Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Sistemas de Informação

Projeto e implementação de testes automatizados para aplicativos mobile: um estudo de caso no aplicativo institucional da Universidade Federal de Juiz de Fora

João Paulo Dias

JUIZ DE FORA
AGOSTO, 2022

Projeto e implementação de testes automatizados para aplicativos mobile: um estudo de caso no aplicativo institucional da Universidade Federal de Juiz de Fora

João Paulo Dias

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Sistemas de Informação

Orientador: Igor de Oliveira Knop

JUIZ DE FORA

AGOSTO, 2022

Projeto e implementação de testes automatizados para aplicativos mobile: um estudo de caso no aplicativo institucional da Universidade Federal de Juiz de Fora

João Paulo Dias

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM SISTEMAS DE INFORMAÇÃO.

Aprovada por:

Igor de Oliveira Knop
D.Sc. Modelagem Computacional

André Luiz de Oliveira
D.Sc. Ciência da Computação

Luiz Felipe Mendes
M.Sc. Modelagem Computacional

Vânia de Oliveira Neves
D.Sc. Ciência de Computação e Matemática Computacional

JUIZ DE FORA
10 DE AGOSTO, 2022

À minha mãe.

Ao meu irmão.

Aos meus amigos.

Resumo

Os aplicativos para dispositivos móveis são ubíquos na sociedade atual, utilizados para as mais diversas tarefas como comunicação, *e-commerce*, *internet banking* e para entretenimento. Independente de sua finalidade, a qualidade das funcionalidades do aplicativo é um dos requisitos que fideliza o usuário. Portanto, as atividades de teste são cruciais para garantir sua qualidade e, consequentemente, aumentar a sua vida útil. Uma preocupação das equipes de desenvolvimento é manter a disponibilidade e a qualidade do aplicativo em diferentes plataformas como Android e *iOS*, demandando cada vez mais por ferramentas de apoio à criação e execução de teste de código-fonte para diversas plataformas operacionais móveis. O aplicativo institucional da UFJF possui cerca de cinco mil usuários, atualmente, e uma nova versão em plataforma de desenvolvimento híbrido está para ser lançada. Neste contexto, este trabalho propõe e relata a implantação de um novo processo de teste no grupo de desenvolvimento. O processo foi definido e registrado ao longo de seis meses de desenvolvimento e a aceitação da equipe técnica foi avaliada por meio de questionário, bem como a cobertura de código inicial medida para posterior acompanhamento.

Palavras-chave: teste de software; dispositivos móveis; aplicativos híbridos, qualidade de software.

Abstract

Mobile applications are ubiquitous in today's society, used for various tasks such as communication, *e-commerce*, *internet banking* and for entertainment. Regardless the purpose, the quality of a mobile application is a requirement to build user's trust. Software testing activities are crucial to achieve the product quality and increase its lifespan. Software teams are concerned in mantaining both quality and availability of mobile applications across Android and *iOS* platforms, increasing the demand for techniques and tools that support the creation and execution of test cases across different mobile operating systems.The current version of UFJF institutional application has about five thousand users, and a new version in hybrid development platform is about to be released. In this context, This study reports a software testing process and its application in the development of UFJF institutional mobile application. The process was defined and reported over six months of development. We evaluated the acceptance of the proposed process with the development team through a survey, as well as an initial code coverage measured for later follow-up.

Keywords: Software testing; mobile devices; hybrid applications, software quality.

Agradecimentos

Agradeço primeiramente a minha fé de que os obstáculos podem ser superados.

À minha mãe e meu irmão pelo apoio e força para seguir nesta caminhada em uma dupla jornada de trabalho e faculdade.

Às minhas tias: Juracy e Aparecida por ajudarem na minha formação como pessoa e como estudante e me ensinarem, que apesar dos obstáculos da vida, sempre devemos tentar superá-los.

Aos meus amigos, em especial, à Fernanda, por sempre demonstrar a preocupação e o amor em forma de amizade. À Gabriele, por me apoiar nas mudanças mais arriscadas e não deixar me esquecer da minha capacidade. Ao Pedro, por não me deixar desistir e, inclusive, me ajudar para que não desistisse. Ao Rian, por todos os ensinamentos e apoio durante os dias na universidade.

Ao professor Igor pela orientação, amizade e principalmente, pela paciência, sem a qual este trabalho não se realizaria.

À professora Vânia, pelas oportunidades durante à carreira acadêmica de desenvolver a paixão por teste de software com ensinamentos com paciência e atenção.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

À equipe de servidores do CGCO e aos bolsistas do projeto do UFJFApp, Alice, Maycon e Thiago, principalmente, pelo tempo disposto para apoiar o este trabalho.

'Desistir nunca foi uma opção, mas, por muitas vezes, a desejei. Estou aqui representando muitos e muitas e digo a eles: Me formei'. *João Paulo Dias*

Conteúdo

Lista de Figuras

Lista de Abreviações e Siglas

ADB *Android Debug Bridge*. 30, 51

API *Application Programming Interface*. 33, 36, 38, 43, 68

APK *Android Application Pack*. 17, 53, 54

CGCO Centro de Gestão do Conhecimento Organizacional. 8, 11, 31, 60, 61

CSS *Cascading Style Sheets*. 18

DOM *Documento Object Model*. 19, 20, 36, 38, 39

GET-SI Grupo de Ensino Tutorial de Sistemas de Informação. 31

GPS *Global Positioning System*. 17

HTML *HyperText Markup Language*. 18–20

HTTP *HyperText Transfer Protocol*. 8, 28, 52

IDE *Integrated Development Environment*. 16

IRA Indíce de Rendimento Acadêmico. 31, 33, 64

JSON *JavaScript Object Notation*. 20, 24, 39, 40

JSX *Javascript Syntax Extension*. 19

NFC *Near Field Communication*. 17

NPDT Núcleo de Pesquisa e Desenvolvimento Tecnológico. 31

NPM *Node Package Manager*. 40

OHA *Open Handset Alliance*. 16

SDK *Software Development Kit*. 16–18

SIGA Sistema Integrado de Gestão Acadêmica. 11, 31, 33, 36

SO Sistema operacional. 17, 20

UFJF Universidade Federal de Juiz de Fora. 11, 30, 31, 33, 35, 36, 65–68

USB *Universal Serial Bus*. 30, 52

1 Introdução

1.1 Visão Geral

A computação móvel atingiu um crescimento contínuo, de modo que sua presença diária nas tarefas da sociedade é um dos maiores e mais recentes avanços da tecnologia. Em 2021, existiam 6,37 bilhões de usuários de *smartphones* no mundo. Até o final de 2022, esse número está previsto alcançar para 6,64, equivalendo a 83,37% da população mundial, conforme o site BankMyCell[1].

Esse avanço demanda que as organizações disponibilizem seus serviços por meio dos aplicativos móveis, executados nos sistemas operacionais dos aparelhos. O Android é o principal sistema operacional móvel, estando presente em 72,83% dos *smartphones* comercializados e em segundo lugar está o *iOS* com 26,35%, conforme o site StatCounter[2].

Para atender a alta demanda por aplicativos em um mercado dividido entre dois sistemas operacionais, o desenvolvimento híbrido vem ganhando adesão das empresas de desenvolvimento de software. A aderência a esse método de desenvolvimento permite o custeio e gerência de uma única equipe responsável pela criação de aplicações que funcionem nos sistemas operacionais *mobile* citados, possibilitando o aumento da velocidade de desenvolvimento e a redução de custos dos produtos (SANTOS, 2018 apud BOUSHEHRINEJADMORADI et al., 2015, p.21).

Outro aspecto que viabiliza o lucro e diminui os custos ao longo do ciclo de vida da aplicação é sua qualidade. Para identificar defeitos e apoiar a garantia à qualidade, utiliza-se o processo de teste de *software*. Esse processo possibilita que erros sejam descobertos mais cedo durante o desenvolvimento da aplicação, visto que quanto mais tarde um erro for descoberto no ciclo de vida do *software*, mais caro é corrigi-lo (MYERS; SANDLER; BADGETT, 2004; PRESSMAN; MAXIM, 2016).

[1]Empresa de compra e venda de celulares. Disponível em ⟨https://www.bankmycell.com/blog/how-many-phones-are-in-the-world#part-1⟩

[2]Serviço popular para colher métricas de páginas. Disponível em ⟨https://gs.statcounter.com/os-market-share/mobile/worldwide⟩

1.2 Contextualização

A Universidade Federal de Juiz de Fora (UFJF) disponibiliza desde 2018 um aplicativo para a plataforma Android e, mais recentemente, a versão para iOS visando possibilitar acesso do corpo universitário às informações sobre a universidade, além de permitir a obtenção de dados relacionados ao desempenho estudantil e dados da biblioteca realizando o *login* pelo Sistema Integrado de Gestão Acadêmica (SIGA). Atualmente, o aplicativo nativo se encontra instalado em cerca de cinco mil dispositivos. Ainda em 2018, iniciou-se o processo para a migração para uma versão de desenvolvimento híbrido, visando que uma única equipe consiga manter o aplicativo para os dois sistemas operacionais. Todo o processo é realizado por uma equipe de desenvolvimento composta por bolsistas de treinamento profissional, analistas de sistema do CGCO e professores.

1.3 Descrição do Problema

Como o processo é dependente de projetos de treinamento profissional, há uma grande rotatividade de bolsistas e uma demora para adaptação dos alunos às tecnologias empregadas. O processo de testes é realizado, manualmente, pela equipe de desenvolvimento. Acarretando problemas como tempo elevado de execução, tendo que ser feito em cada funcionalidade do aplicativo, sendo propenso a erros daqueles que executam os testes. Portanto, as técnicas de teste de *software* e a automação de testes não foram implementados, possibilitando a oportunidade para melhoria.

1.4 Justificativa e Motivação

A migração para o aplicativo híbrido garantiu vantagens como a criação única de um código-fonte para ambas plataformas *mobile*, economizando tempo e custo de manutenção. Agora, melhorar sua qualidade garantirá uma boa experiência com uma aplicação móvel escalável e disponível para os alunos e servidores. Isso conduz à nossa questão principal deste trabalho: como um processo de testes para aplicativos móveis, desenvolvidos na plataforma híbrida, contribui para a qualidade do software desenvolvido?

Para responder esta questão, este trabalho visa desenvolver um processo automatizado de testes de software para diminuir o custo da execução de testes que pode consumir de 30 a 60% do custo total do ciclo de vida de uma aplicação, dependendo da sua complexidade e criticidade (POLO et al., 2013) e aumentar a segurança sobre o que está sendo testado. Esse protocolo é composto por técnicas de teste estruturais e funcionais que ajudarão a garantir a qualidade no projeto cada vez mais cedo, dependendo da fase do ciclo de desenvolvimento onde eles são incluídos, pois, os defeitos podem ser encontrados mais cedo, diminuindo o custo de manutenção deles (MYERS; SANDLER; BADGETT, 2004; PRESSMAN; MAXIM, 2016).

1.5 Objetivos

Visto que este trabalho aplicará testes de software automatizados, em diferentes níveis (unitários e *end-to-end*) no aplicativo da UFJF, os seguintes objetivos específicos foram elencados:

- Caracterização dos métodos e ferramentas utilizados para testes de aplicativos híbridos;
- Avaliação do processo de desenvolvimento e revisão de funcionalidades para o aplicativo;
- Definição de um novo processo, baseado em testes para criação e revisão de funcionalidades;
- Aplicação do método em um conjunto de elementos do aplicativo, para servirem de referência;
- Aplicação de um formulário de aceitação do processo junto à equipe técnica do UFJFApp;
- Mapear os pontos de melhoria para implantação a longo prazo.

A partir dos objetivos alcançados, espera-se criar um ambiente de desenvolvimento que priorize a qualidade e possibilite a criação de um produto com custo de manutenção reduzido.

1.6 Organização do Documento

Este trabalho está organizado em cinco capítulos. Além desta Introdução, no Capítulo 2 são apresentados os conceitos necessários para a compreensão deste trabalho. O Capítulo 3 apresenta o método utilizado para a pesquisa e desenvolvimento, o Capítulo 4 detalha todas as atividades realizadas para finalizar a entrega deste trabalho de conclusão de curso. Por fim, são feitas considerações e descritas limitações que viabilizam trabalhos futuros no Capítulo 5.

2 Fundamentação

Este capítulo apresenta os estados da arte no que tange o desenvolvimento *mobile* nas Seções 2.1 e 2.2, bem como as técnicas atuais de Teste de Software para a área na Seção 2.3.

2.1 Desenvolvimento para Smartphones

O desenvolvimento de aplicações para celulares tem se tornado cada vez mais comum e popular. Dado o mercado crescente do uso de *smartphones*, empresas privadas e instituições governamentais têm o interesse de exploração da área(LECHETA, 2013). Além de ser um dos produtos mais bem sucedidos do mercado, o número de usuários de *smartphones* foi previsto em 6,64 bilhões ao final do ano de 2022.

Os celulares têm sido utilizados para diferentes fins. As funcionalidades mais comuns entre os usuários de dispositivos móveis dizem respeito a redes sociais, jogos, telefonemas, serviços de localização geográfica e, por fim, aplicações que fornecem informações, tal como bancárias, notícias e acadêmicas (NOVAC; MARCZIN; NOVAC, 2016). Os *smartphones* permitem às empresas públicas e privadas inovarem e proporcionarem aplicativos em um nicho com grande impacto e alto número de usuários (FAYYAZ et al., 2020).

Nos últimos anos, empresas de diferentes ramos têm se beneficiado do grande número de usuários de *smartphones*: Uber[3], AirBnb[4], Spotify[5] e Netflix[6] são exemplos de empresas que exploram o mercado e se beneficiaram dos serviços que oferecem via aplicativos móveis (NALKECZ, 2019). Além disso, entidades governamentais também têm se beneficiado, tal como o Governo Federal Brasileiro, por meio da iniciativa do *e-gov* (MESQUITA, 2020) que oferece uma suíte de serviços para os cidadãos. Logo, cada aplicativo, em sua área, permite um novo modo de vida a seus usuários.

[3]Disponível em: https://www.uber.com/br/en/about/
[4]Disponível em: ⟨https://www.airbnb.com/⟩
[5]Disponível em: ⟨https://www.spotify.com/⟩
[6]Disponível em: ⟨https://www.netflix.com/br/⟩

Para atender as diferentes demandas de usuários e garantir qualidade, os aplicativos para *smartphones* possuem necessidades específicas que devem ser abordadas. Entre essas estão: o uso de biometria, serviços de localização, diferentes categorias de hardware, *interfaces* gráficas específicas para cada dispositivo, compartilhamento de informações entre diversos aplicativos simultâneos, a alternância de disponibilidade de sinais de *internet*, cuidado com a privacidade de dados, entre outras. Além dos desafios específicos, necessidades gerais do software também são existentes, tal como segurança, desempenho, confiabilidade, escalabilidade, portabilidade, reusabilidade, entre outros (PRESSMAN; MAXIM, 2021).

A qualidade das aplicações mobile é um tema de interesse para diferentes atores envolvidos, desde as equipes de desenvolvimento até os usuários finais. As técnicas de Teste de Software, tal como testes funcionais, regressivos, exploratórios e de segurança, são ferramentas que auxiliam a manter e garantir qualidade dos aplicativos (TRAMONTANA et al., 2019).

Um dos grandes desafios para garantir a qualidade de software para aplicativos mobile são os diferentes sistemas operacionais que equipes de desenvolvimento precisam lidar. Atualmente, o Android e o iOS são os sistemas operacionais que dominam o mercado de *smartphones* (GOADRICH; ROGERS, 2011). Dessa maneira, o desenvolvimento para os celulares pode ser nativos, com uma base de código específica para cada sistema operacional ou; híbrido, com uma única base de código compilada para ambas as plataformas (MÜLLER; SOARES, 2018). As seções 2.2.1 e 2.2.3 descreverão mais detalhes cada uma das abordagens.

2.1.1 *iOS*

Presente em mais de 1 bilhão de aparelhos pelo mundo em 2021[7], o *iOS* é o sistema operacional criado para iPhones pela Apple em 2007. Hoje, ele está na versão 15 e, apesar de ter sido projetado para iPhones, hoje está em iPads, iPods e Apple TV que são, respectivamente, *tablets*, tocadores musicais e televisores inteligentes.

O sistema operacional da Apple teve a sua arquitetura baseada no Unix. Para

[7]Disponível em https://www.theverge.com/2021/1/27/22253162/iphone-users-total-number-billion-apple-tim-cook-q1-2021

o desenvolvimento de aplicações que executam nesse sistema operacional e utilizam os recursos do iPhone, engenheiros de software devem usar o *Software Development Kit* (SDK) da Apple. A *Integrated Development Environment* (IDE) utilizada é o Xcode[8] e as linguagens de programação que podem ser utilizadas são Objective-C[9] ou Swift[10]. Por fim, a disponibilização de aplicativos ocorre na Apple Store[11] (OLIVEIRA, 2017).

Um dos desafios ao se criar aplicações para o iOS é a restrição de conexão a aparelhos que não sejam da própria fabricante Apple. Essa abordagem de mercado da empresa visa garantir exclusividade entre seus produtos e padronizar novas funcionalidades lançadas aos seus dispositivos (SARTORELI; KUCHAUSKI, 2014).

2.1.2 Android

Originalmente, o Android foi desenvolvido pela empresa Android Inc., adquirida pela Google em 2005. Em 2021, a Google anunciou que 3 bilhões de dispositivos celulares no mundo usavam o Android, além de ser responsável por cerca de 72,83% das vendas dos dispositivos comercializados[12]. A expansão do uso do Android se dá pelo consórcio *Open Handset Alliance* (OHA)[13], formado por 84 empresas de tecnologia, cujo objetivo é aprimorar a plataforma Android, além de buscar inovações.

Sua arquitetura é baseada no *kernel* do GNU/Linux[14] com adaptações para dispositivos móveis. Por ser *open-source*, ou seja, seu código-fonte está disponível para visualização, modificação e distribuição, atraiu grande parte dos produtores de eletrônicos, como LG, Motorola, Samsung. O objetivo dessas empresas é a customização de funcionalidades do sistema operacional e o desenvolvimento de modificações, conforme a necessidade dos diferentes negócios (HECK, 2014). No momento da escrita deste texto, a plataforma está na versão 12[15], lançada em outubro de 2021 e com contínuo trabalho para a próxima maior versão, sem sinais de desaceleração.

