UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

KENYO ABADIO CROSARA FARIA

Uma solução baseada em economia colaborativa para escalar o teste de aplicações Android em dispositivos reais

Goiânia
2019

sistema de bibliotecas ufg

PRPG
PRÓ-REITORIA DE PÓS-GRADUAÇÃO

UFG

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[ ] Dissertação** **[X] Tese**

2. Identificação da Tese ou Dissertação:

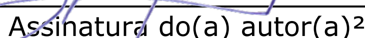
Nome completo do autor: Kenyo Abadio Crosara Faria

Título do trabalho: **Uma solução baseada em economia colaborativa para escalar o teste de aplicações Android em dispositivos reais**

3. Informações de acesso ao documento:

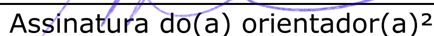
Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.



Assinatura do(a) autor(a)[2]

Ciente e de acordo:



Assinatura do(a) orientador(a)[2]

Data: 27 / 04 / 2019

[1]Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

Casos de embargo:
- Solicitação de registro de patente
- Submissão de artigo em revista científica
- Publicação como capítulo de livro
- Publicação da dissertação/tese em livro

[2]A assinatura deve ser escaneada.

KENYO ABADIO CROSARA FARIA

Uma solução baseada em economia colaborativa para escalar o teste de aplicações Android em dispositivos reais

Trabalho apresentado ao Programa de Pós–Graduação em Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação.

Orientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi

Goiânia
2019

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Abadio Crosara Faria, Kenyo
Uma solucao baseada em economia colaborativa para escalar o teste de aplicacoes Android em dispositivos reais [manuscrito] / Kenyo Abadio Crosara Faria. - 2019.
CVII, 107 f.: il.

Orientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi.
Tese (Doutorado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação em rede (UFG/UFMS), Goiânia, 2019.
Bibliografia.
Inclui siglas, gráfico, tabelas, algoritmos, lista de figuras, lista de tabelas.

1. teste de software. 2. economia colaborativa. 3. teste ui. I. Marcelo Rizzo Vincenzi, Auri, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

Ata de Defesa de Tese de Doutorado

Aos dois dias do mês de abril de dois mil e dezenove, no horário das catorze horas, foi realizada, nas dependências do Instituto de Informática da UFG, a defesa pública da Tese de Doutorado do aluno Kenyo Abadio Crosara Faria, matrícula no. 2014100103, intitulada **"Uma solução baseada em economia colaborativa para escalar o teste de aplicações Android em dispositivos reais"**.

A Banca Examinadora, constituída pelos professores:

Prof. Dr. Auri Marcelo Rizzo Vincenzi – DC/UFSCar - orientador

Prof. Dr. Plínio de Sá Leitão Júnior – INF/UFG

Prof. Dr. Fabrízzio Alphonsus Alves de Melo Nunes Soares – INF/UFG

Prof. Dr. José Carlos Maldonado – ICMC/USP

Prof. Dr. Eduardo Noronha de Andrade Freitas - IFG

emitiu o resultado:

(X) Aprovado

( ) Aprovado com revisão

(A Banca Examinadora deve definir as exigências a serem cumpridas pelo aluno na revisão, ficando o orientador responsável pela verificação do cumprimento das mesmas.)

( ) Reprovado

com o seguinte parecer: ______________________________

____

____

____

Prof. Dr. Auri Marcelo Rizzo Vincenzi

Prof. Dr. Plínio de Sá Leitão Júnior

Prof. Dr. Fabrízzio Alphonsus Alves de Melo Nunes Soares

Prof. Dr. José Carlos Maldonado – ICMC/USP

Prof. Dr. Eduardo Noronha de Andrade Freitas

All rights reserved. The total or partial reproduction of this work is prohibited without permission from the university, author, and advisor.

Kenyo Abadio Crosara Faria

Kenyo Abadio Crosara Faria recebeu seu diploma de graduação em Ciência da Computação pela Universidade Católica de Goiás em 2004; Seu título de Mestre em Engenharia Elétrica e de Computação pela UFG em 2006, e seu Título de Doutor em 2019 pelo Instituto de Informática da UFG. De 2004 até 2008 atuou como diretor da empresa Conceito Informação e Tecnologias Associadas, trabalhando com projetos corporativos. De 2008 até 2011 atuou como pesquisador em projetos contratados pela Eletronorte e executados pela Escola de Engenharia Elétrica e de Computação da UFG. Desde 2008 atua como Professor/Pesquisador pelo Instituto Federal de Goiás. Entre os anos de 2012 e 2014, atuou como Líder Técnico de um projeto de inovação pela empresa Oobj. Sua trajetória lhe deu experiência em desenvolvimento de software utilizando tecnologias diversas.

