UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
ENGENHARIA DE COMPUTAÇÃO

GERSON LUIS FERREIRA DA SILVA FILHO

DESENVOLVIMENTO DE APLICATIVO PARA ADOÇÃO DE ANIMAIS ABANDONADOS UTILIZANDO A LINGUAGEM DE PROGRAMAÇÃO KOTLIN E PROGRAMAÇÃO REATIVA

TRABALHO DE CONCLUSÃO DE CURSO

CURITIBA
2017

GERSON LUIS FERREIRA DA SILVA FILHO

DESENVOLVIMENTO DE APLICATIVO PARA ADOÇÃO DE ANIMAIS ABANDONADOS UTILIZANDO A LINGUAGEM DE PROGRAMAÇÃO KOTLIN E PROGRAMAÇÃO REATIVA

Trabalho de Conclusão de Curso apresentado aos Departamentos Acadêmicos de Eletrônica e Informática como requisito para obtenção do título de Bacharel em Engenharia da Computação - Universidade Tecnológica Federal do Paraná - Câmpus Curitiba
Orientador:Prof. Dr. Rubens Alexandre de Faria

CURITIBA
2017

Gerson Luis Ferreira da Silva Filho

Desenvolvimento de aplicativo para adoção de animais abandonados utilizando a linguagem de programação Kotlin e programação reativa

Trabalho de Conclusão de Curso apresentado aos Departamentos Acadêmicos de Eletrônica e Informática como requisito para obtenção do título de Bacharel em Engenharia da Computação - Universidade Tecnológica Federal do Paraná - Câmpus Curitiba
Orientador:Prof. Dr. Rubens Alexandre de Faria

Trabalho aprovado. Curitiba, 2017:

Prof. Dr. Rubens Alexandre de Faria
UTFPR

Prof. Dra. Anelise Munaretto Fonseca
UTFPR

Prof. Me. Ricardo Umbria Pedroni
UTFPR

Curitiba
2017

RESUMO

FILHO, G. L. F. da S. *Desenvolvimento de aplicativo para adoção de animais abandonados utilizando a linguagem de programação Kotlin e programação reativa.* Trabalho de conclusão de curso - Engenharia de Computação - UTFPR - Campus Curitiba, 2017.

Um dos grandes problemas encontrados nas mais diversas cidades brasileiras é a grande quantidade de animais abandonados. Existem diversas ONGs e grupos que atuam no sentido recolher animais das ruas, dar uma primeira assistência necessária e encaminhá-los para um adotante responsável. Nesse sentido, esse projeto visa auxiliar nesse processo de três formas, primeiramente facilitando ao adotante encontrar o seu futuro animal de estimação de acordo com algumas preferências, em seguida possibilitar o encontro entre o adotante e o tutor temporário do animal. Por fim, auxiliar no acompanhamento do animal pelo seu resgatante à fim de garantir uma adoção responsável. Para desenvolvimento de aplicativos móveis temos inúmeras opções no mercado, e uma das mais expoentes no momento é o desenvolvimento de aplicativos para a plataforma *Android* utilizando a linguagem de programação *Kotlin*. Unida ao cunho social desse projeto, temos também uma contribuição acadêmica relevante, que é a utilização das mais recentes técnicas de engenharia de *software*, aplicadas no processo de desenvolvimento *Android*.

Palavras chaves: *Adoção de Animais, Aplicativos Móveis, Android, Kotlin, Programação Reativa.*

ABSTRACT

FILHO, G. L. F. da S. *Development of an Application for abandoned pet adoption using Kotlin programming language and reactive programming.* Term Paper, 2017.

One of the worst problems found on different brazilian cities is the amount of abandoned animals. There are many non-governamental institutions and groups that act to shelter animals from streets, give a first veterinary assitency and send them to a responsible adopter. On this way, this project is trying to help on this process in three ways, firstly making easier to the adopter find his future pet accondinly some preferences. After that, we want to enhance the communication between the adopter and the animal temporary tutor. Finally, improving the contact after the adoption. Nowadays, there are many options to develop mobile software, and one of the most growing is Android development using Kotlin programming language. This project wants to put together an academmic and a social goal, explaining a professional way to develop mobile software, bringing some of the most modern concepts in these subjects.

Key-words: *Pet Adoption, Mobile Application, Android, Kotlin, Reactive Programming*.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

UTFPR	Universidade Tecnológica Federal do Paraná
ONG	Organização Não Governamental
SPAC	Sociedade Protetora dos Animais de Curitiba
ONU	Organização das Nações Unidas
TDD	*Test Driven Development*
UML	*Universal Modeling Language*
UI	*User Interface*
UIPA	União Internacional Protetora dos Animais
SDK	*Software Development Kit*
NDK	*Native Development Kit*
ART	*Android Runtime*
HAL	*Hardware Abstraction Layer*
API	*Application Programming Interface*
JVM	*Java Virtual Machine*
OOP	*Object Oriented Programming*
MVC	*Model View Controller*
MVP	*Model View Presenter*
IoC	*Inversion of Control*
DI	*Dependency Injection*
IDE	*Integrated Developemnt Environment*
IBGE	Instituro Brasileiro de Geografia e Estatística
MSDN	*Microsoft Developer Network*

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta a contextualização do assunto deste trabalho, bem como a definição do problema, o objetivo geral e os objetivos específicos.

1.1 MOTIVAÇÃO E JUSTIFICATIVA

A relação entre humanos e seus animais de estimação se torna cada vez mais evidente no nosso dia-a-dia. Esse fenômeno não é novo, porém observa-se um grande aumento da importância dada à essa relação nas famílias de todo mundo, inclusive no Brasil. Mesmo com esse fenômeno, observa-se grandes quantidades de cães abandonados nos centros urbanos. Isso ocorre por alguns fatores como a facilidade de reprodução dos animais, abandono por parte dos responsáveis ou criação de animais para a reprodução e comercialização.

Segundo a organização não governamental Sociedade Protetora dos Animais (SPAC), em Curitiba temos aproximadamente 450 mil cães, sendo que destes 229 mil estão nas ruas. Até 2005, os animais recolhidos pela Prefeitura de Curitiba eram sacrificados (SILVA, 2006). Com a prática proibida, hoje, a Prefeitura só realiza o recolhimento de animais agressivos, doentes ou que possam passar alguma enfermidade a seres humanos. Sendo assim, torna-se extremamente importante que haja um controle sobre esses animais, tanto para a saúde deles, como para a da população (ARAUJO, 2016).

Além das políticas públicas desenvolvidas pelos órgãos governamentais existem ONGs que realizam o trabalho de resgate, tratamento e encaminhamento dos animais para adoção responsável. Essas organizações, que muitas vezes dependem apenas de trabalho voluntário e doações da comunidade, utilizam diversos meios para encontrar adotantes para seus animais, como feiras de adoção, anúncio em sites especializados e principalmente mídias sociais como *Facebook* e *Whatsapp*.

Apesar de muito importantes para a divulgação e expansão dos trabalhos, os meios disponíveis não abordam alguns pontos importantes como facilidade na criação de filtros para busca dos animais, acesso fácil e intuitivo, acompanhamento pós adoção, busca de animais por geo-localização e ainda facilidade no cadastro de novos animais pelas ONGs.

Este projeto visa criar uma nova forma das pessoas encontrarem animais para adoção, de forma prática e organizada. Além disso, tem por finalidade chamar atenção ao tema, para mostrar que existem inúmeros animais, das mais diversas características, disponíveis para adoção. Busca-se ainda, ressaltar o ótimo trabalho realizado pelos voluntários que acolhem os animais das ruas e os colocam para adoção.

1.2 PROBLEMATIZAÇÃO E OBJETIVOS

Na etapa de elaboração do projeto, foi pensado um aplicativo para ajudar a informar locais onde existissem animais abandonados nas ruas. Apesar de ser muito importante tirá-los das ruas, depois de alguma pesquisa com diversas organizações de apoio aos animais, percebe-se que a maior demanda é na busca por pessoas que os adotem e que a grande maioria dos abrigos já estão super lotados.

Outra demanda muito importante das ONGs é obrigatoriedade da adoção responsável, que não apenas seleciona um animal para o dono, mas também verifica se o dono atende todas as necessidades exigidas pelo animal (como espaço, condições de higiene, atenção e suporte veterinário) (ORLANDO, 2014). Para realizar esse trabalho de acompanhamento, utilizam-se as redes sociais, como *Facebook* e *Whatsapp*.

Para auxiliar nessa busca e facilitar todo o processo de adoção, será criado um aplicativo que pode ser dividido em três passos principais. Primeiramente, o cadastro dos animais pelas ONGs ou lares temporários. Em seguida, um mecanismo de busca e visualização muito popular em redes de relacionamentos, a fim de facilitar a busca de um animal por seus futuros adotantes. Por fim, uma forma fácil para as instituições acompanharem esse animal após a sua adoção.

1.2.1 Objetivo Geral

Este projeto tem como finalidade facilitar a comunicação entre as instituições de apoio aos animais, e pessoas interessadas em adotá-los, por meio de uma aplicação desenvolvida para a plataforma *Android*.

1.2.2 Objetivos Específicos

Tendo como ponto de partida o objetivo geral, definiram-se os objetivos específicos para o desenvolvimento deste projeto. Estes objetivos são:

- Desenvolver uma aplicação que seja como um catálogo de animais disponíveis para adoção.
- Criar uma plataforma para uma comunicação mais ágil e organizada, entre as ONGs de proteção animal, e os candidatos à adoção dos mesmos.
- Utilizar a nova linguagem de desenvolvimento *Kotlin*, para a plataforma *Android*.
- Aplicar conceitos de engenharia de *software*, a fim de demonstrar suas vantagens.

- Realizar a inclusão inicial de animais reais disponíveis para a adoção na aplicação.

1.3 APRESENTAÇÃO DO DOCUMENTO

A estrutura do presente trabalho divide-se da seguinte maneira:

O capítulo 1 contém a *Introdução*, na qual é apresentada a motivação e justificativa do trabalho além de uma visão geral do problema.

O capítulo 2 trata da *Fundamentação Teórica* que envolve o tema de abandono e adoção de animais e traz alguns conceitos importantes nessa área.

O capítulo 3 trata da *Fundamentação Técnica* que envolve os métodos de programação e conhecimentos utilizados pelo sistema aqui apresentado.

O capítulo 4 refere-se ao *Desenvolvimento* do sistema para auxílio na adoção de animais. Neste capítulo são apresentadas as principais características, os requisitos funcionais e não funcionais do *software*, a arquitetura do sistema bem como o seu funcionamento.

O capítulo 5 trata da *Gestão do Projeto*. Neste capítulo são apresentadas as tarefas estimadas à serem executadas, as horas necessárias para cada uma delas e ainda os custos associados ao desenvolvimento e manutenção do projeto.

O capítulo 6 aborda os *Resultados obtidos*, e tem por finalidade ressaltar o que foi conseguido com o desenvolvimento deste projeto, ressaltando os problemas enfrentados durante o desenvolvimento.

O capítulo 7, *Conclusão*, apresenta os resultados obtidos com o desenvolvimento do sistema, os problemas e dificuldades encontradas, juntamente com sugestões para futuros desenvolvimentos.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo introduz o conceito de abandono de animais, traz explicação de como funcionam as ONGs que trabalham nessa causa, e aborda alguns dos recursos utilizados por essas organizações no resgate, busca de um lar e tratamento destes animais de rua.

2.1 ANIMAIS DE ESTIMAÇÃO

Um assunto muito discutido na sociedade atual e que vem ganhando cada vez mais força é a relação entre os seres humanos e os seus animais de estimação. Segundo (HEIDEN, 2012) alguns fatores que diferenciam os animais dos seres humanos são responsáveis pela proximidade desta relação. As diferenças principais são o sentimento de uma emoção de cada vez e a maior sensibilidade a linguagem corporal, que são mais intensos nos animais. Apesar das diferenças, existem alguns pontos que aproximam ainda mais esta convivência, como o grande apreço pelo contato social e a forma de aprendizado que se dá por observação. A Associacão Americana de Medicina Veterinária define a relação humano animal como "uma relação dinâmica e mutuamente benéfica entre pessoas e outros animais, influenciada pelos comportamentos essenciais para a saúde e bem estar de ambos. Isso inclui as interações emocionais, psicológicas e físicas entre pessoas, demais animais e ambiente" (FARACO, 2008).

2.2 ABANDONO DE ANIMAIS

O abandono de animais é um ato cruel, porém muito comum na nossa sociedade. Pode-se observar diversas histórias de pessoas que abandonam o animal que fica indefeso em lugares perigosos, como estradas ou campos. Segundo (ALMEIDA, 2011), o número de animais abandonados cresce no período que antecede as festas de fim de ano e as férias escolares, quando as famílias não têm onde deixar o animal para ir viajar.

Para evitar esse tipo de atitude aconteça, precisa-se investir na educação da sociedade e definição de regras mais claras para a punição das pessoas que praticam esses atos (DELABARY, 2012). O processo de educação deve ser no sentido de orientar as pessoas que adotam animais de todas as responsabilidades perante o mesmo, além de identificar exatamente se todas as suas necessidades são atendidas, como tratamento veterinário, espaço necessário e até evitar que os animais fiquem sozinhos por muito tempo.

2.3 ASSOCIAÇÃO DE PROTEÇÃO AOS ANIMAIS

Existem diversas associações públicas e privadas que são dedicadas à proteção animal. As organizações privadas muitas vezes são chamadas de ONGs (Organizações não governamentais), termo criado pela ONU na década de 1950, e é delimitado pelo Manual sobre as Instituições Sem Fins Lucrativos, como organizações constituídas legalmente, que estão fora do aparelho estatal, que não distribuem lucros, que gerenciam suas próprias atividades — têm autonomia — de maneira voluntária e podem ser constituídas de modo livre e por qualquer pessoa ou grupo de pessoas (LAZARIN, 2014).

Segundo o mapeamento do terceiro setor, realizado pelo Instituto Brasileiro de Geografia e Estatística (IBGE), em 2010 existiam 2.242 instituições voltadas à proteção de animais, ou do meio ambiente no Brasil. Dessas ONGs, muitas são dedicadas ao resgate, tratamento e, por fim, encontrar um lar para os animais abandonados nas grandes cidades, a fim de proporcionar uma vida mais digna e feliz para eles.

Como muitas destas instituições dependem exclusivamente de trabalho voluntário e doações para realizar o seu trabalho, algumas pessoas resgatam os animais e os acolhem nas suas próprias casas, até encontrarem adotantes definitivos. São os chamados lares temporários. Para adotar um animal nessas instituições normalmente os adotantes são obrigados a assinar um termo de compromisso e ainda o animal deve ser castrado (ou ter a castração agendada) e vacinado.

Com o advento da internet e das redes sociais, surgiram diversos meios de comunicação para as pessoas encontrarem os cães disponíveis para adoção. A principal forma das ONGs divulgarem os animais são as páginas do *Facebook*, auxiliando também na avaliação do adotante pelo seu perfil na rede social. Já a a comunicação entre os tutores temporários e os futuros adotantes é por meio de mensagens enviadas pelo aplicativo *Whatsapp*.

2.3.1 Lares temporários

São chamados de lares temporários, as residências de voluntários que acolhem os animais abandonados, até que os mesmos sejam adotados pelos seus tutores definitivos. Normalmente esses lares estão lotados, não conseguindo acolher todos os animais encontrados. A pessoa responsável pelo lar temporário é a tutora dos animais, até que os mesmos sejam adotados, e é dever dela garantir que os mesmos tenham uma adoção responsável.

