Implementando Testes Automáticos para o Aplicativo Android AppAlpha: Um Relato de Experiência

Ana Paula de Lima Borges Silva[1]

Departamento de Ciências Exatas (DCX) –
Universidade Federal da Paraíba (UFPB) - Campus IV
Cep 58297-000 – Rio Tinto – PB – Brasil

ana.paula@dcx.ufpb.br

Abstract. *In the world of software development, one of the biggest challenges is the issue of product quality, including its maintainability. It is possible to support this process through automated tests, which aim to bring more security and quality during development. In this context, the present work aims to present an experience report on the implementation of unit and user interface tests in the AppAlpha Android application, describing this process, the difficulties encountered and lessons learned in order to support developers of Android Applications.*

Resumo. *No mundo de desenvolvimento de software um dos grandes desafios é a questão da qualidade do produto, incluindo sua manutenibilidade. É possível dar suporte a este processo através de testes automáticos, que visam trazer mais segurança e qualidade durante o desenvolvimento. Neste contexto, o presente trabalho visa apresentar um relato de experiência sobre a implementação de testes unitários e de interface do usuário no aplicativo Android AppAlpha, descrevendo esse processo, as dificuldades encontradas e as lições aprendidas para dar suporte a desenvolvedores de aplicativos Android.*

1. Introdução

Com a popularização das metodologias ágeis no meio de desenvolvimento de software, torna-se cada vez mais importante buscar formas de desenvolver rapidamente código com qualidade focando em aspectos como manutenibilidade. Testes automáticos são uma das estratégias usadas para alcançar esse objetivo.

Ao desenvolver sistemas Android, como o sistema AppAlpha [Dantas, Borges e Dantas 2019], um aplicativo desenvolvido por meio do projeto de extensão Apps4Society, é importante explorar estratégias que garantam que o aplicativo possa ser mantido ao longo do tempo, sofrendo alterações de forma segura e ágil. Uma das estratégias para permitir isso é a implantação de testes automáticos.

Considerando esse cenário, o presente trabalho tem como objetivo buscar entender mais como aplicar testes automáticos em um sistema Android, visando relatar como foi feita a implementação de testes no AppAlpha e descrever os desafios, benefícios e as lições

[1] Trabalho de conclusão de curso, sob orientação da professora Ayla Débora Dantas de Souza Rebouças submetido ao Curso de Bacharelado de Sistemas de Informação do Centro de Ciências Aplicadas e Educação (CCAE) da Universidade Federal da Paraíba, como parte dos requisitos necessários para obtenção do grau de BACHAREL EM SISTEMAS DE INFORMAÇÃO.

aprendidas durante esse processo para dar suporte a outros desenvolvedores de aplicativos Android.

Para alcançar esse objetivo, as atividades previstas para este Trabalho de Conclusão de Curso foram divididas em 2 etapas. Na primeira foi feito um estudo sobre testes automáticos, que teve como finalidade entender o papel e as características dos mesmos no desenvolvimento de software. Na segunda etapa, foi iniciada a implementação dos testes automáticos na base de código existente do AppAlpha, junto com a documentação desse processo e das principais lições aprendidas durante o mesmo.

As demais seções deste artigo estão organizadas conforme descrito a seguir: a Seção 2 apresenta um referencial teórico relativo às ferramentas e técnicas aplicadas durante o desenvolvimento deste TCC; a Seção 3 apresenta a metodologia utilizada no trabalho; a Seção 4 apresenta o projeto que foi usado como base de código, o processo da implementação dos testes automáticos e as lições aprendidas durante o mesmo; e, por fim, a Seção 5 apresenta as considerações finais e propostas de trabalhos futuros.

2. Fundamentação Teórica

Nesta seção serão apresentados os conceitos e ferramentas utilizados no desenvolvimento deste trabalho, a fim de facilitar a compreensão do que foi realizado. Inicialmente, faremos uma breve contextualização do que são objetos de aprendizagem. A seguir, falaremos sobre testes.

2.1 Objetos de Aprendizagem para Alfabetização

Com a popularização dos meios digitais, a inclusão de dispositivos como tablets e smartphones na sala de aula se tornou uma oportunidade de deixar o ensino mais dinâmico. Segundo Wiley (2000), um Objeto de Aprendizagem (OA) é "qualquer recurso digital que pode ser reusado para suportar a aprendizagem". OAs ajudam a explorar o potencial da tecnologia e usá-la a favor do ensino.

Um dos focos dos OAs é a alfabetização. Segundo Freire (2003, p. 13), "A alfabetização é a criação ou a montagem da expressão escrita da expressão oral". Com isso, já existem alguns OAs que têm como objetivo ajudar nesse processo, seja através de sons, imagens e a conexão entre os dois.

