Análise de portabilidade dos testes automatizados entre as plataformas android e iOS: um relato de experiência.

Portability Analysis of Automated Tests Between Android and iOS Platforms: an experience report

Felipe Bernard de Oliveira Soares Diniz[1], **Sidney Nogueira**[1]

[1] [1]Departamento de Computação, Universidade Federal Rural de Pernambuco (UFRPE)
Rua Dom Manoel de Medeiros, s/n, Dois Irmãos, Recife - PE, Brasil

felipe.bernardd@ufrpe.br, sidney.nogueira@ufrpe.br

Abstract. *Multiplatform testability is an essential and challenging aspect of software development, especially due to the diversity of devices and platforms available on the market. This work evaluates the portability of automated test scripts using Robot Framework and Appium on Android and iOS platforms. The methodology involved executing a set of tests on both platforms, mapping, and analyzing the observed discrepancies. The results indicate that, although Robot Framework and Appium are widely used and effective tools, there are still significant challenges in fully porting the scripts, requiring specific adaptations for each platform. The research documents these limitations and provides recommendations to improve the efficiency of automated tests and the quality of mobile software projects.*

Keywords. *Automated Testing; Test Script Portability; Robot Framework; Appium; Android; iOS; Cross-platform Development*

Resumo. *Testabilidade multiplataforma é uma característica essencial e desafiadora no desenvolvimento de software, especialmente devido à diversidade de dispositivos e plataformas disponíveis no mercado. Este trabalho avalia a portabilidade de scripts de teste automatizados utilizando Robot Framework e Appium nas plataformas Android e iOS. A metodologia incluiu a execução de um conjunto de testes em ambas as plataformas, mapeando e analisando as divergências observadas. Os resultados indicam que, embora Robot Framework e Appium sejam ferramentas amplamente utilizadas e eficazes, ainda existem desafios significativos na portabilidade completa dos scripts, exigindo adaptações específicas para cada plataforma. A pesquisa documenta essas limitações e oferece recomendações para melhorar a eficiência dos testes automáticos e a qualidade em projetos de software móvel.*

Palavras-Chave. *Teste Automatizado; Portabilidade de Scripts de Teste; Robot Framework; Appium; Android; iOS; Desenvolvimento Multiplataforma*

1. Introdução

Os sistemas de software fazem parte da vida e rotina de bilhões de pessoas pelo mundo: processos que antes eram completamente analógicos e feitos de forma manual, hoje estão digitalizados e são realizados com o suporte de sistemas e aplicações de software. Dado o grau de dependência atual com soluções de software, falhas podem causar diversos prejuízos. Quando softwares não funcionam corretamente, muitos problemas podem surgir, incluindo perda de dinheiro, tempo ou reputação comercial e, em casos extremos, até ferimentos ou morte [IST 2024].

Para que a ocorrência de falhas seja minimizada, é necessário fazer uso de estratégias no desenvolvimento do software, nas quais se inclui o uso de testes. Na criação de softwares e sistemas, um erro pode surgir nas especificações do projeto, do design, na implementação do sistema, entre outros artefatos. Um erro que está presente no projeto pode levar a um defeito, que pode se manifestar para o usuário final como uma falha (mais conhecida como Bug).

O processo de testagem é importante para detectar problemas em todas as fases do desenvolvimento de um projeto de software, contribuindo para a qualidade do software. Quando a engenharia de software ainda dava os seus primeiros passos, muitas empresas tendiam a ver o processo de teste de software como um custo desnecessário, uma vez que aumentava o tempo de desenvolvimento e atrasava a chegada do produto até as mãos do cliente. Hoje é notório que dedicar esforços para testar um software é um investimento. De acordo com [Myers et al. 1979] o custo em correção de defeitos achados no software cresce exponencialmente 10 vezes para cada estágio que o projeto avança.

O esforço de testar pode ser maior do que o esforço para codificar o software, especialmente quando os testes são realizados manualmente [Kushwaha and Misra 2008]. Estudos indicam que mais de 50% do custo de desenvolvimento de software é destinado ao teste, sendo essa porcentagem ainda maior para softwares críticos [Kushwaha and Misra 2008]. Uma prática eficiente para reduzir o tempo total dos testes manuais é a automação de testes, que pode ser aplicada em diferentes camadas da aplicação. Nas camadas mais baixas, a automação foca em testar métodos e componentes isoladamente, enquanto nas camadas superiores o foco é testar funcionalidades completas. À medida que se sobe pelas camadas de abstração do software, os testes tornam-se mais complexos e mais demorados [IST 2024].

Desenvolver e testar uma aplicação em múltiplas plataformas requer um esforço considerável, pois é necessário adaptar o código da aplicação e dos scripts de teste para cada plataforma. O esforço para criar e manter esses scripts de teste pode ser multiplicado pelo número de plataformas para as quais a aplicação precisa ser verificada. Para minimizar esse esforço, ferramentas e frameworks de automação têm sido propostos, buscando reduzir o trabalho necessário para criar e manter tanto o código da aplicação quanto os scripts de teste.

Uma das tecnologias mais populares para desenvolvimento multiplataforma é o React, que permite que o mesmo código de aplicação seja executado em diferentes plataformas com poucas modificações para se adaptar às particularidades de cada uma. Seguindo essa filosofia, tecnologias e ferramentas como as propostas por

[Behrang and Orso 2018], [Menegassi and Endo 2020] e [Qin et al. 2019] foram desenvolvidas para criar e manter scripts de teste reutilizáveis entre diferentes plataformas, diminuindo o esforço de adaptação. Dentre as ferramentas mais utilizadas para essa finalidade está a combinação Robot Framework e Appium, que permite a reutilização de um mesmo script de teste em diferentes plataformas, sem a necessidade de criar versões específicas para cada uma. A comunidade por trás dessas ferramentas é bastante ativa e extensa: o Robot Framework, por exemplo, conta com mais de 9.000 estrelas e mais de 10.000 utilizadores no GitHub [RobotFrameworkFoundation 2024], enquanto o Appium tem mais de 18.000 estrelas e mais de 300 contribuidores [AppiumProject 2024]. Além disso, ambas as ferramentas possuem fóruns de discussão, como o Appium Discuss e o Robot Framework Forum, que são muito movimentados e recebem contribuições regulares da comunidade global de desenvolvedores.

Apesar do crescente uso de ferramentas de automação, testes implementados para múltiplas plataformas ainda podem se comportar de forma diferente em cada uma delas [do Nascimento Mendes and Dias-Neto 2016]. Isso acontece porque a testabilidade multiplataforma impõe requisitos adicionais ao design da aplicação. Enquanto componentes simples como botões têm um comportamento semelhante entre Android e iOS, componentes mais complexos, como listas de seleção e seletores de data e hora, podem apresentar diferenças notáveis [Bordi 2018].

No Robot Framework/Appium, os scripts de teste são escritos usando *keywords*, que são descrições em alto nível dos procedimentos dos testes. Embora a ideia seja que as *keywords* sejam adaptadas automaticamente às especificidades de cada plataforma, em muitos casos, isso não ocorre perfeitamente. Como observado em fóruns de discussão (GitHub Appium - Issues, StackOverflow - Questions), certos testes podem falhar em uma plataforma enquanto passam em outra, devido problemas como a forma que as plataformas renderizam a árvore de elementos. Isso resulta em falsos negativos, onde o teste falha mesmo quando a funcionalidade está correta.

Apesar da popularidade de Robot/Appium, poucos estudos acadêmicos têm investigado esses problemas. Este trabalho busca preencher essa lacuna ao expor as limitações de portabilidade das *keywords* de Robot/Appium através de um estudo de caso. O projeto de testes proposto inclui scripts automatizados para Android e iOS, desenvolvidos para uma aplicação React, com o objetivo de identificar as divergências nos comportamentos dos testes entre as plataformas e entender as causas dessas falhas.

Este estudo visa responder a pergunta ”Quais as diferenças que se podem observar ao executar um mesmo conjunto de testes automatizados usando Robot framework e Appium nas plataformas Android e iOS?”e o trabalho tem como objetivo avaliar a portabilidade de scripts automáticos desenvolvidos com Robot/Appium para as plataformas Android/iOS. A escolha dessas ferramentas se justifica por serem amplamente adotadas no mercado, de código aberto, e gratuitas, além de oferecerem suporte multiplataforma, o que as torna ideais para a automação de testes em diferentes sistemas operacionais. Além disso, as ferramentas se destacam pela flexibilidade e pela vasta comunidade de usuários e desenvolvedores, facilitando a integração de novas funcionalidades e a resolução de problemas técnicos.

O estudo irá explorar as principais diferenças observadas na execução de um mesmo conjunto de testes automatizados nas plataformas Android e iOS, com o objetivo de identificar os componentes móveis que causam as maiores divergências, bem como entender o nível de limitação dessas tecnologias no contexto dos testes de dispositivos móveis.

