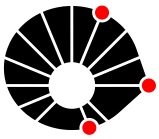

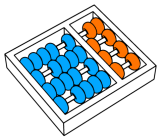

MVP Limpo: Uma Arquitetura para Aplicações Android baseada no estilo MVP e em Arquitetura Limpa

Bernardo do Amaral Teodosio

Relatório Técnico - IC-PFG-20-31
Projeto Final de Graduação
2020 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

MVP Limpo: Uma Arquitetura para Aplicações Android baseada no estilo MVP e em Arquitetura Limpa

Bernardo do Amaral Teodosio

Resumo

O campus da Unicamp é um lugar diverso, onde diferentes experiências podem ser vivenciadas. Cada pessoa que passa pelo campus faz o seu próprio caminho, e tem suas próprias experiências.

Com base nesta temática, o aplicativo Mapa Afetivo foi desenvolvido de forma a permitir que as pessoas possam registrar seus caminhos, suas rotas e, principalmente, descrever as experiências vividas durante seus percursos. Outras pessoas podem, assim, descobrir novos caminhos e vivenciar sentimentos outrora nunca experienciados.

Durante o levantamento de requisitos da aplicação, foi tomada a decisão de que a melhor forma de colocar a ideia em prática seria a partir do desenvolvimento de uma aplicação Android. Naturalmente, um aplicativo eficaz necessita de uma arquitetura eficiente para suportá-lo, garantindo uma boa performance do mesmo e propiciando um processo de desenvolvimento e manutenção adequado. Este trabalho trata da proposta e desenvolvimento da arquitetura do aplicativo desenvolvido, que teve como base o estilo arquitetural MVP (Model-View-Presenter) e adotou princípios de Clean Architecture. Com base nisso, uma arquitetura híbrida foi desenvolvida com o objetivo de apoiar o desenvolvimento da aplicação, considerando um cenário evolutivo. A arquitetura em questão foi criada de tal forma que não esteja fortemente acoplada aos conceitos e funcionalidades do Mapa Afetivo - suas características e estruturas podem ser utilizadas em outros projetos de software, acadêmicos ou comerciais.

Sumário

1 Introdução

Diariamente, milhares de pessoas passam pela Universidade Estadual de Campinas (UNICAMP). Cada uma destas pessoas tem seus próprios objetivos e motivações - cada uma fazendo seu próprio trajeto. Frequentemente, em meio à agitação em que vivemos, os caminhos são percorridos com o foco no objetivo, sem dar muita atenção ao trajeto de fato realizado.

Sendo o campus de Campinas da UNICAMP um lugar amplo e também diverso - com diferentes espaços físicos que vão desde institutos, cantinas, árvores e lugares inesperados, é possível que cada uma dessas pessoas, ao cruzar as diferentes localizações do Campus, experiencie seus próprios sentimentos. Uma caminhada pode trazer consigo muitas experiências - um modo de se relacionar com os ambientes [1], ou ser um simples gesto de ação estética. Pode ainda ser desde um procedimento a um dispositivo criativo [1] - permitindo assim a vivência de diferentes experiências.

Partindo deste pressuposto, surgiu a ideia da criação do Mapa Afetivo - um lugar onde as diferentes pessoas podem registrar seus caminhos percorridos e, principalmente, quais pontos de cada caminho se destacaram. Um ponto de destaque pode ser uma árvore na qual o indivíduo parou um pouco para descansar, uma escultura admirada, uma parede com um grafite - literalmente, qualquer localização com uma latitude e longitude onde o sujeito teve um sentimento.

No Mapa Afetivo, as rotas e pontos podem ser registradas, com os sentimentos e memórias salvos - com um texto ou uma imagem. Estas rotas tornam-se, então, disponíveis para todos os outros usuários da comunidade da UNICAMP - que podem assim conhecer novas rotas, descobrir novos lugares, caminhos e também sentimentos. Além de permitir a descoberta de novos locais, o Mapa Afetivo proporciona aos seus usuários a possibilidade de experienciar sentimentos em lugares nunca antes imaginados.

Partindo da ideia de que um aplicativo móvel seria a forma mais adequada de transformar o Mapa Afetivo num produto real, naturalmente torna-se necessária a criação de uma arquitetura para suportar o mesmo. Como a maioria dos grandes softwares, aplicativos criados para a plataforma Android também se beneficiam de uma arquitetura adequada - que permite que o aplicativo seja não só desenvolvido como também receba manutenções de uma forma fácil e ágil. Para isso, a aplicação de boas práticas de desenvolvimento de software é necessária. Além disso, a arquitetura adotada deve garantir um bom funcionamento da aplicação nos mais diversos modelos de celulares Android existentes, permitindo uma criação fácil de novas funcionalidades por diferentes desenvolvedores, inclusive aqueles não envolvidos inicialmente na criação do projeto.

Considerando o contexto e escopo inicial do projeto, buscou-se uma arquitetura capaz de permitir o trabalho paralelizado dos desenvolvedores, incluindo padrões de código, classes, métodos e entidades do sistema que tornassem o código-fonte como um todo coerente.

Dentre as diferentes arquiteturas existentes atualmente voltadas para aplicações móveis, foi criada uma nova arquitetura, com base em princípios da Arquitetura MVP (*Model-View-Presenter*) [4] e também de Arquitetura Limpa (do inglês, *Clean Architecture*) [2]. A arquitetura proposta permitiu o desenvolvimento do aplicativo de forma paralelizada entre os diferentes desenvolvedores participantes, facilitando o desenvolvimento iterativo e

incremental da aplicação.

Ainda, é válido destacar que a arquitetura aqui descrita pode ser útil a muitos outros projetos além do próprio Mapa Afetivo - existindo assim a possibilidade da mesma ser útil à comunidade acadêmica e também ao desenvolvimento de aplicações em geral.

O restante deste trabalho está organizado da seguinte forma: a Seção 2 apresenta os principais conceitos necessários para a fundamentação teórica do trabalho. A Seção 3 descreve o método de trabalho utilizado para o desenvolvimento do mesmo. Na Seção 4, estão descritos os detalhes técnicos - conceituais e de implementação da arquitetura. Na Seção 5, podemos ver um exemplo da arquitetura em uso. Na Seção **??**, é descrito o comportamento da arquitetura com um módulo particular, tratado como uma dependência externa. Por fim, na Seção 7, encontramos as conclusões extraídas deste trabalho e do processo de desenvolvimento realizado com a arquitetura em questão.

2 Fundamentação Teórica

A arquitetura do Mapa Afetivo teve como base duas arquiteturas bem definidas e de uso comum no mercado de aplicações móveis - mas que não são exclusivas deste tipo de aplicação. Cada uma das arquiteturas tem suas vantagens - e a junção de ambas resultou na arquitetura utilizada pelo aplicativo.

2.1 Clean Architecture

A arquitetura do projeto foi criada como tendo principal base os princípios de *Clean Architecture*, cujo autor é Robert C. Martin [3]. Nessa abordagem, o software é separado em diferentes camadas [2], como pode ser visto na Figura 1.

A ideia principal é que o software siga a "Regra das Dependências", que é a restrição de uma arquitetura clássica em camadas [27]. Assim, cada círculo na figura representa uma camada do software, e a regra que diz que as dependências entre as camadas só podem existir no sentido "externo" - isto é, as camadas externas podem depender das camadas internas, mas o contrário não pode ocorrer.

A divisão em camadas seguindo a Regra das Dependências traz vantagens - sendo uma das principais a facilidade para a realização de testes no software. Como o software está bem modularizado, testar as camadas mais internas é simples - uma vez que possuem nenhuma ou poucas dependências. Testar as camadas mais externas também não é difícil - uma vez que é simples criar dublês de teste para os objetos necessários das camadas internas para realizar os testes da maneira adequada. Isso também garante que os testes possam ser feitos de forma independente e paralelizada. Para softwares grandes, onde o processo de construção (do inglês, *build*) é lento e demorado, paralelizar os testes também é uma vantagem. Aí encontramos mais um benefício da Clean Architecture - a mesma promove escalabilidade, e funciona bem tanto para projetos pequenos como para grandes softwares.

Ainda na Figura 1, notamos que há quatro camadas exemplificadas - uma vez que há quatro círculos. A Clean Architecture não fixa um número específico de camadas - uma vez que cada software tem suas próprias especificidades. No caso do Mapa Afetivo, foram utilizadas três camadas - como veremos nas subseções posteriores.

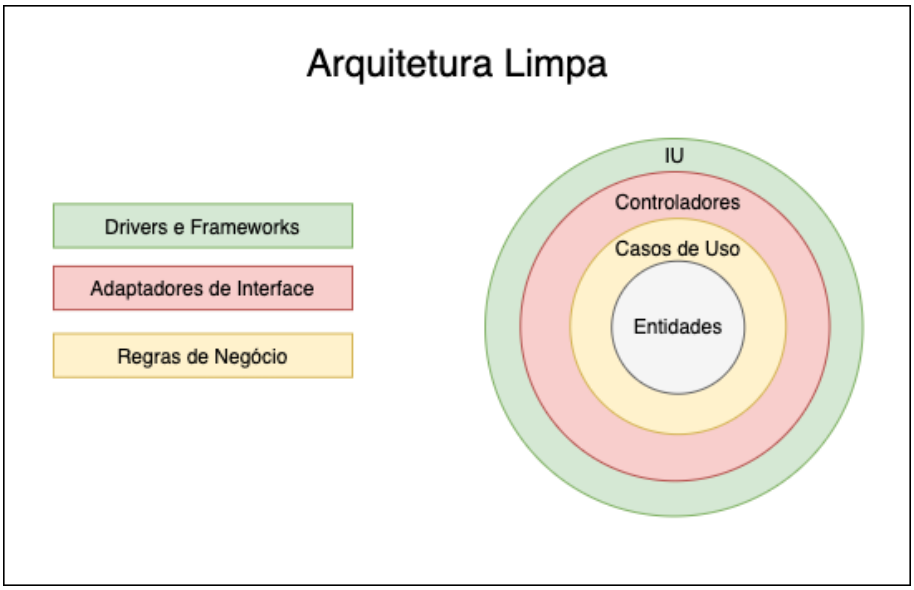

Figura 1: Clean Architecture - uma arquitetura em camadas. Adaptado de [2].

