VINICIUS VANDERLEI HILÁRIO QUILICE

DESENVOLVIMENTO DE UM APLICATIVO ANDROID PARA CADASTRO E GERENCIAMENTO DE DADOS NA OVINOCULTURA

LAVRAS – MG
2019

VINICIUS VANDERLEI HILÁRIO QUILICE

DESENVOLVIMENTO DE UM APLICATIVO ANDROID PARA CADASTRO E GERENCIAMENTO DE DADOS NA OVINOCULTURA

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Engenharia de Controle e Automação, para a obtenção do título de Bacharel.

Prof. Dr. Leonardo Silveira Paiva

Orientador

Prof. Dra. Iraides Ferreira Furusho Garcia

Coorientadora

LAVRAS – MG
2019

VINICIUS VANDERLEI HILÁRIO QUILICE

DESENVOLVIMENTO DE UM APLICATIVO ANDROID PARA CADASTRO E GERENCIAMENTO DE DADOS NA OVINOCULTURA

DEVELOPMENT OF AN ANDROID APP FOR REGISTRATION AND DATA MANAGEMENT IN THE SHEEP INDUSTRY

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Engenharia de Controle e Automação, para a obtenção do título de Bacharel.

APROVADA em 13 de novembro de 2019.

Dr. Wilian Soares Lacerda UFLA

Me. David Augusto Ribeiro UFLA

____

Prof. Dr. Leonardo Silveira Paiva

Orientador

Prof. Dra. Iraides Ferreira Furusho Garcia

Coorientadora

LAVRAS – MG
2019

AGRADECIMENTOS

Gostaria primeiramente de agradecer a todos os meus familiares, pois sem eles eu jamais conseguiria chegar até aqui, devido à todas as dificuldades que enfrentei durante a graduação.

Em especial, agradeço aos meus pais, Edna e Vanderlei, que sempre me apoiaram em todas as minhas escolhas e também me deram toda a base educacional necessária para caminhar com os meus próprios pés, respeitando sempre à todos em meu meio de convivência. Eles são sem sombra de dúvidas, o meu maior motivo de orgulho.

Aos meus avós, Celina, Juvenil, Virgínia e Horácio, que literalmente foram todos como pais para mim, me ajudando e auxiliando em toda a minha base educacional e em todas as ocasiões que tive mais dificuldades, sejam elas psicológicas, estruturais, educacionais ou financeiras. Os quatro puderam me proporcionar momentos magníficos, daqueles que eu sempre desejei que nunca tivessem terminado.

A todos os meus tios e primos, que são muitos, porém indispensáveis para me mostrar através de muito amor e carinho, o verdadeiro valor de uma família.

A minha namorada Lays, que me acolheu em sua vida, me dando muito amor e força de vontade para seguir em frente e nunca desistir em alcançar os meus objetivos. Nos dias de hoje, não consigo me ver sem ela do meu lado, é uma pessoa extremamente fundamental para mim, em todo os pontos da minha vida.

Ao professor doutor Leonardo, que abraçou meu projeto no momento que eu mais precisava de apoio para executá-lo. Sou extremamente grato a ele.

A professora doutora Iraides, que juntamente com o Grupo de Apoio à Ovinocultura da UFLA e minha namorada Lays, propuseram a mim o tema deste trabalho e me deram total orientação para que eu pudesse executá-lo.

A todos os meus amigos. Em especial: Diego, Leonardo e Rosana, que foram fundamentais para mim durante toda esta caminhada, principalmente no início da minha trajetória acadêmica.

E por fim, agradeço a Universidade Federal de Lavras pela oportunidade de ter me graduado em uma das maiores universidades do país.

MUITO OBRIGADO!

“ A ciência é a lenta revelação do plano de Deus. ” (*Hatiie Gerst)*

RESUMO

Com o início da terceira revolução industrial e o grande aumento populacional a partir do século XX, tornou-se necessário a implementação de medidas para aumentar a produção de bens de consumo para atendimento de uma grande demanda. A automação industrial veio então para agilizar e padronizar os meios de produção, de forma a abastecer o mercado de forma mais eficiente. Posteriormente com a evolução tecnológica dos meios de comunicação a partir do final século XX, e principalmente, com a grande disseminação dos aparelhos móveis, cresceu também o desenvolvimento de aplicativos mobile para auxiliar ainda mais a automatização de todos os setores produtivos da economia mundial. Tendo em vista que para o segundo trimestre de 2019 o setor pecuário tem perspectiva de crescimento de 2,7% no PIB brasileiro (IPEA,2019), a automação do mesmo possui muita demanda de serviços. Neste contexto, este presente trabalho visa projetar e desenvolver uma versão teste de aplicativo Android, visando auxiliar os produtores de ovinos a gerenciarem melhor os dados de seus rebanhos, estimulando um aumento qualitativo na produção de carne de cordeiro, que é atualmente o principal produto produzido no setor.

Palavras-chave: Automação. Aplicativos mobile. Android. Dados. Ovinocultura.

ABSTRACT

With the beginning of the third industrial revolution and the great population increase from the twentieth century, it became necessary to implement measures to increase the production of consumer goods to meet a high demand. Industrial automation then came to streamline and standardize the means of production in order to supply the market more efficiently. Subsequently with the technological evolution of the media from the late twentieth century, and especially with the widespread spread of mobile devices, the development of mobile applications to further assist the automation of all productive sectors of the world economy. Given that for the second quarter of 2019 the livestock sector has a growth perspective of 2.7% in Brazilian GDP (IPEA, 2019), its automation has a high demand for services. In this context, this work aims to design and develop a trial version of Android application, aiming to help sheep producers to better manage the data of their herds, stimulating a qualitative increase in lamb meat production, which is currently the main product produced in the sector.

Keywords: Automation. Mobile apps. Android. Data. Sheep farming.

LISTA DE FIGURAS

Figura 1 – Atividades realizadas por uma linguagem de acordo com o seu nível 15
Figura 2 – Representação de um programa em linguagem de máquina 15
Figura 3 – Representação da soma de dois números na linguagem assembly 16
Figura 4 – Representação da soma de dois inteiros em linguagem c 16
Figura 5 – Resumo linguagens de programação 17
Figura 6 – Comparação de programações 18
Figura 7 – Diagrama esquemático da arquitetura da plataforma android sdk 19
Figura 8 – Execução de um programa na linguagem java. 20
Figura 9 – Composição da plataforma java e esquematização de seu funcionamento........... 20
Figura 10 – Exemplo de um código java 21
Figura 11 – Classe universidade com sua implementação. 22
Figura 12 –Instanciando objeto ufla da cidade de lavras, possuindo 23 cursos. 21
Figura 13 – Encapsulamento da classe universidade... 23
Figura 14 – Arraylist de universidades juntamente com o método de adição. 24
Figura 15 – Utilização dos conceitos de herança no exemplo de universidades...................... 25
Figura 16 – Utilização dos conceitos de interface no exemplo de universidades 26
Figura 17 – Fluxograma de funcionamento de um sistema sgbd 27
Figura 18 – Modelo relacional (tabelas) 28
Figura 19 – Criação do banco de dados pessoa e da tabela pessoas....................................... 29
Figura 20 – Exemplo da utilização de alguns comandos no sqlite 30
Figura 21 – Exemplo de um código na linguagem xml 32
Figura 22 – Layout da tela implementada através do código da figura 20 32
Figura 23 – Android studio 36
Figura 24 – Activity menu principal 38
Figura 25 – Activity sobre o aplicativo 39
Figura 26 – Activity representativa do gao 40
Figura 27 – Activity do sub menu gerenciamento do criadouro. 41
Figura 28 – Activity cadastrar dados de uma ovelha 42
Figura 29 – Activity consulta banco de dados 43
Figura 30 – Activity atualiza ou remove dados. 44
Figura 31 – Layout representativo dos dados que serão concatenados no listview. 45
Figura 32 – Tela de ajuda 1 46

Figura 33 – Tela de ajuda 2 46
Figura 34 – Tela de ajuda 3 47
Figura 35 – Classe ovelha 48
Figura 36 – Classe databasehelper utilizada para criação do banco de dados 49
Figura 37 – Tabela table_sheeps 49
Figura 38 – Método para adição de dados 50
Figura 39 – Método para remoção de dados 51
Figura 40 – Método para edição de dados 51
Figura 41 – Método para consultar dados por número de identificação da ovelha 52
Figura 42 – Classe CustomAdapter e seus três primeiros métodos 53
Figura 43 – Implementação do método getview na classe customadapter 54
Figura 44 – Implementação de um adaptador para o listview da classe GerallSheeps 54
Figura 45 – Implementação para o edittext do número de uma olveha 55
Figura 46 – Utilização da classe intent para mudança de activity. 55
Figura 47 – Utilização da classe toast para confirmar que os dados foram salvos. 56
Figura 48 – Entrando com os dados da primeira ovelha 58
Figura 49 – Confirmação da operação de salvamento de dados 58
Figura 50 – Consultando os dados adicionados 59
Figura 51 – Tela de remoção e atualização de dados com os dados selecionados 60
Figura 52 – Dados da ovelha 01 alterados e salvos 60
Figura 53 – Banco de dados vazio após a remoção da ovelha número 01 61
Figura 54 – Método de pesquisa em funcionamento 62

SUMÁRIO

1. INTRODUÇÃO 11

2. OBJETIVOS 12

2.1 Objetivo geral 13

2.2 Objetivos específicos 13

3. REFERENCIAL TEÓRICO 14

3.1 Linguagens de programação 14

3.2 Programação orientada à objetos 17

3.3 Android Studio 18

3.3.1Linguagem Java 19

3.3.1.1 Classe 21

3.3.1.2 Objeto 22

3.3.1.3 Encapsulamento 23

3.3.1.4 Coleções 24

3.3.1.5 Herança 25

3.3.1.6 Interface 26

3.3.1.7 Método onCreate 26

3.3.2 Sistema de Gerenciamento de Banco de Dados (SGBD) 26

3.3.2.1 SQLite 28

3.3.2.1.1Comando e Instruções biblioteca SQL 29

3.3.3 Desenvolvimento de Layouts das activities e Linguagem XML 31

3.4 Escrituração zootécnica e índices da ovinocultura 33

3.4.1. Escrituração zootécnica 33

3.4.2 Índices da ovinocultura utilizados 34

4. MATERIAL E MÉTODOS 36

4.1 Metodologia 36

4.1.1 Projeto das activities 37

4.1.2 Desenvolvimento dos layouts das activities 37

4.1.2.1 Menu principal 37

4.1.2.2 Sobre o aplicativo 38

4.1.2.3 Grupo de Apoio à Ovinocultura 39

4.1.2.4 Gerenciamento do criadouro 40

4.1.2.5 Cadastramento de dados 41

4.1.2.6 Consulta ao banco de dados 42
4.1.2.7 Atualização e remoção de dados 43
4.1.2.8 Dados impressos na ListView 44
4.1.2.9 Telas de Ajuda 45
4.1.3 Programação das activities utilizando a linguagem Java 47
4.1.3.1 Criação da classe Ovelha 47
4.1.3.2 Criação do Banco de Dados 48
4.1.3.2.1 Criação da tabela 49
4.1.3.2.2 Criação dos métodos de operação 49
4.1.3.2.2.1 Adição de dados 50
4.1.3.2.2.2 Remoção de dados 51
4.1.3.2.2.3 Edição de dados 51
4.1.3.2.2.4 Consultar dados 52
4.1.3.3 Classe CustomAdapter 52
4.1.3.4 Classe GetAllSheeps 54
4.1.3.5 Programação de funcionamento dos EditTexts 55
4.1.3.6 Mudanças de activities 55
4.1.3.7 A classe Toast 56
5. RESULTADOS 57
5.1 Navegação entre activities 57
5.2 Operações de gerenciamento do banco de dados 57
5.2.1 Salvamento de dados 57
5.2.2 Edição de dados 59
5.2.3 Remoção de dados 61
5.2.4 Pesquisa de dados 61
5.3 Viabilidade para uso do ovinocultor 62
5.4 Projetos futuros 62
6. CONCLUSÃO 64
REFERÊNCIAS 65

1. INTRODUÇÃO

A história dos dispositivos móveis começa a ser contada em vias de fato, em março de 1983, ano em que a Motorola apresentou ao mercado mundial o primeiro aparelho móvel portátil. O DynaTAC 8000X, como era chamado, conseguiu obter êxito ao realizar a primeira chamada móvel, ao estabelecer comunicação com um telefone fixo.

Apesar de possuírem pouca eficiência devido a grandes sensibilidades de interferência de sinal, a geração de dispositivos móveis chamada de 1G, impactou profundamente a sociedade na segunda metade do século XX, com transmissão de voz via radiofrequência.

Com o surgimento dos aparelhos 2G no início dos anos 90, os dispositivos móveis deixaram para trás a tecnologia analógica, começando a operar com sinais digitais. A partir desta mudança, os dispositivos começaram a ter uma nova função que foi muito difundida na época, a troca de mensagens via SMS. Observando o alto potencial que este recurso tivera, as grandes empresas de dispositivos móveis começaram a aperfeiçoar cada vez mais seus aparelhos, proporcionando a cada ano, melhorias inovadoras em seus produtos.

Em 2007 a Apple anunciou o dispositivo móvel que viria a revolucionar de vez a forma com que a sociedade desenvolveria suas relações via internet. O primeiro Iphone pode ser considerado de fato, o primeiro smartphone inteligente já criado (TABOADA; CÓRCOLES, 2015).

Apesar de sua criação ter sido por Andy Rubin em 2003, na cidade de Palo Alto na Califórnia, o sistema operacional Android começou a ser distribuído pela Google em 2007, baseado no sistema Linux Kernel (NARMATHA; KRISHNAKUMAR, 2016). Com o seu aperfeiçoamento devido a elaboração constante de novas versões mais atualizadas, o Android foi se tornando cada vez mais popular, chegando a ser o sistema operacional mais utilizado no Brasil em setembro de 2019, estando presente em 85,57% dos smartphones vendidos (STATCOUNTER, 2019), sendo que o país possui mais de um smartphone por habitante, sendo assim, há um total de cerca de 230 milhões de aparelhos na totalidade.

Com a evolução cada vez maior dos smartphones, dos sistemas operacionais e também com um uso cada vez mais frequente destes dispositivos pela população, houve um grande aumento no investimento em desenvolvimento de aplicativos móveis, visando atender vários setores econômicos e sociais. Os “apps” como são popularmente conhecidos atualmente, são softwares programados para dispositivos móveis, que vêm tornando-se opção indispensável para a automação de tarefas e atividades dos setores mais variados possíveis. De aplicações

industriais e acadêmicas, até jogos e entretenimento. E no setor agropecuário brasileiro, a situação não é diferente.