2.3.2 Adoção Responsável

A adoção responsável deve seguir algumas regras básicas, a fim de garantir a saúde, a segurança e o conforto dos animais. Segundo a UIPA uma adoção responsável deve seguir as seguintes regras (ORLANDO, 2014):

- Ao decidir-se por acolher um animal, tenha em mente que ele viverá cerca de 12 anos, ou mais, e que necessitará de seus cuidados, independentemente das mudanças que sua vida venha a sofrer no decorrer desse período;
- Prefira sempre adotar, a comprar um animal. Ao adotar um animal, luta-se não só contra o abandono, mas contra o comércio de animais praticado por criadores, que se perfaz à custa de extrema crueldade. É preciso ter consciência de que adquirir um animal de criador implica, necessariamente, patrocinar o abusivo comércio de animais;
- Certifique-se de que poderá cuidar do animal durante o período de férias e no decorrer de feriados;
- Escolha o animal que possua características de comportamento e de tamanho condizentes com o espaço de que dispõe e com os seus próprios hábitos;
- Ministre-lhe assistência veterinária ;
- Providencie para que seja o animal, macho ou fêmea, esterilizado para evitar crias indesejadas que resultam em abandono e em superpopulação de animais;
- Vaciná-lo, anualmente, a partir dos 60 (sessenta) dias de vida;
- Não abandoná-lo em caso de doença, de idade avançada, de viagem, de agressividade ou de outra hipótese;
- Proporcionar-lhe alimentação adequada à espécie; gatos não devem ser alimentados com ração para cães e vice-versa;
- Proporcionar-lhe água fresca (água estagnada acumula larvas de mosquitos, que são prejudiciais à saúde);
- Provê-lo de espaço adequado, ao abrigo do sol e da chuva. Melhor é que se tenha o animal dentro de casa, mas se isso não for possível, dê-lhe ao menos uma casinha, que deve ser colocada ao abrigo do sol, da chuva e do vento;
- Não prendê-lo a correntes, cordas ou a aparato similar. Dê ao animal um lar, e não uma prisão;
- Zelar para que o animal não fuja de casa, providenciando para que os portões de casa sejam resistentes e estejam sempre bem fechados;

- Telar as janelas, caso more em prédio de apartamentos;
- Mantê-lo em boas condições de higiene (a água do banho deve ser quente);
- Jamais submetê-lo a maus-tratos, nem sob o pretexto de educá-lo;
- Passear com o animal para que ele se exercite, sempre preso à coleira e à guia para evitar fuga, atropelamento, ataques a outros animais. Evite levá-lo para passear em horário de sol forte, pois o contato com o solo quente pode causar desconforto e até queimaduras;
- Dar afeto e atenção ao animal;
- Proporcionar-lhe conforto e espaço adequado; áreas descampadas, estacionamentos e garagens não são recomendáveis para animais;
- Amenizar-lhe a sensação de frio, por meio de roupas e cobertores; animais sentem frio tanto quanto os humanos.

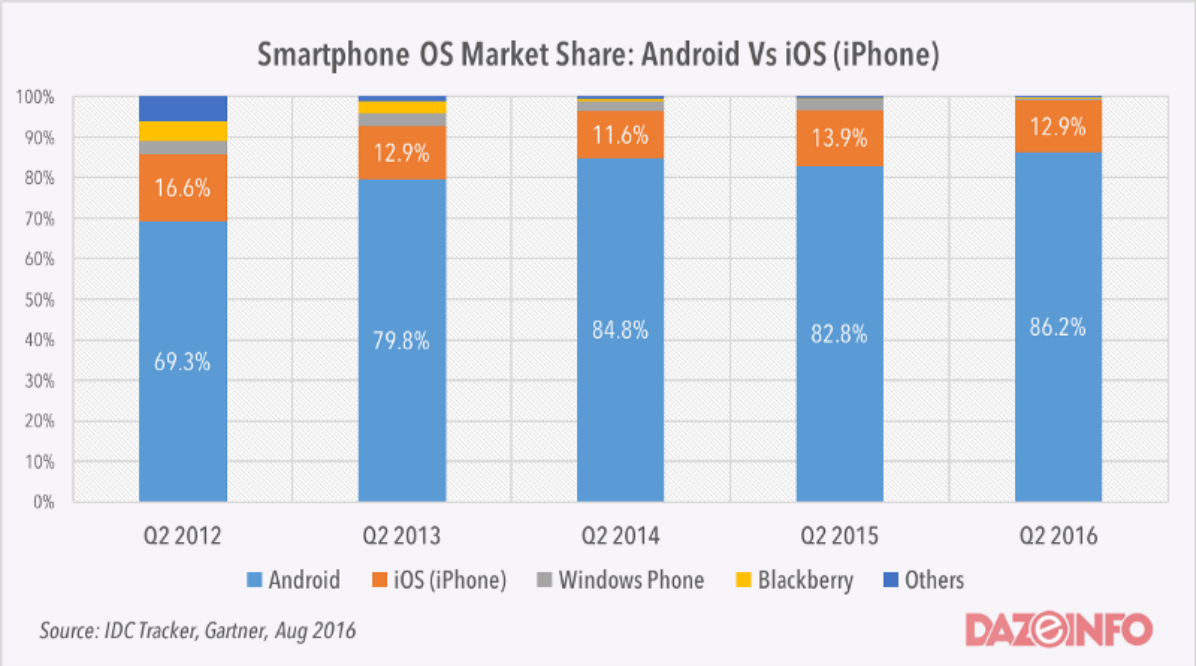
3 FUNDAMENTAÇÃO TÉCNICA

Neste capítulo são apresentados os métodos e os conhecimentos utilizados para o desenvolvimento do sistema, como programação para dispositivos móveis, paradigmas de programação, padrões de arquitetura de sistemas e de *design*.

3.1 PROGRAMAÇÃO PARA DISPOSITIVOS MÓVEIS

Os dispositivos móveis são produtos eletrônicos projetados para serem facilmente levados de um lugar ao outro, hoje em dia muito ligados aos *smartphones*, *tablets* e até os dispositivos vestíveis, como relógios e óculos. Os sistemas operacionais mais utilizados nestes dispositivos são o *Android* e o *iOS*, tendo o primeiro utilizado em mais de 85% dos dispositivos no segundo quarto do ano de 2016, como pode ser visto na figura abaixo (LTD, 2017):

Figura 1 – Utilização de Sistema operacional em dispositivos móveis



Fonte: https://android.jlelse.eu (2017)

Devido às diferenças de design e tecnologias, o desenvolvimento nativo para *Android* e *iOS* se dá em linguagens e ambientes diferentes, obrigando praticamente todos os desenvolvedores, que queiram atingir o maior número de usuários possíveis, a desenvolver para ambas as plataformas. Outra possibilidade existente é a utilização de plataformas de desenvolvimento híbridas, que trazem benefícios em questões de reaproveitamento de código e manutenção, porém pecam em aspectos como *design* exclusivo para cada plataforma e desempenho.

Neste projeto decidiu-se utilizar a plataforma *Android* com desenvolvimento nativo devido ao número elevado de usuários, sua característica de *software* aberto, garantir a qualidade e desempenho de algumas características visuais do *App*, utilizando conceitos novos, porém muito bem aceitos no mercado.

3.1.1 Desenvolvimento Android

Para criar aplicações executáveis em dispositivos móveis com o sistema operacional *Android*, utilizamos o kit de desenvolvimento de software (SDK) fornecido pelo *Google* para a plataforma. Neste SDK, tem-se algumas ferramentas para criar os aplicativos, exemplos de código fonte, ferramentas de desenvolvimento, emuladores e bibliotecas necessárias (CORDEIRO, 2017).

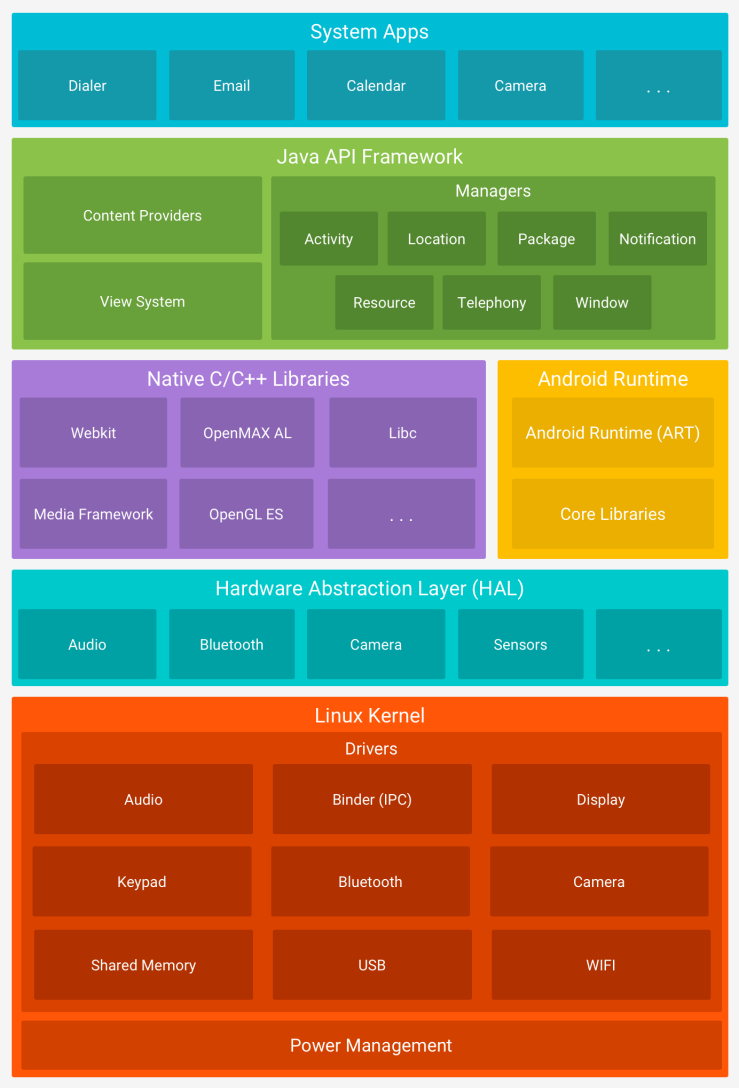
Os aplicativos são escritos para serem executados na ART e Dalvik, máquinas virtuais personalizadas e projetadas para rodar dentro dos dispositivos *Android*, que funcionam sobre um Kernel Linux.

Inicialmente as linguagens com suporte oficial do Google para desenvolvimento *Android*, era o Java e o C++. Recentemente, a empresa anunciou que a nova linguagem Kotlin, desenvolvida pela JetBrains, será suportada como uma linguagem oficial para desenvolvimento de aplicações nativas *Android* (2, 2017).

3.1.1.1 Camadas *Android*

Devido à quantidade e flexibilidade de suas aplicações, o sistema operacional *Android* possui diversas camadas para garantir a segurança de suas aplicações. Essas camadas podem ser observadas na figura 2 (GOOGLE, 2017).

Figura 2 – Camadas do sistema Android



Fonte: https://developer.android.com/guide/platform/index.html (2016)

3.1.1.1.1 kernel Linux

O *kernel Linux* é a camada fundamental para o sistema operacional. Nela encontram-se as implementações de *threads*, gerenciamento de memória, *drivers* e outras funcionalidades de baixo nível. Ele traz uma vantagem em relação à segurança devido ao grande e contínuo desenvolvimento voltado ao núcleo do *Linux*.

3.1.1.1.2 Camada de Abstração de Hardware - HAL

Esta camada possibilita a flexibilidade de dispositivos e fabricantes suportados pelo sistema operacional *Android*. Nela são criados protocolos que servem como abstrações para a utilização de diferentes componentes de *hardware*, como câmera, *bluetooth* e outros. Estes protocolos são seguidos pelos fabricantes, a fim de suportar as chamadas do sistema e dos aplicativos.

3.1.1.1.3 Android Runtime

A partir da versão 5.0 do *Android*, utiliza-se a ART (*Android runtime*) para executar os aplicativos. Com ela cada *app* tem seu próprio processo e uma instancia única para executar seus arquivos DEX, que são binários, otimizados, criados especialmente para o *Android*. Para as versões anteriores são utilizadas as máquinas virtuais Dalvik, porém todos os aplicativos desenvolvidos para a nova ART são compatíveis para rodar também nos sistemas mais antigos.

3.1.1.1.4 Bibliotecas Nativas C/C++

As duas camadas anteriores foram desenvolvidas para abstrair o código C/C++ que interfaceia com o *Kernel do Linux*, a partir de uma API Java. Porém, para alguns tipos de desenvolvimento, é interessante utilizar diretamente recursos do sistema. Para isso, existe a *Android* NDK, plataforma de desenvolvimento nativa do *Android*. Exemplos de uso para o ndk são os jogos que utilizam *OpenGL*, para melhorar performance dos gráficos.

3.1.1.1.5 Java API Framework

O conjunto de bibliotecas disponíveis para programação *Android*, escrita em Java, se encontra nesta camada. As diversas abstrações para controle da interface gráfica, componentes de sistema e serviços, que simplificam a reutilização de código, são expostas por este módulo. Tanto as aplicações que gerenciam o sistema, quanto as aplicações instaladas pelo usuário utilizam estes componentes.

3.1.1.1.6 Aplicações do Sistema

Os sistemas operacionais *Android* geralmente vêm com algumas aplicações previamente instaladas, como câmera, *email*, telefone, agenda, entre outras. Estas aplicações possuem uma camada própria para que sejam empacotadas com o sistema operacional e possam ser utilizadas para as mais diversas funcionalidades. No mesmo grau de abstração que as aplicações de sistema se encontram as aplicações instaladas pelos usuários por meio da loja, ou dos pacotes de aplicativo APK.

3.2 LINGUAGENS DE PROGRAMAÇÃO

Nesta seção serão apresentadas duas linguagens utilizadas para desenvolvimento de aplicações *Android*. As linguagens *Java* e *Kotlin*. Ambas são executadas, da mesma forma, por uma máquina virtual que abstrai componentes do sistema operacional. Como foi visto nas camadas *Android* tem-se uma máquina virtual para cadas aplicação.

3.2.1 Java

Java é uma linguagem de programação orientada a objetos, desenvolvida na década de 1990 pela empresa *Sun Microsystems*. Os programas escritos em Java são compilados para *bytecode*, que é um código interpretado por uma máquina virtual. A máquina virtual mais popular para executar os programas em Java é a JVM (*Java Virtual Machine*), cujo interpretador está instalado nas principais plataformas de execução do mercado, incluindo os principais navegadores *Web* (ORACLE, 2017).

3.2.2 Kotlin

Assim como Java, *Kotlin* é uma linguagem de programação baseada em uma máquina virtual JVM. Ela foi desenvolvida pela empresa *JetBrains* e foi anunciada em 2011. *Kotlin* surgiu como uma alternativa ao Java, que com o passar do tempo se tornou muito "verborosa", o oposto da principal característica da nova linguagem, que é a sintaxe enxuta. Um ponto muito interessante do *Kotlin*, que facilita muito a adesão de desenvolvedores Java, é a sua total interoperabilidade com as bibliotecas da linguagem mais antiga, ou seja, podem-se utilizar todos os recursos do Java nos programas escritos em *Kotlin* (JETBRAINS, 2016).

A *JetBrains* declara que trabalha para que o tempo de compilação do *Kotlin* seja tão rápido quanto é o do Java. Em 2012, a empresa abriu o seu código fonte para desenvolvedores de código aberto, o que segundo a mesma fez crescer o interesse da comunidade pela linguagem.