2.2 Arquitetura MVP

A segunda principal influência utilizada para a arquitetura do Mapa Afetivo foi a arquitetura MVP. MVP é um acrônimo para Model-View-Presenter - sendo estes os três principais componentes da arquitetura.

A arquitetura MVP surgiu originalmente como uma variação da arquitetura MVC (Model-View-Controller) [4] [5], e ganhou bastante popularidade nas últimas décadas, especialmente com o aumento de aplicações móveis desenvolvidas, tornando-se muito comum no desenvolvimento de aplicações Android.

2.2.1 Model

O *Model* é um componente responsável pelo gerenciamento dos dados e implementação de regras de negócio da aplicação. Nesta arquitetura, todas as estruturas de dados e também as classes e entidades que acessam fontes de dados - como repositórios, por exemplo, são considerados parte desse componente. Comumente, o *Model* é também referenciado como sendo a camada de dados da aplicação.

2.2.2 View

Nesta arquitetura, a *View* representa a interface de interação do usuário com a aplicação. Usualmente, em aplicações Android, as *Views* são tratadas como interfaces. Tais interfaces são implementadas por classes que herdam de outras classes do Sistema Android que tratam das interações do usuário com a aplicação - como Activities e Fragments.

Um dos principais pontos da Arquitetura MVP é a ideia de que as *Views* não devem

tomar ações e/ou decisões. As *Views* devem ser estritamente pontos de interação do usuário com a aplicação - e portanto não devem possuir nenhum tipo de lógica de negócios implementada.

2.2.3 Presenter

Na arquitetura MVP, o *Presenter* é o responsável por fazer a ligação entre a *View* e o *Model*. No *presenter*, podem ser tomadas algumas decisões, deve ser realizado o controle da *view* e também pode ser realizado o tratamento de dados.

Em aplicações Android, é comum que a *View* repasse todos os eventos de interação do usuário diretamente para o *Presenter*, que é o responsável por, de fato, tratar os eventos e realizar as ações necessárias para atender às requisições desejadas - fazendo uso, para isso, da camada *Model*. Ao realizar o seu trabalho, o *Presenter* deve invocar a *View* para exibir ao usuário o resultado da sua requisição.

Nesta estrutura, é comum que o *Presenter* sempre tenha uma referência direta para a *View* - pois precisa da mesma para exibir os resultados das requisições realizadas pelo usuário.

Na Figura 2 podemos observar uma ilustração de como a Arquitetura MVP é usualmente utilizada numa aplicação Android.

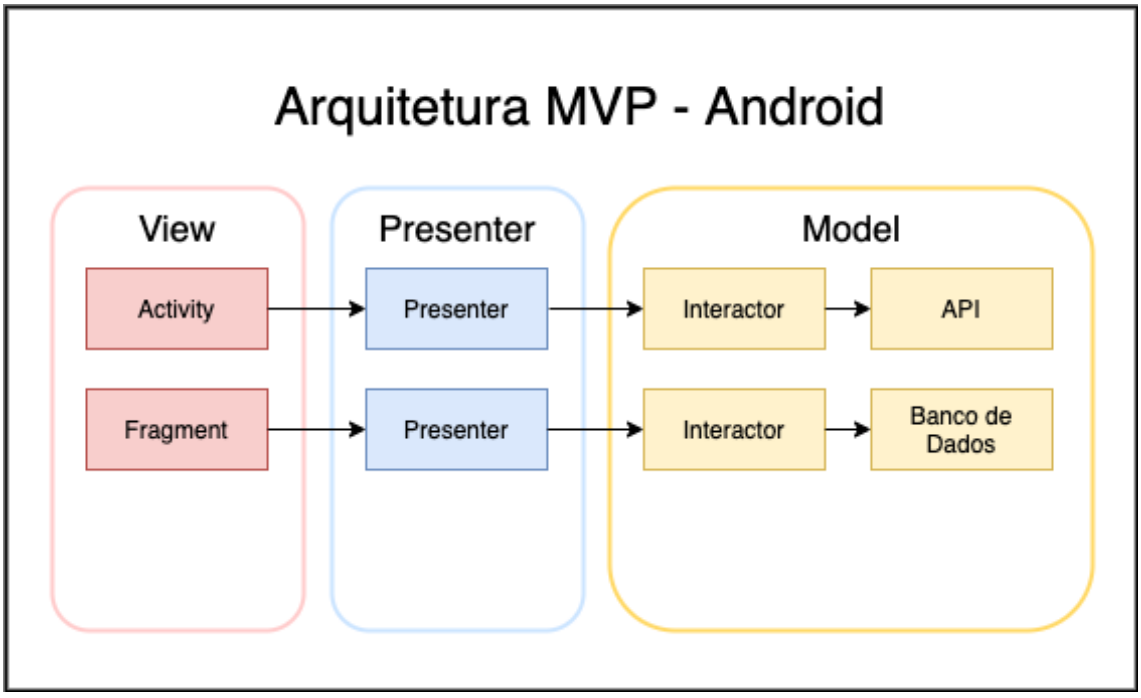

Figura 2: Uso comum da Arquitetura MVP em uma aplicação Android. Adaptado de [6].

3 Método de Trabalho

Antes de iniciar o desenvolvimento do projeto propriamente dito, foi utilizada a plataforma OpenDesign [11] para clarificar o problema a ser resolvido pelo aplicativo e realizar um levantamento inicial de requisitos da aplicação, realizando ainda a priorização dos mesmos. Nessa plataforma, foram identificados stakeholders importantes, problemas e possíveis soluções, além dos requisitos propriamente ditos. O OpenDesign foi útil nesta fase inicial e, a partir dos itens registrados na ferramenta, foi possível utilizá-los no GitLab como issues (itens a serem desenvolvidos) para o desenvolvimento do aplicativo. Isso é realizado de forma automatizada, uma vez que a plataforma OpenDesign é integrada ao Gitlab.

Após esta fase, foi realizado o *setup* inicial do projeto, com a criação das classes base da Arquitetura, junto à estrutura de entidades, camadas e pacotes definidos - de forma a permitir que fosse possível dar sequência ao desenvolvimento do aplicativo.

Sendo uma aplicação Android, o Mapa Afetivo foi desenvolvido utilizando a IDE padrão de desenvolvimento Android, o Android Studio - uma ferramenta da Google desenvolvida com base no IntelliJ.

O aplicativo foi desenvolvido utilizando a linguagem Kotlin [16]. Por ser uma linguagem mais nova, Kotlin possui diversas funcionalidades que facilitam e agilizam o processo de desenvolvimento [26]. Tais funcionalidades podem trazer benefícios como a criação de um código mais limpo - o que, por sua vez, abre possibilidades para economia de tempo de desenvolvimento e manutenção.

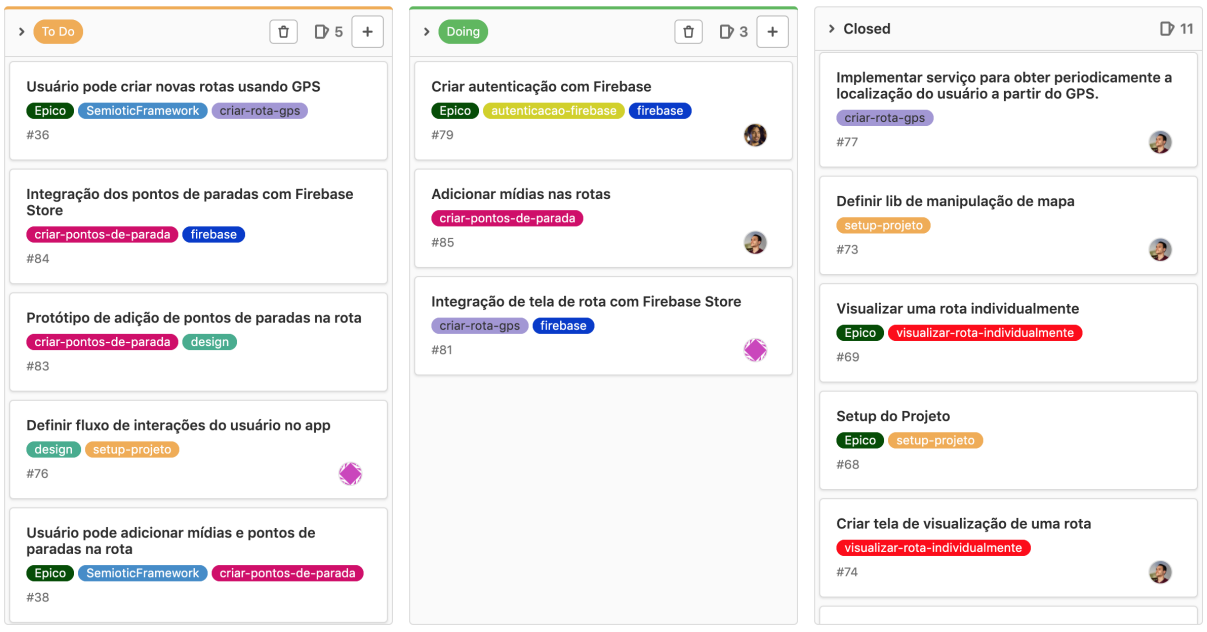

Figura 3: Atividades A Fazer, Em Progresso e Concluídas em um determinado estado do desenvolvimento do Mapa Afetivo

Para gerenciamento de configuração, foi utilizado o sistema de controle de versão Git

[7] junto ao serviço GitLab [8] - ferramentas comuns para o gerenciamento do código-fonte de projetos de software. Trabalhamos em diferentes branches do repositório, de forma a poder paralelizar o desenvolvimento, mantendo uma branch "master" como sendo a padrão da aplicação, para o código integrado e estável. Ao realizar uma tarefa, realizamos a integração do código desenvolvido para a mesma com a branch master, de forma a unificar todo o trabalho desenvolvido. O Gitlab também foi utilizado para gerenciar o progresso do desenvolvimento, utilizando a prática de backlog do produto, onde as tarefas a serem desenvolvidas e seus respectivos estados (A Fazer / Em Progresso / Concluído) são disponibilizados em um quadro Kanban, conforme a Figura 3. O código fonte [9] do projeto, disponível no GitLab, pode ser encontrado no endereço `https://gitlab.ic.unicamp.br/ra164468/mapa-afetivo`.

