UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO
ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Rodrigo Celso Gobbi

AVALIAÇÃO DA APLICAÇÃO DE ALGUMAS TÉCNICAS BLACK-BOX TESTING EM APLICATIVOS HÍBRIDOS

Porto Alegre
2018

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Rodrigo Celso Gobbi

AVALIAÇÃO DA APLICAÇÃO DE ALGUMAS TÉCNICAS BLACK-BOX TESTING EM APLICATIVOS HÍBRIDOS

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Especialista em Engenharia de Software, pelo curso de Pós-Graduação Lato Sensu em Engenharia de Software da Universidade do Vale do Rio dos Sinos – UNISINOS.
Orientadora: Prof. Dra. Margrit Reni Krug

Porto Alegre

2018

Avaliação da aplicação de algumas técnicas black-box testing em aplicativos híbridos

Rodrigo Celso Gobbi[1]

[1]Universidade do Vale do Rio do Sinos – UNISINOS - Porto Alegre – RS - Brasil

{rodrigo.gobbi.7}@gmail.com

Abstract. *After the burst of solutions facing the mobile environment, new technologies are emerging to improve the development of those applications. In this way, the use of hybrid frameworks allowed to achieve a higher variaty of mobile devices using just one development phase. The software testing phase related to that kind of applications are very important in order to prove that the software is working as expected. This article researchs the software techniques called black-box, aiming its applicability over a mobile app developed in a hybrid manner, analysing their effectiveness. The techniques State-Transition and Equivalence Class Partitioning, when combined, found inconsistencies over the specification and codification of the hybrid solution.*

Resumo. *Após a explosão de soluções voltadas ao âmbito mobile, novas tecnologias surgiram para acelerar e permitir uma maior flexibilidade no desenvolvimento destas aplicações. Nesse sentindo, o uso de tecnologias híbridas permitiu atingir uma gama maior de dispositivos utilizando apenas uma fase de desenvolvimento. A etapa de teste de software relacionada a esse processo de desenvolvimento possui vital importância para a verificação do artefato desenvolvido. Esta pesquisa realizou um levantamento das técnicas de teste de software chamadas black-box, aplicando duas delas em um aplicativo desenvolvido de forma híbrida, analisando a sua efetividade. As técnicas State-Transition e Equivalence Class Partitioning, quando combinadas, encontraram incoerências na especificação e codificação da solução híbrida.*

1. Introdução

O surgimento de dispositivos móveis ao longo dos anos promoveu a oferta de aplicações para os mais variados segmentos do mercado. Através desses dispositivos, tecnologias emergiram para permitir o seu desenvolvimento. Esses dispositivos variam em suas características de uma forma abrangente, afetando o processo de Engenharia de Software, principalmente as etapas relacionadas ao teste de software.

As características variam em níveis físicos e lógicos, pois dispositivos móveis possuem tamanhos, recursos computacionais e sistemas operacionais distintos. Da mesma forma, para o desenvolvimento dessas aplicações, ou *apps*, o desenvolvedor deve, muitas vezes, utilizar tecnologias específicas de cada fabricante através de seus respectivos *Software Development Kits – SDK* - e *Guidelines* (IOS e ANDROID, 2018). A complexidade de controlar e gerir múltiplas aplicações destinadas a diversos dispositivos-alvo, promoveu o surgimento de soluções híbridas ou *cross-plataform*, permitindo que um único artefato seja desenvolvido atendendo a maior gama de dispositivos.

No processo de desenvolvimento de software, o teste possui um papel muito importante uma vez que verifica a conformidade do artefato com o que foi codificado e especificado, garantindo que o produto esteja internamente consistente (JENKINS, 2008). As duas técnicas predominantes em testes de software são *white-box* e *black-box* (MYERS, 2004). A última, foco deste trabalho, visualiza o programa como uma “caixa-

preta": não se preocupa com a sua estrutura interna, ou seja, busca encontrar cenários no qual o artefato em questão não se comporta de acordo com o que foi especificado.

Nesse contexto, o trabalho propôs o seguinte objetivo de pesquisa: analisar a utilização de algumas técnicas *black-box* em um aplicativo híbrido de uma pequena organização do setor de alimentação.

Este objetivo geral foi atingido, a partir dos seguintes objetivos específicos:

1. Identificação das técnicas de teste (*black-box*) a serem utilizadas no âmbito *mobile*;
2. Análise de ferramentas disponíveis para aplicar as técnicas identificadas a partir de critérios pertinentes ao estudo de caso;
3. Aplicação da ferramenta de teste escolhida em um aplicativo híbrido existente, por meio de um estudo de caso;
4. Compilação dos resultados gerados pela ferramenta selecionada, analisando e comparando a efetividade das técnicas selecionadas.

O artigo está organizado da seguinte forma: na seção 2 será apresentando um referencial teórico sobre teste de software, técnicas de teste e tópicos relacionados ao desenvolvimento *mobile*. Na seção 3, será abordada a metodologia aplicada, a descrição da unidade de análise e a aplicação das técnicas. Nas seções 4 e 5 serão apresentados os resultados coletados e as considerações finais desta pesquisa.

2. Referencial Teórico

2.1. Teste de Software

O teste de software é um processo, ou uma série de processos, destinado a verificar se o software realiza o que ele foi designado a fazer e se ele desempenha alguma ação que não tenha sido designado a fazer (MYERS, 2004). Com o passar do tempo, este conjunto de processos passou por grandes evoluções, seja pelo surgimento de novas metodologias de desenvolvimento - como a *Test Driven Development,* conhecida como *TDD* - seja pela disponibilização de ferramentas e tecnologias amplamente testadas por outros desenvolvedores, se considerarmos o universo *open source*. No caso da metodologia, pratica-se um processo inverso ao tradicional, no qual o desenvolvimento do teste antecede o desenvolvimento do software propriamente dito.

O processo de teste de software tem como objetivos executar o programa para tentar encontrar falhas, mensurar a qualidade, aumentar a confiabilidade e analisar documentações para prevenir futuras falhas. Segundo Myers (2004), as tarefas relacionadas a teste envolvem questões financeiras e psicológicas. A primeira ocorre pois, em um mundo ideal, é desejável remover todos os erros do software que está em teste; porém, isso não é possível em termos de tempo e recursos físicos, pois um simples software pode possuir diversas combinações de entrada e saída, o que torna impraticável a criação de cenários de testes para todas essas possibilidades. A segunda, incide sobre a atitude do testador, que muitas vezes é mais importante que o próprio processo já que se deve ter em mente a premissa de que o processo tem como objetivo encontrar erros no software e não comprovar que o mesmo não possui erros. Dessa forma, o processo é considerado destrutivo, ou seja a presença de erros deve ser assumida como verdadeira.

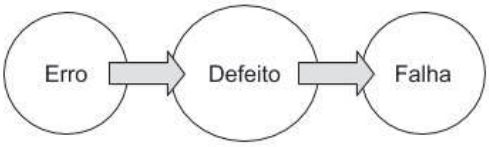
Para discorrer mais sobre o tema, alguns conceitos devem ser introduzidos para uma melhor interpretação desta pesquisa. Spillner, Lin e Schaefer (2014) revisam os seguintes tópicos de acordo com a *International Software Testing Qualification Board* (ISTQB, 2018):

- Erro: consequência de equívocos cometidos durante o processo de desenvolvimento em etapas como codificação ou modelagem. Também podem ocorrer em decorrência de uma má interpretação dos requisitos do sistema.

• Defeito (*Bug*): ocorre a nível sistêmico causado pela presença de erros. Podem ou não impedir o funcionamento do sistema.

• Falha: representa a manifestação de um defeito, conforme visto na Figura 1. Ilustra um cenário cujo a expectativa não é refletida na realidade, ou seja, existe uma discrepância entre o resultado ou o comportamento corrente versus o desejado e esperado.

Figura 1. Relação erro, defeito e falha



Fonte: elaborado pelo autor.

• Caso de teste: refere-se a um conjunto de entrada de valores e resultados esperados do objeto testado.

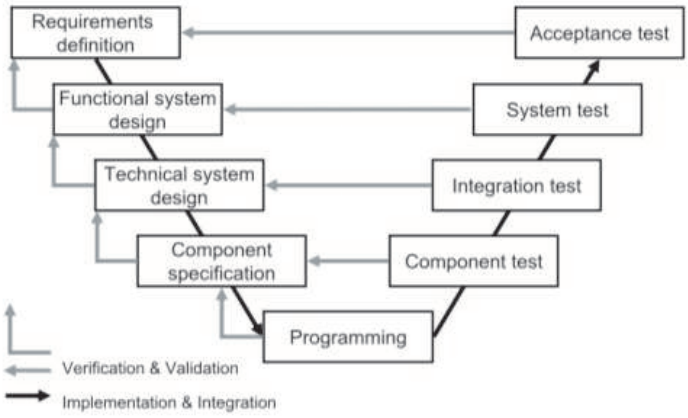
• Suíte de teste: inclui a execução de um ou mais casos de teste.

• Cobertura do teste: o grau no qual itens de cobertura especificados foram determinados ou exercitados por uma suíte de teste.

2.2. Teste no Ciclo de Desenvolvimento de Software

Segundo Spillner, Linz e Schaefer (2014), cada projeto de desenvolvimento de software deve ser planejado e executado utilizando um modelo de ciclo de vida. Dentre estes modelos encontram-se, por exemplo, o Cascata, Modelo-V, Ágil, entre outros. Do ponto de vista de testes, o Modelo-V representa uma correspondência entre as tarefas de desenvolvimento de acordo com a Figura 2.

Figura 2. Modelo-V de desenvolvimento



Fonte: elaborado por (SPILLNER; LINZ; SCHAEFER, 2014).

Este modelo será utilizado para apresentar e revisar os diferentes níveis de teste existentes no processo de desenvolvimento.

2.3. Teste de Componente

Componentes são unidades do software, que dependendo da linguagem de programação podem possuir nomenclaturas como classes, funções ou módulos. Os testes são realizados de forma individual e isolada (*unit*) do restante do sistema. Testes neste nível devem ser comparados com a especificação do componente, verificando que todas as funcionalidades presentes funcionam da forma correta (SPILLNER; LINZ; SCHAEFER, 2014).

2.4. Teste de Integração

Esse nível de teste promove a integração de componentes previamente testados verificando se a colaboração entre os elementos funciona corretamente. Como a nível de unidade os componentes já foram previamente testados, aqui o objetivo é verificar que as interfaces e interações entre eles estão de acordo com o esperado.

Muitas vezes, tanto em nível de sistema quanto em nível de componente, não se pode realizar o teste com os objetos existentes, pois os mesmos não podem ser executados de forma *standalone* sem que os demais componentes que dependam do objeto testado estejam prontos. Por exemplo: se considerarmos um cenário no qual desejamos realizar o teste de uma unidade chamada A, que manipula um banco de dados sem que a unidade responsável pelo banco de dados, B, esteja pronta, pode-se utilizar artefatos adicionais chamados *driver* e *stub modules*. No caso da unidade B não estar pronta, podemos utilizar um *stub A1* para simular o futuro comportamento daquele componente. Se considerarmos o cenário invertido, A não está pronto e B já está, pode-se utilizar um *driver* B1 para representar as possíveis entradas que serão transmitidas para este artefato (MYERS, 2004).

2.5. Teste de Sistema

O nível seguinte de teste no Modelo-V verifica, na perspectiva do usuário, se o software atende às especificações do sistema. Ao invés de *drivers* e *stubs*, o cenário utilizado para teste é o mais próximo da realidade, verificando aspectos funcionais, não funcionais e possíveis inconsistências nas especificações (SPILLNER; LINZ; SCHAEFER, 2014).

2.6. Teste de Aceitação

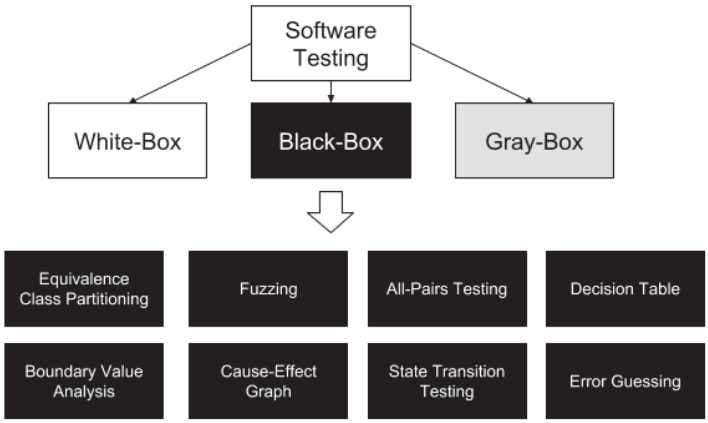
Todos os níveis anteriores de teste descrevem etapas que antecedem a apresentação do artefato perante o cliente. O teste de aceitação utiliza o cliente de uma forma efetiva para garantir que o produto atenda às suas necessidades e que os requisitos foram compreendidos da forma correta na perspectiva de seu negócio (SPILLNER; LINZ; SCHAEFER, 2014).

