NICOLAS DE OLIVEIRA AQUINO BARBOSA

DESENVOLVIMENTO DE TESTES AUTOMATIZADOS NA TECHNOLOG SISTEMAS

LAVRAS – MG

2024

NICOLAS DE OLIVEIRA AQUINO BARBOSA

DESENVOLVIMENTO DE TESTES AUTOMATIZADOS NA TECHNOLOG SISTEMAS

Relatório de estágio supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. DSc. Ricardo Terra Nunes Bueno Villela

Orientador

LAVRAS – MG

2024

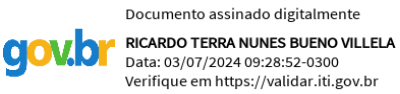
NICOLAS DE OLIVEIRA AQUINO BARBOSA

DESENVOLVIMENTO DE TESTES AUTOMATIZADOS NA TECHNOLOG SISTEMAS

Relatório de estágio supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADA em 17 de maio de 2024.

Prof. DSc. Ricardo Terra Nunes Bueno Villela	UFLA
Prof. DSc. Paulo Afonso Parreira Júnior	UFLA
Valdeci Soares da Silva Junior	TECHNOLOG



Prof. DSc. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2024

AGRADECIMENTOS

À minha família por todo suporte durante a graduação, que mesmo diante de todas as dificuldades, sempre apoiou meus objetivos e torceu por minhas conquistas.
Ao meu orientador, professor Ricardo Terra, por toda a paciência e auxílio prestado na elaboração deste trabalho.
À Universidade Federal de Lavras, em especial ao Departamento de Ciência da Computação, pelo suporte e conhecimento fornecidos.
À Technolog e sua equipe de desenvolvimento pela confiança no meu trabalho e por toda a experiência profissional vivenciada.

RESUMO

O presente documento apresenta um relatório de estágio realizado ao longo de onze meses na área de qualidade de software da empresa Technolog Sistemas, sediada na cidade de Lavras. Durante a execução das atividades, que consistiam na aplicação das técnicas de testes de caixa preta, caixa branca, usabilidade e segurança, o estagiário iniciou sua jornada de automação de testes, anteriormente manuais, para as aplicações voltadas a dispositivos móveis da empresa. Com o uso das tecnologias e ferramentas Appium, Cucumber e Flutter Test, foram obtidas melhorias nos processos de testes de aplicações Android nativas e de aplicações Flutter. Além disso, foram identificadas diferenças relevantes entre as aplicações geradas de forma nativa e as aplicações de compilação cruzada geradas com o *framework* Flutter. No caso de aplicações Android nativas, ocorreu uma exposição mais abrangente de seus elementos, possibilitando a integração com o Appium e, consequentemente, o teste da aplicação em ambiente de desenvolvimento e de produção. Por outro lado, no caso de aplicações Flutter, observaram-se dificuldades na exposição dos elementos, o que inviabilizou os testes com o Appium, tornando necessário o uso da ferramenta própria do *framework*, Flutter Test, que permitiu a automação dos testes apenas em ambiente de desenvolvimento. Durante o estágio, o estagiário —e também autor deste documento— adquiriu conhecimentos essenciais de gestão e análise de qualidade de software, além de experiência prática relevante no mercado de trabalho ao aplicar os conhecimentos acadêmicos.

Palavras-chave: Software. *Mobile*. Appium. Cucumber. Flutter. Testes. Qualidade. Garantia da qualidade.

ABSTRACT

This document presents a internship report carried out over eleven months in the software quality area of the company Technolog Sistemas, located in the city of Lavras. During the activities, which consisted of applying black-box, white-box, usability, and security testing techniques, the intern began his journey of test automation, previously manual, for the company's mobile applications. Using the technologies and tools Appium, Cucumber, and Flutter Test, the intern improved the testing processes of native Android applications and Flutter applications. Furthermore, the trainee identified relevant differences between natively generated applications and cross-compiled applications generated with the Flutter framework. In the case of Android native apps, there was a more comprehensive exposure of its elements, enabling integration with Appium and, consequently, testing of the application in development and production environments. On the other hand, in the case of Flutter applications, the intern faced difficulties in exposing the elements, which made testing with Appium unfeasible, requiring the use of the framework's own tool, Flutter Test, which allowed automation of tests only in the development environment. During the internship, the intern—and also author of this document—acquired essential knowledge of software quality management and analysis, as well as relevant practical experience in the job market by applying the academic knowledge.

Keywords: Software. Mobile. Appium. Cucumber. Flutter. Testing. Quality. Quality Assurance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

A área de Engenharia de Software vem sendo pautada desde meados de 1970, período esse conhecido por diversos fatores depreciativos dos produtos digitais da época, principalmente a qualidade (MICHAEL *et al.*, 2006). Desde então, esse aspecto vem sendo estudado e aprimorado visando que a denominada crise de software não venha a se repetir.

De acordo com TGT e Unicamp (2023), em sua pesquisa denominada 'O Futuro da Qualidade de Software', 46% das empresas brasileiras estão insatisfeitas com processos de qualidade de seus produtos, sendo um desafio a capacitação dos colaboradores em processos de automação por conta dos rápidos avanços das ferramentas e *frameworks*. Ainda conforme TGT e Unicamp (2023), tais questões reforçam a necessidade de esforços voltados para o interesse dos colaboradores para aprendizado e aplicação dos conceitos que possibilitem a melhoria da qualidade dos produtos, desde o início de sua jornada de trabalho.

O presente trabalho tem como objetivo apresentar as atividades realizadas ao longo do estágio supervisionado na empresa Technolog Sistemas, durante o período de 24 de abril de 2023 a 10 de abril de 2024, na área de qualidade de software.

A primeira parte deste trabalho consiste na caracterização da empresa concedente de estágio e do setor de alocação do estagiário, introduzindo os conceitos relevantes do segmento de gestão de frotas que corroboram com a importância da gestão da qualidade dos produtos da empresa. Em seguida, são apresentadas as noções introdutórias aos conceitos, ferramentas e atividades do estagiário, além de suas contribuições e aprendizados.

As atividades do estagiário incluíam testes de aplicações web e *mobile*, aplicando os conceitos de testes de caixa branca e caixa preta, testes de usabilidade, segurança das aplicações e banco de dados. A partir dessas atividades, surgiu o interesse em aprimorar os processos e reduzir o tempo de execução dos

testes das aplicações móveis, iniciando-se o estudo e a aplicação de ferramentas de automação de testes. Durante o estágio, foram utilizadas as ferramentas Appium e Cucumber para testes das aplicações Android nativas, e Flutter Test para as aplicações de compilação cruzada desenvolvidas em Dart com Flutter. O estagiário identificou pontos positivos e negativos de ambas as ferramentas, bem como as dificuldades do contexto em que estava inserido, aplicando a teoria aprendida na Universidade à prática do mercado de trabalho.

Ao analisar as aplicações móveis, constatou-se que, nas aplicações Android nativas, os identificadores dos elementos eram mais facilmente acessíveis. Isso possibilitou a realização de testes tanto nas versões de desenvolvimento quanto nas de produção dos aplicativos, utilizando a ferramenta Appium. Por outro lado, as aplicações Flutter não apresentaram o mesmo padrão. Para interagir com os elementos identificados por suas chaves (*keys*), foi necessário utilizar a ferramenta específica Flutter Test. Essa diferença não inviabilizou a automação dos testes, mas tornou-a impraticável nas versões de produção dos aplicativos Flutter. Ao final, foram identificados avanços nos processos de testes e pontos de melhoria na qualidade dos produtos, desde o desenvolvimento até a manutenção.

A organização deste documento é configurada da seguinte forma: na Seção 2, é realizada uma caracterização mais detalhada da empresa e do setor de alocação do estagiário, incluindo os ritos e práticas da equipe. Na Seção 3, são introduzidos os conceitos relevantes para o entendimento das atividades realizadas e dos fluxos de trabalho do estagiário. Na Seção 4, são apresentadas as atividades e averiguações padrões realizadas na área de alocação do estagiário, bem como o demonstrativo do fluxo dessas atividades. Por fim, nas Seções 5 e 6 são apresentadas, respectivamente, as principais atividades contributivas do estagiário para a empresa e seus aprendizados, e as considerações finais do relatório.

2 TECHNOLOG

No presente momento, com sede estabelecida na cidade de Lavras – MG, a Technolog Sistemas encontra-se como uma entidade empresarial dedicada ao desenvolvimento de soluções tecnológicas nos setores de logística e transporte. Essa empresa, com nove anos de atuação no mercado, tem como missão prover produtos que aprimorem a capacidade de seus clientes em gerir suas frotas de maneira eficaz.

Atualmente, munida de informações advindas de tecnologia de telemetria, a base de dados da empresa contabiliza um acervo que ultrapassa a marca dos 100 milhões de registros, todos eles armazenados e atualizados em tempo real, provenientes das frotas de veículos dos mais de dez clientes ativos. Por conseguinte, impõe-se a necessidade de um processo rigorosamente definido para o desenvolvimento e análise de qualidade das soluções que são disponibilizadas. Nesse contexto, a Technolog Sistemas adota a metodologia ágil de desenvolvimento orientado ao comportamento, conhecida como *BDD - Behaviour Driven Development*, em que se desenvolve baseando-se em requisitos ou especificações (BINAMUNGU; EMBURY; KONSTANTINOU, 2018), bem como emprega técnicas de testes de aplicações fundamentadas em Caixa Preta, Caixa Branca e Banco de Dados (SOMMERVILLE, 2011).

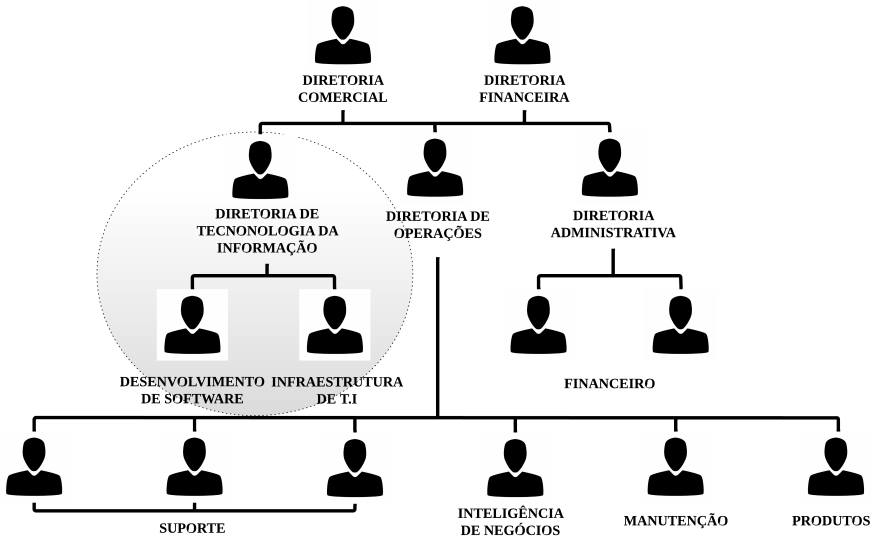
Dividindo-se em duas vertentes, os produtos da empresa concedem informações críticas que servem de base para análises voltadas ao *Eco-Driving*[1] e à *Safe Driving*[2], proporcionando, como resultado principal, a redução de custos para seus clientes. Destacam-se entre os produtos de seu acervo, o Darwin Portal, uma plataforma web que disponibiliza funcionalidades de gestão, e os produtos *mobile*

[1] *Eco-Driving* refere-se a um conjunto de técnicas e decisões estratégicas, táticas e operacionais aplicadas para reduzir o consumo de combustível durante a condução de um veículo (SIVAK; SCHOETTLE, 2012).

[2] *Safe Driving* é um conjunto de técnicas aplicadas ao monitoramento da direção, abrangendo aspectos visuais relacionados ao motorista e aspectos não visuais que envolvem os parâmetros de rodagem do veículo (KAPLAN *et al.*, 2015).

direcionados a gestores e colaboradores encarregados da condução das frotas sob responsabilidade dos clientes. No contexto organizacional atual, a empresa está estruturada em departamentos, seguindo a hierarquia delineada na Figura 2.1, com destaque para o departamento de Tecnologia da Informação, onde o estagiário e também o autor deste relatório estava alocado.

Figura 2.1 – Organograma Technolog Sistemas



Fonte: Autor (2024)

No âmbito do setor de Tecnologia da Informação, os colaboradores, subdivididos em diretor, coordenador, desenvolvedores e analistas de qualidade, reúnem-se regularmente para orquestrar as tarefas que serão executadas ao longo das *sprints*[3]. Diariamente, esses profissionais dialogam sobre as atividades em curso e, ao final de cada ciclo, procedem a uma minuciosa avaliação. Nesse contexto, o estagiário foi designado para a função de Analista de Qualidade de Software, sendo responsável pela execução de processos de avaliação e testes, em estrita conformidade com as normas e protocolos estabelecidos pela empresa.

[3] De forma sucinta, *Sprint* refere-se ao bloco de tempo definido pelas equipes em que as tarefas delegadas devem ser executadas, normalmente de 1 a 2 semanas (FIRDAUS *et al.*, 2019).

