I SEMINÁRIO DE DESENVOLVIMENTO MOBILE E CLOUD COMPUTING
INSTITUTO NACIONAL DE TELECOMUNICAÇÕES – INATEL
AGOSTO DE 2019

e-ISSN 2595-8186
ISSN-CD Rom 2447-2352

Testes automatizados para aplicativos Android

Isabel Francine Mendes & Paulo César Siécola

Abstract: Mobile app users are looking for the best possible experience when choosing an app for their smartphones. They do not tolerate failures, spread the errors found, uninstall that application and look for a product that can meet their needs. In this market, where there are more than 3 million application available, there is no chance for products with screen failures or that put their customer´s data at risk.

To ensure quality, user experience (UX) satisfaction, and rid businesses of failed troubles, automated testing should be an essential and undeniable part of mobile application development. With them, it is possible to test from the smallest part of the code to the interactions that the user will have with buttons, text boxes, and other screens components.

This article will examine the types of tests that can be automated, the tools used, and an actual application using UI tests.

Index terms: automated testing, Android Espresso, testing tools.

Resumo: Usuários de aplicativos para dispositivos móveis buscam a melhor experiência possível ao escolherem uma aplicação para seus *smartphones*. Não toleram falhas, espalham os erros encontrados, desinstalam a aplicação e procuram um produto que possa suprir suas necessidades. Nesse mercado, onde há mais de 3 milhões de aplicativos disponíveis não há chance para produtos com simples falhas de telas ou que ponham em riscos os dados de seus clientes.

Para garantir qualidade, satisfazer a experiência do usuário (UX) e livrar as empresas de transtornos com *bugs*, os testes automatizados devem ser parte fundamental e indiscutível do desenvolvimento de aplicativos para os dispositivos móveis. Com eles é possível testar desde a menor parte do código até as interações que o usuário final terá com botões, caixas de textos e outros componentes presentes nas telas.

Nesse artigo será visto os tipos de testes que podem ser automatizados, as ferramentas utilizadas e uma aplicação real utilizando testes de interface do usuário (UI).

Palavras chave: testes automatizados, Android Espresso, ferramentas para testes.

I. INTRODUÇÃO

No ano de 2018 foram realizados, aproximadamente, 205 bilhões de downloads de aplicativos para dispositivos móveis. Quase 3 milhões desses aplicativos utilizavam tecnologia Android e estavam disponíveis na Google Play Store [1] [2].

Para se destacar neste imenso mercado, o aplicativo deve assegurar a melhor experiência de usuário: funcionamento total dos recursos, menus e botões intuitivos, resposta rápida a erros encontrados, estética agradável e princípios éticos respeitados.

Muitas dessas questões podem ser resolvidas através dos testes de software utilizados para aplicativos de dispositivos móveis, que passam desde o teste unitário dos métodos das classes do programa até a interação final do usuário com a tela do aplicativo.

Nesse artigo serão mostrados quais são os tipos de testes e como devem fazer parte do ciclo de desenvolvimento de um aplicativo para dispositivos móveis. Também serão apresentadas algumas ferramentas disponíveis no mercado e um comparativo entre elas.

O maior destaque será dado aos testes de interface de usuário (UI), com a ferramenta *Espresso Test Recorder* [3] , que permite a gravação de testes simulando a experiência do usuário com a tela do aplicativo.

A figura 1 mostra o aplicativo para dispositivo móvel que foi utilizado para desenvolvimento dos testes.

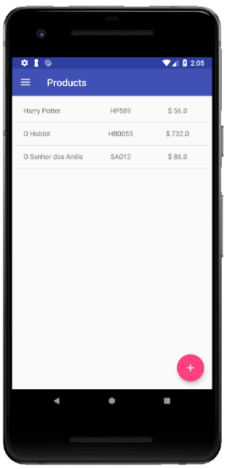

Figura 1: Aplicativo - Criação e lista de produtos

II. TIPOS DE TESTES PARA APLICATIVOS DE DISPOSITIVOS MÓVEIS

Os testes são parte fundamental do ciclo de desenvolvimento de softwares, garantindo produtos de qualidade, com alto desempenho e livres de erros elementares. A cultura de testar o aplicativo ainda não é amplamente usada pelos desenvolvedores, mas isso tem mudado com a disponibilidade de diversas ferramentas e investimento da própria Google na caça aos *bugs* de aplicativos de grandes empresas como PayPal, Spotify, Tesla entre outras [4].

e-ISSN 2595-8186
ISSN-CD Rom 2447-2352

Abaixo, estão algumas falhas em aplicativos que renderam grande transtorno aos usuários e prejuízos financeiros às empresas:

- Atualização do iOS 11.1: Em 2017, muitos usuários notaram que ao digitarem a letra "I" o aplicativo tocava por "A?". Inatel se transformaria em A?natel [5];
- Waze: Também em 2017, houve grande congestionamento (8,7 km de lentidão) na rua 23 de maio em São Paulo por conta de falha no aplicativo que indicava essa via como o melhor caminho para diversos usuários [6];
- Stagefright: Em 2016, uma falha de segurança expôs dados de 20% dos usuários que usavam versões de Android abaixo de 4.0 [7];
- *Bug* nos termostatos da Nest: O consumo rápido da bateria resultou em casas geladas no auge do inverno [8].