2.7. Tipos de teste

Cada nível de teste descrito anteriormente possui objetivos e tipos diferentes quando aplicados no processo de desenvolvimento de teste; cabe salientar dois: testes funcionais e não funcionais. O primeiro, inclui todos os tipos de testes que verificam, em nível sistêmico, o comportamento de acordo com os requisitos funcionais da especificação de software. Já o segundo, está relacionado aos requisitos não funcionais da especificação, os quais descrevem atributos dos comportamentos funcionais do sistema, tais como confiabilidade, usabilidade, robustez, segurança, etc. (SPILLNER; LINZ; SCHAEFER, 2014).

Algumas técnicas de design podem ser aplicadas nos dois tipos (testes funcionais e não funcionais) e em diversos níveis, sendo organizadas em três grandes grupos conforme visto na Figura 3: *white-box, black-box* e *gray-box*.

Figura 3. Taxonomia das estratégias de teste



Fonte: elaborado pelo autor.

Tradicionalmente, as duas primeiras técnicas são as mais utilizadas e permitem combater os desafios de custo e de análise combinatória, já citados anteriormente. Um fator importante durante o processo de teste é entender, através destes grupos, como minimizar a eventual quantidade de testes em um conjunto finito e gerenciável, considerando os riscos do que deve ou não ser testado (JAMIL, M. A. et al., 2016).

2.8. White-box

A técnica *white-box* é baseada na forma em que o sistema foi construído, levando em consideração a sua estrutura interna (BLACK e MITCHELL, 2011). Dessa forma, são considerados responsabilidades dos desenvolvedores, já que para cada bloco de código existente, deve-se garantir que o mesmo seja executado pelo menos uma vez, que o seu comportamento seja verificado (RATZMANN e YOUNG, 2003) e que todos os caminhos de controle sejam completamente testados (MYERS, 2004) atingindo um determinado nível de cobertura.

2.9. Black-box

Esta técnica é muitas vezes referenciada como baseada em especificação ou comportamento, no qual os testes são construídos da forma em que o sistema deve funcionar (BLACK e MITCHELL, 2011). O objeto é testado como uma "caixa-preta", não se preocupando com o seu comportamento interno ou sua estrutura, como na técnica mencionada no item anterior. Essa estratégia se restringe a considerar apenas as saídas geradas a partir de entradas selecionadas (NIDHRA, 2012), por isso o termo "caixa-preta" é aplicado. O testador apenas conhece a composição das entradas e o resultado gerado por elas a partir da especificação do produto. Pode ser aplicado em diversos níveis conforme resumido na Tabela 1.

Tabela 1. Domínios de testes de acordo com o Modelo-V

Etapa do Modelo-V	Domínio
Componente	*White* e *Black-box*
Integração	*White* e *Black-box*

Sistema	*Black-box*
Aceitação	*Black-box*

Fonte: elaborado pelo autor.

2.10. Técnicas de Teste de Software *Black-Box*

Fazem parte da estratégia de *black-box* inúmeras técnicas de teste; algumas serão revisadas, com base na literatura, visando ao primeiro objetivo específico deste artigo: *Equivalence Class Partitioning (ECP), Boundary Values Analysis (BVA)*, *Fuzzing*, *Cause-Effect Graph*, *All-Pairs*, *State Transition*, *Decision Tables* e *Error Guessing*.

1. ECP: essa técnica identifica classes ou partições de uma determinada grandeza, como valores de entrada, valores de saída, valores internos, etc, que se comportam e são tratadas de uma forma semelhante pelo sistema (MYERS, 2004). São subdividas da seguinte forma:

a. *Classes válidas*: descrição na qual o sistema trata a classe de uma forma normal.

b. *Classes inválidas:* descrição na qual o sistema trata a classe de uma forma inválida, rejeitando a grandeza em questão.

2. BVA: pode ser considerada como um refinamento da técnica anterior. Prioriza os testes nas bordas de cada classe, quando aplicável. Para a representação de valores limites deve-se compreender questões arquiteturais de como estes valores estarão representados em memória, já que pode existir um impacto nos limites (MYERS, 2004).

3. Fuzzing: técnica permite detectar *bugs* de implementação a partir da injeção de dados inválidos, mal formatados ou inesperados, fazendo com que o sistema entre em colapso. Essa técnica está dividida em duas categorias: *mutation-based* ou *generation-based*. A primeira utiliza uma amostra de dados de entradas válidas, alterando randomicamente o seu conteúdo para a produção de uma segunda amostra que será utilizada como entrada do sistema (KHAN, 2011). A segunda, não utiliza amostras, construindo o conjunto de dados do zero. Em ambos os casos, os dados devem ser "minimamente" válidos para que não sejam descartados pelo programa.

4. Cause-effect Graph: permite identificar uma relação entre combinações de entradas (*cause*) e saídas (*effect*). *Cause* refere-se a uma condição de entrada que promove uma mudança interna no sistema. *Effect* representa o resultado gerado pela combinação de causas. Pode ser utilizada em situações em que determinadas saídas do sistema dependam de combinações de entradas (KHAN, 2011). Por exemplo: as combinações de entrada para um terminal bancário (ATM) poderiam descrever "cartão é válido", "a entrada de senha está correta", etc; Da mesma maneira, os efeitos para este exemplo seriam: "rejeitar o cartão", "solicitar uma nova entrada de senha", etc. (SPILLNER; LINZ; SCHAEFER, 2014).

5. All-Pairs (pair-wise): pode ser utilizada quando interações entre diferentes parâmetros são desconhecidas. A partir das classes de equivalência, um valor representativo é escolhido e combinado com todos os outros valores representativos das outras classes (SPILLNER; LINZ; SCHAEFER, 2014).

6. Decision Tables: permite realizar testes voltados a determinadas ações do sistema, quando os cenários envolvidos nestas ações dependem apenas de

entradas ou pré-condições que existem em um determinado momento. Verifica se as ações geradas pelo sistema correspondem às esperadas quando determinadas combinações de condições ocorrem.

a. Collapsing Columns: permite realizar a "mesclagem" de colunas da *Decision Table*, desconsiderando condições que não afetam determinadas ações.

7. State-transition: deve ser utilizada quando ações do sistema dependem de condições que existiram no passado; verifica se o sistema atinge estados incorretos a partir de eventos ou estados históricos.

a. State-transition - superstate e substate: em determinados cenários é possível representar estados complexos através de uma forma macro, ou *superstate*, definindo o seu conteúdo através de *substates.* Dessa forma, é possível atingir um nível de cobertura mais granular pois novas transições internas e externas serão expostas.

b. *State-transition Diagram e State-transition Table*: a *State-transition Diagram,* ou *State Machine (SM)*, representa o sistema através de estados, ações e transições. Já a *State-transition Table*, ilustra as mesmas informações além de possíveis combinações que podem não estar representadas no sistema por uma falha na compreensão de requisitos.

c. State-transition - Switch-Coverage: critério mais simples de cobertura. Percorre cada estado e cada transição do *State-transition Diagram.*

8. Error guessing: nenhuma metodologia é empregada nesta técnica e sim o somatório de experiências e intuições das pessoas envolvidas com o processo de teste. Permite explorar determinados cenários de possíveis problemas já vivenciados durante a construção de testes (MYERS, 2004).

As técnicas definidas acima possuem metodologias de derivação de casos de testes únicas, conforme a Tabela 2.

Tabela 2. Derivação dos casos de testes

Técnica Black-box	Metodologia de Teste
ECP	1. Identificação das classes ou partições; 2. Elaborar o caso de teste para cobertura das classes válidas; Pode-se utilizar o mesmo caso de teste para cobrir outras classes válidas independentes; 3. Elaborar o caso de teste para classes inválidas. Cada caso de teste só poderá incluir apenas uma classe inválida.
BVA	1. Identificação dos limites de cada partição; 2. Pode-se combinar no mesmo caso de teste a cobertura para bordas válidas. Bordas inválidas devem ser consideradas separadamente.
Fuzzing	Não existe uma metodologia padrão. Deve-se realizar a mutação da amostra de dados existentes para criar novos dados ou realizar a definição de dados baseada em modelos de entrada.
Cause-effect Graph	1. Dividir especificação em pequenos grupos de condições e ações; 2. Representação dos grupos através de um grafo e símbolos pré-definidos (operadores lógicos); 3. Converter em uma tabela de decisão finita a partir do grafo; 4. Casos de testes serão derivados de cada coluna da tabela.
All-Pairs	1. Identificar valores representativos de cada classe de equivalência; 2. A partir disso, construir uma tabela que combina os valores representativos;

	3. Casos de testes serão derivados de cada linha.
SM	1. Definir estado inicial e final para o caso de teste; 2. Garantir a cobertura de estados e transições tendo como início e fim os estados definidos anteriormente.
State-transition Table	1. Utilizar testes derivados do *State-transition Diagram*; 2. Enumerar estados, eventos e condições construindo uma tabela; 3. Garantir a cobertura dos testes para todas as linhas definidas; 4. Escolher um teste que atinja um estado com representação indefinida na tabela. A partir disso, realizar a tentativa de modificá-lo para incluir a combinação indefinida de evento e ação.
Decision Tables	1. Construir tabela a partir das regras de negócios (entradas e saídas esperadas); 2. Casos de testes são derivados para cada coluna.
Error guessing	Não existe uma metodologia padrão (intuição e experiência do testador ao considerar possíveis premissas adotadas pelo desenvolvedor - ex: divisão por zero, parâmetros inválidos, ponteiros, etc).

Fonte: elaborado pelo autor.

Para a construção destes casos, deve-se seguir estas metodologias utilizando informações do artefato que se deseja testar.

2.11. Aplicativos Mobile

O surgimento de dispositivos portáteis promoveu novas necessidades de negócios para os mais variados segmentos de mercado. Para aumentar a competitividade, as empresas têm apostado em incluir nos seus catálogos de serviços uma nova forma de contato com os clientes através de softwares que executam nesses dispositivos móveis, os chamados *apps* ou *mobile apps*.

Os aplicativos móveis incluem no portfólio das empresas novos canais de vendas diretas ou uma nova forma de realizar *advertisement*. De qualquer forma, em todos os possíveis cenários de uso, pode-se usufruir de capacidades funcionais presentes nestes dispositivos com o objetivo de melhorar a *User Experience (UX)*. As facilidades de usabilidade impostas por recursos como *touch screen* e GPS ampliaram o número de possibilidades se compararmos com um sistema desenvolvido para executar em uma estação de trabalho normal.

Os desafios presentes no processo de desenvolvimento dessas aplicações são inúmeros já que a quantidade de dispositivos existentes no mercado é alta. Além desse fator de disponibilidade, podemos também considerar outros fatores como, por exemplo, o sistema operacional que é executado nestes dispositivos, a geolocalização destes aparelhos, a velocidade de conectividade, o consumo de bateria e a variação dos tamanhos de *display*. Todos estes fatores, que impactam de alguma forma no desenvolvimento, no desempenho e na aceitação do artefato, devem ser considerados durante o processo de desenvolvimento, para garantir que a aceitação do *app* perante os usuários seja positiva.

Se considerarmos que a maior parte das aplicações interage com sistemas na nuvem (*backend*) para o armazenamento de dados, mecanismos de autenticação, entre outros, é preciso ter em mente que essa comunicação deve ser eficiente e rápida para não impedir a *UX*. Caso se sinta desconfortável durante a utilização da aplicação, seja pela ineficiência de comunicação seja pelo excesso de consumo de bateria, o usuário desistirá de utilizá-lo, desinstalando-o de seu aparelho em poucos segundos. O índice de rejeição pode aumentar se esse perfil de usuário visualizar as avaliações (*ratings)* de outros usuários do aplicativo das lojas em que os *apps* são publicados.