Com a população nacional ultrapassando a casa de 210 milhões de habitantes (IBGE, 2019), a produção de matéria prima básica para a confecção de alimentos, bebidas, condimentos, vestimentas, entre outros produtos, vem necessitando cada vez mais de processos automatizados que possam garantir maior qualidade e produtividade para atender a uma demanda crescente. Este processo de controle e automação rural, não ocorre somente com maquinários pesados e robustos. Agora, os produtores rurais estão cada vez mais buscando auxílio nos aplicativos móveis, que são projetados e executados de acordo com as necessidades dos produtores rurais (SEBRAE, 2015).

No setor pecuarista, os produtores rurais podem contar com uma variada gama de aplicativos, para auxiliá-los nas mais diversas áreas, como: Rentabilidade do rebanho, controle de pragas, gerenciamento de vacinas, controle de medicamentos, gerenciamento do ambiente, evolução de rebanho, coleta e registro de dados dos animais.

Em busca do desenvolvimento de um exemplar específico desta última área citada, aplicado à ovinocultura, foi estipulado neste presente trabalho, a interdisciplinaridade entre os cursos de Engenharia de Controle e Automação e Zootecnia, tendo no primeiro, parte principal do desenvolvimento, já que o aplicativo foi desenvolvido por diferentes tipos de linguagens de programação, para tornar a coleta e gerenciamento de dados mais eficiente.

O “Ovelheiro” então, surge a partir dos ideais de um aluno de Engenharia de Controle e Automação, juntamente com o Grupo de Apoio à Ovinocultura da Universidade Federal de Lavras.

2. OBJETIVOS

A partir da necessidade que o produtor de ovinos possui em gerenciar melhor os dados de seu rebanho, torna-se viável projetar e desenvolver um aplicativo Android para coletá-los e manipulá-los de forma mais eficiente e otimizada.

2.1. Objetivo Geral

O principal objetivo deste trabalho é desenvolver um aplicativo de teste no Android Studio, que seja funcional e facilite a interação entre o produtor e os dados de seu animal. De

forma que ele possa armazenar os dados de todas as ovelhas de seu rebanho, coletados em campo, em um banco de dados interno a seu smartphone ou tablet, para que ele possa consultar os dados de determinado animal em tempo real, sem a necessidade de uma conexão à internet.

2.2. Objetivos Específicos

Para conseguir obter êxito neste presente trabalho, os seguintes objetivos foram estipulados:

- Adquirir um maior conhecimento em Programação Orientada à Objetos, para criação das classes, objetos e métodos requisitados durante o desenvolvimento do aplicativo;
- Utilização da linguagem Java para programar o funcionamento e a modularização do aplicativo;
- Utilização da linguagem XML para a confecção de layouts funcionais e simples, para não dificultarem a utilização do aplicativo para nenhum usuário;
- Criação de telas de ajuda ao usuário, visando explicar de forma mais clara a utilização de algum determinado recurso do aplicativo;
- Criação e gerenciamento de um Banco de Dados para armazenar todas as entradas digitadas pelo usuário e mostrar o histórico de coletas de dados de um determinado animal, facilitando que o usuário possa acompanhar a rotina do seu rebanho ao longo do tempo;
- Utilização da biblioteca SQLite, tornando-se possível ao usuário fazer uma consulta específica dos dados de um animal em tempo real, sem precisar de conexão à Internet;
- Adquirir maior conhecimento sobre a criação de Ovinocultura, para poder enxergar melhor os problemas da área, buscando propor alternativas de controle e automação para otimizar o setor.

3. REFERENCIAL TEÓRICO

Para o desenvolvimento deste trabalho, é necessário entender alguns conceitos sobre linguagens de programação, principalmente linguagem java orientada à objetos, demonstrando sua importância para o desenvolvimento de aplicativos no Android Studio.

E nesta seção serão apresentados alguns conceitos sobre sistema de gerenciamento de banco de dados, especialmente a biblioteca SQLite, linguagem XML utilizada para o desenvolvimento dos layouts deste aplicativo e também breves conceitos sobre escrituração zootécnica e ovinocultura.

3.1. Linguagens de programação

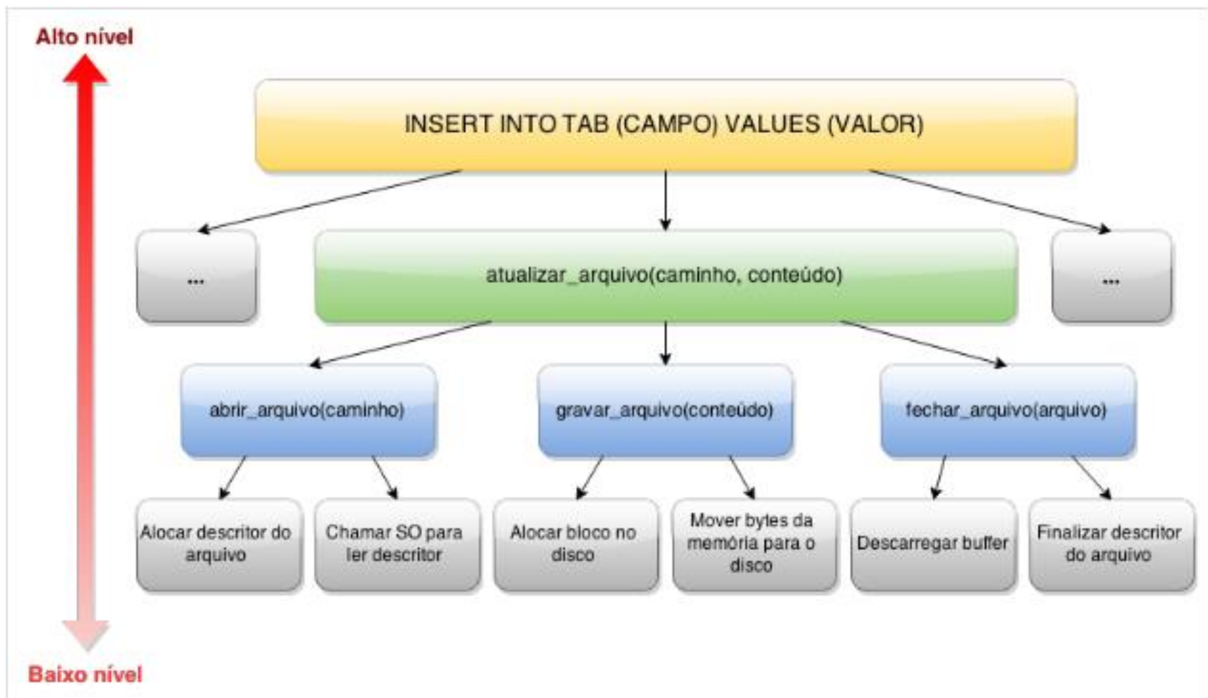
Apesar da grande evolução dos hardwares de microcomputadores para atender as necessidades da sociedade, os mesmos não poderiam realizar suas atividades sozinhos, sem auxílio de um operador. Para isso, tornou-se necessário programá-los para que realizassem as mais diversas tarefas, de acordo com um princípio de funcionamento pré-estabelecido. E para isso, começaram a ser desenvolvidas (e ainda são) vários tipos de linguagens computacionais.

Uma linguagem de programação pode ser definida como sendo um conjunto limitado de instruções (vocabulário), associado a um conjunto de regras (sintaxe) que define como as instruções podem ser associadas, ou seja, como se pode compor os programas para a resolução de um determinado problema (WILLRICH, 2000).

As linguagens de programação podem ser classificadas de acordo com o seu nível, conforme a mesma se aproxima com a linguagem em que o processador do computador torna-se apto a interpretá-la, o nível da mesma torna-se mais baixo.

A Figura 1 nos mostra algumas atividades que fazem com que uma linguagem possa ser classificada de acordo com o seu nível.

Figura 1 – Atividades realizadas por uma linguagem de acordo com o seu nível.



Fonte: Ricardo (2015).

Existem três tipos básicos de linguagem de programação: Linguagem de máquina, linguagens assembly e linguagens de alto nível.

A linguagem de máquina é a de mais baixo nível, correspondendo a linguagem interna de funcionamento do computador. Tendo em vista que o hardware do mesmo é constituído por ligações elétricas e eletrônicas, esta linguagem é baseada em um sistema binário de numeração como mostrado na Figura 2, que representa tanto os dados, quanto as operações (WILLRICH, 2000).

Figura 2 – Representação de um programa em linguagem de máquina

0	0	1	0	0	1	1
1	1	1	1	1	1	0
1	0	0	1	1	1	1
0	0	1	0	0	0	1

Fonte: Do Autor (2019).

Segundo Willrich (2000), embora seja a linguagem diretamente executável pelos processadores, a programação de aplicações diretamente em linguagem de máquina é impraticável, sendo assim, torna-se necessário a utilização de outro recurso para que a linguagem de máquinas passe a ser compreendida.

A linguagem Assembly, representa um código em linguagem de sistema numérico binário através de mnemônicos, tornando-se praticável para programação de máquinas. A Figura 3 mostra um exemplo de código implementado pela linguagem assembly.

Figura 3 – Representação da soma de dois números na linguagem assembly.

```
MOV AX,B       ; registro AX recebe o valor de memória contida na variável B
ADD AX,C       ; AX recebe a soma de AX (valor de B) com o valor de C
MOV A,AX       ; variável A recebe valor de AX
```

Fonte: Willrich (2000).

Por último, as linguagens de alto nível, que são aquelas que a escrita está ainda mais próxima da linguagem do ser humano (*while*, *if*, *else*, *do*, *for*, *then*, *real*) e permitem a manipulação dos dados nas mais diversas formas (números inteiros, reais, vetores, listas), enquanto a linguagem *Assembly* trabalha com bits, bytes, palavras, armazenados em memória (WILLRICH, 2000). Sendo assim, as linguagens de alto nível necessitam de um compilador para serem interpretadas pelo processador da máquina. São exemplos de linguagens de alto nível: C, C++, Java, Fortran, Cobol, Python, Pascal, entre outras. A Figura 4 mostra um exemplo de linguagem de alto nível, representado pela linguagem C, enquanto a Figura 5, apresenta um resumo das linguagens de programação.

Figura 4 – Representação da soma de dois inteiros em linguagem C.

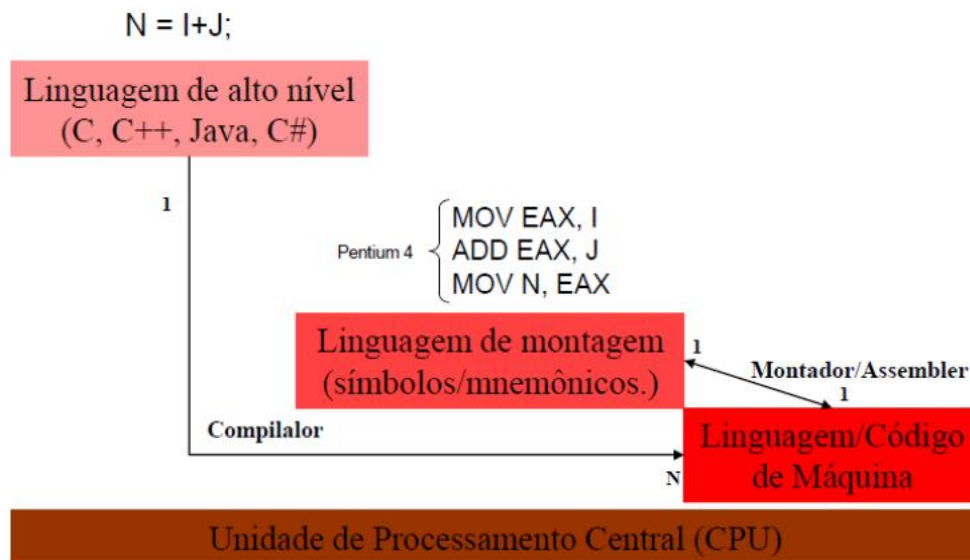
```c
#include <stdlib.h>
#include <stdio.h>

void main(void)
{
  char num1[80], num2[80];

  printf("digite o primeiro: ");
  gets(num1);
  printf("digite o segundo: ");
  gets(num2);
  printf("A soma é: %d.", atoi(num1) + atoi(num2));
}
```

Fonte: Schildt (1996).

Figura 5 – Resumo linguagens de programação.



Fonte: Rocha (2017).

3.2. Programação orientada à objetos

O conceito de orientação à objetos data do final da década de 60 e início da década de 70, surgindo com uma necessidade de modelar sistemas mais complexos através de classes e objetos. Sendo assim, a programação orientada à objetos foi criada para aproximar o mundo digital do mundo real, de forma que quando temos um sistema à ser modelado, pensamos primeiramente nos objetos que fazem parte do problema ao invés de pensarmos nas estruturas de dados em si, ou em rotinas, como observado na Figura 6.

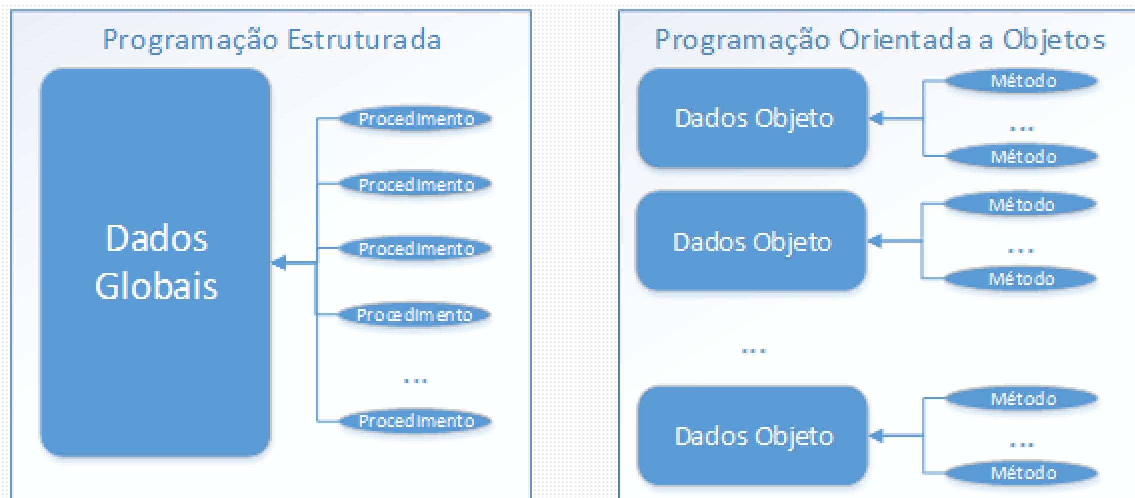
Um dos conceitos mais importantes aqui, é o conceito de abstração. Desta maneira, este tipo de programação consiste em trabalhar um objeto se preocupando com suas propriedades mais importantes, desviando o foco de propriedades mais desnecessárias para modelar o sistema.