Após execução dos testes e análise dos resultados, uma das principais inferências deste estudo é que a implementação de testes automatizados não deve ser feita considerando apenas uma plataforma, deve-se realizar uma verificação básica em ambas as plataformas durante a escrita dos scripts de teste para aumentar a chance de uma execução efetiva e precisa. Este trabalho também destaca que algumas keywords do Robot Framework e do Appium não funcionam conforme o esperado entre as plataformas. Com base nessas descobertas, o trabalho sugere que a comunidade de desenvolvedores dessas ferramentas precisa continuar aprimorando suas funcionalidades, incluindo a correção de comportamentos inconsistentes. A experiência deste estudo serve como referência para medir a evolução e as melhorias nas versões futuras do Robot/Appium.

A estrutura deste trabalho segue da seguinte forma: A Seção 2 introduz os conceitos e tecnologias subjacentes utilizados ao longo do trabalho. A Seção 3 explora os trabalhos relacionados no campo da automação de testes de software, revisando pesquisas e estudos prévios que abordam a portabilidade de testes automatizados. Na Seção 4, detalhamos a metodologia empregada para a realização dos experimentos e a coleta de dados, incluindo a configuração do ambiente, criação dos scripts de teste, e execução dos testes automatizados. A Seção 5 descreve como os testes automatizados foram executados, organização dos resultados e monitoramento das execuções. A Seção 6 apresenta os resultados dos testes realizados, discutindo as descobertas e insights sobre a portabilidade dos scripts entre as plataformas Android e iOS. Finalmente, na Seção 7, resumimos os principais pontos discutidos e os resultados alcançados.

2. Teste automático multiplataforma

Esta seção apresenta uma visão geral dos conceitos necessários para entendimento deste trabalho como Robot Framework, Appium e das outras tecnologias necessárias para a execução dos testes automatizados.

2.1. Automação de testes

A automação de testes é um processo essencial que oferece várias vantagens, incluindo a redução do tempo de execução dos testes, aumento da cobertura dos testes e a detecção precoce de defeitos. Para entender os casos de teste, alguns conceitos precisam ser abordados, como as partes que o formam: a descrição do teste, que detalha o objetivo e as condições iniciais; o corpo do teste, onde são definidos os passos a serem seguidos; e os vereditos, que indicam se o teste foi bem-sucedido ou falhou. Além disso, uma asserção é uma condição que deve ser verdadeira para que o teste passe, sendo essencial para validar o comportamento esperado do software. Com esses conceitos em mente, a automação de testes, utilizando ferramentas como Appium e Robot Framework, pode ser configurada para verificar as funcionalidades em aplicativos móveis.

2.2. Tecnologias Utilizadas

Este estudo faz uso de diversas tecnologias e ferramentas para automatizar testes em diferentes plataformas móveis. A seguir, são apresentadas as principais tecnologias utilizadas.

Python é a base para a instalação e operação de pacotes essenciais, como o Robot Framework e bibliotecas adicionais necessárias para a automação com Appium, devido à sua simplicidade e extensibilidade.

Node.js é utilizado para a execução de código JavaScript no servidor, sendo essencial para a instalação e operação de pacotes distribuídos via npm, como o Appium. Ferramentas como o appium-doctor também são usadas para verificar a configuração do ambiente de automação.

Robot Framework é um framework de automação extensível baseado em Python, utilizado para testes de aceitação e desenvolvimento orientado a testes de aceitação (ATDD), desenvolvimento orientado a comportamento (BDD) e automação de processos robóticos (RPA). Este framework permite a criação de testes automatizados em alto nível, facilitando a manutenção e compreensão dos testes. O Robot Framework organiza os dados de teste em seções como Settings, Variables, Keywords, e Test Cases, conforme mostrado na Tabela 1 e na Figura 1, onde a seção Settings começa na linha 1, a seção Variables na linha 6, a seção Keywords na linha 12, e a seção Test Cases começa na linha 28.

Seção	**Usado para**
Settings	Importar bibliotecas, definir metadados para suítes e casos de teste.
Variables	Definir variáveis usadas em outros locais nos dados de teste.
Keywords	Criar keywords do usuário a partir de keywords de nível inferior existentes.
Test Cases	Criar casos de teste a partir de palavras-chave disponíveis.

Tabela 1. Seções usadas no Robot Framework

Para escrever código com Robot é preciso utilizar *keywords*, um exemplo de código feito com Robot Framework pode ser observado na Figura 1. Na linha 13, está definido o nome da *keyword* criada pelo usuário, enquanto a linha 14 contém os argumentos que ela recebe. Já nas linhas 15 a 18, estão outras *keywords*, que podem ser importadas de bibliotecas ou definidas pelo próprio usuário. Nessas mesmas linhas, é possível observar o uso de variáveis que são passadas como argumentos para essas *keywords*, facilitando a reutilização e parametrização do código.

Appium é uma ferramenta de automação de testes de código aberto que permite a automação de aplicativos móveis em diversas plataformas, como Android e iOS. É amplamente utilizada para testar a interface de usuário (UI) em aplicações nativas, híbridas e web, utilizando os padrões do WebDriver [App 2024]. Com suporte multiplataforma, o Appium permite a reutilização de scripts de teste entre diferentes sistemas operacionais móveis.

AppiumLibrary é uma biblioteca que integra o Appium com o Robot Framework, permitindo a automação de testes em dispositivos móveis Android e iOS. Ela oferece

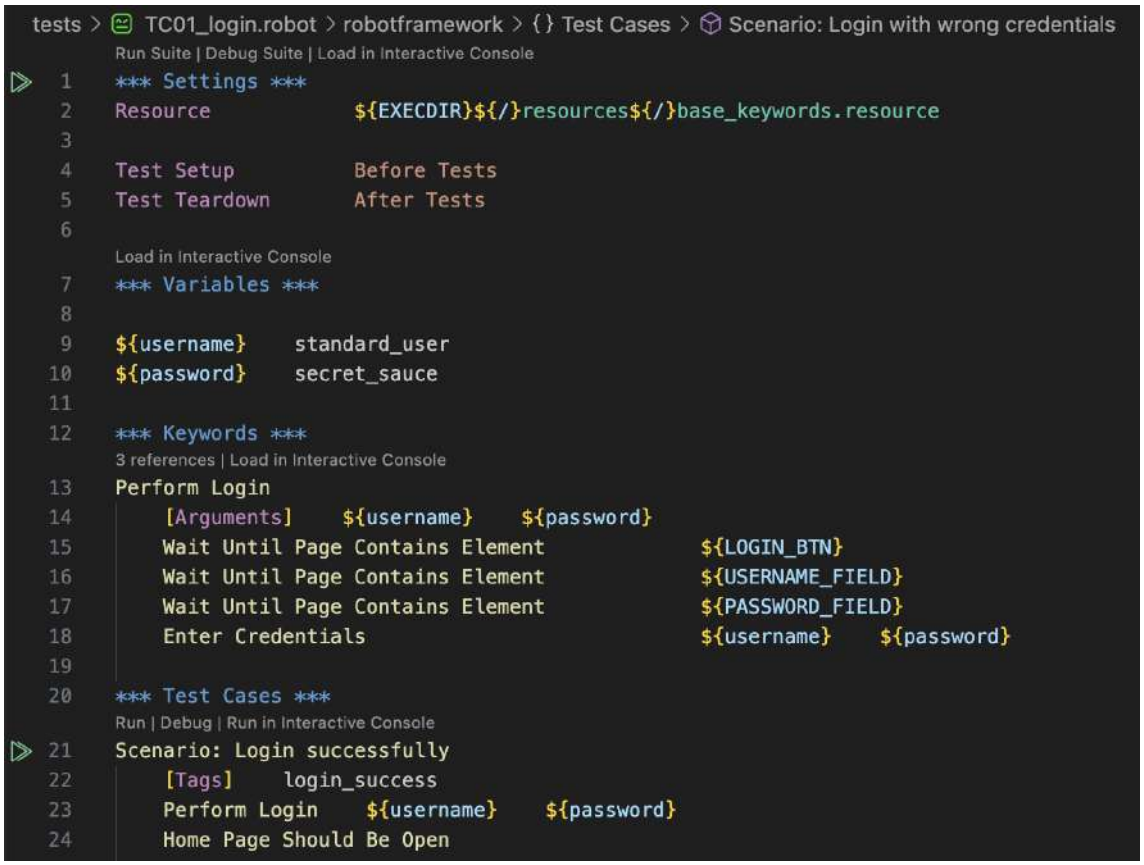

Figura 1. Seções do robot framework. Elaborado pelo autor (2024).

um conjunto de *keywords* para interagir com a interface de usuário (UI) de aplicativos móveis, facilitando a criação de scripts de teste automatizados. Embora muitas *keywords* da AppiumLibrary funcionem em ambas as plataformas, Android e iOS, existem algumas que são específicas para uma plataforma devido a diferenças nas APIs e componentes de UI de cada sistema operacional [Bolsu 2024].