No Android, o SDK é formado principalmente pela linguagem de programação

[8]Disponível em: https://developer.apple.com/xcode/
[9]Disponível em https://developer.apple.com
[10]Disponível em ⟨https://developer.apple.com/swift/⟩
[11]Disponível em https://www.apple.com/app-store/
[12]Disponível em ⟨https://bit.ly/over3bilion⟩
[13]http://www.openhandsetalliance.com
[14]⟨https://www.gnu.org/gnu/linux-and-gnu.pt-br.html⟩
[15]https://developer.android.com/about/versions/12

Java [16]e o SDK é o Android Studio [17]. Existe ainda a alternativa de desenvolvimento de aplicações utilizando outra linguagem de programação, a Kotlin[18]. O aplicativo é disponibilizado e instalado a partir da loja Google Play [19], também é possível gerar o aplicativo em formato *Android Application Pack* (APK) e instalar diretamente no dispositivo móvel facilitando os seus testes.

2.2 Métodos de Desenvolvimento

As aplicações *mobile* vêm ganhando evidência nos últimos anos e muitos investimentos da indústria e da academia têm sido realizados para permitir a entrega de novas aplicações, além da manutenção e garantia de qualidade de todas. Para atender as diferentes necessidades por software, algumas categorias de desenvolvimento de aplicações mobile podem ser praticadas: nativo, *WebView* e híbrido (AHMAD et al., 2018).

2.2.1 Desenvolvimento Nativo

O desenvolvimento nativo é o primeiro dos métodos utilizados para o desenvolvimento de aplicativos mobile. Esse método consiste em utilizar o SDK específico de cada Sistema operacional (SO) móvel, fornecido pelo mantenedor do sistema. Esse conjunto de *software* é composto por ferramentas e linguagens de programação apropriadas para gerar um aplicativo executado no celular com determinado SO. Esse modelo de desenvolvimento permite o acesso completo às funcionalidades de *hardware* do dispositivo, como câmera, *Global Positioning System* (GPS), bússola e *Near Field Communication* (NFC) (MÜLLER; SOARES, 2018).

Dentre as vantagens obtidas com o desenvolvimento nativo estão o alto desempenho e o uso de recursos nativos do dispositivo devido ao acesso direto ao *hardware*. Apesar dessas vantagens, há a demanda por trabalhadores especializados em cada um dos SDKs necessários para o desenvolvimento nativo. Manter duas equipes pode aumentar o tempo de desenvolvimento do aplicativo para uma ou ambas as plataformas Android e

[16]Disponível em: ⟨https://www.java.com/en/⟩
[17]Disponível em ⟨https://developer.android.com/studio⟩
[18]Disponível em ⟨https://developer.android.com/kotlin⟩
[19]https://play.google.com/store/games

iOS. Isso pode inviabilizar o projeto, dado o custo de manter dois grupos com profissionais capacitados para cada um dos SDKs (TOLEDO; DEUS, 2012).

2.2.2 Desenvolvimento Webview

Para diminuir o custo e o tempo da criação de aplicativos para ambas as plataformas móveis, algumas iniciativas para tornar o desenvolvimento agnóstico à plataforma foram criadas, como o *PhoneGap.* que se tornou o Apache Cordova [20], um *framework open-source* gratuito criado pela Nitobi, empresa adquirida pela *Adobe* em 2011[21].

Seu funcionamento utilizava o navegador *web* disponível aos desenvolvedores nativamente, ou o *WebView* para executar *websites* internamente a uma aplicação padrão, desenvolvida em código nativo de cada plataforma. O desenvolvedor do aplicativo, no que lhe concerne, o constrói usando *HyperText Markup Language* (HTML), *Cascading Style Sheets* (CSS) e *Javascript* [22] em um único projeto que pode ser distribuído para ambas plataformas Android e iOS. Porém, como desvantagem, devido ao uso de um aplicativo intermediário para utilizar as funções nativas do aparelho, o tempo de execução era lento (BEZERRA; SCHIMIGUEL, 2016).

2.2.3 Desenvolvimento Híbrido

A abordagem híbrida utiliza um único SDK para desenvolver um aplicativo para múltiplos sistemas operacionais. Com isso, há uma maior eficiência no desenvolvimento do aplicativo e no seu desempenho, pois um único código-fonte é responsável por gerar executáveis que funcionarão tanto no Android quanto no *iOS* e não há um aplicativo intermediário realizando a comunicação com o *hardware*. Sendo, então, uma alternativa para ampliar a gama de usuários de modo mais barato e em menos tempo (ARAUJO, 2019).

Atualmente, há bibliotecas que possibilitam o desenvolvimento híbrido, Santos (2018) elencou os principais *frameworks* para desenvolvimento de Aplicativos móveis, dentre eles, o *Xamarin*[23], *framework* mantido pela Microsoft, apresentando tipo de aquisição

[20]Disponível em ⟨https://cordova.apache.org/⟩

[21]Disponível em https://www.businesswire.com/news/home/20111003006347/en/Adobe-Announces-Agreement-to-Acquire-Nitobi-Creator-of-PhoneGap

[22]Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[23]https://dotnet.microsoft.com/en-us/apps/xamarin

pago para empresas e gratuito para desenvolvedores individuais, tornando uma opção mais custosa para empresas. Já o React Native, que há especial interesse neste trabalho e será detalhado a seguir, possui a licença gratuita e *open-source*, tornando-se atrativo para ser utilizado pelos desenvolvedores individuais e empresas pelo mundo.

React

Para falar sobre as ferramentas de desenvolvimento híbrido, é necessário explicar alguns conceitos relacionados. Um deles é sobre o React, também conhecido como *React.js*. Ele é uma biblioteca JavaScript desenvolvida pela Meta[24] e publicado como *open-source* em 2013, visando a fácil criação de *interfaces* web. A motivação de sua criação foi a necessidade da Meta em lidar com *interfaces* que realizavam a atualização dos dados depois de um determinado tempo (DANIELSSON, 2016).

Quando a página *web* é carregada no navegador, um *Documento Object Model* (DOM) é criado contendo aquela página *web*. O DOM é a representação, em forma de uma árvore, dos elementos HTML e seus estados. Quando uma ação é realizada pelo usuário, como a submissão de um formulário, os dados são enviados ao servidor e a página é recarregada por completo, logo o DOM também. Para evitar esse comportamento custoso, o React utiliza o conceito do DOM Virtual, responsável por renderizar apenas os elementos que tiveram seu estado modificado, como no caso do formulário e com o uso de componentes React é possível diminuir o retrabalho com código. (DANIELSSON, 2016).

Componentes e JSX

Componentes React são como blocos de construção, funções JavaScript que recebem como parâmetro o estado da aplicação e dados chamados *props*, quando esse estado é modificado, todos os componentes são notificados e caso haja um dado atualizado, o componente renderiza novamente, modificando apenas parte do DOM (DANIELSSON, 2016).

Os componentes são criados utilizando *Javascript Syntax Extension* (JSX)[25], sendo uma representação de objetos baseada em XML, facilitando a leitura da estrutura

[24] Conglomerado de empresas responsável pelo *Facebook*, *Instagram* e *WhatsApp*, dentre outros. ⟨https://about.facebook.com/⟩

[25] Disponível em: ⟨https://pt-br.reactjs.org/docs/introducing-jsx.html⟩

de árvore dos componentes React (DANIELSSON, 2016).

Cada componente pode apresentar uma estrutura DOM ao ser renderizado, por isso, ela pode ser utilizada para realizar asserções acerca do dado que é demonstrado em seu interior. Assim, a cada modificação do DOM do componente, é possível gerar um arquivo com a árvore de componentes em diversos formatos, incluindo em *JavaScript Object Notation* (JSON) (DANIELSSON, 2016). Esse arquivo é chamado *snapshot* sendo utilizado em testes dos componentes React, como será demonstrado no Capítulo 4.

React Native

O React Native foi introduzido em 2015 pela Meta ao mundo do desenvolvimento mobile, um framework cujo objetivo era revolucionar a forma que as aplicações *mobile* eram criadas. Com o propósito de simplificar o desenvolvimento para diferentes plataformas, o React Native utiliza a tecnologia já empregada no React para criar componentes, mas com um foco diferente, em vez de elementos HTML, o DOM virtual é composto de elementos nativos do SO. A aplicação é escrita em uma única linguagem e o código é transformado em componentes nativos de cada plataforma *mobile* por *JavaScript engines*, o JavaScriptCore no *iOS* e o V8 no Android que renderizam a aplicação, conforme demonstrado na Figura 2.1 (DANIELSSON, 2016).

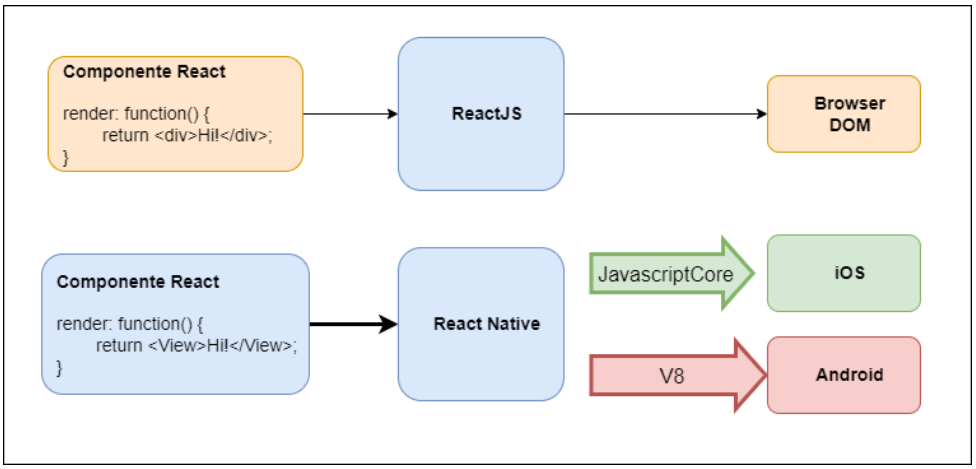

Figura 2.1: Renderização no React e no React Native (EISENMAN, 2016)

Essa forma de renderização é conhecida pela Meta como "Aprenda uma vez, escreva em qualquer lugar". Com um único código, a aplicação é executada em diferentes SO. aumentando a produtividade e diminuindo o custo de criação do aplicativo

(DANIELSSON, 2016).

2.3 Qualidade e Teste de Software

A eficiência no desenvolvimento de uma aplicação também é medida pela sua qualidade ou conformidade aos requisitos funcionais e não funcionais estipulados. Quando há uma não-conformidade ou defeito em um aplicativo móvel, pode significar perda de usuários em escala global e prejuízos financeiros, além de má reputação. Para evitar incidentes, cada vez mais deve-se investir em ferramentas e metodologias que permitam encontrar defeitos na concepção do aplicativo, evitando que os custos aumentem (PRESSMAN; MAXIM, 2016).

Assim, para revelar a presença de defeitos no *software* e ajudar a garantir a qualidade, utiliza-se o processo de teste de *software* o mais cedo possível no ciclo de desenvolvimento. Já que quanto mais tarde um defeito demorar para ser descoberto , mais caro é corrigi-lo (MYERS; SANDLER; BADGETT, 2004).

Após a execução dos testes, caso não seja encontrado erros, não há garantia que a aplicação é livre deles, mas haverá confiança que suas funcionalidades testadas estão funcionando como o esperado e os usuários terão segurança na aplicação (PRESSMAN; MAXIM, 2016).

Portanto, ao desenvolver um aplicativo, o processo de teste se faz necessário e não pode deixar de existir em seu ciclo de vida. Porém, por si só, a atividade de testes é um desafio para a engenharia moderna, no contexto de desenvolvimento móvel, há ainda mais desafiadora, devido ao variado número de dispositivos móveis no mercado e diferentes sistemas operacionais e o ambiente móvel, tornando a criação de planos de teste robustos para as aplicações em uma tarefa complexa (ALBIERO, 2017).

Na área de testes de *software*, deve-se conceituar pontos relacionados a defeito, erro e falha, além de níveis de testes, seus tipos e técnicas.

2.3.1 Defeito, Erro e Falha

Todo o processo de desenvolvimento do software pode estar suscetível a erros. Os erros podem ocorrer em diferentes momentos e em diferentes níveis, seja nas fases de elicitação de requisitos ou em fases de desenvolvimento. Todavia, independentemente da fase em que ocorra, o erro acaba afetando a qualidade do produto final que será entregue (PRESSMAN; MAXIM, 2021). Nesse cenário, alguns conceitos são importantes para a garantia da qualidade do software.

O defeito está ligado a um processo ou a um passo que ocasiona a definição de um dado incorreto num sistema. Em um sistema computacional, esse defeito pode se manifestar por uma instrução ou comando incorreto via código-fonte inserido no programa (DELAMARO; JINO; MALDONADO, 2013).

Todavia, além do defeito, existe o erro. O erro está ligado a diferença entre um valor que deveria ser o correto e esperado, e o valor que um software emite. Nesse caso, essa diferença entre esses dois valores caracteriza a presença de um erro. Comumente, um defeito ocasiona um erro no software (DELAMARO; JINO; MALDONADO, 2013).

Por fim, a falha é a inabilidade do sistema ou componente realizar a função requerida, considerando as questões de desempenho exigidas (DELAMARO; JINO; MALDONADO, 2013). A Figura 2.2 demonstra esses conceitos e os universos em que se manifestam.

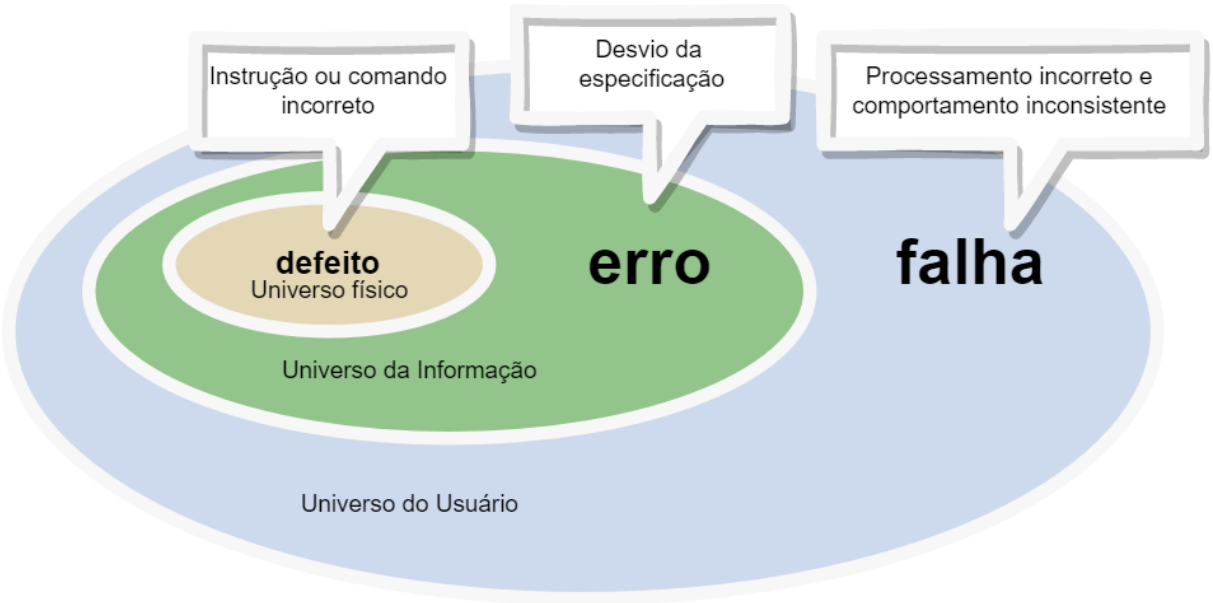

Figura 2.2: Defeito, erro e falha em seus universos

Dessa maneira, em suma, o Teste de Software é uma atividade dinâmica que exercita o software, descobrindo defeitos ou inadequações, analisando os dados de saída e observando anomalias. A realização de testes em um software tem por finalidade descobrir defeitos que podem ocasionar um erro durante a execução do programa, podendo ainda levar a uma falha. A descoberta destes defeitos durante o processo de desenvolvimento de software e não apenas no produto final tende a produzir softwares de maior qualidade e também com custos menores. Pois, as alterações podem ser realizadas ainda no processo de desenvolvimento e isso diminuirá a necessidade de retrabalho, diminuindo os custos de uma manutenção realizada posteriormente (SOMMERVILLE, 2020).

2.3.2 Níveis de Teste

Os níveis de testes são relacionados com o nível de desenvolvimento do software, possuindo objetivos específicos a serem alcançados. Dentre os níveis existentes, no trabalho, dois são utilizados: testes de unidade e teste de sistema, baseados na referência Muller et al. (2019), porém não é consenso que esses termos sejam os mesmos utilizados academicamente (BARBOSA et al., 2000).

Testes de unidade

São responsáveis por testar as menores partes do software onde o objeto ou artefato que será testado é a estrutura da aplicação, suas classes, componentes, linhas de código, fluxo de dados e de controle (AHMED; SADATH; NAGARIA, 2019).

Buscam reduzir riscos, verificar comportamentos funcionais e não funcionais da unidade sob teste conforme a especificação, construir confiança na qualidade dos artefatos. Além de evidenciar defeitos, evitando que se espalhem para níveis mais altos de teste. São automatizados e de rápida execução, diminuindo o tempo de resposta se alterações no artefato impactaram artefatos existentes (MULLER et al., 2019).

Dentre os *frameworks* para automatizar testes unitários, há o *JUnit*[26] do Java, *xUnit*[27] do C#, *unittest*[28] do Python e no Javascript há o *Mocha*[29] e, de particular

[26]https://junit.org/junit5/
[27]https://xunit.net/
[28]https://docs.python.org/3/library/unittest.html
[29]https://mochajs.org/

interesse deste projeto, o *Jest*[30], criado pela Meta. Sua escolha foi visando a simplicidade na instalação, execução e apresentação de resultados, ele possibilita também análises de cobertura de código, ou seja, qual a porcentagem de linhas de código da aplicação estão cobertas por testes, facilitando a criação de testes para fluxos de decisão. E também é indicado para testar aplicativos em React Native e seus componentes.

Também devem ser testados ao nível unitário os componentes React, visando verificar o comportamento dos mesmos quando uma interação ocorre, tal como cliques de botões ou gestos executados nas telas dos dispositivos. Soma-se a essa categoria de teste, a verificação da renderização do componente, por exemplo, a verificação de atributos como cor, ícones dependendo do estado informado ao componente ou validações de formulários (ANTONIO, 2015).

