, será exibido diretamente o valor do *LiveData* contido no *ViewModel*, e, se houver mudanças nesse *LiveData*, o TextView também irá mudar.

Referenciar o *ViewModel* diretamente no XML do layout usando *DataBinding* faz o sistema gerar um vínculo implícito entre o componente *ViewModel* e a *View*. Toda alteração no *ViewModel* é automaticamente delegada para a View de forma reativa.

Portanto, é nessa integração que aspectos da programação reativa ficam mais visíveis na arquitetura. O vínculo entre componente gráfico e propriedade do *ViewModel* ocorre na seguinte linha do código:

```
<TextView ... android:text="@{viewModel.helloWorld}" />
```

Pode-se simplificar a sintaxe dessa linha de código da sua seguinte forma:

```
TextView.text = viewModel.helloWorld
```

Como visto anteriormente, na programação reativa, a instrução acima não apenas representa que a propriedade `text` de `TextView` deve receber o valor da propriedade `helloWorld` de `viewModel`; ela também representa que toda vez que o

valor de `helloWorld` for modificado, o valor da propriedade `text` de `TextView` será automaticamente atualizado também.

Para utilizar *DataBinding* no projeto, são necessárias algumas modificações no projeto Android. Mais especificamente, é necessário integrar uma biblioteca externa ao projeto, o que é feito nos arquivos *gradle* do projeto, que gerenciam as dependências de projetos Android. Além disso, são necessárias mudanças no método `onCreateView` dos fragmentos.

Código 3 - Trecho de código extraído da classe `OtherFragment` do pacote `view`. Método inicial de um fragmento.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    viewModel = ViewModelProviders.of(this).get(OtherViewModel::class.java)
    val binding = FragmentOtherBinding.inflate(inflater, container, false).apply {
        lifecycleOwner = this@OtherFragment
        viewModel = this@OtherFragment.viewModel
        executePendingBindings()
    }
    return binding.root
}
```

Fonte: produção própria.

Esse método é utilizado para carregar o layout, o componente declarativo localizado na pasta `res` (XML). Esse método inicialmente obtém a referência do *ViewModel* usando a classe `ViewModelProviders` do Android Jetpack.

Em seguida, ocorre a configuração do *DataBinding*. O carregamento do XML se dá pela classe `FragmentOtherBinding`. Essa classe é gerada automaticamente pelo *DataBinding* em tempo de compilação a partir do nome do XML, mudando a formatação do texto e adicionando a palavra *Binding*. É nessa classe gerada que são criadas as abstrações necessárias para que a *View* observe as mudanças do *ViewModel* e atualize a interface gráfica, de forma que o programador não precisa explicitamente fazer essas conexões no código Kotlin da *View*.

O método `inflate` carrega o XML propriamente e retorna uma instância de `FragmentOtherBinding`, a representação programática do XML. O fragmento é então passado para essa instância como o objeto que possui um ciclo de vida (`lifecycleOwner`). O `viewModel` também é passado para o `FragmentOtherBinding`, de forma que ele possa usá-lo no layout. É nessa linha que é realizado o vínculo propriamente dito entre o *ViewModel* e o XML. É então executado o método `executePendingBindings` para realizar alguma atualização de variáveis que possa ter ficado pendente. Por fim, é retornado o `binding.root` que contém a *View* raiz do layout, que é uma instância da classe `FrameLayout` referenciada no componente de layout do XML.

5.3.2.3 RecyclerView

Um componente especial utilizado para a renderização gráfica de listas é o `RecyclerView`. Esse componente usa o conceito de reutilização do componente gráfico para mostrar o conteúdo da lista. Por exemplo, dada uma lista de mil elementos, `RecyclerView` não irá gerar mil componentes gráficos. Ao invés disso, é gerada apenas a quantidade que cabe na tela. Conforme o usuário desce ou sobe a lista, o `RecyclerView` realiza um mecanismo de reutilizar o componente gráfico do componente que deixou de ser exibido para carregar o conteúdo do novo item da lista a ser exibido.

Código 4 - Trecho de código extraído do arquivo XML *fragment_boilerplate.xml*. Mostra a declaração de um `RecyclerView`.

```
<androidx.recyclerview.widget.RecyclerView
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   app:adapter="@{adapter}"
   app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
/>
```

Fonte: produção própria.

O código acima mostra a declaração de um `RecyclerView` no XML de layout do fragmento `BoilerplateFragment`.

Para o RecyclerView funcionar, ele precisa de dois atributos essenciais. Um deles é o LayoutManager, que irá configurar a disposição dos itens na tela. No exemplo acima, foi utilizado um LinearLayoutManager. Isso significa que o RecyclerView irá dispor os itens da lista da mesma maneira que o layout LinearLayout. O LinearLayout dispõe os itens verticalmente, um embaixo do outro, ou horizontalmente, um ao lado do outro. O padrão é dispor verticalmente.

O outro atributo necessário para o funcionamento correto do RecyclerView é o *adapter*. Ele tem o propósito de receber os dados a serem visualizados e configurar a disposição de cada item da lista.

A classe Adapter é uma classe abstrata interna do RecyclerView. Para criar-se um *adapter* é necessário criar uma classe que herde de RecyclerView.Adapter, além de uma outra classe interna que herde de ViewHolder. A subclasse precisa sobrescrever os métodos necessários para o funcionamento do RecyclerView. O mais importante deles é o onCreateViewHolder. É nesse método que é criada a instância da classe interna ViewHolder. Essa classe irá ser responsável por realizar a configuração de cada item da lista.

Código 5 - Trecho de código extraído da classe ArchComponentsListAdapter. Mostra o método que carrega o XML e passa para o ViewHolder.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
   val inflater = LayoutInflater.from(parent.context)
   val binding = DataBindingUtil.inflate<AdapterArchComponentBinding>(inflater,
R.layout.adapter_arch_component, parent,false);

   return ViewHolder(binding);
}
```

Fonte: produção própria.

O código acima mostra a implementação da função onCreateViewHolder no boilerplate apresentado, onde é carregado o layout XML correspondente ao item gráfico da lista. Como esse layout XML possui elementos de DataBinding, o carregamento é semelhante ao do método onCreateView apresentado

anteriormente. O retorno desse método é uma instância de `ViewHolder` que recebe como parâmetro no construtor o objeto do *DataBinding*.

Outro método importante criado na classe de *adapter* é o `onBindViewHolder()`. É esse método que o `RecyclerView` usará para a reutilização do componente gráfico.

Código 6 - Trecho de código extraído da classe `ArchComponentsListAdapter`. Mostra o método que chama o método que configura o item de lista.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
   holder.bind(archComponents[position]);
}
```

Fonte: produção própria.

Esse método recebe uma instância de `ViewHolder`, e a posição do item a ser carregado. O corpo do método então chama o método `bind` do `ViewHolder` passando como parâmetro o *ViewData* correspondente à posição da lista.

Código 7 - Trecho de código extraído da classe `ArchComponentsListAdapter.ViewHolder`. Mostra o método de ViewHolder que configura o item de lista.

```
fun bind(archComponent: ArchComponentViewData) {
    binding.apply {
        this.archComponent = archComponent
        executePendingBindings()
    }
}
```

Fonte: produção própria.

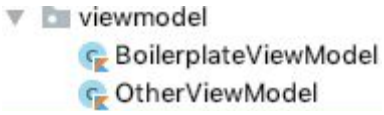
Este método é o único da classe `ViewHolder`. Seu propósito é atribuir ao layout o *ViewData* recebido do *adapter*.

5.3.3 ViewModel

O pacote `viewmodel` contém uma coleção de classes com um sufixo *ViewModel*. Cada uma dessas classes herdam de `ViewModel`, classe dos

Componentes de Arquitetura do Android Jetpack que implementa o papel do *ViewModel* da arquitetura MVVM. A figura 17 mostra a estrutura de diretórios do pacote.

Figura 17 - Estrutura de diretórios do pacote `viewmodel`.



Fonte: Android Studio.

A figura mostra as classes de *ViewModel* presentes no *boilerplate* desenvolvido. A classe `BoilerplateViewModel` contém um exemplo completo de utilização de um *ViewModel*, enquanto que `OtherViewModel` é apenas uma classe mais enxuta de *ViewModel* para ser editada pelo programador que utilizar o *boilerplate* apresentado.

Código 8 - Trecho de código da classe `BoilerplateViewModel`.

```
class BoilerplateViewModel(
   private val archComponentRepository: ArchComponentRepositoryInterface =
ArchComponentRepository()
) : ViewModel() {
   val archComponents: MutableLiveData<List<ArchComponentViewData>> by lazy {
       MutableLiveData<List<ArchComponentViewData>>()
   }
   fun fetch() {
       archComponentRepository.fetch {
           archComponents.postValue(it.map {archComponent: ArchComponent ->
               ArchComponentViewData(
                   archComponent.name,
                   archComponent.description,
                   LibraryViewData(
                       archComponent.library.name,
                       archComponent.library.version.joinToString(".")
                   )
               )
           })
       }
   }
```

```
}
```

Fonte: produção própria.

Esse é o código do `BoilerplateViewModel`, uma classe de *ViewModel*. Essa classe possui `archComponents` como atributo do tipo `MutableLiveData<List<ArchComponentViewData>>`. Ela também recebe pelo construtor o `archComponentRepository`, que é o repositório de onde o *ViewModel* buscará os dados. `MutableLiveData` é uma implementação de `LiveData` que permite mudanças, e `List<ArchComponentViewData>` é uma lista de `ArchComponentViewData`, um *ViewData* que agrupa os dados básicos de um `ArchComponent` que serão mostrados na tela.

O método `fetch()` é chamado por algum componente de *View* que tenha acesso a esse *ViewModel*. No contexto do *boilerplate*, ele é chamada pelo fragmento. O *ViewModel* utiliza o repositório para buscar os dados. No parâmetro do método `fetch()` do repositório é passada a função, ou *callback*, que é chamado pelo repositório quando o dado retorna. Assim é chamado o método `postValue` do *LiveData* para atualizar seu valor, e, consequentemente, atualizar a interface gráfica.

Como o repositório retorna uma lista de objetos do modelo, é necessário uma transformação para que o *LiveData* receba uma lista de *ViewData*. É executado o método `map` para isso. Itera-se por cada objeto de modelo da lista e cria-se um ViewData com os atributos do objeto de modelo formatados corretamente para serem exibidos na tela.

5.3.4 Model

As classes de lógica de negócio da aplicação variam bastante de aplicação para aplicação. Como o *boilerplate* apresentado não possui nenhuma funcionalidade complexa, ele também não possui classes de lógica de negócio, embora os diagramas apresentados anteriormente mostrem como essas classes se encaixam na arquitetura e se comunicam com os demais componentes.

Além disso, para o contexto desse *boilerplate*, a fonte de dados (*data source*) utilizada pelo repositório contém apenas dados fixos, e fica dentro do próprio

repositório. As classes contidas no pacote `model` possuem então as entidades e os repositórios conforme mostra a figura 18.

Figura 18 - Estrutura de diretórios do pacote `model`.

Fonte: Android Studio.

Os repositórios ficam em seu próprio pacote dentro do pacote `model`, junto com a sua interface, enquanto as classes de entidades da aplicação ficam diretamente dentro do pacote `model`, assim como as classes de lógica de negócio também ficariam.

Código 9 - A *Data Class* `Library`.

```
data class Library(
   val name: String,
   val version: List<Int>
)
```

Fonte: produção própria.

O código acima mostra o exemplo de uma classe de dados (*Data Class*). Essa classe representa uma das entidades do domínio da aplicação usada de exemplo no *boilerplate*.

Código 10 - A interface `ArchComponentRepositoryInterface`.

```
interface ArchComponentRepositoryInterface {
   fun fetch(callback: (List<ArchComponent>) -> Unit )
}
```

Fonte: produção própria.

O código acima contém a interface do repositório que buscará objetos de modelo. O método `fetch()` recebe o parâmetro *callback*. O *callback* é uma função sem valor de retorno (representado por `Unit` no Kotlin) e tem como parâmetro a lista de objetos de modelo. Quem chamar o método `fetch()` passará como parâmetro o *callback* que será executado quando os objetos de modelo estiverem disponíveis.

Código 11 - A classe `ArchComponentRepository`.

```
class ArchComponentRepository: ArchComponentRepositoryInterface {
   private val archComponents = listOf(
       ArchComponent("Data Binding",
                     "Serve para o XML da view ter acesso direto ao View Model",
                      Library("com.android.databinding:compiler", listOf(3, 3, 2))
       ), ...)
   override fun fetch(callback: (List<ArchComponent>) -> Unit) {
       return callback(archComponents)
   }
}
```

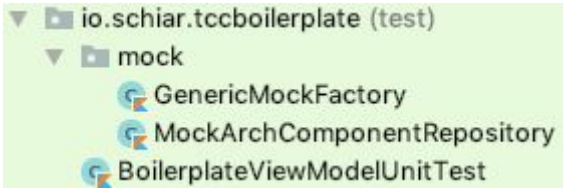
Fonte: produção própria.

Esta é uma parte da classe de repositório `ArchComponentRepository`. Ela implementa a interface do repositório e possui como atributo a lista de objetos de modelo, que para o *boilerplate* apresentado simplesmente é criada uma lista fixa. O método `fetch()`, que é sobrescrito da interface, apenas chama o *callback* do parâmetro passando o atributo. Em uma aplicação real, esse dado poderá vir de uma requisição de uma API, banco de dados ou quaisquer fontes de dados externas.

5.3.5 Testes

O diretório de testes unitários possui a mesma estrutura de diretório do código programático da aplicação. O Android Studio adiciona *(test)* ao lado para identificação.

Figura 19 - Estrutura de diretórios do pacote de testes.



Fonte: Android Studio.

Os objetos *mock* ficam no pacote `mock`,enquanto os arquivos de testes unitários ficam direto no pacote `tccboilerplate`.

Código 12 - A classe `MockArchComponentRepository`.

```
class MockArchComponentRepository : ArchComponentRepositoryInterface {
   override fun fetch(callback: (List<ArchComponent>) -> Unit) {
       val archComponent = ArchComponent(
           "Mock",
           "...",
           Library("org.mockito:mockito-core", listOf(2, 27, 0))
       )
       callback(listOf(archComponent))
   }
}
```

Fonte: produção própria.

O código acima mostra o *mock* utilizado para fins de demonstração no *boilerplate*. É criado uma lista com um único objeto de modelo fixo. Desse modo, sempre que o teste executar, o mesmo resultado será retornado. Nesse *boilerplate* o repositório real da aplicação não fará diferença, porém ao utilizar fontes de dados verdadeiras esse *mock* se tornará útil, pois continuará como um repositório confiável simulado que retorna sempre o mesmo resultado, independentemente de quantas vezes é chamado seu método `fetch()`.

Código 13 - A classe de teste `BoilerplateViewModelUnitTest`.

```
class BoilerplateViewModelUnitTest {
    @get:Rule
    val rule = InstantTaskExecutorRule()
    private val archComponentViewData = ArchComponentViewData(
        "Mock",
        "...",
        LibraryViewData("org.mockito:mockito-core", "2.27.0")
    )
    private lateinit var boilerplateViewModel: BoilerplateViewModel
    @Before
    fun prepare() {
        boilerplateViewModel = BoilerplateViewModel(MockArchComponentRepository())
    }
    @Test
    fun fetch_archComponentsPostsCorrectArchComponentViewData() {
        val observer: Observer<List<ArchComponentViewData>> = mock()
        boilerplateViewModel.archComponents.observeForever(observer)

        boilerplateViewModel.fetch()
        verify(observer).onChanged(listOf(archComponentViewData))
    }
}
```

Fonte: produção própria.

Está é uma visão simplificada da classe de teste unitário `BoilerplateViewModelUnitTest`. Possui como atributo uma regra (`rule`). É através dela que é permitido testar *LiveData*. Essa regra de execução de testes faz com que as instâncias de *LiveData* não dependam mais do gerenciamento de *threads* do framework Android, fazendo com que testes unitários *JUnit* possam ser executados fora do sistema Android sem que funcionalidades do framework Android precisem ser simuladas. A classe `BoilerplateViewModelUnitTest` também possui como atributo `archComponentViewData`, que é utilizado nos testes. Essa instância representa exatamente o *ViewData* gerado a partir do objeto de entidade fixo fornecido pelo repositório.

Existe também nessa classe o método `prepare`, usado para o framework de testes JUnit executá-lo antes de executar cada teste. É criado uma instância do objeto de *ViewModel*, e é passado como argumento no construtor o *mock* apresentado acima. Assim, o *ViewModel* irá utilizar esse *mock* de repositório para buscar os dados ao invés do repositório original do *ViewModel*.

No teste mostrado, é verificado se, chamando o método `fetch()` do *ViewModel,* o que é retornado pelo *LiveData* é realmente uma lista contendo o atributo `archComponentViewData`. Existem mais testes nessa classe, aqui ocultados para simplificação.

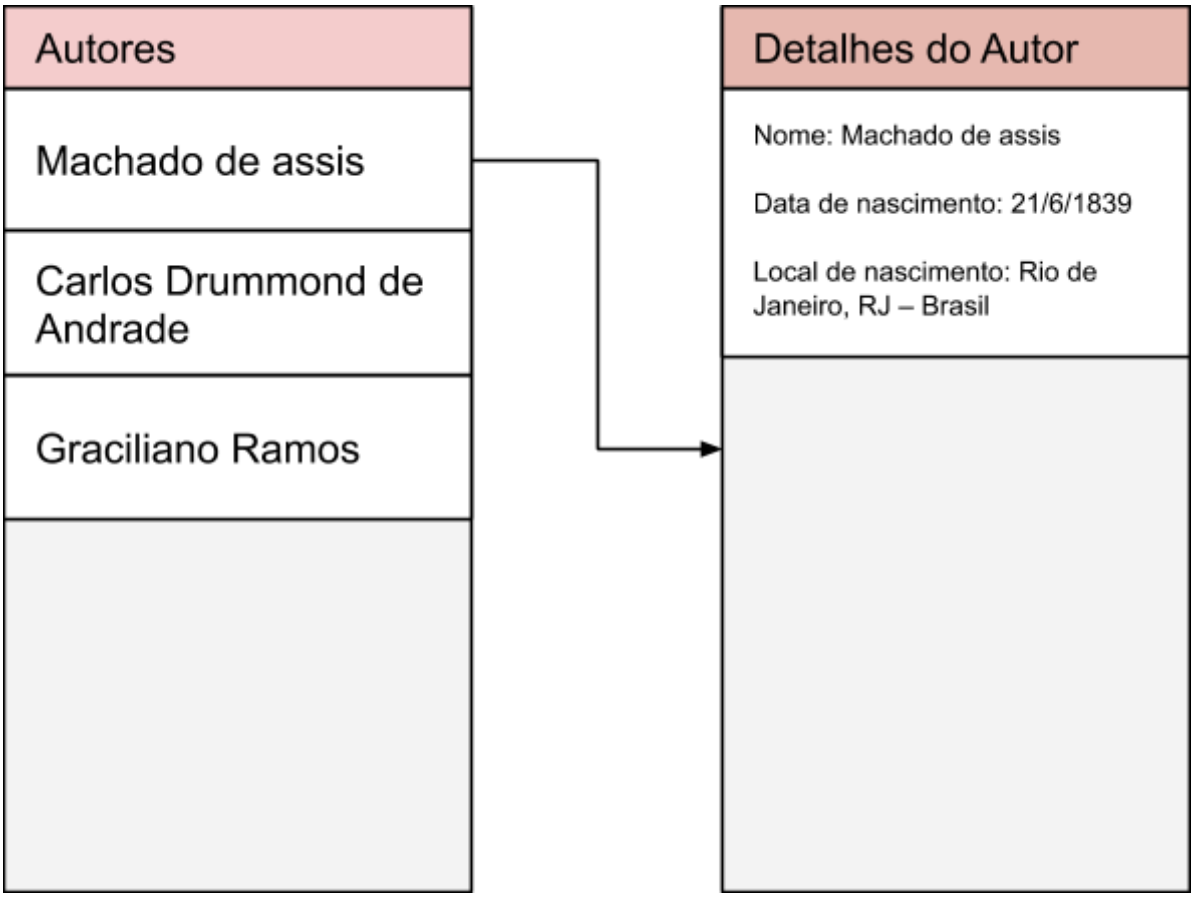
6. Criando uma aplicação com o boilerplate

Neste capítulo é apresentado um passo a passo de como desenvolver uma aplicação utilizando o *boilerplate* desenvolvido.

6.1 Especificação da aplicação

A aplicação a ser desenvolvida a fim de exemplificar a utilização desse *boilerplate* conta com 2 telas. A primeira mostra uma lista de autores, e a segunda, os detalhes de um autor, como mostra a figura 20.

Figura 20 - Esquemático das telas da aplicação.



Fonte: produção própria.

Existe uma API de *webservice* que possui duas rotas, as quais retornam objetos no formato JSON:

- `/autores` - a lista de autores;
- `/autores/{id}` - detalhes de um autor, com `{id}` sendo o id do autor.

Código 14 - Exemplo de JSON do resultado de `/autores`

```
[
    {"id": "1", "nome": "Machado de Assis"},
    {"id": "2", "nome": "Carlos Drummond de Andrade"},
    {"id": "3", "nome": "Graciliano Ramos"}
]
```

Fonte: produção própria.

Código 15 - Exemplo de JSON do resultado de `/autores/1`

```
{
    "id": "1",
    "nome": "Machado de Assis",
    "nascimento": { "dia": 21, "mes": 6, "ano": 1839},
    "localDeNascimento": {
        "cidade": "Rio de janeiro",
        "estado": "RJ",
        "pais": "Brasil"
    }
}
```

Fonte: produção própria.

Com o *boilerplate* aberto no Android Studio pode-se começar a editar os arquivos do projeto para construir a aplicação.

6.2 Construção do modelo

Como visto anteriormente, o *boilerplate* proposto não contém classes de lógica de aplicação, embora os diagramas da arquitetura mostrem como elas devem se comunicar com os outros componentes da arquitetura. O *boilerplate* define, porém, onde as classes de modelo devem ser colocadas na estrutura de diretórios do projeto, além de exemplificar como as classes de entidades e repositórios podem ser criadas.

6.2.1 Entidades básicas

Começando com o modelo, podem-se criar as *data classes* que representam as entidades básicas de modelo.

Código 16 - Classe `Autor`.

```
data class Autor(
   val id: String,
   val nome: String,
   val nascimento: Data,
   val localDeNascimento: Local
)
```

Fonte: produção própria.

Código 17 - Classe `Local`.

```
data class Local(
   val cidade: String,
   val estado: String,
   val pais: String
)
```

Fonte: produção própria.

Código 18 - Classe `Data`.

```
data class Data(
   val dia: Int,
   val mes: Int,
   val ano: Int
)
```

Fonte: produção própria.

Nos trechos de código acima são criadas então algumas *data classes* com os objetos e atributos definidos pelo JSON de retorno da API. Também devem ser criados métodos que convertem os objetos JSON nessas entidades, omitidos nesse exemplo, por simplificação.

6.2.2 Repository

Com as entidades básicas do modelo criadas, o próximo passo é o pacote `repository`.

Nos mesmos moldes da interface existente no *boilerplate*, cria-se a interface `AutorRepositoryInterface`, que servirá de contrato de acesso ao repositório.

Código 19 - A interface `AutorRepositoryInterface`.

```
interface AutorRepositoryInterface {
   val autores: List<Autor>
   fun fetch(callback: (List<Autor>) -> Unit )
   fun fetchDetails(id: String, callback: (Autor) -> Unit )
}
```

Fonte: produção própria.

Cria-se então um repositório que seguirá o contrato estipulado por `AutorRepositoryInterface`, implementando o método `fetch()`, que irá realizar a requisição da lista de autores.

Código 20 - Classe `AutorRepository`.

```
class AutorRepository: AutorRepositoryInterface {
    override lateinit var autores: List<Autor>

    override fun fetch(callback: (List<Autor>) -> Unit) {
       Requisição().buscaJSON("https://exemplo.com/autores") { resultado ->
           autores = parseJSON(resultado)
           callback(autores)
       }
    }
    override fun fetchDetails(id: String, callback: (Autor) -> Unit) {
        Requisição().buscaJSON("https://exemplo.com/autores/$id") { resultado ->
            val autor: Autor = parseJSON(resultado)
            callback(autor)
        }
    }
}
```

Fonte: produção própria.

A classe acima demonstra a criação do `AutorRepository`, que será responsável por buscar a lista do JSON de autores e chamar o *callback* de parâmetro com o resultado. Também é responsável por buscar detalhes de um autor. É importante salientar que, por ser um repositório simples, ele tem acesso direto a aspectos intrínsecos da comunicação com os *webservices*. Numa aplicação real, recomenda-se que todo o código específico dessa fonte de dados seja colocado em classes separadas.

`Requisição` representa a abstração de uma biblioteca que faz requisições HTTP fictícias. O método `buscaJSON` possui como primeiro parâmetro a URL que origina o JSON, enquanto o segundo parâmetro é o *callback* que recebe o resultado da requisição.

Ambos os métodos recebem o resultado da requisição e chamam o *callback* de parâmetro com os objetos do modelo já convertidos do JSON pelo método fictício `parseJSON()`.

Por ser um exemplo simples, não existem classes específicas de lógica de negócio (*Business Logic*). Numa aplicação real, o repositório possuiria instâncias das classes da lógica de negócio e chamaria seus métodos para implementar a lógica necessária. O *boilerplate*, entretanto, auxilia na criação do repositório e mostra como ele se comunica com os outros componentes. Além disso, boas práticas também são demonstradas, como o uso de uma interface para expor as funcionalidades do repositório para o *ViewModel*.

6.3 Construção do *ViewModel*

Com o pacote de modelo totalmente criado, pode-se construir o *ViewModel* da seguinte forma.

Código 21 - A classe `AutorViewModel`.