Um exemplo é o Embaralhado, que de acordo com Silva (2017), "possui uma temática de apresentar de forma desorganizada as letras que constituem uma palavra que está associada a uma imagem. A partir disso, o usuário precisa clicar e arrastar as letras as soltando em espaços de formato quadricular de coloração azulada, de forma que ao final do desafio, a palavra esteja corretamente escrita". Na Figura 1 é possível ver o menu principal e a tela com um dos desafios desse aplicativo.

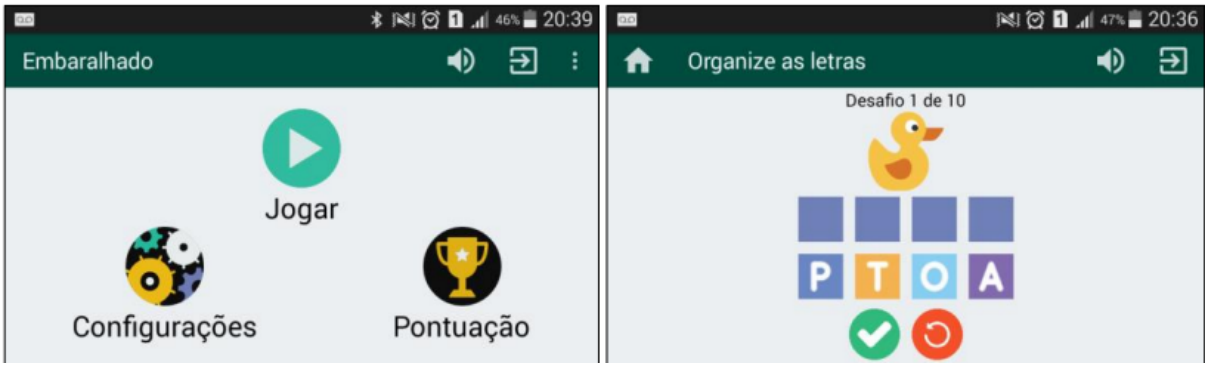

Figura 1. Menu e tela principal do Embaralhado

Com base em trabalhos como o de Silva (2017), foi criado o AppAlpha, que segundo Dantas, Borges e Dantas (2019), "visa apoiar o aprendizado de palavras e a familiarização dos estudantes com o alfabeto". O jogo se baseia no jogo da forca para induzir o usuário a fazer a conexão entre o som que pode ser reproduzido e a imagem que está sendo mostrada na tela. Mais detalhes sobre este aplicativo serão apresentados na Seção 4.1.

2.2 Testes automáticos

Durante o ciclo de desenvolvimento de software, 70% do tempo é gasto na compreensão do programa, como por exemplo em atividades de inspeção com o uso de *debugger* para depuração de código (Minelli, Mocci e Lanza, 2015). Para diminuir o tempo gasto nessa atividade e evitar defeitos, uma alternativa é explorar o teste de software, que tem como alguns dos seus objetivos, segundo Sommerville (2011, p.144), mostrar que um programa faz o que se propõe a fazer e descobrir os defeitos do programa antes do seu uso.

No entanto, testes podem demandar muito tempo, principalmente quando são apenas manuais. Para acelerar esse processo, recomenda-se utilizar testes automatizados, que podem ser definidos como o uso de software para controlar a execução de testes e a comparação entre os resultados esperados e os resultados reais obtidos [Avcioglu e Demirer 2015]. Por exemplo, a execução de testes de regressão e a configuração de ambiente pode ser feita de forma rápida, já que o propósito dos testes automatizados é acelerar os ciclos de execução, fornecer *feedback* imediato, liberar testadores de tarefas repetitivas e reduzir o esforço humano [Klammer e Ramler 2017].

Dentro da área de testes existem duas abordagens que orientam esse processo: a visão interna (teste de caixa branca) e a visão externa do produto (teste de caixa preta). De acordo com Pressman e Maxim (2016, p. 499-500), o teste de caixa preta examina alguns aspectos fundamentais de um sistema, com pouca preocupação em relação à estrutura lógica interna do software. Já o teste caixa branca é uma abordagem de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste. Exemplos de testes caixa branca são os testes unitários, conhecidos também como testes de componente ou módulo. O principal intuito deste método de testes, segundo Campos (2014), é verificar o correto ou incorreto funcionamento de cada unidade através de asserções entre valores devolvidos pelo bloco em teste e valores previamente esperados.

Uma ferramenta que se tornou muito popular para se escrever testes em Java é o JUnit. Foi criada no final dos anos 90 por Erich Gamma e Kent Beck, e rapidamente se tornou o padrão para desenvolver testes em Java [Massol e Husted 2004]. Através dela é possível

escrever asserções[2], predicados que são inseridos no programa para verificar uma condição que o desenvolvedor supõe que seja verdadeira em determinado ponto.

