Automatizando Testes de Interface para Dispositivos Móveis

Eduardo Finotti, Anita Maria da Rocha Fernandes

Pós-Graduação em Qualidade e Engenharia de Software – Universidade do Vale do Itajaí (UNIVALI) Rodovia SC 401, 5025 – 88.032-005 – Florianópolis – SC – Brasil

eduardo.finotti@hotmail.com, anita.fernandes@univali.br

Abstract. *The demand for quality is increasing in every aspect of our society, especially with regard to technology, in view of the increase in the consumption of mobile devices. Nobody wants something that was working efficiently, to stop working, both from the user's point of view and (and especially) from the point of view of the companies. To minimize this type of problem it was described how to start an automated testing project for mobile devices using BDD.*

Resumo. *A exigência de qualidade está cada vez maior em todos os aspectos de nossa sociedade, principalmente no que se diz respeito a tecnologia, tendo em vista o aumento no consumo de dispositivos móveis. Ninguém quer que algo que estava funcionando com eficiência, deixe de funcionar, tanto do ponto de vista do usuário quanto (e principalmente) do ponto de vista das empresas. Para minimizar esse tipo de problema foi descrito como iniciar um projeto de testes automatizados para dispositivos móveis utilizando BDD.*

1 Introdução

Com o aumento na utilização dos dispositivos móveis, a exigência dos consumidores também aumentou. A maioria dos consumidores necessita de produtos com alta qualidade. Segundo a ISO/IEC 8402 de 1994 [NBR ISO 8402 1994], qualidade de um produto se define da seguinte forma: “Conjunto de propriedades de um produto ou serviço, que lhe conferem aptidões para satisfazer as necessidades explícitas ou implícitas.”

Nos últimos anos houve um aumento exponencial no uso de aplicativos móveis, assim como cresceu também o número de plataformas e fabricantes de celulares [Starov *et al* 2015]. A necessidade de entregar software de qualidade em ciclos curtos veio com esse crescimento, pois a exigência dos usuários está cada vez maior, em todos os aspectos (funcional, usabilidade, performance e segurança) [Starov *et al* 2015; Gao *et al* 2014; Dantas 2009].

Por outro lado, também existem as empresas cujo é foco reter mais clientes para obter mais lucros. Um fator prezado pelas empresas é a satisfação dos clientes em relação ao produto que está sendo entregue. Para atingir essa satisfação, é preciso que o software tenha boa qualidade, em todos os sentidos, como possuir boa usabilidade, atingir os objetivos propostos e atender as necessidades do cliente, por exemplo.

Para que esses erros ou falhas nos softwares sejam identificados, é feito o uso dos métodos de testes. Segundo Hetzel [1983 *apud* Hetzel 1987], "teste é qualquer atividade que vise a avaliar uma característica ou recurso de um programa ou sistema. Teste é a medida da qualidade do software".

Teste de software é um processo extremamente importante, devido a uma série de benefícios que ele oferece, tais como o resultado eficaz, aumento da qualidade do software, evidenciando os erros e possibilitando a correção antes de chegar ao consumidor final [Dantas 2009]. Muitas vezes, uma alteração na aplicação pode acarretar em impactos não mensurados no sistema, fazendo com que outras partes da aplicação parem de funcionar ou venha a exibir falha. Sendo assim necessário uma re-execução dos casos de testes que já existiam.

Conforme Pressman (2011, p. 404), esse teste “[...] pode ser executado manualmente, reexecutando um subconjunto de todos os casos de teste ou usando ferramentas automáticas de captura/re-execução.", sendo esse um dos principais motivos para a sua automatização, o que é o foco deste trabalho. Ou seja, quando ocorre uma re-execução de um conjunto de casos de testes depois de uma alteração na aplicação, isso garante que o que estava funcionando antes, ainda funciona perfeitamente.

De acordo com Bartié (2002, p. 196), a definição de testes automatizados é "[...] a utilização de ferramentas de testes que possibilitem simular usuários ou atividades humanas de forma a não requerer procedimentos manuais no processo de execução dos testes.". Mesmo sendo muito eficiente, muitas empresas não utilizam dessa prática, pois acreditam que o esforço (preço) para o desenvolvimento destes testes é muito alto.

Ainda segundo Bartié (2002, p. 181), "A automação exige um esforço inicial de criação, porém possibilita uma incomparável eficiência e confiabilidade, impossível de ser atingida com procedimentos manuais".

Complementando essa afirmação, Molinari (2003, p. 104) fala que testes automatizados "[...] permitirá aumentar a profundidade e abrangência dos casos de testes envolvidos.". Por isso, fica claro que essa prática nos dá muito mais confiabilidade na aplicação, que será entregue para o cliente final.

Este artigo tem como objetivo apresentar uma solução de automação de testes para dispositivos móveis utilizando BDD (especificação baseada em comportamento) onde está prática garante que a aplicação entregue para o cliente estará com sua qualidade garantida quando acorrem mudanças em suas funcionalidades, ou seja, garante que nada parou de funcionar em sua aplicação.

