UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FERNANDO WEBER ALBIERO

Uma abordagem de teste para aplicativos Android utilizando os cenários do Behavior Driven Development

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Profª. Drª. Érika Cota

Porto Alegre

2017

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Weber Albiero, Fernando
Uma abordagem de teste para aplicativos Android utilizando os cenários do Behavior Driven Development / Fernando Weber Albiero. -- 2017.
75 f.

Orientadora: Érika Fernandes Cota.

Dissertação (Mestrado) -- Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, Porto Alegre,
BR-RS, 2017.

1. Teste de Sistema. 2. Teste Automatizado. 3. Android. 4. Computação Móvel. 5.Behavior Driven Development. I. Fernandes Cota, Érika, orient. II. Título.

RESUMO

Os aplicativos móveis, desenvolvidos originalmente para a área do entretenimento, hoje estão presentes nos mais diversos domínios, sendo comuns inclusive em áreas de alto valor agregado, como: varejista, logística, bancária, médica, entre outras. Portanto, a qualidade e correção dos aplicativos móveis tornam-se obrigatórios e as atividades de teste essenciais. Porém a qualidade das aplicações móveis nem sempre é satisfatória. Isso ocorre devido ao fato dessas aplicações sofrerem com a pressão do mercado e passarem por um processo muito rápido de desenvolvimento, onde geralmente a fase de testes é negligenciada ou realizada de forma superficial, pela própria equipe de desenvolvimento, comprometendo assim a qualidade da aplicação. Este trabalho propõe uma abordagem baseada no *Behavior Driven Development* para ajudar na definição de testes de sistema para aplicativos nativos do Android. A abordagem proposta utiliza os arquivos de leiaute da aplicação para extrair informações sobre os componentes da interface e sobre os eventos esperados pelo sistema. A partir dessas informações, é possível verificar a cobertura dos cenários existentes em relação aos eventos disponíveis na interface com o usuário. Além disso, é possível identificar elementos do leiaute que não são exercitados pelos cenários existentes. A abordagem proposta é implementada por uma ferramenta chamada *Android Behavior Testing Tool* que, por meio da interpretação dos cenários do *Behavior Driven Development*, fornece uma visão geral do fluxo comportamental da aplicação ao testador (visão hoje não disponível), proporcionando assim uma noção de fácil compreensão sobre a cobertura dos testes em relação aos elementos da interface do aplicativo. Desta forma, o testador pode julgar a integridade dos casos de teste disponíveis em relação às funcionalidades implementadas e, se necessário, implementar novos testes. A ferramenta também faz uso dos arquivos de leiaute do aplicativo para identificar os componentes da interface que não foram testados e gera, neste caso, modelos de cenários no formato do BDD, automatizando assim a tarefa de escrita dos mesmos. A abordagem proposta foi utilizada em quatro aplicativos Android e se mostrou útil, uma vez que, em três estudos de caso foram detectados bugs oriundos de inconsistências lógicas nos cenários ou elementos não exercitados pelos cenários.

Palavras-chave: Teste de sistema. Automatização de cenários. Android. Computação móvel. Behavior Driven Development.

A test approach for Android apps using the Behavior Driven Development scenarios

ABSTRACT

Mobile applications, originally developed for entertainment, nowadays are present in a wide range of domains, being common even in areas of high value such as retailer, logistics, banking, and medical, among others. However, the quality and correctness of mobile applications become mandatory and testing activities are essential. However, the quality of mobile applications is not always good enough. This is because these applications suffer from market pressure and pass through a very rapid development process where the testing phase usually is neglected or superficially performed by the development team itself, thus compromising the quality of the application. This work proposes an approach based on Behavior Driven Development to help to define system tests for native Android applications. The proposed approach uses the application's layout files to extract information about the interface components and the events expected by the system. From this information, it is possible to check out the coverage of existing test scenarios against events available in the user interface. In addition, it is possible to identify unexercised usage scenarios from the existing test scenarios. The proposed approach is implemented by a tool called Android Behavior Testing Tool which, through the interpretation of the BDD usage scenarios, provides to the tester an overview of the behavioral flow of the application (otherwise unavailable), thus providing a notion of easy understanding of test coverage in relation to the application interface elements. In this way, the tester can judge the integrity of the available test cases in relation to the functionalities implemented and, if necessary, implement new tests. The tool also makes use of the application's layout files to identify untested interface components and in this case generates test scenario models in the BDD format, thus automating the writing task of the scenarios. The proposed approach was used in four Android applications and proved to be useful, since in three case studies bugs were detected. Detected bugs originated from logical inconsistencies in the test scenarios or elements that were not exercised by the scenarios.

Keywords: System test. Scenarios automatization. Android. Mobile computing. Behavior Driven Development.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABTT	Android Behavior Testing Tool
API	Application Programming Interface
A^2T^2	Android Automatic Testing Tool
BDD	Behavior Driven Development
CSS	Cascade Style Sheet
GUI	Graphical User Interface
GPS	Global Positioning System
GV	Graphviz
HTML	Hyper Text Markup Language
IMC	Índice de Massa Corporal
SO	Sistema Operacional
TDD	Test Driven Development
TXT	Text
UI	User Interface
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

A constante evolução da indústria do hardware permite a fabricação de dispositivos móveis cada vez mais complexos. Quando comparados com os dispositivos móveis fabricados há alguns anos, os dispositivos atuais possuem componentes mais rápidos, como processadores e memórias, e oferecem serviços de melhor qualidade, como conexões mais rápidas com a internet e suporte a diferentes formatos de mensagens. Além disso, possuem sistemas operacionais mais eficientes e um maior número de sensores, o que possibilita a utilização de aplicativos mais complexos e robustos. Aplicativos que antes eram uma facilidade, hoje em dia tornaram-se uma necessidade, e estão cada vez mais presentes na vida dos usuários. As aplicações móveis, originalmente desenvolvidas para o setor de entretenimento, hoje encaixam-se nas mais diversas categorias e são cada vez mais comuns em domínios de alto valor agregado (varejista, logístico, transporte aéreo) ou mesmo críticos (setores bancários, médico, etc). Em muitos casos, a interface móvel do negócio é crucial para sua manutenção e crescimento. Portanto, a qualidade e correção dos aplicativos móveis tornam-se obrigatórias, e as atividades de testes, essenciais.

O teste de software é uma importante ferramenta em busca da qualidade de uma aplicação, que deve ser confiável, consistente e previsível, não apresentando surpresas para seus usuários. O teste de software pode ser definido como a atividade, ou o conjunto de atividades, realizadas para verificar se o comportamento de um software está em conformidade com o especificado (PIRES DE SOUZA e GASPAROTTO, 2013). O principal objetivo do teste é encontrar defeitos no sistema, pois sabe-se que, à medida que o desenvolvimento avança, mais caro e difícil torna-se a solução de um problema (MYERS, BADGETT e SANDLER, 2011). Revelar o máximo de falhas no sistema o mais cedo possível, minimiza os custos com manutenção e com consequentes transtornos aos usuários (NETO, 2007). A gravidade de uma falha de software é relativa. Existem falhas que não afetam o sucesso da aplicação, com as quais usuários podem conviver. Porém, em outros casos, a falha do programa representa um completo fracasso comercial ou até mesmo um risco para a segurança física dos usuários (KOSCIANSKI e SOARES, 2007).

Existem diversas técnicas e ferramentas para a aplicação de testes de software nos mais variados estágios do ciclo de desenvolvimento de uma aplicação. Porém, em aplicações móveis, essa atividade ainda é deficiente. Segundo Amalfitano (2011) e Souza e Gasparotto (2013), isso ocorre principalmente, devido ao fato dos processos de desenvolvimento atuais serem muito rápidos. Isso tem impacto direto na atividade de testes, que muitas vezes é

negligenciada ou realizada de forma superficial pela própria equipe de desenvolvimento, que geralmente não possui a qualificação adequada para a realização dos testes de software, comprometendo assim, a qualidade da aplicação. Além disso, o desenvolvimento móvel é caracterizado por um rigoroso *time-to-market*, ciclos de vida mais curtos e um acelerado processo de evolução da tecnologia, fazendo com que os desenvolvedores concentrem esforços nos novos desafios tecnológicos e nos requisitos dos usuários finais, que são os mais importantes para o negócio. Enquanto isso as atividades de verificação tornam-se mais desafiadoras e demandam mais esforço da equipe de desenvolvimento (MUCCINI, DI FRANCESCO e ESPOSITO, 2012).

Existem diferentes técnicas para verificação e validação de software. Essas técnicas podem ser classificadas em dois grandes grupos: testes estáticos e testes dinâmicos. Testes estáticos são aqueles onde a execução do código não é necessária. Esse tipo de teste inclui inspeções e análise dos artefatos de software. Já os testes dinâmicos são aqueles onde o teste é realizado através da execução da aplicação com entradas reais (AMMANN e OFFUTT, 2008). De maneira simplificada, pode-se dizer que a validação é o processo utilizado para determinar se o software atende aos requisitos do usuário, enquanto que, a verificação é o processo empregado para determinar se o software está sendo desenvolvido corretamente.

A automação de teste aborda alguns importantes desafios da verificação no domínio móvel. As plataformas de desenvolvimento móvel fornecem grande suporte à automação dos testes unitários e, além disso, existem diversas ferramentas de terceiros que também suportam testes de sistema (ZEIN, SALLEH e GRUNDY, 2016). Cada ferramenta lida com diferentes aspectos do teste e/ou do aplicativo e/ou da plataforma de execução. Assim, algumas ferramentas interagem diretamente com o código fonte da aplicação (ex: Robotium), outras fornecem funcionalidades de captura/replay (ex: MonkeyTalk), algumas são exclusivas de uma única plataforma (ex: IOS Driver para Apple e UIAutomator para Android), enquanto outras suportam múltiplas plataformas (ex: Calabash), e assim por diante.

Apesar do extenso suporte fornecido pelas ferramentas de automação de testes de sistema, os desenvolvedores de dispositivos móveis ainda enfrentam um desafio importante, que é a definição de testes eficazes e eficientes, isto é, um conjunto de casos de testes que sejam rápidos, manuteníveis e capazes de detectar as maiores (e mais críticas) falhas. Algumas das características do desenvolvimento móvel tornam essa tarefa difícil. Primeiro, os curtos ciclos de desenvolvimento focam na implementação e geralmente a documentação é esparsa, fazendo com que a lista completa das funcionalidades necessárias não esteja bem documentada e/ou acessível ao testador do sistema. Por outro lado, caminhos inválidos podem

estar presentes na implementação do aplicativo e representar caminhos indesejados ou errôneos. Além disso, as especificidades das diferentes plataformas de execução podem afetar o comportamento detalhado de alguma funcionalidade, fazendo com que testes específicos sejam necessários.

Em um ambiente de desenvolvimento comum, um analista de negócios define as funcionalidades necessárias em um documento de especificação e a equipe de testes implementa os testes automatizados baseados nesse documento. No entanto, os casos de testes derivados de documentos de especificação tendem a se concentrar no comportamento esperado da aplicação, enquanto testes eficazes devem exercitar a aplicação além das funcionalidades conhecidas. Sendo essa uma tarefa manual, presente em um ambiente de curtíssimo *time-to-market*, pode ocorrer de algum teste importante ser simplesmente esquecido. Assim, as ferramentas que automatizam, ou pelo menos suportam a tarefa de definir casos de teste, também são importantes neste domínio.

Testes funcionais automatizados, baseados no uso dos cenários do *Behavior Driven Development* (BDD) têm sido usados há algum tempo e estão se tornando uma importante tendência também na comunidade de desenvolvimento móvel (WYNNE e HELLESOY, 2012). Uma razão para isso é o fato das aplicações móveis serem consideradas sistemas baseados em eventos, onde são basicamente compostas por um conjunto distinto de interfaces que, ao receberem entradas dos usuários, alteram o estado da aplicação (MACHIRY, TAHILIANI e NAIK, 2013). Além disso, os cenários, que são escritos em linguagem natural, são mais fáceis de manter e, o mais importante, refletem a perspectiva do usuário final do sistema. Finalmente, muitas ferramentas de automação de teste interagem com ferramentas baseadas no uso do BDD, oferecendo suporte para o ciclo de vida completo do teste.

Mesmo que esta prática melhore a documentação das funcionalidades do sistema, lidando assim com um importante desafio enfrentado pelos testadores de aplicações móveis, continua sendo necessário um esforço considerável para a atividade de escrita dos cenários.

1.1 OBJETIVOS

Este trabalho propõe uma abordagem para auxiliar o testador móvel na definição de cenários do BDD para testes de sistema em aplicativos Android. A abordagem proposta faz uso dos arquivos de cenários do BDD, em conjunto com os arquivos de leiaute da aplicação, para fornecer uma visão geral do sistema ao testador (visão hoje não disponível), para que ele possa avaliar a integridade dos casos de teste disponíveis em relação às funcionalidades

implementadas. Além disso, são gerados cenários no formato do BDD, auxiliando o testador na implementação de novos testes. A abordagem proposta apresenta três vantagens principais: i) fornece uma noção visual e de fácil compreensão dos cenários do BDD; ii) mostra a cobertura dos cenários em relação aos campos acionáveis da interface de usuário (UI) e; iii) gera um conjunto de possíveis cenários, auxiliando o testador na criação de novos testes.

A abordagem proposta é suportada pela implementação de uma ferramenta chamada de *Android Behavior Testing Tool* (ABTT), desenvolvida em linguagem Java e executada no mesmo ambiente de desenvolvimento do aplicativo Android.

A ferramenta ABTT foi validada em quatro aplicativos Android utilizados como estudos de caso. Dentre o conjunto de aplicativos escolhidos, três eram nativos do Android e foram escolhidos por possuírem características diferentes, onde um foi desenvolvido segundo a metodologia do BDD (possuía cenários), outro pela possibilidade de contato com um dos seus desenvolvedores, o que facilitou a interpretação dos resultados obtidos e o último por apresentar um nível de complexidade maior que os anteriores. Os aplicativos nativos foram escolhidos a fim de validar a abordagem proposta neste trabalho. Além disso, um aplicativo híbrido (que não é alvo da nossa abordagem) foi utilizado como artefato de teste para a identificação de possíveis limitações na ferramenta.

Os resultados experimentais obtidos durante a validação da ABTT mostram que a abordagem proposta é de grande valia para o testador, uma vez que em todos os aplicativos nativos foram detectados bugs que passaram despercebidos pelos processos de desenvolvimento das aplicações.

1.2 ORGANIZAÇÃO DO TEXTO

O texto desta dissertação encontra-se organizado como indicado nos parágrafos a seguir.

No Capítulo 2 é apresentado o embasamento teórico necessário para o entendimento deste trabalho, onde serão abordados os seguintes tópicos: desenvolvimento móvel, desenvolvimento Android, testes de software em aplicações desktop, web e mobile e desenvolvimento baseado em comportamento. Neste capítulo serão discutidos ainda outros trabalhos que visam aprimorar a tarefa de definição de testes em aplicações móveis.

No Capítulo 3 é apresentada a abordagem proposta neste trabalho para incrementar o conjunto de testes de sistema a partir de informações de leiaute de uma aplicação Android.

O Capítulo 4 apresenta a ferramenta que suporta a abordagem proposta, detalhando sua arquitetura, funcionamento e um exemplo de utilização.

Já o Capítulo 5 apresenta os estudos de caso e os resultados experimentais da utilização da ferramenta.

Finalmente, no Capítulo 6 são apresentadas as conclusões juntamente com os trabalhos futuros que podem ser realizados para dar continuidade a este trabalho.

2 REFERENCIAL TEÓRICO

2.1 DESENVOLVIMENTO MÓVEL

Aplicativos móveis podem ser resumidamente definidos como aplicações sendo executadas em dispositivos portáteis que recebem informações contextuais de entrada (MUCCINI, DI FRANCESCO e ESPOSITO, 2012). A complexidade dessas aplicações vem aumentando com o passar dos anos, graças à constante evolução do mercado de hardware, que permite o desenvolvimento de aplicações mais complexas. Porém, desenvolver uma aplicação móvel não é uma tarefa trivial. Problemas comuns à engenharia de software tradicional também fazem partes dos problemas enfrentados pela engenharia de software em ambiente móvel (WASSERMAN, 2010). Além disso, a computação móvel conta ainda com problemas específicos, como: interação com outras aplicações e com um variado número de sensores, número limitado de recursos, muitos padrões, protocolos e tecnologias de rede, e um grande conjunto de diferentes modelos de dispositivos (SANTOS e CORREIA, 2015). Aplicações móveis devem ainda lidar com entradas dos usuários assim como mudanças constantes de contexto.

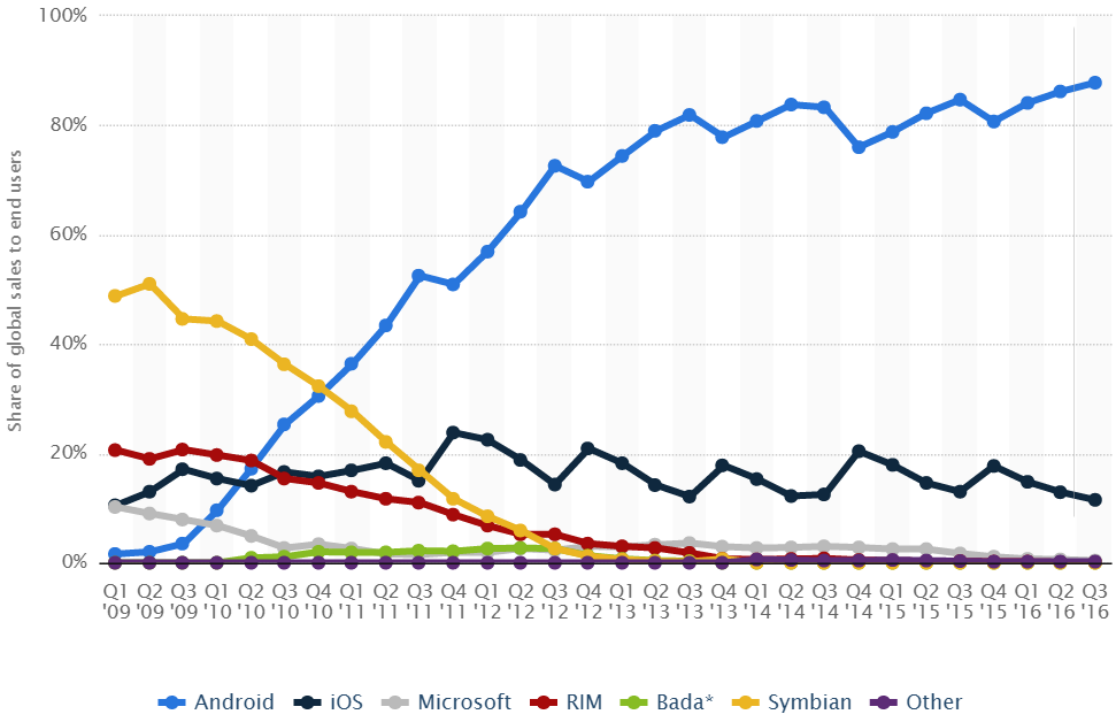
Segundo uma pesquisa realizada por Joorabchi (2013), as principais dificuldades encontradas no desenvolvimento dessas aplicações são as fragmentações, geradas tanto pelos sistemas operacionais como pelos modelos de dispositivos existentes. Além disso, as atuais práticas de desenvolvimento utilizadas, por serem muito rápidas, tornam um desafio a tarefa de desenvolvimento de uma aplicação móvel.

Atualmente, segundo o site Statista (2016), as principais plataformas móveis existentes no mercado são o Android, o IOS (Iphone OS) e o Windows Phone, gerenciados pela Google, Apple e Microsoft, respectivamente. A Figura 2.1 mostra a divisão do mercado de plataformas móveis nos últimos oito anos. Nota-se o aumento expressivo do domínio da plataforma Android nos cinco últimos anos. A Tabela 2.1 mostra as porcentagens de venda de cada sistema operacional no período final de 2015 até a o terceiro trimestre de 2016, deixando claro o constante aumento/domínio do sistema operacional Android sobre as outras plataformas.

Cada umas dessas plataformas utiliza uma linguagem de programação diferente e um conjunto distinto de ferramentas e kits de desenvolvimento. Isso significa que uma mesma aplicação, para poder ser executada nos três sistemas operacionais, deve passar pelo processo de desenvolvimento três vezes, um para cada plataforma. Além disso, os sistemas

operacionais (SO) móveis contam com um grande número de atualizações em um período relativamente curto de tempo, o que muitas vezes impossibilita que uma aplicação desenvolvida para uma versão mais antiga de um SO móvel seja executada em uma versão nova ou mais recente (ZEIN, SALLEH e GRUNDY, 2016). Esse tipo de fragmentação prejudica o reuso e dificulta a integração entre as plataformas. A Figura 2.2, mostra uma linha do tempo com as diferentes versões lançadas do sistema operacional Android.

Figura 2.1 – Gráfico do mercado de SO móvel.



Fonte: (Statista, 2016).

Tabela 2.1 – Porcentagem de vendas de SO móvel.

