

Análise comparativa de ferramentas para testes em aplicativos Android

Samuel Vieira Pinto, Marco Antônio Pereira Araújo

Centro de Ensino Superior de Juiz de Fora – Juiz de Fora – MG – Brasil

samuelsjn@yahoo.com.br, marco.araujo@pucminas.cesjf.br

***Abstract.** Mobile computing appears as an innovative technology in systems development. Mobile applications are being used in many areas and a failure can cause great losses. Therefore, it is vital to ensure the quality of its processes. The activity test is the most widely used resource in software engineering to ensure the quality of a product. The use of tools that assist in the execution of these activities becomes increasingly necessary, but a poorly designed test can significantly increase project costs. In this context, the article provides a case study for automated testing with open source frameworks in mobile apps on Android operating system, unit level and system interface. At the end of the study will be presented a critical analysis of each tool.*

***Resumo.** A computação móvel aparece como uma tecnologia inovadora na área de desenvolvimento de sistemas. Aplicativos de celulares estão sendo utilizados em diversas áreas de atuação e uma falha pode ocasionar grandes perdas. Portanto, é vital garantir a qualidade de seus processos. A atividade de testes é o recurso mais utilizado na Engenharia de Software para se garantir a qualidade de um produto. O uso de ferramentas que auxiliam na execução dessas atividades se torna cada vez mais necessário, mas um teste mal elaborado pode aumentar consideravelmente os custos do projeto. Neste contexto, o artigo proporciona um estudo de caso para testes automatizados com frameworks de código aberto em aplicativos móveis em sistema operacional Android, no nível de unidade e interface de sistema. Ao final do estudo será apresentada uma análise crítica de cada ferramenta.*

Palavras chaves: teste de software, Android, Android Studio.

1. Introdução

Com o crescente aumento de vendas de aparelhos celulares, o desenvolvimento de aplicativos se tornou um setor muito atraente. Muitas empresas de sistemas tem investido em capacitação de seus funcionários tornando-os capazes em desenvolver aplicativos para celulares, auxiliando assim, várias ações no cotidiano de pessoas e empresas. Com isso, um enorme interesse dessas empresas em aperfeiçoar métodos e buscar uma redução nos custos dos projetos. Com essa abordagem, surge à necessidade de criar estratégias que auxiliam na qualidade do produto apresentado.

Segundo Pressman (2011), a Engenharia de Software é uma ciência que utiliza ferramentas, métodos e processos que garantem qualidade em um sistema informatizado. Um desses métodos é a exaustiva utilização de atividades de testes de software em processos do desenvolvimento de sistemas. Testes procuram identificar e eliminar defeitos e falhas que persistem em diferentes partes do produto de software.

Mesmo assim, a execução de testes em modo manual pode dificultar a sua repetição e a garantia de que foram executados corretamente.

Nesse sentido, a intenção desse artigo é coletar informações de ferramentas de testes na utilização de aplicativos móveis para Android, como Robotium, Espresso e JUnit, para auxiliarem nas rotinas de testes desses aplicativos. Será abordada a instalação e configuração no Android Studio exemplificando suas principais funções em um estudo de caso.

O artigo está organizado em 5 seções. Na Seção 2 são apresentados os conceitos sobre teste de software utilizado para o desenvolvimento do estudo de caso. Na Seção 3 são apresentadas a estrutura do aplicativo na plataforma Android, as ferramentas de testes com suas informações e os métodos de testes obtidos pelas ferramentas. Na Seção 4 é apresentado o resultado obtido pelo artigo e na Seção 5 as conclusões seguidas das referências do trabalho.

2. Referencial Teórico: Teste de Software

O controle de qualidade de uma empresa que possua sua mercadoria utilizada por terceiros é uma das práticas principais a serem adotadas para um produto de excelência. Esse controle é tão significativo para o desenvolvimento de um produto que atualmente possui setores específicos que buscam essa qualidade. No desenvolvimento de software não é diferente, inclusive encontrando muitos desenvolvedores que pensam de forma equivocada. A garantia de qualidade de um software engloba uma extensa lista de atividades que tem como resultado um produto com o mínimo de erros. Nesta seção do artigo será abordada uma dessas atividades, o teste de software.

Teste de Software, segundo Pressman (2011), é um conjunto de tarefas planejadas que possuem a intenção de investigar todas as partes de um sistema, aplicando rotinas pré-definidas com o intuito de encontrar erros, anomalias e imperfeições no projeto. Essas tarefas necessitam de uma análise fácil para que toda a equipe de teste possa entender e implementar sem sobrecarregar o orçamento do projeto.

Pressman (2011) divide os casos de testes em duas maneiras, a primeira é quando o testador conhece a função para qual o produto foi projetado e a segunda, quando se conhece o funcionamento interno do produto. A primeira maneira denominada como teste de caixa preta, são testes realizados na interface do software, não se preocupando em realizar provas na estrutura lógica do sistema, enquanto a segunda maneira é conhecida como testes de caixa branca, que são rotinas criadas com o intuito de realizar exaustivos exames na estrutura lógica do sistema. Dessa forma, Pressman (2011), defini níveis de testes com o intuito de executar testes durante o processo de desenvolvimento de um software, que são: unidade, integração, sistema, integração de sistemas e de aceitação.

Nesse artigo serão abordados dois níveis de testes ensinados por Pressman, o Teste de Unidades e Testes de Instrumentação, com o intuito de implementar os casos de testes no código e na interface do aplicativo. Testes de Unidades são executados na máquina onde a *JVM* (Java Virtual Machine) esteja instalada, minimizando o tempo de execução do código que não depende de nenhuma relação com a plataforma Android. E os Testes de Instrumentação, que necessitam de um hardware ou emulador que utilizam e permitem acesso à API do Android (ANDROID, 2016).

3. Estrutura da aplicação Android

A ferramenta Android Studio é a IDE (*Integrated Development Environment* – Ambiente de Desenvolvimento Integrado) oficial para desenvolvimento Android (ANDROID STUDIO, 2016). É uma ferramenta de acesso gratuito e foi desenvolvida pela empresa Google para facilitar e agilizar o desenvolvimento de aplicativos móveis na plataforma Android. Essa ferramenta oferece muitos recursos, como criação, alteração e depuração de código, compilação e ferramentas de desempenho, ajudando assim, todo profissional interessado no desenvolvimento de dispositivos móveis. A versão utilizada neste estudo é a 2.1.3 e foi disponibilizada no site oficial da ferramenta.

Ao iniciar um novo projeto com a IDE Android Studio, alguns arquivos de configuração são criados automaticamente. Para a utilização das ferramentas de testes será necessário alterar e acrescentar algumas informações em arquivos de configuração, sendo o *build.gradle* o principal arquivo. Nesse são configuradas algumas funções importantes, como compiladores para testes, ferramentas para depuração de código, *plug-ins*, anotações, dependências entre outras. Serão detalhados posteriormente os códigos necessários para configurar cada ferramenta no projeto. Os projetos criados na IDE Android Studio contém módulos para a apresentação dos arquivos de código fonte. Assim, para encontrar os arquivos de configuração basta selecionar o módulo de projeto Android e localizar a pasta chamada *Gradle Scripts*, local que se encontram todos os arquivos *Gradle*.