Antonio (2015) defende que os testes unitários para componentes React são muito importantes, dada a quantidade de componentes existentes e as diversas customizações que são realizadas durante as fases do desenvolvimento de software. Ainda segundo o autor, esses testes buscam garantir maior segurança, confiabilidade e qualidade para as aplicações que serão desenvolvidas.

Dentre os frameworks para testes de componentes React, existe o React Testing Library[31] e Enzyme[32] para ReactJS. O Enzyme é uma ferramenta de código aberto para React que busca facilitar a execução dos testes. Além de ter sido criada pelo AirBnb[33], ele permite a criação de componentes e simula o resultado da renderização do DOM do componente. Esse resultado é exibido em formato JSON e permite a verificação do *snapshot* criado pela simulação (PAUL; NALWAYA, 2019).

Testes de Sistema

Os testes de sistema são responsáveis por validar se as funcionalidades estão sendo executadas como o esperado em dispositivos móveis ou em emuladores. Quando comparados às outras técnicas de teste, eles apresentam grandes desafios, como grande tempo gasto para criação, tempo de execução, além de problemas com características dos simuladores

[30] https://jestjs.io/
[31] https://testing-library.com/docs/react-testing-library/intro/
[32] https://enzymejs.github.io/enzyme/docs/installation/
[33] https://airbnb.io/projects/enzyme/

e dos dispositivos celulares, (SILVA, 2019).

Todavia, os testes de sistema são importantes no ciclo de vida do software, dada a simulação do comportamento do sistema que executam. O foco desses testes está nos requisitos funcionais e não funcionais. O principal objetivo é demonstrar possíveis defeitos que os usuários sentirão, além de apontar as características que prevalecem do software ao ser utilizado (FACEBOOK, 2021).

Dentre os *frameworks* para testes de sistemas, há o *Selenium*[34] para testes web. Para testes mobile, o UI Automator[35] é utilizado no Android e o XCuiTest[36] para iOS.

2.3.3 Tipos de Teste

Conforme Muller et al. (2019) diz, tipos de Teste são atividades de teste destinadas a testar características de um software ou parte dele com base em objetivos específicos. Esses objetivos podem ser: avaliação de características funcionais como integridade, correção e adequação, tendo os requisitos funcionais como fonte do que é esperado; avaliação da estrutura ou arquitetura da unidade do sistema se está correta, completa e especificada. Sendo utilizados testes funcionais e de caixa branca para avaliar cada objetivo respectivamente.

Testes de Caixa Branca

Os testes de caixa branca avaliam a estrutura interna do sistema, o que pode incluir código, arquitetura, fluxos de trabalho e fluxo de dados. Sua eficácia pode ser medida através da cobertura estrutural, sendo a extensão em que um elemento estrutural foi testado sendo expressa como a porcentagem do elemento a ser coberto, como linhas de código, fluxos condicionais exercitados (MULLER et al., 2019).

No nível de testes de unidade, a cobertura de código trata de um conceito que analisa a quantidade de código de um software que está sendo executado em um ambiente dinâmico de análise. A cobertura de código pode ser medida em diversas dimensões, como

[34]https://www.selenium.dev/
[35]https://developer.android.com/training/testing/other-components/ui-automator
[36]https://appium.io/docs/en/drivers/ios-xcuitest/

o grau de instrução coberto pela automatização, quantidade de linhas, blocos, funções ou classes. Todavia, o principal objetivo da cobertura de código é garantir que a maioria possível da aplicação está sendo monitorada e tendo comportamentos inadequados expressos (RAHMANI; MIN; MASPUPAH, 2020).

Em aplicações o conceito de cobertura de código está relacionado a exercitar a trechos de código, visando executar simular diferentes comportamentos do código, envio de mensagens curtas e outras características presentes nos dispositivos, como abertura de câmeras. A cobertura de código deverá abranger as funcionalidades principais dos sistemas e permitir que sejam testadas em diferentes níveis (HUANG et al., 2015).

A modelagem e execução do teste caixa-branca exige habilidades ou conhecimentos acerca de como o código é construído, conceitos de lógica de programação e orientação a objetos, por exemplo, são necessários para entender qual o objetivo daquela unidade. Ferramentas podem auxiliar a demonstrar os fluxos alternativos que devem ser testados (MULLER et al., 2019).

Testes Funcionais

Conforme o (MULLER et al., 2019), um sistema deve possibilitar que os usuários executem funcionalidades durante seu uso, para isso, durante a fase de ideação do aplicativo são elicitados requisitos funcionais que devem ser seguidos para que o usuário seja atendido. Assim, testes funcionais consideram o resultado esperado de um determinado requisito funcional a partir de uma determinada entrada, caso o resultado concorde com o esperado, o teste exercitou aquela funcionalidade e garantiu que para aquele conjunto de entradas, o requisito é atendido. Caso isso não ocorra, há um defeito que deve ser corrigido.

A eficácia dos testes funcionais pode ser medida através da cobertura funcional, onde a partir de uma funcionalidade são criados testes que exercitam conjuntos de entradas e saídas válidas e inválidas. Esses conjuntos são baseados nos requisitos funcionais e após a execução, é avaliado se todos os conjuntos pertinentes foram cobertos (MULLER et al., 2019).

A modelagem e execução de testes funcionais exigem habilidades ou conhecimentos acerca dos problemas de negócio que o software visa resolver ou o papel específico

que ele desempenha (MULLER et al., 2019).

2.3.4 Técnicas de Teste

Técnicas de teste ajudam a identificar as condições de teste e os dados que serão utilizados neles. Elas auxiliam o testador a obter melhores resultados do esforço de teste. O uso dessas técnicas nas atividades de teste dependem do contexto em que o projeto está inserido, incluindo a maturidade do processo de desenvolvimento e de testes, restrições de tempo, ferramentas disponíveis, complexidade do componente ou sistema sob teste (MULLER et al., 2019). Dentre essas técnicas, há aquelas com foco em testes estruturais, conhecidas como técnicas de teste de caixa branca baseadas na análise da arquitetura, do detalhamento do projeto, da estrutura interna ou o código do objeto de teste. Na literatura, duas técnicas indicadas são a cobertura de instruções e cobertura de decisão (MULLER et al., 2019).

A cobertura de instruções analisa quantas linhas do código são exercitadas durante o teste e, a partir dessa informação, mede a cobertura dividindo o número de instruções executadas pelos testes pelo número total de instruções existentes. Já a cobertura de decisão analisa as decisões presentes no código. Para realizar isso, são mapeados os fluxos de controle e os valores que podem assumir. Por exemplo, como uma instrução `if` que pode assumir um valor verdadeiro ou falso, a partir desse resultado, um fluxo alternativo do código pode ser executado, sendo necessário cobrir esses cenários para que a cobertura de decisão seja maior e diferentes cenários sejam exercitados. Neste trabalho, elas serão necessárias para definir quais testes serão realizados para cobrir unidades do sistema, como classes de negócio e componentes React Native. As técnicas utilizadas serão a cobertura de instruções e cobertura de decisão (MULLER et al., 2019).

Mocks e *stubs*

Testes de unidade devem responder rapidamente e não depender de informações de outras unidades do sistema, visto que a unidade que está sendo testada e não a integração entre unidades. Portanto, para evitar que haja a dependência externa ou uma configuração lenta que possa afetar os testes, utilizam-se instäncias que imitam as chamadas e suas respostas,

conhecidos como *mocks*. Isso permite que o código seja executado com a informação necessária para sua execução. Desta forma, é possível que ele execute e conte quantas vezes um método foi chamado, se uma propriedade foi alterada ou não (KADU, 2021).

Outro conceito é o de *stub*, sendo uma rotina responsável por, a partir de uma entrada pré-definida de dados, retornar um resultado específico ao ser chamada. Então, com isso é possível testar possibilidades de erro, exceções sem alterar componentes externos (KADU, 2021). Neste projeto serão utilizados tanto os *mocks* quanto *stubs* para criar o teste de comportamentos que dependem de chamadas externas ou de acesso à *internet*.

2.3.5 Automação de Testes de Sistema com Appium

Aplicativos nativos, web ou híbridos precisam de ferramentas para serem testados. O Appium [37] é um framework de código aberto, que além de apoiar, oferece uma estrutura de automação de teste para as diferentes categorias de software mobile. O Appium foi concebido para gerar *scripts* de um evento a ser testado e permitir com que equipes de desenvolvimento aumentem a qualidade dos softwares a serem gerados (SINGH; GADGIL; CHUDGOR, 2014).

O trabalho de (SINGH; GADGIL; CHUDGOR, 2014) apresenta que o Appium auxilia na qualidade do software por meio das suas características de automação. Os autores defendem que a criação de *scripts* sem a automatização pode se tornar uma tarefa difícil e que ocasiona em gasto de tempos elevados. Dessa maneira, a automação permite com que o tempo seja reduzido e os desenvolvedores foquem exatamente em problemas que possam aparecer.

Na Figura 2.3, é demonstrado como é o funcionamento do *Appium*. O script de testes realiza requisições HTTP ao servidor Appium, que a partir dos dados recebidos, decide qual o dispositivo deve ser conectado, qual ferramenta de automação deve ser chamada e quais ações serão executadas, como cliques, digitação e instalação de aplicativos no dispositivo/emulador. Após a conclusão de cada ação, registros são retornados informando se a ação foi bem sucedida e o estado atual da aplicação, permitindo a asserção

[37] Disponível em ⟨https://appium.io/⟩

sobre os comportamentos esperados.

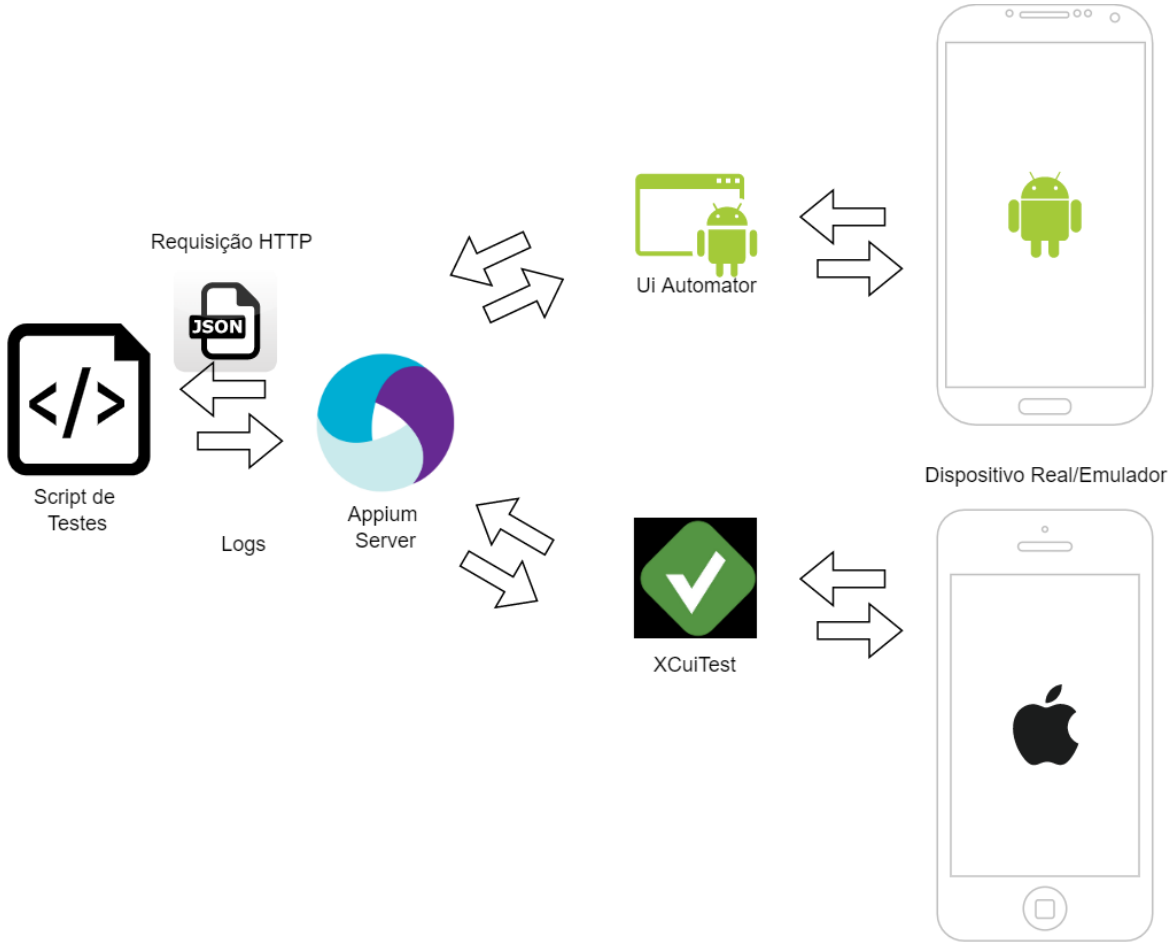

Figura 2.3: Estrutura de testes com o Appium

Os *scripts* a serem desenvolvidos com o Appium podem ser feitos em diversas linguagens de programação, entre elas Python, Ruby e Javascript. Essa flexibilidade da ferramenta permite que as aplicações *mobile* possam entregar uma maior qualidade para os usuários finais, além de permitir testes para diferentes aplicações. Além disso, os testes automatizados permitem uma maior eficiência, acurácia e custo-benefício quando comparado aos testes manuais (ALOTAIBI; QURESHI, 2017).

2.3.6 Emuladores e ADB

Os emuladores são essenciais para o desenvolvimento de aplicações híbridas. Eles consistem em ferramentas que proporcionarão à equipe de software diferentes sistemas operacionais de dispositivos móveis, sem haver a necessidade física de realmente possuir esses dispositivos. O seu objetivo é permitir que as dificuldades em se obter diferentes dispositi-

vos seja diluída, tal como permitir um desenvolvimento e testes em diferentes dispositivos (ALAMRI; MUSTAFA, 2014).

Todavia, o trabalho de (ALAMRI; MUSTAFA, 2014) apresenta desafios oferecidos pelos emuladores aos desenvolvedores e testadores, são alguns deles: O consumo de recursos computacionais, o tempo de execução e a portabilidade desses emuladores. Entretanto, os benefícios acabam sendo superiores, dadas as possibilidades e as dificuldades em se obter diferentes dispositivos.

Outra opção durante o desenvolvimento de aplicações é o *Android Debug Bridge* (ADB) [38]. A ferramenta de linha de comando é responsável por garantir a comunicação entre um dispositivo de desenvolvimento e dispositivos celulares e busca facilitar a instalação e depuração de aplicações, garantindo que o desenvolvedor e testador tenham ferramentas e recursos adequados para utilizar no projeto de software.

O ADB consiste em três componentes, um que envia comandos do computador de desenvolvimento para o dispositivo móvel utilizando um cabo *Universal Serial Bus* (USB) como conexão entre ambos; o segundo trata de um serviço que executa comando nos dispositivos e; o terceiro trata de um servidor que gerencia a comunicação entre o primeiro e o segundo. Ele é uma das principais ferramentas de desenvolvimento e uma das mais utilizadas pelas equipes de desenvolvimento para Android (MEDNIEKS; DORNIN; NAKAMURA, 2012).

2.4 Aplicativo UFJF

A UFJF começou o desenvolvimento de seu aplicativo institucional em 2015. A primeira versão publicada foi um projeto em Android nativo em 2018, e uma segunda equipe publicou a versão em *iOS* um ano depois.

As versões contam com as seguintes funcionalidades:

- Informações sobre a estrutura organizacional da UFJF;
- Informações sobre os cursos de graduação e pós-graduação;
- Acesso aos dados da biblioteca;

[38] https://developer.android.com/studio/command-line/adb

- Acesso às notícias da UFJF;
- Acesso ao Calendário Acadêmico e Serviços para a Comunidade;
- Acesso aos dados do SIGA como notas, horários de aula, histórico escolar e Indíce de Rendimento Acadêmico (IRA).

O projeto foi desenvolvido na modalidade de Treinamento Profissional pelo Núcleo de Pesquisa e Desenvolvimento Tecnológico (NPDT) do CGCO. O aplicativo conta com cerca de 5 mil usuários no Android e 450 no *iOS*. A Figura 2.4 apresenta algumas funcionalidades presentes no UFJFApp, sendo elas: login, informações para discentes, Serviços para Comunidade e Informações sobre a UFJF.

Em 2019, o desenvolvimento de uma nova versão do aplicativo teve início em um trabalho conjunto com o Grupo de Ensino Tutorial de Sistemas de Informação (GET-SI), visando ter apenas uma equipe de desenvolvimento para Android e iOS. O desenvolvimento híbrido foi realizado via React Native e mantém todas as funcionalidades básicas dos aplicativos nativos, com pequenas adaptações de usabilidade e design.

Nesse processo, as funcionalidades foram implementadas e os testes foram realizados manualmente pelos bolsistas e servidores envolvidos, exigindo maior tempo para o lançamento até que a revisão seja finalizada.

Os conceitos envolvendo as categorias de sistemas operacionais, os modelos de desenvolvimento de software mobile, assim como as técnicas e as ferramentas que apoiam os modelos de desenvolvimento, são essenciais para a proposta do trabalho que busca apresentar a implementação de testes automatizados no aplicativo mobile da UFJF.

2.4.1 Sugestão de Processo de Teste no UFJFApp

Segundo Muller et al. (2019), o processo de teste não é universal, pois depende de muitos fatores do contexto do desenvolvimento do software, alguns deles são: modelo do ciclo de vida de desenvolvimento; níveis de teste e tipos de teste considerados; complexidade do produto; restrições de orçamento.

Por isso é observado esses fatores para iniciar o desenho de uma estratégia de testes para o contexto do UFJFApp.

