ROBERT GOMES MELO

FREVO – UM FRAMEWORK E UMA FERRAMENTA PARA AUTOMAÇÃO DE TESTES

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2016

Robert Gomes Melo

FREVO – Um Framework e uma Ferramenta para Automação de Testes

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

ORIENTADOR(A): **Juliano Manabu Iyoda, Ph.D.**

RECIFE
2016

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

M528f Melo, Robert Gomes
FREVO: um framework e uma ferramenta para automação de testes / Robert Gomes Melo. – 2016.
95 f.: il., fig., tab.

Orientador: Juliano Manabu Iyoda.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.
Inclui referências.

1. Engenharia de software. 2. Automação de testes. I. Iyoda, Juliano Manabu (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2017-44

Robert Gomes Melo

FREVO - Um Framework e uma Ferramenta para Automação de Testes

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 29/02/2016

BANCA EXAMINADORA

Prof. Dr. Alexandre Marcos Lins de Vasconcelos
Centro de Informática / UFPE

Prof. Dr. Sidney de Carvalho Nogueira
Departamento de Estatística e Informática /UFRPE

Prof. Dr. Juliano Manabu Iyoda
Centro de Informática / UFPE
(Orientador)

Este trabalho é dedicado às pessoas mais importantes de minha vida, meus pais, Sr. Agnelo Alves de Melo e Sr.ª Maria Rozete Gomes de Melo, e meu irmão Albert E. Gomes Melo.

AGRADECIMENTOS

Primeiramente, quero agradecer a Deus por me conceder força, perseverança e dedicação suficientes para chegar à reta final do curso.

Especialmente, agradeço a meus pais e à minha tia Roza Maria Gomes Liberato por seu apoio incondicional, pois sem eles eu não teria nem concluído a graduação em Ciência da Computação. Eles me formaram como pessoa, como ser humano e também contribuíram para minha formação acadêmica e profissional, enfrentando diversas dificuldades financeiras.

Eu não posso esquecer de uma amizade especial que começou na infância, passou pelos tempos de escola, pelos tempos de faculdade e que perdura até hoje na área acadêmica e profissional, o meu amigo-irmão Alex Antônio Candido Silva. Posso afirmar que a sua importância foi crucial na conclusão deste trabalho.

Agradeço à minha namorada Edlla pelo incentivo, paciência e apoio incondicional. Agradeço a todos os meus amigos e familiares pelo apoio constante e natural.

Agradeço ao meu professor orientador Juliano Iyoda, por estar presente na etapa mais importante de minha vida acadêmica. Obrigado por me aceitar como seu orientando no meio do curso, pois nos momentos em que eu achava que não ia conseguir, você acreditou e me motivou para que eu pudesse construir um belo trabalho.

Agradeço ao convênio Cin/UFPE Motorola pela oportunidade de poder contribuir para o avanço e desenvolvimento das atividades realizadas no projeto, em especial Eduardo Tavares e Gergana Angelova, duas pessoas as quais tenho um respeito e gratidão enorme, pois diversas decisões desse trabalho tiveram o consentimento e apoio deles.

Finalmente, me sinto realizado de ter construído um trabalho que colabora significativamente nas atividades profissionais de dezenas de pessoas, sinto que os dias de trabalho, as noites não dormidas valeram a pena.

“O único lugar onde sucesso vem antes do trabalho é no dicionário!” (Albert Einstein.)

RESUMO

Com o avanço da computação móvel e da Internet, dispositivos móveis como *smartphones* e *tablets* estão gradativamente se tornando parte essencial de nossas vidas. A quantidade e complexidade dos softwares que funcionam nesses dispositivos trazem grandes desafios para os fornecedores de aplicativos e fabricante de telefones. A automação de testes é vista como uma solução para esses desafios, uma vez que qualidade é um fator crítico para o sucesso do produto. Os *frameworks* atuais de automação de teste têm foco exclusivo na automação de um único teste. Normalmente, a execução de testes acontece em lotes (suítes de testes), e funcionalidades importantes no nível de uma automação da execução da suíte são necessárias, mas os frameworks existentes não proveem funcionalidades eficientes voltadas para o gerenciamento de suítes de testes. Isto torna a execução de uma suíte um trabalho semiautomático e penoso. Neste projeto, propomos uma extensão aos frameworks tradicionais de teste para oferecer maior automação na execução de uma suíte de teste. Funcionalidades como timeout e reexecução (dentre outras) foram propostas para estender os frameworks atuais. Nesse projeto, apresentamos FREVO (*Faster Results, Execution and Visualization*), dois componentes integrados (*framework* e ferramenta) que separam de maneira coesa as atividades de desenvolvimento de *scripts* de testes individuais das atividades de automação e gerenciamento da execução de suítes de testes. O *framework* proposto fundamentalmente adiciona novas propriedades ao *framework* UI Automator e cria um padrão na escrita de testes automáticos. A ferramenta integra-se com esse *framework*, criando um ambiente de gerenciamento de execução de testes maduro e consistente, combinado a uma visualização de resultados de casos de teste intuitiva e usual por meio de uma interface gráfica. Por fim, após a implantação em alguns projetos, conduzimos um estudo de caso dentro do contexto de um projeto de pesquisa realizado pela Motorola Mobility em parceria com o Centro de Informática da Universidade Federal de Pernambuco que constatou que FREVO apresentou um ganho de produtividade em 11 dos 18 produtos testados.

Palavras-chave: **Engenharia de Software. Automação de Testes. Android. UI Automator.**

ABSTRACT

With the advance of mobile computing and the Internet, mobile devices like smartphones and tablets are gradually becoming an essential part of our life. The amount and complexity of software running on these devices bring great challenges to application providers and phone manufacturers. Test automation is regarded as a solution for these challenges, since quality is a critical factor for the product success. The current test automation frameworks have focuses exclusively on automation of a single test. Usually, the tests execution is performed in batches (test suites), so important features related to automation of the execution of a suite is necessary, but the existing frameworks do not provide efficient features aimed at the automation of tests suites. This makes the suite execution a semi-automatic and painful work. In this project, we present FREVO (Faster Results, Execution and Visualization): two integrated components (framework and tool) that separate cohesively the script development of a single test from the automation of management and execution of tests suites. The proposed framework primarily adds new features to the UI Automator framework and defines a standard around the writing of automated test suites. The tool is integrated with this framework, creating a friendly test execution and management environment through a graphical interface. Finally, after the deployment in some projects, we performed a case study inserted in the context of a research project conducted by Motorola Mobility in partnership with the Centro of Informatics of the Federal University of Pernambuco that found that FREVO has presented a gain of productivity in 11 of the 18 tested products.

Keywords: **Software Engineering. Test Automation. Android. UI Automator.**

LISTA DE ABREVIATURAS / SIGLAS

Termo	**Descrição**
ADB	*Android Device Bridge*
API	*Application Programming Interface*
CIn	*Centro de Informática*
FREVO	*Faster Results, Execution and Visualization*
GUI	*Graphical User Interface*
IEEE	*Institute of Electrical and Electronics Engineers*
JAR	*Java ARchive*
NTAF	*NHN Testing Automation Framework*
SDK	*Software Development Kit*
SGBD	*Sistema de Gerenciamento de Banco de Dados*
UI	*User Interface*
XML	*eXtensible Markup Language*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O acesso à informação (principalmente à Internet) tem se tornado uma característica cada vez mais independente de conexões físicas (cabeadas). Nesse contexto, podemos destacar a figura dos smartphones, que são dispositivos móveis e compactos que tomam cada vez mais o espaço de outros dispositivos como notebooks e desktops que, além de serem maiores, boa parte também precisa ter acesso a uma rede de Internet fixa, já que não dispõem de dados móveis como os smartphones.

O acesso à Internet nos smartphones é realizado de maneira instantânea, em tempo real e o mais importante: em qualquer lugar, no carro, no trem, na escola e na rua. Essa condição de poder obter informações de maneira rápida e precisa, de se comunicar com diferentes pessoas, dos mais variados lugares a qualquer momento, tudo ao alcance das mãos, pode ser caracterizada como mobilidade. A facilidade de comunicação e o acesso à informação tem modificado a maneira como as pessoas se relacionam e resolvem problemas. Os smartphones não são mais usados apenas para ler e responder e-mails como antes. A enorme quantidade de aplicativos, para diferentes propósitos, como jogos, redes sociais, e até aplicações com acesso à rede bancária, muda a maneira como as pessoas se relacionam.

Todas essas características precisam estar em pleno funcionamento para que esses produtos conquistem e mantenham seu público. Para isso, as empresas de desenvolvimento de software têm dado considerável importância para atividades de teste de software como sendo uma das principais etapas para melhorar a qualidade de um produto em desenvolvimento, colaborando na revelação de erros o mais cedo possível no ciclo de desenvolvimento e, assim, tentando diminuir os custos com correções. Segundo (PRESSMAN, 2004), o custo para correção de um erro na fase de manutenção é de sessenta a cem vezes maior do que corrigi-lo durante o desenvolvimento. (MYERS, 2004) destaca que essa atividade tem apresentado progressivamente um maior grau de abrangência e de complexidade dentro do processo de desenvolvimento.

A execução de casos de testes pode ser realizada de maneira manual ou automática. A automação das atividades de teste visa o alcance de um processo de testes menos suscetível a erros humanos, mais fácil de reproduzir, mais ágil em atividades repetitivas e menos dependente da interação e/ou interpretação humana. Automação pode ser empregada em diversas atividades de teste, desde o planejamento dos testes, passando pela criação de *scripts* de testes, chegando até a sua execução. A automação de testes funcionais é realizada através

da criação e da execução de *scripts* de teste a partir de ferramentas e *frameworks* que permitem que os *scripts* sejam programados ou simplesmente gravados durante a realização de um teste manual, podendo ser alterados programaticamente. Para programá-los, evidentemente, é necessário conhecer a linguagem de *script* utilizada pela ferramenta e/ou *framework.* Uma vez criados, os *scripts* poderão ser executados, substituindo os testes manuais correspondentes.

No que diz respeito às ferramentas e *frameworks* de automação de testes funcionais, existem *frameworks* voltados para testes de interfaces desktop, interfaces Web, interfaces de dispositivos móveis, entre outros. Podemos citar alguns exemplos: IBM Rational Functional Tester (KELLY, 2006), Selenium (SELENIUM, 2016), Robotium (ROBOTIUM, 2015), Appium (SHAO, 2015), UI Automator (ANDROID, 2016). Independente das linguagens de programação utilizadas nos *frameworks* citados acima, todos eles oferecem funcionalidades para a implementação e execução de um caso de teste, bem como para a execução de uma suíte de testes onde os testadores também precisam definir programaticamente o conjunto de testes a serem executados. Entretanto, em ambientes reais, faz-se necessária a constante criação e/ou edição de suítes de testes, e a atividade de criar/editar suítes de testes programaticamente traz problemas relacionados ao custo de manutenção de suítes de testes. Além disso, esses *frameworks* não têm a capacidade de inferir contextualmente as informações dos resultados dos testes executados, não disponibilizando, portanto, diversas características, tais como a capacidade de disparar comandos para reexecução de testes com falhas/erros ou definir um tempo máximo para a execução de um teste no contexto da execução de uma suíte de testes. A ausência dessas características torna a criação, edição e execução de suítes de testes uma tarefa semiautomática e tediosa para os testadores.

O objetivo geral deste trabalho é contribuir com a automação de testes funcionais em dispositivos móveis Android. Para isto, desenvolvemos uma extensão aos *frameworks* tradicionais de automação, que irá prover algumas facilidades de automação no nível de uma suíte, como: monitoramento da execução e tratamento de exceções, a reexecução de casos de testes que tenham falhado ou que tenham alcançado estados de erro durante a execução, a adição de *timeout* como condição de parada à execução de algum *script* e armazenamento das telas de erro. Para complementar as funcionalidades do *framework* e facilitar seu uso, a definição dos parâmetros acerca da quantidade de reexecuções dos casos de testes que falharam e/ou tiveram algum erro durante a execução, ou o tempo de duração máximo de execução de um *script* pode ser feita através de uma ferramenta interativa. Adicionalmente, através da ferramenta, será possível a manipulação da exibição de detalhes da execução dos

scripts automáticos, como, por exemplo, a comparação entre os resultados esperados e os resultados atuais e também a visualização dos *screenshots* que servem de artefatos das validações executadas.

Esse trabalho está inserido no contexto de um projeto de pesquisa realizado pela Motorola Mobility em parceria com o Centro de Informática da Universidade Federal de Pernambuco. As tecnologias propostas foram materializadas na implementação de uma extensão do UI Automator (ANDROID, 2016), uma plataforma aberta oficial do Android amplamente utilizada pela comunidade de automação de testes e bastante usada dentro dos projetos. A atividade principal exercida por esse projeto de parceria é a realização de diversos tipos de testes nos mais variados modelos de smartphones projetados pela Motorola. Muitos desses testes são executados de maneira automática, sendo executados repetidas vezes ao longo do ano. Diferentes falhas podem ser detectadas: falhas essas devido à aplicação sob teste ou, em alguns casos, devido a mudanças no ambiente de execução, como ausência de rede Internet, ausência de rede celular, *force close* (ação que ocorre quando uma aplicação é finalizada anormalmente), ou condição inesperada de exceção pelo *script* de teste. Tais falhas podem acabar comprometendo uma execução deixando o *script* automático parado esperando indefinidamente até que o testador responsável pela execução note o problema e então desbloqueie a execução (no melhor caso, sem perca de resultados parciais da execução), ou tenha que reiniciar a mesma (pior caso, pois os resultados parciais da execução são perdidos). Este cenário industrial ilustra bem as limitações dos *frameworks* atuais na prática.

Neste contexto do projeto CIn-Motorola, realizamos um estudo de caso em que o *framework* e a ferramenta foram aplicadas no teste de 18 produtos. Em nosso estudo de caso, constatamos ganhos de produtividade oriundos da adoção da ferramenta para a execução dos testes automáticos em 11 dos 18 produtos testados.

Em resumo, as principais contribuições desta dissertação são:

- Proposta de extensão dos *frameworks* de automação atuais para promover o nível de automação ao nível de uma suíte de testes, ao invés de um teste individual;
- Materialização da proposta como uma extensão do *framework* UI Automator para o *framework* FREVO para possibilitar uma execução de suítes de testes mais inteligente, permitindo: monitoramento automático da execução, tratamento de exceções, reexecução de casos de testes que tenham falhado ou que tenham alcançado estados de erro, introdução de *timeout* como condição de parada à execução de algum *script*, armazenamento das telas de erro;

- Desenvolvimento de uma ferramenta interativa FREVO para permitir o testador definir e configurar a execução de uma suíte de testes utilizando todas as facilidades do *framework* de forma simples;
- Estudo de caso comparando a produtividade na execução de suítes de forma tradicional (semiautomática) versus a execução utilizando FREVO e, consequentemente, o UI Automator estendido, onde constatamos um ganho de produtividade em 11 dos 18 produtos testados.

Os capítulos subsequentes deste trabalho estão organizados da seguinte forma. No Capítulo 2, o referencial teórico que serve de base para a elaboração deste trabalho é apresentado. No Capítulo 3, os objetivos do trabalho são apresentados em detalhes, bem como a construção dos dois sistemas integrados: o *framework* FREVO (extensão ao UI Automator) e a Ferramenta de Execução de Testes FREVO. No Capítulo 4, acontece a avaliação empírica do trabalho, exibindo os resultados obtidos com a adoção da solução proposta. No Capítulo 5 é realizado um estudo comparativo com alguns trabalhos relacionados ao proposto. Por fim, no Capítulo 6, discutem-se as conclusões e possíveis desdobramentos deste trabalho.

2 FUNDAMENTOS

2.1 TESTE DE SOFTWARE

Uma vez implementado o código de uma aplicação, o mesmo deve ser testado para descobrir tantos erros quanto possível, antes da entrega do produto de software ao seu cliente.

O teste é um conjunto de atividades que podem ser planejadas antecipadamente e conduzidas sistematicamente. Por essa razão, um gabarito para teste de software - um conjunto de passos no qual podemos incluir técnicas de projeto de casos de teste e métodos de teste específicos - deve ser definido para o processo de software.

O significado do verbete teste segundo o dicionário Aurélio FERREIRA (1986) é o seguinte:

> "S. m. 1. Exame, verificação ou prova para determinar a qualidade, a natureza ou o comportamento de alguma coisa, ou de um sistema sob certas condições. 2. Método, processo, procedimento ou meios utilizados para tal exame, verificação ou prova." A palavra teste deriva da palavra inglesa test, que por sua vez tem origem na palavra latina testum, que representa um pote de barro utilizado para determinar a presença ou medir o peso de vários elementos (HETZEL, 1988).

Para MALDONADO (2001), teste de software é uma atividade de verificação e validação do software e consiste na análise dinâmica do mesmo, isto é, na execução do produto de software com o objetivo de verificar a presença de erros no produto e aumentar a confiança de que o mesmo está correto.

Além da identificação e consequente correção de erros, os testes têm outros benefícios: o teste demonstra que o software funciona de acordo com a especificação (requisitos e projeto), bem como trata os requisitos não funcionais que são necessários/desejados no produto de software.

Segundo PRESSMAN (2001) "O teste de software é um elemento crítico da garantia de qualidade de software e representa a última revisão da especificação, do projeto e da codificação". Por isso, o processo de teste deve ser executado durante todo o ciclo de vida do projeto do software. Do ponto de vista de custo, segundo MEYERS (1979), aproximadamente 50% do tempo e mais de 50% do custo total de desenvolvimento de um produto de software é gasto em teste; HARROLD (2000) indica que nos casos de software crítico essa porcentagem é ainda maior. PRESSMAN (2001) diz que 40% do esforço de desenvolvimento é gasto em teste e que em projetos de software crítico (por exemplo, controle de voo ou monitoramento

de reatores nucleares) o seu custo pode chegar a três vezes o custo de todas as outras fases de desenvolvimento.

Para HETZEL (1988) “Testar é o processo de certificar que o programa faz o que era suposto fazer”.

Para MYERS (1979) “Testar é o processo de executar um programa ou sistema com o objetivo de encontrar erros”.

Alguns importantes princípios de teste são destacados por PRESSMAN (2002) e PFLEEGER (2004):

- Teste completo não é possível. Mesmo para sistemas de tamanho moderado, pode ser impossível executar todas as combinações de caminhos durante o teste.
- Teste deve ser conduzido por terceiros. Os testes conduzidos por outras pessoas que não aquelas que produziram o código têm maior probabilidade de encontrar defeitos. O desenvolvedor que produziu o código pode estar muito envolvido com ele para poder detectar defeitos mais sutis.
- Testes devem ser planejados bem antes de serem realizados. Um plano de testes deve ser utilizado para guiar todas as atividades de teste e deve incluir objetivos do teste, abordando cada tipo (unidade, integração e sistema), como serão executados e quais critérios a serem utilizados para determinar quando o teste está completo.

Uma vez que os testes estão relacionados aos requisitos dos clientes e usuários, o planejamento dos testes pode começar tão logo a especificação de requisitos tenha sido elaborada. À medida que o processo de desenvolvimento avança (análise, projeto e implementação), novos testes vão sendo planejados e incorporados ao plano de testes.

2.2 PROCESSO DE TESTE DE SOFTWARE

Para MALDONADO (2001) e PFLEEGER (2004) o processo de teste envolve quatro atividades principais:

Planejamento de Testes: trata-se da definição das atividades de teste, das estimativas dos recursos necessários para realizá-las, dos objetivos, estratégias e técnicas de teste a serem adotadas e dos critérios para determinar quando uma atividade de teste está completa.

Projeto de Casos de Testes: é a atividade chave para um teste bem-sucedido, ou seja, para se descobrir a maior quantidade de defeitos com o menor esforço possível. Os casos de

teste devem ser cuidadosamente projetados e avaliados para tentar obter um conjunto de casos de teste que seja representativo e envolva as várias possibilidades de exercício das funções do software (cobertura dos testes).

Execução dos testes: consiste na execução dos casos de teste e registro de seus resultados.

Avaliação dos resultados: detectadas falhas, os defeitos deverão ser procurados. Não detectadas falhas, deve-se fazer uma avaliação final da qualidade dos casos de teste e definir pelo encerramento ou não de uma atividade de teste.

2.3 TÉCNICAS DE TESTE DE SOFTWARE

Teste de software é uma das etapas mais importantes no ciclo de vida do desenvolvimento de um software, tendo em vista que ela está diretamente ligada com satisfação do cliente ao receber um produto de qualidade. Para que o software seja bem testado, é preciso definir o melhor conjunto de casos de teste, bem como as melhores técnicas para cada etapa do ciclo de desenvolvimento. Com isto, exercitamos as principais funcionalidades do sistema, garantindo que os resultados gerados da execução de cada passo do caso de teste correspondam a um estado esperado do teste e, consequentemente, concluir se o objetivo do teste foi alcançado.

Para isso, várias técnicas de teste têm sido utilizadas para projetar casos de teste. Segundo MALDONADO (2001), essas técnicas podem ser classificadas, segundo a origem das informações utilizadas para estabelecer os objetivos de teste, em, dentre outras categorias, técnicas funcional, estrutural ou baseada em máquinas de estado.

Para PRESSMAN (2002), os testes caixa-preta são empregados para demonstrar que as funções do software estão operacionais, que a entrada é adequadamente aceita e a saída é corretamente produzida e que a integridade da informação externa (uma base de dados, por exemplo) é mantida.

MYERS (2004) compara essa abordagem de teste como uma caixa-preta, uma vez que a pessoa responsável por executar os testes não tem acesso ao código fonte do sistema. Portanto, o intuito dessa técnica é encontrar situações onde o comportamento do sistema não satisfaça as condições esperadas na especificação.

Ainda para PRESSMAN (2002) os testes estruturais ou caixa-branca estabelecem os objetivos do teste com base em uma determinada implementação, verificando detalhes do código buscando encontrar erros em sua estrutura interna através da:

- Execução de testes que percorram cada caminho lógico do código pelo menos uma vez;
- Execução de testes que exercitem estados de verdadeiro e falso, bem como decisões lógicas;
- Execução de testes que exercitem todas as estruturas de dados utilizadas na implementação.

Por fim, para MALDONADO (2001) os testes baseados em máquinas de estado são projetados utilizando o conhecimento subjacente à estrutura de uma máquina de estados para determinar os objetivos do teste.

2.4 NÍVEIS DE TESTE E TIPOS DE TESTE

Os testes podem ser executados em diferentes momentos durante o ciclo de vida de desenvolvimento do software, podendo variar entre os níveis conforme ilustrado na Figura 2.1, que exibe o modelo de desenvolvimento em V, oriundo do modelo de desenvolvimento em cascata.

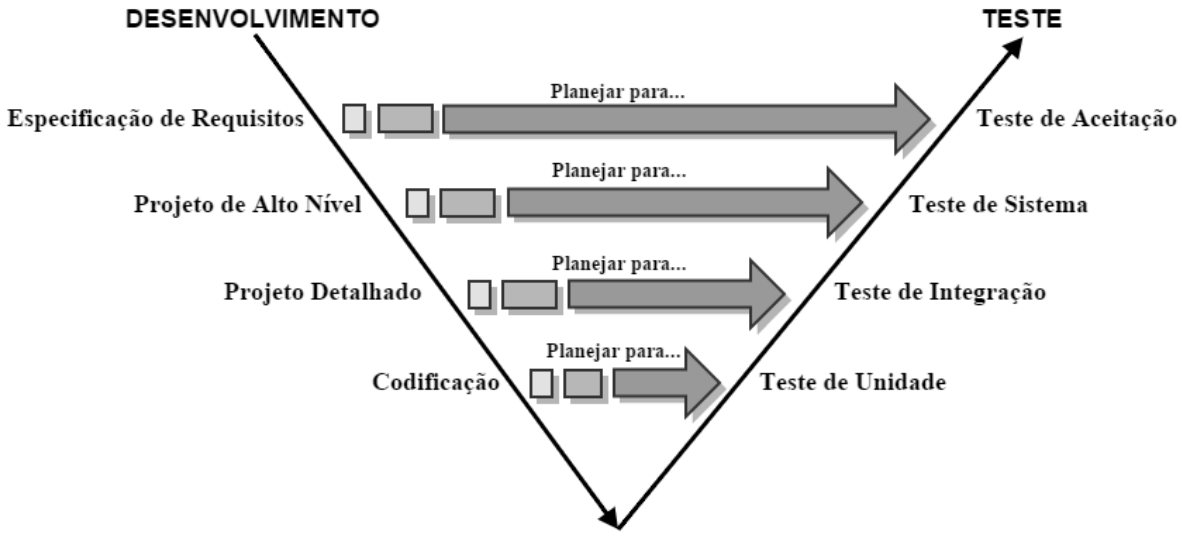

Figura 2.1: Modelo V, paralelismo entre as atividades de desenvolvimento e teste de software.
Fonte: (CRAIG e JASKIEL, 2002).

2.4.1 Níveis de Teste

Segundo CRESPO (2004), nível de teste é a definição da fase de desenvolvimento do software em que o teste será testado. Ele comenta que o nível de teste é uma das etapas na elaboração do planejamento do teste e que depende da fase de desenvolvimento do software. Os diferentes níveis de teste são:

Teste de Unidade: geralmente são testes de caixa branca que têm por objetivo verificar um determinado elemento que possa ser localmente tratado como uma unidade de implementação. "O menor pedaço de software que um desenvolvedor cria" é a definição dada por COPELAND (2004). Segundo TIAN (2005), “uma unidade pode corresponder a uma função, um procedimento ou uma sub-rotina para linguagens de programação estruturadas tradicionais como C, PASCAL ou FORTRAN, ou corresponder a um método numa linguagem orientada a objeto como C++, Java e Smalltalk”.

Teste de Integração: são considerados como testes de caixa-branca por ainda serem realizados por desenvolvedores, que têm por objetivo verificar como as unidades desenvolvidas separadamente se comportam corretamente quando em conjunto com as demais unidades já integradas. Esses testes não se preocupam com um comportamento de uma unidade específica como o teste de unidade descrito anteriormente (RAINSBERGER, 2005). Apesar dos testes de unidades isoladas (teste unitário) terem sido realizados com êxito, ainda não são suficientes para garantir que a combinação com outras unidades se comporte de maneira esperada. Também é importante destacar que os testes de integração devem ser realizados somente após o êxito dos testes de unidade (BURNSTEIN, 2003).

Teste de Sistema: DUSTIN (2003) comenta que diferentemente dos testes unitários e dos testes de integração, é preferível que os testes de sistemas sejam executados em um ambiente isolado e com configuração idêntica ou similar ao ambiente encontrado em produção. BURNSTEIN (2003) também espera que, neste nível, já tenha ocorrido a integração das unidades com sucesso, e que o sistema ou parte dele esteja pronto para ser testado por uma equipe que não conheça as principais características ou funcionalidades do sistema (requisitos funcionais), bem como aspectos no que diz respeito ao desempenho (requisito não funcional). BURNSTEIN (2003) ainda destaca que os testes de sistema podem ser encarados como uma preparação para os próximos testes que serão os testes de aceitação.

Teste de Aceitação: os testes de aceitação têm por objetivo validar o produto, ou seja, verificar se este atende aos requisitos especificados inicialmente e as necessidades atuais do cliente. Geralmente é executado em um ambiente real ou se possível o mais semelhante

possível ao ambiente real de execução. Normalmente um grupo de usuários finais é responsável por realizar essa execução e não pode ser de responsabilidade do time de desenvolvimento. MYERS (2004) define que o teste de aceitação é realizado para determinar se o produto satisfaz as necessidades do cliente. Os testes de aceitação podem ser divididos em testes funcionais e não funcionais.

2.4.2 Tipos de Teste

Para CRESPO (2004), os tipos de teste referem-se às características do software que podem ser testadas. Um tipo de teste é focado em um objetivo particular do teste, isto é, dependendo dos objetivos, o teste é organizado de forma diferente.

MALDONADO (2001) destaca alguns dos principais tipos de teste de software:

Testes de regressão: executam novamente um subconjunto de testes previamente executados. Seu objetivo é assegurar que alterações em partes do produto não afetem as partes já testadas.

Teste funcional: verifica se o sistema integrado realiza as funções especificadas nos requisitos.

Teste de desempenho: verifica se o sistema integrado atende aos requisitos não funcionais do sistema (eficiência, segurança, confiabilidade, etc.).