Foi desenvolvido para o artigo um aplicativo na plataforma Android com o intuito de apresentar as ferramentas de testes. Esse aplicativo, apresentado na Figura 1, apresenta um fragmento de um sistema para controlar a entrada e saída de produtos em um sistema de controle de estoque. Esse aplicativo contém três telas de interface e uma classe de conexão com a base de dados.

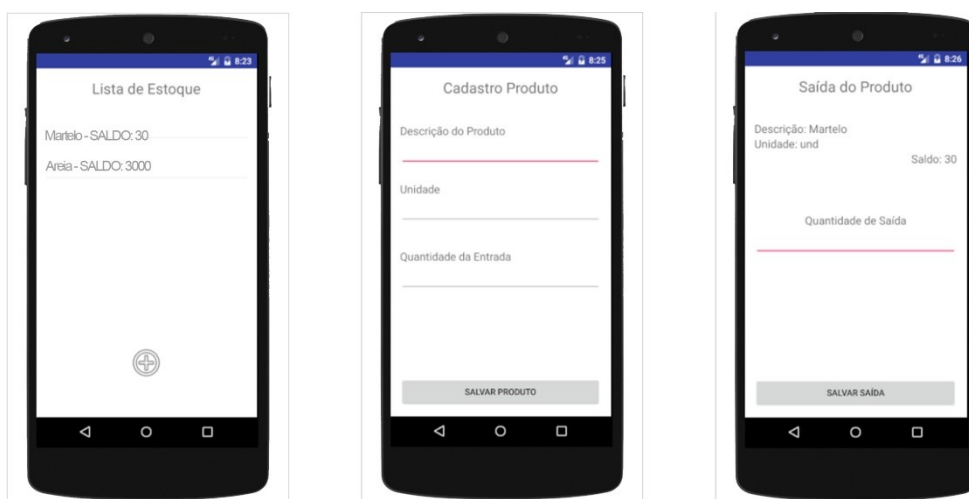


Figura 1 - Telas do Aplicativo Estoque

As telas de interface, na programação Android, são arquivos em *XML* que possuem todas as informações dos componentes ali alocados. Nesse *XML*, as *tags* dos elementos representam os componentes, como *EditViews*, *TextViews*, *ListView* entre outros, e as *tags* dos atributos representam as propriedades dos componentes, como exemplo, *layout_width*, *layout_height*, *text*, *id*, *textSize* entre outros. Assim, o arquivo

tem como principal função a definição dos parâmetros de *layout* para a visualização dos componentes em tela.

Em conjunto com os arquivos XML, a ferramenta Android Studio cria em paralelo os arquivos em Java para implementarem a parte lógica da interface como, por exemplo, os eventos dos componentes, métodos do ciclo de vida da aplicação entre outras funções. Esses eventos são as ações exercidas por componentes como cliques, seleção, foco, saída de foco entre outros. E o do ciclo de vida da aplicação, é função de gerenciar as telas da aplicação como, por exemplo, pausar a aplicação caso o aparelho receba uma chamada telefônica (ANDROID, 2016).

O arquivo *BancoDados.java* possui a classe Java denominada de *BancoDados*. Essa realiza a ligação dos eventos dos botões do aplicativo com a base de dados. Foi criado um método *onCreate()* que verifica se no aparelho já possui um banco de dados e suas tabelas criadas. Os outros métodos dessa classe são para consultar ou inserir informações na base de dados.

O conjunto de arquivos, *activity_main.xml* e *MainActivity.java*, representam a primeira tela do sistema. O arquivo *activity_main.xml* possui os componentes *TextView*, com sua propriedades *id* (propriedade que define uma identificação do componente) estabelecido como *txtvwEstoqueId* e a propriedade *text* (propriedade que define a *string* que irá aparecer em tela) definida como “Lista de Estoque”. O componente *ListView*, sua identificação (*id*) ficou definida com o nome *lstvwListaEstoqueId* e o componente *ImageButton* como *imgbtnIncluirId*. Assim, o arquivo *MainActivity.java* possui a classe *MainActivity* que herda as características da classe *Activity*. Essa classe é responsável por fazer as interações com o usuário, possuindo métodos importantes que serão sobrescritos pela classe *MainActivity*.

O método sobrescrito da classe *Activity* será o *onCreate*, que possui a responsabilidade de carregar os *layouts* (arquivos *XML*) e operações de inicialização. Na classe *MainActivity*, esse método buscará as informações dos componentes que estão no arquivo *XML*. Após essa busca, será instanciado um objeto da classe *BancoDados*, que, quando chamado, irá criar a base de dados do aplicativo, caso essa ainda não exista. Logo após será implementado o evento *click* do botão que possui a função de chamar a nova tela para cadastro de produtos.

Um importante método sobrescrito da classe *Activity* é o método *onResume*, esse será chamado toda vez que a classe *MainActivity* receber novamente o foco. Assim, o método *preencherLista*, que preenche o componente *ListView* com os produtos que estão cadastrados no banco de dados para a visualização do usuário.

A próxima tela é a de cadastro de produtos, que contém os arquivos *activity_entrada_produto.xml* e o *EntradaProduto.java* como referência para a visualização e interação com o usuário. O arquivo *activity_entrada_produto.xml* possui os componentes *EditText* para os parâmetros descrição, unidade, quantidade de entrada e um componente *Button*. O arquivo *EntradaProduto.java* possui a classe *EntradaProduto* que também herda as funções da classe *Activity*. A classe sobrescreve o método *onCreate* implementando o evento *onClick* para o componente *Button*, instanciando o objeto da classe *BancoDados* e utilizando o método para salvar as informações na base de dados.

A terceira e última tela do aplicativo é a de saída de produtos, essa tela é composta pelos arquivos *activity_saida_produto.xml* e *SaidaProduto.java*. No arquivo *activity_saida_produto.xml* foi adicionado quatro componentes *TextView* com a função de demonstrar as informações do produto selecionado na lista da tela anterior, além de um *EditText*, local para digitar o valor da quantidade que será retirada do saldo total do produto, e um *Button* para salvar a quantidade de saída no banco de dados. Assim, a classe *SaidaProduto* implementará o evento do componente *Button* salvando a informação na base de dados.

3.1. Ferramentas

As ferramentas JUnit, Espresso e Robotium (Seções 3.1 a 3.3) serão apresentadas com suas principais características e funcionalidades na execução de testes em aplicativos Android. A JUnit e Espresso foram escolhidas por pertencerem ao pacote de desenvolvimento do Android Studio e o *framework* Robotium, por ser uma ferramenta gratuita e por ter um número significativo de usuários registrados em seu site oficial (ROBOTIUM, 2016).

3.1.1. JUnit 4

JUnit (JUNIT, 2016) é um popular *framework open-source* que auxilia na criação de testes automatizados na linguagem Java. Atualmente, essa ferramenta já é instalada e configurada na IDE Android Studio na sua mais recente versão. A ferramenta JUnit facilita a criação e a apresentação dos resultados gerados. Por ter essa facilidade, muitas ferramentas de testes a utilizam como padrão para a implementação de suas rotinas de teste de unidade.