O paradigma para este tipo de programação busca sempre a exatidão e obviedade, sendo mais direto ao ponto onde se quer chegar.

Em resumo, a programação orientada à objetos utiliza-se da ideia de modelar o sistema a ser desenvolvido, através da criação de objetos simples e com propriedades bem definidas, determinando o comportamento de cada um e como esses comportamentos interagem entre si, ou seja, como um objeto pode ou não influenciar outro dentro do sistema como um todo.

Atualmente, existem muitas linguagens de programação que podem ser voltadas à orientação à objetos, sendo as principais: C++, C#, VB.NET, Java, Object Pascal, Objective-C, Python, SuperCollider, Ruby e Smalltalk (ENGHOLM, 2013).

Figura 6 – Comparação de programações.



Fonte: Devmedia (2014).

3.3. Android Studio

O Android Studio é um software livre desenvolvido pela Google, que tem como finalidade permitir que seus usuários criem aplicativos para serem utilizados nos mais diversos setores. Para poder oferecer um ambiente adequado para este tipo de desenvolvimento, o Android Studio foi criado como uma plataforma SDK.

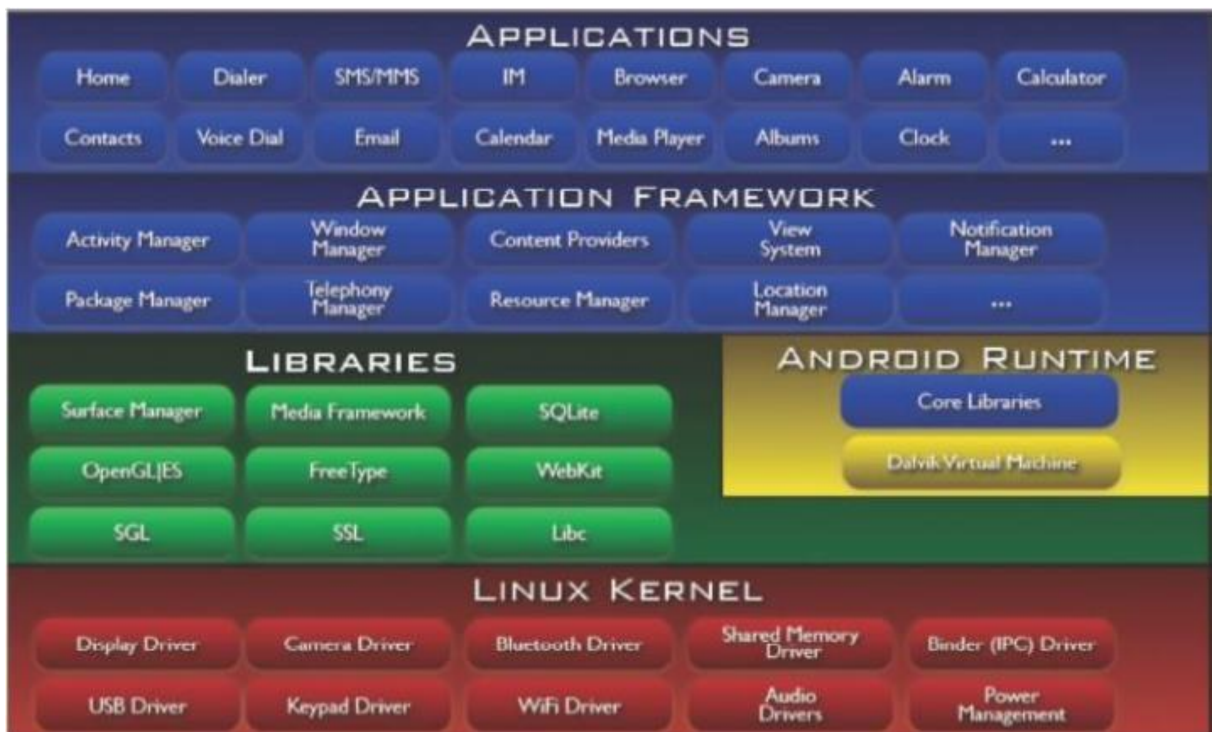
Software Development Kit (SDK) contem API's necessárias para a criação de aplicativos, e inclui um conjunto abrangente de ferramentas de desenvolvimento, incluindo um depurador, bibliotecas de software com códigos prontos, emulador de dispositivo, documentação, exemplos de códigos e tutoriais (GOOGLE DEVELOPER TRAINING TEAM, 2016).

A plataforma SDK possui vários recursos para o desenvolvimento de aplicativos, porém neste presente trabalho, algumas terão mais destaque por terem sido mais utilizadas. Entre elas estão: *Dalvik virtual machine*, que permite a programação em linguagem Java para ser aplicada no funcionamento do aplicativo; *SQLite*, que é o sistema gerenciador de banco de dados (SGBD) do Android, permitindo o armazenamento e manipulação dos dados de entrada do aplicativo; Linguagem *XML*, para confecções dos layouts das telas (*activities*) e

Suporte de Multimídias, para adição das imagens que serão utilizadas no desenvolvimento do aplicativo.

A Figura 7 resume a composição da plataforma Android SDK.

Figura 7 – Diagrama esquemático da Arquitetura da plataforma Android SDK.

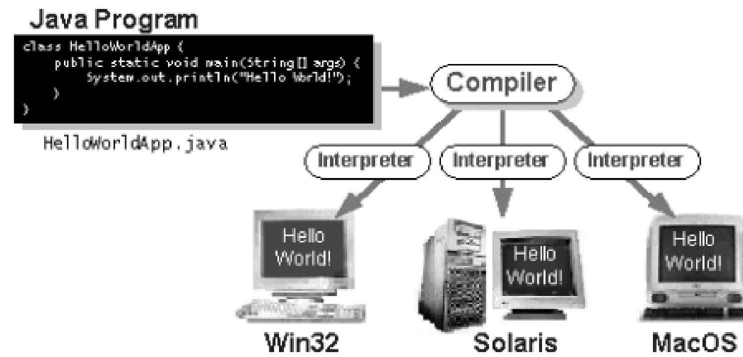


Fonte: SHIBLY (2016).

3.3.1. Linguagem Java

Segundo Claro e Sobral (2008), a linguagem java é amplamente utilizada desde então em aplicações orientada à objetos. Esta popularidade ocorre muito pela grande vantagem que ela possui em relação à outras linguagens, principalmente no quesito de sua execução. Na maioria das linguagens de programação, é necessário compilar ou interpretar um programa para que ele seja executado no computador. A linguagem java, entretanto, consegue realizar as duas ações, compilando e interpretando um programa (MENGUE, 2002). A Figura 8 ilustra a execução de um programa em java.

Figura 8 – Execução de um programa na linguagem java.



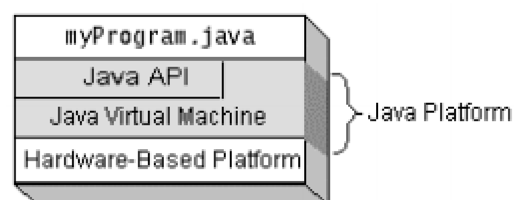
Fonte: Mengue (2002).

Ela consegue realizar esta ação devido à uma linguagem intermediária chamada de bytecode, que independe de qual plataforma o usuário esteja utilizando para executar o seu programa.

Diferente de outras plataformas, a plataforma java não possui um hardware em sua composição, tendo em vista que ela se utiliza de hardwares de outras plataformas para sua execução. Segundo Mengue (2002), a plataforma java é formada por dois componentes: A *Java Virtual Machine* (JVM), que é a principal responsável pelo gerenciamento das execuções de arquivos e a *Java Application Programming Interface* (API), que é a biblioteca que contêm várias utilidades pré-estabelecidas, para as mais diversas aplicações que o programador possa vir a precisar utilizar.

A Figura 9 ilustra a composição de uma plataforma java, integrada ao hardware e à área de escrita do programa, enquanto que a Figura 10, ilustra um trecho de código básico na linguagem java, com utilização de uma classe.

Figura 9 – Composição da plataforma java e esquematização de seu funcionamento.



Fonte: Mengue (2002).

Figura 10 – Exemplo de um código java.

```
class TestaIdade {

    public static void main(String[] args) {
        // imprime a idade
        int idade = 20;
        System.out.println(idade);

        // gera uma idade no ano seguinte
        int idadeNoAnoQueVem;
        idadeNoAnoQueVem = idade + 1;

        // imprime a idade
        System.out.println(idadeNoAnoQueVem);
    }
}
```

Fonte: Caelum (2014).

3.3.1.1. Classe

A classe representa uma generalização de determinado componente do sistema, que pode ser concreto ou abstrato, servindo de modelo para que se crie instâncias a partir de si. Estas instâncias serão os objetos, que se pertencentes à mesma classe, obrigatoriamente deverão possuir suas mesmas características, ou seja, os mesmos atributos e métodos que foram implementados dentro de si.

Os atributos serão as características responsáveis por definirem determinado objeto, enquanto os métodos, serão ações responsáveis para a definição de seu comportamento.

Para que ocorra a instanciação de objetos a partir de determinada classe, a mesma deve conter um construtor, que é um método sem retorno responsável por definir as operações que serão realizadas quando um objeto for criado, servindo de modelo para inicializar os atributos do mesmo.

A Figura 11 mostra uma classe com seu construtor, seus atributos e métodos.

Figura 11 – Classe *Universidade* com sua implementação.

```
public class Universidade {

    //Atributos
    String nome;
    String cidade;
    int numeroDeCursos;

    //Construtor
    public Universidade(String nome, String cidade, int numeroDeCursos) {

    }

    //Métodos
    public String getNome() {
        return nome;
    }
    public String getCidade() {
        return cidade;
    }
    public int getNumCursos() {
        return numeroDeCursos;
    }
}
```

Fonte: Do autor (2019).

3.3.1.2. Objeto

Como o próprio nome já diz, um objeto é uma representação de algum componente que está sendo utilizado no programa e desenvolvido, de forma a aproximá-lo do mundo real. Ele é uma representação mais específica que uma classe, tendo em vista que ele é uma instância da mesma.

A forma com que os objetos interagem entre si via requisições por mensagem (chamada a uma função ou procedimento) é o que determinará as características do programa e dará as diretrizes de sua funcionalidade. A Figura 12 mostra um exemplo de como um objeto é instanciado a partir de uma classe, utilizando o comando "new".

Figura 12 –Instanciando objeto UFLA da cidade de Lavras, possuindo 23 cursos.

```
//Instanciando um novo objeto para a classe Universidade

Universidade universidade = new Universidade("UFLA", "Lavras", 23);
```

Fonte: Do autor (2019).

O estado de um objeto está diretamente relacionado com os seus atributos e valores que os mesmos assumem, dado algum momento. Já o seu comportamento é definido por seus métodos.

3.3.1.3. Encapsulamento

Quando utilizamos um sistema, seja de qual natureza for, desejamos que o seu funcionamento seja adequado e de maneira clara, sem na maioria das vezes, nos preocuparmos em como os funcionamentos internos deste sistema realizam determinada ação. Em programação orientada à objetos utilizando a linguagem java, este princípio é chamado de encapsulamento. Isso quer dizer de forma mais técnica, que faz parte das boas práticas de programação, esconder os estados dos objetos, tendo em vista que a necessidade de interação do código com o usuário, se dê somente por meio dos métodos. Fazendo isso, criamos uma interface para o sistema, em que podemos acessar suas funcionalidades somente pelos seus métodos, não necessitando de acesso à implementação.

Para que a classe tenha um bom design então, é viável que seus atributos sejam privados e seus métodos públicos, para aumentar seu grau de coesão e em contrapartida, diminuir o acoplamento com as outras classes, fazendo com que futuras alterações na implementação de uma classe afetem menos as outras, não prejudicando o funcionamento do sistema como um todo. A Figura 13 exemplifica bem o princípio de encapsulamento aplicado à classe Universidade.

Figura 13 – Encapsulamento da classe *Universidade*.

```
public class Universidade {

    //Atributos privados
    private String nome;
    private String cidade;
    private int numeroDeCursos;

    //Construtor
    public Universidade(String nome, String cidade, int numeroDeCursos) {

    }

    //Métodos públicos
    public String getNome() {
        return nome;
    }
    public String getCidade() {
        return cidade;
    }
    public int getNumCursos() {
        return numeroDeCursos;
    }
}
```

Fonte: Do autor (2019).

3.3.1.4. Coleções

Quando criamos várias classes que a partir delas serão instanciados uma enorme quantidade de objetos, torna-se necessário que tenhamos algum tipo de estrutura para armazená-los e gerenciá-los de forma adequada. Para isso, a biblioteca java possui em sua documentação algumas estruturas de dados para realizarem este tipo de tarefa, são as chamadas coleções.

Uma dessas coleções mais utilizadas é a classe *ArrayList*, que é um tipo de *array*, porém possui a sua capacidade de armazenamento dinamicamente redimensionável, ou seja, ele possui tamanho flexível que varia de acordo com a quantidade de elementos que queremos adicionar a ele.

A biblioteca java possui também algumas operações básicas que podemos utilizar para manipulação de dados de um *ArrayList*. Entre elas, podemos adicionar um novo elemento, removê-lo, conferir se o *ArrayList* está vazio, acessar a posição de algum elemento, entre outras. A Figura 14 mostra a implementação de um ArrayList juntamente com um método para adicionar um novo elemento a ele.

Figura 14 – ArrayList de universidades juntamente com o método de adição.

```java
//Criando ArrayList
private ArrayList<Universidade> universidades = new ArrayList<Universidade>();

//Método para adicionar uma nova universidade no ArrayList
public void adicionarUniversidade(Universidade univ) {
    universidades.add(univ);
}
```

Fonte: Do autor (2019).

3.3.1.5. Herança

Diversas vezes quando estamos projetando um sistema utilizando a linguagem java orientada à objetos, deparamos com classes muito semelhantes, de forma que muitos atributos e métodos acabam sendo os mesmos para ambas.