2.12. Aplicativos Nativos

Uma das possibilidades para o desenvolvimento das aplicações móveis é a utilização de ambientes, ferramentas e tecnologias fornecidas pelos fabricantes dos sistemas operacionais que executam nestes dispositivos. Cada fabricante fornece um *Software Development Kit – SDK* - específico para a sua plataforma; e através dele é possível interagir com os mais variados recursos de hardware presentes em cada dispositivo como acesso a câmera, Wi-Fi, GPS, agenda, entre outros.

Para cada *SDK*, o fabricante adota uma determinada linguagem de programação conforme demonstrado resumidamente na Tabela 3. Em casos como a Apple, não existe flexibilidade na escolha do ambiente que será utilizado para o desenvolvimento sendo impositivo o ambiente oficial.

Tabela 3. Características de desenvolvimento nativo

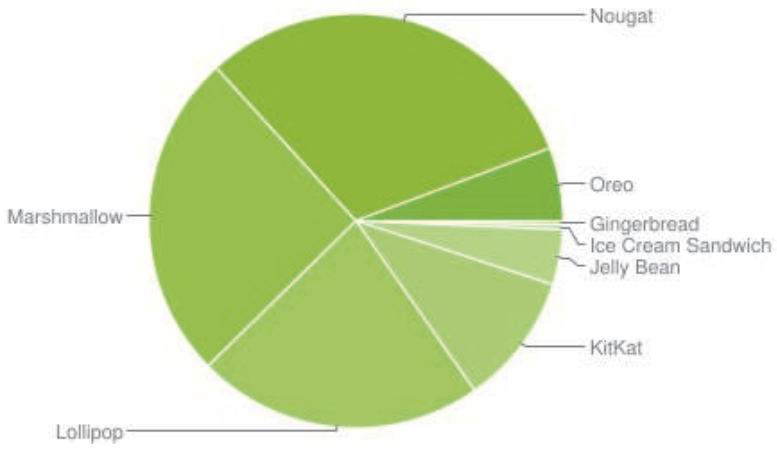
Plataforma	Linguagem	Ambiente de desenvolvimento
Android	Java, C, C++, Kotlin	Eclipse, IntelliJ, Netbeans, Android Studio
iOS	C, Objective-C, Swift (versões 1.0, 1.1, 1.2, 2.0, 3.0, 4.0, 4.1)	Xcode

Fonte: elaborado pelo autor.

A Tabela anterior ilustra um primeiro impacto ao se optar pelo desenvolvimento nativo pois o desenvolvedor deve possuir uma alto nível de proficiência em diversas linguagens de programação para a construção e manutenção desses aplicativos.

Se considerarmos o fator sistema operacional citado na seção anterior, é possível identificar que um dos dois maiores sistemas operacionais existente em celulares e *tablets* - o Android - possui uma grande quantidade de versões. De forma pública, é possível observar a distribuição dessas versões conforme a Figura 4. Observa-se que as maiores fatias de uso das versões do sistema se concentram em Lollipop (Android 5) com 22.4%, Marshmallow (Android 6) com 25.5% e Nougat (Android 7) com 31.1%[1].

Figura 4. Distribuição das versões de Android



Fonte: [ANDROID1].

[1] Segundo ANDROID1 (2018)

[2] Arquitetura genérica para o Ionic versão 1.

[3] Dados não ilustram a quantidade de instalações ativas

Essa informação ilustra um segundo grande impacto no desenvolvimento nativo, pois cada versão de sistema possui uma versão diferente de *SDK* e de Nível de *Application Programming Interface (API)* conforma visto na Tabela 4.

Tabela 4. Versões de Android x Versões de *SDK*

Versão de Android	*SDK* – Nível de *API*
Lollipop (Android 5)	21 e 22
Marshmallow (Android 6)	23
Nougat (Android 7)	24 e 25

Fonte: elaborado pelo autor com base em [ANDROID1].

Muitas vezes em eventuais atualizações do sistema nos dispositivos físicos, serão incluídos recursos adicionais ou evoluções de acessos a recursos existentes que só serão disponibilizados em uma versão futura de *SDK* e *API*. Dessa forma, o *app* desenvolvido em um nível de *API* anterior deverá ser atualizado para usufruir de forma íntegra essas novas funcionalidades.

2.13. Aplicativos Web

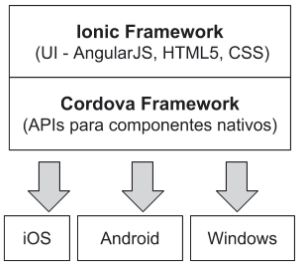
A segunda alternativa para o desenvolvimento de *apps* é a utilização de tecnologias voltadas à *Web* (EL-KASSAS et al., 2017) amplamente padronizadas pela *World Wide Web Consortium* (*W3C*). Esta organização permite que sites se comportem da mesma forma, independentemente de hardware, software ou versão do *browser*. A ideia é permitir que o dispositivo acesse o *Web App* através do *browser* do sistema operacional em questão, permitindo uma aceleração no desenvolvimento, já que não exigirá instalação e publicação nas respectivas lojas, reduzindo o processo burocrático de disponibilização.

Através do HTML5, formato padrão para o desenvolvimento de sites e aplicações *web*, é possível interagir com novos recursos, como áudio e vídeo, o que não era possível em versões anteriores. Porém, esse tipo de solução possui desvantagens quando se tem a necessidade de utilizar recursos específicos que o padrão atual não suporta, como por exemplo a câmera do dispositivo.

2.14. Aplicativos Híbridos

A terceira alternativa para o desenvolvimento é a combinação dos aspectos positivos dos aplicativos Nativos e Web. Esta abordagem usufrui dos aspectos padronizados pela *W3C* para a construção do aspecto visual, uma vez em que existe uma consistência na construção desses elementos. O uso de soluções híbridas passa pela utilização de *frameworks* como Ionic (IONIC, 2018) - Figura 5 -, *Xamarin* (XAMARIN, 2018) e *Titanium* (TITANIUM, 2018).

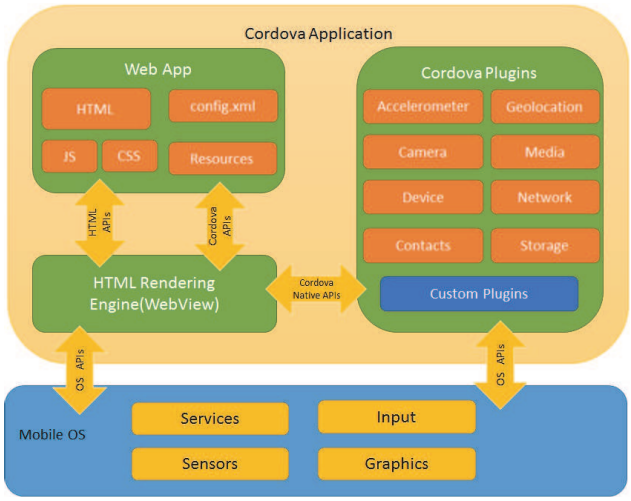
Figura 5. Arquitetura de um aplicativo construído com o framework Ionic[2]



Fonte: elaborado pelo autor.

O mecanismo de funcionamento desses *frameworks*, ilustrado na Figura 6, se baseia em um componente presente nos sistemas operacionais dos celulares, chamado *WebView*, que permite a abertura de páginas *Web* em *apps* desenvolvidos de forma nativa. Ao utilizar a abertura de um *browser*, a padronização citada anteriormente entra em vigor, permitindo uma utilização consistente dos elementos visuais de cada plataforma, não deixando de lado o acesso a componentes do dispositivo.

Figura 6. Arquitetura do framework Cordova



Fonte: [CORDOVA].

A maior parte destes *frameworks* combina tecnologias distintas. No caso do Ionic, podemos citar, em sua arquitetura, outros componentes e suas responsabilidades:

- AngularJS: *framework* escrito em Javascript - JS - que auxilia na construção do *frontend* de páginas dinâmicas.

- Cordova, Apache Cordova ou PhoneGap: permite a construção de aplicações destinadas a dispositivos móveis utilizando HTML5, CSS e JS. Abstrai o conjunto de funções específicas de cada plataforma em *wrappers* nas tecnologias citadas anteriormente, permitindo o acesso a funcionalidades específicas através dessas abstrações.

---

[2] Arquitetura genérica para o Ionic versão 1.

- Cordova *plugins*: As funcionalidades nativas são expostas através de *plugins* - Figura 6 - que disponibilizam recursos como acelerômetro, câmera, sistema de arquivos, entre outros. Os desenvolvedores utilizam JS para a interação com estes componentes.

3. Metodologia

Nessa seção serão descritos aspectos relacionados a pesquisa, abordando a metodologia utilizada, coletas de documentos realizadas e a descrição do estudo de caso.

3.1. Delineamento da Pesquisa

O método de pesquisa utilizado foi a metodologia qualitativa, pois, segundo Richardson (1999), pode descrever a complexidade de um determinado problema. O nível de pesquisa exploratório foi utilizado, pois "se tem a necessidade de identificar, conhecer, levantar ou descobrir informações sobre um determinado tema" (MALHOTRA, 2001). As estratégias utilizadas no processo de investigação dessa pesquisa foram: pesquisa bibliográfica e estudo de caso. A primeira etapa permitiu a construção de um referencial teórico a respeito das técnicas de *black-box testing* existentes e quais dessas possuíam relevância para o estudo de caso posterior. O segundo, utilizou uma investigação empírica, que verifica um fenômeno contemporâneo dentro de seu contexto de vida real (YIN, 2010), tendo como seu componente a seguinte unidade de análise: um aplicativo desenvolvido pelo pesquisador.

Através dessas estratégias, foi possível analisar e utilizar ferramentas para aplicar as técnicas de teste previamente selecionadas. Para a validação do estudo de caso, os resultados obtidos através das ferramentas foram analisados verificando a efetividade das técnicas selecionadas.

3.2. Técnicas de Coleta de Dados

A coleta de dados foi realizada em dois momentos. O primeiro, a documental, pois cujo os dados puderam ser recolhidos no momento da ocorrência do fato ou fenômeno (LAKATOS e MARCONI, 2009). Os seguintes documentos se enquadram para a coleta: especificação de requisitos, casos de uso e relatórios de estatísticas de uso da unidade de análise. Todos os documentos citados foram produzidos pela empresa X da qual o autor é sócio. O segundo, a partir dos resultados das técnicas de teste aplicadas via ferramenta selecionada.

3.3. Definição da Unidade de Análise

Foi utilizado um aplicativo desenvolvido pelo autor como base para a análise desta pesquisa. O *app* foi desenvolvido de forma híbrida, através do Ionic, permitindo que ocorresse apenas um ciclo de desenvolvimento atendendo ambos os requisitos de sistema: Android e iOS. Ambas as versões estão dispostas em produção conforme as Tabelas 5 e 6.

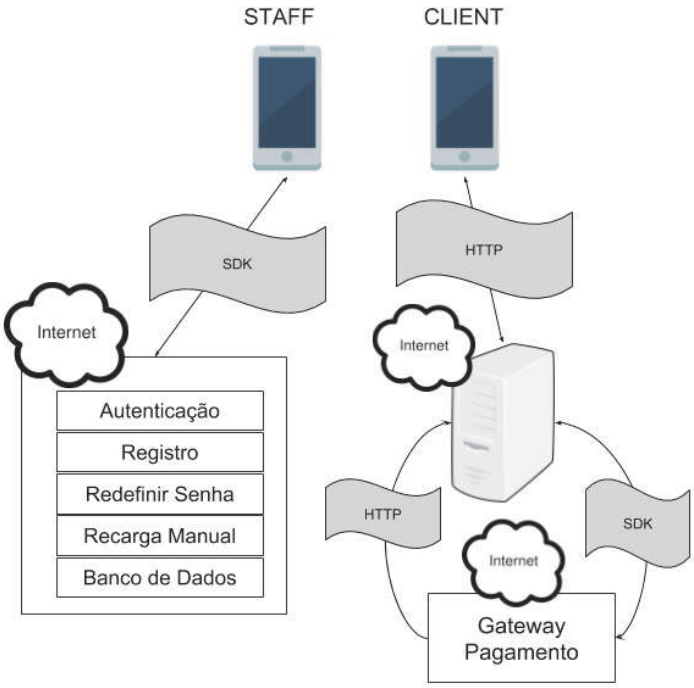
Tabela 5: Distribuição de instalações da unidade de Análise

Loja	Instalações
Apple Store	84
Google Play	49
Total	133

Fonte: elaborado pelo autor com base em [GOOGLE e ITUNES][3]

O contexto de utilização do artefato é um estabelecimento do ramo de alimentação localizado dentro de uma academia de Porto Alegre, Rio Grande do Sul. Tal entidade fornece, através do aplicativo, um mecanismo de *self-checkout* dos produtos para os seus clientes, evitando a utilização de funcionários. A partir de duas versões de operação, *Staff* (funcionário) e *Client* (cliente), o aplicativo concentra funcionalidades como o registro de produtos, cadastro de usuários, recarga de créditos e compra de produtos (ver Apêndice para o diagrama de casos de uso completo). Após o registro, o *Client* pode realizar a recarga a partir de valores fixos previamente definidos (são escolhidos através de botões na tela de recarga) e optar pela compra de produtos via o seu código de barras. A recarga ocorre através da integração com um *Gateway* de pagamento, que fornece esses recursos de forma independente. A manipulação do código de barras do produto pode ser realizada de forma manual ou através da câmera do celular para ambas as versões do aplicativo.