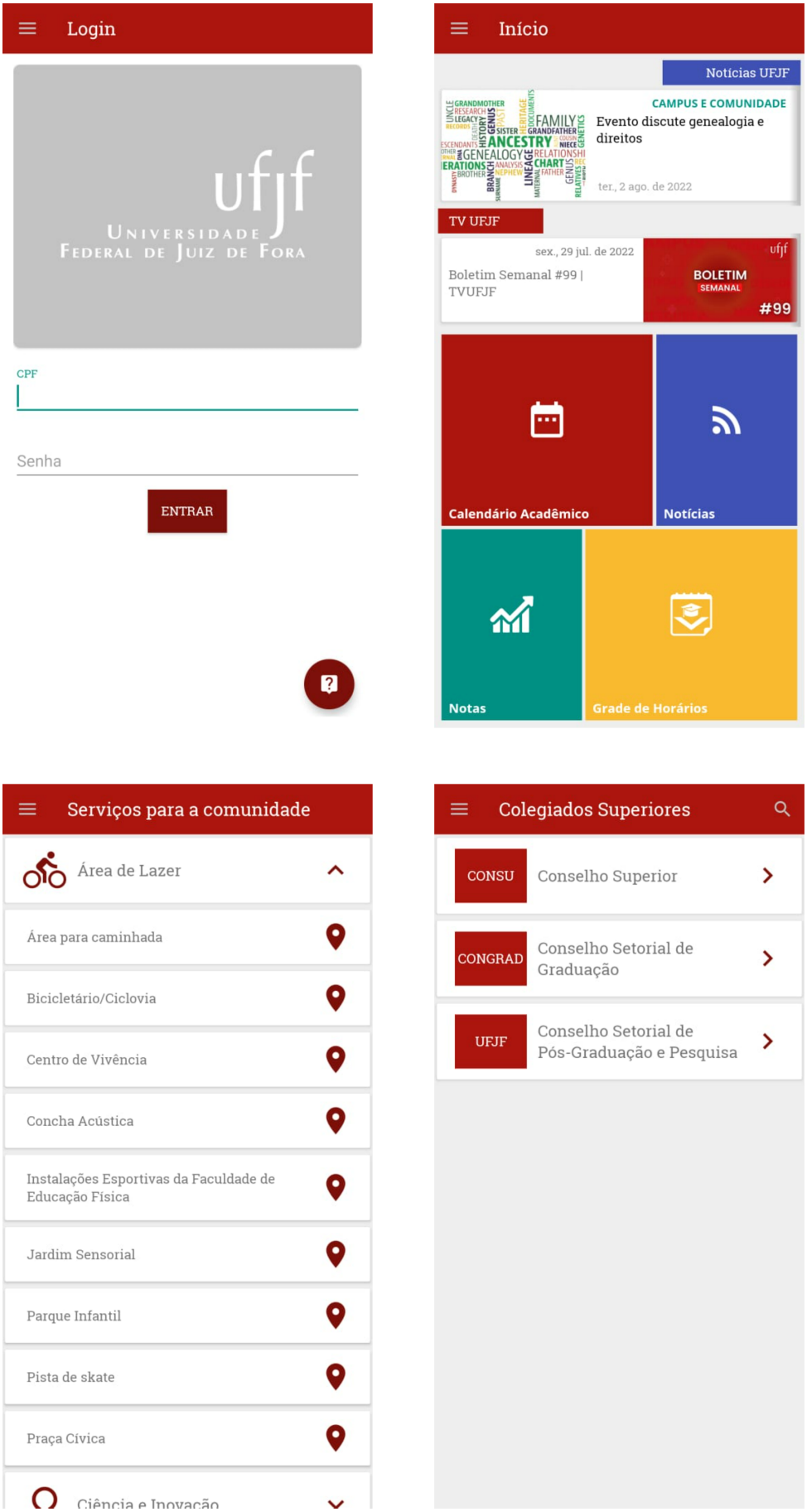

Figura 2.4: Funcionalidades do aplicativo UFJFApp

O Modelo de ciclo de vida de desenvolvimento do aplicativo UFJFApp atualmente é o Kanban. Nele, grupos de funcionalidades são especificadas, projetadas, construídas e testadas juntas em uma série de ciclos ou iterações. Cada ciclo pode incluir novas funcionalidades ou corrigir existentes, não há atualmente um tamanho fixo de tempo para o final de cada ciclo. Mas quando a iteração é finalizada a entrega é feita ao cliente. Isso pode gerar entregas mais rápidas ao usuário e exige que as funcionalidades existentes e as novas estejam funcionando como o esperado, logo, precisam ser testadas a cada iteração (MULLER et al., 2019).

Portanto, a cada finalização de um ciclo de entrega, os testes devem ser executados de forma rápida e a tendência é que o número de testes cresça cada vez mais, exigindo a utilização de ferramentas que possam automatizar esse fluxo.

Os níveis de teste que podem ser aplicados no aplicativo são os testes unitários e testes de sistema com técnicas de teste caixa-preta e caixa-branca devido à sua estrutura, a aplicação de outros tipos de testes exigem um ambiente específico de homologação.

O UFJFApp apresenta diversas funcionalidades que integram com a *Application Programming Interface* (API) do SIGA e isso adiciona uma complexidade e limitações aos testes, pois não há um ambiente para homologar as funcionalidades com dados fictícios, portanto testes das funcionalidades que necessitam de um usuário logado para visualizar as informações, como as informações de notas de disciplinas e IRA não poderão ser executados.

Como o contexto atual da UFJF envolve restrições orçamentárias e cortes, os testes não podem ser executados em dispositivos reais próprios da universidade, sendo necessário, portanto, o uso de emuladores de dispositivos ou dispositivos dos desenvolvedores e testadores.

A partir deste cenário, foi diagramado o fluxo atual de desenvolvimento, presente na Figura 2.5, onde é demonstrado que a tarefa de testes é realizada diretamente pelos servidores da UFJF e já na fase final de entrega do produto. O fluxo sugerido é apresentado na Figura 4.9 no Capítulo 4.

Portanto, o processo de teste seguirá os pontos levantados visando atingir os objetivos específicos definidos no início deste trabalho e para isso ser possível será elencada

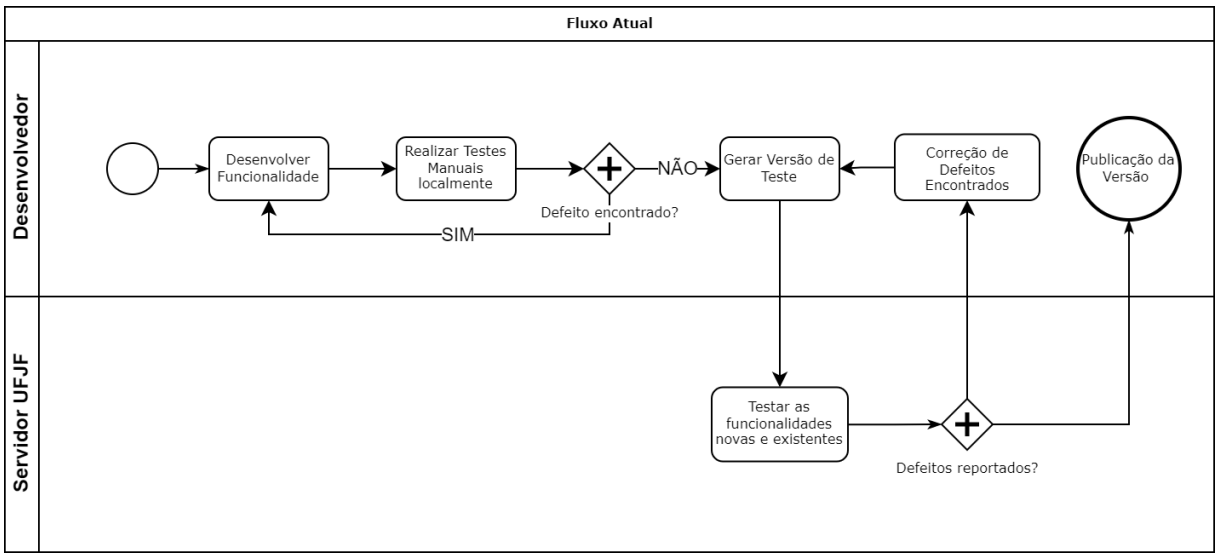

Figura 2.5: Fluxo atual de desenvolvimento do aplicativo UFJFApp

a metodologia utilizada no próximo capítulo.

Este capítulo fez uma revisão conceitual de desenvolvimento móvel, testes e qualidade de software que serão utilizados no desenvolvimento. O Capítulo 3 detalha o método utilizado para execução e avaliação deste trabalho.

3 Método

Este projeto tem o objetivo de melhorar a qualidade no processo de desenvolvimento do aplicativo institucional da UFJF. Foram desenvolvidas abordagens de teste que buscam aumentar a qualidade do código criado, esses testes serão utilizados como referência para serem aplicados por todos os desenvolvedores do projeto durante todo seu ciclo de vida.

A partir da criação do modelo de testes, foi possível criar uma documentação contendo os principais pontos explicativos sobre as abordagens criadas e será definido como objetivos a serem alcançados para que uma funcionalidade esteja pronta. Sendo, portanto, uma pesquisa de natureza qualitativa, semelhante à bibliográfica e de campo.

Para avaliar a eficiência do processo proposto foi utilizado a análise quantitativa de cobertura de código e qualitativa a partir de entrevistas e enquetes com desenvolvedores envolvidos no projeto para definir o sucesso da implementação e seus pontos de melhoria. Para alcançar esse resultado os seguintes passos foram adotados:

- Avaliar o estado atual de testes na aplicação;
- Criar testes unitários observando as camadas da aplicação;
- Avaliar a cobertura de código das classes testadas unitariamente;
- Criar os scripts de automação de testes funcionais;
- Explicar aos bolsistas o funcionamento dos scripts funcionais e solicitar que criem testes;
- Avaliar junto aos bolsistas os scripts criados;
- Coletar avaliações dos bolsistas sobre o novo processo.

Com base nos dados coletados, é esperado que o método possa ser seguido e os exemplos replicados pelos membros da equipe. Se a avaliação for positiva e a cobertura de código conseguir ser expandida, teremos indícios que os objetivos do projeto foram alcançados.

3.1 Visão Geral do Protocolo de Testes

Nesta seção é apresentada a visão geral do protocolo de testes que será aplicado no UFJFApp. Cada camada, apresentada na Figura 3.1, engloba classes que realizam ações específicas para o projeto e se integram para gerar as funcionalidades nele presente.

Na camada *Screens* está presente o núcleo do sistema, ela representa as telas do UFJFApp. Nela são instanciados os componentes com os dados obtidos da API do SIGA UFJF e ela consome todas as outras. Portanto, testes funcionais auxiliarão a validar as funcionalidades relacionadas com a obtenção de dados externos.

Já na camada *Services*, todas as classes presentes realizam acessos externos ao aplicativo, seja consumindo a API do SIGA ou autenticando dados nela, também é onde o acesso ao armazenamento do *smartphone* é realizado para salvar as credenciais obtidas da API. Será possível, utilizando testes unitários, avaliar se os valores salvos são os esperados.

Para manter os dados não sensíveis no *smartphone* e evitar constantes requisições à API, é utilizada a camada *Hooks* que apresenta a classe que representa o *Cache* do aplicativo. Nesta camada, uma possibilidade é avaliar quais dados serão utilizados para validar a memória transitória do aplicativo e quais valores são armazenados, mas como seus métodos tem efeito por todo o aplicativo, os testes podem garantir mais qualidade se feitos ao nível funcional.

Além dessas, há a camada *Components* que apresenta todos os componentes genéricos React utilizados nas Telas do aplicativo, esses componentes apresentam um comportamento que pode ser generalizado e permite que sua utilização seja em diversas telas, garantindo uma maior padronização e menor custo de manutenção. Os testes nessa camada buscarão analisar se os comportamentos dos componentes estão sendo refletidos durante o tempo de execução, utilizando a geração de seus *snapshot* e asserção sob o DOM criado.

Por fim, a camada *Business* representa os dados do usuário como CPF, *Token* de autenticação, usuário e perfis de acesso. Ela é utilizada em diversas camadas como *Hooks* e *Screens* para ser instanciada com dados obtidos após o *login* no aplicativo. Testá-la unitariamente permitirá verificar se os métodos presentes na classe estão gerando mode-

los apropriados e se as condicionais estão sendo executadas corretamente. A Figura 3.1 sumariza o protocolo de testes selecionado.

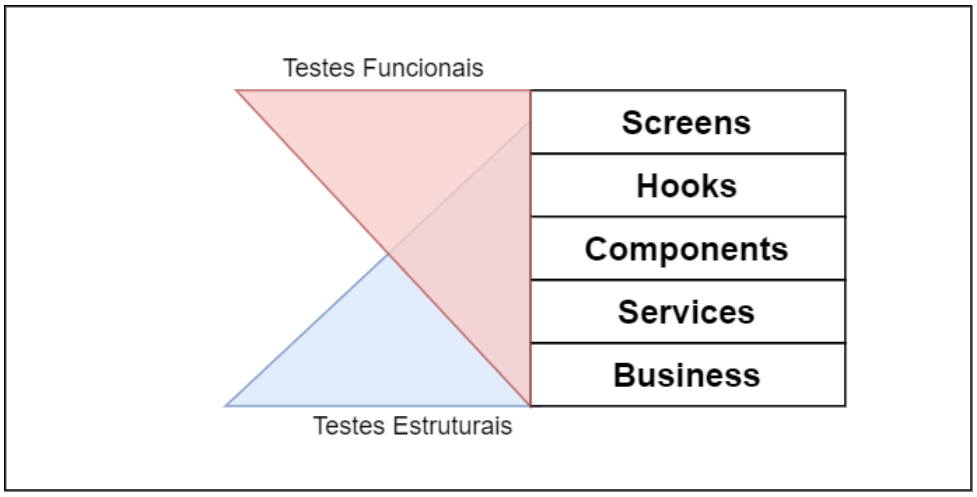

Figura 3.1: Sugestão de abordagem de testes conforme as camadas do projeto UFJFApp

3.2 Avaliação

O protocolo utilizado para avaliar a garantia de qualidade na estrutura do aplicativo será a aplicação de testes unitários em certas camadas e testes funcionais em outras, servindo como exemplo para que os desenvolvedores possam cada vez mais replicar e melhorar os conceitos aplicados nos testes em outras classes, garantindo maior cobertura de código e segurança na publicação de novas versões do UFJFApp.

Ainda, para auxiliar na garantia de qualidade dos testes funcionais, foram feitas sessões explicativas para os bolsistas do projeto sobre a importância dos testes e como automatizá-los no contexto *mobile* utilizando *Appium*. Sendo pedido, ao fim das sessões, a criação de novos testes automatizados para validar funcionalidades definidas do aplicativo.

4 Desenvolvimento

Este capítulo apresenta o desenvolvimento do protocolo de testes de unidade e funcionais do UFJFApp, bem como a avaliação de cada etapa da abordagem. O processo para executar testes estruturais foi dividido em uma sequência de passos da seguinte forma, iniciando com a Seção 4.1 apresenta a configuração dos *frameworks* de teste. Já as Seções 4.2, 4.3, 4.4 identificam os componentes que serão testados e relata as técnicas de testes caixa-branca. A Seção 4.5 traz a avaliação geral, feita pela análise da cobertura de código.

Já o processo para execução de testes funcionais automatizados foi descrito na Seção 4.6 com todas as ações realizadas e em seguida foi feita a avaliação qualitativa, junto aos bolsistas, descrita na Seção 4.7.

4.1 Configuração dos Frameworks para Testes Unitários

Esse processo consistiu em realizar a instalação dos *frameworks* necessários no repositório do aplicativo. O UFJFApp utiliza o yarn[39] como gerenciador de pacotes, ele é responsável por gerenciar as dependências do projeto, instalando ou atualizando as bibliotecas necessárias para que a compilação ocorra sem problemas. O código-fonte do UFJFApp é gerenciado em um projeto privado no *Github*, e para sua montagem é necessário utilizar o ambiente de execução JavaScript *Node.js*[40].

O projeto UFJFApp utiliza o Expo[41] como framework para desenvolvimento em React Native. É uma ferramenta que facilita no desenvolvimento de aplicativos mobile, já que ele abstrai todas as partes complexas de configuração do ambiente e permite acesso rápido e fácil a várias APIs nativas.

Os pacotes necessários instalados foram o *enzyme*, responsável pela renderização do DOM virtual dos componentes React Native e a geração de seus *snapshots*. Para isso ser possível, também foram instalados estruturas que o adaptam para o React

[39] https://yarnpkg.com/
[40] https://nodejs.org/en/about/
[41] Disponível em: ⟨https://expo.dev/⟩

Native 16, presentes no *enzyme-adapter-react-16*.

Como ferramenta para execução de testes automatizados e asserções, o *Jest*[42] foi utilizado. O arquivo de renderização do DOM de um componente deve ser em JSON para facilitar a manipulação das informações da árvore de elementos, portanto, o pacote *enzyme-to-json* será utilizado para transformar o *snapshot* do componente.

O *Jest* ainda deve reconhecer os pacotes que o sistema utiliza, para simular seus comportamentos utilizando *mocks* e compilá-los para executar os testes. Por isso um *preset* deve ser definido. Um *preset* é o framework responsável por esse reconhecimento e, no caso deste trabalho foi necessário, instalar o *jest-expo*[43].

O conjunto completo de pacotes para instalação, para o ambiente de desenvolvimento, é observado na Listagem 4.1.

Listagem 4.1: Instalação de pacotes necessários para o ambiente de testes.

```
yarn add -D enzyme enzyme-adapter-react-16 enzyme-to-json jest jest-expo
```

Para que a adaptação do *Enzyme* ao React Native 16 ocorra, foi necessário criar um arquivo de configuração dos testes na raiz do projeto, chamado `setupTests.js`. O código apresentado na Listagem 4.2, na linha 3 configura através do método *configure* importado na linha 1 do pacote *Enzyme* o adaptador importado do pacote *enzyme-adapter-react-16* na linha 2.

```
1 import { configure } from 'enzyme';
2 import Adapter from 'enzyme-adapter-react-16';
3 configure({ adapter: new Adapter() });
```

Listagem 4.2: Arquivo de configuração `setupTests.js` do *Enzyme*

Ao utilizar o *Jest*, foi necessário configurar a forma que ele é executado. Essa configuração foi feita no `package.json`, sendo um arquivo próprio do *Node.js* para definir as dependências e configurações adicionais das bibliotecas instaladas. A configuração do *Enzyme* foi adicionada ao arquivo de configuração do *Jest*, conforme é apresentado na

[42] https://jestjs.io/
[43] Sugerido em: ⟨https://docs.expo.dev/guides/testing-with-jest/⟩

Listagem 4.3.

```
1 "jest": {
2    "preset": "jest-expo",
3    "setupFiles": [
4      "./setupTests.js"
5    ],
6    "snapshotSerializers": [
7      "enzyme-to-json/serializer"
8    ]
9  }
```

Listagem 4.3: Trecho do arquivo `package.json` para configurar o *Enzyme* no Jest.