3 *BACKGROUND*

Esta seção tem como propósito fornecer ao leitor os conceitos essenciais para compreender aplicações móveis. Na Seção 3.1, são introduzidas noções relativas ao desenvolvimento móvel, bem como desenvolvimento nativo Android e desenvolvimento com o *framework* Flutter. Já na Seção 3.2, são discutidos conceitos e ferramentas pertinentes aos testes em plataformas de desenvolvimento móvel e seu processo de automação.

3.1 DESENVOLVIMENTO PARA PLATAFORMAS *MOBILE*

Aplicativos *mobile*, aplicativos móveis, ou apps, são as nomenclaturas utilizadas para definir as aplicações de software para dispositivos móveis como *smartphones*, *tablets* e *smartwatches* (PEREIRA; SILVA, 2020). Enquanto aplicações web são feitas com intuito de funcionamento em computadores e *notebooks* com o auxílio de um navegador, e não são instaláveis em seus dispositivos de acesso, as aplicações *mobile* contam com maior flexibilidade quanto ao tipo de dispositivo, podem ser desenvolvidas em diferentes plataformas utilizando diferentes linguagens de programação e podem ser inseridas nos aparelhos (HERTZ, 2022).

Em contrapartida, tanto aplicações web como *mobile*, podem ser desenvolvidas de diversos modos em relação à utilização de linguagem e tecnologias (SHAH; SINHA; MISHRA, 2019), sendo o desenvolvimento de aplicações móveis divididas em:

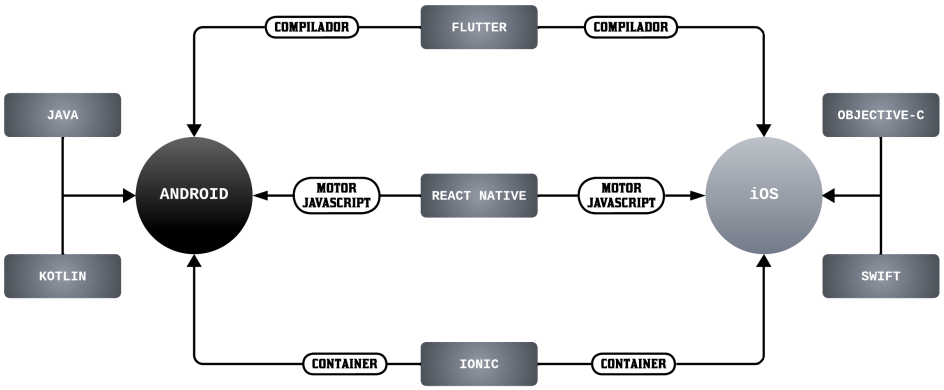
Desenvolvimento Nativo: Trata-se do desenvolvimento *mobile* na linguagem nativa da plataforma alvo, tornando sua instância em questão exclusiva para uso no ambiente selecionado. Sua principal desvantagem é a necessidade de refazer o aplicativo em caso de uso em outra plataforma, ocasionando custos adicionais (SMUTNÝ, 2012). Dentre as principais linguagens e plataformas, tem-se

Java[1] e Kotlin[2], as mais populares para o sistema operacional Android de propriedade da Google, e Objective-C[3] e Swift[4], as mais populares para o sistema iOS de propriedade da Apple (HERTZ, 2022).

Desenvolvimento Não Nativo ou Multiplataforma: Trata-se do desenvolvimento *mobile* em que o código fonte é escrito uma única vez e se utilizam *frameworks*, APIs[5] ou *containers* que possibilitam a geração de arquivos utilizáveis em diferentes plataformas, dos quais podem ser citados Flutter, React Native e Ionic (RISSI; DALLILO, 2022).

Conforme pode ser observado na Figura 3.1, no modo de desenvolvimento não nativo, podem ser originadas pelo menos três tipos de aplicações para dispositivos móveis (EL-KASSAS *et al.*, 2017):

Figura 3.1 – Visão geral dos modos de desenvolvimento e seus tipos de aplicações



Fonte: Autor (2024)

Aplicações Não Nativas - Híbridas: São desenvolvidas em geral utilizando os mesmos artefatos de aplicações web, como HTML (*HyperText Markup Language*

[1] https://www.java.com/pt-BR/
[2] https://kotlinlang.org/
[3] https://developer.apple.com/documentation/objectivec/objective-c_runtime
[4] https://www.apple.com/br/swift/
[5] API (*Application Programming Interface*) é uma interface de software reutilizável por agentes externos à origem de desenvolvimento (LAMOTHE; GUÉHÉNEUC; SHANG, 2021).

ou Linguagem de Marcação de Hipertexto), CSS (*Cascading Style Sheets* ou Folha de Estilo em Cascatas) e JavaScript. Após serem embutidas em *containers* web nativos específicos que possibilitam o acesso ao *hardware* do dispositivo móvel, conseguem acessar as funções nativas e executar como uma aplicação própria do sistema (RAHUL RAJ; TOLETY, 2012). Para esse tipo de desenvolvimento, tem-se o Ionic[6] como um possível candidato que, anteriormente, por meio do Apache Cordova[7] e atualmente por meio do Capacitor[8], possibilita o desenvolvimento multiplataforma (WAHLBRINCK; BONIATI, 2017).

Aplicações Não Nativas - Interpretadas: Diferente das aplicações híbridas que necessitam de um intermediário para realização de comunicação com o dispositivo móvel, esse tipo de aplicação acessa diretamente o aparelho. Por meio de seu motor JavaScript onde é interpretada, consegue acesso às funcionalidades nativas do dispositivo (DA SILVA; SANTOS, 2014). Uma das ferramentas utilizadas para gerar aplicações desse modelo é o React Native[9] que utiliza-se da mesma linguagem e estrutura do React[10], *framework* de desenvolvimento de aplicações web. Isso possibilita facilidade para construção de aplicações *mobile* devido a semelhança com o desenvolvimento web (NEVES; JUNIOR, 2020).

Aplicações Não Nativas - Compilação Cruzada: Nesse tipo de aplicação, escreve-se o código em uma única linguagem e o compilador utilizado se encarrega de gerar arquivos executáveis das plataformas desejadas e possibilitadas pela ferramenta utilizada (RAHUL RAJ; TOLETY, 2012). Um dos candidatos de destaque que possibilitam esse tipo de criação é o Flutter[11], um *framework* desenvolvido

[6] https://ionicframework.com/

[7] Apache Cordova é um *framework* de código aberto que atua como um *container* que envolve o aplicativo originalmente desenvolvido em padrão web e possui uma coleção de *plug-ins* para acessar funções nativas dos dispositivos *mobile* (BOSNIC; PAPP; NOVAK, 2016).

[8] https://capacitorjs.com/

[9] https://reactnative.dev/

[10] https://react.dev/

[11] https://flutter.dev/

pela Google que compila códigos desenvolvidos na linguagem Dart e gera arquivos multiplataformas (GOOGLE, 2023).

3.1.1 ANDROID

De propriedade da Google, o Android constitui-se de um sistema operacional de código aberto para dispositivos móveis. Em quesito desenvolvimento, semelhante a uma aplicação web que possui sua interface desenvolvida basicamente com HTML e CSS, as aplicações Android nativas possuem interfaces desenvolvidas com XML (*Extensible Markup Language*), utilizando-se de construções em *Views* através de componentes designados como *Activities* para representação de sua interface de interação com o usuário (DANIELSSON, 2016).

Em um *layout* de desenvolvimento Android nativo, são encapsuladas diversas representações de objetos como botões, imagens, textos e outros, por meio do conceito de agrupamentos chamados *ViewGroups* que constituem um *layout* da aplicação Android (OLSSON, 2020). Nesse contexto, a interface das aplicações é construído declarativamente em arquivos XML separados dos arquivos Java, nos quais são concentradas as lógicas de comportamento dos componentes (CHEON; CHAVEZ, 2021).

De acordo com Google e Alliance (2023) em sua documentação para desenvolvedores da plataforma, o desenvolvimento nativo Android utiliza as linguagens Java e Kotlin em conjunto com bibliotecas utilitárias para criação das aplicações. Além disso, conta com um conjunto de conceitos específicos no cenário de dispositivos móveis, dos quais elencam-se a seguir os de principais relevância para entendimento deste trabalho:

1. `View`: Componente de área retangular que representa o bloco de contrução básico de componentes da interface do usuário e tratamento de seus eventos relacionados;

2. `ViewGroup`: São componentes formados a partir da junção de uma ou mais *Views* e fazem parte da base para *layouts* e contêineres de visualização;
3. `Activity`: São representações de janelas contendo *Views* ou *ViewGroups* que permitem interação com o usuário; e
4. `Android Manifest`: Consiste em um arquivo obrigatório, de extensão XML, que descreve os componentes utilizados na aplicação Android bem como a listagem de *Activities*, permissões necessárias de uso do dispositivo e recursos mínimos de hardware e software para execução.

3.1.2 FLUTTER

O *framework* Flutter, desenvolvido pela Google, constitui uma ferramenta de código aberto para o desenvolvimento de aplicativos móveis, web e *desktop*. Segundo Ferrero (2022), ele é composto por um SDK (*Software Development Kit* ou Kit de Desenvolvimento de Software) que utiliza a linguagem Dart para a criação, além de um conjunto de elementos de interface reutilizáveis, conhecidos como *widgets*, na construção dos aplicativos. Ademais, emprega as linguagens C e C++ em seu compilador.

Por meio da construção de *widgets*, o *framework* forma uma árvore de elementos similar à construção da árvore DOM (*Domain Object Model* ou Modelo de Documento por Objetos) do HTML em aplicações web. De acordo com Fentaw (2020), isso possibilita a realização de uma nova renderização da árvore caso seja detectada uma alteração de estado em algum dos seus nós, representados pelos *widgets*. Esses *widgets*, inspirados nos componentes do *framework* React, permitem a construção da interface do aplicativo por meio de reutilização, dispensando a necessidade de criação própria. São disponibilizados diversos componentes semelhantes aos das aplicações web, como texto, linha, coluna, pilha e *containers*, além de opções mais avançadas, como listas e gráficos (GOOGLE, 2023). Ao contrário das aplicações Android nativas, cujos compo-

nentes são expressos separadamente em arquivos de extensão XML, os do Flutter estão presentes nos *scripts* Dart, juntamente com as lógicas de comportamento da aplicação (CHEON; CHAVEZ, 2021).

Com esses recursos, o *framework* também oferece a funcionalidade de recarregamento rápido (*Hot Reload*) que permite a aplicação de correções no código durante o desenvolvimento. Conforme Olsson (2020), essas correções são injetadas pela Máquina Virtual Dart, possibilitando um recarregamento ágil do aplicativo no dispositivo físico ou virtual.

Segundo Wang (2022), ao gerar executáveis tanto para a plataforma Android quanto para iOS, a estrutura básica de diretórios de um projeto desenvolvido com Flutter consiste em:

1. `lib`: diretório principal da aplicação, onde os códigos são desenvolvidos na linguagem Dart;
2. `android`: diretório que contém os arquivos responsáveis pela assistência na geração do executável para a plataforma Android;
3. `ios`: diretório que abriga os arquivos responsáveis pela assistência na geração do executável para a plataforma iOS;
4. `web`: diretório onde ficam os arquivos responsáveis pela assistência na geração do executável para a plataforma web; e
5. `test`: diretório onde são localizados os arquivos utilizados para testes de unidade e de integração (*widget tests*).

Todos os diretórios previamente descritos se localizam na raiz do projeto, onde também se encontra o arquivo de configurações do *framework*, o `pubspec.yaml`, responsável pela definição das dependências, versões e outras características do projeto.

3.2 TESTES EM PLATAFORMAS *MOBILE*

Testes de software em geral são elaborados e executados para garantir sua qualidade e que seu funcionamento esteja de acordo com o planejado (SOMMERVILLE, 2011). Já se tratando de plataformas *mobile*, diferente das aplicações web, os testes podem variar de acordo com o sistema operacional, tamanho de tela, plataforma e memória do dispositivo (VILLANES; BEZERRA COSTA; DIAS-NETO, 2015). Apesar da grande variedade de testes que podem ser aplicados em ambos os tipos de aplicações, os testes de caixa branca, nos quais se tem acesso ao código fonte, e os testes de caixa preta, nos quais se testa o arquivo executável final da aplicação, são comumente aplicados (SOMMERVILLE, 2011).

3.2.1 TESTES FUNCIONAIS

O foco deste trabalho são os testes funcionais, que constituem um dos tipos de testes de caixa preta. Esses testes têm como objetivo avaliar as funcionalidades da aplicação, considerando os aspectos sob a perspectiva do usuário, analisando a interface e validando os requisitos do sistema (OLEGÁRIO *et al.*, 2019). Comumente, esses testes são denominados *end-to-end* por testarem o software do ínicio ao fim ou de uma ponta à outra, podendo ser manuais ou automatizados a depender da complexidade e crescimento da aplicação testada (FERNANDES; FONSECA, 2020).

Neste relatório, o conhecimento relevante sobre testes funcionais se dá no intuito de automatizá-los, ou seja, executá-los via *scripts* desenvolvidos em linguagem específica. Tais testes se diferenciam dos testes manuais por promoverem reúso através de sua utilização como testes de regressão, que são testes que viabilizam testar diferentes versões de uma mesma aplicação e, por conseguinte, funcionalidades já anteriomente testadas (VIANA, 2006). Em geral, são realiza-

dos com auxílio de ferramentas de automação como o Selenium[12] e Cypress[13] para aplicações web e o Appium[14] (*Selenium for Apps*) para aplicações *mobile*.