PowerShell é utilizado como uma solução de automação de tarefas multiplataforma, suportando Windows, Linux e macOS. **Pipenv** é empregado para gerenciar ambientes virtuais em Python, garantindo a consistência e isolando as dependências necessárias. Finalmente, o **Visual Studio Code** serve como o editor de código-fonte, permitindo a criação de um ambiente de desenvolvimento versátil e voltado para várias linguagens de programação.

3. Trabalhos Relacionados

A automação de testes em aplicativos móveis tem sido um tema de crescente interesse na literatura. Um dos primeiros trabalhos a investigar esse campo foi o de [Salonen 2012], que explorou a importância dos testes de portabilidade automáticos, focando na criação de ambientes de teste para múltiplas plataformas, especialmente sistemas desktop e web. A abordagem de Salonen difere da do presente estudo, que se concentra nos desafios específicos da automação de testes em dispositivos móveis, particularmente em Android

e iOS.

Mais adiante, [El-Kassas et al. 2016] estudou a conversão de código em desenvolvimento móvel multiplataforma, apresentando abordagens aprimoradas para facilitar esse processo. O estudo destacou a importância de ferramentas que auxiliem na migração de código, mas também apontou limitações, especialmente em relação ao suporte a bibliotecas e à manutenção da performance nativa dos aplicativos.

Na sequência, a análise comparativa realizada por [Anjum et al. 2017] avaliou diversas ferramentas de automação de testes móveis, considerando fatores de qualidade essenciais, como usabilidade, robustez e compatibilidade de plataforma. O estudo concluiu que, apesar de ferramentas como Appium e Robotium serem eficazes para testar funcionalidades e interfaces de usuário, ainda havia uma escassez de soluções que abordassem extensibilidade, manutenibilidade e escalabilidade de forma abrangente.

Em 2018, [Bordi 2018] explorou o uso do Appium para automação de testes, destacando suas capacidades e limitações em ambientes multiplataforma. Apesar do avanço proporcionado por esses trabalhos, poucos investigaram a integração do Appium com outras tecnologias, como o Robot Framework, o que confere ao presente estudo seu caráter inovador.

Pesquisas mais recentes, como a de [Mahmoud et al. 2023], introduziram um modelo baseado em compiladores para a conversão de código de serviços web em aplicativos móveis, evidenciando ainda mais a importância de ferramentas que suportem a migração de código. Já [Yu et al. 2023] discutiu os desafios na geração e migração de scripts de teste utilizando modelos de linguagem natural, destacando a potencial integração dessas abordagens com ferramentas de transcompilação. Esse trabalho complementa os achados anteriores ao integrar o Robot Framework, proporcionando uma análise mais abrangente das limitações e capacidades dessa combinação de ferramentas.

A evolução dos estudos, desde [Salonen 2012] até pesquisas mais recentes como [Yu et al. 2023], mostra o interesse contínuo na portabilidade e eficiência dos testes automatizados em ambientes multiplataforma. No entanto, embora trabalhos anteriores como [El-Kassas et al. 2016], [Anjum et al. 2017], [Bordi 2018], [Mahmoud et al. 2023] e [Yu et al. 2023] abordem tecnologias e frameworks similares, nenhum deles explora os desafios específicos de portabilidade entre Android e iOS com foco nas tecnologias Robot Framework e Appium, como ilustrado pela tabela 2. Este estudo contribui com um ineditismo ao examinar as limitações e adaptações necessárias para garantir a eficácia dos testes nessas plataformas, integrando múltiplas ferramentas para automação em aplicativos móveis.

Estudo	**Portabilidade Multiplataforma**	**Foco em Testes Automatizados**	**Uso de Appium**	**Uso de Robot Framework**
Salonen (2012)	X			
Wafaa (2016)	X			
Anjum et al. (2017)	X	X	X	
Bordi (2018)	X	X	X	
Mahmoud et al. (2023)	X			
Este trabalho	**X**	**X**	**X**	**X**

Tabela 2. Comparação dos Estudos sobre Portabilidade e Testes Automatizados

4. Planejamento

Descrevemos as etapas que foram seguidas neste trabalho para responder a pergunta de pesquisa.

Para ilustrar o processo executado na pesquisa foi desenvolvido o fluxograma da Figura 2.

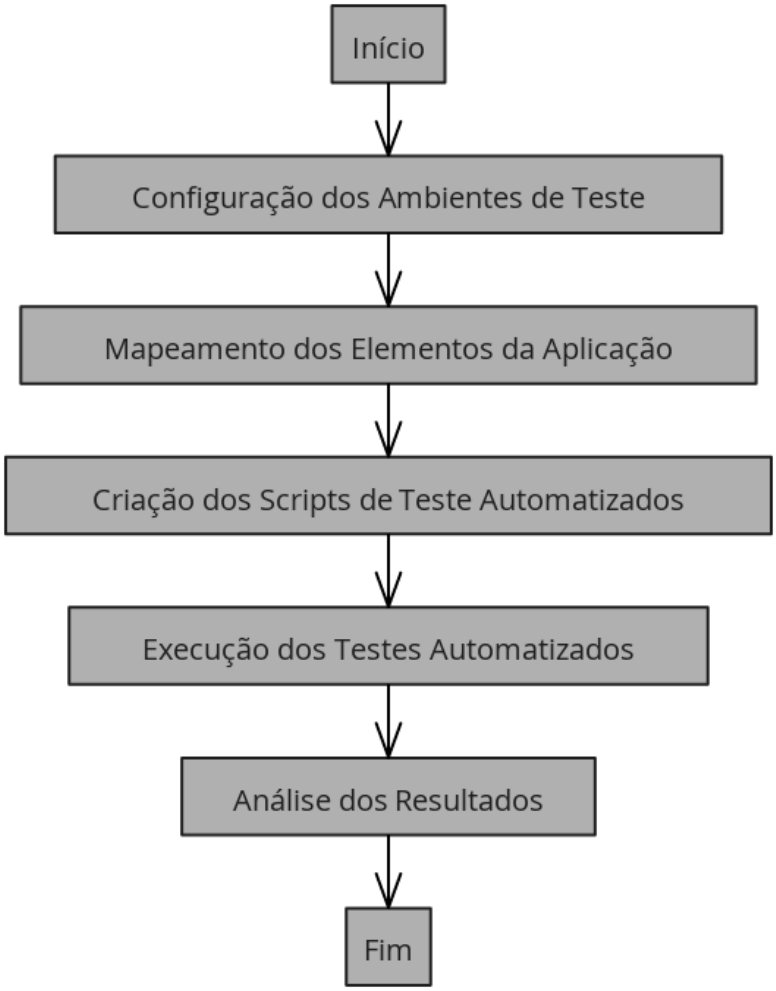

Figura 2. Processo da pesquisa em formato de fluxograma

4.1. Seleção da Aplicação para Testes

Foi escolhida a aplicação Swag Labs Mobile, por funcionar em dispositivos móveis e por possuir código aberto, tanto o código quando os binários da aplicação podem ser baixados no repositório da aplicação[1]. Entre os motivos para a escolha, podemos citar a popularidade da aplicação e a variedade de elementos de interface que ela apresenta é ideal para um estudo de portabilidade de testes.

[1] `https://www.github.com.saucelabs/sample-app-mobile`

4.2. Configuração dos Ambientes de Teste

Para realizar os testes, foi necessário configurar ambientes que simulassem as plataformas Android e iOS. Utilizamos um sistema macOS, pois ele suporta o desenvolvimento e a simulação de ambas as plataformas de forma nativa. A preparação do ambiente envolveu a instalação das ferramentas necessárias e a configuração em cada uma delas, este processo foi detalhado e encontra-se no repositório criado no Github do autor[2]. Ao final da configuração do ambiente e das ferramentas, foi verificado se tudo ocorreu bem utilizando a ferramenta Appium Doctor, para isso, foi utilizado o comando `appium-doctor --ios` e o comando `appium-doctor --android`, para verificar a eficácia das instalações para dispositivos iOS e Android, respectivamente.