É claro que por suportar todas as chamadas em Java, *Kotlin* também pode ser utilizada para criar aplicações *Android*. Com o passar dos anos, ela vem se desenvolvendo cada vez mais para esta plataforma, e já é utilizada por grandes empresas como padrão para desenvolvimento dos seus aplicativos. Em Maio de 2017, a *Google* oficializou *Kotlin* como uma linguagem de desenvolvimento *Android* suportada pela empresa, e ainda anunciou que a utilizarão em algumas bibliotecas da plataforma (2, 2017).

3.3 PARADIGMAS DE PROGRAMAÇÃO

No desenvolvimento de *software*, uma parte fundamental é a abstração em relação ao mundo real. Os paradigmas de programação são utilizados como estruturas que facilitam esta abstração, e criam uma estrutura lógica para o funcionamento dos programas. Serão apresentados três paradigmas de programação nesta seção, programação estruturada, que tem uma importância histórica, programação orientada a objetos que é um paradigma consagrado no mercado e, se tornando cada vez mais popular, a programação orientada a fluxos ou programação reativa.

Paradigmas de programação são seguidos por linguagens, porém é importante ressaltar que, ao utilizar uma linguagem que é desenvolvida baseando-se em um paradigma, não se garante que o mesmo será totalmente implementado. Um exemplo seria um programa escrito em Java, uma linguagem orientada a objetos, porém, utilizando uma única classe principal, chamando funções e rotinas de forma estruturada.

3.3.1 Programação Estruturada

A programação estruturada tem como ponto fundamental três estruturas que podem ser utilizadas para criação de qualquer programa de computadores. As estruturas são as rotinas, as decisões e as repetições. Este padrão ficou conhecido no final da década de 1950 e foi instituído a fim de facilitar o entendimento do código criado, evitando os saltos muito utilizados antigamente com as chamadas *"GoTo"*, que tornam o código difícil de entender e de fazer modificações. A programação estruturada, mesmo sendo uma forma muito antiga de programação, continua sendo muito importante, pois geralmente é a porta de entrada para estudantes aprenderem lógica de programação. Além disso, dentro de outros padrões, como a programação orientada a objetos, pode-se observar conceitos da programação estruturada.

3.3.2 Programação Orientada a Objetos

A programação orientada a objetos (OOP), é atualmente o paradigma de programação mais popular para desenvolvimento de aplicações. O conceito fundamental para este paradigma é a criação de objetos, que são abstrações do mundo real, ou do meio em que o *software* será utilizado. Os 4 pilares da programação orientada a objetos são a abstração, o encapsulamento, a herança e o polimorfismo (MACHADO, 2017).

3.3.2.1 Abstração

Como posto anteriormente, a abstração é conceito fundamental para a orientação a objetos. Nela, elementos reais são convertidos em objetos que possuem um nome único, propriedades específicas que modelam as características do objeto, e por fim as ações que o objeto pode executar.

3.3.2.2 Encapsulamento

O encapsulamento é uma das principais técnicas que define a programação orientada a objetos. Trata-se de um dos elementos que adicionam segurança à aplicação, pelo fato de esconder as propriedades, criando uma espécie de caixa preta da classe.

A maior parte das linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas, ligadas a métodos especiais chamados *getters* e *setters*, que irão retornar e setar o valor da propriedade, respectivamente. Essa atitude evita o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação.

Para se fazer um paralelo com o que é visto no mundo real, tem-se o encapsulamento em outros elementos. Por exemplo, quando se clica no botão ligar da televisão, não se sabe o que está acontecendo internamente. Pode-se, então, dizer que os métodos que ligam a televisão estão encapsulados.

3.3.2.3 Herança

A herança é uma forma de tornar a programação mais segura e reaproveitável. Com ela é possível criar uma hierarquia entre os objetos e assim economizar linhas de código, reaproveitando características já mapeadas. Por exemplo, um objeto carro possui uma cor, rodas, portas, porém podem-se criar vários modelos de carros que possuem todos esses elementos e até alguns outros.

3.3.2.4 Polimorfismo

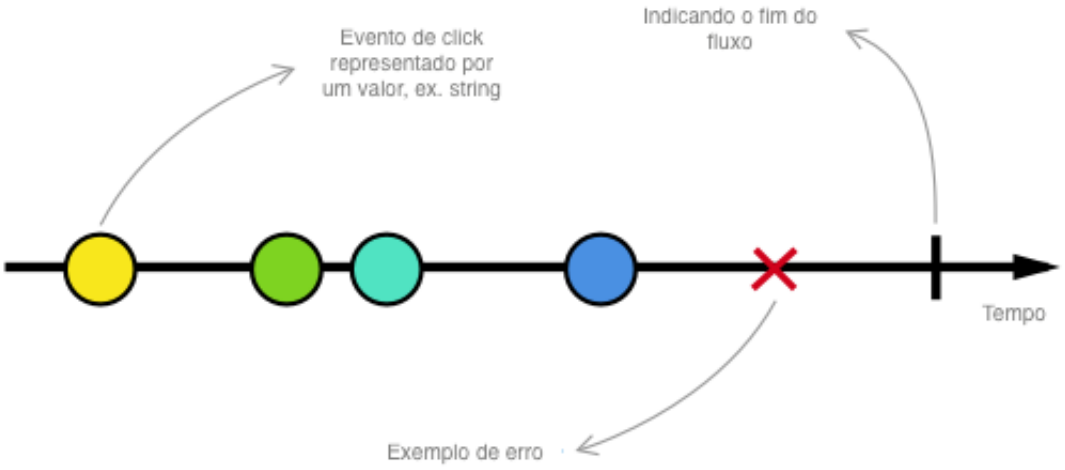
Polimorfismo está diretamente ligado com herança. Esta característica garante que filhos de um objeto pai podem realizar ações de forma diferente, dependendo da sua implementação. Um exemplo seria um objeto eletrodoméstico, ele pode ter os filhos televisão e geladeira. Os dois objetos tem uma ação de ligar, porém cada um liga de uma forma diferente, e mais, executa ações diferentes quando são ligados.

3.3.3 Programação Reativa

Programação reativa é um paradigma de programação que se utiliza de fluxo de dados assíncronos e a propagação de mudanças (BLACKHEAT, 2016). Nesse paradigma, eventos (e.g., cliques em botões da tela) formam um fluxo (*stream*) de eventos assíncronos. Um fluxo é uma

sequência contínua de eventos ordenadas no tempo, como ilustrado na figura 3.

Figura 3 – Fluxo de dados



Fonte: (STALTZ, 2016)

O oposto de programação reativa, ou que utiliza-se normalmente é a programação imperativa. Um exemplo tradicional para comparar os dois paradigmas seria utilizando a expressão $a := b + c$, em que define o valor de a como sendo o somatório entre b e c. Em uma programação imperativa se esta expressão é executada e em seguida alteramos o valor de b ou c o valor de a permanece o mesmo. Já utilizando programação reativa, o valor de a fica dependente dos valores de b e c e as mudanças nos valores destes, propagam valores pelo fluxo de dados, alterando também o valor de a (OLIVEIRA, 2016).

Observa-se que a programação reativa é indicada para utilizar com as aplicações existentes no mercado, pois com a *Internet* e os aplicativos para dispositivos móveis, tem-se uma grande exigência por separação de atividades e organização de diferentes fluxos de informação ao mesmo tempo. Por exemplo, em uma aplicação *mobile*, podem-se ter três fluxos simultâneos de dados, os vindos da *View* que são as interações do usuário, os do processamento do programa em si e os vindos de um serviço *web*.

3.4 PADRÕES DE ARQUITETURA

Os padrões de arquitetura são abstrações criadas a fim de aumentar a produtividade, facilitar a manutenção e melhorar a escalabilidade de projetos de *software*. Nesses padrões, são criadas camadas para separar as funcionalidades e criar uma regra com um fluxo de como as informações devem percorrer as classes do programa. Estas camadas de maneira geral aumentam a atomicidade de cada elemento, tornando-os facilmente substituíveis para melhorias nas

aplicações, ou simuláveis para testar elementos que se utilizem dele.

A escolha do padrão de arquitetura a se utilizar em um projeto deve ser em função das características que o projeto apresenta. Dentre as inúmeras opções de arquitetura, foram escolhidas as que são mais utilizadas para desenvolvimento de dispositivos móveis atualmente.

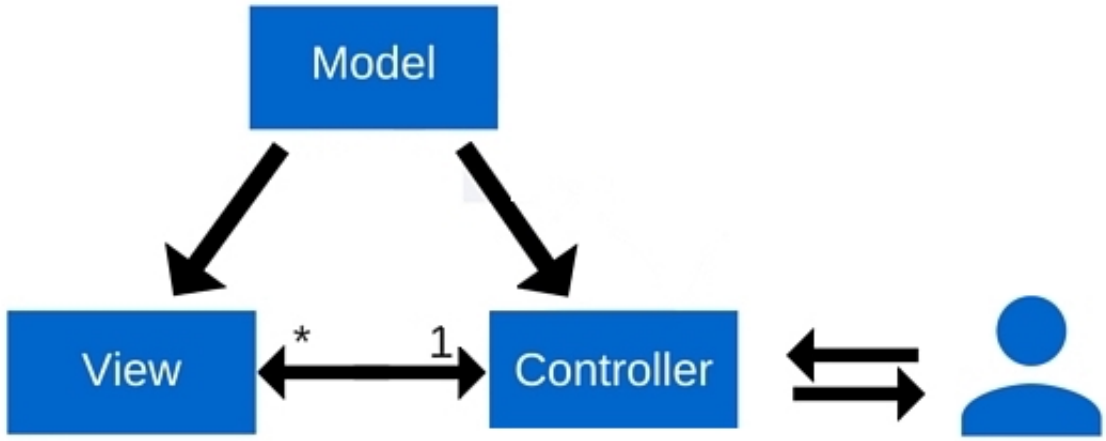
3.4.1 Padrão MVC

O padrão *Model-View-Controller* (MVC), criado na década de 1970, visa fundamentalmente separar a aplicação em três camadas principais, o modelo, a interface gráfica e o controlador (MICROSOFT, 2016).

- *Model* - Nesta camada encontram-se os dados da aplicação, regras de negócios, lógicas e funções.
- *View* - Na *view* estão todos os elementos das camadas gráficas do programa, como as telas, as interações delas e seus efeitos e transições. Uma *view* é uma representação dos dados obtidos na camada de modelo do sistema.
- *Controller* - O controlador é a camada responsável por coordenar os estados do sistema. Ele recebe e envia comandos para a *view* a fim de coordenar o que será apresentado na tela baseado em condições e informações retiradas pelo *model*.

Este padrão é muito utilizado em aplicações *web* e para dispositivos móveis, onde vê-se claramente a separação da camada de *UI* da lógica do sistema. Outro ponto positivo é a exigência de sistemas *multi-threading*, em que uma rotina deve sempre monitorar as atualizações do sistema, e outras devem aguardar chamadas assíncronas vindas do banco de dados ou de servidores *web*.

Figura 4 – Padrão MVC



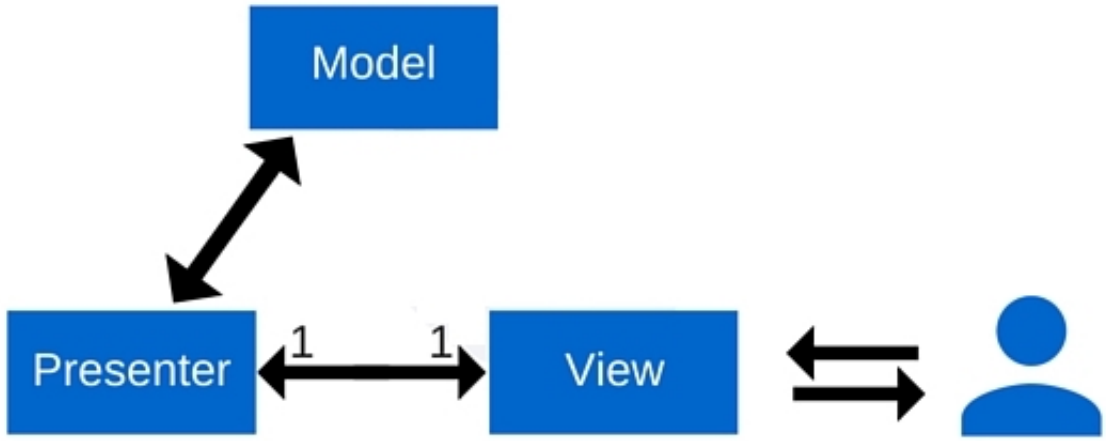
Fonte: (RISHABH, 2016)

3.4.2 Padrão MVP

O padrão MVP é muito similar ao MVC, porém o *controller* é substituído pelo *presenter* e a *view* não tem acesso direto aos dados dos modelos da aplicação. Em outras palavras, o *presenter* tem mais responsabilidades que o *controller*, pois todo o acesso dos dados e lógica de negócios da aplicação são realizados através dele, seja do *model* em relação à *view* (uma notificação *web*) ou da *view* em relação ao *model* (o clique de um botão)(RISHABH, 2016).

- *Model* - Assim como no MVC o *model* possui os dados da aplicação, regras de negócio e lógica da aplicação, porém ele não tem acesso direto à *view*, todas as requisições devem ser feitas ao *presenter*.
- *View* - Na *view* estão todos os elementos das camadas gráficas do programa, como as telas, as interações, seus efeitos e transições. No padrão MVP a *view* só recebe ou envia comandos a partir do *presenter*.
- *Presenter* - O *presenter* é a camada responsável por coordenar os estados do sistema e por todas as atualizações e consultas vindas tanto da *view* para o *model*, quanto do *model* para a *view*.

Figura 5 – Padrão MVP



Fonte: (RISHABH, 2016)

3.5 PADRÕES DE *DESIGN* (*DESIGN PATTERNS*)

Padrões de *design* são um conjunto de regras utilizadas no desenvolvimento de *software* que independem de linguagem ou arquitetura utilizada, e servem como guia para resolver problemas muito comuns encontrados por programadores. O conceito foi introduzido no final da década de 1970 e teve muitas mudanças e publicações relacionadas desde então. Uma publicação muito importante para o conceito atual de *design patterns* é o livro *Design Patters:Elements of Reusable Object-Oriented Software* de 1994, escrito por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Neste livro, são divididos os padrões de *design* em 3 grupos fundamentais (GAMMA *et al.*, 1994):

- Padrões de Criação: relacionado à criação de objetos.
- Padrões Estruturais: tratam das associações entre classes e objetos.
- Padrões Comportamentais: tratam das interações e divisões de responsabilidade entre as classes.

Atualmente existem diversos padrões de *design*, alguns mais e outros menos utilizados no desenvolvimento de *software*. Deve-se ressaltar que uma aplicação não deve obrigatoriamente utilizar os *design patterns*, porém seus conceitos facilitam muito na manutenção e escalabilidade do código, além de que por serem padrões universais, facilitam a inteligibilidade do código por

outros programadores. Na sequência, serão apresentados os padrões mais utilizados no desenvolvimento de aplicações *mobile*, que é o foco principal deste projeto.

3.5.1 Singleton

Este padrão talvez seja o mais conhecido e o utilizado pelos programadores. O padrão *singleton* garante a existência de apenas uma instância de uma classe durante a execução da aplicação (GEARY, 2003). Outro ponto importante deste padrão é o acesso a esse objeto, que deve ser público, tornando o *singleton* uma boa alternativa na hora de compartilhar recursos entre os elementos do programa. O passo-a-passo para a construção de um objeto *singleton* é:

1. Tornar o construtor privado, não possibilitando a criação do objeto por outros.
2. Criar um atributo privado e estático do mesmo tipo da classe.
3. Criar um método estático de instância (eg. *getInstance()*) que verifica se a variável já foi criada, se não foi cria a mesma, e sempre retorna essa variável, criada uma única vez.
4. Para acessar esse objeto, utilizamos sempre a estrutura *Classe.getInstance()*.