Teste de instalação: algumas vezes o teste de aceitação é feito no ambiente real de funcionamento, outras não. Quando o teste de aceitação for feito em um ambiente de teste diferente do local em que será instalado, é necessário realizar testes de instalação.

2.5 ERRO, DEFEITO E FALHA

No processo de desenvolvimento de software é comum encontrar os termos defeito, erro e falha. Embora esses termos estejam diretamente ligados ao processo de desenvolvimento de software, muitos profissionais desconhecem seus significados e suas diferenças.

O IEEE – *Institute of Electrical and Electronics Engineers* – (IEEE 610, 1990), apresenta o erro como uma manifestação concreta de um defeito num artefato de software. A diferença entre o valor obtido e o valor esperado ou qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro.

O defeito em um sistema pode ocorrer devido à omissão de informações, definições de dados ou comandos/instruções incorretas, dentre outros fatores. Se um determinado defeito não for encontrado, pode causar uma falha no funcionamento do software.

De acordo com o IEEE 610 (1990) o defeito é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto.

Já a falha, de acordo com o IEEE 610(1990) é o comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha.

A Figura 2.2 expressa a diferença entre esses conceitos.

Figura 2.2: Diferenças entre Defeito, Erro e Falha.
Fonte: Defeito x Erro x Falha, disponível em http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035.

Defeitos fazem parte do universo físico (a aplicação propriamente dita) e são causados por pessoas, por exemplo, através do mal-uso de uma tecnologia. Defeitos podem ocasionar a manifestação de erros em um produto, ou seja, a construção de um software de forma diferente ao que foi especificado (universo de informação). Por fim, os erros geram falhas, que são comportamentos inesperados em um software que afetam diretamente o usuário final da aplicação (universo do usuário) e pode inviabilizar a utilização de um software.

2.6 TESTE AUTOMÁTICO

A principal motivação da automação de testes é descrita por KANER (2001) como a utilização de estratégias e ferramentas com o objetivo de reduzir o envolvimento humano nas atividades manuais e repetitivas de teste de software.

A fim de tornar o processo de teste mais ágil, ou seja, menos repetitivo e consequentemente menos suscetível a erros humanos, a automação dessas atividades é de fundamental importância, uma vez que através da automação será mais fácil reexecutar uma ou mais funcionalidades automaticamente de forma independente da interferência e intepretação humana (QUENTIN, 1999).

Através da automação é possível a execução de testes de regressão com maior amplitude e profundidade. Por exemplo, em um cenário onde se deseja testar um sistema em aspectos relacionados à performance e stress através da simulação do acesso simultâneo de 1.000 usuários, testes manuais seriam incapazes de atender as necessidades mínimas exigidas para esses tipos de testes. Uma vez que seria necessário um número elevado de passos ou procedimentos repetitivos que levaria a pessoa responsável pela realização dos testes a ficar cansada podendo cometer erros que podem comprometer os resultados finais.

Esses, dentre outros fatores, fazem com que a aplicação de um processo de automação de testes seja um investimento bastante atraente. Automação irá eliminar ou reduzir a interação humana e, com isso, a possibilidade de ocorrerem possíveis alterações de resultados devido à interpretação errada de um caso de teste, ou um erro humano causado por um estado emocional ou até mesmo por cansaço. Enquanto que quando um caso de teste está automatizado, ele sempre irá executar os passos pré-definidos e seu resultado nunca será alterado pelos fatores humanos supracitados.

Adicionalmente, no processo de desenvolvimento de software dinâmico, é cada vez mais difícil que campanhas de testes manuais consigam acompanhar a grande quantidade e o volume de testes exigido durante o ciclo do desenvolvimento, obrigando em alguns casos que o produto seja lançado sem ter completado o processo de testes devido a limitações de tempo. Outra característica acerca de automação de testes que merece destaque é no que diz respeito ao tempo de execução de testes automatizados, uma vez que esses sempre são mais rápidos que os testes manuais, além de que podem ser planejados para serem executados a qualquer hora, maximizando assim o tempo quando executados durante a noite por exemplo.

Entretanto, vale salientar que a implantação de um processo de automação de testes não é trivial. É preciso que as organizações tenham conhecimento das etapas necessárias e dos riscos envolvidos na introdução de ferramentas de automação. Além disso, o tempo gasto para automatizar um caso de teste precisa ser recompensado pelo tempo salvo pela sua execução automática, ou então a automação só trará prejuízos.

2.7 FERRAMENTAS PARA AUTOMAÇÃO DE TESTES

Vários tipos de ferramentas estão disponíveis para apoiar os diferentes propósitos de testes e etapas do desenvolvimento de software (DUSTIN, 2003). Em FEWSTER et al. (1999) são descritos alguns tipos (Figura 2.3):

- Ferramentas de Projeto de Testes (*Test Design Tools*): usadas para derivar dados de teste a partir de uma especificação;
- Ferramentas de Projeto Lógico (*Logical Design Tools*): utilizadas para gerar casos de teste;
- Ferramentas de Projeto Físico (*Physical Design Tools*): usadas para manipular dados existentes ou gerar dados de teste;
- Ferramentas de Gerenciamento (*Test Management Tools*): suportam o plano de testes, possibilitando manter a relação dos testes que serão executados. Ferramentas para ajudar na rastreabilidade de testes e ferramentas de rastreamento de defeitos também estão incluídas nesta categoria;
- Ferramentas de Análise Estática (*Static Analysis Tools*): usadas na detecção de defeitos em código fonte, sem a necessidade de execução do código;
- Ferramentas de Cobertura (*Coverage Tools*): usadas para avaliar o quanto a aplicação testada tem sido exercitada nos testes;
- Ferramentas de Depuração de Código (*Debugging Tools*): usadas na depuração de código, portanto, não são consideradas, a rigor, ferramentas de teste. No entanto, oferecem suporte eficiente para a correção de defeitos encontrados em testes;
- Ferramentas de Análise Dinâmica (*Dynamic Analysis Tools*): usadas para avaliar o comportamento do sistema enquanto o software é executado;
- Simuladores (*Simulators*): simulam condições que permitem a execução de partes de um sistema;
- Ferramentas de Teste de Desempenho (*Performance Testing Tools*): usadas para medir o tempo de resposta do software;
- Ferramentas de Execução e Comparação (*Test Execution and Comparison Tools*): usadas para automatizar a execução de testes, comparando as saídas do teste com as saídas esperadas. São utilizadas em todos os estágios

(testes unitários, testes de integração, testes de sistema e testes de aceitação). Ferramentas de *Record and Playback* são ferramentas de execução e comparação;

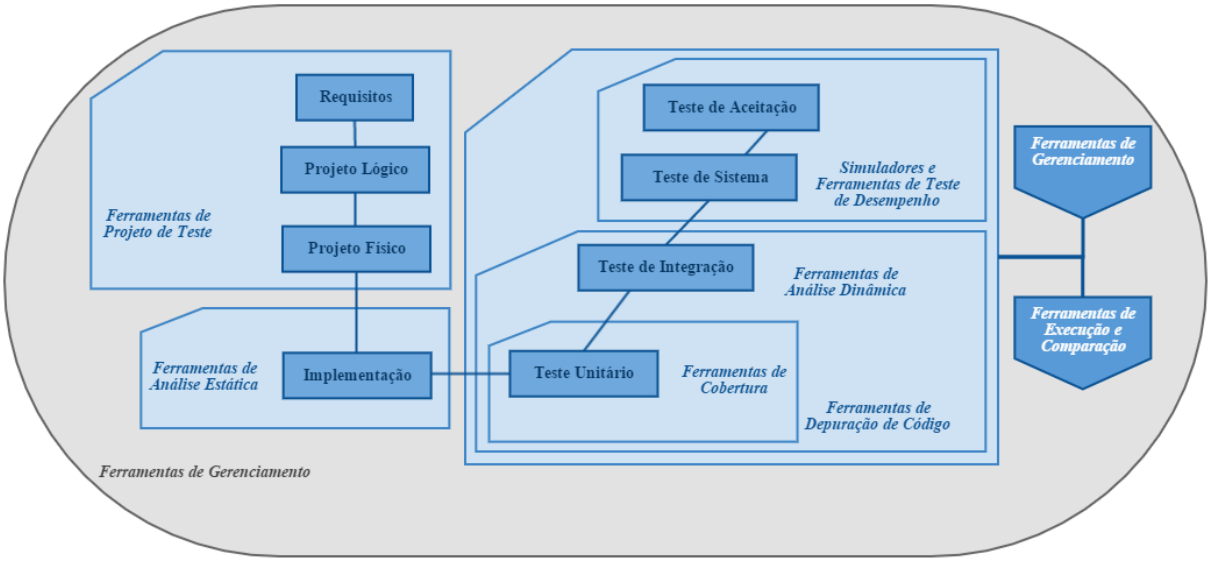

Figura 2.3: Tipos de Ferramentas de Suporte a Testes.

2.8 SCRIPTS DE AUTOMAÇÃO

Como este trabalho aborda a automação de testes funcionais em dispositivos móveis Android, faz-se necessário o entendimento acerca de como esse tipo de automação acontece, bem como, como se dá sua execução.

Nas seções que seguem, ilustraremos o funcionamento dos *frameworks* através da descrição do UI Automator e JUnit. Estes *frameworks* são amplamente adotados na comunidade de desenvolvimento Android e Java, respectivamente e ilustram suas facilidades e limitações. Porém, a descrição que segue é também (guardadas as devidas transformações sintáticas e algumas sutilezas de implementação) comum a todos os *frameworks* de automação de teste.

A automação de testes ocorre pelo desenvolvimento de *scripts* de testes suportados por um determinado *framework*. O *framework* em questão é o UI Automator, um *framework* para o desenvolvimento de testes de interface gráfica do usuário, que possibilita a abstração de ações de um ser humano interagindo com o dispositivo móvel sob teste, como por exemplo, gestos de toque (*click*), pressionar (*longPress*) arrastar (*dragTo*). É necessário conhecer a linguagem e o *framework* no qual os scripts são desenvolvidos para que os *scripts* possam ser escritos e então executados substituindo os testes manuais existentes.

Dessa forma, pode-se concluir que *scripts* são encarados como pequenos programas escritos em uma linguagem de *script* específica com o intuito de testar um conjunto de funcionalidades no software sob teste. Com a adição de *scripts* de teste, é possível ter um ganho no que diz respeito à profundidade e a quantidade de testes executados, de modo que através da criação de *scripts* automáticos torna-se possível a reprodução dos testes automaticamente a qualquer momento. Ou seja, um conjunto de testes pode ser executado quantas vezes for necessário e sob diferentes tipos de configurações de ambiente, como versão de hardware e software.

Desse modo, quando a interface gráfica da aplicação sob teste é terminada, ou considerada como pronta (*software lock down*), a atividade de criação ou de adaptação de *scripts* é aconselhada a acontecer. Desenvolver scripts automáticos para teste funcionais de interface gráfica é uma atividade complexa devido à grande quantidade de variações de hardware, software e também de componentes gráficos utilizados pelos aplicativos. Assim, faz-se necessário o conhecimento avançado do *framework* a fim de desenvolver scripts bem escritos que possibilitem reuso e que necessitem o mínimo possível de adaptação futura devido a mudanças na interface gráfica.

Além dos passos principais do teste, o corpo do *script* contém pontos de verificação, validações ou comparações (*Asserts*). Isto permite comparar o resultado esperado com o resultado atual da aplicação. Dessa forma, é feita uma avaliação se o teste passou ou falhou baseado nos resultados das validações das suas verificações.

Uma vez adotadas ferramentas para automação da execução dos testes, os resultados não aparecem instantaneamente (QUENTIN, 1999). É sabido que existe um aumento no esforço para desenvolver *scripts* de teste automático, diferentemente de testes manuais. Muitas vezes, o tempo levado para organizar o ambiente e iniciar a execução automática chega a ser maior que o tempo gasto para executá-lo manualmente. Além da curva de aprendizado, que ocasiona uma perda de produtividade durante a fase inicial do processo de automação.

Para (QUENTIN, 1999) o esforço empregado nas atividades de planejamento de testes, projeto de casos de teste, criação de relatórios de teste e análise de relatórios de teste provoca uma diminuição média de 75% do esforço total destas atividades quando se utiliza maciçamente ferramentas de automação em um processo que antes era totalmente manual.

O tempo de execução de testes automáticos é na maioria das vezes mais rápido que o tempo necessário para realização dos mesmos testes manuais. Quando essa comparação é levada para um conjunto de casos de teste como, por exemplo, em uma suíte de casos de teste,

o tempo da execução automática torna-se ainda mais relevante e vantajoso quando comparado com o tempo da execução manual. Automação justifica assim, o esforço para a construção e manutenção dos *scripts* automáticos diante da economia gerada após sucessivas execuções de *scripts*.

2.9 JUNIT

O JUnit é um *framework* de testes unitários de código aberto que dá suporte à criação de testes automatizados em Java. Ele habilita a criação de código para a automação de testes com geração de resultados. A partir das classes disponibilizas pelo *framework*, é possível verificar se cada ação de uma classe funciona da maneira esperada, exibindo prováveis falhas e/ou erros. Para garantir que a corretude de uma classe ou método está de acordo com a especificação, basta que, após a execução de um teste, todas as condições verificadas resultem em sucesso, incrementando assim, a qualidade daquela unidade.

MASSOL (2003) apresenta os três principais objetivos do JUnit:

- Auxiliar a escrita de testes úteis;
- Ajudar na criação de testes que conservam o seu valor ao longo do tempo;
- Ajudar a reduzir o custo de escrever testes com a reutilização de código.

O JUnit é um *framework* simples e robusto pronto para auxiliar os programadores nas atividades de testes de software. De uma maneira mais atraente, criou-se uma forma alternativa de realizar testes por meio da criação de programas auxiliares responsáveis por realizar verificações específicas em um sistema. É através desse conceito que o JUnit permite deixar a fase de teste de unidades bem mais agradável para os desenvolvedores de um sistema. Para criar um teste, o programador precisa apenas criar uma classe que estenda *junit.framework.TestCase*, a principal classe do JUnit. A partir dessa classe, as validações podem ser feitas e após a instanciação de uma classe desse tipo, um ambiente de testes é definido e as ações declaradas no teste podem ser executadas.

A Figura 2.4 apresenta um diagrama que contém as principais classes do *framework*.

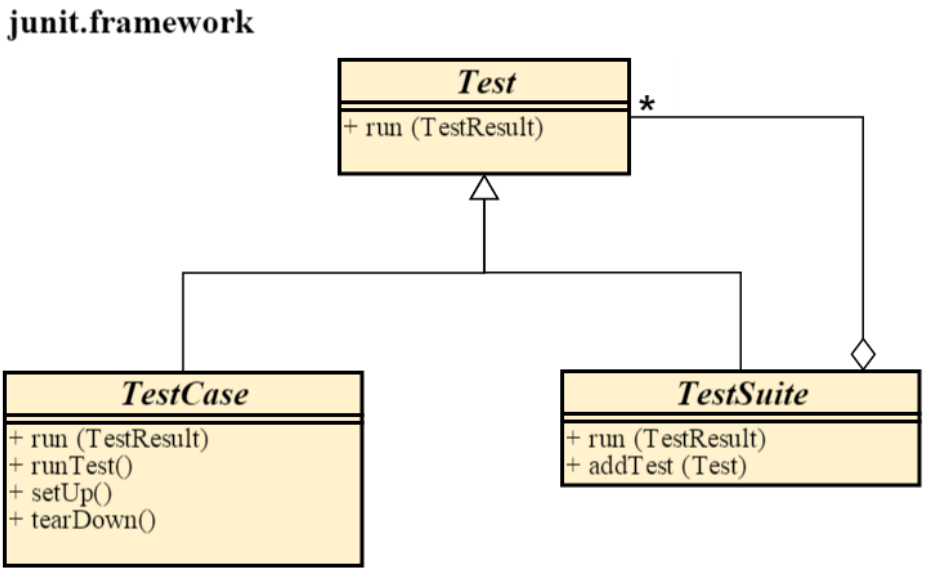

Figura 2.4: Diagrama das principais classes do JUnit.

A interface *Test* possui o método *run(TestResult)* que, uma vez implementado em uma classe, tem a responsabilidade de habilitar a execução de um teste ou suíte de testes e enviar os resultados para o objeto *TestResult.*

A classe abstrata *TestCase* define o ambiente de execução de um caso de teste. O método *run(TestResult)* executa os testes e envia os resultados para o objeto *TestResult*. O método *runTest()* é um método conveniente que executa o método *run(TestResult)*. O método *setUp()* é usado para configurar o ambiente inicial de um caso de teste, por exemplo, abrir uma conexão de rede. Este método é chamado antes da execução de um teste. O método *tearDown()* é usado para finalizar o ambiente de um caso de teste, por exemplo, fechar uma conexão de rede. Este método é chamado depois que um teste é executado.

A classe *TestSuite* é uma composição de casos de teste. Ela habilita a execução de uma coleção de testes. O método *addTest(Test)* permite a adição de um teste na coleção de testes. O método *run(TestResult)* é responsável pela execução e coleta de resultados de todos os testes da suíte.

Um teste de unidade é uma porção de código escrita por um desenvolvedor com o objetivo de executar uma funcionalidade específica no sistema a fim de verificar um determinado comportamento ou estado. Na literatura, essa afirmação ou validação é chamada de *assertion.* Um *assertion* é basicamente uma comparação entre uma entrada variável (sistema a ser testado) e uma saída pré-definida. A Tabela 2.1 apresenta uma lista com alguns dos *assertions* que são disponibilizados pelo JUnit (o comando utilizado para verificar um *assertion* é o *assert*).

Tabela 2.1: Lista de *asserts* do JUnit.

Assert	Ação
assertEquals(expected, actual)	Assegurar que duas entradas são iguais
assertNotEquals(expected, actual)	Assegurar que duas entradas não são iguais
assertTrue(condition)	Assegurar que uma condição é verdadeira
assertFalse(condition)	Assegurar que uma condição é falsa
assertNull(object)	Assegurar que um objeto é nulo/inválido
assertNotNull(object)	Assegurar que um objeto não é nulo/inválido

Para ilustrar um caso de teste, a Figura 2.5 mostra a implementação de um teste JUnit baseada no diagrama de classes (Figura 2.4) e na lista de *asserts* (Tabela 2.1). A classe *ExemploJUnit* estende a classe *TestCase*, sobrescreve os métodos *setUp* e *tearDown*, e cria o método de teste.

```
import junit.framework.TestCase;

public class ExemploJUnit extends TestCase {

    @Override
    protected void setUp() throws Exception {

        super.setUp();
        // Método usado para executar ações antes do início da execução caso de teste
        // TODO Implementar ações...
    }

    public void test() throws Exception {

        // Execução do teste

        // Efetuar uma operação de multiplicação (5 x 4) e garantir que a operação será igual a 20
        int expected = 20;
        int operation = 5 * 4;
        assertEquals(expected, operation);
    }

    @Override
    protected void tearDown() throws Exception {

        super.tearDown();
        // Método usado para executar ações após a execução caso de teste
        // TODO Implementar ações...
    }
}
```

Figura 2.5: Exemplo de um caso de teste JUnit.

Segundo CLARK (2005), existem algumas razões pelas quais o JUnit é recomendado:

- JUnit é gratuito;
- JUnit é elegante e simples. Quando testar um programa se torna algo complexo e demorado, na maioria dos casos não existe motivação para o programador fazê-lo;

- Pode-se criar uma hierarquia de testes que permitirá testar todo o sistema ou apenas parte dele;
- JUnit verifica os resultados dos testes e fornece uma resposta imediata na forma de um relatório com os testes que passaram e falharam na execução.

O JUnit possui integração com diversas ferramentas de desenvolvimento Java, como o Eclipse (ECLIPSE, 2016), IntelliJ (INTELLIJ, 2016), Netbeans (NETBEANS, 2016), entre outras. Adicionalmente, ele foi adaptado para outras linguagens de programação, tais como C++ (CPPUNIT, 2016), C# (NUNIT, 2016) e Python (UNITTEST, 2016).

2.10 FRAMEWORKS DE TESTES ANDROID

O *framework* de testes Android, parte integrante do ecossistema de desenvolvimento de aplicações móveis Android, provê uma arquitetura e ferramentas poderosas que ajudam times de desenvolvedores e testadores a testar todos os aspectos de uma aplicação Android em todos níveis: desde testes simples de unidade até testes complexos de performance.

Segundo ANDROID (2016), o *framework* de testes Android tem as seguintes características fundamentais:

- Os testes são baseados no JUnit. É possível criar um teste JUnit simples para testar uma classe que não tem dependência com a *Application Programming Interface* (API) do Android, ou também usar extensões do JUnit para Android a fim de testar componentes do sistema Android;
- As extensões do JUnit para Android proveem classes de casos de teste específicos para cada componente. Essas classes fornecem métodos auxiliares para a criação de objetos e métodos que ajudam a controlar o ciclo de vida de um componente de simulação;
- Os testes estão contidos dentro de pacotes que são similares aos pacotes principais da aplicação, o que elimina a necessidade de aprender um novo conjunto de técnicas ou ferramentas durante a concepção e construção de testes;
- O *Software Development Kit* (SDK) do Android também fornece uma API para habilitar o teste de dispositivos através de programas escritos na linguagem Python, além de possuir uma ferramenta de linha de comando voltada para testes de estresse de interfaces de usuário (UI) ativadas pelo envio de comandos pseudoaleatórios para um dispositivo.

A Figura 2.6 resume o *framework* de testes Android.

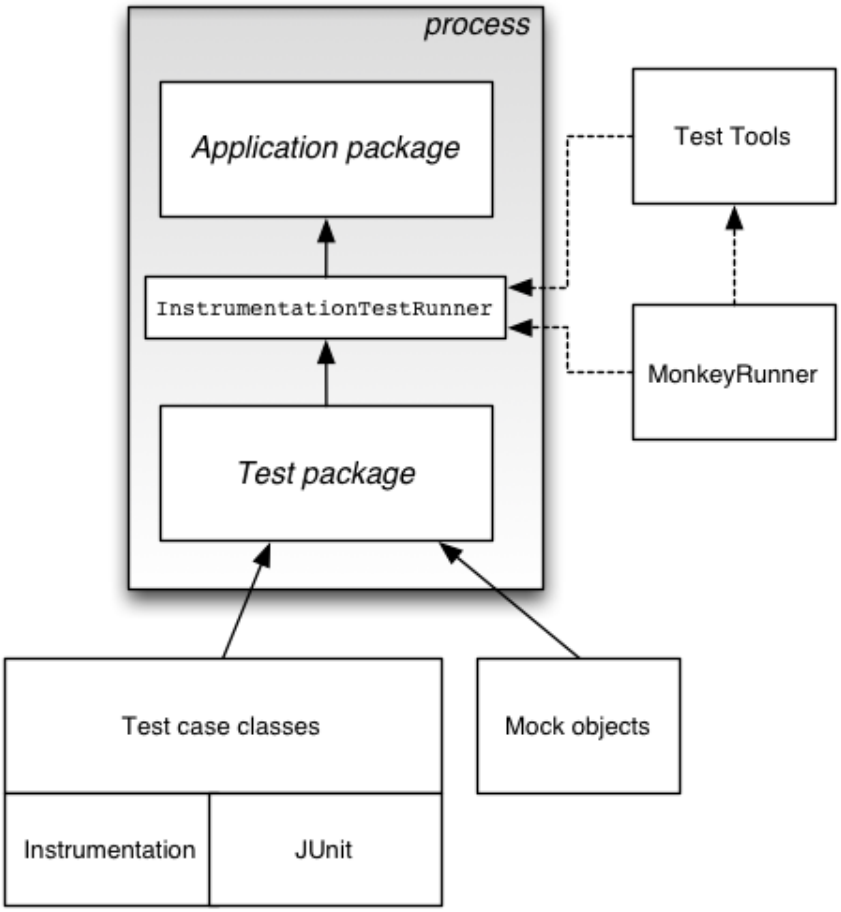

Figura 2.6: Diagrama do *framework* de testes Android.

Na Figura 2.6, o bloco *process* apresenta, de maneira resumida, a estrutura de uma aplicação Android com três pequenos blocos. No bloco *Application package* encontra-se todo o código de uma aplicação. O bloco *InstrumentationTestRunner* é uma classe Android com a capacidade de executar qualquer classe que estende a classe *TestCase* do JUnit. Essa classe possui também um conjunto de serviços de controle no sistema Android. Por meio destes serviços, é possível controlar um componente Android independentemente do seu ciclo de vida normal, bem como controlar como o Android carrega as aplicações. Os blocos *Test Tools* e *MonkeyRunner* ligados à *InstrumentationTestRunner* representam os conjuntos de APIs ou ambientes de execução de testes. No bloco *Test package* encontra-se todo o código de testes de uma aplicação. Os blocos *Test case classes*, *Instrumentation*, *JUnit* e *Mock objects* representam o conjunto de classes usadas no desenvolvimento de testes Android.

2.10.1 UI Automator

Dentro desse conjunto de ferramentas fornecidas pelo Android, existe o UI Automator, um *framework* para criação de testes de interface gráfica baseado no JUnit 3. O UI Automator fornece um conjunto de APIs para construir testes de UI (*User Interface*) que simulam as ações de um usuário. Suas APIs permitem que um desenvolvedor simule e execute operações como clicar em um botão para abrir o menu de aplicações ou abrir um aplicativo em um dispositivo móvel Android. O UI Automator é um *framework* adequado para escrever testes caixa-preta, onde o código de teste não depende de detalhes internos de implementação de uma aplicação. As principais características do UI Automator incluem:

- Um visualizador (Figura 2.7) para capturar e analisar os componentes gráficos que estão sendo apresentados na tela de um dispositivo. Com essa ferramenta, um desenvolvedor pode inspecionar a hierarquia de layout da tela, visualizar as propriedades dos componentes que aparecem na tela do dispositivo e, a partir dessas informações, escrever testes caixa-preta. A Figura 2.7 mostra o UI Automator Viewer detalhando a estrutura da interface gráfica da tela de configuração de conexões de rede;

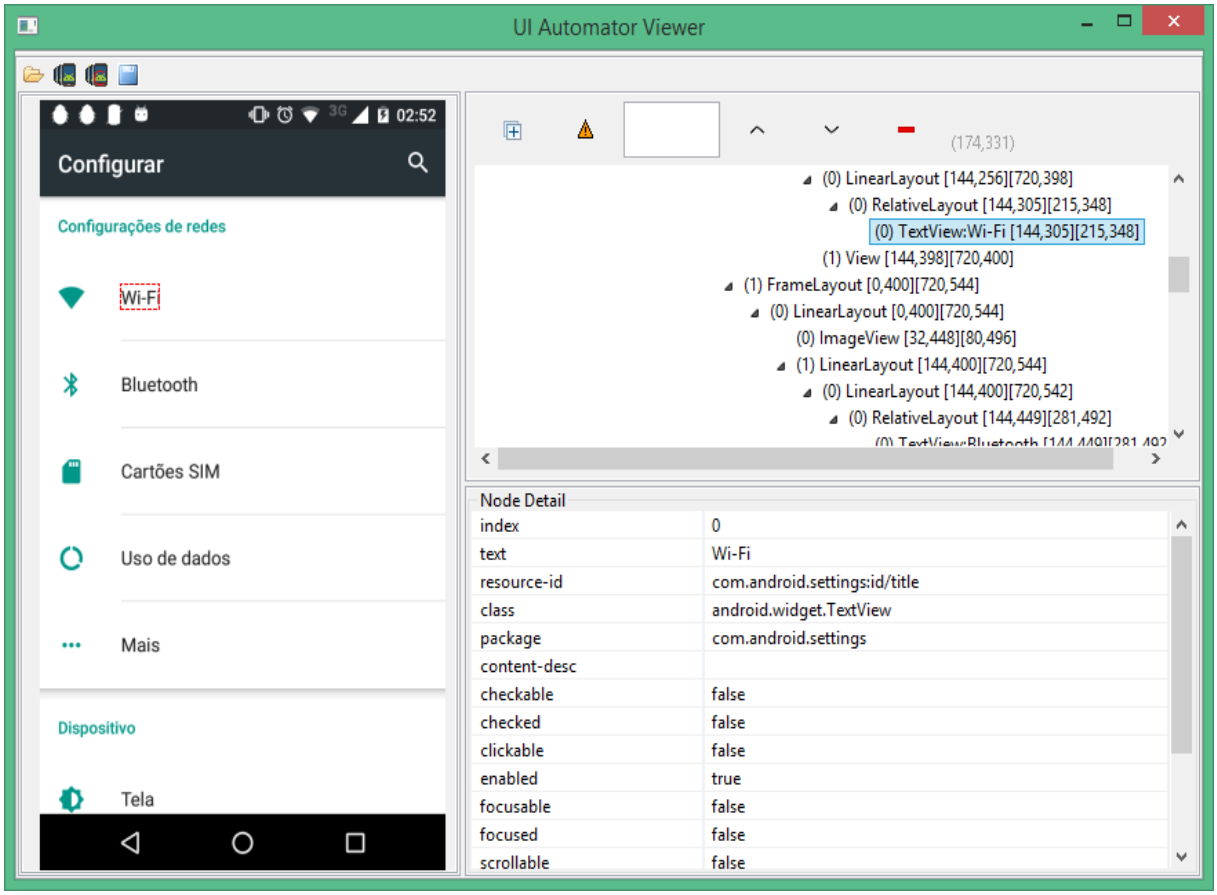

Figura 2.7: Captura de tela da ferramenta UI Automator Viewer.