Figura 7. Arquitetura do objeto da Unidade de Análise



Fonte: elaborado pelo autor.

Para o desenvolvimento desse aplicativo usufruiu-se de uma metodologia ágil adaptada ao contexto da empresa da qual o autor é sócio. O processo de desenvolvimento não realizou etapas de teste de software voltado a este artefato, o que promoveu a motivação para esta pesquisa.

Observa-se na Figura 7 que a arquitetura desse aplicativo está subdivida com outros sistemas hospedados na nuvem os quais fornecem serviços essenciais para o *app*. As camadas de autenticação, registro, redefinição de senha, recarga manual e banco de dados são fornecidas por um serviço de nuvem chamado Y, o qual permite que o acesso seja realizado pelos seus próprios *SDKs*.

Tabela 6: Versões do *app* para ambos os sistemas Android e iOS

[3] Dados não ilustram a quantidade de instalações ativas

Versões do Aplicativo
1.0.0
1.0.1
1.0.2
1.0.3
1.0.4
1.0.5
1.0.6
1.0.7
1.0.8

Fonte: elaborado pelo autor.

Durante o ciclo de vida do aplicativo, novas versões foram disponibilizadas com o objetivo de corrigir defeitos detectados em produção, conforme as informações da Tabela 7.

Tabela 7: Descrição de *bugs* conhecidos da Unidade de Análise

BugID	Versão que corrigiu o *Bug*	Descrição do *Bug*
#343	1.0.5	Esqueci minha senha provoca travamento em versões de Android KitKat
#345	1.0.6	Comprar um item de valor igual a quantia de créditos provoca erro
#346	1.0.7	A lista de vendas não é carregada corretamente.
#350	(ainda não foi corrigido)	Código de barras padrões EAN_13 vs UPC_A

Fonte: elaborado pelo autor.

As informações acima foram extraídas e selecionadas a partir do *bug tracking* da empresa X, que pertence ao sistema de gerenciamento de projetos, chamado Redmine (REDMINE, 2018).

3.4. Técnicas de Análise de Dados

A análise documental foi utilizada como técnica para a análise de dados, tratando e interpretando todas as informações reunidas durante a investigação científica (AZEVEDO, MACHADO e DA SILVA, 2011).

3.5. Limitações da Pesquisa

O estudo de caso foi limitado para o uso de técnicas *black-box testing* em um aplicativo desenvolvido de forma híbrida através do *framework* Ionic versão 1 (IONIC, 2018), não sendo possível generalizar os resultados aqui apresentados para outras versões deste *framework* e para outras ferramentas de desenvolvimento de aplicações híbridas.

3.6. Trabalhos Relacionados

A seguir estão listados os artigos relacionados com o assunto desta pesquisa, enumerando os seus principais pontos.

Tabela 8: Compilação de trabalhos relacionados

Título do Artigo	Pontos importantes sobre o artigo	Referência
Mobile Testing	O artigo cita dois tipos de testes voltados a *mobile*: *device tests* e *testing in the wild*. O primeiro está relacionado com o uso de diversos dispositivos móveis para garantir compatibilidade. O segundo, presume os testes no contexto em que o *app* será utilizado (teste exploratório). Em ambos os casos, as opções de infraestrutura serão: emuladores, dispositivos locais e dispositivos disponíveis na *cloud*. Para garantir o mínimo de cobertura funcional e escalabilidade, pode-se construir casos de testes automatizados que usufruem de interações da *GUI (Graphical User Interface)* via identificadores únicos para a manipulação e consulta (padrão para testes de aplicações *Web*).	(HALLER, 2013)
Functional Testing of an Android Application	No artigo é observado a citação para o *framework* do Google chamado Espresso, que permite a construção de testes funcionais e não funcionais. O *framework* foi escolhido para a construção de testes pois, segundo os autores, necessita de menos configuração para funcionar quando comparado com outros. Além disso, por ser mantido pelo Google promove uma forte ligação com o sistema Android, facilitando os testes nessa plataforma. Pode ser utilizado para técnicas *black-box* quando deixa-se de utilizar identificadores internos do Android para a manipulação dos elementos.	(BÅNGERIUS; FRÖBERG, 2016)
Analysis of various testing techniques	O artigo aborda as diversas formas de teste além de realizar uma análise comparativa entre as técnicas. O estudo demonstrou o uso das técnicas em dez programas desenvolvidos na linguagem C. Observou-se que as maiores quantidades de erros foram encontradas através das técnicas *ECP* e *BVA*.	(KAUR; KHATRI; DATTA, 2014)

Fonte: elaborado pelo autor.

Os trabalhos relacionados tiveram importância para essa pesquisa pois, permitiram a construção de um embasamento sobre as técnicas mais efetivas de *black-box* (KAUR, KHATRI e DATTA, 2014), permitindo que o autor selecionasse uma ferramenta de teste de código aberto de fácil configuração (BÅNGERIUS; FRÖBERG, 2016) além de ilustrar como realizar a manipulação de elementos visuais durante a utilização das técnicas de teste (HALLER, 2013).

3.9 Aplicando técnicas de teste a partir de Ferramentas de Teste

Após a enumeração das técnicas de *black-box* na Seção 2, foram selecionadas duas técnicas para a aplicação na unidade de análise. Para o critério de seleção, foi considerada fortemente a inexistência de um processo de teste de software para o *app* em questão. Dessa forma, um dos critérios para a seleção das técnicas foi considerar que elas seriam um ponto de partida para o processo de teste desse tipo de aplicação para a empresa X. Por exemplo, a técnica *Fuzzing* citada anteriormente visa encontrar, na

maioria dos casos, falhas voltadas à segurança ou observar o comportamento do artefato quando alimentado com dados mal formatados, o que, nesse estágio, possui um grau de importância inferior quando comparado com outras técnicas mais elementares.

A técnica *ECP* foi escolhida, pois, segundo Kaur, Khatri e Datta (2014), ela, em comparação com outras técnicas, conseguiu encontrar a maior quantidade de erros dentre dez programas diferentes. Também ponderou-se que se deve escolher um conjunto mínimo de entradas para que exista a maior probabilidade de encontrar erros com uma quantidade mínima de casos de teste. Dessa forma, Myers (2004) cita que a técnica *ECP* reduz esse número de forma a atingir um nível de teste razoável. Já Spillner, Linz e Schaefer (2014) definem que a técnica *ECP* em conjunto com técnica *BVA,* devem ser aplicadas para cada objeto testado, porém neste caso a combinação complementar não foi utilizada.

A segunda técnica selecionada, *State-Transition Diagram (SM)*, deve ser aplicada quando diferentes estados possuem influência no objeto testado, o que na unidade de análise é verdadeiro, pois podemos verificar casos de uso no qual determinadas ações dependem de estados anteriores, como, por exemplo, a compra de um produto está condicionada à disponibilidade de saldo do cliente. Essa técnica é a única que verifica a cooperação de estados e transições (SPILLNER; LINZ; SCHAEFER, 2014).

As técnicas *all-pairs*, *cause-effect graph* e *decision table* não foram escolhidas, pois tiveram um peso menor já que não foram identificadas dependências nas entradas dos dados. Um exemplo disso é que as grandezas de descrição e preço de produto do *app* não possuíam uma relação como a citada por Spillner, Linz e Schaefer (2014), entre navegadores, idioma e tamanho de tela. A presença de múltiplos parâmetros com dependências não conhecidas, para justificar a aplicação destas técnicas (SPILLNER; LINZ; SCHAEFER, 2014) também não foi detectada, pois o *app* em questão possui um conjunto reduzido de parâmetros de entrada (código de barras, nome e preço do produto e valor de recarga).

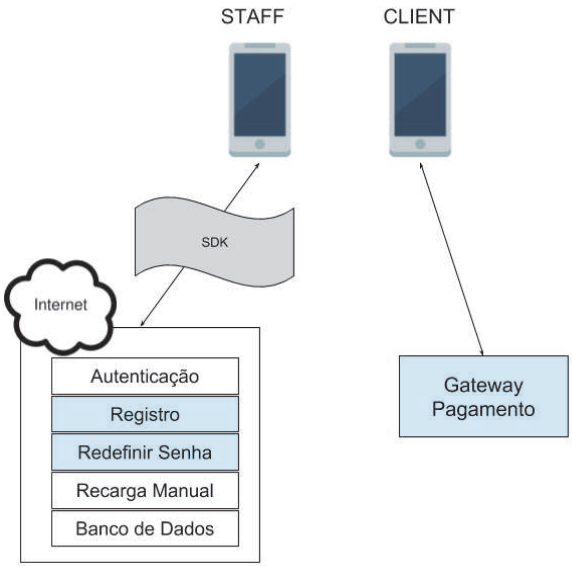
Os níveis possíveis de aplicabilidade das técnicas selecionadas para a unidade de análise podem ser componente, integração e sistema, se considerarmos as variáveis: o papel de desenvolvedor e testador do autor e a não participação do cliente no processo de teste. Optou-se por testar a aplicação em nível de integração, pois não foi possível encontrar documentações suficientes sobre a unidade de análise a nível de componente (interfaces requeridas e providas) para se realizar o teste a nível de unidade. A forma possível seria analisar em nível estrutural a aplicação, porém descaracterizaria o conceito de "caixa-preta" para esta pesquisa. O teste sistêmico foi descartado, pois possui uma complexidade maior em função da dependência de outros sistemas observados na arquitetura da unidade de análise. Por exemplo, durante as ações de registro de usuário um e-mail, com um código de ativação, é enviado para o cliente para que o mesmo realize a sua ativação. Da mesma forma, durante o processo de recarga de saldo, o *Gateway* de pagamento pode possuir diversas combinações de estados intermediários em decorrência do cartão de crédito utilizado. Em ambos os casos, uma análise profunda seria necessária com o objetivo de avaliar a melhor forma de testes de integração com esses outros sistemas. Para mitigar a quantidade e influência desses componentes ao escolher o nível de integração, algumas ações gerais foram necessárias para que o processo de teste fosse iniciado:

- abstração da existência de componentes como *Gateway* de pagamento e troca de informações via e-mail, considerando que os mesmos componentes não estão prontos, conforme visto na Figura 8, através da inserção de *stubs* (MYERS, 2004);

- sincronização do ambiente de desenvolvimento do autor;

- migração da base de dados de produção para desenvolvimento, permitindo a criação de novos usuários e a manipulação da base de dados (inserção de novos produtos, realização de recargas manuais, etc);
- criação de um usuário do tipo *Staff*;
- criação de um usuário do tipo *Client*;
- geração do *app* para as plataformas Android e iOS através do Ionic.

Figura 8. Arquitetura do objeto da Unidade de Análise com *stubs*



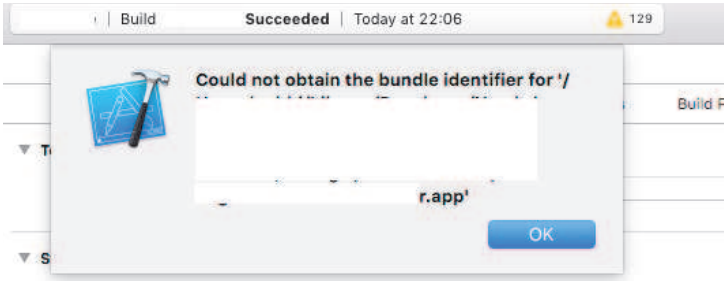
Fonte: elaborado pelo autor.