5. Execução

5.1. Mapeamento dos Elementos da Aplicação

Os elementos presentes na aplicação Swag Labs foram mapeados para serem utilizados nos testes automatizados. Esse mapeamento envolveu a identificação de IDs, classes e outros atributos dos componentes da interface do usuário, utilizando ferramentas como o Appium Inspector. durante o mapeamento, foi possível identificar componentes que apresentaram diferenças claras em seus atributos entre as plataformas Android e iOS, como exemplificado nas Figura 3 para iOS e 4 para Android, onde é possível observar do lado esquerdo das figuras o elemento selecionado e no lado esquerdo uma lista de atributos desse elemento. Essas discrepâncias foram fundamentais para a escrita dos testes, uma vez que guiariam a identificação e documentação das falhas ocorridas durante a execução dos testes automatizados. O mapeamento incluiu a análise dos seguintes aspectos: Identificação dos identificadores únicos (IDs) utilizados para referenciar os componentes da interface do usuário. Estes IDs permitem a interação automatizada, possibilitando que os scripts de teste localizem e interajam com os elementos corretos; E Análise dos atributos específicos de cada plataforma que podem influenciar o comportamento dos testes. Por exemplo, um botão no Android pode ter um conjunto diferente de atributos em comparação com o mesmo botão no iOS. Essas diferenças foram documentadas para garantir que os scripts de teste possam lidar com essas variações de forma eficaz.

5.2. Criação dos Scripts de Teste Automatizados

Os scripts de teste foram desenvolvidos utilizando o Robot Framework em conjunto com o Appium. O processo de criação dos scripts incluiu:

Arquitetura do Projeto de Testes Automatizados:

A arquitetura do projeto de testes foi definida para garantir modularidade e reutilização. O arquivo *requirements.txt* lista as dependências necessárias, enquanto o diretório *resources* contém as *keywords* e variáveis usadas nos testes. O script *run_tests.ps1* automatiza a execução dos testes, e o diretório *tests* armazena os casos de teste estruturados no formato do *Robot Framework*. Essa organização facilita a manutenção e expansão do projeto.

Script para Paralelização da Execução dos Testes

[2] `https://www.github.com/bern4rd/qa-swagLabs`

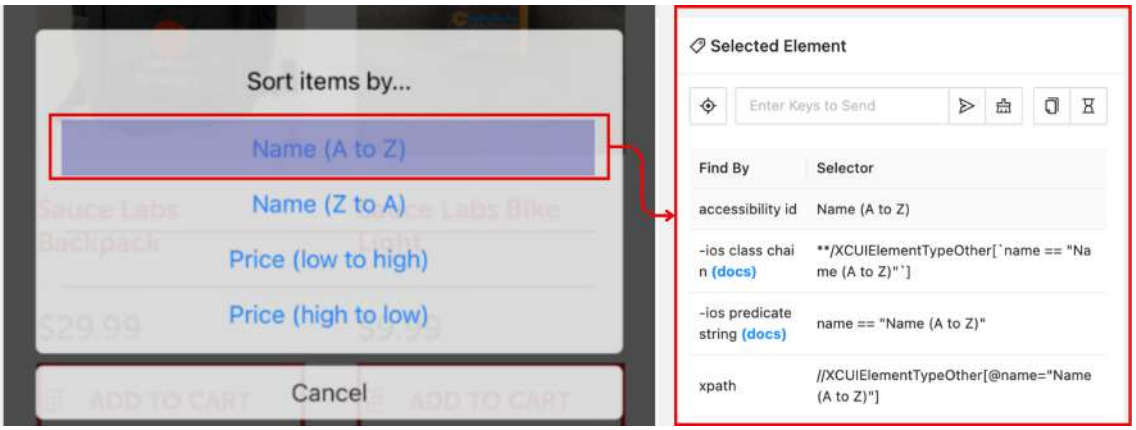

Figura 3. Appium Inspector - iOS

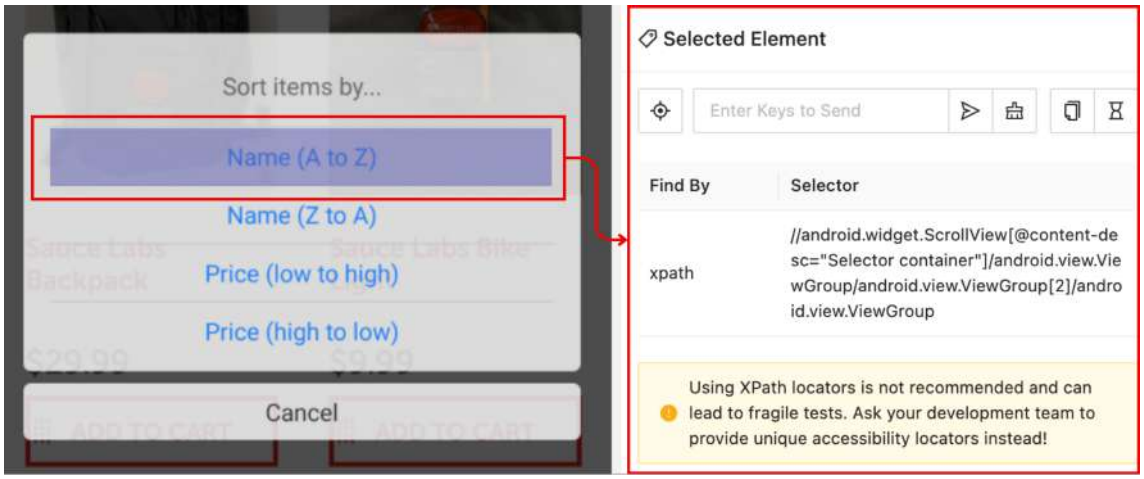

Figura 4. Appium Inspector - Android

Foi criado e utilizado um código para a paralelização da execução dos testes automatizados em ambientes Android e iOS, encontra-se também no repositório do github supracitado. O código lê os caminhos dos testes a partir de um arquivo de texto, executa os testes em paralelo para ambas as plataformas, seu uso também permitiu visualizar a execução sincronizada das execuções, o que facilitou a identificação de gargalos durante a execução de maneira visual.

Definição e escrita dos Casos de Teste

Nessa etapa, aconteceu a seleção de funcionalidades chave da aplicação para serem testadas a fim de utilizar os elementos que mapeamos como possíveis causadores de falhas. As funcionalidades escolhidas contemplaram o login, logout e a aplicação de filtros de pesquisa dos produtos.

Para a escrita dos casos de teste, foi escolhida a abordagem *Gherkin*, como demonstrado na Figura 5, onde a primeira linha da imagem mostra o início da seção de testes, a segunda linha mostra o nome do cenário de teste, a terceira a *tag* do cenário de teste e as linhas seguintes as *keywords* utilizadas no teste em questão. O uso de *Gherkin*

se deu pois permite a definição de testes em uma linguagem de alto nível, legível tanto por desenvolvedores quanto por stakeholders não técnicos. Segundo [Islam 2024], o uso de Gherkin em Behavior Driven Development (BDD) promove uma melhor comunicação entre todas as partes envolvidas no desenvolvimento, facilitando a criação de uma visão compartilhada do projeto. Essa característica é particularmente importante para garantir que os testes automatizados reflitam corretamente os requisitos e expectativas do usuário final, além de permitir uma manutenção mais fácil e ágil dos scripts de teste.

```
*** Test Cases ***
Run | Debug | Run in Interactive Console
Scenario: Login successfully
    [Tags]    login_success
    Given login page is open
    When user perform login    ${username}    ${password}
    Then the home page is open
```

Figura 5. Teste da funcionalidade login escrito em Gherkin

O uso de Gherkin possibilita a escrita de cenários de teste de forma simples e estruturada, seguindo o formato ”Dado-Quando-Então”. Este formato facilita a compreensão e a colaboração entre equipes de desenvolvimento e negócios, garantindo que todos os envolvidos tenham uma visão clara dos testes automatizados. Essa abordagem foi escolhida para assegurar que os testes fossem não apenas funcionais, mas também fáceis de entender, dado o contexto do trabalho, e modificar conforme necessário.

Como esta pesquisa é independente do desenvolvimento da aplicação utilizada para testes, a fase de design das regras de negócio não aconteceu e por consequência não houve a escrita dos requisitos funcionais, dos quais normalmente derivam os casos de uso e, subsequentemente, os casos de teste [Leffingwell and Widrig 2003]. A impossibilidade de acessar os requisitos fez com que fosse necessária a simulação da escrita dos mesmos para as funcionalidades de interesse do trabalho, especificamente **login, logout e aplicação de filtros na listagem dos produtos**.

Uma simulação dos requisitos funcionais e os casos de uso para as funcionalidades de login, logout e aplicação de filtros de ordenação nos produtos são descritos a seguir:

Requisitos Funcionais para Login

- **RF01:** O sistema deve permitir que o usuário insira um nome de usuário.
- **RF02:** O sistema deve permitir que o usuário insira uma senha.
- **RF03:** O sistema deve possuir um botão ”Login”para o envio das credenciais.
- **RF04:** O sistema deve validar as credenciais do usuário.
- **RF05:** O sistema deve exibir uma mensagem de erro caso as credenciais sejam inválidas.
- **RF06:** O sistema deve redirecionar o usuário para a tela principal após um login bem-sucedido.