- O desenvolvedor pode usar as classes da API para capturar e manipular os componentes gráficos em qualquer aplicativo sem precisar do código fonte. A Tabela 2.2 resume as principais classes utilitárias do *framework.*

Tabela 2.2: Lista das principais classes do UI Automator.

Classe	Descrição
UiAutomatorTestCase	Define um ambiente para execução de testes
UiDevice	Fornece acesso a informações sobre o dispositivo, bem como um conjunto de ações
UiObject	Representa um elemento de interface gráfica que está visível na tela do dispositivo
UiCollection	Enumera os elementos de interface gráfica de um contêiner com a finalidade de contagem ou acesso a informação dos subelementos
UiScrollable	Fornece suporte para a pesquisa de itens em um elemento
UiSelector	Representa uma consulta para um ou mais elementos de interface gráfica em um dispositivo
Configurator	Permite a configuração de parâmetros fundamentais para a execução de testes

- Uma API específica para efetuar operações e recuperar as informações de estado de um dispositivo. Através da classe *UiDevice*, o desenvolvedor pode chamar vários métodos pré-definidos para efetuar ações no dispositivo. Algumas ações podem ser vistas na Tabela 2.3;

Tabela 2.3: Lista de métodos da classe UiDevice.

Método	Ação
openNotification()	Abrir a barra de notificações do dispositivo
openQuickSettings()	Abrir a barra de configurações rápidas do dispositivo
pressBack()	Simula um clique no botão BACK do dispositivo
pressHome()	Simula um clique no botão HOME do dispositivo
wakeUp()	Simula um clique no botão LIGA/DESLIGA para acender a tela do dispositivo
setOrientationLeft()	Simula a mudança de orientação da tela do dispositivo para a esquerda

Para ilustrar um caso de teste usando o *framework*, a Figura 2.8 mostra a implementação de um teste UI Automator baseado nas principais classes do *framework* (Tabela 2.2). Todo teste UI Automator deve estender a classe *UiAutomatorTestCase*, uma classe do *framework* que é baseada na classe *junit.framework.TestCase* (Figura 2.4). A classe *ExemploUIAutomator* estende a classe *UiAutomatorTestCase*, sobrescreve os métodos *setUp* e *tearDown*, e cria o seu método de teste.

```
import com.android.uiautomator.core.UiDevice;
import com.android.uiautomator.core.UiObject;
import com.android.uiautomator.core.UiScrollable;
import com.android.uiautomator.core.UiSelector;
import com.android.uiautomator.testrunner.UiAutomatorTestCase;

public class ExemploUiAutomator extends UiAutomatorTestCase {

    private UiDevice mDevice;

    @Override
    protected void setUp() throws Exception {

        super.setUp();

        mDevice = UiDevice.getInstance();

        // Acende a tela do dispositivo e pressiona a tecla HOME
        mDevice.wakeUp();
        mDevice.pressHome();
    }

    public void test() throws Exception {

        // Recuperar o botão central para abrir a bandeja de aplicativos
        UiObject appTrayItem = new UiObject(new UiSelector().description("Apps"));
        appTrayItem.click();

        UiScrollable scrollableList = new UiScrollable(new UiSelector().scrollable(true));

        // Ao abrir a bandeja, faz um scroll pesquisando a aplicação Chrome
        scrollableList.scrollTextIntoView("Chrome");

        UiObject chromeItem = new UiObject(new UiSelector().text("Chrome"));

        // Efetua um clique para abrir o browser Chrome
        if (chromeItem.exists()) {
            chromeItem.clickAndWaitForNewWindow();
            assertEquals("com.android.chrome", mDevice.getCurrentPackageName());
        }
    }

    @Override
    protected void tearDown() throws Exception {

        super.tearDown();

        // Pressiona a tecla HOME e apaga a tela do dispositivo
        mDevice.pressHome();
        mDevice.sleep();
    }
}
```

Figura 2.8: Exemplo de um caso de teste UI Automator.

No exemplo anterior, a classe *ExemploUIAutomator* possui um atributo do objeto *UiDevice* (Tabela 2.2). Esse atributo dá acesso às informações do dispositivo. O método *setUp()*, que é executado antes do caso de teste, prepara o ambiente para a execução do teste. O comando *UiDevice.getInstance()* disponibiliza uma instância do dispositivo sob teste. O comando *wakeUp()* é acionado para acender a tela do dispositivo e o *pressHome()* para mover o dispositivo para a tela principal. O método *test()* refere-se ao caso de teste propriamente dito. O objetivo desse caso de teste é abrir uma aplicação chamada ***Chrome***. Para isso, o teste cria um *UiObject* que contém a descrição **"Apps"**, referente ao botão central, que dá acesso à bandeja de aplicações. O teste clica no botão para acessar a bandeja de aplicações. Após isso, o teste precisa navegar pela tela pesquisando o texto ***"Chrome"***. O método *scrollTextIntoView(String)* tem a finalidade de efetuar um *scroll* na tela pesquisando determinado texto. Após efetuar a rolagem para o texto ***"Chrome"***, o usuário cria um novo *UiObject* que contém o texto ***"Chrome"*** com o objetivo de clicar nesse objeto e iniciar a aplicação. A cláusula *assertEquals(expected, actual)* verifica se a aplicação desejada foi executada comparando o pacote atual que o dispositivo está apresentando com um pacote previamente definido. O método *tearDown()*, que é executado depois do caso de teste, finaliza o ambiente de execução do teste. O comando *pressHome()* move o dispositivo para a tela inicial e o *sleep()* apaga a tela do dispositivo, finalizando a execução do caso de teste.

Todo projeto de testes automáticos implementado através do *framework* UI Automator é distribuído em um arquivo compactado no formato JAR (*Java ARchive*). Esse arquivo contém todas as classes e metadados que constituem o projeto.

A Figura 2.9 mostra as duas ações necessárias para executar um teste UI Automator em um dispositivo.

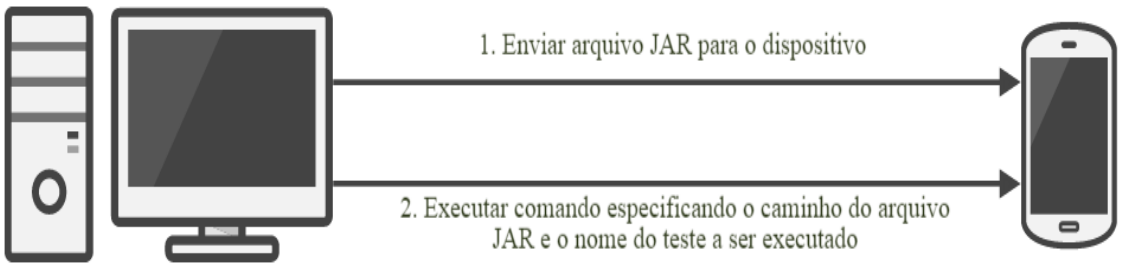

Figura 2.9: Ações necessárias para executar um teste UI Automator.

As duas ações mencionados na Figura 2.9 são executadas através da ferramenta ADB (*Android Device Bridge*), uma ferramenta versátil de linha de comando que habilita a comunicação de um computador com um emulador ou dispositivo Android conectado à máquina. A Figura 2.10 apresenta os comandos que representam as ações 1 e 2 vistas na Figura 2.9.

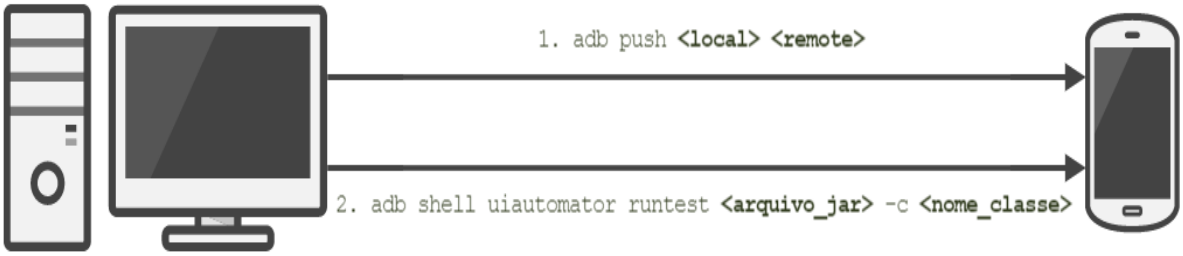

Figura 2.10: Comandos ADB para executar um teste UI Automator.

A Figura 2.11 ilustra a tela de *prompt* de comando do computador de um usuário que utilizou os comandos vistos na Figura 2.10 para executar o teste *ExemploUiAutomator* da Figura 2.8.

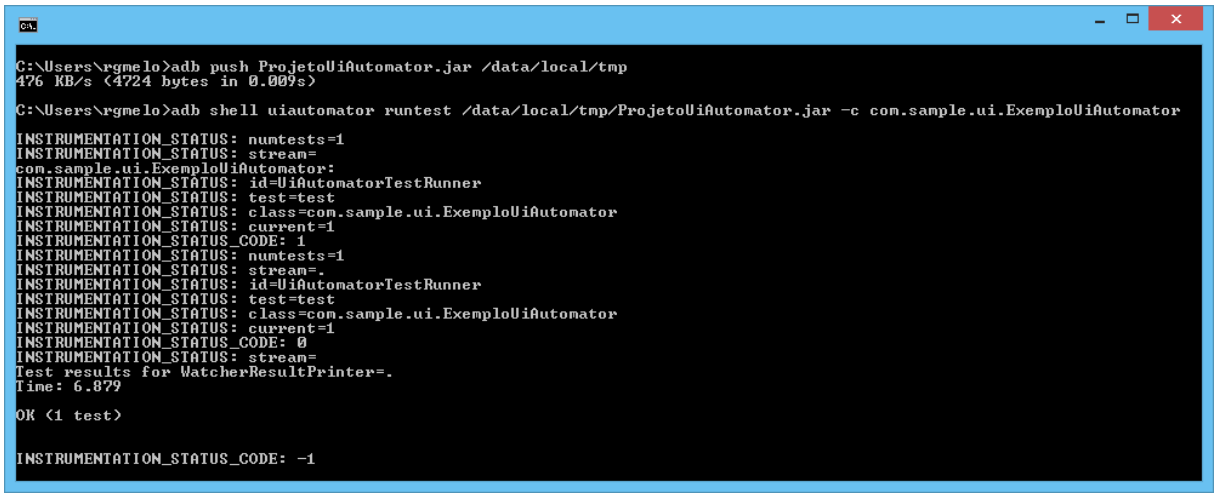

Figura 2.11: Captura de tela de *prompt* de comando invocando as ações para a execução de um teste UI Automator (cenário 1 – sucesso).

Na Figura 2.11, pode-se observar o resultado da execução do teste *ExemploUiAutomator*. As informações textuais apresentadas após o comando *adb shell uiautomator runtest* são a única saída provida pelo *framework*. Portanto, o resultado do teste deve ser extraído a partir desse log de informações. Com base na mensagem *"OK (1 test)"*, infere-se que o resultado do teste resultou em sucesso.

Observe as informações apresentadas após uma execução do teste *ExemploUiAutomator* em um cenário que resultará em **falha** no teste (Figura 2.12).

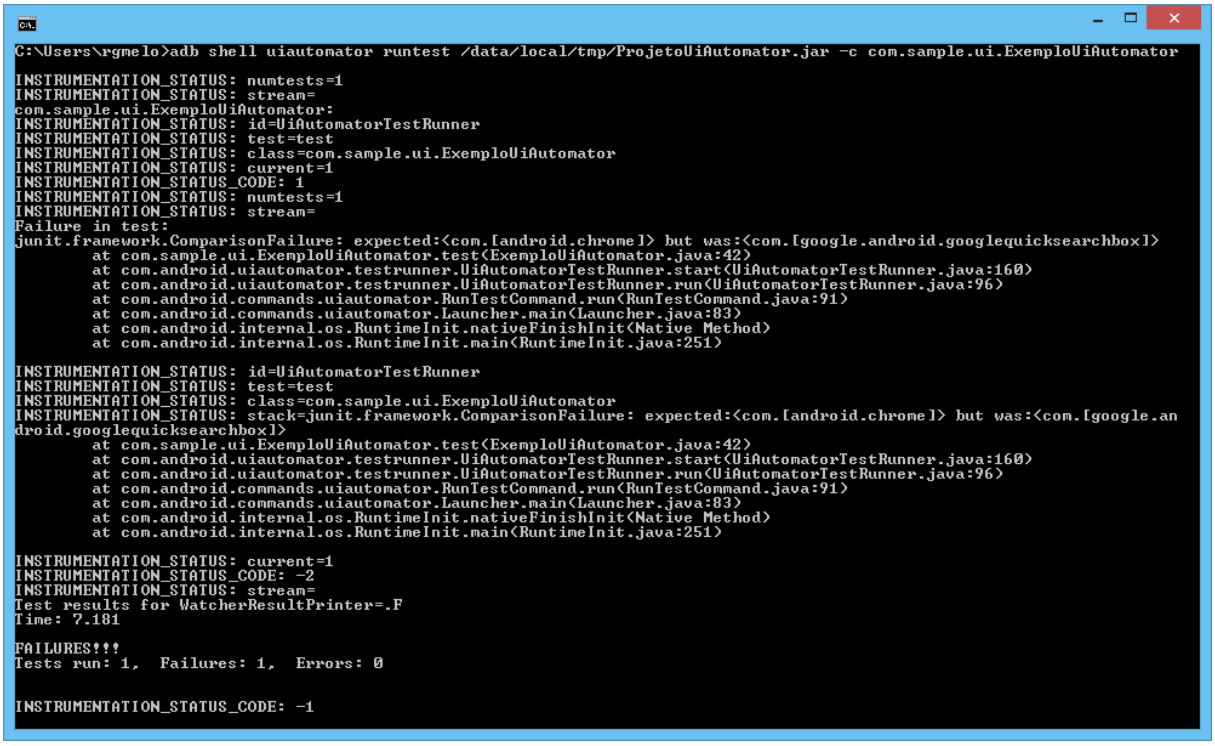

Figura 2.12: Captura de tela de *prompt* de comando invocando as ações para a execução de um teste UI Automator (cenário 2 – falha).

Analisando a mensagem *"FAILURES!!! Tests run: 1, Failures: 1, Errors: 0"*, pode-se inferir que o resultado do teste resultou em falha.

Agora, observe as informações apresentadas após uma execução do teste *ExemploUiAutomator* em um cenário que resultará em **erro** no teste (Figura 2.13).

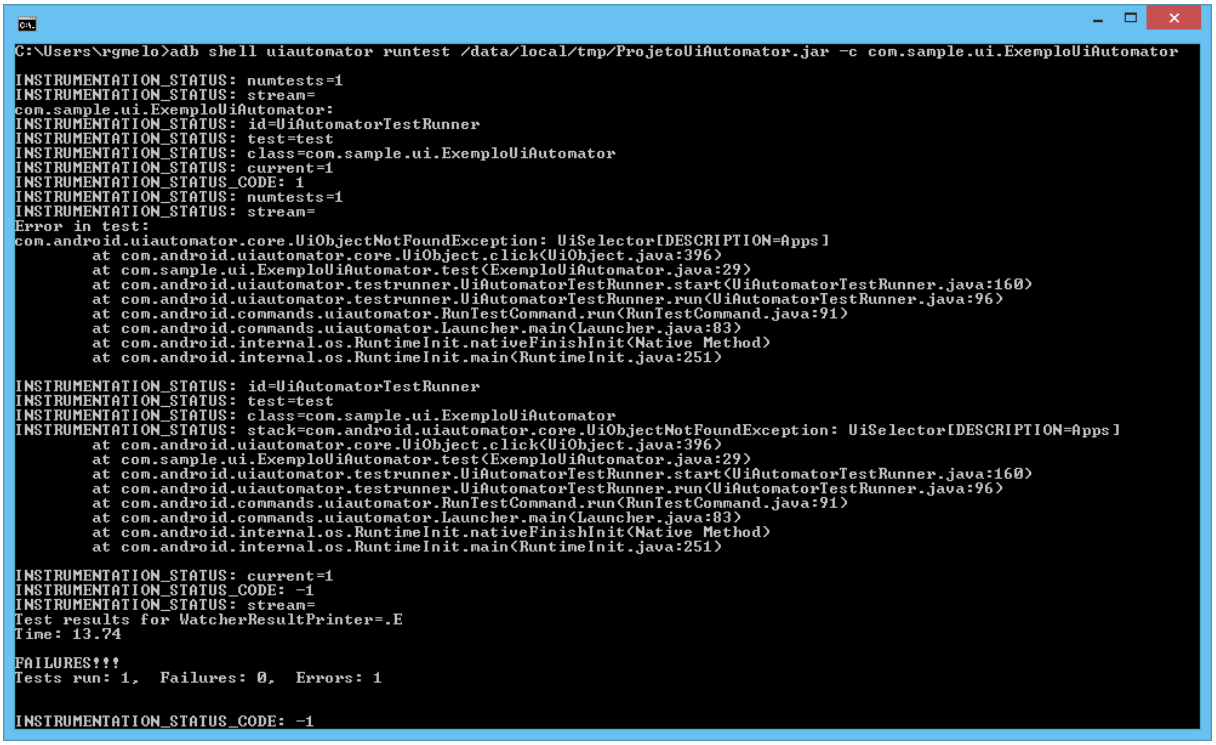

Figura 2.13: Captura de tela de *prompt* de comando invocando as ações para a execução de um teste UI Automator (cenário 3 – erro).

Baseado na mensagem *"FAILURES!!! Tests run: 1, Failures: 0, Errors: 1"*, infere-se que o resultado do teste resultou em erro.

Através das Figuras 2.11, 2.12 e 2.13, pode-se perceber a dificuldade na obtenção dos resultados de um caso de teste, visto que a única fonte de avaliação do teste é um log de informações e, na maioria das vezes, esse log contém vários textos distribuídos que confundem bastante os testadores no momento de analisar e interpretar cada uma das informações. O *framework* não provê, por exemplo, um arquivo estruturado contendo o resultado final do teste e as informações relativas a cada validação interna, como também não provê uma interface usual para análise de resultados.

Vale salientar que os cenários hipotéticos abordados nas Figuras 2.11, 2.12 e 2.13 estão efetuando somente uma validação no caso de teste. Considerando que um caso de teste pode ter várias validações internas, o grau de dificuldade na análise de um caso de teste é proporcional ao número de validações, uma vez que o log de informações cresce de acordo com o número de asserções realizadas. E, se considerarmos um caso cotidiano no contexto em que estamos inseridos, onde diariamente são executadas diversas suítes de testes contendo dezenas de casos de teste automáticos, a análise do log toma um tempo considerável. De fato, esta realidade comprova o quanto a análise dos resultados por meio de logs de informações é

custosa. Até então, consideramos somente a dificuldade imposta pelo *framework* no que diz respeito à análise de resultados por meio de informações textuais.

Outro fator crítico que merece destaque é o custo para a criação/edição e execução de suítes de testes, uma vez que o UI Automator não dá suporte à execução de uma classe do tipo *junit.framework.TestSuite* (Figura 2.4). Por esta razão, os testadores costumam criar um arquivo no formato *.bat* ou .sh, dependendo do sistema operacional, para representar a suíte de testes. Hipoteticamente, se uma suíte tem *N* casos de teste, o arquivo em lotes terá *N* comandos para invocar a execução de um teste UI Automator (*adb shell uiautomator runtest*). A Figura 2.14 apresenta uma suíte com 10 casos de teste, representada em um arquivo no formato *.bat*.

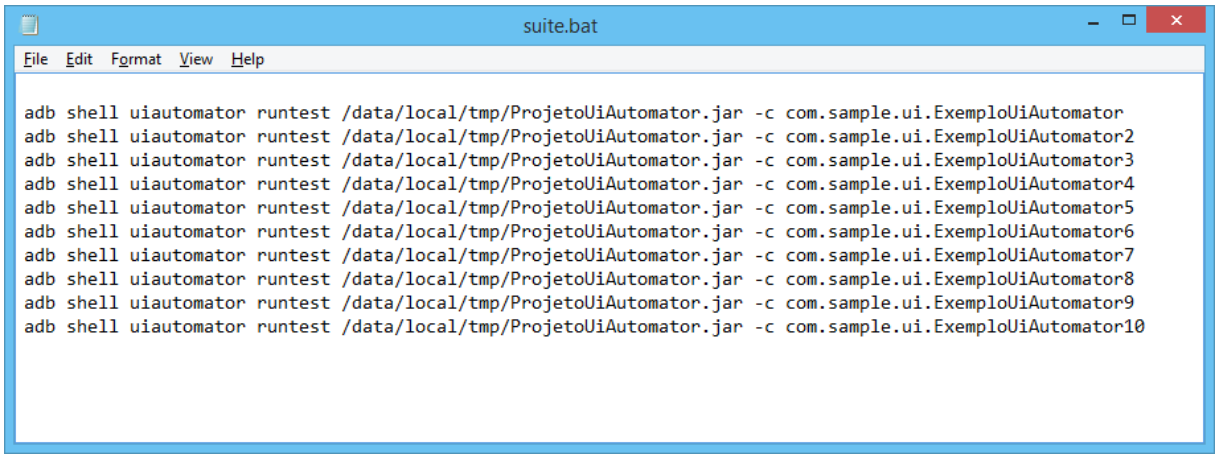

Figura 2.14: Suíte com 10 casos de teste, representada em um arquivo .bat.

Em alguns casos, os testadores usam uma variação do comando *adb shell uiautomator runtest* e combinam os *N* casos de teste em uma única linha de comando, como pode ser visto na Figura 2.15.

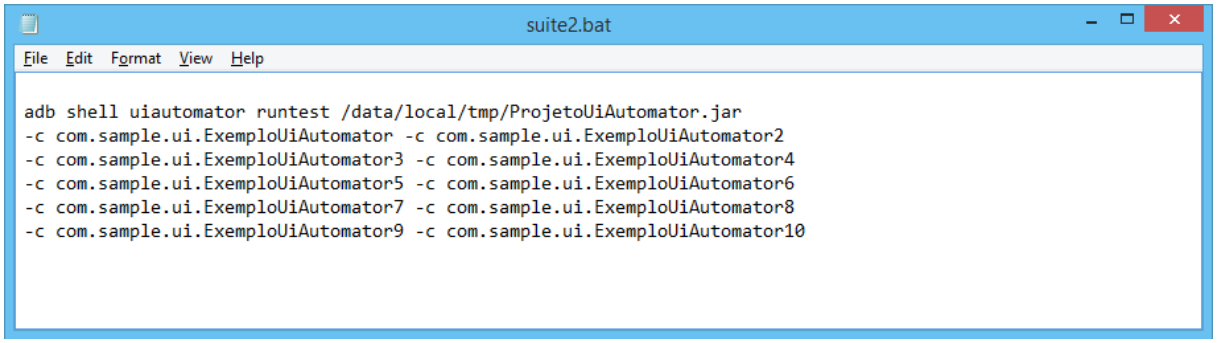

Figura 2.15: Suíte combinada com 10 casos de teste, representada em um arquivo .bat.

Após a criação do arquivo em lotes referente à suíte de testes, os testadores iniciam a execução da suíte invocando o arquivo criado. Um fato bastante comum na execução de suítes de testes é a necessidade de reexecutar os testes que apresentaram falhas e/ou erros durante a

execução. O *framework* não tem a capacidade de inferir o resultado final da execução e reexecutar os testes que apresentaram falhas/erros. Assim, os testadores precisam analisar o log de informações da suíte, identificar os testes com falhas/erros e, por fim, editar a suíte de testes e executar o arquivo novamente.

Ainda sobre execução de testes UI Automator, não é possível definir um número de tentativas de reexecução (*retry*) de um teste com falha/erro. Adicionalmente, o *framework* não disponibiliza uma opção onde o desenvolvedor possa estabelecer um tempo máximo de execução (*timeout*) para um caso de teste. A ausência dessa capacidade pode implicar em situações críticas dentro de um projeto. Por exemplo, um testador tem uma demanda de execução de uma suíte com 30 testes automáticos. Digamos que o primeiro teste a ser executado precisa receber uma mensagem de texto e verificar se a mesma foi entregue corretamente. Por questões de problemas na rede de telefonia, a mensagem não chega e o teste fica esperando pela mensagem de texto infinitamente, bloqueando a execução da suíte de testes e impactando severamente o andamento das atividades do projeto.

Com relação à atividade de implementação de testes automáticos de UI, vimos que, além de disponibilizar uma ferramenta auxiliar para captura e análise de componentes gráficos que são apresentados na tela de um dispositivo, o UI Automator provê também um conjunto de APIs fundamentais para um desenvolvimento rápido e fácil de testes automáticos. No entanto, o UI Automator apresenta diversos problemas no que diz respeito à execução de testes e visualização de resultados.

2.11 CONSIDERAÇÕES FINAIS

Neste capítulo, introduzimos os conceitos fundamentais para o entendimento de testes de software, enfatizando as definições de automação de testes e as ferramentas utilizadas nas atividades de automação. Tendo em vista que o foco do trabalho está centrado na automação de testes funcionais em dispositivos móveis Android, descrevemos a arquitetura do *framework* JUnit e do *framework* de testes Android. Dentre as ferramentas providas pelo Android, destacamos o UI Automator, um *framework* para criação de testes de interface gráfica baseado no JUnit, composto por um conjunto de APIs e ferramentas auxiliares adequadas para escrever testes funcionais, mas que apresenta dificuldades no gerenciamento e execução de suítes de testes.

Os *frameworks* descritos neste capítulo são a base de instanciação e implementação do *framework* FREVO. Além de herdar as características dos *frameworks* descritos acima, o

principal objetivo do *framework* é solucionar os problemas existentes proporcionando um ambiente de desenvolvimento mais amplo, habilitando a existência de um ambiente de execução gerenciável. Vale ressaltar que os problemas descritos nesse capítulo não são particulares deste *framework*, mas sim de qualquer *framework* de teste, como Robotium (ROBOTIUM, 2015), Appium (SHAO, 2015), entre outros.

3 FREVO: FRAMEWORK E FERRAMENTA

Todos *frameworks* de automação de teste têm foco na execução de um único teste, mas, como ilustrado no Capítulo 2, pelos exemplos de UI Automator e JUnit, não há nenhuma facilidade de execução de suítes de testes. Neste capítulo, descreveremos como o *framework* FREVO e a ferramenta FREVO promovem o conceito de automação de teste para uma dimensão maior de suítes de teste.

Para materializar nossos conceitos escolhemos estender o *framework* UI Automator por ser o *framework* padrão de teste caixa-preta de Android e por ser adotado no contexto deste trabalho dentro do projeto CIn-Motorola. Para a implementação da ferramenta, adotamos Java pela compatibilidade com o Android. Entretanto, as extensões propostas aqui podem ser implementadas em qualquer framework de automação de teste similares ao UI Automator e Java.

3.1 OBJETIVOS

Os principais objetivos deste trabalho estão centrados no preenchimento das lacunas existentes nos *frameworks* de automação de teste por meio da construção de dois componentes integrados:

- *Framework* de desenvolvimento de *scripts* baseado no UI Automator;
- Ferramenta de gerenciamento e execução de testes.