A ideia de utilizar a arquitetura em questão surgiu a partir de conhecimentos teóricos [10] e práticos adquiridos a partir do trabalho realizado e observado em projetos anteriores que fazem uso do estilo arquitetural MVP e da Arquitetura Limpa. A proposta de utilizar, então, uma arquitetura híbrida baseada na Arquitetura Limpa com conceitos provindos do estilo MVP foi realizada para o grupo de desenvolvimento, que acolheu a ideia. Ao longo do desenvolvimento, reuniões de alinhamento foram realizadas com os desenvolvedores do projeto, para que a arquitetura ficasse clara e pudesse ser respeitada por todos.

4 A Arquitetura do Mapa Afetivo

4.1 Uma Arquitetura Híbrida

Embora existam inúmeros benefícios pretendidos, o estilo arquitetural MVP pode ser insuficiente para, por si só, garantir uma estrutura e um padrão de código a ser seguido em projetos de médio e grande porte.

É fácil definir os componentes *View* e *Presenter* nessa arquitetura, mas o componente *Model* - que tende a ocupar uma grande parte da base de código acaba por não ter uma estrutura interna bem-definida. Por este motivo, torna-se relevante a definição de uma estrutura mínima também para este componente, de forma a garantir que o código escrito por todo o projeto siga um padrão bem-definido, mantendo as propriedades de manutenibilidade, fácil leitura e organização.

Com base neste cenário, uma arquitetura híbrida pode ser uma solução interessante - escolhida para o Mapa Afetivo. O aplicativo carrega consigo um misto entre as duas arquiteturas previamente mencionadas - MVP e Clean Architecture.

4.2 Três camadas

A arquitetura principal do Mapa Afetivo é dividida em três camadas, conforme pode ser observado na Figura 4: **data**, **domain** e **presentation**.

A ideia principal foi fazer com que tais camadas estejam alinhadas aos princípios da Arquitetura Limpa, respeitando a Regra das Dependências: a camada **presentation** tem dependências exclusivamente da camada **domain**. Por sua vez, a camada **domain** tem dependências exclusivamente da camada **data**.

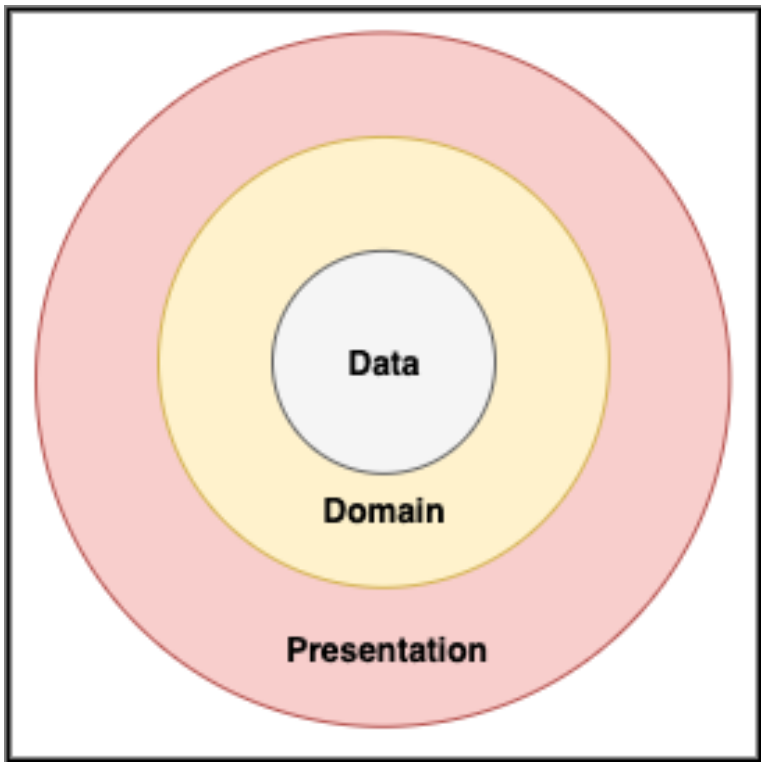

Figura 4: As três camadas da arquitetura do Mapa Afetivo

No Mapa Afetivo, a divisão do aplicativo a partir das camadas foi feita utilizando o recurso de **pacotes** das linguagens Kotlin e Java. A partir da raiz da estrutura do código-fonte, existem três pastas - *data*, *domain* e *presentation*, que contém respectivamente o código relativo às suas partes da arquitetura, como pode ser visualizado na Figura 5.

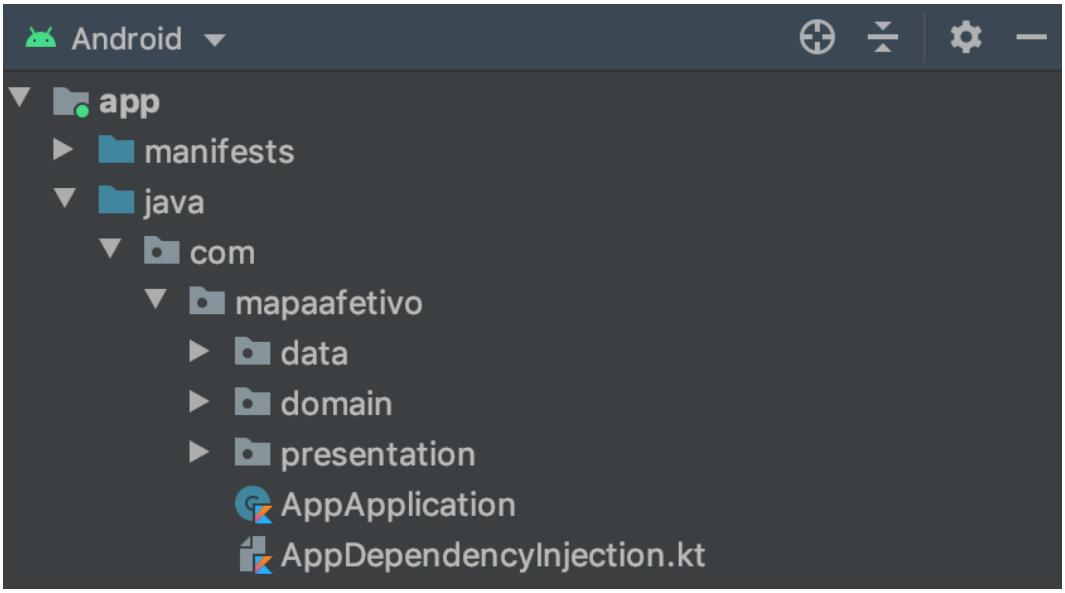

Figura 5: As três camadas divididas em pacotes a partir da interface do Android Studio

4.2.1 Presentation

Na camada *presentation*, estão contidos dois dos três componentes da Arquitetura MVP: a *View* e o *Presenter*. A ideia desta principal desta camada é abranger toda a lógica presente na interação do usuário com o aplicativo.

As principais classes de interação com componentes do Sistema Android também estão contidos nesta camada - como as *Activities*, *Fragments* e *Adapters* do aplicativo.

Dentro do pacote desta camada, há um subpacote para cada funcionalidade do aplicativo (Figura 6). E, dentro do pacote de cada funcionalidade, há ainda dois subpacotes: um pacote nomeado *view* e um pacote nomeado *presenter*.

Cada *view* representa uma interface de interação do usuário com a aplicação. Através da *view*, o usuário pode interagir com aplicativo, adicionando dados ou mesmo visualizando. Para cada *view* disponível no aplicativo, existe um *presenter* associado à mesma. A ideia aqui é que o *presenter* seja responsável por controlar a *view* e fazer as interações necessárias com a camada *domain*. Há uma exceção para a criação de *presenters*: quando a *view* pode ser considerada trivial. Nestes casos, há pouca ou nenhuma lógica envolvida no controle da *view*, o *presenter* pode ser omitido.

Views são criadas como interfaces na linguagem Kotlin. Estas interfaces são, por sua vez, implementadas por classes que estendem classes padrão do sistema Android - como Ac-

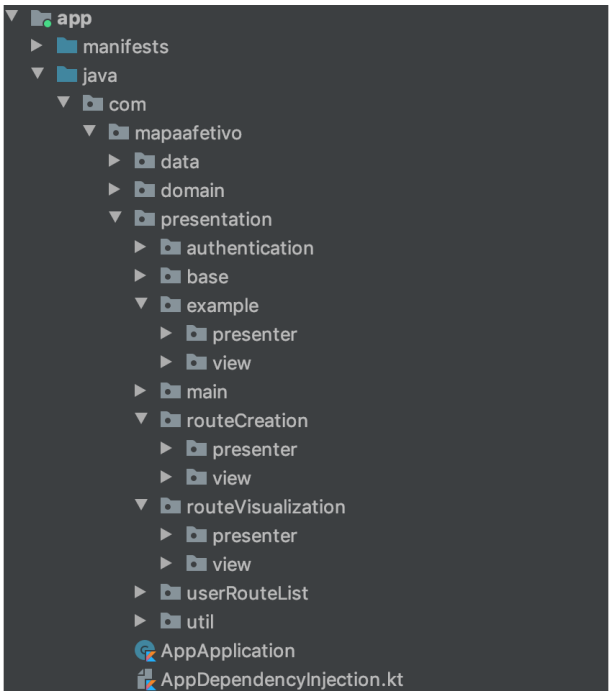

Figura 6: Estrutura de pacotes dentro da camada *presentation*.

tivities e Fragments. No Mapa Afetivo, interfaces gráficas individuais são sempre *Activities*. As Activities, por sua vez, são *views* que têm um *presenter* associado. Os *presenters* são criados como classes, que recebem em seu construtor uma instância da *view* ao qual devem se associar.