Para descrever os cenários de testes será utilizado BDD (*Behaviour Driven Development* ou ainda uma tradução Desenvolvimento Guiado por Comportamento [North 2016]. Segundo Dan North (2003), criador do BDD, "o desenvolvimento conduzido por comportamentos é sobre a implementação de um aplicativo por descrevendo o seu comportamento a partir da perspectiva de seus *stakeholders*".

Não será usada nenhuma empresa como base para este artigo, e como exemplo será utilizado um aplicativo grátis disponível para qualquer usuário. Com este artigo busca-se contribuir para a disseminação do conhecimento de automação de testes para dispositivos móveis.

2 Testes de software

Um teste de software serve para provar que aquilo que foi pedido pelo cliente está funcionando corretamente.

Segundo Sommerville (2011, p. 144) diz que um teste mostra que um sistema faz o que foi proposto e também para descobrir os defeitos antes que o cliente use. "Os resultados do teste são verificados a procura de erros, anomalias ou informações sobre os atributos não funcionais do programa.".

De acordo com Pressman (2011, p. 402): "Teste é um conjunto de atividades que podem ser planejadas com antecedência e executadas sistematicamente". Na mesma ideia, os testes de software são formados por um processo sistemático que tem como principal objetivo o de encontrar erros [Bartie 2002].

Apesar das definições serem diferentes, elas se complementam, nos trazendo os conceitos de testes de software.

Porém, segundo a IEEE, uma das definições mais populares sobre teste é: "*The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component*".

O teste de software pode ser visto como um tipo específico de teste, cujo componente ou processo é submetido a uma avaliação de acordo com os resultados produzidos em determinado ambiente.

Algumas definições básicas dentro de testes de software segundo Delamaro et. al. (2007), é:

- **Defeito**: passo, processo ou definição de dados incorretos;
- **Engano**: ação humana que produz um defeito;
- **Erro**: durante a execução de um programa é caracterizado por um estado inconsistente ou inesperado; e
- **Falha**: resultado produzido é diferente do resultado esperado, pode ser ocasionado por um erro

2.1 Testes mobile

Testes em dispositivos móveis possuem características especiais, e não devem ser comparados com testes para web e desktop [Knott, 2015]. Uma das grandes dificuldades dentro dos testes de aplicações *mobile*, é a dificuldade em validar em diversos aparelhos, pois esses possuem suas particularidades, como câmera, GPS, sistema operacional, entre outras. Ou seja, todos os *devices* possuem essas características, mas muitos funcionam de maneira diferente um do outro, assim, muitas vezes limitando a quantidade de dispositivos testados.

Além desse desafio, não podemos esquecer da atualização do sistema operacional, que a

cada versão pode trazer mudanças em seu funcionamento, fazendo do teste automatizado um facilitador na hora de validar se sua aplicac a o ainda funciona nessa nova versão do sistema operacional.

Para uTest (2015), outras preocupac o es sa o com o consumo de bateria, assim como a usabilidade e facilidade de instalac a o. Esses sa o alguns dos fatores que podem fazer os seus usua rios escolherem entre o seu aplicativo ou do seu concorrente.

2.2 Testes automatizados

Testes automatizados são "[...] a utilizac a o de ferramentas de testes que possibilitem simular usua rios ou atividades humanas de forma a na o requerer procedimentos manuais no processo de execuc a o dos testes." [Bartie , 2002, p. 196]. Assim, Molinari (2003, p. 104), diz que a utilização de testes automatizados "[...] permitira aumentar a profundidade e abrange ncia dos casos de testes envolvidos.".

Para Bartie (2002, p. 181), uma vantagem de sua utilizac a o e que "A automac a o exige um esforc o inicial de criac a o, pore m possibilita uma incompara vel e ficie ncia e con fiabilidade, impossi vel de ser atingida com procedimentos manuais.".

A automac a o de testes na o faz parte de um ni vel especi fico de teste de software,porém este garante que os testes de regressa o sejam realizados. De acordo com Pressman (2011, p. 404) esse teste "[...] pode ser executado manualmente, reexecutando um subconjunto de todos os casos de teste ou usando ferramentas automa ticas de captura/reexecuc a o.".

Ou seja, testes de regressa o podem ser executados manualmente, ou de forma automatizada, e este último será apresentado neste artigo.

2.3 BDD - Behavior-Driven Development

Behavior-Driven Development, ou desenvolvimento orientado ao comportamento, foi inicialmente proposto por Dan North. O BDD envolve práticas ágeis e foi projetado para torná-las mais acessíveis e e ficazes para as equipes ágeis na entrega de um software.

De acordo com Smart (2014), O BDD é um conjunto de práticas de engenharia de software desenvolvido para auxiliar equipes a construir e entregar software de maior valor e qualidade de forma mais rápida.