Na Listagem 1 é possível ver um exemplo de asserção em JUnit. É criado o objeto *Calculator* na linha 6, e na linha 10 é verificado se o retorno do método *add* com os parâmetros 1 e 1 é igual a 2.

```
1 import static org.junit.jupiter.api.Assertions.assertEquals;
2 import example.util.Calculator;
3 import org.junit.jupiter.api.Test;
4
5 class MyFirstJUnitJupiterTests {
6   private final Calculator calculator = new Calculator();
7
8   @Test
9   void addition() {
10      assertEquals(2, calculator.add(1, 1));
11  }
12
13 }
```

Listagem 1. Exemplo de asserção dentro de um teste feito com JUnit.
Fonte: JUnit User Guide (https://junit.org/junit5/docs/current/user-guide/#writing-tests)

3. Metodologia

Inicialmente, o objetivo geral deste trabalho foi o de relatar a experiência de implementar *Test Driven Development* no desenvolvimento de um aplicativo Android, apontando benefícios e desafios no processo. Para isso, fez-se necessário buscar fundamentação teórica sobre as práticas de refatoração, testes e TDD, entender os conceitos relacionados, a importância dessas práticas para o desenvolvimento e ser capaz de poder aplicá-las na base de código do aplicativo AppAlpha, descrito na Seção 4.1.

Contudo, durante a análise do código se percebeu que para implantar o processo de TDD em uma base de código já escrita se fazia necessário primeiramente desenvolver testes automáticos que irão assegurar que mudanças futuras não quebrem o código. Além disso, os testes desenvolvidos durante este trabalho e experiência obtida podem servir de inspiração para outros desenvolvedores implementarem testes antes de novas implementações em aplicativos Android.

Tendo isso em mente, se iniciou o processo de desenvolvimento dos testes automáticos para o aplicativo AppAlpha e se buscou identificar lições aprendidas e desafios para testar aplicativos Android. Para isso, este trabalho focou-se no teste para a fachada do aplicativo, que concentra suas principais funcionalidades e lógica, e no teste de classes do tipo *Activity* (comuns em aplicativos Android) e buscamos documentar a experiência para apoiar outros desenvolvedores de aplicativos Android.

[2] https://pt.wikipedia.org/wiki/Asserção

4. AppAlpha e Implementação de seus Testes

Nesta seção será feita a descrição da solução desenvolvida, começando com uma apresentação do AppAlpha, aplicativo que serviu de base de código para a implementação dos testes. Em seguida, serão apresentados os testes desenvolvidos e as lições aprendidas durante esse processo, além de alguns desafios encontrados.

4.1 AppAlpha

O aplicativo AppAlpha é um Objeto de Aprendizagem desenvolvido durante o projeto Apps4Society [Dantas, Borges e Dantas 2019], tendo como objetivo auxiliar o processo de alfabetização através do clássico jogo da forca. Para isso, é usada a conexão entre o som e letras para exercitar a soletração e a familiarização dos estudantes com o alfabeto. A primeira versão do AppAlpha foi desenvolvida pela autora deste trabalho por meio do projeto de extensão Apps4Society[3] e o aplicativo foi evoluído com a contribuição de outros discentes da UFPB, estando a versão atual disponível no repositório do projeto no GitHub[4].

Também foi feita uma avaliação do aplicativo a fim de verificar se o mesmo pode auxiliar na alfabetização de adultos em turmas de alfabetização, assim como uma avaliação com professores via questionário online. Os resultados demonstraram uma grande aceitação dos dois grupos (alunos adultos e professores), mostrando que o AppAlpha tem potencial para apoiar o processo de alfabetização.

Na Figura 2 é apresentado um dos fluxos de interação do usuário com o aplicativo usado durante as avaliações, e algumas telas do mesmo. Na tela 1, ao iniciar o jogo no botão de play o usuário é levado para a tela 2, de temas. Nela, existem seis opções de temas, agrupando desafios pelo contexto em comum. Ao escolher um, o mesmo é levado para a tela 3 com o desafio escolhido, onde é mostrada a forca, a imagem do desafio em questão, algumas linhas representando as letras a serem soletradas e um teclado com as letras do alfabeto.

Caso acerte a palavra ou erre 6 vezes, o usuário é levado para a tela 4 de resposta, onde a imagem da palavra é destacada e a mesma é soletrada. O ciclo das telas 3 e 4 são repetidos algumas vezes, e, por fim, é apresentada a tela 5 com a pontuação e a opção de salvar seu recorde.