Como este projeto é voltado para desenvolvimento *Android*, foram escolhidos dois exemplos de implementação de um *singleton*, um na linguagem Java e outro na linguagem *Kotlin*. Ambos estão demonstrados na Figura 6.

Figura 6 – Código padrão *Singleton Java* e *Kotlin*

```
//Singleton em Java
public class ClassicSingleton {
   private static ClassicSingleton instance = null;
   protected ClassicSingleton() {
      // Exists only to defeat instantiation.
   }
   public static ClassicSingleton getInstance() {
      if(instance == null) {
         instance = new ClassicSingleton();
      }
      return instance;
   }
   public void facaAlgumaCoisa()
   {
     //
   }
}

//Uso do singleton em Java
ClassicSingleton.getInstance().facaAlgumaCoisa()


//Singleton em Kotlin
public object ClassicSingleton {
    public fun facaAlgumaCoisa() {
    }
}

//Uso do singleton em Kotlin
ClassicSingleton.facaAlgumaCoisa()
```

Fonte:Autoria Própria

3.5.2 Inversão de controle

Conhecido pela sigla *IoC (Inversion of Control)* é um padrão de design que prega que ao invés de um programador determinar qual procedimento será executado, ele apenas determina quando esse procedimento será executado. Em outras palavras, uma classe utiliza código externo para executar uma ação, o que esta ação irá executar fica a cargo de outra classe.

As principais motivações para a utilização de inversão de controle são: o reaproveitamento do código que contém o fluxo de controle, o reaproveitamento do *design* e um baixo acoplamento entre os elementos da aplicação, tornando a mesma mais escalável e aumentando a facilidade na implementação de testes automáticos.

Existem alguns tipos de inversão de controle, porém o mais utilizado e conhecido é a injeção de dependência.

3.5.2.1 Injeção de dependência

O mais popular tipo de Inversão de controle é a Injeção de dependência (*DI*). A injeção de dependência consiste em tornar possível que uma determinada funcionalidade seja inserida em uma classe, sem que a mesma saiba como ela implementa as suas funcionalidades ou como ela foi criada (FOWLER, 2004).

Para tornar a explicação mais clara, será utilizado um trecho de código escrito em *Kotlin*. Primeiramente vê-se em um exemplo uma classe que deseja utilizar uma implementação de arquivos de *log* sem utilizar injeção de dependência (Figura 7), criando um objeto próprio:

Figura 7 – Exemplo de classe sem Injeção de Dependência

```
class ExemploDI {
    var logger: Logger = null
    constructor() {
        //Cria uma nova instance de logger
        logger = Logger("log.txt")
        logger.debug("Construtor OK")
    }
}
```

Fonte:Autoria Própria

Como qualquer *design pattern* a injeção de dependência não é obrigatória para o desenvolvimento, porém se veem problemas que normalmente acontecem. Como exemplo, se muda o *contrutor* na classe "*Logger*", ou utilizar outra classe de *log* na classe "ExemploDI", precisaria-se alterar todas as chamadas de *"Logger"* contidas no programa. A Figura 8 apresenta como é a implementação da classe "ExemploDI" utilizando o padrão de injeção de dependência.

Figura 8 – Exemplo de classe com Injeção de Dependência

```
class ExemploDI {
    constructor(var logger: Logger) {
        logger.debug("Construtor OK")
    }
}
```

Fonte:Autoria Própria

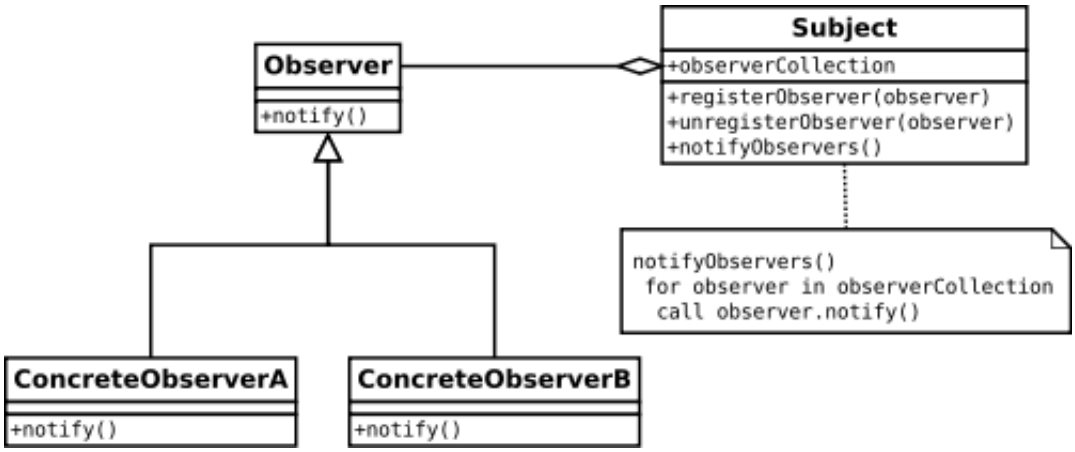
Observa-se que se continua utilizando a classe "*Logger*" da mesma forma, para inserir uma mensagem de *debug* no arquivo. Com isso, a utilização da classe "ExemploDI" torna-se muito mais fácil. Quem for utilizá-la não precisa saber como o construtor de "*Logger*" funciona.

Obviamente esse exemplo apresentado seria ainda interessante utilizando um protocolo, ou interface para a classe "*Logger*", porém o objetivo era mostrar como utilizamos a injeção de dependência, e não esse outro *design pattern*.

3.5.3 Observáveis

Um observável é um padrão de *design* que define uma dependência um para muitos entre os objetos, de modo que quando um objeto muda o estado, todos os seus dependentes são notificados e atualizados automaticamente. O padrão também pode se chamado de *Publisher-Subscriber, Event Generator e Dependents*. Os observáveis possuem três componentes principais, o sujeito, o observável e o observador (HUMBERTO, 2017). Pode-se verificar na figura 9 o diagrama UML que representa todos os elementos básicos de um objeto observável.

Figura 9 – Diagrama UML do padrão Observável



Fonte: (NTT.CC, 2009)

Para criar uma analogia da utilização deste padrão, primeiramente deve-se pensar como funciona uma editora de jornal (sujeito). Ela possui uma lista de clientes (observadores). Quando sai uma nova publicação (observável) todos os observadores recebem uma cópia do jornal, sendo que a qualquer momento um novo cliente pode se registrar, ou cancelar o recebimento do jornal. A partir deste exemplo, observam-se as funções de cada módulo do padrão utilizando observáveis:

- Sujeito - Objeto que possui os observáveis;
- Observável - Objeto que deve informar caso um elemento tenha sido alterado;
- Observadores - Objetos que são informados à cada mudança no elemento a ser observado;

Pode-se notar que o padrão dos observáveis assemelham-se muito a um conceito já visto anteriormente, quando abordada a programação reativa. O uso de observáveis não quer dizer que está sendo usado o conceito de programação reativa, porém a programação reativa faz uso deste padrão de *design* para a sua implementação.

3.6 TESTES DE *SOFTWARE*

Teste de *software* é o processo de execução de um produto para determinar se ele atingiu suas especificações, e se funcionou corretamente no ambiente para o qual foi projetado. O seu objetivo, é revelar falhas em um produto para que as mesmas possam ser corrigidas o mais rápido possível. No desenvolvimento de *software*, toda falha é gerada por erro humano e, apesar do uso das melhores técnicas, os erros permanecem presentes nos códigos desenvolvidos pelos mais experientes profissionais. Para minimizar a quantidade e gravidade destas falhas existem os testes. São diversos os tipos de testes de *software*. Será utilizada a divisão proposta pelo livro "Qualidade de Software – Teoria e Prática" (ROCHA, 2001) para explicar cada um deles:

- Teste de unidade: também conhecido como testes unitários. Tem por objetivo explorar a menor unidade do projeto, procurando falhas ocasionadas por defeitos de lógica e de implementação em cada módulo separadamente. O universo deste tipo de teste são os métodos dos objetos ou mesmo pequenos trechos de código. Este tipo de teste é o mais utilizado em metodologias de desenvolvimentos ágeis voltadas a testes como *TDD* (desenvolvimento voltado a testes).

- Teste de Integração: visa provocar falhas associadas as interfaces entre os módulos, quando estes são integrados para construir a estrutura do *software*. No mundo do desenvolvimento móvel, estes são conhecidos como testes funcionais, em que testam funcionalidades do sistema.

- Teste de Sistema: é o teste que visa reproduzir falhas utilizando o sistema da mesma forma que o usuário final. Dessa maneira, os testes são executados nos mesmos ambientes, condições e dados de entrada que um usuário iria utilizar no dia-a-dia. Em sistemas mais modernos, pode-se automatizar os testes de sistemas, com conhecidos testes de interface gráfica, ou *UI*.

- Teste de Aceitação: é a última etapa das validações, em que são realizados testes com um determinado grupo de usuários reais utilizando a aplicação. Este é o único procedimento apresentado que não pode ser automatizado.

- Teste de Regressão: este não é um nível de teste, e sim uma rotina de aplicar todos os testes já descritos, a cada nova versão de *software* liberada, a fim de que antigas funcionalidades não apresentem erros devido às novas implementações.

4 DESENVOLVIMENTO

Neste capítulo é apresentado o desenvolvimento do aplicativo para adoção de animais trazendo seus requisitos, características, arquitetura e funcionamento.

O nome escolhido para a aplicação é *PetFunding*. Do inglês *Pet* significa animal de estimação e *funding* quer dizer financiamento. Inicialmente o projeto iria ser desenvolvido apenas como uma forma das pessoas adotarem os animais, porém foram observados dois outros fatores. O primeiro é que a grande demanda das instituições de proteção animal, além de encontrar adotantes, é conseguir ajuda financeira para manter os animais acolhidos. Outro fator é, que muitas pessoas dispostas a ajudar, não podem adotar um animal por motivos diversos: por já terem animais em casa, não possuírem espaço e outros. Dessa forma, inclui-se no sistema de adoção um botão para que as pessoas possam fazer doações voluntárias para as instituições que acolherão os animais.

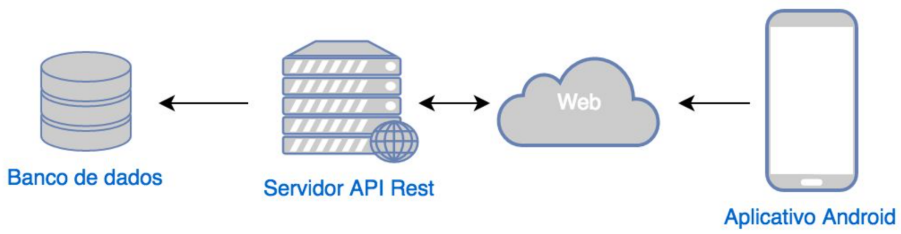
Outro motivo para a escolha do nome, é a similaridade com o termo muito popular *crowdfunding*, utilizado para financiamentos coletivos em geral.

Esta primeira versão será dedicada no propósito inicial de adoção dos animais, porém em implementações futuras, se quer adicionar a funcionalidade de doação para as instituições ou lares temporários. Desta forma os mesmos terão mais recursos para manter o seu trabalho.

4.1 CARACTERÍSTICAS DO PROJETO

Inicialmente foi definido que o projeto seria desenvolvido para a plataforma *mobile Android*. Foi arquitetado o fluxo de compartilhamento dos dados, para que os usuários possam cadastrar ou buscar animais, de acordo com as suas preferências. Para ocorrer esse compartilhamento, o projeto precisa ter um aplicativo móvel, e um servidor *web* que reunirá todas as informações dos usuários e dos animais. Na Figura 10 uma visão geral dos componentes da solução.

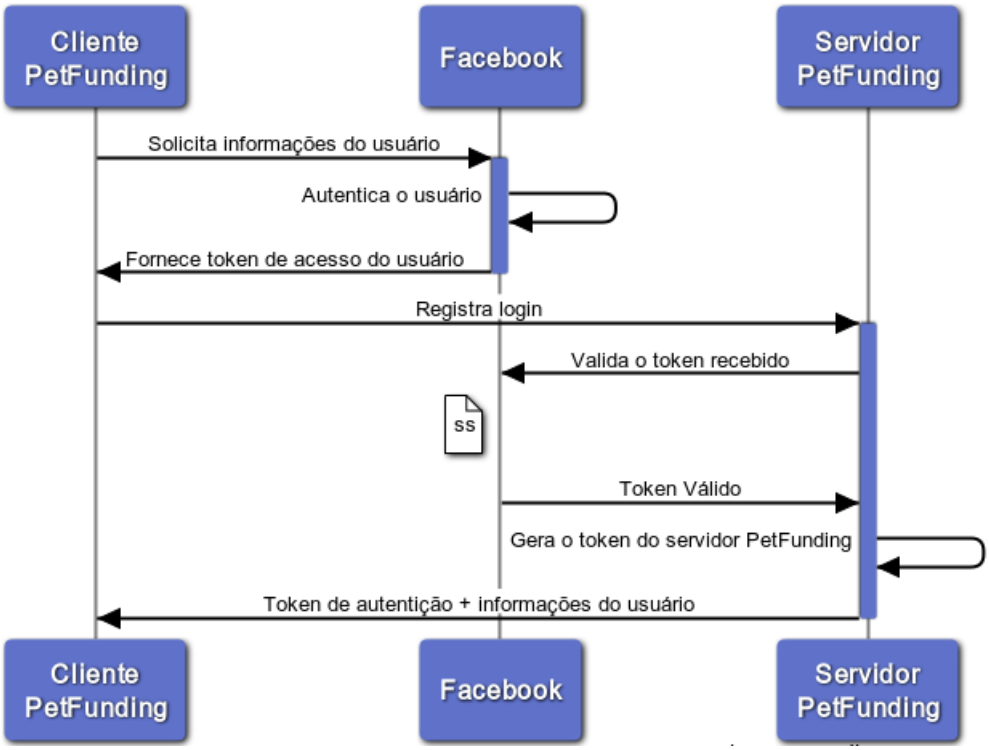
Figura 10 – Estrutura da solução



Fonte: Autoria própria

Por questões de segurança, todos os usuários do sistema deverão realizar um *login* e preencher um cadastro prévio no aplicativo. Foi observado que, utilizando um *login* pela rede social *Facebook* para autenticar os usuários, tem-se uma maior facilidade tanto para o usuário realizá-lo, como para o aplicativo obter as informações necessárias. Dessa forma, optou-se por realizar o *login* exclusivamente pelo *Facebook* nessa primeira versão do aplicativo. O fluxo desta funcionalidade é realizado da seguinte forma:

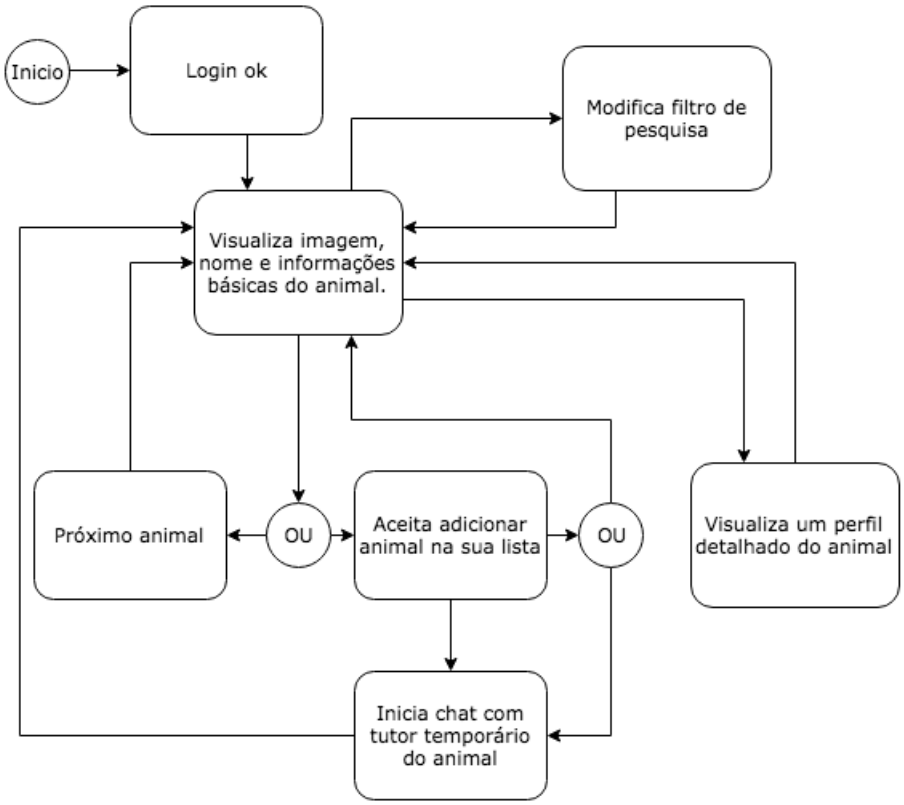
Figura 11 – Estrutura da solução



Fonte: Autoria própria

Uma forma muito interessante de apresentar opções, utilizada por aplicativos de relacionamento, é mostrar cada opção de uma vez, permitindo selecioná-la para obter mais informações. Pode-se ainda aceitar a opção deslizando para a direita, ou obter uma nova opção deslizando para a esquerda. Adaptado para a realidade de adoção de animais, o fluxo básico do usuário comum pode ser definido pela Figura 12.