Segundo Gohil (2011), o BDD indica como o sistema deve se comportar, escrito com linguagem padrão (português, inglês, etc.), facilmente compartilhada entre os *stakeholders* (participantes do projeto). Na mesma linha, para Evans (2003), o BDD visa promover um entendimento comum do negócio entre os *stakeholders* de negócio e a equipe de desenvolvimento, pois descreve o comportamento do sistema.

De acordo com Smart (2014), o BDD promove a escrita de especi ficações executáveis que orientam a implementação em todos os níveis de projeto. As metas do negócio de finidas por analistas e *stakeholders* do negócio são chamadas especi ficações de alto nível. Ou seja, o BDD pode ser usado como documentação de negócio do sistema.

A notação Dado que-Quando-Então é denominada de *Gherkin*. Com ferramentas apropriadas, cenários escritos desta forma podem ser automatizados e executados automaticamente.

Cada cenário *Gherkin* é composto por um conjunto de cláusulas pré-de finidas, Dado que 'contexto'...Quando 'evento'...Então 'resultado'.

- **Dado** que descreve as pré-condições para o cenário e prepara o ambiente de teste. Serve para colocar o sistema no estado conhecido antes que o usuário (ou sistema externo) comece a interagir com o sistema.
- **Quando** descreve a ação-chave que o usuário executa, ou estado de transição.
- **Então** é usado para descrever os resultados esperados. As observações devem estar relacionadas com o valor de negócios/benefício na descrição da funcionalidade.
- **E e Mas** são cláusulas adicionais usados para unir as cláusulas anteriores e proporcionar uma forma mais legível para especi ficar a funcionalidade

A seguir tem-se um exemplo de BDD:
Funcionalidade: Entrar na aplicação
Como usuário do sistema
Eu quero me logar para poder acessar o aplicativo.

Cenário: Acessar a aplicação com dados válidos
Dado que o usuário esteja na tela de login
E informa um usuário e senha válidos

Quando clicar no botão 'Entrar'
Então mostra a tela principal exibindo o nome do usuário logado "João da Silva"

De acordo com Adzic (2014), os exemplos em Gherkin de uma funcionalidade são agrupados em um arquivo de texto chamado *feature file* (arquivo de funcionalidade). O *feature file* tem uma descrição da funcionalidade, seguido por um número de cenários, ou exemplos formalizados de como a funcionalidade funciona, como no exemplo acima.

3 Metodologia

Em termos de metodologia, o trabalho apresentado neste artigo pode ser caracterizado como uma pesquisa aplicada, utilizando dados qualitativos [Marconi e Lakatos 2003].

O trabalho foi dividido nas seguintes etapas, apresentadas a seguir:

1. Criar projeto;
2. Apresentação do aplicativo a ser testado e qual parte dele será automatizada;
3. Elaborar o BDD a ser automatizado;
4. Automatizar teste;
5. Verificar o resultado; e
6. Conclusão.

Na etapa um, foi criado projeto, baixando um *template* pronto com a estrutura base. Nessa etapa será explicada qual a função de cada arquivo e pasta presente na estrutura do projeto. Além disso, será dado o passo a passo de como obter as configurações básicas para funcionar o projeto.

Já com o projeto criado, será apresentado o aplicativo escolhido, mostrando qual parte da aplicação será testada. O aplicativo utilizado nesse trabalho é nativo para o Android, com acesso ao código fonte, e ainda está em desenvolvimento. A aplicação se trata de um sistema de *check-ins* para oficiais de justiça poderem confirmar que estiveram em um determinado local.

Depois de escolhido a aplicação a ser testado, remete-se projeto de automação de testes criado para elaborar o BDD para a funcionalidade a ser testada, definindo os passos dos testes.

Assim que o BDD for elaborado, os testes são elaborados de fato, ou seja, automatiza-se o BDD escrito, e executa-se o código, para que seja possível analisar os resultados.

4 Cenário de estudo

Esta seção apresenta o desenvolvimento e a análise do cenário de estudo.

4.1 Ferramentas utilizadas

Para realizar a automação de testes desse trabalho, foram utilizadas algumas ferramentas *open source*, sendo que um breve resumo será descrito a seguir.

O Cucumber é uma ferramenta que executa testes de aceitação automatizados escritos em um estilo de desenvolvimento orientado por comportamento (BDD). Ele permite que os comportamentos de software esperados sejam especificados em um idioma lógico que os clientes possam entender. Sendo assim, o Cucumber executa de documentação de sistemas escrita em texto, voltado para empresas. O Cucumber pode ser encontrado em: https://cucumber.io/.