Este arquivo de configuração é descrito no formato JSON, como uma representação chave-valor de dados. A chave `jest` dessa seção apresenta cada um dos atributos de configuração do *Jest*, inicialmente o `preset` informa a configuração base do *Jest*, neste caso, foi utilizado o `jest-expo`. O atributo `setupFiles` define um conjunto de arquivos que devem ser executados antes do teste começar. Atualmente, é definido apenas o adaptador do *Enzyme* e o `snapshotSerializers` escolhe o tradutor do *Enzyme* para JSON possibilitando a realização de asserções no teste de componentes React Native. Ainda no `package.json`, na seção de *scripts*, foi adicionado os *scripts* de execução dos testes do *Jest*:

```
1  "scripts": {
2    "test": "jest --u --passWithNoTests --runInBand",
3    "test:watch": "npm test -- --watch",
4    "test:unit": "npm test -- --coverage --testMatch **/__tests__/**/*.spec.js"
5  }
```

Listagem 4.4: Trecho do arquivo `package.json` para configurar os comandos para executar os testes do Jest.

Cada um desses atributos definidos na chave `scripts` equivalem a chamadas de comandos que podem ser executados via linha de comando, todos executados através do *Node Package Manager* (NPM). Dentre eles, foram definidos os comandos que podem ser utilizados para executar o *Jest* sempre que o comando `npm test` for chamado, executando

todos os testes paralelamente. O `npm test:watch` é utilizado para realizar o teste em tempo real, cada vez que um arquivo é salvo, todos os testes relacionados são executados. Já o `npm test:unit` é utilizado para executar o comando de teste adicionando a análise de cobertura de código ao fim da execução.

4.2 Testes na Camada *Business*

Apresentando apenas a classe Credenciais, a camada *Business* possui a menor complexidade do projeto para ser testada, conforme a Listagem 4.5. Seus testes unitários são realizados para validar os valores informados a ela, garantindo que eles são atribuídos corretamente quando necessário e quando não são atribuídos.

Sua cobertura de código inicialmente estava em 0% para toda ela. Após as adições de testes, esse valor aumentou para 93,94% de cobertura de instruções e 98,33% de cobertura de decisão.

```
1  export default class Credenciais {
2
3    objCredenciais = {
4      token: "",
5      data_expira: "",
6      id_pessoa: 0,
7      nome_pessoa: "",
8      login: "",
9      profiles: [],
10     idProfileAtivo: "",
11     tokenX: {
12       access_token: "",
13       expires_in: -1,
14       date_expire_in: "",
15       scope: "",
16       profiles: [],
17       nome: "",
18       profileAtivo: "",
19       jti: "",
20       cpf: ""
21     }
22   }
23   constructor(jsonCredenciais) {
24     if (jsonCredenciais) {
25       this.setToken(jsonCredenciais);
26       this.setDadosUsuario(jsonCredenciais);
27       this.setTokenX(jsonCredenciais.tokenX);
```

```
28     }
29   }
30  }
```

Listagem 4.5: Trecho da classe `Credenciais`.

Inicialmente, a classe apresenta um construtor que chama três métodos que possuem alguns fluxos alternativos, como o método `setTokenX`, reproduzido na Listagem 4.6.

```
1  setTokenX(jsonTokenX) {
2     if (jsonTokenX) {
3       this.objCredenciais.tokenX.access_token = !!jsonTokenX.access_token ?
            jsonTokenX.access_token : "";
4       this.objCredenciais.tokenX.expires_in = !!jsonTokenX.expires_in ?
            jsonTokenX.expires_in : -1;
5       this.objCredenciais.tokenX.date_expire_in = !!jsonTokenX.date_expire_in ?
            jsonTokenX.date_expire_in : "";
6       this.objCredenciais.tokenX.scope = !!jsonTokenX.scope ? jsonTokenX.scope : "";
7       this.objCredenciais.tokenX.profiles = !!jsonTokenX.profiles ?
            jsonTokenX.profiles : [];
8       this.objCredenciais.tokenX.nome = !!jsonTokenX.nome ? jsonTokenX.nome : "";
9       this.objCredenciais.tokenX.profileAtivo = !!jsonTokenX.profileAtivo ?
            jsonTokenX.profileAtivo : "";
10      this.objCredenciais.tokenX.jti = !!jsonTokenX.jti ? jsonTokenX.jti : "";
11      this.objCredenciais.tokenX.cpf = !!jsonTokenX.cpf ? jsonTokenX.cpf : "";
12    }
13   }
```

Listagem 4.6: Trecho do método `setTokenX` presente na classe `Credenciais`.

No método `setTokenX`, os dados são setados na classe de Domínio utilizando uma condicional ternária para informar valores padrões, caso os mesmos não existam no objeto de transporte, visto essa situação, o teste criado resolve abordar os caminhos de decisão que o código precisa realizar.

Para isso, antes de cada teste é criado um objeto padrão com os dados fictícios de domínio, conforme a Listagem 4.7.

```
1  beforeEach(function () {
2         tokenXObject ={
3             access_token:faker.random.uuid(),
4             expires_in:"expires_in",
5             date_expire_in:"date_expire_in",
6             scope:"scope",
7             profiles:{profile:faker.name.jobTitle()},
```

```
8             nome:faker.name.findName(),
9             profileAtivo:true,
10            jti:"jti",
11            cpf:"54050155044"
12        }
13    }
```

Listagem 4.7: Criação do objeto `TokenX`.

Uma vez criado o objeto `tokenXObject`, foi passado por parâmetro para a construção da classe `Credenciais` e, em seguida, o teste avaliou se os atributos da instância criada apresentaram as mesmas chaves e valores do objeto passado por parâmetro. O que assegurou isso é o método `expect`, ele esperava que o valor informado seja igual em valores ao valor passado para o método `toEqual()`, conforme a Listagem 4.8.

```
1  it('Quando instancio o objeto da Classe Credenciais, informando tokenX, os valores
       devem ser adicionados corretamente ao objeto Credenciais', () => {
2         const jsonCredenciais = {tokenX:tokenXObject}
3         const sut = new Credenciais(jsonCredenciais)
4         expect(sut.objCredenciais).toBeTruthy()
5         expect(sut.objCredenciais.tokenX).toEqual(tokenXObject)
6     });
```

Listagem 4.8: Teste da criação do objeto `TokenX` e asserção de valores.

4.3 Testes na Camada *Services*

A camada *Services* apresenta três arquivos que abstraem o consumo de serviços externos como APIs e *Secure Storage*[44]. Inicialmente, foi feito o teste para a classe `AppLocalStorage` que utiliza o *Secure Storage* e o *Async Storage*[45] para armazenar as credenciais do usuário no dispositivo móvel. Para essa categoria de cenário, um teste importante é o que garante que o valor informado ao método de armazenamento `setItemSecure`, por exemplo, é o mesmo valor armazenado ao chamar as bibliotecas externas de armazenamento, além de exercitar as condicionais presentes no código demonstrado na Listagem 4.9.

```
1  export async function setItemSecure(key, value) {
```

[44]https://docs.expo.dev/versions/latest/sdk/securestore/
[45]https://reactnative.dev/docs/asyncstorage

```
2     if (value && value.length > 2048) {
3     throw new Error('Tamanho da string armazenada excede 2048 caracteres, que é o
          limite do SecureStorage!');
4     }
5     if (Platform.OS === 'ios' || Platform.Version >= 23) {
6         await SecureStore.setItemAsync(key, value);
7     } else {
8         await AsyncStorage.setItem(key, value);
9     }
10 }
```

Listagem 4.9: Método `setItemSecure` presente na classe `AppLocalStorage`.

Assim, para que o teste ocorra da forma prevista, as classes externas devem ser representadas por *mocks*. Portanto, antes da execução do teste e também da importação das bibliotecas, todas as classes externas tem seus comportamentos alterados, conforme a Listagem 4.10, garantindo que os cenários que serão testados estejam sob o controle do teste unitário e não de classes externas.

```
1  const mockPlatform = (OS,Version) => {
2     jest.resetModules();
3     jest.doMock("react-native/Libraries/Utilities/Platform", () => ({ OS,Version,
          select: objs => objs[OS] }));
4  };
5  jest.mock('expo-secure-store', () => {
6     return {
7             getItemAsync: jest.fn(),
8             setItemAsync: jest.fn(),
9             deleteItemAsync: jest.fn()
10    };
11 });
12 jest.mock('@react-native-async-storage/async-storage')
13 import faker from 'faker'
14 import * as SecureStore from 'expo-secure-store';
15 import AsyncStorage from '@react-native-async-storage/async-storage';
```

Listagem 4.10: *Mocks* de classes externas.

Com os *mocks* configurados, o teste para verificar os valores que serão armazenados pode ser realizado.

```
1
2  it('Quando chamo o metodo setItemSecure e a plataforma e Android 23, o SecureStore
       deve ser chamado', async () => {
3             let key = faker.database.column()
```

```
4           let value= faker.random.alphaNumeric()
5           mockPlatform("android",23);
6           await setItemSecure(key,value)
7           expect(SecureStore.setItemAsync).toHaveBeenCalledWith(key,value)
8       });
```

Listagem 4.11: Teste do método `SetItemSecure` para validar a decisão tomada quando a versão Android é maior ou igual a 23.

O teste apresentado na Listagem 4.11 realiza a criação de dados falsos para simular valores armazenados no *SecureStore*, em seguida, ele configura a plataforma para ser *Android*, versão 23, assim, ao chamar o método, é validado se o método interno `setItemSecure` do `SecureStore` foi chamado e com os valores passados por parâmetro. Com esse teste, a cobertura de linhas testadas para a classe foi para 35,71%.

4.4 Testes na Camada *Components*

Componentes são estruturas do React que dividem a *interface* gráfica do aplicativo em pequenos pedaços que são independentes e a reutilizáveis em diversas funcionalidades da aplicação. Ao serem criados, eles recebem como parâmetro a variável `props`, objetos *Javascript* enviados para os componentes exibirem dados ou mudar o seu comportamento durante o processo de renderização em tela. Portanto, é possível adicionar valores às `props`, validar internamente no componente para que certa parte do componente seja mostrada ou não. Como exemplo, há o componente `SocialMedia` apresentado na Listagem 4.12 que decide, baseado nos valores passados por parâmetro ao componente, o que será renderizado em tela. Para facilitar o entendimento, será apresentado o trecho do código pertinente aos testes realizados.

```
1  export default props => {
2
3     return (
4        </View>
5             <View
6                 style={[
7                   ViewStyles.viewTextsWithIcon,
8                   ViewStyles.shadow,
```

```
9                   styles.iconsSocialMedia,
10                ]}
11            >
12              {(props.redes_sociais.instagram) && (
13                  <TouchableWithoutFeedback
14                  onPress={() =>
15                  props.navigation.navigate(IndicesMenu.webView, {
16                    title: props.nome,
17                    url: "instagram.com/" + props.redes_sociais.instagram,
18                    secondaryScreen: true,
19                    })
20                  }
21                >
22                  <Instagram style={styles.icon} size={38}
                        color={Colors.colorPrimaryDark}/>
23                </TouchableWithoutFeedback>
24              )}
25              {(props.redes_sociais.facebook) && (
26                  <TouchableWithoutFeedback
27                      onPress={() =>
28                      props.navigation.navigate(IndicesMenu.webView, {
29                      title: props.nome,
30                      url: "facebook.com/" + props.redes_sociais.facebook,
31                      secondaryScreen: true,
32                      })
33                      }
34                  >
35                    <View>
36                      <Facebook size={38} color={Colors.colorPrimaryDark}
                            style={styles.icon}/>
37                    </View>
38                  </TouchableWithoutFeedback>
39              )}
40            </View>
41        </View>
42    )
43  }
```

Listagem 4.12: Componente `SocialMedia`.

Na Listagem 4.12, na linha 1, o componente é declarado, recebendo como parâmetro o objeto `props` e retorna um componente criado a partir da linha 3, após utilizar componentes nativos do React Native, nas linhas 12 e 25 são visualizados condicionais que quando os valores dos atributos `redes_sociais.instagram` e `redes_sociais.facebook` são verdadeiros, o componente é renderizado com o `TouchableWithoutFeedback` [46] refe-

[46] https://reactnative.dev/docs/touchablewithoutfeedback

rido.

Portanto, para essa categoria de componente, a partir da técnica cobertura de decisão, o teste verificou se, quando o valor é válido e quando inválido, o componente tem sua estrutura renderizada da forma correta, ou seja, apresentando ou não o `TouchableWithoutFeedback`.

Para isso, é necessário renderizar o componente utilizando o *Enzyme* e informando os valores necessários como *props* e analisar seu *snapshot* e o que foi renderizado. Portanto, o teste unitário foi criado com esse objetivo no arquivo `SocialMedia.spec.js`, conforme a Listagem 4.13.

```
1  import * as React from 'react';
2  import { shallow, mount } from "enzyme";
3  import SocialMedia from '../components/SocialMedia';
4  const createTestProps = (props) => ({
5      nome:"UFJF",
6      navigation: {
7        navigate: jest.fn(),
8        setOptions: jest.fn()
9      },
10     ...props
11   });
12 describe('Componente SocialMedia', () => {
13
14     it('Social Media renderizado com sucesso', () => {
15         let props = createTestProps({redes_sociais:{linkedin:true}});
16         const component = shallow(<SocialMedia {...props}/>);
17         expect(component).toMatchSnapshot();
18     });
19
20     it("SocialMedia renderiza o ícone do Instagram quando atributo instagram é
           verdadeiro", () => {
21         let props = createTestProps({ redes_sociais: { instagram: true } });
22         const component = mount(<SocialMedia {...props} />);
23         expect(component).toMatchSnapshot();
24         expect(component.find("TouchableWithoutFeedback")).toHaveLength(1);
25         expect(
26          component.find("TouchableWithoutFeedback Icon").instance().props.name
27         ).toBe("instagram");
28     });
29
30      it('SocialMedia não renderiza o ícone do Instagram quando atributo instagram é
            falso', () => {
31         let props = createTestProps({ redes_sociais: { instagram: false } });
32         const component = mount(<SocialMedia {...props} />);
33         expect(component).toMatchSnapshot();
```

```
34          expect(component.find("TouchableWithoutFeedback")).toHaveLength(0);
35      });
36      it("SocialMedia renderiza o ícone do Facebook quando atributo facebook é
            verdadeiro", () => {
37          let props = createTestProps({ redes_sociais: { facebook: true } });
38          const component = mount(<SocialMedia {...props} />);
39          expect(component).toMatchSnapshot();
40          expect(component.find("TouchableWithoutFeedback")).toHaveLength(1);
41          expect(
42           component.find("TouchableWithoutFeedback Icon").instance().props.name
43          ).toBe("facebook-square");
44    });
45    it("SocialMedia não renderiza o ícone do Facebook quando atributo facebook éfalso",
          () => {
46          let props = createTestProps({ redes_sociais: { facebook: false } });
47          const component = mount(<SocialMedia {...props} />);
48          expect(component).toMatchSnapshot();
49          expect(component.find("TouchableWithoutFeedback")).toHaveLength(0);
50    });
51  })
```

Listagem 4.13: Arquivo de testes `SocialMedia.spec.js`: Criado para testar a estrutura do componente `SocialMedia`.

A Listagem 4.13 apresenta alguns aspectos importantes para um teste unitário. O primeiro é a importação das bibliotecas responsáveis pelos testes entre as linhas 1 e 3, isso inclui o `React`, o `enzyme` e o componente a ser testado, `SocialMedia`. Em seguida é criado o método `createTestProps` que retorna um objeto *mock*. O método, presente na linha 4, recebe como parâmetro `props` do componente e adiciona valores fixos ou simulacros conforme a necessidade, na linha 6, foi declarado o objeto simulácro `navigation`, utilizado pelo componente para realizar a navegação entre telas do aplicativo, seus métodos receberam funções *mocks*[47] do *Jest*, através do método `jest.fn()`, para simular o comportamento do componente, assim as chamadas externas foram feitas sem gerar erros ou exceções, elas podem ser configuradas para retornar valores específicos, mas não foi necessário neste caso.

Feita a configuração inicial, é possível iniciar o teste unitário do componente `SocialMedia`, o primeiro teste verifica se ele é renderizado com sucesso. Começando na linha 15, o objeto `props` é criado ao receber o retorno do método `createTestProps` que teve como parâmetro o atributo `redes_sociais` contendo o atributo `linkedin` com

[47] https://jestjs.io/docs/mock-functions

valor `true`. Assim é criado um objeto `props` contendo os atributos necessários para a instanciação do componente. Na linha 16, a instância é criada pelo método `shallow`, ele é responsável por instanciar o componente isoladamente. Na linha 17, é verificado se o *snapshot* do componente foi renderizado corretamente.

Após a verificação da renderização via *snapshot*, foi definido dois testes para validar as decisões tomadas pela condicional, conforme o valor verdadeiro ou falso dos atributos presentes nas linhas 12 e 25 na Listagem 4.12. Portanto, em cada teste, foi atribuído ao atributo `instagram` um valor `boolean` diferente para testar cada cenário.

Começando na linha 21, é criado o objeto `props` com o atributo `instagram` com valor `true`. O componente é instanciado com o objeto `props` na linha 22, utilizando o método `mount`, ele permite que a instância do componente possua seus componentes filhos renderizados e, consequentemente, a validação dos valores que esses componentes filhos apresentem. Neste caso, o componente filho é o `TouchableWithoutFeedback`. Na linha 24, foi realizado a busca pelo componente filho sendo esperado que, neste caso, a busca retorne 1 item apenas e que esse item tenha, entre seus atributos, um Icon, componente nativo do React Native, com o atributo `name` igual à `"instagram"` conforme apresentado na linha 26.

O caso de teste presente na linha 26 realiza a validação quando o componente apresenta o atributo com valor falso. Portanto, após a criação do componente nas linhas 31 es 32, é esperado que não seja encontrado, após a busca na linha 34 , o componente filho `TouchableWithoutFeedback`, ao verificar e assim o tamanho do objeto resultado da busca seja igual a 0.Foram também criados Os casos de teste para a renderização do ícone do facebook, presentes nas linhas 36 e 45, aplicados na atribuição do valor do atributo `facebook`.