Requisitos Funcionais para Logout

- **RF07:** O sistema deve permitir que o usuário acesse o menu de navegação a partir da tela principal.
- **RF08:** O sistema deve fornecer uma opção de ”Logout”no menu de navegação.
- **RF09:** O sistema deve encerrar a sessão do usuário quando a opção ”Logout”for selecionada.
- **RF10:** O sistema deve redirecionar o usuário para a tela de login após o logout bem-sucedido.

Requisitos Funcionais para Aplicação de Filtros de Ordenação

- **RF11:** O sistema deve permitir que o usuário acesse a lista de produtos.
- **RF12:** O sistema deve fornecer um ícone ou botão para acessar as opções de filtros de ordenação.
- **RF13:** O sistema deve exibir as opções de filtros de ordenação ao usuário.
- **RF14:** O sistema deve permitir que o usuário selecione uma opção de filtro de ordenação.
- **RF15:** O sistema deve aplicar o filtro de ordenação selecionado e atualizar a lista de produtos.

Caso de Uso 1: Login
Descrição: Permitir que o usuário realize login na aplicação utilizando suas credenciais.

1. O usuário abre o aplicativo.
2. O usuário é direcionado para a tela de login.
3. O usuário insere o nome de usuário no campo apropriado.
4. O usuário insere a senha no campo apropriado.
5. O usuário clica no botão ”Login”.
6. O sistema redireciona o usuário para a tela principal da aplicação.

Caso de Uso 2: Logout
Descrição: Permitir que o usuário realize logout da aplicação, encerrando a sessão atual.

1. O usuário navega até a tela principal da aplicação.
2. O usuário acessa o menu de navegação.
3. O usuário clica na opção ”Logout”.
4. O sistema encerra a sessão do usuário.
5. O sistema redireciona o usuário para a tela de login.

Caso de Uso 3: Aplicação de Filtros de Ordenação
Descrição: Permitir que o usuário aplique filtros de ordenação aos produtos listados na aplicação.

1. O usuário navega até a lista de produtos.
2. O usuário clica no ícone ou botão de filtros de ordenação.
3. O sistema exibe as opções de filtros de ordenação.
4. O usuário seleciona uma opção de filtro (preço ascendente, preço descendente, nome A-Z, nome Z-A).

5. O sistema aplica o filtro de ordenação selecionado e atualiza a lista de produtos.

Após a etapa em que foram simuladas a modelagem de requisitos e a criação dos casos de uso, a criação dos casos de teste foi realizada utilizando a linguagem Gherkin. Os casos de teste foram escritos conforme os cenários identificados nas etapas anteriores. Abaixo estão os exemplos dos cenários de teste criados:

CT01 - Login com sucesso

```
Scenario: Login successfully
    [Tags]    login_success
    Given login page is open
    When user perform login    ${username}    ${password}
    Then the home page is open
```

CT02 - Logout com sucesso

```
Scenario: Logout successfully
    [Tags]    logout_success
    [Setup]   user should be logged in
    Given the home page is open
    When user perform logout
    Then login page should be open
```

CT03 - Filtrar produtos por preço (ascendente)

```
Scenario: Filter products by price (low to high)
    [Tags]    filter_products
    [Setup]   user should be logged in
    Given the home page is open
    When user filter products by price (low to high)
    Then products should be filtered
```

5.3. Escrita dos Scripts

Nesta etapa, os scripts de teste foram implementados para automatizar os casos de teste definidos anteriormente.

5.3.1. Keywords de Configuração

Foi criado uma *keyword* para configuração das plataformas, chamada de **Before Tests**, onde é utilizada para configurar os parâmetros iniciais necessários para a execução dos testes automatizados em ambas as plataformas, Android e iOS. Ela define as configurações padrão e específicas para cada plataforma. O código da *keyword* utiliza uma estrutura condicional que ajusta as configurações conforme a plataforma em uso.

5.3.2. Keywords do Usuário

As *keywords* do usuário foram implementadas com base nas *keywords* já existentes nas bibliotecas presentes no projeto, incluindo a Appium Library e as *keywords* builtin do Robot Framework. A seguir, exemplos de *keywords* criadas para a implementação dos testes automatizados:

```
login page is open
    Wait Until Element Is Interactive   ${USERNAME_FIELD}
    Wait Until Element Is Interactive   ${PASSWORD_FIELD}
    Capture Page Screenshot

user filter products by price (low to high)
    Filter Products     filter_type=Price (low to high)

Filter Products
    [Arguments]    ${filter_type}
    Wait Until Element Is Interactive    ${BTN_FILTER}
    Click Element                        ${BTN_FILTER}
    Wait Until Element Is Visible        accessibility_id=${
       filter_type}    timeout=5
    Click Element                        accessibility_id=${
       filter_type}
```

O uso de *keywords* facilita a reutilização e a manutenção dos scripts de teste. Elas integram funcionalidades essenciais, como verificação de elementos, captura de screenshots e navegação no menu, que são comuns aos cenários de teste de login, logout e aplicação de filtros.

5.4. Execução dos Testes

Os testes automatizados foram executados em ambos os emuladores (Android e iOS) para garantir a cobertura multiplataforma. Utilizando o script criado anteriormente, foi possível realizar a execução paralela e organizar os resultados em pastas separadas por plataforma. Para realizar a execução dos testes, foi inserido no terminal o seguinte comando:

```
pwsh ./run_tests.ps1
```

Este comando executa o script PowerShell `run_tests.ps1`, que gerencia a execução paralela dos testes em ambas as plataformas. O script coleta os resultados de cada execução e os organiza de acordo com a estrutura definida.

Cada execução foi monitorada e seus resultados foram documentados. O script gerou como saída no terminal um resumo geral do teste, indicando o número total de testes executados, quantos passaram, quantos falharam e indica onde os testes foram armazenados, o caminho para arquivos de log e report, os resultados também foram categorizados por funcionalidade de teste, o que possibilita a análise e a identificação de falhas

específicas. A estrutura de diretórios para os resultados foi organizada da seguinte forma, conforme a Figura 6:

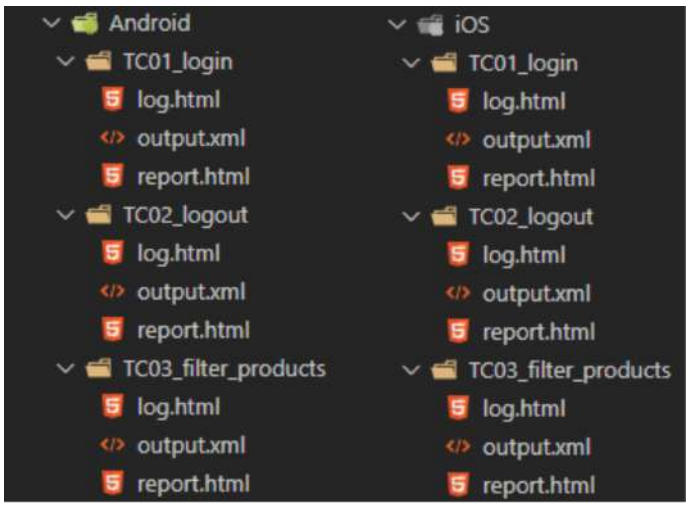

Figura 6. Organização dos logs - Android e iOS. Elaborado pelo autor (2024).

Esta disposição permite a navegação e análise dos resultados, possibilitando a rápida localização de logs e relatórios específicos de cada funcionalidade testada em cada plataforma.

5.5. Análise dos vereditos de teste

Os resultados dos testes foram documentados e analisados para identificar discrepâncias e problemas de compatibilidade. Um visão geral dos resultados pode ser vista na 5.5, as análises focaram em identificar **Discrepâncias de Comportamento**: Diferenças nos comportamentos dos testes entre as plataformas baseadas nos resultados das execuções dos testes automatizados e; **Problemas de Compatibilidade**: Componentes ou funcionalidades que apresentaram problemas específicos em uma das plataformas.

Plataforma	Cenário 1	Cenário 2	Cenário 3
Android	Passou	Passou	Falhou
iOS	Passou	Falhou	Passou

Tabela 3. Resultados dos Cenários de Teste nas Plataformas Android e iOS

5.5.1. Análise da execução do TC01: Login

A execução do *TC01_Login* em ambas as plataformas (Android e iOS) obteve resultados de sucesso. Indicando que a funcionalidade está funcionando corretamente em ambas as plataformas, evidenciado na Figura 7, onde é possivel ver que todos as keywords foram executadas e passaram.