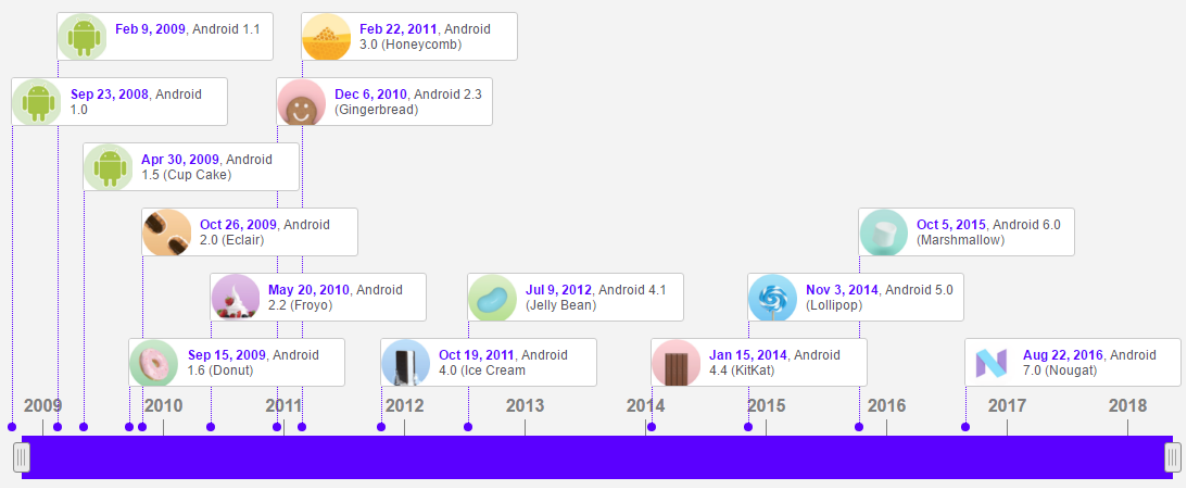
Período	Android	IOS	Windows Phone	Outros
2015Q4	79.6%	18.7%	1.2%	0.5%
2016Q1	83.5%	15.4%	0.8%	0.4%
2016Q2	87.6%	11.7%	0.4%	0.3%
2016Q3	87.8%	11.5%	0.4%	0.3%

Fonte: Adaptado de (IDC - Analyze the Future, 2016).

Para os dispositivos móveis e suas aplicações, o processo de engenharia de software não deve apenas levar em consideração as propriedades do hardware do dispositivo, mas

também deve abordar os problemas de gerenciamento de projetos e os aspectos exclusivos do desenvolvimento de aplicações móveis mencionados anteriormente. Desenvolver para esses dispositivos tem sido um desafio, já que é preciso balancear todas essas características (WASSERMAN, 2010).

Figura 2.2 – Linha do tempo de versões do sistema operacional Android.



Fonte: Elaborado pelo autor.

2.2 DESENVOLVIMENTO ANDROID

A escolha do Android como plataforma alvo para o estudo realizado neste trabalho, se justifica pelo fato de ser o sistema operacional móvel líder do mercado, o que permite que a pesquisa aqui apresentada possa ser utilizada em um maior número de aplicativos.

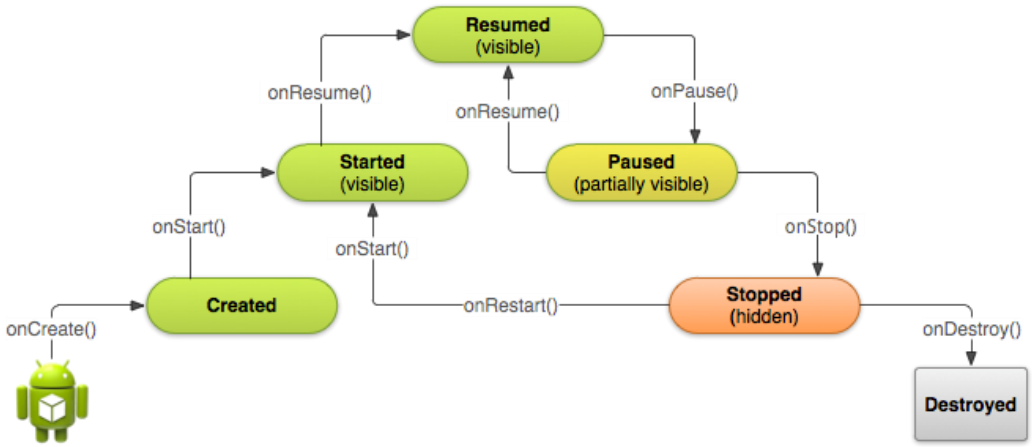
A arquitetura do sistema Android fornece quatro componentes básicos que permitem que uma aplicação interaja com o sistema: *activities*, *services*, *content providers* e *broadcast receivers*. A *activity* é uma das mais importantes classes de um aplicativo Android e serve como ponto de entrada e interação do usuário com o aplicativo. *Services* não possuem uma interface visual e são utilizados para executar processamentos em segundo plano. Os *content providers*, por sua vez, servem para disponibilizar dados específicos de uma aplicação para outras aplicações. Por fim, os *broadcast receivers* são componentes que ficam inativos e respondem a eventos. Para um aplicativo executar, pelo menos um desses componentes deve ser implementado (WINK, 2016). Para este trabalho, serão necessários os conceitos de *activities* e implementação de interfaces, que serão abordados a seguir.

O desenvolvimento de uma aplicação Android é dado basicamente através da utilização de *activities*, que interagem entre si. Uma *activity* geralmente representa uma tela

da aplicação, com a qual o usuário pode interagir e efetuar ações como fazer uma ligação, tirar uma foto, exibir um mapa, entre outros (Android Developers, 2016).

Diferentemente de outros paradigmas de programação em que os aplicativos são iniciados com um método *main*(), o sistema Android inicia o código em uma instância de *activity*, chamando métodos específicos de retorno de chamada, que correspondem a determinados estágios do ciclo de vida da *activity* (Android Developers, 2016). O ciclo de vida de uma *activity* é apresentado na Figura 2.3.

Figura 2.3 – Ciclo de vida de uma *activity*.



Fonte: (Android Developers, 2016).

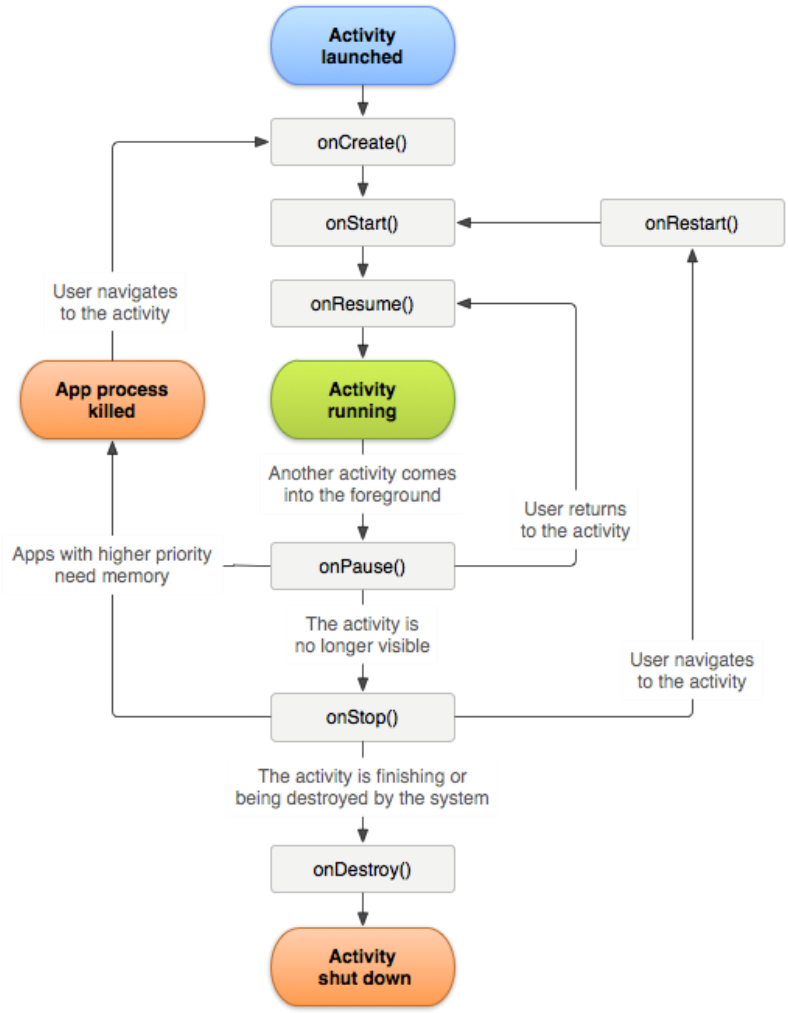
Normalmente, uma das *activities* da aplicação é especificada como a *activity* "principal", que é apresentada ao usuário ao iniciar uma aplicação pela primeira vez. Assim que a aplicação é iniciada, a *activity* principal é criada através da chamada do método *onCreate*(), onde são carregadas as configurações iniciais e o leiaute da *activity*, que é dado por um arquivo com extensão .XML (Android Developers, 2016). Uma *activity* é composta basicamente por três estados: em execução, pausada e parada.

Após a criação da *activity*, os métodos *onStart*() e *onResume*() são chamados e a *activity* passa para o estado “em execução”, tornando-se então visível para o usuário. Esse momento pode ser interpretado como o momento em que a *activity* está realmente sendo executada. Caso uma *activity* seja interrompida por algum motivo fazendo com que ela perca o foco, o método *onPause*() é chamado e a *activity* passa para o estado “pausada”, ficando parcialmente visível para o usuário. Uma *activity* em pausa está completamente viva, permanece conectada ao gerenciador de janelas e seu objeto de atividade é mantido na memória, assim como todas as informações de estado. Porém ela pode ser destruída pelo

sistema em situações onde a disponibilidade de memória do dispositivo encontra-se extremamente baixa (Android Developers, 2016).

Quando uma *activity* perde o foco completamente e é sobreposta por outra *activity* (ex: no caso de uma aplicação estar sendo executada e o dispositivo móvel receber uma chamada), o método *onStop*() é chamado e a *activity* passa para o estado "parada". A *activity* parada também continua viva e com todas as informações de estado salvas, porém ela não se encontra mais conectada ao gerenciador de janelas e nem visível para o usuário. Assim como acontece com a *activity* pausada, uma *activity* parada também pode ser destruída pelo sistema em caso de necessidade de memória (Android Developers, 2016). Para destruir uma *activity*, utiliza-se o método *onDestroy*(). Todo o processo descrito acima é ilustrado pela Figura 2.4.

Figura 2.4 – Ciclo de vida de uma *activity*.



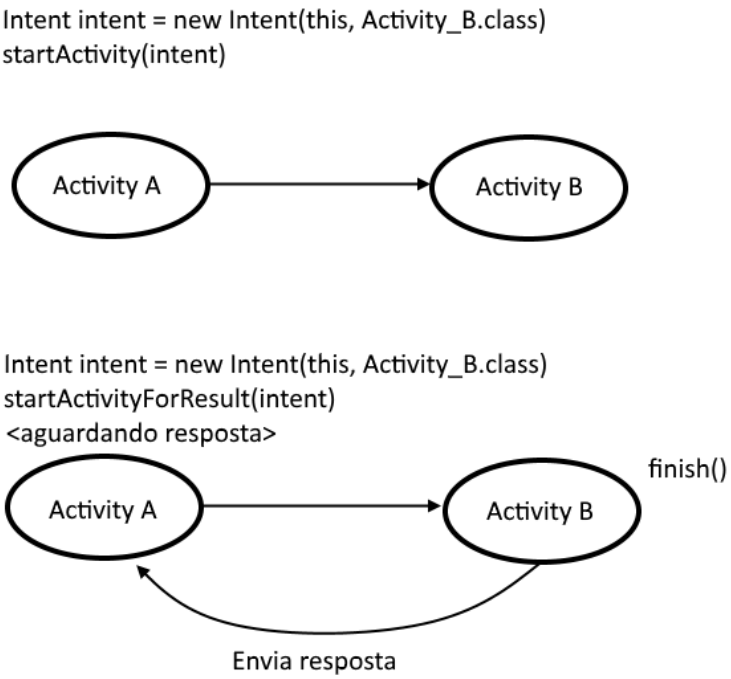
Fonte: (Android Developers, 2016).

Uma aplicação Android normalmente é formada por um conjunto de várias *activities* que trocam informações e se comunicam entre si. Para comunicação entre *activities* o método

utilizado é o *startActivity*(*intent*). Esse método passa como parâmetro um objeto do tipo *intent*, que fornece vínculos em tempo de execução entre componentes separados (como duas *activities*). O *intent* representa uma "intenção de fazer algo" do aplicativo (Start Another Activity, 2017). No construtor de um *intent* são passados como parâmetro a *activity* em execução e a *activity* que se deseja executar.

Porém, iniciar outra *activity* não precisa ser algo unidirecional. Pode-se também iniciar outra *activity* e receber um resultado de volta. Para receber um resultado, utiliza-se como forma de iniciar uma nova *activity* o método *startActivityForResult*(*intent*) (em vez de *startActivity*(*intent*)). Dessa maneira, após a execução da nova *activity*, um resultado é enviado de volta para *activity* anterior, e sua execução continua normalmente. A Figura 2.5 exemplifica a diferença de utilização dos dois métodos.

Figura 2.5 – Funcionamento dos diferentes métodos de comunicação entre *activities*.



Fonte: Elaborado pelo autor.

No primeiro caso, o método *startActivity*(*intent*) é utilizado pela *activity* A para dar início à execução da *activity* B que, após a chamada do método, inicia sua execução. No segundo caso, o método utilizado para dar início a execução da *activity B* é o *startActivityForResult*(*intent*). Após a chamada do método, a *activity* A, que fez a chamada, fica aguardando por uma resposta da *activity* B, que ao ser finalizada com o método *finish*(), envia a resposta.

2.2.1 Leiaute

Em uma aplicação, a estrutura visual da interface de usuário é dada pelo leiaute que, no Android, pode ser definido de duas maneiras: por meio de um arquivo XML ou em tempo de execução. As boas práticas de programação, assim como a documentação oficial do Android, orientam a criação dos leiautes por meio dos arquivos XML. A vantagem é separar melhor a apresentação do aplicativo do código que controla seu comportamento. As descrições de interface são externas ao código do aplicativo, ou seja, é possível modificá-las ou adaptá-las sem modificar e recompilar o código-fonte. Além disso, é possível criar, para uma mesma *activity*, um conjunto semelhante de leiautes XML, porém com diferentes orientações, tamanhos e formatos de tela. É possível também criar leiautes XML para representar telas em diferentes idiomas (Android Developers, 2016). Um arquivo de leiaute XML é composto por *tags*, que identificam os diferentes componentes da tela e suas características. Um exemplo de arquivo XML de leiaute e sua representação como interface de usuário são apresentados nas Figuras 2.6 (a) e 2.6 (b), respectivamente.

Figura 2.6 – Exemplo de arquivo de leiaute XML e sua representação.

(a) Interface de usuário definida em XML.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

Fonte: (Android Developers, 2016).

(b) Tela correspondente ao arquivo XML.

Fonte: Elaborada pelo autor.

2.3 TESTE DESKTOP, WEB E MOBILE

O desenvolvimento de software teve origem com as tradicionais aplicações desktop, que se caracterizam pela execução direta em um computador, sem a necessidade de conexão com a internet. Com a evolução da tecnologia e o surgimento da internet, as aplicações web ganharam espaço e, da mesma maneira, hoje ganham destaque os aplicativos móveis.

Atualmente, as aplicações podem ser divididas em três categorias, de acordo com a plataforma em que são executadas, em: desktop, web e mobile. Porém, essas plataformas diferem entre si e suas características têm impacto direto no teste, devendo ser levadas em consideração.

O objetivo do teste, independentemente da plataforma utilizada, é o mesmo: encontrar o maior número de falhas possíveis no software. Porém, devido às diferenças entre as plataformas, testes específicos são realizados de acordo com suas necessidades e características.

Independentemente da plataforma, as atividades de teste de uma aplicação podem ser divididas em níveis, de acordo com o escopo de aplicação do teste. Assim, podem haver testes unitários, testes de integração, testes de sistema, testes de aceitação, entre outros.

Os testes unitários consistem em testar individualmente os subprogramas, sub-rotinas, classes ou módulos de um software. De maneira geral, ao invés de testar incialmente o

programa como um todo, os testes concentram esforços nos blocos de construção menores (MYERS, BADGETT e SANDLER, 2011).

Os testes de integração, que sucedem os testes unitários, têm por objetivo combinar e testar em conjunto os módulos testados individualmente na fase anterior. Na fase de testes de integração os módulos previamente testados pelos testes de unidade são agrupados de acordo com o estipulado no plano de teste. Esse agrupamento resulta num sistema integrado e preparado para a próxima fase de teste, o teste de sistema.

A fase de testes de sistema é utilizada para garantir que o sistema funcione como um todo. Nessa fase busca-se comparar se o sistema ou programa sob teste atende aos objetivos originais para o qual foi desenvolvido. Para isso são utilizados testes do tipo caixa preta, pois não é necessário conhecimento sobre a estrutura (lógica) interna do sistema, somente dos aspectos gerais (MYERS, BADGETT e SANDLER, 2011).

Neste trabalho, busca-se aprimorar a atividade de definição de testes de sistema em aplicações Android. É interessante notar, porém, que nesta categoria de aplicações, alguns testes de sistema podem corresponder a testes unitários. Isso se deve à estrutura baseada em *activities*. Uma *activity* pode ser considerada como uma unidade testável daquele sistema, embora seja muitas vezes responsável pela execução completa de uma funcionalidade, incluindo a interface com o usuário. Além dos níveis de teste, diferentes tipos e objetivos de teste devem ser considerados nas diferentes categorias de aplicações.

Aplicações desktop por exemplo, geralmente concentram esforços em testes de sistema e segurança. Isso se justifica pelo fato de ser a plataforma com maior poder de processamento, onde são possíveis diversas combinações de hardware, impactando diretamente no funcionamento do software. Além disso, as informações do usuário são armazenadas localmente e devem estar protegidas em casos de ataques.

As aplicações web, por sua vez, são essencialmente aplicações cliente-servidor, onde o cliente é um navegador e o servidor é uma aplicação ou um servidor web. Por isso, o teste dessas aplicações concentra esforços em conectividade, segurança e compatibilidade. Testes de conectividade são realizados para assegurar que, independentemente da velocidade de conexão do usuário, o conteúdo da aplicação seja carregado de maneira correta e em tempo satisfatório. Os testes de segurança, assim, como nas aplicações desktop, procuram proteger os dados dos usuários, que agora não estão mais armazenados localmente, tornando-os ainda mais suscetíveis a ataques de terceiros. Os testes de compatibilidade, por sua vez, são efetuados afim de garantir que a aplicação seja executada corretamente nos diferentes tipos de navegadores utilizado pelo usuário (MYERS, BADGETT e SANDLER, 2011).

O teste de software em ambiente móvel, por si só, é um desafio para a engenharia de software moderna. O variado número de dispositivos disponíveis no mercado, a fragmentação dos sistemas operacionais, os diferentes tipos de aplicativos e as diversas ferramentas disponíveis para testes, são só alguns dos fatores que fazem com que a maioria dos profissionais de teste considerem o teste em ambiente móvel mais complexo e desafiador do que aquele realizado em outras plataformas. Na verdade, são os dispositivos e o ambiente móvel, mais do que o aplicativo em si, que impõem tal desafio. Esses dois fatores adicionam muitas variáveis e complexidades que podem distorcer ou mascarar problemas em um aplicativo móvel, tornando difícil a tarefa de projetar um plano de teste robusto para essas aplicações (MYERS, BADGETT e SANDLER, 2011). Como as aplicações móveis estão cada vez mais presentes em áreas críticas, elas não estão somente mais complexas, mas também mais difíceis de testar. As aplicações móveis exigem uma abordagem diferente em relação a qualidade e a confiabilidade e necessitam de uma abordagem de teste eficaz, capaz de criar um software de qualidade (ZEIN, SALLEH e GRUNDY, 2016).

No ambiente móvel, também são utilizadas as abordagens de teste mencionadas nas plataformas anteriores. Testes de desempenho são utilizados devido as limitações de hardware dos dispositivos, testes de conectividade para testar os muitos padrões de rede disponíveis (2G, 3G, 4G, WiFi, etc.) e testes de segurança para assegurar que as informações pessoais armazenadas no dispositivo estejam seguras. Além disso, o teste funcional muitas vezes é realizado junto com os outros testes, o que não acontece em outras plataformas, onde a divisão entre teste unitário e de sistema é muito clara.

Mesmo entre as próprias aplicações móveis existem diferenças que impactam no teste de software. Os aplicativos móveis podem ser classificados em três categorias: aplicativos nativos, web e híbridos. Aplicativos nativos são aqueles que necessitam instalação e ficam armazenados no próprio dispositivo do usuário. São disponibilizados para *download* diretamente nas lojas virtuais, de acordo com o sistema operacional utilizado. Além disso, os aplicativos nativos são desenvolvidos por meio das linguagens de programação e kits de desenvolvimento específicos de cada sistema operacional móvel e caracterizam-se por terem acesso a todas as funcionalidades do sistema e recursos do dispositivo, como câmera, GPS, acelerômetro, ou qualquer outro que o dispositivo venha a ter.

Os aplicativos web, por sua vez, são considerados páginas web que podem ser acessadas através de um navegador. São geralmente desenvolvidos em HTML 5 e *Cascading Style Sheets* 3 (CSS), o que permite dar uma aparência de uso semelhante a de um aplicativo nativo. Apesar da semelhança, os aplicativos web não têm acesso aos recursos do sistema nem

podem utilizar opções de notificações para os usuários. Além disso, eles dependem de conexão com a internet para o bom funcionamento (OLIVEIRA, 2016).

