

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS CORNÉLIO PROCÓPIO
DIRETORIA DE GRADUAÇÃO E EDUCAÇÃO PROFISSIONAL
ENGENHARIA DE COMPUTAÇÃO

BRUNO MOURA CAVALCANTE

**APOIO A LOCALIZAÇÃO DE DEFEITOS DURANTE A EXECUÇÃO
DE SEQUÊNCIAS DE TESTE PARA APLICAÇÕES ANDROID**

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO

2015

BRUNO MOURA CAVALCANTE

**APOIO A LOCALIZAÇÃO DE DEFEITOS DURANTE A EXECUÇÃO
DE SEQUÊNCIAS DE TESTE PARA APLICAÇÕES ANDROID**

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina Trabalho de Conclusão de Conclusão de Curso 2, do curso de Engenharia de Computação da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para a obtenção do título de Bacharel.

Orientador: Prof. Dr. André Takeshi Endo

CORNÉLIO PROCÓPIO

2015

Dedico este trabalho a meus pais que sempre incentivaram o meu estudo, aos meus amigos que estiveram presente nos cinco anos de curso e a todos os professores que transmitiram seu conhecimento ao longo de toda minha vida.

AGRADECIMENTOS

Agradeço ao meu orientador Prof. Dr. André Takeshi Endo, por todo o tempo dispendido em ensinar e ajudar no desenvolvimento deste trabalho, bem como por sua dedicação e esforço nesta orientação.

À minha família, em especial meus pais que sempre se esforçaram para que eu tivesse o melhor na minha vida, em especial nos estudos, me incentivando e apoiando em momentos difíceis.

Aos meus amigos da faculdade que estiveram comigo ao longo desses cinco anos de estudos, dificuldades e acima de tudo muita felicidade. Agradeço especialmente a minha amiga Aline Ferreira da Silva, pela grande amizade construída ao longo desses cinco anos e pela ajuda na validação do projeto.

Também quero registrar minha gratidão a todos os professores, coordenadores e demais funcionários da Universidade Tecnológica Federal do Paraná de Cornélio Procópio que contribuíram para minha formação e realização desta pesquisa.

RESUMO

CAVALCANTE, Bruno M. **Apoio a Localização de Defeitos Durante a Execução de Sequências de Teste para Aplicações Android**. 2015. Trabalho de Conclusão de Curso (Graduação) – Engenharia de Computação. Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2015.

O crescente aumento no número de vendas de *smarthphones* e *tablets* que utilizam o sistema operacional Android favorece o desenvolvimento de novas tecnologias e ferramentas de Engenharia de Software que ajudem no processo de criação de novas aplicações. Durante a fase de teste revelam-se defeitos, estes que muitas vezes são de difícil localização. Utilizar o Teste Baseado em Modelo (TBM) para o desenvolvimento dos testes, cria um modelo confiável para desenvolver o teste, com isso foi escolhido o TBM como modelo de teste para modelar as sequências de teste deste projeto. Assim, este trabalho de conclusão de curso exhibe o desenvolvimento da ferramenta Visual DDroid, que coleta *screenshots* e *logs* durante a execução de sequências de teste para aplicações Android e exhibe os dados coletados para o usuário. Foi realizada uma avaliação preliminar da ferramenta e nesta foi verificado que os dados coletados são úteis à fase de depuração das sequências de teste e a ferramenta ajudou na localização dos defeitos revelados pelas sequências de teste.

Palavras-chave: Teste Baseado em Modelo. Aplicações Android. Depuração.

ABSTRACT

CAVALCANTE, Bruno M. **Supporting Fault Localization in the Execution of Test Sequences for Android Applications**. 2015. Course Conclusion Work (Undergraduate) – Engenharia de Computação. Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2015.

The increasing number of Android smartphones and tablets favors the development of new technologies and Software Engineering tools that assists the development of new applications. Through the testing phase, faults are often difficult to be located. Using the Model-Based Test (MBT) for the development of the tests, creates a reliable model to develop the test, thereby was chosen the MBT to modeling the test sequences of this project. Therefore, this work shows the development of the Visual DDroid tool that gathers screenshots and logs while test sequences for Android applications are executed and displays the collected data to the user. A preliminary evaluation of the Visual DDroid tool was performed. We verified that the collected data by the tool are useful for the debugging phase of the test sequences and the tool helped in locating the faults revealed by the test sequences.

Keywords: Model-based Testing. Android Application. Debugging.

LISTA DE FIGURAS

Figura 1 - Camadas da plataforma Google Android.	13
Figura 2 - Tela Principal Favorite Twitter®.....	15
Figura 3 - Route Tracker.	15
Figura 4 - ESG da aplicação Route Tracker.....	17
Figura 5 - Ambiente de Depuração.	19
Figura 6 – Sequência de teste da aplicação Favorite Twitter® Searches.	25
Figura 7 - ESG da aplicação Favorite Twitter® Searches	26
Figura 8 - Exemplo de código que executa um teste.	27
Figura 9 - Métodos da classe Adaptor.....	29
Figura 10 - Arquitetura da Ferramenta Visual DDroid.	31
Figura 11 – Visual DDroid, tela inicial.....	32
Figura 12 – Visual DDroid, tela inicial com <i>screenshots</i> carregados.....	32
Figura 13 - Exibição do <i>screenshot</i> junto com o respectivo <i>log</i> na exibição de um defeito.	33
Figura 14 - Exibição do <i>screenshot</i> junto com o respectivo <i>log</i>	34
Figura 15 - Passos do estudo realizado.	37

SUMÁRIO

1 INTRODUÇÃO.....	9
1.1 MOTIVAÇÃO.....	10
1.2 OBJETIVO.....	11
1.3 ORGANIZAÇÃO.....	11
2 REVISÃO BIBLIOGRÁFICA.....	12
2.1 Android.....	12
2.1.1 Aplicações Móveis para Android.....	14
2.1.2 Favorite Twitter® Searches.....	14
2.1.3 Route Tracker.....	15
2.2 Teste Baseado em Modelo.....	16
2.2.1 Event Sequence Graph.....	17
2.3 Depuração em Aplicações Móveis.....	18
2.3.1 Android Debug Bridge.....	18
2.3.2 Dalvik Debug Monitor.....	20
2.3.2.1 LogCat.....	20
2.3.3 Java Debug Wire Protocol.....	21
2.3.4 Android Virtual Device.....	21
2.4 Trabalhos Relacionados.....	21
3 DESENVOLVIMENTO.....	24
3.1 Execução de uma Sequência de Teste.....	24
3.1.1 Exemplo de Sequência de Teste.....	25
3.1.2 Sequências de Teste Desenvolvidas.....	30
3.2 Desenvolvimento da Ferramenta.....	30
3.2.1 Arquitetura da Ferramenta.....	31
4 AVALIAÇÃO PRELIMINAR DA FERRAMENTA.....	35
4.1 Configuração do Estudo.....	35
4.2 Análise dos Resultados.....	38
4.2.1 Perfil dos Participantes.....	42
4.2.2 Sugestões de Melhorias na Ferramenta.....	43
5 CONSIDERAÇÕES FINAIS.....	44
REFERÊNCIAS.....	45

1 INTRODUÇÃO

Computação móvel é um paradigma de computação onde o usuário utiliza um dispositivo de fácil transporte e conexão de dados sem fio com independência da localização física do aparelho. A esta independência da localização, é dada o nome de mobilidade, onde o usuário pode mudar sua posição geográfica, porém mantendo sua conexão de dados. Já o fato do dispositivo ser de fácil transporte, ou seja, pequeno, leve e com conexão de dados sem fio, é dado o nome de portabilidade (FORMAN; ZAHORJAN, 1994).

Os conceitos de computação móvel são aplicados no desenvolvimento de *smartphones* e *tablets*, estes que são informados dispositivos móveis. As aplicações desenvolvidas para estes aparelhos foram inicialmente voltadas para o entretenimento, porém algumas tecnologias baseadas nos conceitos de computação móvel são desenvolvidas para outras áreas, como sistema financeiro, de saúde e indústrias (MUCCINI et al., 2012).

O mercado de dispositivos móveis pode gerar um lucro de aproximadamente 350 bilhões de dólares em 2015 (GAO et al., 2014). Em 2012, segundo a ABI Research (2012), a receita com ferramentas de teste para aplicações móveis poderia exceder os 200 milhões de dólares, já a previsão para 2017 é que este valor chegue perto de 800 milhões. Estes dispositivos contêm tanto aplicações nativas e Web, sendo que estas necessitam cada vez mais de ferramentas de teste melhores (GAO et al., 2014).

O domínio da plataforma Android é evidente, segundo uma avaliação da *International Data Corporation* (IDC), cerca de 80,2% de todos os *smartphones* contém esse sistema operacional. Segundo a mesma avaliação, no ano de 2014 era esperado um crescimento de 25,6% de vendas de dispositivos Android (IDC, 2014). Junto com o crescimento das vendas de dispositivos móveis, cresce também a preocupação com a confiabilidade das aplicações funcionando nestes aparelhos (MUCCINI et al., 2012).

Segundo Gao et al. (2014), o teste de aplicações móveis se refere a diferentes tipos de teste, como testes em aplicações nativas, testes em aplicações *web* e em dispositivos. Para esta atividade são utilizados métodos de teste de software bem definidos e ferramentas que garantem qualidade em funções, comportamentos e desempenho, além de características como conectividade e segurança.

Entre as técnicas de teste existentes, o Teste Baseado em Modelo (TBM), que consiste da geração de casos de teste automáticos baseados em um modelo estrutural, apresenta algumas vantagens, como a geração automática de casos de teste, alta taxa de detecção de defeitos, alto nível de automatização e adequação a mudanças em requisitos (BLACKBURN et al., 2004; UTTING; LEGEARD, 2006; GRIESKAMP et al., 2011). No TBM, os casos de teste gerados podem assumir a forma de sequências de teste, estas que representam uma série de eventos que devem ser aplicados como entradas e observados como saídas para verificar determinadas funcionalidades do software em teste. As sequências de teste são geradas por ferramentas de auxílio ao TBM e de acordo com o modelo formal que descreve a aplicação a ser testada, um exemplo de modelo é o *Event Sequence Graphs* (ESG).

Aplicações utilizadas em ambientes críticos, como a saúde, finanças e compras, necessitam de testes sistemáticos e repetíveis. Utilizar teste automatizado e formal, que segundo Bernardo e Kon (2008) são programas ou *scripts* simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos, é uma forma de se conseguir isso, pois estes são baseados em modelos matemáticos, como o ESG, que é utilizado para representar como se espera que a aplicação funcione (BELLI, et al., 2006).

O processo de encontrar problema em um software em tempo de execução é a definição dada por Krajci e Cummings (2013) para a depuração ou *debug*. O programa responsável por realizar o processo de depuração é chamado de depurador ou *debugger*.

1.1 MOTIVAÇÃO

No desenvolvimento de aplicações móveis, a fase de teste ganha destaque, pois é neste estado do desenvolvimento que podem-se encontrar demasiados problemas. Em um relatório de Gartner (2014) é afirmado que em 2015 75% das aplicações para Android falharam em testes básicos, em relação a segurança dos dados. Assim o desenvolvimento de ferramentas que auxiliem no processo de teste pode vir a reduzir este número.

Alguns problemas podem ocorrer no momento do teste, como por exemplo, as sequências de teste serem muito longas (dificultando a identificação de defeitos no teste, caso alguma ocorra) ou o desenvolvedor pode não estar familiarizado com o

TBM, ou seja, ele não entenderá como os testes funcionam, entre outros problemas. Assim o desenvolvimento de ferramentas que auxiliem no processo de teste pode reduzir os problemas citados acima.

A coleta de dados durante a fase de testes apresenta a vantagem de as informações coletadas serem únicas pois, ao se repetir os testes, alguns resultados podem variar. Outro fator positivo em coletar os dados durante a fase de teste está no fato do custo de reexecução dos testes ser alto, já que alguns testes são realizados em emuladores (estes que apresentam um baixo desempenho) ou em múltiplos dispositivos.

1.2 OBJETIVO

Neste contexto o objetivo deste trabalho de conclusão de curso é apresentar a ferramenta Visual DDroid esta que recupera *screenshots* e *logs* após a execução de sequências de teste para aplicações Android e exibe os dados coletados para o usuário. A ferramenta visa auxiliar a identificação de possíveis defeitos que podem ocorrer durante a execução das sequências de teste.