[3] https://apps4society.dcx.ufpb.br/
[4] https://github.com/a4s-ufpb/AppAlpha

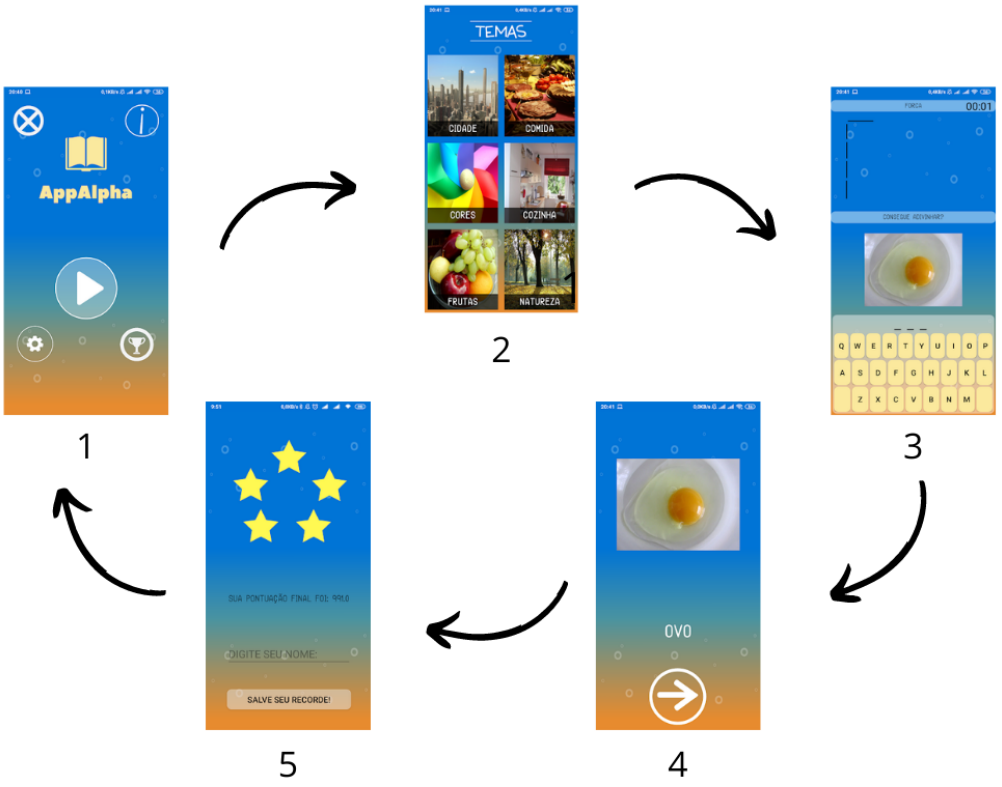

Figura 2. Fluxo de interação do usuário com o aplicativo

Na Figura 3 é possível ver a estrutura do projeto, que está dividida nos seguintes pacotes: *control*, *model* e *activities*.

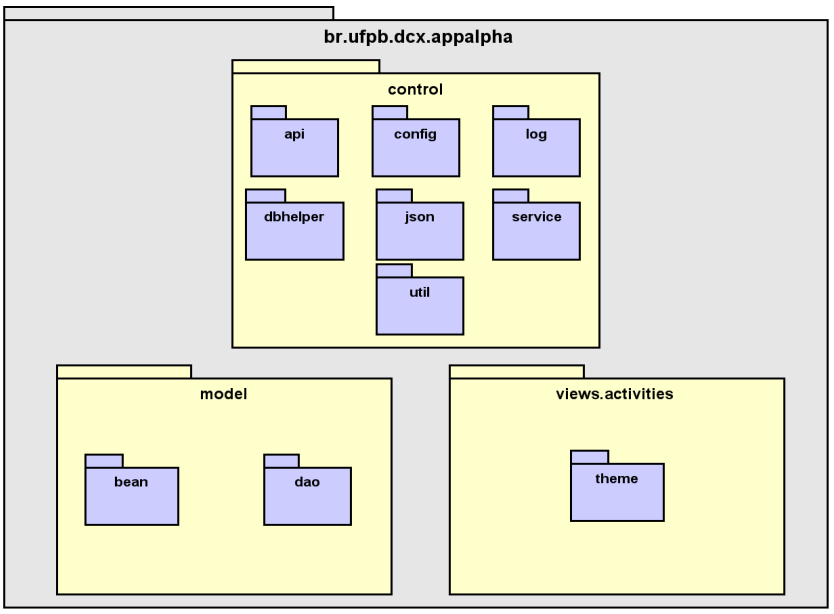

Figura 3. Estrutura dos pacotes no projeto

No pacote *control* ficam as classes relacionadas ao acesso aos desafios - tanto os que vêm pela API quanto os que são criados programaticamente - o registro de logs e as classes *utils*. No pacote *model* ficam as representações dos temas, desafios e recordes do usuário. No pacote *activities* ficam as telas do aplicativo, onde é possível ver e interagir com os temas e desafios escolhidos.