4.5 Avaliação da Cobertura de Código

A avaliação da cobertura de código após a implementação dos testes unitários foi obtida pela execução do comando `npm run test:unit`, seu resultado é apresentado na Figura 4.1. Foram criados 31 testes que conseguiram alcançar um total de 59,5% de cobertura de instruções e 73,78% de cobertura de decisão, houve avanços perante a estrutura inicial

que não apresentava testes estruturais e há oportunidades de melhorias para trabalhos futuros que possam se basear nesse conjunto inicial de testes feitos.

```
> yarn test -- --coverage --testMatch  **/__tests__/**/*.spec.js

yarn run v1.22.19
warning package.json: No license field
warning package.json: "dependencies" has dependency "jest-expo" with range "^45.0.1" that collides with a dependency in "devDe
cies" of the same name with version "^42.0.1"
warning From Yarn 1.0 onwards, scripts don't require "--" for options to be forwarded. In a future version, any explicit "--"
e forwarded as-is to the scripts.
$ jest --u --passWithNoTests  --runInBand --silent --coverage --testMatch **/__tests__/**/*.spec.js
 PASS  __tests__/Credenciais.spec.js
 PASS  __tests__/AppLocalStorage.spec.js
 PASS  __tests__/SocialMedia.spec.js
-----------------------------|---------|----------|---------|---------|-------------------
File                         | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------------|---------|----------|---------|---------|-------------------
All files                    |    59.5 |    73.78 |   30.55 |   59.58 |
 assets                      |     100 |      100 |     100 |     100 |
  Colors.js                  |     100 |      100 |     100 |     100 |
  CommonStyles.js            |     100 |      100 |     100 |     100 |
  Fonts.js                   |     100 |      100 |     100 |     100 |
 assets/fonts                |     100 |      100 |     100 |     100 |
  OpenSansBold.ttf           |     100 |      100 |     100 |     100 |
  OpenSansBoldItalic.ttf     |     100 |      100 |     100 |     100 |
  OpenSansItalic.ttf         |     100 |      100 |     100 |     100 |
  OpenSansLight.ttf          |     100 |      100 |     100 |     100 |
  OpenSansLightItalic.ttf    |     100 |      100 |     100 |     100 |
  OpenSansRegular.ttf        |     100 |      100 |     100 |     100 |
  OpenSansSemibold.ttf       |     100 |      100 |     100 |     100 |
  OpenSansSemiboldItalic.ttf |     100 |      100 |     100 |     100 |
  RobotoSlabBold.ttf         |     100 |      100 |     100 |     100 |
  RobotoSlabLight.ttf        |     100 |      100 |     100 |     100 |
  RobotoSlabRegular.ttf      |     100 |      100 |     100 |     100 |
  RobotoSlabThin.ttf         |     100 |      100 |     100 |     100 |
 business                    |   93.93 |    98.33 |   88.88 |   93.84 |
  Credenciais.js             |   93.93 |    98.33 |   88.88 |   93.84 | 263-274
 components                  |   31.57 |      100 |   27.77 |   31.57 |
  Icon.js                    |   30.76 |      100 |   30.76 |   30.76 | 9-30,59,74-95
  SocialMedia.js             |   33.33 |      100 |      20 |   33.33 | 38-83
 constants                   |     100 |       50 |     100 |     100 |
  Constantes.js              |     100 |      100 |     100 |     100 |
  Endpoints.js               |     100 |       50 |     100 |     100 | 5
 navigation                  |     100 |      100 |     100 |     100 |
  IndicesMenu.js             |     100 |      100 |     100 |     100 |
 services                    |   35.71 |       50 |   16.66 |   35.71 |
  AppLocalStorage.js         |   35.71 |       50 |   16.66 |   35.71 | 34-84
 utils                       |       4 |        0 |       0 |    4.05 |
  AlertUFJF.js               |    3.63 |        0 |       0 |     3.7 | 15-151
  Links.js                   |       5 |        0 |       0 |       5 | 10-100
-----------------------------|---------|----------|---------|---------|-------------------

Test Suites: 3 passed, 3 total
Tests:       31 passed, 31 total
Snapshots:   9 passed, 9 total
Time:        8.846s, estimated 10s
Done in 9.47s.
```

Figura 4.1: Resultado do comando para realizar os testes e analisar a cobertura de código produzida por eles.

4.6 Testes na Camada *Screens*

A camada de *Screens* será testada utilizando a abordagem automatizada funcional, devido à impossibilidade de utilização das técnicas estruturais como cobertura de decisão e sentença. Esta impossibilidade foi originada devido ao alto acoplamento entre as clas-

ses presentes nesta camada e outras, inviabilizando o processo de *mocking* e *stubing* de métodos e classes, logo as unidades não estavam aptas a serem testadas unitariamente.

No contexto de testes *mobile*, ainda há dificuldades para execução de testes automatizados funcionais, devido à necessidade de extensas configurações de ferramentas como o *Appium* ou diversidade de aparelhos presentes no mercado e usado pelos clientes, portanto para este trabalho, foram definidos cenários de teste que validassem as funcionalidades básicas do aplicativo. Sendo elas:

- Validar a visualização de Notícias da UFJF na tela principal do aplicativo;
- Validar a visualização da seção de Serviços para a comunidade;
- Validar a visualização das informações do Aplicativo;

Os demais testes que poderiam ser desenvolvidos no aplicativo apresentaram a necessidade de um ambiente de homologação com usuários para teste, o que não foi possível obter, sendo uma limitação que será melhor descrita no Capítulo 5.

4.6.1 Configuração em Dispositivos Reais

Para este trabalho foram utilizados dispositivos físicos Android das marcas Samsung e Motorola para a execução dos testes. Os três dispositivos precisaram ter o modo desenvolvedor ativado para ser possível executar os testes funcionais via ADB. Feita a conexão, foi possível visualizar via terminal, pelo comando `adb devices`, o dispositivo conectado, conforme a Figura 4.2.

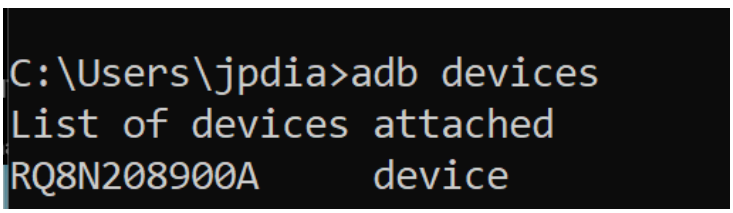

Figura 4.2: Resultado do comando *adb devices*: o nome do dispositivo é *RQ8N208900A* e o seu status *device* indica que a depuração USB está ativa e o dispositivo pode ser usado para testes.

Feita esta configuração inicial, é necessário instalar o *Appium* para realizar a integração dos testes de sistema com o dispositivo Android.

4.6.2 Configuração dos *Frameworks* para Testes Funcionais

Para sua instalação, foi necessário a execução do programa `yarn` no repositório do aplicativo UFJFApp. O primeiro comando é responsável por instalar globalmente, ou seja, para acesso no sistema operacional e não em um único projeto. O segundo comando inicializa o *Appium* para o uso, como demonstrado na Figura 4.3.

```
1 yarn global add appium
2 appium
```

Listagem 4.14: Comandos de Instalação do *Appium*.

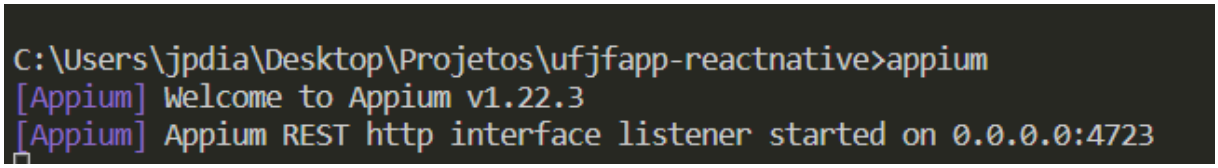

Figura 4.3: Inicialização do *Appium*: A partir da inicialização, é possível visualizar o endereço HTTP e a porta em que o servidor está respondendo às requisições.

Com esta instalação concluída, o próximo passo foi a configuração do *script* de testes e sua execução, já que os testes funcionais realizados no aplicativo utilizam algumas bibliotecas além do *Appium*. A primeira é o *Jest* para execução dos testes, a segunda é o WD.js[48], é um *framework* que permite a conexão ao *Appium* Server, abstraindo as requisições que devem ser feitas e as respostas obtidas do servidor, ambos foram instalados pelo comando `yarn add wd jest`. A biblioteca *path*[49] é nativa do *Node.js* e utilizada para facilitar a manipulação de caminhos relativos e absolutos de arquivos.

A partir deste ponto, a execução dos testes tornou-se viável, portanto, foi necessário a criação do arquivo de configuração para que a conexão ao servidor *Appium* pudesse ser realizada. Na Listagem 4.15, é demonstrado o arquivo `config.js`. Na linha 1, a biblioteca `path` foi importada para ser utilizada, nas linhas 2 e 3 foram definidas constantes relacionadas a versão do Android e o nome do dispositivo conectado ao USB do computador, obtido através do `adb devices`. As capacidades do Android são definidas na linha 4, essas informações são enviadas ao *Appium* para ele poder reconhecer e se conectar ao dispositivo via depuração. As configurações de conexão ao *Appium* obtidas

[48]Disponível em: ⟨http://admc.io/wd/⟩
[49]Disponível em: ⟨https://nodejs.org/api/path.html⟩

na sua inicialização são instanciadas na linha 11. Por fim, foi necessário que o time de Desenvolvimento gerasse o arquivo `.apk` do aplicativo e disponibilizasse para testes, assim, ele foi salvo no diretório base do UFJFApp e nas linhas 15 e 17 são criados o caminho absoluto do APK. Finalizando a configuração, esses objetos são exportados para serem utilizados durante os testes do *Appium* na linha 18.

```
1  import path from 'path';
2  const NOME_DISPOSITIVO = "RQ8N208900A";
3  const VERSAO_ANDROID = "12";
4  const capacidadeAndroid = {
5      platformName: 'Android',
6      deviceName: NOME_DISPOSITIVO,
7      platformVersion: VERSAO_ANDROID,
8      appWaitForLaunch: true,
9      app: undefined
10 };
11 const configAppium = {
12     host: 'localhost',
13     port: 4723
14 };
15 const CAMINHO_BASE = path.resolve(__dirname);
16 let caminhoApp;
17 caminhoApp = path.resolve(CAMINHO_BASE, 'ufjf-app.apk');
18 export {
19     caminhoApp, capacidadeAndroid,
20     configAppium
21 };
```

Listagem 4.15: Arquivo de configuração do *Appium*: `config.js`.

Tornou-se possível executar o *Appium* em conjunto com o *Jest* e o WD.js Com essas instruções, o que será descrito na próxima seção.

4.6.3 Escrita e Execução dos Testes Funcionais

Para a execução dos testes, foi criado o arquivo `app.test.js`, é inicialmente feita a importação da biblioteca WD.js e os dados do arquivo *config.js* na linha 1 e na linha 2 da Listagem 4.16 respectivamente. Na linha 3 é configurado o *timeout* como tempo limite de cada um dos testes executados pelo *Jest*, em milissegundos. O valor corresponde a 10 minutos e demonstrou ser o necessário devido à diferença de tempo entre a requisição realizada ao *Appium* e a execução da ação requisitada no dispositivo físico.

```
1 import wd from "wd";
2 import { caminhoApp, capacidadeAndroid, configAppium } from "./config";
3 jest.setTimeout(600000);
```

Listagem 4.16: Arquivo de testes `app.test.js`: importações e configuração de tempo de execução.

Concluída a parte de importação e configuração do *Jest*, foi iniciado o processo de criação dos testes funcionais. O *Jest* possui alguns métodos que auxiliam à criação dos testes dividindo em passos e generalizando ações comuns a todos os testes e todos os métodos do WD.js são assíncronos[50], ou seja, necessitam ser aguardados para a obtenção do resultado para a próxima ação ser feita por meio da palavra-chave `await` e os métodos que possuem essa palavra-chave devem ser sinalizados como assíncronos pela palavra-chave `async`.

Na Listagem 4.17, o método `describe` [51], presente na linha 1, agrupa em blocos, testes relacionados a um contexto, no caso é o UFJFApp. Na linha 2, a variável `driver` foi declarada para ser instanciada posteriormente no método `beforeEach`[52], responsável por executar antes dos testes presentes no `describe` um bloco de código comum a todos eles. Assim, na linha 3 sua chamada foi feita sendo passado como parâmetro uma função assíncrona responsável pela inicialização do driver com os valores importados do arquivo `config.js`. A conexão ao servidor Appium é feita na linha 4, o método `promiseChainRemote` recebe como parâmetro os dados de conexão do *Appium* e retorna uma instância do driver, quando a conexão é bem sucedida, sendo necessário que o *Appium* esteja instalado e ativo.

Com a conexão feita, foi necessário iniciar o driver através do método `init` na linha 5, tendo como parâmetros as capacidades do Android configuradas na Listagem 4.15 e o caminho do APK do aplicativo UFJFApp gerado para testes. A constante `capacidadeAndroid` foi atribuída via desestruturação[53] para que suas propriedades sejam passadas por parâmetro como variáveis. Ao passar o caminho do instalador do aplicativo,

[50] ⟨https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/async_function#specifications⟩

[51] https://jestjs.io/docs/api#describename-fn

[52] https://jestjs.io/docs/api#beforeeachfn-timeout

[53] ⟨https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment⟩

o *Appium* é responsável pela instalação no dispositivo físico do aplicativo e sua execução inicial.

Cada ação do teste automatizado pode ter que esperar certo tempo para ser feita, como, por exemplo, o tempo de carregamento da tela para em seguida clicar em um botão, para isso, na linha 9 é definido o tempo de espera de 5 segundos para todas as ações feitas pelo WD.js. Entre as linhas 10 e 12, é feito a asserção sobre qual a Activity[54]e o *Package* do aplicativo que está sendo demonstrado na tela do dispositivo móvel, é esperado que sejam informações do UFJFApp. Caso isso não ocorra, a execução será finalizada e os testes não serão executados. Portanto, toda essa preparação é feita a cada teste executado, permitindo segurança de que o aplicativo está carregado e pronto para uso da automação.

Ao fim de cada teste, o método `afterEach`[55] é chamado, finalizando a conexão do WD.js com o *Appium* pelo método `quit`, conforme a linha 18, garantindo que uma nova conexão deve ser criada e o aplicativo deva ser reaberto no dispositivo móvel, assim todos os testes partirão do mesmo ponto, a tela inicial do UFJFApp.

```
1  describe("UFJF App - ", () => {
2    let driver;
3    beforeEach(async () => {
4      driver = await wd.promiseChainRemote(configAppium);
5      await driver.init({
6        ...capacidadeAndroid,
7        app: caminhoApp,
8      });
9      await driver.setImplicitWaitTimeout(5000);
10     const activity = await driver.getCurrentActivity();
11     const pkg = await driver.getCurrentPackage();
12     expect(`${activity}${pkg}`).toBe(
13       "host.exp.exponent.MainActivitybr.ufjf.ufjfapp"
14     );
15   });
16     ....
17   afterEach(async () => {
18     await driver.quit();
19   });
20 });
```

[54]https://developer.android.com/reference/android/app/Activity
[55]⟨https://jestjs.io/docs/api#aftereachfn-timeout⟩

Listagem 4.17: Arquivo de testes `app.test.js`: Métodos auxiliares para criação dos testes.

Para automatizar os cenários de testes definidos, foi necessário que o time de desenvolvimento adicionasse em alguns elementos interativos e botões o atributo `accessibilityLabel` que permitiu a fácil identificação deles pela ferramenta de automação. Inicialmente, foi criado um teste para validar o funcionamento da tela de Notícias presente na tela principal, este cenário é demonstrado na Figura 4.4.

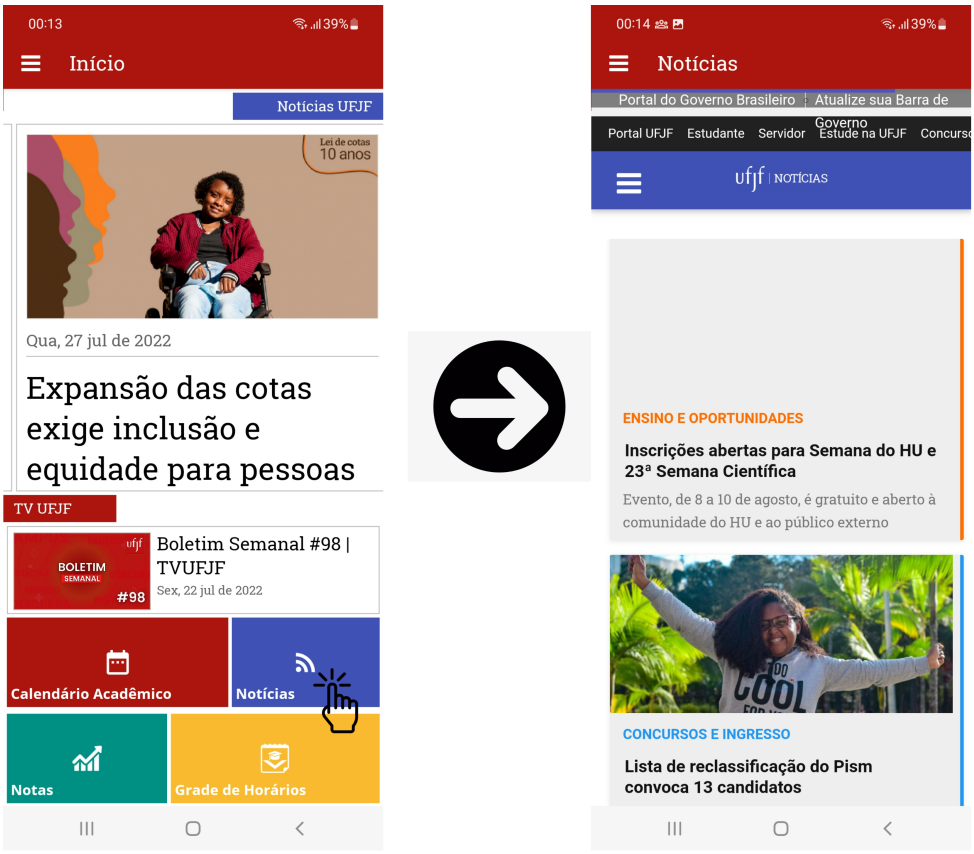

Figura 4.4: Cenário 1: *Validação da Tela de Notícias* consiste em clicar no botão *Notícias* na primeira tela e ser redirecionado para a segunda tela, a página de notícias.