Ao contrário das versões anteriores, a versão 4 da ferramenta JUnit é uma das mais significativas atualizações, fazendo com que o código fique mais compacto e descritivo na sua apresentação. Nessa nova versão, foi adotado o conceito de anotações, que são marcações escritas ao longo do código que possibilita o compilador Java interpretá-las. Assim, essa funcionalidade remove várias necessidades que as versões anteriores existiam, como, nomear os métodos de teste com o prefixo “*test*”.

Em uma classe implementada com a ferramenta de teste JUnit4, pode-se utilizar as seguintes anotações para o processamento dos testes:

- *@Before*: essa anotação serve para identificar um bloco de código que será iniciado antes de cada método de teste. Nas versões anteriores da ferramenta era utilizado o método *setUp()*;
- *@After*: anotação utilizada para identificar um bloco de código que será iniciado após cada método de teste. Nas versões anteriores era utilizado o método *tearDown()*;
- *@Test*: anotação para marcar um método de teste. Uma classe de teste pode conter vários métodos com a anotação *@Test*;
- *@Rule*: em conjunto com classes do projeto que permite adicionar ou redefinir o comportamento de um objeto. A biblioteca JUnit oferece duas classes para o pacote Android, *ActivityTestRule* e *ServiceTestRule*;

- *@BeforeClass*: essa anotação especifica um método estático, ou seja, o método será executado apenas uma vez, nesse caso antes do método de testes;
- *@AfterClass*: essa anotação especifica um método estático, ou seja, o método será executado apenas uma vez, nesse caso após o método de testes;
- *@RunWith*: anotação que referencia a forma de execução da classe.

A configuração da ferramenta JUnit4 no projeto é necessário identificar as dependências da biblioteca no arquivo *build.gradle* do módulo *app*, com o código descrito na Tabela 1.

Tabela 1 – Configuração do Compilador e Dependência no *build.gradle*

Área no <i>Gradle</i>	Código adicionado	Definição
<i>defaultConfig</i>	testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"	Exibe o resultado da análise dos testes
<i>dependencies</i>	androidTestCompile ('com.android.support.test:runner:0.5'){ exclude module: 'support-annotations' } androidTestCompile ('com.android.support.test:rules:0.5'){ exclude module: 'support-annotations' }	Dependências para os compiladores da ferramenta JUnit 4

3.1.2. Espresso

Espresso (ESPRESSO, 2016) é uma ferramenta para teste disponibilizado pelo Google para auxiliar desenvolvedores na criação de testes de Instrumentação. A ferramenta sincroniza as ações de testes executando os segmentos de interfaces. Desde a versão 2.0 a ferramenta está disponível no repositório de suporte de teste do Android.

Para a utilização da ferramenta será necessário verificar se o pacote *Android Support Repository* encontra-se instalado no Android Studio. Para verificar, deve-se acessar o menu ferramentas, e a opção Android, *SDK Manager*. Na pasta “*Extras*” seleciona-se a biblioteca *Android Support Repository*, caso a biblioteca não esteja instalada, basta clicar no botão instalar. O próximo passo é configurar o arquivo *build.gradle* do projeto, será necessário adicionar algumas informações importantes para que a ferramenta Espresso execute suas funções. Seguem as informações para serem preenchidas no arquivo *build.grade*, apresentadas na Tabela 2, para que a ferramenta funcione.

Tabela 2 - Informação do *Gradle*

Área no <i>Gradle</i>	Código adicionado	Definição
<i>defaultConfig</i>	testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"	Exibe o resultado da análise dos testes

<i>dependencies</i>	<pre> androidTestCompile('com.android.support.test.espresso: espresso-core:2.2') { exclude group: 'com.android.support', module: 'support-annotations' } androidTestCompile('com.android.support.test:runner:0.3 ') { exclude group: 'com.android.support', module: 'support-annotations' } </pre>	Dependências para os compiladores da ferramenta Espresso
---------------------	--	--

Para escrever os testes com Espresso é necessário entender como funciona suas principais classes: *Espresso*, *ViewMatchers*, *ViewActions* e *ViewAssertions*.

Espresso:

Classe principal do pacote possui métodos que criam interações com os componentes de interface do aplicativo, conforme métodos principais apresentados na Tabela 3.

Tabela 3 - Métodos da classe Espresso

Métodos Disponíveis	Descrições
<i>onView()</i>	Recebe um <i>ViewMatchers</i> com informações do componente
<i>onData()</i>	É utilizado quando o componente procurado utiliza informações a partir de um adaptador, como uma <i>ListView</i>

ViewMatchers:

Classe do pacote Espresso que retorna um *matcher Views* com a instância do componente procurado. Pode receber várias informações combinadas para facilitar a localização do componente, conforme apresentado na Tabela 4.

Tabela 4 - Métodos Marchers

<i>Matchers</i> Disponíveis	Descrição
<i>withText()</i>	Procura o componente com o texto especificado
<i>withId()</i>	Procura o componente com o <i>id</i> especificado
<i>Hamcrest Matchers</i>	Métodos que auxiliam nas execuções de objetos complexos

ViewActions:

Permite executar um evento do componente. Seus principais métodos são apresentados na Tabela 5.

Tabela 5 - Métodos *Actions*

<i>Actions</i> Disponíveis	Descrição
<i>click()</i>	Clica no componente selecionado
<i>pressKey()</i>	Quando for digitado algum caracter no componente
<i>clearText()</i>	Limpa as informações digitadas no componente

ViewAssertions:

Utiliza métodos para retornar afirmações sobre o estado de um determinado componente selecionando, conforme Tabela 6.

Tabela 6 - Métodos *Assertions*

<i>Assertions</i> Disponíveis	Descrição
<i>matches()</i>	Retorna uma afirmação sobre o componente procurado

A Tabela 7 demonstra a estrutura de uma linha de código com a ferramenta Espresso.

Tabela 7 - Estrutura de uma linha de código com a ferramenta Espresso

<code>onView(ViewMatchers).perform(ViewActions).check(ViewAssertions);</code>

É importante observar que se a classe Espresso não encontrar nenhuma referência do componente apresentado com a classe *ViewMatchers*, inclui todo o resultado encontrado na mensagem de erro, sendo muito útil no momento da análise do problema.

3.1.3. Robotium

Robotium (ROBOTIUM, 2016) é um *framework* para testes de caixa preta e caixa branca criado para facilitar a escrita de testes de unidade e de instrumentação em aplicativos Android. Essa ferramenta simula eventos como toques, cliques, entrada ou modificações de textos e outras ações permitindo assim testar várias funcionalidades do pacote Android, tais como, *Activity*, *Dialogs*, *Toasts*, *Menus*, *Context*, entre outras. Essa *API* foi desenvolvida na linguagem Java e toda sua execução e funcionalidade são herdadas do *framework* de testes JUnit. Será abordada nesse artigo a versão mais recente do Robotium (v. 5.6.1), que se encontra no site oficial da ferramenta.

