UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS CORNÉLIO PROCÓPIO
DIRETORIA DE PESQUISA E PÓS - GRADUAÇÃO
PROGRAMA DE PÓS - GRADUAÇÃO EM INFORMÁTICA

THIAGO BOTTI DE ASSIS

AMPLIFICAÇÃO DE TESTES AUTOMATIZADOS PARA APLICAÇÕES MÓVEIS MULTIPLATAFORMAS BASEADA EM TEST PATTERNS

DISSERTAÇÃO DE MESTRADO

CORNÉLIO PROCÓPIO
2019

THIAGO BOTTI DE ASSIS

AMPLIFICAÇÃO DE TESTES AUTOMATIZADOS PARA APLICAÇÕES MÓVEIS MULTIPLATAFORMAS BASEADA EM TEST PATTERNS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Universidade Tecnológica Federal do Paraná – UTFPR como requisito parcial para a obtenção do título de “Mestre em Informática”.

Orientador: Prof. Dr. André Takeshi Endo

CORNÉLIO PROCÓPIO
2019

Dados Internacionais de Catalogação na Publicação

A848 Assis, Thiago Botti de

Amplificação de testes automatizados para aplicações móveis multiplataformas baseada em *Test Patterns* / Thiago Botti de Assis. – 2019.
68 f. : il. color. ; 31 cm.

Orientador: André Takeshi Endo.
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Informática, Cornélio Procópio, 2019.
Bibliografia: p. 65-68.

1. Aplicativos móveis. 2. Software - Testes. 3. Sistemas operacionais (Computadores). 4. Informática – Dissertações. I. Endo, André Takeshi, orient. II. Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Informática. III. Título.

CDD (22. ed.) 004

Biblioteca da UTFPR - Câmpus Cornélio Procópio
Bibliotecário/Documentalista responsável:
Romeu Righetti de Araujo – CRB-9/1676

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Cornélio Procópio
Programa de Pós-Graduação em Informática

Título da Dissertação Nº 67:

"AMPLIFICAÇÃO DE TESTES AUTOMATIZADOS PARA APLICAÇÕES MÓVEIS MULTIPLATAFORMAS BASEADA EM TEST PATTERNS".

por

Thiago Botti de Assis

Orientador: **Prof. Dr. André Takeshi Endo**

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM INFORMÁTICA – Área de Concentração: Computação Aplicada, pelo Programa de Pós-Graduação em Informática – PPGI – da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Cornélio Procópio, às 15h30 do dia 15 de agosto de 2019. O trabalho foi _____ pela Banca Examinadora, composta pelos professores:

_____
Prof. Dr. Willian Massami Watanabe
(Presidente – UTFPR-CP)

_____
Prof. Dr. Cléber Gimenez Corrêa
(UTFPR-CP)

_____
Prof. Dr. Paulo Augusto Nardi
(UTFPR-CP)

_____
Prof. Dr. Marcelo Medeiros Eler
(EACH-USP)
Participação à distância via _____

Visto da coordenação:

_____
Danilo Sipoli Sanches
Coordenador do Programa de Pós-Graduação em Informática
UTFPR Câmpus Cornélio Procópio

A Folha de Aprovação assinada encontra-se na Coordenação do Programa.

AGRADECIMENTOS

Gostaria de iniciar agradecendo ao Professor André Takeshi Endo, pois suas orientações me mostraram inúmeras vezes o caminho prático para o aprendizado, além de sua persistência e paciência, que me incentivaram durante esse período.

Agradeço também minha família pelo apoio, em especial a minha esposa pela compreensão e motivação constante em cada etapa desta pesquisa.

Por último, não posso deixar de agradecer igualmente ao meu colega de mestrado, André Augusto Menegassi.

RESUMO

DE ASSIS, Thiago Botti. **AMPLIFICAÇÃO DE TESTES AUTOMATIZADOS PARA APLICAÇÕES MÓVEIS MULTIPLATAFORMAS BASEADA EM TEST PATTERNS**. 2019. 67 f. Dissertação – Mestrado – Programa de Pós-graduação em Informática. Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2019.

Contexto: Desenvolvedores de aplicações móveis possuem a necessidade de disponibilizar seus produtos para uma grande variedade de dispositivos com particularidades e sistemas operacionais (SOs) distintos. Assim, é cada vez mais comum o desenvolvimento multiplataforma, ou seja, a mesma aplicação e suas funcionalidades são desenvolvidas para os diferentes SOs. A busca pela qualidade das aplicações móveis cresceu junto com sua propagação. Os usuários exigem que as aplicações móveis sejam confiáveis, robustas e eficientes. Como consequência, os desenvolvedores de software devem adotar técnicas de garantia de qualidade. A realização de testes de aplicações móveis pode ser desafiadora, considerando que erros podem existir devido à interação dos usuários, como girar o dispositivo ou usar vários gestos para rolar ou ampliar a tela durante a execução de uma aplicação móvel. Com isso é possível definir *Test Patterns,* estratégias utilizadas para testar eventos relacionados à interface do usuário, para que sejam verificadas ações comuns nas aplicações móveis, evitando que erros inesperados ocorram durante a realização do teste de uma funcionalidade da aplicação móvel. **Objetivo:** esta dissertação apresenta uma abordagem de amplificação de testes em aplicações móveis multiplataforma. Para isso, foram aplicados quatro *Test Patterns* que verificam as características conhecidas da computação móvel e amplificam conjuntos de testes funcionais existentes. **Método:** a fim de amplificar casos de testes funcionais em aplicações móveis multiplataforma, foi investigada na literatura o uso de *Test Patterns* em aplicações móveis, levantando erros recorrentes motivados por eventos realizados nos dispositivos móveis. Em seguida, foram desenvolvidos scripts para a execução dos *Test Patterns* de maneira automatizada. Por fim, foi conduzido um experimento, analisando o resultado quanto ao número de erros encontrados e o tempo da execução dos testes. **Resultados:** a abordagem proposta foi implementada em uma ferramenta capaz de gerar scripts de testes e foi avaliada com nove aplicações multiplataforma, gerando testes automatizados amplificados. Os dados coletados no experimento mostraram que o conjunto de testes amplificados identificaram 23 erros em oito das nove aplicações móveis, consumiu em média 2,5 vezes o tempo gasto em um caso de teste sem amplificação no Android e 1,5 vezes para o iOS.

Palavras-chave: Dispositivos Móveis, Aplicações Móveis, Aplicações Multiplataforma, Teste de Software.

ABSTRACT

DE ASSIS, Thiago Botti. **AMPLIFICATION OF AUTOMATED TESTS FOR CROSS-PLATFORM MOBILE APPLICATIONS BASED ON TEST PATTERNS.** 2019. 67 p. Dissertação – Mestrado – Programa de Pós-graduação em Informática. Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2019.

Context: Mobile application developers need to make their products available to numerous devices with distinct features and operating systems (OSs). In this way, multiplatform development is increasingly common, that is, the same application and its functionalities are developed for the different OSs. The search for the quality of mobile applications has grown along with its spread, users require that mobile applications are reliable, robust and efficient; therefore, software developers must adopt quality assurance techniques. Mobile app testing can be challenging considering that crashes may exist due to user interaction such as rotate the device or use various gestures to scroll or magnify the screen while running a mobile application. This allows you to define Test Patterns, strategies used to test events related to the user interface, so that common events in the mobile applications are checked, preventing unexpected failures during the test of a functionality of the mobile application. **Objective:** This dissertation aims at investigating the amplification of tests in multiplatform mobile applications. Four Test Patterns were applied to check the known characteristics of mobile computing and amplify existing sets of functional tests. **Method:** To amplify functional test cases in cross-platform mobile applications, we have investigated the use of Test Patterns in mobile applications, raising recurrent failures motivated by events performed on mobile devices. Then, scripts were developed to execute the Test Patterns in an automated way. Finally, an experiment was conducted, and the result was analyzed for the number of failures encountered and the time of execution of the tests. **Results:** The proposed approach was implemented in a tool able of generating test scripts and was evaluated with nine multiplatform applications, generating automated amplified tests. The data collected in the experiment showed that the amplified set of tests found 23 unique bugs in eight of the nine mobile applications and consumed on average 2.54 times the time spent in a test case without amplification on Android and 1.55 times for iOS.

Keywords: Mobile devices, Mobile Applications, Cross-platform Applications, Software Testing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

SOs	Sistemas Operacionais
IDC	*International Data Corporation*
OHA	*Open Handset Alliance*
UWP	*Universal Windows Apps*
APIs	*Application Programming Interfaces*
XAML	*eXtensible Application Markup Language*
XML	*Extensible Markup Language*
UI	*User Interface*
QP	Questão de Pesquisa
ER	Erros
FL	Falhas
FP	Falso Positivo
RAM	*Random Access Memory*
GB	*Gigabyte*

SUMÁRIO

1. INTRODUÇÃO

Os dispositivos de computação móvel, como *smartphones*, *tablets, e-readers* e *wearables*[1], estão dominando o consumo de informação digital, tanto que o tráfego de dados proveniente de dispositivos móveis tem crescido exponencialmente nos últimos anos. As tecnologias existentes tiveram uma grande evolução com a chegada dos dispositivos móveis. Atualmente eles podem ser utilizados para inúmeras tarefas diárias, como navegação na web, bate-papo on-line, realização de transações bancárias e acesso a redes sociais (Dzhagaryan e Milenkovi, 2016).

As aplicações móveis mudaram radicalmente o estilo de vida de bilhões de pessoas ao redor do mundo, sendo utilizadas por várias horas, todos os dias, ajudando os usuários a realizar uma grande variedade de atividades (Amalfitano *et al.*, 2017). Para atender à ampla gama de usuários, milhões de aplicações foram desenvolvidas e estão disponíveis para *download* (Rumee e Liu, 2013). Essas aplicações são executadas a partir de Sistemas Operacionais (SOs) que gerenciam funcionalidades como conexões de rede e gerenciador de bateria, realizando a comunicação com o hardware (Pastore, 2013).

As aplicações móveis possuem uma variedade de entrada de informações realizadas pelo usuário como a digitação de um texto no teclado, eventos executados através do toque, ou por fatores de ambiente como o acelerômetro e o *Global Positioning System* (GPS) (Liu, Gao e Long, 2010). Segundo a IBM (2012), existem três principais abordagens para o desenvolvimento de aplicações móveis, que são definidas como nativas, web ou híbridas. As aplicações nativas interagem diretamente com o SO e são capazes de acessar todas as funcionalidades do dispositivo móvel. As aplicações web utilizam as tecnologias *Hypertext Markup Language* (HTML5), Cascading Style Sheets (CSS3) e JavaScript (W3C, 2019) e são executadas diretamente nos navegadores móveis dos dispositivos. E por fim, as aplicações híbridas, onde são combinadas tecnologias nativas e web. Nessa abordagem todas as funcionalidades dos dispositivos móveis podem ser acessadas com o auxílio de uma ponte JavaScript para a comunicação.

É cada vez mais comum o desenvolvimento da mesma aplicação e suas funcionalidades para os diferentes SOs. Com isso, são utilizadas linguagens e

[1] Objetos eletrônicos vestíveis como óculos, pulseiras e relógios.

bibliotecas de programação distintas para produzir apenas uma aplicação que esteja disponível em várias plataformas (Joorabchi, Ali e Mesbah, 2015; El-kassas *et al.*, 2016). Com essa realidade, empresas de desenvolvimento de aplicações móveis estão utilizando o desenvolvimento multiplataforma que tem como conceito o desenvolvimento de apenas uma aplicação móvel, ou seja, escrever apenas uma vez o código-fonte e possibilitar a compilação e execução em diferentes SOs (El-kassas *et al.*, 2016).

A demanda por qualidade nas aplicações móveis cresceu em conjunto com a sua propagação, seus usuários exigem que elas sejam confiáveis, robustas e eficientes; como consequência, os desenvolvedores de software devem adotar técnicas de garantia de qualidade adequadas (Amalfitano *et al.*, 2017). Para garantir a qualidade das aplicações móveis, elas precisam passar por fases de testes durante todo o período de seu desenvolvimento. O teste de software pode ser definido como o processo de execução de um sistema com o objetivo de encontrar erros (Goodenouth e Gerhart, 1975; Myers, Thomas e Wiley, 2004). Visando o aumento da produtividade, é possível utilizar o teste automatizado, desta forma pode-se realizar a execução de casos de teste quantas vezes forem necessárias (Hoffman, 2001).

Em aplicações móveis, é comum que existam erros devido à interação dos usuários, como girar o dispositivo ou usar vários gestos para rolar ou ampliar a tela durante a execução da aplicação (Zaeem, Prasad e Khurshid, 2014). Um estudo realizado por Adamsen, Mezzetti e Møller (2015) apresentou uma abordagem baseada na visão de que os casos de testes em aplicações móveis se forem executados em condições adversas, podem influenciar na qualidade do software. Com isso é possível definir *Test Patterns* para que sejam testados eventos comuns nas aplicações móveis, evitando que erros inesperados ocorram durante a realização do teste de uma funcionalidade da aplicação móvel.

Os *Patterns* podem estar diretamente relacionados à interface de usuário (UI), como a presença da barra de ações, gaveta lateral ou eventos do sistema, como girar a tela, receber uma chamada ou perder a conexão com a Internet (Coimbra Morgado e Paiva, 2015). As estratégias utilizadas para testar os *Patterns* são chamadas de *Test Patterns* (Morgado e Paiva, 2015).

1.1. MOTIVAÇÃO

Atualmente os desenvolvedores de aplicações móveis possuem a necessidade de disponibilizar seus produtos para uma grande variedade de dispositivos móveis, que executam diferentes SOs, como Android, iOS e Windows, por isso é cada vez mais comum a adoção de ferramentas de desenvolvimento multiplataforma (Boushehrinejadmoradi *et al.*, 2015).

Uma abordagem comum para a automatização de teste de software é utilizar uma ferramenta de captura e repetição com a qual os testadores podem gravar automaticamente *scripts* manipulando a aplicação móvel que está em teste, em seguida, reproduzem os scripts gravados de forma automática (Liu, Gao e Long, 2010). Normalmente o teste automatizado é desenvolvido para atender o aspecto funcional, testando um determinado requisito para garantir que funcione de acordo com o esperado. Algumas características e ações que são realizadas nos dispositivos móveis, geralmente não são testadas de forma automática e podem ser incluídas nos casos de teste para garantir maior qualidade no teste realizado.

Devido às peculiaridades dos dispositivos, as aplicações móveis devem ser testadas com abordagens elaboradas, e os processos de teste devem dar atenção aos eventos específicos existentes nos dispositivos móveis, como por exemplo, receber uma chamada, alterar o estado das conexões de rede ou alterar a orientação do dispositivo enquanto a aplicação está em execução (Amalfitano *et al.*, 2017). Com isso, a amplificação de teste foi investigada na literatura, podendo ser definida como a extensão de casos de teste existentes com o objetivo de verificar outras propriedades do software não identificados nos casos de teste funcionais. Essa é uma estratégia promissora para aplicações móveis suscetíveis a erros relacionados à rotação de dispositivos, interrupções, perda de recursos, eventos e componentes exclusivos da UI dos dispositivos móveis (Danglot *et al.*, 2019).

Com o esforço utilizado para a criação do teste automatizado, aumentar sua abrangência incluindo *Test Patterns* pode ser interessante. Dessa forma, é possível testar diferentes características de uma aplicação durante a execução dos testes funcionais, obtendo maior retorno sobre o esforço gasto no desenvolvimento do caso de teste. Além disso, novos *Test Patterns* podem ser definidos para aplicações multiplataforma. Por exemplo, quando há a necessidade da interface da aplicação ter a mesma reação independente do dispositivo que ela esteja sendo executada.

Considerando a dificuldade na realização dos testes em aplicações móveis multiplataforma citada por Menegassi (2018), e o esforço gasto no desenvolvimento de testes automatizados dessas aplicações, este estudo busca a amplificação dos testes nas aplicações móveis, automatizando um catálogo de *Test Patterns* identificados na literatura que serão utilizados para encontrar erros provenientes de fatores externos que normalmente não são testados nos casos de testes funcionais.

1.2. OBJETIVOS

As aplicações multiplataforma trazem desafios para a execução da atividade de teste de software. Nesse contexto, esta dissertação de mestrado tem como principal objetivo amplificar testes automatizados para aplicações móveis multiplataformas. De maneira específica, foi investigado o uso de *Test Patterns* nas aplicações móveis e como eles podem ser integrados em uma ferramenta. Dessa forma, esta abordagem foi dividida em três etapas. A primeira etapa foi a realização de um levantamento na literatura dos erros recorrentes motivados por eventos realizados nos dispositivos móveis; esses eventos são descritos como *Test Pattens*. A segunda etapa, foi o desenvolvimento de scripts para a execução dos *Test Patterns* de maneira automatizada. E na terceira etapa, foi realizado um estudo que utilizou algumas aplicações como base e casos de testes já existentes, onde foram adicionados os scripts dos *Test Patterns* e analisado o resultado quanto ao número de erros encontrados e o tempo da execução dos testes.

1.3. ORGANIZAÇÃO DO TEXTO

Esta dissertação de mestrado está organizada da seguinte forma: no Capítulo 2 são apresentados os principais conceitos sobre computação móvel, aplicações móveis, aplicações multiplataformas e testes automatizados.

No Capítulo 3 é descrita a abordagem proposta, detalhando a elicitação dos *Test Patterns* na literatura, em seguida os conceitos de *Test Patterns* e a implementação da ferramenta utilizada.

O Capítulo 4 apresenta os resultados esperados, demonstrando a definição do estudo, o procedimento, as ameaças a validade, além de analisar e discutir os resultados da avaliação.

Finalmente o Capítulo 5 apresenta a conclusão do estudo, os trabalhos futuros e a divulgação dos resultados.

2. REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta um embasamento teórico para contextualizar aplicações móveis e testes automatizados. Ele está organizado da seguinte forma: a Seção 2.1 aborda as principais características da computação móvel. Na Seção 2.2, são apresentados os principais conceitos sobre aplicações móveis. A Seção 2.3 apresenta uma visão geral sobre aplicações multiplataforma. Por fim, na Seção 2.4, são abordados os principais conceitos sobre testes automatizados, além de apresentar detalhes sobre a ferramenta utilizada e os *Test Patterns*.

2.1. COMPUTAÇÃO MÓVEL

A computação móvel é a manipulação de dispositivos portáteis por meio de aplicações móveis com o objetivo de trocar informação independente da sua localização física (Jing, Helal e Elmagarmid, 1999). É possível citar vantagens obtidas pela computação móvel, tais como o acesso imediato uniforme à informação e como consequência, a execução de tarefas de maneira rápida sem depender da localização física do usuário (Dzhagaryan e Milenkovi, 2016). Por outro lado, pelo menos três fatores interferem e podem limitar tecnologias que envolvem a computação móvel: (i) variedade de dispositivos móveis, (ii) conectividade e (iii) mobilidade (Gaddah e Kunz, 2003). A variedade de dispositivos móveis, a conectividade e a mobilidade definem algumas outras restrições à computação móvel, tais como tamanho de telas, menor poder de processamento, mecanismos de comunicação instáveis e baixa disponibilidade de energia (Muccini, Di Francesco e Esposito, 2012).

As tecnologias existentes tiveram uma grande evolução com a chegada da computação móvel, além de gerar novas demandas por desenvolvimento de software (Dzhagaryan e Milenkovi, 2016). Os dispositivos móveis possuem SOs desenvolvidos para gerenciar o *hardware* e funcionalidades específicas, como gerenciador de bateria, e de conexões de rede, facilitando a comunicação com as aplicações (Pastore, 2013).

De acordo com um estudo realizado pela *International Data Corporation* (IDC), os fornecedores mundiais de *smartphones* comercializaram um total de 312,9 milhões de

unidades durante o primeiro trimestre de 2019, sendo 86,7% de dispositivos utilizando o Android como SO e 13,3% com o iOS (IDC, 2019).

Android. O SO foi desenvolvido com a liderança do Google após uma aliança chamada de *Open Handset Alliance* (OHA), um grupo de grandes nomes do mercado de telefonia móvel (Lecheta, 2015). O Android foi construído baseado no Linux e sua UI é rica e dá suporte a funcionalidades existentes nos dispositivos móveis, além de conter diversas aplicações já instaladas. O Android possui o código-fonte aberto, e isso permite que os fabricantes possam editar e customizar o SO para seus dispositivos móveis (Lecheta, 2015).

iOS. O iOS é um SO desenvolvido pela Apple, e sua execução é restrita a dispositivos móveis da marca. As versões mais atuais do iOS garantem a execução dos novos *hardware* existentes nos dispositivos móveis Apple, como acelerômetro, GPS e Câmera (Milani, 2014). As linguagens de programação oficiais para a plataforma iOS são Objective-C e o Swift que é uma linguagem com sintaxe simples e moderna (Lecheta, 2015).

2.2. APLICAÇÕES MÓVEIS

Segundo Muccini, Di Francesco e Esposito (2012), uma aplicação móvel pode ser considerado todo software que é executado em um dispositivo que possa se mover, como leitores de músicas digitais, câmeras e telefone celular. Inicialmente as aplicações móveis mais utilizadas eram as redes sociais, como as versões para dispositivos móveis do *Facebook*[2] e *Twitter* [3](Pastore, 2013). No entanto, atualmente os dispositivos móveis estão equipados com processadores de alta tecnologia, sensores e alta capacidade de armazenamento (Boushehrinejadmoradi *et al.*, 2015). Com isso, houve um aumento na utilização de dispositivos móveis para acesso a diversos tipos de aplicações, como bancos, negócios e educação (Pastore, 2013). A criação dessas aplicações móveis tem se tornado cada vez mais simples, considerando que os ambientes de desenvolvimento, ferramentas de apoio e as plataformas de computação móvel evoluíram (Wasserman e Fosser, 2010).

[2] www.facebook.com
[3] www.twitter.com

As aplicações móveis possuem características em comum; uma delas é a variedade de entrada de informações, que podem ser realizadas pelo usuário digitando um texto no teclado e por eventos executados através do toque, ou por fatores de ambiente como o acelerômetro e o GPS (Liu, Gao e Long, 2010). Muitas empresas no momento de iniciar o processo de desenvolvimento de aplicações móveis estão se deparando com a necessidade de definir uma abordagem de desenvolvimento nativa, web ou híbrida, que podem interferir no cronograma, no orçamento e até mesmo na funcionalidade específica do software (IBM, 2019).

Aplicações Nativas. As aplicações nativas interagem diretamente com o SO móvel. Elas são capazes de acessar todas as *Application Programming Interfaces* (APIs) disponibilizadas pelo fornecedor do SO, e possuem arquivos binários que são baixados diretamente para o dispositivo móvel. Essas aplicações são facilmente encontradas para *download* nas lojas de aplicações móveis como a App Store[4] da Apple, o Google Play[5] para o Android ou a Microsoft Store[6] para dispositivos com Windows (IBM, 2019). Portanto, o desenvolvimento de aplicações móveis nativas é realizado de forma específica para cada SO (Latif e Nfaoui, 2016).

Aplicações web. Atualmente as aplicações web possuem um padrão de desenvolvimento que utiliza as tecnologias HTML5, CSS3 e JavaScript (W3C, 2019). Essas aplicações são executadas dentro de navegadores que estão disponíveis nos dispositivos móveis atuais. Com essa possibilidade, houve um aumento das ferramentas JavaScript, como *dojox.mobile*, *Sencha Touch* e *jQuery Mobile*, que possibilitam a criação de UI parecidas com as existentes nas aplicações móveis nativas. No entanto como essas aplicações dependem do navegador para acessar às APIs do SO, muitos recursos nativos dos dispositivos móveis não estão disponíveis (IBM, 2019).

Nessa abordagem, a aplicação é desenvolvida apenas uma vez de forma otimizada, considerando os diversos tamanhos de tela, para que seja possível se adaptar a qualquer dispositivo; dessa forma uma vantagem das aplicações web é estarem disponíveis para uma diversidade de navegadores e consequentemente dispositivos móveis (Latif e Nfaoui, 2016).

[4] https://www.apple.com/br/ios/app-store/
[5] https://play.google.com/store
[6] https://www.microsoft.com/pt-br/store/b/home

Aplicações híbridas. As aplicações híbridas realizam a combinação do desenvolvimento nativo com a tecnologia web, onde é utilizada uma abordagem desenvolvendo trechos de código-fonte web e trechos nativos da aplicação móvel, permitindo que a aplicação híbrida consiga utilizar todos os recursos que os dispositivos móveis possuem (IBM, 2019). No entanto, as interfaces híbridas são inferiores em desempenho quando comparadas com as nativas, uma vez que a execução acontece no mecanismo de renderização do navegador e adiciona as funcionalidades nativas através do uso de uma ponte JavaScript.

A ponte JavaScript é um método utilizado para estabelecer a comunicação entre o SO e a aplicação web. Ela é capaz de receber as mensagens enviadas pelas APIs e traduzir essas informações (Méndez-porras, Quesada-lópez e Jenkins, 2015).

2.3. APLICAÇÕES MULTIPLATAFORMAS

Com o objetivo de aumentar a quantidade de usuários utilizando as aplicações móveis, é cada vez mais comum o desenvolvimento da mesma aplicação e suas funcionalidades para os diferentes SOs. Desta forma, acabam sendo utilizadas linguagens e bibliotecas de programação distintas, para produzir apenas uma aplicação que está disponível em várias plataformas (Joorabchi, Ali e Mesbah, 2015), (El-kassas *et al.*, 2016).

Atualmente, empresas de desenvolvimento de aplicações móveis estão utilizando o desenvolvimento multiplataforma, que tem como conceito o desenvolvimento de apenas uma aplicação móvel, ou seja, escrever apenas uma vez o código-fonte e possibilitar a compilação e execução em diferentes SOs (El-kassas *et al.*, 2016).

No mercado atual, existem algumas ferramentas de desenvolvimento multiplataforma comerciais e de código-fonte aberto, que possibilitam o desenvolvimento de aplicações móveis para plataformas diferentes sem a necessidade de reescrever todo o código-fonte, diminuindo o tempo e o custo do projeto. Algumas das principais ferramentas são apresentadas a seguir.

React Native: criada pelo *Facebook*, essa ferramenta possibilita que o desenvolvimento seja realizado utilizando uma biblioteca JavaScript chamada *React*, gerando aplicações móveis nativas, que após compilada não possui diferença entre uma aplicação desenvolvida com Objective-C ou Java (Facebook, 2019).

Com o *React Native,* o desenvolvimento móvel é realizado utilizando apenas JavaScript, possibilitando gerar pacotes para cada plataforma. Dessa forma as aplicações móveis podem ser publicadas diretamente em suas respectivas lojas. No entanto, a ferramenta disponibiliza a possibilidade de intercalar código-fonte nativo no desenvolvimento, ou seja, é possível desenvolver trechos nas linguagens de cada plataforma, como Objective-C ou Java (Facebook, 2019).

Outra particularidade do *React Native* é que para renderizar os componentes para as aplicações móveis, existe uma camada de abstração, conhecida como ponte, que permite que o *React Native* invoque as APIs diretamente do iOS ou Android (Eisenman, 2015). Na Figura 1, está representado como o *React Native* funciona. Ao invés de renderizar para o navegador como no ReactJS, é utilizada a ponte JavaScript para invocar as APIs nativas dos SOs.

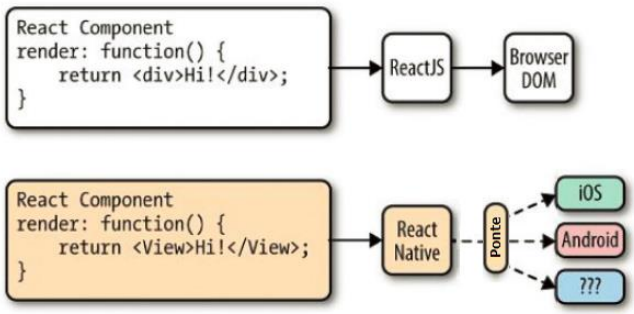

Figura 1 – Funcionamento do *React Native* (Adaptado de Eisenman, 2015).

Xamarin: adquirido pela Microsoft em fevereiro de 2016, o Xamarin oferece o C# ou F# como linguagem comum e compila executáveis para as três maiores plataformas atuais: Android, iOS e Windows (Microsoft, 2019a). Quando as aplicações *Xamarin* são compiladas, é gerado um pacote para cada plataforma; esses arquivos são nativos e podem ser publicados exatamente do jeito que foram gerados (Microsoft, 2019a).

A ferramenta possui duas abordagens para o desenvolvimento de aplicações móveis, a primeira é a *Xamarin.Forms*, que é geralmente utilizada para aplicações móveis que exigem pouca funcionalidade específica de cada plataforma e possui grande compartilhamento de código-fonte. A segunda abordagem é *Xamarin.iOS* e *Xamarin.Android*, que possui abrangência nativa para acesso às funcionalidades e

particularidades de cada plataforma, no entanto o compartilhamento do código-fonte é reduzido (Microsoft, 2019c).

Segundo a Microsoft, por utilizar a linguagem C# ou F# e uma linguagem de marcação declarativa chamada *eXtensible Application Markup Language* (XAML) para o desenvolvimento, em média 75% do código-fonte das aplicações são compartilhados. Caso seja utilizado Xamarin.Forms, o compartilhamento de código-fonte pode chegar próximo dos 100% (Microsoft, 2019b). A Figura 2 ilustra a arquitetura utilizada no Xamarin, onde é possível observar que o código-fonte compartilhado abrange o banco de dados, os serviços em nuvem, e as camadas de acessos a essas informações, além da camada de negócio onde é desenvolvido o código-fonte com as regras dos sistemas. Já nas camadas de aplicação e interface, o código-fonte é escrito de forma específica para cada SO.

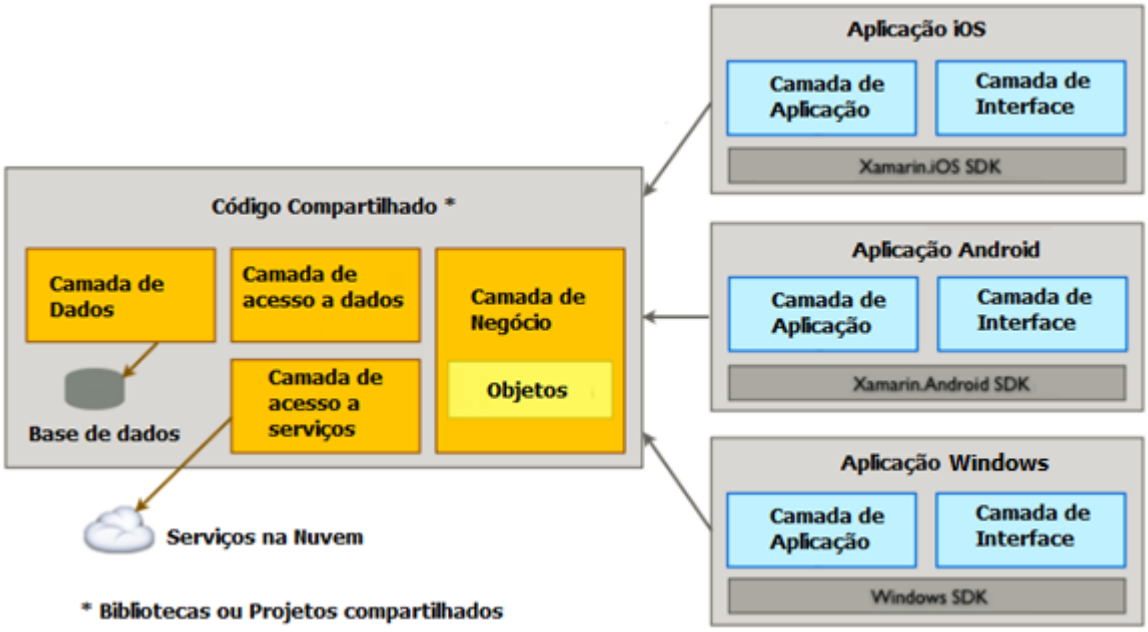

Figura 2 – Arquitetura de uma aplicação Xamarin (Adaptado de Microsoft, 2019e).

Apache Cordova: é uma ferramenta de desenvolvimento de aplicações móveis híbridas, que possui o código-fonte aberto e permite que sejam desenvolvidas aplicações móveis com tecnologias padrões da web, como HTML5, CSS3 e JavaScript (Apache, 2019). Essas tecnologias são executadas dentro de um padrão para cada plataforma e depende de conexões com APIs compatíveis para acessar as

particularidades de cada dispositivo, como sensores, status da rede, entre outros (Malavolta *et al.*, 2015) (Bosnic, Papp e Novak, 2016).

A estrutura padrão de uma aplicação Cordova é baseada em três componentes conforme a Figura 3. O primeiro é a Aplicação web, onde fica localizado o código-fonte da aplicação móvel, mais especificamente páginas HTML, que possui referências de CSS, JavaScript, imagens, arquivos de mídia e outros recursos web. O segundo componente são os Plugins Cordova, que são integrados a ferramenta. Eles fornecem uma interface entre o Cordova e os componentes nativos do dispositivo móvel para realizar a comunicação entre eles. E por fim o WebView que realiza a renderização de toda interface com o usuário (Apache, 2019).

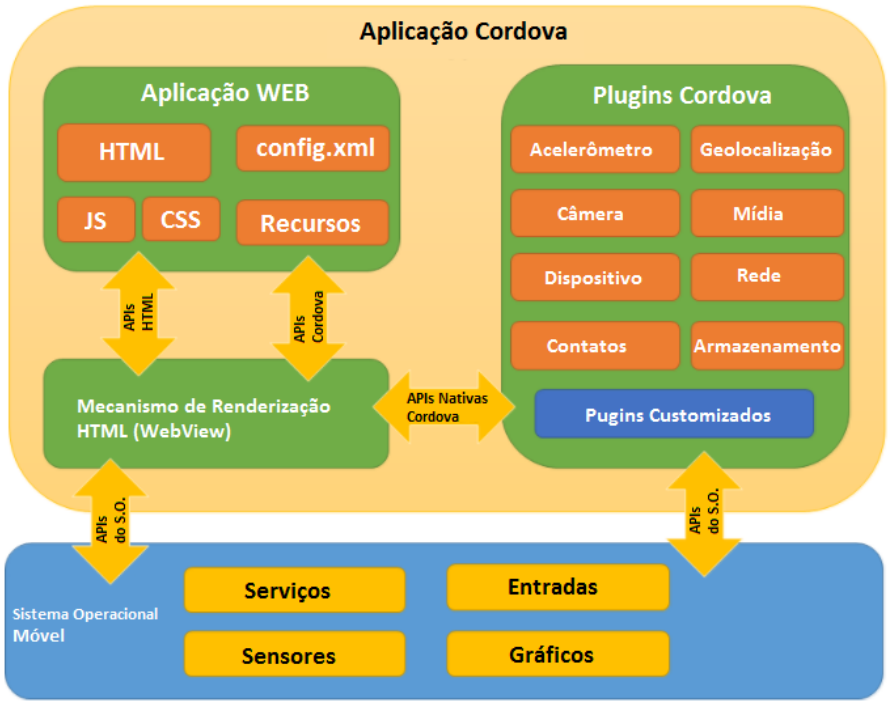

Figura 3 - Arquitetura de uma aplicação Cordova (adaptado de Cordova, 2019)

Frameworks como Ionic (Lynch, Sperry e Bradley, 2019) e PhoneGap (Systems, 2019) utilizam o Cordova para desenvolver aplicações móveis multiplataformas híbridas para os SOs mais conhecidos do mercado como Android, iOS e Windows.