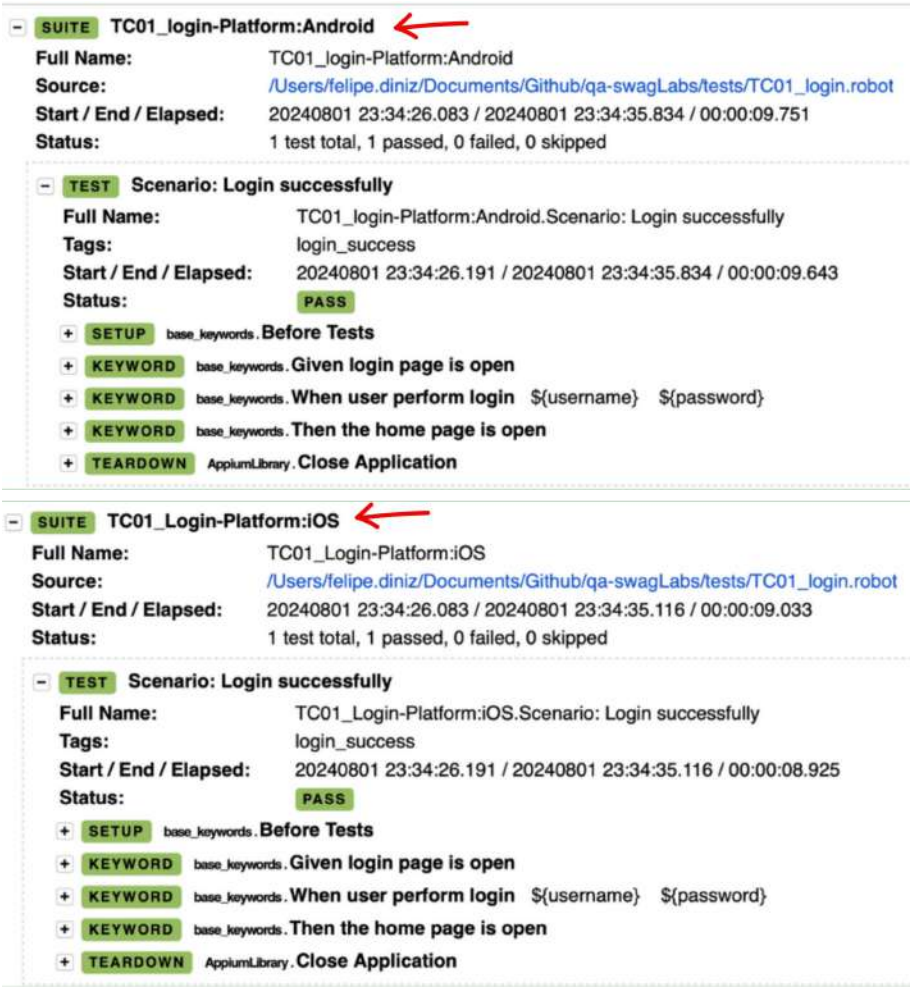

Figura 7. Resultados do TC01: Login - Android e iOS.

5.5.2. Análise da execução do TC02: Logout

Ao executar o *TC02_Logout*, observamos que o teste foi bem-sucedido na plataforma Android, mas falhou na plataforma iOS. A falha no iOS ocorreu porque o sistema relatou que o elemento do botão de logout estava presente na tela, conforme a Figura 8, onde é possível ver que no relatório de execução iOS, o teste retornou o erro *Page Should not have contained element 'accessibility_id=test-LOGOUT'*, ao analisar a tela no momento da falha o erro foi possível caracterizar a falha como um falso negativo, uma vez que o sistema relatou que o botão de logout estava presente na tela quando, na verdade, não estava (conforme a Figura 9). Esse problema de compatibilidade sugere a necessidade de ajustes no reconhecimento dos elementos de interface no iOS para evitar falsos negativos [Chen et al. 2008].

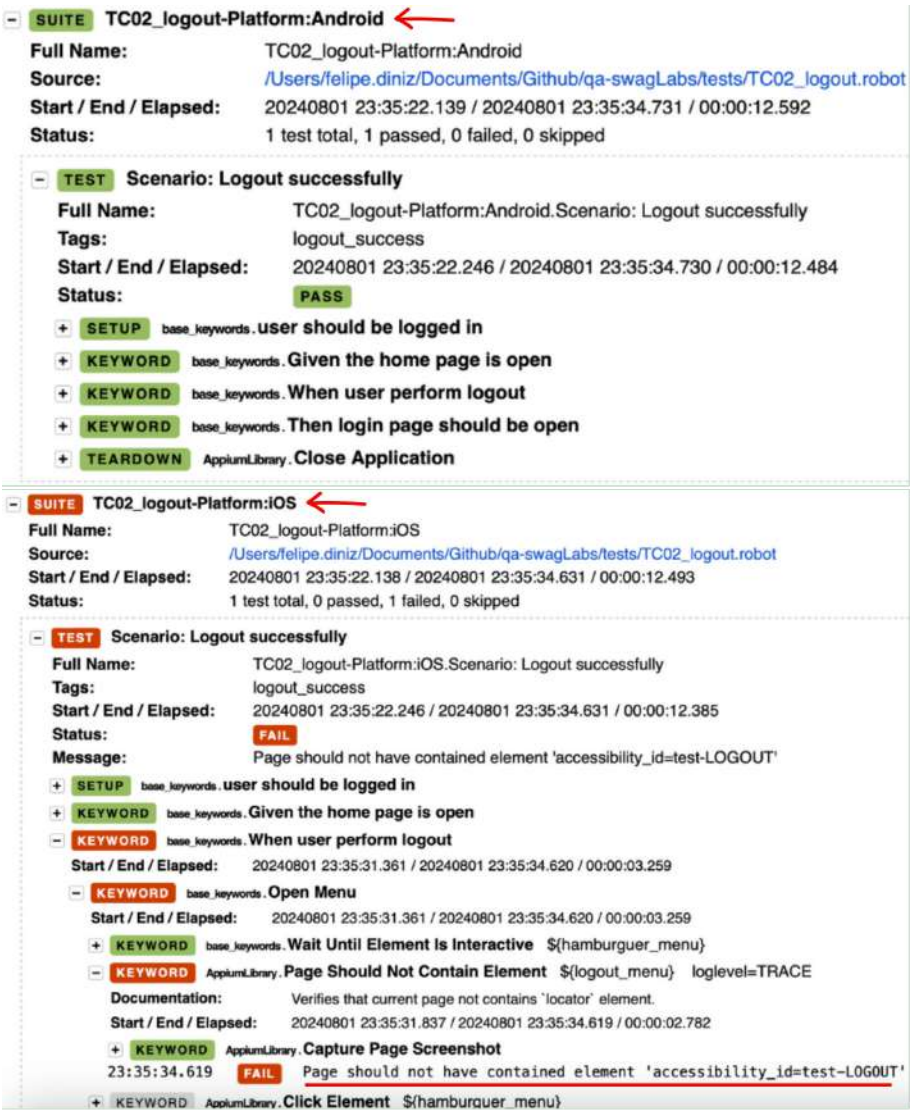

Figura 8. Resultados do TC02: Logout - Android e iOS.

5.5.3. Análise da execução do TC03: Filtro de Produtos

Na execução do *TC03_Filter Products*, todos os testes dessa funcionalidade falharam na plataforma Android. A falha ocorreu porque o elemento `accessibility_id=${filter_type}` não estava sendo encontrado conforme a Figura 10. Manualmente, analisamos a interface da aplicação no momento que o teste automatizado reporta a falha e foi verificado que apesar do teste não encontrar o elemento `accessibility_id=$filter_type`, ele era exibido na interface (conforme a Figura 11), podendo caracterizar a falha como um falso negativo. Após refazer uma inspeção utilizando a ferramenta *Appium Inspector*, foi constatado que, apenas no Android, os elementos em questão não possuíam o atributo *accessibility id*.