Para meus pais, Alaor e Maria de Fátima, por tudo que me ensinaram enquanto vivi com eles. Para minha esposa, Andriana, que sempre me apoiou, e meus filhos Vítor e Helena, razão maior da minha vida.

Agradecimentos

Gostaria de agradecer ao meu orientador, Prof. Dr. Auri pelo seu companheirismo, paciência e dedicação demonstrados durante a realização desta tese. Tenho certeza que fiz mais um amigo.

Obrigado ao Prof. Dr. José Carlos Maldonado pelos ensinamentos e disposição em ajudar sempre que acionado.

Obrigado ao time de professores e demais funcionários do programa de Doutorado do Instituto de Informática da UFG e também aos meus colegas do programa.

Gostaria de agradecer a todos os meus amigos, que contribuíram em alguma medida para realização deste trabalho. Ao Sr. Marcos Ventura, minha influência para o estudo da Ciência da Computação e inspiração para várias das coisas que aprendi. Prof. Dr. Gelson Júnior que me influenciou para experimentar da pesquisa, obrigado por sempre tentar me ajudar quando o procurei. Prof. Dr. Cássio Vinhal que me orientou durante o Mestrado, sempre presente na minha vida. Obrigado aos meus grandes amigos Prof. Dr. Eduardo Freitas e Jonathas Carrijo por vossa participação e investimento na vida da minha família.

Meu especial reconhecimento à minha maior conquista, minha família. Minha esposa Adriana, 14 anos de companheirismo e paciência, sempre me apoiando na carreira. Aos meus filhos Vítor e Helena, sempre aprendo muito com eles.

Meu especial agradecimento aos meus pais. Meus heróis de todos os momentos. Obrigado pelo amor e dedicação empregados na minha vida. Obrigado a minha Mãe pelas orações e por tentar me ajudar até hoje, e ao meu Pai, pelo companheirismo e por sempre me encorajar a executar meus projetos. Jamais teria conseguido chegar até aqui sem os esforços de vocês, sou eternamente grato.

Finalmente, obrigado a Deus por me permitir viver esta experiência e ter o privilégio de finalizar este trabalho. Espero poder honrar esta oportunidade, empregando da melhor forma tudo que aprendi, a fim de ajudar ao próximo para tua Glória. Lhe dedico não apenas este trabalho, mas a minha vida.

“Quando um homem melhora, torna-se cada vez mais capaz de perceber o mal que ainda existe dentro de si. Quando um homem piora, torna-se cada vez menos capaz de captar a própria maldade..”

C. S. Lewis,
Cristianismo puro e simples.

Resumo

Faria, Kenyo Abadio Crosara Faria. **Uma solução baseada em economia colaborativa para escalar o teste de aplicações Android em dispositivos reais**. Goiânia, 2019. 104p. Tese de Doutorado, Instituto de Informática, Universidade Federal de Goiás.

Os testes de softwares para dispositivos móveis apresentam desafios adicionais se comparado aos testes de aplicações desktop e web, especialmente em ambientes fragmentados como é o caso do ecossistema Android. Atualmente são mais de 24 mil diferentes modelos de dispositivos, com diferentes tamanhos e densidades de tela, versão de sistema operacional e outras configurações que contribuem para a instabilidade de aplicativos que endereçam este ecossistema. Vários *frameworks* e plataformas de *TaaS* abordam a validação destes aplicativos, especialmente no que se refere à construção e execução de testes de interface gráfica. No entanto, limitações arquiteturais existentes, resultam em alto custo e baixa diversidade de dispositivos reais a serem utilizados durante a validação de *apps* Android. Inspirada neste contexto e no paradigma da Economia Colaborativa, esta tese propõe uma arquitetura disruptiva, que permite a execução de testes de aplicativos *Android* de forma distribuída, com o uso de dispositivos ociosos ao redor do mundo, reduzindo o custo de infraestrutura com dispositivos reais e com potencial de geração de um novo mercado. Experimentos demonstraram a robustez da arquitetura quanto à execução de testes UI em dispositivos geograficamente distribuídos, e uma análise financeira indica uma redução de 85.67% no custo com infraestrutura de dispositivos físicos alocados para testes, ao mesmo tempo que mostra a viabilidade de funcionamento da plataforma construída.