O Appium é um framework para a automação de testes mobile, podendo ser utilizada para aplicativos nativos, híbridos e web. Utiliza como base a API do Selenium WebDriver, ou seja, os mesmos recursos para automação web podem ser utilizados, como, por exemplo, encontrar elementos pelo id, class, name e *XPath*. Esse *framework* possui um bom suporte para o Android, além de permitir que os testes sejam escritos em diversas linguagens, tais como *Java, Ruby, JavaScript, C#, Perl e PHP. O Appium pode ser encontrado em:* http://appium.io/

O aplicativo utilizado nesse trabalho é nativo para o Android, com acesso ao código fonte. Por isso, não será necessária uma ferramenta para identificar o mapeamento dos elementos da aplicação. Porém, caso necessário, para identificar os elementos pode ser utilizada a ferramenta UI Automator Viewer, fornecida pelo próprio Android. Basicamente, essa ferramenta apresenta uma interface gráfica que permite fazer o mapeamento dos elementos do aplicativo de forma hierárquica, mostrando informações de algumas propriedades, tais como *resource-id*, class, name e text.

O GitHub é uma plataforma de hospedagem de código-fonte com controle de versão. Ele permite que programadores, utilitários ou qualquer usuário cadastrado na plataforma contribuam em projetos privados e/ou *Open Source* de qualquer lugar do mundo. Ou seja, para obter o projeto base (com a estrutura inicial), iremos utilizar/pegar do GitHub. Pode ser acesso por: https://github.com/.

4.2 Criar projeto

Nessa seção será apresentado como iniciar o projeto, sua estrutura e principais configurações iniciais.

4.3 Copiar projeto base para o computador

Para iniciar o projeto, pode ser feito o download da estrutura básica. Essa estrutura pode ser encontrada em: https://github.com/eduardofinotti/mobile-testes-interface-artigo-pos.

Acessando o terminal do computador, navegando até uma pasta de escolha do usuário, deve ser executado o seguinte código:

```
$git clone  https://github.com/eduardofinotti/mobile-testes-interface-artigo-pos
```

Assim, o projeto já está no computador do usuário. Basta acessa-lo por um editor de código que me melhor se encaixe no gosto do desenvolvedor.

Assim, a estrutura inicial do projeto é apresentada na Figura 1.

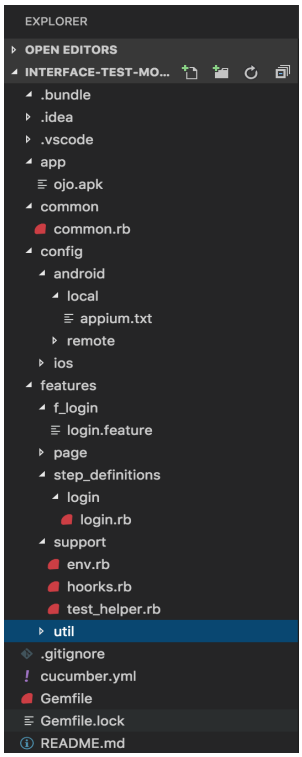

Figura 1. Estrutura inicial do projeto

Iniciando a configuração e estruturação básicas do projeto, deve ser feito as seguintes alterações:

1. Dentro do arquivo *README.md*, existe as instruções para instalação de alguns itens necessários para o projeto. Além disse deve ser instalada as dependências do projeto, executando no terminal, dentro da pasta do projeto:

   ```
   $bundle install
   ```

2. Na pasta *APP* deve ser colocada o .apk do projeto, gerada pelo programador (Figura 2):

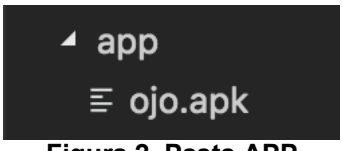

Figura 2. Pasta APP

3. No arquivo *appium.txt* (econtrado em: *config* > *android* > *local*), deve ser colcoado as informações da aplicação:
   [caps]
   platformName = "Android" => Representa em qual sistema irá rodar os testes.
   DeviceName = "0010180973" => Nome do device
   app = "app/ojo.apk" => Localização do .apk no projeto (ver item 1)

appPackage = "com.softplan.ojo" => Nome do package da aplicação
appActivity = "host.exp.exponent.LauncherActivity" => Tela principal da aplicação
newCommandTimeout = 3600 => Tempo de espera o comando antes de encerrar os testes
noReset=fals => indica que sempre inicia a aplicação com os dados limpos
[appium_lib]
wait = 20

O resultado deste arquivo é apresentado na Figura 3:

```
[caps]
platformName = "Android"
deviceName = "0010180973"
app = "app/ojo.apk"
appPackage = "com.softplan.ojo"
appActivity = "host.exp.exponent.LauncherActivity"
newCommandTimeout = 3600
noReset=false
[appium_lib]
wait = 20
debug = "true"
```

Figura 3. Arquivo de configuração appium.txt

4.4 Apresentação do aplicativo a ser testado e qual parte dele será automatizada

O aplicativo utilizado nesse trabalho e nativo para o Android, com acesso ao co digo fonte, e ainda está em desenvolvimento. A aplicação se trata de um sistema de *check-ins* para o ficiais de justiça poderem con firmar que estiveram em um determinado local.

Para acessar a aplicação é necessário fazer um login, e é para essa funcionalidade que será desenvolvido testes automatizados (Figura 4).