Os testes são categorizados, segundo Martin Fowler [9], numa pirâmide contendo três partes: testes unitários, testes de integração e testes de interface do usuário (UI).

A proporção de testes em cada camada varia de acordo com as funcionalidades do aplicativo envolvido. A recomendação no site do Google é:

- Testes unitários são mais rápidos e devem cobrir 70% dos casos de testes;
- Testes de integração devem cobrir 20%;
- E os de UI 10% do total de testes executados [3].

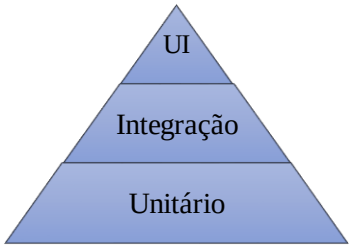

Figura 2. Pirâmide de testes

Abaixo, será mostrado um resumo dos tipos de testes, as ferramentas e *frameworks* desenvolvidas pelo Google para aplicativos para dispositivos Android e uma abordagem prática dos testes de UI.

A. *Testes unitários*

São aqueles responsáveis por testar a menor parte do código: um método dentro de uma classe.

Tudo o que fazem é verificar se o método está entregando o que foi planejado interferindo o mínimo possível em outros métodos e classes e sem depender de emuladores ou dispositivos para serem executados.

Para isso acontecer, utiliza-se o *framework* JUnit, escrito por Erich Gamma e Kent Beck, para a linguagem Java [10]. Além de executar esses testes de forma automatizada, esse conjunto de bibliotecas valida e exibe os resultados de maneira simples e compreensível ao desenvolvedor.

Por serem rápidos e de fácil execução, esses testes podem ser rodados todas as vezes que o desenvolvedor mudar qualquer parte do seu código, garantindo assim que as outras partes não foram afetadas. Caso necessário, os testes deverão ser reescritos para aumentar sua cobertura de validação no código principal.

No Android Studio, ambiente onde o código principal e os testes são escritos, os testes aprovados são sinalizados com cor verde pelo JUnit, caso contrário uma linha vermelha será mostrada nos testes que falharam [11] [12].

A seguir, são apresentados alguns testes unitários inseridos no código do aplicativo e o resultado apresentado no terminal do Android Studio.

```
public class CriarProdutoTest {

    @Test
    public void criaProdutoValido() {
        ProductCreationService productCreationService = ProductCreationService.getInstance();
        Product product = productCreationService.createProduct( name: "prod1",  description: "Desc1",  code: "COD123",  price: 10);

        assertEquals( expected: "prod1", product.getName());
    }

    @Test(expected = ProductWithoutNameException.class)
    public void criaProdutoSemNome() {
        ProductCreationService productCreationService = ProductCreationService.getInstance();
        productCreationService.createProduct( name: null,  description: "Desc1",  code: "COD123",  price: 10);
    }
```

```
Run:  CriarProdutoTest
Tests passed: 3 of 3 tests – 0 ms
CriarProdutoTest (br.com.siecola.androidespresso)  0 ms
    criaProdutoSemNome  0 ms
    criaProdutoPrecoInvalido  0 ms
    criaProdutoValido  0 ms
"C:\Program Files\Android\Android
Process finished with exit code 0
```

Figura 3: Resultado dos testes unitários no terminal do Android Studio

B. *Testes de Integração*

Os testes de integração são aqueles que simulam cenários onde há a necessidade de junção entre os métodos das classes, aplicações externas, como API (Interface de Programação de Aplicativos) e banco de dados, ou ainda outros componentes do Sistema Android. Esses testes apresentam as seguintes características [13]:

- São testes de média complexidade;
- Verificam se as unidades estão trabalhando de forma coordenada e sem erros;
- Cobrem um volume maior do sistema, por isso são mais eficientes que os testes unitários;
- A velocidade da execução dos testes cai à medida que a cobertura aumenta;
- Realizam testes com objetos reais ou objetos *mocks*;
- Dependendo do que será testado será necessário o uso de emuladores ou dispositivos físicos.

Muitas vezes as aplicações externas não estão disponíveis ou prontas para uso. Para sanar esse problema, o desenvolvedor lança mão de *mocks*, que são objetos que simulam o comportamento de objetos reais, como por exemplo, uma API dos Correios.

Um dos *frameworks* utilizados para testes unitários e de integração em aplicações Java é o *Mockito* [14], que apresenta vasta documentação, comunidade ativa e uso simples e intuitivo.

O *Mockito* apresenta diversos outros recursos como verificar se os métodos de um ou vários *mocks* foram

e-ISSN 2595-8186
ISSN-CD Rom 2447-2352

chamados numa ordem específica ou ainda encontrar invocações redundantes. O uso desses recursos poderá ser utilizado dependendo das funcionalidades de cada aplicativo.

O aplicativo apresentado nesse artigo utiliza o *Google Firestore* [15] como banco de dados não relacional da aplicação Ele armazena os produtos criados e permite a edição e exclusão destes produtos.