A Figura 3.2 e o Código 3.1 apresentam, respectivamente, um exemplo de interface para uma aplicação de agendamento de eventos e um teste funcional automatizado da funcionalidade de salvar um evento utilizando o Appium.

Figura 3.2 – Exemplo de interface de aplicação para agendamento de eventos

Fonte: Autor (2024)

```
@Test
public void testeSalvarEvento() {
    driver.findElementById("descricao").click().sendKeys("Apresentacao de Seminario");
    driver.findElementById("dt-inicio").click().sendKeys("20/06/2023");
    driver.findElementById("hr-inicio").click().sendKeys("15:00:00");
    driver.findElementById("dt-termino").click().sendKeys("20/06/2023");
    driver.findElementById("hr-termino").click().sendKeys("16:30:00");
    driver.findElementById("btn-salvar").click();

    assertEquals("Evento salvo com sucesso", driver.findElemenById("id-toast").getText());
    assertTrue(driver.findElementById("id-evento").getText().matches("[0-9]+"));
    assertEquals("Nicolas de Oliveira", driver.findElementById("evento-responsavel").getText());
}
```

Código 3.1 – Exemplo de teste funcional com Appium

[12] https://www.selenium.dev/
[13] https://www.cypress.io/
[14] https://appium.io/

No início do teste, são encontrados os campos de entrada de texto da interface utilizando a função `findElementById()` que recebe como parâmetro uma *string* com o id do campo buscado e são utilizadas as funções `click()` e `sendKeys()` para clicar no elemento e digitar o respectivo texto desejado (linhas 3 a 7). Em seguida, é realizada a busca e clique no botão de salvar (linha 8), redirecionando para a tela com as informações do evento salvo para que seja possível validar a mensagem de sucesso ao salvar (linha 10), e o texto dos campos de id `"id-evento"` e `"evento-responsavel"` por meio das assertivas de verificação do teste (linhas 11 e 12).

3.2.2 AUTOMAÇÃO DE TESTES FUNCIONAIS

Em um processo de automação de testes, podem ser utilizadas ferramentas e artefatos facilitadores para a interação com a aplicação por meio de *scripts* automatizados. Embora a ideia deste estudo fosse utilizar o Appium em todas as aplicações da Technolog, não foi possível utilizá-lo em aplicações Flutter por motivos descritos na Seção 5.4.1. Portanto, esta seção introduz, também, o Flutter Test que foi utilizado na automação de testes em um dos aplicativos da empresa (Seção 5.4.2). Com isso, são apresentadas a seguir noções introdutórias sobre as ferramentas relevantes para este trabalho.

Appium: Desenvolvido pela OpenJS Foundation, o Appium é um *framework* de código aberto utilizado para automação de testes de interface em dispositivos móveis, navegadores, *desktops* e outros, disponibilizando suporte a diversas linguagens, como Java, Python, Ruby e JavaScript (OPENJS, 2012). Utilizando como base de abstração o *Selenium Webdriver*, um servidor próprio HTTP (*Hypertext Transfer Protocol*) denominado *Appium Server*, e um *proxy* para comunicação com o dispositivo físico ou virtual que executará os *scripts* desenvolvidos, o Appium viabiliza a recorrência de realização de massas de testes para vários cenários (SINGH; GADGIL; CHUDGOR, 2014). Da mesma forma de alguns *fra-*

meworks de automação para web, como o Selenium, o Appium possibilita a geração de *scripts* através do método *Record and Playback*[15], com sua ferramenta auxiliar *Appium Inspector*, ou a criação manual utilizando o próprio cliente e um ambiente de desenvolvimento a escolha do usuário (WANG; WU, 2019).

Cucumber: Desenvolvido pela SmartBear Software, o Cucumber consiste em uma ferramenta *open-source* para desenvolvimento orientado a comportamento que realiza validações de passos definidos por seu usuário que em conjunto configuram um cenário (SMARTBEAR, 2019). Seguindo conjuntos de regras com escopo e gramáticas bem definidas, também chamadas de *Gherkin*, a ferramenta consegue dividir um cenário projetado em etapas para sua efetiva validação e geração de relatório de tempo de execução e resultados obtidos. As validações são feitas baseadas na grámatica proposta no formato *Gherkin*, que utiliza as palavras chaves *Given* (Dado), *When* (Quando), *Then* (Então), *And* (E) e *But* (Mas) que, apesar de serem interpretadas de forma igualitária, possibilitam legibilidade e escrita em linguagem natural dos cenários de testes (MARCHESE; ZEN; VILLAFIORITA, 2013). Os Códigos 3.2 e 3.3 correspondem a um exemplo básico de um cenário para funcionalidade de *login*, e os passos gerados e validados pelo Cucumber, respectivamente.

```
#language: pt

Funcionalidade: Login

  Cenario: Realiza login com sucesso
    Dado que estou na tela de login
    Quando preencher os dados dos campos login e senha corretamente
    E clicar em entrar
    Entao devo ter acesso a tela inicial da aplicacao
```

Código 3.2 – Exemplo de Cenário de *Login* em uma Aplicação

[15] *Record and Playback* consiste em um recurso de captura de interações com a tela de uma aplicação para posterior reprodução, inclusas as ações de digitação, clique e movimentação de mouse (TUOVENEN; OUSSALAH; KOSTAKOS, 2019).

```
@Dado("^que estou na tela de login$")
public void queEstouNaTelaDeLogin() { }

@Quando("^preencher os dados dos campos login e senha corretamente$")
public void preencherOsDadosDosCamposLoginESenhaCorretamente() { }

@E("^clicar em entrar$")
public void clicarEmEntrar() { }

@Entao("^devo ter acesso a tela inicial da aplicacao$")
public void devoTerAcessoATelaInicialDaAplicacao() { }
```

Código 3.3 – Exemplo de passos validados pelo Cucumber

Flutter Test: De acordo com Google (2023), a ferramenta de testes própria do Flutter consiste em uma dependência de desenvolvimento que proporciona acesso a bibliotecas contendo métodos e funcionalidades para a realização de três tipos de testes em aplicativos desenvolvidos com o *framework*: (i) testes de unidade, (ii) testes de *widgets* e (iii) testes de integração. O primeiro é utilizado para verificar o funcionamento de trechos específicos de código, como funções ou classes. O segundo é empregado para verificar o comportamento de um elemento (*widget*) específico em uma tela, enquanto o último avalia o funcionamento conjunto de diversos componentes da aplicação como um todo, de ponta a ponta (*end-to-end*), conforme definido explicitamente na Seção 3.2.1 e também sendo o foco deste trabalho. O Código 3.4 exemplifica um teste de *widget*, semelhante à um teste de integração, utilizando a dependência Flutter Test.

```
group("Teste Login", () {
    testWidgets("Deve realizar login com sucesso", (WidgetTester tester) async {
      app.main();
      await tester.pumpAndSettle();
      await tester.tap(find.byKey(const Key("campoLogin")));
      await tester.enterText(find.byKey(const Key("campoLogin")), "meu login");
      await tester.tap(find.byKey(const Key("campoSenha")));
      await tester.enterText(find.byKey(const Key("campoSenha")), "minha senha");
      await tester.tap(find.byKey(const Key("btnLogin")));
      expect("Meu nome de usuário", tester.getText(find.byKey(const Key("nomeUsuario"))));
    });
  });
}
```

Código 3.4 – Exemplo de *widget* com Flutter Test

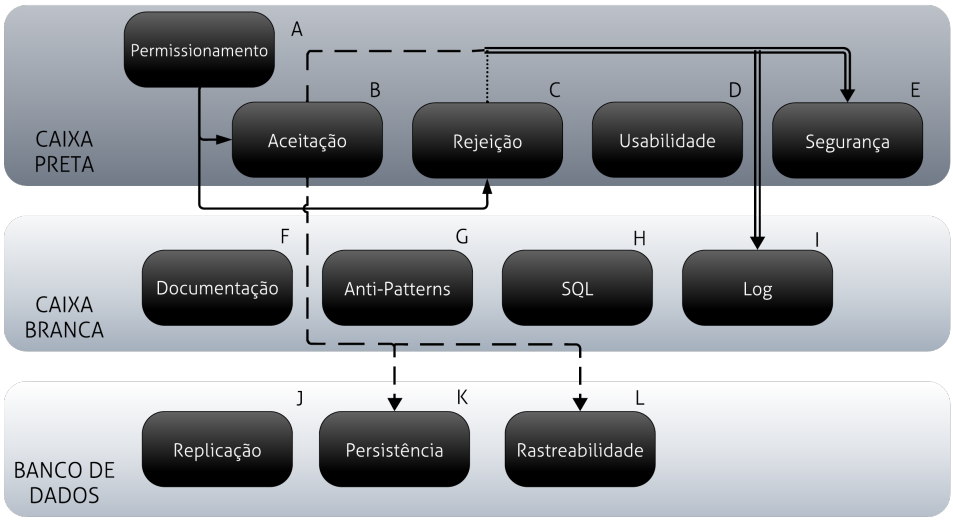
No início do código, são incorporadas duas funções que têm o propósito de notificar o *framework* sobre quais linhas subsequentes representam, respectivamente, a execução de um grupo ou suíte de testes, bem como o próprio teste ou testes a serem executados, identificados nas linhas 1 e 2 pelas funções `group()` e `testWidgets()`. A primeira recebe como parâmetro uma *string* identificadora do grupo ao qual os testes subsequentes pertencem, enquanto a segunda recebe uma *string* identificadora da unidade de teste a ser executada dentro do bloco e um elemento da classe `WidgetTester`, que contém uma série de funções auxiliares para interação com os componentes da aplicação.

Na linha 3, é invocada a função `main()`, que é responsável pela abertura da tela inicial da aplicação testada, nesse caso, a tela de *login*. Na linha 4, a função `pumpAndSettle()` é chamada, colocando a execução do código em espera até que o primeiro quadro da aplicação seja renderizado com seus *widgets*. As linhas 5, 7 e 9 têm a responsabilidade de realizar a ação de clique em um *widget* por meio da função `tap()`. Essa função recebe como parâmetro um *widget* e, em ambos os casos, o *widget* foi selecionado utilizando a função de busca `find.byKey()` com o identificador correspondente do *widget* em seu parâmetro. Por sua vez, as linhas 6 e 8 executam a ação de escrita em um campo por meio da função `enterText()`. Essa função também recebe um *widget* como parâmetro, além de um texto a ser inserido no respectivo campo desse *widget*. Ao realizar, respectivamente, as ações de clique e escrita no campo de *login*, clique e escrita no campo de senha e clique no botão de *login* (linhas 5 a 9), espera-se, na linha 10, que seja possível visualizar o nome do usuário o qual deve ser igual ao primeiro parâmetro da função `expect()`.

4 PROCESSO DE TESTES NA TECHNOLOG SISTEMAS

As seções seguintes apresentam em detalhes os testes realizados na concedente de estágio, sendo a Seção 4.1 referente aos Testes de Caixa Preta ou *Black Box*, a Seção 4.2 referente aos Testes de Caixa Branca ou *White Box*, e a Seção 4.3 referente aos Testes de Banco de Dados. A Figura 4.1 apresenta o fluxo de execução dos três testes citados anteriormente.

Figura 4.1 – Fluxograma de testes



Fonte: Autor (2024)

4.1 TESTES DE CAIXA PRETA

No âmbito da Technolog, durante os testes de Caixa Preta, ocorre uma análise abrangente do funcionamento, usabilidade, segurança e permissões das funcionalidades exigidas no módulo em avaliação (item `A` da Figura 4.1). Em relação ao funcionamento, são aplicados testes de aceitação (item `B`) para avaliar os resultados do sistema para um conjunto de entradas válidas, bem como testes de rejeição (item `C`), quando aplicáveis, para verificar os resultados em relação a um conjunto de entradas inválidas, juntamente com suas tentativas de recuperação

em caso de falhas. Os casos de teste utilizados são detalhadamente documentados para que os erros identificados possam ser reproduzidos pelo desenvolvedor responsável. Além disso, são anexadas evidências visuais, como capturas de tela e vídeos, conforme necessário para esclarecer a complexidade da descrição textual do problema. Se um objetivo não for atendido ou se ocorrer algum problema relacionado à codificação da funcionalidade testada, o teste é considerado como falha, e a tarefa é devolvida ao desenvolvedor responsável para correção.

Nos testes de usabilidade (item `D`), são avaliados aspectos relacionados à intuitividade, tempo de resposta e características consideradas essenciais para a categoria do módulo testado, de acordo com um *checklist* mínimo estipulado pela equipe. Por exemplo, em telas de visualização de tabelas, são requisitadas funcionalidades básicas, tais como a capacidade de reordenar, ocultar, expandir e ordenar dados em colunas, bem como a opção de baixar os dados exibidos em formatos PDF e XLS, conforme figura 4.2. Além disso, espera-se que a visualização da tela seja atualizada de forma apropriada e que seja possível maximizar e minimizar a janela de exibição da funcionalidade testada. Nos testes de usabilidade, são consideradas falhas a falta de cumprimento das características básicas, tempos de resposta que excedam os limites das solicitações, a utilização de campos clicáveis e campos de entrada e saída de dados sem título e *placeholder* ou informações minimamente coerentes com sua finalidade atribuída.