A construção desses dois componentes integrados separa de maneira coesa as atividades de desenvolvimento de *scripts* das atividades de gerenciamento e execução de suítes de testes. No texto que segue, descreveremos as extensões instanciadas para UI Automator e Java, porém as extensões são genéricas para qualquer *framework*.

Para a automação ser promovida ao nível de suíte de testes, propusemos e implementamos as seguintes facilidades, tanto no nível de *framework* quanto no nível de ferramenta GUI:

- *Timeout* para que um caso de teste finalize sua execução;

- Armazenamento e visualização do resultado de um teste de forma mais rica (ao invés de apenas passou e falhou, incluímos a descrição do teste e exibimos as *screenshots* de cada validação);

- Novos *asserts*, como por exemplo, uma asserção para verificar se determinado componente gráfico existe na tela e uma asserção para efetuar a comparação de expressões regulares (muito útil na validação de formatos de data, hora, entre outros).

- Criação de um banco de dados de execuções de suítes (pré e pós-execução);

- Criação de um gerenciador e controlador de execução de uma suíte, incluindo cronometragem do tempo de execução, definição do idioma e armazenamento de resultados;

- GUI para criar, editar, executar e visualizar os resultados de uma suíte sem necessidade de manipulação de arquivos em lote (*batch*), que são bastante limitados para definir *timeouts*, reexecuções e armazenamento de resultados.

- GUI integrada com ferramentas externas de repositório de testes, como Test Central e Dalek.

3.2 DESCRIÇÃO DO FRAMEWORK

O *framework* FREVO caracteriza-se como uma evolução do *framework* UI Automator. A proposta do *framework* está centrada na solução das limitações existentes no UI Automator e na adição de novas características, incrementando a capacidade do *framework*. Visando a integração com a ferramenta FREVO, o *framework* cria um novo formato de saída de dados.

3.2.1 Arquitetura

A Figura 3.1 descreve os pacotes do *framework*.

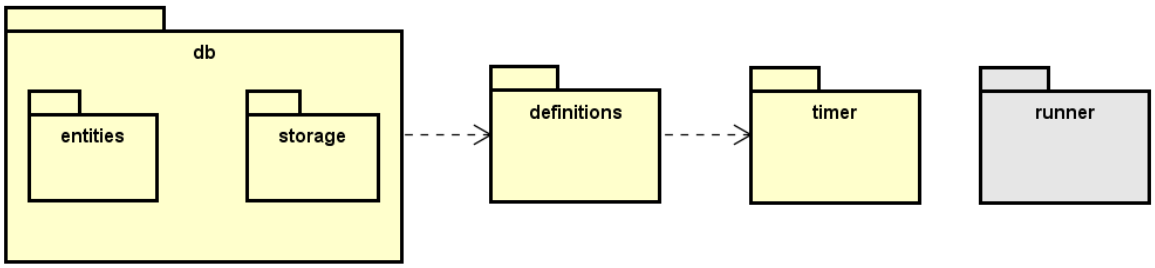

Figura 3.1: Diagrama de pacotes do *framework* FREVO.

O *framework* FREVO está agrupado em quatro pacotes de classes (*db*, *definitions*, *timer* e *runner*). O pacote ***db*** contém as entidades básicas definidas pelo *framework* e as classes responsáveis pelo gerenciamento (criação, inserção, edição) do banco de dados das execuções. O pacote ***definitions*** contém a principal classe do *framework*, que é responsável pelo controle da execução do teste e possui as novas características adicionadas ao sistema. O pacote ***timer*** abriga a classe responsável pela cronometragem da execução de um teste. O pacote ***runner*** contém um conjunto de classes utilitárias responsáveis pela geração de um banco de dados que conterá todos os casos de teste de um projeto UI Automator.

A Figura 3.2 apresenta os pacotes do *framework* com suas respectivas classes.

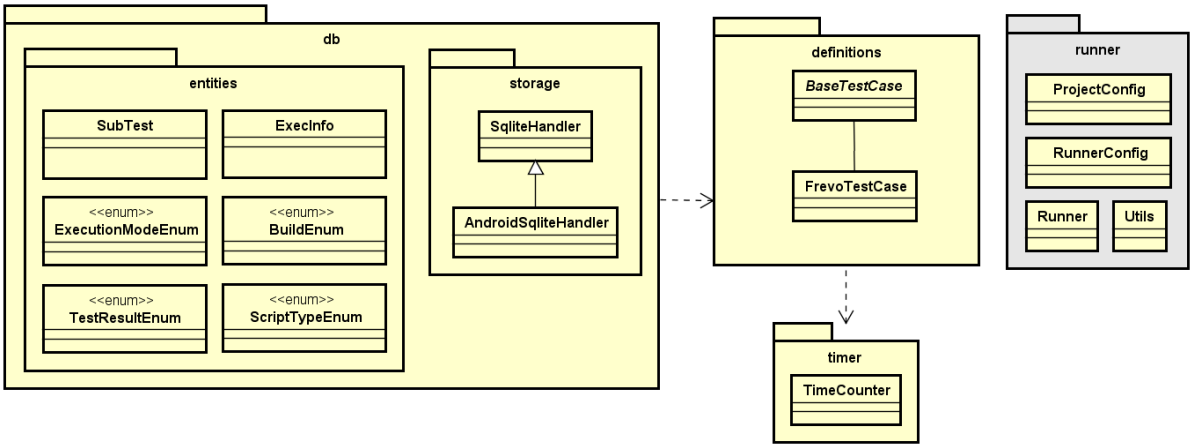

Figura 3.2: Diagrama de classes do *framework* FREVO.

3.2.2 Classe BaseTestCase

A principal classe do *framework* Frevo é a *BaseTestCase*, que estende a classe *UiAutomatorTestCase* (Tabela 2.2) do UI Automator de modo a garantir o ambiente padrão de execução de um teste. As principais funções da classe *BaseTestCase* são:

- Definição do fluxo de execução;
- Padronização na codificação de *scripts*;
- Controle e monitoramento da execução, considerando o *timeout* de um caso de teste;
- Armazenamento de informações/características do caso de teste*;*
- Gravação dos resultados em um banco de dados;
- Captura de *screenshots* em cada *assert* efetuado*;*
- Disponibilização de novos *asserts*.

O fluxo de execução de um caso de teste executado utilizando o *framework* FREVO é baseado no diagrama de estados da Figura 3.3.

Figura 3.3: Diagrama de estados da execução de um teste no *framework* FREVO.

Os métodos *init(), setUpTestCase(), main()* e *tearDownTestCase()* apresentados acima representam não só o fluxo de execução do teste, mas também a padronização da codificação dos *scripts* do *framework* FREVO. Obrigatoriamente, toda classe que estende a classe *BaseTestCase* automaticamente implementará os quatro métodos acima, garantindo uniformidade no código dos *scripts.*

A Figura 3.4 detalha os principais atributos e métodos da classe *BaseTestCase* e suas ligações com outras entidades.

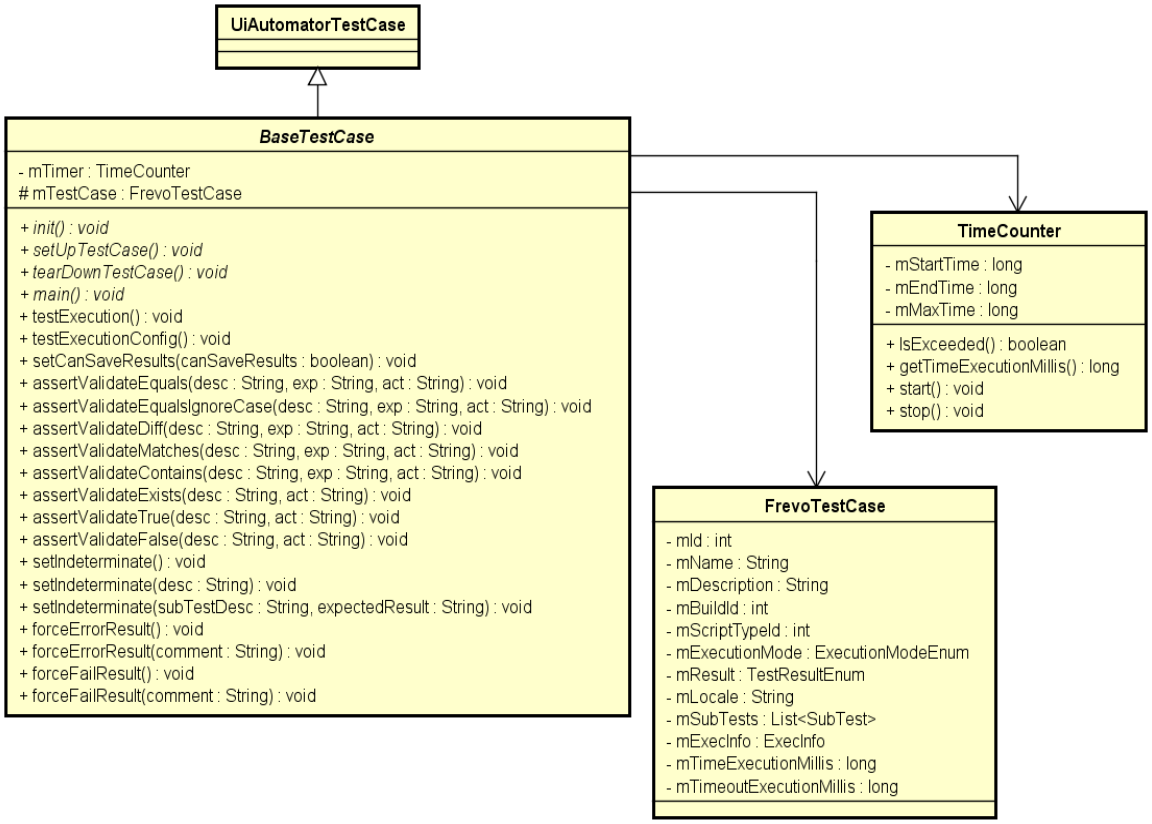

Figura 3.4: Diagrama de classes com os principais atributos e métodos da classe *BaseTestCase*.

A partir do diagrama de classes acima, é possível visualizar a apresentação de alguns *asserts* e outras funções úteis para os desenvolvedores de *scripts*. Além disso, é possível visualizar a ligação da classe *BaseTestCase* com outras duas classes, *TimeCounter* e *FrevoTestCase*. A classe *TimeCounter* é responsável por medir o tempo de execução e controlar o tempo máximo permitido para cada teste. A classe *FrevoTestCase* é responsável por estruturar todos os dados da execução (informações específicas do teste, idioma de execução, validações internas do teste, tempo de execução e resultado do teste).

A classe *FrevoTestCase* contém atributos usados para armazenar informações externas referentes aos scripts. O armazenamento dessas informações é o ponto de partida para a comunicação com ferramentas externas voltadas para o gerenciamento de plano de testes, uma vez que os resultados dos testes executados podem ser enviados a estas ferramentas automaticamente.

Na concepção da geração de um banco de dados estruturado com todas informações da execução de um teste, decidiu-se considerar cada validação (*assert*) como um subteste. Isto é, toda vez que o desenvolvedor insere um *assert* em seu código, o *framework* automaticamente trata este *assert* como um subteste, capturando todas as informações da asserção (descrição, valor atual e esperado, *screenshot* da validação e o resultado). Portanto, para garantir a

gravação das informações no banco de dados, o desenvolvedor deve usar somente os *asserts* disponibilizados na classe *BaseTestCase*. Dessa maneira, a determinação do resultado de um teste é baseada nos resultados existentes dentro de cada subteste, artificio este que já é considerado na classe.

A Tabela 3.1 faz uma relação entre os nomes dos métodos padrões dos *frameworks* UI Automator e FREVO que têm funcionalidades equivalentes.

Tabela 3.1: Comparação entre os métodos padrões dos *frameworks* UI Automator e FREVO.

UI Automator	FREVO	Ação
setUp()	*setUpTestCase()*	Configuração do ambiente de execução do teste
test()	*main()*	Execução do teste propriamente dito
tearDown()	*tearDownTestCase()*	Finalização do ambiente de execução do teste
	init()	Preenchimento de informações específicas do teste

Para ilustrar a implementação de um teste usando o *framework* FREVO, a Figura 3.5 traz a implementação do teste UI Automator visto na Figura 2.8. Agora, em vez de estender a classe *UiAutomatorTestCase*, o caso de teste deve estender a classe *BaseTestCase*. A classe *ExemploFrevoUIAutomator* estende a classe *BaseTestCase*, implementa os métodos obrigatórios e adiciona as ações do teste.

```
public class ExemploFrevoUiAutomator extends BaseTestCase {

    private final String LAUNCH_APP = "Chrome";
    private final String PACKAGE_APP = "com.android.chrome";
    private UiDevice mDevice;

    @Override
    public void init() throws Exception {
        // Define descrição do teste
        mTestCase.setDescription("Abrir a aplicação Chrome no dispositivo sob teste");

        // Define a versão do sistema para a qual o script é indicado
        mTestCase.setBuildId(BuildEnum.LOLLIPOP.getId());

        // Define um timeout de 30s (30000ms) para o teste.
        mTestCase.setTimeoutExecutionMillis(30000);
    }

    @Override
    public void setUpTestCase() throws Exception {
        mDevice = UiDevice.getInstance();

        // Acende a tela do dispositivo e pressiona a tecla HOME
        mDevice.wakeUp();
        mDevice.pressHome();
    }

    @Override
    public void main() throws Exception {
        // Recuperar o botão central para abrir a bandeja de aplicativos
        UiObject appTrayItem = new UiObject(new UiSelector().description("Apps"));
        appTrayItem.click();

        UiScrollable scrollableList = new UiScrollable(new UiSelector().scrollable(true));

        // Ao abrir a bandeja, faz um scroll pesquisando a aplicação definda em
        // LAUNCH_APP
        scrollableList.scrollTextIntoView(LAUNCH_APP);

        UiObject appItem = new UiObject(new UiSelector().text(LAUNCH_APP));

        // Efetua um clique para abrir a aplicação definida em LAUNCH_APP
        if (appItem.exists()) {
            appItem.clickAndWaitForNewWindow();
            sleep(3000);

            mSubTestDescription = "1. Verifique que a aplicação Chrome foi executada com sucesso";
            mExpectedResult = PACKAGE_APP;
            mActualResult = mDevice.getCurrentPackageName();

            this.assertValidateEquals(mSubTestDescription, mExpectedResult, mActualResult);
        }
    }

    @Override
    public void tearDownTestCase() throws Exception {
        // Pressiona a tecla HOME
        mDevice.pressHome();
    }
}
```

Figura 3.5: Exemplo de um caso de teste UI Automator usando o *framework* FREVO.

Igualmente à classe *ExemploUiAutomator* da Figura 2.8, a classe *ExemploFrevoUiAutomator* também tem como objetivo principal abrir uma aplicação chamada ***Chrome***. Como mostrado na Tabela 3.1, os métodos *setUp()*, *test()* e *tearDown()* do

UI Automator, são representados no *framework* FREVO por *setUpTestCase()*, *main()* e *tearDownTestCase()*, respectivamente.

Ao comparar as Figuras 2.8 e 3.5, é possível perceber que os códigos relativos às ações do teste permaneceram inalterados nas duas classes. A única diferença entre elas é a estruturação imposta pela classe *BaseTestCase*, com a adição do método *init()*, indicado para preenchimento das informações do teste, e a mudança na utilização do *assert* da classe *BaseTestCase*. Portanto, essa pequena diferença na estruturação dos *scripts* e na utilização dos *asserts* internos deverão estar presentes em todos os *scripts* que são desenvolvidos usando o *framework* FREVO.

A Figura 3.6 ilustra a tela de *prompt* de comando do computador de um usuário que executou o teste *ExemploFrevoUiAutomator* da Figura 3.5.

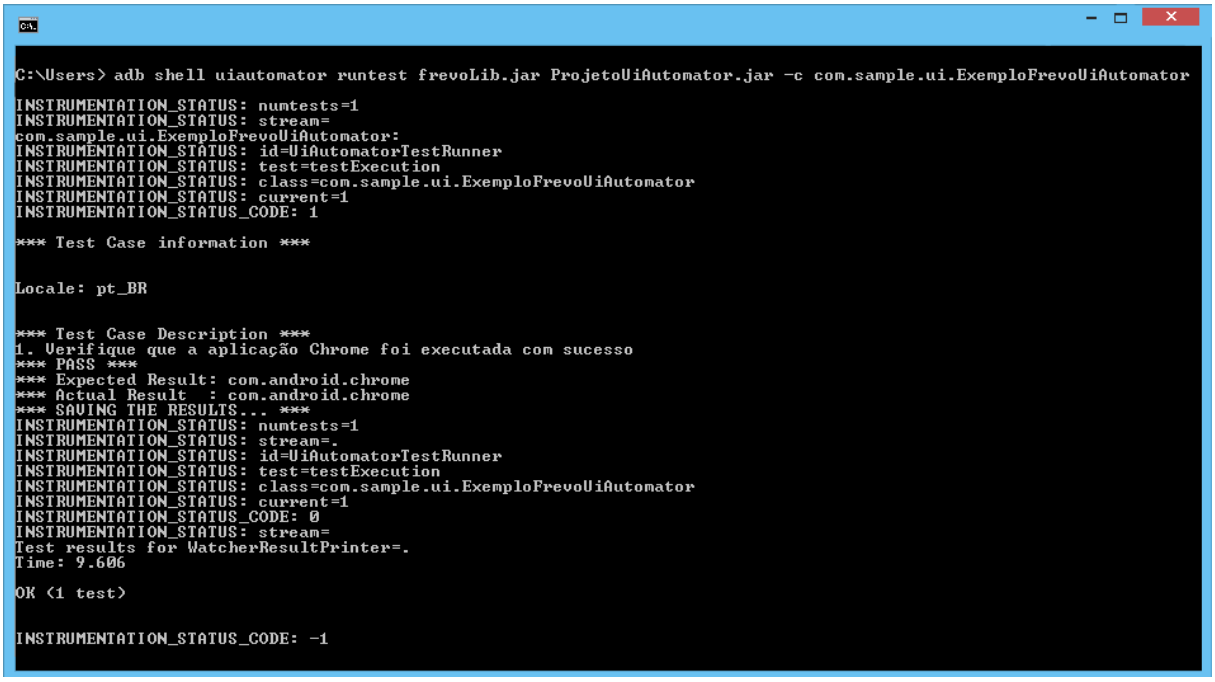

Figura 3.6: Captura de tela de *prompt* de comando invocando as ações para a execução de um teste usando o *framework* FREVO.

Na Figura 3.6, pode-se observar o resultado da execução do teste *ExemploFrevoUiAutomator*. As informações textuais apresentadas após o comando *adb shell uiautomator runtest* não é mais a única fonte de saída provida pelo *framework* FREVO. Além de adicionar mais informações ao log (informações do teste e das validações), a execução passa a ter um banco de dados e todas as *screenshots* referentes às validações. Dessa maneira, os testadores passam a ter uma fonte de dados com informações estruturadas (banco de dados) e intuitivas (*screenshots)* para auxiliá-lo no processo de revisão dos resultados dos testes. Munido dessas informações, a melhoria no processo de testes passa a ser bastante

significativa, visto que a quantidade de informações úteis será essencial para um processo de testes eficaz. As Figuras 3.7 e 3.8 abaixo apresentam os artefatos adicionais gerados na execução da classe *ExemploFrevoUiAutomator*.

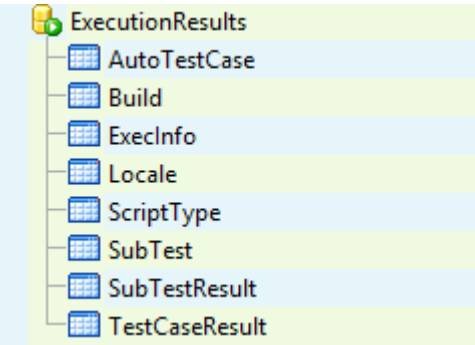

Figura 3.7: Banco de dados resultante da execução da classe *ExemploFrevoUiAutomator*.

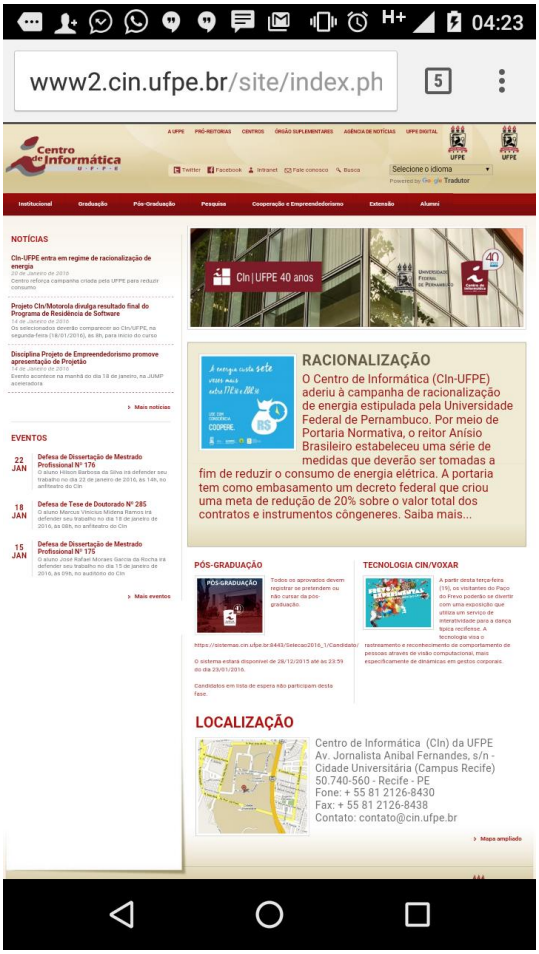

Figura 3.8: Captura de tela proveniente de uma validação efetuada na classe *ExemploFrevoUiAutomator*.

3.2.3 Modelo do banco de dados do *framework*

O projeto de banco de dados foi uma das etapas mais difíceis na construção do *framework*, pois na fase concepção e modelagem dos dados foi necessário abstrair todas as entidades envolvidas no contexto do *framework* e refletí-las em um sistema de gerenciamento de banco de dados (SGBD) relacional, onde os dados são organizados na forma de tabelas.

Segundo (HEUSER, 2008), no projeto de construção de um novo banco de dados, normalmente são considerados dois níveis de abstração de modelo de dados:

- Modelo conceitual;
- Modelo lógico.

O projeto de banco de dados do *framework* FREVO foi realizado baseado nas duas fases descritas acima. Na modelagem conceitual, construímos um modelo na forma de um diagrama entidade-relacionamento (Figura 3.9). Este modelo capturou as necessidades do *framework* em termos de armazenamento de dados de forma independente de implementação, ou seja, não foi preciso registrar como os dados seriam armazenados em um SGBD.

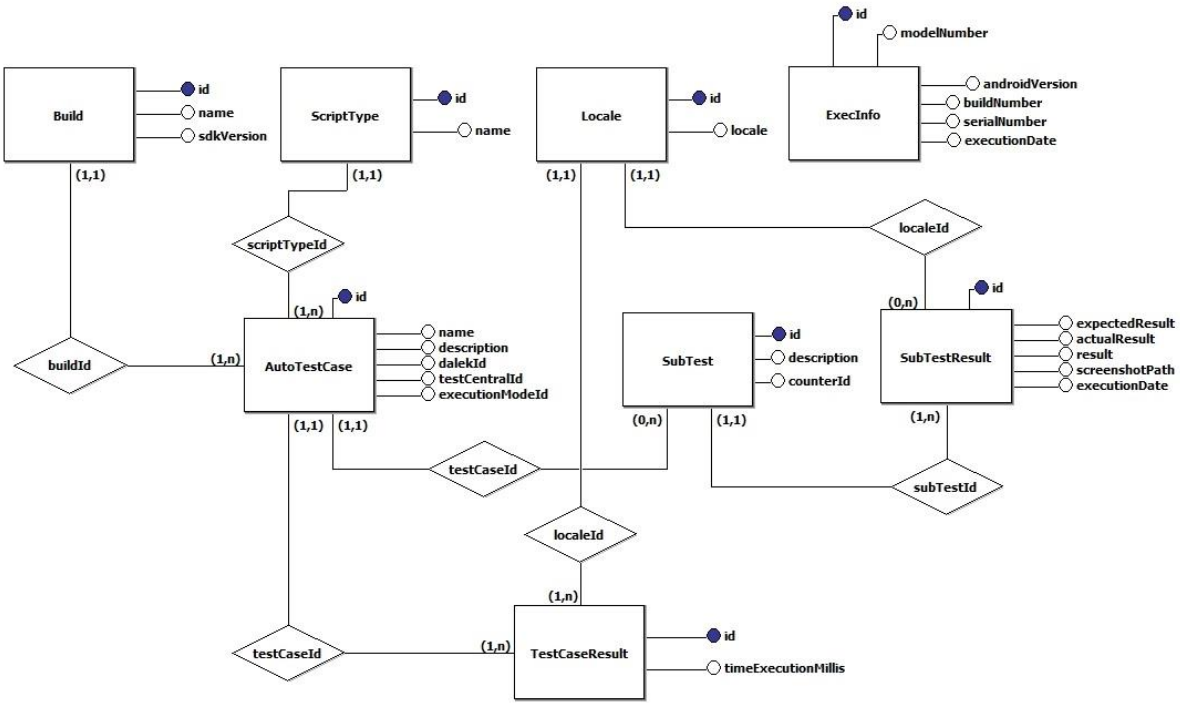

Figura 3.9: Diagrama entidade-relacionamento do banco de dados gerado pelo *framework* FREVO.

Na fase de projeto lógico, o principal objetivo é transformar o modelo conceitual obtido na primeira fase em um modelo lógico. O modelo lógico define como o banco de dados será implementado em um SGBD especifico. A Figura 3.10 ilustra o modelo lógico gerado a partir do diagrama entidade-relacionamento (Figura 3.9).

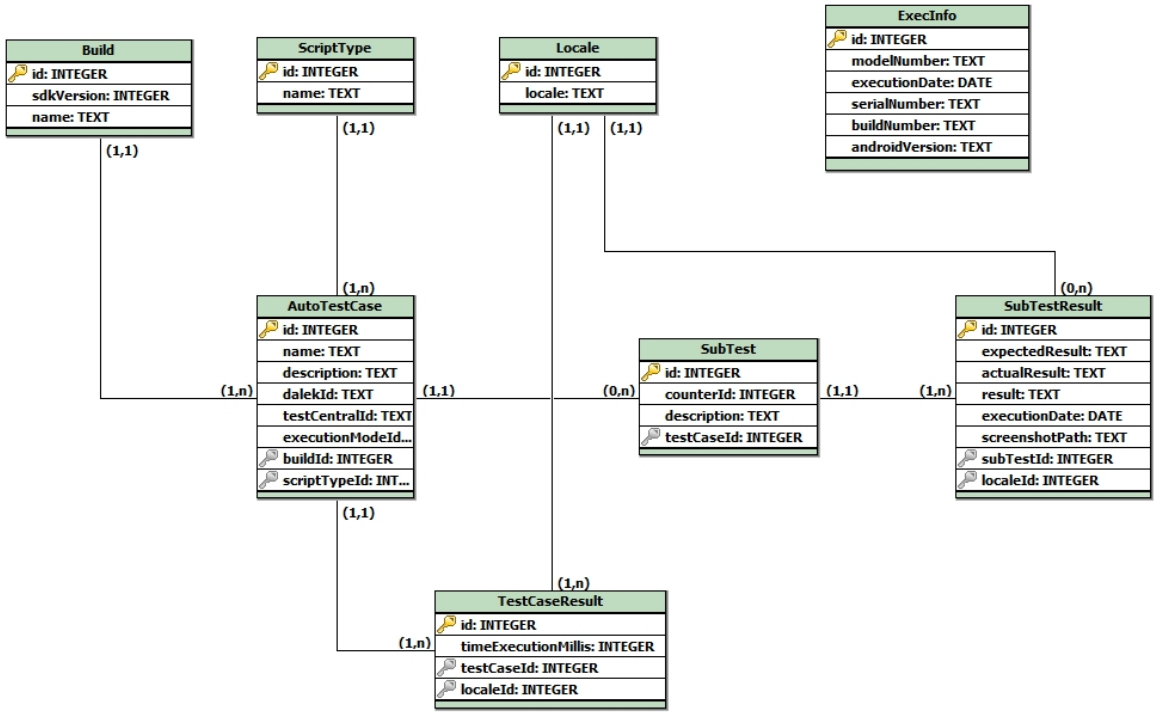

Figura 3.10: Modelo lógico do banco de dados gerado pelo *framework* FREVO.