A complexidade para desenvolver testes de integração simulando ou usando o *Firestore* é alta e está fora do escopo desse artigo.

C. Testes Interface de Usuário (UI)

Os testes de interface de usuário (UI) são aqueles que devem simular as reais ações dos usuários, desde preencher um campo com o nome, clicar num botão ou selecionar algo na tela.

Esse tipo de teste é o mais demorado para se desenvolver, cobrindo apenas 10% do total de testes. Porém, é o com mais valor agregado para o usuário final, já que é a etapa final que garante que o aplicativo atende aos requisitos funcionais, se comporta como o esperado e respeita os padrões de usabilidade.

Duas ferramentas do Android possibilitam a automação dos testes de UI [16]:

- Android Espresso: Permite a automação de cenários complexos, cobrindo o conjunto de ações de único aplicativo. Nesse artigo será dado foco a essa ferramenta;
- UIAutomator: Verifica o comportamento correto entre diferentes aplicativos: por exemplo se um aplicativo de medição de batimentos cardíacos compartilha os dados corretamente com outro de saúde.

D. Teste de UI usando Android Espresso

A estrutura de testes do Espresso contém uma API fluente e concisa que permite simular as interações dos usuários. Essa API é aberta, permitindo personalização dos testes.

Essa estrutura faz parte da biblioteca Android Testing Support, assim como o AndroidJUnitRunner, que é também necessário para executar os testes de UI nos dispositivos com sistema Android.

Dentro do *Espresso* há a ferramenta *Espresso Test Recorder* que permite a criação de testes sem a necessidade de escrita do código. Para isso, basta gravar um cenário de teste com as interações do usuário, usando um dispositivo móvel ou emulador e adicionar declarações para verificação dos elementos da UI. Depois de gravado, o Espresso salva e gera os testes correspondentes, que podem ser personalizados, como por exemplo, diminuir o tempo de espera entre uma tela e outra. O teste criado poderá ser usado quantas vezes necessário.

III. TESTES AUTOMATIZADOS USANDO ESPRESSO TEST RECORDER

A. Configurações Iniciais

Um simples aplicativo para dispositivo móvel foi previamente desenvolvido com duas telas: Na primeira tela é possível ver a lista dos produtos criados, editá-los e deletá-los. E na segunda tela, o usuário pode registrar produtos com nome, código, descrição e preço.

Para que os testes fossem executados no Android Studio, foi necessária a configuração dos seguintes parâmetros no arquivo *build.gradle*:

- Versão do Android:

```
{
   compileSdkVersion27
   defaultConfig {
        minSdkVersion 26
        targetSdkVersion 27
        versionCode 1
        versionName "1.0"
        vectorDrawables.useSupportLibrary =true
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
}
```

- Parâmetro responsável por gerar os relatórios dos testes:

```
debug {
    testCoverageEnabled = true
}
```

- Parâmetro responsável por mostrar os resultados dos testes unitários:

```
testOptions.unitTests.all {
   testLogging {
   events 'passed', 'skipped','failed', 'standardOut', 'standardError'
   }
}
```

- Dependências:

```
{
    implementation 'com.android.support.test.espresso:espresso-idling-resource:3.0.2'
    testImplementation 'junit:junit:4.12'

    //Dependency to unit tests
    testImplementation 'org.mockito:mockito-all:1.10.19'
    testImplementation 'org.hamcrest:hamcrest-all:1.3'
    testImplementation 'org.powermock:powermock-module-junit4:1.6.2'
    testImplementation 'org.powermock:powermock-api-mockito:1.6.2'

    //Dependency from Android Testing Support Library's runner and rules
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test:rules:1.0.2'

    //Dependency from Espresso UI Testing.
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-contrib:3.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-intents:3.0.2'
}
```

B. Passo a passo para realizar os testes de UI com Espresso Test Recorder

Os códigos de testes criados pelo *Espresso Test Recorder* são construídos quando há interação do usuário com a UI do aplicativo para dispositivos móveis. Além de garantirem alta cobertura dos testes, são editáveis e de fácil compreensão mesmo para desenvolvedores iniciantes.

As etapas a seguir são necessárias para a criação de casos de testes em que um novo produto será criado, editado e excluído do aplicativo móvel [17]:

- No Android Studio vá em *Run* e depois *Record Espresso Test*;

e-ISSN 2595-8186
ISSN-CD Rom 2447-2352

- Selecione o emulador ou device e clique em ok;
- A caixa de diálogo *Record Your Test*, o painel *Debugger* e a tela do emulador serão exibidos, como pode ser visto na figura 4.