2.4. TESTES AUTOMATIZADOS

Com o objetivo de garantir a qualidade do software, uma das atividades de apoio utilizada é o teste, que pode ser definido como o processo de execução de um sistema com o objetivo de encontrar erros (Goodenouth e Gerhart, 1975)(Myers, Thomas e Wiley, 2004). O teste de software é formado por casos de teste, ou seja, uma tripla de condições de entradas, condições de execução e resultados esperados, que são elaborados para identificar erros ou percorrer um caminho especificado pelo testador (IEEE, 1990). O teste de aplicações móveis é realizado por meio de atividades executadas utilizando métodos bem definidos e ferramentas que garantem a qualidade nas funcionalidades, comportamentos, desempenho, mobilidade, conectividade, segurança, usabilidade, privacidade e interoperabilidade (Gao *et al.*, 2014).

No cenário do teste manual, é exigida do testador a tarefa de verificar manualmente o comportamento do sistema, determinando se o teste falhou ou foi executado com sucesso (Barr *et al.*, 2015). Já no teste automatizado, as reproduções dos testes podem ser feitas quantas vezes forem necessárias, possibilitando a validação dos resultados gerados, aumentando a produtividade e diminuindo os custos (Oliveira, 2012).

Algumas ferramentas de testes automatizados estão disponíveis no mercado. Neste trabalho, foi adotado o *Framework* Appium que é utilizado para automatizar teste em aplicações móveis nativas, web ou híbridas. O Appium é multiplataforma e possibilita automatizar testes para as tecnologias iOS e Android usando a mesma API, permitindo assim a reutilização de código-fonte entre as plataformas. Para a realização dos testes com o Appium, o desenvolvedor não precisa recompilar a aplicação ou fazer qualquer adaptação, é necessário apenas definir o diretório e preencher as informações da aplicação que será testada que a conexão já é estabelecida. A ferramenta utiliza uma API do Selenium WebDriver, que possibilita o desenvolvimento de casos de testes em várias linguagens de programação, como Java, Ruby, C#, dentre outras (Appium, 2019).

Na Figura 4, é apresentado o código-fonte da realização do teste de adicionar um novo registro na aplicação que está sendo testada. Inicialmente são definidas as configurações do Appium (Linhas 28-31), em seguida, na linha 37 é selecionado o elemento do botão adicionar, e na linha 41 é realizado o *click* nesse botão. O próximo

passo é selecionar o campo de texto na linha 43, após essa etapa, na linha 47 é clicado no campo encontrado, e então na linha 48 é inserido o texto “Oferta 01”. Por fim, na linha 50 busca-se o botão salvar, e em seguida na linha 54 é realizado o *click* nesse botão.

```
18      [TestClass]
19      public class AddOferta
20      {
21          AppiumDriver<IWebElement> _driver = null;
22          DesiredCapabilities _capabilities = new DesiredCapabilities();
23          LocatorStrategy _locator = null;
24
25          [TestMethod]
26          public void TestMethodMain()
27          {
28              _capabilities.SetCapability("platformName", ProjectConfig.PlataformName);
29              _capabilities.SetCapability("platformVersion", ProjectConfig.PlatformVersion);
30              _capabilities.SetCapability("deviceName", ProjectConfig.DeviceName);
31              _capabilities.SetCapability("appPackage", ProjectConfig.AppPackage);
32
33              _driver = new AndroidDriver<IWebElement>(defaultUri, _capabilities, TimeSpan.FromSeconds(3000));
34              _locator = new LocatorStrategy(_driver, null);
35
36              string[] selectors = new string[0];
37              selectors = new string[] {"hierarchy/android.widget.FrameLayout/android.view.View[2]/" +
38                                        "android.widget.FrameLayout/android.widget.ImageButton"};
39              string[] selectorsType = new string[] {"AbsolutePath"};
40              IWebElement botao_add = _locator.FindElementByXPath(selectors[0], selectorsType[0]);
41              botao_add.Click();
42
43              selectors = new string[] {"hierarchy/android.widget.LinearLayout[2]/" +
44                                        "android.widget.LinearLayout/android.widget.EditText"};
45              string[] selectorsType = new string[] {"AbsolutePath"};
46              IWebElement txt_nome_oferta = _locator.FindElementByXPath(selectors[0], selectorsType[0]);
47              txt_nome_oferta.Click();
48              txt_nome_oferta.SendKeys("Oferta 01");
49
50              selectors = new string[] {"hierarchy/android.widget.LinearLayout[2]/"+
51                                        "/android.support.v7.widget.LinearLayoutCompat/android.widget.TextView"};
52              string[] selectorsType = new string[] {"AbsolutePath"};
53              IWebElement botao_salvar = _locator.FindElementByXPath(selectors[0], selectorsType[0]);
54              botao_salvar.Click();
55
56              Exec.Instance.EndSuccefull();
57          }
58      }
```

Figura 4 - Tela de configuração do Appium (Fonte própria).

2.4.1. x-PATeSCO

Segundo Menegassi (2018), as aplicações móveis multiplataforma compartilham uma dificuldade na realização de testes automatizados, pois apesar de serem muito parecidas visualmente, quando cada aplicação é compilada para a execução no SO, os elementos de UI são transformados em *Extensible Markup Language* (XML), e sua estrutura difere para cada plataforma. A Figura 5 apresenta um elemento de uma das telas da aplicação Ofertados, que após ser compilado, contém diferenças no XML para cada SO. Além da quantidade de propriedades gerada para cada plataforma, a

informação apresentada para o usuário no Android pelos atributos “*content-desc”* e *“text”,* são apresentadas no iOS pelos atributos *“label”* e “value”.

Figura 5 – Diferença entre arquivos XML do Android e iOS (Fonte Própria)

Com o objetivo de solucionar o problema encontrado nas aplicações móveis multiplataforma, a ferramenta `x-PATeSCO` define um mecanismo automático de teste para as aplicações móveis, através do desenvolvimento de *scripts* que possam ser executados em diversas configurações (Menegassi, 2018). A abordagem proposta pelos autores possui três passos principais para o funcionamento: primeiro um dispositivo móvel de referência é escolhido para cada plataforma, em seguida é definido um modelo baseado em eventos para representar os casos de testes e as expressões individuais, que são métodos utilizados para localizar os elementos de interface da aplicação móvel que está sendo testada, em seguida é definida uma estratégia para combinar as expressões individuais e gerar um único script de teste.

Os autores investigaram seis expressões individuais, que foram utilizadas para localizar os elementos de interface nas aplicações testadas, essas expressões são detalhadas a seguir:

- ***AbsolutePath***: uma expressão específica de cada plataforma, baseada no caminho absoluto do elemento dentro da estrutura XML. É uma alternativa bem conhecida quando o elemento não tem um atributo de identificação e em alguns casos, os índices são necessários para identificar a posição do elemento dentro da estrutura XML.

- ***IdentifyAttributes***: uma expressão baseada nos valores dos atributos que identificam o elemento, como *resource-id* para a plataforma Android e *name* para a plataforma iOS.
- ***CrossPlatform***: os autores propuseram esta estratégia com o objetivo de definir uma única expressão para diferentes plataformas. Essa expressão está preparada para selecionar um elemento específico da UI da aplicação móvel independente de sua plataforma de execução. Ela combina os principais atributos (Android: *content-desc* ou *text* e iOS: *label* ou *value*) de elementos e seus valores em ambas as plataformas.
- ***ElementType***: uma expressão para encontrar um elemento baseado na combinação de seus atributos (Android: *content-desc* ou *text* e iOS: *label* ou *value)* e identificadores de cada plataforma (Android: *resouce-id* ou iOS: *name*).
- ***AncestorIndex***: uma expressão específica de cada plataforma baseada no índice do elemento desejado definindo sua posição exata, neste caso, dentro do bloco anterior na hierarquia do XML.
- ***AncestorAttribute***: É semelhante à expressão anterior, mas a localização do índice é substituída por uma localização baseada nos valores dos atributos-chave.

Em seguida, os autores apresentaram duas estratégias de localização combinando essas estratégias, são elas:

- ***ExpressionsInOrder:*** As expressões são classificadas pelo tipo e executadas sequencialmente. Se a primeira expressão falhar, a próxima é executada, e assim por diante. Essa estratégia visa evitar a execução incompleta de um caso de teste devido a um elemento não encontrado. A ordem que foi definida foi *CrossPlatform*, *ElementType*, *IdentifyAttributes*, *AncestorAttributes*, *AncestorIndex*, *AbsolutePath*.
- ***ExpressionsMultiLocator:*** nesta estratégia, todas as expressões são executadas e o elemento é selecionado por critérios de votação. Esses critérios são propostos com base em estratégias de confiabilidade para determinar pesos para cada tipo de expressão. A confiabilidade e os pesos das expressões foram discutidos e definidos, atribuindo para *CrossPlatform*, *ElementType* e *AncestorAttributes* uma confiabilidade alta com peso de

0,25, com a confiabilidade média, ficou *IdentifyAttributes* com peso 0,15. Por fim com a confiabilidade baixa ficaram *AbsolutePath* e *AncestorIndex,* ambas com peso 0,05. Para a realização dessa estratégia, foi desenvolvido um algoritmo que busca os elementos utilizando as seis expressões. Para cada elemento encontrado, é calculado o peso de acordo com a expressão. Um elemento retornado por diferentes expressões tem os pesos acumulados e no final, o elemento com a maior somatória de pesos é usado na execução do caso de teste. Caso nenhum elemento seja encontrado, uma exceção é lançada.

A ferramenta criada pelos autores utiliza o Appium para conectar-se a dois dispositivos móveis, um Android e um iOS, realiza o acesso aos elementos de interface da aplicação móvel que está sendo testada e grava os eventos indicados pelo testador, salvando esta gravação em um projeto de teste no Microsoft Visual Studio. Por fim, a `x-PATeSCO` indica para o testador a aplicabilidade das seis expressões individuais para o caso de teste gravado (Menegassi, 2018).

2.4.2. TEST PATTERNS

Os *Patterns* são eventos externos realizados pelo usuário e estão diretamente relacionados à UI, como a presença da barra de ações, gaveta lateral ou eventos do sistema, tais como girar a tela, receber uma chamada ou perder a conexão com a internet (Coimbra Morgado e Paiva, 2015). As estratégias utilizadas para testar esses *Patterns* são descritas como *Test Patterns* (Morgado e Paiva, 2015).

Um aspecto importante para a execução do *Test Pattern* é a definição do catálogo de teste, ou seja, o que deve ser tratado como regra para que possa ser utilizada para a realização dos testes. Morgado e Paiva (2015b) realizaram uma pesquisa para testar erros recorrentes em aplicações móveis Android. Desta forma, desenvolveram uma ferramenta que utiliza um catálogo de *Test Patterns* baseado no SO Android. Os *Test Patterns* definidos por eles estão descritos a seguir:

- **Padrão de gaveta lateral**: um menu transitório que é aberto quando o usuário desliza a tela da esquerda para o centro ou clica no botão menu da aplicação. O teste realizado verifica se a gaveta está implementada corretamente, ou seja, se ele mantém o padrão proposto pelo SO.

- **Padrão de orientação**: ao girar o dispositivo, a tela da aplicação também gira e seu *layout* é atualizado. No entanto, existem dois aspectos principais que devem ser testados: nenhum dado de entrada do usuário deve ser perdido, ou seja, todo o conteúdo inserido pelo usuário deve estar presente após a rotação, e as informações que já estavam sendo apresentadas na tela não podem ser perdidas, como por exemplo, uma lista de registros.
- **Padrão de dependência de recursos**: muitas aplicações móveis usam recursos externos como GPS ou 3G, é importante verificar se a aplicação móvel não falha quando o recurso fica indisponível.

O estudo realizado por Adamsen, Mezzetti e Møller (2015) apresentou uma abordagem baseada na visão de que os testes existentes, se forem executados em condições adversas, podem influenciar na qualidade do software. Com isso foi avaliada, e implementada, uma ferramenta projetada para aplicações Android. Nessa ferramenta foram adicionados os seguintes *Test Patterns* durante a realização dos testes:

- **Pausa e Retorna:** ocorre quando é bloqueada a tela e em seguida volta para a aplicação móvel;
- **Pausa, Pára e Recomeça**: ocorre quando o usuário pressiona e segura o botão “home” para exibir as aplicações abertas e retorna para a aplicação que está sendo testada;
- **Pausa, Pára, Encerra e Cria:** ocorre quando o dispositivo móvel é girado ou entra em modo de carregamento de energia;
- **Solicita-Abandona o foco no áudio:** ocorre quando uma aplicação solicita o foco no controle do volume e retorna para a aplicação testada;
- **Perda de precisão do GPS:** ocorre quando o dispositivo móvel está em uma zona onde o GPS é menos preciso e em seguida volta para uma zona precisa;
- **Alteração GSM:** ocorre quando o dispositivo móvel altera a rede GSM quando está em *roaming*;
- **3G-WiFi-3G:** ocorre quando a conexão de rede muda de 3G para WiFi ou do WiFi para o 3G durante a realização do teste.

Os autores concluíram que as aplicações testadas demonstraram que a abordagem é eficaz em encontrar erros críticos. Ressaltaram também que os resultados mostram que o custo adicional no tempo de teste é baixa, comparado ao número de erros encontrados.

Linares-vásquez *et al.* (2017) realizaram uma pesquisa na literatura e após concluir que não existiam estudos em grande escala que classificassem os erros das aplicações Android, fizeram um levantamento para identificar os erros em aplicações. Com base em uma análise manual definiram um conjunto de operadores de mutação específicos do Android, implementaram uma ferramenta para inserir automaticamente mutações em aplicações Android e após a validação dos dados, concluíram que sua abordagem pode ser possivelmente ampliada e utilizada por outros pesquisadores. Com base nos erros encontrados, alguns *Test Patterns* podem ser citados:

- **Falha na localização do GPS:** ocorre quando o dispositivo móvel perde a conexão com o GPS durante a execução da aplicação;
- **Bloqueio do acesso ao Wifi:** o acesso WiFi pode ser interrompido a qualquer momento por algum evento externo durante sua utilização;
- **Perda de conexão 3G:** a conexão com a rede móvel durante a execução da aplicação pode ser interrompida por fatores como entrar em uma área com WiFi salvo ou sair de uma zona com acesso ao 3G;
- **Rotação do dispositivo:** ação de girar o dispositivo fazendo com que a tela seja redimensionada;
- **Lentidão na conexão de rede:** excesso de lentidão na conexão que pode afetar o envio e recebimento de dados acessados pela aplicação móvel.

A abordagem de Zaeem, Prasad e Khurshid (2014) abrange o problema da geração automatizada de teste para aplicações móveis. Foi realizando um estudo em erros relatados nas aplicações mais populares de código-fonte aberto que se revelou a existência de uma quantidade significativa de erros devido à interação dos usuários, que foram definidas da seguinte forma:

- **Dupla Rotação:** A ação de girar o dispositivo e em seguida girar novamente de volta para o estado inicial, e a aplicação deve manter seu estado original.

- **Encerrando e recomeçando:** O SO pode escolher encerrar e em seguida, reiniciar uma aplicação por várias razões (por exemplo, memória baixa), assim como na Dupla Rotação, a aplicação deve recuperar seu estado original.
- **Pausando e retomando:** A aplicação pode ser pausada, por exemplo, ao clicar no botão “home” e depois retomada. Essa é uma transição do ciclo de vida da atividade que todas as aplicações devem suportar.
- **Botão Voltar:** O botão Voltar nos dispositivos Android leva a aplicação móvel para a tela anterior.
- **Menus de abertura e fechamento (Menu):** o botão de menu em dispositivos Android abre e fecha menus personalizados que cada aplicação móvel define.
- **Aumentando Zoom:** O zoom em uma tela deve exibir um subconjunto do que estava originalmente na tela.
- **Diminuindo o Zoom:** Diminuir o zoom de uma tela deve resultar em um superconjunto da tela original.
- **Scroll:** O scroll deve deslocar para baixo (ou para cima) e exibir uma imagem que compartilhe partes da imagem anterior.

Com isso, foi desenvolvida uma ferramenta com o objetivo de testar exaustivamente as interações com o usuário que as aplicações possuem, alguns erros foram encontrados, e os autores consideraram necessário à avaliação de um conjunto maior de aplicações (Zaeem, Prasad e Khurshid, 2014).

Joorabchi, Ali e Mesbah (2015) desenvolveram uma ferramenta chamada *Checking Compatibility Across Mobile Platforms* (CHECKCAMP), com o objetivo de localizar e visualizar inconsistências de UI entre as versões para Android e iOS de uma mesma aplicação. Os autores apontam que uma aplicação multiplataforma deve conter a mesma funcionalidade e comportamento nas diferentes plataformas suportadas. O CHECKCAMP gera um modelo baseado na UI para cada plataforma e realiza comparações para identificar as inconsistências. Com a ferramenta foram avaliados 14 pares de aplicações multiplataforma, sendo capaz de encontrar 32 inconsistências durante a comparação das versões das aplicações nas diferentes plataformas. As inconsistências foram classificadas em ordem de impacto e estão descritas a seguir:

- **Funcionalidade:** inconsistências funcionais estão relacionadas às diferenças entre os modelos da aplicação para cada plataforma. Por exemplo, na aplicação Android, um registro não pode ser excluído enquanto na iOS existe a opção de exclusão, ou ao clicar em enviar, é solicitado a confirmação no Android, no entanto não solicita o mesmo no iOS.
- **Dados:** quando a apresentação de qualquer tipo de dados é diferente para as plataformas, tanto na ordem de exibição, texto escrito no componente ou formato da hora; por exemplo, o botão de busca possuir o texto “Buscar Eventos” no Android e no iOS apresenta o texto “Buscar”.
- **Layout:** quando um elemento de UI possui um layout diferente, tanto no tamanho, quanto na ordem ou disposição na tela. Um exemplo para essa situação é quando um botão no Android está no lado esquerdo da tela, e no iOS o botão está centralizado.
- **Estilo:** são considerados para esse item as cores, estilo do texto ou diferenças de design, por exemplo, no iOS a galeria de imagens possui um fundo azul enquanto no Android possui uma galeria com um fundo branco.