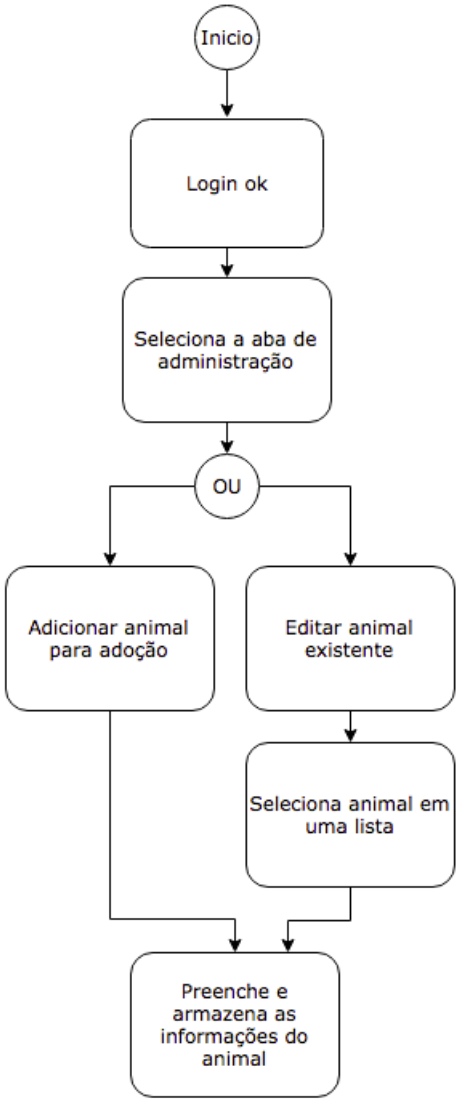
Figura 12 – Fluxo de ações do usuário básico



Fonte: Autoria própria

Como definição do projeto, somente os usuários cadastrados como lares temporários e ONGs, validados pelo sistema, poderão incluir novos animais para adoção. Dessa forma, após a etapa de login, os usuários com permissão para incluir animais terão essa opção extra, que seguirá o fluxo na Figura 13.

Figura 13 – Fluxo de ações para adicionar e editar animais



Fonte: Autoria própria

Por fim, o ponto que liga os adotantes aos tutores temporários dos animais é um *chat*, em que os dois podem tirar maiores dúvidas e marcar um encontro para poder oficializar a adoção. Dessa forma, tem-se reunidas todas as etapas do processo de adoção em um lugar só, facilitando a gestão das ONGs e aumentando as chances de quem procura, encontrar o animal mais adequado para as suas condições.

4.2 REQUISITOS

Com as funcionalidades mapeadas, deverão ser coletados os requisitos funcionais e não funcionais do sistema.

Os requisitos são objetivos, ou restrições estabelecidas por clientes e usuários para definir as propriedades do sistema (FILHO, 2000) ou algo que ele deverá estar apto a realizar para alcançar os objetivos (AVILA; SPINOLA, 2008)

4.2.1 Requisitos Funcionais

Os requisitos de funcionalidades definidos para uma aplicação são exigências mínimas que a mesma deve atender após a sua implementação. Para este projeto, definiu-se que os requisitos funcionais são:

- RF01: A aplicação deve ser capaz de rodar em dispositivos móveis.
- RF02: A aplicação móvel deve ser capaz de receber informações de um servidor na *web*.
- RF03: A aplicação para *smartphone* deverá mostrar fotos de animais abandonados.
- RF04: A aplicação deverá permitir que um grupo de usuários previamente selecionados adicionem novos animais à serem adotados.
- RF05: A aplicação deverá permitir a comunicação entre um usuário interessado em adotar os animais, e um usuário responsável pela inclusão do cadastro do animal.
- RF06: A aplicação deverá permitir que os usuários selecionem animais com interesse em adotar.
- RF07: A aplicação deverá permitir que os usuários visualizem os animais selecionados anteriormente por eles.
- RF08: A aplicação deverá permitir que o usuário, responsável pelo cadastro do animal para adoção, edite o cadastros do mesmo.
- RF09: A aplicação deverá realizar *login* e coletar as informações dos usuários.
- RF10: A aplicação deverá enviar notificações para os usuários responsáveis pelos animais, quando os mesmos receberem mensagens no *chat*.
- RF11: O sistema deverá possuir integração com a rede social *Facebook*.

4.2.2 Requisitos Não Funcionais

Algumas exigências para o desenvolvimento de uma aplicação não estão diretamente ligadas as suas funcionalidades. Estas exigências podem ser chamadas de requisitos não funcionais. Para a aplicação desenvolvida neste trabalho, os requisitos não funcionais são:

- RNF01: O sistema deve ser desenvolvido utilizando o SDK *Android*.
- RNF02: O sistema deverá utilizar a linguagem de programação *Kotlin*.
- RNF03: O sistema deverá ser capaz de funcionar a partir da versão 4.0 do sistema operacional *Android*.
- RNF04: A comunicação entre o servidor e o aplicativo deve ser criptografada.
- RNF05: Os animais cadastrados deverão possuir no mínimo uma foto.
- RNF06: Os dados coletados dos animais devem ser flexíveis e podem ser utilizados para qualquer tipo de animal.

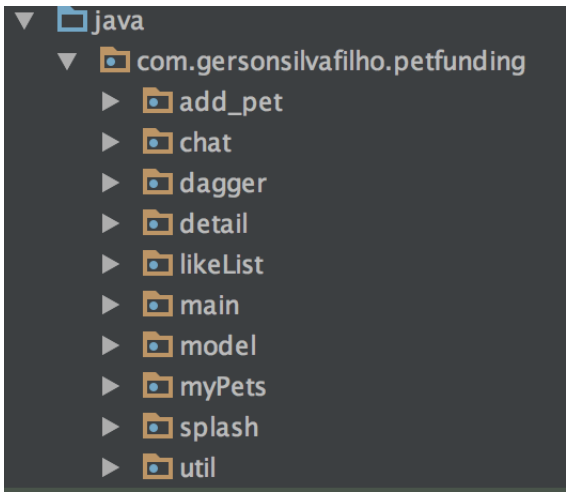
4.3 ARQUITETURA

O projeto utilizará vários padrões tratados anteriormente. O padrão de arquitetura é uma variação do padrão MVP, em que ocorre grande desacoplamento entre as classes.

4.3.1 Organização de Arquivos

Primeiramente ao criar um projeto na IDE *Android Studio*, deve-se pensar como organizar os pacotes e pastas para manter a organização do projeto. Utilizando a abordagem da página do *Google* (ANDROID, 2016), cada pasta ou pacote dentro do projeto será relacionada a uma funcionalidade do sistema. Dessa forma, fica mais fácil identificar onde cada elemento se encontra, e ainda, adicionar ou remover funcionalidades do sistema.

Figura 14 – Organização das pastas do sistema

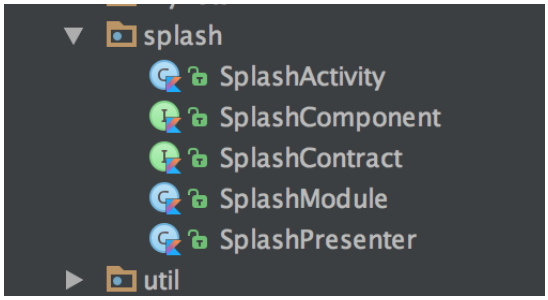


Fonte: Autoria própria

Pode-se observar na Figura 15 que, além das funcionalidades do sistema, têm-se 3 pastas adicionais. A pasta *"dagger"* refere-se ao *framework* de injeção de dependência utilizado, que precisa de algumas classes para configurar o projeto. A pasta *"model"* refere-se a camada com o mesmo nome no padrão MVP, e contém todos os objetos modelo, e as classes de comunicação com banco de dados e servidor web. Por fim, a pasta *"util"* é composta por algumas classes criadas para facilitar o desenvolvimento, e não estão ligadas à uma funcionalidade específica.

Dentro de cada pasta, se observa alguns arquivos base para as funcionalidades. A funcionalidade mais básica, e que possui um padrão replicável de arquivos, é a *"splash"*, que é a tela inicial do aplicativo.

Figura 15 – Arquivos da pasta "splash"

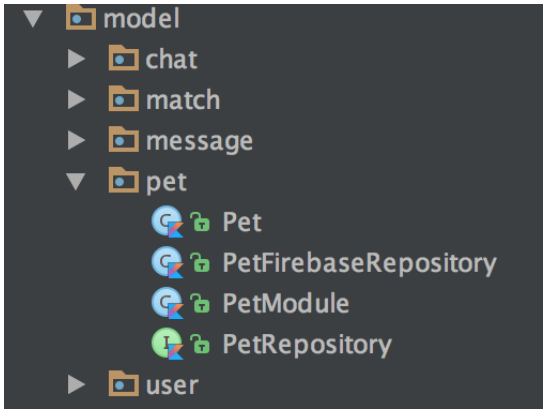


Fonte: Autoria própria

Trazendo para o modelo de aplicação MVP, cada funcionalidade possui sua *view* e seu *presenter*. Além deles, temos outros 3 arquivos, o *"contract"* que é uma interface que serve de modelo para a *"view"* e para o *"presenter"*, o *"module"* que é o módulo que define quais objetos o *framework dagger* irá injetar nas dependências, e o *"component"*, que define o escopo da aplicação para as dependências injetadas. A explicação mais detalhada sobre como funciona o *framework Dagger* e a injeção de dependência utilizada serão abordados mais a frente.

Outro ponto interessante de se ressaltar na organização dos arquivos é a pasta *"model"*. Esta pasta está organizada por tipo de objeto, e cada objeto segue uma estrutura padrão como se observa na Figura 16.

Figura 16 – Arquivos da pasta "model"



Fonte: Autoria própria

Dentro da pasta *"pet"*, temos 4 arquivos que podem ser considerados como um padrão para as outras pastas de modelos. O arquivo *"Pet"* é a classe base para a criação de um objeto que representa um animal.

O arquivo *"PetRepository"* é mais um padrão conhecido no desenvolvimento de *software* em que se criam modelos de repositórios para buscar dados de uma fonte, seja ela *web* ou local. Esse arquivo é na verdade uma interface que define como deve ser implementado um repositório de *"Pets"*. A implementação dele, baseada nas chamadas do servidor utilizado, chamamos de *"PetFirebaseRepository"*.

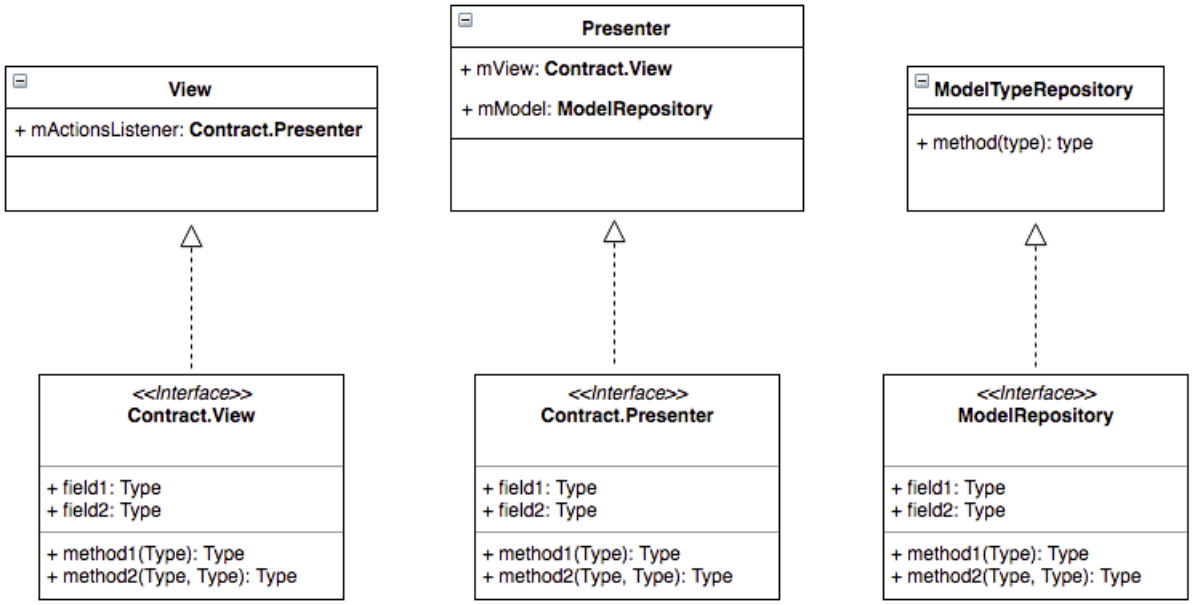
O arquivo *"PetModule"*, como na explicação das pastas de funcionalidades, está se referindo a uma configuração do *framework* de injeção de dependência, e sinaliza qual é o repositório a ser passado quando for injetada uma dependência do tipo *"PetRepository"*.

4.3.2 Modificação no Padrão MVP

O padrão MVP, como já foi exposto, diminui o nível de coesão das classes, tornando assim, mais fácil de realizar testes e modificar o *software*. Porém, ainda se tem uma certa dependência entre as classes, pois uma deve importar a outra para poder utilizar os seus métodos.

Para evitar esse semi-acoplamento, e tornar ainda melhor a arquitetura do padrão MVP, é adicionado um elemento no meio das relações, que chama-se contrato. Esse contrato, nada mais é que uma interface que contém tudo que aquela classe precisa para ser do tipo desejado. Então, para utilizar uma classe "A", a classe "B"não precisa importar a classe "A", basta importar sua interface e chamar os métodos padrão da mesma.