1.3 ORGANIZAÇÃO

O desenvolvimento deste trabalho está dividido da seguinte forma: o Capítulo 2 consiste na fundamentação teórica, detalhada em revisão bibliográfica que apresenta os principais assuntos desta proposta: Aplicações Móveis, Teste Baseado em Modelo e Depuração. Neste capítulo também é exibida a seção de trabalhos relacionados. Já o Capítulo 3 consiste em apresentar o desenvolvimento de sequências de teste e o desenvolvimento da ferramenta. No Capítulo 4 é apresentado a avaliação preliminar da ferramenta e a análise dos resultados. Por fim, o Capítulo 5 expõe as considerações finais.

2 REVISÃO BIBLIOGRAFICA

Neste capítulo será apresentada uma revisão da literatura dos temas relacionados a este trabalho, visando auxiliar o leitor a compreender o mesmo. As seguintes subseções serão apresentadas: Android, Sequências de Teste e Depuração em Aplicações Móveis.

2.1 Android

Segundo Ableson et al. (2012) a plataforma Android revolucionou o mercado de celulares pois é a aplicação móvel *open source* com maior presença no mercado mundial. O Android em 2015 é mantido pelo Google, porém recebe apoio de desenvolvimento da *Open Handset Alliance* (OHA), este que é um grupo formado por 84 empresas de tecnologia (OPEN HANDSET ALLIANCE, 2014), algumas destas são: HTC, LG, Motorola, Samsung, Sony Ericsson, Toshiba, Srint Nextel, China Mobile, T-Mobile, ASUS, Intel e Garmun.

A plataforma Android possui código aberto, padronizada e única, ou seja, apresenta a vantagem de que os desenvolvedores possam colaborar com o seu aprimoramento, já que estes podem acrescentar novas funções ou remover defeitos, além do fato de que os fabricantes de hardware possam alterar seu código fonte para moldar o sistema a seu modo (MEDNIEKS ET AL., 2012).

Android é uma plataforma desenvolvida para dispositivos móveis, ele inclui um *kernel* baseado em Linux, uma interface com diversos elementos para serem utilizados, dispõe de bibliotecas de código, *frameworks* e suporte a multimídia (ABLESON et al., 2012). Os componentes referentes ao Sistema Operacional (SO) são escritos em C ou C++, já as aplicações são desenvolvidas em Java e são interpretadas pela Máquina Virtual (MV) chamada de Dalvik (ABLESON et al., 2012).

A plataforma Android é subdividida em cinco camadas, cada uma apresenta suas características. Compreender cada uma delas melhora o desenvolvimento de futuras aplicações (ABLESON et al., 2012; LECHETA, 2013). Abaixo segue as definições sobre cada camada (ABLESON et al., 2012; LECHETA, 2013).

1. *Applications*: é onde se encontram as aplicações, tais como clientes e-mail, navegador Web, contatos entre outros.

2. *Application Framework*: é responsável por disponibilizar os recursos necessários para o desenvolvimento de aplicações Android.

3. *Libraries*: dispõe de um conjunto de bibliotecas em C/C++ estas que permitem aos aplicativos trabalharem com arquivos de mídia (MP3, JPG, PNG, MPEG4, H.264, AAC e AMR).

4. *Android Runtime*: controla as bibliotecas centrais do núcleo e também controlar as instâncias da máquina virtual Dalvik e seus processos.

5. *Linux Kernel*: é responsável pelos serviços centrais do Android, tais como gerenciamento de memória, processos, segurança e tarefas

A Figura 1 representa a esquematização das camadas citadas.

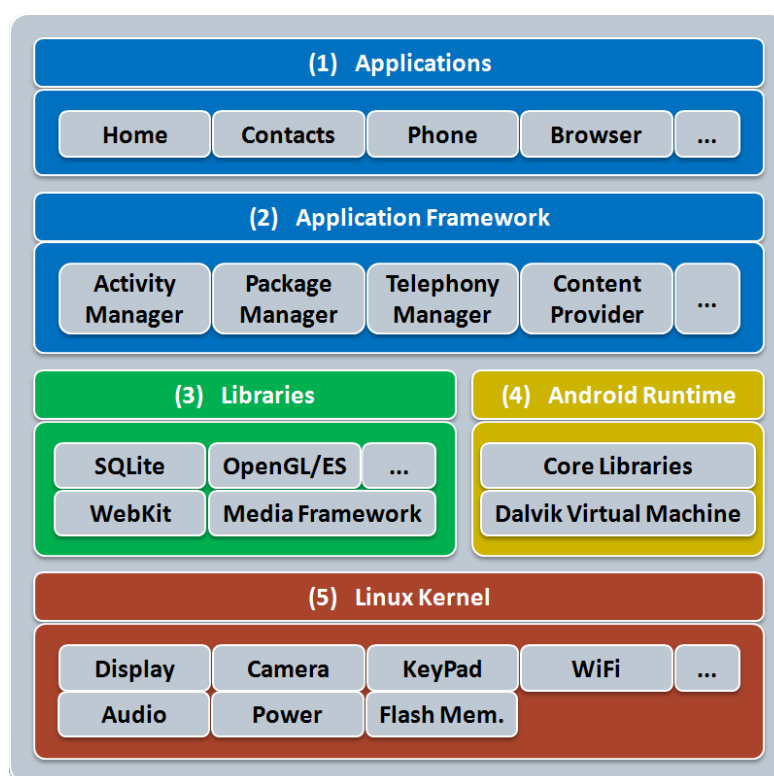


Figura 1 - Camadas da plataforma Google Android.
Fonte: Extraído de FARTO (2015).

2.1.1 Aplicações Móveis para Android

As aplicações desenvolvidas para Android são implementadas em linguagem Java. Segundo Deitel et al. (2012), a escolha pelo Java é baseada no fato das aplicações desenvolvidas poderem ser executadas em diversos dispositivos Android, além do fato da linguagem Java ser *free e open source*.

Para o desenvolvimento das aplicações, pode ser utilizada uma *Integrated Development Environment* (IDE), o Android Studio e o Eclipse são exemplos que podem ser utilizados. Porém é necessário o uso do *Android Software Development Kit* ou Android SDK. O Android SDK disponibiliza bibliotecas e recursos necessários para desenvolver a aplicação móvel para Android, passando pelas fases de construção, teste e depuração (LECHETA, 2013).

As aplicações móveis desenvolvidas para a plataforma Android, podem utilizar recursos como banco de dados embarcado, sensores, integração com redes sociais e consumidoras (*Web Services*). As aplicações podem ser desenvolvidas para sistemas de entretenimento e multimídia, sistemas de tempo real e críticos.

A seguir serão descritas duas aplicações que utilizam alguns dos recursos citados acima.

2.1.2 Favorite Twitter® Searches

Esta aplicação permite ao usuário registrar uma cadeia de caracteres e salvá-la com um determinado nome. Ao salvar, são gerados na tela dois botões, o primeiro contém o nome dado a cadeia de caracteres e o segundo é chamado *edit*. Ao clicar no primeiro botão o usuário é direcionado para o navegador de internet do dispositivo e lá são exibidos os *tweets* mais recentes que contenham a cadeia de caracteres salva. Já ao clicar em *edit*, é possível modificar tanto a cadeia de caracteres quanto o nome que lhe foi atribuído (DEITEL et al., 2012). A Figura 2 exibe a tela principal desta aplicação.

2.1.3 Route Tracker

Esta aplicação permite ao usuário calcular o seu trajeto utilizando o serviço de localização do dispositivo, o *Global Positioning System* (GPS). Para isto, basta clicar no botão *start tracking*, que o aplicativo começará a registrar o seu trajeto e exibi-lo no mapa. Quando desejar terminar de registrar o caminho basta clicar no botão *stop tracking* e logo em seguida será exibida a distância percorrida e a velocidade média. A Figura 3 exibe a tela principal da aplicação.

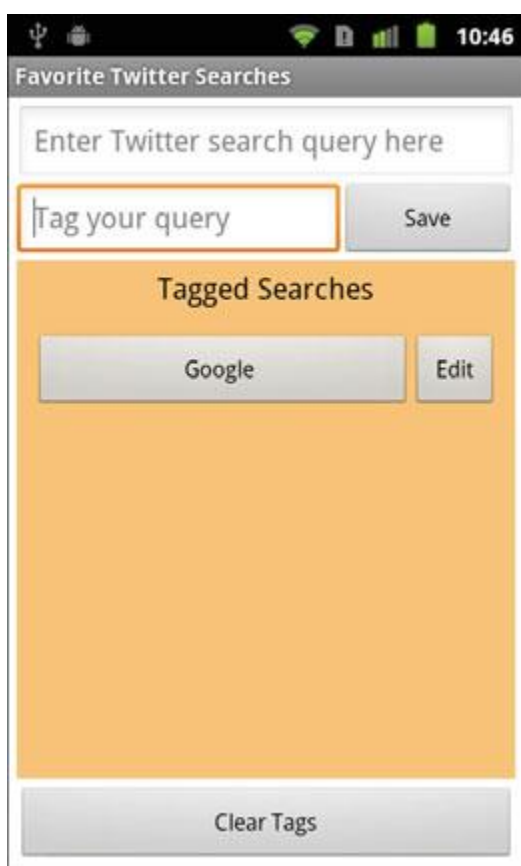


Figura 2 - Tela Principal Favorite Twitter® Searches.
Fonte: Extraído de DEITEL et al. (2012).

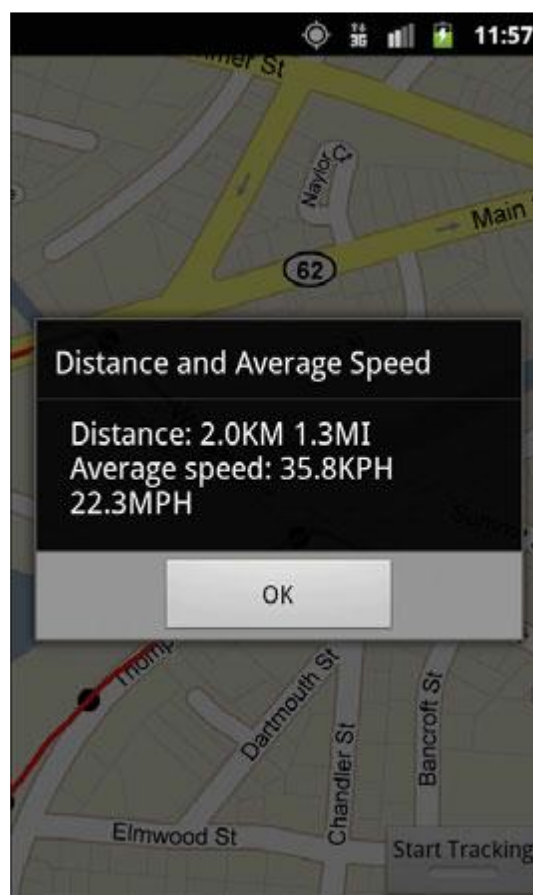


Figura 3 - Route Tracker.
Fonte: Extraído de DEITEL et al. (2012).

2.2 Teste Baseado em Modelo

A geração automática de casos de teste que utiliza um modelo comportamental ou estrutural, que pode ser chamado de modelo de teste é a definição dada por Sinha e Smidts (2006) sobre o Teste Baseado em Modelo. Um caso de teste é um par ordenado de uma entrada e uma saída esperada do sistema em teste (BELLI, et al., 2006). O TBM é uma solução eficiente e eficaz que pode ser utilizado para testes requeridos por diversas classes de software. Na definição do processo de teste, ressaltam-se quatro passos principais (SINHA; SMIDTS, 2006):

1. Modelagem: é construído um modelo de teste com o objetivo de representar o que deve ser testado no sistema.

2. Geração de casos de teste: utilizando o modelo de teste, os casos de teste são definidos utilizando algum critério de seleção definido.

3. Concretização: os casos de teste são concretizados com o objetivo de serem executados no sistema em teste. Assim adaptadores podem ser definidos para intermediar os diferentes níveis de abstração.

4. Execução dos testes: os casos de teste e os adaptadores são utilizados para executar os testes no sistema em teste.

O modelo de teste, descrito no item 1, pode ser projetado utilizando diferentes técnicas de modelagem, como a Máquina de Estados Finitos (MEFs) (LEE; YANNAKAKIS, 1996), *labelled transition systems* (LTS) (TRETMANS, 1995), *Unified Modeling Language* (UML) (OMG, 2009) e ESG (Belli et al., 2006).