Por último, existem os aplicativos híbridos, que são uma espécie de fusão entre aplicativos nativos e aplicativos web. Os aplicativos híbridos, também conhecidos como *cross-platform*, podem ser definidos como aqueles que possuem uma parte suficientemente grande (ou todo ele) feita em uma tecnologia que não é a específica da plataforma alvo, ou que não seja a mais comumente utilizada (LIMA, 2016). Esses aplicativos são desenvolvidos a partir da linguagem e tecnologia dos aplicativos web e utilizam frameworks de conversão, que convertem a linguagem de forma que possam ser instalados nos dispositivos dos usuários. Assim como os aplicativos nativos, são disponibilizados nas lojas virtuais e também possuem acesso aos recursos do sistema, porém não de forma tão eficiente como em um aplicativo originalmente nativo, por esse motivo problemas de compatibilidade podem ocorrer. Muitos desses aplicativos utilizam um recurso chamado "*web view*" que simplesmente renderiza e insere diretamente uma página da web no aplicativo.

Para abordar todos os aspectos citados acima, assim como os citados na seção 2.1, testes adicionais podem ser realizados, além do teste funcional que é mandatório. A Tabela 2.2 mostra os diferentes tipos de testes adicionais que podem ser realizados em aplicações móveis:

Tabela 2.2 – Tipos de teste.

Tipo de Teste	Objetivo
Teste de Usabilidade	Garantir que o aplicativo seja fácil de usar e forneça uma experiência de usuário satisfatória para os clientes.
Testes de Compatibilidade	Testar o aplicativo em diferentes dispositivos móveis, navegadores, tamanhos de tela e versões de SO, de acordo com os requisitos.
Teste de Interface	Testar as opções de menu, botões, marcadores, histórico, configurações e fluxo de navegação do aplicativo.
Teste de Serviço	Testar os serviços da aplicação on-line e off-line.
Teste de Recursos	Testar o uso de memória, exclusão automática de arquivos temporários, problemas de desenvolvimento de banco de dados local, etc.
Teste de Desempenho	Testar o desempenho do aplicativo, alterando a conexão de 2G para 3G, 3G para WIFI, compartilhando documentos, o

	consumo de bateria, etc.
Teste Operacional	Testar *backups* e planos de recuperação (caso a bateria termine, ou ocorra perda de dados ao atualizar o aplicativo).
Testes de Instalação	Testar o aplicativo através da instalação/desinstalação nos diferentes modelos de dispositivos.
Testes de Segurança	Testar o aplicativo para validar se o sistema protege os dados ou não.

Fonte: (MYERS, BADGETT e SANDLER, 2011)

Neste trabalho, é abordado parcialmente o teste funcional além dos testes de interface com base na especificação fornecida pelos cenários do BDD. Esses cenários refletem as funcionalidades mais importantes do sistema, porém nem sempre de forma completa. Pode ocorrer de algum elemento da interface não ser exercitado pelos cenários, fazendo com que alguma falha no aplicativo passe despercebida. Sendo assim, utiliza-se os arquivos de leiaute da aplicação em conjunto com o modelo de cenário do BDD para gerar e documentar testes de sistema a fim de aumentar a cobertura dos testes e exercitar todos os elementos da interface da aplicação.

2.4 DESENVOLVIMENTO BASEADO EM COMPORTAMENTO

O desenvolvimento baseado em comportamento, do inglês *Behavior Driven Development*, criado em 2006 por Dan North, é uma abordagem ágil de desenvolvimento de software onde os requisitos do sistema são descritos utilizando linguagem natural, porém estruturada (linguagem ubíqua), fornecendo assim uma documentação mais viva e acessível dos requisitos da aplicação e especificações menos ambíguas. O BDD, além de melhorar a comunicação entre a equipe de desenvolvimento, as partes interessadas e os analistas do negócio, gera especificações executáveis que podem ser utilizadas como testes de aceitação automatizados (NORTH, 2006) (SMART, 2014).

As especificações do BDD são compostas por um conjunto de exemplos (chamados cenários) que ilustram o comportamento do sistema do ponto de vista do usuário. Por esta razão, os cenários são descritos em um idioma livre de jargões técnicos ou detalhes de implementação (linguagem ubíqua). Uma vez confirmados pelo cliente, os cenários podem ser detalhados e implementados como testes automatizados, transformando-se assim em testes de aceitação automatizados. Por linguagem ubíqua entende-se uma linguagem comum entre

desenvolvedores e usuários, baseada no modelo de domínio da aplicação, livre de termos técnicos e comum a diferentes setores presentes no processo de desenvolvimento de um software.

Através do uso dessa linguagem, o BDD facilita o entendimento e estimula a aproximação das diferentes equipes que participam do processo de desenvolvimento do software, potencializando o diálogo e diminuindo as diferenças e os erros gerados a partir das falhas de comunicação. Uma das principais vantagens da utilização do BDD é a aproximação entre as diversas equipes envolvidas no processo de criação do software, principalmente entre as equipes de testes e desenvolvimento (MATTÉ, 2011).

Clientes e analistas de negócios trabalham com este tipo de artefatos, listando cenários interessantes que devem ser posteriormente implementados pelos desenvolvedores. A equipe de desenvolvimento utiliza esses cenários para projetar e implementar o sistema enquanto que, as equipes de teste, usam esses cenários para implementar testes automatizados.

O BDD pode ser visto ainda como uma evolução do *Test Driven Development* (TDD), uma vez que os testes também são escritos antes do código funcional da aplicação e guiam o desenvolvimento do software, porém o foco no BDD é outro. O BDD não leva em consideração detalhes de implementação, concentrando esforços somente no comportamento da aplicação. Uma das vantagens dessa abordagem é que os comportamentos se modificam menos que os testes. Sendo assim, tipicamente tais modificações refletem a necessidade de inclusão de funcionalidades adicionais no sistema (LAPOLLI, CRUZ, *et al.*, 2010).

2.4.1 Histórias e Cenários

A aplicação do BDD é dada a partir da criação de cenários, que definem o comportamento esperado da aplicação diante da interação do usuário com determinada funcionalidade do sistema. Para a criação desses cenários, utiliza-se a linguagem ubíqua mencionada anteriormente, a fim de expor de maneira clara e simplificada, para qualquer participante do projeto, o que está sendo desenvolvido e qual o resultado esperado.

Antes da escrita dos cenários é recomendado que seja criada uma breve narrativa, chamada de *user story* (história do usuário), que descreve a utilização de um recurso no âmbito da aplicação, juntamente com seu benefício para o negócio, do ponto de vista do cliente ou do usuário final do sistema (COWAN, 2016).

O BDD recomenda a utilização de um modelo para a criação das *user stories*, porém esse modelo não precisa ser rigorosamente seguido. Ele é sugerido devido à comprovada

eficiência nos mais diferentes tipos de projetos (NORTH, 2006). O modelo de narrativa sugerido é o modelo tradicional das *user stories* que é apresentado na Figura 2.7.

Figura 2.7 – Modelo de *user story*.

```
1 Título [uma linha descrevendo a história]
2 Narrativa:
3 Como [papel]
4 Eu quero [funcionalidade]
5 Para que [benefício]
```

Fonte: Adaptado de (NORTH, 2007).

O modelo é composto pelo título e pela narrativa, que é formada pelas sentenças: *Como*, *Eu quero* e *Para que*. O título deve descrever, de maneira clara e objetiva, a atividade que será executada, tornando assim, mais fácil para a equipe de testes ou qualquer participante do projeto, o entendimento do objetivo a ser alcançado pelo cenário proposto (NORTH, 2007). Através dos elementos da narrativa e sua estrutura papel-funcionalidade-benefício, é possível identificar quem está propondo determinada funcionalidade e qual a sua importância para o sistema. Dessa maneira, fica explícito para o usuário ou para os interessados no software, o valor das funcionalidades disponibilizadas pela aplicação (COWAN, 2016). Um exemplo de utilização do modelo apresentado acima é mostrado na Figura 2.8.

Figura 2.8 – Exemplo de uma *user story*.

```
1 História:  Saque em caixa eletrônico
2 Narrativa:
3 Como cliente de um banco.
4 Eu quero sacar dinheiro em um caixa eletrônico.
5 Para que eu possa retirar dinheiro enquanto o banco está fechado.
```

Fonte: Adaptado de (NORTH, 2007).

Apesar da notória semelhança com um caso de uso, uma história pode ser vista como um diferente artefato no processo de desenvolvimento de software, podendo ser utilizada para descrever e identificar um conjunto de requisitos. A próxima etapa para a aplicação do BDD é a criação dos cenários, que capturam os critérios de aceitação do software. Um critério de aceitação é representado pelo comportamento do sistema, ou seja, se o sistema apresenta todos os comportamentos esperados, inclusive diante das situações inusitadas, ele atende a todos os critérios de aceitação, caso contrário, não atende (LAPOLLI, CRUZ, *et al.*, 2010).

Para a criação dos cenários, assim como nas *user stories*, é utilizada a linguagem ubíqua aplicada a um modelo que faz uso de sentenças pré-determinadas. A utilização dessas sentenças facilita a criação e a escrita dos cenários (NORTH, 2006). O modelo utilizado para a escrita dos cenários é apresentado na Figura 2.9.

Figura 2.9 – Modelo de cenário.

```
1  Cenário 1: Título
2  Dado que [contexto inicial]
3      E [mais contexto]
4  Quando [evento]
5      E [mais eventos]
6  Então [comportamento esperado]
7      E [mais comportamentos esperados]
```

Fonte: Adaptado de (NORTH, 2007).

O modelo é composto pelo título do cenário e pelas sentenças predefinidas: "*Dado que*" (*Given*), "*Quando*" (*When*) e "*Então*" (*Then*). A sentença "*Dado que*" é utilizada para representar o estado inicial do sistema, a sentença "*Quando*" para representar as ações do usuário sobre a aplicação, como cliques, preenchimento de formulários, movimentos de rolagem ou qualquer outra interação possível com o sistema, e a sentença "*Então*" para representar o estado final da aplicação, ou seja, o comportamento do software diante das ações realizadas pelo usuário. Cada uma dessas sentenças pode ser estendida através do uso do conector "E" (*And*). Dessa maneira é possível ampliar o contexto inicial do cenário assim como as interações do usuário e as repostas do sistema.

No ambiente de testes, a linguagem ubíqua é representada através da linguagem *Gherkin*, que é muito semelhante à linguagem natural, porém contém algumas palavras chaves. Essa linguagem é específica de domínio e é entendida pelas principais ferramentas de teste baseadas em BDD (HANSSON, 2017). Um exemplo da representação de cenário utilizando Gherkin é mostrado na Figura 2.10.

Figura 2.10 – Exemplo de um cenário de teste.

```
1  Cenário 1: Cliente possui saldo positivo
2  Dado que o saldo da conta corrente do cliente seja de R$100,00.
3      E o cliente insira o cartão de crédito no caixa eletrônico.
4      E o cartão de crédito do cliente seja válido.
5      E o caixa eletrônico contenha dinheiro suficiente.
6  Quando o cliente solicitar sacar R$20,00.
7  Então o caixa eletrônico deve entregar R$20,00
8      E o saldo da conta do cliente deve passar para R$80,00.
9      E o cartão de crédito do cliente deve ser devolvido.
```

Fonte: Adaptado de (NORTH, 2007).

Através dos cenários é possível representar situações em que o usuário interage com o software e um comportamento específico é esperado por parte da aplicação, como exemplificado na Figura 2.9, na situação fictícia onde um cliente de um banco tem R$100,00 de saldo e faz um saque de R$20,00 no caixa eletrônico. Supondo que o caixa eletrônico possua dinheiro suficiente e que o cartão do usuário seja válido, o saque é realizado com sucesso e o cliente deve ficar com um saldo de RS80,00. A partir do cenário, a funcionalidade sob teste é implementada atendendo ao proposto pelo cenário. Dessa maneira espera-se capturar os erros de implementação durante a fase de desenvolvimento aumentando assim a qualidade do software.

Segundo Santos (2015), métodos de desenvolvimento ágeis permitem à equipe de desenvolvimento responder rapidamente a mudanças e, quando comparado com o método tradicional de desenvolvimento, onde a fase de testes é executada ao final do desenvolvimento, diminuem os riscos associados ao projeto.

2.4.2 Desafios do BDD

O BDD, apesar de ser uma técnica simples de desenvolvimento, ainda conta com muitos desafios e dificuldades em face da sua utilização. O principal desafio enfrentado quando utiliza-se o BDD está relacionado com a escrita dos cenários. Um cenário deve ser simples e de fácil compreensão para qualquer indivíduo, de forma a facilitar a comunicação e estimular a aproximação entre as diferentes partes envolvidas no processo de desenvolvimento de um software. Além disso, um cenário deve capturar de maneira correta os critérios de aceitação do sistema. Porém escrever um cenário com qualidade, não é uma tarefa trivial, pois depende muito do conhecimento e da experiência adquirida pelo responsável pela escrita do mesmo.

Normalmente, no ambiente empresarial, os cenários são escritos de maneira manual pelos testadores ou pelos responsáveis pelo projeto. Porém esses indivíduos detêm um nível de conhecimento diferente sobre o funcionamento do software. A equipe de testes nem sempre possui o conhecimento do software como um todo, principalmente em grandes sistemas coorporativos. Já os responsáveis pelo projeto, possuem uma visão geral do negócio, mas muitas vezes não conhecem algum funcionamento específico da aplicação. Essa diferença de conhecimento sobre o software acaba criando uma dificuldade sobre o nível de abstração que deve ser utilizado na escrita dos cenários. Se um nível de abstração muito alto for utilizado, os critérios de aceitação podem não ser capturados de forma integral, fazendo

com que algum critério passe despercebido pelo teste. Por outro lado, se um nível de abstração muito baixo for utilizado e detalhes de implementação forem levados em consideração, o BDD pode acabar sendo descaracterizado. Dessa maneira será necessária a utilização de termos técnicos para descrição dos detalhes de implementação que, ao invés de facilitar o entendimento e a comunicação entre as partes, poderá criar barreiras de interpretação.

Outra dificuldade presente na utilização do BDD é o entendimento de quais termos utilizar para descrever um cenário. Por exemplo, uma caixa de texto pode ser descrita de diferentes maneiras, como: formulário, caixa de texto, campo, *text box*, entre outras. Isso ocorre pois não existe uma padronização de termos para a escrita dos cenários. Esse conjunto de diferentes palavras utilizadas para representar um mesmo componente pode acabar gerando confusão e dificultar o entendimento.

Outro desafio existente na escrita dos cenários é a redundância. Muitas vezes um critério de aceitação é capturado diversas vezes por diferentes cenários, o que acaba ocasionando um desperdício de tempo e esforço. Por outro lado, erros são comuns e, geralmente por esquecimento, nem todos os componentes do leiaute são testados, fazendo com que algum critério de aceitação passe despercebido. Além disso, como já dito anteriormente, os cenários são escritos de forma não automática, pois atualmente não existe ferramenta disponível no mercado que atenda a essa necessidade. Sendo assim a escrita dos cenários demanda tempo e pessoal.

2.4.3 Execução dos Cenários do BDD

Para automatizar a execução de uma determinada funcionalidade, os testadores devem primeiro implementar o comportamento implícito em cada etapa (passo) do cenário. Assim, por exemplo, o passo "*Dado que*" (linha 2 na Figura 2.11) indica o estado inicial da aplicação para a funcionalidade descrita.

Figura 2.11 – Exemplo de cenário do BDD

```
1 Cenário 2 : Caso em que a busca não é possível.
2 Dado que estou na página inicial.
3     E não estou logado.
4 Quando eu tento realizar uma busca.
5 Então eu recebo uma mensagem de erro.
6     E eu devo permanecer na página inicial.
```

Fonte: Elaborado pelo autor.

Para executar esse cenário automaticamente, esse estado inicial deve ser definido por uma etapa de teste correspondente, que pode, por exemplo, abrir o navegador e ir para a página inicial pré-definida na aplicação. As etapas de teste são implementadas em uma linguagem de programação regular, como Java, Ruby, etc. A Figura 2.12 mostra uma possível implementação da etapa "*Dado que*" (Figura 2.11) em Java.

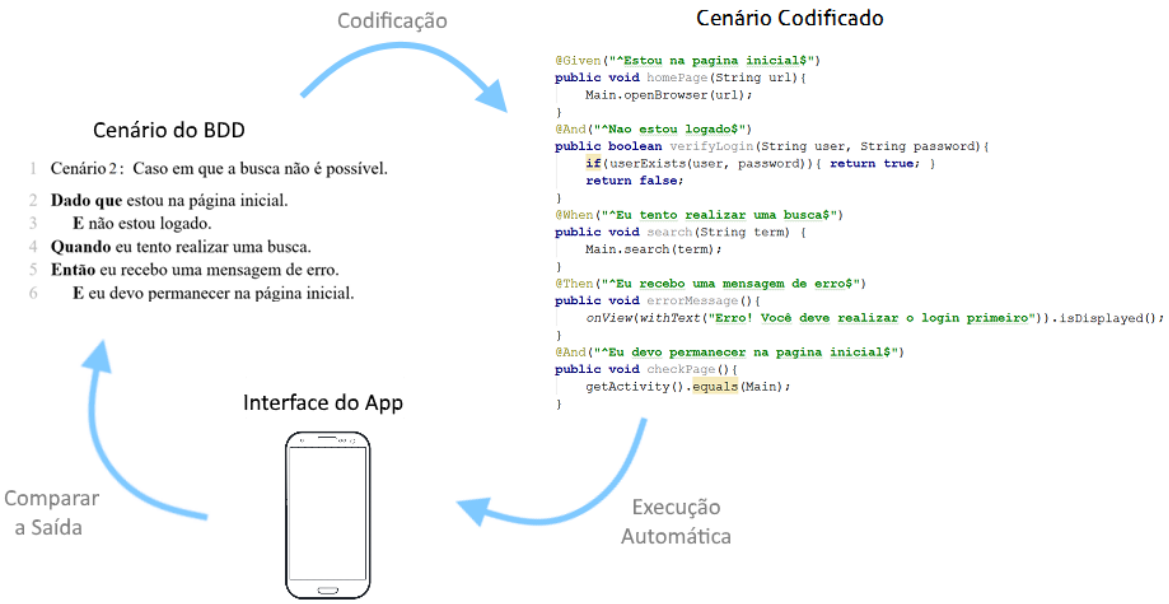
Figura 2.12 – Implementação em Java de um passo do cenário do BDD.

```
@Given("^Estou na pagina inicial$")
public void homePage(String url){
    Main.openBrowser(url);
}
```

Fonte: Elaborado pelo autor.

Além da implementação de todas as etapas de teste presentes nos cenários do BDD, a execução automática dos cenários assume uma interação com a interface do aplicativo, uma vez que representam a perspectiva do usuário. Sendo assim, é necessário algum mecanismo que simule essas interações (descritas nos cenários) e colete a resposta da aplicação, para ser comparada com o resultado esperado. A Figura 2.13 ilustra o processo de execução automática de cenários BDD.

Figura 2.13 – Processo de automação de um teste escrito em BDD.



Fonte: Elaborado pelo autor.

Hoje existem várias ferramentas disponíveis para suportar o uso do BDD em diferentes plataformas, incluindo a móvel. Essas ferramentas têm três objetivos principais: 1) analisar os cenários escritos em linguagem de alto nível e encontrar a implementação correspondente de cada etapa do cenário; 2) executar as etapas de teste implementadas e simular as interações necessárias do usuário; e 3) coletar informações do aplicativo após a aplicação de algum estímulo e comparar com a resultado esperado. Cucumber (Cucumber, 2017), JBehave (Jbehave, 2017), Appium (Appium.io, 2017) são alguns exemplos de ferramentas que analisam os cenários relacionando-os com as etapas de teste. Robotium (Robotium Tech, 2017), Espresso (Espresso, 2017), UIAutomator (UIAutomator, 2017), entre outros, fornecem APIs para simular interações e eventos de usuários em aplicativos móveis, bem como para coletar as respostas da aplicação.

O BDD não é uma técnica de teste e seu principal objetivo não é a automação do teste. Sua principal vantagem e objetivo é promover a comunicação e a aproximação entre todas as partes interessadas no desenvolvimento do software e fazê-los focar no valor de negócio de cada funcionalidade do sistema em desenvolvimento. Um primeiro resultado desse objetivo é a geração de especificações mais confiáveis. No entanto, a estrutura da especificação do BDD e o suporte das ferramentas ajudam a verificar constantemente e automaticamente que os exemplos listados na especificação (que representam as funcionalidades necessárias da aplicação) estão realmente funcionando no software entregue.

Vale ressaltar que um cenário do BDD pode ser escrito em diferentes níveis de abstração. Na sua origem e melhor utilização, os cenários do BDD utilizam conceitos e sentenças de muito alto nível, sem referência a detalhes tecnológicos ou de interface de usuário. É natural, portanto, que as especificações do BDD estejam incompletas em relação ao sistema implementado e sua interface. De fato, o sistema implementado combina exemplos de funcionalidades mais completas enquanto que os cenários de alto nível representam apenas os cenários de uso mais interessantes da aplicação que surgiram durante a fase de elicitação de requisitos. Por outro lado, não é raro observar equipes de projeto que utilizam apenas esta especificação incompleta para definir testes de sistemas. Também não é incomum ver equipes de projeto utilizando os cenários do BDD de nível mais baixo, ou seja, cenários baseados em interações com a interface de usuário, como ambos, seu documento de especificação principal é base para a criação dos testes de sistema. Cenários do BDD baseados em componentes de UI estão mais perto da implementação do sistema, mas não necessariamente geram testes eficazes. Nesse nível, o número de cenários cresce rapidamente e pode-se facilmente esquecer ou perder o rastreamento de possíveis interações.