Para iniciar os testes com a ferramenta Robotium, foi realizado o *download* da biblioteca (*robotium-solo-5.6.1.jar*) e adicionado na pasta “*lib*” do projeto. Caso seja necessário importar o arquivo, deve-se alterar o menu de visualização de módulos do Android Studio para a opção Projeto, assim, será possível visualizar toda árvore de pastas da estrutura do projeto. Arrastando o arquivo JAR até a pasta “*lib*” é o bastante para a importação do pacote Robotium e sua utilização no projeto.

Para configurar a ferramenta, é necessário adicionar a linha apresentada na Tabela 8, no campo *dependencies* do arquivo *build.gradle* (*Module app*). Após a inclusão da linha será necessário sincronizar (*Sync*) o Android Studio para que a IDE funcione em conjunto com a ferramenta Robotium.

Tabela 8 - Informação para o arquivo *Gradle*

<code>androidTestCompile 'com.jayway.android.robotium:robotium-solo:5.6.1'</code>

Após a configuração da ferramenta na IDE Android Studio, observa-se a sua classe principal.

Classe Solo

Solo é classe do pacote Robotium que encapsula todas as ações de interação com as interfaces gráficas para os aplicativos Android. Essa classe permite adicionar textos

em componentes *Edit*, clicar em componentes, seleccionar elementos, alterar *Activity*s entre outras várias funções importantes para manipular os testes. Os métodos dessa classe serão apresentados na Tabela 9.

Tabela 9 - Métodos da classe Solo

Método	Descrição
<i>getView(int id)</i>	Pesquisa na Activity o componente com a ID referenciada
<i>assertCurrentActivity(text, Activity.class)</i>	Assegura que o texto passado se encontra na Activity
<i>waitForText(text)</i>	Espera por um texto na tela, padrão de 5 segundos
<i>clickOnButton(text)</i>	Clica no botão com o texto do parâmetro
<i>clickOnText(text)</i>	Procura o texto na interface atual e clica no componente
<i>enterText(editText, text)</i>	Alterar o valor do campo EditText
<i>searchText(text)</i>	Procura o texto na interface atual e retorna verdadeiro caso encontrado
<i>searchButton(text)</i>	Procura o botão que contém o texto na interface atual
<i>goBack()</i>	O botão de voltar é pressionado
<i>waitForActivity(SegundoActivity.class, 2000)</i>	Espera a Activity referenciada o tempo (em segundos) o valor referenciado como parâmetro
<i>waitForText(text, 2000)</i>	Espera o tempo referenciado aparecer na tela
<i>waitForView(R.id.view)</i>	Espera o ID referenciado aparecer na tela

3.2. Casos de Testes: Instrumentação e Unidade

Nesta seção serão criados casos de testes utilizando as ferramentas apresentadas na seção anterior visando atender os conceitos de testes de instrumentação e unitário em aplicações desenvolvidas em Android. Serão abordados também alguns casos de testes em arquivos XML, demonstrando o funcionamento das ferramentas na execução de testes nas informações contidas nesses arquivos.

3.2.1. Testes de Instrumentação

A Seção 3.4.1 mostra como se aplicam casos de testes de forma prática com as ferramentas para teste funcional apresentadas nesse artigo. Serão realizados casos de testes possíveis de serem executados no aplicativo desenvolvido empregando os recursos das ferramentas Espresso e Robotium.

3.2.1.1. Caso de Teste: Verificar se a tela principal do aplicativo foi aberta

Esse primeiro caso de teste irá verificar se o aplicativo foi aberto. A tela *MainActivity* apresenta um componente *TextView* com a sua propriedade *text* com a expressão “Lista Estoque”. Para criar o método de teste *testAbrirMainActivity()*, será procurada a *string* em questão na tela.

A Tabela 10 apresenta o método *testAbrirMainActivity()* utilizando a ferramenta Espresso. Nesse código, o método *onView()* recebe o *ViewMatcher* retornado do método *withText()*, e esse recebe como parâmetro a *string* que será procurada, que no exemplo será “Lista de Estoque”.

O método *check* espera um *ViewAssertion*, que tem como função retornar verdadeiro se a apresentação da *string* em tela for válida. Para que isso ocorra, será informado o método *matches(isDisplayed())*, que recebe um *ViewMatcher* e retorna um *ViewAssertion*. O método *isDisplayed()* é um *ViewMatcher* que verifica se algo foi exibido em tela e retorna um *ViewAssertion*.

Tabela 10 - Método *testAbrirMainActivity()* escrito com a ferramenta Espresso

```
@Test
public void testAbrirMainActivity() throws Exception{
    onView(withText("Lista de Estoque")).check(matches(isDisplayed()));
}
```

Na Tabela 11 é exibido o método *testAbrirMainActivity()* com a ferramenta Robotium. O código apresenta o método *waitForActivity()* da classe *Solo*, esse método aguarda alguns segundos o carregamento da classe *Activity* que foi passada como parâmetro. Assim, foi instanciado um objeto da classe *TextView* para que receba as informações do componente *txtvwEstoqueId* do projeto. Com isso, será executado o método do pacote JUnit, *assertTrue()*, aguardando a resposta da informação esperada.

Tabela 11 - Método *testAbrirMainActivity()* escrito com a ferramenta Robotium

```
public void testAbrirMainActivity() throws Exception {
    solo.waitForActivity("MainActivity", 2000);
    TextView cabecalhoTelaPrincipal = (TextView) solo.getView(R.id.txtvwEstoqueId);
    assertTrue(cabecalhoTelaPrincipal.getText().toString().equals("Lista de Estoque"));
}
```

3.2.1.2. Caso de Teste: Verificar se a janela do cadastro de produto foi aberta

O segundo caso de teste será para verificar se o botão, que se encontra na tela principal, abre a tela de cadastro de produtos. O tipo de botão adicionado é um *ImageView*, este será denominado de *imgbtnIncluirId* e irá instanciar a nova classe *EntradaProduto*.

O *testAbrirTelaCadastro()*, demonstrado na Tabela 12, foi implementado utilizando a ferramenta Espresso. No código foi adicionado uma variável *string* de nome *finalTexto* que recebe o valor “Cadastro Produto”. Após a definição da resposta esperada, é adicionada uma linha no código com a intenção de executar o evento *click* do componente. Para que a ferramenta Espresso consiga esse evento, será de extrema importância localizar o componente *imgbtnIncluitId* com o método *onView()*. Com o componente do botão será utilizado o método *perform*, para retornar um objeto *ViewInteraction*, que é uma interface primária para executar uma ação. Esse objeto retornado recebe como parâmetro um *ViewAction* que no código está representado pelo método *click*. Assim, será procurado o valor da variável *finalTexto* na nova tela que será aberta.