Figura 4. Login do sistema

Logo Após o *login* realizado, é veri ficado se realmente logou na aplicação, onde é validado do *header* da próxima pagina (Figura 5), garantindo que o nome do usuário logado é exibido corretamente.

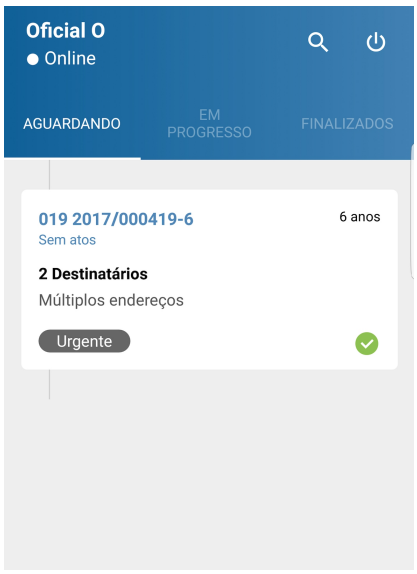

Figura 5. Tela incial da aplicação

4.5 Elaborar o BDD a ser automatizado

Já com as configurações básicas do projeto prontas (ver arquivos como *env.rb*, *hooks.rb e Gemfile*), vamos começar a descrever as funcionalidades a serem automatizadas.

A funcionalidade escolhida para ser automatizada é o *login* da aplicação. Então agora vamos descrever como essa funcionalidade deve se comportar. Como já vimos o BDD indica como o sistema deve se comportar, escrito com linguagem padrão.

Seguindo esse padrão, o BDD dessa funcionalidade, deve ser adicionado no arquivo: *features > f_login > login.feature*, e fica da seguinte forma:

login.feature

```
#language: pt

@login
Funcionalidade: Entrar na aplicação
  Como um usuário
  Eu quero me logar na aplicação
  Para poder acessar o aplicativo

  Cenário: Acessar aplicação com dados válidos
    Dado que é informado um usuário e senha válidos
    Quando clicar no botão 'Entrar'
    Então mostra a tela principal com o nome do usuário logado "Oficial O"

  Cenário: Acessar aplicação com dados inválidos
    Dado que é informado um usuário e senha inválidos
    Quando clicar no botão 'Entrar'
    Então mostra mensagem "Senha ou usuário inválido."
```

Figura 6. BDD da funcionalidade

Então os passos do primeiro cenário 1 são:

- Informar usuário e senha válidos
- Clicar no botão Entrar
- Verificar se logou corretamente.

Os passos do cenário 2 são:

- Informar usuário e senha inválidos
- Clicar no botão Entrar
- Verificar mensagem de erro ao realizar login.

4.6 Automatizar teste

Os BDDs criados são basicamente nossos testes, pois cada linha deste, gera um método, que produz

uma ação, e é dentro deste método que deve ser codificado os testes.

Para ter certeza que cada linha chama um método, pode ser executado esse BDD sem a criação dos testes de fato. Para isso, devemos configurar e iniciar o Appium.

Para configurar o Appium, o mesmo deve ser inciado, clicando em Start Server.

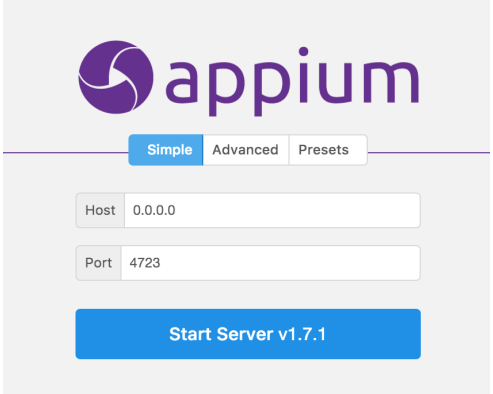

Figura 7. Tela inicial Appium

Com o Appium já iniciado, deve ser executado no terminal, dentro da pasta do projeto o seguinte comando (com o celular conectado no comutador via USB):

$cucumber PN=android OA=local -p run

Onde:

PN: informa que nosso dispositivo é Android.

OA: informa que vamos rodar os testes localmente, e não em um computador remoto.

Run: pega as informações presentes no arquivo cucumber.yml, onde dentro dele existem as tags (que diz quais BDDs queremos executar), além de informar que será gerado um relaatórios dos resultados gerados com a execução dos testes.