Por isso, este trabalho propõe utilizar a estrutura bem definida dos cenários do BDD para ajudar o testador a definir testes de sistemas eficazes para aplicativos nativos do Android. O método proposto ajuda o testador de duas maneiras: primeiro, ajuda a avaliar quanto da interface do sistema (e das funcionalidades correspondentes) é exercitada pelos cenários disponíveis. Em seguida, ajuda a propor ações adicionais sobre a interface que auxiliam o testador a pensar em novos cenários que correspondem (ou utilizam) essas ações.

2.5 TRABALHOS RELACIONADOS

A automação de teste no domínio móvel tem recebido muita atenção recentemente e uma ampla gama de recursos e ferramentas está atualmente disponível. Muitas abordagens lidam com a questão da automação do teste em diferentes dispositivos e plataformas. Por exemplo, várias plataformas online permitem a e execução de um mesmo script de teste em vários dispositivos simultaneamente (KAASILA, FERREIRA, *et al.*, 2012); (Perfecto Mobile, 2017); (TestObject, 2017), entre outros. Outras abordagens lidam com questões de segurança e visam detectar vulnerabilidades na aplicação (CHAN, HUI e YIU, 2012). Muitas ferramentas comerciais e de código aberto se concentram na automação da execução do teste, simulando interações do usuário (como MonkeyTalk (Oracle, 2017)), ou fornecendo APIs para acessar a interface do aplicativo (como Robotium (Robotium Tech, 2017), Selendroid (DARY, 2017), etc).

Algumas abordagens, no entanto, foram propostas recentemente para automatizar e/ou apoiar o testador na tarefa de definir testes para aplicativos Android. Para automatizar essa tarefa precisa-se entender o fluxo de informação e controle da aplicação. Isso tem sido feito através da análise da interface do aplicativo.

Amalfitano (2011), propõe uma ferramenta chamada *Android Automatic Testing Tool* (A^2T^2), que extrai automaticamente uma visualização gráfica dos possíveis fluxos da aplicação, com base em simulações das interações do usuário (utilizando valores aleatórios) com a interface do aplicativo. Através das interações cria-se uma árvore, onde os nós representam as interfaces de usuário (telas do aplicativo) e as arestas representam os eventos que conectam as diferentes interfaces. O comportamento da ferramenta é baseado no modelo conceitual de uma *Graphical User Interface* (GUI) Android, onde são definidos os conceitos de interface (com propriedades), *Widgets* (elementos acionáveis da tela), eventos (parametrizados ou não) e *Event Handlers*. Uma vez que o modelo de árvore é criado, os casos de testes podem ser gerados automaticamente seguindo todos os caminhos da raiz até as

folhas. Os testes gerados por essa abordagem visam detectar falhas geradas por exceções não capturadas e, podem ser utilizados futuramente como testes de regressão.

Em um trabalho mais recente, Amalfitano (2015) propôs uma ferramenta chamada MOBIGuitar, uma evolução do A^2T^2, onde um modelo sensível ao estado da aplicação é extraído no formato de uma máquina de estados e os casos de teste são gerados com base em um critério de cobertura par a par. Com base nesse critério, cada par de estados da máquina dá origem a um caso de teste.

Machiry (2013) propôs uma ferramenta chamada Dynodroid, que automaticamente gera entradas para aplicativos Android. A ferramenta primeiramente identifica os eventos que são relevantes para um determinado estado da aplicação. Em seguida, um desses eventos é selecionado e executado, levando a aplicação a um novo estado. O processo se repete para um certo número (parametrizado) de eventos. Um evento na interface do aplicativo, é considerado relevante se possuir um *listener* dedicado na implementação do aplicativo.

Semelhante a esse trabalho, as abordagens citadas visam fornecer suporte à geração de testes funcionais. No entanto, eles não assumem nenhum conhecimento prévio sobre o comportamento do aplicativo e exigem um esforço considerável para extrair um modelo de fluxo de controle. Embora essa extração seja automatizada, não é claro como a manutenção e a evolução do caso de teste são realizadas, o que é crucial no domínio móvel. A abordagem proposta neste trabalho é, portanto, complementar a essas. Na abordagem proposta, tenta-se utilizar as informações conhecidas sobre o fluxo da aplicação (o que é explicito em cenários de uso) e propor cenários adicionais para o testador, aproveitando a cadeia de ferramentas de automação de teste já utilizadas pela equipe de projeto. Acredita-se que tal abordagem auxilia o processo de qualidade geral do aplicativo móvel, pois promove o aprimoramento dos testes e o compartilhamento de informações entre os analistas do negócio, desenvolvedores e testadores.

3 PROPOSTA

Considere um modelo de projeto onde os cenários do BDD de alto nível (não baseado em componentes da interface) sejam utilizados como fonte de informação para a criação dos testes de sistema. A equipe de teste recebe esses cenários do BDD e implementa as etapas correspondentes ao teste. Assim, à medida que a aplicação evolui, toda a equipe conhece o percentual de cenários implementados. Essa prática pode levar a equipe a confiar cegamente nos cenários como suficientes para testar o sistema implementado, pois apenas utilizam como medida de confiança a quantidade de cenários que foram desenvolvidos. Naturalmente, também é possível medir a cobertura de código alcançada pela execução dos testes. Porém não há incentivo para avaliar como os cenários originais realmente exercitam as funcionalidades do sistema. Espera-se que os cenários descritos cubram uma alta porcentagem dos recursos implementados, porém nem sempre isso ocorre. Por exemplo, uma única interface de usuário pode ser compartilhada por múltiplas funcionalidades separadamente e/ou parcialmente documentadas. Como outro exemplo, componentes comuns da interface, como um botão de “voltar” ou um menu, podem apresentar ou até mesmo exigir comportamentos distintos quando utilizados em diferentes pontos do aplicativo, e essa situação pode não estar descrita nos cenários originais do BDD.

Este trabalho propõe uma abordagem baseada no BDD para ajudar na definição de testes de sistema eficazes para aplicações nativas do Android. A abordagem proposta utiliza os arquivos de leiaute do Android para extrair informações sobre eventos e transições de tela esperados no sistema. Além disso, os cenários do BDD de baixo nível são utilizados para documentar os testes de sistema que serão executados através da interface de usuário. Considerando os cenários de sistema do BDD descritos a nível da interface, isto é, que explicitamente fazem referência aos elementos do leiaute, é possível realizar tanto a análise de cobertura de utilização dos elementos da interface quanto a geração automática de testes adicionais. Isso permite também que os testes de sistema tirem proveito das ferramentas existentes para automação de teste associadas ao BDD.

A abordagem proposta é composta de quatro partes:

1) Visualização gráfica dos cenários do BDD: O arquivo contendo os cenários do BDD é lido e interpretado dando origem a um grafo que representa o comportamento do aplicativo. A leitura do arquivo de cenários é feita linha por linha, onde são identificadas sentenças especiais que dão origem aos componentes do grafo. Neste ponto são identificados

os possíveis elementos do leiaute (descritos nos cenários com base na UI) bem como as telas a que eles pertencem.

2) Avaliação dos cenários do BDD em relação a utilização dos componentes da interface: Os arquivos de leiaute são utilizados para identificar quais os elementos da interface não foram exercitados pelos cenários. Por meio da análise sintática dos arquivos de leiaute, são identificados e comparados os verdadeiros elementos com aqueles extraídos das especificações dos cenários. Se algum dos elementos do leiaute não for encontrado em nenhum dos cenários significa que este elemento não foi testado.

3) Visualização gráfica dos elementos identificados como não testados: Todos os elementos identificados como não testados são conectados a um nó especial do grafo chamado "Alerta". O nó "Alerta" é apresentado com destaque no grafo a fim de facilitar a visualização dos elementos não testados.

4) Sugestão de novos testes: Os elementos do leiaute que foram identificados como não testados dão origem a um novo cenário.

As equipes de projeto que já utilizam os cenários do BDD descritos em termos da UI, podem utilizar a abordagem proposta diretamente. As equipes de projeto que utilizam os cenários de alto nível (não baseados na UI) ainda podem utilizar a abordagem proposta, porém devem primeiro traduzir os cenários originais para cenários baseados na UI. Acredita-se que este trabalho de tradução adicional seja compensado por ganhos de automação na geração dos testes de sistema e análise de cobertura. Além disso, em posse desses dois artefatos (cenários de alto nível e cenários com base na UI), pode-se relacionar os testes de sistema com os requisitos originais, aumentando a rastreabilidade do teste.

Embora seja comum ver equipes de desenvolvimento utilizarem o BDD puramente como um artefato de teste, este trabalho não defende o uso do BDD somente para esse propósito.

A partir da abordagem proposta, acredita-se que a estrutura de um cenário do BDD possa ser utilizada em conjunto com os arquivos de leiaute da aplicação para gerar cenários que exercitem completamente os elementos da interface do aplicativo. Cada parte da abordagem proposta é detalhada a seguir por meio de um exemplo.

3.1 APLICATIVO EXEMPLO: BLUETAPP

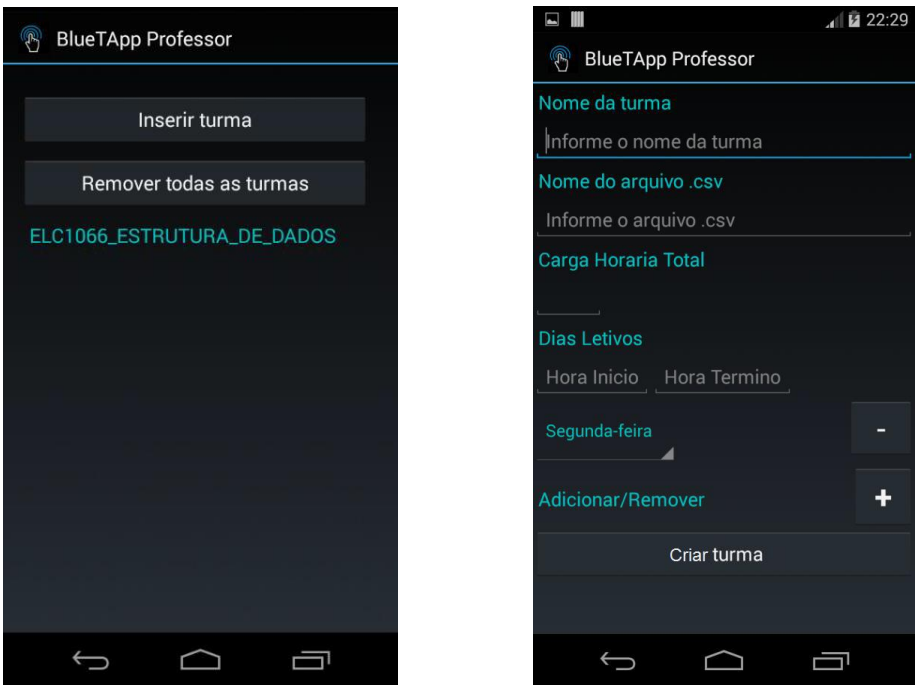
Para explicar a abordagem proposta, um aplicativo chamado BlueTApp, será utilizado como exemplo. BlueTApp é um aplicativo que permite o controle da frequência acadêmica

em instituições de ensino. O aplicativo possibilita controlar a frequência dos discentes manualmente, ou através da comunicação via Bluetooth com seus dispositivos. O aplicativo é composto por oito *activities*, cada uma representada por uma tela na aplicação. Além disso, possui dez funcionalidades, entre gerenciamento de estudantes e classes, e quatorze possíveis transições entre as *activities*. A Figura 3.1 apresenta duas telas do aplicativo BlueTApp, enquanto a Figura 3.2 apresenta seus leiautes representados através de arquivos XML.

A primeira tela apresenta duas características: criação de uma nova classe e remoção de todas as classes registradas. Já a segunda, apresenta um formulário para a criação de uma nova classe.

Figura 3.1 – Telas do aplicativo BlueTApp.

(a) Tela principal. (b) Tela de criação de uma nova classe.



Fonte: Elaborado pelo autor.

Através da análise da Figura 3.2, pode-se observar como os componentes do leiaute são organizados. Com o uso de *tags*, cada componente e suas propriedades são descritos. Vale notar que cada componente pode possuir um atributo id e um atributo *text* associado (linhas 4, 5, 9, 10, 14 na Figura 3.2 (a) e linhas 3, 7, 12, 15, 17, 22, 25, 27, 32, 35 na Figura 3.2 (b)). Essas propriedades são utilizadas para identificar os componentes do leiaute.

Figura 3.2 – Arquivos XML correspondente às telas do BlueTApp mostradas na Figura 3.1.

(a) Arquivo de leiaute da tela principal.

```xml
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"

    <Button android:id="@+id/bt_inserir_turma"
        android:text="Inserir Turma"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />

    <Button android:id="@+id/bt_remover_todas_turmas"
        android:text="Remover Todas as Turmas"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />

    <ListView android:id="@+id/lv_turmas_main"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
</RelativeLayout>
```

Fonte: Elaborado pelo autor.

(b) Arquivo de leiaute da tela de criação de uma nova turma.

```xml
<RelativeLayout android:layout_width="match_parent" android:la
    <RelativeLayout android:id="@+id/ll_turma" android:layout_
        <TextView android:id="@+id/tv_nome_turma"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <EditText android:id="@+id/et_nome_turma"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Insira o nome da turma" />

        <TextView android:id="@+id/tv_nome_arquivo_csv"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Arquivo CSV" />

        <EditText android:id="@+id/et_nome_arquivo_csv"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Insira o nome do arquivo CSV" />

        <TextView android:id="@+id/tv_carga_horaria_total"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Carga Horária" />

        <EditText android:id="@+id/et_carga_horaria_total"
            android:layout_width="67dp"
            android:layout_height="fill_parent" />
    </RelativeLayout>
    <LinearLayout android:layout_width="fill_parent" android:la
        <Button android:id="@+id/bt_criar_turma"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Criar Turma" />
    </LinearLayout>
</RelativeLayout>
```

Fonte: Elaborado pelo autor.

3.1.1 Visualização gráfica dos cenários do BDD

Como exemplo, considere um ambiente onde o aplicativo BlueTApp possua somente um cenário escrito em alto nível (não baseado na UI), descrevendo os passos para a criação de uma nova turma. Antes de utilizar a abordagem proposta é preciso efetuar a tradução desse cenário para um cenário de baixo nível (com base na UI).

Se os cenários do sistema sob teste estiverem descritos em termos dos elementos da interface, a relação entre o arquivo de cenários e o grafo gerado é direta. No entanto, como mencionado anteriormente, os cenários do BDD devem ser descritos utilizando somente conceitos de alto nível, completamente livres de relações com os elementos do leiaute. Nesse caso, não é possível relacionar diretamente os cenários do BDD com os componentes do leiaute. Para as equipes de desenvolvimento que seguem esta boa prática, propõe-se que um novo arquivo no estilo do BDD seja criado e utilizado apenas para fins de testes de sistema. Esse novo arquivo deve conter todos os cenários do BDD de alto nível (não baseados na UI) descritos em termos dos componentes e comandos da interface de usuário.

A Figura 3.3 (a) mostra o cenário em alto nível com os passos para a criação de uma nova turma no aplicativo BlueTApp, enquanto a Figura 3.3 (b) mostra o mesmo cenário descrito em termos dos elementos do leiaute.

Figura 3.3 – Cenários do BDD com diferentes níveis de abstração**.**

(a) Cenário para criação de uma nova classe em termos do usuário.

```
1  Cenário 1: Criação de uma nova turma.
2  Dado que eu abri o aplicativo.
3      E quero adicionar uma nova turma.
4  Quando forneço as informações necessárias.
5      E confirmo a operação.
6  Então eu devo visualizar a nova turma.
```

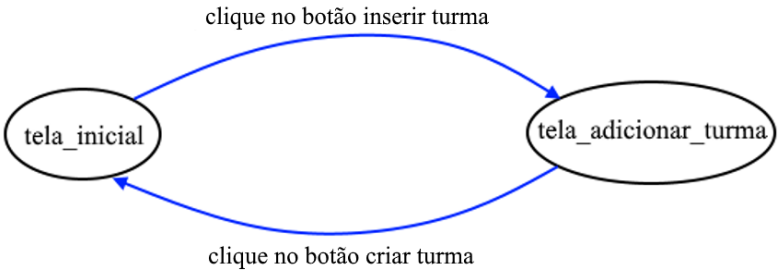
(b) Cenário para a criação de uma nova classe em termos dos elementos do leiaute**.**

```
1  Cenário 1: Criação de uma nova turma em termos dos elementos de UI.
2  Dado que estou na tela “inicial”.
3      E clico no botão “inserir turma”
4      E aparece a tela “adicionar turma”.
5  Quando preencho todos os campos do formulário.
6      E clico no botão “criar turma”.
7  Então deve aparecer a tela “inicial”.
8      E deve aparecer uma mensagem de confirmação.
9      E deve aparecer a nova turma.
```

Fonte: Elaborado pelo autor.

A partir do cenário de baixo nível (Figura 3.3 (b)), é fornecida a visualização gráfica do cenário por meio de um grafo, como mostra a Figura 3.4.

Figura 3.4– Grafo gerado utilizando o cenário do BDD.



Fonte: Elaborado pelo autor.

Analisando o grafo é possível perceber que tanto as telas como os elementos responsáveis pelas transições (mudanças de tela), descritos nos cenários, foram identificados corretamente. Da mesma maneira, o comportamento esperado por parte da aplicação diante da interação do usuário com o sistema, também foi representado corretamente através do grafo.

A abordagem proposta realiza a leitura e interpretação dos cenários linha a linha e inicia a análise com a busca pela sentença (Cenário:) utilizada para demarcar o início da descrição de um cenário. A partir desse ponto, todo o texto encontrado, é associado a um objeto utilizado para representar um cenário. O processo de associação do texto ao objeto se repete até o início de uma nova descrição de cenário e o processo de leitura até o final do arquivo. O texto capturado é analisado por um algoritmo onde um dicionário de sentenças pré-definidas é utilizado para identificar telas, elementos gráficos e transições que farão parte do grafo. Os algoritmos utilizados nessa etapa da abordagem, para captura e análise do texto, são apresentados respectivamente nas Figuras 3.5 e 3.6.

Figura 3.5 – Pseudocódigo do algoritmo utilizado para capturar o texto dos cenários.

Algoritmo de Captura do Texto

```
1.  Seja Arquivo o arquivo contendo os cenários do BDD baseados em UI.
2.  Seja TagCenario a tag "Cenario:"
3.  Seja Texto o objeto utilizado para armazenar todo o texto capturado do arquivo.
4.  Seja TextoCenario o objeto utilizado para armazenar o texto de um cenário.
5.  Seja ListaCenarios uma lista para armazenar os objetos que representam um cenário.
6.  Texto ← getTexto(Arquivo)
7.  Para cada Linha em Texto faça
8.    Se Linha contém TagCenario então
9.      Repita
10.       Linha ← Le_Linha(Texto)
11.       Adiciona_Texto(TextoCenario, Linha)
12.     Enquanto Linha não contém TagCenario
13.     Cenario ← Cria_Cenario(TextoCenario)
14.     Adiciona_Cenario(ListaCenario, Cenario)
15.     Retrocede_Uma_Linha()
16.   Fim_Se
17. Fim_Para
```

Fonte: Elaborado pelo autor.

Figura 3.6 – Pseudocódigo do algoritmo utilizado para analisar o texto dos cenários.

Algoritmo de Análise do Texto

```
1.  Seja ListaCenarios uma lista de objetos que representam um cenário.
2.  Seja Dicionario um dicionário com sentenças pré-definidas utilizadas para identificar no
3.  texto dos cenários os elementos do grafo.
4.  Para cada Cenario em ListaCenarios faça
5.    Texto ← get_Texto(Cenario)
6.    Repita
7.      Linha ← le_linha(Texto)
8.      Para cada Sentenca em Dicionario faça
9.        Se Linha contém Sentenca então
10.         Elemento ← Cria_Elemento(get_Nome_Entre_Aspas(Linha))
11.         Adiciona_Elemento(Cenario, Elemento)
12.       Fim_Se
13.     Fim_Para
14.   Enquanto Linha != null
15. Fim_Para
```

Fonte: Elaborado pelo autor.

3.1.2 Avaliação dos cenários do BDD em relação a utilização dos componentes da interface

Para avaliar a cobertura total dos cenários em relação aos elementos do leiaute do aplicativo, é preciso identificar primeiro quais elementos gráficos fazem parte da interface da aplicação. Para isso, realiza-se uma análise sintática nos arquivos XML de leiaute.

Da mesma forma que na etapa anterior, a leitura dos arquivos de leiaute também é realizada linha a linha. Para isso utiliza-se um algoritmo que procura por tags específicas do Android, utilizadas para a declaração dos elementos gráficos nos arquivos XML. Nessa etapa da abordagem são considerados somente os principais elementos responsáveis pelas transições de estados entre as *activities*. São eles: botões, imagens em forma de botões, listas e itens (componentes/botões dos menus). O algoritmo utilizado para a identificação desses elementos é apresentado na Figura 3.7.

Figura 3.7 – Pseudocódigo do algoritmo de identificação dos elementos do leiaute.