Figura 17 – Modificação no padrão MVP



Fonte: Autoria própria

Com a utilização de interfaces, ganham-se mais duas peças importantes na garantia de qualidade de *software*. Primeiro, pode-se substituir facilmente qualquer componente descrito por uma interface, pois se o novo componente implementar seus métodos, ele funcionará com o resto do código.

Outro ponto positivo para o uso de interfaces, é a facilidade com que se criam testes. Imaginando a situação acima, criam-se testes unitários para uma função que utilize o repositório, que comunica com a *web*. Com a interface pode-se criar um objeto simulado, muitas vezes chamado "*mockup*"para chamar a função e obter uma resposta controlada. Este modelo de desenvolvimento permite inúmeras melhorias, e variações de código aumentando sua escalabilidade e facilidade de manutenção.

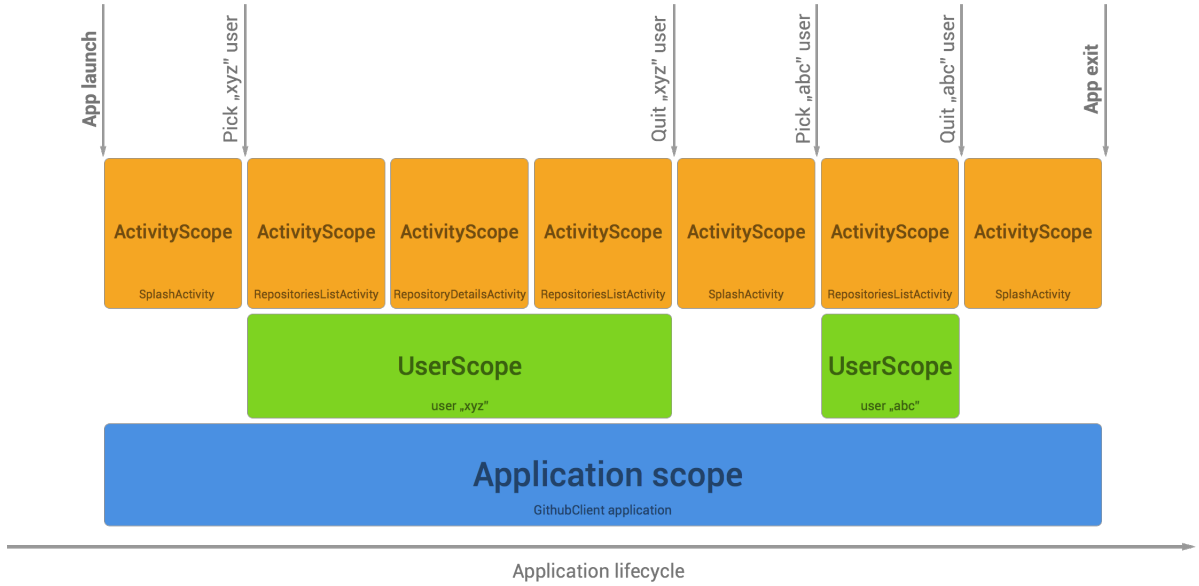
4.3.3 Injeção de dependência

Para implementar a Injeção de dependência utilizou-se o *framework* mais popular para essa finalidade no desenvolvimento *Android*, o *Dagger* em sua versão 2. A utilização do *framework* não é fundamental para a implementação da DI, porém um argumento que pesou para a sua utilização foi o auxílio na criação de escopos na aplicação.

A injeção de dependência tem diversas vantagens que já foram tratadas, mas uma desvantagem é a manutenção de cada dependência em seu devido lugar, durante a execução do código. Um aplicativo *Android* tem diversos escopos, e seu uso pode trazer confusão ao desenvolvimento.

A aplicação utiliza a mesma divisão de escopos proposta por (STANEK, 2015), que apresenta a estrutura com 3 escopos principais. Um escopo de aplicação global, um escopo de usuário e um escopo de tela ("*view*"). Cada escopo tem um ciclo de vida diferente, que está explicado na figura 18.

Figura 18 – Ciclo de vida dos escopos



Fonte: (STANEK, 2015)

- Escopo de aplicação - É o escopo global. Inicia quando o usuário abre a aplicação e encerra quando o mesmo fecha o *app*. Para o aplicativo *PetFunding* os componentes globais são o acesso as API do servidor, e algumas variáveis do sistema.
- Escopo de usuário - Este escopo inicia quando o usuário faz o *login* na plataforma, e encerra quando o mesmo faz *logout*. Os principais recursos compartilhados por esse escopo são as informações do usuário ativo no momento.
- Escopo de Tela - Cada tela no *Android* tem um escopo próprio. Este escopo muda dependendo de cada tela, e muitas vezes não possui nenhum recurso compartilhado. Também conhecido como *ActivityScope*, pois no desenvolvimento *Android* uma *Activity* é o elemento que controla a "*View*".

4.3.4 Programação Reativa

No projeto foi utilizado o *framework RxKotlin*, que já traz a implementação de todos os componentes da programação reativa, com exigências do sistema operacional *Android*, e ainda utilizando todos os recursos da linguagem *Kotlin*. A programação reativa foi utilizada de diferentes formas durante o desenvolvimento.

Primeiramente, utiliza-se para realizar chamadas ao servidor, devido a natureza assíncrona das mesmas, e a facilidade de tratamento de erros da programação reativa. A função apresentada na Figura 19 é um exemplo de como foi utilizada a programação reativa, expondo uma chamada assíncrona de conexão com o *Facebook*.

Figura 19 – Exemplo de chamada utilizando programação reativa

```
//Método que utiliza programação reativa
fun loginWithFacebook(token:String):Observable<Boolean> {
    val credential = FacebookAuthProvider.getCredential(token)
    return RxFirebaseAuth
        .signInWithCredential((mAuth), credential)
        .map { authResult -> authResult.user!= null }
        .toObservable()
}

//Chamada do metodo acima
mUserRepository.loginWithFacebook(token)
    .take(1)
    .subscribe{
        //Execute algo quando receber
    }
```

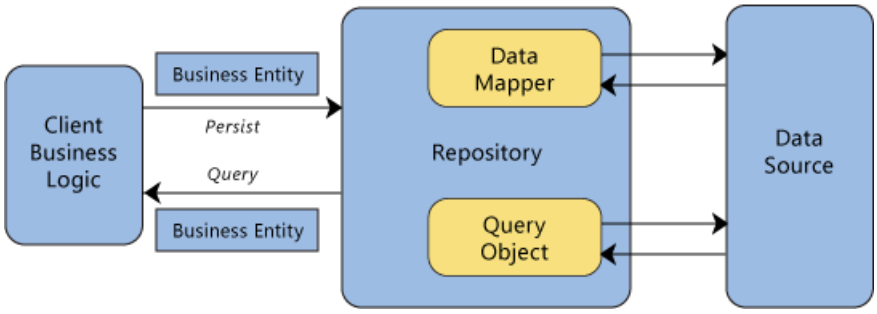
Fonte:Autoria Própria

Pode-se observar algumas vantagens da programação reativa. Primeiro, é a forma de tratar chamadas assíncronas, que é praticamente transparente utilizando funções lambda da linguagem *Kotlin*. Outro ponto nesse trecho de código são as funções "*map*"e "*take(1)*", que trazem outra vantagem da programação reativa, que pode aplicar operações sobre as chamadas ou sobre o fluxos de dados.

4.3.5 Repositórios

O padrão de repositórios vem ganhando popularidade desde que foi apresentado, no *Doman-Driven Design* em 2004. Ele é uma abstração dos dados que provém operações em uma base local, ou em um servidor remoto, a fim de separar a camada de processamento da camada de dados do sistema (*presenter* e *model*, respectivamente neste projeto). Um exemplo desta relação pode ser observada na Figura 20.

Figura 20 – Estrutura de um repositório



Fonte: (MSDN, 2016)

Outro fator muito importante para a utilização do conceito de repositórios, é o aumento da facilidade na criação de testes automáticos, com simulação de dados. Pode-se criar um repositório padrão com alguns dados simulados, visando assim, obter uma resposta conhecida a algumas funcionalidades do sistema. Desta forma, podem-se realizar tanto testes unitários, quanto testes de funcionalidades de forma automática.

Um exemplo desta utilização pode ser encontrada no projeto, e está descrito na Figura 21, onde tem-se a classe "*PetFirebaseRepository*" que busca um objeto do tipo "*Pet*" no servidor *web*.

Figura 21 – Exemplo classe de repositórios

```
class PetFirebaseRepository : PetRepository {
    override fun getPetFromKey(petId: String): Observable<Pet> {
        val key = petsRef.child(petId)
        return RxFirebaseDatabase
            .observeSingleValueEvent(key, Pet::class.java)
            .toObservable()
    }
}
```

Fonte:Autoria Própria

Pode-se criar uma classe "*PetMockRepository*" que simula o repositório com dados pré determinados. Na Figura 22, ao invés de buscarmos um novo "*Pet*" no servidor, retorna-se um observável criado internamente, com o nome "Nino".

Figura 22 – Exemplo de *mockup* da classe *PetRepository*

```
class PetMockRepository : PetRepository {
    override fun getPetFromKey(petId: String): Observable<Pet> {
        return Pet("Nino").toObservable()
    }
}
```

Fonte:Autoria Própria

4.3.6 Testes automáticos

Um grande motivo para a utilização dos padrões de engenharia de *software*, no desenvolvimento profissional, é a melhoria das condições para a criação de testes automatizados. Neste projeto, cada classe criada foi pensada a fim de que pudesse ser testada, tornando assim o código mais confiável e seguro.

Podem-se dividir os testes criados em testes unitários e testes de funcionalidade. A plataforma de desenvolvimento do *Android* já possui camadas pré-determinadas para estes dois tipos de testes.

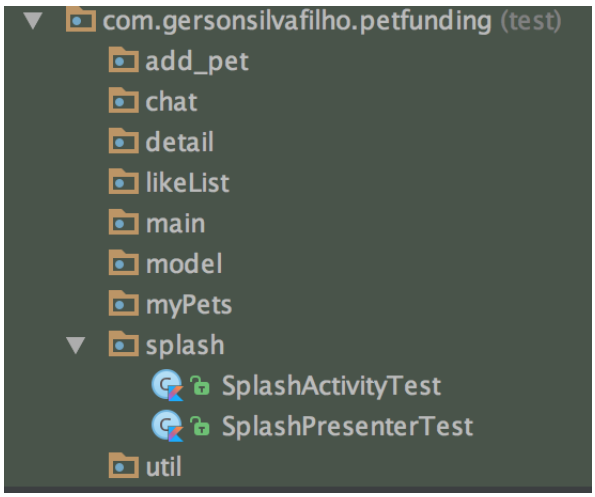
4.3.6.1 Testes unitários

Os testes unitários são testes das funções das classes do sistema. É um teste de um trecho curto do código. Este tipo de teste é muito importante para os "*Presenters*" do sistema. Um método de fazê-los mais testáveis, na plataformas *Android*, é torná-los independentes de qualquer componente da plataforma, ou até de bibliotecas externas.

Os componentes utilizados, são os oficiais sugeridos pela *Google* para testes unitários no *Android*. São eles o "*JUnit*", que provém uma API com várias funcionalidades para avaliar os testes, e o "*Mokito*" que é uma biblioteca que facilita na criação de objetos simulados para os testes.

A estrutura da pasta de testes unitários se assemelha a estrutura de pastas do projeto principal. Dentro de cada pasta temos duas classes, uma que testa o "*presenter*" e outra que testa o a "*view*".

Figura 23 – Estrutura de pastas de testes unitários



Fonte: Autoria própria

Um exemplo de teste unitário criado para a classe "*SplashPresenter*" está na classe "*SplashPresenterTest*" (Figura 24). Para criar um teste, utiliza-se a biblioteca *Mokito* para criar os objetos simulados, então chama-se o construtor do *presenter* a partir deles.

Figura 24 – Exemplo da inicialização dos testes

```
class SplashPresenterTest{
     @Before
     fun init()
     {    //Inicializa os objetos simulados
          mockView = mock(SplashContract.View::class.java)
          mockUserRepository = mock(UserRepository::class.java)
          //Inicializa o presenter a ser testado
          presenter = SplashPresenter(mockView,mockUserRepository)
     }
 }
```

Fonte:Autoria Própria

Em seguida, criou-se um teste para a função que executa um trecho de código quando o *login* realizado, no *Facebook*, retorna com sucesso. O teste, que está na Figura 25, se resume a chamar a função no "*presenter*" a ser testado, e em seguida verificar se a função da "*view*" que está dentro dele foi invocada apenas uma vez.

Figura 25 – Exemplo de um teste unitário

```
@Test
fun testOkFacebook() {
    presenter.facebookSuccess()
    Mockito.verify(mockView, times(1)).facebookSuccess()
}
```

Fonte:Autoria Própria

4.3.6.2 Testes de Funcionalidade

A principal diferença entre os testes de funcionalidade, e dos unitários, é que o primeiro é executado em um dispositivo, seja ele físico ou virtual, e testa as funcionalidades do sistema. Dentro desse tipo de teste encontram-se os testes de *UI*, ou interface gráfica, que executam ações na tela do dispositivo e verificam o retorno gráfico das mesmas.

Para criar testes de funcionalidade utiliza-se a biblioteca "*Espresso*", que também é indicada pela *Google* para a realização desse tipo de validação. O trecho, presente na figura 26, mostra como é realizado o teste que simula o clique do botão de *login* com o *Facebook*, na tela inicial do aplicativo. Em seguida, é utilizada a tela de menu para clicar no botão *logout*. Tudo isso respeitando os tempos que estas funcionalidades levam para serem executadas.

Figura 26 – Exemplo de um teste de UI

```
public class SplashActivityTest {
    @Test
    public void loginLogoutTest() {
        ViewInteraction loginButton = onView(
                allOf(withId(R.id.fbLoginButton),
                withText("Continue with Facebook"),
                isDisplayed()))
        loginButton.perform(click())

        sleep(4000)

        ViewInteraction appCompatImageButton = onView(
                allOf(
                withContentDescription("Open navigation drawer"),
                        withParent(withId(R.id.toolbar)),
                        isDisplayed()));
        appCompatImageButton.perform(click());

        sleep(2000)

        ViewInteraction appCompatCheckedTextView = onView(
                allOf(withId(R.id.design_menu_item_text),
                withText("Sair"),
                isDisplayed()));
        appCompatCheckedTextView.perform(click());
    }
}
```

Fonte:Autoria Própria

O teste será considerado como "sucesso" quando estes comandos forem executados, respeitando os tempos contidos no código, e todas as telas aparecerem como planejado. Caso contrário, o teste retornará como "erro".

4.4 FUNCIONAMENTO DA APLICAÇÃO

Nesta seção será demonstrado o funcionamento do aplicativo desenvolvido, utilizando as telas do mesmo, a fim de dar um melhor panorama sobre a aplicação.

4.4.1 Tela de *login*

Nesta tela, observada na Figura 11, é mostrado o ícone do aplicativo, e o botão de *login* com o *Facebook*. A única opção no momento para entrar na aplicação é apertando o botão de *login*. Caso ocorra algum erro, uma mensagem é apresentado ao usuário. Esta tela não é mostrada

caso o usuário já tenha realizado o *login* anteriormente.