Pela forma como o Sistema Android funciona, *Activities* e *Fragments* possuem seu próprio ciclo de vida - que é controlado pelo Sistema de acordo com as interações realizadas pelo usuário com a aplicação. Os *presenters*, por serem criados dentro das *views* e para controlá-las, estão portanto intrinsecamente atrelados aos seus ciclos de vida.

Como os eventos de interação são repassados pelo sistema às *views*, é responsabilidade do desenvolvedor repassar tais eventos aos *presenters* - uma vez que o *presenter* deve controlar a *view*, e não o contrário. Desta forma, para todo evento de interação do usuário com uma *view* e para todo evento disparado pelo Sistema para uma *view* que é utilizado pelo Mapa Afetivo, há um método equivalente no *Presenter* da respectiva *view*, que deve ser invocado pela *view* no momento em que o evento é recebido. Ao ter um método invocado, o *Presenter* é o responsável por realizar a operação desejada e, após a finalização da operação, invocar a *view* (através de sua instância, recebida no construtor) para informar o resultado da operação. Nesse cenário, um *presenter* pode invocar métodos da *view* diversas vezes caso necessário, para alterar seu estado. Uma vez que a *view* também representa aquilo que é "visto" pelo usuário da aplicação, é natural que seja necessário alterar o estado da mesma para refletir o atual estado do aplicativo após o usuário solicitar a realização de uma ação.

Considere o seguinte cenário como exemplo desse comportamento: ao iniciar a criação

de uma rota, o aplicativo começa imediatamente a coletar a localização do usuário, disparando um serviço em *background* que coleta a localização conforme o usuário se move. No momento do início da criação, a *view* recebe um evento do sistema, indicando que o botão "criar rota" foi acionado. Imediatamente, a *view* invoca um método do *presenter* - `onCreateRouteClick(Context)` - para que a captura de pontos se inicie. Neste momento, o *presenter* imediatamente invoca o método `displayCreatingRouteLayout()` da *view*, informando a mesma que a sua exibição deve ser alterada para exibir que uma criação de rota está em progresso. Na implementação deste método, a *view* altera o seu layout para ocultar o botão "criar rota" e exibir, em seu lugar, um círculo de *loading* indicando que a criação de uma rota está em progresso. A Figura 7 ilustra a situação:

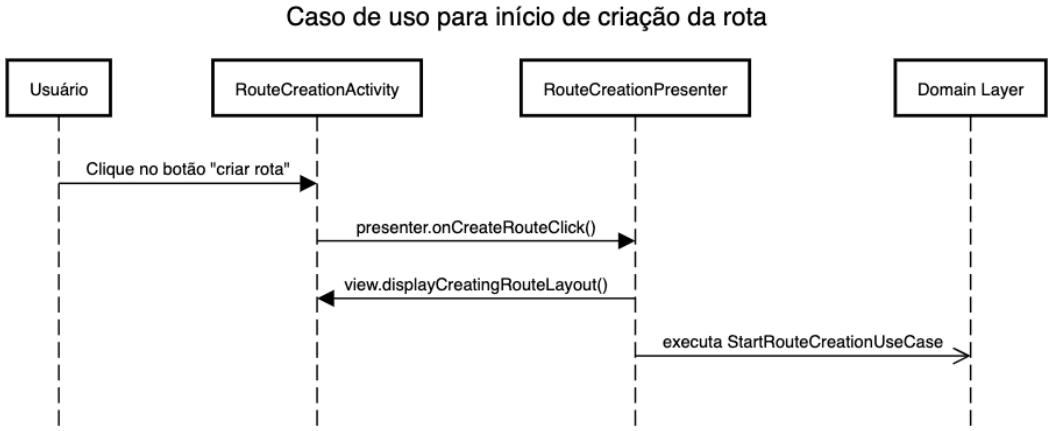

Figura 7: Sequência de interação entre a *view (Activity)* e o *presenter* para a criação de uma rota

Assim como na Arquitetura MVP, um princípio fundamental é mantido aqui: a *view* deve ser passiva - isto é, não deve "tomar iniciativas" por si própria ou mesmo ser responsável por nenhuma lógica. Toda a lógica da aplicação deve estar contida no *presenter* e nas camadas mais profundas, sendo a *view* responsável exclusivamente pela interação com o usuário - recebendo eventos e exibindo resultados.

4.2.2 Domain

As camadas *domain* e *data* formam o que seria o *Model* na Arquitetura MVP. Na camada *domain*, encontramos representada toda a lógica de negócio da aplicação. Esta lógica é primariamente representada através dos casos de uso, que são invocados pelo *presenter* da camada *presentation*. Por sua vez, os casos de uso fazem uso da camada *data* como para obter dados e também persistir. Uma outra forma de enxergar esta camada é tratá-la como um *middleware* entre a *view* e as fontes de dados.

Nesta camada, a ideia é que não haja mais interações com classes do Sistema Android. Nem sempre é possível manter esta restrição - quando, por exemplo, é necessária uma instância de um *Context* na camada *data*, para interação com alguma biblioteca ou módulo externo, é necessário que o mesmo seja passado pela camada *domain* - isto pode ser visto no

fluxo de criação de rotas e nos casos de uso relacionados. A principal vantagem de isolar os componentes do sistema Android na camada *presentation* é facilitar o uso de testes unitários para testar casos de uso, a camada com um todo e também a camada *data*. Quando não há componentes do sistema envolvidos, os testes podem ser realizados com a JVM, sem a necessidade de emular um dispositivo com sistema Android ou utilizar um dispositivo real para a execução dos testes. Em geral, isso poupa tempo dos responsáveis pelos testes - o que facilita os aplicativos que seguem tal arquitetura a serem escaláveis.

A estrutura de pacotes da camada *domain* é apresentada na Figura 8. Esta estrutura é simples uma vez que há um subpacote "usecase", que é dividido em subpacotes separados de acordo com a funcionalidade da aplicação - assim como acontecia na camada *presentation*. Há também um pacote *model*, que agrupa algumas entidades de uso comum da aplicação. Tais entidades tem como funcionalidade estritamente representar dados de forma estruturada, e não tem lógica de negócio envolvida. Por fim, há um pacote *base*, que agrega a classe base para a criação de um caso de uso.

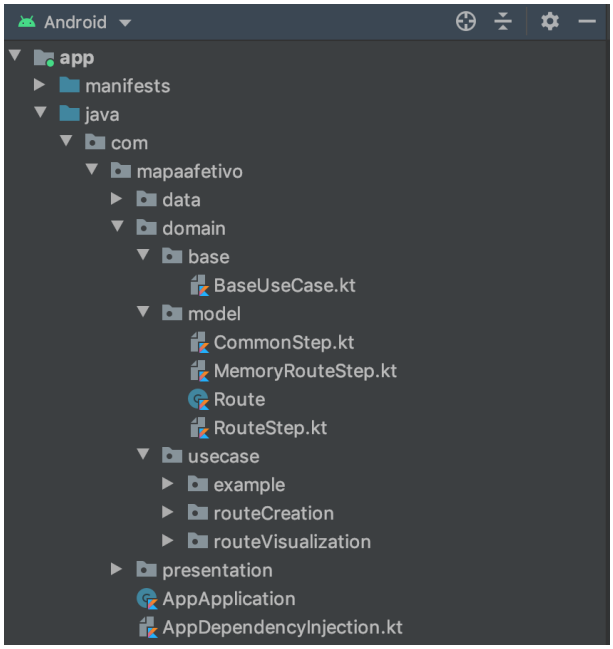

Figura 8: Ilustração do fluxo de interação entre a *view* e o *presenter* para iniciar a criação de uma rota

De fato, é notável que os casos de uso são a parte mais relevante desta parte da arquitetura. Não por menos, são de extrema importância, pois contém em si a lógica de negócio da aplicação. Cada caso de uso é representado por uma classe, cujo nome é sufixado pelo termo *UseCase*. O design dos casos de uso é realizado de tal forma que todo caso de uso deve, obrigatoriamente, estender a classe pai *BaseUseCase*. Os nomes das classes que representam classes de uso devem ser bem descritivos, e indicar exatamente qual a função do caso de uso em questão. Alguns exemplos de nomes de casos de uso encontrados no Mapa Afetivo são: *StartRouteCreationUseCase*, *FinishRouteCreationUseCase*, *AddImage-*

MediaToRouteUseCase.

Cada caso de uso também deve obrigatoriamente conter um método público cujo nome é *execute*. A quantidade e os tipos de parâmetros deste método são variáveis, específicos de cada caso de uso. O comportamento geral deste método, no entanto, deve ser uniforme em todos os casos de uso - ele deve executar a ação para a qual o caso de uso foi executado.

Exemplificando, para o caso de uso *AddImageMediaToRouteUseCase*, o método *execute()* deve adicionar uma imagem à rota que está sendo criada no momento da invocação. Para isso, a assinatura deste método é declarada da seguinte forma:

```
fun execute(
    title: String,
    description: String,
    imageMedia: File
): Completable
```

A assinatura do método para este caso de uso indica que ele possui três parâmetros: um título - que deverá ser associado à mídia - uma descrição - que também deverá ser associada à mídia - e por fim o arquivo de mídia em si.

No exemplo anterior, pode-se observar que o método *execute* tem um tipo de retorno chamado *Completable*. Esta é mais uma característica da arquitetura implantada: todo método *execute()* de um *UseCase* deve obrigatoriamente ter um dos três tipos de retorno a seguir: *Observable, Single* ou *Completable*. Tais classes são parte da biblioteca RxJava 2 [13], que é amplamente utilizada nesta arquitetura.