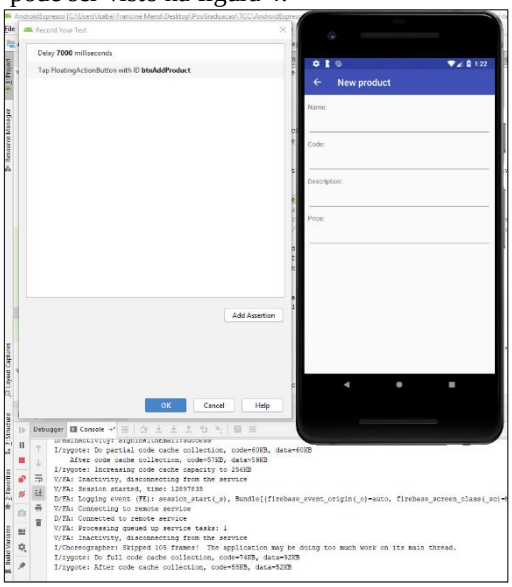

Figura 4: Primeira tela do Espresso Test Recorder

- No emulador ou dispositivo, toque no botão de adição (+) na tela de criação de novo produto do aplicativo. Adicione um produto com nome, código, descrição e preço e salve (ou clique no ícone voltar).
- A janela *Record Your Test* mostra as ações sendo gravadas (*btnAddProduct*, *edtName*, *Navigate up).*
- Na figura 5, é possível observar as ações gravadas e o novo produto exibido na lista.

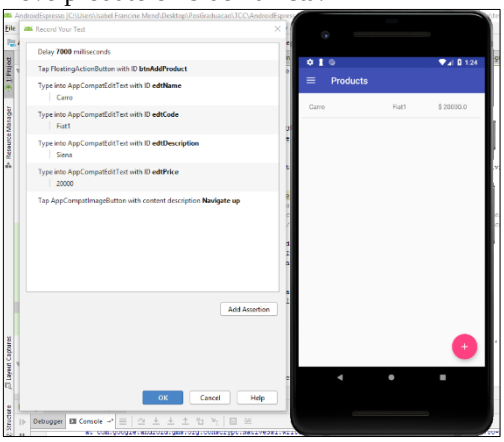

Figura 5: Ações gravadas pelo Record Espresso Test

- Clique em *Add Assertion* na janela *Record Your Test*. Uma captura de tela da interface do usuário do aplicativo é exibida em um painel no lado direito da janela.
- Selecione cada item da lista da interface do usuário que se deseja verificar (na tela à direita) e escolha o texto esperado na tela *Edit Assertion.*
- Salve todas as *Assertions* necessárias, finalizando o teste com *Save Assertion*, como mostrado na figura 6.

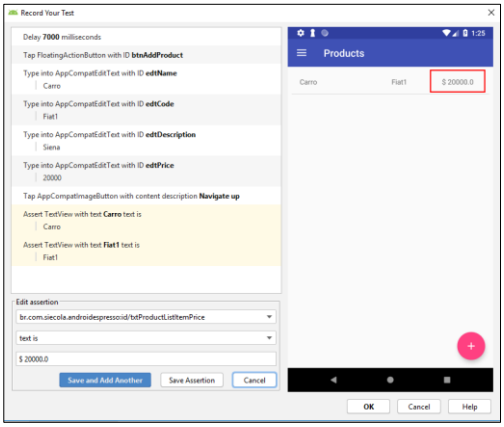

Figura 6: Assertion

- Volte ao aplicativo, clique e pressione o botão do mouse no produto criado. Os ícones de editar e excluir serão exibidos, como mostrado na figura 7. Clique no excluir.
- Clique no botão Ok e salve o teste.
- O teste será criado em *br.com.siecola.androidespresso(androidTest)*, local onde são criados os testes instrumentados do Android Studio.

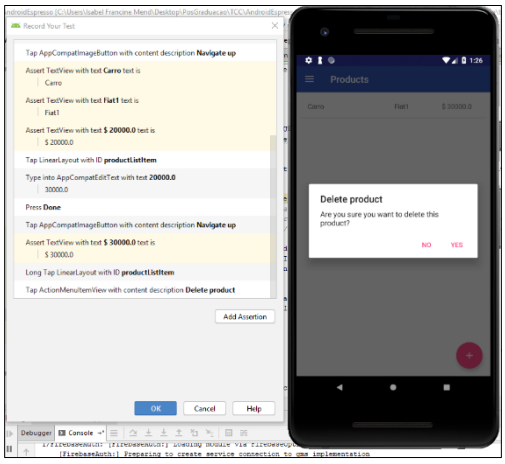

Figura 7: Ação de excluir o produto

C. Análise do Código de Teste

A seguir, o código do teste criado foi separado em algumas partes para melhor entendimento:

```
public class testeTCC3 {
    @Rule
    public ActivityTestRule<MainActivity> mActivityTestRule = new ActivityTestRule<>(MainActivity.class);

    @Test
    public void testeTCC3() {
        try {
            Thread.sleep( millis: 2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

A primeira parte do código apresenta as seguintes notações:

- @Rule: notação responsável por instanciar a Activity e validá-la.
- @Test: responsável por dizer ao JUnit que há casos de testes que serão executados.

e-ISSN 2595-8186
ISSN-CD Rom 2447-2352

- Por padrão, o Record configura um tempo (Thread.sleep) de sete segundos para dar início a aplicação. Nesse teste foi necessário diminuir o tempo para 2 segundo e adicioná-lo também logo depois da ação do botão "*Navigate up*". Dessa forma, há um tempo suficiente para que todos os campos fossem preenchidos e verificados corretamente.