Foi proposta uma abordagem capaz de detectar automaticamente essas inconsistências. Os pesquisadores concluíram que existem oportunidades de melhoria na abordagem, uma delas é a utilização do CHECKCAMP em aplicações móveis que já possuem script de testes.

Segundo Fazzini e Orso (2017), devido à variedade de configurações dos dispositivos móveis que utilizam o Android, as aplicações multiplataforma podem conter inconsistências em seu comportamento. Para identificar essas inconsistências, os autores propuseram uma ferramenta chamada DIFFDROID. O DIFFDROID executa a aplicação móvel e gera automaticamente um conjunto de entradas de teste para a aplicação. Em seguida executa a aplicação com essas entradas em um dispositivo de referência e cria um modelo de UI. E então, é executada a aplicação com as mesmas entradas em configurações de dispositivos móveis diferentes. Finalmente, o DIFFDROID compara os modelos gerados com a referência definida inicialmente e relata as diferenças identificadas. Os tipos de inconsistências identificadas no estudo são:

- **Estrutural:** é realizada a validação da estrutura do código-fonte, calculando a semelhança e verificando se o dispositivo móvel de referência possui propriedades compatíveis com o que está em teste;
- **Visual:** é realizado o cálculo de representação visual, validando se os componentes são exibidos mantendo o mesmo padrão tanto para o dispositivo móvel de referência quanto para o que está em teste.

Os testes utilizando o DIFFDROID foram realizados em 5 aplicações móveis reais, totalizando mais de 130 combinações de configurações entre tamanho de tela e versões do Android para os dispositivos móveis testados. Os autores concluíram que a ferramenta é capaz de identificar inconsistências em aplicações multiplataforma reais de forma eficiente e com um número limitado de falsos positivos.

Holl e Elberzhager (2014) realizaram uma classificação dos erros comuns específicos para dispositivos móveis que podem ser utilizadas para focar na garantia de qualidade. Os autores inicialmente realizaram uma revisão na literatura para identificar quais pontos de vista existem em relação a erros de aplicações móveis. Além da literatura, o estudo também é baseado nas experiências feitas em projetos internos de desenvolvimento de aplicações móveis para uso comercial. Com base nos trabalhos encontrados na literatura e nas experiências de projetos, incluindo entrevistas com desenvolvedores e testadores, foram definidos os seguintes conjuntos de eventos que podem causar erros nas aplicações móveis:

- **Rede:** rede temporariamente desconectada, alteração de rede 3G para WiFi, ou WiFi para outra WiFi, ou localização física como GPS temporariamente desconectado;
- **Energia:** bateria fraca; alto consumo de energia devido a excesso de utilização de memória; sem bateria;
- **Configuração:** configurações sobre localidade, idioma, data ou hora diferente entre o cliente e o servidor;
- **Interface de Usuário:** alteração entre a visualização vertical e horizontal, alterações entre resolução de tela, funcionalidades nativas do SO como o botão voltar ou o home;
- **Interrupções:** recebimento de ligação, mensagens de texto ou mensagens padrões do SO;

- **Uso de Dados:** termino inesperado em um processo da aplicação, possibilidade de acessar informações sem permissão, função de envio de informações acidentalmente.

Os autores concluíram que essa classificação pode ser avaliada e otimizada para aplicar garantia de qualidade das aplicações móveis.

2.5. CONSIDERAÇÕES FINAIS

Neste capítulo, foram introduzidos os principais conceitos e características sobre a computação móvel e aplicações multiplataformas. Foram descritas também algumas ferramentas para desenvolvimento dessas aplicações. Foram apresentados os principais conceitos sobre teste de software, e alguns *Test Patterns* para aplicações móveis que foram propostos na literatura.

No próximo capítulo, é apresentada uma abordagem de amplificação de teste de software para aplicações móveis multiplataforma baseada nos *Test Patterns* encontrados na literatura, a qual é a proposta desta dissertação.

3. ABORDAGEM PROPOSTA

Neste capítulo é apresentada a abordagem proposta. Na Seção 3.1 são apresentados os estudos contendo os *Test Patterns* encontrados na literatura. Na Seção 3.2 são apresentadas as definições de cada *Test Pattern*. Na Seção 3.3 é descrita a implementação dos testes automatizados contendo a amplificação.

O objetivo da abordagem proposta é a amplificação de casos de testes existentes e sua execução de forma automatizada em aplicações multiplataforma. A Figura 6 demonstra as seguintes etapas: a) Criação de *scripts* automatizados dos *Test Patterns* encontrados na literatura. b) Aplicação dos *scripts* dos *Test Patterns* nos casos de teste funcionais existentes. c) Geração dos casos de teste com os *scripts* amplificados. d) Execução de casos de testes amplificados nas aplicações móveis multiplataforma.

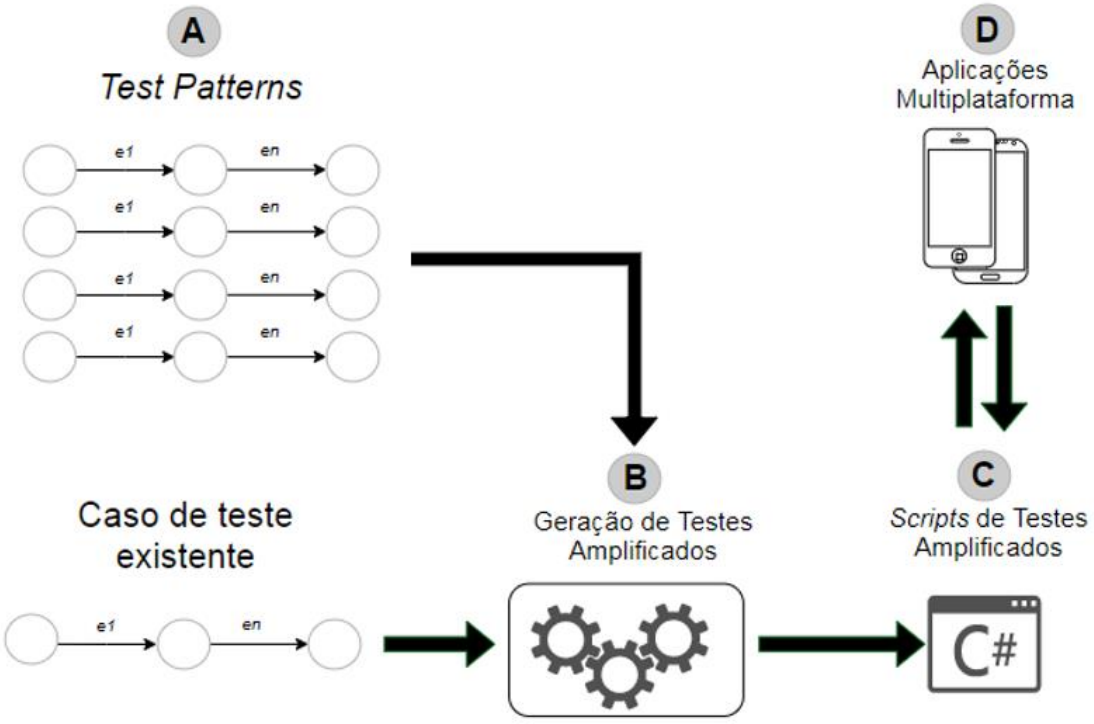

Figura 6 - Visão geral da abordagem

3.1. ELICITAÇÃO DOS TEST PATTERNS NA LITERATURA

O levantamento na literatura foi baseado principalmente no estudo de Zein, Salleh e Grundy (2016). Neste estudo são apresentados diversos *Test Patterns*, que foram analisados e sumarizados. Na Tabela 1 foram agrupados os os estudos contendo *Test Patterns* encontrados na literatura, e apresenta uma breve descrição da ação realizada para a execução do teste além das referências separadas pelas seguintes classes:

Rotação: ações de rotação de tela dos dispositivos móveis. Durante a execução dos casos de teste, os dispositivos podem ser girados sem gerar erros;

Exibição: ações como aumentar o zoom ou redimensionar uma tela, não podem gerar erros;

Conexão: nesta classe são avaliadas interrupções de serviços externos como o bloqueio do acesso a conexão com a internet durante a execução de um caso de teste;

Padrões de UI: a classe de padrões de interface foi dividida em duas, pois os SOs possuem suas particularidades, como padrões de menus e botões. No entanto algumas funcionalidades são similares como o botão *home*;

SOs: ações como colocar a aplicação em segundo plano e em seguida retomar a aplicação, ou configurações de idioma, data e hora diferentes podem ser testadas.

Baseado nessas informações coletadas, foram definidos quatro *Test Patterns* para serem utilizados nesse estudo. Foi selecionado pelo menos um *Test Pattern* de cada classe, além da possiblidade de serem testados tanto no Android quanto no iOS. O primeiro foi o *Lost Connection*, encontrado na classe Conexão. O próximo *Test Pattern* definido foi o *Back Event* e este por sua vez pode ser encontrado na classe de Padrões de Interface, tanto no subitem Android quanto no iOS. O *Side Drawer Menu* também está na classe de Padrões de Interface, e apesar de não aparecer em nenhum dos estudos a realização de testes para o iOS, ele pode ser testado em ambos SOs. Para as classes SOs, Exibição e Rotação, foi definido o *Test Pattern Don't Change State*.

Tabela 1 - *Test Patterns* mais citados na literatura

Classe		Nome	Descrição Geral	Referência
Rotação		Orientação da Tela	Ao girar o dispositivo, a tela da aplicação também gira e seu layout é atualizado.	Morgado e Paiva (2015b), Linares-vásquez et al. (2017), Zaeem, Prasad e Khurshid (2014), Holl e Elberzhager (2014)
		Dupla Rotação	Girar o dispositivo e em seguida girar novamente de volta para o estado inicial.	Zaeem, Prasad e Khurshid (2014)
Exibição		Aumentando Zoom	O zoom em uma tela deve exibir um subconjunto do que estava originalmente na tela.	Zaeem, Prasad e Khurshid (2014)
		Diminuindo o Zoom	Diminuir o zoom de uma tela deve resultar em um superconjunto da tela original.	Zaeem, Prasad e Khurshid (2014)
		Redimensionamento de tela	Aumentar o diminuir o tamanho da tela.	Linares-vásquez et al. (2017), Holl e Elberzhager (2014)
		Scroll	O scroll deve deslocar para baixo (ou para cima) e exibir uma imagem que compartilhe partes da imagem anterior.	Zaeem, Prasad e Khurshid (2014)
Conexão		Perda de precisão do GPS	Dispositivo móvel entra em uma zona onde o GPS é menos preciso ou sem sinal algum e em seguida volta para uma zona precisa.	Adamsen, Mezzetti e Møller (2015), Morgado e Paiva (2015b), Linares-vásquez et al. (2017), Holl e Elberzhager (2014)
		Alteração GSM	Dispositivo móvel altera a rede GSM quando está em roaming.	Adamsen, Mezzetti e Møller (2015), Holl e Elberzhager (2014)
		3G-WiFi-3G	Quando a conexão de rede muda de 3G para WiFi ou do WiFi para o 3G.	Adamsen, Mezzetti e Møller (2015), Morgado e Paiva (2015b), Holl e Elberzhager (2014)
		Bloqueio do acesso ao Wifi	Acesso ao WiFi bloqueado durante a realização dos testes.	Linares-vásquez et al. (2017)
		Lentidão na conexão de rede	Conexão lenta.	Linares-vásquez et al. (2017), Holl e Elberzhager (2014)
		Dependência de recursos	Perca de recursos externos como GPS ou 3G durante a execução da aplicação móvel.	Morgado e Paiva (2015b), Linares-vásquez et al. (2017), Holl e Elberzhager (2014)
Padrões de Interface	Android	Gaveta lateral	Um menu transitório que é aberto quando o usuário desliza a tela da esquerda para o centro ou clica no botão menu da aplicação.	Morgado e Paiva (2015b)
		Botão Voltar	Botão Voltar dos dispositivos Android, que leva a aplicação móvel para a tela anterior.	Zaeem, Prasad e Khurshid (2014), Holl e Elberzhager (2014)
		Menus de abertura e fechamento	Botão de menu dos dispositivos Android, que abre e fecha menus personalizados que cada aplicação móvel define.	Zaeem, Prasad e Khurshid (2014), Holl e Elberzhager (2014)
	iOS	Barra de navegação	A barra de navegação possui o botão voltar nos dispositivos iOS, levando a aplicação móvel para a tela anterior	Apple, 2017b
		Botão Home	Além de outras funções como leitor de digital o botão home do iOS leva a aplicação móvel para o segundo plano exibindo a página inicial do dispositivo	Apple, 2017a
SOs		Encerrar e recomeçar	O SO encerra e em seguida reinicia a aplicação.	Zaeem, Prasad e Khurshid (2014), Zaeem, Prasad e Khurshid (2014), Holl e Elberzhager (2014)
		Pausa e Retorna	Bloqueio da tela e em seguida volta para a aplicação móvel; receber uma ligação, SMS ou alerta padrão do SO	Adamsen, Mezzetti e Møller (2015), Zaeem, Prasad e Khurshid (2014), Holl e Elberzhager (2014)
		Pausa, Para e Recomeça	Pressionar e segurar o botão "home" para exibir as aplicações abertas e retornar para a aplicação que está sendo testada	Adamsen, Mezzetti e Møller (2015), Holl e Elberzhager (2014)
		Solicita-Abandona o foco no áudio	Outra aplicação solicita o foco no controle do volume e retorna para a aplicação testada.	Adamsen, Mezzetti e Møller (2015), Holl e Elberzhager (2014)
		Pausa, Para, Encerra e Cria	Girar o dispositivo móvel ou entrar em modo de carregamento de energia.	Adamsen, Mezzetti e Møller (2015), Holl e Elberzhager (2014)
		Configuração	Configurações sobre localidade, idioma, data ou hora diferente entre o cliente e o servidor.	Holl e Elberzhager (2014)

3.2. TEST PATTERNS

Esta seção descreve os quatro padrões de teste adotados neste trabalho. Todos os padrões de teste assumem que um caso de teste é fornecido como uma sequência de eventos, ou seja, ações da UI, como clicar em um botão, selecionar um item ou preencher um campo de texto. A Figura 7 apresenta um caso de teste original que transita entre três telas; a tela `T1orig`, em seguida a `T2orig` e por fim acessa a tela `T3orig`. Neste caso a aplicação acessa três telas diferentes e encerra o caso de teste. Na transição de cada estado, existe um evento. No caso do exemplo abaixo. `e1` e `e2` são eventos realizados para mover o usuário para uma próxima tela. O `e1` move da `T1orig` para a `T2orig` e o `e2` redireciona da `T2orig` para a `T3orig`.

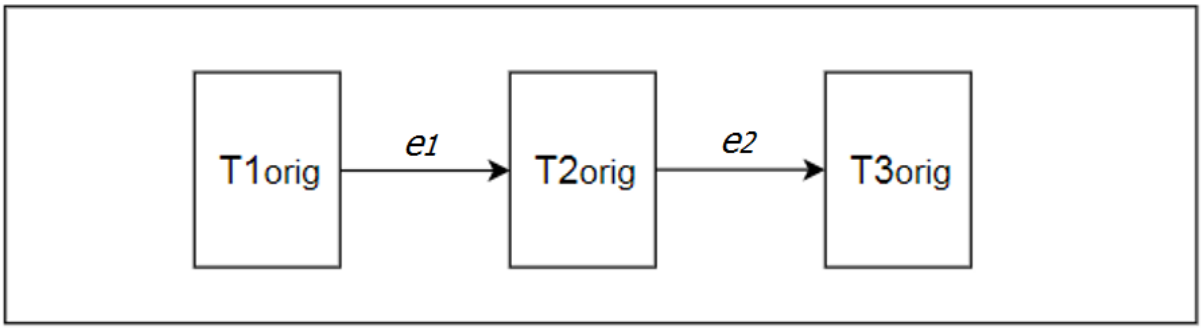

Figura 7 - Caso de teste original

Após a definição dos quatro *Test Patterns* adotados, foi verificado que os *scripts* necessários para a automatização dos testes não poderiam possuir apenas uma sequência predefinida de eventos específicos a serem executados, mas sim um mecanismo para identificar erros em potencial. Com isso, para cada *script* foram desenvolvidas lógicas de código-fonte específicas. As exceções são capturadas registrando as informações dos erros em um log. Os detalhes de cada *Test Patterns* estão descritos a seguir:

Lost Connection. A aplicação que está sendo testada é validada através de seu comportamento, verificando sua reação quando a conexão com a Internet é perdida (Holl e Elberzhager, 2014; Morgado e Paiva, 2015; Linares-vásquez *et al.*, 2017). Como os dispositivos móveis são suscetíveis a redes instáveis, a aplicação deve estar preparada para uma queda de conexão. Como retorno, assume-se que

alguma mensagem deve ser fornecida ao usuário final, exibindo mensagem de erro ou aviso (Android Core App Quality, 2019; Apple, 2019).

É possível verificar na Figura 8 a representação da amplificação de testes utilizando o *Test Pattern Lost Connection*. O caso de teste amplificado com o *Lost Connection* inicia desligando a conexão com a Internet do dispositivo móvel e em seguida acessa a primeira tela do sistema `T1LC`, validando assim se as telas possuem o mesmo estado antes e depois da perda de conexão (`T1LC = T1orig`). Com essa validação, a amplificação verifica se é possível acessar essa tela sem erros mesmo se a conexão for perdida. Em seguida, a aplicação é reiniciada, o Wifi é religado, e só é desligado novamente antes de acessar uma tela que não foi testada, no caso a `T2LC`. Novamente é validado se, sem conexão as telas possuem o mesmo estado (`T2LC = T2orig`). O mesmo procedimento é realizado para a tela `T3LC`. Caso uma das validações verifique que os estados das telas não conferem, ou seja, o sistema se comporta de maneira diferente existindo a perda de conexão, é verificado se a aplicação móvel exibe alguma mensagem informando que houve falha na conexão, caso não seja exibido, um log é gravado. Mesmo ocorrendo alguma falha, após a gravação do log, a aplicação móvel é reiniciada e a realização dos testes continua para a próxima tela.