Após a seleção das duas técnicas, o segundo objetivo específico foi contemplado a partir do levantamento de ferramentas que permitiriam o uso de tais técnicas. A busca por ferramentas para a aplicação encontrou diversas opções porém foi fixado o requisito de considerar ferramentas oficiais usufruindo da máxima quantidade de recursos de cada sistema. Foi considerada uma opção para cada plataforma de destino do *app*: XCTest (XCTEST, 2018) para iOS e Espresso para o Android (ESPRESSO, 2018). A segunda requer o mínimo de configuração possível para a construção de testes funcionais e não funcionais (BÅNGERIUS; FRÖBERG, 2016). Ambas as ferramentas fornecem o recurso de "*Record and Play*" para testes de *UI (User Interface)*, sendo possível realizar a construção de casos de teste a partir de ações gravadas. Esse é um recurso interessante quando utilizado em técnicas de *black-box* pois a partir dessa interface as intervenções são construídas de forma simplificada, rápida e, em alguns cenários, sem a nenhuma necessidade de recompilar o *app*.

A partir da Tabela 5 observa-se que a maior quantidade de instalações envolve o sistema iOS, e como o autor desta pesquisa possui um ecossistema propício para a utilização deste sistema (ambiente de desenvolvimento, licença de desenvolvedor e um aparelho iPhone), optou-se pela tentativa de utilização do XCTest. Após o *build* com sucesso da plataforma iOS através do *framework* Ionic, o projeto foi incluído no ambiente de desenvolvimento oficial com o objetivo de validar minimamente a construção de casos de teste através do recurso de gravação. As tentativas de adaptações de *build* com a inclusão de testes gerados via gravação ocasionaram o erro de execução apresentado na Figura 9. Algumas tentativas foram realizadas com o objetivo de contornar o problema, porém sem sucesso, em decorrência da maneira em que os *targets*

de compilação do Cordova são definidos pelo Ionic. Não foram encontradas documentações oficiais e objetivas para contornar o problema.

Figura 9: Erro de execução na utilização de testes de *UI*



Fonte: elaborado pelo autor.

Após o revés da primeira tentativa, optou-se pela construção de testes voltados para a plataforma Android. O *build* para a plataforma foi realizado com sucesso e o projeto foi adaptado (Figura 10) através do ambiente oficial, Android Studio, para a inclusão de um teste mínimo realizado através do recurso de gravação do Espresso.

Figura 10: Configuração do Espresso para a plataforma Android

```
252
253  dependencies {
254      implementation fileTree(dir: 'libs', include: '*.jar')
255      // SUB-PROJECT DEPENDENCIES START
256      implementation(project(path: "CordovaLib"))
257      // SUB-PROJECT DEPENDENCIES END
258
259      compile 'com.android.support.test.espresso:espresso-web:3.0.2'
260      compile 'com.android.support.test:runner:1.0.2'
261      compile 'com.android.support.test:rules:1.0.2'
262      compile 'com.android.support.test.espresso:espresso-core:3.0.2'
263  }
264
```

Fonte: elaborado pelo autor.

O teste de gravação mínimo demonstrou um novo revés, dessa vez na utilização do recurso de gravação (Apêndice F). O recurso se baseia na manipulação das *Views* nativas, não sendo compatível para o *app* da unidade de análise pois o mesmo é executado em uma *WebView*. Para componentes desenvolvidos dessa forma, o Espresso permite, através de seu componente – Espresso Web – a construção de testes voltados a aplicativos híbridos, permitindo a busca e a manipulação visual dos elementos pertencentes a *WebView* de forma manual, conforme visto na Figura 11.

Figura 11: Utilizando *XPATH*[4] para a manipulação manual dos componentes

```
// Register screen
onWebView().withElement(findElement(Locator.XPATH,  value: "//*[@id=\"tabsControllerStaff-tabs1\"]/div/a[1]"))
        .perform(webClick());
onWebView().withElement(findElement(Locator.XPATH,  value: "//*[@id=\"page3\"]/ion-content/div[1]/p"))
        .check(webMatches(getText(), containsString( substring: "Adicionar Produto")));
```

Fonte: elaborado pelo autor.

Após a utilização do Espresso Web, foi possível construir um caso de teste mínimo comprovando a eficiência da ferramenta, permitindo a sua utilização para essa

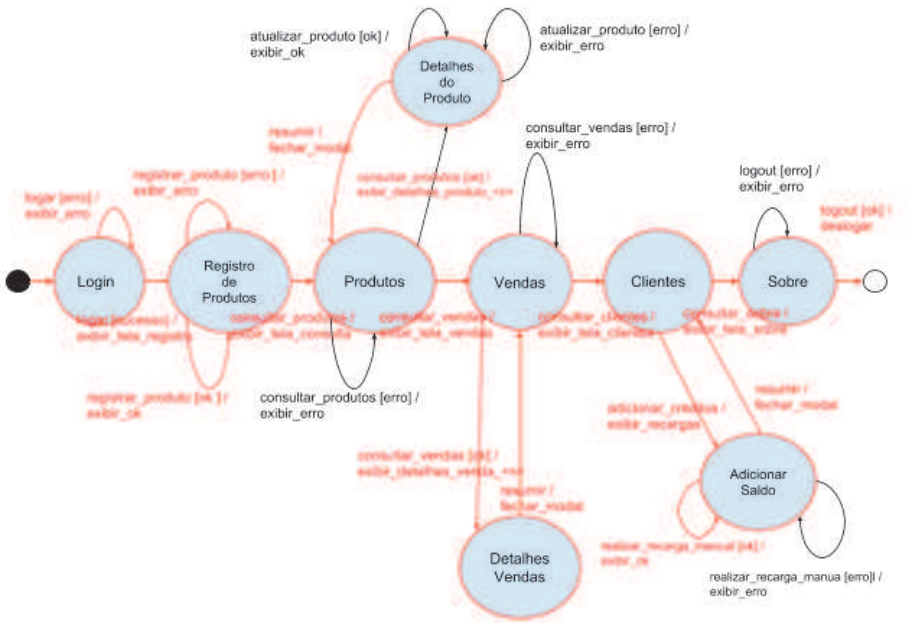
[4] *XPATH* é uma forma padrão definida pela *W3C* de realizar a navegação de elementos no HTML

pesquisa. A partir desse resultado, as técnicas selecionadas anteriormente começaram a ser aplicadas. Para aplicar as técnicas, o recurso padronizado pela *W3C*, entidade que regulamenta padrões voltados a *Web*, foi utilizado, o que corrobora com Haller (2013), que diz que os casos de teste baseados em *GUI* podem utilizar identificadores únicos para a manipulação de elementos. Para localizar esses identificadores, o recurso *serve* do Ionic foi utilizado para permitir o *debugging* local via *browser* e suas ferramentas, conforme visto no Apêndice D. Este processo foi efetuado de forma manual para cada elemento visual necessário, permitindo a construção de asserções (verificação do valor de entrada *versus* valor esperado).

Para a técnica *State-Transition Diagram (SM)*, inicialmente, a elaboração das máquinas de estados e transições foram construídas, pois este documento não foi encontrado nas bases privadas da empresa X. Nas Figuras 12 e 13, observa-se o resultado dessa construção. A partir das máquinas de estado para ambas as versões, a derivação dos casos de teste seguiu o modelo definido na seção 2 para essa técnica, onde estados e transições tiveram a sua cobertura definida conforme os seguintes critérios:

- versão *Staff*: estado inicial "Login" e estado final "Logout".

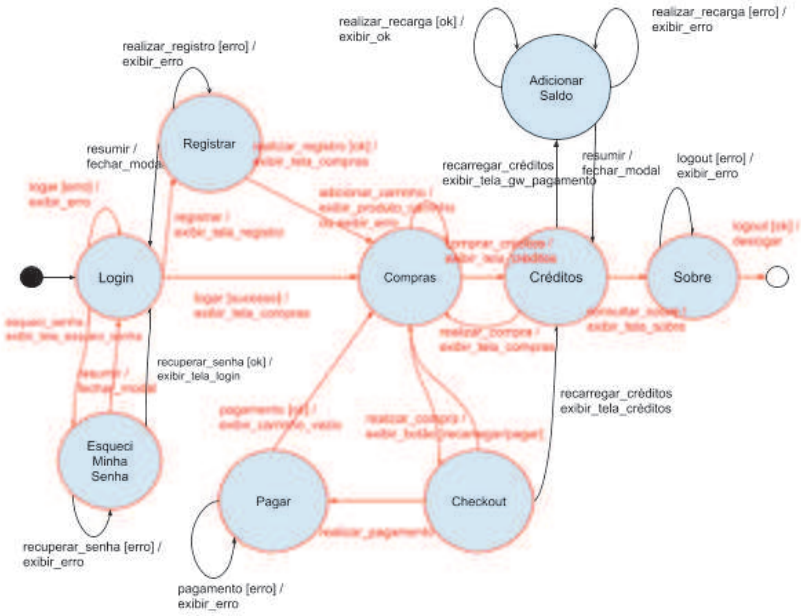
Figura 12: *SM* da versão *Staff*



Fonte: elaborado pelo autor.

- versão *Client*: estado inicial "Login" e estado final "Logout".

Figura 13: *SM* da versão *Client*



Fonte: elaborado pelo autor.

Para a técnica *ECP*, as classes tiveram que ser identificadas para a correta derivação dos casos de teste. Para ambas as versões, foram identificadas três classes conforme demonstra a Tabela 9.

Tabela 9: Identificação das classes: Versão *Staff*

#	Entrada	*ECP* Válida	*ECP* Inválida	Casos de teste do código fonte #2
1	Nome do Produto	Letras e números	-	A,B,C,D,E,G,H
2		-	Apenas números	M
3		-	Sem nome	N
4		-	Mínimo 5 caracteres	O
1	Preço do Produto	Valor Positivo	-	A,B,C,D,E,F,G,H,I,J,K
2		-	Valor Negativo	L
3		-	Valor zero	P
4		-	Caracteres inválidos	Q
1	Código de Barras do Produto	CODE128 Auto	-	A,E
2		CODE128 A	-	B,E
3		CODE128 B	-	C,E
4		CODE128 C	-	D,E
5		EAN13	-	E
6		EAN8	-	F,E
7		UPC	-	G,E

8		CODE 39	-	H,E
9		ITF14	-	I,E
10		ITF	-	J,E
11		PHARMACODE	-	K,E

Fonte: elaborado pelo autor.

Analisando a especificação do produto, não foi possível determinar quais são os formatos de códigos de barras suportados pela aplicação para a classe código de barras. A partir da base de dados corrente, é possível elencar os padrões EAN-13 e UPC, porém optou-se por estender essa quantidade através de uma ferramenta (BARCODE, 2018), conforme visto nas linhas das Tabelas 9 e 10.

Tabela 10: Identificação das Classes: Versão *Client*

#	Entrada	*ECP* Válida	*ECP* Inválida	Casos de teste do código fonte #4
1	Código de Barras do Produto	CODE128 Auto	-	A
2		CODE128 A	-	B
3		CODE128 B	-	C
4		CODE128 C	-	D
5		EAN13	-	E
6		EAN8	-	F
7		UPC	-	G
8		CODE 39	-	H
9		ITF14	-	I
10		ITF	-	J
11		PHARMACODE	-	K

Fonte: elaborado pelo autor.

Após o levantamento de dados e comprovação de que a ferramenta de teste escolhida funciona, a construção de casos de teste continuou utilizando o Android Studio. Quatro novos arquivos foram incluídos no projeto, conforme Tabela 11, que ilustra a quantidade de casos de teste construídos para cada técnica. Para as técnicas *ECP*, a cobertura de cada classe é identificada através do nome do caso de teste na última coluna das Tabelas 9 e 10. Para a técnica *SM*, as Figuras 12 e 13 destacam em vermelho a cobertura de transições e estados de cada máquina considerando algumas premissas:

- a cobertura de determinados arcos que envolvem falhas não foram incluídas, pois seria necessária a injeção de *stubs* em pontos estratégicos, uma vez em que essas transições dependem de serviços fornecidos por *SDKs* (ex: erro ao não ser possível realizar a leitura do banco de dados) de terceiros ou de disponibilidade de conectividade.

- a disponibilidade de conectividade pode ser alterada durante a execução dos casos de teste, porém, foi observado que é possível apenas de forma manual, o que inviabilizaria o ganho de automatização promovido pelo Espresso.