4.2 Implementação de Testes para a Fachada do Sistema

O *framework* utilizado para os testes unitários foi o JUnit 5, que é a *release* mais atual na data de escrita deste trabalho. Para permitir a execução dos testes no ambiente Android é necessário incluir o plugin android-junit5[5] no build.gradle do projeto, assim como mostrado na linha 8 da Listagem 2:

```
1 buildscript {
2    repositories {
3        google()
4        jcenter()
5    }
6    dependencies {
7        classpath 'com.android.tools.build:gradle:4.2.2'
8        classpath "de.mannodermaus.gradle.plugins:android-junit5:1.7.1.1"
9    }
10 }
```

Listagem 2. Importação do plugin do JUnit 5

Após isso, é necessário incluir a dependência do JUnit 5 e da sua *engine* no build.gradle do aplicativo, como mostra a Listagem 3:

```
// https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api
    testImplementation group: 'org.junit.jupiter', name:
'junit-jupiter-api', version: '5.8.1'
    testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.8.1'
```

Listagem 3. Importação das dependências do JUnit 5

O primeiro teste foi desenvolvido na fachada do sistema, que no caso do AppAlpha é a classe `ChallengeFacade`. Nela está contida toda a lógica de verificar se a tentativa do

[5] https://github.com/mannodermaus/android-junit5

usuário para aquela palavra está correta, quantos erros o mesmo cometeu até o momento e o tempo levado para cada desafio. Caso o mesmo acerte todas as letras, esta classe também é responsável por retornar um novo desafio.

Para criar o teste, foi usada a anotação do JUnit 5 chamada *@BeforeEach*, que faz com que o método onde foi usada essa anotação seja chamado antes de cada teste executado. O mesmo ficou responsável por criar os objetos necessários para inicializar o objeto `ChallengeFacade`, que é onde estão os métodos a serem testados. Na Listagem 4 é possível ver como foi feita essa configuração.

```
1 @BeforeEach
2     void setUp() {
3         Theme kitchenTheme = new Theme("cozinha",
Integer.toString(R.drawable.cozinha),
  Integer.toString(R.raw.cozinha), null);

4      Challenge spoonChallenge = new Challenge("colher", null,
Integer.toString(R.raw.colher), null,
Integer.toString(R.drawable.colher));

5        Challenge knifeChallenge = new Challenge("faca", null,
Integer.toString(R.raw.faca), null,
Integer.toString(R.drawable.faca));

6        challengeFacade = ChallengeFacade.getInstance();
7        challengeFacade.init(Arrays.asList(spoonChallenge,
knifeChallenge), kitchenTheme);
8    }
```

Listagem 4. Configuração feita antes de cada teste

Após isso foram criados os testes para cada método da classe. Em cada um foi usada a notação *@Test* para sinalizar que aquele era um teste a ser executado e também a *@DisplayName*, que serve para criar um título personalizado para cada um. No *@DisplayName* foi usado o conceito *Given-When-Then*[6], que visa criar um template de fácil entendimento para a documentação dos testes. No *Given* é descrito o contexto daquela situação de teste, suas características. O *When* é o gatilho daquela situação, o que é exercitado no teste, já o *Then* é o comportamento que se espera que seja verificado.

Esse template também foi usado na organização do código dentro do teste. Como mostrado na Listagem 5, primeiramente foi configurado o contexto na fase *Given*, após isso foi feita a ação descrita no *When* e no *Then* foi feita a asserção para verificar se o comportamento foi o esperado.

[6] https://abap101.com/2014/01/15/given-when-then-but-uma-maneira-simples-de-documentar-qualquer-coisa/

```
1 @DisplayName("Given user attempt a right letter for the word 'colher', " +
2            "When they click on it, " +
3            "Then it must return ATTEMPT_ACCEPTED")
4    @Test
5    void checkAttemptWithRightLetterTest() {
6        // Given
7        String letter = "o";
8        // When
9       int result = challengeFacade.checkAttempt(letter);
10       // Then
11       assertEquals(ChallengeFacade.ATTEMPT_ACEPTED, result);
12   }
```

Listagem 5. Teste organizado através do template Given-When-Then

Ao fim do processo de implementação desses testes foi possível chegar a uma cobertura de 72% dos métodos dessa classe, sendo os outros 28% correspondentes aos métodos *get* e *set* dos atributos da mesma. Para verificar a cobertura dos testes foi usada uma ferramenta própria do Android Studio. Para acessá-la, basta clicar na opção 'Run tests with Coverage', como mostrado na Figura 4.