```
class AutorViewModel(
   private val autorRepository: AutorRepositoryInterface = AutorRepository()
) : ViewModel() {
   val autores: MutableLiveData<List<AutorViewData>> by lazy {
```

```
            MutableLiveData<List<AutorViewData>>()
        }

        val autor: MutableLiveData<AutorDetalhesViewData> by lazy {
            MutableLiveData<AutorDetalhesViewData>()
        }

        fun fetch() {
            autorRepository.fetch {
                   autores.postValue(it.map { autor: Autor ->
                       AutorViewData(autor.nome)
               })
            }
        }

        fun fetchDetail(position: Int) {
            autorRepository.fetchDetails(autorRepository.autores[position].id) {
                val autorViewData = AutorDetalhesViewData(
                    autor.nome,
                    "$nascimento.dia/$nascimento.mês/$nascimento.ano",
                    "$cidade, $estado - $pais"
                )
                autor.postValue(autorViewData)
            }
        }
    }
```

Fonte: produção própria.

Seguindo o modelo do *boilerplate* proposto, a classe `AutorViewModel` recebe como parâmetro uma instância que implementa a interface `AutorRepositoryInterface`. Por padrão, é criada uma instância de `AutorRepository`. Em testes unitários, poderia se criar uma `MockAutorRepository` que retorne resultados controlados.

Na arquitetura proposta neste trabalho, a classe possui como atributos alguns *LiveData*. Mais especificamente, eles são instâncias de `MutableLiveData`, por eles serem mutáveis. O primeiro *LiveData* se chamada `autores`, que é um `MutableLiveData<List<AutorViewData>>`, ou seja, um *LiveData* mutável que engloba uma lista de *ViewData* do tipo `AutorViewData`. Como visto anteriormente,

esses objetos *LiveData* são expostos para a *View* observar e atualizar seus componentes gráficos automaticamente quando os objetos *ViewData* armazenados internamente são modificados. O outro atributo *LiveData* se chama `autor`, que é um `MutableLiveData<AutorDetalhesViewData>`, ou seja, um *LiveData* mutável que engloba um *ViewData* do tipo `AutorDetalhesViewData`. As classes `AutorViewData` e classe `AutorDetalhesViewData` serão criadas logo adiante.

A classe `AutorViewModel` possui também os métodos que serão chamados pela *View* para que os dados necessários comecem a ser carregados. Esses são os métodos `fetch` e `fetchDetail`. Ambos, como resultado, geram atualizações nos `LiveData` do `ViewModel`.

Seguindo a recomendação do *boilerplate*, objetos *LiveData* nunca devem englobar e expor entidades do modelo para a View. Por isso, os *LiveData* do *boilerplate* e deste exemplo englobam classes *ViewData*, as quais contêm apenas as informações básicas necessárias para serem mostradas pela visão. Portanto, foram criados diferentes *ViewData*, cada um para ser usado em cada situação. Seguir essa boa prática na criação do *ViewModel* é uma das vantagens de ter usado o boilerplate como base para a aplicação.

6.4 Construção da *View*

O pacote `viewmodel`, para esse caso em específico, devido a baixa complexidade da aplicação, ficou simples, com apenas uma classe. Pode-se a seguir criar a *View*.

6.4.1 Construção dos *ViewData*

Código 22 - Classe `AutorViewData`.

```
data class AutorViewData(
   val nome: String
)
```

Fonte: produção própria.

Código 23 - Classe `AutorDetalhesViewData`.

```
data class AutorDetalhesViewData(
   val nome: String,
   val nascimento: String,
   val localDeNascimento: String
)
```

Fonte: produção própria.

O `AutorViewData` é usado para concentrar num único objeto os dados de um autor que serão exibidos em cada item na tela que exibe a lista de autores, enquanto que `AutorDetalhesViewData` é usado para exibir mais informações sobre um autor na tela de detalhes de um autor. Nota-se que todos os atributos dos *ViewData* são `String`, e não há o atributo `id`. Quando o `viewModel` busca as entidades de modelo, essas entidades são formatadas em objetos *ViewData*. Para a lista, definiu-se que apenas o nome dos autores será mostrado. Logo, o `AutorViewData` não possui dados de data e local.

6.4.2 Construção do layout da lista de autores e seu adapter

Criados os *ViewData*, pode-se criar então os layouts. O primeiro passo é criar o layout dos fragmentos da aplicação.

Código 24 - O XML que representa o fragmento da tela de lista de autores.

```
<layout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   xmlns:app="http://schemas.android.com/apk/res-auto"
>
   <data>
       <variable
           name="adapter"
           type="io.schiar.tccboilerplate.view.AutoresListAdapter"
       />
   </data>
   <LinearLayout
       android:orientation="vertical"
       android:id="@+id/boilerplate"
```

```
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            tools:context=".view.AutorFragment"
        >
            <androidx.recyclerview.widget.RecyclerView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                app:adapter="@{adapter}"
                app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
            />
        </LinearLayout>
    </layout>
```

Fonte: produção própria.

Esse XML acima possui uma variável vinculada ao fragmento pelo *DataBinding*. Cria-se a *tag* `variable`, que declara a variável `adapter` do tipo `AutoresListAdapter`. Na criação da `RecyclerView`, mais abaixo, define-se então esse `adapter` como o atributo `adapter` da `RecyclerView`, junto com seu `layoutManager`.

Criado o layout do fragmento da lista de autores, cria-se então a classe do `adapter` usado.

Código 25 - Classe de adapter `AutoresListAdapter`.

```
class AutoresListAdapter(
    private val autores: List<AutorViewData>,
    private val autorSelecionadoListener: AutorSelecionadoListener
): RecyclerView.Adapter<AutoresListAdapter.ViewHolder>() {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        val binding = DataBindingUtil.inflate<AdapterAutorBinding>(
        inflater, R.layout.adapter_autor, parent, false);
        return ViewHolder(binding, autorSelecionadoListener);
    }
    override fun getItemCount(): Int {
        return autores.size
    }
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bind(autores[position], position);
    }
```

```
    class ViewHolder(
        private val binding: AdapterAutorBinding,
        private val autorSelecionadoListener: AutorSelecionadoListener
    ) : RecyclerView.ViewHolder(binding.root) {
        fun bind(autor: AutorViewData, posicao: Int) {
            binding.apply {
                this.autor = autor
                this.posicao = posicao
                this.autorSelecionadoListener = autorSelecionadoListener
                executePendingBindings()
            }
        }
    }
}
```

Fonte: produção própria.

A classe ficou semelhante à classe disponível no *boilerplate*. Ela foi adaptada para receber uma lista de *ViewData* de autores e utilizar o layout `adapter_autor` para representar o layout de um item da lista. Como visto anteriormente, o papel do `adapter` é informar à `RecyclerView` quantos e quais dados ela deve mostrar. A classe recebe então a lista de *ViewData* de autores no seu construtor.

Também foi necessário ter na classe um atributo de `AutorSelecionadoListener`, que é repassado para o `binding` (instância de `AdapterAutorBinding` que implementa o *DataBinding* de cada item da lista). Também é passado para o `binding` a posição do autor na lista. Essas informações são passadas para que cada item da lista possa avisar o *listener* quando for selecionado. Como o método de buscar detalhes é chamado na classe do fragmento, o fragmento implementa essa interface `AutorSelecionadoListener`, como vista abaixo, e é passado para o `adapter`, como será visto a seguir.

Código 26 - A interface `AutorSelecionadoListener`.

```
interface AutorSelecionadoListener {
    fun onAutorSelecionado(posicao: Int)
}
```

Fonte: produção própria.

Cria-se então o layout que o `adapter` utiliza para representar cada autor da lista.

Código 27 - O XML de um autor.

```
<layout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
>
   <data>
       <variable name="autor"
type="io.schiar.tccboilerplate.view.viewdata.AutorViewData"/>
       <variable name="posicao" type="Int"/>
       <variable name="autorSelecionadoListener"
                 type="io.schiar.tccboilerplate.view.AutorSelecionadoListener"
       />
   </data>
   <LinearLayout
       android:orientation="vertical"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:layout_marginBottom="15dp"
       android:onClick="@{(view) ->
autorSelecionadoListener.onAutorSelecionado(posicao)}"
   >
       <TextView
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:textSize="@dimen/content_size"
           android:text="@{autor.nome}"
       />
   </LinearLayout>
</layout>
```

Fonte: produção própria.

É possível perceber a declaração de uma *tag* `variable` para cada uma das informações passadas para o `binding` anteriormente, como nome do autor (variável `autor`), `posicao` e `autorSelecionadoListener`. O nome do autor é vinculado então, através de *DataBinding*, à propriedade `text` do `TextView`, enquanto `posicao`

e `autorSelecionadoListener` são usados para tratar o evento de clique no item da lista (propriedade `onClick` do layout).

Dessa forma, o *boilerplate* demonstra como o componente *RecyclerView* do framework Android pode ser integrado com os Componentes de Arquitetura do Android Jetpack de forma a manter o baixo acoplamento entre os componentes. A abordagem adotada nessa seção faz com que as classes de `adapter` não possuam conhecimento de entidades do modelo, o que é uma característica desejável.

6.4.3 Construção do fragmento da lista de autores

Tendo o `adapter`, o layout de um item e o layout do fragmento da lista, cria-se então a classe do fragmento da lista de autores.

Código 28 - A classe de fragmento `AutorFragment`.

```
class AutorFragment : Fragment(), AutorSelecionadoListener {
   private lateinit var viewModel: AutorViewModel
   override fun onCreateView(
       inflater: LayoutInflater, container: ViewGroup?,
       savedInstanceState: Bundle?
   ): View {
       viewModel = ViewModelProviders
                   .of(requireActivity())
                   .get(AutorViewModel::class.java)
       val binding = FragmentAutorBinding.inflate(inflater, container, false).apply
       {
           lifecycleOwner = this@AutorFragment
           executePendingBindings()
       }
       viewModel.fetch()
       val view = binding.root
       viewModel.autores.observe(this, Observer {
           binding.adapter = AutoresListAdapter(it, this)
       })
       return view
   }

   override fun onAutorSelecionado(posicao: Int) {
       viewModel.fetchDetail(posicao)
       val navId = R.id.autor_to_autor_detalhes
```

```
            Navigation.findNavController(view ?: return).navigate(navId)
        }
    }
```

Fonte: produção própria.

Reutiliza-se boa parte da classe de fragmento do *boilerplate*. Troca-se os nomes para o *DataBinding* usar o `viewModel` da aplicação. Troca-se também o `adapter` que o `RecyclerView` utiliza para o tratamento e a configuração dos items da lista. O `viewModel` é inicialmente usado para buscar a lista de autores. Além disso, observa-se seu *LiveData* `autores` e atribui-se um novo `AutoresListAdapter` ao `binding` sempre que a lista for modificada.

O fragmento implementa a interface `AutorSelecionadoListener` para então ser passada como parâmetro para o `adapter`. O método `onAutorSelecionado` realiza a navegação do fragmento `AutorFragment` para o fragmento `AutorDetalhesFragment`, que será definido a seguir. Nesse método é também usado o `viewModel` novamente para solicitar os detalhes do autor selecionado.

6.4.4 Construção do layout da tela de detalhes

Cria-se agora o layout da tela de detalhes de um autor:

Código 29 - O XML que representa o fragmento de detalhes de um autor.

```
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
>
    <data>
        <variable
            name="viewModel"
            type="io.schiar.tccboilerplate.viewmodel.AutorViewModel"
        />
    </data>
    <LinearLayout
        android:orientation="vertical"
        android:id="@+id/boilerplate"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
            tools:context=".view.AutorFragment"
        >
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="@dimen/content_size"
                app:label="@{@string/nome}"
                app:value="@{viewModel.autor.nome}"
            />
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="@dimen/content_size"
                app:label="@{@string/nascimento}"
                app:value="@{viewModel.autor.nascimento}"
            />
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="@dimen/content_size"
                app:label="@{@string/localDeNascimento}"
                app:value="@{viewModel.autor.localDeNascimento}"
            />
        </LinearLayout>
    </layout>
```

Fonte: produção própria.

Nota-se a reutilização do *BindAdapter* que contém no *boilerplate*, porém, para essa aplicação, utiliza-se o atributo `autor` presente no `AutorViewModel`. Seguem-se boas práticas usando o arquivo de `strings` do projeto para guardar as constantes de textos da aplicação.

Código 30 - O arquivo `strings.xml` localizado na pasta `res/`.

```
<resources>
  <string name="app_name">Autores</string>
  <string name="label_value">%1$s: %2$s</string>
  <string name="nome">Nome</string>
  <string name="nascimento">Data de Nascimento</string>
  <string name="localDeNascimento">Local de nascimento</string>
</resources>
```

Fonte: produção própria.

6.4.5 Construção do fragmento que mostra os detalhes de um autor

Com o layout criado, cria-se agora o fragmento que representa a tela que exibe os detalhes de um autor.

Código 31 - A classe de fragmento `AutorDetalhesFragment`.

```
class AutorDetalhesFragment : Fragment() {
   private lateinit var viewModel: AutorViewModel
   override fun onCreateView(
       inflater: LayoutInflater, container: ViewGroup?,
       savedInstanceState: Bundle?
   ): View {
       viewModel = ViewModelProviders
           .of(requireActivity())
           .get(AutorViewModel::class.java)
       val binding = FragmentAutorDetalhesBinding
           .inflate(inflater, container, false).apply {
               lifecycleOwner = this@AutorDetalhesFragment
               viewModel = this@AutorDetalhesFragment.viewModel
               executePendingBindings()
           }
       return binding.root
   }
}
```

Fonte: produção própria.

6.4.6 Construção do *Navigation*

Com os fragmentos definidos com seus respectivos layouts, *adapters* e interfaces necessárias, falta apenas criar o arquivo que define a navegação entre os dois fragmentos.

Código 32 - O arquivo `navigation.xml` localizado no diretório `res/navigation`.

```
<navigation
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
```

```
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/navigation"
    app:startDestination="@id/autor"
>
    <fragment
        android:id="@+id/autor"
        android:label="@string/app_name"
        android:name="io.schiar.tccboilerplate.view.AutorFragment"
        tools:layout="@layout/fragment_boilerplate"
    >
        <action
            android:id="@+id/autor_to_autor_detalhes"
            app:destination="@id/autor_detalhes"
            app:enterAnim="@anim/nav_default_enter_anim"
            app:exitAnim="@anim/nav_default_exit_anim"
            app:popEnterAnim="@anim/nav_default_pop_enter_anim"
            app:popExitAnim="@anim/nav_default_pop_exit_anim"
        />
    </fragment>
    <fragment
        android:id="@+id/autor_detalhes"
        android:label="@string/app_name"
        android:name="io.schiar.tccboilerplate.view.AutorDetalhesFragment"
    />