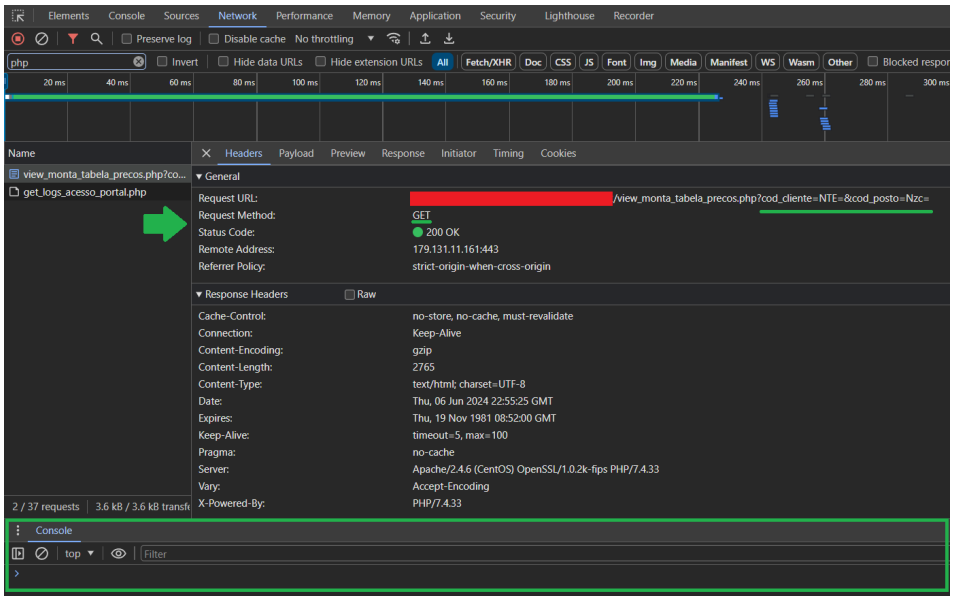
No que diz respeito aos testes de segurança (item `E`), particularmente no contexto de aplicações web, faz-se uso do recurso de inspeção do navegador para analisar todas as requisições efetuadas durante os testes de funcionamento e usabilidade. Falhas são identificadas quando são observadas exibições desnecessárias, advertências (*warnings*) ou erros no *console* do navegador, bem como requisições do tipo GET que envolvam a transmissão não criptografada de dados sensíveis, conforme figura 4.3. Essas verificações são especialmente relevantes dada a natu-

Figura 4.2 – Exemplo de aplicação dos testes de usabilidade

Fonte: Autor (2024)

reza do setor em que a empresa atua, lidando com informações de geolocalização, detalhes de veículos e seus respectivos condutores.

Figura 4.3 – Exemplo de aplicação dos testes de segurança



Fonte: Autor (2024)

Além disso, durante a execução de cada um dos testes mencionados anteriormente – testes de aceitação, rejeição, usabilidade e segurança – os resultados também são avaliados levando em consideração as permissões associadas ao usuá-

rio logado no sistema. Isso assegura que cada usuário tenha acesso apenas às funcionalidades correspondentes ao seu nível de privilégio. No âmbito da empresa, são identificados os seguintes tipos de usuários:

1. Usuário interno da Technolog (que é sempre administrador)
2. Usuário externo não administrador
3. Usuário externo administrador

Nos testes de permissões, verifica-se e valida-se a disponibilidade de cada usuário para realizar operações específicas, acessar informações de placas e navegar pelos menus relacionados ao seu grau de privilégio.

4.2 TESTES DE CAIXA BRANCA

No âmbito da Technolog, os testes de Caixa Branca são conduzidos em todos os arquivos que passaram por alterações devido à codificação requerida em uma tarefa específica. Tais arquivos são obrigatoriamente informados pelo desenvolvedor responsável.

Nesses testes, são examinados diversos aspectos, incluindo documentação, *Anti-Patterns*[1], análise de consultas SQL (*Structured Query Language*) e registro de erros (*log*). No que concerne aos testes de documentação (item `F` da Figura 4.1), são inspecionadas as funções criadas, seus parâmetros, explicações relacionadas ao fluxo de execução, especialmente quando há complexidade, e o cumprimento das normas de desenvolvimento específicas do projeto, seja ele web ou *mobile*. Por sua vez, nos testes de *Anti-Patterns* (item `G`), identificam-se partes de código e comentários irrelevantes, bem como código obsoleto. Os testes de SQL (item `H`) concentram-se na verificação do uso de *NOLOCK* em transações no banco de dados, na utilização de *Upper Case* (letras maiúsculas) em comandos e na formatação das consultas SQL em arquivos DAO (*Data Access Object*).

[1] *Anti-Patterns* são práticas e soluções de *design* ruins que prejudicam a qualidade, manutenção e evolução de um software (OUNI *et al.*, 2015)

Por fim, os testes de registro de erros (item `I`) tem como finalidade averiguar que durante os testes de Caixa Preta não tenham sido registrados problemas no *log*.

4.3 TESTES DE BANCO DE DADOS

No âmbito da Technolog, os testes de banco de dados têm como objetivo a verificação da integridade dos dados e a rastreabilidade das alterações realizadas. Dentro dessa categoria, nos testes de replicação (item `J` da Figura 4.1), são conduzidos testes de verificação da criação e replicação de *jobs*, *procedures* e tabelas. Já os testes de persistência (item `K`) verificam se os artefatos gerados por meio dos testes de aceitação foram salvos em banco de dados. Por fim, os testes de rastreabilidade (item `L`) têm como objetivo a confirmação da existência e preenchimento das colunas de rastreabilidade. Essas colunas têm a responsabilidade de registrar informações sobre o momento e o autor da última modificação efetuada em um registro da tabela. A aplicação desses testes visa confirmar as criações e modificações que foram efetuadas durante os testes de caixa preta.

O Código 4.1 exemplifica uma consulta a uma tabela fictícia de viagens. Nessa consulta, são selecionadas as colunas de rastreabilidade (linhas 5 a 8) que são preenchidas seguindo o fluxo de B para L, conforme Figura 4.1. Além disso, a consulta utiliza o comando *NOLOCK* (linha 10), que permite ao banco realizar consultas sem a necessidade de espera pela liberação de bloqueios decorrentes de outras transações em andamento que utilizem a mesma tabela. Isso ocorre porque, no segmento organizacional onde a empresa está inserida, a disponibilidade tem um grau de importância superior à exatidão dos dados. Essa prioridade difere, por exemplo, do segmento bancário, no qual a exatidão é preferível à disponibilidade.

```
SELECT
        placa,
        latitude,
        longitude,
        usuario_resp_cadastro,
        data_cadastro,
        usuario_resp_alteracao,
        data_alteracao
FROM
        tabela_viagens WITH(NOLOCK)
```

Código 4.1 – Exemplo de consulta SQL

Com a tabela resultante, verifica-se a igualdade dos códigos de usuários registrados nas colunas de cadastro e alteração com o código do usuário executor das ações, juntamente com a data e hora da execução das modificações.

5 ATIVIDADES REALIZADAS

Esta seção lista as principais atividades executadas pelo estagiário e fornece informações sobre as contribuições feitas à empresa no que diz respeito ao processo de qualidade descrito na Seção 4.

5.1 ESTUDO DAS FERRAMENTAS DE AUTOMAÇÃO DE TESTES

No decorrer do período de estágio, foram efetuados testes em novas funcionalidades e correções nos produtos web e *mobile*. Com o decorrer do tempo, identificou-se que o estagiário demonstrou maior afinidade pela vertente das aplicações móveis. Tendo em vista o tempo investido na realização de testes manuais para esses aplicativos e a oportunidade de aprimorar os processos internos da organização, deu-se início ao projeto de testes automatizados para as aplicações *mobile*.

Nesse início do processo de automação, tornou-se necessário aprofundar o conhecimento acerca das funcionalidades e limitações das ferramentas selecionadas, descritas na Seção 3.2.2, para otimizar o fluxo de automação.

Inicialmente, no que diz respeito ao *Gherkin*, foi realizada uma análise das aplicações das palavras-chave em relação às necessidades dos cenários a serem validados. Foi identificado que, além das anteriormente mencionadas na Seção 3.2.2 que são palavras de validação na linguagem natural, existem também as seguintes palavras de descrição de cenários e contextualização (SMARTBEAR, 2019):

'Feature' (Funcionalidade): Essa palavra-chave é utilizada para descrever funcionalidades 'simples', nas quais ocorre uma validação ou caso de teste por vez. O exemplo anteriormente apresentado no Código 3.2 na Seção 3.2.2 ilustra esse tipo de situação.

'Background' (Contexto): Essa palavra-chave é empregada para descrever um ou mais pré-requisitos comuns a uma funcionalidade, englobando um ou mais cenários. Nesse caso, o contexto é validado antes da execução de cada um desses

cenários. Continuando com o exemplo do Código 3.2, seria possível incluir um cenário de *login* com credenciais inválidas, adicionando um contexto no qual, como pré-requisito, o usuário deve possuir uma conta registrada no sistema.

'Scenario Outline' (Esquema do Cenário): Essa palavra-chave é empregada para descrever um cenário com múltiplas entradas de dados através de uma tabela de '*Examples*' (Exemplos), na qual a primeira linha contém chaves e as seguintes linhas os respectivos valores associados. A utilização dessa palavra-chave do *Gherkin* permite ao Cucumber identificar que o cenário deve ser executado para cada uma das linhas da tabela, possibilitando testes com massas de dados.

Com base nessas informações, foi concluído, pelo estagiário, que o uso do 'Esquema do Cenário' da linguagem descritiva *Gherkin* e a ferramenta Cucumber seria fundamental para a automação dos testes e proporcionaria uma redução considerável no tempo dedicado a verificações manuais. Além disso, sempre que possível, a combinação com o 'Contexto' proveria melhora na legibilidade e na compreensão das funcionalidades testadas.

Durante o estudo e tentativa de sua aplicação, o estagiário identificou algumas limitações no uso do 'Cenário de Fundo' com o Cucumber. O estagiário tentou gerar dinamicamente as linhas da tabela de exemplos a partir de um arquivo de entrada, mas não obteve sucesso. Ao buscar solucionar essa questão, as alternativas encontradas estavam voltadas para o uso de estruturas de repetição e leitura de arquivos, sem a necessidade de recorrer à tabela. No entanto, isso não era viável, uma vez que divergia da proposta de utilização do Cucumber como uma ferramenta de auxílio nesse processo. Portanto, a necessidade de leitura estática dos dados na tabela de exemplos foi considerada uma limitação e, em caso de alterações nas entradas, foi preciso efetuar as modificações diretamente na tabela do arquivo da *feature*.

Em relação ao Appium, buscou-se o entendimento de suas funcionalidades e semelhanças com o Selenium, que era uma ferramenta previamente utilizada

pelo estagiário. Já em relação ao Flutter Test, foi realizado um estudo da documentação de sua ferramenta de testes mas com uma comparação com o Appium. Foram elencadas as funções essenciais e indispensáveis nos *scripts* de automação de testes. Nesse entendimento, o estagiário identificou e priorizou as seguintes funcionalidades das ferramentas (OPENJS, 2012; GOOGLE, 2023):

Busca de elemento pelo identificador único: Essa funcionalidade, considerada a mais necessária e amplamente utilizada, constitui a base da interação com os aplicativos nos testes automatizados. Utilizando a função `AndroidFindById()`, o Appium permite referenciar um elemento visível na tela por meio de um identificador previamente estabelecido no código-fonte da aplicação e a subsequente realização de interações com o mesmo. Já na ferramenta de testes do Flutter, o processo é realizado através da classe `Finder` e seu método `byKey()`.

Busca de elemento por identificador de acessibilidade: Similar à funcionalidade anterior e representada pela função `AndroidFindByAcessibilityId()` no Appium, essa função permite a captura e interação com um elemento por meio de seu identificador de acessibilidade, que é um identificador secundário ao mencionado anteriormente. Geralmente, essa funcionalidade é empregada em elementos que não possuem o identificador único, o que pode ocorrer em casos nos quais a aplicação é gerada por ferramentas específicas que não permitem a inserção desse elemento de unicidade. A representação dessa funcionalidade na ferramenta do Flutter é por meio da função `bySemanticsLabel()` da classe `Finder`.

Ação de toque (Clique): Essa funcionalidade possibilita a interação com elementos clicáveis, como botões, caixas de seleção, campos de texto, elementos arrastáveis, entre outros. No Appium, a interação ocorre por meio do retorno das funções `AndroidFindBy*()`, que disponibilizam um `MobileElement` com a opção de uso da função `click()`. Seguindo o mesmo padrão, a ferramenta do Flutter possibilita

essa interação por meio do método `tap()`, disponível após o retorno do método `by*()` descrito anteriormente para casos em que o elemento seja clicável.

Ação de digitar: Relacionada ao retorno das funções `AndroidFindBy*()`, no Appium a função `sendKeys()` permite o envio de uma sequência de caracteres para um campo de texto. No Flutter, o padrão se repete por meio da utilização da função `enterText()`.