Tabela 12 - Método `testAbrirTelaCadastro()` escrito com Espresso

```
@Test
public void testAbrirTelaCadastro() throws Exception{
    final String finalTexto = "Cadastro Produto";
    onView(withId(R.id.imgbtnIncluirId)).perform(click());
    onView(withId(R.id.txtvwTituloEntradaEstoqueld)).check(matches(withText(finalTexto)));
}
```

Na Tabela 13, será demonstrado o mesmo código visto anteriormente com a ferramenta Robotium. Foi criada uma *string*, denominada de *finalTexto*, com o valor “Cadastro Produto”. A próxima linha de código é a utilização o objeto da classe *Solo* com o método *clickOnView()*. Esse método serve para implementar o evento *click* em um componente *View*, que no exemplo é representado pelo componente *imgbtnIncluirId*. Após clicar no componente são esperados dois segundos para a construção da nova *Activity*, *EntradaProduto*. Assim, será instanciado um objeto da classe *TextView* que receberá as informações do componente da tela através do método *getView*, pertencente à classe *Solo*. Com isso, serão comparadas as informações da *string finalTexto* com a propriedade *text* do componente *TextView*, denominado *cabecalhoTelaCadastro*.

Tabela 13 - Método `testAbrirTelaCadastro()` escrito com Robotium

```
public void testAbrirTelaCadastro() throws Exception{
    final String finalTexto = "Cadastro Produto";
    solo.clickOnView(solo.getView(R.id.imgbtnIncluirId));
    solo.waitForActivity(EntradaProduto.class, 2000);
    TextView cabecalhoTelaCadastro = (TextView)
solo.getView(R.id.txtvwTituloEntradaEstoqueld);
    assertTrue(finalTexto.equals(cabecalhoTelaCadastro.getText().toString()));
    solo.goBack();
}
```

3.2.1.3. Caso de Teste: Teste de cadastro de produto

O intuito desse teste será digitar informações nos componentes *EditTexts* da *Activity EntradaProduto* e, após esse preenchimento, salvar essas informações na base de dados da aplicação. Nos dois casos de testes criados serão definidas três variáveis que representaram as informações para serem incluídas nos componentes *EditTexts*. Essas informações para os campos descrição, unidade e quantidade.

Para utilizar a ferramenta Espresso será pesquisado na janela principal o componente *ImageButton* e após será utilizado o evento *click*. Com isso, a nova tela *EntradaProduto* será aberta e, assim, as informações das variáveis *descricao*, *unidade* e *quantidade* serão adicionados nos componentes *EditTexts* com o método *perform()*. O método *perform* recebe o método *closeSoftKeyboard()* para fechar o teclado virtual. Após esse preenchimento dos campos, será chamado o evento *click* do componente *Button btnSalvarProdutoId* para salvar as informações e retornar à tela principal. Na tela principal será procurado no componente *ListView* a frase “Martelo – SALDO: 10”, caso encontrado o caso de teste terá validade.

Tabela 14 - Método *testTelaCadastroProduto()* com Espresso

```
@Test
public void testTelaCadastroProduto() throws Exception{
    final String descricao = "Martelo";
    final String unidade = "unid";
    final int quantidade = 10;
    onView(withId(R.id.imgbtnIncluirId)).perform(click());
    onView(withId(R.id.edtxtDescricaoId)).perform(typeText(descricao), closeSoftKeyboard());
    onView(withId(R.id.edtxtUnidadeId)).perform(typeText(unidade), closeSoftKeyboard());
    onView(withId(R.id.edtxtQuantidadeId)).perform(typeText(String.valueOf(quantidade)),
    closeSoftKeyboard());
    onView(withId(R.id.btnSalvarProdutold)).perform(click());
    onView(allOf(withId(R.id.lstvwListald), withText("Martelo - SALDO: 10")));
}
```

Com a ferramenta Robotium será utilizado o método *enterText()* do objeto Solo para preencher os campos *EditTexts* encontrados na *Activity EntradaProduto*. Após o procedimento de preenchimento será utilizado o método *clickOnButton()* para simular o evento *click* no componente *Button* da tela. Se o cadastro das informações ocorrer conforme previsto, aparecerá uma mensagem utilizando a classe *Toast* do pacote Android com a seguinte frase “O Produto foi salvo com sucesso!”. Se a frase for encontrada, o teste ficará validado.

Tabela 15 - Método *testTelaCadastroProduto()* com Robotium

```
public void testTelaCadastroProduto() throws Exception{
    final String desc = "Madeira";
    final String unid = "m3";
    final String qnt = "3000";
    solo.clickOnView(solo.getView(R.id.imgbtnIncluirId));
    solo.waitForActivity(EntradaProduto.class, 2000);
    EditText descProduto = (EditText) solo.getView(R.id.edtxtDescricaoId);
    EditText unidProduto = (EditText) solo.getView(R.id.edtxtUnidadeId);
    EditText quantProduto = (EditText) solo.getView(R.id.edtxtQuantidadeId);
    solo.enterText(descProduto,desc);
    solo.enterText(unidProduto,unid);
    solo.enterText(quantProduto,qnt);
    Button botaoSalvar = (Button) solo.getView(R.id.btnSalvarProdutold);
    solo.clickOnButton(botaoSalvar.getText().toString());
    assertTrue(solo.waitForText("O Produto foi salvo com sucesso!"));
    solo.goBack();
}
```

3.2.1.4. Caso de Teste: Teste de saída de produto

O próximo caso de teste é verificar a saída do produto do estoque. Essa função é realizada quando um item na lista for selecionado e, assim, a classe *SaidaProduto* é acionada.

O teste *testTelaSaidaProduto()*, feito com a ferramenta Espresso, é implementado utilizando método *onData()*. O método *onData()* é um método estático que recebe um *Matcher* como parâmetro e retorna um objeto da classe *DataInteraction*. A classe *DataInteraction* possui a função de criar uma *interface* que relaciona dados exibidos em um *AdapterView*, com o elemento utilizado para o preenchimento do componente *ListView* na tela *MainActivity*. O método *onData()* seleciona um item do

componente *ListView*, marcando a posição referente ao *index* passado com o método *atPosition()*, no exemplo será selecionado o *index* zero, ou seja, a primeira linha. Assim, a nova tela de saída de produto aparecerá com o campo *EditText* para a saída do produto. Utilizando o método *perform()*, será digitado no campo *EditText* o valor representado pela variável *qntSaida* e, após esse procedimento, o evento *click* do componente *Button* será chamado salvando a informação na base de dados. Quando retornar à tela inicial, será comparado o valor da resposta com a informação que está apresentada na lista.

Tabela 16 - Método com Espresso

```
@Test
public void testTelaSaidaProduto() throws Exception{
    final int qntSaida = 2;
    onData(anything()).inAdapterView(withId(R.id.lstvwListald)).atPosition(0).perform(click());
    onView(withId(R.id.edttxtQuantidadeSaidald)).perform(typeText(String.valueOf(qntSaida),
closeSoftKeyboard());
    onView(withId(R.id.btnSalvarSaidald)).perform(click());
    onView(allOf(withId(R.id.lstvwListald), withText("Martelo - SALDO: 8")));
}
```

O teste da tela de saída de produto utilizando a ferramenta Robotium segue os mesmos passos do teste anterior. Será criada uma variável do tipo *string*, denominada de *desc* e definida com o valor “Madeira”. Com essa informação, será localizado e clicado, na tela principal, com o método da classe *Solo* em *clickOnText()*. Assim, a nova *Activity SaidaEstoque* será criada para o usuário. Após a abertura da nova janela será adicionado o valor da variável “*qnt*” que, no exemplo, será 1000 no campo *EditText*. Quando terminado, será selecionado o evento *click* do botão para salvar a quantidade de saída do produto e, após, será verificado se a resposta foi satisfatória para validar o caso de teste.