```
ViewInteraction floatingActionButton = onView(
        allOf(withId(R.id.btnAddProduct),
                childAtPosition(
                        allOf(withId(R.id.layoutProductsList),
                                childAtPosition(
                                        withId(R.id.container),
                                        position: 0)),
                        position: 1),
                isDisplayed()));
floatingActionButton.perform(click());
```

A segunda parte apresenta:

- *ViewInteraction*: classe responsável por executar as ações (usando o método *perform()*) e asserções (usando o método *check()*) nos elementos da *View* que nesse código representam os botões de adicionar, editar e excluir e ainda as caixa de texto;
- O método *childAtPosition()* : responsável por encontrar um componente gráfico na tela através de um *Id*;
- *perform(click)*: método para clicar no botão de adição (+).

```
ViewInteraction textView = onView(
        allOf(withId(R.id.txtProductListItemName), withText("Carro"),
                childAtPosition(
                        allOf(withId(R.id.productListItem),
                                childAtPosition(
                                        withId(R.id.rcvProducts),
                                        position: 0)),
                        position: 0),
                isDisplayed()));
textView.check(matches(withText("Carro")));
```

- Na terceira parte é possível ver a instrução *textView.check(matches(withText("Carro")))* que é responsável por confirmar o estado atual de uma View. Nesse exemplo, verifica-se se o nome do produto que foi criado como Carro é realmente esse.

```
ViewInteraction recyclerView = onView(
        allOf(withId(R.id.rcvProducts),
                childAtPosition(
                        withId(R.id.layoutProductsList),
                        position: 0)));
recyclerView.perform(actionOnItemAtPosition( position: 0, click()));
```

- E também a instrução *recyclerView.perform(ActionOnItemPosition(click))*: é realizada ao se selecionar o produto da posição zero da lista de produtos.

```
ViewInteraction actionMenuItemView = onView(
        allOf(withId(R.id.product_delete), withContentDescription( text: "Delete product"),
                childAtPosition(
                        childAtPosition(
                                withId(R.id.action_mode_bar),
                                position: 2),
                        position: 1),
                isDisplayed()));
actionMenuItemView.perform(click());

ViewInteraction appCompatButton = onView(
        allOf(withId(android.R.id.button1), withText("Yes"),
                childAtPosition(
                        childAtPosition(
                                withClassName(is( value: "android.widget.ScrollView")),
                                position: 0),
                        position: 3)));
appCompatButton.perform(scrollTo(), click());
```

- Na quarta parte do código, há a ação de excluir um produto, com a instrução *actionMenuItemView.perform(click)*.

```
private static Matcher<View> childAtPosition(
        final Matcher<View> parentMatcher, final int position) {

    return new TypeSafeMatcher<View>() {
        @Override
        public void describeTo(Description description) {
            description.appendText("Child at position " + position + " in parent ");
            parentMatcher.describeTo(description);
        }

        @Override
        public boolean matchesSafely(View view) {
            ViewParent parent = view.getParent();
            return parent instanceof ViewGroup && parentMatcher.matches(parent)
                    && view.equals(((ViewGroup) parent).getChildAt(position));
        }
    };
}
```

- E por último é mostrada a instrução da classe *Matcher* que retorna um *ViewAssertion* afirmando que a *View* existe.

D. Análise dos Relatórios Gerados pelo Android Espresso

Depois que os testes foram realizados é possível obter os resultados das falhas e sucessos através dos relatórios que o Android Studio fornece. Para isso, vá até a aba do *Gradle*, abra a seção *app*, *Tasks* e *verification*. Dê duplo clique em *createDebugCoverageReport*. Com o emulador aberto, todos os testes do Android Espresso serão realizados.

Ao finalizar a verificação, os relatórios de cobertura de testes serão criados na pasta onde está o projeto Android.

O primeiro relatório pode ser encontrado em C:\Desktop\SeuProjeto\app\build\reports\androidTests\connected\index. Na figura 8 e 9, é possível verificar os resultados dos testes, tempo de duração, o tipo de emulador usado e os testes divididos por pacote ou classe.

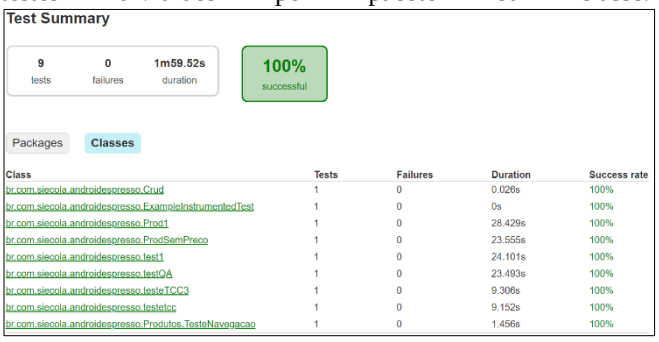

Figura 8: Relatório de testes por classe

Figura 9: Detalhes do relatório de um teste específico

e-ISSN 2595-8186
ISSN-CD Rom 2447-2352

Também é possível ver a cobertura de testes no projeto como um todo através do caminho C:\Desktop\SeuProjeto\app\build\reports\coverage\debug\index:

Na figura 10, é possível verificar a cobertura de testes por pacote em todo o projeto.

debugAndroidTest

Element	Missed Instructions	Cov.
br.com.siecola.androidespresso		54%
br.com.siecola.androidespresso.service		0%
br.com.siecola.androidespresso.fragment		90%
br.com.siecola.androidespresso.exception		0%
br.com.siecola.androidespresso.viewmodel		85%
br.com.siecola.androidespresso.util		0%
br.com.siecola.androidespresso.repository		99%
br.com.siecola.androidespresso.adapter		100%
br.com.siecola.androidespresso.model		100%
Total	464 of 1,764	73%

Figura 10: Cobertura de testes por pacote

Ao se entrar num pacote, é possível ver a cobertura em todas as classes, como mostrado na figura11.

br.com.siecola.androidespresso.fragment

Element	Missed Instructions	Cov.
ProductFragment		83%
ProductsListFragment		95%
ProductsListFragment.new ActionMode.Callback() {...}		98%
Total	49 of 537	90%

Figura 11: Cobertura de testes por classe

Dentro da classe, observa-se há cobertura de teste em cada método, conforme a figura 12.

ProductFragment

Element	Missed Instructions	Cov.
hideKeyboard()		0%
onPause()		91%
lambda$onCreateView$0(View)		0%
onActivityCreated(Bundle)		100%
lambda$onResume$1(Product)		100%
onCreateView(LayoutInflater, ViewGroup, Bundle)		100%
onResume()		100%
ProductFragment()		100%
Total	38 of 230	83%

Figura 12: Cobertura de testes por método

A seguir, há uma demonstração de um fragmento do código que não foi coberto por testes.

Esse trecho é responsável por deixar o teclado invisível quando o usuário clica fora dos elementos de texto ou botões.