2.2.1 Event Sequence Graph

O ESG é um grafo direcionado, cujos nós são eventos e arcos são sequências válidas de eventos. O nó definido por “[” representa uma entrada enquanto o “]” define a saída. Utilizando o aplicativo *Route Tracker*, descrito na Subseção 2.1.3, é apresentado na Figura 4 um exemplo de ESG. Qualquer sequência se inicia com “[” e é finalizada com “]” é chamada de *complete event sequence* (CES).

Segundo Belli et al. (2006), o ESG apresenta muitas vantagens. Dentre essas, ressalta-se o refinamento incremental das especificações que podem estar rudes ou nem existirem no início do processo de teste ou quando a abordagem pode ser implantada em teste baseado em código. Um exemplo disso é quando se usa o código fonte do sistema em teste como a especificação final e diagrama de fluxo de controle.

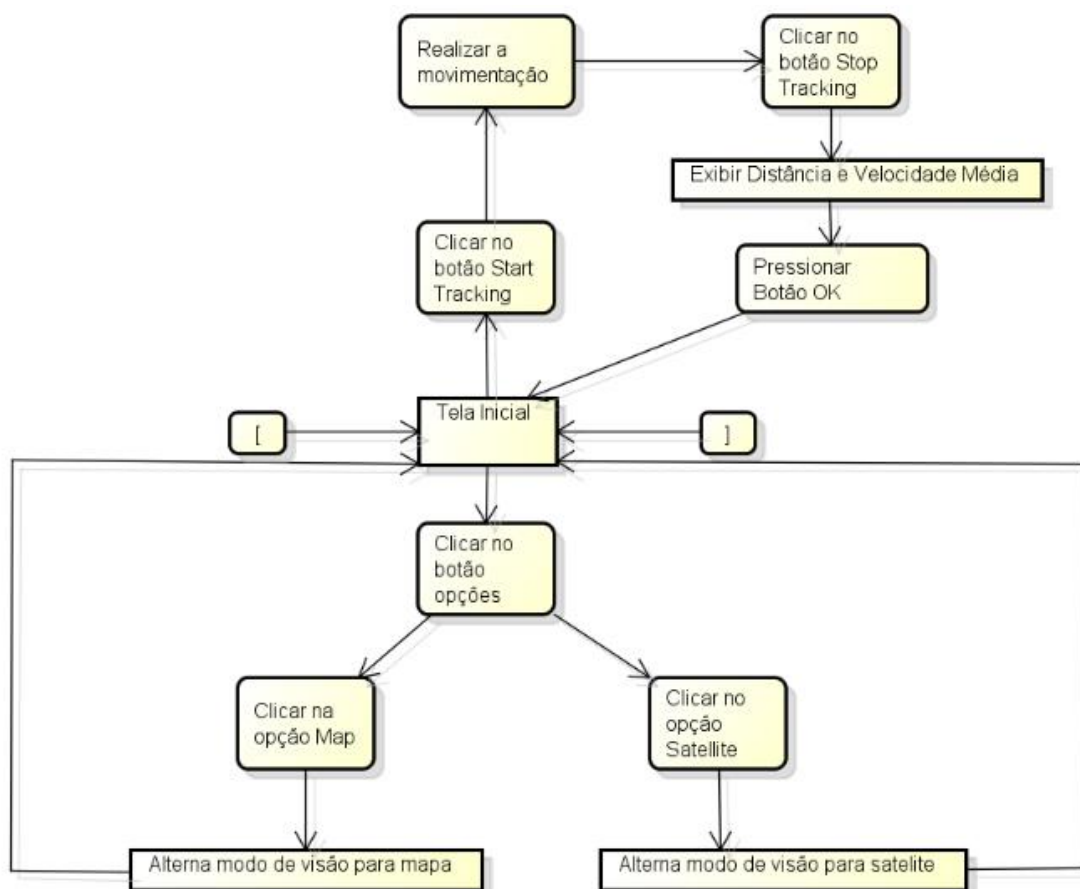


Figura 4 - ESG da aplicação Route Tracker.
Fonte: Autoria própria.

2.3 Depuração em Aplicações Móveis

A definição dada por Krajci e Cummings (2013) para a depuração ou *debugging* é: processo de encontrar um defeito em um software em tempo de execução. O programa responsável por realizar o processo de depuração é chamado de depurador ou *debugger*.

No desenvolvimento de aplicações móveis para Android, o uso do Android SDK é obrigatório, pois ele dispõe de diversas ferramentas que são necessárias para realizar a depuração da sua aplicação (ANDROID, 2015a). Como já foi citado na seção 2.1.1, o desenvolvimento de aplicativos para Android, é baseado na linguagem Java, porém para realizar a comunicação entre o *debugger* e a Máquina Virtual Java (MVJ), é utilizado o *Java Debug Wire Protocol* (JDWP) (ORACLE, 2015), este que será explicado na Subseção 2.3.3.

Outra ferramenta necessária para realizar o *debug* da sua aplicação é o *Android Debug Bridge* (ADB), este que atua como uma conexão entre o dispositivo em que ocorrerá a depuração e o DDMS (ANDROID, 2015a), o ADB será explicado na Subseção 2.3.1. O dispositivo onde ocorre a depuração pode ser tanto um dispositivo físico ou um *Android Virtual Device* (AVD), que será explicado na subseção 2.3.4.

O *Dalvik Debug Monitor Server* (DDMS) é outra ferramenta útil no processo de depuração, pois ele é uma aplicação gráfica que realiza a comunicação do JDWP e o ADB (ANDROID, 2015a). O DDMS será explicado na Subseção 2.3.2. A Figura 5, exibe todo o processo de depuração para uma aplicação Android, nas subseções a 2.3.1, 2.3.2, 2.3.3 e 2.3.4, serão explicados cada um desses passos.

2.3.1 Android Debug Bridge

O ADB é uma ferramenta de linha de comando que permite a comunicação entre uma instância de um AVD ou um dispositivo físico e o desenvolvedor da aplicação. Ele pode ser considerado uma aplicação cliente servidor e é dividida em três componentes (ANDROID, 2015b):

1. Cliente: é executado na máquina em que está se desenvolvendo. Ele é chamado a partir de uma linha de comando ADB. O cliente pode ser criado a partir do DDMS também.

2. Servidor: é executado na máquina em que está se desenvolvendo, como um processo em segundo plano. Sua função é gerenciar a comunicação entre o cliente e o ADB que está sendo executado no dispositivo ou AVD.

3. *Daemon*: é executado no dispositivo ou AVD, como um processo de fundo.

Ao iniciar o ADB, ele verifica se já existe alguma instância de cliente iniciada, caso não exista, ele inicia o servidor. A primeira tarefa realizada pelo servidor é encontrar todas as instâncias de dispositivos e/ou AVD que estão conectadas na faixa de portas que vão de 5555 até 5585. Quando o servidor encontra um *daemon*, ele inicia a comunicação com ele na porta que aquele está (ANDROID, 2015b).

Após o servidor configurar a comunicação com todos os *daemon*, ele passa a aceitar os comandos ADB, estes que podem ser direcionados para cada uma das instâncias conectadas (ANDROID, 2015b).

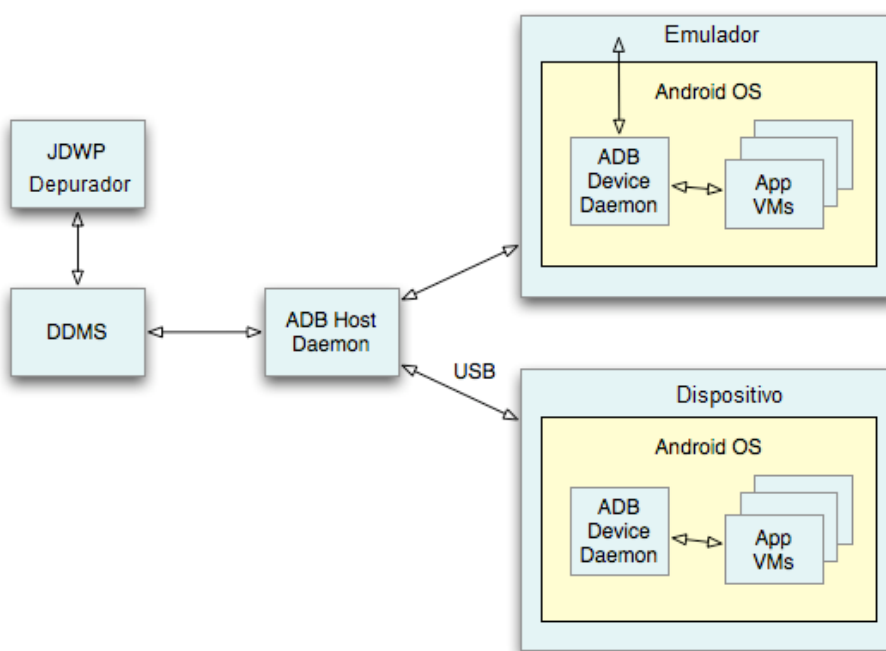


Figura 5 - Ambiente de Depuração.
Fonte: Adaptado de ANDROID (2015a).

2.3.2 Dalvik Debug Monitor Server

O DDMS é uma ferramenta de depuração utilizada no desenvolvimento de aplicações para Android. Ela fornece serviços de encaminhamento de porta, captura de tela no dispositivo ou AVD, informações sobre o dispositivo ou AVD, *threads* e *heap* executando no dispositivo ou AVD, *logcat*, processo e informação sobre o estado de rádio, chamadas e SMS (ANDROID, 2015c).

O processo de funcionamento do DDMS é baseado na comunicação entre o DDMS e a MVJ (pelo ADB). Cada aplicação executada no dispositivo ou AVD, possui sua própria MVJ, assim, o DDMS consegue recuperar informações sobre cada aplicação funcionando.

2.3.2.1 LogCat

Ferramenta utilizada para visualização dos *logs* do sistema Android. Utilizando o LogCat, é possível coletar e visualizar as saídas de *debug* do sistema. É possível também visualizar e coletar os *logs* das aplicações que estão sendo executadas pelo dispositivo (ANDROID, 2015e).

É possível para o programador gerar *logs* de dentro da própria aplicação ou de sequências de teste, esses *logs* também são exibidos pelo LogCat. Para isto é necessário utilizar a classe *Log* do Android (ANDROID, 2015f). Os níveis de mensagens de *log* que podem ser gerados são (ANDROID, 2015f): *Assert*, *Debug*, *Error*, *Info*, *Verbose* e *Warn*.

2.3.3 Java Debug Wire Protocol

O JDWP é um protocolo que realiza a comunicação entre o *debugger* e a MVJ. A utilização do JDWP permite que o processo de depuração ocorra em processos diferentes no mesmo computador ou em um computador remoto (ORACLE, 2015). A JDWP difere de muitos protocolos de especificação que só detalham o formato e o *layout*, não o que é transportado. Porém deve-se utilizar uma *Application Programming Interface* (API) para diferentes mecanismos de transporte (ORACLE, 2015).

2.3.4 Android Virtual Device

O AVD é um emulador de Android, que permite ao desenvolvedor modelar o que ele necessita, permitindo tanto a configuração de hardware quanto a de software. O AVD consiste das seguintes configurações (ANDROID, 2015d):

1. Perfil de Hardware: define as características de hardware do dispositivo.
2. Mapeamento da Imagem do Sistema: define qual versão do Android irá atuar no emulador.
3. Outras Opções: define a máscara que será utilizada no emulador, as dimensões da tela e se o emulador terá um cartão SD.
4. Memória de Armazenamento Dedicada para o Emulador: define a memória interna do emulador, onde serão armazenados os aplicativos e dados.

O papel do AVD ou do dispositivo físico é o mesmo na fase de depuração.

2.4 Trabalhos Relacionados

Alguns projetos desenvolvidos apresentam relação com este trabalho. Deve-se ressaltar que a ferramenta desenvolvida ajuda na localização de defeitos durante a execução de sequências de teste, já os trabalhos apresentados em Hausman et al. (2013) e Nguyen et al. (2013) são ferramentas de depuração e o trabalho apresentado em Machado et al. (2013) é um *plugin* que trabalha em cima de uma técnica de teste diferente do TBM.

















Em Hausman et al. (2013) é apresentada uma ferramenta de depuração, chamada de *Universal JavaScript Debugger*, para aplicações móveis que são desenvolvidas no ambiente AppConKit (APPCONKIT, 2014). Esta é uma plataforma de desenvolvimento que gera aplicações de multi plataforma, para Android e iOS, utilizando a linguagem JavaScript. Na validação da ferramenta foram realizados 5216 casos de teste, deste total 84,7% foram realizados com sucesso. As falhas e erros que ocorreram foram devido a: perda de dependências, problemas na *engine* do JavaScript e no momento de checar a igualdade do código (HAUSMAN et al., 2013).