Tabela 17 - Método com Robotium

```
public void testTelaSaidaProduto() throws Exception{
    final String desc = "Madeira";
    final int qnt = 1000;
    solo.clickOnText(desc);
    solo.waitForActivity(SaidaProduto.class, 2000);
    EditText quantProduto = (EditText) solo.getView(R.id.edttxtQuantidadeSaidald);
    solo.enterText(quantProduto, String.valueOf(qnt));
    Button botaoSalvar = (Button) solo.getView(R.id.btnSalvarSaidald);
    solo.clickOnButton(botaoSalvar.getText().toString());
    assertTrue(solo.waitForText("Saída de estoque cadastrada com sucesso!"));
    solo.goBack();
}
```

3.2.2. Testes de Unidade

Todos os testes implementados até o momento foram teste de interface, ou seja, as ferramentas utilizavam as interfaces gráficas do aplicativo para a execução das suas rotinas. Os próximos casos de testes demonstrados utilizaram o *framework* JUnit para criar rotinas de testes em métodos que não utilizam a interface gráfica com o usuário. No aplicativo Estoque foi criada a classe *BancoDados* como exemplo para a realização desses testes.

3.2.2.1. Caso de Teste: Teste para listagem de produto

Na classe *BancoDados* foi implementado um método, denominado *onSelect()*, esse possui o intuito de retornar a resposta de qualquer consulta *SQL* inserida na base de dados. O método retorna um objeto da classe *Cursor*, que possui uma interface de acesso de leitura e escrita a um conjunto retornado pela consulta *SQL*. Assim, foi desenvolvido um caso de teste para verificar se o retorno desse método é o esperado.

É observado na Tabela 18 o código para o teste *meuTesteListarProdutos()*. Esse método utiliza uma variável do tipo *string* que possui o comando *SQL* para retornar todos os produtos cadastrados na base de dados. Com o objeto instaciado da classe *BancoDados* será passado como parâmetro o comando *SQL* para o método *onSelect()* retornando as informações para o objeto lista da classe *Cursor*. Assim, verifica-se se a quantidade de linhas retornadas é maior que zero.

Tabela 18 - Caso de Teste meuTesteListarProdutos()

```
@Test
public void meuTesteListarProdutos(){
    String sql = "Select * from produto";
    Cursor lista = banco.onSelect(sql);
    assertTrue("Lista não está vazia!", lista.getCount() > 0);
}
```

3.2.2.2. Caso de Teste: Teste salvar produto na base de dados

O próximo método (Tabela 19) a ser visto é o teste *meuTesteSalvarProduto()* que apresenta uma rotina para verificar se o método *saveProduto()* da classe *BancoDados* consegue executar suas funções como esperado.

Foi instanciado um objeto da classe *BancoDados* denominado de “banco”, para chamar o método e passar os parâmetros necessários para o cadastro do produto. Assim, será chamado o método *onSelect()* para consultar a informação e retornar a quantidade de informação encontrada no base de dados. Caso a resposta for maior que zero, o caso de teste será validado.

Tabela 19 - Caso de Teste meuTesteSalvarProduto()

```
@Test
public void meuTesteSalvarProduto(){
    banco.saveProduto("Brita","m3",2000);
    String sql = "Select idproduto from produto where descricao = 'Brita'";
    Cursor lista = banco.onSelect(sql);
    assertTrue("Possui o produto no banco de dados!",lista.getCount() > 0);
}
```

3.2.2.3. Caso de Teste: Teste salvar saída de estoque

Um caso de teste que pode ser implementado é o de saída de produto (Tabela 20). Com o objeto instanciado da classe *BancoDados* chama-se o método *saveEstoque()* e passa-se o *idproduto* e a quantidade de saída desse produto. Assim, verifica-se se a resposta aguardada será igual à retornada pelo método *saldo()*.

Tabela 20 - Caso de Teste meuTesteSalvarSaida()

```
@Test
public void meuTesteSalvarSaida(){
    banco.saveEstoque(100,1);
    assertEquals(1900,banco.saldo(1));
}
```

3.2.2.4. Caso de Teste: Teste método meuTesteSaldo()

O método *saldo* será verificado utilizando um caso de teste unitário (Tabela 21). Será informado o saldo esperado do produto e comparado com o resultado encontrado na base de dados.

Tabela 21 - Caso de Teste meuTesteSaldo()

```
@Test
public void meuTesteSaldo(){
    assertEquals(1900,banco.saldo(1));
}
```

3.2.3. Testes de Unidade em arquivos XML

Na programação de aplicativos Android existem os arquivos XML que possuem a finalidade de armazenar variáveis que servirão como padrão para o projeto. Esses arquivos possuem informações como tamanho e tipo de fonte, altura e largura do componente, imagens e cores utilizados no projeto entre outras informações que os desenvolvedores podem informar como padrão. Assim, realizar testes nesses arquivos é de vasta importância para ter um projeto consistente. Nessa seção serão apresentados casos de testes em alguns arquivos *XML*, utilizando a classe *R* nativa do Android e criada automaticamente no projeto pelo Android Studio, para demonstrar a funcionalidade das ferramentas.

3.2.3.1. Caso de Teste: Nome do aplicativo

O arquivo *strings.xml* tem a função de concentrar as *strings* utilizadas no projeto. Essas *strings* permitem ao desenvolvedor padronizar textos em apenas um local, assim, quando houver a necessidade de modificá-los, não precisará percorrer todo o sistema para encontrá-las. Esse mecanismo é muito utilizado na internalização do aplicativo, ou seja, traduzir o aplicativo para outras línguas.

No caso de teste da Tabela 22 foi utilizado a ferramenta Robotium para instanciar o objeto da classe *Solo* e utilizar a classe *R* do Android. A classe *R* possui a função de acessar os recursos da programação Android, como informações de componentes e informações de arquivos da plataforma. Com isso, será buscado o arquivo *strings.xml* e selecionada a variável responsável por armazenar o nome do aplicativo. Assim, utiliza-se *assertEquals()* para comparar o nome armazenado na variável *app_name*.

Tabela 22 - Caso de Teste testNomeAplicativo()

```
public void testNomeAplicativo(){
    String nome_aplicativo = solo.getString(R.string.app_name);
    assertEquals(nome_aplicativo,"Estoque");
}
```

3.2.3.2. Caso de Teste: Testar dimensões de componentes.

Outro arquivo importante é o *dimens.xml*, que armazena as dimensões definidas para os componentes no projeto (Tabela 23). No aplicativo Estoque foi criada uma dimensão para servir como exemplo, foi adicionada na variável *altura_tela* que define a altura do componente *ListView* alocado na classe *MainActivity*. No Android as dimensões são definidas pela unidade *dip* (*Density-independent Pixels*), relativa à resolução de tela.