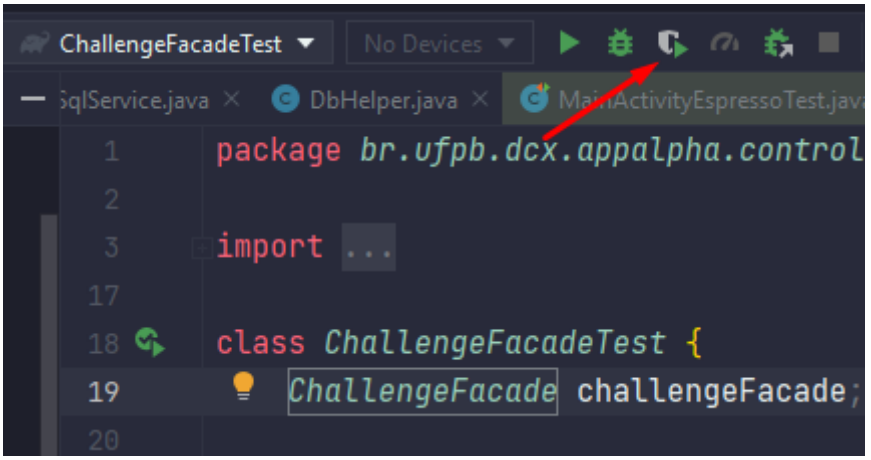

Figura 4. Opção de executar testes avaliando a Cobertura

4.3 Implementação de Testes para classes do tipo Activity

O *framework* utilizado para os testes da *activity* foi o Espresso[7], que faz parte da biblioteca de suporte para teste Android. Quando se cria um novo projeto Android pelo Android Studio, suas dependências já são importadas automaticamente, mas para importá-las manualmente é preciso adicionar as seguintes linhas no build.gradle do aplicativo, conforme ilustra a Listagem 6.

[7] https://developer.android.com/training/testing/espresso

```
androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.0'
androidTestImplementation 'androidx.test:runner:1.1.0'
androidTestImplementation 'androidx.test:rules:1.1.0'
```

Listagem 6. Dependências do framework Espresso

Ao criar a classe de teste da *activity* é preciso usar a notação *@RunWith* para sinalizar que é um teste e que vai ser rodado pela classe que foi passada em seu argumento, que nesse caso é a *AndroidJUnit4* (ver linha 13 da Listagem 7). Após isso, é preciso usar a notação *@Rule* para instanciar a *activity* como uma *ActivityScenarioRule* (linha 17 da Listagem 7), que é uma classe responsável por criar a *activity* antes do teste e fechá-la após o mesmo terminar.

Tendo configurado a classe de teste e a criação da *activity,* foi feito um teste para a mesma, também usando a notação *@Test* (linha 18 da Listagem 7). Nele, foi verificado se o botão de recordes - mostrado na tela 1 da Figura 2 - estava sendo mostrado na tela (linha 20 da listagem 7). Para isso, foram utilizados os seguintes métodos:

- *onView()*

  Cria uma visualização do componente de interface passado em seu parâmetro

- *withId()*

  Retorna o componente de interface que foi passado como um id pelo seu parâmetro

- *check()*

  Verifica se o componente passado em seu parâmetro é igual ao componente selecionado

- *matches()*

  Verifica se o componente existe na hierarquia da interface

- *isDisplayed()*

  Retorna o componente que está sendo mostrado na tela

```java
package br.ufpb.dcx.appalpha.view.activities;

import static androidx.test.espresso.Espresso.onView;
import static androidx.test.espresso.assertion.ViewAssertions.matches;
import static androidx.test.espresso.matcher.ViewMatchers.isDisplayed;
import static androidx.test.espresso.matcher.ViewMatchers.withId;

import androidx.test.ext.junit.rules.ActivityScenarioRule;
import androidx.test.ext.junit.runners.AndroidJUnit4;
import androidx.test.filters.LargeTest;

import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;

import br.ufpb.dcx.appalpha.R;


@RunWith(AndroidJUnit4.class)
@LargeTest
public class MainActivityEspressoTest {

    @Rule
    public ActivityScenarioRule<MainActivity> activityTestRule
            = new ActivityScenarioRule<>(MainActivity.class);

    @Test
    public void whenActivityIsLaunched_shoudDisplayGoToRecordsButton() {
        onView(withId(R.id.btnRecords)).check(matches(isDisplayed()));
    }

}
```

Listagem 7. Teste da classe *activity*

Para rodar esse tipo de teste é necessário ter um celular conectado ou mesmo um emulador que o próprio Android Studio oferece. No caso deste trabalho foi usado um celular, e antes de executá-lo é necessário fazer algumas configurações no mesmo:

- Ir em configurações -> opções do desenvolvedor -> permitir depuração e instalação via USB;
- Ainda em opções do desenvolvedor, desativar a escala de animação da janela, de transição e do Animator;
- Instalar o aplicativo que vai ser testado, ir em informações do aplicativo e permitir o início automático. Isso é necessário pois quando for instalado o apk com os testes o mesmo, de forma automatizada, abre o aplicativo e faz as verificações necessárias que são pedidas no teste.