A biblioteca RxJava facilita o uso de programação reativa e a realização de tarefas assíncronas [14] nas linguagens Java e Kotlin, além de adicionar funções de uso geral que podem ser úteis para diversas situações. No caso do Mapa Afetivo, ela é utilizada em todos os UseCases, uma vez que o tipo de retorno de um UseCase deve ser obrigatoriamente um dos três tipos mencionados. As operações assíncronas costumam ser executadas com alta frequência em aplicações Android - mas no entanto, o sistema não possui por padrão nenhuma forma trivial de implementar tais operações.

Um dos principais conceitos existentes na programação de aplicações Android é o da *main thread* - ou *UI Thread* - a thread de execução principal de um aplicativo. Por padrão, as operações de um aplicativo são sempre executadas nesta *thread*. No entanto, determinadas operações possuem um tempo de execução longo - como uma requisição HTTP - e, portanto, devem ser executadas em uma *thread* em background para que a execução do aplicativo não fique travada. Uma das funcionalidades mais úteis da biblioteca RxJava é possibilitar facilmente a troca de threads para determinadas operações, através do uso de *Schedulers* [23] [24].

Podemos tratar *Observables*, *Singles* e *Completables* como promessas. Imaginemos que tais objetos contém a **promessa** da execução de um determinado trecho de código. Com isso, em todo objeto que seja de um desses tipos, podemos invocar o método *subscribe()* - que fará com que a promessa seja executada e o código prometido seja, de fato, executado. Desta forma, para executar um caso de uso, não basta invocar o método *execute()* - é necessário invocar, em seguida, o método *subscribe()* para que a execução seja realizada.

A interação com os *Schedulers* é realizada como o padrão de projeto *Observer*[28]. O método *subscribeOn(Scheduler)* permite escolher em qual thread a promessa será, de fato, executada - o parâmetro Scheduler representa a thread de execução. O método *observeOn(Scheduler)* permite escolher em qual thread o resultado da execução será entregue.

Além da biblioteca RxJava2, fazemos uso também da biblioteca RxAndroid, que possui um Scheduler específico para representar a *main thread* do sistema Android. Assim, podemos facilmente configurar a execução de um UseCase para que seja realizada em uma *thread* em background, tendo seu resultado entregue na *main thread* - o que é extremamente útil.

O trecho de código abaixo mostra como um caso de uso pode ser executado, de forma que a execução seja feita em uma thread em background e o resultado seja entregue na thread main:

```
fun onImageFetched(title: String, description: String, image: File) {
    addImageMediaToRouteUseCase
        .execute(title, description, image)
        .setupCommonSchedulers()
        .subscribe()
}

   ...

fun Completable.setupCommonSchedulers(): Completable {
    return this
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
}
```

Podemos observar a execução do *UseCase AddImageMediaToRouteUseCase*, mencionado anteriormente. Antes de invocar o método *subscribe* e efetivar a execução, é invocado o método *setupCommonSchedulers()*. Conforme pode ser visto no segundo trecho de código, este método nada mais é do que uma *Extension Function* [17] criada para configurar corretamente a execução em background utilizando o *Scheduler* de entrada e saída, e configurar o retorno para ser executado na *main thread*.

Um caso de uso deve ser sempre bem definido e ter um propósito único: realizar uma ação específica. A ação realizada deve estar bem descrita no nome da classe do caso de uso, e a mesma deve ser executada a partir da invocação do método *execute()* do caso de uso. Para realizar o seu trabalho, os casos de uso utilizam a camada *data* da aplicação, acessando os repositórios e fontes de dados da aplicação para obter informações, e também para persisti-las.

Frequentemente, veremos classes de casos de uso que terão um tamanho pequeno - estas representam casos de uso sem muita lógica de negócio envolvida. Nesses casos, a classe acaba por funcionar como um *proxy* entre a camada *data* e a camada *presentation*. Outras vezes, porém, os casos de uso podem ser grandes, a depender de quantos repositórios da camada de *data* precisam ser acessados para que o caso de uso tenha sua ação executada. Os casos de uso também podem atuar como *mappers*, cuja função é mapear dados da camada

de *data* para o formato esperado pela *view*. Como trabalhamos com *views* passivas, este mapeamento deve ser feito pelo próprio caso de uso ou pelo *presenter*, conforme fizer mais sentido.

Na Figura 9, podemos ver um exemplo de interação entre um caso de uso, a camada *data* e a camada *presentation*. *RouteCreationActivity* e *RouteCreationPresenter* fazem parte da camada *presentation*. *RouteRepositoy* faz parte da camada *data*.

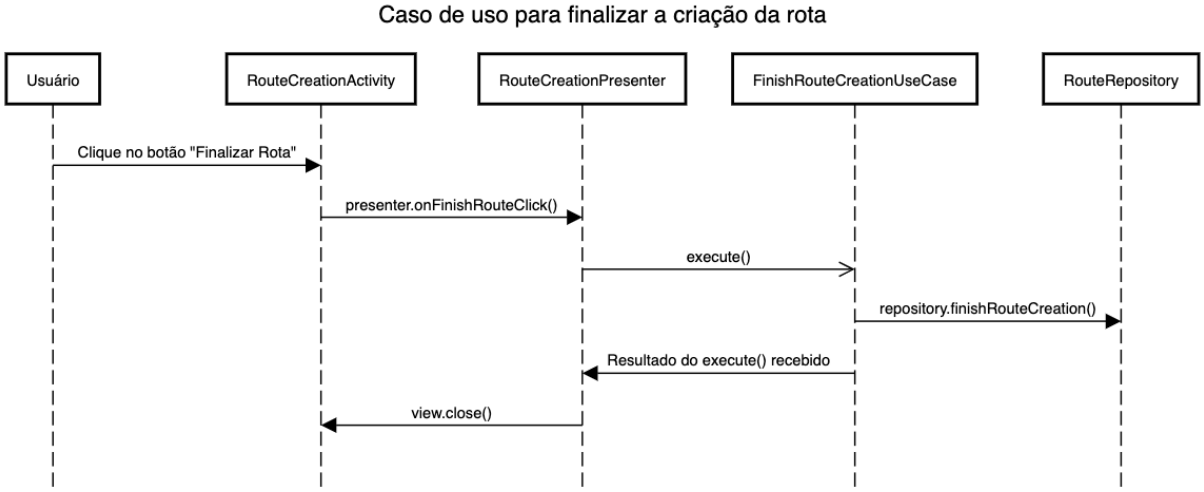

Figura 9: Ilustração do fluxo de interação entre um *UseCase* e as camadas *data* e *presentation*.

4.2.3 Data

Com a camada *data*, fechamos o que seria o restante do *Model* numa arquitetura MVP tradicional. Esta camada é responsável por realizar todas interações da aplicação com o meio externo - o que inclui interações com o *Firebase [21] Firestore [22]* (serviço utilizado para persistência de dados) e interações com o serviço de coleta de pontos.

Esta camada foi desenhada para atuar com a estrutura de repositórios. Nesta estrutura, temos diversas interfaces cujo nome é sufixado com a palavra *Repository*. Tais interfaces são implementadas por classes que funcionam como entrada e saída de dados. Métodos são declarados nas interfaces *Repository* para executar as funções necessárias.

Os repositórios criados na camada *data* são utilizados pelos casos de uso da camada *domain*. No método construtor de um *usecase*, este deve receber como parâmetros todos os repositórios necessários para executar a sua ação.

Diferentemente dos *usecases*, que tem uma a função de executar uma ação específica cada um, os repositórios - por atuarem como fonte de dados são mais gerais. Um mesmo repositório pode ter diversas funções e, portanto, pode ser reutilizado por diferentes casos de uso. Também por isso, os repositórios da aplicação são tratados como objetos *singleton* - o que será visto em mais detalhes na Seção 4.4.

No trecho de código abaixo, podemos ver um exemplo da interface do repositório de rotas da caminhada - *RouteRepository*. Este repositório é responsável pelas funções relacionadas à criação de uma rota:

```
interface RouteRepository {

    fun startRouteCreation(context: Context): Completable

    fun isRouteCreationInProgress(): Single<Boolean>

    fun finishRouteCreation(): Single<List<MapaAfetivoLocation>>

    fun addTextMedia(title: String, text: String): Completable

    fun addImageMedia(title: String, description: String,
                      imageMedia: File): Completable

}
```

Ao observar este repositório, podemos observar que o tipo de retorno segue um padrão. Assim como nos casos de uso, os repositórios também devem retornar exclusivamente objetos dos seguintes tipos: *Single, Completable* ou *Observable*. As razões para esta restrição são similares às já explicadas anteriormente em relação aos casos de uso - facilidade de execução do código em diferentes threads conforme a necessidade. Como os repositórios são usados diretamente pelos casos de uso, há mais uma vantagem em manter os retornos limitados a objetos destes três tipos: é fácil para um caso de uso atuar como um *mapper* entre a camada de dados e a camada de apresentação, uma vez que os tipos de retorno são limitados. O caso de uso deve alterar apenas alterar o tipo interno retornado (o que não é necessário no caso de um retorno *Completable*, já que não há tipo interno) - e isto é algo trivial de realizar utilizando algumas das facilidades da biblioteca RxJava, fazendo uso das funções *map* e *flatMap*, disponíveis para *Observables* e *Singles*.

4.3 Classes base

Além da divisão estruturada dos pacotes anteriormente explicitada, algumas classes base foram também desenvolvidas - de forma a garantir que, especialmente as classes da camada *presentation* do aplicativo, seguissem um padrão.

Uma destas classes base é a *BasePresenter*, que é a base de todos os *presenters* da aplicação - todo *presenter* do Mapa Afetivo deve obrigatoriamente herdar desta classe.

Sempre que um *UseCase* é executado, é retornado um objeto do tipo *Disposable* [25]. Esta é uma classe padrão do RxJava, que representa uma ligação entre o *Observable, Completable* ou *Single* observado e o observador que o executou - a partir da invocação do método *subscribe()*. Após a execução de um caso de uso, é necessário invocar o método *dispose()* do *Disposable* associado, de forma a garantir que a ligação seja desfeita - uma vez que não é mais necessária, e mantê-la ativa pode gerar vazamentos de memória.