Em Nguyen et al. (2013) é apresentada uma ferramenta de depuração, chamada de GROPG *graphical onphone debugger*. Ela apresenta uma interface gráfica para a depuração e é uma aplicação para o dispositivo móvel, por exemplo, quando um *break point* é inserido no código, no momento em que a depuração chega nele a aplicação exibe no dispositivo as informações da depuração. Para validação da ferramenta, foi realizada uma comparação com outra ferramenta de depuração que é executada no dispositivo móvel, chamada de *DroidDebugger*, esta que exibe as informações da depuração em texto, via *shell*. Na primeira comparação, foram utilizadas três aplicações, em todos os testes, o GROPG consumiu maior quantidade de memória RAM, porém seu tempo de execução foi 66% menor em relação ao *DroidDebugger*. Na segunda comparação, foi verificado a quantidade de passos que cada ferramenta levava para colocar um *break point* no código. Por ser uma ferramenta gráfica, utiliza-se apenas 39 passos para se colocar um *break point*, já utilizando o *DroidDebugger* leva-se 112 passos, já que ele é uma ferramenta que utiliza linhas de comando (NGUYEN et al., 2013).

Em Machado et al. (2013) é apresentado um *plugin* para a IDE Eclipse, chamado MZoltar. Sua função é encontrar defeitos em aplicações móveis para Android, baseado na técnica *Spectrum-based fault localization* (SFL). Para validar a ferramenta foram realizados testes com três aplicações *open source* e uma aplicação exemplo, todas desenvolvidas para plataforma Android. Foi confirmado que a infraestrutura para coletar informações necessárias é 5,75% mais leve em relação a aplicações que utilizar o SFL, enquanto a precisão no diagnóstico do relatório que é gerado pela aplicação é similar as aplicações com o mesmo propósito dessa (MACHADO et al., 2013).

Como foi visto nos trabalhos a cima, existem ferramentas que são criadas com o propósito de facilitar o processo de localização de defeitos na depuração de aplicações móveis para Android. Em todas os trabalhos citados, os resultados apresentados foram satisfatórios, porém nenhum deles apresenta o que ocorre com a aplicação em cada momento do teste, ou seja, caso um defeito ocorra, o testador não sabe o que estava ocorrendo com a aplicação no momento em que o defeito acontece. Assim, uma ferramenta que exiba *screenshots* junto com o *log* do momento da execução daquele teste auxilia na localização do defeito, caso ele ocorra.

No Quadro 1 é realizada uma comparação entre a ferramenta desenvolvida neste projeto, chamada de Visual DDroid e as três ferramentas citadas, *Universal JavaScript Debugger*, GROPG e MZoltar.

	Universal JavaScript Debugger	GROPG	MZoltar	Visual DDroid
Ferramenta é Multiplataforma¹				
Exibe o ponto onde a sequência de teste falhou²				
Consumo de Memória RAM³	-	25 Mb	-	100 Mb
Exibe logs da aplicação				
Trabalha com aplicações de diferentes plataformas⁴				

Quadro 1 - Comparação entre ferramentas.
Fonte: Autoria própria.

¹ Considerando as plataformas Windows, Lunux e Mac OS.

² Considerando um *screenshot* da tela do dispositivo no momento no qual o defeito ocorre

³ O GROPG é executado no dispositivo móvel enquanto o Visual DDroid é uma aplicação Desktop.

⁴ Considerando as plataformars Android e iOS.

3 DESENVOLVIMENTO

Nesta seção são apresentadas as descrições das atividades que foram desenvolvidas para alcançar o objetivo deste projeto. Essa seção foi subdividida de acordo com as atividades planejadas: desenvolvimento e execução de uma sequência de teste e desenvolvimento da ferramenta.

3.1 Execução de uma Sequência de Teste

Antes de se iniciar a execução de uma sequência de teste é necessário compreender três definições envolvidas no processo de desenvolvimento dos mecanismos do TBM (FARTO, 2015):

- **Artefatos:** No desenvolvimento de aplicações móveis são utilizados artefatos específicos, que são denominados recursos em plataformas de mobilidade. Um conjunto de recursos exibe diversas características da aplicação: sequências de animação e transição de telas, componentes visuais, elementos gráficos e folhas de estilo. O conjunto de artefatos são definidos em formato *Extensible Markup Language* (XML), podendo ser manipulados por ferramentas de apoio, tanto para sua geração dinâmica como para a interpretação e extração de informações úteis na realização de testes automatizados (FARTO, 2015).

Para o TBM, o principal artefato gerado de forma automática é o conjunto de casos de teste, porém alguns artefatos precisam de construídos de forma manual pelo testador, como um modelo de teste e um adaptador para a execução dos testes automatizados (FARTO, 2015).

- **Ferramentas:** Algumas ferramentas usadas no processo de teste foram desenvolvidas para auxiliar o testador, sistematizando passos repetíveis predispostos a erros (FARTO, 2015). A ferramenta que é utilizada para realizar o processo de execução de testes deste trabalho é o Robotium (ROBOTIUM, 2015).

O Robotium é definido com um *framework* de teste automatizado para Android (ROBOTIUM, 2015a). Com este *framework*, é possível escrever casos de teste que simulam interações do dispositivo com o usuário, como entrada de textos e

cliques. É possível utilizar o Robotium tanto na IDE Eclipse quanto no Android Studio (ROBOTIUM, 2015b).

- Processo: Como foi apresentado na Subseção 2.2, o processo de TBM é constituído de quatro passos: modelagem, geração de testes, concretização e execução de testes. Cada um desses passos, se forem realizados de forma incorreta, ou se não forem realizados, podem resultar em um gasto desnecessário de tempo. (FARTO, 2015).

3.1.1 Exemplo de Sequência de Teste

Nesta subseção será apresentado um exemplo de uma sequência de teste automatizada para a aplicação Favorite Twitter® Searches, apresentada na Subseção 2.1.2. Na Figura 7 é exibido o ESG da aplicação Favorite Twitter® Searches. A sequência que foi desenvolvida pelo autor demonstra o processo de deleção de uma busca salva, ela é baseada na sequência de teste da Figura 6.

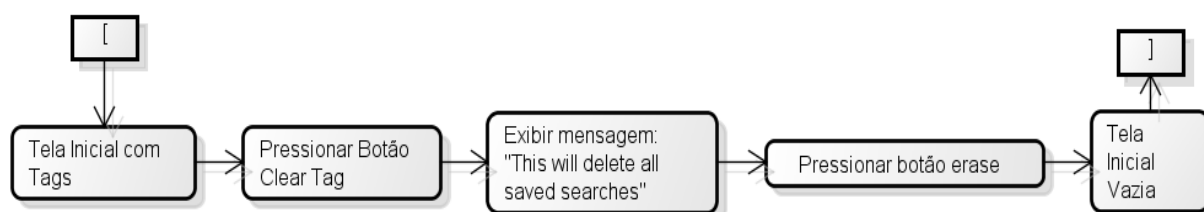


Figura 6 – Sequência de teste da aplicação Favorite Twitter® Searches.
Fonte: Autoria própria.

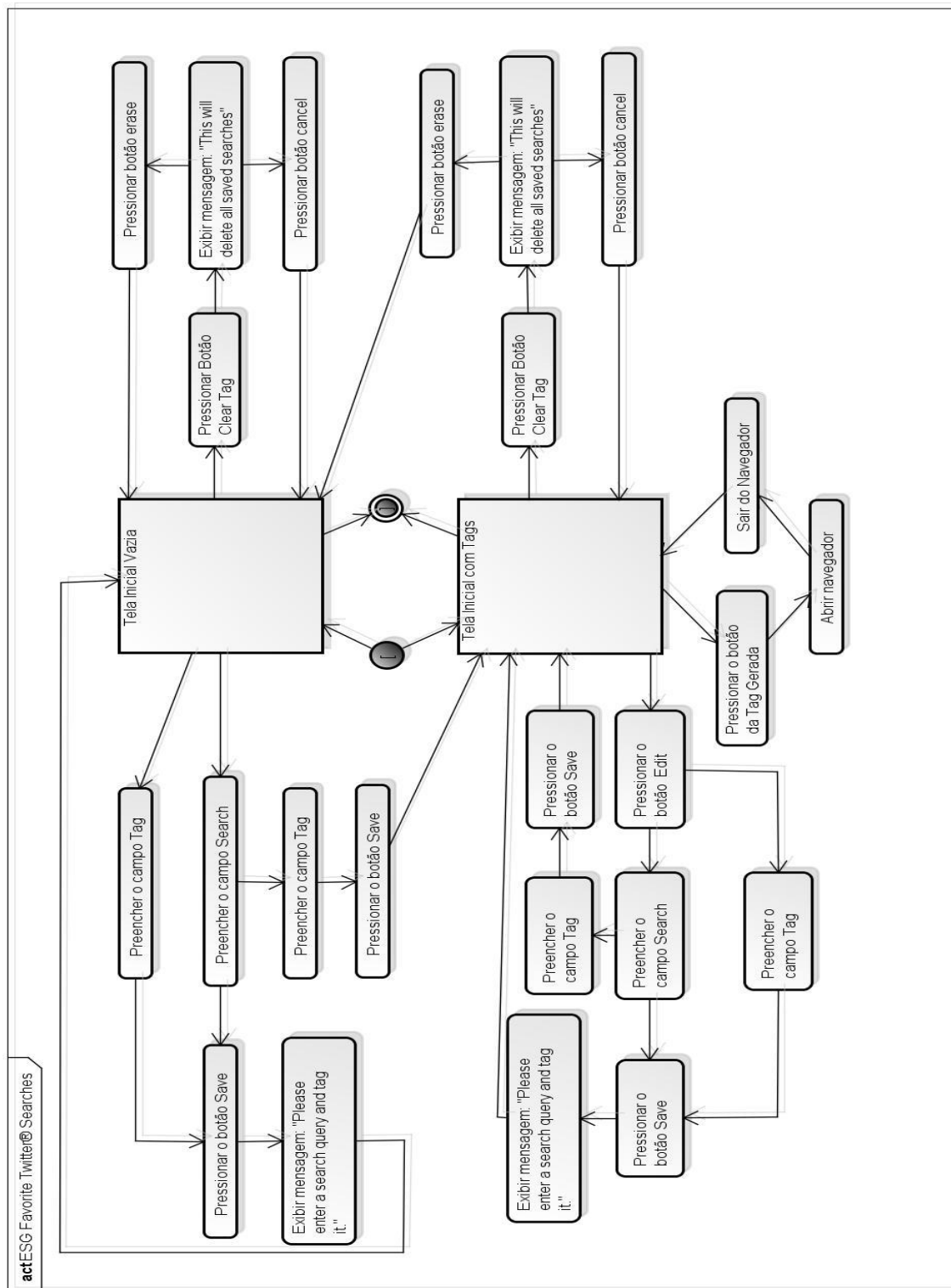


Figura 7 - ESG da aplicação Favorite Twitter® Searches .
 Fonte: Autoria própria.

Na Figura 6, percebe-se o uso do “[” e do “]”, assim a sequência é considerada uma CES. A Figura 8 apresenta um método que executa a deleção de uma busca salva pelo aplicativo. Foi utilizada a sequência da Figura 6 como base para o desenvolvimento do código de teste. O código foi desenvolvido na linguagem Java e ele pertence à classe `CesTest`, também desenvolvida pelo autor. Essa classe contém todos os métodos de teste desenvolvidos. A String `ces`, define quais passos serão executados, estes que são divididos pela vírgula e o método `runner.executeCompleteEventSequence` executa de fato o teste.

```
public void testCES07(){
    nomeTeste = "CES07";
    Logs.info(nomeTeste+"#inicio");
    String ces="telaInicialcomTags,pressBotaoClearTag,exibirMsgThisWillDelete," +
        "pressBotaoErase,telaInicial";
    Assert.assertTrue(runner.executeCompleteEventSequence(adaptor, ces,solo,nomeTeste));
    Logs.info(nomeTeste+"#fim");
}
```

Figura 8 - Exemplo de código que executa um teste.
Fonte: Autoria própria.