Inicialmente, os testes foram rodados pela interface do Android Studio. Porém, esta indicava que nenhum teste era executado e não mostrava nenhum feedback do motivo. Após algumas pesquisas na internet foi descoberto o comando *gradlew connectedAndroidTest*, que também executa os testes mas gera um log dos métodos executados durante o mesmo.

Ao fim desse comando foi mostrado o erro "*No tests found. This usually means that your test classes are not in the form that your test runner expects (e.g. don't inherit from TestCase or lack @Test annotations).*", como mostrado na Figura 5. As pesquisas feitas mostraram que esse é um erro bastante genérico, não explicando exatamente qual o problema. Foram usados outros comandos que geram mais logs, como *--info*, *--debug* e *--stacktrace*, mas nenhum deles retornou a real causa da falha nos testes.

```
com.android.build.gradle.internal.testing.ConnectedDevice > No tests found.[M2101K7BNY - 11]
 FAILED
No tests found. This usually means that your test classes are not in the form that your test
 runner expects (e.g. don't inherit from T
estCase or lack @Test annotations).

> Task :app:connectedDebugAndroidTest FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':app:connectedDebugAndroidTest'.
> There were failing tests. See the report at: file:///C:/Users/User/Documents/github/AppAlp
ha/app/build/reports/androidTests/connecte
```

Figura 5. Erro durante a execução dos testes

Para verificar se o erro era algo relacionado à implementação do teste, foi criado um novo projeto do zero composto por uma tela inicial com dois *selects* e um botão. Todo o processo de importação das dependências do Espresso e da configuração da classe de teste foi similar ao anterior, como é possível ver na Listagem 8.

```
1  package com.example.consultasipacapp;

2  import static androidx.test.espresso.Espresso.*;
3  import static androidx.test.espresso.assertion.ViewAssertions.matches;
4  import static androidx.test.espresso.matcher.ViewMatchers.isDisplayed;
5  import static androidx.test.espresso.matcher.ViewMatchers.withId;

6  import androidx.test.ext.junit.rules.ActivityScenarioRule;
7  import androidx.test.ext.junit.runners.AndroidJUnit4;
8  import androidx.test.filters.LargeTest;

9  import org.junit.Rule;
10 import org.junit.Test;
11 import org.junit.runner.RunWith;



12 @RunWith(AndroidJUnit4.class)
13 @LargeTest
14 public class MainActivityTest {

15    @Rule
16    public ActivityScenarioRule<MainActivity>
mainActivityActivityScenarioRule
            = new ActivityScenarioRule<>(MainActivity.class);

17    @Test
18    public void
whenActivityIsLaunched_shoudDisplayCampusAndAidInputAndSubmitButton() {
19        onView(withId(R.id.campus_id)).check(matches(isDisplayed()));
20        onView(withId(R.id.aid_id)).check(matches(isDisplayed()));
21        onView(withId(R.id.submit_id)).check(matches(isDisplayed()));
22    }
23 }
```

Listagem 8. Implementação do teste no projeto novo

Na Figura 6 é possível ver que o teste foi executado com sucesso, mostrando que a implementação não está errada mas que pode ser algo relacionado a permissões de leitura e escrita no dispositivo e no banco de dados do aplicativo AppAlpha que pode estar levando ao problema anteriormente descrito.