Sendo assim, se o programa conter muitas classes semelhantes, fica inviável temos que escrever todos os métodos e atributos em comum para todas as classes que os possuem, pois, além de aumentar muito o trabalho do programador, existe um grande risco de acontecer duplicações de código, dificultando futuras edições.

A solução mais viável para amenizar este problema, é a utilização de herança, que segundo Ricarte (2001) é um mecanismo que permite que características comuns a diversas

classes sejam fatoradas em uma classe base, ou superclasse. A partir de uma classe base, outras classes podem ser especificadas. Cada classe derivada ou subclasse apresenta as características (estrutura e métodos) da classe base e acrescenta a elas o que for definido de particularidade para ela. Ou seja, uma subclasse possui todos os métodos e atributos de uma superclasse, e ainda pode possuir alguns a mais para complementar suas características.

Para identificarmos uma classe como sendo subclasse de outra, utilizamos a palavra reservada *extends*, traduzindo a ideia de que uma classe está estendendo as características e a funcionalidade de sua superclasse (GAÚCHO, 2016).

Um importante conceito para o uso de herança é o método *super().* Este método representa a chamada ao construtor da superclasse. Se não for chamado explicitamente o compilador irá gerar código para chamar o construtor default da *superclasse*. No entanto, se desejamos chamar explicitamente um construtor com argumentos devemos usar o método *super()* (OLIVEIRA; MACIEL, 2002).

Devemos tomar cuidado ao usar herança, pois se não sobrescrevermos algum método herdado de uma superclasse por uma subclasse, o compilador java entenderá que este método está sendo herdado da classe *Object*, uma classe intrínseca à API Java. A anotação *@Override* deve preceder os métodos que serão sobrescritos na subclasse.

A Figura 15 mostra um exemplo de aplicação em código dos conceitos de herança.

Figura 15 – Utilização dos conceitos de herança no exemplo de universidades.

```
//Subclasse Universidade_Particular herdando da superclasse Universidade
public class Universidade_Particular extends Universidade {

    private double mensalidade;
    private int diasLetivos;
    private int precoAula;

    public Universidade_Particular(String nome, String cidade, int numeroDeCursos,
            double mensalidade, int diasLetivos, int precoAula) {

        //Palavra reservada super, representando os atributos vindo da superclasse
        super(nome, cidade, numeroDeCursos);
    }

    //Sobrescrição do método calcularMensalidade()
    @Override
    public double calcularMensalidade() {
        return mensalidade = diasLetivos*precoAula;
    }
}
```

Fonte: Do autor (2019).

3.3.1.6. Interface

Um dos contratempos que podem ocorrer em implementações em java, é o chamado problema de “herança múltipla”, tendo em vista que este tipo de herança não é permitido pela plataforma java. Este problema ocorre quando uma determinada classe requisita uma herança múltipla, ou seja, ela precisa herdar atributos ou métodos de mais de uma superclasse.

Para auxiliar na resolução parcial deste problema, existem as interfaces, que são formadas por um ou mais métodos que não possuem implementação. A palavra reservada implements garante que uma classe está implementando uma interface. A Figura 16 mostra um exemplo de implementação de interface.

Figura 16 – Utilização dos conceitos de interface no exemplo de *Universidades*.

```
//Interface criada para demonstrar que para este exemplo a Universidade Particular é semestral
public interface Semestral {

    //Método sem implementação
    public void periodoSemestral();
}
```

```
//Subclasse Universidade_Particular implementando a Interface
public class Universidade_Particular implements Semestral {
```

Fonte: Do autor (2019).

3.3.1.7. Método onCreate

Este método é fundamental no Android Studio, pois é nele que todo o código java de funcionamento de uma *Activity* do aplicativo, vai estar implementado. Se a implementação estiver fora deste método, a aplicação não funcionará.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
```

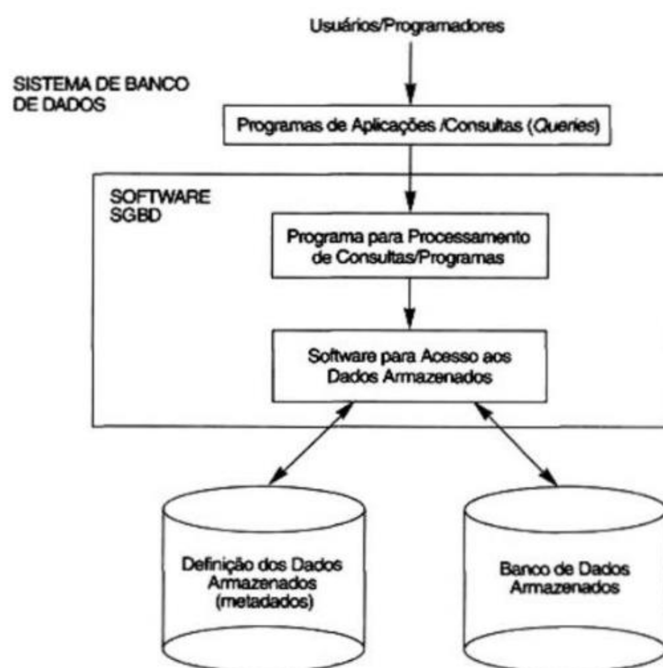
3.3.2. Sistema de Gerenciamento de Banco de Dados (SGBD)

Em sua grande maioria, os aplicativos desenvolvidos no Android Studio necessitam de um sistema de gerenciamento de banco de dados, para que as entradas no aplicativo possam ser armazenadas e gerenciadas de forma segura e eficiente, utilizando-se de várias operações. Entre

elas, um sistema para consultar os dados cadastrados, de forma que o SGBD criado gerencie estas pesquisas e operações no banco de dados.

A Figura 17 resume uma integração de um SGBD ao programa desenvolvido pelo usuário no Android Studio.

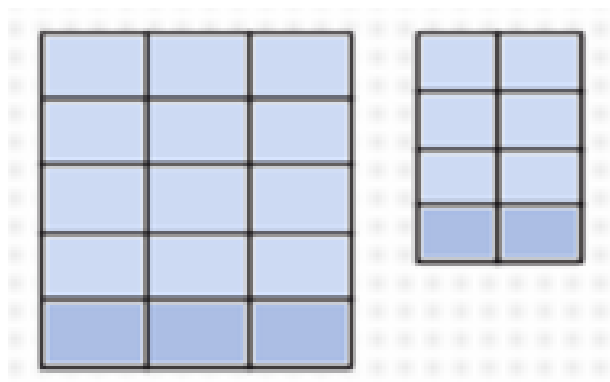
Figura 17 – Fluxograma de funcionamento de um sistema SGBD.



Fonte: Researchgate (2018).

Estes sistemas possuem vários modelos de base de dados, sendo que neste presente trabalho, será usado o modelo relacional, adequado a ser o modelo subjacente de um Sistema Gerenciador de Banco de Dados (SGBD), que se baseia no princípio em que todos os dados estão guardados em tabelas (ou matematicamente falando, relações). Toda sua definição é teórica e baseada na lógica de predicados e na teoria dos conjuntos (DEVMEDIA, 2014). A Figura 18 ilustra um modelo de banco de dados relacional, evidenciando as linhas e colunas de uma tabela. Desta forma, algum tipo de biblioteca deve ser utilizado para realizar estas ações de forma eficaz.

Figura 18 – Modelo Relacional (tabelas).



Fonte: Devmedia (2014).

3.3.2.1. SQLite

Segundo Coelho e Prado (2015), o SQLite é um Banco de Dados *Open Source* e relacional, sendo uma influente biblioteca de banco de dados baseado em SQL (Structured Query Language) que tem como função ser parecido com um gerenciamento de banco de dados, no qual alguma das suas funções é manter o controle de diversos bancos de dados e simultaneamente suas tabelas.

Este modelo de banco de dados é muito utilizado no desenvolvimento de aplicativos móveis Android, já que a plataforma JDK suporta este tipo de biblioteca, além dele possuir requisitos muito eficientes para a realização desta tarefa.

Primeiramente a facilidade da utilização do SQLite é que toda a criação e gerenciamento do banco de dados desenvolvido, fica interno à aplicação em um único arquivo, fazendo com que sua utilização não dependa de terceiros e também de uma conexão à rede de internet. Além disso, sua base de dados pode ter tamanho superior a 2 *terabytes* e ele é um gerenciador de banco de dados "autossuficiente", ou seja, independe de uma estrutura cliente-servidor (COELHO; PRADO, 2015).

A classe em que o Banco de Dados será implementado deve estender a superclasse *SQLiteOpenHelper*, pois ela contém os principais métodos para garantir o funcionamento e gerenciar a criação do mesmo. O primeiro método que deve ser sobrescrito é o método *onCreate(),* que só será chamado enquanto o banco de dados ainda não foi criado. Portanto, é o método para garantir que o banco de dados já foi criado e está pronto para ser usado. O segundo método é o *onUpgrade()*, que só é chamado se a versão atual do aplicativo não

corresponde ao do construtor e a do banco de dados, garantindo então que a versão utilizada é a mesma para todos os componentes.

Além destes métodos, temos também o construtor com os parâmetros necessários para utilização dos recursos provenientes da superclasse.

A Figura 19 mostra um exemplo de uma classe responsável por estender a superclasse *SQLiteOpenHelper*.

Figura 19 – Criação do banco de dados pessoa e da tabela pessoas.

```
public class DataBaseHelper extends SQLiteOpenHelper {

//Atributos do banco de dados
    public static String DATABASE_NAME = "pessoa";
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_PESSOA = "pessoas";
    private static final String KEY_ID = "id";
    private static final String KEY_IDADE = "idade";
    private static final String KEY_DATA_NASCIMENTO = "nascimento";
    private static final String KEY_PESO = "peso";

//Criação da tabela PESSOA
    private static final String CREATE_TABLE_PESSOA = "CREATE TABLE "
            + TABLE_PESSOA + "(" + KEY_ID
            + " INTEGER PRIMARY KEY AUTOINCREMENT," KEY_IDADE + "
TEXT, " + KEY_DATA_NASCIMENTO + " TEXT, " + KEY_PESO + " TEXT

//Construtor estendendo atributos da superclasse SQLiteOpenHelper
public DataBaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
        Log.d("table", CREATE_TABLE_PESSOA);
    }
//Método onCreate
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_TABLE_PESSOA);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        db.execSQL("DROP TABLE IF EXISTS '" + TABLE_PESSOA + "'");
        onCreate(db);
    }
```

Fonte: Do autor (2019).

3.3.2.1.1. Comandos e Instruções biblioteca SQL

O SQLite é uma variação da biblioteca SQL, só que aplicado à dispositivos móveis, devido as vantagens citadas no tópico anterior. Sendo assim, os comandos utilizados para um banco de dados SQL, podem ser utilizados também para a criação e manipulação de dados, usando a biblioteca SQLite.

Alguns comandos para a manipulação de dados são de suma importância, tendo em vista que são eles que ditarão as ações que podemos realizar no banco de dados através de comandos gerados pelo aplicativo. Eles possuem dois tipos de instruções principais: As DDL e as DML.

Para Elmasri e Navathe (2011), as instruções (ou comandos) DDL são utilizadas para a definição da estrutura do modelo físico do banco de dados, através de alguns comandos básicos. As mais importantes para o entendimento deste presente trabalho são:

- CREATE: Responsável pela criação das tabelas;
- DROP: Remove uma tabela caso ela já exista.

Já as operações DML são utilizadas para se fazer alterações no banco de dados em si. As quatro principais e mais importantes são:

- INSERT: Insere registros na tabela selecionada;
- UPDATE: Atualiza os dados de uma tabela selecionada;
- DELETE: Semelhante à instrução INSERT, porém, é utilizada para fazer a remoção de algum registo da tabela;
- SELECT: Importa dados pré-selecionados específicos de determinada tabela. Deve ser utilizada prescindido de outras instruções, como:
  FROM + NOME DA TABELA: indica de qual tabela os dados serão importados;
- SELECT*: Semelhante à instrução SLECT simples, porém, importa todos os dados de uma tabela específica;
- QUERY: Utilizado para fazer buscas de informação em alguma tabela.

A Figura 20 mostra um exemplo com algumas destas instruções e comandos sendo utilizados para manipulação de dados de uma tabela de pessoas.

Figura 20 – Exemplo da utilização de alguns comandos no SQLite.

```
            //Criando uma tabela chamada 'pessoas' com duas variáveis
            bancoDeDados.execSQL("CREATE TABLE IF NOT EXISTS pessoas(
nome VARCHAR, idade INT(3))");

            //inserindo duas pessoas na tabela
            bancoDeDados.execSQL("INSERT INTO pessoas(nome, idade)
VALUES('Vinicius', 27)");
            bancoDeDados.execSQL("INSERT INTO pessoas(nome, idade)
VALUES('João', 42)");

            //Importando dados 'nome' e 'idade' da tabela 'pessoas'
            Cursor cursor = bancoDeDados.rawQuery("SELECT nome, idade
FROM pessoas", null);
```

Fonte: Do Autor (2019).

3.3.3. Desenvolvimento de Layouts das activities e Linguagem XML

O Android Studio permite que seus layouts sejam elaborados através da linguagem XML, que apesar de possuir este nome, não é uma linguagem de programação e sim uma linguagem de marcação. Ao invés de se utilizar de comandos universais das linguagens de programação de alto nível, como instruções *for* e *if* por exemplo, a linguagem XML delimita informações, deixando bem claro onde uma instrução começa e onde termina.

É uma linguagem semelhante ao HTML para a criação de layouts de web sites, mas aqui, iremos usá-la para criar layouts de aplicativos móveis no Android Studio.

Cada arquivo de layout deve conter exatamente um elemento raiz, que deve ser um objeto *View* ou *ViewGroup*. Com o elemento raiz definido, é possível adicionar objetos ou *widgets* de layout extras como elementos filho para construir gradualmente uma hierarquia de *View* que define o layout (ANDROID DEVELOPERS, 2019).

O Android Studio permite trabalharmos com dois tipos de layouts pré-definidos para implementação do XML. Primeiramente temos o *Relative Layout*, em que criamos um elemento na tela a partir da relação com outro. Para criar-se um botão por exemplo, podemos usar um texto como referência de posicionamento. O segundo tipo é o *Linear Layout*, que permite que alinhemos nossos elementos na tela seguindo dois tipos de orientações, vertical ou horizontal. Sendo assim, ambos os tipos possuem o mesmo objetivo, posicionar os elementos do layout criado, de forma adequada e organizada.

A Figura 21 deixa claro do por que XML é uma linguagem de marcação, onde a instrução começa com “<” e termina com “/>” através de um layout com um texto e um simples botão. Foi utilizado o *Relative Layout*, em que o botão está “58p” de distância abaixo da tela em relação ao texto.

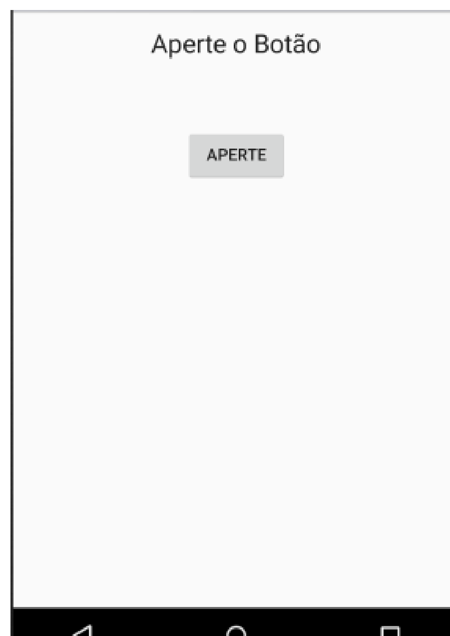
A Figura 22 nos mostra o layout da tela implementado na figura 21.

Figura 21 – Exemplo de um código na linguagem XML.