Cada um dos passos na String `ces` já foram previamente programados na classe `Adaptor` (o adaptador citado na Subseção 3.1.1), ou seja, cada uma das substrings separadas por vírgula são métodos da classe `Adaptor`. Os atributos do método `runner.executeCompleteEventSequence` são: um objeto da classe `Adaptor`, a String `ces`, um objeto da classe `Solo` (que realiza as ações no Android, como pressionar um botão na tela) e a String `nomeTeste`. O objeto `runner`, é uma instância da classe `EventRunner`. Essa classe contém um método que busca na classe `Adaptor` qual método corresponde a ação contida na String `ces`. Possui também o método `executeCompleteEventSequence`, que retorna *true* caso a ação tenha sido executada com sucesso ou *false* caso qualquer erro ocorra durante a execução da ação.

Utilizando o retorno do método `executeCompleteEventSequence`, o método `Assert.assertTrue` (Figura 8), verifica se a ação ocorreu sem erro ou não. Ele compara o resultado retornado com o booleano *true*. Caso a comparação seja verdadeira, a execução da sequência ocorreu com sucesso, caso contrário é gerada uma exceção.

Os métodos da classe `Adaptor` responsáveis por cada uma das ações dentro da `String ces`, Figura 8, são exibidos na Figura 9. Os códigos exibem as seguintes ações: verificar se existe alguma *tag* na tela, pressionar o botão “*Clear Tags*”, procurar pela mensagem “*This will delete all saved searches*” na tela, pressionar o botão “*Erase*” e verificar se está na tela inicial. As ações ocorrem de forma automática no dispositivo. A `annotation@Event` mostra que quando a `String ces` (Figura 8), conter por exemplo a substring “`pressBotaoErase`”, este método será invocado e executado, caso a ação seja executado com sucesso ele retorna *true*, indicando que essa fase do teste ocorreu com sucesso. Caso a ação não seja executada, o método retorna *false*, indicando que nessa fase do teste ocorreu um problema.

```

public class Adaptor
{
    //...
    @Event(label = "telaInicialcomTags")
    public boolean telaInicialcomTags(){

        if(solo.searchButton("Edit"))
            return true;
        return false;
    }

    @Event(label="pressBotaoClearTag")
    public boolean pressBotaoClearTag(){
        if(solo.searchButton("Clear Tags")){
            solo.clickOnButton("Clear Tags");
            return true;
        }
        return false;
    }

    @Event(label="exibirMsgThisWillDelete")
    public boolean exibirMsgThisWillDelete(){
        return solo.searchText("This will delete all saved searches");
    }

    @Event(label="pressBotaoErase")
    public boolean pressBotaoErase(){
        if(solo.searchButton("Erase")){
            solo.clickOnButton("Erase");
            return true;
        }
        return false;
    }

    @Event(label = "telaInicial")
    public boolean telaInicial(){

        if(solo.searchButton("Edit")){
            return false;
        }
        if(!solo.getEditText(0).getText().toString().equals("")){
            return false;
        }
        if(!solo.getEditText(1).getText().toString().equals("")){
            return false;
        }
        return true;
    }

    //...
}

```

Figura 9 - Métodos da classe Adaptor.
Fonte: Autoria própria.

3.1.2 Sequências de Teste Desenvolvidas

Para o projeto, foram desenvolvidas dez sequências de teste para a aplicação Favorite Twitter® Searches. As estruturas dos métodos são semelhantes à da Figura 8 e eles pertencem a classe CestTest. Todos os métodos foram baseados no ESG apresentado na Figura 7. Para a classe Adaptor, foram desenvolvidos doze métodos para os eventos modelados para a aplicação Favorite Twitter® Searches. A estrutura de cada método dessa classe é baseada na Figura 9. Os códigos foram desenvolvidos na IDE Eclipse (ECLIPSE, 2015).

3.2 Desenvolvimento da Ferramenta

Para concretizar o objetivo deste projeto, foi desenvolvida uma ferramenta que coleta dados durante a fase de execução dos testes automatizados. Para sua implementação, foi utilizada a linguagem de programação Java visto que a ferramenta de teste Robotium (ROBOTIUM, 2015) é utilizada apenas nesta linguagem. Outro fator que contribuiu para o uso do Java é o fato de que as aplicações desenvolvidas para Android são desenvolvidas nessa linguagem, como foi citado na Subseção 2.1.1.

A IDE utilizada para o desenvolvimento da ferramenta foi o NetBeans (NETBEANS, 2015), já a IDE utilizada para executar as sequências de teste foi o Eclipse (ECLIPSE, 2015). Para o desenvolvimento do ESG, foi utilizado o diagrama de atividade, criado na ferramenta Astah Community (ASTAH, 2014).

Os dados coletados pela ferramenta são *screenshots* da tela no momento da execução do teste e os eventos que ocorrem no console de *log*, chamado de LogCat (ANDROID, 2015e). Nas sequências de teste, foram adicionados códigos para inserir no *log* as seguintes informações: o início e fim de cada evento de teste e o início e o fim de cada evento presente na String *ces*, ver Figura 8.

A utilização de *screenshots* para visualização de erros pode auxiliar o testador a verificar onde o problema se encontra, juntamente com o *log* do momento da execução daquele teste, mostrando ao testador as informações e exceções (caso haja um defeito) que estavam ocorrendo no momento da captura do *screenshot*.

A ferramenta é capaz de exibir ao testador as *screenshots* capturadas na ordem em que foram tiradas e permite a ele salvá-las em seu computador. Junto com as imagens é exibido o respectivo *log* da execução daquele teste que gerou o

screenshot. Caso um evento falhe executado devido a um defeito, a imagem é exibida com uma borda vermelha. Caso ele seja executado sem falha, a imagem é exibida com uma borda verde.

3.2.1 Arquitetura da Ferramenta

A arquitetura da ferramenta é baseada na Figura 10. A interação 1 ocorre quando o usuário (leia usuário como Programador/Testador) está no processo de desenvolvimento ou teste de uma aplicação Android, pois são nessas fases que o usuário utiliza a IDE para o desenvolvimento. A interação 3, ocorre quando o usuário deseja executar, depurar a aplicação ou uma sequência de teste, pois nesta fase, a IDE precisa se comunicar com o dispositivo (leia dispositivo como emulador ou dispositivo físico) e esta comunicação ocorre por intermédio do ADB (ver subseção 2.3.1). As interações 4 e 6 ocorrem quando o ADB faz qualquer tipo de acesso ao dispositivo, independente de quem requisitou a ação. Esse acesso pode ser gerado pela IDE ou por qualquer aplicação que necessite acessar ou executar alguma tarefa no dispositivo. A interação 2 ocorre da seguinte forma: Ao acessar a ferramenta, o usuário (leia usuário como Programador/Testador) tem duas possibilidades (Figura 11):

1. Carregar os *screenshots* e o arquivo de *log* direto do emulador ou dispositivo físico (opção: Carregar do Cartão SD).

2. Carregar os *screenshots* e o arquivo de *log* de uma pasta do sistema (opção: Carregar da Pasta).

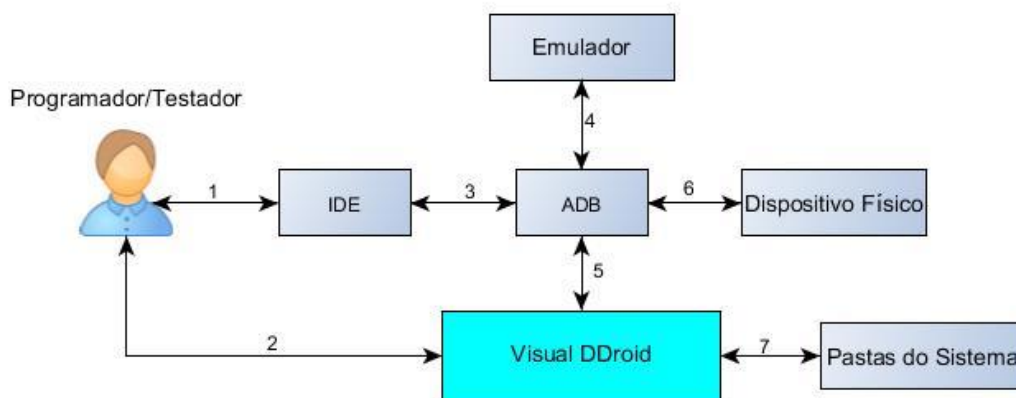


Figura 10 - Arquitetura da Ferramenta Visual DDroid.
Fonte: Autoria própria.

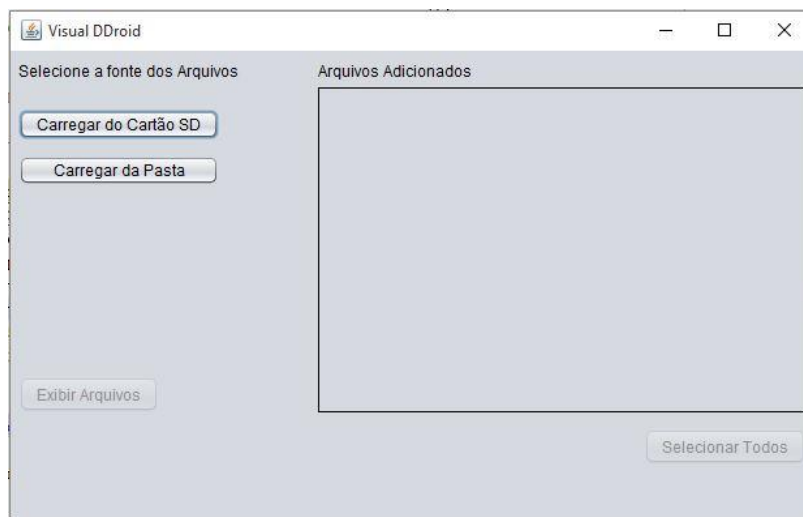


Figura 11 – Visual DDroid, tela inicial.
Fonte: Autoria própria.

Caso se escolha a primeira opção, a ferramenta pergunta ao usuário em qual pasta ele deseja salvar as informações recuperadas. Após se escolher a pasta, a ferramenta recupera os *screenshots* do dispositivo criando uma conexão com o ADB e executando o comando para transferir arquivos do dispositivo para a pasta selecionada, interação 5.

Caso se escolha a segunda opção, a ferramenta pergunta ao usuário de qual pasta ele deseja recuperar as informações, interação 7.

Após a escolha de umas das possibilidades, é exibida uma lista de *checkbox* contendo todas as imagens recuperadas (Figura 12), sendo possível escolher quais *screenshots* serão exibidas, interação 2.

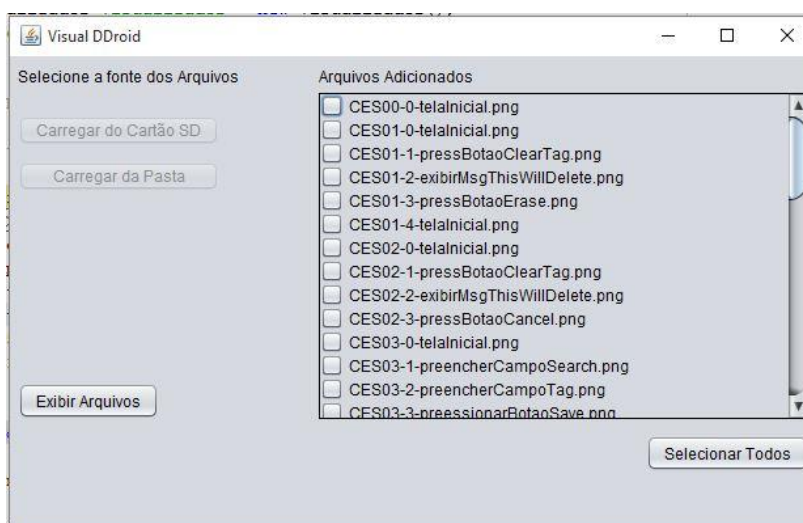


Figura 12 – Visual DDroid, tela inicial com *screenshots* carregados.
Fonte: Autoria própria.

Depois da seleção das imagens, a ferramenta exibe ao usuário a primeira *screenshot* escolhida, juntamente com o *log* do momento da execução da ação que resultou naquela *screenshots* (Figura 14). O *log* pode ser filtrado, permitindo exibir o *log* completo (exibe todas os *logs* que foram capturadas durante a execução da ação que resultou na *screenshot*), o *log* da aplicação (exibe todos os *logs* referentes à aplicação, durante a execução da ação que resultou na *screenshot*) ou sem *log* (não é exibido nenhum *log*).