Ação de capturar atributo: Também relacionada ao retorno das funções `AndroidFindBy*()`, essa função permite a obtenção dos atributos de um `MobileElement`, por exemplo, o texto. No Appium, essa funcionalidade é representada pela função `getAttribute()`, que recebe como parâmetro o nome do atributo a ser capturado. No Flutter, essa execução ocorre utilizando o método de mesmo nome a depender do *driver* utilizado. Para se obter o texto de um *widget* utilizando a classe `Finder`, por exemplo, pode se utilizar o método `evaluate().single.widget` realizando um *casting* para `Text` acrescentando `as Text` no seu final.

Abrir e fechar o aplicativo: São funcionalidades essenciais que permitem o controle durante a execução de diversos casos de teste, além do gerenciamento da instância em execução para a devida desalocação de memória ao final da execução. São representadas pelas funções `launchApp()` e `quit()`, respectivamente no Appium. No Flutter, as funções são, respectivamente, `startApp()` e `exitApplication()`.

Ação de voltar (undo): Essa funcionalidade permite o retorno da tela do dispositivo a um estado anterior ao atual. Ela se faz necessária nos casos de dispositivos e aplicativos que não possuam um botão específico para essa finalidade e é representada no Appium pela ação conjunta das funções `navigate().back()`. No Flutter, essa ação ocorre por meio da classe `NavigatorState` e seu método `pop()`.

Definições de capacidades desejadas (desired capabilities): Essa funcionalidade permite ao usuário definir os detalhes relacionados à plataforma, dispositivo e aplicativo de teste. As principais capacidades do Appium definidas pelo estagiário durante a elaboração dos *scripts* de automação foram as seguintes:

1. `PLATFORM_NAME`: Define o tipo de plataforma utilizada, por exemplo, Android ou iOS;
2. `AUTOMATION_NAME`: Define o *driver* utilizado para comunicação entre o dispositivo e o *framework*;
3. `DEVICE_NAME`: Define o dispositivo utilizado para o teste, sendo identificado pelo seu nome;
4. `appium:appPackage`: Define o aplicativo a ser testado, identificado pelo seu nome de pacote;
5. `appium:appActivity`: Define a primeira tela a ser executada ao iniciar um teste, identificada pelo seu nome de *Activity*.

No Flutter, a maioria dessas capacidades já estão pré-definidas no arquivo `pubspec.yaml`, podendo ser alteradas inclusive por meio da execução dos testes via linha de comando.

Com as informações resultantes desse estudo sobre as ferramentas, o estagiário encontrou-se apto a iniciar o processo de automação e verificar a compatibilidade dos aplicativos e funcionalidades com as ferramentas selecionadas.

5.2 DESENHO DA AUTOMAÇÃO DE TESTES PARA APPS

Inicialmente, conduziu-se um estudo e uma coleta de informações referentes aos tipos e tecnologias associadas a alguns dos produtos da Technolog, conforme documentado na Tabela 5.1.

Tabela 5.1 – Análise de Aplicações da Technolog

Aplicativo	Tipo	Linguagem	Tecnologia	Deploy
AppU	Nativo	Java	-	Sim
AppV	Nativo	Java	-	Sim
AppX	Nativo	Java	-	Sim
AppW	Interpretada	JavaScript	*React Native*	Sim
AppY	Compilação Cruzada	Dart	*Flutter*	Sim
AppZ	Compilação Cruzada	Dart	*Flutter*	Não

A partir desse estudo, constatou-se que: (i) a maioria das aplicações selecionadas já estava em produção e (ii) predominava o desenvolvimento nativo, especialmente com Java. Além disso, ao considerar que a maioria dessas aplicações era destinada a clientes externos, tomou-se a decisão de iniciar os estudos de automação de testes para as aplicações aqui denominadas 'AppU' e 'AppV' por questões de confidencialidade.

O desfecho desse processo de estudo foi fator decisório pela utilização dos *frameworks* Appium e Cucumber. A escolha pelo Appium se deu devido à sua semelhança com o Selenium, que proporciona um suporte robusto e bem estabelecido. Além disso, o Cucumber foi escolhido devido às melhorias que oferece em termos de legibilidade na criação e execução dos cenários a serem automatizados.

5.3 AUTOMAÇÃO DE TESTES PARA 'APPU' E 'APPV'

Esta seção lista as principais atividades executadas pelo estagiário no que diz respeito ao processo de automação de testes para os aplicativos nativos 'AppU' e 'AppV'.

5.3.1 DECISÕES DE PROJETO

Antes de iniciar o processo de automação em si, o estagiário teve a necessidade de definir questões relacionadas à estruturação do projeto. Por se tratar de uma iniciativa inexistente no repertório da organização, foi necessário tomar decisões sobre os seguintes dois aspectos:

Localização do repositório: Durante as fases iniciais de implementação das dependências do projeto em relação às ferramentas escolhidas, o estagiário enfrentou problemas relacionados ao versionamento de componentes das aplicações. Isso ocorreu devido ao fato de que, seguindo a regra interna de execução em uma versão específica do Android, algumas dependências não ofereciam suporte para versões mais recentes das ferramentas Appium e Cucumber. Isso resultou em problemas de execução ao tentar executar no mesmo repositório da aplicação. Portanto, considerando que a execução dos *scripts* necessitava apenas da versão .apk do aplicativo ou da instalação prévia em um dispositivo, o estagiário optou por criar um repositório exclusivo para testes, no qual continha apenas as classes, assertivas e dependências dos *frameworks* de testes. Isso permitiu a instalação das ferramentas em suas versões mais atuais e seguras, sem conflitos com os componentes da aplicação em si, bastando que a instância desejada do aplicativo fosse instalada no dispositivo, ou referenciada ao aloca-lá em um dos diretórios desse repositório.

Geração de valor para a organização: Outra decisão de projeto tomada pelo estagiário foi a escolha da ferramenta de geração automatizada de relatórios de testes, o *Allure Report*[1]. Essa ferramenta possibilita a geração gráfica dos resultados, detalhando sucessos, falhas, tempo de execução e outros detalhes, além de oferecer a capacidade de exportar os resultados em diferentes formatos. Na visão do estagiário, essa ferramenta permite análises gerenciais para gestores que não possuem profundo conhecimento em programação, pois pode haver falta de compreensão dos resultados por pessoas de diferentes níveis de conhecimento em outros setores da organização.

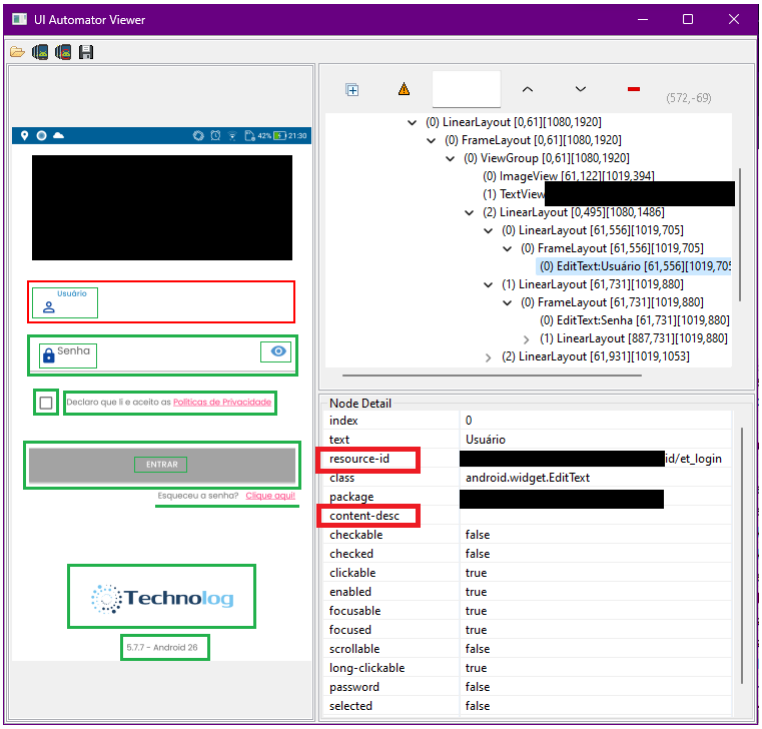
Com base nessas decisões, o estagiário efetivamente deu início ao processo de automação dos testes manuais na Technolog para o ‘AppU’ e o ‘AppV’.

[1] https://allurereport.org/

5.3.2 PROCESSO DE AUTOMAÇÃO

Após a conclusão dos estudos e definições das seções anteriores, deu-se início ao processo de mapeamento das interfaces dos aplicativos selecionados utilizando o *driver* de automação *UiAutomator2*[2], comum a todas as aplicações Android. Esse *driver* possibilita a captura de telas (*snapshots*) fiéis ao estado atual das aplicações, bem como a obtenção dos atributos individuais de cada componente presente nas capturas. A Figura 5.1 ilustra o resultado obtido por meio dessa ferramenta, incluindo a marcação dos componentes para fins de mapeamento.

Figura 5.1 – *Snapshot* Aplicativo 'U'



Após a finalização do mapeamento das aplicações, foram verificadas as funcionalidades correspondentes a cada tela e seus respectivos elementos mapeados, criando, assim, cenários utilizando o padrão *Gherkin* para a interpretação do Cucumber. Para cada cenário, foram geradas massas de testes por meio de consul-

[2] https://github.com/appium/appium-uiautomator2-driver

tas dinâmicas ao banco de dados de cada cliente previamente selecionado. Essas consultas permitiram a seleção de candidatos para os testes que seriam realizados nos cenários.

Com posse desses candidatos, foi implementada uma lógica de conversão do conteúdo do arquivo contendo a massa de testes, alterando os seus delimitadores para o padrão de delimitação de uma tabela de Exemplos do *Gherkin*. Cada linha dessa tabela contém um caso e o seu resultado esperado, conforme exemplificado no Código 5.1.

```
1  #language: pt
2
3  Funcionalidade: Login
4
5    Esquema do Cenario: Realiza login com sucesso
6      Dado que estou na tela de login
7      Quando preencher os dados dos campos login "<login>" e senha "<senha>" corretamente
8      E aceitar as politicas de privacidade
9      E clicar em entrar
10     Entao devo ter acesso a tela inicial da aplicacao com o email "<email>"
11
12     Exemplos:
13       | login  | senha       | email    |
14       | login1 | minhaSenha1 | Email 1  |
15       | login2 | minhaSenha2 | Email 2  |
16       | login3 | minhaSenha3 | Email 3  |
```

Código 5.1 – Exemplo de Cenário com Tabela de Exemplos do *Gherkin*

Nessa *feature* que aborda a funcionalidade de *login*, para cada linha da tabela de exemplos (linhas 13 a 16), também conhecida como tabela *Gherkin*, os valores correspondentes às chaves das colunas serão atribuídos aos campos nas linhas 7 e 10. Isso permitirá a realização de assertivas em relação a esses valores quando o Cucumber fornecer os passos dos testes a serem implementados. Logo, essa abordagem facilita a reusabilidade para diversas massas de teste.

Com todos os arquivos de *features* prontos, juntamente com seus respectivos cenários e massas de teste, deu-se início à implementação das lógicas de acesso aos componentes de cada tela dos aplicativos. Utilizou-se o padrão de projeto *Page Object*, comumente empregado na automação de testes web com

Selenium, no qual se prioriza a orientação a objetos, visando facilitar a manutenção dos códigos e promover a reusabilidade de componentes (YIMEI *et al.*, 2019).

Dessa forma, os componentes e ações comuns a todas as telas foram implementados em uma classe denominada `BaseScreen`, enquanto cada componente e função específica foram colocados em suas respectivas classes `Screen`, as quais herdam os atributos e métodos da classe `BaseScreen`.

No Código 5.2, as linhas 6 a 17 representam métodos comuns a todas as telas das aplicações e, portanto, foram inseridos na classe `BaseScreen`, pois a única diferença entre eles está no tipo de ação realizada com o `MobileElement` passado como parâmetro em cada uma, como pode ser observado nas linhas 7, 10 e 13.

```
1  public abstract class BaseScreen {
2      public BaseScreen() {
3          PageFactory.initElements(new AppiumFieldDecorator(AppiumDriverHelper.getDriver(),
4                                                            Duration.ofSeconds(120) ), this);
5      }
6      public void preencherCampo(MobileElement elemento, String texto) {
7          elemento.sendKeys(texto);
8      }
9      public void clicarEmElemento(MobileElement elemento) {
10         elemento.click();
11     }
12     public String getTextoElemento(MobileElement elemento) {
13         return elemento.getText();
14     }
15     public void voltar() {
16         AppiumDriverHelper.voltar();
17     }
18 }
```

Código 5.2 – Exemplo da classe `BaseScreen`

No Código 5.3, é apresentada uma parte do trecho da classe `LoginScreen` na qual ocorre o mapeamento dos `IDs` dos elementos da tela por meio da anotação `@AndroidFindBy` (linhas 3, 5 e 7). Além disso, são definidos os *getters* para cada um desses elementos, permitindo sua utilização nos métodos da classe pai `BaseScreen` por meio da importação da biblioteca *Lombok*[3] e uso da anotação `@Getter` (linha 1).