Um dos objetivos da classe *BasePresenter* é garantir que as ligações *Disposable* sempre sejam propriamente encerradas - uma vez que os casos de uso sempre são invocados a partir de *presenters*, é natural que todo *presenter* seja responsável por encerrar suas próprias ligações.

```
open class BasePresenter {

    private val pendingDisposable: CompositeDisposable
            = CompositeDisposable()

    protected fun addDisposable(disposable: Disposable) {
        pendingDisposable.add(disposable)
    }

    fun onDestroy() {
        pendingDisposable.clear()
    }

}
```

Como podemos observar no código da classe, exibido acima, ela garante que todo *presenter* possua um método *onDestroy()*, responsável por encerrar os *Disposables* ativos. O método *addDisposable* existe para que, sempre que um caso de uso seja executado, os *presenters* possam manter as referências das ligações na propriedade *pendingDisposable* - de forma que todas sejam encerradas futuramente, quando o método *onDestroy()* for invocado.

O método *onDestroy()* sempre é invocado por uma *view*, a partir de um evento disparado pelo Sistema Android informando de sua destruição. Naturalmente, a *view* repassa a informação da destruição para o *presenter* - através do *onDestroy()*, que é responsável por encerrar os *disposables* existentes e, possivelmente, outras atividades pendentes.

Além do *BasePresenter*, há também duas outras classes base importantes: *BaseFragment* e *BaseAppCompatActivity*. Ambas as classes desempenham funções semelhantes - são, respectivamente, as classes base de *Fragments* e *Activities*, os dois únicos tipos de *views* presentes na arquitetura do Mapa Afetivo.

Como dito anteriormente, uma das funções destas classes é receber o evento de destruição da *view* do Sistema Android e repassá-la para *presenter* associado. Naturalmente, pode-se perceber uma função implícita de *BaseAppCompatActivity* e *BaseFragment*: realizar a ligação entre o *presenter* e a *view*.

```
abstract class BaseFragment<T : BasePresenter> : Fragment(), KodeinAware {

    private val nonGenericsPresenter by instance<BasePresenter>()
    protected val presenter by lazy { nonGenericsPresenter as T }

    override val kodein: Kodein = Kodein.lazy {
```

```
        extend(AppApplication.kodein)
        import(fragmentModule())
    }

    protected open fun fragmentModule(): Kodein.Module = Kodein.Module(
    "General Fragment Module"
    ) {

    }

    override fun onDestroy() {
        super.onDestroy()
        presenter.onDestroy()
    }

}
```

No trecho acima, podemos ver a implementação da classe *BaseFragment*. Assim como *BaseAppCompatActivity*, ela é uma classe abstrata que recebe um tipo T, que obrigatoriamente estende um *BasePresenter*. Isso força com que as classes herdeiras obrigatoriamente tenham um *presenter*, do tipo T - onde T é um tipo genérico que representa o *presenter* da *view* específica descrita pela classe concreta. Notamos que a classe disponibiliza para suas herdeiras uma referência para o *presenter*, a partir da propriedade de mesmo nome. Esta é a forma fundamental que uma *view* tem para acessar seu *presenter*: através desta propriedade.

4.4 Injeção de Dependências

As classes base *BaseFragment* e *BaseAppCompatActivity* também desempenham outra função importante: são responsáveis por integrar parte do código responsável pela injeção de dependências do projeto.

No Mapa Afetivo, a biblioteca Kodein [15] foi utilizada para realizar a injeção de dependências - uma técnica amplamente utilizada na arquitetura para ajudar a manter o código limpo e ajudar na criação de testes unitários, caso isto venha a ser executado no futuro.

O Kodein funciona de maneira simples: cria-se uma instância da classe Kodein no *Application* [18] da aplicação, e em seguida são adicionados módulos a esta instância. Os módulos são responsáveis por **descrever como deve ocorrer a instanciação de classes** através do *framework* de injeção de dependências. No Mapa Afetivo, foram criados módulos de escopo global específicos para a criação de repositórios e casos de uso.

A criação de repositórios é realizada a partir do *repositoryModule*. Os repositórios da aplicação são tratados como *singletons* - criados uma vez, quando seu uso é necessário, permanecem "vivos" até que o ciclo de vida da aplicação seja encerrado. Como os repositórios podem ser usados em diferentes pontos da aplicação, por diferentes casos de uso, não há

necessidade de criar múltiplas instâncias de um mesmo repositório - e aqui a construção de um *singleton* faz sentido.

Os casos de uso são criados a partir do *useCaseModule* - que também é global. No entanto, os casos de uso não são tratados como *singletons* - como cada caso de uso é específico e existe para realizar uma ação, faz mais sentido que os mesmos sejam criados como instâncias comuns. Desta forma, o *useCaseModule* funciona como um agrupador de funções para criar casos de uso. No exemplo abaixo, podemos ver um trecho do *useCaseModule*, onde é especificada a criação do caso de uso *StartRouteCreationUseCase*:

```
val useCaseModule = Kodein.Module("Use Cases Module") {
    ...
    bind<StartRouteCreationUseCase>() with provider {
        StartRouteCreationUseCase(instance())
    }
    ...
}
```

O *StartRouteCreationUseCase* recebe, em seu construtor, uma instância de *RouteRepository*. A invocação da função *instance()* na criação do caso de uso indica ao *Kodein* que uma instância de *RouteRepository* deve ser criada para ser injetada no *StartRouteCreationUseCase*. O *Kodein* cria esta instância a partir de seus módulos - quando a função *intance()* é invocada, a biblioteca procura nos módulos associados à instância atual do *Kodein* uma função que possa instanciar um *RouteRepository*. Felizmente, esta função é encontrada no *repositoryModule* - um trecho do mesmo é exemplificado abaixo:

```
val repositoryModule = Kodein.Module("Repository Module") {
    bind<ExampleRepository>() with singleton { ExampleDataRepository() }
    bind<RouteRepository>() with singleton { AndroidRouteRepository() }
    bind<UserRepository>() with singleton { LocalUserRepository(
    instance("local_user_preferences"
    ), instance()) }
}
```

Como podemos ver, este módulo descreve como uma instância de *RouteRepository* pode ser criada.

4.4.1 AppApplication

A instância global do *Kodein* utilizada pela aplicação é configurada na classe *AppApplication* - uma classe que estende *Application* [18] - e é responsável por realizar configurações iniciais do aplicativo, toda vez que ele é inicializado pelo sistema. No trecho abaixo, ilustramos a criação de uma instância de *Kodein*, usando módulos previamente citados e também outros utilizados pela aplicação:

```
class AppApplication : Application(), KodeinAware {

    companion object {
        lateinit var kodein: Kodein
    }

    override val kodein: Kodein = Kodein.lazy {
        import(repositoryModule)
        import(useCaseModule)
        import(firebaseModule)
        import(generalModule(this@AppApplication))
    }

    override fun onCreate() {
        super.onCreate()
        Companion.kodein = kodein
    }

    ...

}
```

A instância de *Kodein* é criada a partir da invocação da função *Kodein.lazy*. Nesta função, associamos os módulos previamente descritos à instância em criação.

4.4.2 Injeção de dependências na criação de rotas

A *view* relativa à criação de rotas é uma *Activity*, cuja classe recebe o nome *RouteCreationActivity*. Esta *Activity* estende a classe *BaseAppCompatActivity*, utilizando o tipo T do *Presenter RouteCreationPresenter*, e também implementa a interface *RouteCreationView* - interface de comunicação do *presenter* com a *view*.

No trecho abaixo, podemos observar um trecho da classe *BaseAppCompatActivity*:

```
abstract class BaseAppCompatActivity<T : BasePresenter> :
    AppCompatActivity(), KodeinAware {

    private val nonGenericsPresenter by instance<BasePresenter>()
    protected val presenter by lazy { nonGenericsPresenter as T }

    private val applicationKodein by closestKodein()

    override val kodein: Kodein = Kodein.lazy {
        extend(applicationKodein)
        import(activityModule())
    }
```

```
    protected open fun activityModule(): Kodein.Module =
        Kodein.Module("General Activity Module") {
    }

    ...

}
```

Esta classe implementa a interface *KodeinAware*, que faz parte também da biblioteca *Kodein*. Isto indica que a classe em questão deve ter associada à mesma um objeto do tipo *Kodein*. O objeto em questão é declarado na mesma classe, e é importante destacar dois pontos.

Primeiro, a instância de *Kodein* criada nesta classe terá acesso a todos os módulos da instância de *Kodein* da classe *AppApplication* - e, portanto, será capaz de prover dependências criadas nestes módulos. Isto é feito a partir do comando *extend(applicationKodein)*.

O segundo ponto de destaque é a inserção do comando *import(activityModule())* - que faz com que a instância de *Kodein* desta classe também tenha acesso às dependências criadas pelo módulo retornado pela função *activityModule()*.

Pois bem, nota-se que esta função é declarada como *open*, e pode ser sobrescrita pelas classes herdeiras de *BaseAppCompatActivity*. A função deve, portanto, ser sobrescrita por cada uma destas classes **que necessitem de dependências não providas pela instância global de *Kodein* da aplicação**.

Observa-se no trecho a seguir uma pequena parte da *Activity RouteCreationActivity*:

```
class RouteCreationActivity :
    BaseAppCompatActivity<RouteCreationPresenter>(),
    RouteCreationView,
    OnMapReadyCallback {

    ...

    override fun activityModule(): Kodein.Module =
    Kodein.Module("Route Creation Module") {
        bind<BasePresenter>() with provider {
            RouteCreationPresenter(
                this@RouteCreationActivity,
                instance(),
                instance(),
                instance(),
                instance()
            )
        }
    }
```

```
    ...
}
```

É possível notar que há uma sobrescrita do método *activityModule()* da *BaseAppCompatActivity*. Nesta sobrescrita, é criado um novo módulo, que indica como a criação de instâncias da clase *RouteCreationPresenter* - o *presenter* desta *view* - deve se realizada. Nota-se novamente o uso da função *instance()*, para que o *Kodein* possa automaticamente procurar as dependências necessárias para os parâmetros do construtor do *RouteCreationPresenter* e criá-las - simplificando imensamente o trabalho do desenvolvedor, que outrora deveria manualmente instanciar as dependências neste trecho do código. De forma geral, além de poupar tempo de desenvolvimento, o uso do *Kodein* como *framework* de injeção de dependências ajuda massivamente a manter um código limpo, de fácil leitura, sem *boilerplate* e de fácil manutenção.