O modelo lógico da Figura 3.10 é composto por sete tabelas:

- Tabela ***Build*** – Armazena informações pré-definidas de *builds* de diversas versões do sistema operacional Android;
- Tabela ***ScriptType*** **–** Armazena informações pré-definidas para classificar o tipo de teste;
- Tabela ***Locale*** **–** Armazena informações dos idiomas em que o teste foi executado;
- Tabela ***ExecInfo*** **–** Armazena informações gerais do dispositivo onde o teste foi executado;
- Tabela ***AutoTestCase*** **–** Armazena informações básicas do(s) teste(s) executados em um dispositivo. Essa tabela armazena informações referentes à *build* do dispositivo (tabela *Build*) e ao tipo do teste (tabela *ScriptType*);
- Tabela ***SubTest*** **–** Armazena as informações básicas das validações de um subteste (validação) realizadas pelo teste executado. Todos os registros dessa tabela são relacionados a um teste (tabela *AutoTestCase*);
- Tabela ***SubTestResult*** **–** Armazena as informações referentes ao resultado de um subteste (validação). Todos os registros dessa tabela estão

relacionados diretamente com um subteste (tabela *SubTest*) e com um idioma (tabela *Locale*);

- Tabela ***TestCaseResult*** **–** Armazena as informações referentes ao resultado final de um teste. Todos os registros dessa tabela possuem relação direta com um teste (tabela *AutoTestCase*) e com um idioma (tabela *Locale*).

Como pode ser visto na Figura 3.7, o banco de dados resultante da execução de um teste escrito usando o *framework* FREVO contém as sete tabelas apresentadas nos diagramas das figuras 3.9 e 3.10.

3.2.4 Geração de banco de dados de todos *scripts* de um projeto UI Automator

Sabemos que a distribuição de projetos UI Automator se dá por meio de um arquivo compactado no formato JAR, que contém todas as classes de teste e outros recursos construídos dentro de um projeto.

Para uma execução convencional de testes UI Automator, os testadores precisam conhecer o nome completo das classes de teste e combiná-las com os comandos ADB vistos na Figura 2.10 para finalmente poder iniciar a execução dos testes. Foi nesse contexto que nos deparamos com uma das maiores dificuldades na construção do *framework*: definir uma abordagem automática responsável por mapear todas as classes de teste existentes em um projeto UI Automator. Após esse mapeamento, o *framework* deveria ser capaz também de distribuir essas informações em um formato de dados estruturados.

No diagrama de classes do *framework* FREVO (Figura 3.2), podemos visualizar o pacote ***runner***. Esse pacote contém um conjunto de classes utilitárias responsáveis por mapear os testes dentro um projeto UI Automator e pela consequente geração de um banco de dados contendo todos os testes do projeto.

A Figura 3.11 apresenta uma classe *DatabaseConfig* que esboça a configuração simples de um projeto UI Automator.

```java
public class DatabaseConfig {

    public static void main(String[] args) {

        RunnerConfig runnerConfig = new RunnerConfig();

        // Caminho do arquivo JAR do framework
        String frevoLibJarPath = "C:/Users/rgmelo/frevoLib.jar";

        // Caminho do arquivo JAR do projeto
        String projectJarPath = "C:/Users/rgmelo/ProjetoUiAutomator.jar";

        // Nome do pacote das classes de teste
        String packageScripts = "com.sample.scripts.frevo";

        runnerConfig.setLocalFrameworkJarFilePath(frevoLibJarPath);

        runnerConfig.addProjectConfig(projectJarPath, packageScripts);

        runnerConfig.setTargetFolderDatabase("release");

        Runner.createInitialDatabase(runnerConfig);
    }
}
```

Figura 3.11: Exemplo de classe construída para gerar o banco de dados de um projeto UI Automator.

A classe *DatabaseConfig* é um programa Java que irá gerar o banco de dados de um projeto chamado *ProjetoUiAutomator*. Para gerar um banco de dados de um projeto, o desenvolvedor só precisa de duas classes. A classe *RunnerConfig* é usada para informar as características do projeto. É necessário informar o caminho do arquivo JAR do *framework* FREVO, o caminho do arquivo JAR referente ao projeto e o pacote onde as classes de teste estão presentes. Por fim, a classe *Runner* é usada para iniciar a geração do banco de dados baseado nas configurações existentes em *RunnerConfig*. A Figura 3.12 ilustra os dados da tabela *AutoTestCase* do banco de dados gerado, contendo registros de todas as classes de teste que estão dentro do pacote *com.sample.scripts.frevo* informado na configuração.

id	name	description
1	com.sample.scripts.frevo.ExemploFrevoUiAutomator	Abrir a aplicação Chrome no dispositivo sob teste
2	com.sample.scripts.frevo.ExemploFrevoUiAutomator10	Abrir a aplicação Câmera no dispositivo sob teste
3	com.sample.scripts.frevo.ExemploFrevoUiAutomator2	Abrir a aplicação Settings no dispositivo sob teste
4	com.sample.scripts.frevo.ExemploFrevoUiAutomator3	Abrir a aplicação Agenda no dispositivo sob teste
5	com.sample.scripts.frevo.ExemploFrevoUiAutomator4	Abrir a aplicação Calculadora no dispositivo sob teste
6	com.sample.scripts.frevo.ExemploFrevoUiAutomator5	Abrir a aplicação Relógio no dispositivo sob teste
7	com.sample.scripts.frevo.ExemploFrevoUiAutomator6	Abrir a aplicação Galeria no dispositivo sob teste
8	com.sample.scripts.frevo.ExemploFrevoUiAutomator7	Abrir a aplicação Ajuda no dispositivo sob teste
9	com.sample.scripts.frevo.ExemploFrevoUiAutomator8	Abrir a aplicação Mensagens no dispositivo sob teste
10	com.sample.scripts.frevo.ExemploFrevoUiAutomator9	Abrir a aplicação Telefone no dispositivo sob teste

Figura 3.12: Dados da tabela *AutoTestCase* do banco de dados dos *scripts*.

O banco de dados contendo todas as informações das classes de teste de um projeto implementado usando o *framework* FREVO, juntamente com o arquivo JAR do projeto são os dois artefatos necessários para integrar um projeto FREVO UI Automator com a ferramenta de gerenciamento e execução de testes.

3.3 DESCRIÇÃO DA FERRAMENTA

Sabemos que o *framework* FREVO conseguiu sanar algumas das limitações existentes no *framework* UI Automator, como também incorporou novas características de codificação, melhorando a qualidade dos *scripts* codificados. Não podemos esquecer dos artefatos que são gerados após a execução de um teste implementado usando o *framework* FREVO. Agora, além de melhorar o log de informações dos *scripts*, o *framework* também fornece um banco de dados estruturado com as informações básicas do teste e todas *screenshots* resultante das validações dos testes (por exemplo, quando um teste falha, o sistema automaticamente salva a *screenshot* da tela do celular).

A ferramenta FREVO se enquadra no contexto de aproveitar todos os artefatos providos pela execução de um teste do *framework* FREVO, podendo proporcionar aos testadores uma interface gráfica para visualização de resultados de casos de teste, tornando o processo de análise de resultados mais intuitivo, objetivo e usual.

Além disso, a ferramenta foca no provimento de funcionalidades para solucionar o problema na criação/edição e execução de suítes de testes UI Automator, onde os testadores precisavam criar um arquivo em lotes contendo linhas de comando ADB para cada caso de teste (figuras 2.14 e 2.15). Vale lembrar das dificuldades enfrentadas pelos testadores na necessidade de reexecutar uma suíte de testes, tendo o retrabalho de analisar os testes da execução anterior e editar o arquivo em lotes que representa a suíte.

O foco no aproveitamento dos artefatos gerados pelo *framework* FREVO e na solução dos problemas existentes no gerenciamento e execução de testes UI Automator faz com que a ferramenta FREVO apresente um processo de gerenciamento e execução de testes maduro e consistente.

As seções a seguir descreverão a arquitetura de software da ferramenta, a integração da ferramenta com o *framework* e um exemplo real da ferramenta sendo executada.

3.3.1 Arquitetura

A Figura 3.13 descreve os pacotes da ferramenta.

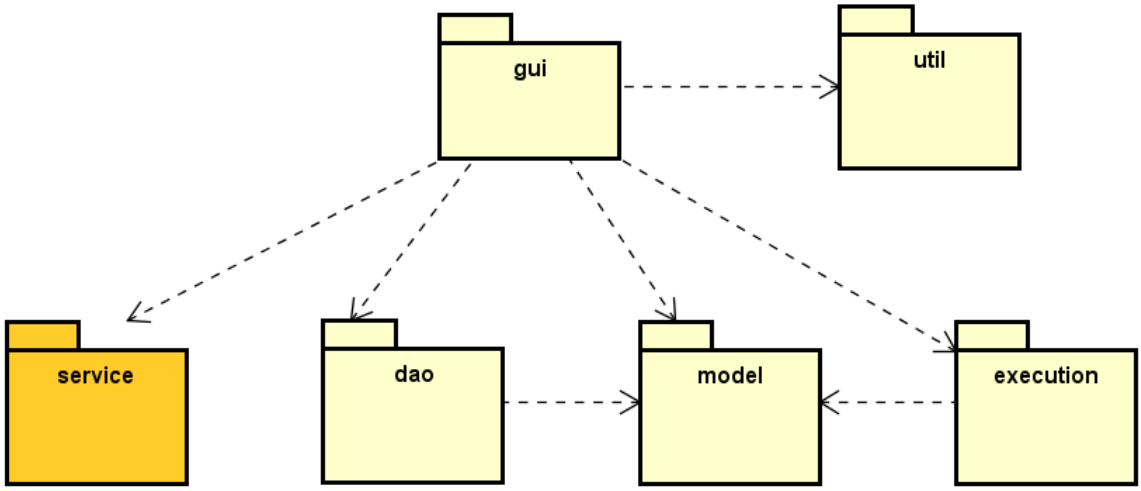

Figura 3.13: Diagrama de pacotes da ferramenta FREVO.

A ferramenta está agrupada em seis pacotes de classes (*gui*, *util*, *service*, *dao*, *model* e *execution*). O pacote ***gui*** contém as classes que compõem a interface gráfica da ferramenta (janelas principais, *dialogs*, etc.). O pacote ***util*** contém um conjunto de classes utilitárias usadas pela ferramenta. O pacote ***service*** abriga as classes responsáveis por realizar a comunicação com ferramentas externas de gerenciamento de testes. Essas ferramentas externas mantém a relação dos testes originais. O pacote ***dao*** contém as classes responsáveis pela leitura dos bancos de dados provenientes do *framework* FREVO e outras classes responsáveis por criar arquivos estruturados no formato XML (*eXtensible Markup Language*) (BRAY, 1998). O pacote ***model*** contém as classes que constituem as entidades fundamentais da ferramenta. O pacote ***execution*** contém as classes responsáveis pelo controle e gerenciamento da execução de testes UI Automator.

O conteúdo interno dos pacotes descritos acima é bastante extenso; isto é, a quantidade de classes e outros recursos é muito grande. As subseções abaixo trazem um diagrama resumido para cada um dos seis pacotes, detalhando somente informações de suas classes e/ou características mais importantes.

3.3.2 Pacote *gui*

A Figura 3.14 apresenta um diagrama resumido das classes do pacote *gui*.

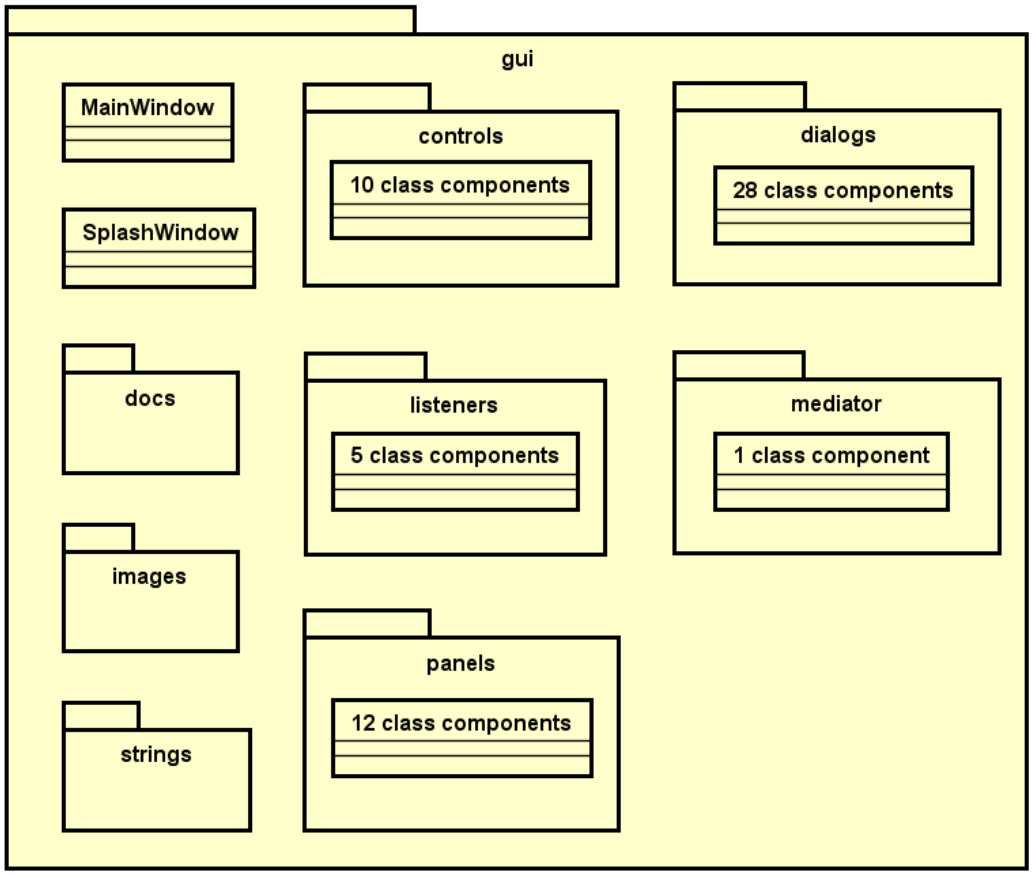

Figura 3.14: Diagrama de classes resumido do pacote *gui*.

O pacote *gui* contém a classe *MainWindow*. Essa classe representa a tela principal da ferramenta e funciona como um contêiner de componentes gráficos. Cada macrofuncionalidade da ferramenta está contida no subpacote *panels*. O subpacote *dialogs* contém classes que não estão embutidas no contêiner principal da ferramenta. Os subpacotes *controls*, *listeners* e *mediator* contém classes utilitárias que usadas para diversos fins. Os subpacotes *docs*, *images* e *strings* referem-se a documentos, imagens e textos, respectivamente, e são usadas por todo o sistema.

3.3.3 Pacote *util*

A Figura 3.15 apresenta um diagrama resumido das classes do pacote *util*.

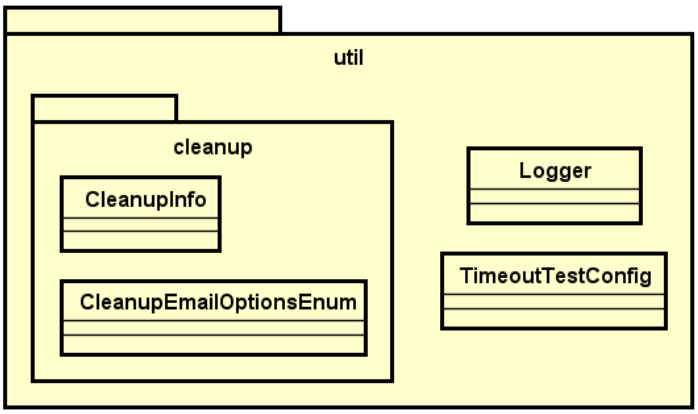

Figura 3.15: Diagrama de classes resumido do pacote *util*.

O pacote *util* contém a classe *TimeoutTestConfig*. Essa classe é a fonte de entrada para a configuração do *timeout* dos testes que serão executados pela ferramenta. A classe *Logger* captura o log de prováveis erros na ferramenta.

3.3.4 Pacote service

A Figura 3.15 apresenta um diagrama resumido das classes do pacote *service*.

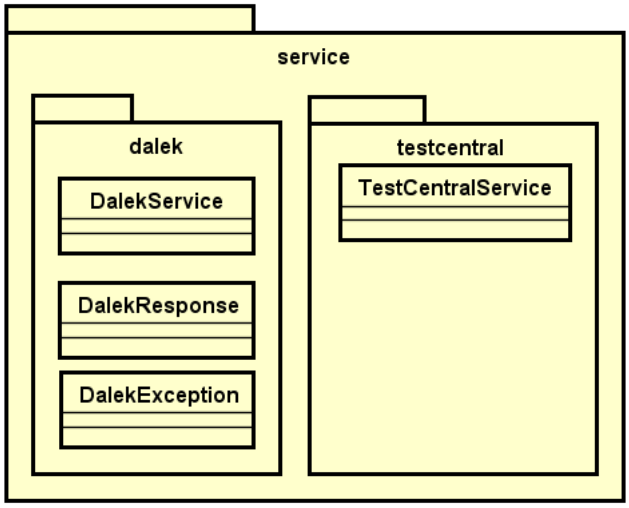

Figura 3.16: Diagrama de classes resumido do pacote *service*.

O pacote *service* tem dois subpacotes, *Dalek* e *TestCentral*. A classe *DalekService* é responsável por efetuar a comunicação com a ferramenta *Dalek.* A classe *TestCentralService*

efetua a comunicação com a ferramenta *TestCentral*, ambas ferramentas externas usadas como banco de dados de testes no projeto CIn-Motorola.

3.3.5 Pacote *dao*

A Figura 3.17 apresenta um diagrama resumido das classes do pacote *dao*.

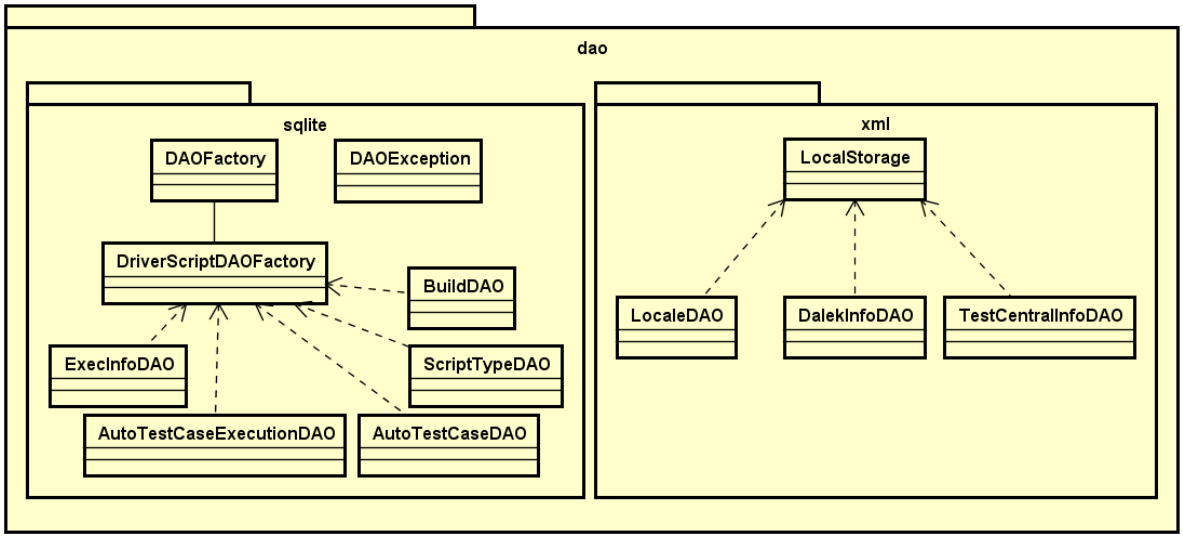

Figura 3.17: Diagrama de classes resumido do pacote *dao*.

O pacote *dao* tem dois subpacotes, *sqlite* e *xml*. A classe *DriverScriptDAOFactory* é responsável por converter o banco de dados proveniente da execução de um teste do *framework* FREVO. As principais funções das classes do subpacote *sqlite* são converter o banco de dados resultante de uma execução (Figura 3.7) em objetos e disponibilizar uma interface de acesso a esses objetos. Observe que algumas classes como *BuildDAO, ExecInfoDAO, ScriptTypeDAO, AutoTestCaseDAO* referem-se às tabelas *Build, ExecInfo, ScriptType, AutoTestCase* definidas no *framework* FREVO (figuras 3.9 e 3.10). A classe *AutoTestCaseExecutionDAO* agrupa os dados das tabelas *SubTest, SubTestResult* e *TestCaseResult* também definidas no *framework.*

O subpacote *xml* é usado para criar, editar e salvar dados locais no formato XML. A classe *LocaleDAO* gerencia uma lista de idiomas suportadas pela ferramenta na execução de testes. As classes *DalekInfoDAO* e *TestCentralInfoDAO* gerencia os dados usados para acessar as ferramentas externas *Dalek* e *TestCentral*, respectivamente.

3.3.6 Pacote *model*

A Figura 3.18 apresenta um diagrama resumido das classes do pacote *model*.

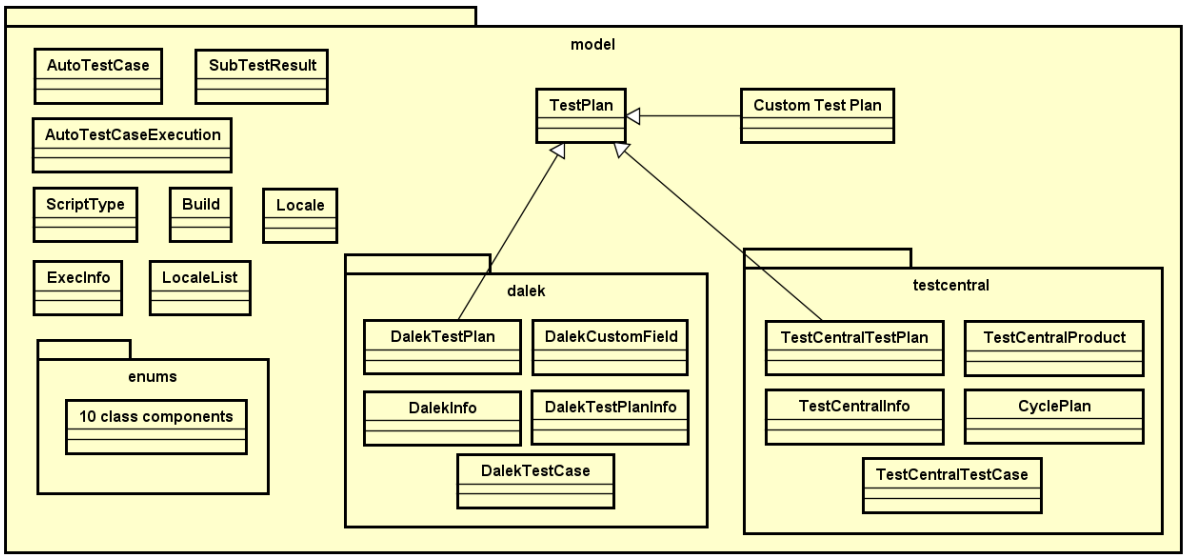

Figura 3.18: Diagrama de classes resumido do pacote *model*.

Esse pacote engloba todas as entidades básicas do sistema. Algumas classes desse pacote são utilizadas pelas classes do pacote *dao* para representar os objetos de cada tabela definida no *framework* FREVO. Por exemplo, as classes *Build*, *ExecInfo*, *ScriptType*, *Locale* e *AutoTestCase* são utilizadas pelas classes *BuildDAO*, *ExecInfoDAO*, *ScriptTypeDAO*, *LocaleDAO* e *AutoTestCaseDAO*, respectivamente. O pacote *model* tem a classe *TestPlan* com três especializações (*CustomTestPlan, DalekTestPlan* e *TestCentralTestPlan*). Essas três classes representam suítes de testes específicas de algumas funcionalidades do sistema.

Os subpacotes *dalek* e *testcentral* disponibilizam algumas classes que são usadas para representar informações que são utilizadas no pacote *service* para enviar e/ou recuperar dados das ferramentas externas.

3.3.7 Pacote execution

A Figura 3.19 apresenta um diagrama resumido das classes do pacote *execution*.

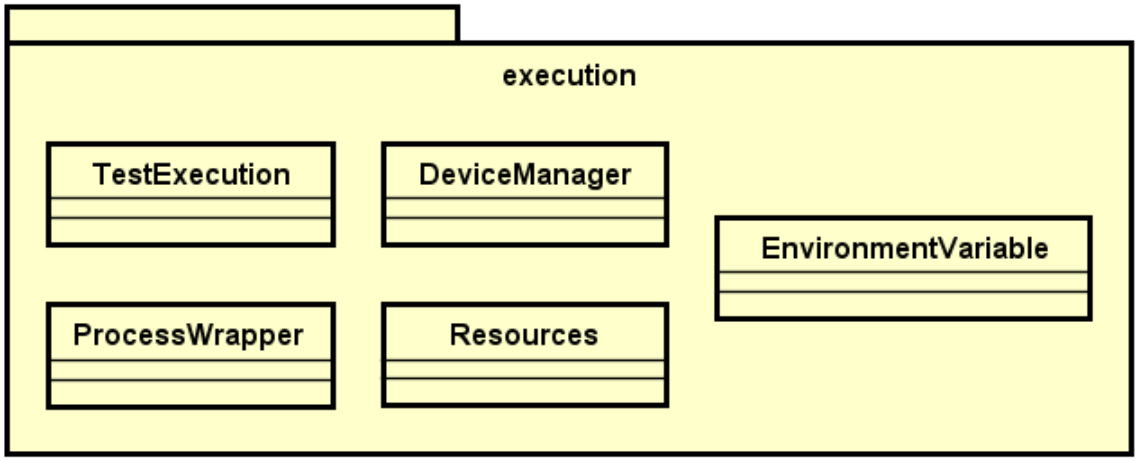

Figura 3.19: Diagrama de classes resumido do pacote *execution*.

O pacote *execution* contém todas as classes utilitárias responsáveis pela execução de *scripts* UI Automator do *framework* FREVO. Umas das principais classes do sistema é a classe *TestExecution*, que é responsável pela orquestração da execução de uma suíte de testes criada usando a ferramenta. Além de definir a ordem de execução, ela também monitora a execução capturando o log de execução dos testes e, no final da execução de uma suíte de testes, captura todas as informações da execução (banco de dados, *screenshots*, logs de execução, etc.). A classe *DeviceManager* é responsável pelo reconhecimento de dispositivos conectados na máquina, disponibilizando-os para a classe *TestExecution* iniciar a execução de uma suíte de testes. A classe *ProcessWrapper* é responsável por enviar comandos ADB para serem executados em um dispositivo conectado.

3.3.8 Integração Ferramenta-Framework

A ferramenta foi arquitetada para integrar a execução de *scripts* UI Automator que foram implementados através do *framework* FREVO. Sabemos que todo projeto UI Automator gera um arquivo JAR contendo os *scripts* e que o *framework* FREVO tem a capacidade de gerar um banco de dados a partir desse arquivo JAR.

Dessa maneira, a integração da ferramenta com projetos UI Automator desenvolvidos com o *framework* FREVO ocorre por meio da disponibilização de dois arquivos: um arquivo JAR do projeto e um banco de dados do projeto gerado no *framework* FREVO.

A Figura 3.20 ilustra os artefatos necessários para consolidar a integração entre o *framework* e a ferramenta.