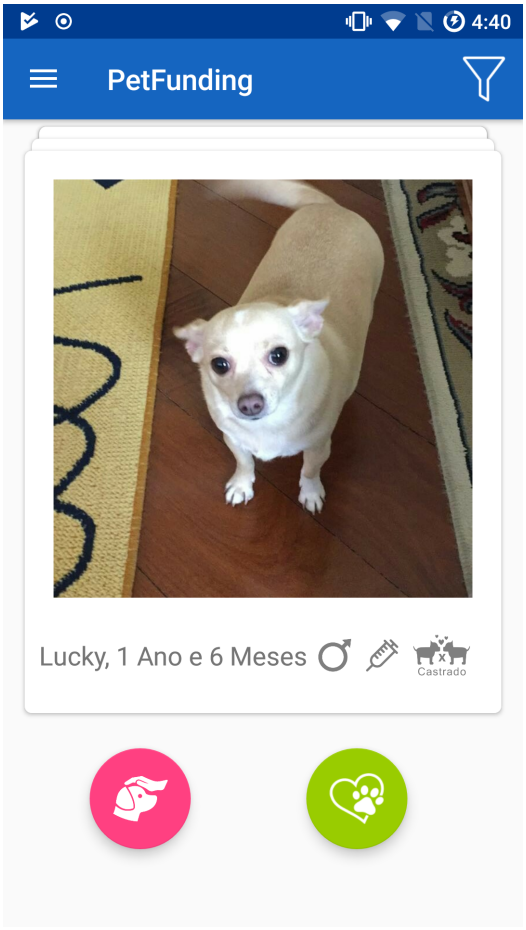
Figura 27 – Tela de *Login*

Fonte: Autoria própria

4.4.1.1 Tela Principal

O menu principal é a tela base da aplicação (Figura 28). Por meio dela é possível acessar as funcionalidades do aplicativo. A principal funcionalidade, que é a de encontrar animais para adoção, é a primeira a ser mostrada para o usuário. Apertando no canto superior esquerdo podem-se observar as opções do menu, e apertando no canto superior direito, podem-se acessar os filtros dos animais a serem mostrados.

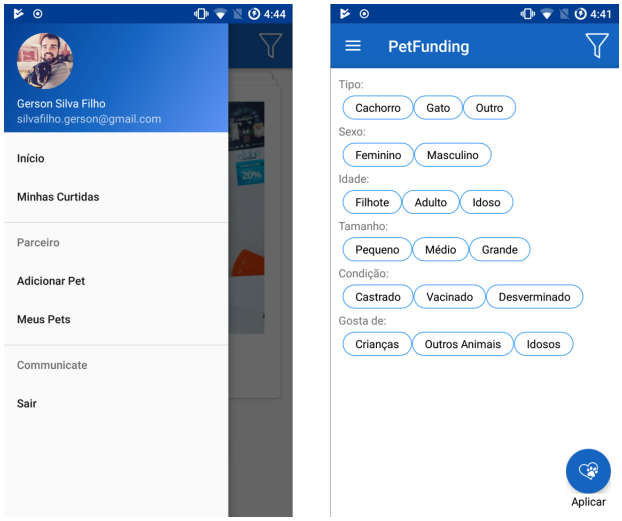
Figura 28 – Tela de Principal



Fonte: Autoria própria

Clicando no canto superior esquerdo, observam-se as opções do menu, e apertando no canto superior direito, podem-se acessar os filtros dos animais a serem mostrados (Figura 29).

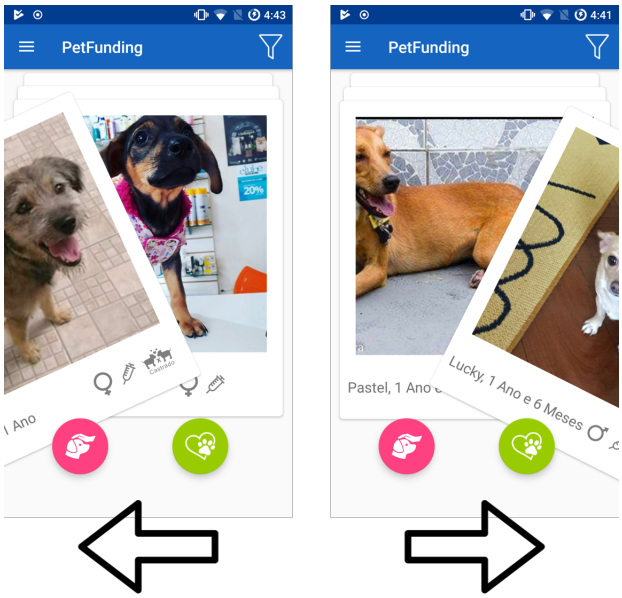
Figura 29 – Menu e filtros



Fonte: Autoria própria

O funcionamento da tela principal (Figura 30), para a seleção dos animais, é muito simples. Caso o usuário arraste o cartão para a direita, ele irá adicionar o animal na sua lista de animais que tem interesse, podendo iniciar uma conversa com o tutor do mesmo. Caso ele arraste para a esquerda, irá visualizar um novo animal.

Figura 30 – Menu e filtros

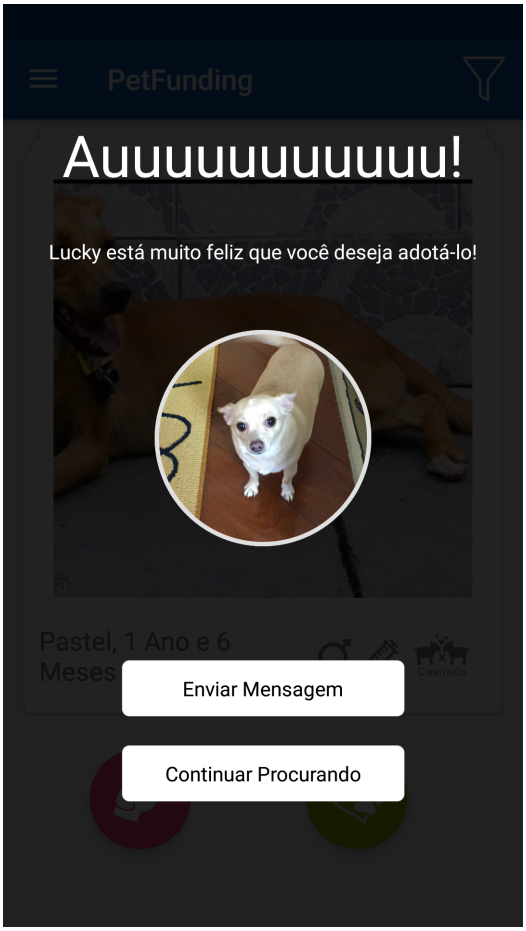


Fonte: Autoria própria

4.4.2 Tela de Seleção

A tela da Figura 31 é invocada quando o usuário adiciona qualquer *pet* na sua lista de curtidas. Ela é uma referência à tela dos aplicativos de relacionamento, onde dois usuários que gostaram um do outro, são informados que isto ocorreu.

Figura 31 – Tela de seleção

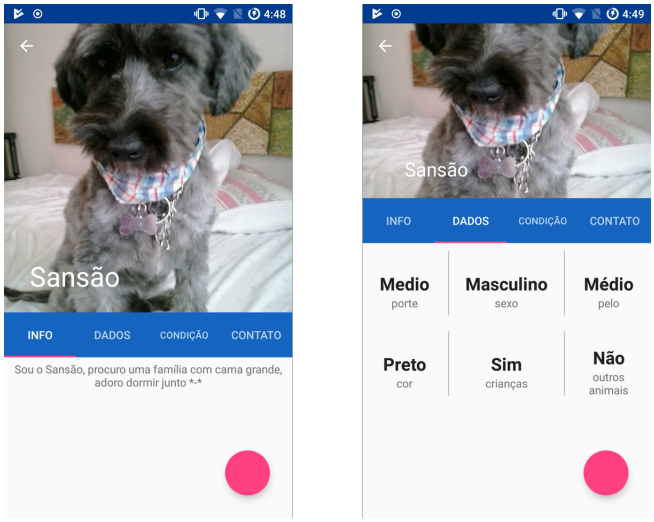


Fonte: Autoria própria

4.4.3 Detalhes do Animal

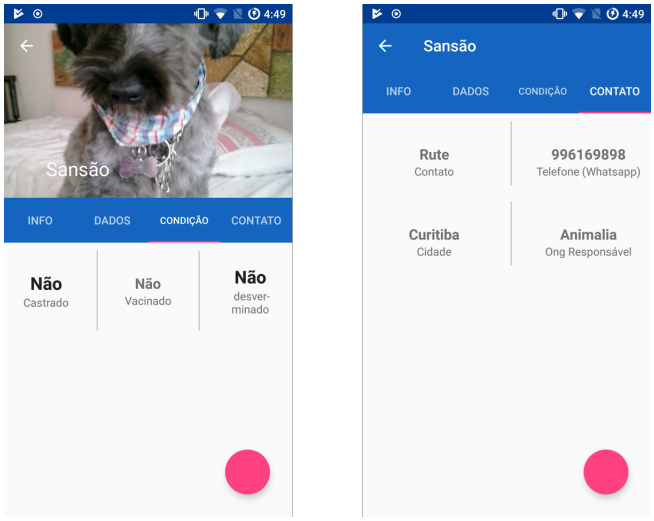
Quando o usuário seleciona a imagem do animal, ele é direcionado para a tela com detalhes do mesmo (Figuras 32 e 33). Os detalhes são as informações detalhadas, para a pessoa poder adotar o animal. São apresentadas diversas imagens do mesmo no topo da tela, e ainda o contato do responsável por ele, caso queira-se comunicar-se por outro meio.

Figura 32 – Detalhes do animal - Info e dados



Fonte: Autoria própria

Figura 33 – Detalhes do animal - Condição e contato

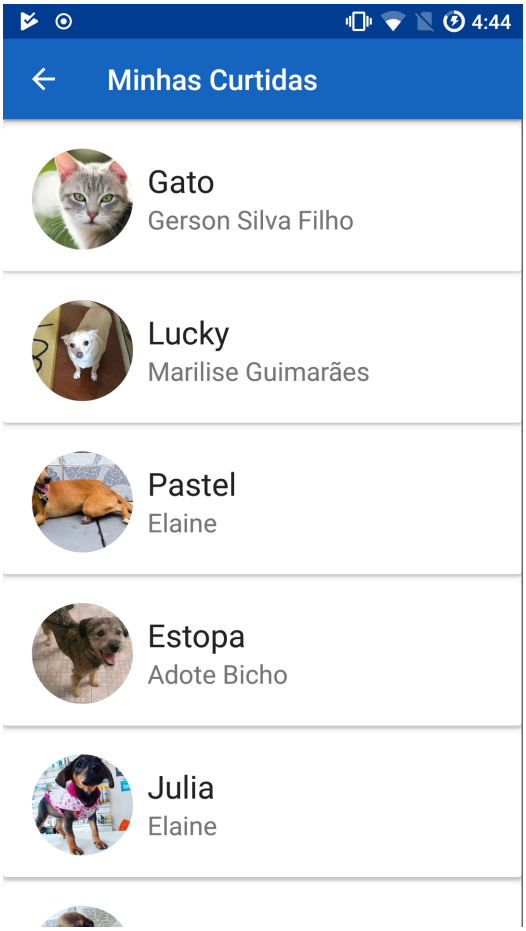


Fonte: Autoria própria

4.4.4 Minhas Curtidas

Na tela minhas curtidas (Figura 34) o usuário pode ver uma lista com todos os animais selecionados por ele. Pode-se também conferir seus perfis, e iniciar um chat com o tutor temporário, para agendar um encontro ou fazer perguntas sobre o animal.

Figura 34 – Tela Minhas Curtidas

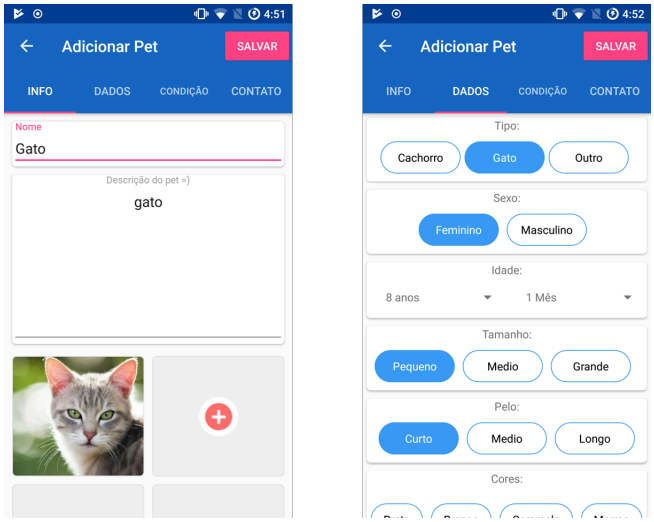


Fonte: Autoria própria

4.4.5 Adicionar *Pet*

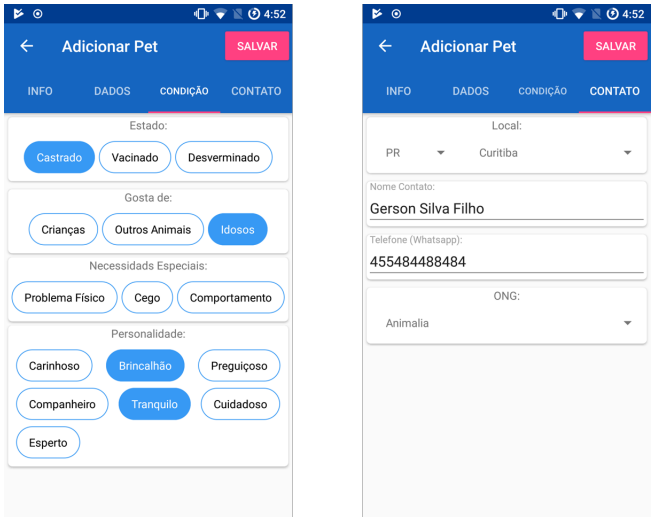
Esta opção só pode ser visualizada por usuários cadastrados como parceiros. Com ela, que pode ser observada na Figura 35, o usuário pode adicionar um novo animal para ser adotado. Todas as opções foram criadas para facilitar e agilizar a inclusão dos animais, como a idade aproximada, e as opções de fácil seleção. A tela foi dividida em abas para permitir um maior número de informações, e ainda, organizar melhor cada etapa do processo de inserção dos dados.

Figura 35 – Adicionar *Pet* - Info e Dados



Fonte: Autoria própria

Figura 36 – Adicionar *Pet* - Condição e Contato

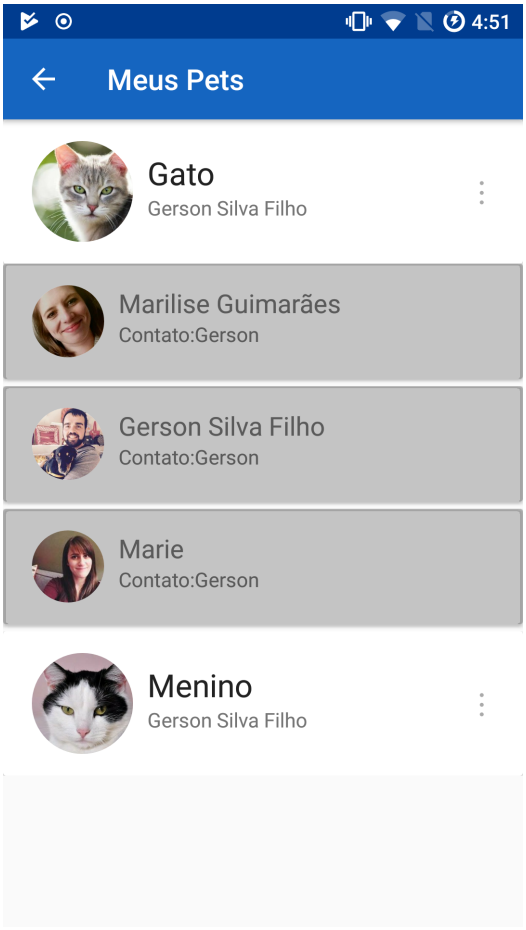


Fonte: Autoria própria

4.4.6 Meus *Pets*

Esta tela também só pode ser visualizada por usuários cadastrados como administradores. Nela, o usuário irá visualizar todos os animais que ele adicionou, e terá um resumo de todos os *chats* que teve com pessoas interessadas em adotá-los (Figura 37). Pode ainda selecionar o menu para visualizar e editar o cadastro do *pet*.