5 A Arquitetura Proposta em Uso

Nesta seção, trazemos como exemplo de uso da arquitetura a funcionalidade de Visualização de Rota. A funcionalidade é simples, e não chega a utilizar todos os elementos da arquitetura - está aqui apenas para ilustrar parte do trabalho também desenvolvido no projeto.

A *view* principal relativa a esta funcionalidade é a interface 'RouteVisualizationView', implementada pela *Activity RouteVisualizationActivity*. A *Activity* é instanciada pelo sistema Android quando o usuário deseja visualizar uma rota específica. Para identificar qual rota deve ser visualizada, a *Activity* deve receber, em seu *Intent* de criação, um *extra* nomeado "EXTRA_ROUTE", que deve conter uma instância de *Route*, indicando a rota do usuário. Para isso, é recomendado que a criação do *Intent* seja feita através do método *newInstance(Context, Route)*, pertencente ao *companion object* da classe *RouteVisualizationActivity*. Abaixo, podemos ver o código deste método:

```
    companion object {

        const val EXTRA_ROUTE = "EXTRA_ROUTE"

        fun newInstance(context: Context, route: Route): Intent {
            return Intent(context, RouteVisualizationActivity::class.java)
                .putExtra(EXTRA_ROUTE, route)
        }

    }
```

Quando a *Activity* é criada, o método *onCreate* da mesma é chamado, pelo Sistema Android. A *Activity* então, atuando como uma *view* passiva, repassa este evento ao método *onInitialize(Route)* do seu *presenter* (*RouteVisualizationPresenter*). O *presenter* então guarda

uma referência da rota recebida, e chama o método *initialize()* da *view*, para que esta possa executar seu processo de inicialização.

Durante sua inicialização, a *view* configura seu *layout* e também inicializa o mapa a ser exibido, fazendo o uso da biblioteca de mapas do Google para isso [19]. Quando a inicialização do mapa é concluída, a *view* avisa o *presenter* deste evento, invocando o método *onMapReady()* do mesmo. Quando este método é invocado, o *presenter* toma a decisão de exibir a rota para o usuário. Para isso, ele invoca o método *displayRoute(Route)* da *view*, utilizando como argumento a rota previamente guardada em uma referência. No método, a *view* exibe, visualmente, a rota para o usuário. Para realizar esta exibição, a arquitetura novamente é utilizada. Os pontos de início e término da rota, além dos pontos salvos manualmente pelo usuário como uma lembrança tem uma exibição especial. Para estes pontos, uma pequena descrição é exibida na parte inferior da tela - junto com uma imagem, nos casos de lembranças com imagens.

Para realizar esta exibição, foi criado um *ViewPager*, que utiliza instâncias de *Fragments* específicas - da classe *RouteVisualizationMemoryFragment*. Cada um destes *Fragments* é responsável por exibir ao usuário as informações de uma memória (ou dos pontos de início e fim da rota). Notavelmente, cada um destes *Fragments* é tratado como uma *View*. A classe *RouteVisualizationMemoryFragment* implementa a interface *RouteVisualizationMemoryView*, que está associada ao *presenter* *RouteVisualizationMemoryPresenter*, seguindo a mesma estrutura da arquitetura. Na Figura 10, podemos ver a estrutura de pacotes desta funcionalidade.

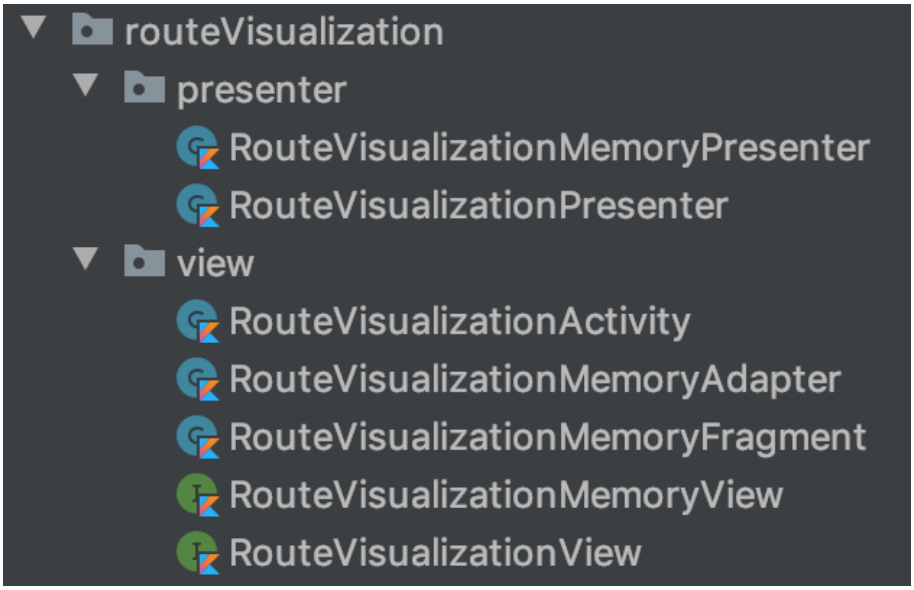

Figura 10: Estrutura de pacotes da funcionalidade de visualização da rota

Notamos também que, como nesta funcionalidade nenhum dado é persistido e todos os dados exibidos (a rota do usuário) são recebidos como parâmetros de entrada da *RouteVisualizationActivity*, não foi necessária a criação de nenhum *UseCase* para esta *feature*. No

entanto, foi criado o *UseCase FetchUserRouteListUseCase*, que é usado para obter a lista de rotas do usuário - este caso de uso é executado antes da visualização da rota acontecer.

A Figura 11 ilustra, de forma simplificada, a execução do fluxo.

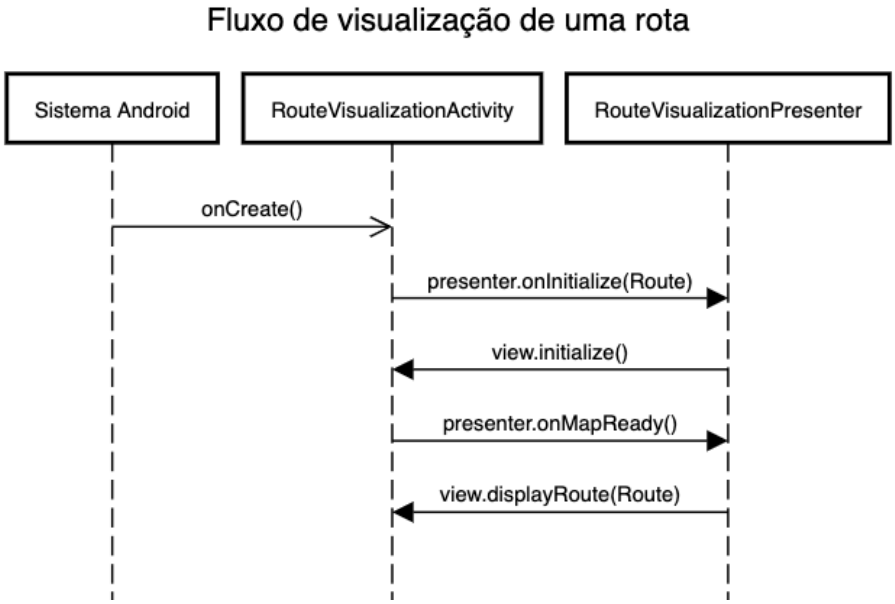

Figura 11: Ilustração do fluxo de visualização de uma rota.

6 A Arquitetura na Coleta de Pontos

Para a coleta de pontos - parte fundamental da funcionalidade de criar rotas, foi desenvolvido um módulo a parte. Este módulo é totalmente independente do Mapa Afetivo - sendo inclusive construído separadamente no processo de *build* do projeto. Desta forma, para que o Mapa Afetivo possa utilizá-lo, ele é importado como uma dependência que o aplicativo possui.

Por ser um módulo a parte, ele é tratado pelo Mapa Afetivo como algo externo ao aplicativo. Desta forma, a interação com este módulo pelo Mapa Afetivo é feita exclusivamente a partir da camada *data* da aplicação. O repositório *RouteRepository* - cujo código da interface foi exibido anteriormente - é responsável pela interação. As implementações de seus métodos se comunicam com o módulo externo e são responsáveis por garantir a criação da rota.

O fluxo de Coleta de Pontos inicia na *view* e passa pelo *presenter* até a execução do caso de uso *StartRouteCreationUseCase*. Este caso de uso é responsável por fazer uso da camada *data* - pelo *RouteRepository* - que, por sua vez, interage com o módulo externo para iniciar a Coleta de Pontos.

6.1 A criação de um módulo externo

Pela forma como os componentes do Sistema Android funcionam, um *Service* é adequado para coletar a localização do usuário em plano de fundo. Dadas as características de um *Service*, nota-se que o mesmo não se enquadra bem em nenhuma das três camadas da arquitetura. Este foi o principal motivo pela qual a criação de um módulo externo foi realizada. Tal módulo é responsável por realizar a coleta de pontos do usuário. Sendo um módulo externo à parte principal do Mapa Afetivo (embora interno ao aplicativo), ele segue a sua própria arquitetura.

A principal classe deste módulo é a *FetchUserLocationService*. Esta classe estende a classe *Service* do Sistema Android, e representa um serviço que roda em plano de fundo e é responsável por coletar a localização do usuário. O funcionamento desta classe é simples: a partir do momento em que o *Service* recebe o comando de inicialização, ele passa a coletar a localização atual do usuário, com um intervalo de um segundo entre cada requisição. Para coletar a localização, é utilizada a biblioteca de Localização do Google [20].