- apesar de existir a possibilidade de compartilhar a *Webcam* do notebook com o simulador do Android, com o objetivo de suportar a câmera, foi detectado que,

em decorrência de permissões do sistema operacional, essa manipulação programática não é trivial, por isso não foi considerada.

- durante a atualização de produtos, por exemplo, não foi possível acessar os elementos visuais de acordo com o *XPATH* em decorrência do Ionic.

A seguir, a síntese dos testes baseados em *SM*:

- *Staff Case 1*: se propõe a realizar transições gerais.
- *Client Case 1*: transições para autenticação e compra de produto.
- *Client Case 2*: se propõe a realizar transições voltadas ao registro de usuário.

Tabela 11: Arquivos fontes dos casos de teste

#	Código Fonte	Qtd. Casos de teste	Casos de teste
1	StaffTest_SM.java	1	Case1
2	StaffTest_SM_ECP.java	17	A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q[5]
3	ClientTest_SM.java	2	Case1, Case2
4	ClientTest_SM_ECP.java	12	A,B,C,D,E,F,G,H,I,J,K,L
	Total	32	-

Fonte: elaborado pelo autor.

Além dos arquivos de testes, um arquivo chamado *ExternalStubsAndDrivers.java* foi incluído para abstrair a comunicação com os componentes externos, garantindo que a comunicação externa espere o tempo necessário para que as requisições sejam atendidas com sucesso.

4. Análise e Apresentação de Resultados

Para a execução dos casos de teste foi utilizado o emulador do Android, conforme citado por Haller (2013). A versão do Android utilizada no emulador levou em consideração as informações listadas no painel de desenvolvedor, que o autor possui acesso (GOOGLE, 2018). Através das estatísticas fornecidas pelo Google neste painel, foi possível detectar que a versão Marshmallow (Android 6) possui maior incidência nos dispositivos que utilizam a aplicação.

Para realizar a análise de efetividade dos testes aplicados, foram consideradas informações de estatísticas de *bugs* nas versões do *app*, a partir de dados do *bug tracking* da empresa X, e dados referentes a especificação do artefato (casos de uso). A partir da coleta, três versões do aplicativo foram selecionadas para a execução dos testes utilizando o Android 6. Essas versões consideraram recursos compatíveis permitindo uma homogeneidade das máquinas de estados e classes de equivalência, tornando possível a repetibilidade dos casos de teste entre as versões. Também foram

[5] O nome dos casos de teste seguiram um padrão diferente nos arquivos #2 e #3 para garantir a ordem léxica de execução.

considerados os *bugs* pertinentes para cada versão selecionada, tendo como objetivo, verificar a cobertura das técnicas frente a estes.

Para a primeira amostra de testes, a última versão do artefato, 1.0.8, foi gerada e instalada no simulador de destino. A partir disso, os testes foram executados e os resultados gerados foram organizados conforme a Tabela 12. Todos os dados foram extraídos com base em relatórios gerados pelo próprio Android Studio, conforme visto no exemplo do Apêndice C.

Tabela 12: Compilação de resultados da versão 1.0.8

Versão	Versão *app*	Técnica	Sucesso	Falha	Tempo
1.0.8	*Staff*	*SM*	Case1	-	02’09’’
1.0.8	*Staff*	*SM+ECP*	D,E,F,G,I,J,N	A,B,C,H,K,L,M,O,P,Q	24’18’’
1.0.8	*Client*	*SM*	Case1, Case2	-	04’06’’
1.0.8	*Client*	*SM + ECP*	A,D,E,F,G,I,J,L	B,C,H,K	18’04’’

Fonte: elaborado pelo autor.

Na segunda amostragem, foi necessário realizar uma manipulação no sistema de controle de versões para que se atingisse o estado da versão 1.0.6, incluindo as alterações de *stubs* dos componentes. A partir disso, o binário final foi gerado e instalado no simulador. Os resultados estão ilustrados na Tabela 13.

Tabela 13: Compilação de resultados da versão 1.0.6

Versão	Versão *app*	Técnica	Sucesso	Falha	Tempo
1.0.6	*Staff*	*SM*	-	Case1	01’24’’
1.0.6	*Staff*	*SM+ECP*	-	A-Q	19’26’’
1.0.6	*Client*	*SM*	Case1,Case2	-	03’51’’
1.0.6	*Client*	*SM + ECP*	A,D,E,F,G,I,J,L	B,C,H,K	17’53’’

Fonte: elaborado pelo autor.

Na terceira e última amostragem, o mesmo processo foi realizado tendo como resultado a Tabela 14.

Tabela 14: Compilação de resultados da versão 1.0.5

Versão	Versão *app*	Técnica	Sucesso	Falha	Tempo
1.0.5	*Staff*	*SM*	-	Case1	01’27’’
1.0.5	*Staff*	*SM+ECP*	-	A-Q	19’26’’
1.0.5	*Client*	*SM*	Case2	Case1	03’36’’
1.0.5	*Client*	*SM + ECP*	A,D,E,F,G,I,J,L	B,C,H,K	18’06’’

Fonte: elaborado pelo autor.

Após a compilação das execuções de cada teste, pode-se realizar a seguinte síntese para a técnica *SM*:

- Na versão 1.0.8 do aplicativo, os testes baseados em máquinas de estados verificaram, de forma efetiva, os estados e coberturas definidas. Nenhum dos

bugs que permanecem em aberto nessa versão (BugIDs: 350, 351 e 353) foram detectados.

• Nas versões 1.0.5 e 1.0.6 versão *Staff*, foi encontrada uma falha na qual o teste manifesta um defeito ao realizar a transição para o estado "Detalhes Vendas" (BugID: #346). Em consequência disso, os casos de teste seguintes não são executados, pois essa transição sempre é verificada.

• Na versão 1.0.5 versão *Client*, não foi possível realizar a compra de produtos utilizando todo o saldo previamente carregado (BugID: #345).

Na técnica *SM + ECP* para a versão *Staff*, observa-se:

- Versão 1.0.8, 1.0.6 e 1.0.5:
  - As classes válidas de códigos de barra CODE128 Auto, CODE128 A, CODE128 B e CODE39 não foram aceitas pelo aplicativo.
  - O código de barras PHARMACODE não foi aceito em decorrência de um mascaramento de falha no caso de teste que realiza a cobertura da classe CODE128 B.
  - Foi possível realizar o registro de produtos com preço negativo, preço com valor zero e com a descrição do produto com caracteres apenas numéricos.
  - Foi observado que a regra de negócio que exige que a descrição do produto contenha pelo menos cinco caracteres não está correta, pois foi possível registrar um produto com quatro caracteres.
  - Foi possível cadastrar o preço de um produto com caracteres alfanuméricos, deixando-o com o valor zero.
- Versão 1.0.6:
  - Não é possível realizar a transição para o estado "Detalhes Vendas" (BugID: #346).

Para a versão *Client*:

- Versão 1.0.8, 1.0.6 e 1.0.5:
  - Não foi possível realizar a compra de produtos das seguintes classes de equivalência válidas: CODE128 A, CODE128 B e CODE39.
  - Não foi possível validar os dados de descrição do produto que foi cadastrado como PHARMACODE em decorrência do caso de teste CODE128 B.

Tabela 15: Compilação final de todos os resultados coletados

Versão	Técnica	Qtd. casos de teste	Qtd. de *Bugs* Conhecidos	Qtd. de *Bugs* Novos	Tempo Total Gasto
1.0.8	*SM (Staff+Client)*	3	0	0	06'15''
1.0.6	*SM (Staff+Client)*	3	1 (BugID: #346)	0	05'15''
1.0.5	*SM (Staff+Client)*	3	2 (BugIDs: #345 e #346)	0	05'03''
1.0.8	*SM + ECP*	29	0	10	42'22''

	(Staff+Client)				
1.0.6	*SM + ECP (Staff+Client)*	29	1 (BugID: #346)	17	37’19’’
1.0.5	*SM + ECP (Staff+Client)*	29	2 (BugIDs: #345 e #346)	4	37’32’’

Fonte: elaborado pelo autor.

Ao observar os resultados acima, pode-se identificar dois comportamentos com relação às técnicas aplicadas. O primeiro é que a técnica *SM* foi capaz de detectar incoerências de transições e eventos nas versões que possuíam problemas com essas características, independentemente da técnica ser aplicada de forma isolada ou em conjunto. Já o uso das classes permitiu observar incoerências na especificação originadas por possíveis falhas no processo de elicitação de requisitos. Isso foi observado no contexto dos códigos de barras quando o domínio da classe foi ampliado com o objetivo de avaliar o comportamento do artefato. Durante essa observação, percebeu-se que o mesmo não está preparado para consumir determinados padrões porém, não é possível afirmar se isso é um impacto para as necessidades de negócios. O resultado também ilustra que a dependência de conectividade com outros sistemas pode ampliar o tempo total de execução dos casos de teste (*ExternalStubsAndDrivers.java).*

5. Considerações Finais

O objetivo geral desta pesquisa, aplicar algumas técnicas *black-box* em um aplicativo híbrido, destacou que o processo de teste voltado ao contexto de dispositivos móveis tende a possuir um grau diferenciado de complexidade, em decorrência de seu amplo âmbito de variáveis presentes inicialmente no processo de desenvolvimento e também posteriormente durante o seu ciclo de manutenção. Os aplicativos híbridos visam minimizar o impacto desse desenvolvimento, fornecendo ferramentas que reduzem o tempo nesse primeiro estágio, porém, durante essa pesquisa, foi observado que a consequência disso incide sobre o processo de teste. Essa incidência foi observada nas tentativas de uso das ferramentas oficiais para a construção dos testes para a unidade de análise, onde foram detectadas dificuldades na integração de *build* e na utilização de recursos de “*Record and Play”,* uma vez em que esse último se baseia em *Views* nativas, o que não é o caso de aplicativos desenvolvidos de forma híbrida.

Pode-se considerar que os aplicativos, de forma geral, tendem a utilizar recursos de hardware específicos e de infraestruturas hospedadas na nuvem, conforme identificado na unidade de análise (câmera e conectividade). Idealmente, deve-se realizar o teste considerando a maior variabilidade de dispositivos físicos, verificando o seu comportamento em diferentes versões de sistemas operacionais, já que este aspecto pode impactar no comportamento destes recursos.

Na unidade de análise, as premissas adotadas, com o objetivo de viabilizar o terceiro objetivo específico, considerando um ambiente de desenvolvimento e teste reduzidos, facilitaram a construção de casos de teste, pois minimizaram a influência destas variáveis externas. A partir dos resultados gerados pelo ambiente de construção de testes, foi possível analisar os resultados comparando com as informações coletadas nas documentações da empresa X.

Foi possível detectar o impacto dessas variáveis, por exemplo, através da ausência de cobertura para *bugs* que dependem da versão do sistema e de recursos específicos, como a câmera, conforme visto nos BugIDs: #350 e #343. Entende-se que nesses casos, o custo para a cobertura de todas as versões do sistema operacional é alto e complexo para um ambiente mínimo, pois depende de altos recursos computacionais, uma vez que as imagens de sistemas simulados são extremamente pesadas, conforme visto no Apêndice G. Além disso, no caso de recursos de hardware (câmera), não se pode abrir mão da automatização fornecida pelos *frameworks* de teste, ainda mais se a

necessidade de se utilizar múltiplos dispositivos existe. Conclui-se que, mesmo não considerando essas variáveis externas, as técnicas selecionadas e aplicadas nesse artigo têm a possibilidade de cobertura para esses *bugs*, já que estão representadas na cobertura dos estados e nas classes identificadas.

Os casos de teste construídos nesta pesquisa, demonstraram a sua efetividade para defeitos já conhecidos do sistema, como por exemplo os BugIDs: #345 e #346. Sendo assim, através destes resultados, foi possível verificar que as técnicas aplicadas na unidade de análise identificaram lacunas na especificação do projeto e inconformidades nas transições das máquinas de estados. Salienta-se que um grande desafio foi encontrado durante a aplicação das técnicas: a interceptação dos elementos visuais via *XPATH* comprovou não ser robusta quando a estrutura do Ionic era alterada, tornando os casos de teste inválidos. Dessa forma, para trabalhos futuros sugere-se:

1. analisar outras formas de localização dos elementos contidos na *WebView*, evitando que alterações de estrutura do HTML5 impactem na execução dos casos de teste.
2. encontrar formas de viabilizar os recursos de gravação para a construção de testes baseados em *WebViews*.
3. encontrar formas de detectar, antecipadamente, os elementos do Ionic, utilizados na construção do *app*, que impossibilitam a construção de testes.
4. aplicar outras técnicas *black-box* em soluções híbridas, além das utilizadas nesta pesquisa.