[3] https://projectlombok.org/

```
@Getter
public class LoginScreen extends BaseScreen {
    @AndroidFindBy(id = "meuIdA")
    private MobileElement campoUsuario;
    @AndroidFindBy(id = "meuIdB")
    private  MobileElement campoSenha;
    @AndroidFindBy(id = "meuIdC")
    private  MobileElement btnEntrar;
}
```

Código 5.3 – Exemplo da classe `LoginScreen`

Finalmente, realizou-se a implementação dos passos do cenário descrito no Código 5.1, preenchendo os métodos relacionados a cada um dos passos com o uso dos componentes correspondentes da tela da funcionalidade a ser testada. Os Códigos 5.4 e 5.5 apresentam, respectivamente, os métodos da classe `LoginSteps` e uma parte da implementação desses métodos, utilizando os componentes da classe `LoginScreen`.

```
@Dado("^que estou na tela de login$")
public void queEstouNaTelaDeLogin() {...}

@Quando("^preencher os dados dos campos login \"([^\"]*)\" e senha \"([^\"]*)\" corretamente$")
public void preencherOsDadosDosCamposLoginESenhaCorretamente(String login, String senha){...}

@E("^aceitar as politicas de privacidade$")
public void aceitarAsPoliticasDePrivacidade(){...}

@E("^clicar em entrar$")
public void clicarEmEntrar(){...}

@Entao("^devo ter acesso a tela inicial da aplicacao com o e-mail \"([^\"]*)\"$")
public void devoTerAcessoATelaInicialDaAplicacaoComOEMail(String email) {...}
```

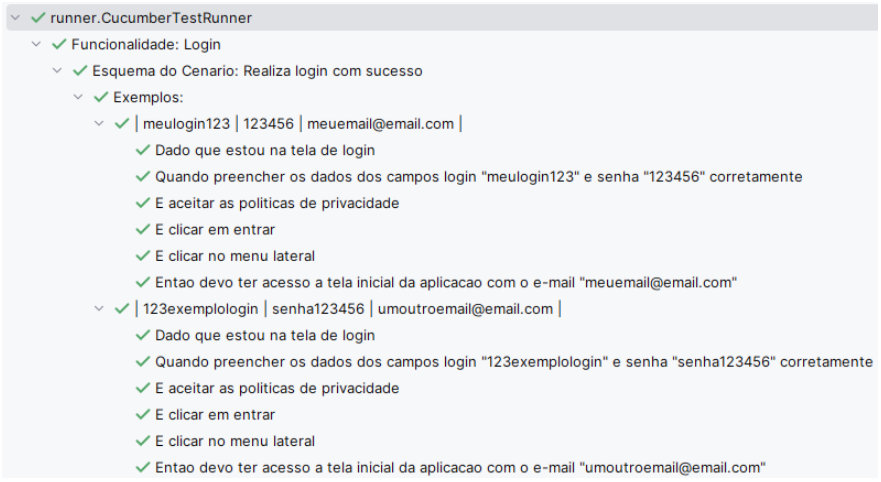
Código 5.4 – Exemplo dos métodos da classe `LoginSteps`

```java
public class LoginSteps {
    private LoginScreen screen = new LoginScreen();

    @After
    public void afterScenario() {
        AppiumDriverHelper.openApp();
    }

    @Dado("^que estou na tela de login$")
    public void queEstouNaTelaDeLogin() {
        this.screen.clicarEmElemento(screen.getCampoUsuario());
    }
    @Quando("^preencher os dados dos campos login \"([^\"]*)\" e senha \"([^\"]*)\" corretamente$")
    public void preencherOsDadosDosCamposLoginESenhaCorretamente(String usuario, String senha)  {
        this.screen.preencherCampo(screen.getCampoUsuario(), usuario);
        this.screen.clicarEmElemento(screen.getCampoSenha());
        this.screen.preencherCampo(screen.getCampoSenha(), senha);
    }
...
}
```

Código 5.5 – Exemplo de implementação dos métodos da classe `LoginScreen`

A Figura 5.2 demonstra os resultados das validações realizadas durante a execução do teste para duas das linhas de exemplos da Tabela *Gherkin*. Por motivos de confidencialidade, as informações de *login* e senha utilizadas foram omitidas.

Figura 5.2 – Resultados da execução da *feature* de *Login*



5.3.3 ADVERSIDADES ENCONTRADAS

No decorrer do processo, identificaram-se desafios devido a estruturação, coleta, armazenagem e tratamento dos dados. Ao final do processo, não foi possível alcançar em sua totalidade os seguintes tópicos devido às restrições impostas pelas tecnologias e pelas regras de negócio:

Verificação da equidade dos dados: Devido à dinamicidade dos dados e à impossibilidade de realizar tratamentos no banco de dados, não foi viável verificar a igualdade das informações exibidas no caso do 'AppU' com os resultados das requisições ao banco. Como exemplo, considere uma companhia aérea com uma frota de 500 aeronaves equipadas com sensores de movimentação e medição de volume de combustível. Suponha que, a cada trinta segundos, os sensores enviem informações sobre o *status* do tanque, localização e velocidade de movimentação. Com base nesse exemplo, prevê-se que haverá um total de 43 milhões e 200 mil envios de dados em um mês. Realizar análises estatísticas desses dados por meio de cálculos a nível de banco de dados torna-se uma tarefa onerosa. Esse cenário ilustrativo guarda semelhanças com a situação observada no 'AppU'. Especificamente para casos como o supramencionado, o processamento de métricas é realizado em um nível diferente do banco de dados.

Configurações de features: Dada a especificidade de cada cliente, tornou-se inviável a criação de *features* reutilizáveis no 'AppV'. Em muitos casos, foi necessário criar recursos exclusivos para cada cliente. Isso ocorreu porque, dependendo do cliente utilizado, certas informações e funcionalidades eram alteradas. A título de ilustração, considere o cenário no qual cada cliente possui acesso a uma lista de tarefas com formatos de preenchimento distintos. Nesse caso, já que não há controle sobre a variedade de formas de acesso aos campos dessas listas, abrangendo todas as configurações possíveis, a solução previamente mencionada, aliada à criação de

uma lista controlada que abarque os principais elementos que se diferenciam entre as diversas listas, foram as abordagens adotadas para superar essa adversidade.

Reutilização da massa de teste: Esse problema, também decorrente da dinamicidade dos dados, foi observado apenas no 'AppU'. Dependendo do momento da execução, era necessário gerar uma nova massa de testes, pois a massa anteriormente utilizada se tornava inválida. As soluções encontradas foram gerar novas massas a cada execução ou utilizar dados com *mock* que não necessariamente representam o mesmo resultado da execução com dados reais.

Essas adversidades, comumente encontradas na Área de Tecnologia da Informação, não impediram que benefícios fossem obtidos com a automação dos testes. Além disso, reforçaram a necessidade de que, mesmo com a conclusão do processo de automação, a presença do profissional responsável pela criação e manutenção dos *scripts* e casos de teste continua indispensável.

5.4 AUTOMAÇÃO DE TESTES PARA 'APPZ'

Esta seção lista as principais atividades executadas pelo estagiário no que diz respeito ao processo de automação de testes para o aplicativo, previamente selecionado, de compilação cruzada, desenvolvido com o *framework* Flutter.

5.4.1 DECISÕES DE PROJETO

Com a conclusão do processo de automação dos testes do 'AppU' e 'AppV', o estagiário deu início à tentativa de integrar os projetos de automação mencionados anteriormente com um dos aplicativos que utiliza a tecnologia Flutter, denominado aqui 'AppZ'.

Enfrentou-se como adversidade o fato de que o *driver* de automação *UiAutomator* não oferecer suporte básico para o principal identificador, a chave `key`, comumente usada nesse tipo de aplicação. Logo, os atributos `id` e

`resource-id`, conforme ilustrado na Figura 5.1 da Seção 5.3.2, não possuem valor, dificultando o acesso aos elementos que os requerem (MANOHAR, 2022).

Além da falta de identificadores, constatou-se que, devido ao encapsulamento dos elementos originalmente codificados como *widgets* na linguagem Dart, para a geração de elementos nativos das plataformas Android e iOS, determinados componentes, como botões, poderiam apresentar identificadores no *UiAutomator* ao utilizar os atributos `text` e `SemanticsLabels` do *framework* Flutter, enquanto componentes como campos de entrada de texto não apresentavam a mesma operabilidade (AMARALISA, 2022). Essa adversidade específica impossibilitou a utilização do modelo de projeto anterior que, fundamentalmente, se baseia nas estratégias de acesso aos elementos através das funções `FindById()` e `findByAccessibilityId()`.

Como medida paliativa para promover o reaproveitamento do projeto previamente desenvolvido para as aplicações nativas, foi considerada a possibilidade de transição do *driver* de automação *UiAutomator* para o *appium-flutter-driver*[4]. Essa mudança permitiu o acesso aos elementos da aplicação Flutter, entretanto, surgiram questões relacionadas à incompatibilidade de versões com outros componentes do projeto. Essa incompatibilidade poderia acarretar mudanças substanciais na estrutura, caso esses componentes precisassem ser substituídos ou removidos devido à impossibilidade de operar em conjunto com o novo *driver* escolhido.

Diante desse cenário, a decisão foi pela adoção da ferramenta de testes nativa do Flutter, em conjunto com a inclusão da dependência *flutter_gherkin*. Essa ação viabilizou a integração do *framework* com a linguagem *Gherkin*, iniciando assim o processo de automação dos testes do aplicativo 'AppZ'.

Foi alcançado sucesso na integração do *framework* Flutter com a dependência *flutter_gherkin*. No entanto, notou-se que a legibilidade alcançada na automação de aplicações nativas não foi atingida na aplicação Flutter. Além disso,

[4] https://github.com/appium/appium-flutter-driver

foi necessário criar manualmente um *runner* (executável) para os cenários com a dependência *build_runner*, devido à falta de criação por parte da dependência referente ao *gherkin*. Ao pesquisar na documentação disponível[5], observou-se a data da última atualização da ferramenta e seus exemplos de uso incompatíveis com a versão atual do Dart (mais recente) utilizada no aplicativo da empresa. Diante dessas questões, o estagiário optou por prosseguir sem a utilização da dependência *flutter_gherkin*, tendo como objetivo suprir sua principal perda: a automação de massas de testes que era possível com o uso do Esquema do Cenário.

5.4.2 PROCESSO DE AUTOMAÇÃO

No processo de automação deste aplicativo, ao contrário dos anteriores, não foi necessário mapear os *IDs* ou chaves, como são chamados no Flutter. Isso porque o desenvolvimento do aplicativo foi acompanhado desde o início pelo estagiário, que, seguindo as boas práticas de desenvolvimento e orientação a testes, prontamente realizava apontamentos dos elementos sem chaves ou fora dos padrões estabelecidos previamente. Com isso, todos os *widgets* que possibilitam a inserção de chaves estáticas foram devidamente configurados. Com esse conhecimento, foi iniciada a criação das classes de tela (*Screens*), com a generalização e abstração de seus métodos de acordo com as funcionalidades.

Inicialmente, no Código 5.6, foram definidos os métodos base da classe `BaseScreen` que são comuns a todas as suas classes filhas ou telas, representados pelas linhas (5, 9 e 13-16). Esses métodos utilizam as funções já apresentadas nas Seções 3.2.2 e 5.1. Adicionalmente, na linha 2, foi acrescentado um `WidgetTester` que representa o *driver* utilitário para interação com o aplicativo e, na linha 3, o construtor da classe que recebe obrigatoriamente esse *driver* para realizar as ações necessárias de interação.