Algoritmo de Identificação dos Elementos do Leiaute

```
1.  Seja ListaArquivos o conjunto de arquivos de leiaute do aplicativo.
2.  Seja Dicionario um dicionário com as principais tags utilizadas para declarar os
3.  elementos gráficos no Android.
4.  Seja id a tag "android:id=".
5.  Seja text a tag "android:text=".
6.  Seja TagDeFechamento a tag "/>".
7.  Seja ListaElementos uma lista com os objetos utilizados para armazenar os elementos
8.  Identificados nos arquivos de leiaute.
9.  Para cada Leiaute em ListaLeiaute faça
10.   Repita
11.     Linha ← Le_Linha(Leiaute)
12.     Para cada Tag em Dicionario faça
13.       Se Linha contém Tag então
14.         Elemento ← Cria_Elemento()
15.         Repita
16.           Linha ← Le_Linha(Leiaute)
17.           Se Linha contém id então
18.             set_Elemento_id(Elemento, get_Id(Linha))
19.           Senão Se Linha contém text
20.             set_Elemento_text(Elemento, get_Text(Linha))
21.           Fim_Se
22.         Enquanto Linha não contém tagDeFechamento
23          Adiciona_Elemento(ListaElementos, Elemento)
24.       Fim_Se
25.     Fim_Para
26.   Enquanto Linha != null
27. Fim_Para
```

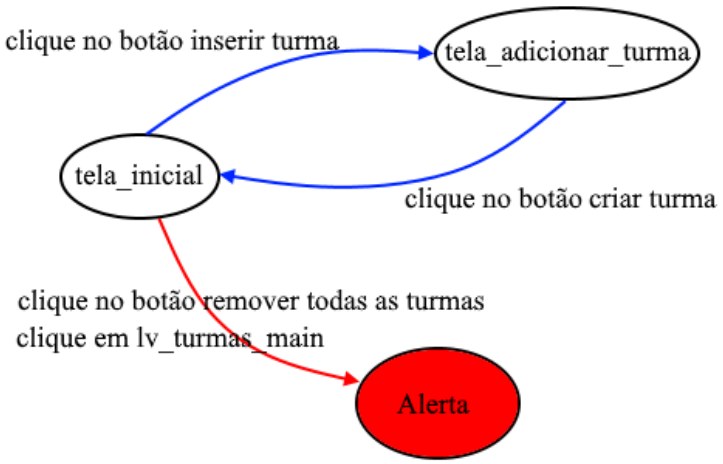
Fonte: Elaborado pelo autor.

Uma vez identificados, esses elementos são comparados com aqueles extraídos da etapa anterior. Se algum dos componentes gráficos identificados não for encontrado em nenhum dos cenários, significa que este elemento não foi exercitado e o elemento é marcado então como não testado.

3.1.3 Visualização gráfica dos elementos identificados como não testados

Os elementos do leiaute que não foram exercitados pelos cenários possivelmente representam casos de testes adicionais. Na abordagem proposta, os testes faltantes são representados no grafo como arcos que ligam os componentes não testados de cada tela a um nó especial do grafo chamado “Alerta”. Eles representam eventos na interface do aplicativo que não foram exercitados pelos cenários do BDD descritos em nível de UI. Por exemplo, a Figura 3.8 mostra o grafo gerado a partir da análise do cenário apresentado na Figura 3.3 (b) e dos leiautes descritos na Figura 3.2.

Figura 3.8 – Grafo mostrando os testes faltantes no aplicativo BlueTApp.



Fonte: Elaborado pelo autor.

Através da análise do grafo nota-se que, tanto o botão “remover todas as turmas” como a lista de itens, foram identificados como não testados. De fato, no cenário apresentado na Figura 3.3 (b), não existe menção a nenhum desses elementos do leiaute. Assim sendo, a abordagem proposta considera que esses elementos darão origem a novos casos de teste.

Além disso, vale ressaltar que as interações com os elementos não testados geram no grafo um arco para o nó especial “Alerta”. Isso acontece pois, somente com a análise sintática dos arquivos de leiaute da aplicação, sem a análise do código fonte, é impossível determinar para qual estado esse elemento levará a aplicação. As mudanças de estado do aplicativo

expressas no grafo, são identificadas através dos passos descritos nos cenários do BDD (seção 3.1.1).

3.1.4 Sugestão de novos testes

Além de mostrar as interações faltantes (testes ausentes), a abordagem proposta gera ainda um modelo de cenário envolvendo os componentes não testados. Um cenário parcial de teste é gerado no formato do BDD e anexado ao arquivo de cenários original. A Figura 3.9 mostra o arquivo de cenários utilizado nesse exemplo após a identificação dos elementos não exercitados.

Figura 3.9 – Modelo de cenário descrevendo novos testes para o aplicativo BlueTApp.

```
1  Cenário 1: Criação de uma nova turma em termos dos elementos de UI.
2  Dado que estou na tela “inicial”.
3      E clico no botão “inserir turma”
4      E aparece a tela “adicionar turma”.
5  Quando preencho todos os campos do formulário.
6      E clico no botão “criar turma”.
7  Então deve aparecer a tela “inicial”.
8      E deve aparecer uma mensagem de confirmação.
9      E deve aparecer a nova turma.
10
11 #### CENÁRIOS PARCIAIS ####
12 Cenário 2: Clique no botão remover todas as turmas.
13 Dado que estou na tela “inicial”.
14 Quando clico no botão “remover todas as turmas”.
15 Então aparece a tela “Alerta”.
16
17 Cenário 3: Clique em lv_turmas_main.
18 Dado que estou na tela “inicial”.
19 Quando clico em “lv_turmas_main”.
20 Então aparece a tela “Alerta”.
```

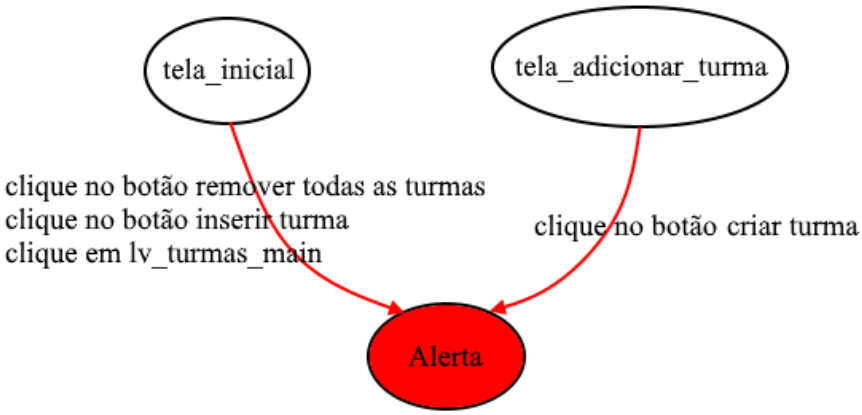
Fonte: Elaborado pelo autor.

Por meio da Figura 3.9, pode-se observar que os cenários adicionais foram criados. Esses cenários podem ser considerados parciais, uma vez que não é possível determinar, de fato, para qual estado os elementos não testados levarão a aplicação. Sendo assim, o testador deve completar esses cenários da forma que achar mais adequada.

A abordagem proposta fornece ainda a possibilidade de automação da tarefa de escrita dos cenários. Quando a aplicação sob teste não possuir nenhum cenário descrito, a análise passa a ser exclusivamente sobre os arquivos de leiaute da aplicação, onde todos os elementos

gráficos são identificados como não testados. Dessa forma, o grafo gerado mostra todos os elementos conectados ao nó “Alerta” e um arquivo de saída com os possíveis cenários de teste é criado. A Figura 3.10 mostra o grafo gerado para os arquivos de leiaute apresentados na Figura 3.2 enquanto a Figura 3.11 apresenta os cenários parciais gerados.

Figura 3.10 – Grafo parcial do aplicativo BlueTApp.



Fonte: Elaborado pelo autor.

Figura 3.11 – Cenários parciais do aplicativo BlueTApp.

```
##### CENÁRIOS PARCIAIS #####
  Cenário 1: Clique no botão inserir turma.
  Dado que estou na tela “inicial”.
  Quando clico no botão “inserir turma”.
  Então aparece a tela “Alerta”.

  Cenário 2: Clique no botão remover todas as turmas.
  Dado que estou na tela “inicial”.
  Quando clico no botão “remover todas as turmas”.
  Então aparece a tela “Alerta”.

  Cenário 3: Clique em lv_turmas_main.
  Dado que estou na tela “inicial”.
  Quando clico em “lv_turmas_main”.
  Então aparece a tela “Alerta”.

  Cenário 4: Clique no botão criar turma.
  Dado que estou na tela “adicionar turma”.
  Quando clico no botão “criar turma”.
Então aparece a tela “Alerta”.
```

Fonte: Elaborado pelo autor.

Para executar a abordagem proposta foi implementada uma ferramenta que, por meio de um script, executa os quatro passos descritos anteriormente. O próximo capítulo detalha o funcionamento da ferramenta.

4 ANDROID BEHAVIOR TESTING TOOL

A abordagem proposta no capítulo 3 foi implementada por meio de uma ferramenta de suporte batizada de *Android Behavior Testing Tool* (ABTT). A ferramenta implementada pode funcionar de três maneiras distintas de acordo com os arquivos de entrada utilizados:

1) Utilizando como parâmetro de entrada apenas os arquivos de cenários do BDD (com base na UI): neste modo, a ferramenta fornece uma visualização gráfica, por meio da criação de um grafo (Figura 3.4), das operações de sistemas exercitadas através da interação com a interface. Neste modo, o testador tem uma visão geral dos testes de sistemas especificados e pode verificar, por exemplo, inconsistências óbvias. Este modo também é útil para o analista de negócios avaliar mais facilmente o conjunto de testes especificado.

2) Utilizando como parâmetro de entrada os arquivos de cenários do BDD em conjunto com os arquivos de leiaute do aplicativo: a ferramenta fornece uma visualização gráfica das operações de sistema exercitadas através da interface e identifica os componentes do leiaute que não foram testados (Figura 3.8). Isso é mostrado explicitamente para o usuário por meio da criação de um nó especial no grafo, chamado “Alerta”. O nodo do grafo que possui o componente de interface identificado como “não testado” é conectado ao nodo “Alerta”. Além disso, os cenários parciais, para os componentes da interface não testados, são automaticamente anexados ao arquivo de cenários do BDD (o mesmo utilizado como parâmetro de entrada).

3) Utilizando como parâmetro de entrada apenas os arquivos de leiaute do aplicativo: a ferramenta fornece uma visão distorcida do sistema, conectando todos os nodos do grafo ao nodo “Alerta” (Figura 3.10). Além disso, é gerado também um arquivo de saída, em formato textual, contendo os cenários parciais de teste, que devem ser completados posteriormente pelo testador. Esse modo de utilização é útil em situações em que o sistema sob teste não possui nenhum cenário criado. Dessa forma, a tarefa de escrita é parcialmente automatizada, servindo como um ponto de partida para a escrita dos testes de sistema.

Para o bom funcionamento da ferramenta, alguns cuidados são necessários:

I. Os arquivos de leiaute do aplicativo devem ter o mesmo nome das *activities* a que pertencem. Isso é necessário pois, como dito anteriormente, por meio da análise sintática exclusiva dos arquivos de leiaute (sem a análise do código fonte) é impossível identificar a qual *activity* o arquivo de leiaute pertence. O nome dos arquivos é utilizado como identificador da *activity* que o leiaute faz referência.

II. Se o aplicativo possui menus, representados por meio de arquivos no formato XML, eles devem conter um comentário no início do arquivo com o seguinte formato: <!- Telas: [nome das *activities* a que o menu pertence, separadas por vírgula] ->. Isso é necessário pois, assim como nos arquivos de interface, é impossível identificar a qual *activity* um menu pertence sem uma análise mais profunda.

III. Os atributos *text* e id dos componentes de interface, definidos nos arquivos XML de leiaute, são utilizados pela ferramenta para dar nome aos elementos gráficos que geram as transições de estado entre os nodos do grafo. Sendo assim, recomenda-se utilizar nos cenários do BDD, para referenciar os componentes do leiaute, os mesmos nomes definidos nesses atributos. Se isto não for feito, isto é, se os cenários do BDD não utilizarem os mesmos nomes definidos nos campos *text* ou id dos arquivos de leiaute, o grafo será gerado errado, mostrando os elementos identificados no arquivo de cenários como testados e aqueles identificados nos arquivos de leiaute como não testados. Isso ocorre pois, nessa situação a ferramenta não consegue identificar que esses componentes são os mesmos.

IV. Os cenários devem respeitar o formato proposto pelo BDD e utilizar as palavras reservadas “Dado que”, “Quando”, “E”, e “Então” para iniciar uma sentença. Além disso, os nomes dos componentes do leiaute devem aparecer entre aspas duplas e as ações devem ser descritas por meio da utilização de uma das sentenças a seguir: “Estou na tela”, “clico no botão”, “clico em”, “digito no”, “faço movimento de” (utilizado para a realização de movimentos como *zoom* ou mudança de orientação do dispositivo).

Como já mencionado, a ferramenta utiliza, em dois modos de operação, os arquivos de leiaute do aplicativo. Caso o utilizador da ferramenta não tenha acesso aos arquivos de leiaute, é possível obtê-los utilizando o arquivo .apk (instalador) da aplicação Android como parâmetro de entrada para a ferramenta Apktool (Apktool, 2017), que utiliza engenharia reversa para, a partir do arquivo de instalação do aplicativo, gerar os arquivos de leiaute. O arquivo de instalação de uma aplicação encontra-se no diretório /data/app do dispositivo do usuário. Porém para ter acesso a esse diretório é preciso ter todos os privilégios de administrador do dispositivo liberados (*root*). Caso contrário o diretório fica oculto para o usuário. Se o usuário não possuir um dispositivo com os privilégios liberados, é possível obter facilmente os arquivos .apk das mais diversas aplicações no site https://apkpure.com/ (ApkPure, 2017).

4.1 ARQUITETURA

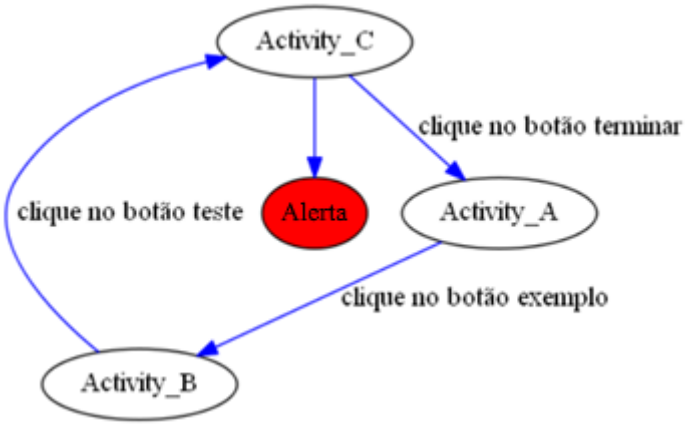
A ABTT trabalha em conjunto com o software Graphviz (Graphviz, 2017), que é uma simples ferramenta de código aberto para a visualização de grafos e possui uma linguagem associada para a descrição desse tipo de estrutura. Assim, a estrutura dos grafos, extraída do arquivo de cenários e/ou arquivos de leiaute, é salva em um arquivo de texto na notação Graphviz. A Figura 4.1 mostra um exemplo da notação utilizada e a Figura 4.2 o grafo correspondente.

Figura 4.1 – Notação utilizada no software Graphviz

```
01 Digraph{
02     edge[color="blue"];
03     Alerta [style=filled, fillcolor=red];
04     Activity_A → Activity_B [label = "clique no botão exemplo"];
05     Activity_B → Actvity_C [label = "clique no botão teste"];
06     Activity_C → Activity_A [label = "clique no botão terminar"];
07     Activity_C → Alerta;
08 }
```

Fonte: Elaborado pelo autor.

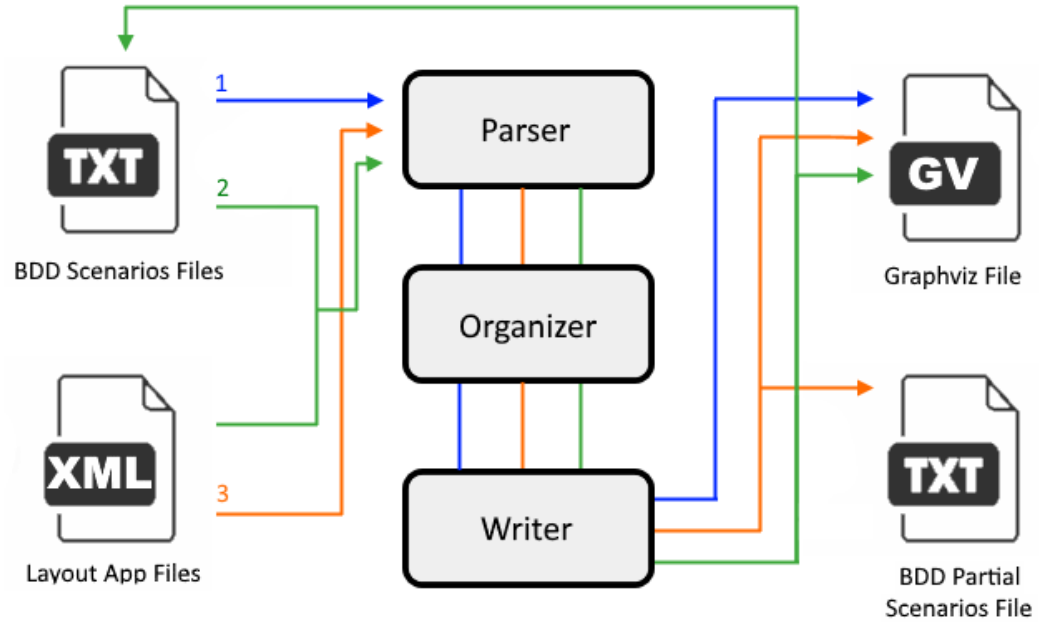
Figura 4.2 – Grafo correspondente a notação utilizada na Figura 4.1.



Fonte: Elaborado pelo autor.

O ABTT é composto de três módulos distintos que cooperam entre si e são responsáveis por diferentes funções. A Figura 4.3 mostra a arquitetura da ferramenta e os três possíveis modos de operação.

Figura 4.3– Arquitetura da ABTT e seus três possíveis modos de operação.



Fonte: Elaborado pelo autor.

Parser: é o responsável pela leitura dos arquivos de entrada fornecidos para a ferramenta. Por meio da análise sintática dos arquivos, a ferramenta é capaz de identificar as telas da aplicação e a maneira como interagem.

Organizer: é o responsável pela remoção de duplicações, tanto de nodos como de arestas, e pela formatação dos dados. Os nodos duplicados ocorrem quando uma tela específica aparece em mais de um cenário. Os arcos duplicados ocorrem quando, em uma tela específica A, podem ser realizadas diversas operações que gerem mais de um arco de saída para uma mesma tela B. A ferramenta cuida de identificar essas situações e concatenar os nós/arcos em um único elemento. Além disso, caso ocorra a situação descrita na seção 2.2.1, de uma mesma *activity* possuir um conjunto de leiautes semelhantes, com os mesmos elementos, porém de tamanhos diferentes, o usuário da ferramenta tem a liberdade de fornecer todos esses arquivos para a ferramenta ou somente um deles. Caso mais de um seja fornecido, a ferramenta se encarrega de identificar os elementos semelhantes e remover as duplicações. O *organizer* é também o módulo responsável pela formatação dos dados, que é realizada para facilitar a tarefa de escrita das saídas da ferramenta que serão executadas pelo próximo e último módulo.

Writer: é o responsável pela escrita dos arquivos de saída gerados pela ferramenta. Um ou dois arquivos de saída podem ser gerados, dependendo dos arquivos fornecidos como

parâmetros de entrada. Nos três modos de operação um arquivo de saída (referente ao grafo) com extensão .GV é criado para ser interpretado posteriormente pelo Graphviz.

No modo de operação 2, quando são fornecidos como entrada para a ferramenta o arquivo de cenários, juntamente com os arquivos de leiaute da aplicação, é gerado, além do arquivo GV, um conjunto de cenários parciais que são anexados no arquivo de cenários utilizado como entrada para a ferramenta. Esse modo de operação é representado na Figura 4.3 pela cor verde.