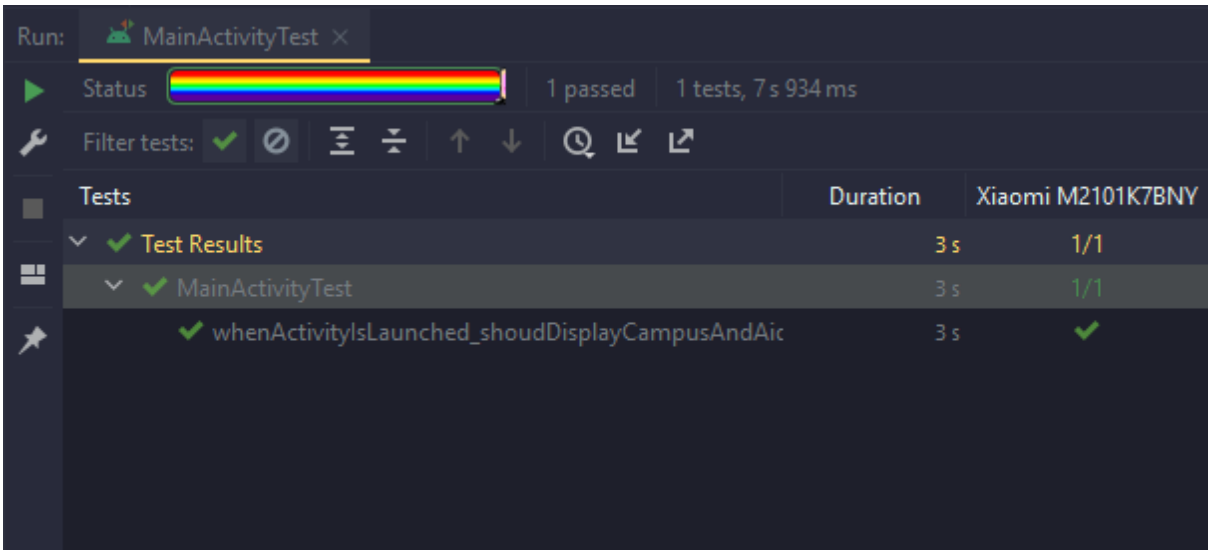

Figura 6. Resultado do teste no projeto novo

5. Conclusões e Trabalhos Futuros

Este trabalho teve como objetivo relatar como implementar testes unitários e de Interface do Usuário no contexto de um sistema Android, a fim de garantir um processo de desenvolvimento mais seguro.

Como mostrado durante o relato, foi possível descrever as configurações e os passos necessários para esse tipo de implementação, atingindo o objetivo principal e podendo servir de documentação para outras pessoas que queiram fazer isto em seus projetos. O uso do Android Studio como IDE e das suas ferramentas - como o Coverage - deixaram o processo de execução bastante simples, além do uso do template Given-When-Then que auxiliou na hora de criar testes unitários mais compreensíveis. Todo o código aqui produzido está disponível para consulta no repositório do projeto[8] no Github.

Em relação ao código, alguns trabalhos futuros são implementar mais testes unitários, a fim de obter uma maior cobertura do projeto. Além disso, também pretendemos iniciar um processo de adoção do método de desenvolvimento TDD, e avaliar se foi possível verificar melhora na qualidade do código e da sua produção.

Em relação ao Objeto de Aprendizagem AppAlpha, visa-se fazer uma nova avaliação desta vez com crianças para verificar sua eficácia no processo de alfabetização focado nessa faixa etária e considerando a última versão do aplicativo.

[8] https://github.com/a4s-ufpb/AppAlpha

Referências

Avcioglu, A., & Demirer, M. (2015). Implementation of system testing automatization on computer aided systems for hardware and software. In 2015 IEEE AUTOTESTCON. IEEE.

Campos, T. F. (2014). Ferramenta para Geração Automática de Testes Unitários a partir de Especificações Algébricas usando Alloy e SMT.

Dantas, E.; Borges, A. P. D. L.; Dantas, A. (2019). Avaliando a Utilização do Aplicativo AppAlpha no Processo de Alfabetização de Adultos. In Anais *do IV Congresso sobre Tecnologias na Educação* (pp. 232-240). SBC.

Finizola, J. S., Dantas, A., Santana, R., & Neves, T. (2016). Achei a Palavra: Um objeto de aprendizagem colaborativo para apoiar a alfabetização. RENOTE.

Freire, P. (2003). A importância do ato de ler.

Klammer, C., & Ramler, R. (2017). A journey from manual testing to automated test generation in an industry project. In 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C) (pp. 591-592). IEEE.

Hodgins, W. The future of learning objects. (2002). In: WILEY, D. A. (Ed.). The instructional use of learning objects. Bloomington: AECT, p.281-298.

Massol, V. and Husted, T. (2004). JUnit in action. Manning.

Minelli, R., Mocci, A., & Lanza, M. (2015). I know what you did last summer-an investigation of how developers spend their time. In 2015 IEEE 23rd International Conference on Program Comprehension). IEEE.

Nogueira, D., Coscarelli, C., Chiamowicz, L., & Prates, R. (2010). Papa Letras: Um jogo de auxílio à alfabetização infantil. Anais do IX Simpósio Brasileiro de Jogos e Entretenimento Digital (SBGames).

Silva, J.; Dantas, A. (2018). Avaliando o Uso do Aplicativo "Embaralhado" no Processo de Alfabetização. RENOTE. 16. 10.22456/1679-1916.86001.