```
private void hideKeyboard() {
    if (getActivity() != null) {
        InputMethodManager imm = (InputMethodManager)getActivity()
                .getSystemService(Context.INPUT_METHOD_SERVICE);
        if ((imm != null) && (getActivity().getCurrentFocus() != null) &&
                (getActivity().getCurrentFocus().getWindowToken() != null)) {
            imm.hideSoftInputFromWindow(getActivity().getCurrentFocus().getWindowToken(), 0);
        }
    }
}
```

Com todos esses relatórios é possível ao desenvolvedor explorar onde seu código, tanto o principal quanto o de teste, pode ser melhorado para aumentar a cobertura de testes.

IV. COMPARATIVO COM OUTRAS FERRAMENTAS

Existe uma grande variedade de ferramentas para se testar aplicativos de dispositivos móveis: algumas são *open source*, outras atendem aplicativos híbridos (aqueles que usam o mesmo código para diferentes sistemas operacionais) e outras suportam diferentes linguagem de programação.

A seguir, é apresentada uma nova pirâmide de testes com mais subníveis, com as ferramentas apresentadas ao longo desse artigo e outras populares no ecossistema Android [18].

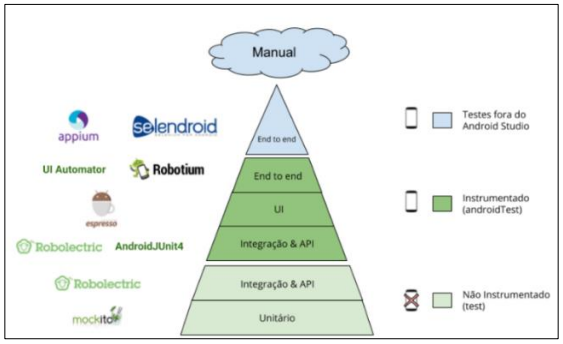

Figura 13: Pirâmide de níveis de testes e respectivas ferramentas

Também é apresentado um comparativo entre essas ferramentas, segundo [19] [20]:

- Linguagens de programação (LP): quais são suportadas pela ferramenta.
- Aplicações (A): se aplicações híbridas, nativas e web são suportadas.
- Suporte (Sup): Se há comunidade, canais de ajuda e suporte para o desenvolvedor.
- *Cross-platform* (CP): os mesmos códigos de testes servem para aplicações Android, iOS.
- *White-box testing* (WB): Indica se o a ferramenta precisa de acesso ao código da aplicação.

TABELA I: FERRAMENTAS PARA AUTOMAÇÃO DE TESTES

Ferramenta	LP	A	Sup	CP	WB
JUnit	Java e Kotlin	Nativa	JUnit	N	S
Mockito	Java	Nativa	Mockito	N	S
Robolectric	Java	Nativa	Robolectric	N	S
Appium	Java, Objective-C, Ruby, Python, entre outras	Híbridas, nativas e web	appium	S	N
Calabash	Ruby	Híbridas, nativas e web	Calabash	S	N
Espresso	Java e Kotlin	Nativa	Espresso	N	N
Selendroid	Java	Híbridas, nativa e web	Selendroid	S	N
UIAutomator	Java	Nativa	ui-automator	N	N
Robotium	Java	Híbridas e nativas	Robotium	N	N

e-ISSN 2595-8186
ISSN-CD Rom 2447-2352

Como pode ser observado, cada ferramenta tem suas características e sua escolha dependerá do que se deseja atender em cada projeto.

V. GERENCIAMENTO DOS CASOS DE TESTES

A cobertura dos casos de testes deve ser conhecida por todos os envolvidos no desenvolvimento do produto, seja os desenvolvedores, analistas de qualidade, gerentes e dono do produto. Dessa forma há coerência entre o que foi planejado e o que será entregue.

Os desenvolvedores e analistas de qualidade devem conhecer a fundo todos os tipos e casos de testes que cobrem o produto, pois, caso haja alteração do código, saberão os testes que necessitam de mudanças ou exclusão.

Há no mercado várias ferramentas que permitem a escrita e gerenciamento dos casos de testes. Duas delas são:

- qTest [21]: é possível escrever, organizar e executar os casos de testes usando uma interface rápida e fácil e ainda disponibilizar relatórios dos testes executados. Além disso é integrada ao *Jira* (ferramenta de gerenciamento de projetos ágeis) e ao *Confluence* (ferramenta de gerenciamento de documentação);
- TestLink [22]: também possibilita a escrita, armazenamento e execução dos casos de teste, sendo uma ferramenta *open source*.

Alguns desenvolvedores iniciam a escrita dos casos de testes numa tabela do Excel. Isso é válido até certo ponto, pois, aumentando-se a cobertura de testes cresce também a necessidade de organização e gerenciamento dos testes unitários, de integração de UI, desempenho e outros que o produto possa exigir.

VI. TRABALHOS FUTUROS