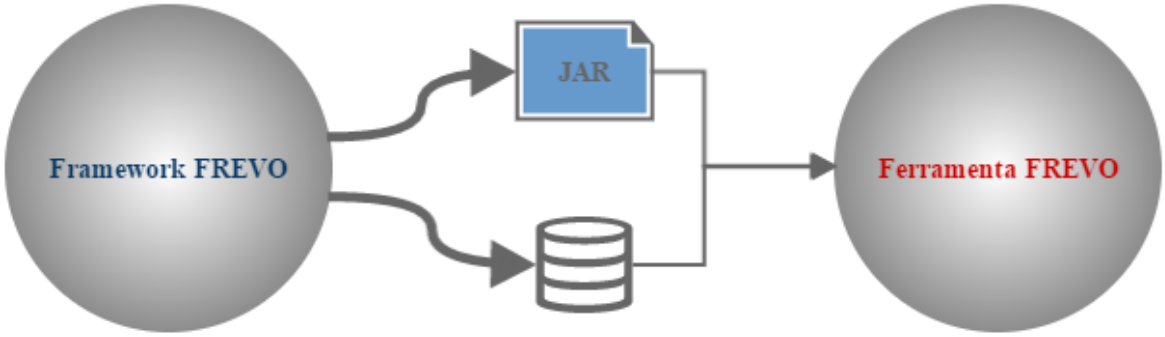

Figura 3.20: Artefatos necessários para efetuar a integração entre o *framework* e a ferramenta.

A Figura 3.21 ilustra um fluxo com os passos necessários para integrar um projeto FREVO UI Automator com a ferramenta.

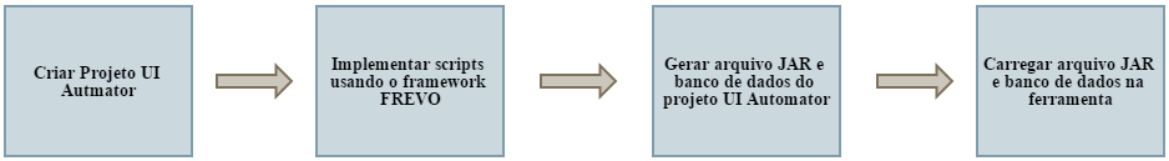

Figura 3.21: Fluxo com os passos necessários para integrar um projeto UI Automator na ferramenta.

A integração do *framework* com a ferramenta habilita a execução de *scripts* UI Automator através de uma interface gráfica intuitiva, sem a necessidade de combinar comandos ADB em um arquivo em lotes como também fornece um conjunto de funcionalidades focadas na apresentação dos resultados da execução. A partir da ferramenta, um testador pode criar uma suíte de testes simplesmente selecionando os testes desejados, podendo executá-la imediatamente. Após a execução, a ferramenta já disponibiliza o resultado dos testes na tela de resultados.

3.4 CONSIDERAÇÕES FINAIS

Neste capítulo, apresentamos os principais objetivos do trabalho, focado na construção de dois sistemas integrados: o *framework* FREVO e a ferramenta de execução de testes. No detalhamento do *framework* FREVO, descrevemos arquitetura e detalhes técnicos de implementação que culminaram na adição de novas características e na solução dos

problemas existentes. Na seção da ferramenta, além de descrevermos a arquitetura, apresentamos todas as funcionalidades que habilitaram a integração com o *framework* FREVO viabilizando a consolidação de um ambiente de execução de suítes de testes automáticos maduro e consistente.

Combinando as facilidades providas pelo *framework* FREVO (*timeout*, reexecução e *screenshot*) com as funcionalidades providas pela ferramenta (montagem de suítes, configuração dos parâmetros do framework, execução e coleta de resultados), fica evidente o enriquecimento da automação de testes para uma dimensão maior de automação de suítes de teste, provendo, dessa maneira, um nível de automação de teste inexistente ao se utilizar os *frameworks* existentes.

4 ESTUDO DE CASO

Este capítulo descreve a utilização da ferramenta FREVO em alguns projetos de testes da Motorola, relatando a experiência e os resultados obtidos ao implantar a ferramenta como uma parte fundamental do processo de automação de testes.

Vimos, na Seção 2.10.1, as dificuldades encontradas pelos testadores tanto nos cenários de criação e edição de suítes de testes, quanto na parte referente à execução de testes (definição de *retry*, *timeout*). É sabido que antes da ferramenta FREVO, as atividades de testes eram realizadas rotineiramente de maneira semiautomática. Esse processo semiautomático funcionava da seguinte maneira: um testador tinha uma atividade designada para execução de todos os testes (manuais e automáticos) de uma suíte. Primeiramente, o testador precisava identificar manualmente quais testes daquela suíte eram automáticos, para então poder criar um arquivo em lotes que representaria a suíte destes testes automáticos. Mostramos, na Seção 2.10.1, a maioria dos problemas que acabam impactando a produtividade dos testadores, como, por exemplo, aqueles cenários onde a execução fica bloqueada devido a um erro no *script* ou quando o testador precisa reexecutar uma suíte somente com testes que apresentaram problemas. Cenários como este são comuns em todos os *frameworks* de automação de testes e sempre precisam do acompanhamento do testador, o que não é ideal.

Com a implantação da ferramenta FREVO nos projetos, o testador não precisa mais gastar tempo com a criação/edição de suítes de testes como também pode continuar com suas atividades paralelas sem se preocupar com o andamento da execução. Agora, basta que o usuário forneça as informações da suíte de testes que a ferramenta identifica os testes automáticos e já cria uma suíte de testes. Para iniciar a execução, o testador só precisa pressionar o botão *Run.*

No processo de testes, após a execução de uma suíte de testes, o testador responsável pela execução inicia o processo de análise dos resultados e logo em seguida pode efetuar o envio dos resultados dos testes para as ferramentas externas.

4.1 FUNCIONAMENTO PASSO-A-PASSO

Essa seção é destinada para descrever o funcionamento da ferramenta, desde as funcionalidades de integrar um projeto FREVO UI Automator até as funcionalidades de criação de suíte de testes e visualização de resultados. Dependendo do tipo de execução, os resultados dos testes já podem ser enviados às ferramentas externas de gerenciamento de testes.

A Figura 4.1 ilustra a tela principal da ferramenta. A tela é composta por três abas:

- TestCentral;
- Dalek;
- Custom execution.

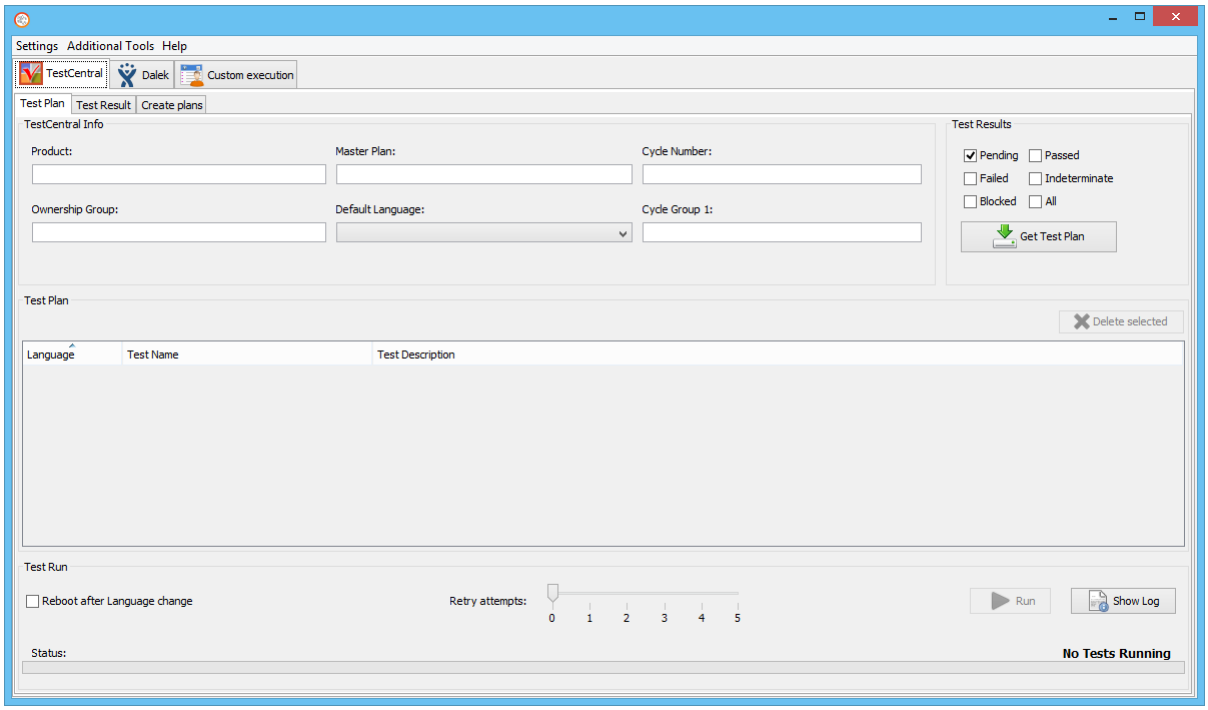

Figura 4.1: Tela principal da ferramenta FREVO.

O primeiro passo para integrar os *scripts* de um projeto FREVO UI Automator na ferramenta é fornecer os dois arquivos necessários (arquivo JAR e banco de dados) para carregar todos os *scripts* e suas informações.

A ferramenta disponibiliza uma opção (menu *Settings* → *Load scripts*) para integrar um novo projeto FREVO UI Automator.

A Figura 4.2 apresenta a caixa de diálogo onde o usuário pode informar o arquivo JAR e o banco de dados do projeto que deseja integrar na ferramenta.

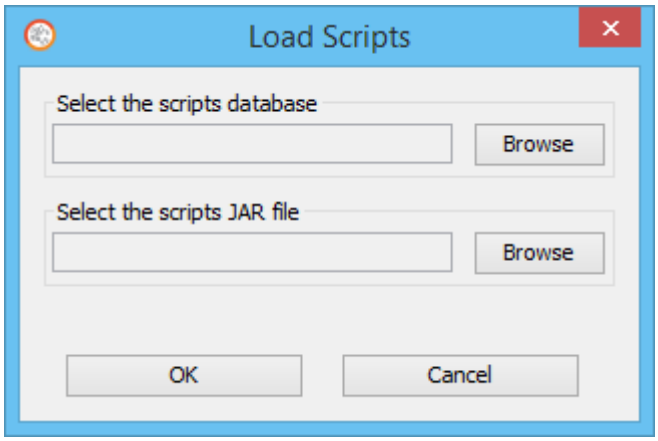

Figura 4.2: Caixa de diálogo para informar o arquivo JAR e o banco de dados de um projeto UI Automator.

Assim que um usuário carrega os dois arquivos necessários, nesse momento a ferramenta está apta para criar suítes usando os testes do projeto carregado e logicamente iniciar a execução desses testes. A Figura 4.3 apresenta a tela da ferramenta (aba *Custom execution*) após o carregamento dos arquivos do *ProjetoUiAutomator* visto na Seção 3.2.4.

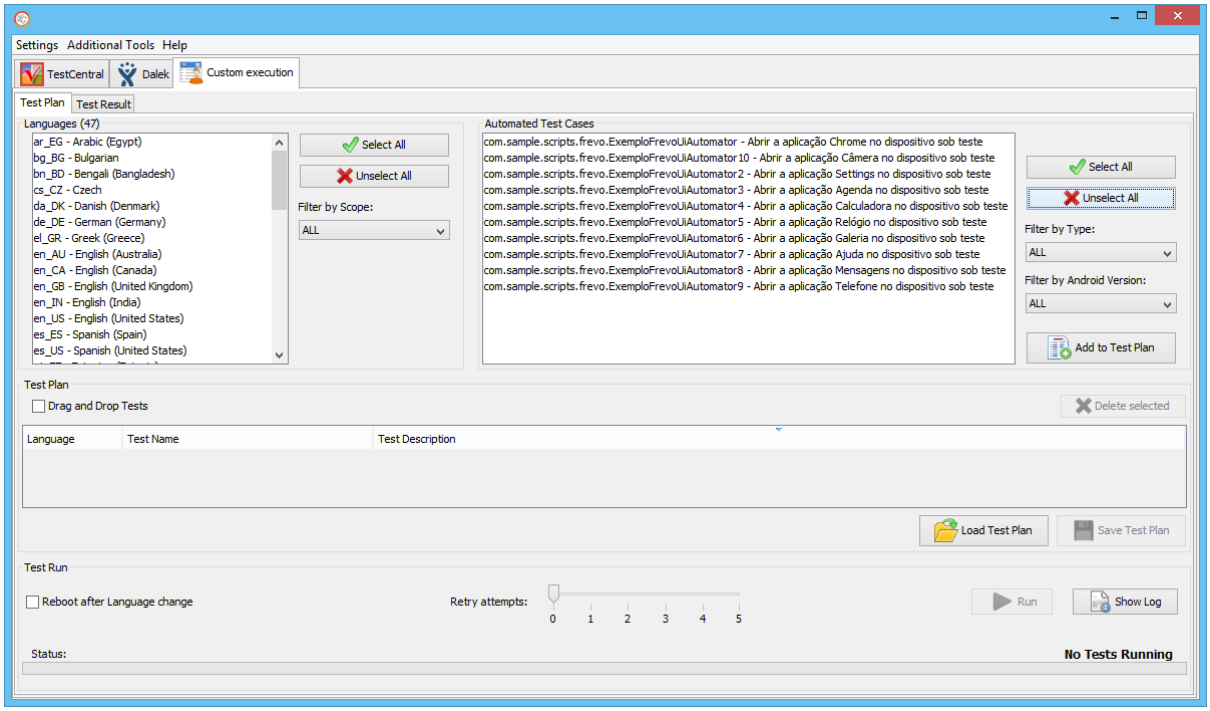

Figura 4.3: Aba *Custom execution* da ferramenta após o carregamento dos arquivos do projeto *ProjetoUiAutomator*.

Na aba *Custom* execution, a lista *Automated Test Cases* contém dez elementos, que representam os testes automáticos contidos no arquivo JAR do projeto carregado. Os dez testes automáticos referem-se exatamente aos testes vistos no banco de dados gerado pelo *framework* FREVO (Figura 3.12).

Agora, vamos supor que um testador deseja criar uma suíte de testes contendo os dez testes automáticos da lista. Para isso, o testador precisa apenas selecionar um idioma e os testes que deseja adicionar na suíte. Após isso, basta clicar no botão *Add to Test Plan* que os testes selecionados serão adicionados à suíte de testes, que está automaticamente pronta para ser executada.

A Figura 4.4 exemplifica a criação de uma suíte selecionando o idioma *en_US - English (United States)* e os dez testes disponíveis na lista de testes automatizados.

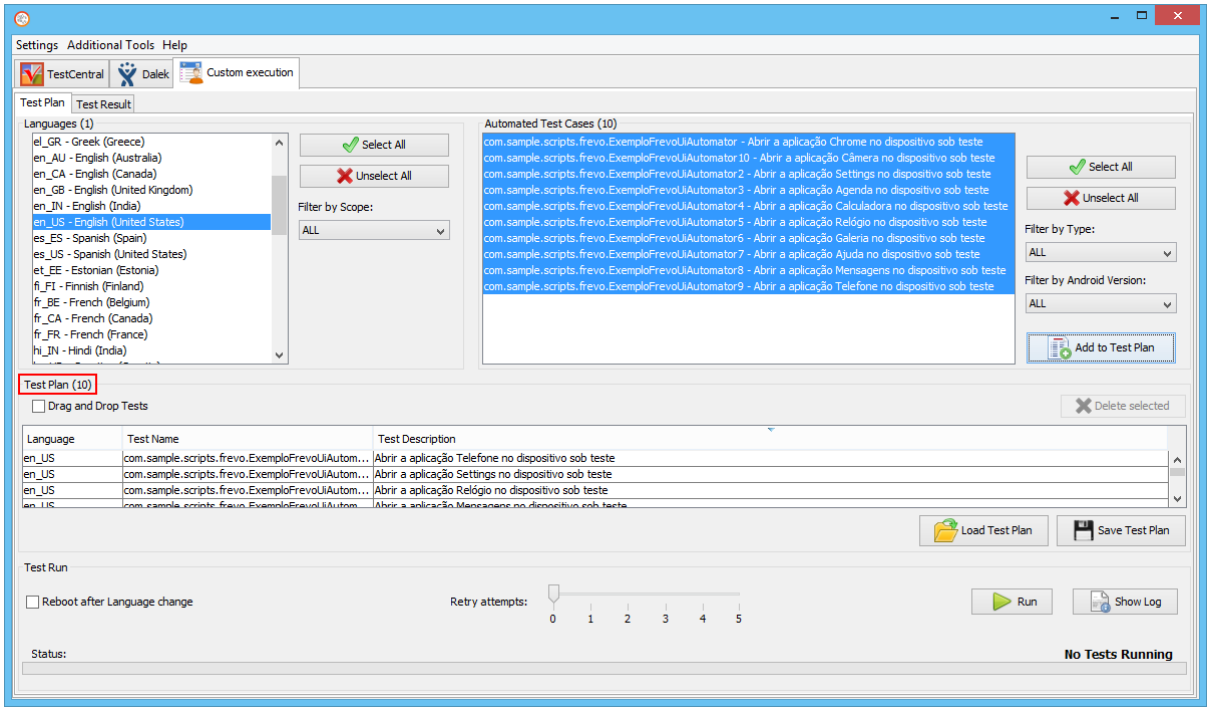

Figura 4.4: Exemplo de criação de uma suíte de testes através da ferramenta

Após selecionar o idioma e os testes desejados e clicar no botão *Add to Test Plan*, a ferramenta adiciona os dez testes automáticos (veja o destaque na Figura 4.4, uma mensagem *Test Plan (10)*) à suíte. Uma vez que a suíte está configurada, o testador pode iniciar a execução dos testes clicando no botão *Run*. Dando prosseguimento ao processo, vamos simular a execução dessa suíte de testes.

A Figura 4.5 apresenta a tela da ferramenta durante a execução dos testes.

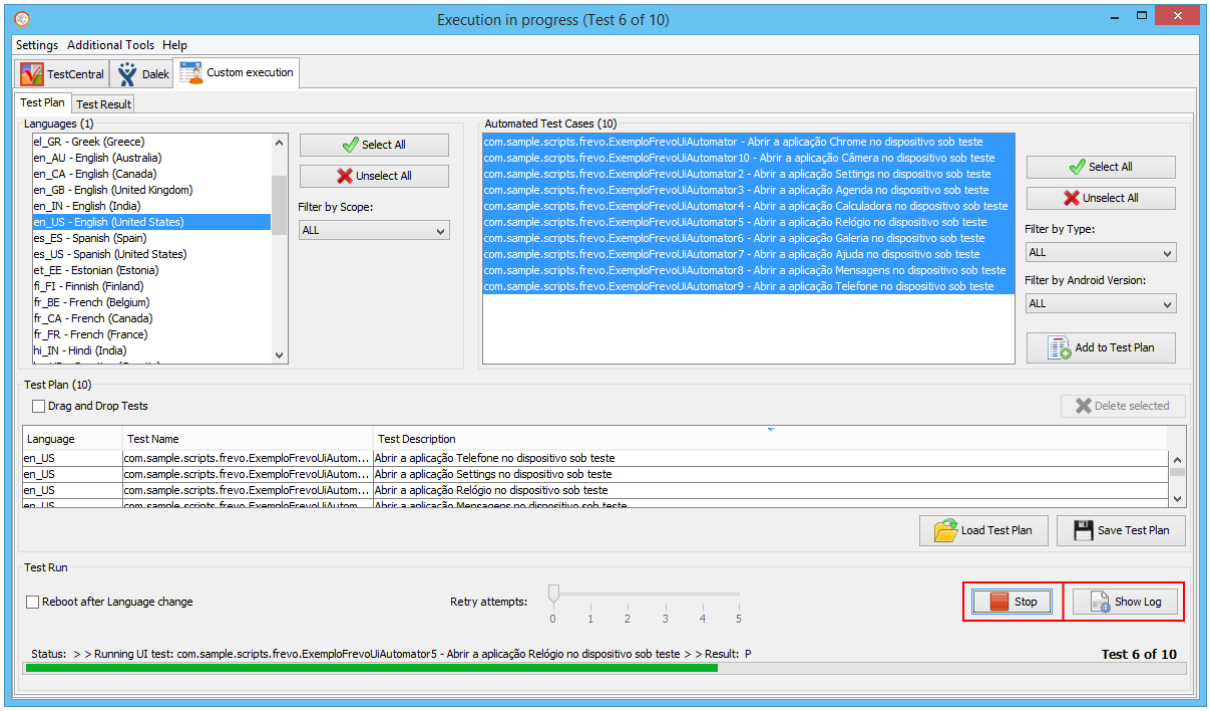

Figura 4.5: Tela da ferramenta durante a execução de testes.

Observe que durante a execução de testes, a ferramenta apresenta o status geral da execução, onde a informação *Test 6 of 10* informa que o sexto teste da suíte está sendo executado. Pode-se observar também um botão *Stop*, onde o usuário pode interromper a execução a qualquer momento. O botão *Show Log* abre uma janela que contém dois logs: um log que ilustra as ações efetuadas pela ferramenta e quais testes foram executados, e o outro log é aquele log de informações gerado pelo *framework* UI Automator, que traz os detalhes da execução do *script*. As figuras 4.6 e 4.7 apresentam as telas respectivas ao log de ações da ferramenta e ao log de informações gerado pelo *framework* UI Automator.

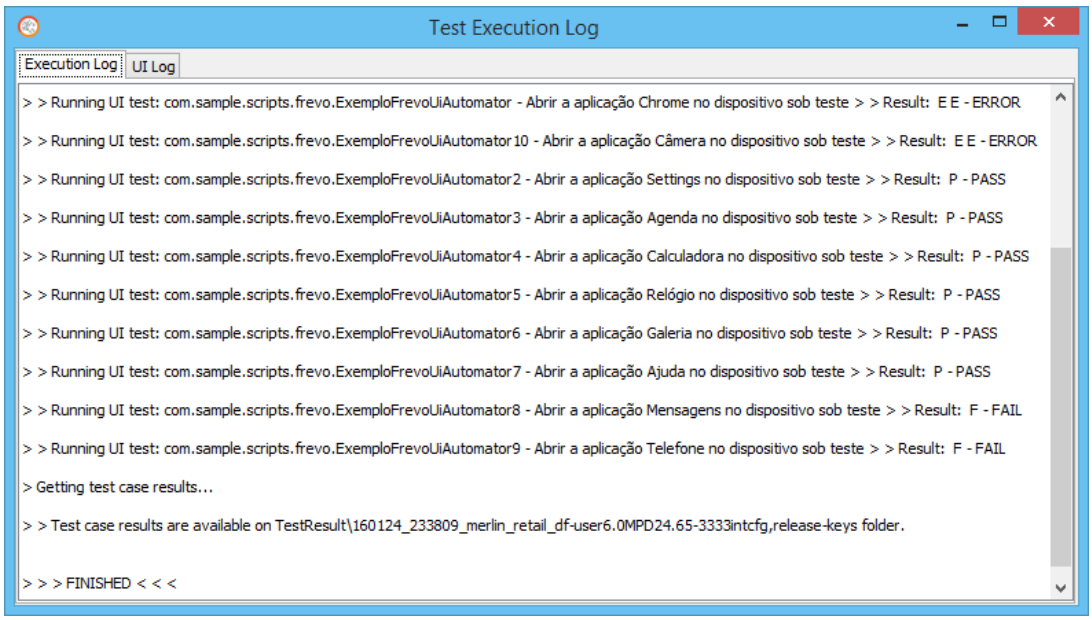

Figura 4.6: Janela que apresenta o log de informações geradas pela ferramenta.

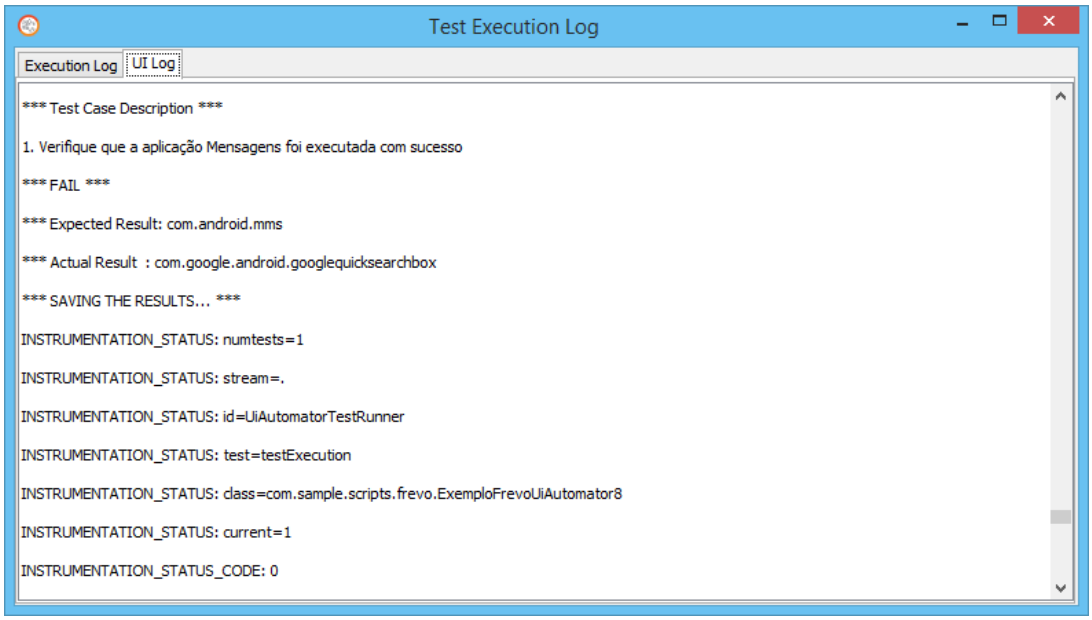

Figura 4.7: Janela que apresenta o log de informações gerados pelo *framework* UI Automator.

Continuando, assim que a ferramenta identifica que todos os testes de uma suíte foram executados, ela notifica o usuário que a execução foi finalizada e faz a captura dos artefatos gerados pela execução (banco de dados, *screenshots* e log de informações). Após a captura desses artefatos, a ferramenta é automaticamente direcionada para uma tela de resultados. A tela de resultados pode ser vista na Figura 4.8.

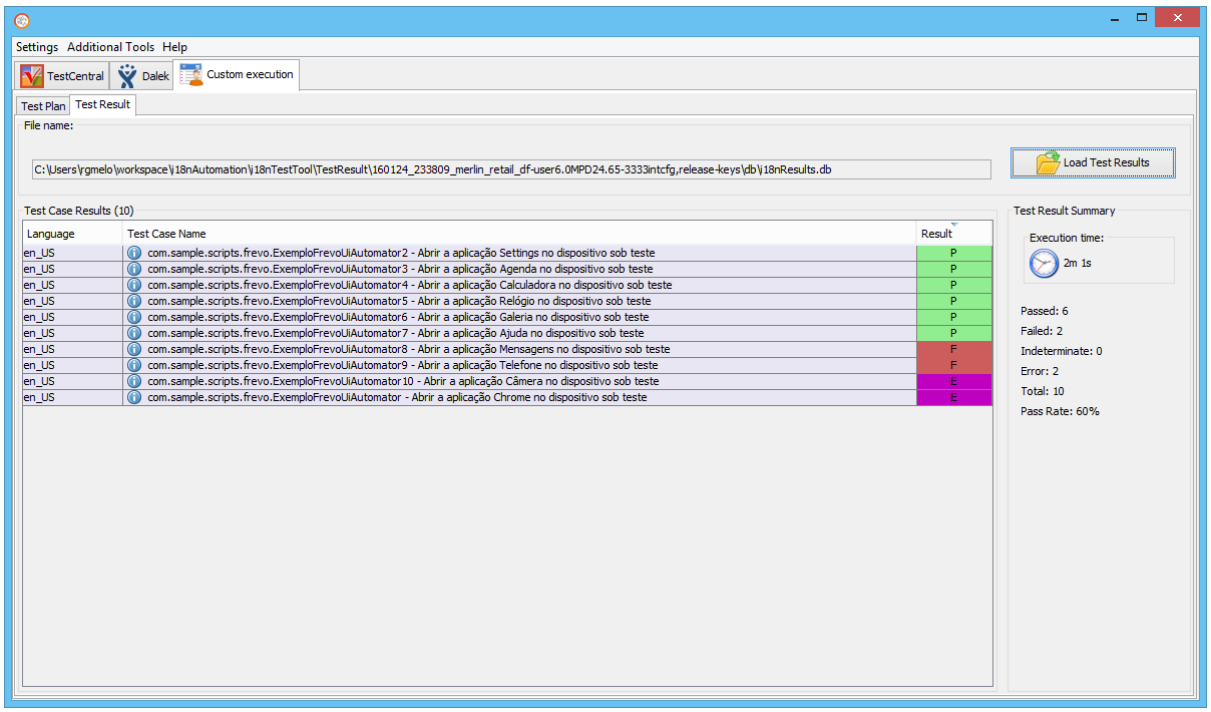

Figura 4.8: Tela de resultados da execução de uma suíte de testes na ferramenta.