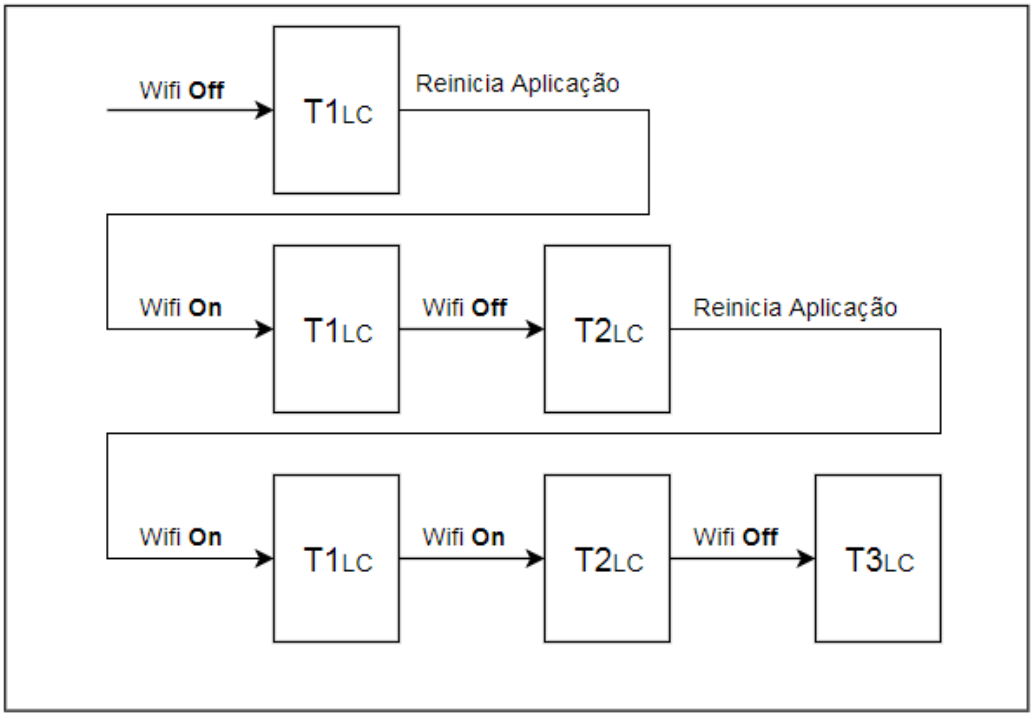

Figura 8 - Visão geral do *Lost Connection*

O Algoritmo 1 apresenta um pseudocódigo do *script* do *Lost Connection.* Na Figura 9 é possível observar que inicialmente o método recebe duas listas de informações (Linha 2 e 3), onde são recebidos todos os eventos do caso de teste que está sendo amplificado e todos os estados das telas que o sistema passou durante a execução do caso de teste antes da amplificação. Em seguida é desligada a conexão com a internet (Linha 4). O *script* então inicia a aplicação que está sendo testada (Linha 5) e em seguida percorre todos os eventos verificando se os estados do sistema que está sendo testado sem a conexão com a internet são iguais aos estados que foram testados com a conexão (Linhas 6-17). Além disso, o *script* verifica se o sistema possui alguma mensagem para o usuário, ou seja, se os estados estiverem diferentes, mas existir uma mensagem informando a falta de conexão com a internet, o *script* não considera como erro (Linha 9). Em algumas aplicações, só é possível acessar uma tela, se houver conexão com a internet para chegar até ela, por isso, o *script* ao final de cada validação de estado, religa a conexão com a internet, reinicia a aplicação e executa os eventos até onde foi testado, em seguida desliga novamente a conexão e segue com os testes para o próximo evento (Linha 12, 13 e 14).

Algoritmo 1 Lost Connection

```
1: procedure LostConnection(allEvents[], origStates[])

2:     input allEvents[] - O caso de teste é um array de eventos
3:     input origStates[] - Mapeamento dos estados coletados na execução original

4:     connection( OFF );
5:     startApp();
6:     for each event ∈ allEvents do
7:         currentState = getActualState();
8:         if currentState != origStates[event] then //Alteração de layout
9:             if ! existLostConnectionMsg(currentState) then // Nenhum retorno
10:                RecordBugInfo();
11:            end if
12:            connection( ON );
13:            bringAppTo( currentState );
14:            connection( OFF );
15:        end if
16:        executeEvent(event);
17:    end for
18: end procedure
```

Figura 9 - Algoritmo do *Pattern Lost Connection*

Back Event. Seu objetivo é avaliar como a aplicação se comporta quando o evento *Back* é acionado durante um caso de teste (Holl e Elberzhager, 2014; Zaeem, Prasad e Khurshid, 2014). Assume-se que, quando em execução, o evento *Back* deve ir para uma tela anterior da aplicação móvel (Android Core App Quality, 2019; Apple, 2019).

Na Figura 10, é possível observar como funciona a amplificação dos testes para o *Back Event*. Neste caso é iniciado validando a primeira tela, chamada de `T1BE`. Assim que a aplicação móvel carrega a primeira tela, é executada a ação *Back Event* e é validado se o fato de executar essa ação faz com que o dispositivo móvel exiba a tela inicial do SO. Em seguida é reiniciada a aplicação. O sistema transita pela tela já testada até chegar na segunda tela, gravando os estados das telas, até chegar na `T2BE`, e novamente é executada a ação *Back Event*. Desta vez, é validado se ao executar o evento *Back*, a aplicação retornou corretamente para a tela anterior, no caso a `T1BE`. Com isso é validado se a ação do *Back Event* está redirecionando para a tela correta. Da mesma forma, o processo é repetido para a tela `T3BE`, transitando pelas telas já testadas e conferindo os estados das telas com relação a execução do evento na última tela. Caso seja identificado que a aplicação não retornou para a tela correta, é gravado um log informado o erro.

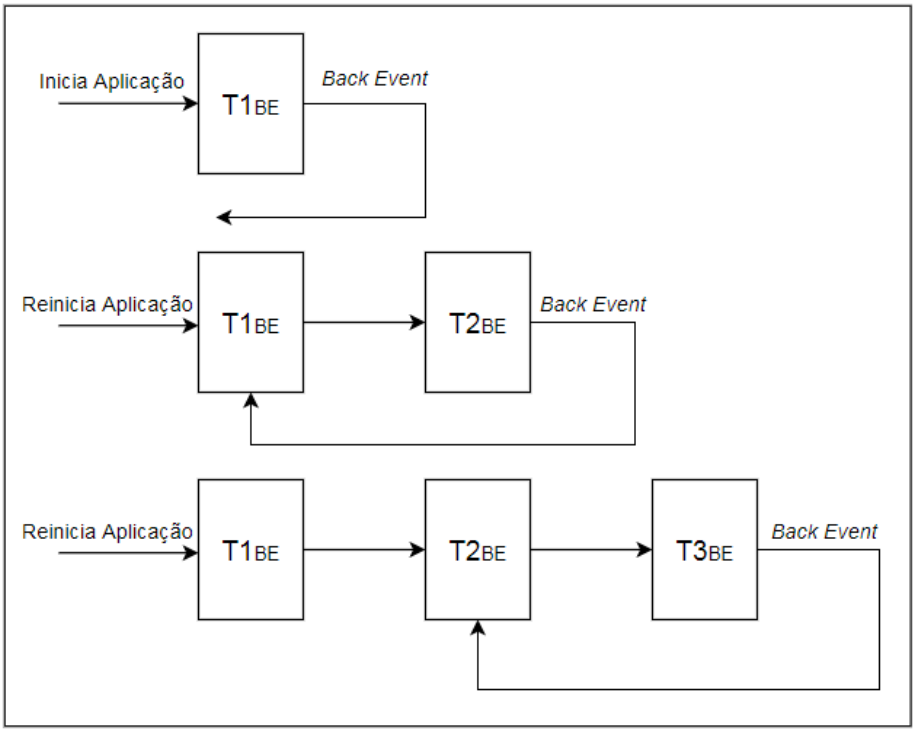

Figura 10 - Visão Geral do *Back Event*

A Figura 11 contém o algoritmo 2, que descreve como o caso de teste é amplificado para verificar a execução do *Back Event*. O *script* recebe como entrada um uma lista de eventos que foram executados no caso de teste original. Em seguida, o *script* inicia a aplicação (linha 3). Em cada iteração, é executado o evento correspondente, então é verificado se o estado após a execução do evento é igual ao estado antes da execução (Linha 8). O procedimento executa o evento *back* (voltar) e valida se a aplicação móvel retornou ao estado anterior (Linhas 9-11). Se os estados forem diferentes depois de executar o evento *back*, é gerado um log desse cenário como um erro (Linha 12). Se os estados forem iguais, o procedimento seguirá a verificação do próximo evento.

Algoritmo 2 Back Event

```
1: procedure BACKEVENT(allEvents[])
2:     input allEvents[] - O caso de teste é um array de eventos
3:     startApp();
4:     for each event ∈ allEvents do
5:         beforeState = getActualState();
6:         executeEvent(event);
7:         afterState = getActualState();
8:         if beforeState != afterState then // transição de telas
9:             ExecuteBack();
10:            afterBackState = getActualState();
11:            if beforeState != afterBackState then // erro detectado
12:                RecordBugInfo();
13:            end if
14:            bringAppTo( afterState );
15:        end if
16:    end for
17: end procedure
```

Figura 11 - Algoritmo do *Pattern Back Event*

Side Drawer Menu. Este *Pattern* tem como objetivo verificar como a aplicação que está sendo testada se comporta ao ter um menu lateral (Zaeem, Prasad e Khurshid, 2014; Linares-vásquez *et al.*, 2017). O menu é aberto quando o usuário desliza a tela para o centro ou clica em um ícone de menu (Morgado e Paiva, 2015). Com isso, foram validados se todos os itens de menu levam a aplicação para uma tela diferente da anterior.

Para o *Side Drawer Menu* foi realizado o teste validando a existência de um menu lateral e se todos os itens do menu, transitavam para alguma tela diferente da tela atual. Como é possível observar na Figura 12 após iniciar a aplicação, é verificado na primeira tela (`T1SDM`) se existe um menu, e se ele transita corretamente para outras telas (Valida Menu). Em seguida, caso haja alguma inconsistência é reiniciada a aplicação, transitando para uma tela que ainda não foi testada, no caso a segunda tela (`T2SDM`) e novamente faz a validação do menu, caso não possua inconsistência, segue para a próxima tela validando o próximo item. E por último, é transitado até a tela `T3SDM` e valida o menu como foi feito para as telas anteriores.

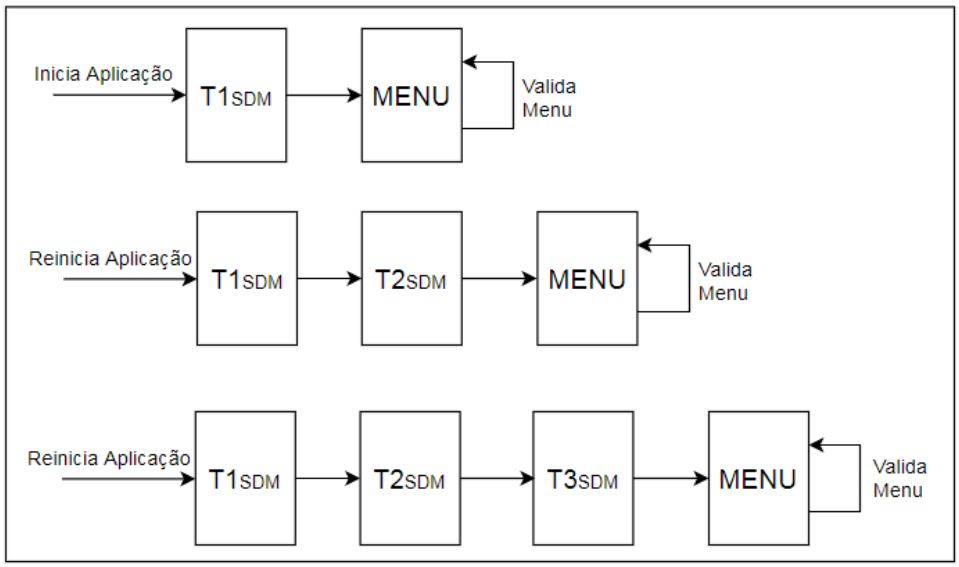

Figura 12 - Visão Geral do Side Drawer Menu

O algoritmo 3 descreve como o caso de teste é amplificado para validar os menus da aplicação móvel. É possível observar na Figura 13 que o *script* recebe como entrada um caso de teste com uma matriz de eventos. Inicialmente, o procedimento inicia a aplicação (linha 3). Para cada iteração, é verificado se a aplicação móvel possui um menu (linhas 6-23). Se houver um menu na aplicação que está sendo testada, o procedimento verifica os itens de menu empiricamente, clicando neles e verificando se a aplicação móvel altera a tela após acessar um item de menu (Linhas 28-33). Se a aplicação móvel não mudar de tela, registra-se esse cenário como um erro (Linha 31). Se nenhum item de menu for encontrado, o procedimento prossegue para a próxima iteração (Linha 35).

Algoritmo 3 Side Drawer Menu

```
1: procedure SIDEDRAWERMENU(allEvents[])
2:     input allEvents[] - O caso de testes é um array de eventos
3:     startApp();
4:     for each event ∈ allEvents do
5:         beforeMenuState = getActualState();
6:         haveMenu = false;
7:
8:         //Tenta localizar um o menu com o swipe horizontal
9:         initialState = beforeMenuState;
10:        Swipe();
11:        finalState = getActualState();
12:        if initialState != finalState then
13:            haveMenu = true;
14:        else
15:            //Tenta localizar um menu pelo botão
16:            initialState = getActualState();
17:            LocateAndClickMenuButton();
18:            finalState = getActualState();
19:
20:            if initialState != finalState then
21:                haveMenu = true;
22:            end if
23:        end if
24:
25:        if haveMenu then
26:            menuState = getActualState();
27:            for each item ∈ getMenuItems() do
28:                tap( item );
29:                afterTapItemState = getActualState();
30:                if afterTapItemState ∈ {beforeMenuState, menuState} then
31:                    RecordBugInfo();
32:                end if
33:                bringAppTo(menuState);
34:            end for
35:            bringAppTo(beforeMenuState);
36:        end if
37:        executeEvent(event);
38:    end for
39: end procedure
```

Figura 13 - Algoritmo do *Pattern Side Drawer Menu*

Don't Change State. Tem como objetivo verificar como a aplicação se comporta quando eventos externos acionados pelo usuário ocorrem durante o caso de teste (Zaeem, Prasad e Khurshid, 2014; Adamsen, Mezzetti e Møller, 2015; Morgado e Paiva, 2015). Como os dispositivos móveis são suscetíveis a essas ações, a aplicação deve estar preparada e evitar erros devido a eventos externos, portanto foi assumido que após a execução de eventos externos, a aplicação móvel deve retornar ao estado anterior (Android Core App Quality, 2019; Apple, 2019).

Para o *Don't Change State*, é validado em cada evento executado pelo caso de teste, se o estado permanece o mesmo. Por exemplo, na Figura 14, pode-se observar que a aplicação inicia, na primeira tela (`T1DCS`) e já valida o *Test Pattern*. Essa

validação pode ser, rotacionar o dispositivo, deixa-lo em segundo plano e depois retornar para a aplicação, dar um *zoom*, um *swipe* ou um *scroll* na tela. Em seguida é validado se os estados conferem, ou seja, se o estado da primeira tela com o *Test Pattern* é igual ao estado da primeira tela com o caso de teste original (`T1DCS = T1orig`). O mesmo é feito para os outros eventos do caso de teste. Caso exista alguma diferença entre os estados comparando com o caso de teste original, é gravado um log informando o erro.

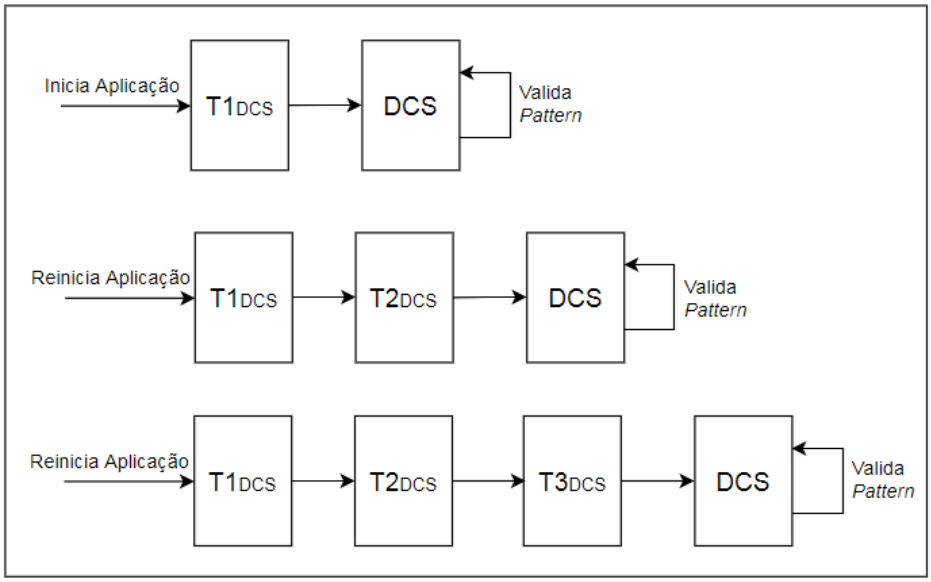

Figura 14 - Visão Geral do Don't Change State

Na Figura 15, é possível verificar o algoritmo 4, que descreve como um caso de teste é amplificado para verificar os eventos externos realizados pelos usuários. Ele recebe como entrada uma lista de eventos e o tipo da ação externa a ser executada. Os tipos atualmente suportados são Rotacionar, Pausa-Para-Retorna, *Zoom*, *Swipe* e *Scroll*. Primeiro, o procedimento inicia a aplicação (linha 4). Para cada iteração, registra o estado antes e depois do evento externo (linhas 7-9). Se os estados forem diferentes (a aplicação não retornou para mesma tela com os mesmos dados) (Linha 10), registra-se o cenário como um erro (Linha 11). O procedimento continua então para o próximo evento (Linha 12).