</navigation>
```

Fonte: produção própria.

Com o exemplo de navegação contido no *boilerplate*, foi apenas necessário editar os `id` dos fragmentos e o `id` do primeiro fragmento a renderizar.

O código inteiro está em um projeto e disponível em https://github.com/giovanischiar/tcc-boilerplate-example.

7. Avaliação

7.1 Definição da avaliação

Para avaliação da arquitetura proposta e do *boilerplate*, foi elaborado um documento de requisitos para uma aplicação simples que contém duas telas, duas requisições a um *webservice* e mostra os dados retornados nas requisições nessas duas telas. Este documento está disponível no apêndice 2. Foram desenvolvidas duas aplicações seguindo o documento de requisitos e utilizando como base apenas o *boilerplate* produzido e a documentação que o acompanha, a qual está disponível no apêndice 3. O objetivo da avaliação é analisar o código desenvolvido nessas aplicações utilizando métricas de qualidade de software definidas por outros autores e avaliar a eficácia ou não do uso do *boilerplate*.

7.2 Métricas aplicadas nos aplicativos resultantes

7.2.1 Coesão

Define-se uma classe coesa como uma classe especializada em desempenhar apenas um papel. Falta de coesão é uma característica de classes que desempenham mais de um papel. Essa é uma característica não desejável, pois desempenhar mais de um papel torna o código da classe mais difícil de entender, manter e testar as funcionalidades [47].

Baixa coesão significa que a classe pode ter sido má projetada ou ter alta complexidade. É desejável que a classe seja decomposta em uma ou mais classes para cada uma desempenhar um papel diferente. Dessa forma, uma classe coesa é desejável pois promove um bom encapsulamento. Em contrapartida, uma classe altamente coesa, com todos os métodos conectados, torna-se mais difícil fazer testes unitários [48].

7.2.1.1 Coesão Justa de Classe

Uma forma sistemática de medir a coesão de uma classe é medindo o quanto seus métodos se relacionam com seus atributos e entre si. A métrica da Coesão Justa de Classe (TCC, do inglês *Tight Class Cohesion*) é uma forma de medir a coesão de uma classe. Considerando os métodos não privados de uma classe, contamos o número de conexões diretas (NDC, do inglês *Number of Direct Connections*) entre os métodos da classe [47]. Os métodos A e B pertencentes a uma mesma classe estão conectados se:

- A e B referenciam um mesmo atributo;
- A e B chamam métodos que usam o mesmo atributo.

Considera-se também o número de conexões possíveis (NP, do inglês *Number of Possible Connections*) com a seguinte fórmula:

Equação 1 - Número de conexões possíveis (NP) [47].

$$NP = N * (N-1)/2$$

onde N é o número total de métodos da classe.

A fórmula da Coesão Justa de Classe pode ser então calculada como:

Equação 2 - Coesão justa de classe (TCC) [47].

$$TCC = NDC / NP$$

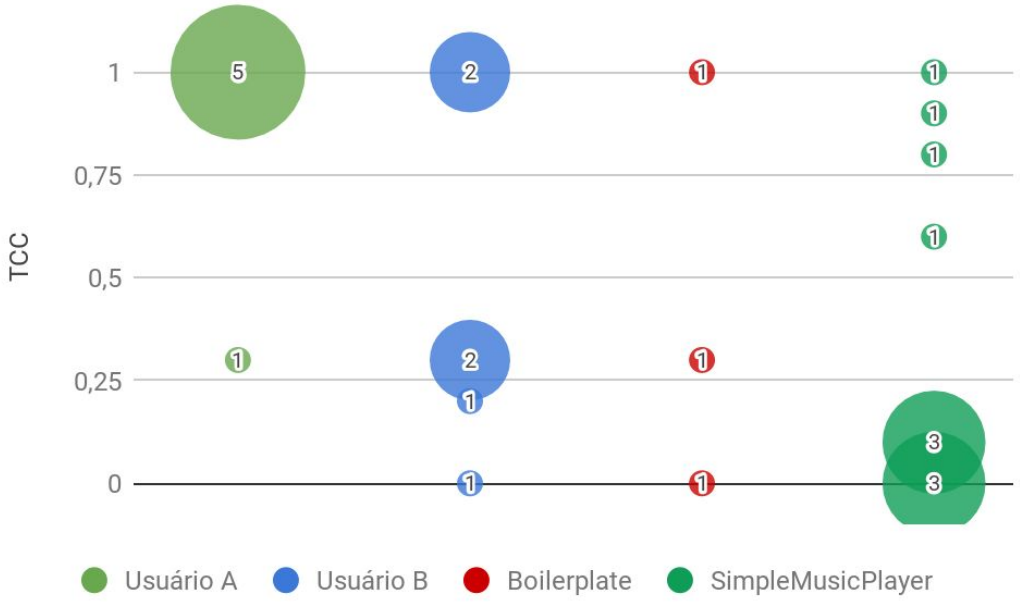
De acordo com os autores da métrica, uma classe com TCC menor que 0,5 é considerada uma classe com baixa coesão. Uma classe com TCC maior que 0,8 é considerada uma classe com alta coesão, sendo um TCC igual a 1 a coesão máxima, onde todos os métodos estão conectados [47]. Mesmo com a desvantagem de, possivelmente, dificultar a criação de testes unitários, define-se

um TCC igual a 1 como o melhor valor possível para essa métrica de alta coesão [48].

7.2.1.2 Aplicação da métrica

Foi aplicada a métrica nas aplicações desenvolvidas, doravante chamados Aplicação A e Aplicação B. A métrica também foi aplicada no *boilerplate* em si, que, apesar de ser uma estrutura genérica, consiste em uma aplicação modelo, a qual deve ser substituída pela aplicação real do usuário. Portanto, acredita-se ser interessante a aplicação da métrica no *boilerplate* também. Além disso, como forma de comparar a coesão das aplicações criadas a partir do *boilerplate* com aplicações Android reais, a métrica também foi aplicada a uma aplicação Android chamada *SimpleMusicPlayer*. Essa aplicação é um tocador de músicas, cujo código está disponível em https://github.com/alexdemagalhaes/simple-music-player como software livre.

Gráfico 1 - Quantidade de classes e TCC correspondentes.



Fonte: produção própria.

Como é uma métrica aplicada a cada classe, o gráfico 1 mostra no eixo *y* os valores obtidos de TCC, enquanto o tamanho das bolhas indica a quantidade de classes que possuem aquele valor de TCC, dentre as classes elegíveis a serem analisadas por essa métrica (classes sem métodos ou com apenas 1 método não são possíveis de serem analisadas pela métrica). Os valores foram arredondados de forma a terem apenas uma casa decimal após a vírgula, para simplificar o gráfico.

Nota-se que usuário B e o *boilerplate* tiveram uma classe sem coesão cada. Os motivos disso acontecer, no caso do usuário B, é que foram necessárias algumas sobrescritas de método no fragmento para suportar funcionalidades que foram necessárias no desenvolvimento e que não necessitavam acessar atributos. Métodos que não acessam atributos da classe prejudicam a nota da classe nessa métrica de coesão. No caso do *boilerplate*, como o projeto é simples, um exemplo de navegação para demonstrar a funcionalidade do *Navigation* resultou na sobrescrita de um método que não usasse atributos da classe do fragmento. Portanto, nota-se que há casos em que sobrescrever métodos de superclasses do framework Android com o objetivo de utilizar funcionalidades do framework pode tornar uma classe menos coesa, de acordo com essa métrica.

Medir a coesão de uma aplicação é difícil [49]. Em geral, porém, nota-se que a maior parte das classes criadas pelos usuários possuem um alto nível de coesão e tiveram um bom resultado na métrica utilizada. É interessante também tentar verificar se o resultado positivo da métrica foi uma consequência da utilização do boilerplate ou, principalmente, mérito do programador.

No caso da aplicação B, todas as classes analisadas pela métrica foram classes derivadas do *boilerplate*, nas quais quem desenvolveu realizou as modificações necessárias para servir aos seus propósitos. As demais classes não eram complexas o suficiente para serem analisadas pela métrica (só possuem 1 método ou nenhum método).

Olhando o resultado da métrica aplicada à aplicação A, apenas 1 classe criada por quem desenvolveu foi elegível à aplicação da métrica. E, mesmo essa classe, obteve a coesão máxima, indicando que as habilidades de quem desenvolveu ajudaram no resultado da métrica. Porém, ainda pode-se notar que a

maior parte do resultado positivo da métrica foi consequência da organização do código das classes derivadas do *boilerplate*.

É possível também comparar o resultado das duas aplicações criadas com a aplicação *SimpleMusicPlayer*, que não foi desenvolvida seguindo o *boilerplate* nem a arquitetura proposta. Nota-se que boa parte das classes se aproximam da baixa coesão, evidenciando que a aplicação possui, em sua maioria, classes com baixa coesão, característica não desejada em aplicações.

7.2.2 Acoplamento

Define-se acoplamento entre classes quando uma classe possui uma referência a outra, possuindo acesso a métodos e/ou atributos da outra. Um programa com classes muito acopladas umas às outras resulta em classes de difícil reuso e encapsulamento, e também dificulta a criação de testes unitários. Quanto menos acoplamento tiver, mais reusáveis são as classes do programa e mais fácil fica criar testes unitários sem muita necessidade de criar *mocks* para as classes dependentes.

7.2.2.1 Fator de acoplamento

O fator de acoplamento (CF - do inglês *Coupling Factor*) é um fator utilizado para a medição do quão as classes de um programa estão acopladas entre si [49]. É definido como um grafo onde os vértices são classes e as arestas são o acoplamento entre as classes. Itera-se nesse grafo contando todas as arestas e os vértices. A fórmula fica a seguinte:

Equação 3 - Fator de acoplamento [50].

$$CF = \frac{\sum_{i=1}^{TC}\left[\sum_{i=1}^{TC} \text{é\_}Cliente(C_i, C_j)\right]}{TC^2 - TC}$$

Onde TC é a quantidade de classes da aplicação, e C_i uma classe específica (um vértice no grafo). O cálculo $\text{é\_}Cliente(C_i, C_j)$ é definido da seguinte

forma: 1 se C_i possui uma referência a um método ou atributo de uma instância de C_j, ou seja, elas estão acopladas entre si, e 0 caso contrário (não estão acopladas).

Houve uma revisão nessa definição e os autores decidiram levar em consideração o $DC(C_i)$, que são os descendentes da classe C_i. A nova fórmula estendida então ficou:

Equação 4 - Fator de acoplamento estendida [50].

$$CF = \frac{\sum_{i=1}^{TC}\left[\sum_{i=1}^{TC} \text{é\_}Cliente(C_i, C_j)\right]}{TC^2 - TC - 2\left[\sum_{i=1}^{TC} DC(C_i)\right]}$$

Os programas desenvolvidos nessa avaliação não definiram novas relações de herança além das impostas pelas classes do framework Android e do Android Jetpack. Portanto, esse novo componente na fórmula estendida sempre será 0 nas aplicações desenvolvidas pelos usuários e a versão estendida da fórmula não afetou os resultados obtidos nelas.

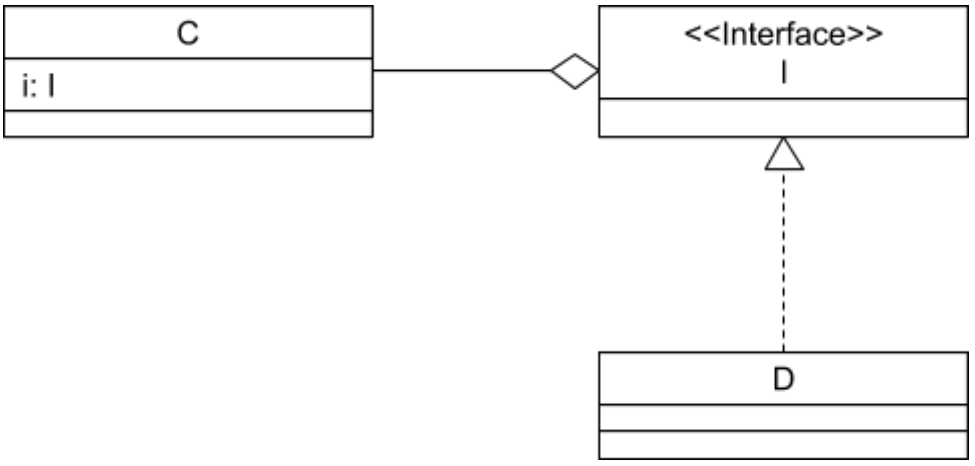
7.2.2.2 Aplicação da métrica

Ao utilizar a métrica original, notou-se algumas dificuldades. Quando se utiliza essa métrica para programas Kotlin, é preciso definir algumas convenções. Em Kotlin, o uso de interfaces permite que classes apenas possuam um acoplamento com um contrato, não com as classes que de fato implementam esse contrato. Porém, a métrica original foi criada para programas em C++, os quais não possuem exatamente o mesmo mecanismo de interfaces. Para a aplicação da métrica foram adotados as seguintes convenções:

Para o cálculo de $\text{é\_}Cliente(C_i, C_j)$, as interfaces são consideradas como classes apenas em C_j, ou seja, não são avaliados os acoplamentos de interface-classe ou interface-interface. Logo, apenas são levados em consideração o acoplamento classe-interface e classe-classe. Para o cálculo de TC, não são consideradas as interfaces.

Dessa forma, se uma instância de uma classe C possui como atributo a interface I que é implementada por D, considera-se que existe um acoplamento entre C e D, mesmo que a instância de C só conheça I e não saiba que, internamente, lá existe uma instância de D, como ilustra a figura 21.

Figura 21 - A classe C possui a interface I, implementada por D.



Fonte: produção própria.

Embora um acoplamento entre classe e interface seja mais desejável do que entre classes concretas, decidiu-se dessa forma para não alterar a fórmula original, onde a fórmula originalmente foi aplicada em uma linguagem que não existe o conceito de interface.

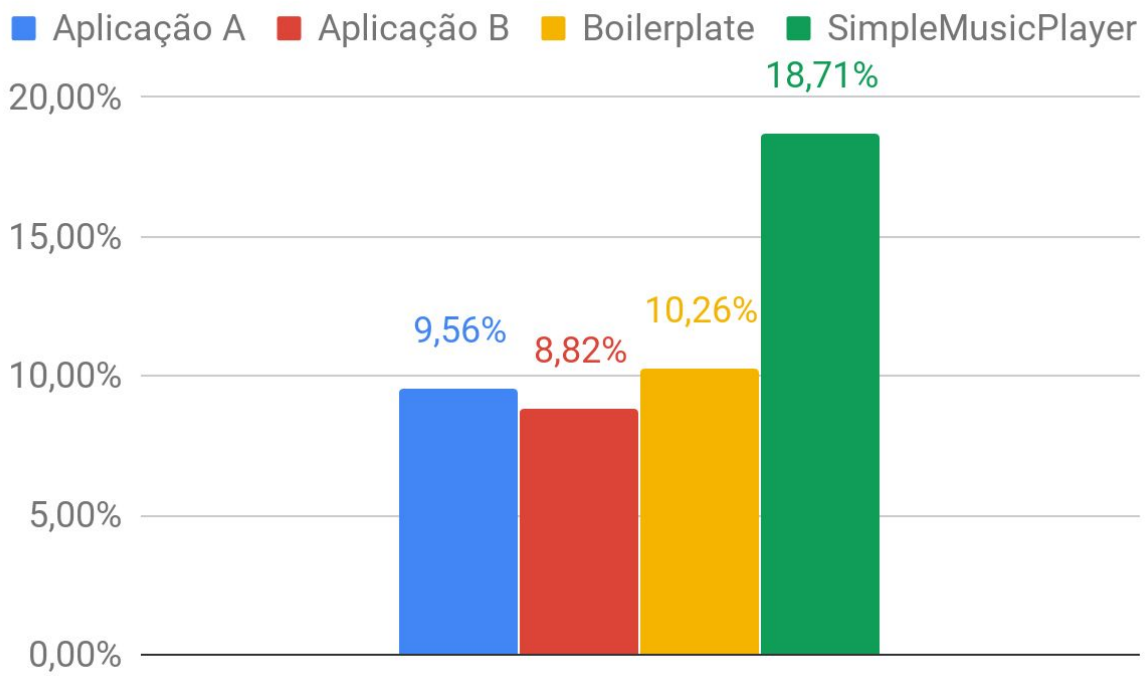
Utilizando essas convenções, aplicou-se a métrica no *boilerplate* em si, nas aplicações desenvolvidas A e B, e também na aplicação *SimpleMusicPlayer*.

A tabela 1 mostra a média, o valor mínimo e o valor máximo obtidos dessa métrica aplicada pelos autores da métrica a 5 artefatos de software diferentes. Em média, os artefatos de software avaliados por eles obtiveram fator de acoplamento de 10,8%.

O gráfico 2 mostra o valor de acoplamento do *boilerplate*, dos programas dos usuários A e B, e da aplicação *SimpleMusicPlayer*. Nota-se que os fatores de acoplamento dos dois programas desenvolvidos ficaram menores do que o do próprio *boilerplate*. Esse resultado mostra que o código novo adicionado durante a criação das aplicações, tanto em classes diretamente derivadas do *boilerplate*

quanto em classes novas, não adicionou mais acoplamento à aplicação, e, portanto, não prejudicou as métricas de acoplamento.

Gráfico 2 - Valor em porcentagem de CF para cada programa.



Fonte: produção própria.

Tabela 1 - Comparação de CF em programas, com o mínimo, a média e o máximo [50].

	Mínimo	Média	Máximo
CF	3,9%	10,8%	17,7%

Fonte: produção própria.

Nota-se também que os dois programas ficaram com o fator de acoplamento abaixo da média de 10,8% obtida pelos autores da métrica, sendo o *boilerplate* inicial o que possui maior índice (10,26%), pela pouca quantidade de classes. Valores abaixo do máximo obtido pelos autores, de 17,7%, são considerados valores aceitáveis [50]. Como as aplicações desenvolvidas obtiveram fator de acoplamento abaixo da média e bem distante no valor máximo, considera-se que elas possuem baixo acoplamento.

A aplicação *SimpleMusicPlayer*, por outro lado, obteve um fator de acoplamento muito superior às aplicações desenvolvidas sob o *boilerplate*, mesmo possuindo razoavelmente uma complexidade semelhante, evidenciando que a boa arquitetura desenvolvida no *boilerplate* incentivou quem desenvolveu a criar programas com um fator de acoplamento inferior à aplicação que não utilizou o *boilerplate* como base.

Assim como foi feito na métrica de coesão, é interessante tentar verificar se o resultado positivo da métrica foi uma consequência do *boilerplate* ou das habilidades de quem desenvolveu. Essa comparação é mais difícil na métrica de acoplamento, visto que as classes se relacionam entre si e é difícil analisar separadamente apenas as classes derivadas do *boilerplate* ou criadas pelo programador. Porém, olhando para as classes criadas e as derivadas diretamente do *boilerplate*, nota-se que 77% das classes da aplicação A e 83% das classes da aplicação B foram derivadas diretamente do *boilerplate*. Com base nisso, nota-se que a maioria das classes das aplicações foram derivadas diretamente do *boilerplate*, e, consequentemente, grande parte do resultado da métrica de acoplamento é diretamente influenciado pelo uso do *boilerplate*.

Outro possível indício de que o *boilerplate* pode ajudar no desenvolvimento de uma aplicação com características desejáveis de acoplamento é o fato do fator de acoplamento das duas aplicações criadas terem ficado bem abaixo do fator de acoplamento da aplicação *SimpleMusicPlayer*, a qual não foi criada com base no *boilerplate*.

Esses resultados podem dar o indício de que o *boilerplate* incentivou o desenvolvimento da aplicação com uma característica de baixo acoplamento, embora não descartam a possibilidade de que as habilidades de quem desenvolveu tenha influenciado na qualidade da aplicação.

8. Conclusão

O objetivo de propor uma arquitetura e construir um *boilerplate* para o desenvolvimento de aplicações Android é justificável quando se afirma que o Android tornou-se o sistema operacional dominante no mercado e a comunidade de desenvolvedores da plataforma enfrenta dificuldades em organizar o código de suas aplicações. Sendo um sistema operacional dominante, uma arquitetura adequada e um *boilerplate* que ilustra seu uso podem potencialmente ser uma solução para muitos programadores.

8.1 Resultados

No desenvolvimento da arquitetura, os Componentes de Arquitetura do Android Jetpack, lançados recentemente, se mostraram apropriados para serem usados numa arquitetura para o desenvolvimento de aplicações Android. Com esse estudo, percebeu-se que uma aplicação desenvolvida com uma única atividade e múltiplos fragmentos se tornou mais recomendado do que uma aplicação com múltiplas atividades, além de permitir a utilização do componente *Navigation* que provém do Android Jetpack. Basear a arquitetura no padrão de arquitetura MVVM também mostrou-se adequado, visto que os Componentes de Arquitetura fornecem as peças para a utilização do MVVM.

O Android evoluiu bastante com esses Componentes de Arquitetura, definindo um guia para a construção de uma arquitetura, embora esses componentes, sozinhos, não definam uma arquitetura de fato. Essa arquitetura desenvolvida precisa então determinar como usar esses componentes e quais boas práticas no seu uso devem ser encorajadas.

Os Componentes de Arquitetura serviram então como as peças de um quebra-cabeça, onde a arquitetura definida no *boilerplate* colocou as peças que faltavam e resultou em uma arquitetura simples, porém completa.

As classes e arquivos desse *boilerplate* foram organizados de forma tão específica quanto necessário para que a arquitetura apresentada possa ser

compreendida, e tão genérica quanto possível para que desenvolvedores possam usá-los no desenvolvimento de qualquer aplicação Android.

Essa arquitetura proposta neste trabalho e definida no *boilerplate* foi avaliada e os resultados mostram que ela é capaz de ajudar os desenvolvedores, tanto iniciantes quanto experientes, a evitar construir as aplicações em volta de atividades, resultando em aplicações com código mais bem distribuído e coeso.

A arquitetura MVVM mostrou-se adequada ao utilizá-la com os componentes do Android Jetpack, e a utilização da linguagem Kotlin permitiu uma grande expressividade para o código que integra os Componentes de Arquitetura presentes no Android Jetpack. A linguagem foi bem aceita pela Google, e atualmente na documentação do Android Jetpack existem diversos exemplos de como utilizar os componentes do Android Jetpack na linguagem Kotlin. Os conceitos de Kotlin apresentados no capítulo de Kotlin cobriu todos os trechos de código apresentados no capítulo de desenvolvimento.

Foram aplicadas métricas de acoplamento e coesão nas aplicações desenvolvidas e foram obtidos resultados satisfatórios com todas as aplicações. Contudo, há limitações que poderiam ser resolvidas em trabalhos futuros.

Pode-se dizer que os objetivos inicialmente descritos foram cumpridos. Os desafios de arquitetura de desenvolver aplicativos Android foram compreendidos e foram feitas análises dos padrões de arquitetura atuais para escolher o padrão de arquitetura que mais se encaixa, considerando-se as vantagens de desvantagens. Estudou-se a fundo os componentes de arquitetura presentes no Android Jetpack, resultando na utilização desses componentes no *boilerplate*, bem como estudou-se também a linguagem de programação Kotlin. O *boilerplate* foi, então, desenvolvido utilizando todas as tecnologias estudadas. Foi criado e exemplificado um programa utilizando o boilerplate. Elaborou-se um documento de requisitos de uma aplicação. Desenvolveu-se duas aplicações utilizando o boilerplate. Analisou-se as aplicações desenvolvidas para verificar indícios de contribuição do *boilerplate* para a boa qualidade das aplicações desenvolvidas. Por fim, o *boilerplate* está disponível em um repositório público no *GitHub* no endereço https://github.com/giovanischiar/tcc-boilerplate, para qualquer usuário programador

utilizar, cumprindo-se assim o último objetivo de disponibilizá-lo à comunidade desenvolvedora.

8.2 Limitações

Ao longo da avaliação feita do *boilerplate* e da arquitetura propostos, foram percebidas algumas limitações e oportunidades de melhoria na solução.

Embora a arquitetura e os diagramas que a acompanham mostram todos os seus componentes e a interação entre eles, foram notadas algumas lacunas no código *boilerplate* desenvolvido. O *boilerplate*, por ser simples, não conta com exemplos de classes de lógica de negócio, nem exemplos de fontes de dados com acesso a banco de dados e requisições a *webservices*; portanto, não há clareza em como implementar a lógica dessa camada, deixando o usuário decidir.

Pela simplicidade do *boilerplate*, a comunicação entre os *ViewModels* também não é definida de forma clara. É orientado no texto que exista um *ViewModel* para cada tipo de dado no domínio da aplicação, mas não se especifica como que a comunicação entre domínios é feita, deixando para o programador resolver. Essa comunicação pode ser feita através das classes de lógica de negócio, ou através dos *ViewModels* acessando múltiplos repositórios. Essas opções provavelmente exigiriam que um mecanismo de compartilhamento de instâncias fosse implementado, como, por exemplo, transformar algumas classes de lógica de negócio ou repositórios em *singletons*. Outra abordagem seria realizar a troca de pequenos dados entre os fragmentos utilizando o recurso do *Navigation*, o que não foi explorado por esse estudo.

O *boilerplate*, apesar da documentação fornecida, não conta com um manual detalhado sobre sua utilização. O arquivo *README.md*, presente na página do *GitHub* onde está disponível o código do *boilerplate* e no apêndice 3, cumpre parcialmente essa necessidade, porém de forma mais superficial.

Por fim, as aplicações criadas não são aplicações complexas e de larga escala, o que pode ter feito com que algumas deficiências da arquitetura ou do *boilerplate* não tenham ficado visíveis.

8.3 Trabalhos futuros

Para trabalhos futuros, é interessante tratar as limitações identificadas. Por exemplo, pode-se adicionar mais diagramas à documentação, referentes a aspectos dinâmicos, exemplificando o fluxo da aplicação. Quanto ao *boilerplate* em si, pode-se adicionar uma orientação, através de exemplos, na comunicação entre repositórios e fontes de dados. A criação de um programa gerador de *boilerplate*, com um nome de pacote personalizado, também é algo pertinente, pois a edição do nome do pacote de um projeto Android é ligeiramente complicada.

É interessante realizar a avaliação do boilerplate com usuários programadores e aplicar um questionário para os usuários avaliarem o uso do *boilerplate.* Até mesmo, inicialmente, solicitar que a aplicação seja feita sem oferecer o *boilerplate* e, num outro momento, fazer a aplicação com o *boilerplate* para fins de comparação. Não foi possível esta pesquisa ser realizada por não ser possível a autorização do comitê de ética a tempo de cumprir o cronograma da pesquisa.

Acredita-se, então, que é interessante haver novas iterações no desenvolvimento da arquitetura e do *boilerplate*, de modo que possíveis deficiências sejam tratadas e disponibilizadas para a comunidade Android.

Por fim, acredita-se que a arquitetura proposta e o *boilerplate* que a acompanha resolvem diversos problemas que desenvolvedores Android enfrentam no dia-a-dia, e servem tanto como uma boa base para a criação de novas aplicações Android, como também como um guia de boas práticas no desenvolvimento de aplicações Android.

8.4 Considerações finais

Considera-se o estudo feito sobre as tecnologias e conceitos de programação para o desenvolvimento do *boilerplate* aqui desenvolvido muito interessante. Foram necessários muitos conceitos aprendidos em disciplinas do currículo de Ciências da Computação para o desenvolvimento deste trabalho. Embora não aprenda-se especificamente sobre programação Android, os conceitos aprendidos nas

disciplinas foram utilizados para o estudo da melhor arquitetura para se aplicar à programação Android.

Considera-se interessante este *boilerplate*, aqui desenvolvido, por oferecer a oportunidade a desenvolvedores Android de desenvolver aplicações melhor estruturadas.

E, por tratar-se de um *boilerplate* para o desenvolvimento de aplicação para a plataforma Android, com essa plataforma crescendo consideravelmente, acredita-se que a escolha da plataforma para a criação deste *boilerplate* fora assertiva, pois aumenta a utilidade do *boilerplate* aqui produzido.

Referências

[1] MOBILE Operating System Market Share Worldwide | StatCounter Global Stats. [2017 - 2018]. Disponível em: <http://gs.statcounter.com/os-market-share/mobile/worldwide>. Acesso em: 26 nov. 2018.

[2] JEMEROV, Dmitry et al. **Using Kotlin for Android Development**. [2017-2018]. Disponível em: <https://kotlinlang.org/docs/reference/android-overview.html>. Acesso em: 26 nov. 2018.

[3] SHAH, Yash; SHAH, Jimil; KANSARA, Krishna. Code obfuscating a Kotlin-based App with Proguard. **Second International Conference on Advances in Electronics, Computers and Communications (ICAECC)**, IEEE, p. 1-5, 2018.

[4] DÜRSCHMID, M. T.; DÖLLNER, J. Towards Architectural Styles for Android App Software Product Lines. **IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)**, Buenos Aires, p. 58-62, 2017.

[5] DUPREE, Kevin. **MVPR: A Flexible, Testable Architecture for Android**. 7 de Jul 2015. Disponível em <https://www.philosophicalhacker.com/2015/07/07/mvpr-a-flexible-testable-architecture-for-android-pt-1/>. Acesso em 4 jun. 2019.

[6] IDESIS, Stanley. **Develop Your First Android Application**: Learn the Model-View-Controller Pattern. 2018. Disponível em: <https://openclassrooms.com/en/courses/4661936-develop-your-first-android-application/4679186-learn-the-model-view-controller-pattern>. Acesso em: 26 nov. 2018.

[7] DAOUDI, Aymen et al. An exploratory study of MVC-based architectural patterns in Android apps. In: **Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing**. ACM, 2019. p. 1711-1720.

[8] Use Android Jetpack to Accelerate Your App Development. Disponível em: <https://android-developers.googleblog.com/2018/05/use-android-jetpack-to-accelerate-your.html>. Acesso em: 3 jun. 2019.

[9] BERGSTROM, Lukas. **Announcing Architecture Components 1.0 Stable.** Disponível em: <https://android-developers.googleblog.com/2017/11/announcing-architecture-components-10.html>. Acesso em: 4 jun. 2019.

[10] LARDINOIS, Frederic. **Kotlin is now Google's preferred language for Android app development.** Disponível em: <https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/>. Acesso em: 4 jun. 2019.

[11] Desenvolvedores Android - Tecnology - Arquitetura da plataforma. Disponível em: <https://developer.android.com/guide/platform>. Acesso em: 3 jun. 2019.

[12] SIMPSON, Ronnie. **Android overtakes Windows for first time**: "Milestone in technology history and end of an era" as Microsoft no longer owns dominant OS. 2017. Disponível em: <http://gs.statcounter.com/press/android-overtakes-windows-for-first-time>. Acesso em: 26 nov. 2018.

[13] LI, L.; BISSYANDÉ, T. F; PAPADAKIS, M; RASTHOFER, S; BARTEL, A.; OCTEAU, D.; KLEIN, J.; TRAON, L. Static analysis of android apps: A systematic literature review. **Information and Software Technology**, v. 88, p. 67-95, 2017.

[14] Desenvolvedores Android - Docs - Guias - Introduction to Activities. Disponível em: <https://developer.android.com/guide/components/activities/intro-activities>. Acesso em: 3 jun. 2019.

[15] YADAV, R.; BHADORIA, R. S. Performance Analysis for Android Runtime Environment. **Fifth International Conference on Communication Systems and Network Technologies**, Gwalior, p. 1076-1079, 2015.

[16] Atividades | Android Developers. Disponível em: <https://developer.android.com/guide/components/activities.html?hl=pt-BR>. Acesso em 5 jun. 2019.

[17] Desenvolvedores Android - Docs - Guias - Fundamentos de aplicativos. Disponível em: <https://developer.android.com/guide/components/fundamentals>. Acesso em: 3 jun. 2019.

[18] Visão geral da IU | Android Developers. Disponível em: <https://developer.android.com/guide/topics/ui/overview.html?hl=pt-BR>. Acesso em: 4 jun. 2019.

[19] Context | Android Developers. Disponível em: <https://developer.android.com/reference/android/content/Context>. Acesso em: 4 jun. 2019.

[20] KHANNA, Gaurav. **Mastering Android context.** 5 de Jun 2018. Disponível em: <https://www.freecodecamp.org/news/mastering-android-context-7055c8478a22/>. Acesso em: 5 jun. 2019.

[21] SOKOLOVA, Karina; LEMERCIER, Marc; GARCIA, Ludovic. Android passive MVC: a novel architecture model for the android application development. **International Conference on Pervasive Patterns and Applications**. 2013.

[22] MUNTENESCU, Florina. **Android Architecture Patterns Part 1: Model-View-Controller**. 1 de Nov 2016. Disponível em <https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6>. Acesso em 4 jun. 2019.

[23] MAXWELL, Eric. **The MVC, MVP, and MVVM Smackdown**. 2017. Disponível em: <https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/>. Acesso em: 26 nov. 2018.

[24] Build effective unit tests | Android Developers. Disponível em: <https://developer.android.com/training/testing/unit-testing/>. Acesso em: 5 jun. 2019.

[25] FOWLER, Martin. **GUI Architectures**. 2006. Disponível em: <https://www.martinfowler.com/eaaDev/uiArchs.html>. Acesso em: 2 jun. 2019.

[26] MOORE, Dana; BUDD, Raymond; BENSON, Edward. **Professional Rich Internet Applications: AJAX and Beyond**. John Wiley & Sons, 2007.

[27] GOSSMAN, John. **Introduction to Model/View/ViewModel pattern for building WPF apps**. 8 de out. 2015. Disponível em <https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/>. Acesso em 5 jun. 2019.

[28] Entendendo o Pattern Model View ViewModel MVVM**.** Disponível em: <https://www.devmedia.com.br/entendendo-o-pattern-model-view-viewmodel-mvvm/18411>. Acesso em: 5 jun. 2019.

[29] The MVVM Pattern | Microsoft Docs. Disponível em <https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10)>. Acesso em 5 jun. 2019.

[30] DORFMANN, Hannes. **Model-View-Intent on Android**. 4 de Mar 2016. Disponível em <http://hannesdorfmann.com/android/model-view-intent>. Acesso em 5 jun. 2019.

[31] MOBILE, Mutual. **Meet VIPER**: Mutual Mobile's application of Clean Architecture for iOS apps. 2014. Disponível em: <https://mutualmobile.com/resources/meet-viper-fast-agile-non-lethal-ios-architecture-framework>. Acesso em: 27 nov. 2018.

[32] MARTIN, Robert C. **The Clean Architecture**. 2012. Disponível em: <http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>. Acesso em: 27 jun. 2018.

[33] Guia para a arquitetura do app | Android Developers. Disponível em <https://developer.android.com/jetpack/docs/guide#overview>. Acesso em 5 jun. 2019.

[34] Android Architecture Components | Android Developers. Disponível em <https://developer.android.com/topic/libraries/architecture/>. Acesso em 5 jun. 2019.

[35] Guia para a arquitetura do app | Android Developers. Disponível em <https://developer.android.com/jetpack/docs/guide>. Acesso em 5 jun. 2019.

[36] ViewModel Overview | Android Developers. Disponível em <https://developer.android.com/topic/libraries/architecture/viewmodel>. Acesso em 5 jun. 2019.

[37] Create a List with RecyclerView | Android Developers. Disponível em <https://developer.android.com/guide/topics/ui/layout/recyclerview>. Acesso em 5 jun. 2019.

[38] STALTZ, André. **The introduction to Reactive Programming you've been missing**. 2014. Disponível em: <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Acesso em: 27 jun. 2018.

[39] NG, Karen. **Android Developers Blog: What's New with Android Jetpack and Jetpack Compose**. 8 mai. de 2019. Disponível em <https://android-developers.googleblog.com/2019/05/whats-new-with-android-jetpack.html>. Acesso em 5 jun. 2019.

[40] JEMEROV, Dmitry et al. **Basic Syntax**. [2016-2018]. Disponível em: <https://kotlinlang.org/docs/reference/basic-syntax.html>. Acesso em: 26 nov. 2018.

[41] ATAMAS, Semyon et al. **Null Safety**. [2016-2018]. Disponível em: <https://kotlinlang.org/docs/reference/null-safety.html>. Acesso em: 26 nov. 2018.

[42] ATAMAS, Semyon et al. **Classes and Inheritance**. [2016-2018]. Disponível em: <https://kotlinlang.org/docs/reference/classes.html>. Acesso em: 26 nov. 2018.

[43] BRESLAV, Andrey et al. **Dynamic Type**. [2014-2018]. Disponível em: <https://kotlinlang.org/docs/reference/dynamic-type.html>. Acesso em: 4 jun. 2019.

[44] IGUSHKIN, Sergey et al. **Higher-Order Functions and Lambdas.** [2016-2018]. Disponível em <https://kotlinlang.org/docs/reference/lambdas.html>. Acesso em 5 jun. 2019.

[45] TSVETINOV, Nickolay. **Learning reactive programming with Java 8**. Packt Publishing Ltd, 2015.

[46] Observer Design Pattern. Disponível em: <https://sourcemaking.com/design_patterns/observer>. Acesso em: 5 jun. 2019.

[47] BIEMAN, James M.; KANG, Byung-Kyoo. Cohesion and reuse in an object-oriented system. **ACM SIGSOFT Software Engineering Notes**, v. 20, n. si, p. 259-262, 1995.

[48] **Cohesion metrics**. Disponível em: <https://www.aivosto.com/project/help/pm-oo-cohesion.html>. Acesso em: 5 jun. 2019.

[49] FONSECA, Wannessa Rocha da. **Ferramenta de extração de métricas para apoio à avaliação de especificações orientadas a objetos**. 2002. Dissertação (Mestrado em Ciência da Computação) – Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis, 2002.

[50] ABREU, F. Brito; GOULÃO, Miguel; ESTEVES, Rita. Toward the design quality evaluation of object-oriented software systems. **Proceedings of the 5th International Conference on Software Quality**, Austin, Texas, USA, p. 44-57, 1995.

[51] **Boilerplate | Definition of Boilerplate by Merriam-Webster**. Disponível em: <https://www.merriam-webster.com/dictionary/boilerplate>. Acesso em: 10 jul. 2019.

Apêndice 1 - Kotlin

A linguagem Kotlin é uma das linguagens suportadas oficialmente para o desenvolvimento de aplicações Android [2]. A comunidade de desenvolvedores Android atualmente recomenda e encoraja o uso da linguagem Kotlin tanto para a criação de novas aplicações Android como também para manutenção de aplicações existentes [10].

Pela presença de construções que facilitem o uso do paradigma de programação reativa e também seguindo as recomendações da comunidade Android, o *boilerplate* proposto neste trabalho foi todo desenvolvido utilizando a linguagem Kotlin. Porém, por Kotlin ter sido criada há poucos anos e ser menos conhecida do que a linguagem Java, alguns aspectos da linguagem podem ser mais difíceis de compreender para o desenvolvedor não habituado com a linguagem. Neste capítulo, então, serão descritos aspectos da linguagem que diferem de Java, de forma que o leitor possa se familiarizar com a linguagem e compreender todos os aspectos do código *boilerplate* proposto.

Kotlin é uma linguagem de programação orientada a objetos com princípios de programação funcional. Ela pode ser compilada para a máquina virtual Java, permitindo seu uso para o desenvolvimento de aplicativos Android, e também pode ser compilada para JavaScript ou para binário nativo através do backend da LLVM. Algumas vantagens de utilizar Kotlin para o desenvolvimento de aplicações Android são [2]:

- Compatibilidade: Kotlin é totalmente compatível com JDK 6, assegurando que poderá executar em dispositivos mais antigos sem problemas. É totalmente suportada pelo Android Studio, principal IDE de desenvolvimento Android, e também é compatível com o sistema de compilação do Android.
- Performance: Uma aplicação Kotlin executa tão rápido quanto uma equivalente em Java graças à estrutura do *bytecode* ser similar.

- Interoperabilidade: Kotlin é 100% interoperável com Java, permitindo-se utilizar todas as bibliotecas Java já existentes em uma aplicação Kotlin.
- Rastros: Kotlin possui uma biblioteca de *runtime* bem enxuta. Em uma aplicação real são adicionados uns cem métodos a mais e menos de 100KB ao tamanho do `apk`, arquivo final da compilação de um programa Android.

1.1 Características da linguagem

1.1.1 Sintaxe

1.1.1.1 Declarando variáveis e constantes

A sintaxe para declaração de variáveis e constantes difere de Java na ordem de declaração. Além disso, Kotlin consegue inferir o tipo pelo literal ou valor atribuído à variável [40]. Exemplo de constante:

```
val b = 10; // constante "b" de tipo Int inferido pela sua atribuição
b = 10; // erro! constante do tipo Int "c" não pode ser reatribuída
```

Segue exemplo de variáveis:

```
var a: Int // variável de nome "a" e do tipo Int
a = 10 // reatribuição permitida da variável "a"
```

Como pode-se notar, o tipo é declarado após o nome da variável, diferentemente de Java. Além disso, não há necessidade de declaração explícita de tipo ao ser declarada e inicializada a variável ou constante.

1.1.1.2 Declarando funções

A palavra reservada `fun` é utilizada para declaração de funções. Segue um exemplo de uma função de nome `exemplo` com dois parâmetros do tipo `Int` e retorno `Int`:

```
fun exemplo(a: Int, b: Int): Int { /* corpo da função */ }
```

1.1.2 Segurança de tipo (null safety)

O sistema de tipos de Kotlin tem como um dos objetivos eliminar o perigo de referências nulas do código [41].

Um dos maiores problemas na programação Java é quando tenta-se acessar métodos de uma referência nula, resultando em uma exceção de referência nula, que em Java é chamada de `NullPointerException`, ou NPE.

O sistema de tipo do Kotlin tem como um dos objetivos eliminar a ocorrência de NPE na execução do código. Em Kotlin só há poucas formas de haver NPE, entre elas:

- Uma chamada explícita usando `throw NullPointerException()`;
- Uso do operador `!!`, descrito a seguir;
- Em alguns casos na interoperação com Java;

No Kotlin, o sistema de tipos diferencia entre referências que podem ser nulas (referências *nullables*) e as que não podem ser nulas (referências não-nulas). Por exemplo, uma variável do tipo `String` que não pode ser nula:

```
var a: String = "abc"
a = null // erro de compilação
```

Para permitir que uma variável seja nula, é necessário explicitamente declarar a variável como sendo do tipo `String?` (note o símbolo "`?`" depois do nome do tipo, representando possível nulidade).

```
var b: String? = "abc"
b = null // sem erros de compilação
```

```
print(b)
```

Como a não pode ser nula, se um método ou propriedade da variável for acessado, é garantido que não cause um NPE. Pode-se então seguramente fazer:

```
val l = a.length
```

Porém, no caso da variável b, a mesma pode ser nula, e acessar alguma propriedade dela diretamente como mostrado acima não seria seguro. Logo, o código a seguir possui um erro de compilação:

```
val l = b.length // error: variable 'b' can be null
```

A seguir pode-se ver como é possível acessar métodos ou propriedades de variáveis cujo valor pode ser nulo.

1.1.2.1 Verificando nulidade em condições

Primeiramente, para acessar propriedades e métodos de referências que podem ser nulas, é possível explicitamente verificar a nulidade usando um condicional:

```
val l = if (b != null) b.length else -1
```

Se a condição for verdadeira, propriedades e métodos da variável podem ser acessados diretamente de forma segura.

1.1.2.2 Chamadas seguras (safe calls)

Uma outra forma de poder chamar métodos ou acessar propriedades de b é utilizando um operador especial, o operador de chamadas seguras (safe call) expresso no código por `.?`.

```
val a = "Kotlin"
val b: String? = null
```

```
println(b?.length)
println(a?.length)
```

O código acima imprime o valor de `b.length` se `b` não for nulo e `null` caso contrário. O tipo dessa expressão é `Int?`.

Safe calls são úteis em chamadas encadeadas. Por exemplo: se Bob, um empregado, pode estar designado a um departamento (ou não) que, por sua vez, pode ter outro empregado como chefe do departamento, então, para obter o nome do chefe do departamento de Bob (se existir), codificamos o seguinte:

```
bob?.department?.head?.name
```

Tal encadeamento resulta em `null` se qualquer uma das propriedades for nula.

1.1.2.3 Operador Elvis

Quando se tem uma referência que pode ser nula `ref`, existe um operador, chamado operador Elvis, que permite usar seu valor, caso exista, ou um valor padrão diferente caso contrário. Um pequeno exemplo dessa expressão codificado usando o operador Elvis:

```
ref ?: x
```

A expressão acima se traduz como "se `ref` não é nulo, use seu valor; caso contrário, use este outro valor não nulo `x`".
Considerando-se novamente a expressão abaixo:

```
val l: Int = if (b != null) b.length else -1
```

Ela pode ser reescrita utilizando-se o operador Elvis, expresso no código por `?:`

```
val l = b?.length ?: -1
```

Se o operando à esquerda, `b?.length`, não for nulo, a expressão retorna seu valor. Caso contrário, a expressão retornará o valor do operando à direita; no caso, `-1`. É importante salientar que a expressão do lado direito só é executada caso a expressão do lado esquerdo seja nula.

Os comandos `return` e `throw` também são expressões em Kotlin, e podem também serem usados no lado direito do operador. Isso pode ser útil, por exemplo, para verificar argumentos de funções:

```
fun foo(node: Node): String? {
   val parent = node.getParent() ?: return null
   val name = node.getName() ?: throw IllegalArgumentException("nome esperado")
   // ...
}
```

1.1.2.4 Operador `!!`

Esse operador diz ao compilador de Kotlin para ignorar a possibilidade de uma variável ou expressão ser nula e, caso realmente o seja, ocorre uma NPE. Pode-se usar para substituir o operador `?.` na chamada de métodos ou no acesso de propriedades.

```
val l = b!!.length
```

Com esse operador, perde-se a segurança de tipo característica de Kotlin. Dessa forma, o código comporta-se da mesma forma que código Java, e toda segurança de tipo deve ser implementada explicitamente pelo programador para evitar que erros de NPE ocorram, como é feito em Java.

1.1.3 Classes

Declara-se classes em Kotlin usando a palavra-chave `class` [42].

```
class Pessoa { ... }
```

A declaração de uma classe consiste no nome da classe, o cabeçalho da classe (especificando os parâmetros, o construtor primário, etc.) e o corpo da classe; todos declarados dentro de chaves. Ambos o cabeçalho e o corpo são opcionais; se a classe não possuir corpo, as chaves podem ser omitidas.

```
class Vazia
```

1.1.3.1 Construtores

Uma classe em Kotlin pode ter um construtor primário e um ou mais construtores secundários. O construtor da classe faz parte do cabeçalho da classe, e ele é declarado logo após a declaração do nome da classe (com parâmetros que podem ser opcionais).

```
class Pessoa constructor(firstName: String) { ... }
```

Se o construtor primário não possui nenhuma *annotation* ou modificadores de visibilidade, a palavra-chave `constructor` pode ser omitida.

```
class Pessoa(firstName: String) { ... }
```

O construtor primário não pode conter nenhum código. O código de inicialização pode ser colocado em um ou mais blocos de inicialização, que são declarados utilizando a palavra `init` seguido do código envolto em chaves.

```
class InitOrderDemo(name: String) {
   val firstProperty = "First property: $name"

   init {
       println("First initializer block that prints ${name}")
   }

   val secondProperty = "Second property: ${name.length}"

   init {
```

```
        println("Second initializer block that prints ${name.length}")
    }
}
```

Nota-se que os parâmetros do construtor primário podem ser usados nos blocos de inicialização. Também podem ser usados na inicialização de propriedades no escopo da classe.

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

Existe também uma sintaxe para declaração de propriedades direto na declaração dos parâmetros no construtor primário:

```
class Person(val firstName: String, val lastName: String, var age: Int) { ... }
```

1.1.3.2 Construtores Secundários

Uma classe também pode possuir construtores secundários, que são declarados utilizando a palavra-chave `constructor`.

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

Se a classe possuir um construtor primário, é necessário chamar o construtor primário passando os parâmetros requeridos por ele. Isso é feito utilizando a palavra-chave `this`:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

É importante notar que o código em blocos de inicialização faz parte do construtor primário, e a chamada ao construtor primário é feita antes de executar o corpo dos construtores secundários. Portanto, todo o código em blocos inicializadores é executado antes de começar a executar o construtor secundário.

1.1.3.3 Criando instâncias de classes

Para criar instâncias de classes, chama-se o construtor como se fosse uma função comum:

```
val invoice = Invoice()
val customer = Customer("Joe Smith")
```

Em Kotlin não existe e não é necessário usar-se a palavra-chave `new`.

1.1.3.4 Herança

Todas as classes de Kotlin tem uma classe em comum chamada `Any`. Esta é a superclasse padrão para todas as classes que não possuem um supertipo.

A classe `Any` difere da original de Java, `java.lang.Object`, por não possuir métodos como, por exemplo, `equals()`, `hashCode()` e `toString()`.

Para declarar um supertipo da classe, coloca-se dois pontos após a declaração do cabeçalho da classe:

```
open class Base(p: Int)
class Derived(p: Int) : Base(p)
```

Se a classe base tem um construtor primário, a classe derivada precisa inicializar o construtor logo após a declaração do supertipo usando os parâmetros do construtor primário.

Se a classe não possuir um construtor primário, então cada construtor secundário precisa inicializar o tipo base usando a palavra-chave `super`, ou chamar

o construtor que irá fazer isso. Note que nesse caso diferentes construtores secundários podem chamar diferentes construtores da classe base.

```
class MyView : View {
   constructor(ctx: Context) : super(ctx)
   constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

1.1.3.5 Sobrescrevendo métodos

Em Kotlin, para criar métodos que podem ser sobrescritos, utiliza-se a palavra-chave *open* antes da declaração do método. Quando é feita uma sobrescrita de método, utiliza-se a palavra-chave *override*.

```
open class Base {
   open fun v() { ... }
   fun nv() { ... }
}
class Derived() : Base() {
   override fun v() { ... }
}
```

A palavra-chave `override` é requerida em `Derived.v()`. Não é permitido ter um método `nv()` na classe `Derived` mesmo com ou sem `override`. Declarar funções com a palavra-chave `open` não possui efeitos em classes declaradas sem a palavra chave `open`.

1.1.3.6 Data Classes

As *Data Classes* são classes cujo único propósito é guardar dados.

```
data class User(val name: String, val age: Int)
```

Quando declara-se uma *Data Class*, o compilador automaticamente cria alguns métodos como `equals()`, `hashCode()`, `toString()`, `copy()`, etc. É possível também criar mais de um construtor para a classe.

1.1.4 Tipagem

A linguagem possui tipagem definida estaticamente (versões da linguagem que não compilam para JVM possuem funcionalidades de tipagem dinâmica [43]). Os tipos de literais são inferidos pelo compilador do Kotlin, e por isso não é necessário declarar o tipo quando são criadas variáveis ou constantes que já sejam inicializadas na mesma sentença. Quando cria-se e inicializa-se uma variável/constante, o compilador consegue inferir o tipo apenas analisando o tipo do que lhe é atribuído.

Apesar de muitos tipos serem inferidos pelo valor atribuído a variáveis e constantes, a tipagem da linguagem pode ser considerada forte. Exemplos são:

- Uma variável não pode ser declarada sem antes atribuir a ela um tipo, seja ele inferido através do Kotlin ou declarado explicitamente:

```
var a = true // variável a de tipo inferido: Boolean
var b: String // variável b de tipo declarado: String
var c /* erro: é necessário declarar um tipo ou inicializá-la (para o tipo ser
        inferido pelo compilador de Kotlin) */
```

- Uma variável, após o tipo definido pela sua declaração, não pode ser reatribuída para um valor de tipo diferente do inicial:

```
var a = 2 // tipo inferido: Int
a = "2" /* erro: o tipo da variável é Int, não pode ser reatribuída
                a outro tipo inferido String pelo Kotlin */
```

1.1.5 Funções de escopo

Kotlin fornece funções de escopo que simplificam a manipulação de várias propriedades de um objeto de uma única vez. Funções de escopo possuem o objetivo, então, de executar um bloco de código dentro do contexto de um objeto. Dentro desse bloco, os atributos e métodos do objeto podem ser acessados sem que seja necessário escrever o nome do objeto. Esse bloco de código pode ser passado para as funções de escopo da forma de expressões lambda [44]. Dois exemplos comuns de funções de escopo são as funções `let` e `apply`. O exemplo abaixo ilustra a utilização da função `let`.

```
Person("Alice", 20, "Amsterdam").let {
    it.moveTo("London")
    it.incrementAge()
}
```

A expressão lambda passada para a função `let` permite que alterações sejam realizadas na instância da classe `Person`. Dentro do escopo do bloco, a instância pode ser referida como `it`.

```
val adam = Person("Adam").apply {
    age = 20 // mesmo que this.age = 20 ou adam.age = 20
    city = "London"
}
```

O exemplo acima mostra o uso da função `apply`. Assim como a função `let`, a função `apply` permite que a instância seja modificada de forma concisa dentro do escopo do bloco passado como parâmetro. No caso da função `apply`, a instância pode ser referenciada dentro do bloco omitindo-se qualquer nome de variável, ou utilizando `this`.

1.2 Integração com Java e Uso no Android

Analisando-se a sintaxe e construções básicas da linguagem Kotlin vistos neste capítulo, nota-se um leve distanciamento da sintaxe de Java. Porém, conceitualmente, são linguagens bastante semelhantes, justamente por ambas compilarem para o mesmo tipo de *bytecode* e poderem ser usadas para o desenvolvimento das mesmas aplicações Android. Uma das muitas vantagens de Kotlin é a sua total interoperação com Java. Isso significa que, se um desenvolvedor decidir utilizar Kotlin no seu projeto já criado em Java, ele pode simplesmente criar código Kotlin normalmente, e toda classe criada terá total acesso ao código Java já existente da aplicação e vice-versa, sem que nenhuma alteração seja necessária no código Java já existente para que isso ocorra. Isso permite também a interação do Kotlin com quaisquer bibliotecas já existentes Java.

Outra grande vantagem de Kotlin é a presença de diversas construções modernas de linguagem que facilitam a programação e legibilidade do código, construções essas que existem também em Java a partir da versão 8. Android atualmente suporta inteiramente apenas Java 7, possuindo poucos recursos do Java 8. Olhando-se a fundo os recursos existentes no Java 8, vê-se que esses recursos facilitam a escrita de aplicações seguindo o paradigma da programação reativa [45]. Embora não seja possível aproveitar todos os recursos do Java 8 no Android, Kotlin faz bem esse papel por já incluir todos esses recursos que facilitam o uso de programação reativa em aplicações Android [2].

Apêndice 2 - Documento de requisitos

Documento de requisitos

O objetivo da aplicação é desenvolver um aplicativo Android simples que busca informações da API do Stack Overflow. O aplicativo Android a ser desenvolvido terá 2 telas.

Primeira tela

Exibe uma lista de posts do Stack Overflow. Será utilizado uma rota de API (endpoint) que retorna uma lista (não paginada para simplificar) de 15 posts com título da pergunta e data de criação.


```
{
    "items": [
        {
            "creation_date": 1557741958,
            "post_id": 56110118,
            "title": "Browser Cache API is not working for cookie authenticated services"
        },
        {
            "creation_date": 1557805700,
            "post_id": 56122748,
            "title": "OSX - Unidentified Developer and bundle format is ambiguous (could
…"
        },
        {
            "creation_date": 1557803528,
            "post_id": 56122494,
            "title": "R error in &#39;[&lt;-.data.frame &#39;C&#39; *temp*&#39;,…
replace…"
        },
        {
            "creation_date": 1557805693,
            "post_id": 56122747,
            "title": "ValueError: shapes (5,14) and (16,) not aligned: 14 (dim 1) != 16
(…"
        },
        {
            "creation_date":1557763292,
            "post_id": 56116017,
            "title": "XDefaultDepth and XDisplayPlanes – what&#39;s the difference?"
        }
    ]
}
```

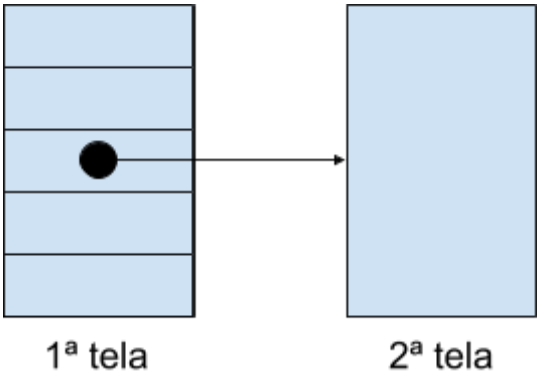

Exemplo de um resultado em JSON da API da primeira tela.

Informações sobre os dados:

- `creation_date`: é a data de criação do post em milissegundos desde 01/01/1970;
- `post_id`: é o ID do post, vai ser utilizado para buscar os detalhes do post;
- `title`: título do post.

Requisitos:

- Mostrar uma lista onde cada item da lista deve conter a `creation_date` e `title`, de forma que o usuário consiga ler todos os campos de todos os itens da lista (descendo a rolagem da tela se necessário);
- Ao clicar em um item dessa lista, a aplicação precisa redirecionar para a segunda tela, a de detalhes do post, e para isso deve-se utilizar o atributo `post_id` como entrada para a rota (endpoint) que retorna o JSON de detalhes.



Esquemático de como a aplicação irá se comportar. A seta representa o redirecionamento da 1ª tela para a 2ª tela. Esse padrão é chamado de master/detail.

Segunda tela

Mostra detalhes de um post do Stack Overflow ao ser selecionado um item da primeira tela. Exibe mais detalhes de uma pergunta; além dos dados da tela anterior, também mostra dados do autor, o corpo da pergunta e um link para a mesma.


```
{
    "items": [
        {
            "owner": {
                "profile_image": "https://www.gravatar.com/avatar/3de2c57425e65b...",
                "display_name": "Deepak Singh"
            },
            "last_activity_date": 1557799078,
            "creation_date": 1557799078,
            "post_id": 56122012,
            "title": "not able to create Event based Task Scheduler using c sharp",
            "link": "https://stackoverflow.com/a/56122012",
```

```
            "body": "<p>my bad some spelling msitakes caused the issue…
        }
    ]
}
```


Exemplo de um resultado em JSON da API da segunda tela.

Informações sobre os dados:

- `owner`: é o JSON de dados do autor da pergunta;
    - `profile_image`: é a URL para o avatar do autor da pergunta;
    - `display_name`: é o nome do autor da pergunta;
- `link`: URL para a pergunta no Stack Overflow;
- `body`: é o corpo da pergunta formatado em HTML, onde é explicada com detalhes;

Requisitos:

- Mostrar uma tela onde deve conter todos os dados (exceto `post_id` e `last_activity_date`);
- Mostrar a foto do autor carregando a imagem disponível na URL;
- Mostrar o corpo da pergunta formatado em HTML (ver https://bit.ly/2vXTAbe)

O objetivo da aplicação é mostrar ao usuários (exceto ID) todos os dados contidos no JSON da primeira tela. Além disso, para cada elemento, ao selecionar um item, redirecionar para a tela de detalhes desse post usando o `post_id` fornecido no JSON.

Será disponibilizado um boilerplate Android com um conjunto de orientações para ser utilizado como base para auxiliar a aplicação. A arquitetura implementada deve seguir a arquitetura apresentada pelo boilerplate. A aplicação deve ser criada apenas com base no boilerplate disponibilizado e na documentação que o acompanha.

Endpoints

JSON da primeira tela:

https://api.stackexchange.com/2.2/posts?order=desc&sort=activity&site=stackoverflow&filter=!iCB*(w(j2UuvtwOC_YVO41

JSON da segunda tela:

https://api.stackexchange.com/2.2/posts/{post_id}?order=desc&sort=activity&site=stackoverflow&filter=!PvzBH.-zD2ypuxhB6jfE6NTVTpsY1a

onde `{post_id}` deverá ser substituído pelo id do post selecionado da primeira tela.
Ex.:
https://api.stackexchange.com/2.2/posts/56122012?order=desc&sort=activity&site=stackoverflow&filter=!PvzBH.-zD2ypuxhB6jfE6NTVTpsY1a
para `post_id = 56122012`.

Requisitos adicionais

- Buscar a lista do Stack Overflow sempre que o app for aberto;

Requisitos não funcionais

- Suportar apenas modo portrait;
- Suportar desde a versão 16 da API do Android (Android Jelly Bean);
- A aplicação a ser desenvolvida deve ser feita em Kotlin;

Informações adicionais

- Poderá utilizar quaisquer bibliotecas para auxiliar o desenvolvimento;
- Não existe um design de interface ideal, o visual da aplicação não precisa ser muito elaborado, apenas mostrar os dados que são requisitados;
- As requisições por restrições da API só podem ser feitas mais de uma vez dentro do intervalo de 10 segundos. Se forem feitas sem esse intervalo a API retorna um erro. Tratar as requisições para só serem feitas depois de 10 segundos não é necessário.

Apêndice 3 - Documentação do boilerplate

app

Packages

Name	Summary
io.schiar.tccboilerplate.model	
io.schiar.tccboilerplate.model.repository	
io.schiar.tccboilerplate.view	
io.schiar.tccboilerplate.view.viewdata	
io.schiar.tccboilerplate.viewmodel	

Index

All Types

app / io.schiar.tccboilerplate.model

Package io.schiar.tccboilerplate.model

Types

Name	Summary
ArchComponent	data class ArchComponent Representa um componente arquitetural.
Library	data class Library Representa uma biblioteca.

app / io.schiar.tccboilerplate.model / ArchComponent

ArchComponent

data class ArchComponent
Representa um componente arquitetural.

Constructors

Name	Summary
<init>	ArchComponent(name: String, description: String, library: Library) Representa um componente arquitetural.

Properties

Name	Summary
description	val description: String descrição do componente.
library	val library: Library biblioteca do componente.
name	val name: String nome do componente.

app / io.schiar.tccboilerplate.model / ArchComponent / <init>

<init>

ArchComponent(name: String, description: String, library: Library)
Representa um componente arquitetural.

app / io.schiar.tccboilerplate.model / ArchComponent / description

description

val description: String
descrição do componente.

Property

description - descrição do componente.

app / io.schiar.tccboilerplate.model / ArchComponent / library

library

val library: Library
biblioteca do componente.

Property
library - biblioteca do componente.

app / io.schiar.tccboilerplate.model / ArchComponent / name

name

val name: String
nome do componente.