```
<?xml version="1.0" encoding="utf-8"?>
//Utilizando o Relative Layout
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.ovelheiro.vinihg.ovelheiro.AperteBotao">
//<Começa uma instrução
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="Aperte o Botão"
        android:id="@+id/textView15"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" />
//Termina uma instrução />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Aperte"
        android:id="@+id/button"
        android:layout_below="@+id/textView15"
        android:layout_alignLeft="@+id/textView15"
        android:layout_alignStart="@+id/textView15"
        android:layout_marginTop="58dp"
        android:layout_alignRight="@+id/textView15"
        android:layout_alignEnd="@+id/textView15" />
</RelativeLayout>
```

Fonte: Do Autor (2019).

Figura 22 – Layout da tela implementada através do código da Figura 20.



Fonte: Do Autor (2019).

Neste presente trabalho, foram utilizados os seguintes elementos para desenvolver os layouts das activities:

- Button: São os elementos que ao ser pressionados gera alguma ação, podendo ser uma mudança de tela, salvamento de dados, remoção de dados, etc;
- TextView: Usados para escrita de textos utilizados durante a execução do aplicativo, como textos de ajuda, nome de telas, descrições, etc;
- ImageView: Elemento para utilização de imagens que serão exibidas nas telas do aplicativo;
- EditText: Caixas de texto que servirão de campo para o usuário entrar com os dados que serão manipulados no aplicativo;
- ListView: Elemento utilizado para listar os dados adicionados pelo usuário;
- ScrollView: Utilizado quando desejamos colocar mais elementos em uma tela, do que sua capacidade de tamanho suporta, permitindo que o layout seja deslocado para poder suportar a adição de todos os elementos necessários para o seu funcionamento.

3.4. Escrituração zootécnica e índices da ovinocultura

Nesta seção serão apresentados os conceitos de escrituração zootécnica, e de como ela é feita atualmente pelos produtores do setor.

Além disso, será apresentado quais as variáveis referentes a zootecnia serão utilizadas no desenvolvimento do aplicativo, e porque elas são importantes para fazer o gerenciamento do criadouro.

3.4.1. Escrituração zootécnica

A escrituração zootécnica é a maneira com que o produtor guarda os dados que são coletados a partir do seu rebanho. Tendo em mãos as coletas de dados, eles precisam ser armazenados para que os profissionais da área possam analisá-los quando for necessário.

Nos dias atuais a escrituração é feita em sua grande parte de maneira manual, em que o produtor coleta os dados de seu rebanho e os armazena em fichas físicas, no papel. Entretanto, existem também a maneira informatizada, com auxílio de computadores e softwares. É nesta segunda, que o aplicativo *Ovelheiros* se encaixa.

Grandes são os benefícios da escrituração informatizada, permitindo maior controle, detalhe e integração da informação, com disponibilização fácil e rápida para o usuário. (LÔBO, R. N. B)

3.4.2. Índices da ovinocultura utilizados

Para uma avaliação preliminar das condições nutricionais, sanitárias e de manejo do animal, bem como do rebanho, alguns dados são imprescindíveis, como a data em que os dados foram coletados, data de nascimento do animal, peso, escore de condição corporal (ecc), Famacha ©, opg (ovos por grama de fezes) e oopg (oocistos por grama de fezes). A seguir é apresentada uma breve explicação das variáveis mais importantes que foram utilizadas:

- O ecc é uma estimativa do estado nutricional em que se encontra o animal. Escala que classifica o animal, em função da avaliação visual e tátil, da cobertura muscular e da massa de gordura, e que vai do 1 (subnutrido) ao 5 (obeso).
- O método Famacha ©, é um método auxiliar ao índice de OPG, atuando no controle de *Haemoncus contortus*, tendo como princípio a avaliação baseada em uma cartela com uma escala de coloração da mucosa ocular, que vai de 1 (coloração de mucosa bem vermelha, praticamente sem traços de anemia) à 5 (mucosa apresenta palidez intensa, necessário que o animal receba suplementação alimentar). Indica subjetivamente a necessidade de vermifugação do animal ou não.
- Opg é baseado no *Haemoncus contortus, que* é um helminto que parasita o sistema gastrointestinal de ovinos, causando a *haemoncose*. A técnica consiste na contagem de número de ovos por gramas de fezes, auxiliando na tomada de decisão na recomendação do tratamento do rebanho com anti-helmíntico ou não, quando a contagem média limite for maior ou igual a 500.
- No oopg, o protozoário *Eimeria* é o causador da doença infecciosa *coccidiose*, desta forma, este índice avalia a forma dos oocistos presentes nas fezes dos animais e a quantidade. Sendo o valor médio limite de 3.

A observação destes dados irá auxiliar na tomada de decisões em qual direção o produtor deverá seguir para melhorar a condição do ambiente em que este animal está inserido, bem como do seu manejo.

4. MATERIAL E MÉTODOS

Neste capítulo será descrito de forma detalhada sobre o caminho percorrido para alcançar os objetivos deste trabalho, descrevendo sobre a criação dos layouts de cada tela e posteriormente como cada uma foi programada para funcionar de acordo com as necessidades descritas no capítulo 2.

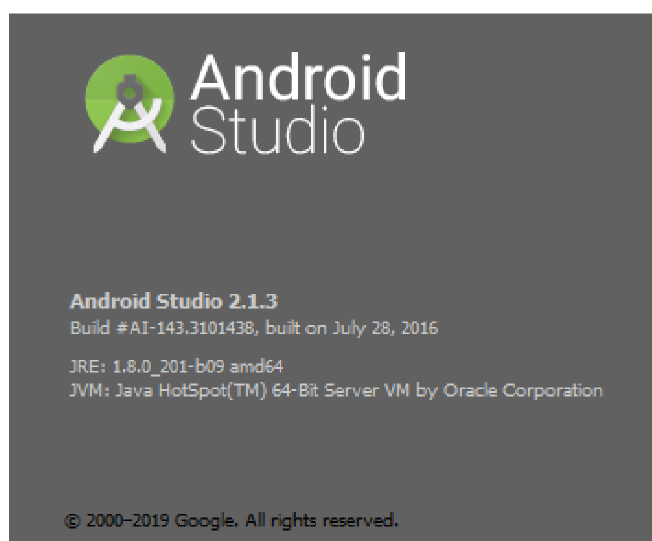
4.1. Metodologia

Este projeto foi todo executado no ambiente de desenvolvimento do Android Studio, versão 2.1.3, como mostrado na Figura 23.

Para garantir todo o funcionamento do aplicativo Ovelheiros de forma eficaz e organizada, as seguintes etapas de execução foram pré-estabelecidas:

- Desenvolvimento dos layouts das telas de acordo com as necessidades que o aplicativo requer;
- Implementação das funcionalidades das telas com a utilização da linguagem JAVA e das funções prontas da biblioteca do Android Studio.
- Implementação do Banco de Dados na biblioteca SQLite.

Figura 23 – Android Studio.



Fonte: Android Developers (2019).

4.1.1. Projeto das activities

Antes de iniciar o desenvolvimento dos layouts das *activities* propriamente ditas, foi necessário projetar quantas *activities* seriam necessárias para o cumprimento dos objetivos estipulados e como elas seriam feitas.

Sendo assim, foi necessário a criação e desenvolvimento de doze, divididas da seguinte forma:

- Uma tela contendo o menu principal;
- Uma tela contendo um sub menu;
- Quatro telas para o gerenciamento dos dados;
- Quatro telas de ajuda ao usuário;
- Uma tela para descrever o aplicativo e suas motivações;
- Uma tela de divulgação ao Grupo de Apoio à Ovinocultura (GAO) da Universidade Federal de Lavras.

4.1.2. Desenvolvimento dos layouts das activities

Tendo em mãos a quantidade de *activities* necessárias, iniciou-se o desenvolvimento dos layouts das mesmas, conforme a ordem estipulada.

Todos os layouts foram desenvolvidos com a utilização da linguagem XML com utilização do *Relative Layout* e com *backgroud* nas cores branca ou cinza, com exceção da tela de menu principal.

Para visualizar as telas depois de prontas, foi utilizado o emulador *Galaxy Nexus API 23*, já embutido no Android Studio.

4.1.2.1. Menu principal

A *activity* do menu principal conta com uma imagem de *background* que foi editada no software *photoshop 2015* versão de teste, além de contar com um TextView para o usuário que precisar de ajuda e três botões.

O primeiro deles é o Button “Entrar” responsável quando pressionado, por levar o usuário até a tela sub menu.

Já o segundo Button, direciona para a divulgação do Grupo de Apoio à Ovinocultura da Universidade Federal de Lavras. Enquanto o último, direciona para uma *activity* que contêm

um breve comentário sobre o aplicativo e os fatores que motivaram o seu desenvolvimento. A Figura 24 exibe como ficou a *activity* após o término.

Figura 24 – Activity menu principal.

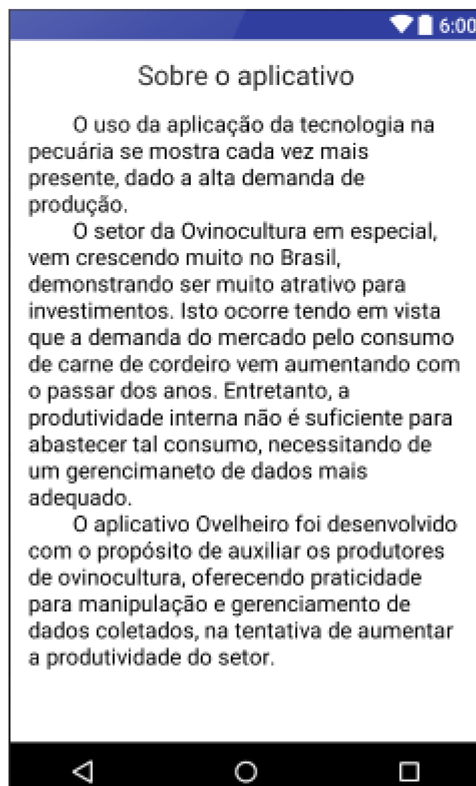
Fonte: Do Autor (2019).

4.1.2.2.Sobre o Aplicativo

Esta *activity* foi desenvolvida através de dois TextView, sendo o primeiro o título e o segundo um breve comentário sobre o desenvolvimento do aplicativo.

A Figura 25 exibe o modelo final da tela.

Figura 25 – Activity Sobre o aplicativo.



Fonte: Do Autor (2019).

4.1.2.3. Grupo de Apoio à Ovinocultura

Activity desenvolvida para divulgar o Grupo de Apoio â Ovinocultura (GAO) da Universidade federal de lavras.

Este layout foi desenvolvido através de quatro ImageView, uma sendo o logo do núcleo e os outros três, para representar três formas de contato do mesmo.

E para finalizar, foi utilizado um TextView para escrever o título do texto, e um ScrollView para redigir o texto institucional de apresentação do núcleo.

A imagem 26 mostra o resultado deste layout pronto após o desenvolvimento.

Figura 26 – Activity representativa do GAO.

Fonte: Do Autor (2019).

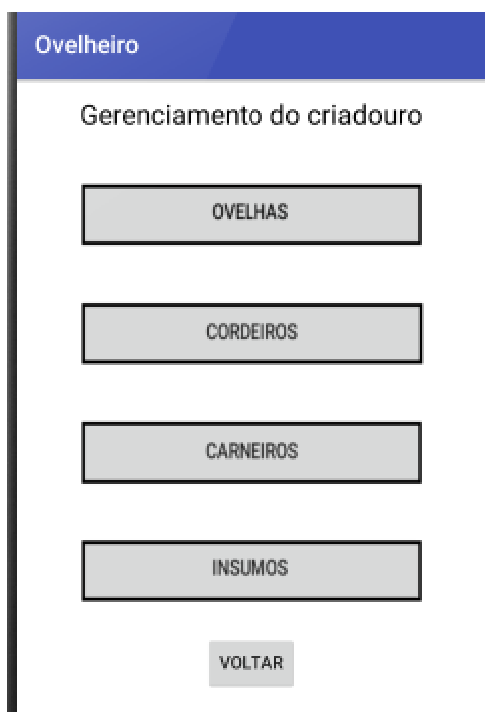
4.1.2.4. Gerenciamento do Criadouro

Nesta *activity*, o usuário poderá selecionar através de cinco Buttons, qual animal ele quer gerenciar: Ovelhas, Cordeiros ou Carneiros, respectivamente.

Também foi criado um botão para gerenciamento de insumos do criadouro e outro de menor dimensão, que se pressionado, leva o usuário novamente ao menu principal.

A Figura 27 mostra a tela final desenvolvida.

Figura 27 – Activity do Sub Menu Gerenciamento do criadouro.



Fonte: Do Autor (2019).

4.1.2.5. Cadastramento de Dados

Esta é a tela onde o usuário entrará com os dados da ovelha na qual ele deseja adicionar, salvando-os no banco criado. Para isso, a tela conta com nove EditTexts, cada um com um modelo escrito dentro de si, de como o usuário deve digitar os dados a serem salvos.

Para salvar os dados, foi criado um Button, juntamente com outros três. Um para consultar os dados adicionados, outro para voltar à *activity* de gerenciamento de criadouro e um último para requisição de ajuda ao usuário.

Foram também utilizados dois TextView para serem escritos o título e o subtítulo da tela, respectivamente.

E para finalizar, todos os itens foram colocados dentro de um ScrollView, de forma que todos os elementos possam ser acessados e visualizados pelo usuário.

A Figura 28 mostra esta tela pronta.

Figura 28 – Activity Cadastrar Dados de uma Ovelha.

Fonte: Do Autor (2019).

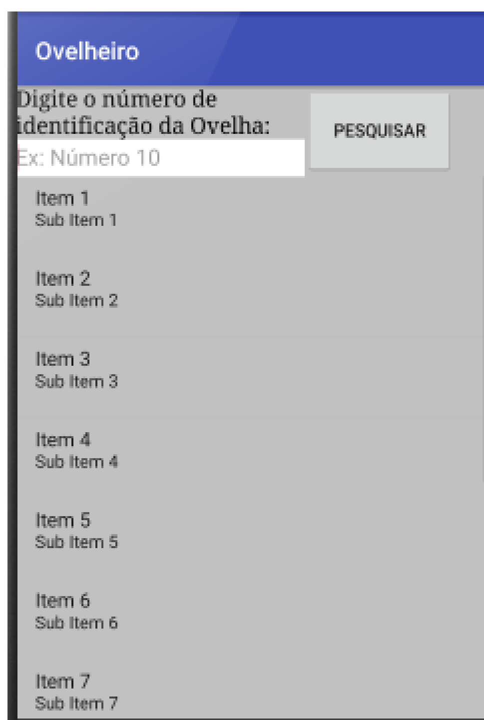
4.1.2.6. Consulta ao Banco de Dados

Esta tela está sendo responsável por exibir todos os dados armazenados no banco e também para consultar o histórico de determinada ovelha, pesquisando pelo número de identificação da mesma.

Ela conta com um Button para fazer a pesquisa, um EditText com dica de preenchimento para o usuário pesquisar a ovelha, um TextView com o texto indicativo da busca a ser feita. E para finalizar, um ListView onde será impresso os dados cadastrais das ovelhas conforme forem adicionadas, sendo que os últimos dados adicionados, fiquem no topo da lista.

A Figura 29 mostra a tela na versão final após o seu desenvolvimento.

Figura 29 – Activity Consulta Banco de Dados



Fonte: Do Autor (2019).

4.1.2.7. Atualização e Remoção de Dados

Nesta *activity* foi desenvolvida de forma semelhante ao layout desenvolvido na tela de cadastramento de dados de uma ovelha, foram utilizados nove EditTexts para a entrada dos dados da ovelha. Só que aqui, ao invés de serem para adicionar, serão para remover a ovelha em questão.

Além de um TextView para o título da tela, há também quatro botões, sendo eles para remover dados, editar dados, voltar à tela de cadastramento ou requisitar por ajuda, respectivamente.

A Figura 30 mostra a tela na versão final após o seu desenvolvimento.

Figura 30 – Activity Atualiza ou Remove Dados.

Fonte: Do Autor (2019).

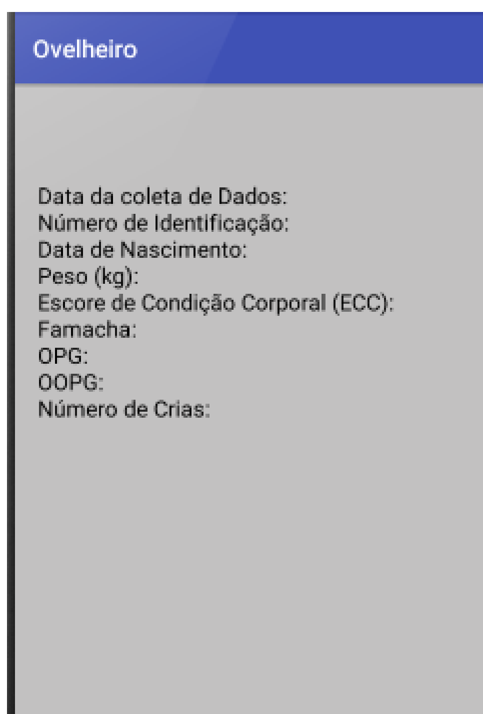
4.1.2.8. Dados impressos na ListView

Esta tela foi desenvolvida apenas com o intuito de servir como um "Item" no ListView criado na *activity* de consulta de banco de dados, de forma que quando o usuário cadastra novos dados, esta tela é concatenada com a lista. Sendo assim, essa *activity* não poderá ser vista por si só, enquanto o aplicativo estiver em funcionamento.

É uma tela simples, contendo só TextView para representação de cada tipo de dado da ovelha.

A Figura 31 mostra o layout desta tela desenvolvida.

Figura 31 – Layout representativo dos dados que serão concatenados no ListView.

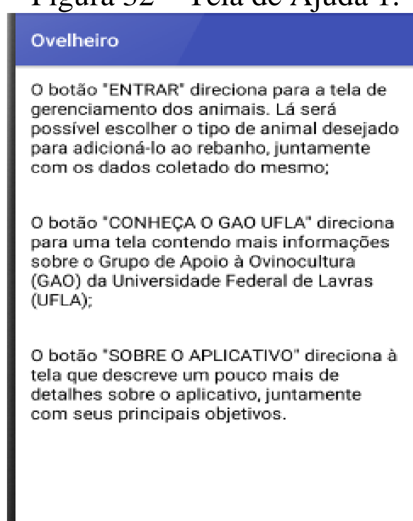


Fonte: Do Autor (2019).

4.1.2.9. Telas de Ajuda

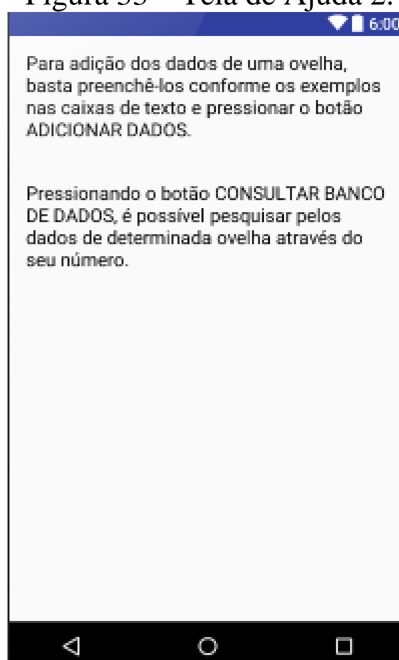
Todas as telas de ajuda foram desenvolvidas apenas com TextView e com escritas breves e diretas para oferecer auxilio ao usuário, caso o mesmo precise de ajuda para realizar alguma tarefa no aplicativo. As Figuras 32, 33 e 34, mostram a tela de ajuda para o menu principal, para cadastramento de dados e para remoção ou atualização de dados, respectivamente.

Figura 32 – Tela de Ajuda 1.



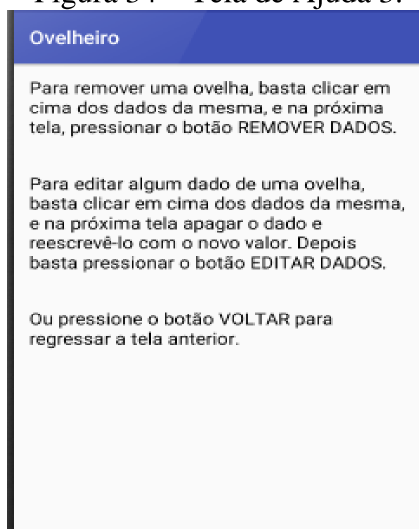
Fonte: Do Autor (2019).

Figura 33 – Tela de Ajuda 2.



Fonte: Do Autor (2019).

Figura 34 – Tela de Ajuda 3.



Fonte: Do Autor (2019).

4.1.3. Programação das activities utilizando a linguagem Java

Após a implementação dos layouts das *activities*, torna-se necessário programá-las para que elas realizem os funcionamentos de maneira eficiente e adequada para que os objetivos estipulados sejam cumpridos. Para isso, será utilizado a linguagem java e também a biblioteca SQLite para fazer o gerenciamento dos dados.

4.1.3.1. Criação da classe Ovelha

Como o trabalho se dá a partir do gerenciamento dos dados de um rebanho de ovelhas, é necessário primeiramente, criar uma classe com os atributos e métodos necessários para isso. O atributo *id* será responsável por fornecer a identificação de uma ovelha ao banco de dados, para que o mesmo a inicialize. Sendo assim, criou-se uma classe chamada *Ovelha*, que será responsável por receber estes atributos que foram explicados na seção 3.4, bem como os métodos *get* e *set* para cada um.

Após estes passos, é necessário implementar a interface *Serializable*, que é responsável por serializar a classe *Ovelha*, de forma que ela é encapsulada em códigos binários, tornando viável a sua utilização em outra classe.

A Figura 35 nos mostra a classe criada com os seus atributos, implementando a interface *Serializable* e uma exemplificação de alguns de seus métodos.

Figura 35 – Classe Ovelha.