Com o resultado deste trabalho, será possível aplicar estas técnicas para outros *apps* desenvolvidos pelo autor, permitindo o início de um processo de teste de software para a empresa X.

6. Referências

ANDROID. Android Guidelines. Disponível em: <https://developer.android.com/guide/index.html>. Acesso em março de 2018.

ANDROID1. Distribution Dashboard. Disponível em: < https://developer.android.com/about/dashboards/>. Acesso em março de 2018.

AZEVEDO, Debora; MACHADO, Lisiane; DA SILVA, Lisiane Vasconcellos. Métodos e procedimentos de pesquisa: do projeto ao relatório final. Unisinos, 2011.

BLACK, Rex; MITCHELL, Jamie. Advanced Software Testing. Vol 3. 1. ed. 2011.

BÅNGERIUS, S.; FRÖBERG, F. Functional testing of an Android application. 2016

BARCODE. Barcode Generator. Disponível em: <http://lindell.me/JsBarcode/generator/>.Acesso em março de 2018.

CORDOVA. Cordova. Disponível em: <https://cordova.apache.org/>. Acesso em março de 2018.

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

GOOGLE. Google Play Console. Disponível em: https://developer.android.com/distribute/console//>. Acesso em março de 2018.

HALLER, K. Mobile Testing. ACM SIGSOFT Software Engineering Notes, v. 38, n. 6, p. 1–8, 2013.

IONIC. Ionic Framework. Disponível em: <https://ionicframework.com>. Acesso em março de 2018.

IOS. iOS Guidelines. Disponível em: <https://developer.apple.com/app-store/guidelines/>. Acesso em março de 2018.

ISTQB. Glossary. Disponível em: <https://glossary.istqb.org>. Acesso em março de 2018.

ITUNES. Itunes Connect. Disponível em: <https://itunesconnect.apple.com/loginAcesso>. Acesso em março de 2018.

JENKINS, Nick. A software Testing Primer – An Introduction to Software Testing. 2008.

KHAN, M. E. DIfferent Approaches To Black Box Testing Techniques For FInding Errors. International Journal of Softwar Engineering & Applications, v. 2, n. 4, p. 31–40, 2011.

KAUR, K.; KHATRI, S. K.; DATTA, R. Analysis of various testing techniques. International Journal of System Assurance Engineering and Management, v. 5, n. 3, p. 276–290, 2014.

LAKATOS, Eva Maria; MARCONI, Marina de Andrade. Técnicas de pesquisa: planejamento e execução de pesquisas, amostragens e técnicas de pesquisas, elaboração, análise e interpretação de dados. 7. ed. São Paulo: Atlas, 2009.

MALHOTRA, N. R. Pesquisa de marketing: uma orientação aplicada. Porto Alegre: Bookman, 2001.

MYERS, Glenford J. The Art of Software Testing. 2. ed. 2004.

RÄTZMANN, Manfred; YOUNG, Clinto De. Galileo Computing Software Testing and Internationalization. 2003.

REDMINE. Redmine. Disponível em: <https://www.redmine.org>. Acesso em março de 2018.

RICHARDSON, Robert Jarry. Pesquisa social: métodos e técnicas. 3. ed. São Paulo: Atlas, 1999.

SPILLNER, Andreas; LINZ, Tilo; SCHAEFER, Hans. Software Testing Foundations. 4. ed. 2014.

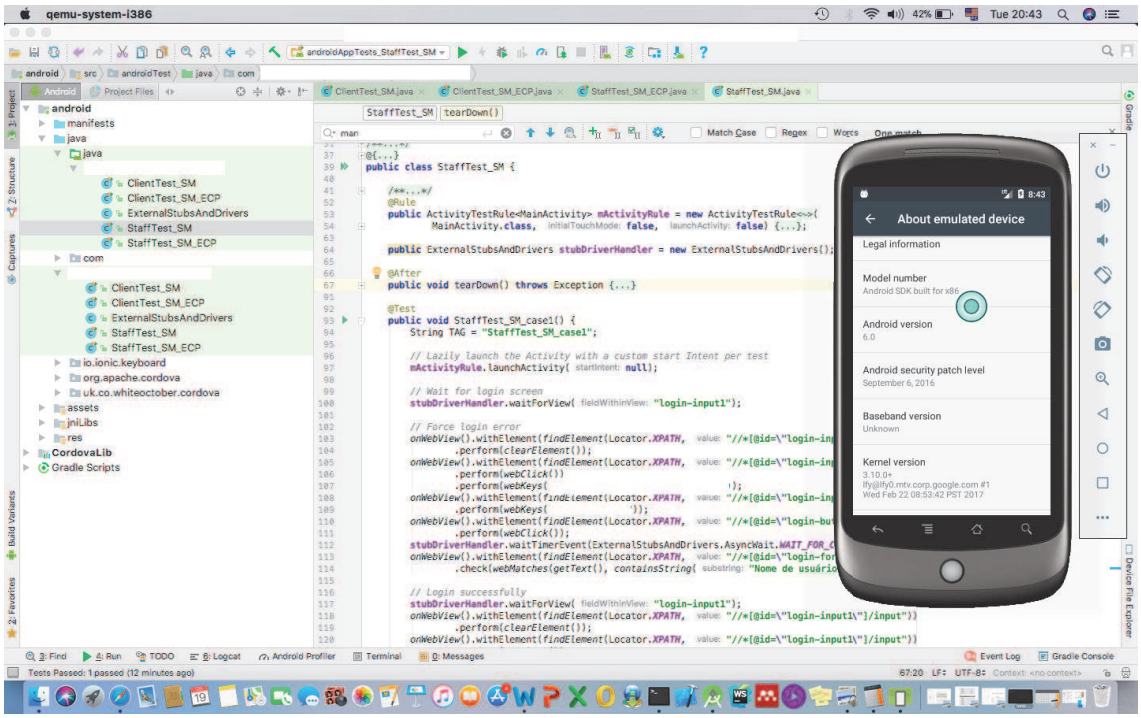
TITANIUM. Titanium. Disponível em: <https://www.appcelerator.com/Titanium/>. Acesso em março de 2018.

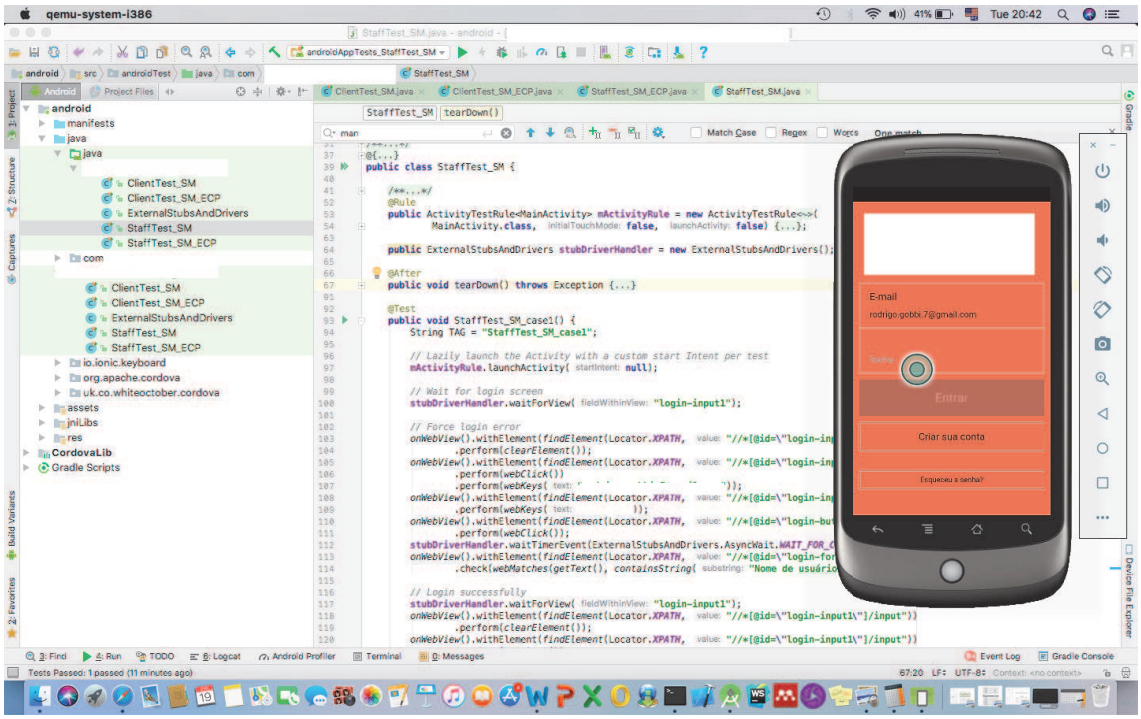
XAMARIN. Xamarin. Disponível em: <https://www.xamarin.com>. Acesso em março de 2018.

YIN, Robert K. Estudo de caso: planejamentos e métodos. 4. ed. Porto Alegre: Bookman, 2010.BÅNGERIUS, S.; FRÖBERG, F. Functional testing of an Android application. 2016.

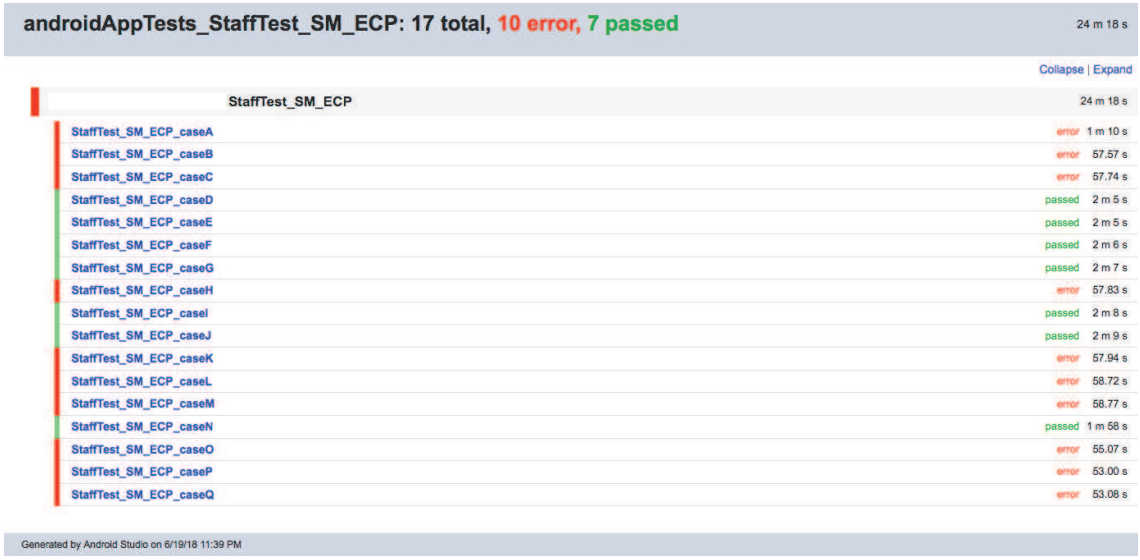
APÊNDICE A – Casos de Uso da Unidade de Análise