O resultado dessa execução é:

```
Funcionalidade: Entrar na aplicação
  Como um usuário
  Eu quero me logar na aplicação
  Para poder acessar o aplicativo

"Driver starting"
  Cenário: Acessar aplicação com dados válidos                          # features/f_login/login.feature:9
    Dado que é informado um usuário e senha válidos                     # features/f_login/login.feature:10
    Quando clicar no botão 'Entrar'                                     # features/f_login/login.feature:11
    Então mostra a tela principal com o nome do usuário logado "Oficial 0" # features/f_login/login.feature:12

"Driver starting"
  Cenário: Acessar aplicação com dados inválidos       # features/f_login/login.feature:14
    Dado que é informado um usuário e senha inválidos  # features/f_login/login.feature:15
    Quando clicar no botão 'Entrar'                    # features/f_login/login.feature:16
    Então mostra mensagem "Senha ou usuário inválido." # features/f_login/login.feature:17

2 scenarios (2 undefined)
6 steps (6 undefined)
0m57.997s

You can implement step definitions for undefined steps with these snippets:

Dado("que é informado um usuário e senha válidos") do
  pending # Write code here that turns the phrase above into concrete actions
end

Quando("clicar no botão {string}") do |string|
  pending # Write code here that turns the phrase above into concrete actions
end

Então("mostra a tela principal com o nome do usuário logado {string}") do |string|
  pending # Write code here that turns the phrase above into concrete actions
end

Dado("que é informado um usuário e senha inválidos") do
  pending # Write code here that turns the phrase above into concrete actions
end

Então("mostra mensagem {string}") do |string|
  pending # Write code here that turns the phrase above into concrete actions
end
```

Figura 8. Execução sem métodos implementados

Ou seja, o resultado é a infomação de que os metodos não estão implementados. Sendo assim dentro de *features > step_definitions > login > login.rb* vamos codificar os metodos de teste.

Para facilitar é possóvel copiar os métodos indicados no terminal, e somente codificar no lugar da linha “pending”.

Contudo, antes da codificação dos testes, devemos criar nossa *Page Object*, que é onde ficará o mapeamento das telas. Importante para reaproveitamento de código, organização e manutenção do mesmo.

Dentro de *features* > *page* > *android* > *login_screen_page.rb*, deve ser colocado a identificação dos elementos da tela, que pode ser por *id*, *name*, *css*, *xpath*, etc. Esses elementos podem ser informados pelo desenvolvedor, ou utilizar de ferramentas que fazem a extração dessas informações, como o UI Automator Viewer, fornecida pelo proprio Android. O arquivo mencionado deve ficar:

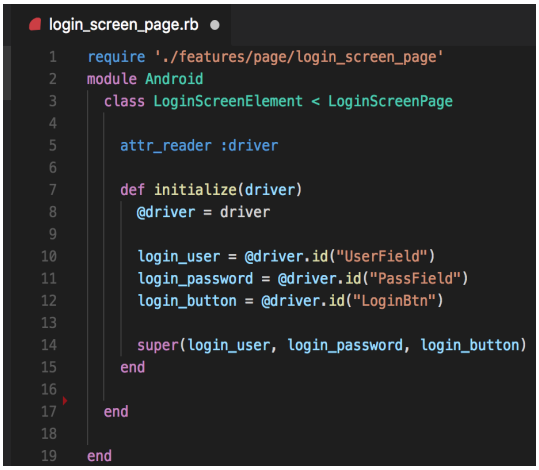

Figura 9. Mapeamento da page login

Para as funções que fazem a ação para o navegador, essas devem ser criadas em *features* > *page* > *login_screen_page.rb*, que é o caminho presente no atributo *require*, no arquivo onde está o mapeamento dos elementos da tela (figura 9). O arquivo com as ações deve ficar (figura 10):

Figura 10. Mapeamento da page login

Em seguinda, depois de mapeado as telas que terão testes automatizados, deve ser implementado os metodos gerados pelo BDD (figura 8).

Cenário: Acessar aplicação com dados válidos

Método: Dado que é informado um usuário e senha válidos

Explicação:

- Nesse método é feita a ação para preencher o usuário e senha para acessar a aplicação.
- Mas antes é necessário instânciar a página, para poder acessar seus atributos:
  - *@login_screen_object=$ENV::LoginScreenElement.new($driver)*
- Em seguida deve ser chamado de ação da página (figura 10)
  - *@login_screen_object.send_login_user('oficial')*
  - *@login_screen_object.send_login_password('132')*

Método: Quando clicar no botão 'Entrar'

Explicação*:*

- Nesse método é feito o *click* no botão Entrar
  - *login_screen_object.click_login_button*
- Nessa aplicação, depois do login, é aberta uma solicitação para dar permissão de acesso a localização do dispositivo. Pensando em reaproveitar o código para o teste de login inválido, é necessário verificar se essa solicitação está presenta na tela, e se estiver presente, aceitar:
  - elemento=$driver.find_element(:id,'com.android.packageinstaller:id/permission_allow_button').displayed?

```
if elemento == true
        $action.waitElementAndClick
                ("com.android.packageinstaller:id/permission_allow_button")
end
```

Método: Então mostra a tela principal com o nome do usuário logado "Oficial O"

Explicação:

- Nesse método deve ser validado se realmente logou na aplicação. Para isso vai ser validado se o nome do usuário logado é exibido na tela inicial, no header na aplicação:

Figura 11. Header aplicação

- Deve ser instânciada a página que representa o header da aplicação (o aquivo com o mapemento dessa tela deve ser criado)
  - *@header_screen_object = $ENV::HeaderScreenElement.new($driver)*
- Porém, antes de tudo, pode ser colocado um *sleep*, que serve para esperar, até fazer uma outra ação
  - *sleep 5*
- Depois, é só fazer a comparação, pra validar se o nome exibido na tela é o mesmo que foi especificado no BDD
  - *if @header_screen_object.getUserName != nome*
    fail("Erro de validaçao")
    end

Cenário: Acessar aplicação com dados inválidos

Método: Dado que é informado um usuário e senha válidos

Explicação:

- Nesse método é feita a ação para preencher o usuário e senha para acessar a aplicação, igualmente feito no cenário anterior, porém, passando dados inválidos.
  - *@login_screen_object=$ENV::LoginScreenElement.new($driver)*
  - *@login_screen_object.send_login_user('xxx')*
  - *@login_screen_object.send_login_password('xxx')*

Método: Quando clicar no botão 'Entrar'

Explicação:

- É o mesmo método utilizado no cenário anterior, pois como o nome do método é igual (Quando clicar no botão 'Entrar'), este pode ser utilizado em todo o projeto, independente da *feature.*

Método: Então mostra mensagem "Senha ou usuário inválido."

Explicação*:*

- Quando o usuário tenta acessar a aplcação com dados inválidos, é um *popup* é exibido, informando que os dados estão errados
- Então deve ser instânciada a página que representa o *popup* de alerta (o aquivo com o mapemento dessa tela, e chamar o metodo que recuperao texto da tela. Igualmente feito para os exemplos anteriores.
  - @login_errado = $ENV::Noti ficacaoScreenElement.new($driver)
    @mensagem = @login_errado.getMensagem
- Após isso, basta apenas comparar a mensagem da tela com a mensagem especi ficada no BDD.
  - if @mensagem != mensagem
      fail("Erro! O teste falhou!")
    end

Ao final o arquivo contendo os métodos fica da seguinte maneira:

login.rb

```ruby
Dado(/^que é informado um usuário e senha válidos$/) do
  @login_screen_object = $ENV::LoginScreenElement.new($driver)
  @login_screen_object.send_login_user('oficial')
  @login_screen_object.send_login_password('132')
end

Quando(/^clicar no botão 'Entrar'$/) do
  @login_screen_object.click_login_button

  begin
    elemento = $driver.find_element(:id, 'com.android.packageinstaller:id/permission_allow_button').displayed?

    if elemento == true
      $action.waitElementAndClick("com.android.packageinstaller:id/permission_allow_button")
    end

  rescue
      p "Não pediu permissão!"
  end
end

Então("mostra a tela principal com o nome do usuário logado {string}") do |nome|
  sleep 5

  @header_screen_object = $ENV::HeaderScreenElement.new($driver)

  if @header_screen_object.getUserName != nome
    fail("Erro de validaçao")
  end
end

Dado(/^que é informado um usuário e senha inválidos$/) do
  @login_screen_object = $ENV::LoginScreenElement.new($driver)
  @login_screen_object.send_login_user('xxx')
  @login_screen_object.send_login_password('xxx')
end

Então("mostra mensagem {string}") do |mensagem|

  @login_errado = $ENV::NotificacaoScreenElement.new($driver)
  @mensagem = @login_errado.getMensagem

  if @mensagem != mensagem
    fail("Erro! O teste falhou!")
  end

end
```

Figura 12. Arquivo login.rb

4.7 Verificar o resultado

Até este ponto, todas classes já estão implementadas, e resta executar o projeto para analisar os resultados.

Dentro da pasta do projeto, no terminal, deve ser executado o seguinte comando:

$cucumber PN=android OA=local -p run

Ao executar este comando, com o celular conectado no comutador via USB, o aplicativo começa a ser executado automaticamente na tela do aparelho, simuando um usuário utilizando o aplicativo. Ao final o resultado é o seguinte:

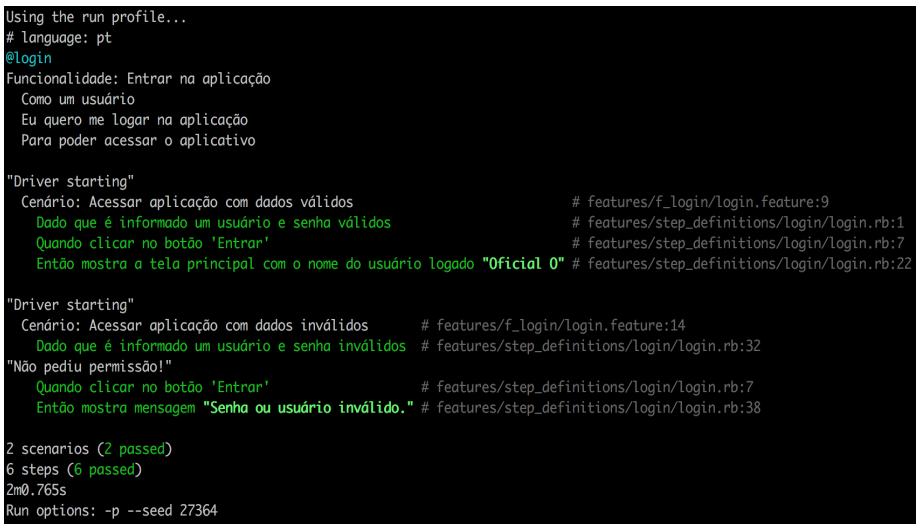

Figura 13. Resultado execução

Além disso, um arquivo chamado *report.html* é gerado, com os resultados da execução.