Algoritmo 4 Don't Change State

```
1: procedure DONTCHANGESTATE(allEvents[], type)

2:     input allEvents[] - O caso de teste é um array de eventos
3:     input type ∈ {Rotacionar, PausaParaRetorna, Zoom, Scroll, Swipe}

4:     startApp();
5:     for each event ∈ allEvents do
6:         executeEvent(event);
7:         beforeState = getActualState();
8:         execute( type ); //executa um dos 5 tipos
9:         afterState = getActualState();
10:        if beforeState != afterState then //não voltou para a mesa tela
11:            RecordBugInfo();
12:            bringAppTo( beforeState );
13:        end if
14:    end for
15: end procedure
```

Figura 15 - Algoritmo do *Pattern Don't Change State*

3.3. IMPLEMENTAÇÃO

Após o levantamento dos *Test Patterns* e o desenvolvimento dos *scripts* para a execução de forma automática desses eventos, foi necessário criar um componente para automatizar a amplificação. Com esse objetivo, o componente foi desenvolvido utilizando APIs do Appium, que fazem a conexão entre a ferramenta `x-PATeSCO` e os dispositivos móveis. O componente foi denominado como `Test Amplificator Generator (TAG)`, que tem como objetivo utilizar os *scripts* de testes já gravados no `x-PATeSCO` (Endo, Menegassi e Assis, 2019) e adicionar os *scripts* criados para os *Test Patterns*. Desta forma, os casos de teste atuais possuem eventos que não foram testados anteriormente, e os casos de teste gravados futuramente também poderão utilizar essa amplificação de forma automática.

Como a `x-PATeSCO` gera um projeto de teste para o Microsoft Visual Studio, o `TAG` foi desenvolvido utilizando a mesma plataforma, mantendo o C# como linguagem de programação. Como os *scripts* são capazes de executar automaticamente os quatro *Test Patterns* no caso de teste original, nenhum esforço manual é necessário para aplicar a amplificação de teste em aplicações móveis multiplataforma.

Na figura 16a é possível observar um trecho de código-fonte do caso de teste desenvolvido na ferramenta `x-PATeSCO`, na linha 47 está presente o comando que

insere valor no campo e na linha 60 o código-fonte faz com que seja clicado em salvar. Na figura 16b existe o script com o código para que o Appium rotacione a tela do dispositivo móvel durante a execução do caso de teste. Por fim na realização do teste o estado da interface deve ser checado para garantir que não há alteração no caso de teste após a rotação do dispositivo.

a)

```
36      string[] selectors = new string[0];
37
38      selectors = new string[] {"hierarchy/android.widget.widget.LinearLayout/"+
39                                "android.widget.LinearLayout/android.widget.EditText"};
40
41      string[] selectorsType = new string[] {"AbsolutePath"};
42
43      IWebElement e = _locator.FindElementByXPath(selectors[0], selectorsType[0]);
44
45      // Clica no elemento e escreve "Oferta 01"
46      e.Click();
47      e.SendKeys("Oferta 01");
48      try { _driver.HideKeyboard(); } catch {}
49
50      selectors = new string[0];
51
52      selectors = new string[] {"hierarchy/android.widget.FrameLayout/"+
53                                "android.support.v7android.widget.TextView"};
54
55      selectorsType = new string[] {"AbsolutePath"};
56
57      IWebElement e = _locator.FindElementByXPath(selectors[0], selectorsType[0]);
58
59      // Clica no botão salvar
60      e.Click();
```

b)

```
36      string[] selectors = new string[0];
37
38      selectors = new string[] {"hierarchy/android.widget.widget.LinearLayout/"+
39                                "android.widget.LinearLayout/android.widget.EditText"};
40
41      string[] selectorsType = new string[] {"AbsolutePath"};
42
43      IWebElement e = _locator.FindElementByXPath(selectors[0], selectorsType[0]);
44
45      // MUDANDO A ORIENTAÇÃO PARA LANDSCAPE
46      _driver.Orientation = ScreenOrientation.Landscape;
47      // -------------------------------
48
49      // Clica no elemento e escreve "Oferta 01"
50      e.Click();
51      e.SendKeys("Oferta 01");
52      try { _driver.HideKeyboard(); } catch {}
53
54      // MUDANDO A ORIENTAÇÃO PARA PORTRAIT
55      _driver.Orientation = ScreenOrientation.Portrait;
56      // -----------------------------------------------
57
58      selectors = new string[0];
59
60      selectors = new string[] {"hierarchy/android.widget.FrameLayout/"+
61                                "android.support.v7android.widget.TextView"};
62
63      selectorsType = new string[] {"AbsolutePath"};
64
65      IWebElement e = _locator.FindElementByXPath(selectors[0], selectorsType[0]);
66
67      // Clica no botão salvar
68      e.Click();
```

Figura 16 - Caso de teste com *script* amplificado.

Na Figura 17 é possível observar o caso de teste original gravado na x-PATeSCO, esse teste realiza um cadastro na aplicação acionando as ações listadas. As ações definidas na ferramenta formam: 1) Clicar no botão Menu. 2) Clicar no botão novo registro. 3) Inserir texto “Thiago” no campo nome. 4) Inserir o texto “a@a.com” no campo e-mail. 5) Inserir o texto “123456” no campo telefone. 6) Clicar no botão salvar. 7) Clicar no botão ok.

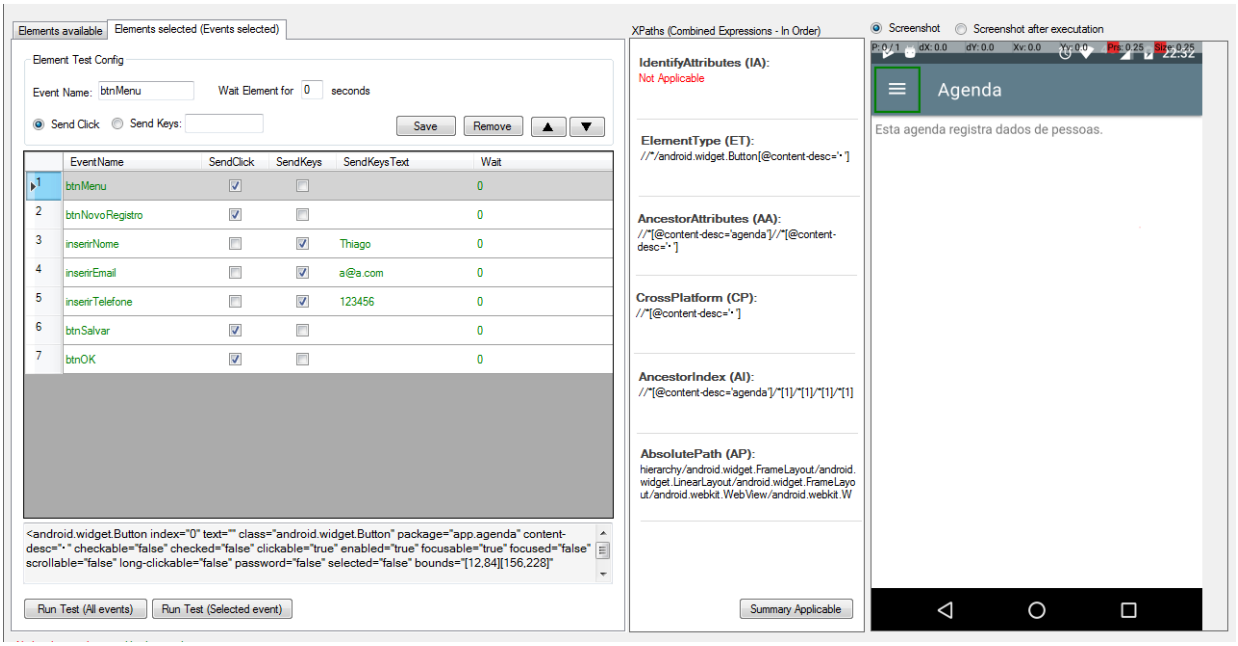

Figura 17 - Caso de teste original

Na Figura 18 é apresentado o módulo da fermenta PATeSCO onde foi incluída a funcionalidade de automatização das amplificações baseadas nos *Test Patterns*, o TAG é a opção que usuário possui de amplificar os testes originais. Na área *Test Pattern Engine* é possível selecionar qual amplificação o projeto gerado deve possuir, ou simplesmente não amplificar aquele projeto. Ao selecionar a opção *Not Apply* a ferramenta gera o projeto contendo o caso de teste original, sem nenhuma amplificação. Caso seja selecionado qualquer outra opção dentro deste grupo de amplificações, o projeto é gerado contendo o caso de teste amplificado com os *scripts* do *Test Pattern* selecionado. Todos os casos de teste originais, não possuem erros, portanto cada erro encontrado com a amplificação é algo novo para qualquer uma das aplicações móveis testadas.

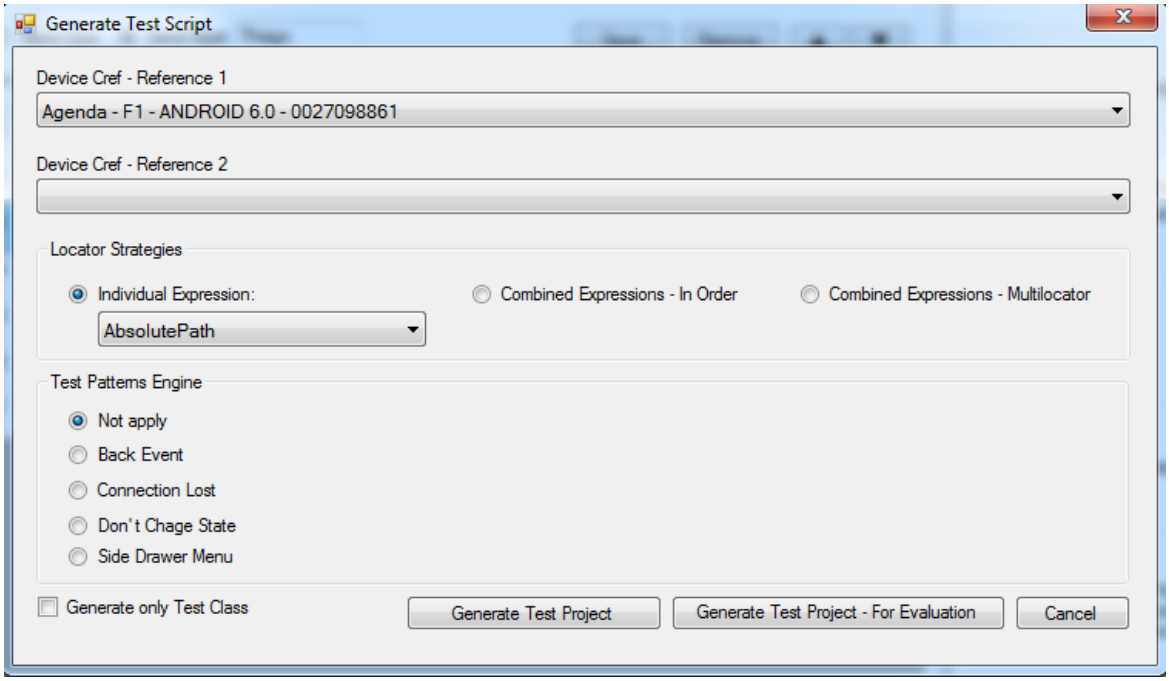

Figura 18 - Geração do projeto de testes

3.4. CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentadas a abordagem e a implementação da ferramenta, a qual são baseadas nas etapas de criação de *scripts* automatizados dos *Test Patterns* encontrados na literatura, aplicação desses *scripts* nos casos de teste funcionais existentes, geração dos casos de teste com os *scripts* amplificados e execução dos casos de testes nas aplicações móveis multiplataforma.

No próximo capítulo, é apresentada avaliação dessa abordagem e da ferramenta utilizada para apoia-la, com o objetivo de investigar a capacidade da contribuição para a área de teste de software.

4. AVALIAÇÃO EXPERIMENTAL

Este capítulo apresenta o estudo experimental conduzido para avaliar a abordagem implementada em uma ferramenta para gerar casos de testes amplificados. As descrições das etapas conduzidas neste estudo experimental são apresentadas a seguir. Na Seção 4.1 são apresentadas definições do experimento, contendo as Questões de Pesquisa (QPs) a serem respondidas e suas métricas para avaliação da abordagem. A Seção 4.2 apresenta o processo de execução do experimento e os recursos utilizados. Na Seção 4.3 é apresentada a análise dos resultados. Na Seção 4.4, é apresentada a discussão dos resultados. Na Seção 4.5, são apresentadas as possíveis ameaças à validade desta pesquisa. Por fim, na Seção 4.5 são apresentadas as considerações finais.

4.1. DEFINIÇÃO DO ESTUDO

Para validar a abordagem de amplificação de testes proposta, foi necessário entender o desempenho da ferramenta em diferentes aplicações móveis, considerando o tempo de execução e a quantidade de erros encontrados, verificando o resultado obtido em várias configurações. Para isso foi realizado uma pesquisa experimental, que é um dos principais métodos para condução de ensaios em Engenharia de Software (Wohlin *et al.*, 2012). Portanto foi conduzido um estudo experimental para comparar a execução dos casos de testes originais e amplificados com o objetivo de responder as seguintes Questões de Pesquisa (QP):

- **QP1:** É possível detectar erros em aplicações multiplataformas utilizando a amplificação de testes baseada nos *Test Patterns*?

- **QP2:** Qual é o impacto no tempo de execução dos casos de testes amplificados por *Test Patterns*?

- **QP3:** Os resultados das amplificações são replicáveis em diferentes configurações? (p. ex., Versão do SO)

Para responder a **QP1** foram registradas a quantidade de erros únicos detectados após a amplificação dos testes. Para cada falha encontrada, foram realizados testes manuais nos dispositivos móveis seguindo as mesmas ações executadas pela amplificação automatizada, desta forma, apenas as inconsistências que também ocorreram manualmente foram consideradas erros, todas as outras foram classificados como falsos positivos.

Na **QP2**, foram medidos os tempos de execução dos casos de testes sem a amplificação proposta e em seguida o tempo de execução dos testes amplificados. A diferença entre o tempo de ambos foi analisada para responder essa questão.

Finalmente, a **QP3** analisa os resultados coletados na **QP1** a partir da perspectiva de diferentes configurações de dispositivos móveis para o Android e iOS.

4.2. PROCEDIMENTO

O experimento foi realizado utilizando como base o estudo de Menegassi (2018), onde foram utilizadas as aplicações multiplataformas listadas na Tabela 2. Para cada aplicação, é apresentado se o código-fonte é aberto ou industrial, sendo que duas aplicações são industriais (MemesPlay e Bargains) e foram fornecidos por empresas parceiras. Os outros sete são projetos de código-fonte aberto obtidos no GitHub. Essas aplicações móveis multiplataforma são implementadas em diferentes *frameworks* (Apache Cordova, React Native e Xamarin). Elas variam de aproximadamente 400 linhas de código-fonte a até 178 mil linhas e cada aplicação móvel tem dois ou três casos de teste (entre 9 e 20 eventos cada).

Tabela 2 - Aplicações Móveis testadas

Aplicação Móvel	Tipo de Codificação	Tipo da Aplicação Multiplataforma	Framework de desenvolvimento	Tamanho - Linhas de Código	#CTs (#Ev.)
Fresh Food Finder	Código Aberto	Híbrido	Cordova	13824	3 (20)
Order App	Código Aberto	Híbrido	Cordova	71565	3 (16)
MemesPlay	Industrial	Híbrido	Cordova	5484	3 (12)
Agenda	Código Aberto	Híbrido	Cordova	1038	3 (18)
TodoListCordova	Código Aberto	Híbrido	Cordova	9304	3 (10)
MovieApp	Código Aberto	Nativo Multiplataforma	ReactNative	2088	3 (10)
TodoList	Código Aberto	Nativo Multiplataforma	ReactNative	405	3 (13)
Tasky	Código Aberto	Nativo Multiplataforma	Xamarin	654	3 (9)
Bargains	Industrial	Nativo Multiplataforma	Xamarin	178266	2 (10)

Todas as aplicações foram instaladas e testadas em dois dispositivos Android e dois iOS. Um ambiente de desenvolvimento específico para Android e outro para iOS foram configurados para a compilação das aplicações selecionadas. Para a compilação dos projetos do iOS, foi necessário assinar a aplicação com um certificado de desenvolvedor fornecido pela Apple. Na Tabela 3 é apresentada a ficha técnica dos dispositivos móveis utilizados no experimento.

Tabela 3 - Dispositivos utilizados

Dispositivo	SO	Tela (Polegadas)	CPU	RAM
Asus ZenFone 3 Max	6.0	5.2	Quad Core 1.3 GHz	2 GB
Samsung J7 Prime	7.0	5.5	Octa Core 1.6 GHz	2 GB
iPhone 6	9.3	4.7	Dual Core 1.4 GHz	1 GB
iPad 3	12.1.3	9.4	Dual Core 1 GHz	1 GB

A `x-PATeSCO` foi utilizada para apoiar o experimento, ela se conecta ao `Appium` que, por sua vez, conecta-se a dispositivos móveis para executar os casos de teste. Na `x-PATeSCO` foram gravados todos os casos de testes originais realizados no estudo de (Menegassi, 2018), em seguida foram extraídos da ferramenta os projetos de testes e executados nos dispositivos móveis, gravando um *log* com o tempo de execução e erros encontrados.