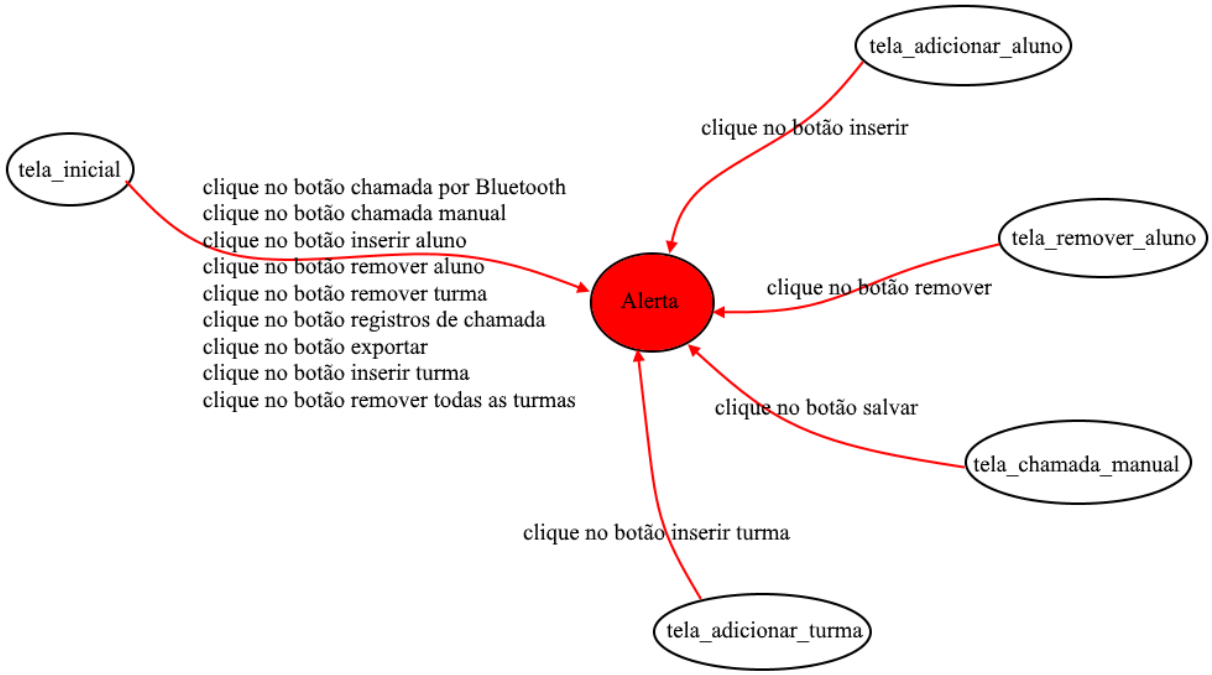
Se os arquivos XML de leiaute da aplicação forem fornecidos para a ferramenta de forma isolada (modo de operação 3, representado na Figura 4.3 pela cor laranja) um arquivo com extensão TXT também é criado, contendo os novos cenários parciais.

4.2 UM EXEMPLO DE USO

O uso do ABTT será exemplificado no contexto do processo de teste do aplicativo BlueTApp. O BlueTApp não foi desenvolvido usando a abordagem do BDD, portanto, nenhum cenário do BDD baseado em UI estava disponível.

Por esse motivo, a ferramenta será utilizada primeiro no modo de operação 3, isto é, fornecendo somente os arquivos de leiaute do aplicativo como parâmetros de entrada. Neste modo, observa-se que somente os componentes do leiaute são identificados, como mostra a Figura 4.4.

Figura 4.4– Grafo parcial do aplicativo BlueTApp.



Fonte: Elaborado pelo autor.

Pode-se observar que todos os arcos apontam para o nodo “Alerta”, o que significa que nenhum teste de sistema foi identificado ainda. Como nenhum cenário foi fornecido à ferramenta, não é possível detectar as interações entre as possíveis telas do sistema. Também deve ser observado que somente algumas *activities* da aplicação foram detectadas nesta etapa, mais especificamente aquelas que possuem algum componente de leiaute que, ao ser exercitado, pode gerar uma transição de telas. O uso da ferramenta no modo de operação 3 gerou ainda um total de quatorze cenários parciais.

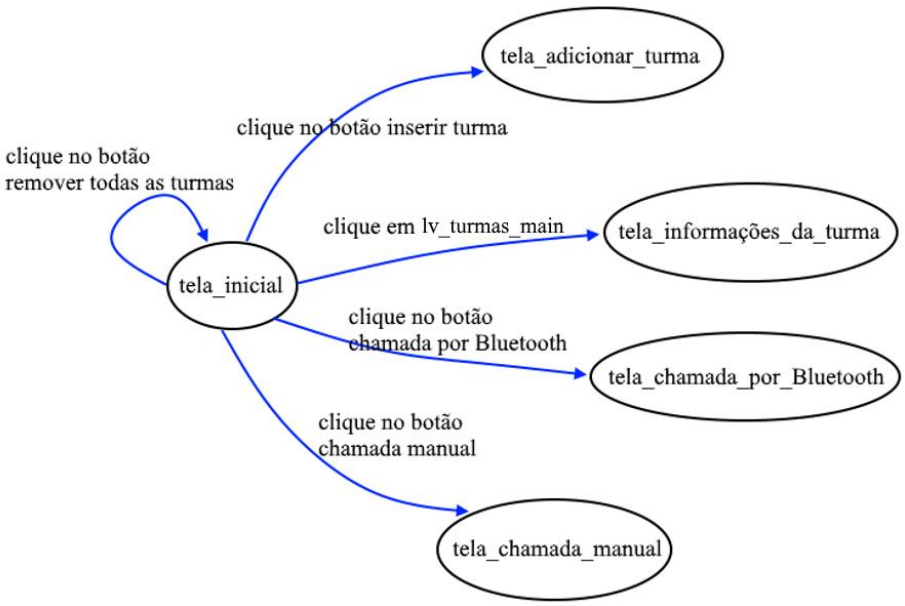
Para exemplificar o uso do ABTT no modo de operação 1, isto é, utilizando somente os arquivos de cenários, cinco dos quatorze cenários parciais gerados pelo uso da ferramenta no modo de operação 3, serão completados. Esses cenários serão utilizados para validar alguns dos componentes da interface que, quando estimulados, alteram o estado da aplicação, produzindo a transição entre as *activities*. A fim de exemplificação os cenários restantes foram desconsiderados. Esses cinco cenários, após completados, foram fornecidos como entrada para a ferramenta, que agora opera no modo 1 (utilizando somente o arquivo de cenários). Os cenários são apresentados na Figura 4.5 e o grafo resultante na Figura 4.6.

Figura 4.5 – Cenários do aplicativo BlueTApp.

```
1  Cenário 1: Clique no botão inserir turma.
2  Dado que estou na tela “inicial”.
3  Quando clico no botão “inserir turma”.
4  Então aparece a tela “adicionar turma”.
5
6  Cenário 2: Clique no botão remover todas as turmas.
7  Dado que estou na tela “inicial”.
8  Quando clico no botão “remover todas as turmas”.
9  Então aparece uma mensagem “operação realizada com sucesso”.
10
11 Cenário 3: Clique em lv_turmas_main.
12 Dado que estou na tela “inicial”.
13 Quando clico em “lv_turmas_main”.
14 Então deve aparecer a tela “informações da turma”.
15
16 Cenário 4: Clique no botão chamada por Bluetooth.
17 Dado que estou na tela “inicial”.
18 Quando clico no botão “chamada por Bluetooth”.
19 Então aparece a tela “chamada por Bluetooth”.
20
21 Cenário 5: Clique no botão chamada manual.
22 Dado que estou na tela “inicial”.
23 Quando clico no botão “chamada manual”.
24 Então deve aparecer a tela “chamada manual”.
```

Fonte: Elaborado pelo autor.

Figura 4.6 – Grafo resultante dos cenários fornecidos à ferramenta.

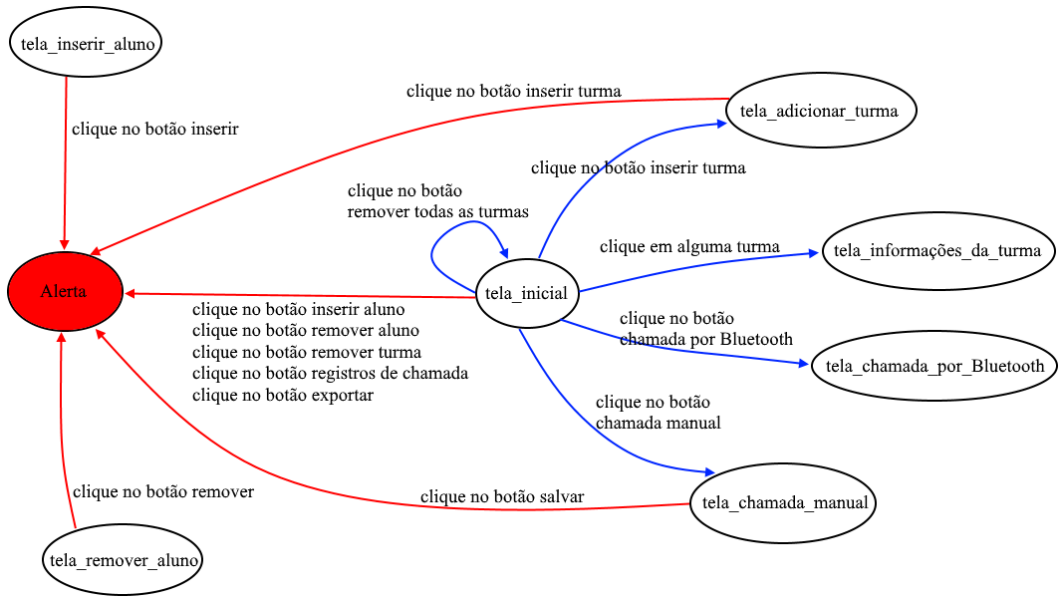


Fonte: Elaborado pelo autor.

Pode-se observar que o grafo representa corretamente o fluxo comportamental da aplicação conforme descrito nos cenários do BDD.

Por último, será exemplificado a utilização da ferramenta no modo de operação 2, isto é, fornecendo os arquivos de cenários (Figura 4.5) juntamente com os arquivos de leiaute como parâmetros de entrada. O grafo obtido através da utilizada do ABTT no modo de operação 2 é apresentado na Figura 4.7.

Figura 4.7 – Grafo de cobertura resultante para o aplicativo BlueTApp.



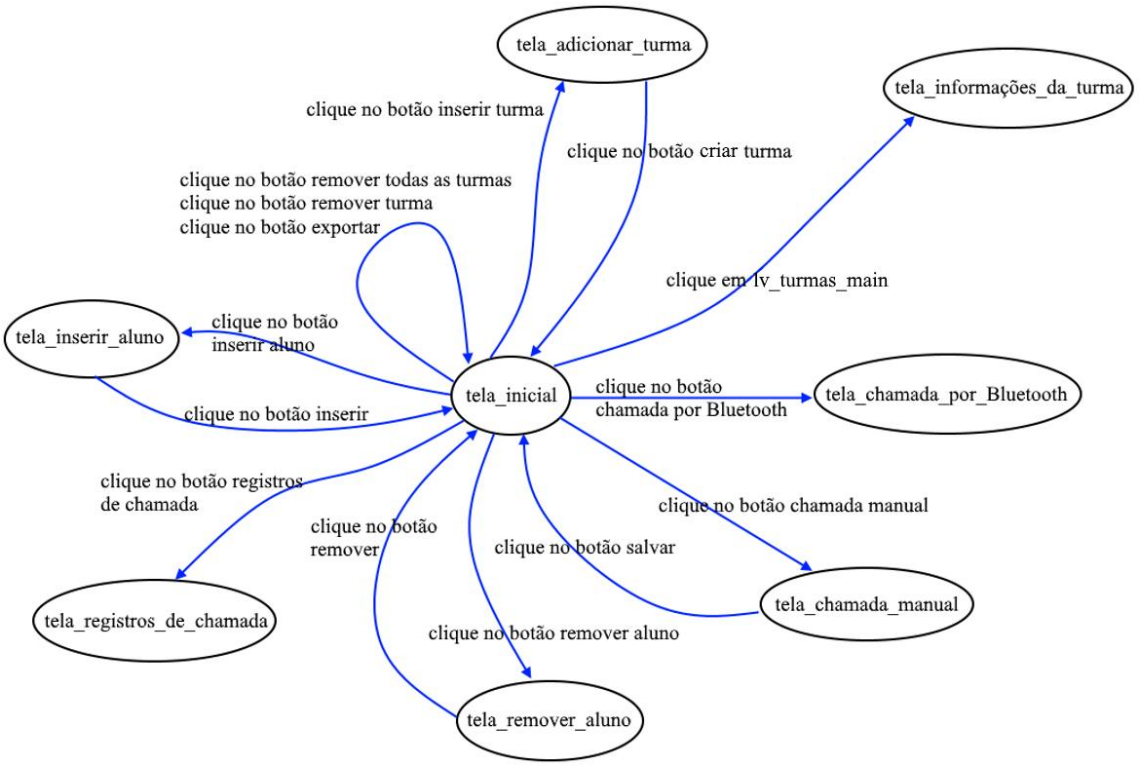
Fonte: Elaborado pelo autor.

No grafo, os arcos azuis representam os eventos exercitados sobre a interface da aplicação, enquanto que os arcos vermelhos, apontando para o nodo “Alerta”, representam os testes faltantes.

Como mencionado anteriormente, a aplicação possui quatorze possíveis transições de estado, das quais apenas cinco foram exercitadas por meio dos cenários utilizados neste exemplo. No total, nove componentes foram identificados como não testados ou não exercitados pelos cenários. Por meio da análise do grafo, o testador pode identificar facilmente quais elementos não foram testados e a quais telas eles pertencem. Desta forma, fica fácil para o testador completar os cenários de forma a estimular todos os componentes da interface.

Nos modos de operação 2 e 3, a ferramenta produziu um conjunto de cenários parciais referentes aos elementos que foram identificados como não testados. O conjunto gerado pelo uso do ABTT no modo de operação 2 (Figura 4.5) é apresentado no Anexo A. O testador pode utilizar o modelo gerado pela ferramenta como um ponto de partida para o processo de escrita dos novos testes do sistema. Completando o arquivo de cenários parciais e aplicando-o como entrada para a ferramenta, obteve-se o gráfico da Figura 4.8, que mostra as catorze possíveis interações com os elementos da interface que foram exercitados pelos cenários fornecidos.

Figura 4.8 – Grafo completo do aplicativo BlueTApp.



Fonte: Elaborado pelo autor.

5 ESTUDOS DE CASO

Um dos principais problemas enfrentados neste trabalho foi a escolha das aplicações que serviriam como caso de testes para a validação da ferramenta proposta. A dificuldade se deve principalmente à complexidade das aplicações disponíveis para *download* (que são muito simples ou muito complexas). Se uma aplicação muito simples fosse escolhida, seria difícil mostrar o potencial envolvido na utilização da ferramenta e os benefícios proporcionados por ela. Por outro lado, se uma aplicação muito complexa fosse utilizada, seria difícil entender o funcionamento da mesma e identificar se as saídas geradas pela ferramenta eram representativas.

A comunicação com os desenvolvedores das aplicações também foi um desafio. Seria desejável ter um olhar crítico sobre a aplicação escolhida, seja por parte do desenvolvedor ou de alguém que possuísse o conhecimento técnico sobre o funcionamento da aplicação como um todo, para confirmar se as saídas geradas pela ferramenta eram ou não fidedignas.

Outra dificuldade encontrada foi a identificação de aplicações desenvolvidas segundo a metodologia proposta pelo BDD, uma vez que não há indícios de aplicações, dentre as disponíveis para *download*, que tenham seguido tal metodologia. Isso, porém, não chega a ser um limitante para o uso da ferramenta, uma vez que é possível a geração automática dos cenários através da utilização exclusiva dos arquivos de leiaute da aplicação.

Dentro dessas condições, escolheu-se quatro aplicativos para serem utilizados como estudos de caso para a ferramenta. O primeiro, chamado Leish Heal, foi desenvolvido pelo autor seguindo a metodologia do BDD. O segundo, chamado Nutri Bovinos, foi escolhido pela possibilidade de contato com um dos seus desenvolvedores e também por apresentar uma complexidade razoável. O terceiro, chamado Controle de Saúde, não foi desenvolvido segundo a metodologia do BDD e não permitia contato com os desenvolvedores. Porém se encaixa no perfil de complexidade desejado. Foi escolhido ainda um aplicativo híbrido (que não é alvo do nosso estudo) de forma a validar o comportamento da ferramenta diante dessa situação.

A seguir são apresentados os aplicativos utilizados como estudo de caso bem como os resultados obtidos a partir da utilização da ferramenta ABTT.

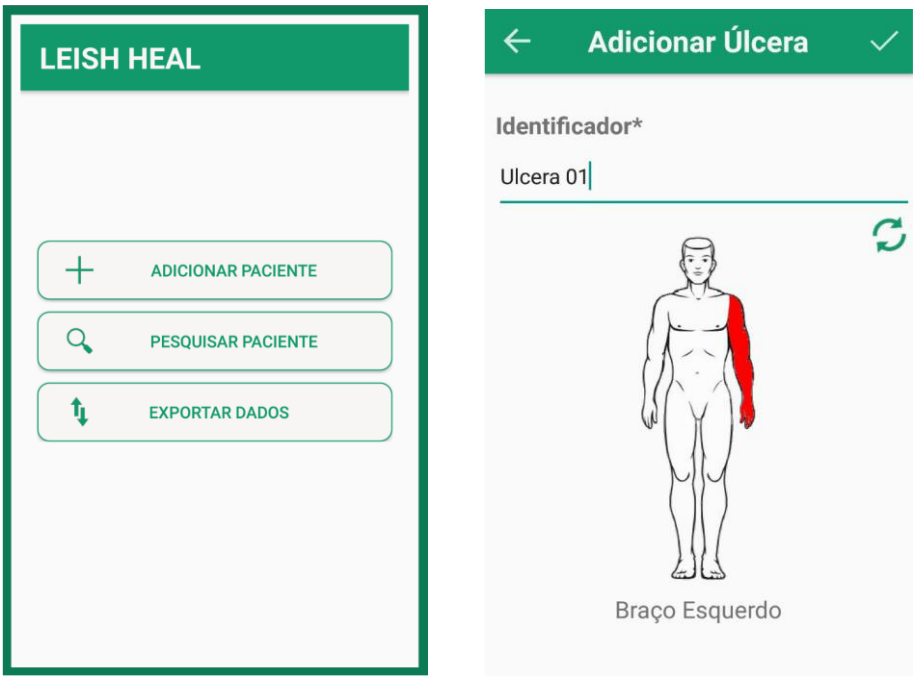
5.1 LEISH HEAL

Esse aplicativo foi desenvolvido segundo a metodologia do BDD em uma parceria entre a UFRGS e a Fiocruz-MG. O Leish Heal é um aplicativo criado com o objetivo de auxiliar médicos, enfermeiros e profissionais da área da saúde que tratam de pessoas diagnosticadas com Leishmaniose Tegumentar.

A Leishmaniose Tegumentar é uma infecção que produz lesões na superfície da pele dos enfermos, fazendo com que esses pacientes necessitem de doses periódicas de medicamento para o tratamento das lesões. O Leish Heal é uma ferramenta que visa facilitar a aplicação do tratamento da doença, que é feito de forma presencial através da visita dos profissionais da saúde aos enfermos. Nessas visitas costuma-se coletar dados sobre as lesões, além de aplicar-lhes as doses necessárias de medicamento por meio de injeções nas extremidades das feridas.

O Leish Heal conta com diversas funcionalidades que visam facilitar o trabalho desses profissionais, como cálculo da área da lesão, cálculo do volume e do limite diário de medicamento que um paciente pode receber, entre outras. Além disso o aplicativo serve ainda como ferramenta de organização, armazenamento, visualização e exportação dos dados coletados. O Leish Heal não está disponível para *download*, pois encontra-se em processo de verificação e homologação pela Fiocruz-MG. A Figura 5.1 apresenta algumas telas do aplicativo.

Figura 5.1 – Telas do aplicativo Leish Heal.



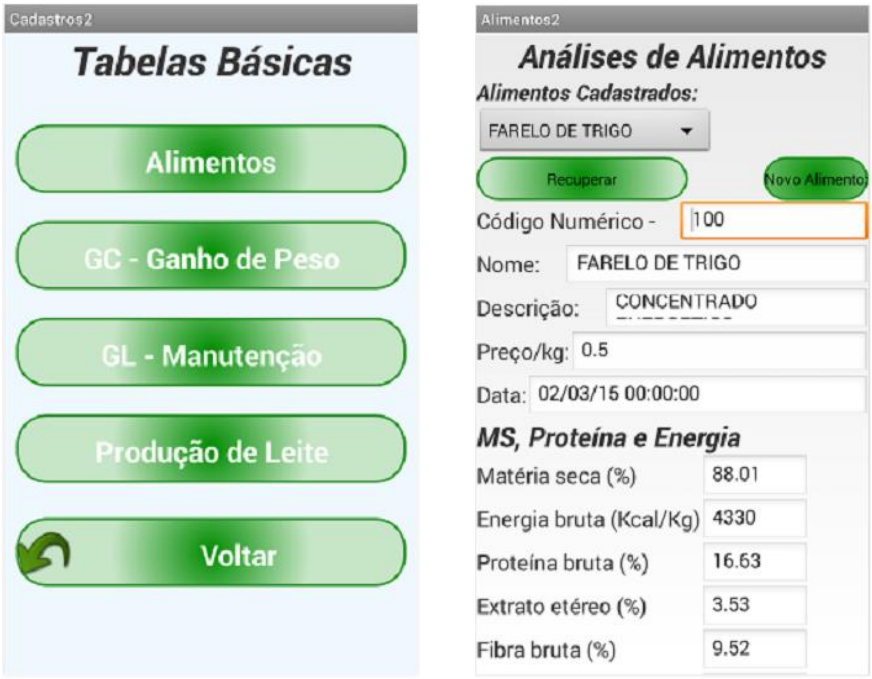
Fonte: Elaborado pelo autor.

5.2 NUTRIBOVINOS

O NutriBovinos não foi desenvolvido segundo a metodologia do BDD. Porém, foi escolhido devido a simplicidade de funcionamento e a possiblidade de contato com um de seus desenvolvedores, que serviu como especialista para afirmar se as saídas da ferramenta estavam corretas ou não.

O aplicativo encontra-se disponível para *download* na *Play Store*[1] e tem por objetivo auxiliar na gestão de rebanhos de corte e leite. Isso é feito através da coleta e manipulação de dados, considerados importantes para a produção, por meio de diferentes tipos de registros, como de alimentos, rações, necessidades nutricionais, entre outros (C7 NutriBovinos CL, 2015). A Figura 5.2 apresenta algumas telas do aplicativo.