Finalizando a simulação da execução de uma suíte, podemos observar que a ferramenta apresentou na tela o resultado da execução dos dez testes da suíte. A tela de

resultados apresenta o resultado específico e traz um resumo geral da execução. Nessa execução, podemos observar que o tempo total da execução foi de 2m 1s. Dos dez resultados, seis resultados tiveram o valor P (*Pass*), dois resultados o valor F (*Fail*) e dois resultados o valor E (*Error*) obtendo uma taxa de sucesso de 60%.

Vamos supor que o testador precise analisar cada um dos resultados da execução, passando pelos testes que passaram e principalmente pelos testes que resultaram em falha e erro, pois esses precisam ser melhor analisados. Para acessar os detalhes da execução de um teste através da ferramenta, basta que o usuário dê um duplo clique no teste desejado que a ferramenta abrirá uma tela de visualização dos resultados, onde todas as informações do teste serão disponibilizadas (descrição das validações, valor atual, valor esperado, *screenshot*, etc.).

A Figura 4.9 ilustra a tela de visualização de resultados de um teste que resultou em falha, o teste *ExemploFrevoUiAutomator8.*

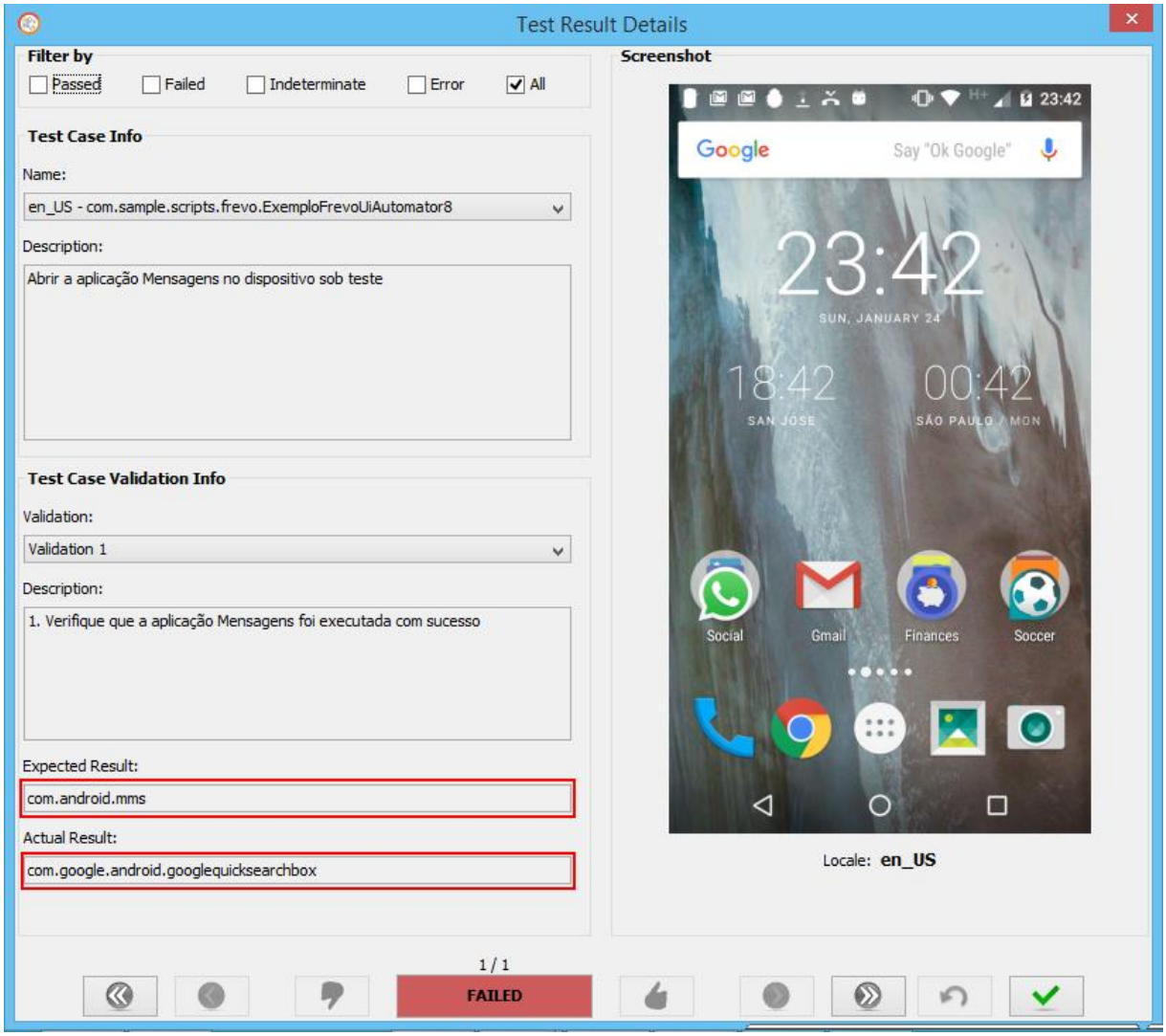

Figura 4.9: Tela de visualização de resultados de um teste.

A tela de visualização de resultados traz todas as informações do teste. É possível visualizar a descrição do teste, navegar pelas validações verificando as informações de resultado atual, resultado esperado e *screenshot*, provendo todas as informações úteis que um testador precisa no momento de analisar o resultado de um teste. No exemplo da Figura 4.9, facilmente identificamos a falha na validação, visto que o resultado esperado da validação era o valor *com.android.mms* e o valor atual foi *com.google.android.googlequicksearchbox*. Além disso, as imagens resultantes das validações também são bastante úteis nos contextos de análise de resultados de testes.

Destacando outra característica importante da ferramenta na parte de gerenciamento da execução de testes, vamos supor que um testador deseja estabelecer um tempo máximo (*timeout)* para a execução de uma suíte de testes. A ferramenta disponibiliza uma opção (menu *Settings* → *Change Timeout Test Execution*) com esta finalidade. Veja a Figura 4.10 abaixo.

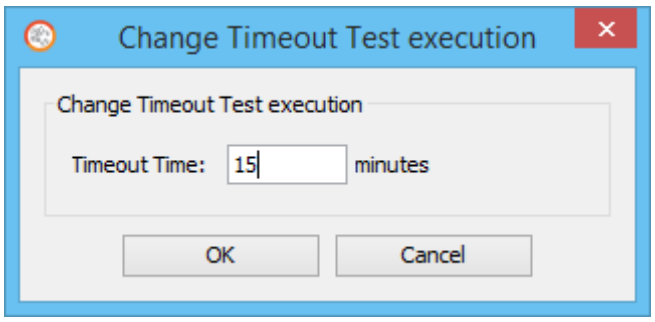

Figura 4.10: Caixa de diálogo para definir um tempo máximo (*timeout*) de execução para cada teste.

Ainda sobre o gerenciamento da execução de testes, um cenário bastante comum na execução de testes automáticos é a necessidade que os testadores têm de reexecutar determinados testes que resultaram em falha ou erro durante a execução. Na maioria das vezes, é preciso analisar quais testes falharam após isso, criar uma nova execução somente com os testes identificados com o resultado falha ou erro. A ferramenta resolve esse problema com a opção *Retry attempts*. Através dessa opção, antes de iniciar a execução de uma suíte de testes, o testador define um número de tentativas – de zero a cinco – de reexecução dos testes que apresentaram problemas. Após a execução da suíte, a ferramenta fará a análise automática dos resultados dos testes e inicia uma nova execução considerando somente aqueles testes que resultaram em falha ou erro.

A Figura 4.11 destaca a opção *Retry attempts* definida na ferramenta.

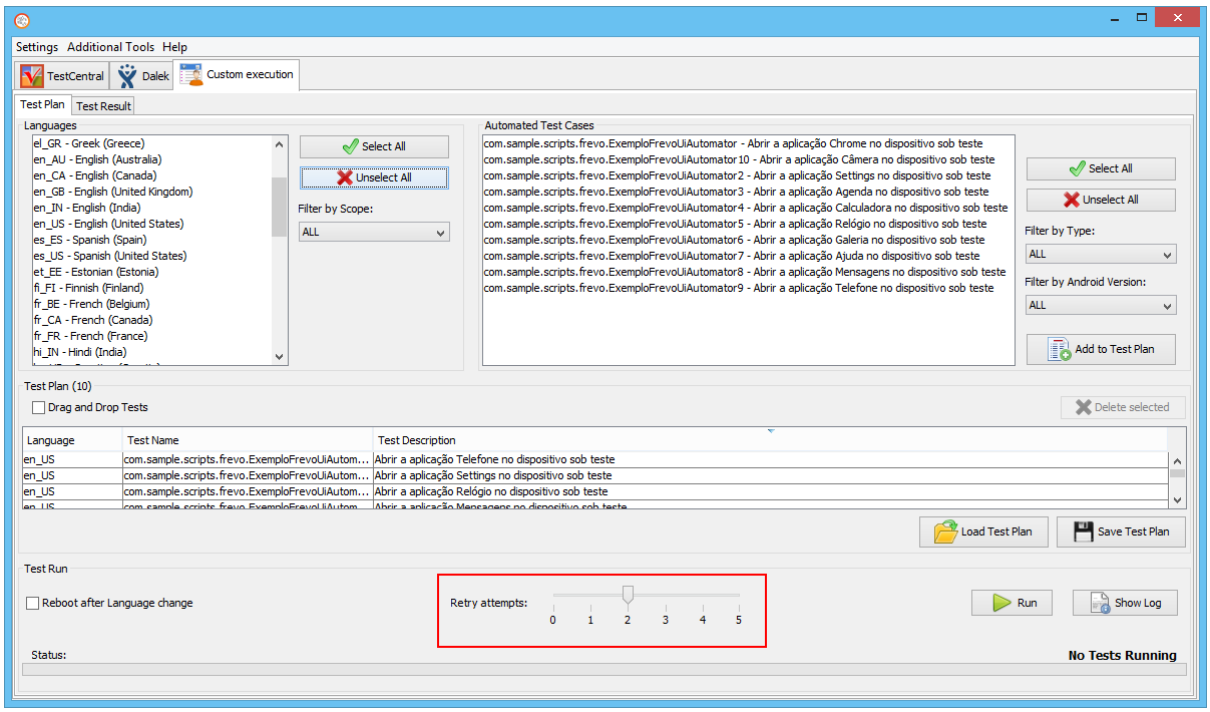

Figura 4.11: Tela da ferramenta destacando a opção *Retry attempts.*

Na Figura 4.11, a opção *Retry attempts* está marcada no valor 2 (dois). Isto significa que, caso a execução de uma suíte apresente falha ou erro em seus resultados, a ferramenta irá reexecutar os testes que apresentaram problemas pelo menos duas vezes. Para demonstrar o funcionamento dessa funcionalidade, simulamos a execução de outra suíte de testes, que obrigatoriamente apresenta problemas na primeira execução.

A Figura 4.12 mostra o log das ações efetuadas pela ferramenta, mostrando que a ferramenta reexecutou alguns testes que apresentaram problemas.

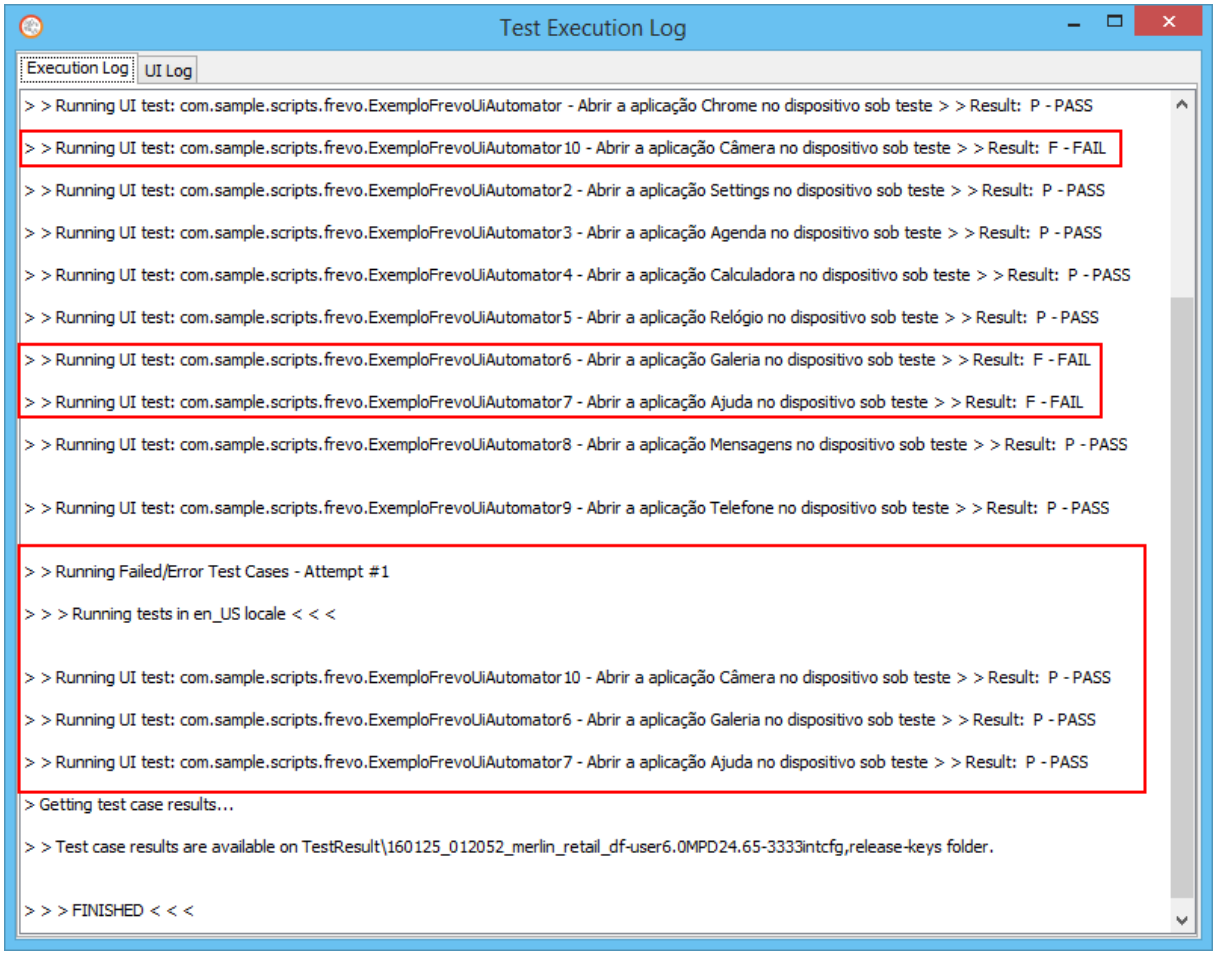

Figura 4.12: Tela que apresenta o log de ações efetuadas pela ferramenta, considerando uma suíte com a opção *Retry attempts* ativada (valor > 0).

Ao analisar a execução através da Figura 4.12, nota-se que, na primeira execução, dos dez testes executados, três deles apresentaram problemas, sendo eles: *ExemploFrevoUiAutomator10, ExemploFrevoUiAutomator6* e *ExemploFrevoUiAutomator7*. Como a opção *Retry attempts* estava marcada com o valor dois (2), a ferramenta automaticamente identificou os testes com problemas e iniciou a execução somente dos testes problemáticos. Observe que na primeira tentativa de reexecução, os três testes executados resultaram em sucesso, não havendo, portanto, necessidade de executar novamente, uma vez que os resultados dos testes já são satisfatórios.

Os exemplos citados aqui foram efetuados na aba *Custom execution*. No entanto, as abas *TestCentral* e *Dalek* possuem as mesmas características e funcionalidades. A única diferença está na execução de suítes de testes, uma vez que a ferramenta carrega as suítes de testes que estão definidas nessas ferramentas externas. Ao carregar suítes de testes das ferramentas externas, a ferramenta faz um mapeamento de verificação de testes automáticos,

ou seja, a ferramenta verifica dentre todos os testes da suíte, quais deles estão presentes na ferramenta e, portanto, podem ser executados de maneira automática.

Ainda sobre a comunicação com ferramentas externas, a ferramenta FREVO também dá suporte à criação de suítes nesses sistemas externos, isto é, os usuários podem criar suítes de testes diretamente da ferramenta FREVO, sem precisar abrir as ferramentas externas para tal finalidade, diminuindo assim o tempo no gerenciamento (criação/edição) e execução de suítes de testes.

4.2 TEMPOS DE EXECUÇÃO

Após descrevermos a utilização do *framework* e ferramenta FREVO em uma execução particular, introduzimos uma descrição das medições de tempo das execuções já realizadas até o momento com a ferramenta. Também coletamos o tempo total da execução da mesma suíte utilizando-se a execução tradicional (semiautomática).

Não está no escopo deste trabalho fazermos uma análise estatística destes dados, mas apenas para apresentar números preliminares de uso da ferramenta em produção e introduzir situações em que seus tempos de execução são, em alguns casos, dominantes em relação à técnica semiautomática. Um experimento controlado e uma análise estatística serão feitas em trabalhos futuros.

Sabendo das diferenças entre uma execução semiautomática e uma execução essencialmente automática, decidimos comparar o tempo de execução entre elas. No processo da captura de dados reais das ferramentas externas, estabelecemos uma regra geral quanto à captura dos tempos inicial e final de cada execução.

Como tempo inicial de execução de cada suíte, definimos o horário em que o testador inicia a execução da suíte de testes, isto é, quando o mesmo realiza uma transição de status (*NEW* → *TESTING*). Na ausência dessa informação, capturamos o horário referente à primeira atualização do resultado de um teste (menor horário dentre as execuções), tanto para as execuções semiautomáticas quanto para as execuções automáticas. Para o tempo final de execução de uma suíte, definimos como sendo o horário em que o testador finaliza a execução da suíte de testes, ou seja, realizando outra transição de status (*TESTING* → *CLOSED*). Do mesmo modo, na ausência dessa informação, consideramos o horário referente à última atualização do resultado de um teste (maior horário dentre as execuções), tanto para as execuções semiautomáticas quanto para as execuções automáticas.

As tabelas que serão apresentadas a seguir trazem os resultados de 100 (cem) execuções, distribuídas em 18 produtos diferentes (variação de produto e/ou versão de sistema operacional). Vale salientar que nessa avaliação é considerada somente uma suíte de testes, composta por 30 casos de teste, ou seja, o número de testes executados é igual para todas as execuções.

A Tabela 4.1 apresenta os resultados das execuções realizadas no Produto 1.

Tabela 4.1: Resultados das execuções realizadas no Produto 1.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	09/03/15	10:41:00	16:54:00	6:13:00
2	Automática	06/05/15	10:07:00	12:42:00	2:35:00
3	Automática	19/05/15	18:14:00	21:49:00	3:35:00
4	Automática	10/06/15	14:13:00	17:08:00	2:55:00
5	Automática	29/06/15	10:31:00	12:01:00	1:30:00
6	Automática	05/10/15	10:32:00	13:23:00	2:51:00

Nas execuções do Produto 1, verificamos que ocorreu uma *dominação completa* do FREVO. Para o Produto 1, conseguimos capturar somente uma execução semiautomática que continha todas as execuções da suíte.

A *dominação completa* ocorre quando o tempo máximo alcançado na execução automática ainda é menor que o tempo mínimo da execução semiautomática.

A Tabela 4.2 apresenta os resultados das execuções realizadas no Produto 2.

Tabela 4.2: Resultados das execuções realizadas no Produto 2.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	09/03/15	12:30:00	19:10:00	6:40:00
2	Semiautomática	15/04/15	14:30:00	19:26:00	4:56:00
3	Semiautomática	22/04/15	14:49:00	17:47:00	2:58:00
4	Automática	19/05/15	14:30:00	16:33:00	2:03:00
5	Automática	27/05/15	10:04:00	14:01:00	3:57:00
6	Automática	01/06/15	12:49:00	15:10:00	2:21:00
7	Automática	08/06/15	12:48:00	14:13:00	1:25:00
8	Automática	28/08/15	10:48:00	12:19:00	1:31:00

Nas execuções do Produto 2, verificamos que não houve *dominação completa* do FREVO.

A Tabela 4.3 apresenta os resultados das execuções realizadas no Produto 3.

Tabela 4.3: Resultados das execuções realizadas no Produto 3.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	10/03/15	13:21:00	16:17:00	2:56:00
2	Semiautomática	22/04/15	9:13:00	13:09:00	3:56:00
3	Semiautomática	26/04/15	13:58:00	16:05:00	2:07:00
4	Automática	16/06/15	14:58:00	17:30:00	2:32:00
5	Automática	22/06/15	15:17:00	19:13:00	3:56:00
6	Automática	29/06/15	11:28:00	13:17:00	1:49:00
7	Automática	30/06/15	11:38:00	13:02:00	1:24:00
8	Automática	31/07/15	9:07:00	10:27:00	1:20:00

Nas execuções do Produto 3, constatamos que não houve *dominação completa* do FREVO, visto que o tempo máximo alcançado da execução automática foi maior que o tempo mínimo da execução semiautomática.

A Tabela 4.4 apresenta os resultados das execuções realizadas no Produto 4.

Tabela 4.4: Resultados das execuções realizadas no Produto 4.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	10/03/15	17:19:00	20:52:00	3:33:00
2	Semiautomática	15/04/15	19:23:00	22:21:00	2:58:00
3	Automática	04/05/15	14:49:00	15:51:00	1:02:00
4	Automática	18/05/15	10:25:00	12:57:00	2:32:00
5	Automática	27/05/15	16:49:00	20:28:00	3:39:00
6	Automática	01/06/15	10:42:00	14:23:00	3:41:00
7	Automática	10/06/15	13:09:00	16:03:00	2:54:00

Nas execuções do Produto 4, também verificamos que não houve *dominação completa* do FREVO, pelo mesmo motivo visto no Produto 3.

A Tabela 4.5 apresenta os resultados das execuções realizadas no Produto 5.

Tabela 4.5: Resultados das execuções realizadas no Produto 5.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	22/04/15	14:04:00	18:10:00	4:06:00
2	Semiautomática	09/06/15	11:19:00	16:47:00	5:28:00
3	Automática	01/06/15	12:55:00	14:39:00	1:44:00
4	Automática	02/07/15	11:52:00	13:45:00	1:53:00
5	Automática	31/08/15	10:53:00	14:32:00	3:39:00
6	Automática	14/09/15	11:21:00	14:53:00	3:32:00
7	Automática	28/09/15	8:14:00	10:01:00	1:47:00

Nas execuções do Produto 5, constatamos que houve *dominação completa* do FREVO, uma vez que as duas condições necessárias foram satisfeitas.

A Tabela 4.6 apresenta os resultados das execuções realizadas no Produto 6.

Tabela 4.6: Resultados das execuções realizadas no Produto 6.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	11/05/15	10:16:00	18:35:00	8:19:00
2	Automática	27/05/15	12:34:00	13:44:00	1:10:00
3	Automática	02/06/15	15:42:00	17:21:00	1:39:00
4	Automática	10/06/15	12:44:00	14:27:00	1:43:00
5	Automática	16/06/15	13:28:00	19:07:00	5:39:00
6	Automática	28/07/15	11:15:00	15:32:00	4:17:00

Nas execuções do Produto 6, podemos constar que houve *dominação completa* do FREVO. Neste produto, conseguimos capturar somente uma execução semiautomática que continha todas as execuções da suíte.

A Tabela 4.7 apresenta os resultados das execuções realizadas no Produto 7.

Tabela 4.7: Resultados das execuções realizadas no Produto 7.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	26/03/15	13:59:00	17:17:00	3:18:00
2	Semiautomática	27/03/15	10:45:00	16:19:00	5:34:00
3	Semiautomática	08/04/15	9:23:00	15:35:00	6:12:00
4	Automática	29/04/15	12:11:00	15:09:00	2:58:00
5	Automática	04/05/15	11:32:00	13:08:00	1:36:00
6	Automática	27/07/15	13:08:00	15:22:00	2:14:00
7	Automática	07/08/15	8:56:00	9:46:00	0:50:00
8	Automática	20/10/15	11:26:00	14:12:00	2:46:00

Nas execuções do Produto 7, averiguamos que houve *dominação completa* do FREVO.

A Tabela 4.8 apresenta os resultados das execuções realizadas no Produto 8.

Tabela 4.8: Resultados das execuções realizadas no Produto 8.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	06/05/15	12:20:00	18:08:00	5:48:00
2	Automática	27/07/15	12:32:00	14:13:00	1:41:00
3	Automática	07/08/15	12:04:00	13:20:00	1:16:00

Nas execuções do Produto 8, averiguamos que houve *dominação completa* do FREVO. Neste produto, somente uma execução semiautomática contendo todas as execuções pôde ser recuperada.

A Tabela 4.9 apresenta os resultados das execuções realizadas no Produto 9.

Tabela 4.9: Resultados das execuções realizadas no Produto 9.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	28/01/15	8:55:00	15:39:00	6:44:00
2	Semiautomática	04/02/15	9:13:00	13:36:00	4:23:00
3	Semiautomática	05/02/15	13:11:00	15:43:00	2:32:00
4	Semiautomática	16/03/15	17:19:00	19:58:00	2:39:00
5	Semiautomática	13/10/15	10:08:00	16:53:00	6:45:00
6	Automática	28/07/15	16:47:00	20:52:00	4:05:00
7	Automática	06/08/15	15:49:00	17:13:00	1:24:00
8	Automática	06/08/15	10:39:00	12:05:00	1:26:00

Nas execuções do Produto 9, verificamos que não houve *dominação completa* do FREVO.

A Tabela 4.10 apresenta os resultados das execuções realizadas no Produto 10.

Tabela 4.10: Resultados das execuções realizadas no Produto 10.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	25/11/14	9:18:00	16:43:00	7:25:00
2	Semiautomática	22/01/15	13:54:00	17:07:00	3:13:00
3	Automática	06/05/15	10:16:00	11:17:00	1:01:00
4	Automática	05/06/15	11:38:00	15:56:00	4:18:00
5	Automática	27/07/15	12:25:00	13:29:00	1:04:00
6	Automática	07/08/15	8:30:00	10:30:00	2:00:00

Nas execuções do Produto 10, verificamos que não houve *dominação completa* do FREVO.

A Tabela 4.11 apresenta os resultados das execuções realizadas no Produto 11.

Tabela 4.11: Resultados das execuções realizadas no Produto 11.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	20/01/15	10:05:00	15:47:00	5:42:00
2	Semiautomática	26/01/15	13:26:00	16:02:00	2:36:00
3	Automática	14/08/15	12:24:00	15:13:00	2:49:00
4	Automática	14/08/15	20:11:00	22:36:00	2:25:00
5	Automática	31/08/15	10:37:00	11:52:00	1:15:00
6	Automática	31/08/15	12:54:00	15:49:00	2:55:00

Nas execuções do Produto 11, pudemos verificar que não houve *dominação completa* do FREVO.

A Tabela 4.12 apresenta os resultados das execuções realizadas no Produto 12.

Tabela 4.12: Resultados das execuções realizadas no Produto 12.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	20/01/15	11:20:00	17:43:00	6:23:00
2	Semiautomática	26/01/15	15:03:00	17:15:00	2:12:00
3	Automática	20/08/15	16:17:00	17:10:00	0:53:00
4	Automática	31/08/15	9:57:00	11:11:00	1:14:00

Nas execuções do Produto 12, verificamos que houve *dominação completa* do FREVO.

A Tabela 4.13 apresenta os resultados das execuções realizadas no Produto 13.

Tabela 4.13: Resultados das execuções realizadas no Produto 13.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	05/02/15	18:09:00	23:24:00	5:15:00
2	Semiautomática	10/02/15	13:34:00	16:23:00	2:49:00
3	Automática	19/08/15	9:44:00	11:51:00	2:07:00
4	Automática	31/08/15	11:17:00	12:48:00	1:31:00

Nas execuções do Produto 13, constatamos que houve *dominação completa* do FREVO.

A Tabela 4.14 apresenta os resultados das execuções realizadas no Produto 14.