Ao iniciar a coleta de localização, o *Service* é colocado em primeiro plano, e uma notificação é exibida ao usuário - indicando que uma coleta está em andamento. A cada um segundo, então, a localização do usuário é obtida. O *Service* possui um mecanismo observável, criado a partir do uso de um *Observable* da biblioteca *RxJava*. Ao obter uma localização, o *Service* emite essa localização utilizando o *Observable*, para que possíveis clientes interessados em obtê-la possam fazer uso da mesma - o que não ocorre no Mapa Afetivo. Ao obter a localização do usuário, um segundo comportamento também é tomado pelo *Service*: é feito um cálculo para determinar se a localização obtida está dentro de um raio de cinco metros da localização mais recente salva. Caso esta seja a primeira localização obtida, ela é adicionada à lista de pontos percorridos do usuário. Caso não seja a primeira localização e esteja num raio de cinco metros da localização mais recente, esta localização é descartada. No último caso - em que a localização obtida está a mais de cinco metros da última localização não descartada, esta localização é adicionada à lista de pontos percorridos pelo usuário.

Quando o *Service* recebe o comando de finalização da rota - a partir de uma interação do usuário - a coleta de pontos é então interrompida, e a lista com todos os pontos coletados é retornada à camada *data* da arquitetura do aplicativo. O *Service* então encerra o seu funcionamento.

Caso, em algum momento durante a criação da rota, o usuário resolva adicionar alguma memória - em forma de texto ou imagem -, é utilizada uma função no *Service* para retornar a última localização disponível do usuário [20]. Esta localização é então obtida, e a mídia é associada ao ponto de localização retornado. Os pontos de mídia ficam associados ao *RouteRepository*.

Ao finalizar a criação da rota, os pontos de memória do usuário são concatenados à lista de pontos percorridos retornada anteriormente. É feita, então, uma ordenação de todos os pontos pelo horário de captura dos mesmos - de forma a garantir que os pontos de memória permaneçam na ordem correta. Os pontos são, então, repassados à camada *domain* da aplicação.

7 Conclusões

A arquitetura proposta não pôde ser seguida em 100%, quando observada a base de código do projeto. Uma das principais razões para este fator é a curva de aprendizado associada à arquitetura e às tecnologias do projeto, ligada ao fator tempo relacionado ao prazo disponível para finalização e entrega do projeto. Isto não é um problema, já que as partes fora da arquitetura podem ser adaptadas para fazerem parte da mesma caso isto faça sentido no futuro, sem prejuízos ao funcionamento da aplicação.

É importante salientar que a arquitetura desenvolvida possui oportunidades para melhorias, especialmente em pontos onde não foram adotadas restrições fortes. Um exemplo onde este ponto chama a atenção é em relação à localização das classes de modelos - cuja função única é armazenar dados em memória de forma estruturada - que hoje não possuem um lugar bem definido na arquitetura. Na implementação atual do aplicativo, algumas classes estão na camada *domain*, enquanto que outras - mais atreladas à persistência e obtenção dos dados estão na camada *data*. O ideal seria definir um local adequado para tais classes, com sentido lógico por trás desta decisão. Um outro ponto de melhorias está relacionado à organização dos repositórios da camada *data* - na implementação atual, diversos repositórios foram criados, mas a decisão entre quais métodos devem ficar em qual repositório não está semanticamente bem definida.

Por fim, é razoável acrescentar que o desenvolvimento de algumas funcionalidades por mim - como a Criação de Rotas e a Visualização de Rotas - seguindo a estrutura original da arquitetura proposta contribuíram para que o restante do projeto seguisse os mesmos padrões previamente definidos. Os códigos desenvolvidos para as funcionalidades em questão - e suas respectivas estruturas - serviram como exemplo para o desenvolvimento do restante das funcionalidades do aplicativo.

A arquitetura utilizada foi suficiente para a organização e estruturação do projeto. Embora a introdução de novas tecnologias - como a linguagem Kotlin - e bibliotecas - como RxJava e KodeIn - fortemente atreladas à arquitetura possam ter contribuído para a ocorrência de uma curva de aprendizado inicialmente íngreme a ser percorrida pelos desenvolvedores do aplicativo, é natural que a introdução de qualquer arquitetura e/ou tecnologias em um projeto causem tal efeito. O conhecimento porém adquirido no desenvolvimento da aplicação - e de sua arquitetura - utilizando os princípios e estruturas aqui apresentados é de valor inestimável. Caso o projeto venha a ter futuras evoluções, com a introdução de novas funcionalidades e/ou mesmo a introdução de testes unitários, é gratificante saber que tais desenvolvimentos estarão bem encaminhados, dada a base de código já existente e a arquitetura preparada para suportar a evolução do projeto.

Referências

[1] ENNES, M., & ROMANINI Jr, M. Caminhada como prática do não-saber: uma reflexão sobre des-com-passos cotidianos. Disponível em: <https://www.publionline.iar.unicamp.br/index.php/simpac/article/download/4400/4404>. Acesso em 14 de jan de 2021.

[2] MARTIN, Robert C. The Clean Architecture. The Clean Code Blog, 13 de ago. de 2012. Disponível em: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>. Acesso em: 09 de jan. de 2021.

[3] MARTIN, Robert C. (2017) Clean Architecture: A Craftsman's Guide to Software Structure and Design : Pearson.

[4] Model-View-Presenter. Wikipédia. Disponível em: <https://en.wikipedia.org/wiki/Model-view-presenter>. Acesso em: 09 de jan. de 2021

[5] MVP architectural pattern. Disponível em: <https://ducmanhphan.github.io/2019-08-05-MVP-architectural-pattern/>. Acesso em: 09 de jan. de 2021.

[6] SORAL, Rakshit. Architectural Guidelines to follow for MVP pattern in Android. AndroidPub, 06 de mar. de 2018. Disponível em: <https://android.jlelse.eu/architectural-guidelines-to-follow-for-mvp-pattern-in-android-2374848a0157>. Acesso em: 15 de jan. de 2021.

[7] Git. Disponível em: <https://git-scm.com/>. Acesso em: 09 de jan. de 2021.

[8] GitLab. Disponível em: <https://about.gitlab.com/>. Acesso em: 09 de jan. de 2021.

[9] Código Fonte do Mapa Afetivo. Disponível em: <https://gitlab.ic.unicamp.br/ra164468/mapa-afetivo>. Acesso em: 17 de jan. de 2021.

[10] Android-CleanArchitecture. Disponível em: <https://github.com/android10/Android-CleanArchitecture/>. Acesso em: 15 de jan. de 2021.

[11] OpenDesign. Disponível em: <https://mc750.opendesign.ic.unicamp.br/>. Acesso em: 09 de jan. de 2021.

[12] Reis, J.; Santos, A.; Duarte, E.; Gonçalves, F.; Nicolau de França, B.; Bonacin, R. and Baranauskas, M. (2020). Articulating Socially Aware Design Artifacts and User Stories in the Conception of the OpenDesign Platform.In Proceedings of the 22nd International Conference on Enterprise Information Systems - Volume 2: ICEIS, ISBN 978-989-758-423-7, pages 523-532. DOI: 10.5220/0009418205230532

[13] RxJava2. Disponível em: <https://github.com/ReactiveX/RxJava/tree/2.x>. Acesso em: 09 de jan. de 2021.

[14] ReactiveX. Disponível em: <http://reactivex.io/>. Acesso em: 09 de jan. de 2021.

[15] Kodein. Disponível em: <https://github.com/Kodein-Framework/Kodein-DI>. Acesso em: 09 de jan. de 2021.

[16] Kotlin. Disponível em: <https://kotlinlang.org/>. Acesso em: 09 de jan. de 2021.

[17] Extensions - Kotlin Programming Language. Disponível em: <https://kotlinlang.org/docs/reference/extensions.html#extension-functions>. Acesso em: 10 de jan. de 2021.

[18] Application — Android Developers. Disponível em: <https://developer.android.com/reference/android/app/Application>. Acesso em: 10 de jan. de 2021.

[19] Google Maps Android SDK. Disponível em: <https://developers.google.com/maps /documentation/android-sdk/overview>. Acesso em: 09 de jan. de 2021.

[20] Google Play Services. Disponível em: <https://developers.google.com/android/guides/overview>. Acesso em: 09 de jan. de 2021.

[21] Firebase. Disponível em: <https://firebase.google.com/>. Acesso em: 09 de jan. de 2021.

[22] Firestore. Disponível em: <https://firebase.google.com/docs/firestore/>. Acesso em: 09 de jan. de 2021.

[23] LEDOUX, Jacques. RxJava2: Schedulers 101 or simplified concurrency, part 1. Medium, 08 de jul. de 2018. Disponível em: <https://medium.com/@softjake/rxjava2-schedulers-101-or-simplified-concurrency-management-40ab0ed1ce1d>. Acesso em: 09 de jan. de 2021.

[24] Scheduler. Disponível em: <http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Scheduler.html>. Acesso em: 09 de jan. de 2021

[25] TAN, Lawrence. Working with RxJava Disposables in Kotlin. raywenderlich.com, 04 de nov. de 2019. Disponível em: <https://www.raywenderlich.com/3983802-working-with-rxjava-disposables-in-kotlin>. Acesso em: 09 de jan. de 2021.

[26] TEODOSIO, Bernardo do Amaral. Motivos para trocar o Java pelo Kotlin ainda hoje. Medium, 15 de jan. de 2018. Disponível em: <https://medium.com/movile-tech/trocar-o-java-pelo-kotlin-8bed76014d99>. Acesso em: 09 de jan. de 2021.

[27] Shaw, Mary, and David Garlan. Software architecture. Vol. 101. Englewood Cliffs: prentice Hall, 1996.

[28] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. “Design patterns: Abstraction and reuse of object-oriented design.” In European Conference on Object-Oriented Programming, pp. 406-431. Springer, Berlin, Heidelberg, 1993.