O usuário pode navegar pelas *screenshots* selecionadas de maneira sequencial (utilizando o botão Avançar ou Voltar) ou escolhendo qual *screenshot* ele deseja exibir, utilizando uma lista de todas as *screenshots* selecionadas. Caso a sequência de teste que gerou a *screenshots* não tenha sido executada devido a um defeito, a imagem é exibida com uma borda vermelha (Figura 13), caso ela tenha sido executada sem falha, a imagem é exibida com uma borda verde (Figura 14).

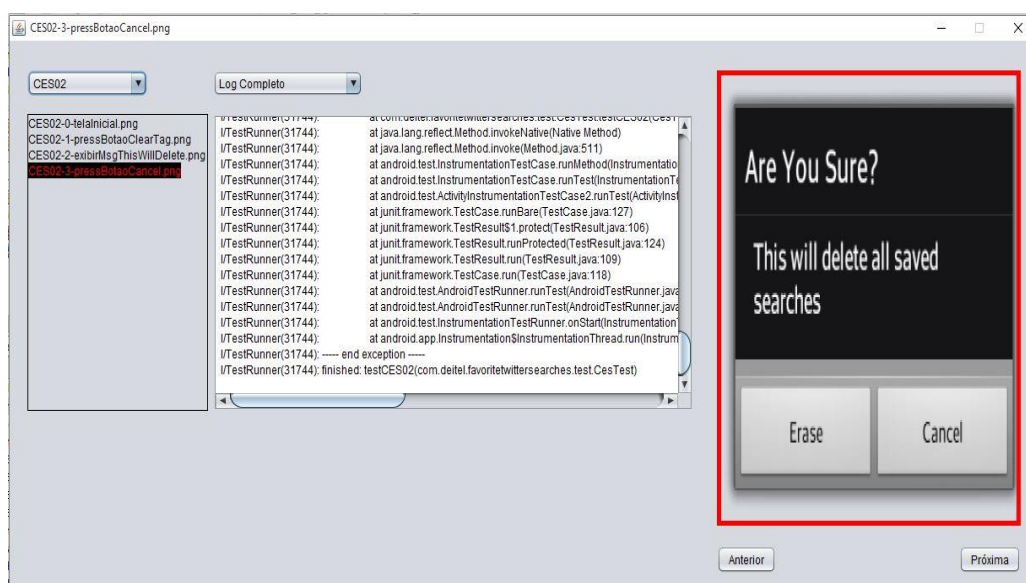


Figura 13 - Exibição do *screenshot* junto com o respectivo *log* na exibição de um defeito.
Fonte: Autoria própria.

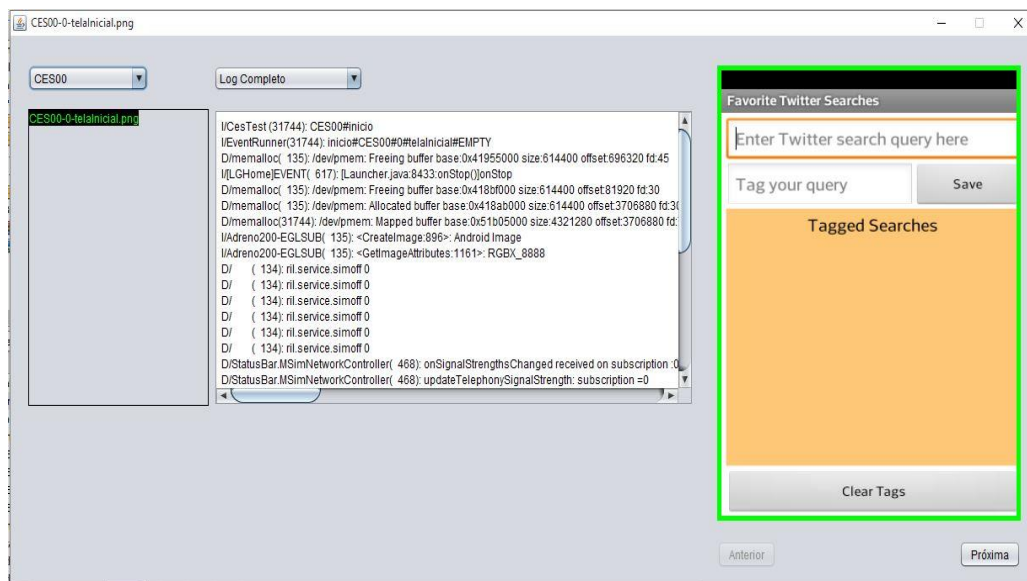


Figura 14 - Exibição do *screenshot* junto com o respectivo *log*.
Fonte: Autoria própria.

As telas da ferramenta Visual DDroid podem ser vistas a partir da Figura 11 até a Figura 14. As dez sequências de teste desenvolvidos juntamente com o código fonte da ferramenta, podem ser encontrados em:

<https://github.com/andreendo/VisualDDroid>

4 AVALIAÇÃO PRELIMINAR DA FERRAMENTA

Nesta seção é apresentada uma avaliação preliminar da ferramenta. A seção foi subdividida nas seguintes atividades: configuração do estudo e execução e análise dos resultados.

4.1 Configuração do Estudo

Para realizar a avaliação preliminar da ferramenta, foram selecionados seis desenvolvedores de uma empresa de desenvolvimento. O teste foi realizado com cada um de forma separada. Foi perguntado a cada um deles o nível que ele apresenta de conhecimento em desenvolvimento de software, com o objetivo de mapear os perfis dos participantes.

A primeira parte do teste foi apresentar uma aplicação Android que gera números aleatórios. Para esta aplicação, foi exibida ao desenvolvedor sua interface e o modo de como a aplicação trabalha. Na sequência, foi apresentada uma visão do teste automatizado para aplicações Android e em seguida foi mostrada a execução de três sequências de teste para a aplicação que gera números aleatórios.

Após isso, foi mostrado como utilizar a ferramenta Visual DDroid, exibindo o processo de como recuperar os *screenshots* e *logs* gerados pela execução das três sequências de teste.

A segunda parte do teste foi mostrar ao participante o funcionamento da aplicação Favorite Twitter® Searches, exibindo sua interface e mostrando todas as funcionalidades da aplicação. Na sequência foi solicitado ao participante para executar as dez sequências de teste desenvolvidas para a aplicação Favorite Twitter® Searches.

Foram inseridos três defeitos, um na classe Adaptor e dois na aplicação Favorite Twitter® Searches. Os defeitos foram inseridos de modo que um defeito não influenciasse na execução das outras sequências de teste. Os três defeitos inseridos foram:

1. Defeito na classe Adaptor: foi inserida a ação de clicar em um botão inexistente na aplicação.

2. Defeito na aplicação: a sequência de teste requer que os dois campos de texto estejam vazios após o clique no botão “*Clear Tags*”, porém no código da aplicação a ação não é realizada.

3. Defeito na aplicação: a sequência de teste requer que uma mensagem de aviso seja exibida após se um dos dois campos de texto estiver vazio (deve-se preencher os dois campos para que a *tag* seja salva sem gerar a mensagem). No código da aplicação foi trocado um “&&” (operador lógico “and”) pelo “||” (operador lógica “or”).

Nessa parte foi explicado para o participante que o objetivo do estudo era localizar os defeitos revelados pelas dez sequências de teste. Para o Defeito 1, foi estipulado um prazo de 15 minutos para que o defeito pudesse ser localizado, para os Defeitos 2 e 3, o prazo máximo para localização do defeito foi de 25 minutos. Caso o defeito não fosse encontrado, era solicitado para que o participante tentasse localizar o próximo defeito.

Para cada participante foi cronometrado o tempo que o mesmo levou para encontrar cada defeito. Foi informado ao participante que as dez sequências de teste estavam corretas, ou seja, os defeitos não poderiam estar localizados na ordem das sequências de testes ou no código da sequência. Também foi informado ao participante que os defeitos estavam ou na classe Adaptor ou no código fonte da aplicação Favorite Twitter® Searches. Foi explicado também que cada participante poderia depurar a sequência de teste.

Após os defeitos serem localizados, foi perguntado ao participante sobre sugestões de melhorias na ferramenta Visual DDroid e quais as facilidades que a ferramenta trouxe ao desenvolvedor.

Na Figura 15 é exibido o fluxo dos passos do estudo realizado.

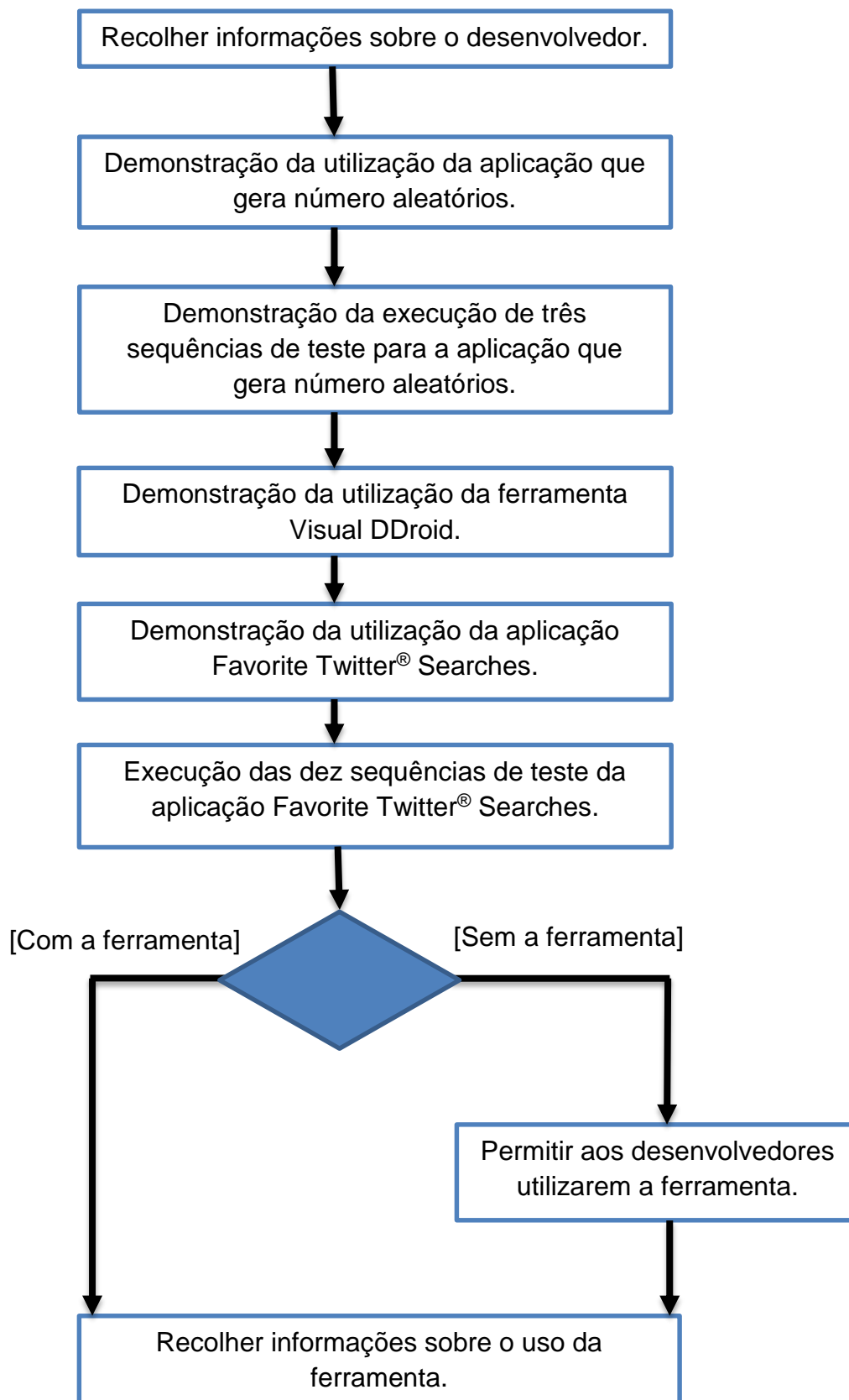


Figura 15 - Passos do estudo realizado.
Fonte: Autoria própria.

4.2 Análise dos Resultados

Após a execução das dez sequências, três delas apresentaram falha. Neste ponto, três participantes utilizaram a ferramenta desenvolvida para poder localizar os defeitos inseridos. Os outros três participantes não puderam utilizar a ferramenta. Cada participante realizou o processo de localização de erros de forma separada. Após cada participante encontrar os três defeitos, (para os participantes que não utilizaram a ferramenta foi permitido que ele a utilizasse) foi solicitado ao mesmo para informar as facilidades que a ferramenta traz para o desenvolvedor e algumas sugestões de melhoria.