O desenvolvimento de testes automatizados está sempre em evolução e necessita de estudos constante para a melhor experiência do usuário com o aplicativo apresentado.

Por tanto, os seguintes assuntos serão tratados em trabalhos futuros:

A. Testes de integração

Os testes de integração serão desenvolvidos em duas etapas:

- Simulando o *Firestore* através de *mocks*;
- E usando o *Firestore* como aplicação externa.

B. Teste de desempenho usando Firebase Test Lab [23]

Todo aplicativo necessita de testes que simulem o uso real de suas funções e telas. Para isso o uso do Firebase Test Lab é mais que bem-vindo no auxílio dessa tarefa, já que possui uma grande variedade de dispositivos e configurações disponibilizados no *data center* do Google. Além de ser todo baseado em nuvem, apresenta relatório de resultados da bateria de testes e o desempenho do hardware do dispositivo durante todo o processo.

Os testes que podem ser executados são:

- Teste de instrumentação: o Test Lab executa testes que foram escritos usando as bibliotecas do Espresso e do UI Automator 2.0. Esses são os testes que serão, futuramente, utilizados na aplicação apresentada nesse artigo;
- Teste Robô: para aplicativos Android, é possível que o próprio Test Lab crie os testes automatizados e em seguida execute-os;
- Teste de loop de jogo: teste que demostra a execução de um cenário do jogo, simulando as ações do jogador;
- Teste de instrumentação com Orquestrador: executa os testes de instrumentação de maneira independente. Os testes são mais demorados, porém, com a vantagem de encerrar apenas os testes que falharam durante a execução.

C. Testes para a iOS

Durante o curso de pós-graduação também houve o desenvolvimento de um aplicativo para dispositivos móveis que utilizam o sistema iOS. Logo, os testes para essa plataforma também serão realizados de acordo com as técnicas e ferramentas fornecidas pela Apple.

VII. CONCLUSÃO

Depois de apresentado os níveis de testes, as ferramentas existentes e como testar uma aplicação, previamente desenvolvida durante o curso de pós-graduação, tem-se evidências que testes automatizados devem fazer parte do desenvolvimento de todo aplicativo para dispositivos Android.

Na parte prática, todos os testes foram desenvolvidos usando as ferramentas nativas do Android. Essa escolha se deu devido a facilidade de uso, tutoriais e guias fornecidos, e pela interação do Android Espresso com o Firebase.

Cada teste deve ser cuidadosamente escrito e elaborado de forma que o usuário final tenha uma satisfatória experiência e não queira desinstalar o aplicativo. Erros são passíveis de acontecer em qualquer tipo de aplicativo para dispositivo móvel, seja ele de grandes empresas ou do desenvolvedor *indie*. Porém com a filosofia do uso de testes automatizados as falhas e bugs tendem ao ínfimo.

VIII. REFERÊNCIAS

[1] "Statista," 2019. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. [Acesso em Janeiro 2019].

[2] "Statista," 2019. [Online]. Available: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/. [Acesso em Janeiro 2019].

e-ISSN 2595-8186
ISSN-CD Rom 2447-2352

[3] "Google Developers," [Online]. Available: https://developer.android.com/training/testing/fundamentals. [Acesso em Fevereiro 2019].

[4] "hackerone," Outubro 2017. [Online]. Available: https://hackerone.com/googleplay. [Acesso em Fevereiro 2019].

[5] S. Larson, "CNN Business," Novembro 2017. [Online]. Available: https://money.cnn.com/2017/11/06/technology/apple-bug-iphone-letter-i-symbols/index.html. [Acesso em Fevereiro 2019].

[6] B. Capelas, B. Ribeiro e P. Mengue, "Exame," Outubro 2017. [Online]. Available: https://exame.abril.com.br/tecnologia/falha-no-waze-por-ter-ajudado-a-triplicar-transito-em-sao-paulo/. [Acesso em Janeiro 2019].

[7] S. Griffiths, "Daily mail," Março 2016. [Online]. Available: https://www.dailymail.co.uk/sciencetech/article-3502470/Stagefright-malware-Worst-Android-bug-history-returns-time-infect-BILLIONS-Android-phones-seconds.html. [Acesso em Janeiro 2019].