Figura 5.2 – Telas do aplicativo NutriBovinos.



Fonte: (C7 NutriBovinos CL, 2015).

5.3 CONTROLE DE SAÚDE

O aplicativo controle de saúde foi escolhido como opção de estudo de caso por se encaixar no perfil de complexidade esperado. O aplicativo tem o objetivo de auxiliar médicos e pacientes a acompanhar e controlar algumas medidas importantes, como: glicose, pressão arterial, batimento cardíaco, peso, altura e IMC (Controle de Saúde, 2014). O aplicativo

[1] https://play.google.com/store/apps/details?id=com.c7nutribovinos&hl

encontra-se disponível para *download* na *Play Store*[2]. A Figura 5.3 apresenta algumas telas do aplicativo.

Figura 5.3 – Telas do aplicativo Controle de Saúde.

Fonte: (Controle de Saúde, 2014).

5.4 WordPress

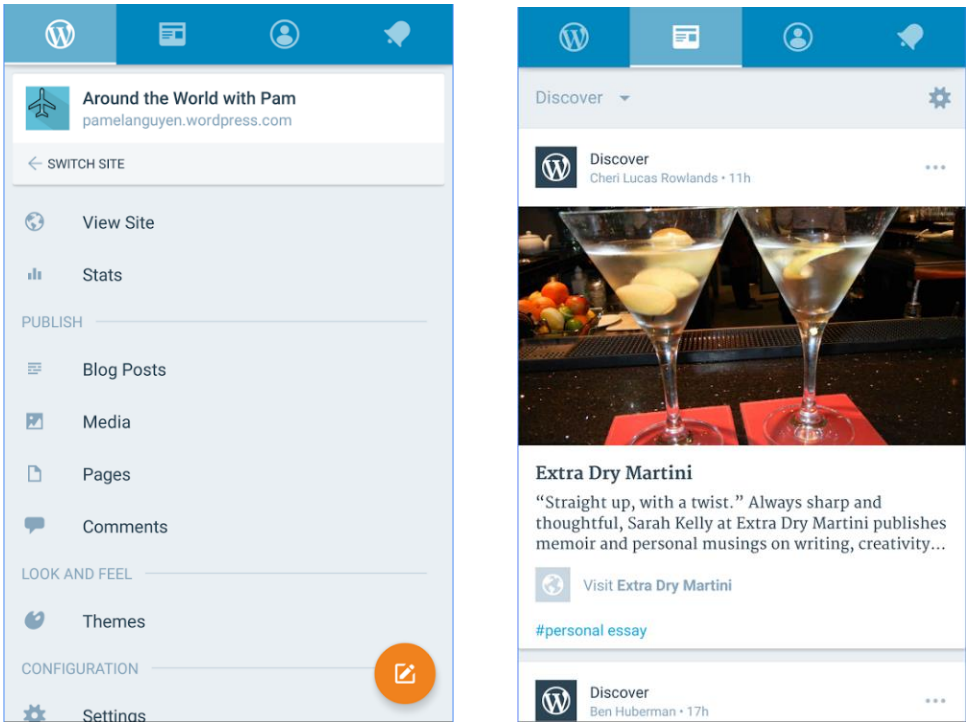
O aplicativo WordPress foi escolhido como estudo de caso por dois motivos principais: i) por ser um aplicativo híbrido e; ii) por ser mais complexo que os aplicativos apresentados anteriormente. Apesar da ferramenta proposta ter sido desenvolvida para aplicações nativas do Android, o WordPress foi utilizado como artefato de teste para identificar o comportamento da ferramenta quando interage com uma aplicação desse tipo e, dessa forma, descobrir as limitações da ferramenta e pensar em soluções de melhoria.

O WordPress é uma plataforma livre, de código aberto, desenvolvida em Java e JavaScript e utilizada para a publicação de conteúdo pessoal na web com foco em estética e usabilidade. Através de um blog é possível gerenciar as informações publicadas bem como as

[2] https://play.google.com/store/apps/details?id=br.com.matheuspiscioneri.controledesaude&hl=pt_BR

interações dos leitores com o blog. O aplicativo encontra-se disponível para *download* na *Play Store*[3]. A Figura 5.4 mostra algumas telas do aplicativo.

Figura 5.4 – Telas do aplicativo WordPress.



Fonte: (WordPress, 2017).

5.5 RESULTADOS

5.5.1 Leish Heal

O aplicativo Leish Heal foi desenvolvido segundo a metodologia de desenvolvimento do BDD, ou seja, os testes foram escritos antes do código funcional da aplicação, e conta com um conjunto de 75 cenários baseados em UI. O aplicativo é composto por 13 *activities*, que podem interagir entre si, formando um conjunto de 17 possíveis transições.

Para a validação do aplicativo Leish Heal utilizou-se como parâmetro de entrada para a ferramenta os cenários do BDD em conjunto com os arquivos de leiaute do aplicativo (modo de operação 2), que produziu um grafo com 13 *activities* e 16 transições. A ferramenta detectou que um elemento do leiaute não havia sido exercitado pelos cenários, o que ocasionou a descoberta de um bug na aplicação. Através da análise do grafo, foi detectada

[3] https://play.google.com/store/apps/details?id=org.wordpress.android&hl=pt_BR

ainda uma inconsistência nos cenários, de forma que outro bug foi detectado. Essa inconsistência era referente a um clique em um botão após a exclusão de um item em uma lista. Dessa maneira, dois bugs, que passaram desapercebidos durante o processo de desenvolvimento, foram detectados, dando origem a dois novos cenários. O grafo que identificou os erros pode ser encontrado na Figura 1 B em anexo.

Também se confirmou com os desenvolvedores do aplicativo que a ferramenta foi capaz de identificar corretamente todas as *activities* do aplicativo, bem como as interações entre elas. Além disso, após os cenários serem completados com os dois testes faltantes, a ferramenta confirmou que, através dos cenários, todos os componentes do leiaute foram testados, atingindo uma cobertura total dos elementos do sistema.

O grafo completo do aplicativo Leish Heal está disponível na Figura 2 B em anexo.

5.5.2 NutriBovinos

O aplicativo NutriBovinos não foi desenvolvido segundo a metodologia de desenvolvimento do BDD. Isso não chega a ser uma limitação para a ferramenta, que pode ser utilizada no modo de operação 3, somente com os arquivos de leiaute. Como o NutriBovinos não possuía nenhum cenário de teste escrito, para começar sua validação utilizou-se ferramenta no modo de operação 3. Dessa maneira obteve-se um total de 52 cenários parciais. Além disso, foram identificadas 13 *activities* através do grafo gerado.

De forma a completar os cenários parciais, navegou-se manualmente através da aplicação, a fim de descobrir qual o comportamento do aplicativo diante das interações do usuário com os elementos do leiaute. Após os cenários parciais serem completados, atingiu-se um total de 57 cenários (telas com entradas de textos e botões dão origem a 2 casos de teste: um clicando no botão antes do preenchimento do campo de texto e outro após). Utilizou-se então o arquivo de cenários como entrada para a ferramenta, juntamente com os arquivos de leiaute (modo de operação 2), a fim de gerar o grafo comportamental da aplicação, e verificar se faltou testar algum dos elementos do leiaute. Para garantir que, tanto os cenários completos quanto o grafo gerado pela ferramenta, estavam corretos, os dois foram apresentados a um dos desenvolvedores do aplicativo, que confirmou a veracidade das informações apresentadas em ambos, bem como o número de *activities* e transições.

Através dos testes realizados, foram detectados 5 bugs na aplicação, todos referentes a cliques em botões de submissão de registros. Os erros aconteciam quando os registros eram submetidos contendo algum dos campos incompletos e levavam a aplicação a parar de

funcionar. Como os cenários foram construídos pela ferramenta, a partir da análise sintática dos arquivos de leiaute da aplicação, todos os componentes da interface foram testados, atingindo assim uma cobertura de 100% dos elementos responsáveis pelas transições de estado entre as *activities*. O grafo completo do funcionamento do aplicativo NutriBovinos, gerado pela ferramenta, pode ser visto na Figura 1 C em anexo.

5.5.3 Controle de Saúde

Assim como o aplicativo NutriBovinos, o aplicativo Controle de Saúde também não foi desenvolvido segundo a metodologia do BDD e, por isso, não possuía nenhum cenário. Sendo assim, iniciou-se sua validação com a utilização da ferramenta no modo de operação 3 (somente arquivos de leiaute), que gerou o grafo parcial da aplicação e um total de 51 cenários parciais. Através da análise do grafo gerado foram identificadas ainda 30 *activities*.

De maneira a identificar o comportamento do aplicativo e completar os cenários parciais, navegou-se manualmente através das telas do sistema, e percebeu-se a presença de um menu genérico em várias dessas telas (Figura 5.3). Analisando os arquivos de leiaute, concluiu-se que o menu era instanciado em tempo de execução, pois não existia nenhum arquivo de leiaute referente a esse elemento. De forma a não comprometer os testes, o menu foi reproduzido por meio de um arquivo XML e novamente a ferramenta foi utilizada no modo de operação 3. Dessa vez, foi gerado um total de 147 cenários e um grafo com 30 *activities*. Esse aumento significativo na identificação de elementos se deve ao fato de muitas das telas do aplicativo possuírem somente esse menu como elemento de interação com o usuário. Além disso, cada opção do menu leva a aplicação a um novo estado, onde o menu também aparece, aumentando assim rapidamente o número de cenários.

Por fim, navegou-se novamente entre as telas do aplicativo, de forma a completar os cenários parciais. Uma vez completados, obteve-se um total de 157 cenários (assim como no NutriBovinos, cenários extras apareceram devido à presença de botões e campos de texto na mesma tela) e através da execução manual dos testes foram identificados 10 bugs na aplicação. Todos os bugs encontrados são referentes a cliques em botões de salvamento de informações, porém nenhum deles levou o aplicativo a parar de funcionar. Os bugs aconteciam diante da tentativa de salvar informações incompletas ou com campos faltantes. Nessas situações uma mensagem de erro deveria aparecer, porém era mostrado ao usuário um quadrado branco sem nenhuma informação. Através dos testes executados, a partir dos cenários, atingiu-se uma cobertura de teste de 100% dos elementos da interface do aplicativo.

A Tabela 5.1 traz um comparativo dos resultados obtidos através do uso da ferramenta na atividade de teste dos aplicativos apresentados.

Tabela 5.1 – Resultados obtidos.

	Leish Heal	NutriBovinos	Controle de Saúde
Desenvolvido com BDD	Sim	Não	Não
Activities	13	13	39
Activities identificadas	13	13	30
Transições	17	24	?
Transições identificadas	16	24	91
Bugs Identificados	2	5	10
Crash	Sim	Sim	Não
Total de cenários	77	57	157
Cobertura	100%	100%	100%

Fonte: Elaborado pelo autor.

Através da análise da tabela nota-se que em todos os estudos de caso foram descobertos bugs, confirmando a utilidade da abordagem proposta neste trabalho. Além disso, em todas as aplicações, uma cobertura de 100% dos elementos do leiaute foi alcançada. Vale destacar ainda que o número total de transições do aplicativo Controle de Saúde é desconhecido (por isso aparece na tabela representado por um ponto de interrogação). Acredita-se que as transições identificadas com o uso da ferramenta correspondem ao número total de transições, porém não se pode afirmar com certeza, uma vez que a aplicação não possuía cenários e também não foi possível o contato com seus desenvolvedores.

5.5.4 WordPress

O teste do aplicativo WordPress, iniciou-se com a utilização da ferramenta no modo de operação 3, uma vez que o aplicativo não foi desenvolvido segundo a metodologia do BDD e por isso não possuía nenhum cenário. Através do grafo gerado pela ferramenta, verificou-se que a grande maioria dos elementos do leiaute não foram identificados.

Uma vez que o aplicativo é de código aberto, iniciou-se então uma inspeção manual dos arquivos da aplicação, tanto dos arquivos de código fonte quanto dos arquivos de leiaute, a fim de descobrir o motivo de tão poucos elementos terem sido identificados. A partir dessa análise foram identificadas algumas limitações da ABTT:

1) Identificou-se a presença do elemento *web view*, discutido na seção 2.3, utilizado para renderizar o blog e as informações postadas. O blog criado pelo usuário, nada mais é que uma página web sitiada em um domínio do WordPress, ou seja, os elementos recuperados pelo *web view*, que funciona como uma espécie de navegador, devem ser acessados através da utilização de tags HTML.

2) O aplicativo WordPress, por concentrar esforços no projeto da interface, utiliza muitos elementos personalizados que não levam às tags tradicionais do Android como elemento de identificação. Esses elementos, por serem criados pelos desenvolvedores do aplicativo, possuem um arquivo XML de leiaute próprio e são carregados dentro do arquivo de leiaute tradicional do Android através de uma tag *include* ou do *path* do arquivo XML dentro do projeto, referente ao elemento customizado. Sendo assim, uma análise da estrutura de diretórios do projeto precisa ser realizada a fim de descobrir os *paths* desses arquivos customizados.

3) Além do uso de elementos de leiaute customizados, o WordPress conta ainda com a presença de uma grande variedade de *widgets* em sua interface. Um *widget* é um pacote de elementos gráficos, agrupados de tal maneira que formam um novo elemento gráfico. Por exemplo, uma *action bar*, um conjunto de abas, são exemplos de *widgets*. Nesse caso, uma inspeção detalhada desses elementos precisa ser realizada, de forma a identificar quais elementos gráficos compõem o *widget*.

4) O WordPress possui ainda alguns elementos instanciados em tempo de execução, ou seja, são elementos que não aparecem nos arquivos de leiaute da aplicação, somente no código fonte, onde são criados e gerenciados.

Como a ABTT não lida ainda com esses tipos de situações, ela não gerou uma boa resposta quando utilizada no processo de teste do WordPress. De qualquer forma, essas descobertas servem como diretriz e motivação para a implementação de novas funcionalidades e continuação do trabalho.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho propôs uma abordagem baseada em BDD para ajudar na definição de testes de sistema efetivos para aplicativos nativos do Android. A abordagem é suportada por uma ferramenta chamada *Android Behavior Testing Tool* (ABTT). A ferramenta pode operar de três maneira distintas de acordo com os arquivos fornecidos como parâmetros de entrada. No modo de operação 1, a ferramenta interpreta os cenários do BDD (baseados em UI) fornecendo uma visão geral do comportamento da aplicação ao testador, facilitando assim a tarefa de julgamento da completude dos cenários propostos. No modo de operação 2, a ferramenta permite, por meio de análise dos cenários em conjunto com os arquivos de leiaute da aplicação, a detecção dos elementos da interface de usuário que não foram exercitados pelos cenários (testes faltantes). Nesse caso, gera-se um modelo parcial de cenário, no formato do BDD, representando os novos casos de teste. No modo de operação 3, a ferramenta analisa exclusivamente os arquivos de leiaute da aplicação e, gera um conjunto de cenários parciais que devem ser posteriormente completados pelo testador. Esse modo de operação automatiza parcialmente a tarefa de escrita dos cenários e serve como um ponto de partida para a criação de novos testes.

A ferramenta foi validada através de três aplicativos nativos do Android que serviram como estudos de caso. Com o uso da ferramenta e, através da execução manual dos cenários gerados, foram detectados 17 bugs no total, que confirmaram a utilidade e a importância de uma ferramenta de auxílio à fase de testes. Além disso, um aplicativo híbrido, que não é o alvo da abordagem proposta, foi utilizado como estudo de caso com o objetivo de levantar dados sobre as limitações da ferramenta. Através desse estudo de caso algumas descobertas foram feitas e limitações foram encontradas. Atualmente a abordagem proposta não identifica elementos criados em tempo de execução, elementos customizados e nem objetos carregados a partir do elemento *web view*.

Como trabalhos futuros pretende-se aumentar o número de funcionalidades da ferramenta e, inspirados no estudo realizado por (PHAM e NGOC-HUNG, 2013), expandir a análise para o código fonte das aplicações. Dessa forma, espera-se identificar as transições de tela sem a necessidade do uso dos cenários do BDD, utilizando-os somente como instrumento de medição da integridade dos testes escritos. Além disso, com a análise do código fonte, espera-se identificar os elementos do leiaute instanciados em tempo de execução e automatizar completamente a tarefa de escrita dos cenários.

Além disso, deseja-se ainda expandir o número de elementos gráficos do Android que são identificados pela ferramenta. Para isso, uma análise da estrutura de diretórios do projeto será realizada, a fim de identificar as tags (*paths*) utilizadas para a declaração dos elementos customizados. Uma análise dos *includes* e *widgets* utilizados nos arquivos de leiaute também será realizada, de forma que aplicativos híbridos também possam ser englobados pelas facilidades oferecidas pela nossa ferramenta.

Dessa maneira, espera-se minimizar o número de limitações da ferramenta para ajudar ainda mais o testador na atividade de desenvolvimento de cenários baseados em UI, aumentando a automatização da escrita e geração de novos casos de teste em aplicações Android.

REFERÊNCIAS

AMALFITANO, D. et al. **MobiGUITAR - A Tool for Automated Model-Based Testing of Mobile Apps**. [S.l.]: [s.n.]. 2015.

AMALFITANO, D.; FASOLINO, A. R.; TRAMONTANA, P. **A GUI Crawling-based technique for Android Mobile Application Testing**. Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. [S.l.]: IEEE. 2011. p. 252-261.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. New York: Cambridge University Press, 2008.

ANDROID Developers. **Activity**, 2016. Disponivel em: <https://developer.android.com/reference/android/app/Activity.html>. Acesso em: 14 Dezembro 2016.

ANDROID Developers. **Componentes de Aplicativos**, 2016. Disponivel em: <https://developer.android.com/guide/components/activities.html>. Acesso em: 14 Dezembro 2016.

ANDROID Developers. **Training**, 2016. Disponivel em: <https://developer.android.com/training/basics/activity-lifecycle/starting.html>. Acesso em: 14 Dezembro 2016.

ANDROID Developers. **Interface do Usuário**, 2016. Disponivel em: <https://developer.android.com/guide/topics/ui/declaring-layout.html>. Acesso em: 14 Dezembro 2016.

APKPURE. **ApkPure**, 2017. Disponivel em: <https://apkpure.com/>. Acesso em: 14 Fevereiro 2017.

APKTOOL. **ibootpeaches**, 2017. Disponivel em: <https://ibotpeaches.github.io/Apktool/>. Acesso em: 14 Fevereiro 2017.

APPIUM.IO. **Appium:** Mobile App Automation Made Awesome, 2017. Disponivel em: <http://appium.io/>. Acesso em: 13 Janeiro 2017.

C7 NutriBovinos CL. **Google Play**, 2015. Disponivel em: <https://play.google.com/store/apps/details?id=com.c7nutribovinos&hl>. Acesso em: 27 nov. 2016.

CHAN, P. P. F.; HUI, L. C. K.; YIU, S. **DroidChecker:** Analyzing Android Applications for Capability Leak. Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. Tucson: Association for Computing Machinery. 2012. p. 125-136.

COHN, M. **User Stories Applied For Agile Software Development**. [S.l.]: Addison-Wesley Professional, 2004.

CONTROLE de Saúde. **Google Play**, 2014. Disponivel em: <https://play.google.com/store/apps/details?id=br.com.matheuspiscioneri.controledesaude&hl=pt_BR>. Acesso em: 16 Fevereiro 2017.

COWAN, A. Cowan+. **Your Best Agile User Story**, 2016. Disponivel em: <http://www.alexandercowan.com/best-agile-user-story/>. Acesso em: 06 Setembro 2016.

CUCUMBER. **Cucumber Limited**, 2017. Disponivel em: <https://cucumber.io/>. Acesso em: 13 Janeiro 2017.

DARY, D. **Selendroid:** Selenium for Android, 2017. Disponivel em: <http://selendroid.io/>. Acesso em: 13 Janeiro 2017.

DELAMARO, M. E.; VICENZI, A. M. R.; MALDONADO, J. C. **A Strategy to Perform Coverage Testing of Mobile Applications**. Proceedings of the 2006 International Workshop on Automation of Software Test. Shanghai: Association for Computing Machinery. 2006. p. 118-124.

ESPRESSO. **Android Testing Support Library**, 2017. Disponivel em: <https://google.github.io/android-testing-support-library/docs/espresso/>. Acesso em: 13 Janeiro 2017.

EVANS, E. **Domain-Driven Design:** Tackling Complexity in the Heart of Software. [S.l.]: Addison Wesley, 2003.

GRAPHVIZ. **Graphviz - Graph Visualization Software**, 2017. Disponivel em: <http://www.graphviz.org/>. Acesso em: 13 Janeiro 2017.

HANSSON, A. Gherkin. **GitHub**, 2017. Disponivel em: <https://github.com/cucumber/cucumber/wiki/Gherkin>. Acesso em: 04 Abril 2017.

IDC - Analyze the Future. **Smartphone OS Market Share**, 2016. Disponivel em: <http://www.idc.com/promo/smartphone-market-share/os;jsessionid=F26A24E3BD6523C2DEC2094174B86799>. Acesso em: 27 Janeiro 2017.

JBEHAVE. **What is JBehave?**, 2017. Disponivel em: <http://jbehave.org/>. Acesso em: 13 Janeiro 2017.

JOORABCHI, M. E.; MESBAH, A.; KRUCHTEN, P. **Real Challenges in Mobile App Development**. 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. Baltimore: IEEE. 2013.