5 Conclusão

Com usuários cada vez mais exigentes a importa ncia da qualidade de software cresce ainda mais, na o sendo diferente com aplicativos mo veis. Com um mercado ta o exigente e com muitos concorrentes, qualquer erro apresentado para os usuários pode ser o motivo para esses desistirem de usar o seu aplicativo, ale m de ficar com uma imagem prejudicada perante a esses usuários insatisfeitos.

A necessidade de ter velocidade nesses testes tambe m e crucial para o lanc amento de novas funcionalidades e/ou correc o es de erros.

Outro problema seria testar todas as funcionalidaes em diversos aparelhos a cada versão disponibilizada. Por isso, o teste automatizado se torna um caminho importante, pois seria possível reexecutar várias vezes o mesmos testes, garantindo que o que funcionava antes, não deixou de funcionar.

Levando em considerac a o esses problemas, este trabalho buscou mostrar como iniciar um projeto testes automatizados para dispositivos móveis, assim contrinuíndo com um assunto que ainda é pouco desenvolvido em nosso país.

Como trabalhos futuros, proõem-se realizar uma comparação de tempo na execução de testes automatizados em relação a testes manuais, exibindo quanto tempo levou a reexecução dos testes de uma maneira e de outra.

6 Referências

NBR ISO 8402 (1994). Gestão da Qualidade e garantia da qualidade. Acessado em https://docslide.com.br/documents/nbr-iso-8402-1994-gestao-da-qualidade-e-garantia-da-qualidade-terminologia.html.

Starov, O., Vilkomir, S., Gorbenko A. e Kharchenko, V. (2015) *"Testing-as-a-Service for Mobile Application: State-of-the-art Survey" in Dependability Problems of Complex Information Systems, p. 55-71.*

HETZEL, Willian. *Guia completo ao teste de software. Rio de Janeiro: Campus, 1987.*

Marconi, Marina de Andrade e Lakatos, Eva Maria (2003), Fundamentos da Metodologia Científica, Atlas, 5a edição.

Dantas, V. L. L. (2009), *Requisitos para Testes de Aplicações Móveis, Universidade Federal do Ceará. Dissertação de Mestrado em Ciência da Computação, 132 p.*

Pressman, Roger S. (2011), *Engenharia de software: Uma abordagem profissional, AMGH, 7ª*

edição.

Bartié, Alexandre (2002), Garantia da qualidade de software: adquirindo maturidade organizacional, Campus

Molinari, Leonardo (2003), Testes de Software: Produzindo Sistemas Melhores e Mais Confiáveis, Érica.

Sommerville, Ian (2011), Engenharia de software, Pearson Prentice Hall, 9a edição.

IEEE. IEEE Computer Dictionary - Compilation of IEEE Standard Computer Glossaries, 610-1990, 1990.

Matts, C.; Adzic, G. "Feature injection: three steps to success". Capturado em: http://www.infoq.com/articles/feature-injection-success, 2014.

McDonald, K. "Beyond Requirements: Analysis with an Agile Mindset". Boston, USA: Pearson Education, 2015

Smart, J. "BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle". New York, USA: Manning Publications, 2014.

North, D. "Introducing bdd". Capturado em: http://dannorth.net/introducing-bdd/, 2006.

Adzic, G. "Specification by Example: How Successful Teams Deliver the Right Software". Connecticut, USA: Manning Publications, 2011.

Smart, J. "BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle". New York, USA: Manning Publications, 2014.

Gohil, K.; Alapati, N.; Joglekar, S. "Towards behavior driven operations (bdops)". In: International Conference on Advances in Recent Technologies in Communication and Computing, 2011, pp. 262 -264.

Evans, E. "Domain-Driven Design: Tacking Complexity In the Heart of Software". Boston, MA, USA: Addison-Wesley, 2003.

Knott, D. (2015), Hands-On Mobile App Testing: A Guide For Mobile Testers and Anyone Involved in the Mobile App Business, Addison-Wesley Professional.

uTest (2015), “The Essencial Guide to Mobile App Testing” in Applause. Disponível em http://go.applause.com/rs/539-CKP-074/images/The-Essential-Guide-to-Mobile- App-Testing.pdf. Maio.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software*. [S.l.]: Elsevier, 2007.