Tabela 23 - Caso de Teste testTamanhoListView()

```
public void testTamanhoListView(){
    String altura_tela = solo.getString(R.dimen.altura_tela).toString();
    assertEquals( altura_tela,"400.0dip");
}
```

4. Resultados obtidos pelas ferramentas

Os métodos de testes criados tanto pelas ferramentas Espresso e Robotium, quanto pela JUnit4 serão executados pela ferramenta JUnit contida na IDE Android Studio. Na IDE existem duas formas de criação de compiladores das ferramentas. A primeira é criada com a seleção da classe de teste com o botão direito do mouse e clicando na opção *Run*. A segunda opção é definir manualmente o compilador com as configurações desejadas. Para definir esse mecanismo na ferramenta é necessário ir ao menu *Run* (Executar) do Android Studio, opção *Edit Configurations...* (Editar Configurações...). Na nova janela serão apresentadas as informações para a criação. Para compilar as ferramentas demonstradas nesse artigo, basta escolher a opção *Android Tests* (Testes Android) e adicionar um novo compilador. O parâmetro *Name* (nome), define o novo compilador e o parâmetro *Module* (Módulo) qual o módulo que sofrerá as rotinas, no exemplo do artigo foi escolhido o módulo “*app*”. Assim, o novo compilador será criado e será responsável em executar as classes de testes do projeto.

Quando executar o compilador para a classe de teste, será aberta uma janela mostrando uma barra com o andamento dos testes. Essa barra aparecerá com cores diferentes de acordo com o andamento do processo: verde, caso todos os testes tenham obtidos resultados satisfatórios, como mostrado nas Figuras 2 a 4, e vermelho, caso algum teste tenha encontrado um resultado não esperado.

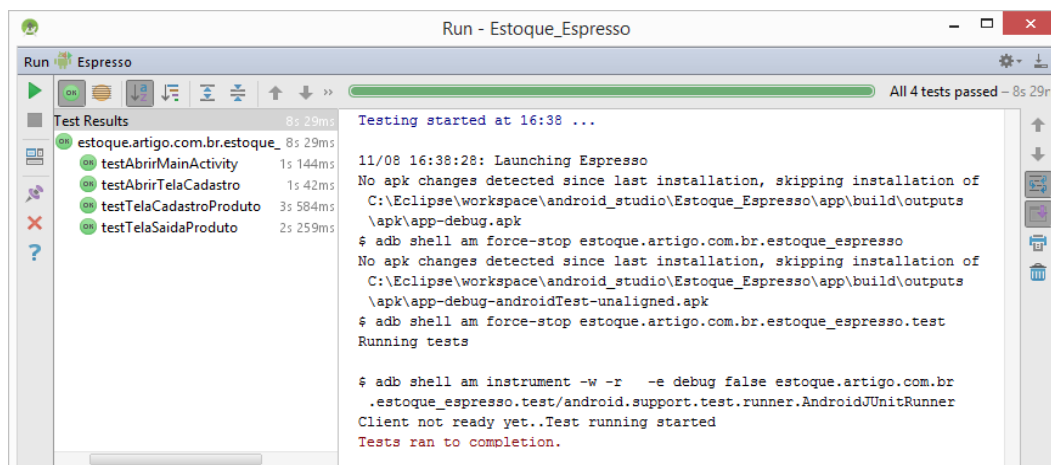


Figura 2 - Resultado da Ferramenta Espresso

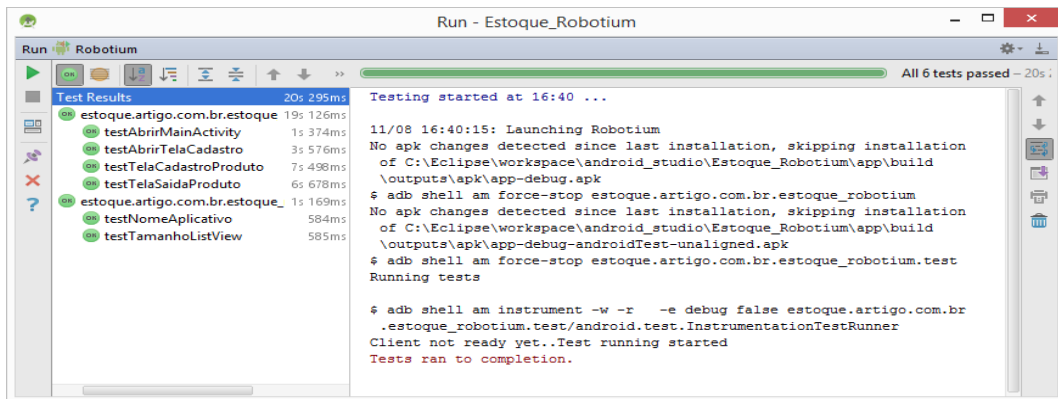


Figura 3 - Resultado da Ferramenta Robotium

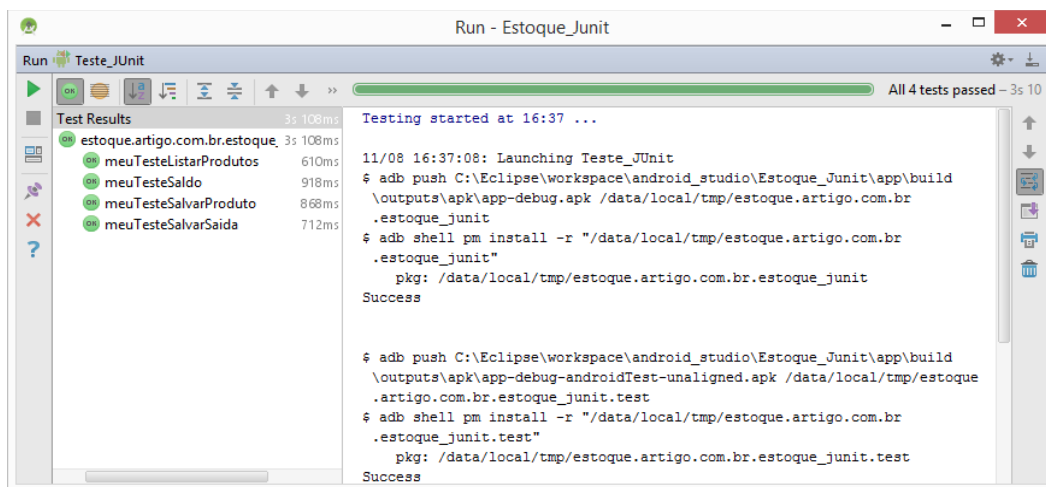


Figura 4 - Resultado da Ferramenta JUnit

As imagens demonstram os testes executados com sucesso, pois todos foram apresentados com a cor verde. Assim, foram comprovados todos os casos de testes, pois todos obtiveram os resultados esperados.

Para exemplificar uma situação de falha, alterou-se o caso de teste criado pela ferramenta Robotium. Foi alterado o caso de teste *testNomeAplicativo*, para que aguarde uma resposta que não condiz com a informação prevista pelo aplicativo.