KAASILA, J. et al. **Testdroid:** automated remote UI testing on Android. Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia. Ulm: Association of Compurter Machinery. 2012. p. 1-4.

KISS, F. YMC. **The top 5 challenges of BDD with Behat/Mink – Best practices needed**, 2013. Disponivel em: <http://www.ymc.ch/de/blog/the-top-5-challenges-of-bdd-with-behatmink-best-practices-needed/>. Acesso em: 02 Janeiro 2017.

KOSCIANSKI, A.; SOARES, M. D. S. **Qualidade de Software - Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software**. 2nd. ed. [S.l.]: Novatec Editora, 2007. 17-42 p.

KUDRYASHOV, K. Inviqa. **The beginner's guide to BDD**, 2015. Disponivel em: <https://inviqa.com/blog/bdd-guide>. Acesso em: 02 Janeiro 2017.

LAPOLLI, F. R. et al. Modelo de Desenvolvimento de Objetos de Aprendizagem Baseado em Metodologias Ágeis e Scaffoldings. **Revista Brasileira de Informática na Educação**, v. 18, p. 19-32, 2010.

LIMA, V. Nativo x Híbrido - A Discussão Final (parte I). **Concrete Solutions**, 2016. Disponivel em: <http://www.concretesolutions.com.br/2016/03/16/nativo-x-hibrido/>. Acesso em: 13 Fevereiro 2017.

MACHIRY, A.; TAHILIANI, R.; NAIK, M. **Dynodroid:** An Input Generation System for Android Apps. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. Saint Petersburg: Association for Computing Machinery. 2013. p. 224-234.

MARTINS, R. J. W. D. A. **Desenvolvimento de Aplicativo para Smartphone com a Plataforma Android**. Pontifícia Universidade Católica do Rio de Janeiro. Rio de Janeiro, p. 50. 2009.

MATTÉ, M. A. **Testes de Software: Uma Abordagem da Atividade de Teste de Software em Metodologias Ágeis Aplicando a Técnica Behavior Driven Development em um Estudo Experimental**. Universidade Tecnológica Federal do Paraná. Medianeira, p. 54. 2011.

MOUNTAIN Goat Software. **User Stories**, 2016. Disponivel em: <https://www.mountaingoatsoftware.com/agile/user-stories>. Acesso em: 06 Setembro 2016.

MUCCINI, H.; DI FRANCESCO, A.; ESPOSITO, P. **Software Testing of Mobile Applications:** Challenges and Future Research Directions. Proceedings of the 7th International Workshop on Automation of Software Test. [S.l.]: IEEE. 2012. p. 29-35.

MYERS, G.; BADGETT, T.; SANDLER, C. **The Art of Software Testing**. 3rd. ed. [S.l.]: John Wiley & Sons, 2011.

NETO, A. C. D. Introdução a Teste de Software. **Engenharia de Software Magazine**, p. 54-59, 2007. Disponivel em: <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035>. Acesso em: Outubro 2015.

NORTH, D. Introducing BDD. **Better Software magazine**, March 2006.

NORTH, D. What's in a Story? **Dan North And Associates**, 2007. Disponivel em: <https://dannorth.net/whats-in-a-story/>. Acesso em: Setembro 2016.

OLIVEIRA, B. Qual a diferença entre Aplicativo Nativo, Mobile Web App e Aplicativo Híbrido. **Mestre do Marketing.com**, 2016. Disponivel em: <http://www.mestredomarketing.com/qual-diferenca-entre-aplicativo-nativo-mobile-web-app-e-aplicativo-hibrido/>. Acesso em: 13 Fevereiro 2017.

ORACLE. **Oracle and CloudMonkey**, 2017. Disponivel em: <https://www.oracle.com/corporate/acquisitions/cloudmonkey/index.html>. Acesso em: 13 Janeiro 2017.

PERFECTO Mobile. **Perform Mobile and Selenium Testing On Real Devices with Perfecto**, 2017. Disponivel em: <https://www.perfectomobile.com/>. Acesso em: 13 Janeiro 2017.

PHAM, V.-C.; NGOC-HUNG, P. **A Method and Tool Support for Automated Data Flow Testing of Java Programs**. Proceedings of International Conference on Context-Aware Systems and Applications. Phu Quoc: Springer International Publishing. 2013. p. 157-167.

PIRES DE SOUZA, ; GASPAROTTO, A. M. S. **A Importância da Atividade de Teste no Desenvolvimento de Software**. XXXIII Encontro Nacional de Engenharia de Produção. Salvador: [s.n.]. 2013. p. 17.

PRESSMAN, R. S. **Software Engineering:** A Practitioner Approach. 5th. ed. New York: McGraw-Hill, 2005. 888 p.

ROBOTIUM Tech. **Robotium**, 2017. Disponivel em: <http://robotium.com/>. Acesso em: 13 Janeiro 2017.

SANTOS, A.; CORREIA, I. **Mobile Testing in Software Industry using Agile:** Challenges and Opportunities. Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). Graz: IEEE. 2015.

SATYANARAYANAN, M. **Fundamental Challenges in Mobile Computing**. Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing. Philadelphia: ACM. 1996. p. 1-7.

SMART, J. F. **BDD in Action**. 1st. ed. [S.l.]: Manning Publications, 2014.

SMARTFACE. **How to Increase Productivity and Reduce Costs in Native iOS and Android Development with Smartface**, 2016. Disponivel em: <https://www.smartface.io/increase-productivity-reduce-costs-native-ios-android-development-smartface/>. Acesso em: 14 Dezembro 2016.

START Another Activity. **Android Developers**, 2017. Disponivel em: <https://developer.android.com/training/basics/firstapp/starting-activity.html?hl=pt-br>. Acesso em: 13 Janeiro 2017.

STATISTA. **Global Mobile OS Market Share in Sales to End Users From 1st Quarter 2009 to 1st Quarter 2016**, 2016. Disponivel em: <https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/>. Acesso em: 13 Dezembro 2016.

TESTOBJECT. **Test Object – Android and iOS Mobile App Testing Made Easy**, 2017. Disponivel em: <https://testobject.com/>. Acesso em: 13 Janeiro 2017.

UIAUTOMATOR. **Android Testing Support Library**, 2017. Disponivel em: <https://google.github.io/android-testing-support-library/docs/uiautomator/>. Acesso em: 13 Janeiro 2017.

WASSERMAN, A. I. **Software Engineering Issues for Mobile Application Development**. Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. Santa Fé: ACM. 2010. p. 397-400.

WINK, A. S. **Android Platform Analysis from the Software Engineering Perpective**. 2016. 49f. Monografia (Especialização) - Curso de Engenharia da Computação, Insituto de Informática, UFRGS. Porto Alegre, 2016. Disponível em <http://hdl.handle.net/10183/147666>. Acesso em: 13 fev. 2017.

WORDPRESS. **Google Play**, 2017. Disponivel em: <https://play.google.com/store/apps/details?id=org.wordpress.android&hl=pt_BR>. Acesso em: 16 Fevereiro 2017.

WYNNE, ; HELLESOY, A. **The Cucumber Book:** Behaviour Driven Development for Testers and Developers. Dallas: The Pragmatic Bookshelf, 2012.

ZEIN, S.; SALLEH, N.; GRUNDY, J. A Systematic Mapping Study of Mobile Application Testing Techniques. **The Journal of Systems and Software**, p. 334-356, 2016.

ANEXO A – CENÁRIOS DO APLICATIVO BLUETAPP

Figura 1 A – Cenários do aplicativo BlueTApp após uso do ABTT no modo de operação 2.

Cenário 1: Clique no botão inserir turma.
Dado que estou na tela "inicial".
Quando clico no botão "inserir turma".
Então aparece a tela "adicionar turma".

Cenário 2: Clique no botão remover todas as turmas.
Dado que estou na tela "inicial".
Quando clico no botão "remover todas as turmas".
Então aparece uma mensagem "operação realizada com sucesso".

Cenário 3: Clique em alguma turma.
Dado que estou na tela "inicial".
Quando clico em "alguma turma".
Então deve aparecer a tela "informações da turma".

Cenário 4: Clique no botão chamada por Bluetooth.
Dado que estou na tela "inicial".
Quando clico no botão "chamada por Bluetooth".
Então aparece a tela "chamada por Bluetooth".

Cenário 5: Clique no botão chamada manual.
Dado que estou na tela "inicial".
Quando clico no botão "chamada manual".
Então deve aparecer a tela "chamada manual".

CENÁRIOS PARCIAIS
Cenário 6: Clique no botão registros de chamada.
Dado que estou na tela "inicial".
Quando clico no botão "registros de chamada".
Então aparece a tela "Alerta".

Cenário 7: Clique no botão registros exportar.
Dado que estou na tela "inicial".
Quando clico no botão "exportar".
Então aparece a tela "Alerta".

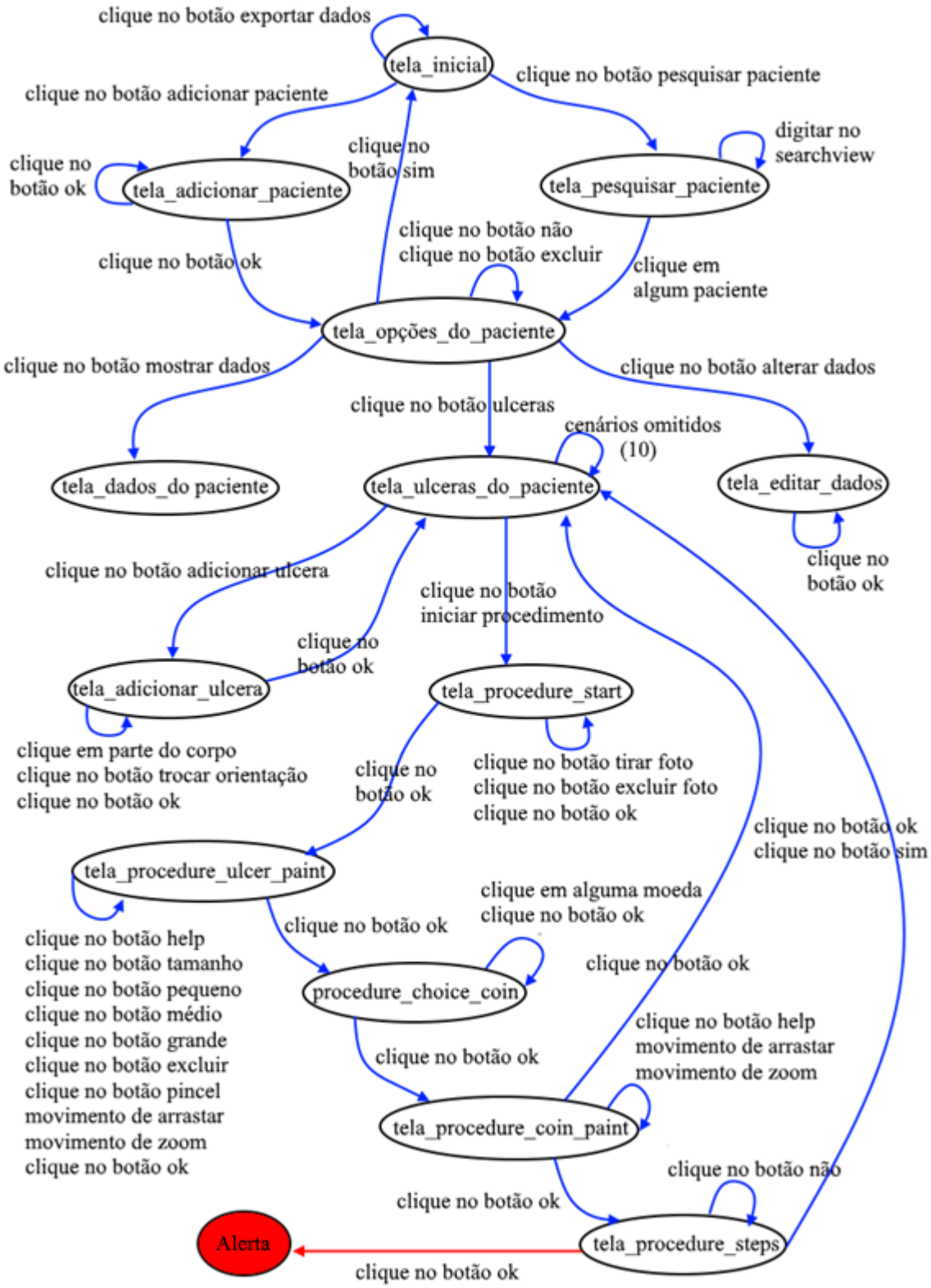
Cenário 8: Clique no botão inserir aluno.
Dado que estou na tela "inicial".
Quando clico no botão "inserir aluno".
Então aparece a tela "Alerta".

41
42 Cenário 9: Clique no botão inserir turma.
43 **Dado que** estou na tela “adicionar turma”.
44 **Quando** clico no botão “criar turma”.
45 **Então** aparece a tela “Alerta”.
46
47 Cenário 10: Clique no botão salvar.
48 **Dado que** estou na tela “chamada manual”.
49 **Quando** clico no botão “salvar”.
50 **Então** aparece a tela “Alerta”.
51
52 Cenário 11: Clique no botão remover aluno.
53 **Dado que** estou na tela “inicial”.
54 **Quando** clico no botão “remover aluno”.
55 **Então** aparece a tela “Alerta”.
56
57 Cenário 12: Clique no botão remover.
58 **Dado que** estou na tela “remover aluno”.
59 **Quando** clico no botão “remover”.
60 **Então** aparece a tela “Alerta”.

Fonte: Elaborado pelo autor.

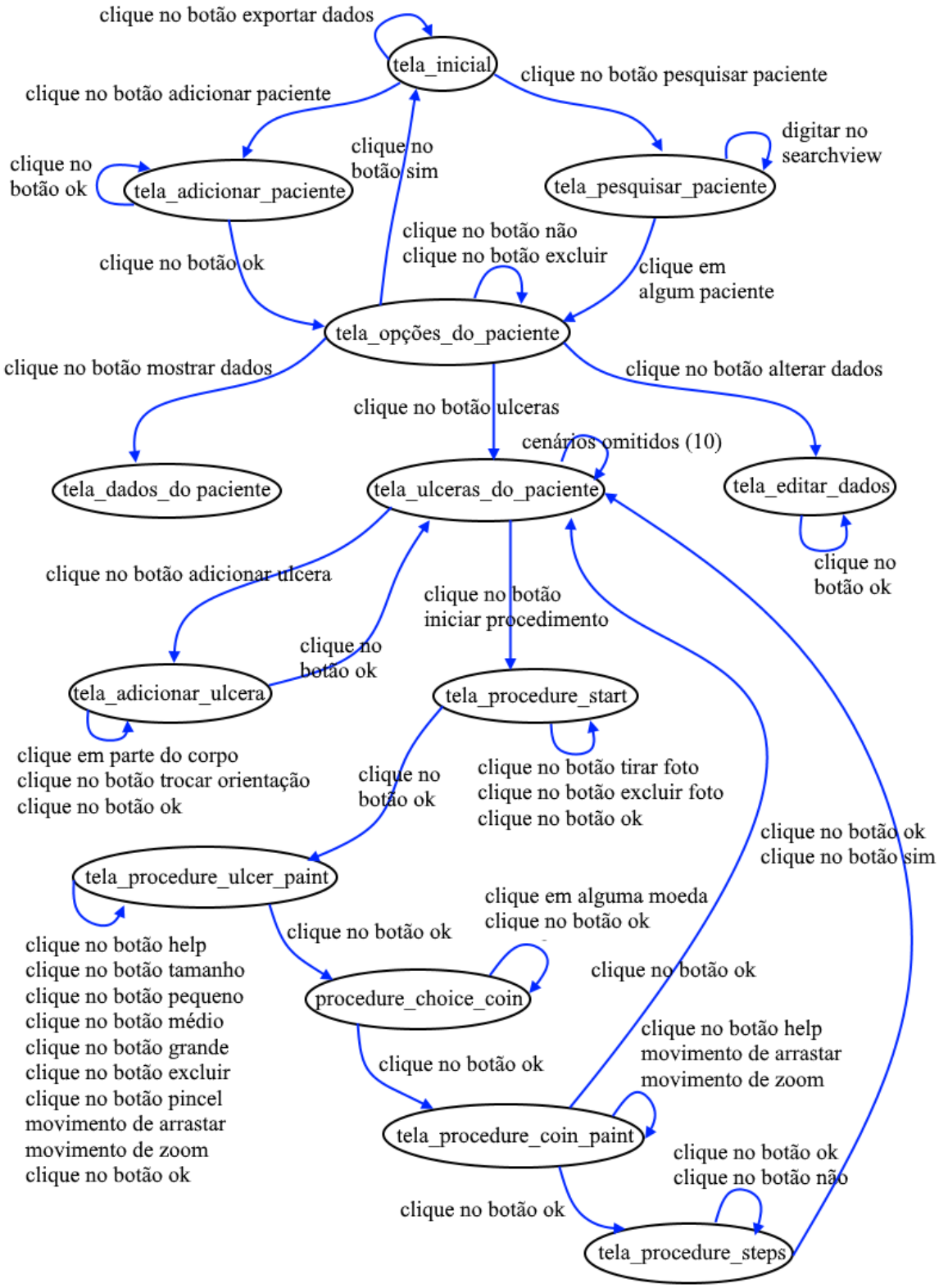
ANEXO B – GRAFOS DO APLICATIVO LEISH HEAL

Figura 1 B – Grafo do aplicativo Leish Heal que identificou um elemento não testado.



Fonte: Elaborado pelo autor.

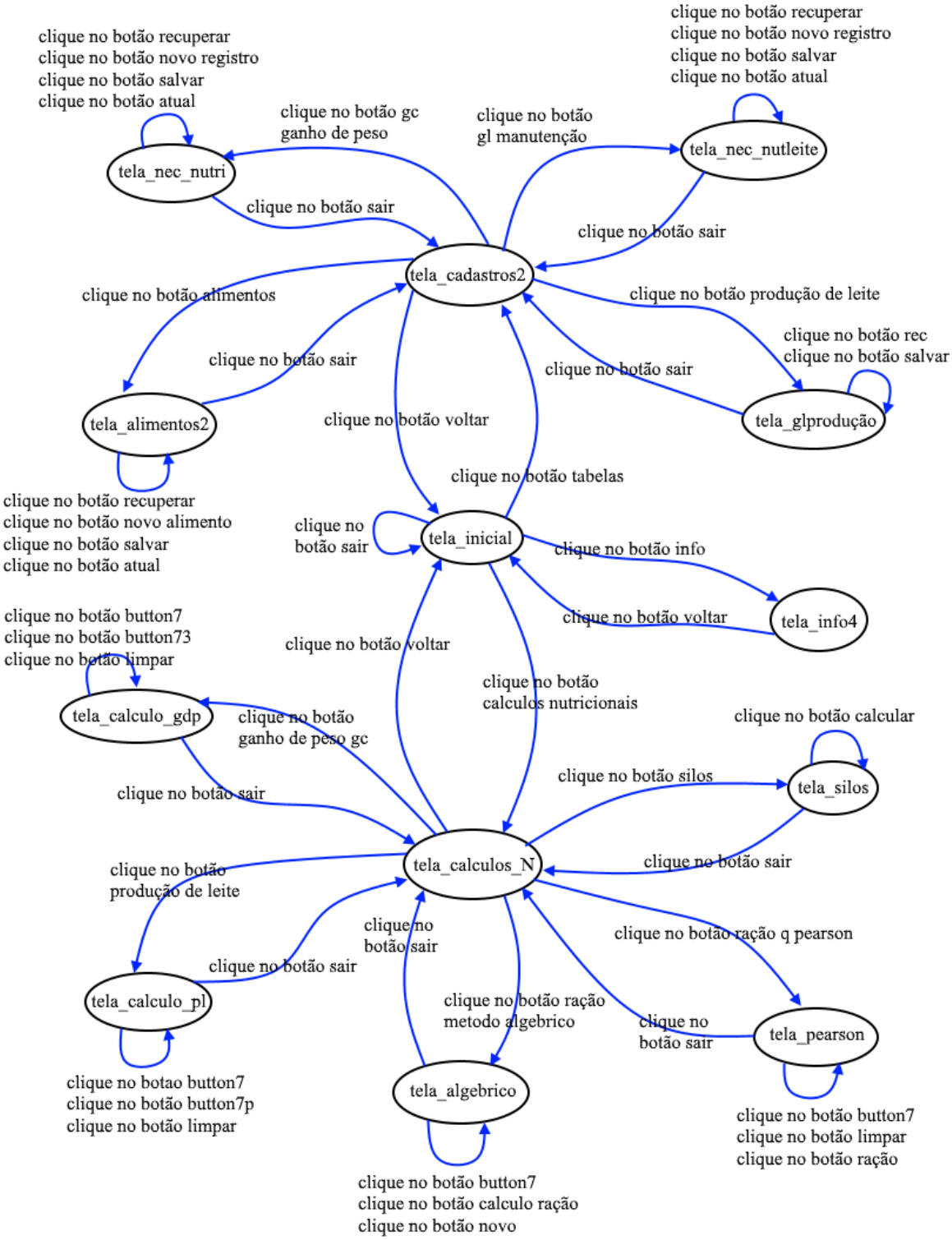
Figura 2 B – Grafo completo do aplicativo Leish Heal.



Fonte: Elaborado pelo autor.

ANEXO C – GRAFO DO APLICATIVO NUTRIBOVINOS

Figura 1 C – Grafo completo do aplicativo NutriBovinos.



Fonte: Elaborado pelo autor.