Tabela 4.14: Resultados das execuções realizadas no Produto 14.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	23/02/15	14:02:00	20:51:00	6:49:00
2	Semiautomática	13/03/15	18:44:00	21:42:00	2:58:00
3	Automática	20/05/15	13:16:00	15:24:00	2:08:00
4	Automática	02/06/15	11:59:00	15:24:00	3:25:00
5	Automática	20/08/15	10:15:00	11:42:00	1:27:00

Nas execuções do Produto 14, averiguamos que não houve *dominação completa* do FREVO.

A Tabela 4.15 apresenta os resultados das execuções realizadas no Produto 15.

Tabela 4.15: Resultados das execuções realizadas no Produto 15.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	13/03/15	10:53:00	15:31:00	4:38:00
2	Automática	02/06/15	13:43:00	16:00:00	2:17:00
3	Automática	19/08/15	11:14:00	12:44:00	1:30:00

Nas execuções do Produto 15, verificamos que houve *dominação completa* do FREVO. Neste produto, somente uma execução semiautomática contendo todos os testes foi recuperada.

A Tabela 4.16 apresenta os resultados das execuções realizadas no Produto 16.

Tabela 4.16: Resultados das execuções realizadas no Produto 16.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	24/02/15	18:36:00	23:23:00	4:47:00
2	Semiautomática	26/02/15	16:45:00	19:42:00	2:57:00
3	Automática	10/08/15	13:04:00	14:59:00	1:55:00
4	Automática	20/08/15	16:21:00	18:43:00	2:22:00

Nas execuções do Produto 16, averiguamos que houve *dominação completa* do FREVO.

A Tabela 4.17 apresenta os resultados das execuções realizadas no Produto 17.

Tabela 4.17: Resultados das execuções realizadas no Produto 17.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	26/02/15	17:48:00	20:44:00	2:56:00
2	Automática	18/08/15	11:27:00	13:31:00	2:04:00

Nas execuções do Produto 17, observamos que houve *dominação completa* do FREVO. Para o Produto 17, conseguimos capturar somente uma execução de cada tipo, uma vez que o produto em questão estava começando a ser utilizado nas atividades de testes automáticos.

A Tabela 4.18 apresenta os resultados das execuções realizadas no Produto 18.

Tabela 4.18: Resultados das execuções realizadas no Produto 18.

Nº execução	Tipo	Data	Tempo inicial	Tempo final	Tempo de execução
1	Semiautomática	23/02/15	13:27:00	17:16:00	3:49:00
2	Semiautomática	26/02/15	17:52:00	20:31:00	2:39:00
3	Automática	02/06/15	14:38:00	16:37:00	1:59:00
4	Automática	11/08/15	9:45:00	11:25:00	1:40:00
5	Automática	21/08/15	13:05:00	14:02:00	0:57:00

Por fim, nas execuções do Produto 18, também verificamos que houve *dominação completa* do FREVO.

A Tabela 3.19 apresenta um resumo das 100 (cem) execuções capturadas no experimento.

Tabela 4.19: Resumo das execuções capturadas no experimento.

Tipo de execução	Quantidade de execuções	Tempo total gasto (horas)
Semiautomática	37	165:13:00
Automática	63	142:07:00

No experimento, averiguamos que o tempo gasto para executar 37 suítes semiautomáticas foi de 165:13:00, enquanto que o tempo gasto para executar 63 suítes automáticas através da ferramenta foi de 142:07:00.

Durante o experimento, constatamos que o tempo máximo da execução automática através da ferramenta apresentou-se menor que o tempo mínimo da execução semiautomática em 11 dos 18 produtos testados. Isto é, a dominação completa do FREVO ocorreu em 61% dos produtos testados.

4.3 CONSIDERAÇÕES FINAIS

Foi feita uma apresentação do uso do *framework* e ferramenta FREVO no ambiente do CIn-Motorola de modo que os componentes foram bem aceitos e rapidamente adotados em alguns projetos. Atualmente, o *framework* e a ferramenta FREVO é parte essencial no processo de desenvolvimento de *scripts*, bem como no processo de gerenciamento de criação e/ou execução de suítes de testes. A ferramenta demonstrou um bom desempenho em 11 dos 18 produtos já testados, apesar de uma confirmação estatística deste fato ainda ser necessária.

5 TRABALHOS RELACIONADOS

Com o avanço das tecnologias utilizadas no desenvolvimento das aplicações móveis, faz-se necessário que os *frameworks* de teste acompanhem essa evolução para que seja possível testar os novos componentes e características adicionadas e assim realizar diferentes tipos de validações nos produtos contribuindo para a qualidade do software.

Este capítulo descreve brevemente alguns dos principais *frameworks* de testes para dispositivos móveis do sistema Android disponíveis no mercado, listando suas principais características, semelhanças e/ou diferenças com o *framework* proposto e trabalhos acadêmicos relacionados ao desenvolvimento de *frameworks* para teste.

5.1 INDÚSTRIA

5.1.1 UI Automator

O *framework* UI Automator (ANDROID, 2016) foi a base para a implementação das propostas deste trabalho, visto que ele é um *framework* padrão de teste de caixa-preta de Android além ser o *framework* adotado no contexto do projeto CIn-Motorola. Sabemos que o UI Automator é um *framework* para criação de testes de interface gráfica baseado no JUnit 3. Ele fornece um conjunto de APIs que simulam as ações de um usuário. É um *framework* adequado para escrever testes caixa-preta, onde o código de teste não depende de detalhes internos de implementação de uma aplicação.

A Seção 2.10.1 no Capítulo 2 aborda todas as características desse *framework* em mais detalhes.

5.1.2 Robotium

Segundo (SHAO, 2015), Robotium era o *framework* de testes Android mais usado nos primeiros dias de mundo Android. Ele é bastante semelhante ao Selenium, mas sendo para Android e fazendo testes de API simples. Robotium é uma biblioteca de código aberto que estende JUnit com grande parte dos métodos úteis para testes de interface do usuário Android. Ele fornece casos de teste automáticos de caixa-preta poderosos e robustos para aplicativos Android.

Segundo a documentação oficial do Robotium (ROBOTIUM, 2015), este é um dos *frameworks* de automação de teste do Android que tem suporte completo para aplicações nativas e híbridas. Os desenvolvedores de casos de teste podem usar o Robotium para

escreverem funções, sistemas e cenários de teste de aceitação do usuário, abrangendo várias atividades do Android.

O Robotium possui uma ferramenta chamada Testdroid Recorder para a criação de *scripts* de teste. Através desta aplicação, é possível realizar ações reais no dispositivo real, gravando cada passo ou ação convertendo-a então em um código Javascript, podendo ser modificado manualmente depois.

5.1.3 Appium

Segundo (SHAO, 2015), Appium é um *framework* (e ferramenta) de automação de teste para aplicações nativas, híbridas e aplicativos web para os sistemas operacionais iOS e Android.

Appium usa o protocolo JSONWireProtocol internamente para interagir com aplicativos iOS e Android usando o WebDriver do Selenium. O suporte à Android ocorre através dos *frameworks* UI Automator (a partir do nível de API 16 ou superior) e Selendroid (SELENDROID, 2016) (nível de API inferior à 16). Com relação ao suporte para o iOS, este ocorre via UI Automation. Por fim, o suporte à aplicativos web ocorre por meio do Selenium driver.

(SHAO, 2015) complementa que uma das maiores vantagens de Appium é a possibilidade de escrever *scripts* Appium em quase qualquer linguagem de programação. Por exemplo, Java (DEITEL, 2000), Objective-C (KOCHAN, 2011), Javascript (FLANAGAM, 2006), PHP (GERKEN, 2000), Python (LUTZ, 1996), C# (LIBERTY, 2005), entre outras). Appium aborda também a questão de liberdade na escolha das ferramentas, compatibilidade entre plataformas (Android e iOS), liberdade de ter instalação e configuração rápida de dispositivos para realizar testes. Além disso, usuários familiarizados com a tecnologia Selenium sentem-se mais confortáveis em usar Appium em testes de aplicativo móvel, uma vez que eles usam o mesmo WebDriver e as características de teste são usadas da mesma maneira.

A Tabela 5.1 faz uma comparação entre os *frameworks* de automação descritos no trabalho, abordando diversas características tais como suporte a testes de componentes web, linguagem de programação utilizada, suporte para reexecução de casos de testes, entre outras.

Tabela 5.1: Tabela comparativa entre *frameworks* de testes suportados no sistema operacional Android.

Framework / **Características**	**Robotium**	**Appium**	**UI Automator**	**Frevo**
Suporte à Android	SIM	SIM	SIM	SIM
Suporte web	SIM	SIM	NÃO	NÃO
Linguagem de Programação	JAVA	DIVERSAS LINGUAGENS	JAVA	JAVA
Ferramenta de Suporte para criação de Testes	TESTDROID RECORDER	APPIUM.APP	UI AUTOMATOR VIEWER	UI AUTOMATOR VIEWER
Níveis de API suportadas	Todas	Todas	A partir da API 16	A partir da API 16
Reexecução de casos de teste (*retry*)	NÃO	NÃO	NÃO	SIM
Tempo limite de execução (*timeout*)	NÃO	NÃO	NÃO	SIM
Detalhes dos resultados do Teste (banco de dados, *screenshots*)	NÃO	NÃO	NÃO	SIM
GUI para criação, configuração e execução de uma suíte	NÃO	NÃO	NÃO	SIM
Base de dados de suítes e execuções realizadas, incluindo os resultados	NÃO	NÃO	NÃO	SIM

No restante deste capítulo, descreveremos trabalhos acadêmicos relacionados ao desenvolvimento de *frameworks* para teste. Devido à natureza destes trabalhos em comparação aos *frameworks* já descritos, faremos uma avaliação separada, pois eles estão relacionados a este projeto em um nível mais superficial.

5.2 ACADEMIA

5.2.1 NTAF

KIM (2009) propôs a criação do NTAF (*NHN Testing Automation Framework*), um *framework* concebido para ser utilizado como uma estrutura extensível através de uma poderosa mistura de várias ferramentas. Tirando proveito de diversas características de plataformas existentes, o NTAF visa prover as seguintes facilidades:

- Suporte à automação de sistemas baseados em web;
- Suporte à distribuição de ambientes de testes;
- Suporte à interface gráfica web e execução concorrente.

O fluxo de execução de testes e os ambientes de teste são declarados como tabelas de dados de entrada e os dados de saída são esperados em ambientes distribuídos. Por exemplo, através do NTAF é possível aplicar novas *builds* de um sistema automaticamente, instalá-las

em máquinas remotas e executar os testes remotamente. Para prover tais habilidades, o NTAF define uma simples e poderosa sintaxe que controla o fluxo de execução através de estruturas de controle. Essas estruturas de controle permitem que o *framework* lide com a lógica dos testes e com o fluxo de execução.

Segundo KIM (2009), múltiplos servidores com clientes podem ser configurados e executados, isto é, vários testes podem ser executados ao mesmo tempo.

Além das características descritas anteriormente, o NTAF possui um serviço de rastreamento de falhas interligado com uma interface para visualização de resultados. Isto é, durante a execução de uma suíte de testes, quando erros ou exceções acontecem, o serviço registra a falha automaticamente e, no final da execução, uma página de resultados é apresentada ao usuário. O NTAF provê também métricas de qualidade relacionadas às execuções.

5.2.2 Um framework multi-plataforma

Atualmente, a maioria das aplicações móveis é desenvolvida para diversas plataformas. Essas aplicações têm sido desenvolvidas de forma independente para cada um dos serviços, como, por exemplo, a aplicação *WhatsApp*, que é desenvolvida em três plataformas: Android, iPhone e Windows Mobile. Como automação de testes tem ganhado bastante atenção, cada plataforma móvel oferece seu próprio *framework* de testes. Dessa maneira, para testar o serviço *WhatsApp*, três aplicações são testadas independentemente para cada uma das plataformas.

SONG (2011) realizou uma pesquisa com o objetivo de desenvolver um *framework* de testes para integrar as plataformas Android e iPhone, de modo que o desenvolvimento de um único *script* já seria suficiente para testar um serviço nas duas plataformas. A base para a construção desse *framework* foram as ferramentas de código aberto disponíveis na comunidade de testes.

Antes de iniciar o desenvolvimento do *framework*, SONG (2011) deu ênfase a três grandes questões:

- Questões específicas da plataforma;
- Questões de gerenciamento de testes;
- Questões de ambientes de testes.

Ao avaliar estas questões, levando em consideração a especificidade de cada plataforma, SONG (2011) definiu o *framework* (ferramentas auxiliares referentes às plataformas Android e iPhone, arquitetura e ambiente de testes).

Na escolha das ferramentas auxiliares, a ferramenta FoneMonkey foi escolhida para a plataforma iPhone, e a ferramenta Robotium (Seção 5.2) foi escolhida para a plataforma Android.

Com relação à arquitetura, tendo em vista que as aplicações são dependentes de plataforma, SONG (2011) construiu uma camada para remover as dependências de plataforma por meio da criação de funções advindas do mapeamento das funções equivalentes entre os *frameworks* auxiliares (FoneMonkey e Robotium). Dessa maneira, foi estabelecida uma tabela estruturada com funções independentes que habilita a unificação na criação de *scripts* de teste.

Segundo SONG (2011), no momento de implementar um novo *script*, não importa qual a plataforma auxiliar está sendo usada, o processo de automação de teste pode ser dividido em três etapas:

- Criação do caso de teste baseado nas funções do *framework* proposto;
- Execução do caso de teste;
- Visualização de resultados.

Para o ambiente de testes, o *framework* proposto usou o *framework* NTAF (Seção 5.4), herdando todas as características de execução desse *framework*, podendo iniciar a execução dos testes em diversos dispositivos (capacidade de distribuição) simultaneamente (capacidade de concorrência) e visualizar os resultados (capacidade de apresentação de resultados).

No entanto, o *framework* não foi testado para a plataforma Android, somente a plataforma iPhone foi testada. Em adição, o *framework* não conseguiu mensurar precisamente o tempo de execução dos testes.

5.2.3 Um framework dirigido ao modelo chave-valor

Até então, os *frameworks* apresentados neste trabalho requerem que os times responsáveis pela automação de testes tenham um nível de baixo a médio de conhecimentos de programação. AMARICAI (2014) propõe o desenvolvimento de um *framework* de automação de testes voltado para times de testes que tem um nível baixo (ou nenhum) de conhecimentos de programação. Partindo desse princípio, o *framework* considera as seguintes opções:

- Manter a lógica e o fluxo dos casos de testes no código do *framework* e usar arquivos externos para representar os *locators* (identificadores que representam componentes gráficos) da tela e os dados de entrada do teste;

- Ou ser mais ousado e usar um alto nível de generalização no código para manter os *locators*, os dados do teste, a lógica dos testes e o fluxo dos testes em arquivos externos.

Segundo AMARACAI (2014), as abordagens apresentadas acima exigem um alto nível de conhecimento de programação por parte das pessoas responsáveis pelo desenvolvimento do *framework*. Em contrapartida, as pessoas envolvidas na atividade de criação e manutenção de *scripts* precisam ter poucos conhecimentos de programação.

Dando continuidade as etapas de construção do *framework*, AMARACAI (2014) apresenta os três tipos de arquivos necessários:

- *Locators*: arquivo que contém os identificadores dos componentes gráficos da tela de uma aplicação;
- *Data*: arquivo que contém as seções de dados dos casos de teste. Nesse arquivo, incluem-se os dados de entrada e os *asserts*. Cada seção tem a ordem específica da aplicação – as seções são separadas visando o reuso em várias partes do teste;
- *Test*: arquivo que armazena os testes, onde cada teste contém o fluxo de ações e as chamadas para as seções de dados.

O *framework* considera basicamente esses três arquivos, independentemente do tamanho da aplicação a ser testada. Diferentemente dos *frameworks* citados acima, a solução proposta apresenta-se como uma abordagem dirigida a chave-valor, isto é, os arquivos externos armazenam os casos de teste, os dados de entrada e os identificadores da aplicação como entidades do sistema.

Tudo que uma equipe de testes precisa fazer para criar um novo teste é definir uma ou mais seções de dados (ou usar as seções de dados existentes) e adicionar uma nova seção no arquivo de testes. Da mesma forma, se os identificadores ainda não estão definidos, basta criar uma seção adicionando os elementos no arquivo de identificadores. Ao solicitar a execução de um teste, a relação complexa entre cada um dos três arquivos fica a cargo do *framework*.

5.3 CONSIDERAÇÕES FINAIS

Nesse capítulo, apresentamos os principais *frameworks* de testes existentes na indústria e alguns trabalhos acadêmicos com propostas de *frameworks* de teste. O estudo desses trabalhos foi fundamental para a consolidação de um trabalho acadêmico com características essenciais voltadas para a indústria de automação de testes.

6 CONCLUSÕES

Com o rápido crescimento do mercado de dispositivos móveis, a automação de testes é uma tendência na área de testes de software, sendo vista como umas das principais medidas para acompanhar essa evolução. Quando executada corretamente, é uma das melhores formas de reduzir o tempo de teste no ciclo de vida do software, diminuindo o custo e aumentando a produtividade das atividades de teste, além de, consequentemente, garantir um ponto fundamental para o sucesso do produto, a qualidade.

A automação de teste consiste em repassar para um computador tarefas de teste que seriam realizadas por um ser humano. Geralmente, esse processo ocorre por meio do uso de *frameworks* de automação de testes. Na realização do trabalho, observamos que os *frameworks* de automação de teste proveem facilidades de programação de testes automáticos isolados. No entanto, pudemos identificar que apresentam diversas limitações referentes à execução de suítes de testes e visualização de resultados (limitados a passar e falhar um teste, sem maior riqueza de detalhes que possam ajudar a depurar o problema).

Neste trabalho, propusemos uma extensão de facilidades dos *frameworks* de automação de testes para o nível de automação de suítes, materializados em uma implementação baseada no UI Automator. A estratégia apresentada nesta dissertação é baseada na separação, de maneira coesa, das atividades de implementação de *scripts* das atividades relativas ao gerenciamento e execução de testes. Esta estratégia requer a construção desses dois componentes integrados: o primeiro é um *framework* de desenvolvimento de *scripts* que essencialmente adiciona novas características ao *framework* UI Automator e define um padrão na escrita de testes automáticos. Estas novas características podem ser estendidas em qualquer outro *framework* (não são particulares ao UI Automator) e englobam facilidades de automação no nível de suítes, como *timeout*, reexecução e base de dados de execuções de suítes. O segundo componente é uma ferramenta interativa criada para definir um processo de execução maduro e consistente, seguido de uma visualização de resultados de casos de teste intuitiva e usual por meio de uma GUI. Tipicamente os *frameworks* produzem resultados baseados em julgamentos booleanos de passou ou falhou. Nossa proposta inclui facilidades de resultados mais ricos onde, por exemplo, a *screenshot* da tela de erro é armazenada. Isto facilita o trabalho posterior do programador na depuração do código.

Uma importante vantagem do uso da abordagem proposta pela pesquisa em substituição à utilização do processo tradicional definido no UI Automator diz respeito à

melhoria na qualidade (evitar a introdução de erros humanos) e produtividade (redução de esforço) nas etapas de criação, edição e execução de suítes de testes automáticos.

Portanto, as principais contribuições do presente trabalho foram: a construção de dois projetos de software, um *framework* e uma ferramenta que estendem as facilidades de automação de testes para o nível de suítes, e a aplicação deles em um ambiente real de produção. O nosso objetivo inicial foi propor uma abordagem capaz de eliminar os passos comuns e repetitivos durante as atividades de testes de modo a viabilizar a otimização do processo de criação, edição e execução de suítes de testes automáticos. Por fim, constatamos através de um estudo de caso, que a solução FREVO alcançou ganhos de produtividade (*dominação completa*) oriundos da sua adoção na execução dos testes automáticos da ordem de 60% dos produtos testados.

Com relação aos trabalhos futuros na ferramenta e *framework* FREVO, identificamos quatro possibilidades de expansão: a primeira seria aumentar a capacidade do *framework* no que diz respeito à automação de testes de voz sobre dispositivos móveis, isto é, prover um conjunto de classes capazes de validar funcionalidades que são realizadas por meio de comandos de voz. Como a tecnologia de comandos de voz sob dispositivos móveis é recente, os *frameworks* de automação ainda são carentes nesse contexto; a segunda seria a construção de uma nova camada no *framework* FREVO, responsável por controlar a *execução* de testes automáticos desenvolvidos em Python; a terceira seria a construção de um *parser* responsável por converter os *scripts* nativos do UI Automator em *scripts* FREVO, eliminando assim, a etapa de adaptação manual dos *scripts* de um projeto UI Automator existente; por fim, executaremos um experimento controlado para analisarmos com precisão o ganho real que FREVO oferece em comparação com o processo semiautomático.

REFERÊNCIAS

(AMARACAI, 2014) AMARICAI, Sabina et al. Designing a Software Test Automation Framework. Informatica Economica, v. 18, n. 1, p. 152-161, 2014.

(ANDROID, 2016) Android Testing Tools. Disponível em: <http://developer.android.com/tools/testing/testing-tools.html>. Acesso em: 19/01/2016.

(BRAY, 1998) BRAY, Tim et al. Extensible markup language (XML). World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210, v. 16, 1998.

(BURNSTEIN, 2003) BURNSTEIN, Ilene. Practical Software Testing: A Process-oriented Approach. Springer-Verlag. Nova York.

(CLARK, 2005) CLARK, M. "JUnit Primer". Web Journal, 2005.

(COPELAND, 2004) COPELAND, Lee. A Practitioner's Guide to Software Test Design. Artech House.

(CPPUNIT, 2016) CppUnit - C++ port of JUnit. Disponível em: <http://sourceforge.net/projects/cppunit/>. Acesso em: 19/01/2016.

(CRAIG, 2002) CRAIG, R.D., JASKIEL, S. P., Systematic Software Testing, Artech House Publishers, Boston, 2002.

(CRESPO, 2004) CRESPO, A. N., SILVA, O. J., BORGES, C. A., SALVIANO, C. F., TEIVE, M., JUNIOR, A., & JINO, M. Uma metodologia para teste de Software no Contexto da Melhoria de Processo. Simpósio Brasileiro de Qualidade de Software, 271-285, 2004.

(DEITEL, 2000) DEITEL, H. M., DEITEL, P. J. "Java: how to program". Third Edition, Prentice Hall, 2000.

(DUSTIN, 2003) DUSTIN, Elfriede. Effective Software Testing: 50 Specific Ways to Improve Your Testing. Addison-Wesley Professional; 1st edition, 2002.

(ECLIPSE, 2016) Using JUnit in Eclipse. Disponível em: <https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml>. Acesso em: 19/01/2016.

(ESPRESSO, 2016) ESPRESSO. Espresso. 2016. Disponível em: https://google.github.io/android-testing-support-library/docs/espresso/index.html. Acesso em: 22/01/2016.

(FERREIRA, 1986) FERREIRA, A. B. H. Novo dicionário Aurélio da língua portuguesa. Ed. Nova Fronteira, 1986.

(FEWSTER et al., 1999) FEWSTER, Mark, GRAHAM, Dorothy. Software Test Automation. Addison-Wesley Professional; 1st edition, 1999.

(FLANAGAM, 2006) FLANAGAN, David. JavaScript: the definitive guide. " O'Reilly Media, Inc.", 2006.

(GERKEN, 2000) GERKEN, Till; RATSCHILLER, Tobias. Web Application Development with PHP. New Riders Publishing, 2000.

(HARROLD, 2000) HARROLD. M. J. Testing: a roadmap. In: 22nd International Conference on Software Engineering. Junho, 2000.

(HETZEL, 1988) HETZEL, W. The complete guide to software testing. 2nd edition. QED Info Science Inc., 1988.

(HEUSER, 2008) HEUSER, C. A. Projeto de banco de dados. 6. ed. Porto Alegre: Bookman, 2008. 282p. (Série Livros Didáticos Informática UFRGS, v. 4).

(IEEE 610, 1990) IEEE Standard 610-1990: IEEE Standard Glossary of Software Engineering Terminology, IEEE Press.

(INTELLIJ, 2016) Testing Frameworks. Disponível em: <https://www.jetbrains.com/idea/help/testing-frameworks.html>. Acesso em: 19/01/2016.

(JUNIT, 2016) Junit development team. JUnit. Disponível em: <http://www.junit.org>. Acesso em: 19/01/2016.

(KANER, 2001) KANER, C, JAMES B, BRET P. Lessons Learned in Software Testing: A Context-Driven Appr. Wiley; 2001.

(KELLY, 2006) KELLY, Michael. Introduction to IBM Rational Functional Tester 7.0. Disponível em: <http://www.ibm.com/developerworks/rational/library/06/1205_kelly>. Acesso em: 22/01/2016.

(KIM, 2009) KIM, Eun Ha; NA, Jong Chae; RYOO, Seok Moon. Implementing an effective test automation framework. In: Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International. IEEE, 2009. p. 534-538.

(KOCHAN, 2011) KOCHAN, S. G. “Programming in Objective-C”. Third Edition, Addison-Wesley Professional, 2011.

(LIBERTY, 2005) LIBERTY, Jesse. Programming C#: Building. NET Applications with C#. " O'Reilly Media, Inc.", 2005.

(LUTZ, 1996) LUTZ, Mark. Programming python. O'Reilly, 1996.

(MALDONADO, 2001) MALDONADO, J. C., FABBRI, S.C.P.F., “Teste de Software”. In: Qualidade de Software: Teoria e Prática, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.

(MASSOL, 2003) MASSOL, V., HUSTED, T., “JUnit in Action”, Manning Publications Co., Greenwich, CT, 2003.

(MYERS, 1979) MYERS, G. The art of software testing. John Wiley and Sons, 1979.

(MYERS, 2004) MYERS, Glenford J. The Art of Software Testing. New York: John Wiley & Sons, Second Edition, 2004.

(NETBEANS, 2016) Writing JUnit Tests in NetBeans IDE. Disponível em: <https://netbeans.org/kb/docs/java/junit-intro.html>. Acesso em: 19/01/2016.

(NUNIT, 2016) NUnit. Disponível em: <http://www.nunit.org/>. Acesso em: 19/01/2016.

(PFLEEGER, 2004) PFLEEGER, S. L. Engenharia de Software: Teoria e Prática, Prentice Hall do Brasil, 2ª Edição, 2004.

(PRESSMAN, 2001) PRESSMAN, R. S. Software engineering - a practitioner’s approach. 5th. ed. McGraw-Hill, 2001.

(PRESSMAN, 2002) PRESSMAN, R. S. Engenharia de software. 5. Ed. São Paulo: McGraw-Hill, 2002. 843 p.: ISBN 9788586804250.

(PRESSMAN, 2004) Roger S. Pressman. Software Engineering: A Practitioner’s Approach. McGraw Hill, 2004.

(QUENTIN, 1999) QUENTIN, G. Automated Software Testing: Introduction, Management and Performance. Softw. Test., Verif. Reliab., 9(4):283–284, 1999. Elfriede Dustin, Jeff Rashka and John Paul, Addison-Wesley, 1999 (Book Review).

(RAINSBERGER, 2005) RAINSBERGER, J.B. and STIRLING, Scott. JUnit Recipes: Practical Methods for Programmer Testing. Manning Publications Co.

(ROBOTIUM, 2016) ROBOTIUM. Robotium. 2016. Disponível em: https://github.com/robotiumtech/robotium. Acesso em: 22/01/2016

(SELENDROID, 2016) SELENDROID. Selendroid. 2016. Disponível em: http://selendroid.io/setup.html. Acesso em: 25/01/2016.

(SELENIUM, 2016) SELENIUM. Selenium. 2016. Disponível em: http://www.seleniumhq.org/docs/. Acesso em: 25/01/2016.

(SHAO, 2016) SHAO, Lingkai. Top 5 Android Testing Frameworks (with Examples). 2016. Disponível em: http://testdroid.com/tech/top-5-android-testing-frameworks-with-examples. Acesso em: 22/01/2016.

(SONG, 2011) SONG, Hyungkeun; RYOO, Seokmoon; KIM, Jin Hyung. An integrated test automation framework for testing on heterogeneous mobile platforms. In: 2011

First ACIS International Symposium on Software and Network Engineering (SSNE). IEEE, 2011. p. 141-145.

(TIAN, 2005) TIAN, Jeff. Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement. John Wiley & Sons.

(UNITTEST, 2016) unittest - Unit testing framework. Disponível em: <https://docs.python.org/2/library/unittest.html>. Acesso em: 19/01/2016.