Figura 37 – Tela Meus *Pets*



Fonte: Autoria própria

5 GESTÃO DO PROJETO

Neste capítulo são apresentados o cronograma, o custo estimado para desenvolvimento e manutenção do projeto.

5.1 ESTIMATIVA DE HORAS

No início do projeto foi feita uma estimativa de atividades e horas a serem utilizadas no desenvolvimento de cada atividade. No Quadro 1, temos uma comparação entre as horas previstas e as utilizadas para a execução do projeto.

Observa-se que, apesar de alguma diferença entre as horas previstas e horas utilizadas, o erro ficou próximo de 10%, que é um valor aceitável para projetos desta natureza.

Quadro 1 – Estimativa de horas

Atividade	**Horas Previstas**	**Horas Utilizadas**
Estudo de viabilidade	10	10
Estudo das tecnologias	40	50
Criação das telas e *wireframe*	12	25
Modelagem dos objetos e relacionamentos	15	12
Criação da estrutura do projeto	20	15
Desenvolvimento - *Login*	10	10
Desenvolvimento - Adicionar *Pet*	15	18
Desenvolvimento - Tela Principal	30	28
Desenvolvimento - *Chat*	20	12
Desenvolvimento - Meus *Pets*	10	12
Desenvolvimento - Meus *Likes*	10	8
Desenvolvimento - Editar *Pet*	15	5
Desenvolvimento - Testes automáticos	30	35
Cadastro e criação do Servidor *Web*	30	25
Criação da automatização de build/testes	10	15
Inclusão de dados no aplicativo	5	10
Redigir a Monografia	50	80
Total	**332**	**370**

5.2 CUSTOS

Os custos para desenvolvimento do projeto podem ser divididos em duas categorias: os iniciais e os recorrentes. As despesas inciais são os valores referentes as horas de desenvolvimento, e custos básicos para criação do projeto. Pagamentos de forma recorrente serão feitos para manter a infraestrutura do aplicativo funcionando. Estes pagamentos são realizados mensalmente.

5.2.0.1 Custos inicias

O custo inicial é basicamente a quantidade de horas utilizadas, multiplicadas pelo valor cobrado pela hora de um programador *Android*. O valor da hora de desenvolvimento varia muito, e é relativo ao nível de conhecimento do desenvolvedor. Neste projeto iremos utilizar um valor de 80 reais por hora, valor médio praticado no mercado. Com este valor temos um custo de desenvolvimento de R$29.600,00.

Outro custo envolvido é o custo da publicação do aplicativo, que para a plataforma *Android* é realizado apenas uma vez por desenvolvedor, no valor de 40 dólares. Com a taxa de câmbio referente a junho de 2017 o valor em reais total é R$131,71.

5.2.0.2 Custos Recorrentes

Por fim, tem-se o custo para manter o servidor *web* funcionando. Este custo é minimizado pela utilização da tecnologia *Firebase* do *Google*. Nela, tem-se um servidor para até 100 usuários simultâneos gratuitamente. Caso se exceda esse número, pode-se adotar o plano "*Frame*"que não possui limitação de usuários, e custa 25 dólares mensais. Com a taxa de câmbio referente a junho de 2017 o valor em reais pode chegar a R$82,32.

6 RESULTADOS E DISCUSSÕES

6.1 RESULTADOS OBTIDOS

Avalia-se que este projeto trouxe bons resultados. Primeiramente do ponto de vista acadêmico, pois a proposta de trazer a linguagem *Kotlin*, para uma aplicação desenvolvida na universidade, já é uma mudança. Há de se ressaltar que durante o desenvolvimento do projeto a linguagem se tornou oficialmente suportada pela *Google*, trazendo maior visibilidade à mesma.

Outro fator de grande valia, foi a tentativa de aproximar um trabalho acadêmico do que se tem de mais eficiente, e de melhor qualidade no desenvolvimento *Android* nativo. Grandes empresas já utilizam a linguagem *Kotlin*, programação reativa, injeção de dependência, o padrão MVP e ainda um desenvolvimento muito voltado a testes automáticos. Muitas vezes observam-se trabalhos sobre esses temas de forma separada, devido a complexidade dos mesmos, porém a agregação dos assuntos, buscando visualizar um panorama mais amplo, é muito válida para fins de entender o desenvolvimento de *software* praticado no mercado.

Observando o caráter social do aplicativo, tem-se uma contribuição muito importante para auxiliar diversas ONGs que trabalham com a adoção de animais, possibilitando um canal de fácil controle e utilização, para quem busca um animal a ser adotado. O aplicativo inicial abriu portas para projetos futuros, não só voltados à adoção, mas também à contribuição para esses seres que merecem uma vida melhor.

Por fim, a natureza do aplicativo se torna quase uma paródia de aplicativos consagrados para relacionamentos, trazendo também a atenção da sociedade para a importância da adoção de animais, evitando o comércio e exploração dos mesmos, e ainda alertando sobre os problemas de saúde que a superpopulação pode trazer à vida dos humanos.

6.2 PROBLEMAS E DIFICULDADES ENCONTRADAS

Foram encontradas diversas dificuldades durante a pesquisa e o desenvolvimento dessa aplicação. Nenhuma causou muitos danos e todas puderam ser contornadas, porém é válida a reflexão para que futuros projetos não passem pelos mesmo problemas.

Uma dificuldade observada, foi a utilização do banco de dados da tecnologia *Firebase*. Para muitas aplicações ele torna-se uma forma fácil de persistir dados em um servidor *web*, porém pela estrutura do seu banco de dados ser não relacional, uma busca encadeada, ou ainda filtros com mais de um relacionamento, se tornam custosos a nível de execução da aplicação.

Outro problema encontrado foi a primeira inserção de dados. Por ser uma tarefa extensa de busca de animais e inserção no aplicativo, deveria ser feita por uma plataforma *web*, ao invés

de ser pelo celular. Os dados coletados dos animais estavam na *internet*, tanto imagens quanto informações, sendo essa inserção de dados em massa se torna quase inviável de ser feita pelo aplicativo.

7 CONCLUSÃO

A adoção de animais é um assunto que tem atraído muita visibilidade nos últimos anos, talvez pelas campanhas vistas, ou pelo crescimento da divulgação nas redes sociais. Apesar disso, continuam-se observando números alarmantes de animais abandonados por donos irresponsáveis, o aumento de animais nas ruas, devido à sua rápida reprodução, e ainda um grande mercado que trabalha com a reprodução de animais, explorando os mesmos para obter o lucro a qualquer custo.

O trabalho de educação e conscientização da população deve ser contínuo, e as formas de divulgar essas iniciativas devem sempre ser atualizados com o passar do tempo.

Por meio do aplicativo desenvolvido neste projeto, é possível às ONGs cadastradas incluir animais disponíveis para adoção, e ainda realizar uma gestão mais eficiente da comunicação e avaliação dos adotantes. A adoção é uma via de mão dupla, sendo que tanto os adotantes, quanto os animais adotados, devem ter suas necessidades respeitadas, a fim de garantir uma melhor adaptação e qualidade de vida a ambos.

Pelo lado dos adotantes a aplicação é muito simples e intuitiva. Pode-se filtrar animais pelas suas características, facilitando assim a procura pelos mais adequados para cada família. O aplicativo tem como fundamento não buscar um animal pela raça, tanto que não temos essa opção no cadastro ou na pesquisa. A pesquisa é realizada pelas características globais do animal, colocando de lado um apelo capitalista, que incentiva um mercado que contribui para a superpopulação de animais nas ruas.

Do ponto de vista acadêmico, foi um grande desafio trazer uma proposta completa de desenvolvimento de aplicativo comercial. Como já exposto anteriormente, são raros os trabalhos que trazem tantos assuntos misturados, porém essa é a realidade do desenvolvimento de *software* atualmente. A utilização de *frameworks*, novas linguagens de programação, e bibliotecas de terceiros é comum no mercado, e deve ser incentivada nos meios acadêmicos a fim de se pensar mais nos novos desafios, e não em refazer os pontos já abordados por outros.

Espera-se que este trabalho possa demonstrar as vantagens de se utilizar as diversas técnicas de engenharia de *software*, que muitas vezes são estudadas de forma separada. Estas técnicas, mesmo parecendo inúteis em projetos menores, são utilizadas no mercado em larga escala, proporcionando uma maior qualidade e facilidade na manutenção dos *softwares*.

Todo o desenvolvimento do aplicativo e o seu código fonte está disponibilizado de forma aberta na plataforma *Github*. Espera-se continuar o desenvolvimento e expansão do projeto contando com o apoio de desenvolvedores cívicos e de simpatizantes das causas dos animais. Desta forma, pode-se esperar que dentro de alguns anos será vista a diminuição do número de animais abandonados nas ruas, e ainda o aumento do número pessoas interessadas na adoção, ao invés da compra de animais.

7.1 DESENVOLVIMENTOS FUTUROS

Como todo *software*, esta aplicação não tem limites para ser aperfeiçoada. Algumas demandas inclusive foram encontradas durante o desenvolvimento, porém não puderam ser incluídas devido ao prazo estabelecido para o projeto.

Como tratado anteriormente, foi observada a enorme demanda por uma plataforma para a doação de recursos exclusiva para campanhas de resgate animal. Este será o ponto principal de futuros desenvolvimentos e melhorias desta plataforma.

Outra demanda já mapeada é a criação de uma aplicação para a plataforma iOS, para atingir um maior número de usuários e assim abranger quase todos os dispositivos móveis.

Pequenas correções no aplicativo ainda serão feitas antes da sua publicação, e ainda, será realizado um trabalho extensivo para o aumento da cobertura de testes do aplicativo.

REFERÊNCIAS

2, G. *Kotlin and Android*. 2017. Disponivel em:<https://developer.android.com/kotlin/index.html>. Acessado em 10/06/2017.

ALMEIDA, E. H. de P. **Maus tratos contra animais**. 2011.

ANDROID. **Automated Performance Testing Codelab**. 2016. Disponivel em:<https://codelabs.developers.google.com/codelabs/android-perf-testing/index.html>. Acessado em 10/06/2017.

ARAUJO, T. M. **A responsabilidade do município na proteção e guarda dos animais abandonados**. 2016.

AVILA, A. L.; SPINOLA, R. O. **Introducao a Engenharia de Requisitos**. 2008. Disponivel em: <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-engenharia-de-requisitos/8034>. Acessado em 30/05/2016.

BLACKHEAT, A. J. S. **Functional Reactive Programming**. [S.l.: s.n.], 2016. 1a ed. Editora Maning,.

CORDEIRO, F. **Android SDK: O que e? Para que Serve? Como Usar?** 2017. Disponivel em: <http://www.androidpro.com.br/android-sdk/>. Acessado em 10/06/2017.

DELABARY, B. F. **ASPECTOS QUE INFLUENCIAM OS MAUS TRATOS CONTRA ANIMAIS NO MEIO URBANO**. 2012.

FARACO, C. B. **INTERACAO HUMANO-ANIMAL**. 2008.

FILHO, W. P. P. **Engenharia de Software: fundamentos, metodos e padroes**. [S.l.: s.n.], 2000. 8ed. Editora LTC,.

FOWLER, M. **Inversion of Control Containers and the Dependency Injection pattern**. 2004. Disponivel em:<https://martinfowler.com/articles/injection.html>. Acessado em 10/06/2017.

GAMMA, E. *et al.* **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.: s.n.], 1994.

GEARY, D. **Simply Singleton**. 2003. Disponivel em:<http://www.javaworld.com/article/2073352/core-java/simply-singleton.html>. Acessado em 10/06/2017.

GOOGLE. **Android Platform Architeture**. 2017. Disponivel em: <https://developer.android.com/guide/platform/index.html>. Acessado em 10/06/2017.

HEIDEN, J. **BENEFICIOS PSICOLOGICOS DA CONVIVECIA COM ANIMAIS DE ESTIMACAO PARA OS IDOSOS**. 2012.

HUMBERTO, C. **Design Patterns - Observer**. 2017. Disponivel em: <http://www.devmedia.com.br/design-patterns-observer/16875>. Acessado em 10/06/2017.

JETBRAINS. **Kotlin**. 2016. Disponivel em: <https://kotlinlang.org/>. Acessado em 10/06/2017.

LAZARIN, L. R. **A Comunicação como Estratégia de Gestão em ONGs: informação, sensibilização e engajamento para sustentar e ampliar a garantia dos direitos dos animais.** 2014.

LTD, M. T. P. **Apple Vs Android - A comparative study 2017**. 2017. Disponivel em: <https://android.jlelse.eu/apple-vs-android-a-comparative-study-2017-c5799a0a1683>. Acessado em 10/06/2017.

MACHADO, H. **Os 4 pilares da Programacao Orientada a Objetos**. 2017. Disponivel em: <http://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264>. Acessado em 10/06/2017.

MICROSOFT. **Model-View-Controller**. 2016. Disponível em: <https://msdn.microsoft.com/en-us/library/ff649643.aspx>. Acessado em 09/11/2016.

MSDN. **The Repository Pattern**. 2016. Disponivel em:<https://msdn.microsoft.com/en-us/library/ff649690.aspx/>. Acessado em 10/06/2017.

NTT.CC. **Design Patterns in ActionScript-Observer**. 2009. Disponivel em:<http://ntt.cc/2009/01/13/gang-of-four-gof-design-patterns-in-actionscript-observer.html>. Acessado em 10/06/2017.

OLIVEIRA, J. V. M. de. **Desenvolvimento de Protótipo de Sistema Web para Gerenciamento de Agência Lotérica Utilizando Programação Funcional Reativa**. 2016.

ORACLE. **Obtenha Informações sobre a Tecnologia Java**. 2017. Disponivel em:<https://www.java.com/pt_BR/about/>. Acessado em 10/06/2017.

ORLANDO, V. T. **Guarda Responsavel**. 2014. Disponivel em: <http://www.uipa.org.br/guarda-responsavel/>. Acessado em 10/06/2017.

RISHABH. **Understanding The Difference Between MVC, MVP and MVVM Design Patterns**. 2016. Disponivel em: <https://www.linkedin.com/pulse/understanding-difference-between-mvc-mvp-mvvm-design-rishabh-software>. Acessado em 10/06/2017.

ROCHA, A. R. C. D. **Qualidade de software - Teoria e Prática**. [S.l.: s.n.], 2001.

SILVA, M. E. T. da. **Resumo Executivo do projeto rede de defesa e proteção animal da cidade de Curitiba**. 2006.

STALTZ, A. **The introduction to Reactive Programming you've been missing**. 2016. Disponivel em:<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Acessado em 10/06/2017.

STANEK, M. **Dependency injection with Dagger 2 - Custom scopes**. 2015. Disponível em: <http://frogermcs.github.io/dependency-injection-with-dagger-2-custom-scopes/>. Acessado em 10/06/2017.