Property
name - nome do componente.

app / io.schiar.tccboilerplate.model / Library

Library

data class Library
Representa uma biblioteca.

Constructors

Name	Summary
<init>	Library(name: String, version: List<Int>) Representa uma biblioteca.

Properties

Name	Summary
name	val name: String nome da biblioteca.
version	val version: List<Int> versão da biblioteca.

app / io.schiar.tccboilerplate.model / Library / <init>

<init>

Library(name: String, version: List<Int>)
Representa uma biblioteca.

app / io.schiar.tccboilerplate.model / Library / name

name

val name: String
nome da biblioteca.

Property

name - nome da biblioteca.

app / io.schiar.tccboilerplate.model / Library / version

version

val version: List<Int>
versão da biblioteca.

Property

version - versão da biblioteca.

app / io.schiar.tccboilerplate.model.repository

Package io.schiar.tccboilerplate.model.repository

Types

Name	Summary
ArchComponentRepository	class ArchComponentRepository : ArchComponentRepositoryInterface Implementação de um repository de componentes arquiteturais. Fornece os dados a respeito dos componentes arquiteturais.

ArchComponentRepositoryInterface	interface ArchComponentRepositoryInterface Contrato de um fornecedor de dados para a aplicação. O padrão repository proporciona uma abstração da camada de dados da aplicação. Além disso, ele centraliza o uso dos objetos do domínio. Através de um repository, outros componentes da aplicação conseguem manejar os objetos do domínio de forma simples, sem precisar conhecer de fato de onde esses objetos vêm e onde são armazenados (internet, banco de dados, caches, etc). Isso permite que todos os componentes que usam o repository possuam um baixo acoplamento com as camadas de serviço e persistência da aplicação.

app / io.schiar.tccboilerplate.model.repository / ArchComponentRepository

ArchComponentRepository

class ArchComponentRepository : ArchComponentRepositoryInterface
Implementação de um repository de componentes arquiteturais. Fornece os dados a respeito dos componentes arquiteturais.
Por motivos de simplificação, essa classe gera dados de componentes arquiteturais para a aplicação e os mantém em memória durante a execução. Numa aplicação real, essa classe se comunicaria com as diferentes camadas de dados da aplicação, como por exemplo serviços e persistência.

Constructors

Name	Summary
<init>	ArchComponentRepository() Implementação de um repository de componentes arquiteturais. Fornece os dados a respeito dos componentes arquiteturais.

Functions

Name	Summary
fetch	fun fetch(callback: (List<ArchComponent>) -> Unit): Unit Busca a lista de componentes arquiteturais a ser exibida na View.

app / io.schiar.tccboilerplate.model.repository / ArchComponentRepository / <init>

<init>

ArchComponentRepository()
Implementação de um repository de componentes arquiteturais. Fornece os dados a respeito dos componentes arquiteturais.
Por motivos de simplificação, essa classe gera dados de componentes arquiteturais para a aplicação e os mantém em memória durante a execução. Numa aplicação real, essa classe se comunicaria com as diferentes camadas de dados da aplicação, como por exemplo serviços e persistência.

app / io.schiar.tccboilerplate.model.repository / ArchComponentRepository / fetch

fetch

fun fetch(callback: (List<ArchComponent>) -> Unit): Unit
Overrides ArchComponentRepositoryInterface.fetch
Busca a lista de componentes arquiteturais a ser exibida na View.

Parameters

callback - usado para receber a lista de componentes arquiteturais buscada.

app / io.schiar.tccboilerplate.model.repository / ArchComponentRepositoryInterface

ArchComponentRepositoryInterface

interface ArchComponentRepositoryInterface
Contrato de um fornecedor de dados para a aplicação. O padrão repository proporciona uma abstração da camada de dados da aplicação. Além disso, ele centraliza o uso dos objetos do domínio. Através de um repository, outros componentes da aplicação conseguem manejar os objetos do domínio de forma simples, sem precisar conhecer de fato de onde esses objetos vêm e onde são armazenados (internet, banco de dados, caches, etc). Isso permite que todos os componentes que usam o repository possuam um baixo acoplamento com as camadas de serviço e persistência da aplicação.

Functions

Name	Summary
fetch	abstract fun fetch(callback: (List<ArchComponent>) -> Unit): Unit Busca a lista de componentes arquiteturais a ser exibida na View.

Inheritors

Name	Summary
ArchComponentRepository	class ArchComponentRepository : ArchComponentRepositoryInterface Implementação de um repository de componentes arquiteturais. Fornece os dados a respeito dos componentes arquiteturais.

app / io.schiar.tccboilerplate.model.repository / ArchComponentRepositoryInterface / fetch

fetch

abstract fun fetch(callback: (List<ArchComponent>) -> Unit): Unit
Busca a lista de componentes arquiteturais a ser exibida na View.

Parameters

callback - usado para receber a lista de componentes arquiteturais buscada.

app / io.schiar.tccboilerplate.view

Package io.schiar.tccboilerplate.view

Types

Name	Summary
ArchComponentsListAdapter	class ArchComponentsListAdapter : Adapter<ViewHolder> Trata a lista de ViewDatas para ser exibida pelo componente RecyclerView.
BindingAdapters	object BindingAdapters Utilizado para tratamento de dados do ViewModel para serem apresentados na View através de data binding.
BoilerplateFragment	class BoilerplateFragment : Fragment Mostra a lista de componentes arquiteturais utilizados neste boilerplate
MainActivity	class MainActivity : AppCompatActivity Atividade que controla toda a navegação dos fragmentos da aplicação.

OtherFragment	class OtherFragment : Fragment Mostra uma string pré criada do ViewModel para fins de demonstração do DataBinding

app / io.schiar.tccboilerplate.view / ArchComponentsListAdapter

ArchComponentsListAdapter

class ArchComponentsListAdapter : Adapter<ViewHolder>
Trata a lista de ViewDatas para ser exibida pelo componente RecyclerView.

Parameters

archComponents - lista de ViewDatas.

Types

Name	Summary
ViewHolder	class ViewHolder : ViewHolder interna ao Adapter. Utilizada por ele para tratar o item gráfico da lista.

Constructors

Name	Summary
<init>	ArchComponentsListAdapter(archComponents: List<ArchComponentViewData>) Trata a lista de ViewDatas para ser exibida pelo componente RecyclerView.

Functions

Name	Summary
getItemCount	fun getItemCount(): Int Quantidade de itens.
onBindViewHolder	fun onBindViewHolder(holder: ViewHolder, position: Int): Unit Usado no momento em que o item da lista é exibido.
onCreateViewHolder	fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder Usado para carregar o XML do layout adapter_arch_component com DataBinding que representa um item da lista.

app / io.schiar.tccboilerplate.view / ArchComponentsListAdapter / ViewHolder

ViewHolder

class ViewHolder : ViewHolder
interna ao Adapter. Utilizada por ele para tratar o item gráfico da lista.

Parameters

binding - o objeto do XML do item da lista.

Constructors

Name	Summary
<init>	ViewHolder(binding: AdapterArchComponentBinding) interna ao Adapter. Utilizada por ele para tratar o item gráfico da lista.

Functions

Name	Summary
bind	fun bind(archComponent: ArchComponentViewData): Unit configura o item gráfico com o viewdata.

app / io.schiar.tccboilerplate.view / ArchComponentsListAdapter / ViewHolder / <init>

<init>

ViewHolder(binding: AdapterArchComponentBinding)
interna ao Adapter. Utilizada por ele para tratar o item gráfico da lista.

Parameters

binding - o objeto do XML do item da lista.

app / io.schiar.tccboilerplate.view / ArchComponentsListAdapter / ViewHolder / bind

bind

fun bind(archComponent: ArchComponentViewData): Unit
configura o item gráfico com o viewdata.

app / io.schiar.tccboilerplate.view / ArchComponentsListAdapter / <init>

<init>

ArchComponentsListAdapter(archComponents: List<ArchComponentViewData>)
Trata a lista de ViewDatas para ser exibida pelo componente RecyclerView.

Parameters

archComponents - lista de ViewDatas.

app / io.schiar.tccboilerplate.view / ArchComponentsListAdapter / getItemCount

getItemCount

fun getItemCount(): Int
Quantidade de itens.
Return a quantintidade de items.

app / io.schiar.tccboilerplate.view / ArchComponentsListAdapter / onBindViewHolder

onBindViewHolder

fun onBindViewHolder(holder: ViewHolder, position: Int): Unit
Usado no momento em que o item da lista é exibido.

Parameters

holder - o objeto que trata o item gráfico da lista.
position - a posição do item na lista.

app / io.schiar.tccboilerplate.view / ArchComponentsListAdapter / onCreateViewHolder

onCreateViewHolder

fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder
Usado para carregar o XML do layout adapter_arch_component com DataBinding que representa um item da lista.

Parameters

parent - componente pai do item

viewType - o id do tipo da View. Aqui ignorado e utilizado o adapter_arch_component.
Return o objeto ViewHolder.

app / io.schiar.tccboilerplate.view / BindingAdapters

BindingAdapters

object BindingAdapters
Utilizado para tratamento de dados do ViewModel para serem apresentados na View através de data binding.

Functions

Name	Summary
setValue	fun setValue(textView: TextView, value: String): Unit É adicionado o valor a um label no TextView

app / io.schiar.tccboilerplate.view / BindingAdapters / setValue

setValue

@JvmStatic fun setValue(textView: TextView, value: String): Unit
É adicionado o valor a um label no TextView

app / io.schiar.tccboilerplate.view / BoilerplateFragment

BoilerplateFragment

class BoilerplateFragment : Fragment
Mostra a lista de componentes arquiteturais utilizados neste boilerplate

Constructors

Name	Summary
<init>	BoilerplateFragment() Mostra a lista de componentes arquiteturais utilizados neste boilerplate

Functions

Name	Summary
onCreateView	fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View É carregado o BoilerplateViewModel para passar ao databinding do XML, assim o XML tem acesso aos atributos e métodos do ViewModel.

app / io.schiar.tccboilerplate.view / BoilerplateFragment / <init>

<init>

BoilerplateFragment()
Mostra a lista de componentes arquiteturais utilizados neste boilerplate

app / io.schiar.tccboilerplate.view / BoilerplateFragment / onCreateView

onCreateView

fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View
É carregado o BoilerplateViewModel para passar ao databinding do XML, assim o XML tem acesso aos atributos e métodos do ViewModel.

Parameters

inflater - usado para carregar o XML do fragmento.
container - o componente pai do fragmento.
savedInstanceState - dados do estado anterior do fragmento.
Return view correspondente ao fragmento.

app / io.schiar.tccboilerplate.view / MainActivity

MainActivity

class MainActivity : AppCompatActivity
Atividade que controla toda a navegação dos fragmentos da aplicação.

Constructors

Name	Summary

<init>	MainActivity() Atividade que controla toda a navegação dos fragmentos da aplicação.

Functions

Name	Summary
onBackPressed	fun onBackPressed(): Unit Utilizar o método do navigation para voltar ao fragmento anterior quando há um evento de botão físico de voltar.
onCreate	fun onCreate(savedInstanceState: Bundle?): Unit Define qual o XML de view que a Atividade vai usar, define a toolbar criada no XML como a barra superior da aplicação, e passar o controle de navegação para ela, para assim poder integrar o navigation a aplicação
onSupportNavigateUp	fun onSupportNavigateUp(): Boolean Utilizar o método do navigation para voltar ao fragmento anterior quando há um evento de botão da barra superior. de voltar

app / io.schiar.tccboilerplate.view / MainActivity / <init>

<init>

MainActivity()
Atividade que controla toda a navegação dos fragmentos da aplicação.

app / io.schiar.tccboilerplate.view / MainActivity / onBackPressed

onBackPressed

fun onBackPressed(): Unit
Utilizar o método do navigation para voltar ao fragmento anterior quando há um evento de botão físico de voltar.

app / io.schiar.tccboilerplate.view / MainActivity / onCreate

onCreate

protected fun onCreate(savedInstanceState: Bundle?): Unit
Define qual o XML de view que a Atividade vai usar, define a toolbar criada no XML como a barra superior da aplicação, e passar o controle de navegação para ela, para assim poder integrar o navigation a aplicação

Parameters
savedInstanceState - guarda dados da última execução permitindo restaurar o estado anterior.

app / io.schiar.tccboilerplate.view / MainActivity / onSupportNavigateUp

onSupportNavigateUp
fun onSupportNavigateUp(): Boolean
Utilizar o método do navigation para voltar ao fragmento anterior quando há um evento de botão da barra superior. de voltar
Return true se foi possível voltar ao fragmento anterior, false caso contrário

app / io.schiar.tccboilerplate.view / OtherFragment

OtherFragment
class OtherFragment : Fragment
Mostra uma string pré criada do ViewModel para fins de demonstração do DataBinding

Constructors

Name	Summary
<init>	OtherFragment() Mostra uma string pré criada do ViewModel para fins de demonstração do DataBinding

Functions

Name	Summary
onCreateView	fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? É carregado o OtherViewModel para passar ao databinding do XML, assim o XML tem acesso aos atributos e métodos do ViewModel.

app / io.schiar.tccboilerplate.view / OtherFragment / <init>

<init>

OtherFragment()
Mostra uma string pré criada do ViewModel para fins de demonstração do DataBinding

app / io.schiar.tccboilerplate.view / OtherFragment / onCreateView

onCreateView

fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View?
É carregado o OtherViewModel para passar ao databinding do XML, assim o XML tem acesso aos atributos e métodos do ViewModel.

Parameters

inflater - usado para carregar o XML do fragmento.
container - o componente pai do fragmento.
savedInstanceState - dados do estado anterior do fragmento.
Return view correspondente ao fragmento.

app / io.schiar.tccboilerplate.view.viewdata

Package io.schiar.tccboilerplate.view.viewdata

Types

Name	Summary
ArchComponentViewData	data class ArchComponentViewData Representação dos componentes arquiteturais do ponto de vista da visão.
LibraryViewData	data class LibraryViewData Representação das bibliotecas do ponto de vista da visão.

app / io.schiar.tccboilerplate.view.viewdata / ArchComponentViewData

ArchComponentViewData

data class ArchComponentViewData

Representação dos componentes arquiteturais do ponto de vista da visão.

Constructors

Name	Summary
<init>	ArchComponentViewData(name: String, description: String, library: LibraryViewData) Representação dos componentes arquiteturais do ponto de vista da visão.

Properties

Name	Summary
description	val description: String descrição de um componente.
library	val library: LibraryViewData biblioteca de um componente.
name	val name: String nome de um componente.

app / io.schiar.tccboilerplate.view.viewdata / ArchComponentViewData / <init>

<init>

ArchComponentViewData(name: String, description: String, library: LibraryViewData)
Representação dos componentes arquiteturais do ponto de vista da visão.

app / io.schiar.tccboilerplate.view.viewdata / ArchComponentViewData / description

description

val description: String
descrição de um componente.

Property

description - descrição de um componente.

app / io.schiar.tccboilerplate.view.viewdata / ArchComponentViewData / library

library

val library: LibraryViewData
biblioteca de um componente.

Property

library - biblioteca de um componente.

app / io.schiar.tccboilerplate.view.viewdata / ArchComponentViewData / name

name

val name: String
nome de um componente.

Property

name - nome de um componente.

app / io.schiar.tccboilerplate.view.viewdata / LibraryViewData

LibraryViewData

data class LibraryViewData
Representação das bibliotecas do ponto de vista da visão.

Constructors

Name	Summary
<init>	LibraryViewData(name: String, version: String) Representação das bibliotecas do ponto de vista da visão.

Properties

Name	Summary
name	val name: String nome de uma biblioteca.
version	val version: String

	versão de uma biblioteca.

Functions

Name	Summary
toString	fun toString(): String Método auxiliar para gerar um nome completo da biblioteca.

app / io.schiar.tccboilerplate.view.viewdata / LibraryViewData / <init>

<init>

LibraryViewData(name: String, version: String)
Representação das bibliotecas do ponto de vista da visão.

app / io.schiar.tccboilerplate.view.viewdata / LibraryViewData / name

name

val name: String
nome de uma biblioteca.

Property

name - nome de uma biblioteca.

app / io.schiar.tccboilerplate.view.viewdata / LibraryViewData / version

version

val version: String
versão de uma biblioteca.

Property

version - versão de uma biblioteca.

app / io.schiar.tccboilerplate.view.viewdata / LibraryViewData / toString

toString

fun toString(): String
Método auxiliar para gerar um nome completo da biblioteca.
Return o nome completo da biblioteca.

app / io.schiar.tccboilerplate.viewmodel

Package io.schiar.tccboilerplate.viewmodel

Types

Name	Summary
BoilerplateViewModel	class BoilerplateViewModel : ViewModel Recebe mensagens da visão solicitando dados. Formata esses dados e os disponibiliza para a visão através dos objetos LiveData.
OtherViewModel	class OtherViewModel : ViewModel Recebe mensagens da visão solicitando dados. Formata esses dados e os disponibiliza para a visão através dos objetos LiveData.

app / io.schiar.tccboilerplate.viewmodel / BoilerplateViewModel

BoilerplateViewModel

class BoilerplateViewModel : ViewModel
Recebe mensagens da visão solicitando dados. Formata esses dados e os disponibiliza para a visão através dos objetos LiveData.

Constructors

Name	Summary
<init>	BoilerplateViewModel(archComponentRepository: ArchComponentRepositoryInterface = ArchComponentRepository()) Recebe mensagens da visão solicitando dados. Formata esses dados e os disponibiliza para a visão através dos objetos LiveData.

Properties

Name	Summary
archComponents	val archComponents: MutableLiveData<List<ArchComponentViewData>> lista atual de componentes arquiteturais.
buttonContent	val buttonContent: MutableLiveData<String> conteúdo do botão da tela.

Functions

Name	Summary
fetch	fun fetch(): Unit Busca os dados de componentes arquiteturais e atualiza o LiveData de archComponents.

app / io.schiar.tccboilerplate.viewmodel / BoilerplateViewModel / <init>

<init>

BoilerplateViewModel(archComponentRepository: ArchComponentRepositoryInterface = ArchComponentRepository())
Recebe mensagens da visão solicitando dados. Formata esses dados e os disponibiliza para a visão através dos objetos LiveData.

app / io.schiar.tccboilerplate.viewmodel / BoilerplateViewModel / archComponents

archComponents

val archComponents: MutableLiveData<List<ArchComponentViewData>>
lista atual de componentes arquiteturais.

Property

archComponents - lista atual de componentes arquiteturais.

Getter Property

archComponents - lista atual de componentes arquiteturais.

Getter

lista atual de componentes arquiteturais.

app / io.schiar.tccboilerplate.viewmodel / BoilerplateViewModel / buttonContent

buttonContent

val buttonContent: MutableLiveData<String>
conteúdo do botão da tela.

Property
buttonContent - conteúdo do botão da tela.

app / io.schiar.tccboilerplate.viewmodel / BoilerplateViewModel / fetch

fetch

fun fetch(): Unit
Busca os dados de componentes arquiteturais e atualiza o LiveData de archComponents.

app / io.schiar.tccboilerplate.viewmodel / OtherViewModel

OtherViewModel

class OtherViewModel : ViewModel
Recebe mensagens da visão solicitando dados. Formata esses dados e os disponibiliza para a visão através dos objetos LiveData.

Constructors

Name	Summary
<init>	OtherViewModel() Recebe mensagens da visão solicitando dados. Formata esses dados e os disponibiliza para a visão através dos objetos LiveData.

Properties

Name	Summary
helloWorld	val helloWorld: MutableLiveData<String> usado no XML para ser mostrado na tela.

app / io.schiar.tccboilerplate.viewmodel / OtherViewModel / <init>

<init>

OtherViewModel()
Recebe mensagens da visão solicitando dados. Formata esses dados e os disponibiliza para a visão através dos objetos LiveData.

helloWorld

val helloWorld: MutableLiveData<String>
usado no XML para ser mostrado na tela.

Property

helloWorld - usado no XML para ser mostrado na tela.

Apêndice 4 - Arquivo README.md do projeto do *boilerplate*

Boilerplate para construção de aplicações Android

Arquitetura

A arquitetura desse boilerplate segue os moldes do MVVM, ao mesmo tempo que usufrui dos mais recentes componentes arquiteturais do Android.

Android Jetpack

Este boilerplate utiliza uma boa parte dos componentes arquiteturais do Android Jetpack:

- ViewModel
- LiveData
- Navigation
- DataBinding

Código de amostra

Com esse boilerplate, é implementado um aplicativo que mostra os componentes arquiteturais contidos no próprio boilerplate, para fins de consulta e a fim de exemplificar o uso da arquiterura. Esse código contém pelo menos um exemplo de cada componente da arquitetura apresentada. É mostrado como integrar o DataBinding, como fazer uso do Navigation, assim como também tem um fragmento simples e um viewmodel prontos para a aplicação ser implementada.

Testes

É apresentado o Mockito para o mock de componentes da arquitetura, de forma a permitir os mais diversos testes unitários das camadas de Model e ViewModel, como por exemplo os objetos LiveData.

Documentação

Todo código do boilerplate está documentado, e é utilizado o Dokka para a geração de páginas de markdown.

Estrutura de diretório

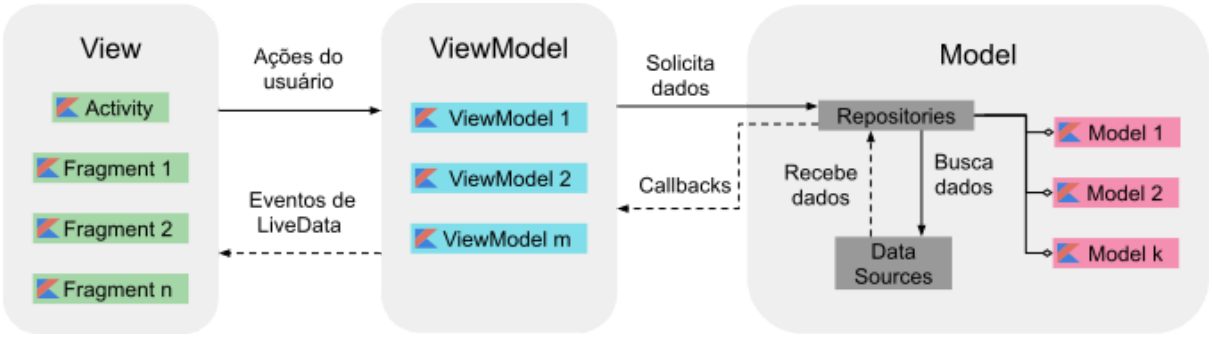
Pasta main

```
.
├── AndroidManifest.xml
├── java
│   └── io
│       └── schiar
│           └── tccboilerplate
│               ├── model
│               │   ├── ArchComponent.kt
│               │   ├── Library.kt
│               │   └── repository
│               │       ├── ArchComponentRepository.kt
│               │       └── ArchComponentRepositoryInterface.kt
│               ├── view
│               │   ├── BindAdapters.kt
│               │   ├── BoilerplateFragment.kt
│               │   ├── MainActivity.kt
│               │   ├── OtherFragment.kt
│               │   └── viewdata
│               │       ├── ArchComponentViewData.kt
│               │       └── LibraryViewData.kt
│               └── viewmodel
│                   ├── BoilerplateViewModel.kt
│                   └── OtherViewModel.kt
└── res
    ├── drawable-*
    │   └── ic_launcher_background.xml
    ├── layout
    │   ├── activity_main.xml
    │   ├── fragment_boilerplate.xml
    │   ├── fragment_other.xml
    │   └── view_arch_component.xml
    ├── mipmap-*
    │   ├── ic_launcher.xml
    │   └── ic_launcher_round.xml
    ├── navigation
    │   └── navigation.xml
    └── values
        ├── colors.xml
        ├── dimens.xml
        ├── strings.xml
        └── styles.xml
```

Esta pasta contém todos os arquivos de código e XMLs necessários para a construção da aplicação.