Para cada um dos três defeitos, citados na subseção 4.1, foi cronometrado o tempo que cada participante levou para encontrar os defeitos. Os tempos de cada foram registrados no Quadro 2.

Tempo de Localização de Defeitos (min)				
Participante	Usou a Ferramenta?	Defeito 1	Defeito 2	Defeito 3
Participante 1	Sim	1 min 32s	7 min 44s	8 min 30s
Participante 2	Sim	1 min 30s	1 min 59s	1 min 33s
Participante 3	Sim	1 min 00s	5 min 38s	4 min 10s
Participante 4	Não	7 min 48s	7 min 18s	7 min 05s
Participante 5	Não	3 min 40s	8 min 00s	6 min 34s
Participante 6	Não	1 min 00s	7 min 15s	2 min 20s

Quadro 2 - Tempo de Localização de cada defeito.

Fonte: Autoria própria.

Os participantes 1, 2 e 3 utilizaram a ferramenta para tentar localizar os defeitos, já os participantes 4, 5 e 6 não utilizaram a ferramenta. A partir dos dados do Quadro 2, foram montados os gráficos 1, 2 e 3. Ainda utilizando o Quadro 2 foi montado o Quadro 3, este que exhibe a mediana do tempo que os desenvolvedores levaram para encontrar, com e sem a ferramenta.

Mediana do Tempo de Localização dos Defeitos (min)			
Uso da Ferramenta	Defeito 1	Defeito 2	Defeito 3
Utilizando a Ferramenta	1 min 30s	5 min 38s	4 min 10s
Não utilizando a Ferramenta	3 min 40s	7 min 18s	6 min 34s

Quadro 3 - Mediana do Tempo de Localização dos Defeitos.

Fonte: Autoria própria.

O Gráfico 1 exibe o tempo que cada participante levou para encontrar o primeiro defeito. Pela leitura do gráfico verifica-se que os participantes que utilizaram a ferramenta apresentaram um desempenho melhor em relação aos que não utilizaram a ferramenta. Utilizando o valor da mediana, obtida do Quadro 3, percebe-se que há uma diferença de mais de dois minutos no tempo necessário para localizar o defeito. O participante 6 apresentou um desempenho semelhante a aqueles que utilizaram a ferramenta, uma provável explicação para o fato é que esse participante, antes de começar a localizar os defeitos, começou a utilizar a aplicação Favorite Twitter® Searches para verificar todas as suas funcionalidades.

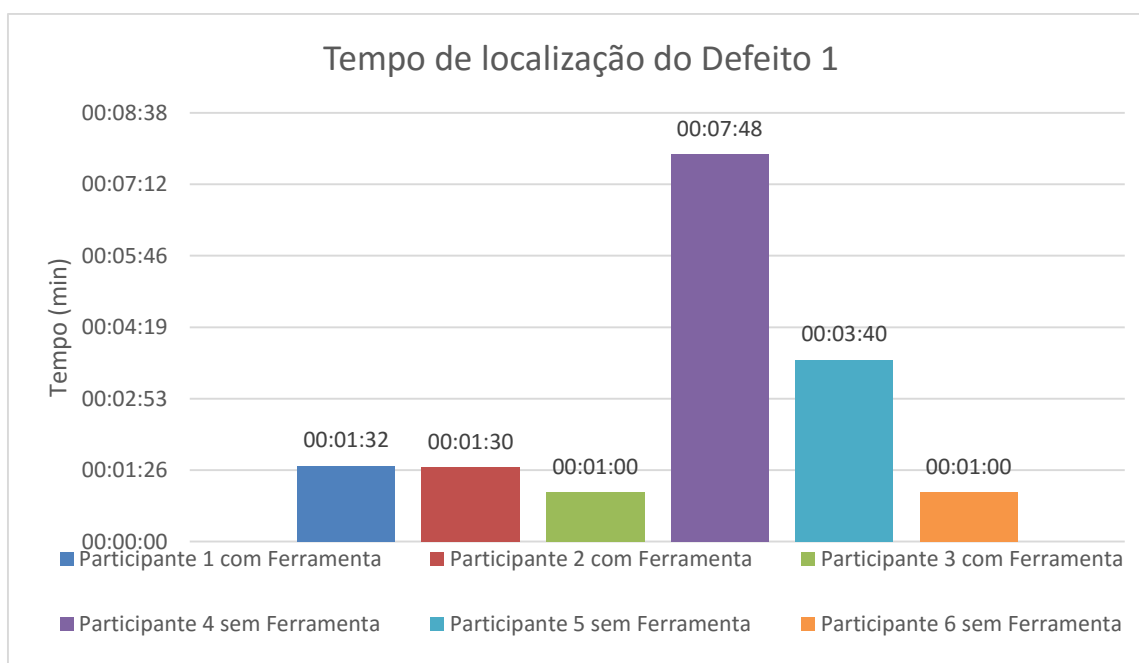


Gráfico 1 - Tempo de localização do Defeito 1.
Fonte: Autoria própria.

O Gráfico 2 exibe o tempo que cada participante levou para encontrar o defeito 2. Apenas pela leitura do gráfico, não fica claro se os participantes que utilizaram a ferramenta apresentam um desempenho melhor em relação aos que não utilizaram. Porém, pela análise da mediana, extraída do 3, verifica-se que os participantes que utilizaram a ferramenta encontraram o defeito 2 em aproximadamente um minuto e trinta segundos antes de quem não utilizou a ferramenta.

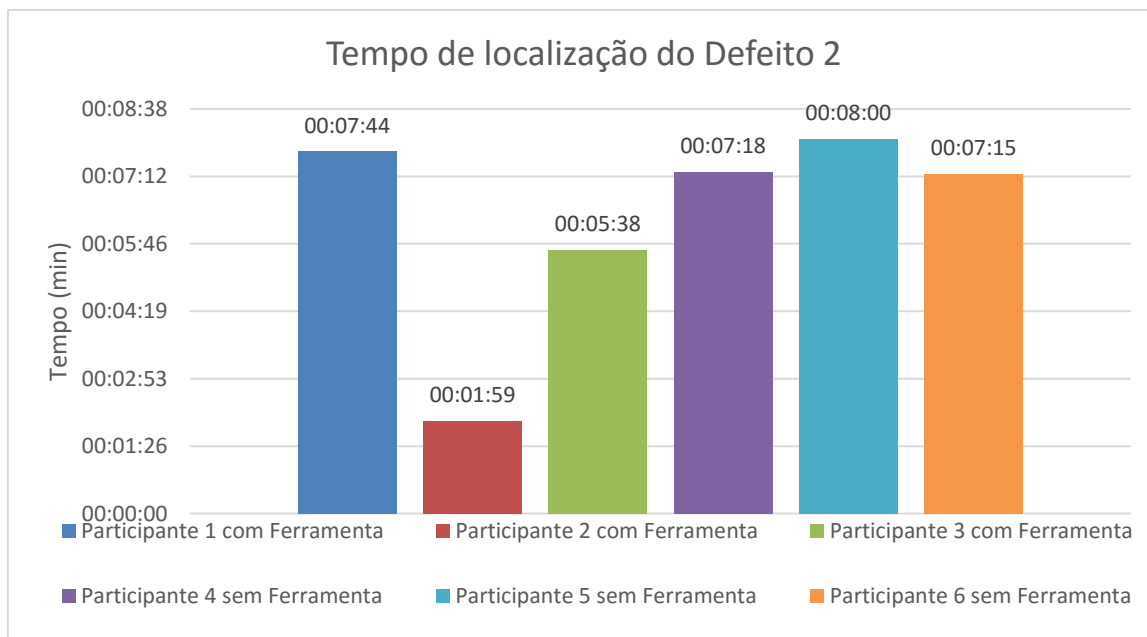


Gráfico 2 - Tempo de localização do Defeito 2.
Fonte: Autoria própria.

O Gráfico 3 exibe o tempo que cada participante levou para encontrar o defeito 3. Apenas pela leitura do gráfico, não fica claro se os participantes que utilizaram a ferramenta apresentam um desempenho melhor em relação aos que não utilizaram. Porém, pela análise da mediana, extraída do Quadro 3, verifica-se que os participantes que utilizaram a ferramenta encontraram o defeito e em aproximadamente dois minutos antes de quem não utilizou a ferramenta. O participante 1 apresentou o pior desempenho entre todos os participantes, mesmo utilizando a ferramenta, uma provável explicação para o fato é que o participante verificou as sequências anteriores (mesmo as que não falharam) antes de tentar localizar o Defeito 3.

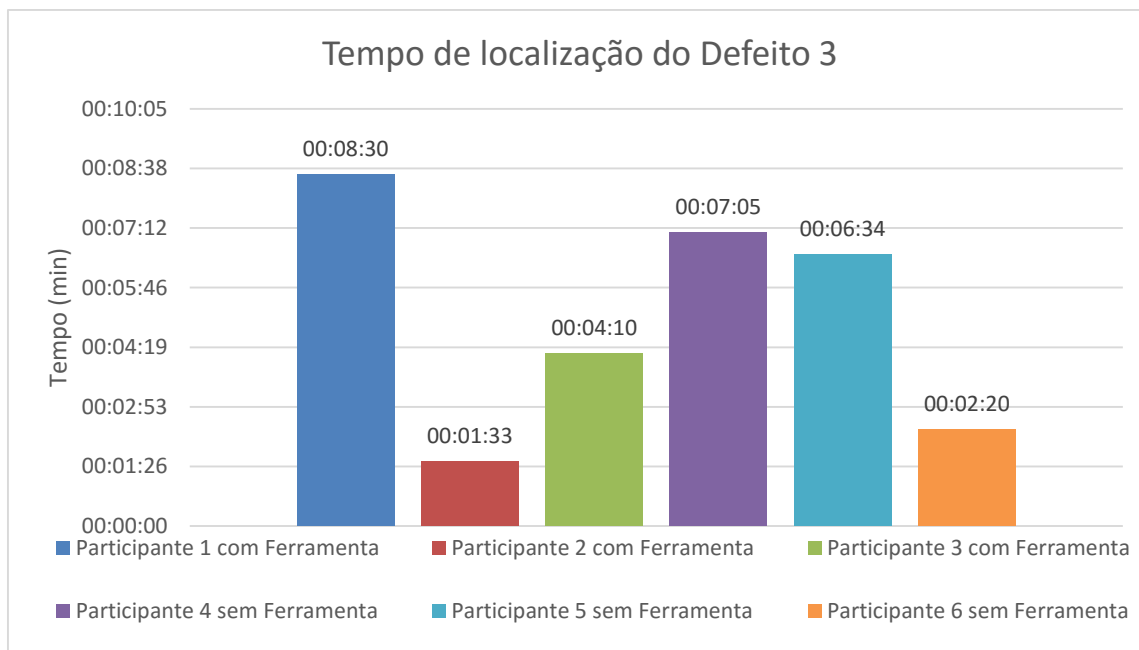


Gráfico 3 - Tempo de localização do Defeito 3.
Fonte: Autoria própria.

O Gráfico 4 exibe a mediana do tempo de localização de cada defeito, com e sem a ferramenta. Pela análise desse gráfico, percebe-se que os participantes que utilizaram a ferramenta conseguiram encontrar os defeitos inseridos mais rápidos do que os participantes que não utilizaram a ferramenta.