APÊNDICE B – Ambiente de Testes





APÊNDICE C – Exemplos de resultados de execução gerados pelo Android Studio

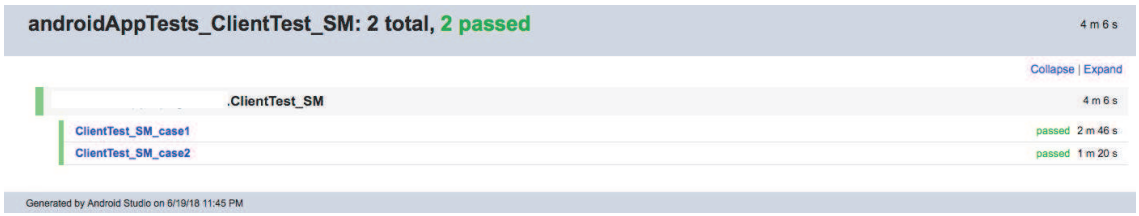


androidAppTests_StaffTest_SM_ECP: 17 total, 10 error, 7 passed — 24 m 18 s

Collapse | Expand

StaffTest_SM_ECP	24 m 18 s
StaffTest_SM_ECP_caseA	error 1 m 10 s
StaffTest_SM_ECP_caseB	error 57.57 s
StaffTest_SM_ECP_caseC	error 57.74 s
StaffTest_SM_ECP_caseD	passed 2 m 5 s
StaffTest_SM_ECP_caseE	passed 2 m 5 s
StaffTest_SM_ECP_caseF	passed 2 m 6 s
StaffTest_SM_ECP_caseG	passed 2 m 7 s
StaffTest_SM_ECP_caseH	error 57.83 s
StaffTest_SM_ECP_caseI	passed 2 m 8 s
StaffTest_SM_ECP_caseJ	passed 2 m 9 s
StaffTest_SM_ECP_caseK	error 57.94 s
StaffTest_SM_ECP_caseL	error 58.72 s
StaffTest_SM_ECP_caseM	error 58.77 s
StaffTest_SM_ECP_caseN	passed 1 m 58 s
StaffTest_SM_ECP_caseO	error 55.07 s
StaffTest_SM_ECP_caseP	error 53.00 s
StaffTest_SM_ECP_caseQ	error 53.08 s

Generated by Android Studio on 6/19/18 11:39 PM



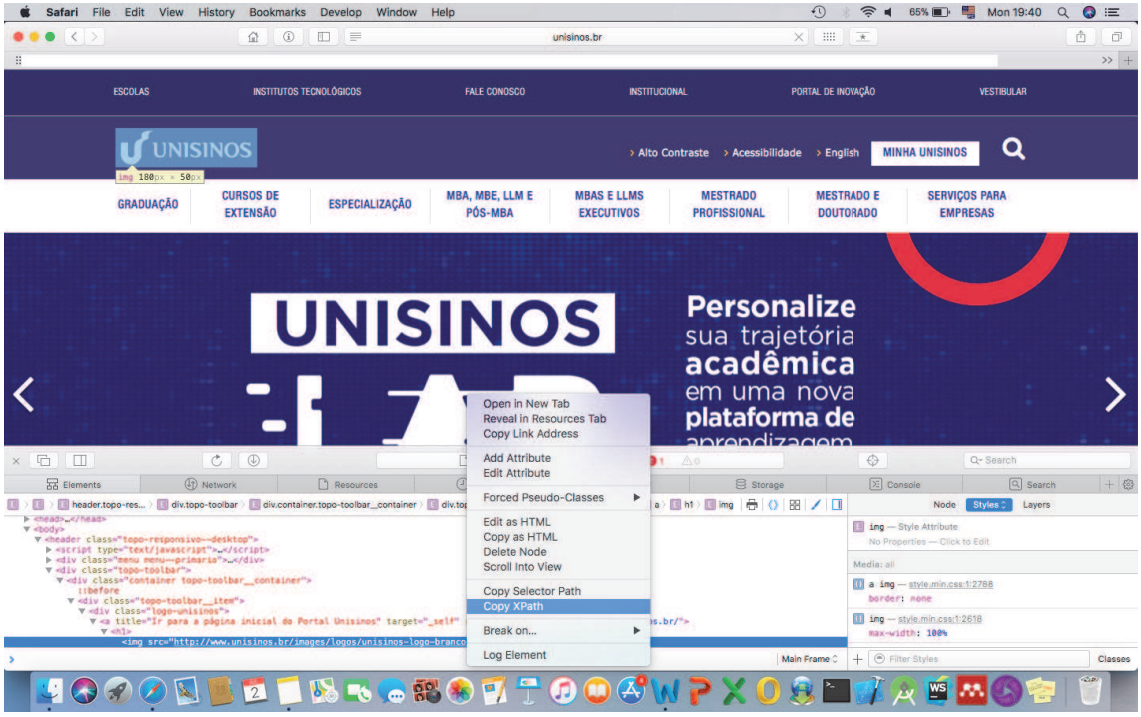
androidAppTests_ClientTest_SM: 2 total, 2 passed — 4 m 6 s

Collapse | Expand

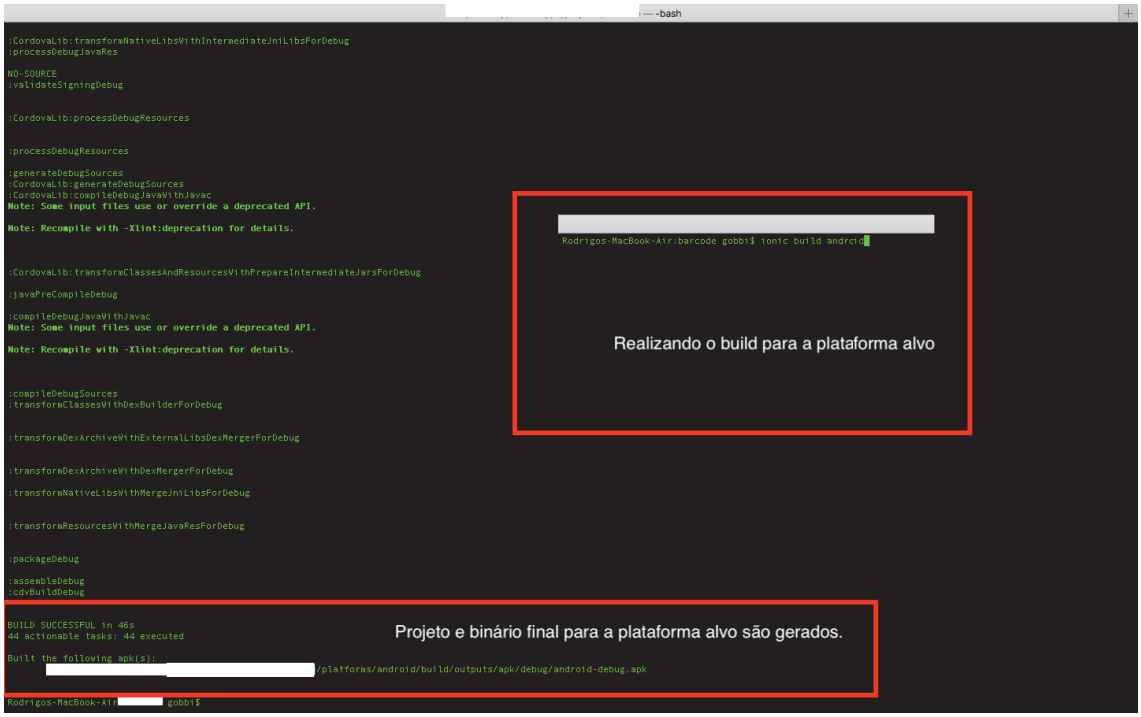
.ClientTest_SM	4 m 6 s
ClientTest_SM_case1	passed 2 m 46 s
ClientTest_SM_case2	passed 1 m 20 s

Generated by Android Studio on 6/19/18 11:45 PM

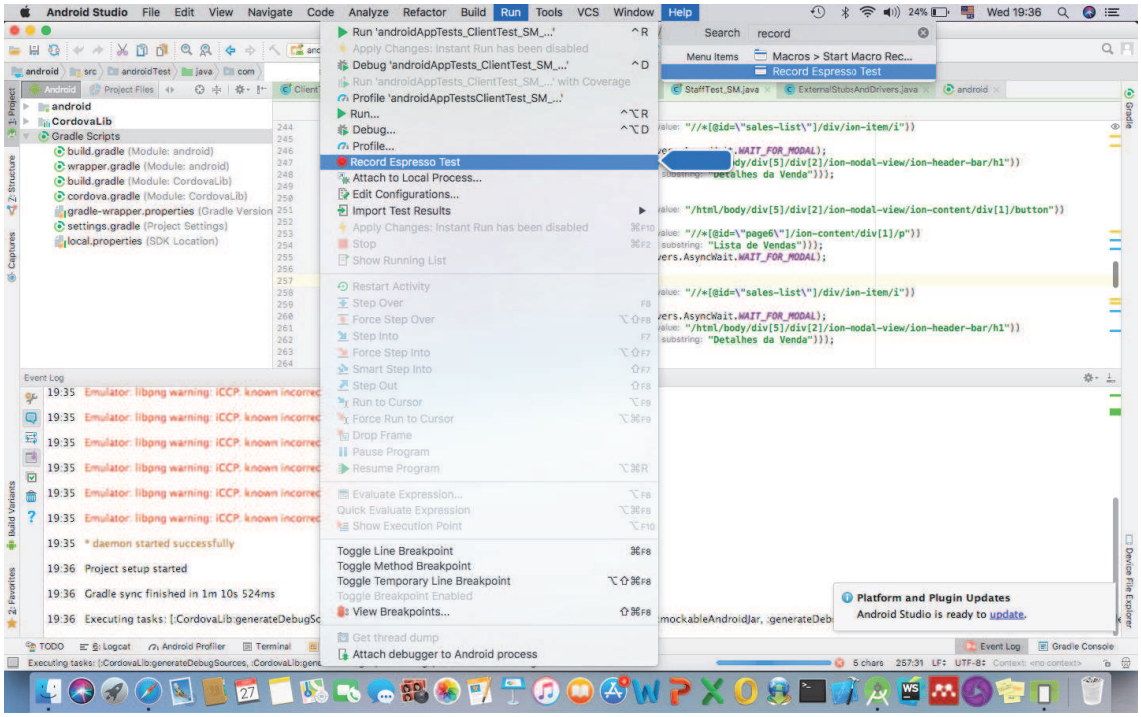
APÊNDICE D – Captura do *XPATH* através do *browser*

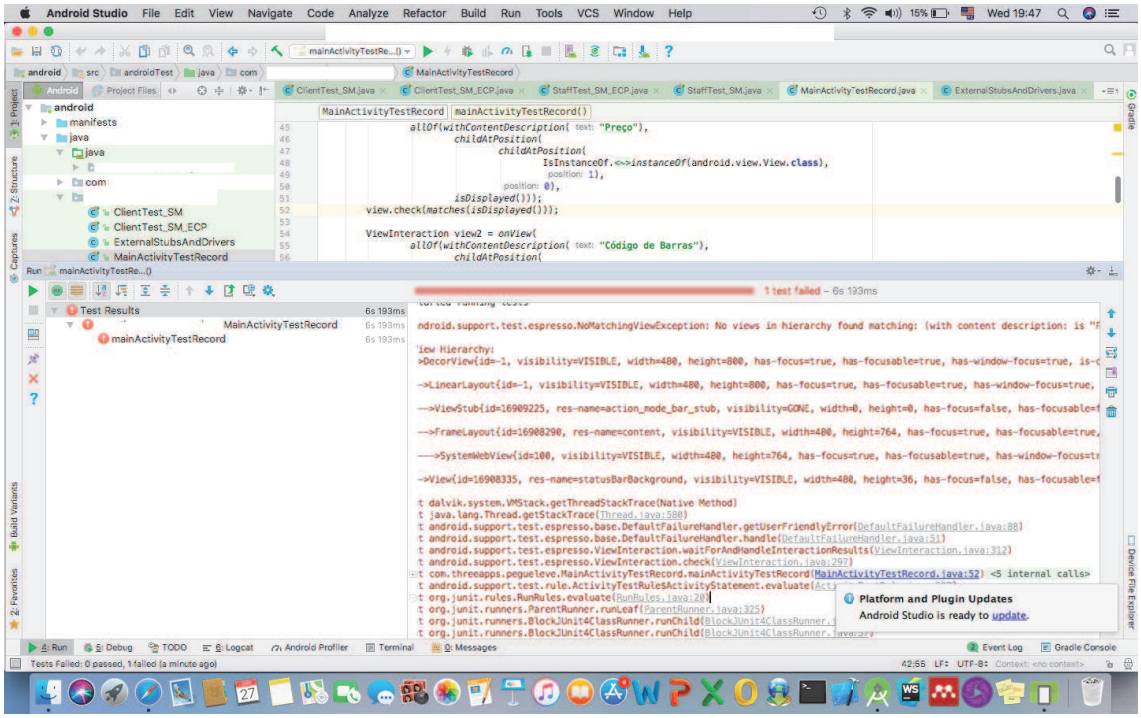


APÊNDICE E – Ionic: realizando o *build* para a plataforma alvo



APÊNDICE F – Tentativa de realizar casos de teste baseado no Espresso Recorder





APÊNDICE G – Tamanho da imagem do Android utilizada nos testes