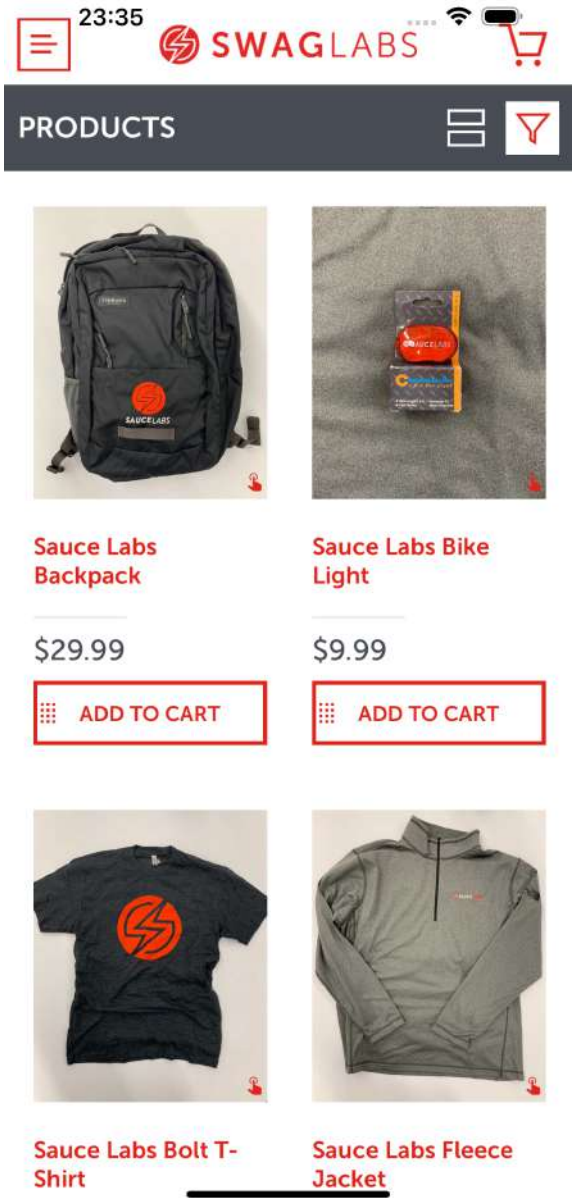

Figura 9. Falso negativo no TC02. Elaborado pelo autor (2024)

6. Discussão de Resultados

A execução dos testes automatizados proporcionou uma visão clara das capacidades e limitações da automação de testes multiplataforma utilizando Appium e Robot Framework. Durante o processo, foi possível identificar diferenças significativas entre as plataformas Android e iOS, especialmente no que diz respeito à identificação e manipulação de elementos de interface, o que impactou diretamente a execução de alguns cenários de teste. De modo geral, os testes que envolviam funcionalidades básicas e elementos simples entre as plataformas apresentaram resultados consistentes. No entanto, as funcionalidades um pouco mais específicas, demandam adaptações nos scripts de teste.

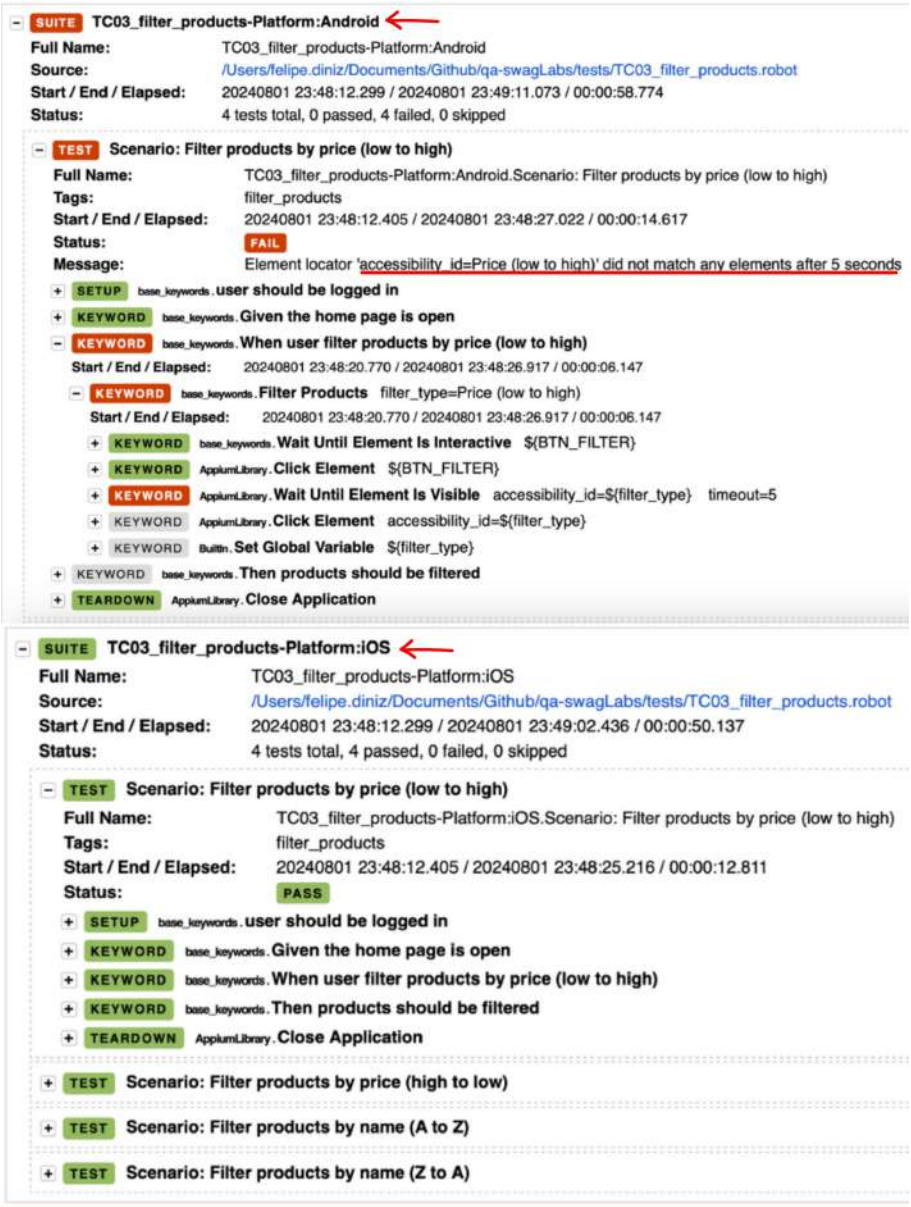

Figura 10. Resultados do TC03: Filtro de Produtos - Android e iOS.

Essa necessidade de ajustes e o comportamento variável entre as plataformas ressaltam a complexidade envolvida na automação de testes multiplataforma e a importância de um planejamento mais criterioso na implementação dos testes.

Os desafios encontrados, como problemas de identificação de elementos e divergências nas características de componentes visuais, reforçam a ideia de que uma única abordagem de automação pode não ser totalmente eficiente para ambos os sistemas operacionais. Ferramentas como o Appium oferecem suporte para a automação em diferentes plataformas, mas, para alcançar resultados mais confiáveis, é necessário levar em consideração as particularidades de cada plataforma. Os resultados também revelaram a necessidade de uma maior padronização entre as plataformas móveis para garantir que

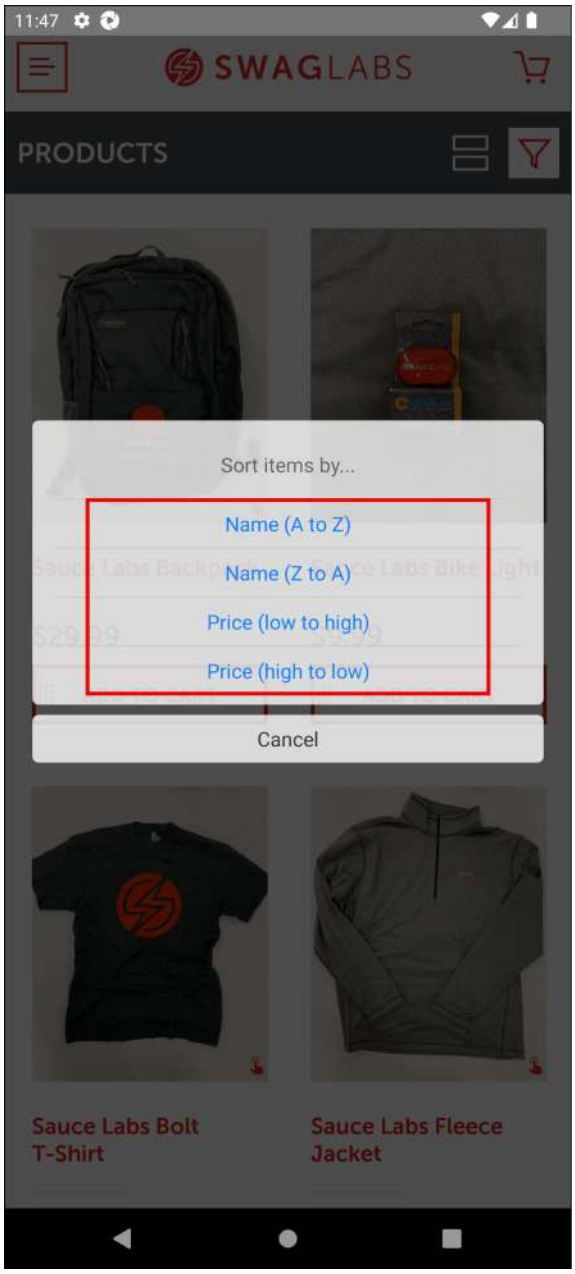

Figura 11. Falso negativo no TC03. Elaborado pelo autor (2024)

os testes automatizados possam ser executados de forma mais homogênea. A falta de consistência entre os atributos de identificação dos elementos entre Android e iOS gerou falhas em alguns testes, indicando que ainda existem desafios técnicos a serem superados para garantir a plena portabilidade dos testes automatizados. Com base nos resultados observados, foi possível montar a Tabela 4, que sumariza para os resultados e recomendações para cada falha observada.