Na Listagem 4.18 é possível visualizar o código que automatiza esse cenário. A linha 1 é a chamada do método `describe` que já foi apresentado e engloba todos os testes e os blocos de código que devem ser executados anterior e posteriormente aos testes omitidos desta Listagem. Com foco na validação do cenário proposto, a linha 3 invoca o método `test` que recebe dois parâmetros: o título do teste e uma função que executará o teste. Em seguida, a linha 4 busca pelo elemento que possui o `AcessibilityLabel` pelo

[56] ⟨https://reactnative.dev/docs/accessibility#accessibilitylabel⟩

método `elementByAcessibilityId`, apesar da diferença na nomenclatura, a informação buscada é a mesma. Ao encontrar esse botão, a variável `botaoNoticias` é instanciada e na linha 5 é chamado o método `click`, caso ele responda com sucesso, o teste é aprovado.

```
1 describe("UFJF App - ", () => {
2     ...
3     test("abrir a tela de inicio e clicar nas notícias na tela Principal", async () =>
          {
4         let botaoNoticias = await driver.elementByAccessibilityId("Notícias UFJF");
5         await botaoNoticias.click();
6     });
7     ...
8 });
```

Listagem 4.18: Arquivo de testes `app.test.js`: Validando a tela de Notícias da UFJF.

Visando a implementação de novos cenários e a passagem de conhecimento ao time de bolsistas do projeto, foram criados dois novos cenários de teste e levantado junto ao time quais seriam as necessidades para que aqueles testes fossem implementados. O time encontrou a necessidade de tornar o menu acessível por meio dos *acessibilityLabels*. Cada uma das opções demonstradas na Figura 4.5 teve a adição do atributo `acessibilityLabel` em sua declaração no React Native. Após essa inclusão, foi possível realizar a automação das interações com o menu.

O próximo cenário, demonstrado na Figura 4.6, foi criado pela bolsista A após sessões explicativas sobre o que é o Appium, como é automatizar validações de aplicativos com ele e os passos a serem desenvolvidos.

A Listagem 4.19 demonstra o código escrito, na linha 4, o teste busca, pelo atributo `acessibilityLabel`, o botão que acessa o Menu do UFJFApp e instancia a variável *botaoMenu*, em seguida, o botão é clicado na linha 5, possibilitando a obtenção do elemento para acessar a seção de Serviços para a Comunidade da mesma forma na linha 6. Logo, na linha 7, o botão é clicado e, feita essa ação, é avaliado se a tela atual, em execução no aplicativo, apresenta um elemento Android, chamado `TextView`[57], com o texto Ciência e Inovação, essa busca é feita na linha 8, utilizando a estrutura de XPath[58] para buscar nós em estruturas baseadas em XML[59]. Por fim, foi feita uma asserção uti-

[57] ⟨https://developer.android.com/reference/android/widget/TextView⟩
[58] ⟨https://www.w3schools.com/xml/xpath_syntax.asp⟩
[59] ⟨https://www.w3schools.com/xml/xml_whatis.asp⟩

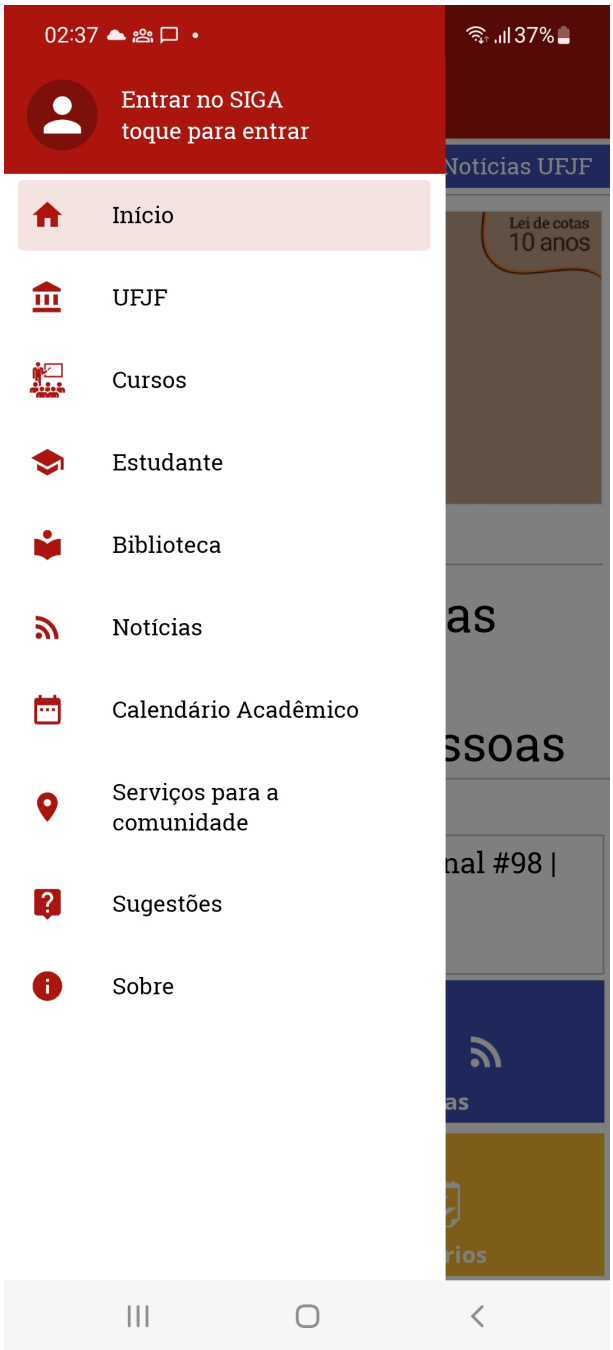

Figura 4.5: Menu do UFJFApp - A adição do atributo *acessibilityLabel* facilitou a navegação do *Appium* pelo aplicativo.

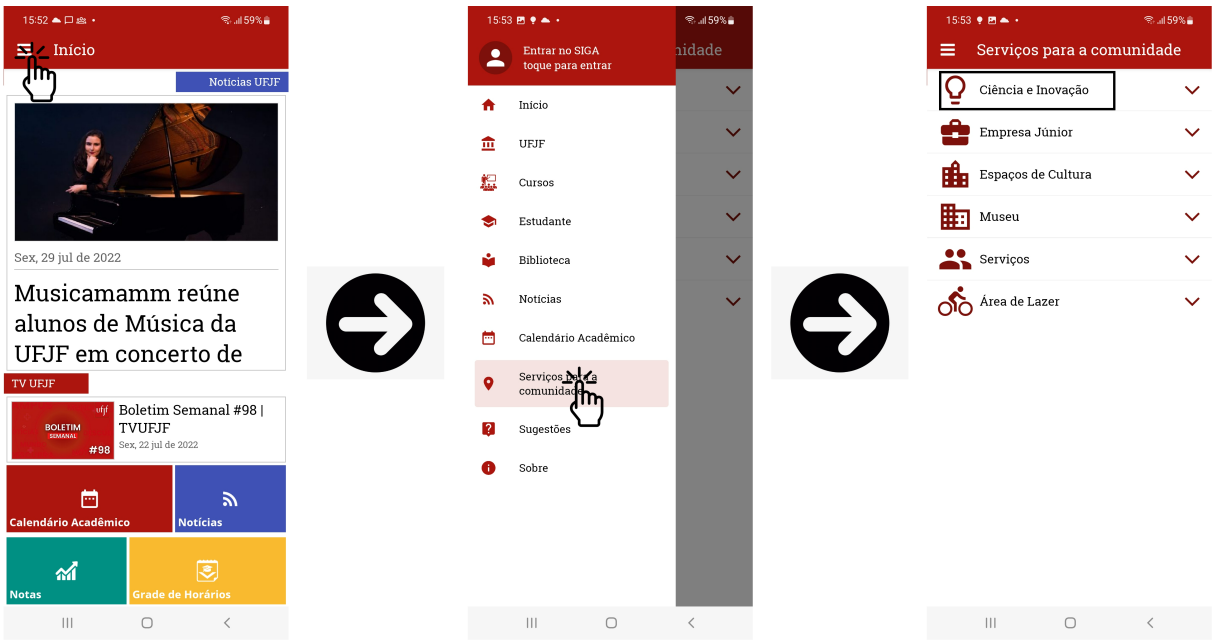

Figura 4.6: Cenário de Teste: *Validar a visualização da seção de Serviços para a comunidade* consiste em abrir o menu do UFJFApp, acessar a seção *Serviços da Comunidade* e validar a presença do texto *Ciência e Inovação*, garantindo que os dados estão sendo apresentados.

lizando o método `expect`[60], em que, é esperado que a variável `conteudoServicos` seja verdadeira, ou seja, não nula. Logo, o teste passou, pois o elemento foi encontrado. Caso ele não seja encontrado em outro momento, o método `elementByXPath` irá retornar nulo e o teste falhará.

```
1  describe("UFJF App - ", () => {
2      ...
3      test("abrir a tela de inicio, clicar na opção do menu: Serviços para a Comunidade
           e visualizar os serviços realizados", async () => {
4          let botaoMenu = await driver.elementByAccessibilityId("Menu");
5          await botaoMenu.click();
6          let botaoServicos = await driver.elementByAccessibilityId("Serviços para a
               comunidade");
7          await botaoServicos.click();
8          let conteudoServicos = await
               driver.elementByXPath("//android.widget.TextView[@text='Ciência e
               Inovação']");
9          expect(conteudoServicos).toBeTruthy();
10     });
11     ...
12 });
```

Listagem 4.19: Arquivo de testes `app.test.js`: Validando a funcionalidade de *Serviços para a Comunidade*.

[60] ⟨https://jestjs.io/docs/expect⟩

Concluindo a implementação de cenários de teste utilizando o *Appium*, foi possível validar a visualização das informações do aplicativo na seção *Sobre*, conforme a imagem 4.7. Ele foi desenvolvido pelo Bolsista B do UFJFApp, após sessões explicativas sobre as tecnologias utilizadas e o cenário proposto.

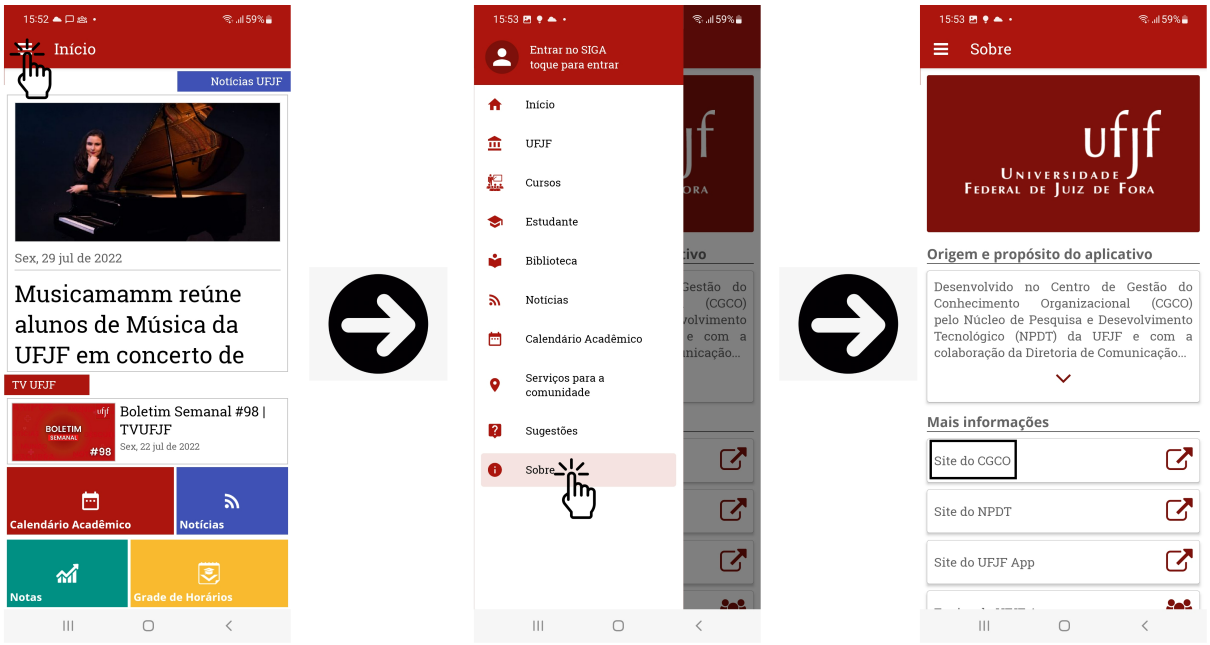

Figura 4.7: Montagem indicando o Cenário de Teste: Validar a visualização da seção Sobre do aplicativo. O teste consiste em abrir o menu do UFJFApp, acessar a seção Sobre e validar a presença do texto Site do CGCO, garantindo que os dados estão sendo demonstrados.

Na implementação do teste apresentado na Listagem 4.20, foi criado um objeto que representa o menu da mesma forma que no teste anterior na linha 4 sendo clicado na linha 5, a grande diferença está no elemento a ser procurado que, neste caso, foi o botão que acessa os dados sobre o aplicativo utilizando o atributo `accessibilityLabel` com o valor `Sobre` na linha 6, concluída a instanciação do objeto `botaoSobre`, ele é clicado através do comando na linha 7. A asserção, assim como no cenário anterior, é feito sobre o texto demonstrado em tela nas linhas 10 e 11.

```
1 describe("UFJF App - ", () => {
2     ...
3     test("abrir a tela de inicio e clicar na opção do menu: Sobre e visualizar as
          informações do Aplicativo", async () => {
4     let botaoMenu = await driver.elementByAccessibilityId("Menu");
5     await botaoMenu.click();
6     let botaoSobre = await driver.elementByAccessibilityId("Sobre");
7     await botaoSobre.click();
```

```
8     let conteudoSobre = await driver.elementByXPath(
9       "//android.widget.TextView[@text='Site do CGCO']"
10    );
11    expect(conteudoSobre).toBeTruthy();
12  });
13    ...
14 });
```

Listagem 4.20: Arquivo de testes `app.test.js`: Validando a funcionalidade Sobre.

Após a criação dos testes, eles foram executados utilizando o comando `npm run test:e2e`, ele invoca o *Jest* que executa o arquivo `app.test.js` e o resultado é demonstrado na Figura 4.8. Os cenários propostos foram executados com sucesso, porém em um tempo acima do esperado. Isso é devido ao processo de realização de requisições ao *Appium* e ele invocar as ações no aparelho.

```
C:\Users\jpdia\Desktop\Projetos\ufjfapp-reactnative>npm run test:e2e

> test:e2e
> yarn test app.test.js

yarn run v1.22.10
warning package.json: No license field
$ jest --u --passWithNoTests  --runInBand --silent app.test.js
 PASS  __tests__/e2e/app.test.js (259.522 s)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        260.165 s
Done in 261.06s.
```

Figura 4.8: Resultado da execução dos testes funcionais automatizados com *Appium*.

4.7 Avaliação dos Bolsistas sobre os Testes Feitos

O método de teste foi avaliado ao instruir os dois alunos bolsistas de treinamento profissional vinculados ao projeto UFJFApp no CGCO. Foi observado se eles conseguiriam entender os benefícios dos testes e como criá-los no contexto de aplicações móveis. A Bolsista A é aluna do curso de Sistemas de Informação e está no terceiro período e o Bolsista B é aluno de Ciência da Computação e está no quinto período. Ambos aprenderam o uso

do React e React Native ao longo de um ano e não tiveram contato prévio com a disciplina de testes, nem na universidade e nem em estágio. Após as sessões de instruções, de forma isolada, ambos configuraram o ambiente de testes em seus computadores e criaram um conjunto de testes. Por fim, foram elaboradas três perguntas para avaliar a percepção do processo de teste por parte dos bolsistas.

A primeira pergunta, *Após as sessões explicativas, qual a importância dos testes para o software?*, visa avaliar o que mais chamou a atenção do bolsista perante à implementação dos testes no UFJFApp. A segunda, *Onde que teve mais dificuldade no uso das ferramentas de teste?*, busca entender os pontos que podem prejudicar o uso contínuo das ferramentas de teste. E a última, *Em um próximo projeto deseja utilizar testes para seu conteúdo?*, busca entender quais os impactos que os testes trouxeram para poderem ser replicados em próximos projetos.

4.7.1 Avaliação da Bolsista A

As questões foram encaminhadas para a bolsista por e-mail, sendo reproduzidas abaixo:

1. Após as sessões explicativas, qual a importância dos testes para o software?

   > Considero os testes muito importantes, pois previnem que alguns problemas cheguem aos usuários finais, além de poupar bastante tempo do desenvolvedor, visto que automatiza a etapa de testes da aplicação que está sendo desenvolvida. Em projetos médios e grandes se tornar essencial a realização desses testes automatizados, pois existem muitas funcionalidades e demandaria um tempo gigantesco testar manualmente cada uma.

2. Onde que teve mais dificuldade no uso do framework?

   > Creio que minha maior dificuldade foi a sintaxe. Sempre que vou aprender qualquer tecnologia nova é o que mais me deixa confuso. Além disso, sou uma pessoa que gosta de aprender por videoaulas, então a falta dessas videoaulas em português brasileiro na Internet também dificulta meu aprendizado.

3. Em um próximo projeto deseja utilizar testes para seu conteúdo?

> Com certeza! Até mesmo em projetos pequenos, pois serviria para eu me aperfeiçoar na tecnologia e ficar melhor preparado para aplicar os testes em grandes projetos. Além disso, mesmo sendo um projeto pequeno, é bom garantir a qualidade do mesmo.

4.7.2 Avaliação da Bolsista B

O bolsista B apresentou respostas mais sucintas que podem dificultar a avaliação por sua parte, sendo elas:

1. Após as sessões explicativas, qual a importância dos testes para o software?

> Eu penso que os testes são importantes principalmente para aumentar a confiança no que está sendo desenvolvido, seja um aplicativo, sistema web, jogo, etc. É interessante poder implementar uma função ou fazer uma mudança de outro tipo sabendo que isso não vai "quebrar" o que já existia antes.

2. Onde que teve mais dificuldade no uso do framework?

> A parte mais difícil foi a configuração de dependência, especificamente as variáveis de ambiente do Java.