Na próxima etapa foram gerados pela ferramenta, os projetos de testes amplificados baseados nos casos de testes originais. Essa amplificação contém o código-fonte do caso de teste original e os *scripts* de validação baseados no *Test Patterns.*

A ferramenta é capaz de gerar um projeto de teste do Visual Studio, onde é possível executar de forma automatizada esses eventos gravados na aplicação que está sendo testada.

Com os projetos amplificados gerados pela `x-PATeSCO`, cada caso de teste foi executado e todo o processo foi repetido para os dispositivos móveis selecionados, gravando *log* com o tempo de execução e erros encontrados. Com isso, cada aplicação foi executada pelo menos dez vezes por dispositivo.

No total considerando as nove aplicações móveis, existem 26 casos de teste originais, cada caso de teste foi amplificado quatro vezes. Cada caso de teste foi executado uma vez para cada *Test Pattern* (*Lost Connection, Back Event, Don't*

Change State e *Side Drawer Menu*), portanto foram executados 130 casos de testes por dispositivo móvel, sendo 26 originais e 104 amplificados.

Os projetos de teste foram executados a partir de dois computadores distintos, o primeiro para executar os testes nas aplicações Android e o segundo para a execução das aplicações iOS. As configurações utilizadas foram:

a) Windows 7 professional, com processador Intel Core i5 dual-core (2,5 GHz) e 8 GB de RAM.

b) macOS High Sierra, com processador Intel Core i5 (1,4 GHz) e 4 GB de RAM.

Ambos sem qualquer processamento adicional ou carga de comunicação para evitar a saturação da CPU e da memória. Um servidor `Appium` também foi instalado em cada uma das máquinas. Além disso, os recursos de software usados no experimento foram os seguintes:

- XCode 8.2 (para compilação das aplicações para o iOS 9.3);
- XCode 10 (para compilação das aplicações para o iOS 10.13);
- Android SDK r24.4.1 (para compilação das aplicações para Android);
- VisualStudio 2017;
- Appium 1.6.2 (Windows OS) e 1.12.1 (MAC OS);
- Selenium WebDriver for C# 2.53.0;
- Appium WebDriver 1.5.0.1;
- Windows 7 OS;
- MAC OS High Sierra.

4.3. AMEAÇAS A VALIDADE

Nesta seção são apresentadas as possíveis ameaças à validade desta pesquisa.

A baixa quantidade de dispositivos móveis presentes no estudo, podem reduzir os resultados obtidos. Outro ponto é a quantidade limitada de aplicações móveis multiplataforma utilizadas nos testes. Com a grande quantidade de variações de dispositivos móveis existentes no mercado, um maior número de dispositivos neste estudo não foi possível, visto que o custo seria inviável. No caso das aplicações, como

existe a necessidade de validar o tempo de execução dos casos de testes, optou-se utilizar as mesmas aplicações contidas no estudo de Menegassi (2018).

Uma outra ameaça são os *hardwares* dos dispositivos móveis selecionados. Os dispositivos utilizados nos testes não possuem uma grande diferença de configuração de *hardware*, portanto um dispositivo com muito mais memória ou um processador muito mais veloz pode apresentar diferenças nos tempos de execução dos casos de testes.

Todos os casos de testes originais foram reproduzidos fielmente à partir dos casos já gravados no estudo de Menegassi (2018) que foram projetados por quatro participantes independentes com experiência em computação móvel.

Os falsos positivos encontrados no estudo podem ser minimizados alterando a forma de comparação dos estados. Atualmente é verificado se as imagens dos estados possuem uma diferença maior que 10%. Isso pode ser aperfeiçoado realizando comparações entre os XML dos estados.

Grande parte das ameaças citadas, podem ser mitigadas no futuro com replicações deste estudo. Desta forma estão disponíveis no GITHUB todos os pacotes contendo a ferramenta com o código-fonte aberto e os dados necessários para a execução.

4.4. ANÁLISE DOS RESULTADOS

Todos os projetos de testes foram executados e gravaram *logs* contendo informações de tempo e erros das execuções para os quatro dispositivos móveis selecionados. Essas informações foram coletadas, processadas e analisadas para responder as questões de pesquisa proposta. Esta seção apresenta e discute todos os resultados obtidos.

A tabela 4 apresenta os dados da execução dos casos de testes nos dispositivos móveis. Ela demonstra nas linhas o sistema operacional de cada dispositivo móvel; nas colunas são exibidos a quantidade de erros (ER*), as falhas (FL*) e os falso positivos (FP*) encontrados em cada *Test Pattern.* As colunas agruparam o *Back Event, Lost Connection*, *Side Drawer Menu* e *Don’t Change State.* A coluna *Don’t Change State* leva em consideração os resultados das execuções dos eventos de Rotação, Pausa-Para-Retorna, *Zoom*, *Scroll e Swipe*. O símbolo '-' indica

que o *Test Pattern* não é aplicável nesta determinada aplicação. Por exemplo, a aplicação `Agenda` não possui conexão com a Internet, portanto, não é aplicável.

Tabela 4 - Número de Erros, Falhas e Falso Positivos

Aplicação Móvel	Dispositivo	*Back Event*			*Lost Connection*			*Side Drawer Menu*			*Don't Change State*			TOTAL		
		ER*	FL*	FP*	ER*	FL*	FP*	ER*	FL*	FP*	ER*	FL*	FP*	ER*	FL*	FP*
Fresh Food Finder	Android 6.0	1	10	0	0	0	0	0	0	0	1	31	15	2	41	15
	Android 7.0	1	10	0	0	0	0	0	0	0	0	16	16	1	26	16
	iOS 9.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	iOS 12.1.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
OrderApp	Android 6.0	1	15	0	1	6	0	-	-	-	1	9	0	3	30	0
	Android 7.0	1	15	0	1	6	0	-	-	-	0	9	9	2	30	9
	iOS 9.3	0	0	0	1	2	0	-	-	-	1	1	0	2	3	0
	iOS 12.1.3	0	0	0	1	2	0	-	-	-	1	1	0	2	3	0
Memesplay	Android 6.0	1	8	0	1	6	0	0	0	0	1	23	21	3	37	21
	Android 7.0	1	8	0	1	6	0	0	0	0	0	17	17	2	31	17
	iOS 9.3	0	0	0	1	2	0	0	0	0	0	0	0	1	2	0
	iOS 12.1.3	0	0	0	1	2	0	0	0	0	0	0	0	1	2	0
Agenda	Android 6.0	1	8	0	-	-	-	0	0	0	1	40	38	2	48	38
	Android 7.0	1	8	0	-	-	-	0	0	0	0	12	12	1	20	12
	iOS 9.3	0	0	0	-	-	-	0	0	0	0	0	0	0	0	0
	iOS 12.1.3	0	0	0	-	-	-	0	0	0	0	0	0	0	0	0
TodoListCordova	Android 6.0	1	3	0	-	-	-	-	-	-	1	11	10	2	14	10
	Android 7.0	1	3	0	-	-	-	-	-	-	0	11	11	1	14	11
	iOS 9.3	0	0	0	-	-	-	-	-	-	0	0	0	0	0	0
	iOS 12.1.3	0	0	0	-	-	-	-	-	-	0	0	0	0	0	0
MovieApp	Android 6.0	1	8	0	2	6	0	0	0	0	3	46	39	6	60	39
	Android 7.0	1	8	0	2	6	0	0	0	0	0	46	46	3	60	46
	iOS 9.3	0	0	0	1	1	0	0	0	0	1	1	0	2	2	0
	iOS 12.1.3	0	0	0	1	1	0	0	0	0	1	1	0	2	2	0
TodoList	Android 6.0	0	0	0	-	-	-	-	-	-	1	18	13	1	18	13
	Android 7.0	0	0	0	-	-	-	-	-	-	0	15	15	0	15	15
	iOS 9.3	0	0	0	-	-	-	-	-	-	0	0	0	0	0	0
	iOS 12.1.3	0	0	0	-	-	-	-	-	-	0	0	0	0	0	0
Tasky	Android 6.0	0	0	0	-	-	-	-	-	-	0	2	2	0	2	2
	Android 7.0	0	0	0	-	-	-	-	-	-	0	2	2	0	2	2
	iOS 9.3	0	0	0	-	-	-	-	-	-	0	0	0	0	0	0
	iOS 12.1.3	0	0	0	-	-	-	-	-	-	0	0	0	0	0	0
Bargains	Android 6.0	1	3	0	1	5	0	-	-	-	2	21	5	4	29	5
	Android 7.0	1	3	0	1	5	0	-	-	-	0	20	20	2	28	20
	iOS 9.3	0	0	0	1	3	0	-	-	-	1	1	0	2	4	0
	iOS 12.1.3	0	0	0	1	3	0	-	-	-	1	1	0	2	4	0
Total	Android 6.0	7	55	0	5	23	0	0	0	0	11	201	136	23	279	143
	Android 7.0	7	55	0	5	23	0	0	0	0	0	148	148	12	226	148
	iOS 9.3	0	0	0	3	6	0	0	0	0	2	2	0	7	11	0
	iOS 12.1.3	0	0	0	3	6	0	0	0	0	2	2	0	7	11	0

ER: Erros FL*: Falhas FP*: Falso Positivos*

Todas execuções automáticas dos casos de testes foram posteriormente reproduzidas manualmente, desta forma, foram considerados como erros, apenas os casos que foram possíveis simular manualmente. Quando não eram possíveis de serem reproduzidos manualmente, foram definidos como falso positivo. Para a quantidade de falhas foi considerado o número de vezes que houve alguma interrupção ou diferença de estados ocasionados pelo erro ou pelo falso positivo detectado. Como a maior quantidade de erros foram encontrados no Android 6.0, a análise foi focada no dispositivo com este SO.

Foi possível observar que o *Back Event* apresentou sete erros em sete aplicações diferentes. Não foram encontrados falsos positivos, uma vez que todas as falhas foram causadas pelos erros identificados. No geral, para este *Test Pattern* foi detectado um cenário defeituoso no qual a aplicação não retorna para uma tela anterior quando executado o botão Voltar. Em vez disso, a aplicação é encerrada, entra em segundo plano ou retorna para uma tela diferente.

O *Test Pattern Lost Connection*, revelou cinco erros em quatro das cinco aplicações nos quais ele é aplicável. Ele demonstrou um número menor de falhas e falso positivos. Isso provavelmente se deve ao baixo número de eventos encontrados nos casos de testes que dependem de uma conexão com a Internet para serem executados corretamente. Por exemplo, a aplicação `OrderApp` não apresenta uma mensagem quando ocorre uma perda de conexão com a Internet. Já a aplicação `MovieApp` encerra a aplicação quando o dispositivo perde a conexão com a Internet.

Não foram encontrados erros, falhas ou falso positivo para o *Test Pattern Side Drawer Menu* em nenhuma das quatro aplicações aptas aos testes. Foram verificados que todos os itens de menu estão acessíveis e navegam para diferentes telas de forma correta.

Para o *Don't Change State* foram encontrados onze erros para o Android e dois no iOS em oito aplicações móveis. É possível observar que esse *Test Pattern* apresentou um alto número de falhas e falsos positivos. Isso ocorreu devido a problemas relacionados ao *framework* Appium. Ações como *Zoom*, *Scroll* e *Swipe* causaram exceções durante a execução do teste automatizado, mas não foi possível reproduzir esses cenários manualmente. Quanto aos erros, `OrderApp`, `TodoList`, `Bargains` e `Memesplay` reiniciaram após uma ação Pausa-Para-Retorna. Já as

aplicações `Agenda`, `Fresh-Food-Finder`, `MovieApp`, `TodoList`, e `Bargains` foram encerradas após executar a ação de *zoom*.

Os testes amplificados detectaram 23 erros únicos em oito das nove aplicações móveis multiplataforma. Esses resultados fornecem evidências que apoiam uma resposta positiva para o **QP1**.

A Tabela 5 apresenta a quantidade de tempo de execução gasto pelos testes amplificados com relação aos casos de testes originais, considerando apenas a amplificação, ou seja, se o caso de teste original demorou um minuto para executar e a amplificação demorou um minuto e meio, é apresentado que o tempo de execução é 1,5x com relação ao caso de teste original. A quantidade de tempo gasta nos *Test Patterns* mostra uma variação entre as aplicações móveis testadas. Por exemplo, para o Android, o *Back Event* possui uma variação de 0,71 a 4,5 vezes em comparação com caso de testes original. Em média, o *Back Event* também apresenta a maior média de tempo gasto (2,97x), enquanto o *Side Drawer* mostra a menor média de tempo (2,06x). O *Don't Change State* e o *Lost Connection* apresentaram médias intermediárias de 2,42x e 2,68x, respectivamente. Já para o iOS, o *Lost Connection* apresenta a maior média de tempo gasto (2,01x) em relação ao caso de teste original. O *Side Drawer Menu* e o *Don't Change State* empatam quanto a média de tempo gasto para a execução dos casos de testes amplificados (1,22x). E o *Back Event* apresenta média intermediária de (1,75).

Para responder a **QP2**, os resultados mostram que os testes amplificados executam de 0,65 a até 5,56 vezes o tempo de execução do conjunto de testes original, portanto o impacto pode ser razoável. Dependendo da aplicação e do *Test Pattern*, o (baixo) custo do tempo de CPU pode ser compensado pelos recursos de detecção de erros da abordagem proposta.

Na Figura 19 é possível observar que o SO que mais detectou erros foi o Android em sua versão 6.0, em segundo lugar também o Android em sua versão 7.0, já os dispositivos com iOS encontraram 7 erros, independentemente da versão

utilizada. Nenhuma versão dos SOs encontrou erros diferentes dos que foram vistos na versão 6.0 do Android.

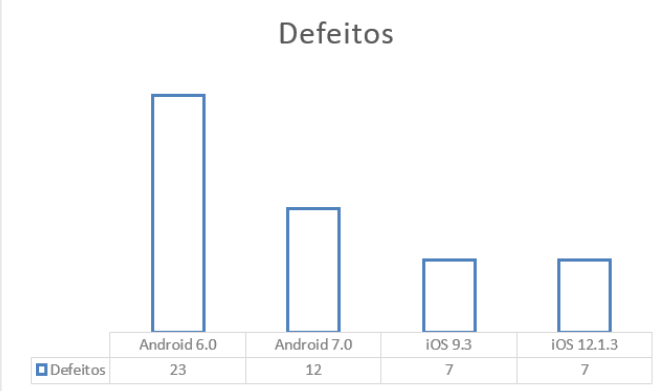

Figura 19 - Erros únicos encontrados por dispositivo

Após a análise das questões anteriores, foram gerados insumos para a resposta da **QP3**. Os resultados mostraram que após a execução dos casos de teste em diferentes configurações de dispositivos e SOs, nenhum erro exclusivo foi encontrado nos dispositivos da Apple. No entanto, os dispositivos Android apresentaram uma variação na quantidade de erros para cada versão do SO.

Tabela 5 - Tempo de execução em relação aos casos de testes originais

Aplicação Móvel	Sistema Operacional	Back Event	Lost Connection	Side Drawer Menu	Don't Change State
Fresh-Food-Finder	Android 6.0	4,50	1,70	1,71	2,59
	Android 7.0	4,55	1,72	1,70	2,69
	iOS 9.3	1,68	1,45	1,39	1,23
	iOS 12.1.3	1,43	1,32	1,15	1,30
OrderApp	Android 6.0	4,25	3,37	-	3,91
	Android 7.0	3,36	2,66	-	3,09
	iOS 9.3	2,29	1,83	-	1,56
	iOS 12.1.3	2,17	1,82	-	1,55
Memesplay	Android 6.0	4,25	5,56	3,09	4,47
	Android 7.0	2,40	3,58	1,96	2,70
	iOS 9.3	1,30	1,22	1,32	1,23
	iOS 12.1.3	1,01	0,98	0,93	0,89
Agenda	Android 6.0	3,44	-	1,82	2,07
	Android 7.0	3,42	-	1,64	2,29
	iOS 9.3	1,66	-	1,52	1,19
	iOS 12.1.3	1,58	-	1,03	1,25
TodoListCordova	Android 6.0	3,71	-	-	2,62
	Android 7.0	4,17	-	-	2,87
	iOS 9.3	2,22	-	-	1,55
	iOS 12.1.3	2,09	-	-	1,53
MovieApp	Android 6.0	2,20	2,54	3,10	2,04
	Android 7.0	0,71	0,82	1,02	0,65
	iOS 9.3	1,73	1,42	1,32	1,27
	iOS 12.1.3	1,53	1,42	0,99	1,15
TodoList	Android 6.0	1,43	-	-	2,40
	Android 7.0	1,39	-	-	1,99
	iOS 9.3	1,34	-	-	1,28
	iOS 12.1.3	1,39	-	-	1,23
Tasky	Android 6.0	1,97	-	-	2,38
	Android 7.0	2,00	-	-	2,43
	iOS 9.3	1,27	-	-	1,17
	iOS 12.1.3	1,41	-	-	1,26
Bargains	Android 6.0	1,41	1,89	-	1,77
	Android 7.0	1,45	1,85	-	1,99
	iOS 9.3	2,22	3,04	-	1,27
	iOS 12.1.3	2,50	2,31	-	1,23
	Android Max	4,50	5,56	3,10	4,47
	Android Min	0,71	0,82	1,02	0,65
	Android Média	2,97	2,68	2,06	2,42
	iOS Max	2,50	3,04	1,52	1,56
	iOS Min	1,01	0,98	0,93	0,89
	iOS Média	1,75	2,01	1,22	1,22

4.5. DISCUSSÃO DOS RESULTADOS

No experimento realizado foram analisados a ferramenta e a abordagem propostas na execução de amplificação automatizada de casos de testes baseado em *Test Patterns* encontrados na literatura. Essa abordagem foi aplicada em dispositivos móveis distintos contendo Android e iOS como SOs. Foram obtidos resultados que forneceram evidências que respondem as questões de pesquisa.