[8] N. Bilton, "New York Time," Janeiro 2016. [Online]. Available: https://www.nytimes.com/2016/01/14/fashion/nest-thermostat-glitch-battery-dies-software-freeze.html?_r=0. [Acesso em Março 2019].

[9] H. Vocke, "Martin Fowler," 26 Fevereiro 2018. [Online]. Available: https://martinfowler.com/articles/practical-test-pyramid.html. [Acesso em Março 2019].

[10] P. Blundell e D. T. Milano, Learning Android Application Testing, Packt Publising Ltd., 2015.

[11] M. Aniche, Testes Automatizados de Software - Um guia prático, Casa do Código, 2017.

[12] "Caelum Ensino e Inovação," [Online]. Available: http://www.caelum.com.br/apostila-java-testes-xml-design-patterns/testes-automatizados/#3-5-junit. [Acesso em Fevereiro 2019].

[13] A. Turcu, "ProAndroidDev," Dezembro 2018. [Online]. Available: https://proandroiddev.com/writing-integration-tests-in-android-b0436978ed7b. [Acesso em Abril 2019].

[14] "Mockito," [Online]. Available: https://static.javadoc.io/org.mockito/mockito-core/2.27.0/org/mockito/Mockito.html#0.1. [Acesso em Abril 2019].

[15] "Cloud Firestore," Google, [Online]. Available: https://firebase.google.com/products/firestore/. [Acesso em abril 2019].

[16] "Android Developers," [Online]. Available: https://developer.android.com/training/testing/ui-testing. [Acesso em Abril 2019].

[17] "Codelabs," Google, [Online]. Available: https://codelabs.developers.google.com/codelabs/android-training-espresso-for-ui-testing/index.html?index=..%2F..android-training#5. [Acesso em maio 2019].

[18] P. Silva, "Medium," Maio 2018. [Online]. Available: https://medium.com/android-dev-br/explorando-a-pir%C3%A2mide-de-testes-no-android-parte-1-18ea135808df. [Acesso em Abril 2019].

[19] R. d. R. B. X. d. Morais, *Estudo exploratório sobre ferramentas para aplicações Android*, Recife, 2018.

[20] "AFour Technologies," 30 Setembro 2015. [Online]. Available: https://afourtech.com/automation-tools-for-ios-and-android-apps/. [Acesso em Abril 2019].

[21] "qTest," [Online]. Available: https://www.qasymphony.com/software-testing-tools/qtest-manager/test-case-management/. [Acesso em Abril 2019].

[22] L. Reis, "Medium," Agosto 2018. [Online]. Available: https://medium.com/@luanasreis/testlink-uma-ferramenta-de-gerenciamento-de-testes-de-software-44001b816f64. [Acesso em Abril 2019].

[23] "Firebase," Março 2019. [Online]. Available: https://firebase.google.com/docs/test-lab/android/overview?hl=pt-br. [Acesso em Abril 2019].

[24] A. Gazola e C. Lopes, "Criação de mocks com Mockito," *Mundo J*, nº 49, p. 5.

[25] J. Viegas, "OneDayTesting," Julho 2016. [Online]. Available: http://blog.onedaytesting.com.br/bugs-em-aplicativos-mobile/. [Acesso em Maio 2019].

[26] "App Annie," Novembro 2014. [Online]. Available: https://www.appannie.com/en/insights/market-data/lessons-learned-from-100-negative-app-reviews/. [Acesso em Maio 2019].

[27] B. Abreu, "OneDayTesting," Janeiro 2018. [Online]. Available: http://blog.onedaytesting.com.br/2017-ano-em-bugs/. [Acesso em Maio 2019].

e-ISSN 2595-8186
ISSN-CD Rom 2447-2352

Isabel Francine Mendes é engenheira de Quality Assurance na empresa WatchGuard em Santa Rita do Sapucaí – MG.

É graduada em Engenharia Biomédica pelo Instituto Nacional de Telecomunicações - Inatel e pós-graduada em Gestão de Projetos pela Faculdade de Administração e Informática – FAI.

Paulo César Siécola é Mestre em ciência da computação pela USP na área de sistemas distribuídos. Professor de curso de pós-graduação no Inatel em disciplinas envolvendo AWS, Azure, Web API, Google App Engine, Android e iOS. Desenvolvedor back-end com Java e Spring Boot. Desenvolvedor Android e iOS. Escritor de livros sobre cloud computing. Líder técnico na WatchGuard, desenvolvendo soluções de segurança em cloud computing.