[5] <https://flutter.dev/>

```dart
abstract class BaseScreen {
  final WidgetTester tester;
  BaseScreen({required this.tester});

  Future<void> clicar(FinderBase finder) async {
    await tester.tap(finder as FinderBase<Element>);
    await tester.pumpAndSettle();
  }
  Future<void> inserirTexto(FinderBase finder, String texto) async {
    await tester.enterText(finder as FinderBase<Element>, texto);
    await tester.pumpAndSettle();
  }
  Finder buscarPorChave(Key key) { return find.byKey(key); }
  Finder buscarPorTipo(Type type) { return find.byType(type); }
  Finder buscarPorTexto(String text) { return find.text(text); }
  String buscarTextoElemento(String key) { return buscarPorChave(key as
       Key).evaluate().single.widget as String; }
}
```

Código 5.6 – Exemplo da classe `BaseScreen` do 'AppZ'

Após a crição da classe base das demais, foi criada a classe para representação dos elementos da tela inicial de *login* do aplicativo. Em tal classe, encontram-se os campos da tela e seus respectivos métodos de acesso. No Código 5.7, nas linhas 2 a 5, foram criadas variáveis que representam cada campo da tela de *login*. Foi adicionado o modificador `late` a essas variáveis, o qual permite que elas sejam inicializadas tardiamente, evitando problemas relacionados à verificação de *null-safety* da linguagem Dart que exige que as variáveis declaradas sejam inicializadas. As linhas 7 a 12 representam o construtor da classe, enquanto as linhas 13 a 16 representam os métodos *setters* dos campos e, por fim, as linhas 17 a 19 representam o método de obtenção de texto de um *widget*.

```dart
class LoginScreen extends BaseScreen {
  late Finder campoLogin;
  late Finder campoSenha;
  late Finder checkboxPoliticas;
  late Finder botaoEntrar;

  LoginScreen({required super.tester}) {
    campoLogin = _setCampoLogin();
    campoSenha = _setCampoSenha();
    checkboxPoliticas = _setCheckboxPoliticas();
    botaoEntrar = _setBotaoEntrar();
  }
  Finder _setCampoLogin() { return buscarPorChave(const Key('exemploChave1')); }
```

```
14   Finder _setCampoSenha() { return buscarPorChave(const Key('exemploChave2')); }
15   Finder _setCheckboxPoliticas() { return buscarPorChave(const Key('exemploChave3')); }
16   Finder _setBotaoEntrar() { return buscarPorChave(const Key('exemploChave4')); }
17   String _getTextoElemento(Finder elemento) {
18     return elemento.evaluate().single.widget as String;
19   }
20 }
```

Código 5.7 – Exemplo da classe `LoginScreen` do 'AppZ'

Após a criação das classes, foi iniciada a lógica para os testes com massa de dados, a principal vantagem da utilização do *flutter_gherkin* que havia sido perdida após sua remoção. Inicialmente, foi criada uma classe específica para conter a massa de dados de teste por meio de um *array* de objetos JSON (*JavaScript Object Notation*) no formato chave-valor. Essa decisão permitiu que inúmeras chaves que caracterizam atributos utilizáveis para testes pudessem ser inseridas e acessadas facilmente. Além disso, possibilitou o acesso de forma dinâmica a seus elementos por conter atributos de tamanho e indexação de elementos por posições. O Código 5.8 exemplifica o descrito anteriormente.

```
1  class MassaDados {
2    static const massaDeDados = [
3      {
4        "login": "exemploLogin1",
5        "senha":"exemploSenha1",
6        "nome": "exemploNome1"
7      },
8      {
9        "login": "exemploLogin2",
10       "senha":"exemploSenha2",
11       "nome": "exemploNome2"
12     },
13     ...
14   ];
15 }
```

Código 5.8 – Exemplo de massa de dados para o 'AppZ'

Para a lógica de acesso aos dados de teste, foi criada uma estrutura de repetição dentro da função `group` da classe responsável pela execução dos testes. Nessa estrutura, é chamada para cada elemento do *array* a função `testWidget`, que executa o cenário em questão conforme o Código 5.9. Por questões de legibi-

lidade e manutenibilidade, as lógicas de testes em si, como acesso e interação aos elementos de determinada tela, foram adicionadas em um arquivo separado, com nomenclatura declarativa da funcionalidade testada. O Código 5.10 exemplifica essa separação das lógicas de testes.

```
void main() {
  const dados = MassaDados.massaDeDados;

  group('Funcionalidade Login', () {
    for (var i = 0; i < dados.length; i++) {
      final usuario = dados[i];
      testWidgets('Deve realizar login com sucesso para o usuário ${usuario["nome"]}',
              (tester) async {
            LoginTester loginTester = LoginTester(tester: tester);
            await loginTester.executarLogin(usuario);
            await loginTester.executarTeste(usuario);
          });
    }
  });
}
```

Código 5.9 – Exemplo da classe principal de testes do 'AppZ'

```
class LoginTester {
  ...

  Future<void>executarLogin(Map<String, String> usuario) async {
    IntegrationTestWidgetsFlutterBinding.ensureInitialized();
    app.main();
    await tester.pumpAndSettle();
    await loginScreen.clicar(loginScreen.campoLogin);
    await loginScreen.inserirTexto(loginScreen.campoLogin, usuario["login"]!);
    await loginScreen.clicar(loginScreen.campoSenha);
    await loginScreen.inserirTexto(loginScreen.campoSenha, usuario["senha"]!);
    await loginScreen.clicar(loginScreen.checkboxPoliticas);
    await loginScreen.clicar(loginScreen.botaoEntrar);
  }

  Future<void>executarTeste(Map<String, String> usuario) async {
    expect(loginScreen.buscarPorTexto(usuario["nome"]!), findsOneWidget);
  }
}
```

Código 5.10 – Exemplo da classe de testes de *login* do 'AppZ'

Como resultado da automação dos testes de *login*, tem-se a Figura 5.3, onde mais uma vez são omitidas algumas informações por questões de confidencialidade.

Figura 5.3 – Resultados da execução da *feature* de *Login* no 'AppZ'

5.4.3 ADVERSIDADES ENCONTRADAS

As principais adversidades encontradas no processo de automação de aplicações desenvolvidas com o *framework* Flutter são elencadas da seguinte maneira:

Indisponibilidade de separação de repositório: Foi necessário utilizar o repositório base da aplicação para a criação e execução dos testes, utilizando o Flutter Test. Tal escolha decorreu da indisponibilidade de acesso às telas e componentes na versão em *deploy* com a tecnologia de testes definida.

Indisponibilidade de testes em produção: Em virtude das adversidades mencionadas anteriormente e das questões elencadas na Seção 5.4.1, tornou-se inviável realizar testes automatizados na versão final da aplicação. Para efetuar tal procedimento, seria indispensável utilizar um *driver* com acesso aos componentes e atributos da aplicação final, sem a necessidade de acesso ao respectivo código fonte.

Legibilidade de código: O estagiário, que possuía maior familiaridade com aplicações desenvolvidas em Java, enfrentou dificuldades no entendimento e reprodução dos cenários ao utilizar a linguagem Dart em conjunto com o *Gherkin*. Manter a orientação a objetos, coesão e acoplamento das classes tornou-se desafiador nesse contexto. Por tais razões, foi descartado o uso do *Gherkin* na automação.

Conflito de dependências: Durante as tentativas de solução dos problemas encontrados, o estagiário deparou-se com diversas propostas de soluções em fóruns e

na documentação de ferramentas. No entanto, essas propostas frequentemente entravam em conflito com as dependências do próprio aplicativo. Ao tentar realizar *downgrade* ou *upgrade* das dependências, surgiam conflitos com a versão do Dart. Essas questões, juntamente com a indisponibilidade de separação entre o repositório de testes e a aplicação em si, foram os maiores desafios encontrados na elaboração da automação de testes.

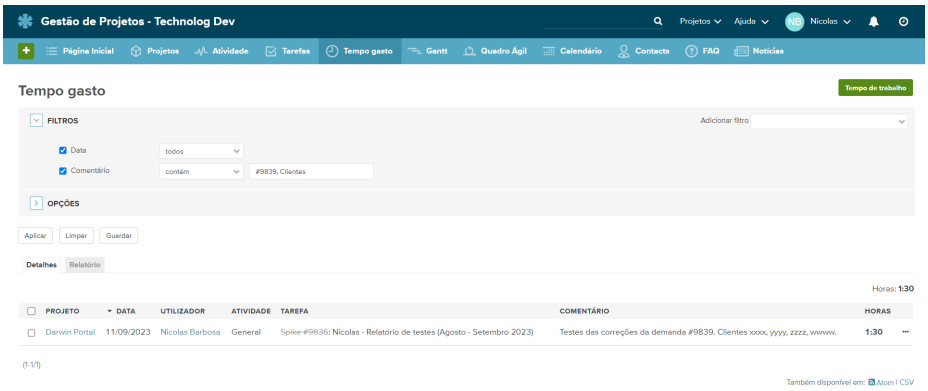
5.5 CONTRIBUIÇÕES E LIÇÕES APRENDIDAS

Como resultados do processo de automação, foram identificados alguns pontos de melhorias em relação aos testes manuais conforme apresentados:

Redução de tempo de teste: A evidência dessa otimização foi imediatamente percebida durante a aplicação do primeiro teste automatizado para um erro de acesso relatado no 'AppU'. Anteriormente, quando se realizavam testes manuais para esse tipo de relato, era comum utilizar uma amostra de teste, geralmente correspondendo a 5 a 10% do tamanho da população disponível para utilização que variava de 50 à 220 elementos por cliente. Isso se justificava devido ao excessivo tempo demandado pelos procedimentos manuais, decorrente do tamanho do conjunto de dados, das instabilidades na conexão, do tempo de carregamento das telas e da possibilidade de erro humano. Com a automação desse teste em específico, foi utilizada a totalidade da população de um dos clientes, o que demandou apenas 29 minutos. Comparativamente ao procedimento anterior, que levava de 1 a 1 hora e 30 minutos, essa otimização representou uma redução significativa de tempo. Isso se deve à análise de tempo necessário para inicialização da aplicação no emulador, que não é reaproveitada, e que consequentemente compreende uma parcela relevante desse escopo, diferentemente dos testes de aplicações web, nos quais não é necessário acessar um dispositivo virtual com sistema operacional próprio. Ao desconsiderar esse tempo, a redução de tempo se torna ainda mais evidente. Essa análise fundamenta-se nos dados registrados na ferramenta de controle interno,

empregada para o registro de tempo e documentação das atividades realizadas, tanto de testes quanto de desenvolvimento conforme Figura 5.4. Ao se comparar a capacidade de uma máquina em operar 24 horas por dia com a jornada diária média de um colaborador, que varia de 6 a 8 horas, torna-se evidente a relevância da redução do tempo de teste. Esse fator é especialmente significativo quando se considera que a máquina é capaz de realizar três a quatro vezes mais horas de trabalho que um colaborador, desconsiderando finais de semana e feriados.

Figura 5.4 – Exemplo de registro de tempo em teste manual na plataforma *plan*.io



Fonte: Autor (2024)

Identificação de pontualidades: Durante os testes manuais, era comum não identificar detalhes críticos, devido aos fatores mencionados anteriormente, como o uso de amostras para as análises. Com a capacidade de realizar um maior número de testes em um menor período de tempo, foi possível aumentar o tamanho das amostras e explorar cenários não abordados nos testes manuais habituais. Como resultado, foram identificadas questões pontuais, a exemplo do erro anterior de acesso, que afetou um usuário específico. Além disso, foi possível distinguir os problemas gerados na aplicação dos problemas gerados do servidor responsável pelo gerenciamento das conexões (APIs), especialmente em cenários nos quais parte dos testes foi concluída com sucesso e outra parte falhou. Ao analisar as causas subjacentes, constatou-se que o problema poderia estar relacionado à res-

posta fornecida pelo servidor. Em um caso específico, foi aberto um chamado para análise de um problema relatado por um usuário em particular. Nesse contexto, o usuário encontrava-se impossibilitado de visualizar os dados na tela principal da aplicação. Após uma investigação meticulosa e comparação com outras instâncias, constatou-se que o problema era singular e não decorrente de falhas na própria aplicação. Ao examinar as respostas fornecidas pela API, verificou-se que, devido a inadequações nas configurações de telemetria, o veículo utilizado pelo usuário durante o acesso à aplicação apresentava pendências de configuração, resultando na não transmissão apropriada dos dados necessários para o funcionamento da aplicação. Essas situações contribuíram significativamente para o processo de automação das verificações realizadas no gerenciador de requisições Postman[6]. Anteriormente, essas verificações eram realizadas manualmente, uma a uma. Ao final do processo, elas passaram a ser executadas em suítes com massa de teste incluindo a realização de assertivas referentes à resposta da API. Esse aprimoramento na automação proporcionou otimização e aumento de eficiência das verificações no gerenciador de requisições, além da identificação dos gargalos relacionados ao tempo de resposta das aplicações.

Ganho de desempenho da equipe: A automação permitiu que o estagiário assumisse uma carga de trabalho mais significativa no setor. Enquanto um cenário estava em execução, era possível dedicar atenção a problemas e questões que exigiam intervenção manual, como documentação, testes de caixa branca e de banco de dados. Outra vantagem notável foi a capacidade dos membros da equipe, especialmente os desenvolvedores, executarem as suítes de testes durante e após suas implementações para avaliar a qualidade de seu trabalho e sua conformidade com os requisitos das aplicações. Para isso, bastou a criação de um arquivo *README*[7]

[6] https://www.postman.com/

[7] *README* é um arquivo contendo informações úteis para compreensão e aprendizado sobre os arquivos disponibilizados em um repositório (WANG; WANG; CHEN, 2023).

com as instruções de execução dos *scripts* para que qualquer membro da equipe pudesse executá-los. Esse nível de flexibilidade e autonomia resultou em um aumento substancial da eficiência e produtividade da equipe.

6 CONCLUSÃO

Este relatório teve como objetivo fornecer ao leitor uma visão das atividades realizadas pelo autor e suas contribuições enquanto estagiário no departamento de Tecnologia da Informação da Empresa Technolog Sistemas, sediada na cidade de Lavras. Durante o período de estágio, o autor pôde aplicar os conhecimentos teóricos adquiridos ao longo de sua graduação em Ciência da Computação, especialmente nas disciplinas de Programação Orientada à Objetos, Banco de Dados, Interação Humano-Computador e Engenharia de Software. Esses conhecimentos foram utilizados no contexto de análise da qualidade e testes dos sistemas de software logísticos da empresa concedente do estágio, provendo ao estagiário a diminuição do *gap* entre teoria e prática.

Com o despertar do interesse do estagiário por aplicações móveis, foi necessário realizar um estudo nesse campo para entender as diferentes aplicações da empresa, além das ferramentas relevantes e atuais utilizadas para avaliar sua qualidade. Após essas duas atividades, o estagiário iniciou seu primeiro desenvolvimento de teste automatizado utilizando a formalização da linguagem *Gherkin* em conjunto com o *framework* Appium, o que proporcionou uma melhor legibilidade dos *scripts*, testes de massa de dados e trouxe a nível de código os critérios de aceitação para cada cenário descrito.