A abordagem apresentou quatro amplificações, e apenas o *Side Drawer Menu* não identificou nenhum erro ou falso positivo. É possível que os menus dessas aplicações foram construídos com componentes de UI bem testados e que os desenvolvedores deram atenção especial a eles devido à sua relevância nas aplicações. Como consequência, os menus são mais testados e menos suscetíveis a erros. Todos os outros encontraram erros adicionais nas aplicações que foram testadas anteriormente nos casos de testes originais. Dos erros encontrados, situações como encerrar a aplicação ou reinicia-las foram comuns.

Das aplicações móveis testadas, somente a `Tasky` não apresentou nenhum erro. Esta aplicação é simples, não possui conexão com a internet e não contém um menu, isso elimina metade das amplificações propostas, portanto apenas o *Back Event* e o *Don't Change State* foram testados. Foram encontradas nesta aplicação dois falsos positivos que tratam de travamento da aplicação no momento da realização do *zoom*. No entanto, não foi possível identificar esse erro manualmente, por isso foi considerado como falso positivo.

Como todos os falsos positivos foram encontrados apenas quando executados pelo Appium, é possível que versões mais recentes sejam mais robustas para testes automatizados, além de que os resultados mostraram que a versão do Android 7.0 gerou menos erros que a 6.0.

Surpreendentemente, todos os erros encontrados pelo *Don't Change State* no Android 6.0 não foram detectados no Android 7.0. As políticas do Android para processos em segundo plano podem diferir entre as versões do sistema operacional. Isso pode causar um comportamento defeituoso em uma versão específica.

Nos testes realizados no iOS, apenas sete erros foram encontrados, e nenhum falso positivo foi identificado. Isso nos mostra que o Appium e o iOS possuem uma integração melhor que com o Android.

O *Lost Connection* encontrou três erros no iOS tanto para a versão 9.3 quanto para a 12.1.13 e o *Don't Change State* encontrou dois erros em ambas versões do SO. Isso nos mostra que para as aplicações testadas com o iOS, não houve diferença entre as versões, inclusive a quantidade de falhas ocorridas nos dispositivos da Apple foram as mesmas. Além disso, não foram encontrados erros diferentes entre os SOs.

A menor quantidade de erros encontrados no iOS, pode ser explicado por algumas particularidades do SO. O fato do código-fonte não ser aberto e rodarem exclusivamente nos dispositivos da Apple, impossibilita que outros fabricantes possam editar e customizar o SO, além de limitar a quantidade de dispositivos móveis.

4.6. CONSIDERAÇÕES FINAIS

A automatização de testes em aplicações multiplataforma pode gerar resultados financeiros positivos para a indústria de software. Isso motiva a utilização da abordagem proposta, visto que os *scripts* de automatização criados realizam testes e encontram erros.

5. CONCLUSÃO

As aplicações móveis recebem diferentes tipos de eventos externos. Esses eventos são executados pelos usuários através do toque ou por fatores externos como acelerômetro e conexão com o GPS (Liu, Gao e Long, 2010). Nessas aplicações, é possível identificar uma quantidade significativa de falhas devido à ações realizadas pelo usuário durante a execução da aplicação (Zaeem, Prasad e Khurshid, 2014). Independente do SO utilizado no dispositivo móvel em que a aplicação foi desenvolvida e instalada, existe a necessidade da realização de testes. A busca por qualidade nas aplicações móveis está cada vez maior, os usuários exigem confiabilidade, robustez e eficiência; isso faz com que os desenvolvedores de software adotem técnicas para garantir que as aplicações estejam adequadas a essa expectativa (Amalfitano *et al.*, 2017).

Nesta dissertação foi apresentada e avaliada uma abordagem para gerar scripts de teste amplificados para aplicações móveis multiplataforma. Para fazer isso, foram utilizados quatro *Test Patterns*: *Lost Connection*, *Back Event*, *Side Drawer Menu* e *Don't Change State*. Para a gravação dos casos de testes foi utilizada uma ferramenta de suporte chamada `x-PATeSCO`. Então foi desenvolvido uma ferramenta de amplificação dos casos de testes gerados pela `x-PATeSCO` chamada de *Test Generator Amplificator.* Nove aplicações multiplataforma e quatro dispositivos móveis foram usados no estudo. Todos os casos de testes originais e amplificados foram executados em cada dispositivo móvel. Ao final foram gravados os logs contendo os erros e o tempo de execução de cada caso de teste.

Os resultados mostraram que das nove aplicações móveis testadas, apenas a `Tasky` não apresentou nenhum erro. Apenas o *Back Event* e o *Don't Change State* foram testados nessa aplicação, como ela não possui conexão com a Internet, não foi possível testar o *Lost Connection*, e o fato dela não possuir um menu, elimina *Side Drawer Menu.* No entanto em todas as outras aplicações foram encontrados erros.

Referente aos erros detectados para cada *Test Pattern* testado, foi possível observar que o *Don't Change State* encontrou a maior quantidade de problemas nas aplicações móveis testadas, identificado um total de onze erros em oito das nove aplicações móveis. O *Back Event* encontrou a segunda maior quantidade, totalizando sete erros. Das nove aplicações testadas, apenas a `TodoList` e `Tasky` não

apresentaram inconsistências ao serem testadas com esse *Test Pattern*. Em terceiro na quantidade de erros detectados ficou o *Lost Connection*, que detectou cinco erros. É de fato uma necessidade a validação deste *Test Pattern.* Das cinco aplicações móveis que utilizam conexão com a Internet (`Fresh-Food-Finder`, `OrderApp`, `Memesplay`, `MoveiApp` *e* `Bargains`) apenas o `Fresh-Food-Finder` não apresentou nenhum erro para o *Lost Connection.* Todas as outras aplicações, apresentaram inconsistências ao perder a conexão com a internet. O único *Test Pattern* que não identificou um erro foi o *Side Drawer Menu*. Apenas quatro aplicações possuíam menu, e todos as validações relacionadas a este *Test Pattern* funcionaram corretamente.

Com relação ao tempo de execução, foi possível concluir que existe uma diferença grande com relação aos dispositivos Android e iOS. Para os Android, a média de execução dos casos de teste amplificados são no mínimo duas vezes maior que o tempo dos casos de teste originais. Do outro lado, para o iOS essa média diminui para uma vez e meia.

Apesar de um aumento de tempo de execução razoável, o fato de ser possível encontrar erros que não foram identificados nos casos de teste originais, pode pesar a favor da utilização da amplificação proposta neste estudo na indústria de software.

5.1. TRABALHOS FUTUROS

Baseado nos resultados deste estudo, os trabalhos futuros são descritos a seguir:

Busca de novas aplicações. Para ampliar os resultados deste estudo, é necessário que novas aplicações sejam testadas. As nove aplicações móveis multiplataforma testadas neste trabalho, possuem poucas funcionalidades complexas, portanto uma nova execução do estudo com outras aplicações pode dar um resultado mais abrangente.

Inclusão de novos *Test Patterns.* Os quatro *Test Patterns* utilizados apresentaram resultados satisfatórios. No entanto é possível identificar novos eventos e situações e incluir na ferramenta, aumentando o poder de amplificação.

5.2. DIVULGAÇÃO DOS RESULTADOS

Durante o desenvolvimento desta dissertação de mestrado, obteve-se a seguinte publicação:

- ASSIS, T. B.; MENEGASSI, A. A.; ENDO, A. T. **Amplifying Tests for Cross-Platform Apps Through Test Patterns.** In: The 31st International Conference on Software Engineering & Knowledge Engineering. 2019. p. 55–60. DOI: 10.18293/SEKE2019-076

Ferramenta desenvolvida e pacote experimental:

- ASSIS, T. B.; MENEGASSI, A. A.; ENDO, A. T. **A tool for Amplifying Tests for Cross-Platform Apps through Test Patterns.** Disponível em: https://github.com/thiagobotti/test-patterns-amplificator

Registro de Software:

- ENDO, A. T.; MENEGASSI, A. A.; ASSIS, T. B. **x-PATeSCO - Cross-Platform App Test Script Recorder**. Data da Publicação: 17/01/2019. Certificado de Registro de Programa de Computador, Registro N° BR512019000397-7.

REFERÊNCIAS

Adamsen, C. Q., Mezzetti, G. e Møller, A. (2015) “Systematic Execution of Android Test Suites in Adverse Conditions”, *Proceedings of the 2015 International Symposium on Software Testing and Analysis*.

Alves, R. e Oliveira, P. De (2012) “Apoio à automatização de oráculos de teste para programas com interfaces gráficas”. doi: 10.11606/D.55.2012.tde-30032012-144613.

Amalfitano, D. *et al.* (2017) “Why does the orientation change mess up my Android application ? From GUI failures to code faults”, *Software Testing, verification and Reliability.*, (September), p. 1–27.

Android Core App Quality (2019) *Android*. Available at: https://developer.android.com/docs/quality-guidelines/core-app-quality (Acessado: 24 de junho de 2019).

Apache (2019) *Cordova*. Available at: https://cordova.apache.org/docs/en/latest/ (Acessado: 21 de março de 2019).

Appium (2019) *Introduction to Appium*. Available at: http://appium.io/docs/en/about-appium/intro/ (Acessado: 13 de fevereiro de 2019).

Apple (2019) *Apple iOS Human Interface Guidelines*. Available at: https://developer.apple.com/design/human-interfaceguidelines/%0Aios/overview/themes/ (Acessado: 12 de janeiro de 2019).

Barr, E. T. *et al.* (2015) “The Oracle Problem in Software Testing : A Survey”, *IEEE Transactions on Software Engineering*, 41(5), p. 507–525. doi: 10.1109/TSE.2014.2372785.

Bosnic, S., Papp, I. e Novak, S. (2016) “The development of hybrid mobile applications with Apache Cordova”, *2016 24th Telecommunications Forum (TELFOR)*.

Boushehrinejadmoradi, N. *et al.* (2015) “Testing Cross-Platform Mobile App Development Frameworks (T)”, *30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015*. doi: 10.1109/ASE.2015.21.

Coimbra Morgado, I. e Paiva, A. C. R. (2015) “Testing approach for mobile applications through reverse engineering of UI patterns”, *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, p. 42–49. doi: 10.1109/ASEW.2015.11.

Danglot, B. *et al.* (2019) *A Snowballing Literature Study on Test Amplification”*, *CoRR*. Available at: https://arxiv.org/abs/1705.10692.

Dzhagaryan, A. e Milenkovi, A. (2016) “Models for Evaluating Effective Throughputs for File Transfers in Mobile Computing”, *Computer Communication and Networks (ICCCN), 2016 25th International Conference on*. doi: 10.1109/ICCCN.2016.7568547.

Eisenman, B. (2015) *Learning React Native*. 1º ed.

El-kassas, W. S. *et al.* (2016) “Enhanced Code Conversion Approach for the Integrated Cross-Platform Mobile Development (ICPMD)”, 42(11), p. 1036–1053.

Endo, A. T., Menegassi, A. A. e Assis, T. B. (2019) “x-PATeSCO - Cross-Platform App Test Script Recorder”. Certificado de Registro de Programa de Computador. Registro N° BR512019000397-7.

Facebook (2019) *Build native mobile apps using JavaScript and React*. Available at: https://facebook.github.io/react-native/ (Acessado: 7 de fevereiro de 2019).

Fazzini, M. e Orso, A. (2017) “Automated Cross-Platform Inconsistency Detection for Mobile Apps”, *32nd IEEE/ACM International Conference on Automated Software Engineering*, p. 308–318.

Gaddah, A. e Kunz, T. (2003) “A survey of middleware paradigms for mobile computing”, *Techical Report*, (July). Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.7633&rep=rep1&type=pdf.

Gao, J. *et al.* (2014) “Mobile Application Testing: A Tutorial”, *Computer*, 47, p. 46–55. doi: 10.1109/MC.2013.445.

Goodenouth, J. B. e Gerhart, S. L. (1975) “Toward a Theory of Test Data Selection”, *IEEE Transactions on Software Engineering*, p. 156–173. doi: 10.1145/390016.808473.

Hoffman, D. (2001) “Using Oracles in Test Automation”, *Proceedings of the 19th Pacific Northwest Software Quality Conference (PNSQC 2001)*.

Holl, K. e Elberzhager, F. (2014) “A Mobile-specific Failure Classification and its Usage to Focus Quality Assurance”, *40th EUROMICRO Conference on Software Engineering and Advanced Applications*. doi: 10.1109/SEAA.2014.19.

IBM (2019) *Native, web or hybrid mobile-app development*. Available at: ftp://public.dhe.ibm.com/software/pdf/mobileenterprise/%0AWSW14182USEN.pdf (Acessado: 5 de janeiro de 2019).

IDC (2019) *Market Share, 2017 Q1*.

IEEE (1990) “Standard Glossary of Software Engineering Terminology”. 1990. ISSN 0-7381-0391-8. ISBN 155937067X. Available at: http://ieeexplore.ieee.org/xpls/absall.jsp?arnumber=159342.

Jing, J., Helal, A. S. e Elmagarmid, A. (1999) “Client-server computing in mobile environments”, *ACM Computing Surveys*, 31(2), p. 117–157. doi: 10.1145/319806.319814.

Joorabchi, M. E., Ali, M. e Mesbah, A. (2015) “Detecting inconsistencies in multi-platform mobile apps”, *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, p. 450–460. doi: 10.1109/ISSRE.2015.7381838.

Latif, M. e Nfaoui, E. H. (2016) “Cross platform approach for mobile aplication development : a survey”, *International Conference on Information Technology for Organizations Development (IT4OD)*, p. 1–5.

Lecheta, R. L. (2015) *Google Android - Aprenda a criar aplicações para dispositivos móveis com o Android SDK*. 4º. Novatec.

Linares-vásquez, M. *et al.* (2017) “Enabling Mutation Testing for Android Apps”, *11th Joint Meeting on Foundations of Software Engineering*. doi: 10.1145/3106237.3106275.

Liu, Z., Gao, X. e Long, X. (2010) “Adaptive random testing of mobile application”, *2nd International Conference on Computer Engineering and Technology (ICCET), 2010*, 2, p. 297–301. doi: 10.1109/ICCET.2010.5485442.

Lynch, M., Sperry, B. e Bradley, A. (2019) *Ionic Framework*, *2013*. Available at:

https://ionicframework.com/docs.

Malavolta, I. *et al.* (2015) “Hybrid Mobile Apps in the Google Play Store : An Exploratory Investigation”, *2nd ACM International Conference on Mobile Software Engineering and Systems*, p. 56–59. doi: 10.1109/MobileSoft.2015.15.

Méndez-porras, A., Quesada-lópez, C. e Jenkins, M. (2015) “Automated Testing of Mobile Applications : A Systematic Map and Review”, *CIBSE 2015 - XVIII Ibero-American Conference on Software Engineering*, (April 2015), p. 195–208. Available at: http://eventos.spc.org.pe/cibse2015/pdfs/3%7B_%7DSET15.pdf.

Menegassi, A. A. (2018) *Testes Automatizados para Aplicações Móveis Multiplataforma*. UTFPR.

Microsoft (2019a) *Introduction to Mobile Development*. Available at: https://docs.microsoft.com/pt-br/xamarin/cross-platform/get-started/introduction-to-mobile-sdlc (Acessado: 11 de abril de 2019).

Microsoft (2019b) *Plataforma Xamarin*. Available at: https://docs.microsoft.com/pt-br/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/understanding-the-xamarin-mobile-platform (Acessado: 19 de abril de 2019).

Microsoft (2019c) *Xamarin*. Available at: https://docs.microsoft.com/pt-br/xamarin/cross-platform/troubleshooting/ (Acessado: 27 de abril de 2019).

Milani, A. (2014) *Programando para iPhone e IPad*. 2º ed. Novatec.

Morgado, I. C. e Paiva, A. C. R. (2015) “The iMPAcT Tool: Testing UI Patterns on Mobile Applications”, *30th IEEE/ACM International Conference on Automated Software Engineering The*, p. 876–881. doi: 10.1109/ASE.2015.96.

Muccini, H., Di Francesco, A. e Esposito, P. (2012) “Software Testing of Mobile Applications: Challenges and Future Research Directions”, *AST 2012, Proceedings of the 7th International Workshop on Automation of Software Test*, p. 29–35. doi: 10.1109/IWAST.2012.6228987.

Myers, G. J., Thomas, T. M. e Wiley, J. (2004) *The Art of Software Testing*. 2. New Jersey: JohnWiley.

Pastore, S. (2013) “Mobile operating systems and apps development strategies”, *2013 International Conference on Systems, Control and Informatics*, p. 350–358.

Rumee, S. T. A. e Liu, D. (2013) “DroidTest : Testing Android Applications for Leakage of Private Information”, *The 16th Information Security Conference (ISC 2013)*.

Systems, A. (2019) *PhoneGap*. Available at: http://docs.phonegap.com/.

W3C (2019) *W3C*. Available at: https://www.w3.org/ (Acessado: 2 de fevereiro de 2019).

Wasserman, A. I. e Fosser (2010) *Software Engineering Issues for Mobile Application Development*, *FoSER 2010 Proceedings of the FSE/SDP workshop on Future of software engineering research*.

Wohlin, C. *et al.* (2012) *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-29044-2.

Zaeem, R. N., Prasad, M. R. e Khurshid, S. (2014) “Automated generation of oracles for testing user-interaction features of mobile apps”, *Proceedings - IEEE 7th*

International Conference on Software Testing, Verification and Validation, ICST 2014, p. 183–192. doi: 10.1109/ICST.2014.31.

Zein, S., Salleh, N. e Grundy, J. (2016) “A systematic mapping study of mobile application testing techniques”, *Journal of Systems and Software*. Elsevier Inc., 117, p. 334–356. doi: 10.1016/j.jss.2016.03.065.