Como apresentado na Figura 5, a barra do JUnit tornou-se de cor vermelha e o método *testNomeAplicativo* modificado aparece selecionado em azul, pois o método em questão não passou no teste. A área a direita da figura descreve, na primeira linha, a falha ocorrida, que no exemplo demonstrado avisa que era esperada a *string* "Estoque", mas estava "Estoque de Produto".

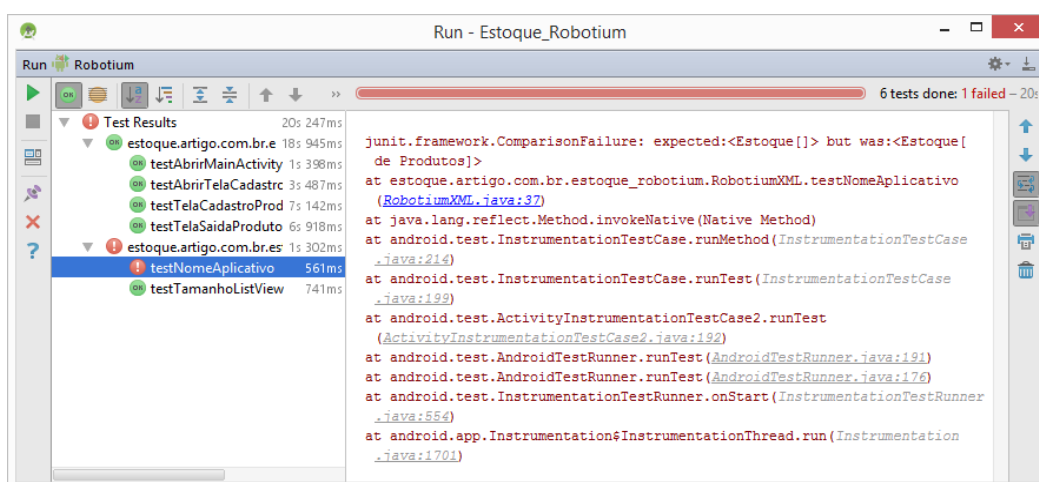


Figura 5 - Falha com a ferramenta Robotium

4.1. Quadro comparativo das ferramentas

Foram utilizados alguns grupos de avaliações de ferramentas de testes desenvolvidos por Veloso (2010) para seguir como parâmetros nesse artigo. Não foram utilizados todos os grupos apresentados por Veloso, pois muitos não possuem alguma referência com esse estudo. Assim, serão identificados os pontos de cada ferramenta utilizada nesse estudo.

As ferramentas se igualam em vários grupos propostos, entretanto a ferramenta Robotium se destacou em utilização e capacidade de implementar testes de unidade e de instrumentação.

Alguns aspectos técnicos relevantes, entre eles o avaliador de cobertura e o gerador de relatório definido pelo usuário, não estão presentes em nenhuma das ferramentas citadas pelo artigo. O avaliador de cobertura seria um diferencial, pois demonstraria a abrangência dos testes no aplicativo, sendo essencial essa informação para a equipe de testadores.

As ferramentas Robotium e Espresso, como já foi visto neste trabalho, utilizam funcionalidades do *framework* JUnit. Assim, muitos dos aspectos técnicos destacados na Tabela 4, foram importadas da ferramenta JUnit.

Tabela 4 - Quadro comparativo das ferramentas

Grupos	Aspectos técnicos	Robotium	Espresso	JUnit
Gerador Manual de teste	Mecanismo de captura-reprodução	X	X	
	Uso de linguagem de alto nível	X	X	
Avaliador de Testes	Avaliador de Cobertura			
Gerador de Relatórios	Gerador de relatório definido pelo usuário			
	Acesso a qualquer informação de testes existentes no modelo de testes	X	X	X

Arquitetura da Ferramenta	Uso de software livre	X	X	X
	Utilizar terminologia adequada ao contexto	X	X	X
Executor de Teste	Gerador de Log de execução de testes	X	X	X
	Simulador de interface de hardware e software			X
	Integração com uma linguagem de script para a configuração do ambiente de testes	X	X	X
	Povoador de dados	X	X	
	Analisador de falhas	X	X	X

5. Conclusão

A utilização das teorias sobre testes de software ficaram relevantes atualmente no ambiente de desenvolvimento de aplicações móveis. Pois se uma empresa não utilizar suas teorias pode acarretar em um produto com baixa credibilidade no mercado. Assim, a utilização de ferramentas que auxiliam nessa importante tarefa é cada vez mais desejada.

Baseado nisso, esse estudo teve o objetivo de demonstrar ferramentas que auxiliam no desenvolvimento de testes de unidade e de instrumentação em uma aplicação Android. Apesar do pouco material encontrado na literatura sobre as ferramentas JUnit, Robotium e Espresso, foi demonstrado como são simples e poderosas essas ferramentas, tanto na instalação quanto na utilização. Dos conhecimentos adquiridos na realização deste trabalho destacam-se, um aprofundamento na teoria de testes de software e de ferramentas de testes para ambiente Android e a implementação de testes nos aplicativos dessa plataforma. Com essas informações, estudos futuros poderão ser realizados, abordando outros casos de testes, como, por exemplo, acessar informações na internet entre outras.

O trabalho ainda apresentou um estudo de caso de forma a evidenciar algumas situações mais comuns que podem ocorrer em um aplicativo. A abordagem de diferentes formas de teste, como teste unitário e teste de instrumentação, contribuem significativamente para a melhoria do produto, objetivando assim, uma maior confiança do usuário final.

Referências

- PRESMAN, Roger S. **Engenharia de Software: Uma abordagem Profissional**. 7. ed. Amgh Editora Ltda, 2011.
- VELOSO, Janielton de Sousa. **Avaliação de Ferramentas de Apoio ao Teste de Sistemas de Informação**. 2010. TCC (Graduação) - Curso de Informática, Departamento de Informática de Estatística, Universidade Federal do Piauí, Teresina, 2010.
- ANDROID, **User's Guide. Test Your App**. Disponível em: <<https://developer.android.com/studio/test/index.html>>. Acesso em: 07 out. 2016.
- ESPRESSO. **Android user interface testing with Espresso**: Tutorial. 2106. Disponível em:

<<http://www.vogella.com/tutorials/AndroidTestingEspresso/article.html>>. Acesso em: 22 out. 2016.

ESPRESSO, **Testing UI for a Single App**. 2016. Disponível em <https://developer.android.com/training/testing/ui-testing/espresso-testing.html>. Acesso em: 22 out. 2016.

ANDROID, Developer. **Building Local Unit Tests**. 2016. Disponível em: <<https://developer.android.com/training/testing/unit-testing/local-unit-tests.html>>. Acesso em: 19 out. 2016.

JUNIT, **Build Local Unit Test**. 2016. Disponível em: <<https://developer.android.com/training/testing/unit-testing/local-unit-tests.html>>. Acesso em: 23 out. 2016.

ANDROID STUDIO. Disponível em <<https://developer.android.com/studio/index.html?hl=pt-br>>

Acesso em 15 out. 2016.

ROBOTIUM. Disponível em <<https://github.com/RobotiumTech/robotium/wiki/Downloads>> Acesso em: 20 out. 2016.