O estagiário enfrentou adversidades relacionadas à reutilização das lógicas implementadas para as aplicações nativas na aplicação Flutter, o que gerou questões como a impossibilidade de realizar testes na versão de produção do App Flutter, conflitos de dependências e, de forma geral, questões relacionadas à divergência de dados devido à dinamicidade do segmento da empresa. Ainda assim, houve sucesso na automação dos testes das aplicações nativas e da aplicação Flutter selecionada, tendo como principais benefícios a redução do tempo de teste, a identificação de pontos de falha e o aumento significativo do tamanho das amostras utilizadas em comparação com os testes manuais.

Durante esse processo, foi identificado que a utilização de *frameworks* mais atuais possibilita a agilidade em processos anteriormente onerosos. Esse é o caso do Flutter que, com um único código fonte, permite a geração de aplicações para as plataformas *Web*, *Desktop*, iOS e Android (GOOGLE, 2023). Esse processo, anteriormente, exigia a geração de códigos nas linguagens nativas de cada uma das plataformas. Apesar da utilidade proporcionada, foram identificadas questões relacionadas à exposição dos elementos das aplicações e seus identificadores, utilizados para sua captura e interação com o aplicativo. Nas aplicações Android nativas, os identificadores (*IDs*) foram expostos de maneira que puderam ser utilizados em conjunto com o Appium, o que possibilitou a automação de testes para as versões de desenvolvimento e produção dos aplicativos. Por outro lado, nas aplicações de compilação cruzada desenvolvidas com Flutter, houve dificuldade na manipulação dos elementos devido à não exposição de seus identificadores (*keys*). Foi necessário, então, utilizar a ferramenta própria do respectivo *framework*, o Flutter Test, para lidar com essa questão de exposição. No entanto, isso trouxe como adversidade a inviabilidade da automação de testes para a versão de produção dos aplicativos, uma vez que era necessário utilizá-lo dentro do próprio repositório com o código fonte da aplicação.

Além disso, ainda no aspecto de testes de software, foi compreendido que a prática difere da teoria ao lidar com situações reais. No caso da empresa onde o estagiário atuou, cujos cenários de testes nem sempre são controlados devido à mutabilidade provocada pelo uso de telemetria no envio e recebimento de informações, foram necessárias manutenções constantes nos *scripts* e criações de lógicas para contornar falhas de assertivas.

Por fim, a experiência do estágio na Technolog proporcionou ao estagiário – e também autor deste relatório – o aprimoramento de suas habilidades desenvolvidas ao longo do curso de graduação, além do desenvolvimento pessoal e profissional ao lidar com colegas e problemas relevantes do segmento de logística.

Espera-se que as experiências e conhecimentos registrados neste relatório possam contribuir para a realização de trabalhos semelhantes em outras aplicações, como aquelas desenvolvidas em *React Native* e *Ionic*, além de discussões acerca das limitações do Flutter no âmbito da realização de testes por meio de ferramentas genéricas como o Appium.

REFERÊNCIAS

AMARALISA. **Automating Flutter Apps with Appium Flutter Driver Using Java Client**. [*S. l.: s. n.*], 2022. Disponível em: <https://medium.com/@amaralisa321/automating-flutter-apps-with-appium-flutter-driver-using-appium-java-client-5af5e2de7cba>. Acesso em: 06 dez. 2023.

BINAMUNGU, Leonard Peter; EMBURY, Suzanne M.; KONSTANTINOU, Nikolaos. Maintaining behaviour driven development specifications: Challenges and opportunities. *In:* **25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [*S. l.: s. n.*], 2018. p. 175–184.

BOSNIC, Stefan; PAPP, Ištvan; NOVAK, Sebastian. The development of hybrid *mobile* applications with Apache Cordova. *In:* **24th Telecommunications Forum (TELFOR)**. [*S. l.: s. n.*], 2016. p. 1–4.

CHEON, Yoonsik; CHAVEZ, Carlos. Converting Android Native Apps to Flutter CrossPlatform Apps. *In:* **8th International Conference on Computational Science and Computational Intelligence (CSCI)**. [*S. l.: s. n.*], 2021. p. 1898–1904.

DA SILVA, Marcelo Moro; SANTOS, Marilde Terezinha Prado. Os paradigmas de desenvolvimento de aplicativos para aparelhos celulares. **Tecnologia, Infraestrutura e Software (TIS)**, v. 3, n. 2, p. 162–170, 2014.

DANIELSSON, William. **React Native application development – A comparison between native Android and React Native**. 2016. Tese (Doutorado) – Department of Computer e Information Science, Linköping University, Suécia.

FENTAW, Awel Eshetu. **Cross platform mobile application development: A comparison study of React Native vs Flutter**. 2020. Tese (Doutorado) – Faculty of Information Technology, University of Jyväskylä, Finlândia.

FERNANDES, Matheus; FONSECA, Samuel Tomkelski. **Automação de testes de software: Estudo de caso da empresa Softplan**. [*S. l.*], 2020. Monografia de Conclusão do Curso de Sistemas de Informação, Universidade do Sul de Santa Catarina, Brasil.

FERRERO, Alejandro. **Development of a Large-Scale Flutter App**. 2022. Tese (Doutorado) – Scuola di Ingegneria Industriale e Dell'Informazione, Politecnico Milano, Itália.

FIRDAUS, Muhammad Bambang *et al.* Agile-scrum Software Development Monitoring System. *In:* **6th International Conference on Electrical, Electronics and Information Engineering (ICEEIE)**. [*S. l.: s. n.*], 2019. v. 6, p. 288–293.

GOOGLE, Inc. **Flutter Documentation**. [*S. l.: s. n.*], 2023. Disponível em: <https://flutter.dev/>. Acesso em: 16 fev. 2023.

GOOGLE, Inc; ALLIANCE, Open Handset. **Glossário da plataforma Android**. [*S. l.: s. n.*], 2023. Disponível em: <https://source.android.com/docs/setup/start/glossary>. Acesso em: 20 fev. 2024.

HERTZ, Günter Matheus. **Desenvolvimento de aplicação *mobile* para avaliação de docentes e disciplinas**. [*S. l.*], 2022. Monografia de Conclusão do Curso de Ciência da Computação, Universidade Federal do Rio Grande do Sul, Brasil.

KAPLAN, Sinan *et al.* Driver Behavior Analysis for Safe Driving: A Survey. **IEEE Transactions on Intelligent Transportation Systems**, v. 16, n. 6, p. 3017–3032, 2015.

EL-KASSAS, Wafaa S. *et al.* Taxonomy of Cross-Platform Mobile Applications Development Approaches. **Ain Shams Engineering Journal**, v. 8, n. 2, p. 163–190, 2017.

LAMOTHE, Maxime; GUÉHÉNEUC, Yann-Gaël; SHANG, Weiyi. A systematic review of API evolution literature. **ACM Computing Surveys**, v. 54, n. 8, p. 1–36, 2021.

MANOHAR, Shoban. **How can we automate native + flutter applications?** [*S. l.: s. n.*], 2022. Disponível em: <https://medium.com/@shoban.manohar/challenges-testing-flutter-mobile-applications-1da67057d72d>. Acesso em: 06 dez. 2023.

MARCHESE, Maurizio; ZEN, Roberto; VILLAFIORITA, Adolfo. **Gherkin* and Cucumber* - A new test case path composition approach to testing ruby on rails web applications**. 2013. Tese (Doutorado) – Department of Information Engineering and Computer Science, University of Trento, Itália.

MICHAEL, Buettner *et al.* **Software Engineering is indeed in a crisis**. [*S. l.: s. n.*], 2006. Disponível em: <https://www.researchgate.net/publication/265070905_Software_Engineering_is_indeed_in_a_crisis/>. Acesso em: 20 abr. 2024.

NEVES, Jonathan; JUNIOR, Vilmar Mendes. **Uma análise comparativa entre Flutter e React Native como *frameworks* para desenvolvimento híbrido de aplicativos *mobile*: Estudo de caso visando produtividade**. [*S. l.*], 2020. Monografia de Conclusão do Curso de Ciência da Computação, Universidade do Sul de Santa Catarina, Brasil.

OLEGÁRIO, Gustavo *et al.* **Um *framework* para geração de testes automatizados para aplicações *mobile***. [*S. l.*], 2019. Monografia de Conclusão do Curso de Ciência da Computação, Universidade Federal de Santa Catarina, Brasil.

OLSSON, Matilda. **A Comparison of Performance and Looks Between Flutter and Native Applications**. 2020. Tese (Doutorado) – Department of Software Engineering, Blekinge Institute of Technology, Suécia.

OPENJS, Foundation. **Appium Documentation**. [*S. l.: s. n.*], 2012. Disponível em: <https://appium.io/docs/en/2.1/>. Acesso em: 24 ago. 2023.

OUNI, Ali *et al.* Web service antipatterns detection using genetic programming. *In:* **16th Conference on Genetic and Evolutionary Computation (GECCO)**. [*S. l.: s. n.*], 2015. p. 1351–1358.

PEREIRA, Vinícius Aielo; SILVA, Aline Aparecida Cintra. **Aplicativo *mobile* de controle de contas**. [*S. l.*], 2020. Monografia de Conclusão do Curso de Tecnólogo em Análise e Desenvolvimento de Sistemas, Faculdade de Tecnologia de Franca, Brasil.

RAHUL RAJ, C.P; TOLETY, Seshu Babu. A study on approaches to build cross-platform *mobile* applications and criteria to select appropriate approach. *In:* **9th India Conference (INDICON)**. [*S. l.: s. n.*], 2012. p. 625–629.

RISSI, Matheus; DALLILO, Felipe Diniz. Flutter um *framework* para desenvolvimento *mobile*. **Revista Científica Multidisciplinar**, v. 3, n. 11, p. 1–12, 2022.

SHAH, Kewal; SINHA, Harsh; MISHRA, Payal. Analysis of Cross-Platform *Mobile* App Development Tools. *In:* **5th International Conference for Convergence in Technology (I2CT)**. [*S. l.: s. n.*], 2019. p. 1–7.

SINGH, Shiwangi; GADGIL, Rucha; CHUDGOR, Ayushi. Automated testing of *mobile* applications using scripting technique: A study on appium. **International Journal of Current Engineering and Technology**, v. 4, n. 5, p. 3627–3630, 2014.

SIVAK, Michael; SCHOETTLE, Brandon. Eco-driving: Strategic, tactical, and operational decisions of the driver that influence vehicle fuel economy. **Transport Policy**, v. 22, p. 96–99, 2012.

SMARTBEAR, Software. **Cucumber Documentation**. [*S. l.: s. n.*], 2019. Disponível em: <https://cucumber.io/docs/guides/overview/>. Acesso em: 24 ago. 2023.

SMUTNÝ, Pavel. *Mobile* development tools and cross-platform solutions. *In:* **13th International Carpathian Control Conference (ICCC)**. [*S. l.: s. n.*], 2012. p. 653–656.

SOMMERVILLE, Ian. **Engenharia de Software**. 9a. [*S. l.*]: Pearson, 2011.

TGT, Consultant; UNICAMP. **O Futuro da Qualidade de Software**. [*S. l.: s. n.*], 2023. Disponível em: <https://lp.inmetrics.com.br/inscricao-estudo-tgt-pesquisa-qualidade>. Acesso em: 14 abr. 2024.

TUOVENEN, J; OUSSALAH, Mourad; KOSTAKOS, Panos. MAuto: Automatic *mobile* game testing tool using image-matching based approach. **The Computer Games Journal**, v. 8, n. 3, p. 215–239, 2019.

VIANA, Virginia Maria Araújo. **Um método para seleção de testes de regressão para automação**. 2006. Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de Pernambuco, Brasil.

VILLANES, Isabel Karina; BEZERRA COSTA, Erick Alexandre; DIAS-NETO, Arilo Claudio. Automated *Mobile* Testing as a Service. *In:* **11th World Congress on Services (WCS)**. [*S. l.: s. n.*], 2015. p. 79–86.

WAHLBRINCK, Kamile A; BONIATI, Bruno B. Aplicações *Mobile* Híbridas: Um Estudo de Caso do *Framework* Ionic para Construção de um Diário de Classe. *In:* **VIII Encontro Anual de Tecnologia da Informação (EATI)**. [*S. l.: s. n.*], 2017. p. 69–76.

WANG, Junmei; WU, Jihong. Research on *Mobile* Application Automation Testing Technology Based on Appium. *In:* **2nd International Conference on Virtual Reality and Intelligent Systems (VRIS)**. [*S. l.: s. n.*], 2019. p. 247–250.

WANG, Tianlei; WANG, Shaowei; CHEN, Tse-Hsun Peter. Study the correlation between the README file of GitHub projects and their popularity. **Journal of Systems and Software**, v. 205, p. 111806, 2023.

WANG, Yusi. **Review and testing of plugins in Flutter for Android and IOS**. 2022. Tese (Doutorado) – Communications and Computer Networks Engineering, Politecnico di Torino, Itália.

YIMEI, Chen *et al.* Research on Page Object Generation Approach for Web Application Testing. *In:* **31st International Conference on Software Engineering and Knowledge Engineering (SEKE)**. [*S. l.: s. n.*], 2019. p. 43–63.