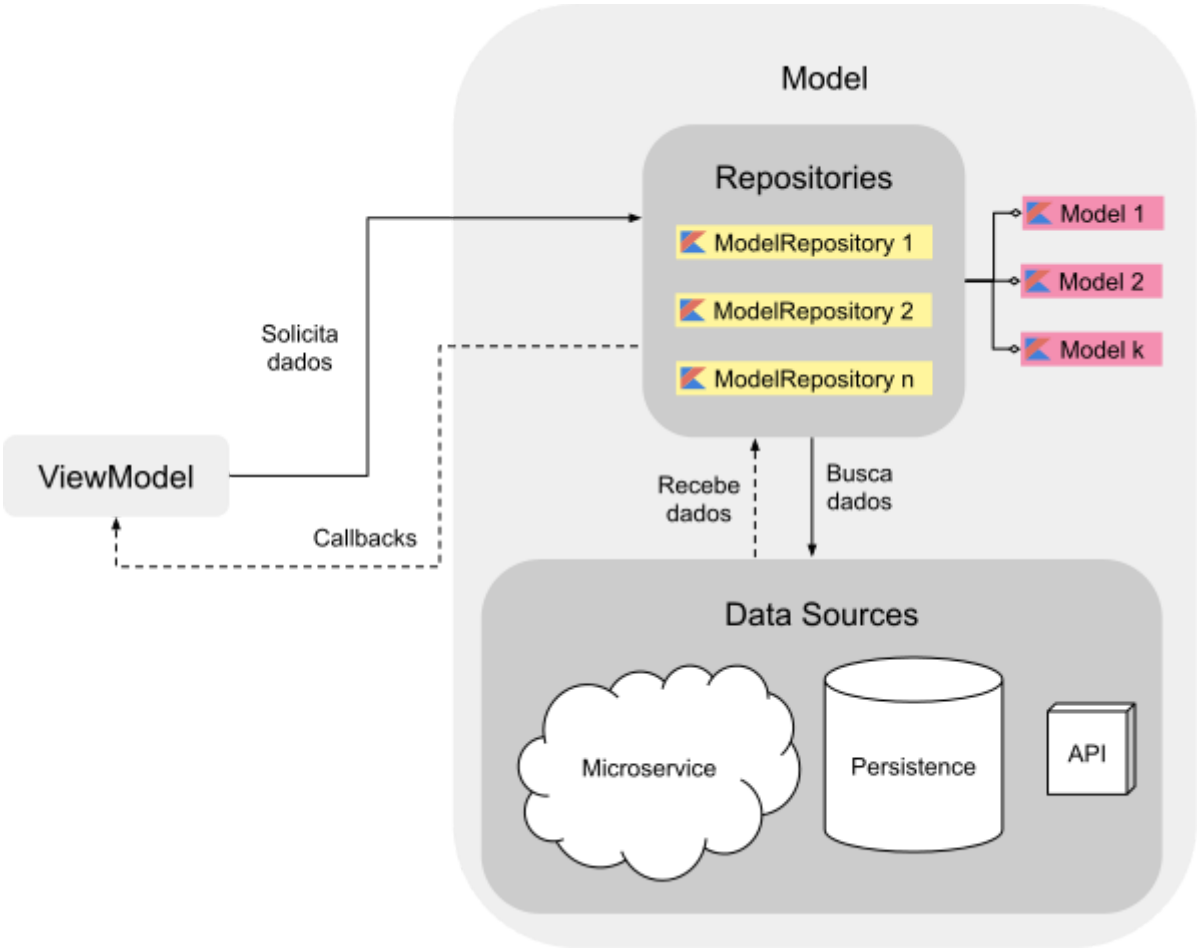
/java

Apesar do nome, dentro dela está todo o código da aplicação, a qual é feita 100% em Kotlin. Os pacotes estão divididos segundo o padrão da arquitetura MVVM. Segue esquemático de uma arquitetura genérica MVVM:



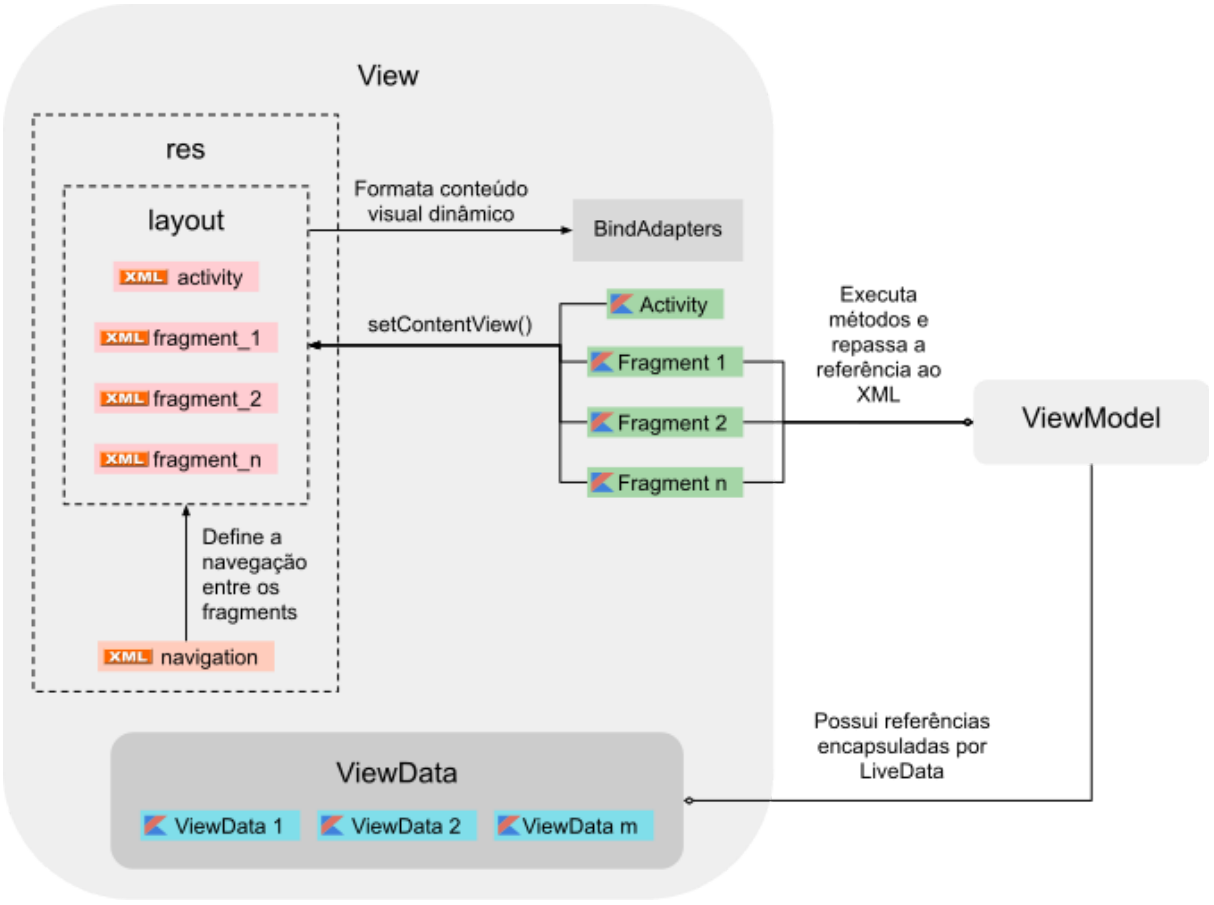
/java/io/schiar/tccboilerplate/model

Onde são implementadas as regras de negócio da aplicação. Dentro desse pacote também estão os repositórios, responsáveis pela requisição de dados. Segue o esquemático de um model genérico:



/java/io/schiar/tccboilerplate/view

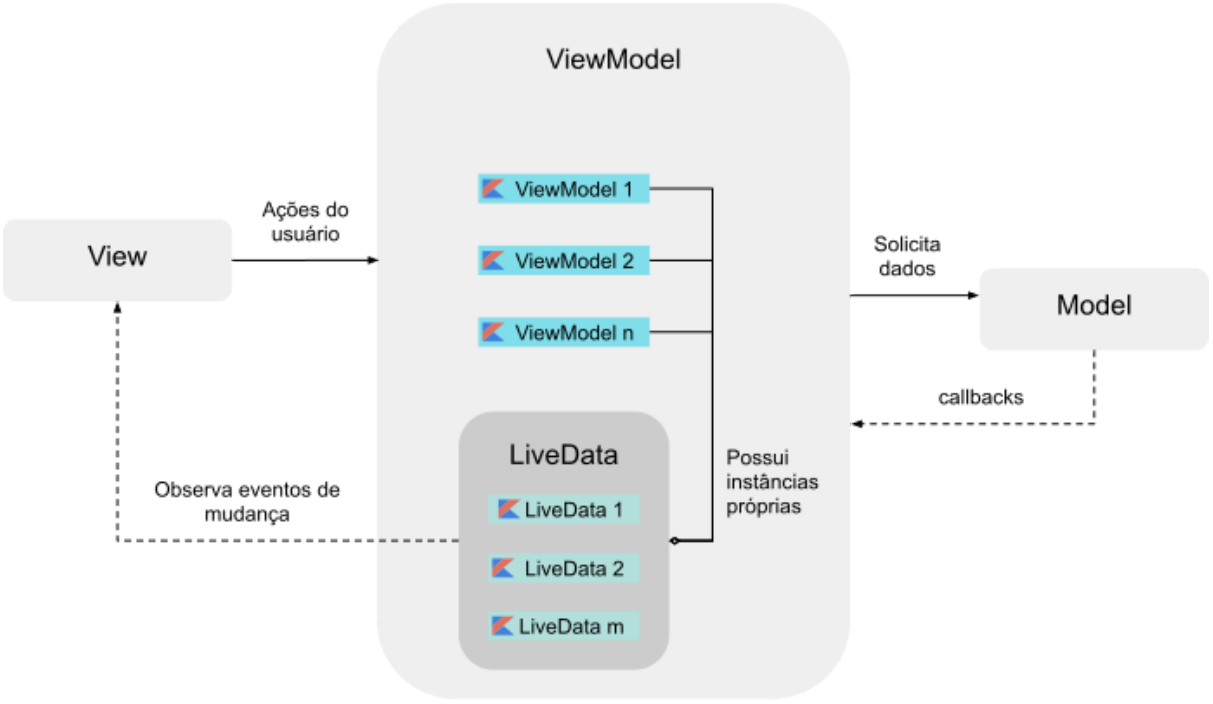
Responsável pelas classes que representam as telas da aplicação. É onde também estão os ViewDatas, utilizados para a formatação de filtragem de objetos do modelo. Com os objetos ViewData, a view conhece estritamente o que é necessário para a exibição dos dados. Segue o esquemático de uma view genérica:



/java/io/schiar/tccboilerplate/view-model

Responsável pela implementação dos LiveData. Esses LiveData expõem os dados que a view precisa mostrar na tela. Para fazer isso, a view pode pegar as informações desses LiveData ou observar quaisquer mudanças que ocorrem neles, atualizando a tela de forma reativa. Esses LiveData podem também ser referenciados diretamente pelos XMLs da view para a exibição dos dados, graças ao uso de DataBinding e os Bind Adapters. Cada mudança de dados encapsulados com LiveData automaticamente notifica todos os lugares em que são observados. É recomendado que os LiveData sejam encapsulados por objetos

ViewData. Segue o esquemático de um view model genérico:



/res

Na pasta res estão os XMLs de apoio à aplicação.

/res/drawable-*e /res/mipmap-*

Carregam todo os ícones (ou eventuais imagens) da aplicação nos mais diversos tamanhos de tela (existe uma pasta pra cada tamanho de Drawablee Mipmap, ocultadas nesse exemplo para simplificação).

/res/layout

Contém os XML que representam os componentes de visão da aplicação. Cada fragmento está aqui representado. No método onCreateView(...)de cada fragmento da pasta /java/io/schiar/tccboilerplate/viewé carregado o seu arquivo XML localizado nessa pasta. Os XML dessas pastas possuem acesso a métodos e LiveData de ViewModels graças ao DataBinding. O fragmento carrega o XML através do Navigation e passa a referência do ViewModel para ele.

/res/navigation

Contém um único XML que serve para descrever todas as transições que devem ocorrer entre os fragmentos da aplicação.

/res/values

Contém constantes diversas do programa: cores, dimensões, strings, e estilos. Cada um fica em seu próprio XML, e todos podem ser utlizados em fragmentos e XMLs.

Pasta test

```
.
└── java
```

```
└── io
    └── schiar
        └── tccboilerplate
            ├── BoilerplateViewModelUnitTest.kt
            └── mock
                ├── GenericMockFactory.kt
                └── MockArchComponentRepository.kt
```

/java/io/schiar/tccboilerplate/

Aqui ficam os testes unitários da aplicação.

/java/io/schiar/tccboilerplate/mock

Muitas vezes é necessária a geração de objetos "falsos" para a execução de testes unitários. Esses objetos são chamados de mock. Irão servir de suporte aos testes.

Gradle

É o gerenciador de pacotes acoplado ao Android. É com ele que é gerenciado as bibliotecas utilizadas em aplicações Android. A seguir serão mostradas as configurações do Gradle personalizadas para esse boileplate.

Plugins

apply plugin: 'com.android.application' /* Módulo padrão de aplicações Android. */

apply plugin: 'kotlin-android' /* Habilita a utilização da linguagem Kotlin. */

apply plugin: 'kotlin-android-extensions' /* Usado para recursos adicionais para suplantar o código Kotlin. */

apply plugin: 'kotlin-kapt' /* Necessário para o DataBinding */

apply plugin: 'org.jetbrains.dokka-android' /* Geração de HTML para documentação */

Dependências

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar']) /* Se existir qualquer biblioteca adicionada manualmente esse comando detecta e inclui na compilação */
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version" /* Biblioteca oficial Kotlin */
    implementation 'androidx.appcompat:appcompat:1.0.2' /* Habilita codificar para versões antigas do Android */
    implementation 'androidx.core:core-ktx:1.0.2' /* Habilita importantes adicionais ao Kotlin como um modo simplificado de acesso ao XML pelo fragment */
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3' /* Layout avançado utilizado em layouts */
```

```
    implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0' /* Biblioteca adicional para utilização de componentes arquiteturais do Jetpack */
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.0.0' /* Permite a utilização de ViewModel */
    implementation 'androidx.legacy:legacy-support-v4:1.0.0' /* Suporte do androidx a versões antigas */
    testImplementation 'junit:junit:4.12' /* Criação de testes unitários */
    testImplementation 'org.mockito:mockito-core:2.27.0' /* Criação de mocks para suporte de testes unitários */
    testImplementation 'android.arch.core:core-testing:1.1.1' /* Criação de mocks de componentes do Android para serem utilizados em testes unitários */
    androidTestImplementation 'androidx.test:runner:1.1.1' /* Teste instrumentado Android */
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1' /* Teste instrumentado Android */
    kapt "com.android.databinding:compiler:3.3.2" /* Permite a utilização de DataBinding à aplicação */
    implementation 'android.arch.navigation:navigation-fragment:1.0.0' /* Permite a utilização de Navigation à aplicação */
    implementation 'android.arch.navigation:navigation-ui:1.0.0' /* Permite a utilização de Navigation à aplicação */
}
```

Há também um código adicional para a utilização do DataBinding:

```
dataBinding {
    enabled = true
}
```

Orientações Gerais

Esse boilerplate foi desenvolvido visando as melhores práticas de engenharia de software para desenvolvimento de aplicações Android. Aqui estão algumas orientações para a melhor utilização desse boilerplate.

- ViewModels não podem possuir referências à classes da View, especialmente aquelas do framework Android;
- ViewModels foram pensados para funcionar como a janela do modelo para a view. Em um fluxo recomendado, o ViewModel, com sua referência ao Repository, busca dados do modelo, monta objetos de visão (ViewDatas) e os deixa disponíveis para a view ter acesso;
- Esse boilerplate incentiva o desenvolvimento de aplicativos de atividade única (single activity). Assim pode-se tirar o máximo de proveito do componente Navigation e a Google também recomenda que aplicações funcionem dessa maneira;
- Recomenda-se a utilização de classes de modelo para executar a lógica da aplicação;
- A utilização de DataBinding junto com BindAdapters é encorajada. Recomenda-se utilizar essa combinação sempre que possível;
- Recomenda-se utilizar sempre que possível os arquivos de constantes localizados em /respara colocar as constantes da aplicação;

Como começar?

- Clone esse repositório
- Abra-o no Android Studio
- Utilize como exemplo as classes pré implementadas. Aqui possui outro exemplo que utiliza as melhores práticas.
- Edite as classes para formar sua própria aplicação. Mude o package id e o nome dos pacotes para ser o da sua própria aplicação.

Apêndice 5 - Artigo

Criação de uma solução arquitetônica para organização de código em aplicações Android

Giovani Lopes Schiar Junior

Instituto de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

Caixa Postal 476 – 88040-900 – Florianópolis – SC – Brasil

giovani.schiar@grad.ufsc.br

Abstract. *Developers of the Android system have not recommended any architecture for the development of applications. Moreover, the Android execution model and the libraries of its SDK provide guidelines that hinder loose coupling and high cohesion between components. Hence, defining the ideal architecture for Android development is a challenge. A boilerplate project that presents a software architecture that intends to help programmers on Android application development was created to help them create their applications with desirable code organization features. The results show that the boilerplate contributed to the development of Android applications with high cohesion and low coupling.*

Resumo. *Os desenvolvedores do sistema Android não definem uma arquitetura recomendada para as aplicações e, além disso, o modelo de execução do Android e as bibliotecas do Android SDK definem diretivas que dificultam o baixo acoplamento e a alta coesão entre os componentes. Definir uma arquitetura ideal é então um grande desafio. Neste trabalho foi criado um boilerplate que apresenta uma arquitetura que visa auxiliar programadores no desenvolvimento de aplicações Android com características desejáveis de organização de código. A avaliação realizada mostrou que o boilerplate contribuiu para o desenvolvimento de aplicações com baixo acoplamento e alta coesão.*

1. Introdução

Android tornou-se o sistema operacional dominante no mercado [1]. Logo, é natural a demanda e interesse por desenvolver aplicações para essa plataforma.

Uma aplicação Android utiliza um modelo de programação baseado em atividades [2], que são classes centrais onde, através delas, todos os recursos do Android podem

facilmente ser utilizados, incluindo iniciar uma outra atividade, manipular componentes gráficos, gerenciamento de áudio, sensores, notificações, etc.

Um dos problemas desse modelo de programação do Android é que desenvolvedores têm a tendência de criar toda a lógica de suas aplicações em volta dessas atividades [3]. Esse comportamento resulta em aplicações onde as classes de atividades são muito grandes e complexas, enquanto as demais classes possuem pouco código. Não há uma arquitetura bem definida e recomendada oficialmente [2]. Cabe ao programador decidir a melhor abordagem e, historicamente, os desenvolvedores têm buscado utilizar a arquitetura Model-View-Controller [4], ou MVC, onde as classes das aplicações ficam divididas em Modelo, Visão e Controlador. Desde o início do desenvolvimento de aplicações para Android, os desenvolvedores têm enfrentado dificuldades em organizar o código e separar a lógica nesses três componentes [3] [5].

Recentemente, o Google lançou o Android Jetpack [6]: uma coleção de componentes com diversas finalidades que facilitam a programação Android. Entre eles, existem vários componentes arquiteturais. Apesar de ainda não existir uma arquitetura recomendada, esses componentes arquiteturais já permitem que, a partir deles, seja possível construir uma arquitetura de forma mais concisa [7]. Embora o Android Jetpack forneça a maioria das peças do quebra-cabeça de uma arquitetura ideal para Android, algumas peças devem ser preenchidas pelos desenvolvedores.

Este trabalho apresenta um código boilerplate de uma arquitetura simples, porém completa, voltada ao desenvolvimento de aplicações Android, e que visa combinar as vantagens dos componentes disponíveis no Android Jetpack com as vantagens de um padrão de arquitetura adequado para aplicações móveis. Uma aplicação Android exemplo explorando o uso da arquitetura proposta também é fornecida para auxiliar na compreensão da arquitetura.

Código boilerplate, no contexto deste trabalho, consiste em todos os arquivos necessários para compor a base de uma aplicação Android qualquer. As classes e arquivos desse boilerplate estão organizados de forma tão específica quanto necessário para que a arquitetura apresentada possa ser compreendida, e tão genérica quanto possível para que desenvolvedores possam usá-los no desenvolvimento de qualquer aplicação Android.

O código boilerplate proposto pretende explorar a arquitetura Model-View-ViewModel (Modelo, Visão, Modelo de Visão), ou MVVM, baseada em conceitos de programação reativa, em conjunto com os componentes arquiteturais contidos no Android Jetpack, que também seguem o paradigma da programação reativa. Esse boilerplate será criado na linguagem Kotlin, que, embora seja nova e menos conhecida do que a linguagem Java, é a linguagem de programação que a comunidade de desenvolvedores Android prefere utilizar para desenvolvimento Android atualmente [8], o que permite que o boilerplate tenha uma aceitação maior pela comunidade.

2. Fundamentação teórica

2.1. Android

O Android é um sistema operacional baseado no núcleo Linux. Foi projetado para ser executado principalmente em dispositivos com telas sensível ao toque como smartphones e tablets [9]. Possui também interface específica para TVs, carros e relógios de pulso. Atualmente, 74.69% dos usuários de smartphones no mundo utilizam o sistema operacional Android [1]. Dados de 2017 também indicam que o sistema operacional está presente em cerca de 37.93% de todos os dispositivos da atualidade, superando até mesmo o sistema operacional Windows [10].

O modelo de programação de um aplicativo Android é baseado principalmente no conceito de atividade; existe uma classe do sistema Android chamada Activity para esse fim. O programador precisa criar uma classe que herda de Activity e sobrescrever métodos específicos que correspondem a estágios específicos do ciclo de vida da atividade. Nesses métodos é feita a customização do comportamento da atividade. O sistema Android se encarrega de invocar os métodos específicos dessa classe conforme a atividade passa pelos seus estágios do ciclo de vida.

Cada atividade pode também possuir um ou mais fragmentos, representados no Android pela classe Fragment, que são componentes menores de elementos de interface gráfica, e que permitem uma melhor organização dos elementos gráficos. Fragmentos e atividades também podem ser chamados no Android de controladores de UI (do inglês *user interface*, ou interface do usuário). Ambos os componentes manipulam componentes gráficos na tela.

Um dos principais elementos do ecossistema Android e que permeia por todos os componentes do framework é o objeto de contexto. O contexto é uma interface de informações globais do ambiente da aplicação. É uma classe abstrata cuja implementação é provida pelo sistema do Android [11]. Com ela é possível acessar recursos e classes específicos da aplicação, bem como realizar chamadas a operações a nível de aplicação, tal como iniciar atividades, receber e enviar *Intents*, etc. O uso dos objetos de contexto em aplicativos Android é um dos maiores desafios para a organização e estruturação de aplicativos Android [12].

O sistema Android não recomenda uma arquitetura específica [2]. Muitos desenvolvedores, ao longo do tempo, tentaram adaptar os modelos de arquitetura existentes para o Android. O mais comum deles sempre foi o padrão MVC [4]. Porém, mesmo para esse padrão de arquitetura sempre houveram muitos questionamentos a respeito de como utilizá-lo. Por exemplo, os controladores de UI fazem parte da visão ou do controlador [13]? Controladores de UI podem conter elementos de interface gráfica, porém também podem ser

o ponto de acesso a recursos do sistema operacional que estejam atrelados à lógica da aplicação.

Como consequência dessa falta de padrão e desses desafios, muitos desenvolvedores, tanto iniciantes quanto experientes, têm a tendência de criar toda a lógica de suas aplicações em volta dos controladores de UI e das classes que possuem acesso a contextos [3]. Esse comportamento resulta em aplicações onde as classes dos controladores de UI são muito grandes e complexas e as demais classes possuem pouco código [13] [14].

2.2. Padrões de arquitetura

Ao longo dos anos, foram criados os mais variados padrões de arquitetura para o desenvolvimento de softwares. Para cada ambiente onde eles foram utilizados, porém, pequenas adaptações precisaram ser feitas para que eles pudessem existir de forma coerente dentro do ambiente [16]. Nesta seção são discutidos os principais desses padrões, que problemas eles solucionam, e que problemas eles criam.

2.2.1. MVC

Inicialmente, a arquitetura que muitos programadores resolveram aplicar foi o MVC, de Model-View-Controller, ou Modelo-Visão-Controlador. Esse padrão tem como objetivo separar as responsabilidades de qual componente se encarrega da lógica da aplicação e dos dados (Modelo), qual se encarrega de mostrar esses dados na interface gráfica para o usuário e coletar os eventos disparados pelo usuário na tela (Visão), e qual componente conecta os dois primeiros realizando a troca de mensagens, tratamento de eventos e o fluxo dos dados (Controlador). O MVC é um padrão de arquitetura muito bem difundido no universo Java, em diversos ambientes [17].

Há uma grande divergência entre membros da comunidade Android a respeito de como conceitos como atividades e fragmentos se encaixam em uma arquitetura MVC [5] [14]. Pode-se ver uma atividade como um controlador que trata os eventos da visão, que no caso seriam os arquivos XML declarativos, e que ao mesmo tempo atualiza o modelo? Por outro lado, pode-se ver uma atividade como parte da visão, onde então o controlador seria um artefato isolado das classes do framework Android [13]?

Os componentes do MVC precisam ser criados pelo desenvolvedor, o que adiciona mais complexidade ao código da aplicação. Componentes do Android, como atividades e serviços, farão parte do MVC, mas precisarão ser inseridos dentro dos componentes do MVC criados. A necessidade dessa abordagem pode ser vista como uma das desvantagens do padrão MVC no desenvolvimento Android.

2.2.2. MVP

O MVP, de Model-View-Presenter, ou Modelo-Visão-Apresentador, é um padrão de arquitetura não muito distante do MVC clássico, porém introduz mudanças no papel do

controlador, chamando-o de presenter, ou apresentador. Uma das vantagens do MVP é permitir uma separação entre a lógica de visão e os componentes gráficos do Android [16]. Dessa forma, é possível criar testes unitários que avaliem a lógica da visão sem precisar simular ou interagir com os componentes gráficos do Android.

Da mesma forma que ocorre com o MVC, no MVP não existe clareza de como organizar e separar o código em visão, modelo e apresentação no contexto do desenvolvimento Android [15]. Isso ocorre porque não existem componentes bem definidos dentro do framework Android que realizam exclusivamente o papel de visão, modelo ou apresentação. Porém, por causa das características do presenter, faz sentido ver as atividades da aplicação como presenters, onde a view seria as definições de layout nos arquivos XML e subclasses de View. Por outro lado, existe ainda o risco de muito código ser colocado dentro das atividades, prejudicando a organização do código e dificultando a manutenção do aplicativo.

2.2.3. MVVM

O MVVM, de Model-View-ViewModel, ou Modelo-Visão-Modelo da Visão, é um padrão de arquitetura apresentado por John Gossman para aplicações da Microsoft que tem como objetivo automatizar as mudanças ocorridas no modelo para apresentá-las na visão [18]. O modelo e visão são semelhantes ao MVC e MVP. O ViewModel ou Modelo da Visão, diferentemente do Presenter, possui um elemento chamado Binder, ou vinculador, responsável por justamente estabelecer um vínculo entre a visão e a lógica de negócio automatizando as suas mudanças de estado. A figura 1 mostra ilustra esse comportamento.

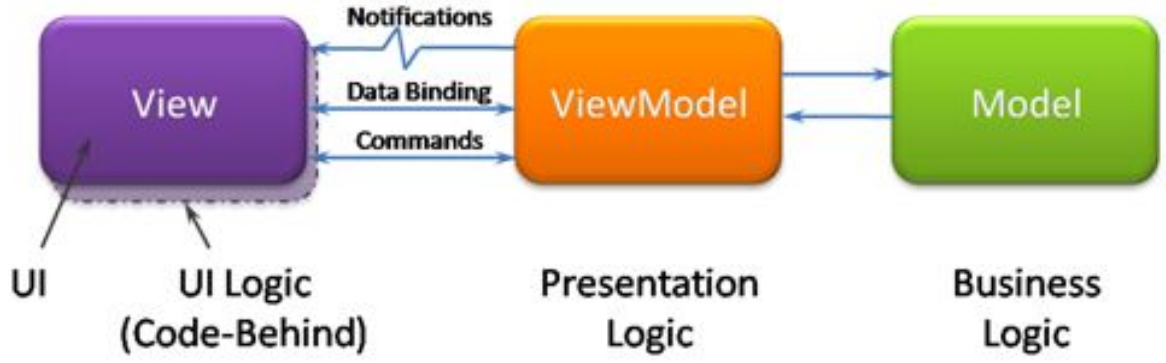


Figura 1. Demonstração do fluxo de dados da arquitetura MVVM.

Uma das vantagens do MVVM é a separação total entre a lógica do negócio e a apresentação da interface gráfica, permitindo que a visão seja modificada sem que o modelo seja afetado, considerando que a interface com o ViewModel continue a mesma. Outra vantagem desse padrão é permitir a criação simplificada de testes unitários do modelo e do ViewModel sem usar a visão [19].

2.3. Android Jetpack

Visando facilitar o desenvolvimento de aplicações Android e incentivar os desenvolvedores a utilizar boas práticas de programação, recentemente o Google lançou o Android Jetpack [6], que consiste em um conjunto de componentes de software que é externo ao Android SDK, complementando-o, com o objetivo de utilizar as novas funcionalidades da plataforma sem perder a compatibilidade com versões antigas do sistema.

2.3.1. Componentes de Arquitetura

Os componentes de arquitetura do Android Jetpack fornecem ferramentas para auxiliar na organização e estruturação do código da aplicação. Além disso, esses Componentes de Arquitetura também tratam internamente diversos aspectos intrínsecos do ecossistema Android [6]. Ao realizar esse tratamento internamente, esses componentes permitem que os desenvolvedores não precisem se preocupar em fazer com que a arquitetura da aplicação tenha que lidar com eles. A arquitetura da aplicação pode então ser focada nos aspectos mais inerentes da lógica própria da aplicação.

Na implementação de uma aplicação utilizando os Componentes de Arquitetura do Android Jetpack, os componentes da arquitetura da aplicação ficam separados conforme ilustrado na figura 2.

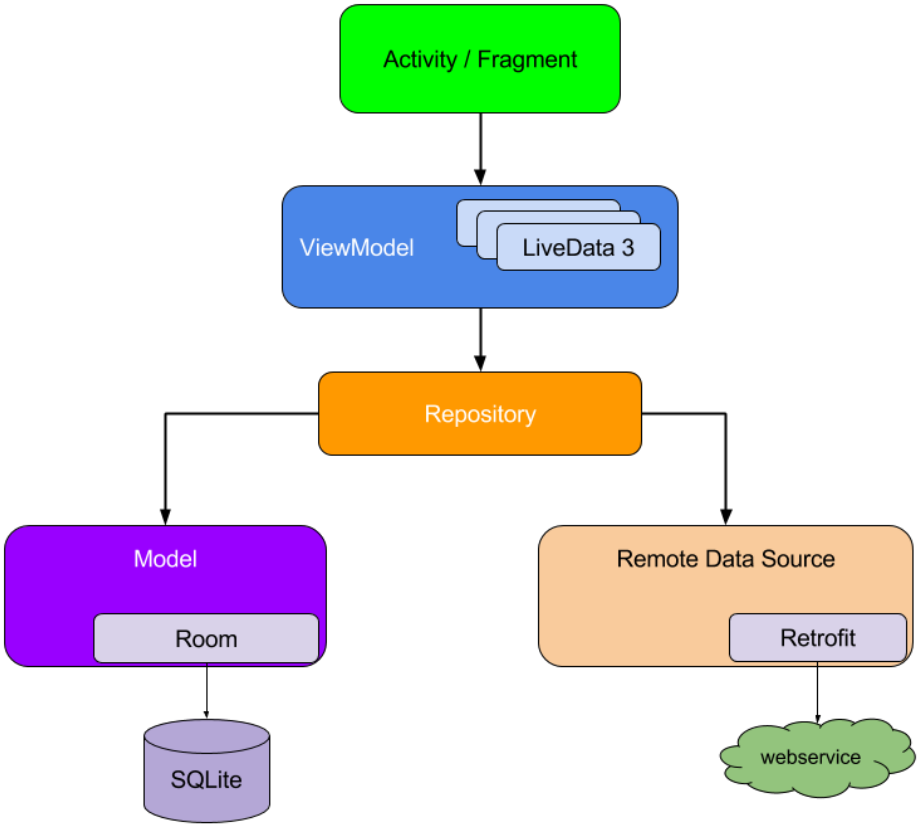


Figura 2. Diagrama da forma recomendada pela Google de utilização do Android Jetpack na arquitetura de aplicações.

2.4. Considerações

Ao longo da fundamentação teórica foram descritos diversos padrões de arquitetura usados no desenvolvimento de aplicações Android. Com base nas vantagens e desvantagens de cada abordagem, pode-se identificar uma arquitetura simples, porém completa, que possa auxiliar no desenvolvimento de novas aplicações Android, suprindo as limitações da plataforma e de sua documentação, ao mesmo tempo que traz os principais benefícios que os desenvolvedores Android esperam de uma arquitetura.

3. Desenvolvimento

Neste capítulo é descrita a arquitetura proposta para o desenvolvimento de aplicações Android. A proposta deste Trabalho de Conclusão de Curso é propor uma arquitetura simples, porém completa, voltada ao desenvolvimento de aplicações Android, e que visa combinar as vantagens dos componentes disponíveis no Android Jetpack com as vantagens de um padrão de arquitetura adequado para aplicações móveis. Essa arquitetura foi proposta na forma de um boilerplate que demonstra como organizar o código da aplicação seguindo a arquitetura.

3.1. Boilerplate

Boilerplate vem de uma expressão em inglês, originalmente do ramo jornalístico, onde boilerplate significa chapa de ebulição [23]. É muito utilizado nas esteiras para a preparação de textos para serem impressos em jornais em massa. No ramo da programação, esse termo ficou conhecido como qualquer trecho de código que pode ser usado como modelo inicial para construção de aplicações.

Para o boilerplate, foram utilizadas as mais recentes recomendações do Google para a arquitetura de aplicações Android [20], como a utilização dos Componentes de Arquitetura do Android Jetpack. Além de serem a recomendação do Google, esses componentes são uma escolha interessante por possuírem uma integração natural com o framework Android, o que reduz o esforço dos desenvolvedores em utilizá-los no ambiente Android.

As classes e arquivos desse boilerplate foram organizados de forma tão específica quanto necessário para que a arquitetura apresentada possa ser compreendida, e tão genérica quanto possível para que desenvolvedores possam usá-los no desenvolvimento de qualquer aplicação Android.

3.2. Arquitetura

A arquitetura desenvolvida é composta principalmente por 3 pacotes: modelo (Model), visão (View) e modelo da visão (ViewModel).

O pacote modelo possui os seguintes componentes:

- repositório (Repository);
- fontes de dados (DataSources);
- lógica de negócio (BusinessLogic);
- e entidades (Entities).

A visão foi dividida entre componentes declarativos (arquivos XML no diretório res) e componentes em Kotlin. Os componentes Kotlin são os seguintes:

- fragmentos (Fragment);
- atividade principal (MainActivity);
- auxiliares de data binding (BindAdapter);
- e dados da visão (ViewData).

Basicamente, o fluxo de dados na aplicação é o seguinte:

1. A *View* recebe ações do usuário e envia comandos ao *ViewModel*;
2. *ViewModel* solicita dados ao *Repository* e/ou delega ações à *BusinessLogic*;
3. *Repository* busca dados das fontes de dados (banco de dados, *webservices*, etc), cria entidades e retorna elas ao *ViewModel* em formato de *callbacks*;
4. *ViewModel* atualiza suas estruturas internas, chamadas de *LiveData*, com as novas informações a serem mostradas para o usuário;
5. View recebe notificações de que os dados do ViewModel mudaram, e assim atualiza a tela.

A figura 3 mostra um diagrama completo da arquitetura.

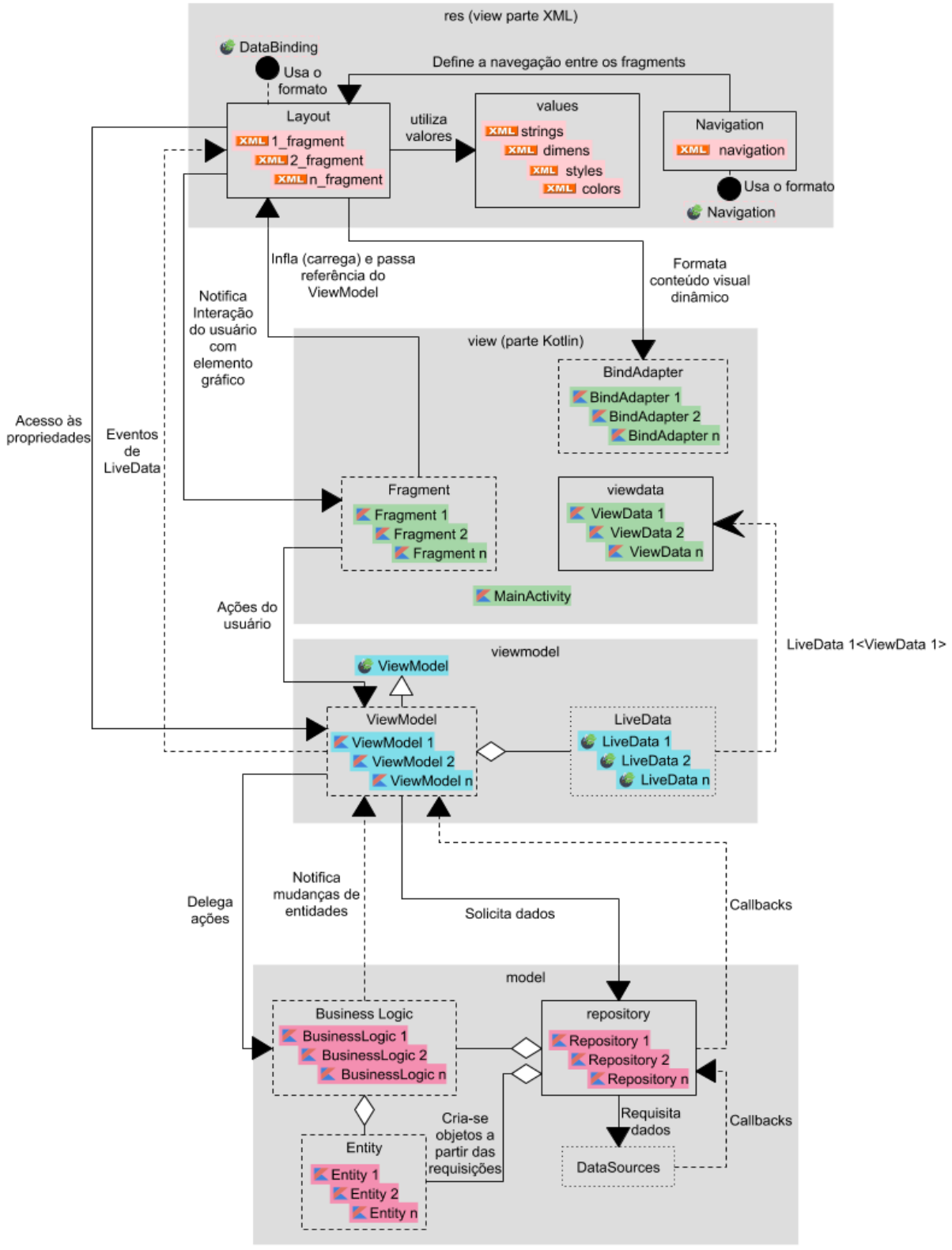


Figura 3. diagrama completo da arquitetura.

3.2.1. View

A figura 4 mostra a View na forma dos dois retângulos cinza superiores com nome de res (view parte XML) e view (parte Kotlin). No Android, o componente View consiste em duas partes distintas: classes Kotlin e arquivos declarativos no formato XML.

As classes Kotlin são primariamente controladores de UI. Além disso, as classes Kotlin da View também incluem BindAdapters e o pacote que contém classes de dados da visão (ViewData), que são os objetos que fornecem dados a serem exibidos na tela e são encapsulados por *LiveData*. Esses objetos devem ser compostos por dados básicos da plataforma, como strings, números e booleanos, e também podem ser agrupamentos desses dados básicos. Nesse *boilerplate* a recomendação é que os *ViewData* possuam apenas as informações suficientes para que os layouts e *BindAdapters* os utilizem para a exibição correta dos dados. É necessário que as informações nos *ViewData* já estejam formatadas, de forma a não necessitar de lógica extra para serem exibidas.

Para aumentar a expressividade dos documentos XML da View, e dessa forma auxiliar na manutenção e testabilidade da aplicação, o componente *View* utiliza vínculos de dados (*data binding*). É uma forma de vincular as propriedades do *ViewModel* (os objetos *ViewData*, que contém os dados necessários para a *View* renderizar na tela) diretamente aos componentes gráficos nos layouts.

Os *bind adapters* são classes Kotlin que customizam o processo de pegar um valor num objeto ViewData e passá-lo para o componente gráfico no layout. Essas classes possuem o propósito de manipular esses objetos *ViewData* e prepará-los para serem fornecidos aos layouts através do *data binding*.

Os arquivos XML estão localizados em uma estrutura de diretórios localizados na pasta res do projeto Android. Contém arquivos de layouts onde são declarados elementos gráficos. Os arquivos no diretório values são arquivos que definem constantes do programa utilizadas nos layouts, enquanto o arquivo navigation que define a navegação entre os fragmentos.

3.2.2. ViewModel

O componente *ViewModel* possui o mesmo papel do *ViewModel* presente na arquitetura MVVM, e utiliza o próprio componente *ViewModel* disponível nos Componentes de Arquitetura do Android Jetpack. Cada classe de *ViewModel* herda da super classe ViewModel presente no Android Jetpack. Esses componentes são cientes dos ciclos de vida dos componentes de visão, como os controladores de UI, e, portanto, permanecem sempre disponíveis independentemente do ciclo de vida dos componentes de visão.

Na arquitetura proposta neste trabalho, recomenda-se que seja criado um *ViewModel* para cada tipo de dado no domínio da aplicação, da mesma forma como podem existir diferentes rotas de *webservices* ou diferentes tabelas no banco de dados representando cada

tipo de dado no domínio da aplicação. Essa separação visa reduzir o acoplamento entre telas da aplicação e aumentar a coesão de *ViewModels*.

Os atributos de um *ViewModel* são constituídos de *ViewData* encapsulados por *LiveData*. O envio de dados entre *View* e *ViewModel* de forma reativa é realizado pelas objetos *LiveData*. É uma classe que implementa o padrão *Observer*. De forma a estender o padrão Observer tradicional, o funcionamento do *LiveData* é ciente do ciclo de vida dos controladores de UI. Para isso, sempre que um novo observador é adicionado ao *LiveData*, é passado também como parâmetro um controlador de UI. Eventos de alteração nos dados observados só são enviados aos observadores se esse controlador de UI encontra-se ativo. Além disso, se o controlador de UI é destruído, o vínculo se encerra e os observadores são removidos.

3.2.3. Model

O modelo é responsável pelas entidades, que consistem nas classes básicas de dados do domínio da aplicação. O modelo também é responsável pelos repositórios, que realizam as requisições de dados, e a lógica de negócio da aplicação.

Entidades representam os objetos básicos do domínio da aplicação e seus atributos. Elas não possuem funcionalidade nenhuma, além de definir o formato dos objetos do domínio com seus respectivos atributos.

Para as requisições de dados, utiliza-se o padrão de repositório (*Repository*). Cada tipo de dado do domínio da aplicação, geralmente representado por uma rota num *webservice* ou uma tabela num banco de dados, possui seu próprio *Repository*. O propósito dos *Repository* é fazer requisições para as fontes de dados (*Data Sources*), que podem ser requisições HTTP para alguma API, uma consulta no banco de dados, ou qualquer recurso que forneça os dados necessários para criar as entidades. Geralmente, a comunicação entre os repositórios e as fontes de dados é realizada através de chamadas assíncronas.

As classes de lógica de negócio são responsáveis por processar os dados e produzir resultados que alterem os objetos do modelo, o que gera notificações para o *ViewModel*. As classes da lógica do negócio são acessadas diretamente pelos repositórios, porém, também podem receber mensagens do *ViewModel* solicitando ações.

4. Avaliação

Para avaliação da arquitetura proposta e do boilerplate, foi elaborado um documento de requisitos para uma aplicação simples que contém duas telas, duas requisições a um webservice e mostra os dados retornados nas requisições nessas duas telas. Foram desenvolvidas duas aplicações seguindo o documento de requisitos e utilizando como base apenas o boilerplate produzido e a documentação que o acompanha. O objetivo da avaliação é analisar o código desenvolvido nessas aplicações utilizando métricas de qualidade de software definidas por outros autores e avaliar a eficácia ou não do uso do boilerplate.

4.1. Métrica de coesão

Uma forma sistemática de medir a coesão de uma classe é medindo o quanto seus métodos se relacionam com seus atributos e entre si. A métrica da Coesão Justa de Classe (TCC, do inglês *Tight Class Cohesion*) é uma forma de medir a coesão de uma classe.

Foi aplicada a métrica nas aplicações desenvolvidas, doravante chamadas Aplicação A e Aplicação B. A métrica também foi aplicada no boilerplate em si, que, apesar de ser uma estrutura genérica, consiste em uma aplicação modelo, a qual deve ser substituída pela aplicação real do usuário. Além disso, como forma de comparar a coesão das aplicações criadas a partir do *boilerplate* com aplicações Android reais, a métrica também foi aplicada a uma aplicação Android chamada *SimpleMusicPlayer*. O resultado é mostrado no gráfico 1.

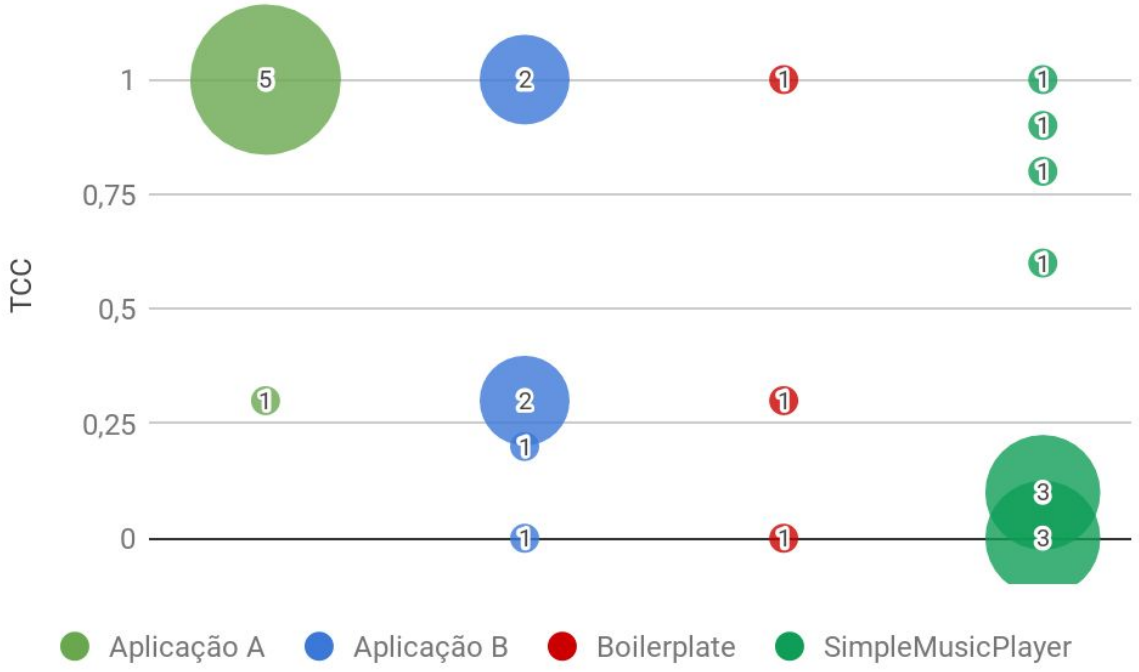


Gráfico 1. resultado da métrica de coesão.

De acordo com os autores da métrica, uma classe com TCC menor que 0,5 é considerada uma classe com baixa coesão. Uma classe com TCC maior que 0,8 é considerada uma classe com alta coesão, sendo um TCC igual a 1 a coesão máxima, onde todos os métodos estão conectados [21].

Como é uma métrica aplicada a cada classe, o gráfico 1 mostra no eixo y os valores obtidos de TCC, enquanto o tamanho das bolhas indica a quantidade de classes que possuem aquele valor de TCC, dentre as classes elegíveis a serem analisadas por essa métrica (classes sem métodos ou com apenas 1 método não são possíveis de serem analisadas pela métrica).

Nota-se que a maior parte das classes criadas pelos usuários possuem um alto nível de coesão e tiveram um bom resultado na métrica utilizada. É possível também comparar o resultado das duas aplicações criadas com a aplicação *SimpleMusicPlayer*, que não foi desenvolvida seguindo o *boilerplate* nem a arquitetura proposta. Nota-se que boa parte das classes se aproximam da baixa coesão, evidenciando que a aplicação possui, em sua maioria, classes com baixa coesão, característica não desejada em aplicações.

4.2. Métrica de acoplamento

Define-se acoplamento entre classes quando uma classe possui uma referência a outra, possuindo acesso a métodos e/ou atributos da outra. Um programa com classes muito acopladas umas às outras resulta em classes de difícil reuso e encapsulamento, e também dificulta a criação de testes unitários. Quanto menos acoplamento tiver, mais reusáveis são as classes do programa e mais fácil fica criar testes unitários sem muita necessidade de criar mocks para as classes dependentes.

O gráfico 2 mostra o valor de acoplamento do boilerplate, dos programas dos usuários A e B, e da aplicação SimpleMusicPlayer. Nota-se que os fatores de acoplamento dos dois programas desenvolvidos ficaram menores do que o do próprio boilerplate. Esse resultado mostra que o código novo adicionado durante a criação das aplicações, tanto em classes diretamente derivadas do boilerplate quanto em classes novas, não adicionou mais acoplamento à aplicação, e, portanto, não prejudicou as métricas de acoplamento.

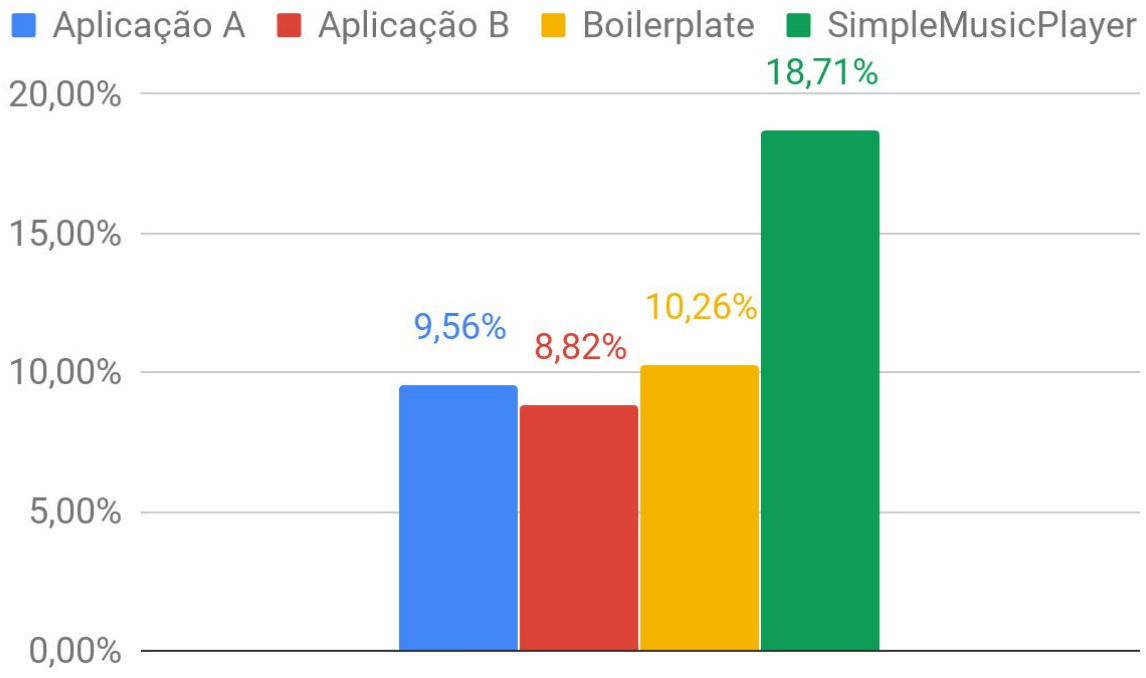


Gráfico 2. resultado da métrica de acoplamento.

Os dois programas ficaram com o fator de acoplamento abaixo da média de 10,8% obtida pelos autores da métrica, sendo o boilerplate inicial o que possui maior índice (10,26%), pela pouca quantidade de classes. Valores abaixo do máximo obtido pelos autores, de 17,7%, são considerados valores aceitáveis [22]. Como as aplicações desenvolvidas obtiveram fator de acoplamento abaixo da média e bem distante no valor máximo, considera-se que elas possuem baixo acoplamento.

5. Conclusão

O objetivo de propor uma arquitetura e construir um boilerplate para o desenvolvimento de aplicações Android é justificável quando se afirma que o Android tornou-se o sistema operacional dominante no mercado e a comunidade de desenvolvedores da plataforma enfrenta dificuldades em organizar o código de suas aplicações. Sendo um sistema operacional dominante, uma arquitetura adequada e um boilerplate que ilustra seu uso podem potencialmente ser uma solução para muitos programadores.

5.1. Resultados

No desenvolvimento da arquitetura, os Componentes de Arquitetura do Android Jetpack, lançados recentemente, se mostraram apropriados para serem usados numa arquitetura para o desenvolvimento de aplicações Android.

Essa arquitetura proposta neste trabalho e definida no boilerplate foi avaliada e os resultados mostram que ela é capaz de ajudar os desenvolvedores, tanto iniciantes quanto experientes, a evitar construir as aplicações em volta de atividades, resultando em aplicações com código mais bem distribuído e coeso.

As classes e arquivos desse *boilerplate* foram organizados de forma tão específica quanto necessário para que a arquitetura apresentada possa ser compreendida, e tão genérica quanto possível para que desenvolvedores possam usá-los no desenvolvimento de qualquer aplicação Android.

Foram aplicadas métricas de acoplamento e coesão nas aplicações desenvolvidas e foram obtidos resultados satisfatórios com todas as aplicações. Por fim,

5.2. Limitações

Ao longo da avaliação feita do *boilerplate* e da arquitetura propostos, foram percebidas algumas limitações e oportunidades de melhoria na solução.

Pela simplicidade do *boilerplate*, a comunicação entre os *ViewModels* não é definida de forma clara. É orientado no texto que exista um *ViewModel* para cada tipo de dado no domínio da aplicação, mas não se especifica como que a comunicação entre domínios é feita, deixando para o programador resolver.

Por fim, as aplicações criadas não são aplicações complexas e de larga escala, o que pode ter feito com que algumas deficiências da arquitetura ou do *boilerplate* não tenham ficado visíveis.

5.3. Trabalhos futuros

Para trabalhos futuros, é interessante tratar as limitações identificadas. Pode-se adicionar uma orientação, através de exemplos, na comunicação entre repositórios e fontes de dados. A criação de um programa gerador de *boilerplate*, com um nome de pacote personalizado, também é algo pertinente, pois a edição do nome do pacote de um projeto Android é ligeiramente complicada.

5.4. Considerações finais

Acredita-se que a arquitetura proposta e o *boilerplate* que a acompanha resolvem diversos problemas que desenvolvedores Android enfrentam no dia-a-dia, e servem tanto como uma boa base para a criação de novas aplicações Android, como também como um guia de boas práticas no desenvolvimento de aplicações Android.

Referências

[1] MOBILE Operating System Market Share Worldwide | StatCounter Global Stats. [2017 - 2018]. Disponível em: <http://gs.statcounter.com/os-market-share/mobile/worldwide>. Acesso em: 26 nov. 2018.

[2] DÜRSCHMID, M. T.; DÖLLNER, J. Towards Architectural Styles for Android App Software Product Lines. IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Buenos Aires, p. 58-62, 2017.

[3] DUPREE, Kevin. **MVPR: A Flexible, Testable Architecture for Android**. 7 de Jul 2015. Disponível em <https://www.philosophicalhacker.com/2015/07/07/mvpr-a-flexible-testable-architecture-for-android-pt-1/>. Acesso em 4 jun. 2019.

[4] IDESIS, Stanley. **Develop Your First Android Application**: Learn the Model-View-Controller Pattern. 2018. Disponível em: <https://openclassrooms.com/en/courses/4661936-develop-your-first-android-application/4679186-learn-the-model-view-controller-pattern>. Acesso em: 26 nov. 2018.

[5] DAOUDI, Aymen et al. An exploratory study of MVC-based architectural patterns in Android apps. In: **Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing**. ACM, 2019. p. 1711-1720.

[6] Use Android Jetpack to Accelerate Your App Development. Disponível em: <https://android-developers.googleblog.com/2018/05/use-android-jetpack-to-accelerate-your.html>. Acesso em: 3 jun. 2019.

[7] BERGSTROM, Lukas. **Announcing Architecture Components 1.0 Stable.** Disponível em: <https://android-developers.googleblog.com/2017/11/announcing-architecture-components-10.html>. Acesso em: 4 jun. 2019.

[8] LARDINOIS, Frederic. **Kotlin is now Google's preferred language for Android app development.** Disponível em: <https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/>. Acesso em: 4 jun. 2019.

[9] Desenvolvedores Android - Tecnology - Arquitetura da plataforma. Disponível em: <https://developer.android.com/guide/platform>. Acesso em: 3 jun. 2019.

[10] SIMPSON, Ronnie. **Android overtakes Windows for first time**: "Milestone in technology history and end of an era" as Microsoft no longer owns dominant OS. 2017. Disponível em: <http://gs.statcounter.com/press/android-overtakes-windows-for-first-time>. Acesso em: 26 nov. 2018.

[11] Context | Android Developers. Disponível em: <https://developer.android.com/reference/android/content/Context>. Acesso em: 4 jun. 2019.

[12] KHANNA, Gaurav. **Mastering Android context.** 5 de Jun 2018. Disponível em: <https://www.freecodecamp.org/news/mastering-android-context-7055c8478a22/>. Acesso em: 5 jun. 2019.

[13] SOKOLOVA, Karina; LEMERCIER, Marc; GARCIA, Ludovic. Android passive MVC: a novel architecture model for the android application development. **International Conference on Pervasive Patterns and Applications**. 2013.

[14] MUNTENESCU, Florina. **Android Architecture Patterns Part 1: Model-View-Controller**. 1 de Nov 2016. Disponível em <https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6>. Acesso em 4 jun. 2019.

[15] MAXWELL, Eric. **The MVC, MVP, and MVVM Smackdown**. 2017. Disponível em: <https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/>. Acesso em: 26 nov. 2018.

[16] FOWLER, Martin. **GUI Architectures**. 2006. Disponível em: <https://www.martinfowler.com/eaaDev/uiArchs.html>. Acesso em: 2 jun. 2019.

[17] MOORE, Dana; BUDD, Raymond; BENSON, Edward. **Professional Rich Internet Applications: AJAX and Beyond**. John Wiley & Sons, 2007.

[18] GOSSMAN, John. **Introduction to Model/View/ViewModel pattern for building WPF apps**. 8 de out. 2015. Disponível em <https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/>. Acesso em 5 jun. 2019.

[19] The MVVM Pattern | Microsoft Docs. Disponível em <https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10)>. Acesso em 5 jun. 2019.

[20] Guia para a arquitetura do app | Android Developers. Disponível em <https://developer.android.com/jetpack/docs/guide>. Acesso em 5 jun. 2019.

[21] BIEMAN, James M.; KANG, Byung-Kyoo. Cohesion and reuse in an object-oriented system. ACM SIGSOFT Software Engineering Notes, v. 20, n. si, p. 259-262, 1995.

[22] ABREU, F. Brito; GOULÃO, Miguel; ESTEVES, Rita. Toward the design quality evaluation of object-oriented software systems. **Proceedings of the 5th International Conference on Software Quality**, Austin, Texas, USA, p. 44-57, 1995.

[23] Boilerplate | Definition of Boilerplate by Merriam-Webster. Disponível em: <https://www.merriam-webster.com/dictionary/boilerplate>. Acesso em: 10 jul. 2019.

Apêndice 6 - Código boilerplate