Plataforma	CT	Elemento	Falha reportada	Recomendação
iOS	2	accessibility_id=test-LOGOUT	Page should not have contained element	Evitar o uso de *keywords* que verifiquem a presença do elemento apenas na árvore lógica da interface. Implementar uma *keyword* que verifique se o elemento está visível na tela para ter um comportamento mais coerente entre as plataformas.
Android	3	accessibility_id=Price (low to high)	Element locator did not match any elements after 5 seconds	Fazer a inspeção dos elementos em todas as plataformas implementadas.
Android	3	accessibility_id=Price (high to low)	Element locator did not match any elements after 5 seconds	Fazer a inspeção dos elementos em todas as plataformas implementadas.
Android	3	accessibility_id=Name (A to Z)	Element locator did not match any elements after 5 seconds	Fazer a inspeção dos elementos em todas as plataformas implementadas.
Android	3	accessibility_id=Name (Z to A)	Element locator did not match any elements after 5 seconds	Fazer a inspeção dos elementos em todas as plataformas implementadas.

Tabela 4. Falhas reportadas e recomendações para a execução dos testes automatizados

7. Conclusão

Este trabalho apresentou um estudo sobre a portabilidade de scripts de teste automatizados utilizando Robot Framework e Appium nas plataformas Android e iOS. O objetivo principal foi avaliar como essas ferramentas lidam com as diferenças entre as duas plataformas, destacando as limitações e os desafios enfrentados na automação de testes multiplataforma. A escolha de Appium e Robot Framework foi motivada por serem tecnologias amplamente adotadas no mercado, de código aberto, e com suporte multiplataforma, características que as tornam adequadas para o contexto de testes de aplicações móveis.

Os testes realizados evidenciaram que, embora funcionalidades básicas possam ser automatizadas de forma eficaz em ambas as plataformas, as diferenças entre os componentes de interface de Android e iOS impactam diretamente a portabilidade dos scripts de teste. Foram observadas falhas relacionadas à identificação de elementos e diferenças nos atributos de acessibilidade entre as plataformas, gerando falsos negativos que comprometem a confiabilidade dos testes automatizados. Esse resultado destaca a necessidade de adaptações específicas para cada plataforma durante o desenvolvimento dos scripts de teste, além de reforçar a importância do uso de ferramentas como o Appium Inspector

para mapear os componentes de interface de maneira eficaz.

Em termos de contribuições, este estudo oferece uma visão inicial sobre as limitações e os desafios enfrentados ao utilizar ferramentas de automação como o Appium e o Robot Framework para testar aplicações móveis em múltiplas plataformas. Com base nos resultados obtidos, propomos que, ao desenvolver scripts de teste automatizados, não se deve focar apenas em uma plataforma. É crucial realizar, no mínimo, uma verificação básica em ambas as plataformas (Android e iOS) para garantir que a execução dos testes tenha uma maior chance de ser efetiva e precisa, evitando falhas causadas por diferenças na estrutura dos componentes de interface. Além disso, uma importante contribuição para o desenvolvimento do Appium e do Robot Framework é a constatação de que algumas *keywords* deveriam funcionar de maneira uniforme em ambas as plataformas, mas falham devido às diferenças na forma como as plataformas renderizam a árvore de elementos. Esse ponto ressalta a necessidade de aprimoramento das ferramentas para que elas considerem essas variações, facilitando a portabilidade dos scripts de teste. A experiência realizada neste trabalho também servirá como uma referência para futuras versões das tecnologias Robot Framework e Appium, permitindo avaliar se as limitações observadas foram solucionadas ou mitigadas nas novas versões, e fornecendo feedback para a melhoria contínua dessas ferramentas.

Ainda existem muitas contribuições que podem surgir a partir de análises futuras. Em particular, há um potencial significativo para a investigação de aplicações geradas em outras linguagens de programação e em outras plataformas, que abrangem os mesmos softwares mobile. Sugerimos a investigação de técnicas mais avançadas para a adaptação automática dos scripts de teste entre plataformas, utilizando aprendizado de máquina ou inteligência artificial para mapear e corrigir discrepâncias entre componentes de interface. Além disso, sugerimos a aplicação do estudo em um conjunto maior de funcionalidades e a diferentes tipos de aplicações poderia fornecer uma visão mais abrangente das limitações e das melhores práticas para a automação de testes em dispositivos móveis.

Referências

[App 2024] (2024). *Appium*. OpenJS Foundation. Acessado em: 15 de junho de 2024.

[IST 2024] (2024). *Certified Tester Foundation Level Syllabus*. ISTQB. Acessado em: 15 de junho de 2024.

[Anjum et al. 2017] Anjum, H., Babar, M. I., Jehanzeb, M., Khan, M., Chaudhry, S., Sultana, S., Shahid, Z., Zeshan, F., and Bhatti, S. N. (2017). A comparative analysis of quality assurance of mobile applications using automated testing tools. *International Journal of Advanced Computer Science and Applications*, 8(7):249–255.

[AppiumProject 2024] AppiumProject (2024). Appium - github repository. `https://github.com/appium/appium`. Acessado em: 24 de setembro de 2024.

[Behrang and Orso 2018] Behrang, F. and Orso, A. (2018). Automated test migration for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 384–385, New York, NY, USA. Association for Computing Machinery.

[Bolsu 2024] Bolsu, S. (2024). Appium library. `https://github.com/serhatbolsu/robotframework-appiumlibrary`.

[Bordi 2018] Bordi, K. (2018). Overview of test automation solutions for native cross-platform mobile applications. Master's thesis, Lappeenranta University of Technology.

[Chen et al. 2008] Chen, J., Zhou, H., and Bruda, S. D. (2008). Combining model checking and testing for software analysis. In *2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 206–209.

[do Nascimento Mendes and Dias-Neto 2016] do Nascimento Mendes, I. and Dias-Neto, A. C. (2016). A process-based approach to test usability of multi-platform mobile applications. In Marcus, A., editor, *Design, User Experience, and Usability: Design Thinking and Methods*, pages 456–468, Cham. Springer International Publishing.

[El-Kassas et al. 2016] El-Kassas, W. S., Abdullah, B. A., Yousef, A. H., and Wahba, A. M. (2016). Enhanced code conversion approach for the integrated cross-platform mobile development (icpmd). *IEEE Transactions on Software Engineering*, 42(11):1036–1053.

[Islam 2024] Islam, G. (2024). *On the real world practice of Behaviour Driven Development*. Phd thesis, University of Glasgow.

[Kushwaha and Misra 2008] Kushwaha, D. S. and Misra, A. K. (2008). Software test effort estimation. *SIGSOFT Softw. Eng. Notes*, 33(3).

[Leffingwell and Widrig 2003] Leffingwell, D. and Widrig, D. (2003). *Managing Software Requirements: A Use Case Approach*. Addison-Wesley object technology series. Addison-Wesley.

[Mahmoud et al. 2023] Mahmoud, A. T., Muhammad, A. A., Yousef, A. H., Medhat, W., Zayed, H. H., and Selim, S. (2023). Compiler-based web services code conversion model for different languages of mobile application. In *2023 Intelligent Methods, Systems, and Applications (IMSA)*, pages 464–469.

[Menegassi and Endo 2020] Menegassi, A. A. and Endo, A. T. (2020). Automated tests for cross-platform mobile apps in multiple configurations. *IET Software*, 14(1):27–38.

[Myers et al. 1979] Myers, G. J., Sandler, C., and Badgett, T. (1979). *The Art of Software Testing*. Wiley Publishing, 1rd edition.

[Qin et al. 2019] Qin, X., Zhong, H., and Wang, X. (2019). Testmig: migrating gui test cases from ios to android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 284–295, New York, NY, USA. Association for Computing Machinery.

[RobotFrameworkFoundation 2024] RobotFrameworkFoundation (2024). Robot framework - github repository. `https://github.com/robotframework/robotframework`. Acessado em: 24 de setembro de 2024.

[Salonen 2012] Salonen, V. (2012). Automatic portability testing.

[Yu et al. 2023] Yu, S., Fang, C., Ling, Y., Wu, C., and Chen, Z. (2023). Llm for test script generation and migration: Challenges, capabilities, and opportunities. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 206–217.