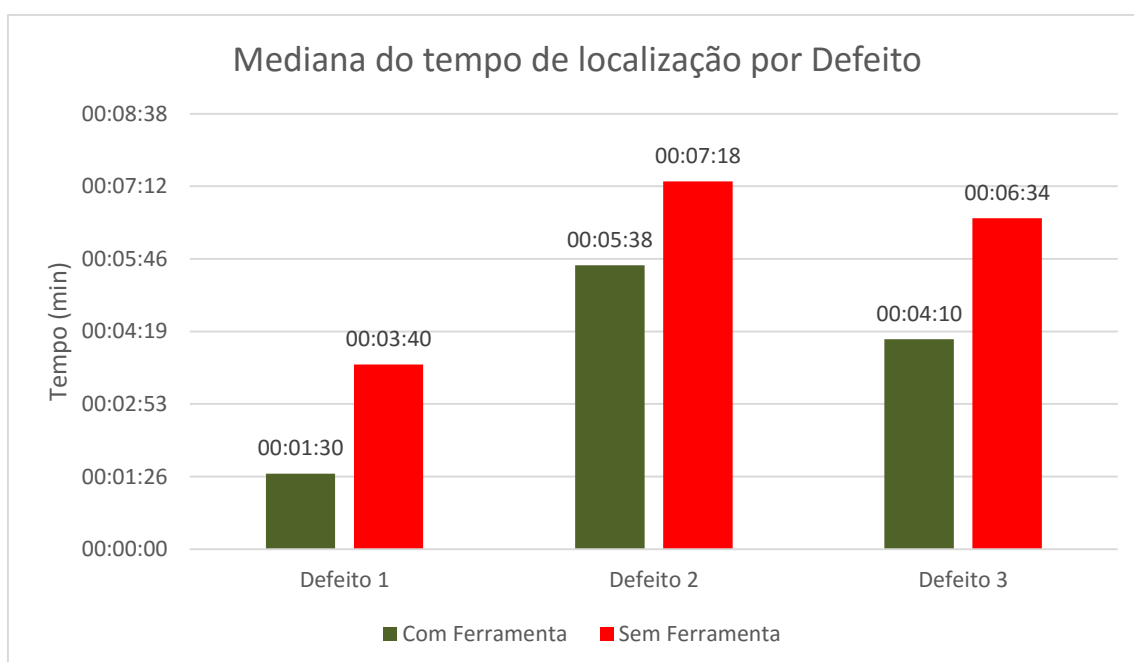


Gráfico 4 - Mediana do tempo de localização por Defeito.
Fonte: Autoria própria.

Em suma, é possível verificar que os participantes que utilizaram a ferramenta Visual DDroid apresentaram um desempenho superior em relação a quem não utilizou a ferramenta, utilizando o parâmetro do tempo total para localizar cada defeito.

4.2.1 Perfil dos Participantes

No início do experimento, foi perguntado à cada participante a quanto tempo o mesmo programava e com quais as linguagens de programação ele já havia trabalhado. Essas informações coletadas podem ser vistas no Quadro 3.

Experiência de cada Participantes		
Participantes	Tempo de Desenvolvimento (em anos)	Linguagens já utilizadas
Participante 1	5 anos	C, Java, C# e HTML
Participante 2	8 anos	C, C# e Delphi
Participante 3	10 anos	C, C++, C#, Pascal, Java e Delphi
Participante 4	4 anos	C, Java, C#, PHP e Javascript
Participante 5	3 anos	C++, Java, PHP e HTML
Participante 6	19 anos	Clipper, Delphi, C++, C# e Javascript

Quadro 4 - Experiência de cada Participante.

Fonte: Autoria própria.

Pelo Quadro 4 verifica-se que todos os participantes apresentam conhecimento em desenvolvimento de software, com no mínimo 3 anos de desenvolvimento. Todos os participantes apresentam conhecimento de programação orientada a objeto, o que auxiliou na localização dos defeitos inseridos na aplicação, visto que a aplicação é desenvolvida na linguagem Java usando a orientação a objeto.

Pelo perfil dos participantes, verifica-se também um fator que poderia prejudicar os resultados do estudo, já que dois dos desenvolvedores nunca trabalharam com a linguagem Java. Porém pelo Quadro 2, verifica-se que ambos participantes obtiveram os melhores desempenhos, um no grupo dos que utilizaram a ferramenta e outro no grupo dos que não utilizaram a ferramenta. Uma provável razão para isto ter ocorrido é o tempo de experiência que ambos apresentam, mais de oito anos. Outro fator que pode mostrar o melhor desempenho de ambos é o fato deles terem utilizado a aplicação Favorite Twitter® Searches após a execução das dez

seqüências de teste, sendo que os outros quatro participantes começaram a procurar os defeitos no código.

4.2.2 Sugestões de Melhorias na Ferramenta

Como foi citado na subseção 4.1, ao fim do estudo, foi perguntado a cada participante algumas sugestões de melhorias na ferramenta e as facilidades encontradas ao utilizar a ferramenta. As facilidades de uso da ferramenta que foram levantadas são:

- Exibição dos *logs*.
- Navegação pelas *screenshots*.
- *Screenshot* exibe o local onde a seqüência falhou.
- Rapidez para encontrar o defeito.

Já as sugestões de melhoria para a ferramenta são:

- Apresentar dois *combobox*, um com as seqüências que falharam e outro com as seqüências que não falharam.
 - Utilizar as setas do teclado para navegar entre as *screenshots*.
 - Poder maximizar a tela.
 - Poder aumentar o tamanho do campo que exibe os *logs*.
 - Na exibição dos *logs*, poder destacar o momento da falha.
 - Exibir somente as seqüências que falharam.
 - Apresentar uma possível solução quando a seqüência falha.

5 CONSIDERAÇÕES FINAIS

Como as vendas de *smartphones* e *tablets* com o sistema operacional Android não param de crescer, o aumento no número de ferramentas de auxílio ao desenvolvimento de aplicações para esta plataforma são uma forma de garantir que as futuras aplicações apresentem um número mínimo de defeitos.

Dentro deste contexto, neste projeto foi desenvolvida uma ferramenta, chamada Visual DDroid, que auxilia testadores e desenvolvedores a localizarem defeitos que ocorrem após a execução de sequências de teste em aplicações Android. Pelo estudo realizado, verificou-se que no contexto do estudo a ferramenta auxiliou na localização dos defeitos. Porém para uma validação mais eficiente é necessário testar a ferramenta com um número maior de desenvolvedores.

Algumas dificuldades foram encontradas no decorrer do projeto, como o aprendizado do *framework* Robotium e compreender a importância do TBM para este projeto.

Como trabalhos futuros, pode-se citar a expansão da ferramenta para trabalhar em outras plataformas, como Windows Phone e iOS. Capturar outras informações do momento da execução de sequências de teste, como o tempo de execução de cada sequência. Realizar a validação da ferramenta com um grupo maior de desenvolvedores. E por último, desenvolver as sugestões feitas pelos participantes no estudo realizado.

REFERÊNCIAS

ABI Research. **\$200 Million Mobile Application Testing Market Boosted by Growing Demand for Automation**, Outubro de 2012. Disponível em: <<https://www.abiresearch.com/press/200-million-mobile-application-testing-market-boos/>>. Acesso em: Março de 2015.

ABLESON, W. F.; SEN, R.; KING, C.; ORTIZ, E. **Android in action**. 3rd ed., Elsevier/Manning Publications Co., Shelter Island, New York, 2012.

ANDROID. **Android Debug Bridge**, 2015 (2015b). Disponível em: <<http://developer.android.com/tools/help/adb.html> >. Acesso em: Abril de 2015.

ANDROID. **Debugging**, 2015 (2015a). Disponível em: <<http://developer.android.com/tools/debugging/index.html>>. Acesso em: Abril de 2015.

ANDROID. **Log**, 2015 (2015f). Disponível em: <<http://developer.android.com/intl/pt-br/reference/android/util/Log.html>>. Acesso em: Outubro de 2015.

ANDROID. **Logcat**, 2015 (2015e). Disponível em: <<http://developer.android.com/tools/help/logcat.html>>. Acesso em: Setembro de 2015.

ANDROID. **Managing Virtual Devices**, 2015 (2015d). Disponível em: <<http://developer.android.com/tools/devices/index.html>>. Acesso em: Abril de 2015.

ANDROID. **Using DDMS**, 2015 (2015c). Disponível em: <<http://developer.android.com/tools/debugging/ddms.html>>. Acesso em: Abril de 2015.

APPCONKIT. **Welcome to Weptun's AppConKit 3.3.3**, 2014. Disponível em: <<http://customer.weptun.de/display/ACK3/Welcome+to+Weptun's+AppConKit+3.3.3>>. Acesso em: Setembro de 2015.

ASTAH. **Astah** (2014). Disponível em: <<http://astah.net/>>. Acesso em: Outubro de 2014.

BELLI F, BUDNIK CJ, WHITE L. **Event-based modelling, analysis and testing of user interactions: approach and case study**. Software Testing, Verification & Reliability 2006.

BLACKBURN, M.; BUSSER, R.; NAUMAN, A. **Why model-based test automation is different and what you should know to get started**. Relatório Técnico, Software Productivity Consortium, 2004.

DEITEL, P. J.; DEITEL, H. M.; DEITEL, A.; MORGANO, M. **Android for Programmers: An App-Driven Approach**. Upper Saddle River, NJ: Prentice Hall, 2012.

ECLIPSE. **Eclipse** (2015). Disponível em: < <https://eclipse.org/>>. Acesso em: Setembro de 2015.

FARTO, Guilherme de Cleva. **Mecanismos de apoio a testes automatizados para aplicações móveis**. 2014. Dissertação – Informática. Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2014.

FORMAN, G. H.; ZAHORJAN, J. **The challenges of mobile computing**. Computer, v. 47, n. 4, p. 38-47, 1994.

GARTNER, Inc. **Gartner Says More than 75 Percent of Mobile Applications will Fail Basic Security Tests Through 2015**, Setembro de 2014. Disponível em: <<http://www.gartner.com/newsroom/id/2846017>>. Acesso em: Março de 2015.

GAO, J.; XIAOYING B.; WEI-TEK T.; UEHARA, T. **Mobile Application Testing: A Tutorial**. Computer, v. 47, n. 2, p. 46-55, 2014.

GRIESKAMP, W.; KICILLOF, N.; STOBIE, K.; BRABERMAN, V. A. **Model-based quality assurance of protocol documentation: tools and methodology**. Software Testing, Verification and Reliability, v. 21, n. 1, p. 55-71, 2011.

HAUSMAN, Christoph; BLITZ, Patrick; BAUMGARTEN, Uwe. **Debugging Cross-Platform Mobile Apps without Tool Break**. Engenharia de Software, 2013 p. 377-390.

IDC. **Smartphone Momentum Still Evident with Shipments Expected to Reach 1.2 Billion in 2014 and Growing 23.1% Over 2013, According to IDC**, Maio de 2014. Disponível em: <<http://www.idc.com/getdoc.jsp?containerId=prUS24857114>>. Acesso em: Março de 2015.

KRAJCI, Iggy.; CUMMINGS, Darren. **Android on x86: An Introduction to Optimizing for Intel® Architecture**. 1º ed. Apress Open, 2013.

LECHETA, R. **Google Android: aprenda a criar aplicações com dispositivos móveis com o Android SDK**. 3rd ed., São Paulo: Novatec, 2013.

LEE, D.; YANNAKAKIS, M. (1996). **Principles and methods of testing nite state machines - a survey**. Proceedings of the IEEE.

MACHADO, Pedro; CAMPOS, José; ABREU, Rui. **MZoltar: Automatic Debugging of Android Applications**. Conferência: Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile, 2013.

MEDNIEKS, Z.; DORNIN, L.; MEIKE, G. B.; NAKAMURA, M. **Programming Android: Java Programming for the New Generation of Mobile Devices**. 2nd ed., O'Reilly Media, 2012.

MUCCINI, H.; DI FRANCESCO, A.; ESPOSITO, P. **Software testing of mobile applications: Challenges and future research directions**. In: Proc. of the 7th

International Workshop on Automation of Software Test (AST), p. 29-35, 2012.

NETBEANS. **NetBeans** (2015). Disponível em: < <https://netbeans.org/>>. Acesso em: Setembro de 2015.

NGUYEN, Tuan A.; CSALLNER, Christoph; TILLMANN, Nikolai. **GROPG: A Graphical On-Phone Debugger**. Conferência: Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile, 2013.

OMG. **Object management group - uml 2.2 superstructure specification**. 2009
Disponível em: <<http://www.omg.org/technology/documents/formal/uml.htm>>.
Acesso em: Março de 2015.

OPEN HANDSET ALLIANCE. **Open Handset Alliance Overview**. Disponível em:
<http://www.openhandsetalliance.com/oha_overview.html>. Acesso em: Março de 2015.

ORACLE. **Java Debug Wire Protocol**. Disponível em:
<<https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jdwp-spec.html>>.
Acesso em: Abril de 2015.

ROBOTIUM. **Robotium – The world’s leading Android test automation framework**, 2015 (2015a). Disponível em: < <https://github.com/robotiumtech/robotium> >. Acesso em: Março de 2015.

ROBOTIUM. **Robotium – Getting Started**, 2015 (2015b). Disponível em:
<<https://github.com/RobotiumTech/robotium/wiki/Getting-Started>>.
Acesso em: Setembro de 2015.

TRETMANS, G. J. **Testing labelled transition systems with inputs and outputs**.
Workshop on Protocol Test Systems VIII - COST 247 Session, Evry,
France, pages 461-476, Evry, France. Institut National des Télécommunications.

UTTING, M.; LEGEARD, B. **Practical model-based testing: A tools approach**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.