```
public class Ovelha implements Serializable {

    private int id;
    private String dataColheitaDados;
    private String numeroOvelha;
    private String dataNascimento;
    private String peso;
    private String ecc;
    private String famacha;
    private String opg;
    private String oopg;
    private String numCrias;


    public int getId() { return id; }

    public void setId(int id) { this.id = id; }

    public String getDataColheitaDados() { return dataColheitaDados; }

    public void setDataColheitaDados(String dataColheitaDados) {...}

    public String getNumeroOvelha() { return numeroOvelha; }

    public void setNumeroOvelha(String numeroOvelha) { this.numeroOvelha = numeroOvelha; }

    public String getDataNascimento() { return dataNascimento; }

    public void setDataNascimento(String dataNascimento) { this.dataNascimento = dataNascimento; }
```

Fonte: Do Autor (2019).

4.1.3.2. Criação do Banco de Dados

Para criar o banco de dados e suas operações, foi criada uma classe chamada *DataBaseHelper* para se tornar subclasse da classe *SQLiteOpenHelper*, que oferece os métodos necessários para se gerenciar os dados.

Os atributos desta classe devem ser todos estáticos para demonstrar que não podem ser alterados após sua criação e a utilização da palavra reservada *final* também reforça esta propriedade.

A Figura 36 mostra a criação do banco de Dados, juntamente com os seus atributos.

Figura 36 – Classe DataBaseHelper utilizada para criação do Banco de Dados.

```
public class DataBaseHelper extends SQLiteOpenHelper {

    public static String DATABASE_NAME = "sheep_database";
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_SHEEP = "sheeps";
    private static final String KEY_ID = "id";
    private static final String KEY_DATA_DADOS = "datadados";
    private static final String KEY_NUMERO_OVELHA = "numovelha";
    private static final String KEY_DATA_NASCIMENTO = "nascimento";
    private static final String KEY_PESO = "peso";
    private static final String KEY_ECC = "ecc";
    private static final String KEY_FAMACHA = "famacha";
    private static final String KEY_OPG = "opg";
    private static final String KEY_OOPG = "oopg";
    private static final String KEY_NUMERO_CRIAS = "crias";
```

Fonte: Do Autor (2019).

Após este passo, foram implementados os métodos *onCreate* e *onUpdate*, necessários para o funcionamento do gerenciador de dados, juntamente com o construtor da classe.

4.1.3.2.1 Criação da Tabela

É importante criar uma tabela para organizar os atributos criados, em colunas de uma tabela, facilitando a localização e manipulação destes dados através de seu *Id*. Para a criação da tabela *TABLE_SHEEP*, foi utilizado o comando *CREATE TABLE* e posteriormente a definição das colunas, em que cada uma, seria representada por um atributo criado anteriormente.

A Figura 37 mostra a criação desta tabela.

Figura 37 – Tabela TABLE_SHEEPS.

```
//Criação da Tabela com as respectivas colunas
private static final String CREATE_TABLE_SHEEPS = "CREATE TABLE "
        + TABLE_SHEEP + "(" + KEY_ID
        + " INTEGER PRIMARY KEY AUTOINCREMENT," + KEY_DATA_DADOS + " TEXT, " + KEY_NUMERO_OVELHA + " TEXT, " + KEY_DATA_NASCIMENTO +
        " TEXT, " + KEY_PESO + " TEXT, " + KEY_ECC + " TEXT, " + KEY_FAMACHA + " TEXT, " + KEY_OPG + " TEXT, " + KEY_OOPG + " TEXT, "
        + KEY_NUMERO_CRIAS + " TEXT );";
```

Fonte: Do Autor (2019).

4.1.3.2.2 Criação dos métodos de operação

Tendo em mãos o banco de dados e os atributos de entrada organizados em uma tabela, é preciso implementar os métodos para manipulação destes dados. Foram criados então, quatro métodos para realizar o gerenciamento. O primeiro para adicionar os dados de entrada, o

segundo para removê-los, o terceiro para o usuário editar algum dado caso necessário e o quarto para buscar uma ovelha através do seu número de identificação. Foi criado também um *ArrayList* para armazenar as ovelhas criadas na classe *Ovelha*, para fazer as operações necessárias. Assim, com a utilização do método *getReadableDatabase* e uma estrutura de repetição *while*, é possível sempre verificar no banco de dados se alguma operação nova é chamada, utilizando um objeto cursor, instanciado a partir da classe *Cursor*.

4.1.3.2.2.1. Adição de dados

Para esta operação, foi implementado um método chamado *addSheepDetail*. A adição de dados foi possível graças a utilização do método *getWritableDatabase* da API Android, pois com a chamada dele, mostramos que queremos que algum dos dados sejam escritos no nosso banco e da classe *ContentValues*, para salvá-los. Para finalizar, o método *insert* é utilizado para o retorno dos dados.

A Figura 38 mostra a implementação final do método para adição de dados.

Figura 38 – Método para adição de dados.

```
public long addSheepDetail(String dataColheitaDados, String numeroOvelha, String dataNascimento, String peso,
                           String ecc, String famacha, String opg, String oopg, String numCrias) {

    SQLiteDatabase db = this.getWritableDatabase();

    //Método put adiciona dados
    ContentValues contentValues = new ContentValues();
    contentValues.put(KEY_DATA_DADOS, dataColheitaDados);
    contentValues.put(KEY_NUMERO_OVELHA, numeroOvelha);
    contentValues.put(KEY_DATA_NASCIMENTO, dataNascimento);
    contentValues.put(KEY_PESO, peso);
    contentValues.put(KEY_ECC, ecc);
    contentValues.put(KEY_FAMACHA, famacha);
    contentValues.put(KEY_OPG, opg);
    contentValues.put(KEY_OOPG, oopg);
    contentValues.put(KEY_NUMERO_CRIAS, numCrias);
    long insert = db.insert(TABLE_SHEEP, null, contentValues);

    return insert;
```

Fonte: Do Autor (2019).

4.1.3.2.2.2. Remoção de dados

Com a implementação do método *deleteOvelha* é possível deletar todos os dados de uma ovelha selecionada, através do *Id* da mesma.

O código abaixo, ilustrado pela Figura 39, realiza esta tarefa.

Figura 39 – Método para remoção de dados.