```
package io.schiar.tccboilerplate.model.repository

import io.schiar.tccboilerplate.model.ArchComponent
import io.schiar.tccboilerplate.model.Library

/**
 * Implementação de um repository de componentes arquiteturais. Fornece os dados a respeito dos
 * componentes arquiteturais.
 *
 * Por motivos de simplificação, essa classe gera dados de componentes arquiteturais para a aplicação
 * e os mantém em memória durante a execução. Numa aplicação real, essa classe se comunicaria
 * com as diferentes camadas de dados da aplicação, como por exemplo serviços e persistência.
 *
 * @property archComponents componentes arquiteturais gerados para serem exibidos na lista de componentes arquiteturais.
 * Numa aplicação real, esses objetos viriam de uma camada de dados da aplicação.
 */
class ArchComponentRepository: ArchComponentRepositoryInterface {
        private val archComponents = listOf(
        ArchComponent("Data Binding",
                "Serve para o XML da view ter acesso direto ao View Model",
                Library("com.android.databinding:compiler", listOf(3, 3, 2))
        ),
```

```
        ArchComponent("ViewModel",
        "Componente que busca e formata os dados para serem exibidos na View",
        Library("androidx.lifecycle:lifecycle-viewmodel-ktx", listOf(2, 0, 0))
        ),

        ArchComponent("LiveData",
        "Contêiner de dados. Quando o dado é modificado, notifica quem o observa",
        Library("androidx.core:core-ktx:", listOf(1, 0, 0))
        ),

        ArchComponent("Navigation",
        "Controla a navegação da aplicação",
        Library("android.arch.navigation:navigation-fragment:", listOf(1, 0, 0))
        )
        )

        /**
        * Busca a lista de componentes arquiteturais a ser exibida na View.
        * @param callback usado para receber a lista de componentes arquiteturais
buscada.
        */
        override fun fetch(callback: (List<ArchComponent>) -> Unit) {
        return callback(archComponents)
        }

}

package io.schiar.tccboilerplate.model.repository

import io.schiar.tccboilerplate.model.ArchComponent

/**
```

```
 * Contrato de um fornecedor de dados para a aplicação.
 * O padrão repository proporciona uma abstração da camada de dados da
aplicação.
 * Além disso, ele centraliza o uso dos objetos do domínio.
 * Através de um repository, outros componentes da aplicação conseguem
 * manejar os objetos do domínio de forma simples, sem precisar
 * conhecer de fato de onde esses objetos vêm e onde são armazenados (internet,
banco de dados, caches, etc).
 * Isso permite que todos os componentes que usam o repository possuam um baixo
 * acomplamento com as camadas de serviço e persistência da aplicação.
 */
interface ArchComponentRepositoryInterface {
        /**
        * Busca a lista de componentes arquiteturais a ser exibida na View.
        * @param callback usado para receber a lista de componentes arquiteturais
buscada.
        */
        fun fetch(callback: (List<ArchComponent>) -> Unit )
}

package io.schiar.tccboilerplate.model

/**
 * Representa um componente arquitetural.
 * @property name nome do componente.
 * @property description descrição do componente.
 * @property library biblioteca do componente.
 */
data class ArchComponent(
        val name: String,
        val description: String,
        val library: Library
```

```
)

package io.schiar.tccboilerplate.model

/**
 * Representa uma biblioteca.
 * @property name nome da biblioteca.
 * @property version versão da biblioteca.
 */
data class Library(
        val name: String,
        val version: List<Int>
)

package io.schiar.tccboilerplate.view.viewdata

/**
 * Representação dos componentes arquiteturais do ponto de vista da visão.
 * @property name nome de um componente.
 * @property description descrição de um componente.
 * @property library biblioteca de um componente.
 */
data class ArchComponentViewData(
        val name: String,
        val description: String,
        val library: LibraryViewData
)

package io.schiar.tccboilerplate.view.viewdata

/**
 * Representação das bibliotecas do ponto de vista da visão.
```

```
 * @property name nome de uma biblioteca.
 * @property version versão de uma biblioteca.
 */
data class LibraryViewData(
        val name: String,
        val version: String
) {
        /**
        * Método auxiliar para gerar um nome completo da biblioteca.
        * @return o nome completo da biblioteca.
        */
        override fun toString(): String = "$name:$version"
}

package io.schiar.tccboilerplate.view

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.databinding.DataBindingUtil
import androidx.recyclerview.widget.RecyclerView
import io.schiar.tccboilerplate.R
import io.schiar.tccboilerplate.view.viewdata.ArchComponentViewData
import io.schiar.tccboilerplate.databinding.AdapterArchComponentBinding

/**
 * Trata a lista de ViewDatas para ser exibida pelo componente [RecyclerView].
 * @param archComponents lista de ViewDatas.
 **/
class ArchComponentsListAdapter(private val archComponents:
List<ArchComponentViewData>):
        RecyclerView.Adapter<ArchComponentsListAdapter.ViewHolder>() {
```

```
        /**
        * Usado para carregar o XML do layout adapter_arch_component com
DataBinding que representa um item da lista.
        * @param parent componente pai do item
        * @param viewType o id do tipo da View. Aqui ignorado e utilizado o
adapter_arch_component.
        * @return o objeto ViewHolder.
        */
        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        val binding = DataBindingUtil.inflate<AdapterArchComponentBinding>(inflater,
R.layout.adapter_arch_component, parent,false);

        return ViewHolder(binding);
        }

        /**
        * Quantidade de itens.
        * @return a quantintidade de items.
        */
        override fun getItemCount(): Int {
        return archComponents.size
        }

        /**
        * Usado no momento em que o item da lista é exibido.
        * @param holder o objeto que trata o item gráfico da lista.
        * @param position a posição do item na lista.
        */
        override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bind(archComponents[position]);
```

```
        }

        /**
        * interna ao Adapter. Utilizada por ele para tratar o item gráfico da lista.
        * @param binding o objeto do XML do item da lista.
        */
        class ViewHolder(private val binding: AdapterArchComponentBinding):
RecyclerView.ViewHolder(binding.root) {
        /**
        * configura o item gráfico com o viewdata.
        */
        fun bind(archComponent: ArchComponentViewData) {
        binding.apply {
                this.archComponent = archComponent
                executePendingBindings()
        }
        }
        }
}

package io.schiar.tccboilerplate.view

import android.widget.TextView
import androidx.databinding.BindingAdapter
import io.schiar.tccboilerplate.R

/**
 * Utilizado para tratamento de dados do ViewModel para serem apresentados na
View através de data binding.
 */
object BindingAdapters {
        /**
```

```
        * É adicionado o valor a um label no [TextView]
        */
        @BindingAdapter(value= ["label", "value"])
        @JvmStatic
        fun setLabelValue(textView: TextView, label: String, value: String?) {
        value ?: return
        val viewText = textView.text
        val labelValue = textView.context.getString(R.string.label_value)
        textView.text = String.format(labelValue, label, value)
        }

}

package io.schiar.tccboilerplate.view

import androidx.lifecycle.ViewModelProviders
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.Observer
import androidx.navigation.Navigation
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import io.schiar.tccboilerplate.R
import io.schiar.tccboilerplate.viewmodel.BoilerplateViewModel
import io.schiar.tccboilerplate.databinding.FragmentBoilerplateBinding
import io.schiar.tccboilerplate.view.viewdata.ArchComponentViewData
import kotlinx.android.synthetic.main.fragment_boilerplate.view.*

/**
```

```
 * Mostra a lista de componentes arquiteturais utilizados neste boilerplate
 * @property viewModel ViewModel necessário para mostrar os dados necessários do modelo na View.
 */
class BoilerplateFragment : Fragment() {

        private lateinit var viewModel: BoilerplateViewModel

        /**
        * É carregado o [BoilerplateViewModel] para passar ao databinding do XML, assim o XML tem acesso aos atributos e
        * métodos do ViewModel.
        * @param inflater usado para carregar o XML do fragmento.
        * @param container o componente pai do fragmento.
        * @param savedInstanceState dados do estado anterior do fragmento.
        * @return view correspondente ao fragmento.
        */
        override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
        ): View {
        viewModel = ViewModelProviders.of(this).get(BoilerplateViewModel::class.java)
        val binding = FragmentBoilerplateBinding.inflate(inflater, container, false).apply {
        lifecycleOwner = this@BoilerplateFragment
        viewModel = this@BoilerplateFragment.viewModel
        executePendingBindings()
        }
        viewModel.fetch()
        val view = binding.root
```

```
view.navigation_example_btn.setOnClickListener(::onNavigationButtonPressed)
        viewModel.archComponents.observe(this, Observer {
        binding.adapter = ArchComponentsListAdapter(it)
        })
        return view
        }

        /**
        * Evento disparado quando o botão de exemplo de navegação é clicado. Ele
navega para [OtherFragment].
        * @param view componente de visão do botão.
        */
        private fun onNavigationButtonPressed(view: View) {
        val navId = R.id.boilerplate_to_other
        Navigation.findNavController(view).navigate(navId)
        }

}

package io.schiar.tccboilerplate.view

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.navigation.NavController
import androidx.navigation.Navigation
import androidx.navigation.ui.NavigationUI
import io.schiar.tccboilerplate.R
import kotlinx.android.synthetic.main.activity_main.*

/**
 * Atividade que controla toda a navegação dos fragmentos da aplicação.
```

```
 * @property navController utilizado para o navigation.
 */
class MainActivity : AppCompatActivity() {

        private lateinit var navController: NavController

        /**
        * Define qual o XML de view que a Atividade vai usar, define a toolbar criada no XML como a barra superior
        * da aplicação, e passar o controle de navegação para ela, para assim poder integrar o navigation a aplicação
        * @param savedInstanceState guarda dados da última execução permitindo restaurar o estado anterior.
        */
        override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setSupportActionBar(toolbar)
        navController = Navigation.findNavController(this, R.id.initial_fragment)
        NavigationUI.setupActionBarWithNavController(this, navController)
        }

        /**
        * Utilizar o método do navigation para voltar ao fragmento anterior quando há um evento de botão da barra superior.
        * de voltar
        * @return true se foi possível voltar ao fragmento anterior, false caso contrário
        */
        override fun onSupportNavigateUp(): Boolean {
        return navController.navigateUp() || super.onSupportNavigateUp()
        }
```

```
        /**
        * Utilizar o método do navigation para voltar ao fragmento anterior quando há um evento de botão físico de voltar.
        */
        override fun onBackPressed() {
        super.onBackPressed()
        navController.navigateUp()
        }
}

package io.schiar.tccboilerplate.view

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.ViewModelProviders

import io.schiar.tccboilerplate.databinding.FragmentOtherBinding
import io.schiar.tccboilerplate.viewmodel.OtherViewModel

/**
 * Mostra uma string pré criada do ViewModel para fins de demonstração do DataBinding
 * @property viewModel ViewModel necessário para mostrar os dados necessários do modelo na View.
 */
class OtherFragment : Fragment() {

        private lateinit var viewModel: OtherViewModel
```

```
        /**
        * É carregado o [OtherViewModel] para passar ao databinding do XML, assim
o XML tem acesso aos atributos e
        * métodos do ViewModel.
        * @param inflater usado para carregar o XML do fragmento.
        * @param container o componente pai do fragmento.
        * @param savedInstanceState dados do estado anterior do fragmento.
        * @return view correspondente ao fragmento.
        */
        override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
        ): View? {
        viewModel = ViewModelProviders.of(this).get(OtherViewModel::class.java)
        val binding = FragmentOtherBinding.inflate(inflater, container, false).apply {
        lifecycleOwner = this@OtherFragment
        viewModel = this@OtherFragment.viewModel
        executePendingBindings()
        }
        return binding.root
        }

}

package io.schiar.tccboilerplate.viewmodel

import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import io.schiar.tccboilerplate.model.ArchComponent
import io.schiar.tccboilerplate.model.repository.ArchComponentRepository
import io.schiar.tccboilerplate.model.repository.ArchComponentRepositoryInterface
```

```
import io.schiar.tccboilerplate.view.viewdata.ArchComponentViewData
import io.schiar.tccboilerplate.view.viewdata.LibraryViewData

/**
 * Recebe mensagens da visão solicitando dados.
 * Formata esses dados e os disponibiliza para a visão através dos objetos
LiveData.
 * @property archComponentRepository fornecedor de objetos de modelo para o
ViewModel.
 * @property archComponents lista atual de componentes arquiteturais.
 * @property buttonContent conteúdo do botão da tela.
 */
class BoilerplateViewModel(
        private val archComponentRepository: ArchComponentRepositoryInterface =
ArchComponentRepository()
) : ViewModel() {

        val archComponents: MutableLiveData<List<ArchComponentViewData>> by
lazy {
        MutableLiveData<List<ArchComponentViewData>>()
        }

        val buttonContent = MutableLiveData<String>().apply { value = "Exemplo de
navegação" }

        /**
        * Busca os dados de componentes arquiteturais e atualiza o LiveData de
[archComponents].
        */
        fun fetch() {
        archComponentRepository.fetch {
        archComponents.postValue(it.map {archComponent: ArchComponent ->
```

```
                ArchComponentViewData(
                archComponent.name,
                archComponent.description,
                LibraryViewData(
                archComponent.library.name,
                archComponent.library.version.joinToString(".")
                )
                )
        })
        }
        }
}

package io.schiar.tccboilerplate.viewmodel

import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel

/**
 * Recebe mensagens da visão solicitando dados.
 * Formata esses dados e os disponibiliza para a visão através dos objetos
LiveData.
 * @property helloWorld usado no XML para ser mostrado na tela.
 */
class OtherViewModel: ViewModel() {
        val helloWorld = MutableLiveData<String>().apply { value = "HelloWorld" }

        /* TODO: Criar LiveDatas, ver exemplo em BoilerplateViewModels */
}

package io.schiar.tccboilerplate.mock
```

```
import org.mockito.Mockito

/**
 * Criação de mocks.
 */
class GenericMockFactory {
	/**
	* Cria mocks onde é possível utilizar classes que recebem parâmetros de tipo (generics).
	* @return apenas ignora o parâmetro de tipo e chama o mock da biblioteca Mockito.
	*/
	companion object {
	inline fun <reified T> mock(): T {
	return Mockito.mock(T::class.java)
	}
	}
}

package io.schiar.tccboilerplate.mock

import io.schiar.tccboilerplate.model.ArchComponent
import io.schiar.tccboilerplate.model.Library
import io.schiar.tccboilerplate.model.repository.ArchComponentRepositoryInterface

/**
 * Mock específico de um repositório de componente arquitetural. Usado para os testes unitários.
 */
class MockArchComponentRepository : ArchComponentRepositoryInterface {
	/**
	* retorna sempre a mesma lista com um único componente arquitetural.
```

```
        * @param callback usado para receber a mesma lista com um único
componente arquitetural.
        */
        override fun fetch(callback: (List<ArchComponent>) -> Unit) {
        val archComponent = ArchComponent(
        "Mock",
        """Serve para simular de forma síncrona e isolada o comportamento de uma
classe real.
                Utilizar uma classe real no teste muitas vezes é impraticável pois
alguns dos seus métodos
                podem ser assíncronos e dependentes de componentes externos
como rede ou banco de dados.
                Embora alguns mocks possam ser criados na mão, mocks genéricos
podem facilmente ser criados
                utilizando o framework Mockito.""",
        Library("org.mockito:mockito-core", listOf(2, 27, 0))
        )
        callback(listOf(archComponent))
        }
}

package io.schiar.tccboilerplate

import androidx.arch.core.executor.testing.InstantTaskExecutorRule
import androidx.lifecycle.Observer
import io.schiar.tccboilerplate.mock.GenericMockFactory.Companion.mock
import io.schiar.tccboilerplate.mock.MockArchComponentRepository
import io.schiar.tccboilerplate.view.viewdata.ArchComponentViewData
import io.schiar.tccboilerplate.view.viewdata.LibraryViewData
import io.schiar.tccboilerplate.viewmodel.BoilerplateViewModel
import org.junit.Assert.assertEquals
import org.junit.Assert.assertNull
```

```
import org.junit.Before
import org.junit.Rule
import org.junit.Test
import org.mockito.Mockito.verify

/**
 * Testa o [BoilerplateViewModel]
 * @property rule permite-se utilizar LiveData no JUnit.
 * @property archComponentViewData usado para verificar se buscando os
componentes arquitetuais do repositório mock
 * retorna um igual a esse.
 * @property boilerplateViewModel usado para testar todos os testes da classe.
 */
class BoilerplateViewModelUnitTest {
        @get:Rule
        val rule = InstantTaskExecutorRule()

        private val archComponentViewData = ArchComponentViewData(
        "Mock",
        """Serve para simular de forma síncrona e isolada o comportamento de uma
classe real.
                Utilizar uma classe real no teste muitas vezes é impraticável pois
alguns dos seus métodos
                podem ser assíncronos e dependentes de componentes externos
como rede ou banco de dados.
                Embora alguns mocks possam ser criados na mão, mocks genéricos
podem facilmente ser criados
                utilizando o framework Mockito.""",
        LibraryViewData("org.mockito:mockito-core", "2.27.0")
        )

        private lateinit var boilerplateViewModel: BoilerplateViewModel
```

```
    /**
    * Executa antes de todos os testes. Cria-se um [BoilerplateViewModel]
passando como parâmetro um repositório mock.
    */
    @Before
    fun prepare() {
    boilerplateViewModel =
BoilerplateViewModel(MockArchComponentRepository())
    }

    /**
    * Verifica se o atributo archComponents de [boilerplateViewModel] é
inicializado com null.
    */
    @Test
    fun archComponents_isInitiallyNull() {
    assertNull(boilerplateViewModel.archComponents.value)
    }

    /**
    * Verifica se o componente buscado no repository definido no construtor
chamando fetch de [boilerplateViewModel]
    * ele atualiza o [boilerplateViewModel]
    */
    @Test
    fun fetch_archComponentsPostsCorrectArchComponentViewData() {
    val observer: Observer<List<ArchComponentViewData>> = mock()
    boilerplateViewModel.archComponents.observeForever(observer)

    boilerplateViewModel.fetch()
    verify(observer).onChanged(listOf(archComponentViewData))
```

```
        }

        /**
        * Verifica se o componente buscado no repository definido no construtor
chamando fetch é igual ao
        * [archComponentViewData]
        */
        @Test
        fun fetch_archComponentsContainsCorrectArchComponentViewData() {
        boilerplateViewModel.fetch()
        assertEquals(listOf(archComponentViewData),
boilerplateViewModel.archComponents.value)
        }
}
```