3. Em um próximo projeto deseja utilizar testes para seu conteúdo?

> Sim. A minha intenção é implementar testes assim que começar a desenvolver um projeto do zero.

A partir das respostas é possível observar que ambos os bolsistas perceberam a importância dos testes para o software e também a segurança que os testes dão perante ao funcionamento esperado do aplicativo. Dentre as dificuldades, temos a falta de conteúdo em português sobre as ferramentas utilizadas, principalmente o *Enzyme*. Ainda há a configuração de variáveis de ambiente para a instalação do emulador quando utilizado para testes funcionais, no início das sessões, mas devido à necessidade de um alto poder

de processamento, foi utilizado os dispositivos físicos para os testes funcionais. Quanto ao interesse por utilizar os testes em próximos projetos que desenvolverem, os bolsistas demonstraram interesse no uso.

4.8 Considerações sobre o Protocolo de testes

O protocolo de testes implementado possibilitou ao projeto UFJFApp o uso testes estruturais e funcionais automatizados, totalizando 31 testes unitários e 3 funcionais criados, além das sessões explicativas e acompanhamento do desenvolvimento dos testes funcionais com os bolsistas. O trabalho demonstrou a configuração necessária para implementar testes em um aplicativo que não possuía nenhuma forma de validação, além da manual. A cobertura de código aumentou de 0 para 56% nas classes testados e possibilitando que os testes unitários sejam reproduzidos para aumentar a cobertura de código.

Os testes funcionais mostraram-se possíveis, mas com a necessidade de mapear os elementos do aplicativo com *acessibilityLabel* para poderem interagir com *Appium*. O tempo de execução dos testes funcionais foi de 1 minuto para cada teste, tornando-se custoso dependendo da necessidade. Atualmente, tecnologias permitem que os testes sejam executados remotamente em dispositivos móveis emulados em nuvem, essa técnica é chamada de *device farm*[61], porém, em termos financeiros, aumenta o custo, sendo em torno de 0,17 dólares por minuto de uso[62].

Dentre limitações relacionados a sua infraestrutura e codificação, o ambiente de testes é inexistente, impedindo o uso de credenciais para utilizar a área logada do aplicativo, dificultando a criação de testes para funcionalidades complexas como Visualização de Notas e do IRA. O código demonstrou-se muito acoplado em camadas como *Screens* e *Components*, impossibilitando o uso de técnicas de testes estruturais para os testes unitários, pois não era possível realizar *mocking* e *stubing* nas respectivas classes presentes nessas camadas.

Por fim, foi criado o diagrama do fluxo sugerido para inserção de testes no ciclo de desenvolvimento do UFJFApp, presente na Figura 4.9. Há a atividade de testes

[61] https://aws.amazon.com/device-farm/
[62] https://aws.amazon.com/device-farm/pricing/

unitários antes da geração da versão de testes, buscando evidenciar defeitos presentes nesta fase de desenvolvimento. Após a criação dos testes unitários e correção de possíveis erros, os bolsistas do projeto criarão e executarão os testes automatizados das funcionalidades novas e existentes, caso não haja defeitos, os servidores testarão as funcionalidades novas e existentes com mais rapidez e, caso qualquer dessas fases apresentem defeitos, será enviado ao desenvolvedor responsável para correção.

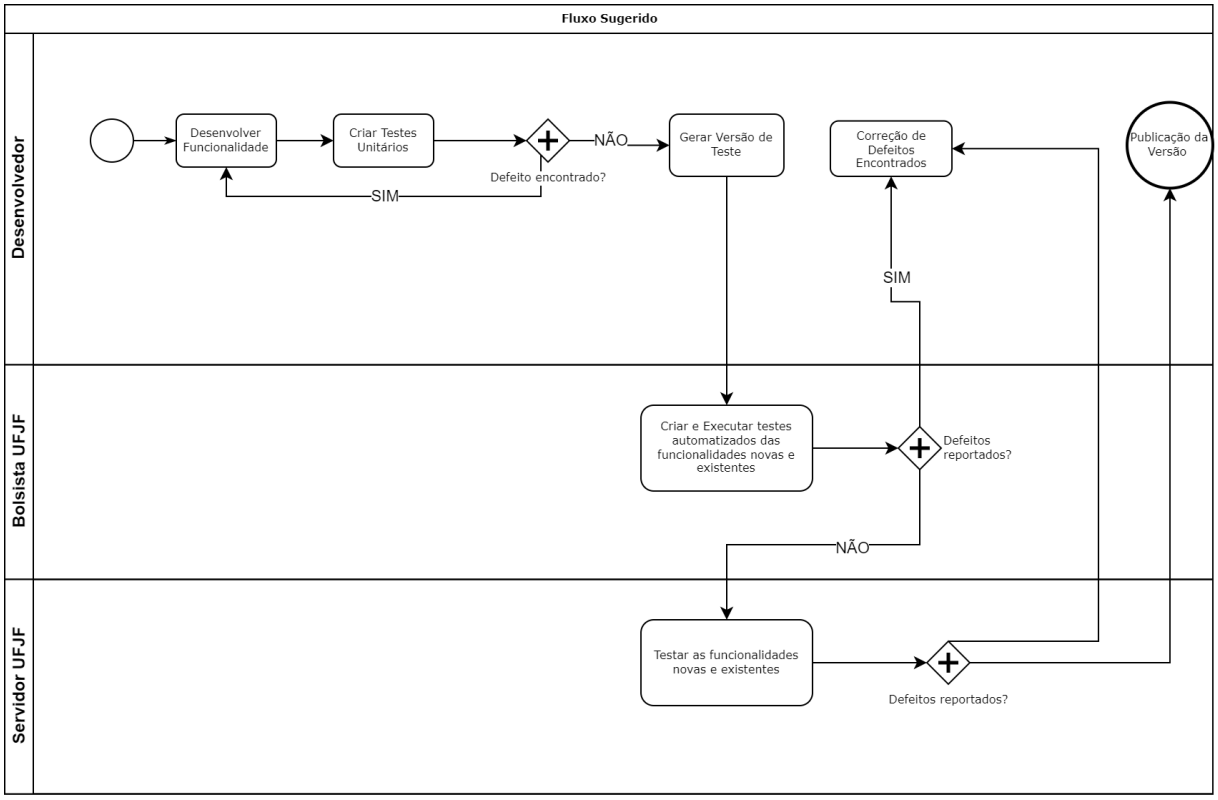

Figura 4.9: Fluxo sugerido de desenvolvimento do aplicativo UFJFApp

Este capítulo apresentou o projeto, configuração e implantação de um protocolo de testes para um aplicativo institucional da UFJF. No próximo capítulo serão feitas as considerações finais deste trabalho e possíveis continuações deste trabalho serão sugeridas.

5 Considerações Finais

Neste capítulo as conclusões sobre a implantação proposta de um novo processo de teste no grupo de desenvolvimento do aplicativo institucional da UFJF são apresentadas, bem como suas limitações e sugestões para trabalhos futuros.

O desenvolvimento de aplicativos mobile é uma tarefa árdua devido ao uso de ferramentas específicas para atender diferentes sistemas operacionais, Android e *iOS*, e para haver êxito em sua entrega a qualidade deve se manter em ambos. O aplicativo da UFJF adequou suas tecnologias para o React Native, ferramenta que permite o desenvolvimento híbrido, possibilitando o acesso de mais de 6 mil estudantes às informações sobre a universidade e também ao seu histórico discente em diferentes dispositivos móveis. O segundo ponto envolve um trabalho manual de validação antes de cada nova publicação do aplicativo, causando uma mobilização de pessoal para realizar os testes no aplicativo e a exigência de testes em funcionalidades novas e existentes.

Para sistematizar a tarefa de testes, foram implementados testes unitários e funcionais no UFJFApp utilizando frameworks *Javascript*, como o *jest*, *Enzyme* e *Appium* que possibilitam dar início a um processo automatizado de garantia de qualidade a cada nova versão, aumentando a cobertura de código, dificultando a inserção de novos defeitos e exercitando cenários de uso do aplicativo em dispositivos móveis reais.

Foi realizada uma observação da capacidade de implementar os testes por parte dos bolsistas do projeto após sessões treinamento da abordagem proposta. Ambos os bolsistas conseguiram realizar os testes direcionados e opinaram positivamente sobre o método.

Como resultado dos esforços da pesquisa foi possível criar os testes na aplicação em diferentes camadas, assim como a capacitação dos bolsistas do projeto para eles poderem dar continuidade na criação de novos testes, assim como a documentação que explica cada categoria de teste implementada em anexo a essa pesquisa, cumprindo os objetivos definidos na seção 3.

Os objetivos específicos definidos para o projeto foram, na maioria, alcançados,

possibilitando a entrega de testes automatizados em diversas camadas. A caracterização dos métodos e ferramentas utilizados foram definidos por pesquisas das ferramentas para testes em aplicativos híbridos com a tecnologia React Native, eles: o *Jest*, o *Enzyme* e o *Appium*.

O processo de desenvolvimento consistiu no levantamento das funcionalidades por parte dos interessados no projeto e a implementação por parte dos bolsistas e servidores dedicados ao projeto. Esse processo se manteve inalterado, pois mudanças na infraestrutura do projeto estão sendo avaliadas pela UFJF.

Após a análise do projeto, foi possível identificar camadas que podem ser testes unitariamente e funcionalidades que podem ser testadas funcionalmente e aplicar testes em cada uma delas, com algumas limitações. Portanto, foram definidas camadas do projeto que pudessem ser testadas seja unitária ou funcionalmente. Foram aplicados testes unitários na camada de *Business*, *Components* e *Services* e as demais camadas foram exercitadas por testes funcionais.

Após a criação dos testes, foram realizadas sessões de diagnóstico com os bolsistas da equipe técnica do UFJFApp, observando os testes realizados na instrução e pedindo para eles reproduzirem o processo em outros. Com as sessões feitas e os testes criados por eles, um formulário foi disponibilizado aos bolsistas para eles poderem opinar sobre o que aprenderam em relação aos testes.

O objetivo de mapeamento dos pontos de melhoria e criação de um planejamento a longo prazo para a implantação não puderam ser realizados, pois, o sistema está sendo reorganizado tecnicamente, contando com mudanças de equipe e também com implementação de novas tecnologias do React Native.

Durante a execução deste trabalho, algumas dificuldades impediram o avanço e criar mais testes no contexto do UFJFApp. Como a lógica de programação utilizada no projeto dificulta a aplicação de testes unitários, pois há contextos que princípios como o de responsabilidade única dos postulados SOLID (MARTIN, 2000) não são seguidos, dificultando o teste unitário, tornando um assunto para trabalhos futuros com foco em refatoração de código do projeto do UFJFApp.

A aplicação de testes funcionais demonstrou ser possível até certo ponto, devido

a uma limitação da API da UFJF quanto a usuários específicos de teste, impossibilitando a realização de testes funcionais que requisitavam recursos com o usuário logado. Portanto, a reestruturação dos ambientes de desenvolvimento, possibilitando a criação de um ambiente próprio para testes, é um tópico interessante para ser trabalhado futuramente.

O problema abordado neste pode ser melhorado com o uso de ambientes separados para homologação e publicação do aplicativo, como também o uso de *device farms*, infraestruturas em nuvem que permitem a execução de testes em dispositivos móveis de forma remota, diminuindo o tempo de instalação da infraestrutura de um emulador e também permitindo a execução dos testes de forma paralela. Porém, os custos da operação podem ser altos a medida que o uso dos recursos da infraestrutura em nuvem aumenta.

Bibliografia

AHMAD, A. et al. An empirical study of investigating mobile applications development challenges. *IEEE Access*, IEEE, v. 6, p. 17711–17728, 2018.

AHMED, S.; SADATH, L.; NAGARIA, J. Software testing and lines of codes-a study on software engineering design patterns. In: *2019 International Conference on Automation, Computational and Technology Management (ICACTM)*. [S.l.: s.n.], 2019. p. 389–394.

ALAMRI, H. S.; MUSTAFA, B. A. Software engineering challenges in multi platform mobile application development. *Advanced Science Letters*, American Scientific Publishers, v. 20, n. 10-11, p. 2115–2118, 2014.

ALBIERO, F. W. Uma abordagem de teste para aplicativos android utilizando os cenários do behavior driven development. 2017.

ALOTAIBI, A. A.; QURESHI, R. J. Novel framework for automation testing of mobile applications using appium. *International Journal of Modern Education & Computer Science*, v. 9, n. 2, 2017.

ANTONIO, C. d. S. Testing react components. In: *Pro React*. [S.l.]: Springer, 2015. p. 281–292.

ARAUJO, G. R. d. Desenvolvimento cross-platform com react native: um estudo de caso do aplicativo naveg. 2019.

BARBOSA, E. F. et al. Introdução ao teste de software. *Minicurso apresentado no XIV Simpósio Brasileiro de Engenharia de Software (SBES 2000)*, 2000.

BEZERRA, P. T.; SCHIMIGUEL, J. Desenvolvimento de aplicações mobile cross-platform utilizando phonegap. 2016.

BOUSHEHRINEJADMORADI, N. et al. Testing cross-platform mobile app development frameworks (t). In: IEEE. *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2015. p. 441–451.

DANIELSSON, W. React native application development. *Linköpings universitet, Swedia*, v. 10, n. 4, p. 10, 2016.

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2013.

EISENMAN, B. *Writing cross-platform apps with react native*. InfoQ, 2016. Disponível em: ⟨https://www.infoq.com/articles/react-native-introduction/⟩.

FACEBOOK. *Testing Overview React Native*. 2021. ⟨https://reactnative.dev/docs/testing-overview⟩. (Accessed on 07/06/2021).

FAYYAZ, Z. et al. Recommendation systems: Algorithms, challenges, metrics, and business opportunities. *applied sciences*, MDPI, v. 10, n. 21, p. 7748, 2020.

GOADRICH, M. H.; ROGERS, M. P. Smart smartphone development: ios versus android. In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. [S.l.: s.n.], 2011. p. 607–612.

HECK, F. S. Sistema móvel de controle de presença. 2014.

HUANG, C.-Y. et al. Code coverage measurement for android dynamic analysis tools. In: IEEE. *2015 IEEE International Conference on Mobile Services*. [S.l.], 2015. p. 209–216.

KADU, A. N. *Optimizing and Vulnerability Testing of a Cloud-Based Intelligent Tutoring System*. Tese (Doutorado) — INDIAN INSTITUTE OF TECHNOLOGY KANPUR, 2021.

LECHETA, R. R. *Google Android-3ª Edição: Aprenda a criar aplicações para dispositivos móveis com o Android SDK*. [S.l.]: Novatec Editora, 2013.

MARTIN, R. C. Design principles and design patterns. *Object Mentor*, v. 1, n. 34, p. 597, 2000.

MEDNIEKS, Z.; DORNIN, G. L.; NAKAMURA, M. Programando o android. *São Paulo: Novatec*, 2012.

MESQUITA, K. A evolução do governo eletrônico no brasil e a contribuição das tic na redefinição das relações entre governo e sociedade. *Comunicologia-Revista de Comunicação da Universidade Católica de Brasília, Brasília*, v. 12, n. 2, p. 174–195, 2020.

MÜLLER, G. da R.; SOARES, I. W. Estudo comparativo sobre ferramentas de desenvolvimento multiplataforma para aplicaç oes móveis. 2018.

MULLER, T. et al. *Certified Tester Syllabus Foundation level - BSTQB*. BSTQB - Brazilian Software Testing Qualifications Board, 2019. Disponível em: ⟨https://bstqb.org.br/b9/doc/syllabus_ctfl_3.1br.pdf⟩.

MYERS, J.; SANDLER, C.; BADGETT, T. The art of software testing. *Jersey City, NJ*, 2004.

NALKECZ, A. *Cutting-Edge Business Models in The Age of Digital Disruption–Examples, Prospects, and Key Economic and Legal Challenges*. Tese (Doutorado) — University of Warsaw, 2019.

NOVAC, O. C.; MARCZIN, R.-G.; NOVAC, M. C. Comparison of hybrid cross-platform mobile applications with native cross-platform applications. *Journal of Computer Science & Control Systems*, v. 9, n. 2, 2016.

OLIVEIRA, D. de J. Uma proposta de arquitetura para single-page applications. 2017.

PAUL, A.; NALWAYA, A. The ecosystem: Extending react native. In: *React Native for Mobile Development*. [S.l.]: Springer, 2019. p. 225–232.

POLO, M. et al. Test automation. *IEEE Software*, v. 30, n. 1, p. 84–89, 2013.

PRESSMAN, R.; MAXIM, B. *Engenharia de Software-8ª Edição*. [S.l.]: McGraw Hill Brasil, 2016.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de software-9*. [S.l.]: McGraw Hill Brasil, 2021.

RAHMANI, A.; MIN, J. L.; MASPUPAH, A. An evaluation of code coverage adequacy in automatic testing using control flow graph visualization. In: IEEE. *2020 IEEE 10th Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. [S.l.], 2020. p. 239–244.

SANTOS, D. S. d. Sistema de recomendação de frameworks para desenvolvimento multiplataforma em dispositivos móveis. Pós-Graduação em Ciência da Computação, 2018. Disponível em: ⟨https://ri.ufs.br/jspui/bitstream/riufs/10685/2/DENISSON_SANTANA_SANTOS.pdf⟩.

SARTORELI, C. E.; KUCHAUSKI, N. A. U. Comparativo entre ios, android e windows phone. *ETIC-Encontro de iniciação científica-ISSN 21-76-8498*, v. 9, n. 9, 2014.

SILVA, C. G. G. d. M. *Estudo comparativo de ferramentas de testes de ponta a ponta automatizados em sistemas web*. Dissertação (B.S. thesis) — Universidade Federal do Rio Grande do Norte, 2019.

SINGH, S.; GADGIL, R.; CHUDGOR, A. Automated testing of mobile applications using scripting technique: A study on appium. *International Journal of Current Engineering and Technology (IJCET)*, Citeseer, v. 4, n. 5, p. 3627–3630, 2014.

SOMMERVILLE, I. *Engineering software products*. [S.l.]: Pearson London, 2020.

TOLEDO, J. M.; DEUS, G. D. de. *Desenvolvimento em Smartphones-Aplicativos Nativos e Web*. 2012.

TRAMONTANA, P. et al. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal*, Springer, v. 27, n. 1, p. 149–201, 2019.