```
public void deleteOvelha(int id) {
    SQLiteDatabase db = this.getWritableDatabase();
    db.delete(TABLE_SHEEP, KEY_ID + " = ?",
            new String[]{String.valueOf(id)});
}
```

Fonte: Do Autor (2019).

4.1.3.2.2.3. Edição de Dados

O método *updateOvelha* foi criado para realizar a edição de dados caso o usuário julgar necessário. A implementação é muito semelhante ao método de adição, com a diferença que o método retornado ao final da implementação é o *update*, que atualiza os valores digitados pela identificação de *id* da Ovelha.

A Figura 40 mostra a implementação final do método.

Figura 40 – Método para edição de dados.

```
public int updateOvelha(int id, String dataColheitaDados, String numeroOvelha, String dataNascimento, String peso,
                        String ecc, String famacha, String opg, String oopg, String numCrias) {
    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put(KEY_DATA_DADOS, dataColheitaDados);
    values.put(KEY_NUMERO_OVELHA, numeroOvelha);
    values.put(KEY_DATA_NASCIMENTO, dataNascimento);
    values.put(KEY_PESO, peso);
    values.put(KEY_ECC, ecc);
    values.put(KEY_FAMACHA, famacha);
    values.put(KEY_OPG, opg);
    values.put(KEY_OOPG, oopg);
    values.put(KEY_NUMERO_CRIAS, numCrias);

    return db.update(TABLE_SHEEP, values, KEY_ID + " = ?",
            new String[]{String.valueOf(id)});
}
```

Fonte: Do Autor (2019).

4.1.3.2.2.4. Consultar dados

Para realização desta tarefa, foi implementado o método *buscarOvelhaPorNúmero*, que como o próprio nome já diz, pesquisa uma ovelha por seu número de identificação e retorna os seus dados.

O método *rawQuery* procura na tabela a ovelha que possui o número pesquisado e a retorna quando encontrada.

A Figura 41 mostra a implementação deste método.

Figura 41 – Método para consultar dados por número de identificação da ovelha.

```
public Ovelha buscarOvelhaPorNumero (String numeroOvelha){

    Ovelha ovelha;
    SQLiteDatabase db = this.getReadableDatabase();

    Cursor cursor = db.rawQuery("SELECT datadados, numovelha, nacimento, peso, ecc, famacha, " +
            "opg, oopg, crias FROM usuario WHERE numovelha = ?", new String[] {numeroOvelha});
    cursor.moveToFirst();

    if(cursor.getCount() > 0){

        ovelha = new Ovelha();
        ovelha.setDataColheitaDados(cursor.getString(0));
        ovelha.setNumeroOvelha(cursor.getString(1));
        ovelha.setDataNascimento(cursor.getString(2));
        ovelha.setPeso(cursor.getString(3));
        ovelha.setEcc(cursor.getString(4));
        ovelha.setFamacha(cursor.getString(5));
        ovelha.setOpg(cursor.getString(6));
        ovelha.setOopg(cursor.getString(7));
        ovelha.setNumCrias(cursor.getString(8));
    } else {
        ovelha = null;
    }

    cursor.close();
    db.close();

    return ovelha;
}
```

Fonte: Do Autor (2019).

4.1.3.3. Classe CustomAdapter

Para personalizarmos no ListView de modo que ele consiga adaptar com que os dados exibidos na Figura 30, se relacionem com cada “item” mostrado através da Figura 28, a classe *CustomAdapter* deve ser criada e estender a superclasse *BaseAdapter*.

Esta superclasse possui quatro métodos que serão fundamentais para fazer esta tarefa. São eles: *getCount*, *getItem*, *getItemId* e *getView*.

A Figura 42 traz a implementação dos três primeiros, de forma comentada, descrevendo a função de cada um deles em relação ao ListView que será criado.

Além dos métodos, temos também os atributos referentes ao contexto da aplicação e o *ArrayList* que receberá a adição das ovelhas, juntamente com o construtor da classe.

Figura 42 – Classe CustomAdapter e seus três primeiros métodos.

```
public class CustomAdapter extends BaseAdapter {

    private Context context;
    private ArrayList<Ovelha> ovelhaArrayList;

    public CustomAdapter(Context context, ArrayList<Ovelha> ovelhaArrayList){

        this.context = context;
        this.ovelhaArrayList = ovelhaArrayList;
    }

    @Override
    //Método getCount é usado pelo ListView para saber quantos itens existem no ovelhaArrayList
    public int getCount() {
        return ovelhaArrayList.size();
    }

    //Faz com que cada Item do ListView receba os dados de acordo com a posição de um termo do ovelhaArrayList
    @Override
    public Object getItem(int position){
        return ovelhaArrayList.get(position);
    }

    //Espera o retorno de qual id do objeto selecionado. Neste caso não possui Id e retorna 0
    @Override
    public long getItemId(int position) {
        return 0;
    }
```

Fonte: Do Autor (2019).

O método *findViewById* é responsável por localizarmos cada componente que temos em uma *activity*, para podermos utilizá-los de forma adequada.

O quarto método da classe *BaseAdapter*, o *getView*, é o método necessário para definir o que será associado a cada item do ListView. Para personalizarmos a criação do nosso ListView de forma otimizada, precisamos utilizar a classe *ViewHolder*, que nos ajuda a adaptar os itens em cada item do ListView, sem que tenhamos que ficar implementando o código novamente para cada adição de dados.

Desta forma, precisamos escrever os textos que serão impressos nos itens do ListView somente uma vez, bem como os seus métodos *findViewById*. Se a classe *ViewHolder* não fosse utilizada, teríamos que implementar tudo para cada vez que uma ovelha fosse adicionada ao banco de dados, aumentando o peso computacional, o trabalho manual e também a ineficiência do sistema.

A Figura 43 mostra este último método da classe *CustomAdaptar* implementado.

Figura 43 – Implementação do método getView na classe CustomAdapter.

```
public View getView(int position, View convertView, ViewGroup parent) {
    ViewHolder holder;

    if (convertView == null){
        holder = new ViewHolder();
        LayoutInflater inflater = (LayoutInflater) context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        convertView = inflater.inflate(R.layout.activity_ovelhas, null, true);

        holder.tvdatadados = (TextView) convertView.findViewById(R.id.dataDadosId);
        holder.tvidentificacao = (TextView) convertView.findViewById(R.id.numIdentificacaoId);
        holder.tvnascimento = (TextView) convertView.findViewById(R.id.dataNascimentoId);
        holder.tvpeso = (TextView) convertView.findViewById(R.id.pesoId);
        holder.tvecc = (TextView) convertView.findViewById(R.id.eccId);
        holder.tvfamacha = (TextView) convertView.findViewById(R.id.famachaId);
        holder.tvopg = (TextView) convertView.findViewById(R.id.opgId);
        holder.tvoopg = (TextView) convertView.findViewById(R.id.oopgId);
        holder.tvcrias = (TextView) convertView.findViewById(R.id.numCriasId);
        convertView.setTag(holder);
    }else {
        holder = (ViewHolder)convertView.getTag();
    }

    holder.tvdatadados.setText("Data da coleta dos dados: " +ovelhaArrayList.get(position).getDataColheitaDados());
    holder.tvidentificacao.setText("Número de Identificação: " +ovelhaArrayList.get(position).getNumeroOvelha());
    holder.tvnascimento.setText("Data de nascimento: " +ovelhaArrayList.get(position).getDataNascimento());
    holder.tvpeso.setText("Peso (kg): " +ovelhaArrayList.get(position).getPeso());
    holder.tvecc.setText("Escore de Condição Corporal (ECC): " +ovelhaArrayList.get(position).getEcc());
    holder.tvfamacha.setText("Famacha: " +ovelhaArrayList.get(position).getFamacha());
    holder.tvopg.setText("OPG: " +ovelhaArrayList.get(position).getOpg());
    holder.tvoopg.setText("OOPG: " +ovelhaArrayList.get(position).getOopg());
    holder.tvcrias.setText("Número de crias: " +ovelhaArrayList.get(position).getNumCrias());
```

Fonte: Do Autor (2019).

4.1.3.4. Classe GetAllSheeps

Esta classe foi implementada para a criação do ListView que receberá os métodos implantados na classe *CustomAdapter*.

A Figura 44 nos mostra como isso foi implementado na classe *GetAllSheeps*.

Figura 44 – Implementação de um Adaptador para o ListView da classe GetAllSheeps.

```
customAdapter = new CustomAdapter(this,ovelhaArrayList);
listView.setAdapter(customAdapter);
```

Fonte: Do Autor (2019).

4.1.3.5. Programação de funcionamento dos EditTexts

Para iniciar o funcionamento dos EditTexts, foi necessário primeiramente conseguir recuperar os dados digitados pelo usuário. Para isso, foi utilizado o método *setText*, recebendo os métodos *get* implementados na classe *Ovelha* para salvar os dados digitados.

Posteriormente, é necessário converter os dados em *Strings*, para que os valores digitados possam ser impressos e lidos pelo usuário.

A Figura 45 exemplifica como a implementação de funcionamento do EditText de entrada para o número de identificação de uma ovelha, foi feita, seguindo o modelo explicado acima, sendo que esta forma, foi aplicada em todos os EditText do aplicativo.

Figura 45 – Implementação para o EditText do número de uma ovelha.

```
etNumIdentificacao.setText(ovelha.getNumeroOvelha());
etNumIdentificacao.getText().toString()
```

Fonte: Do Autor (2019).

4.1.3.6. Mudanças de activities

Para mudar-se de *activity* através de um clique em um Button ou ImageView, é necessário a utilização da classe *Intent*.

A Figura 46 mostra como isso é possível, através de um evento de clique aplicado à um Button. Para isso utiliza-se do método *SetOnClickListener*.

Figura 46 – Utilização da classe Intent para mudança de activity.

```
//Evento clique aplicado ao botão para ir a activity do GAO através do Menu Principal (MainActivity)
botaoGao.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intentt = new Intent(MainActivity.this, Gao.class);
        //Inicializa a mudança de Activity
        startActivity(intentt);
    }
});
```

Fonte: Do Autor (2019).

4.1.3.7. A classe Toast

A classe *Toast* é utilizada quando queremos imprimir alguma mensagem de texto na tela, quando algum evento específico acontece.

A Figura 47 mostra um exemplo de utilização desta classe.

Figura 47 – Utilização da classe *Toast* para confirmar que os dados foram salvos.

```
//Utilização da classe Toast com os seguintes parâmetros:
//Activity em que será aplicado, mensagem que será exibida na tela e duração da mensagem
//O método show é usado para exibir a mensagem
Toast.makeText(TelaOvelha.this,"Dados Salvos com Sucesso!",Toast.LENGTH_SHORT).show();
```

Fonte: Do Autor (2019).

5. RESULTADOS

Após todas as implementações desenvolvidas conforme a descrição no capítulo 4, notou-se que o aplicativo *Ovelheiros* cumpriu bem o seu papel, realizando todos os objetivos que foram estabelecidos no capítulo 2.

5.1. Navegação entre activities

A navegação entre as *activities* funcionou de forma satisfatória. Os botões ao serem pressionados pelo usuário, cumprem bem a suas funções de direcionar o usuário para outras telas, conforme a necessidade.

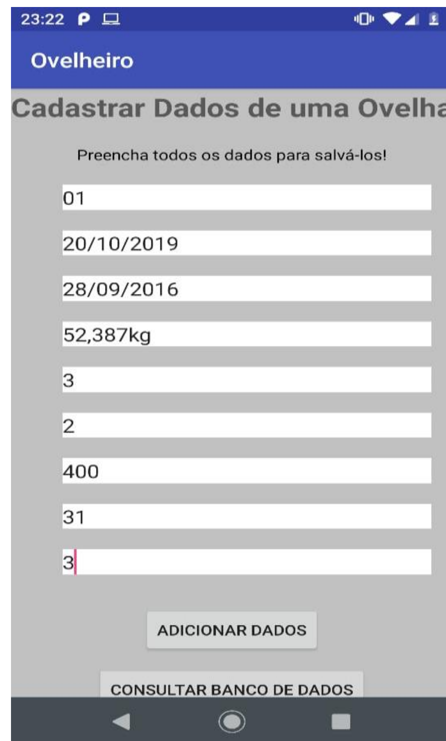
5.2. Operações de gerenciamento do banco de dados

Para esta seção, foram feitos testes utilizando as quatro operações básicas pré-estabelecidas para comprovar o bom funcionamento do aplicativo. Para os testes, foram utilizados dados fictícios, porém, que não fogem do intervalo de normalidade para uma ovelha.

5.2.1. Armazenamento de dados

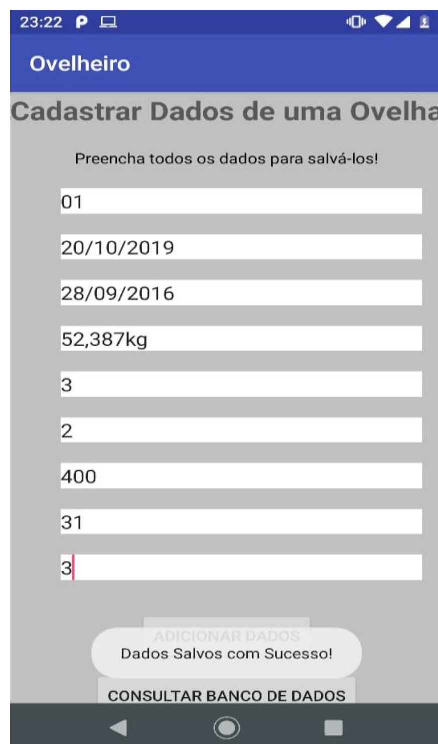
Ao digitar os dados referente a ovelha conforme a Figura 48, e pressionar o botão ADICIONAR DADOS, a mensagem de confirmação irá aparecer conforme a Figura 49. Após o salvamento dos dados, foi pressionado o botão CONSULTAR BANCO DE DADOS, mostrando que os dados referentes a primeira ovelha adicionada já estavam disponíveis para visualização, como é mostrado na Figura 50. Desta forma, comprovou-se que o salvamento de dados foi efetivo.

Figura 48 – Entrando com os dados da primeira ovelha.



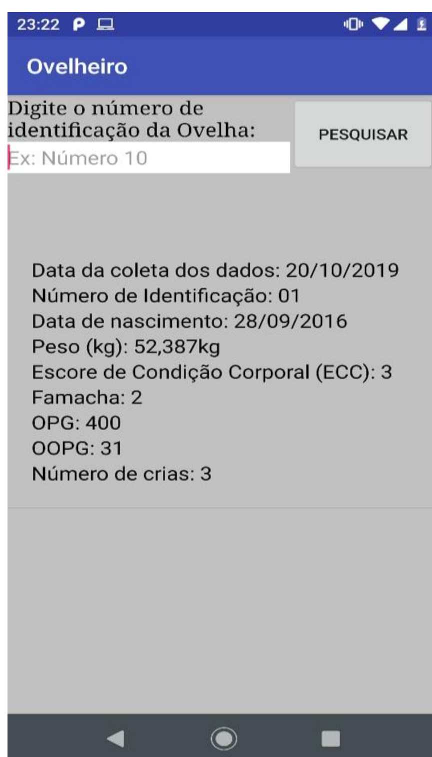
Fonte: Do Autor (2019).

Figura 49 – Confirmação da operação de salvamento de dados.



Fonte: Do Autor (2019).

Figura 50 – Consultando os dados adicionados.



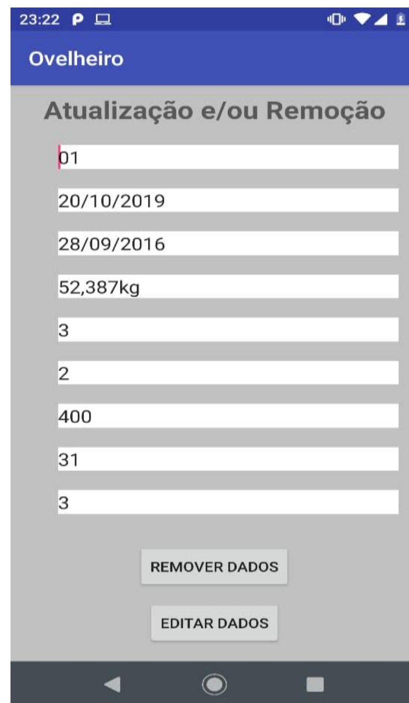
Fonte: Do Autor (2019).

5.2.2. Edição de dados

Ao clicarmos sobre os dados adicionados na Figura 48, abriremos a tela de remoção e atualização de dados. Podemos perceber que os dados que foram adicionados, já vêm preenchidos nos campos de entrada, para realmente confirmar se alguma das operações será realmente executada, conforme mostra a Figura 51.

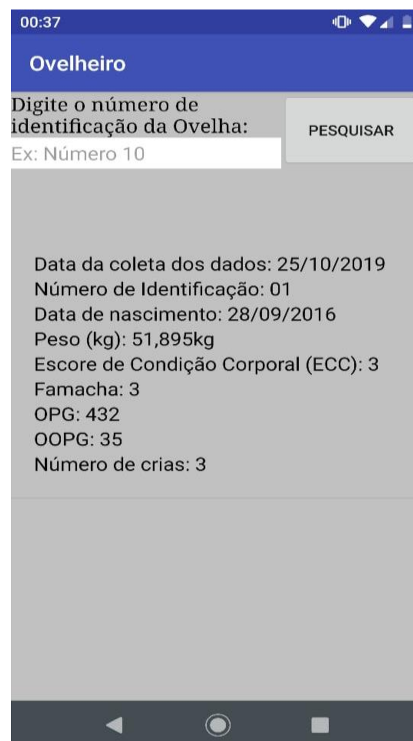
Visando testar esta funcionalidade, as variáveis “data da coleta dos dados”, “peso”, “Famacha ©”, “opg” e “oopg”, tiveram seus valores iniciais alterados. Assim, foi pressionado o botão EDITAR DADOS, depois o botão VOLTAR e depois o botão CONSULTAR BANCO DE DADOS novamente. A operação de edição também foi realizada com sucesso, conforme mostrado na Figura 52.

Figura 51 – Tela de remoção e atualização de dados com dados selecionados.



Fonte: Do Autor (2019).

Figura 52 – Dados da ovelha 01 alterados e salvos.



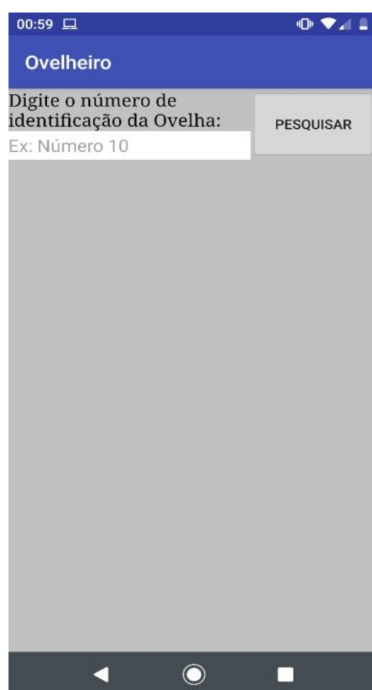
Fonte: Do Autor (2019).

5.2.3. Remoção de dados

Após a edição de dados, foi pressionado os dados na lista da Figura 52 novamente, voltando para a tela representada na Figura 51, mas desta vez, foi pressionado o botão REMOVER DADOS.

Conforme observado na Figura 53, os dados referentes a ovelha 01 foram removidos com êxito, deixando o banco de dados vazio.

Figura 53 – Banco de dados vazio após a remoção da ovelha número 01.



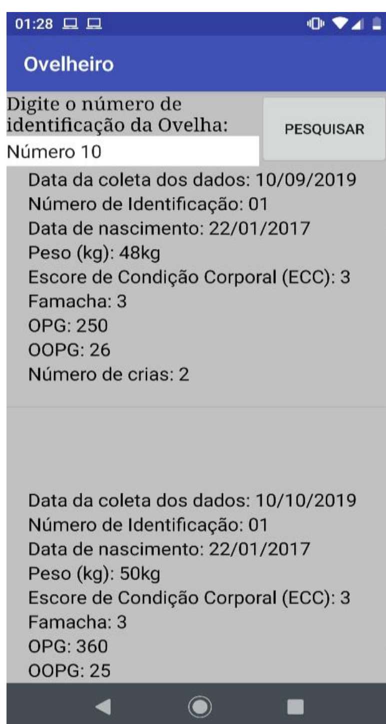
Fonte: Do Autor (2019).

5.2.4. Pesquisa de dados

Para testar se esta funcionalidade, foram adicionadas duas amostras de dados de uma mesma ovelha número 01 e uma amostra de outra ovelha diferente, com número de identificação número 02.

Ao ser inicializado a pesquisa somente para os dados da ovelha com o número de identificação 01 e pressionamento do botão PESQUISAR, houve êxito desta funcionalidade também, conforme mostrado na Figura 54.

Figura 54 – Método de pesquisa em funcionamento.



Fonte: Do Autor (2019).

5.3. Viabilidade para uso do ovinocultor

O aplicativo mostrou-se eficiente ao realizar todas as tarefas de gerenciamento de dados. Com a utilização da biblioteca SQLite, ficou bem funcional para utilização do produtor no campo, pois o SQLite não necessita de serviço cliente servidor e também não depende de conexão à rede internet para funcionar. Entretanto, a capacidade de armazenamento varia de acordo com a capacidade de memória do aparelho que estará sendo utilizado.

Tendo em vista que muitos lugares no campo são isolados das grandes cidades, e por consequência nem sempre tendo conexão à internet, o aplicativo pode vir a se tornar uma opção viável ao produtor que deseja armazenar e gerenciar os dados de seu rebanho de ovinos, tendo em vista que a escrituração zootécnica feita atualmente, ainda é muito arcaica.

5.4. Projetos futuros

Funcionalidade para gerenciamento de todos os animais do rebanho, pois até o momento, o aplicativo foi desenvolvido apenas para ovelhas. Apesar do sub menu possuir a

indicação de outros animais do rebanho e também insumos, o *Ovelheiros* ainda não possui funcionamentos para os mesmos.

Apesar das variáveis utilizadas serem importantes, ainda são poucas em relação a grande gama de dados coletados dos rebanhos nos dias de hoje. O objetivo foi mostrar que é viável e funcional o desenvolvimento com poucas variáveis, significa que com mais também será possível.

Ficar entrando com dados a todo momento por uma tela pequena de celular, pode se tornar um processo muito cansativo para o usuário. É viável então, futuramente ser desenvolvido um banco de dados externo, de forma que o dispositivo móvel e um desktop estejam ligados a ele. Desta forma, fica viável que os dados sejam escritos pelo desktop em algum lugar mais confortável para o usuário como o seu escritório por exemplo, enquanto o aplicativo seja utilizado somente para fazer consulta em tempo real ao lado do animal no campo, assim fica mais fácil analisá-lo, caso ele apresente algum transtorno ou doença.

Gerar gráficos e relatórios, para que o profissional da área possa detectar o ponto exato em que será necessário à aplicação de alguma medida veterinária ou zootécnica ao animal analisado.

6. CONCLUSÃO

Este trabalho apresenta de forma bem detalhada o desenvolvimento de um aplicativo Android simples e prático, todo implementada na linguagem java orientada à objetos e ideal para ser utilizado segundo os seus propósitos de criação. Mesmo sendo fácil de utilizá-lo, todas as telas mais funcionais apresentam centrais de ajuda ao usuário, com textos breves, diretos e simplificados, visando auxiliar na manipulação de cada uma delas.

É ideal para utilização em qualquer lugar do campo, mesmo em zonas mais afastadas dos grandes centros urbanos, tendo em vista que a conexão à internet pode ser dispensando para a utilização. Isso tudo graças a biblioteca SQLite que dispensa este recurso.

Entretanto, o *Ovelheiro* é só um desenvolvimento de teste experimental, tendo em vista que como foi citado nos resultados, o mesmo possui muitas melhorias a serem feitas até que seja ideal para utilização comercial. Ainda assim, o seu desenvolvimento mostrou o seu potencial, sendo um aplicativo móvel Android que possa atender as necessidades do produtor.

Portanto, fica claro com este trabalho a importância da utilização de novas tecnologias no campo, e a gama de processos que podem ser automatizados com o auxílio tecnológico, não só de maquinário pesado, mas sim, com a utilização de aplicativos móveis, que estão cada vez mais difundidos nos dias de hoje.

REFERÊNCIAS

ANDROID DEVELOPERS. **Layouts.** Disponível em: <https://developer.android.com/guide/topics/ui/declaring-layout?hl=pt-br>. Acesso em: 21 de out. 2019.

APOSTILA CAELUM. **Java e Orientação à Objetos.** Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/>. Acesso em: 20 de out. 2019.

CLARO, D.B.; SOBRAL, J. B. M. **Programação em JAVA**. Livro programando em Java 1ª edição, p.12, 2008.

COELHO, T. O.; PRADO, G. P. **Análise Comparativa para Avaliação de Tecnologias de Banco de Dados para Dispositivos Móveis.** Caderno de Estudos em Sistemas de Informação, 1(1), 2015.

DEVMEDIA. **Gerenciamento de Banco de Dados: Análise Comparativa de SGBD's.** Disponível em: <https://www.devmedia.com.br/gerenciamento-de-banco-de-dados-analise-comparativa-de-sgbd-s/30788>. Acesso em: 21 de out. 2019.

ELMASRI, R.; NAVATHE, S. B. **Sistemas de Banco de Dados.** 5 ed. São Paulo: Ed. PEARSON. p.46, 2005.

ENGHOLM, H. **Análise e design orientados a objetos.** Novatec Editora, 2013.

GAÚCHO, F. **Java básico e intermediário.** Disponível em: <https://tadstudy.files.wordpress.com/2016/08/manual.pdf>. Acesso em: 21 de out. 2019.

IBGE. **Projeção da População do Brasil e das Unidades da Federação.** Disponível em: <https://www.ibge.gov.br/apps/populacao/projecao/>. Acesso em: 8 out. 2019.

IPEA. **Economia Agrícola.** Disponível em: <http://www.ipea.gov.br/cartadeconjuntura/index.php/tag/pib-do-setor-agropecuario/>. Acesso em: 8 out. 2019.

LÔBO, R. N. B. **Importância da Escrituração Zootécnica Para o Desenvolvimento da Caprino-Ovinocultura.** Disponível em: <http://srvgen.cnpc.embrapa.br/pagina/escrit.php>. Acesso em: 21 de out. 2019.

RICARTE, I. L. **Programação Orientada a Objetos: uma abordagem com Java**. Ed. Universidade Estadual de Campinas, Campinas, SP, Brasil. p.6, 2001.

MENGUE, F. **Curso de Java Básico.** Disponível em: <http://ftp.unicamp.br/pub/apoio/treinamentos/linguagens/java_basico.pdf >. Acesso em: 20 de out. 2019.

NARMATHA, M.; KRISHNAKUMAR, S.V. **Study on Android Operating System and Its Versions.** International Journal of Scientific Engineering and Applied Science, 2(2), pp.439-444, 2016.

OLIVEIRA, A. E.; MACIEL, V. V. **JAVA NA PRÁTICA**, p.74, 2002.

SEBRAE. **Saiba como aplicativos auxiliam nos negócios do campo.** Disponível em: <http://www.sebrae.com.br/sites/PortalSebrae/artigos/saiba-como-aplicativos-auxiliam-nos-negocios-do-campo,408c7383f9cbe410VgnVCM1000003b74010aRCRD>. Acesso em: 8 out. 2019.

STATCOUNTER. **Mobile Operating System Market Share Brazil.** Disponível em: **<**https://gs.statcounter.com/os-market-share/mobile/brazil**>.** Acesso em: 8 out. 2019.

TABOADA, L.; CÓRCOLES E. **Hiperconectados: En una relación estable con Internet.** Ed. Grupo Planeta Spain; Nov 24, 2015.

GOOGLE DEVELOPER TRAINING TEAM. **Android Developer Fundamentals Course. Learn to develop Android Applications.** Disponível em: <https://google-developer-training.github.io/android-developer-fundamentals-course-concepts/en/android-developer-fundamentals-course-concepts-en.pdf>. Acesso em: 22 de out. 2019.

WILLRICH, R. **Capítulo 4: Linguagens de Programação.** Disponível em: <http://algol.dcc.ufla.br/~monserrat/icc/Introducao_linguagens.pdf>. Acesso em: 17 de out. 2019.