Palavras–chave

teste de software, economia colaborativa, teste UI

Abstract

Faria, Kenyo Abadio Crosara Faria. **DBB:DBB: An Approach Based on Collaborative Economy Applied on Android Application Tests Aiming at Reducing Rosts.**. Goiânia, 2019. 104p. PhD. Thesis
Instituto de Informática, Universidade Federal de Goiás.

Software testing for mobile devices presents additional challenges compared to testing desktop and web applications, especially in fragmented environments such as the Android ecosystem. There are currently over 24 thousands different device models, with different screen sizes and densities, operating system versions, and other configurations that contribute to the instability of applications for this ecosystem. Several frameworks and TaaS platforms assist the validation of these type of applications, especially with regard to the construction and execution of UI tests. However, existing architectural limitations result in high-cost and low-diversity real devices to be used during Android apps validation. Inspired in this context and in the Collaborative Economy paradigm, this study proposes a disruptive architecture, which allows the execution of applications tests in an distributed way, using idle devices around the world, reducing the cost of infrastructure with real devices at the same time with potential of generating a new market. Experiments have demonstrated the robustness of the architecture for performing UI tests on geographically distributed devices, and a financial analysis indicates a reduction of 85.67% in the infrastructure cost of physical devices allocated for testing, while showing the viability of the built platform.

Keywords

software testing, collaborative economy, UI testing

Conteúdo

Lista de Figuras **11**

Lista de Tabelas **12**

1 Introdução **13**
- 1.1 Motivação 15
- 1.2 Objetivos 20
- 1.3 Metodologia 20
- 1.4 Publicações 21
- 1.5 Organização da Tese 22

2 Fundamentação Teórica **23**
- 2.1 Testes de Software 23
  - 2.1.1 Motivações e limitações dos testes 23
  - 2.1.2 Conceitos chave e taxonomia 25
- 2.2 Tipos de Aplicativos 27
  - 2.2.1 Aplicações Nativas 28
  - 2.2.2 Aplicações Web 31
  - 2.2.3 Aplicações Híbridas 33
- 2.3 Testes de apps Android 33
  - 2.3.1 Testes usando o Espresso 38
  - 2.3.2 Provedores de dispositivos em nuvem 40
- 2.4 Economia Colaborativa 42
  - 2.4.1 Economia Colaborativa Aplicada ao Teste de Software 44
- 2.5 Considerações Finais 46

3 DBB: Uma proposta de solução para execução de testes UI distribuídos **47**
- 3.1 Compilação dos testes 48
- 3.2 Execução distribuída dos testes 50
  - 3.2.1 Registro de dispositivos na plataforma 52
  - 3.2.2 Distribuição dos testes 53
- 3.3 Relatório dos testes executados 55
- 3.4 Considerações Finais 57

4 Experimentos e Resultados **58**
- 4.1 Detalhamento dos experimentos 58
  - 4.1.1 Experimento A 59
  - 4.1.2 Experimento B 63
- 4.2 Resultados 64

4.3 Considerações Finais 65

5 Análise da plataforma proposta **66**
5.1 Análise comparativa da escalabilidade e cobertura 66
5.2 Avaliação do custo 69
5.3 Demanda existente 77
5.4 Ameaças a Validade 78
5.5 Considerações Finais 80

6 Trabalhos Correlatos **81**
6.1 Gravação e execução de casos de teste UI 81
6.2 Análise estática e dinâmica 82
6.3 Geração de dados de entrada 83
6.4 Trabalhos que abordam o problema do oráculo 87
6.5 Trabalhos que abordam o custo por meio da priorização 87
6.6 Plataformas em nuvem 88
6.7 Considerações Finais 89

7 Conclusão **90**

Bibliografia **93**

Lista de Figuras

1.1 Fatia de mercado global: *Desktop vs smartphones vs tablets* (STATCOUNTER, 2018) 15
1.2 *Market Share* dos sistemas operacionais em dispositivos móveis (STATCOUNTER, 2018) 16
1.3 Número de fabricantes de dispositivos *Android* (OPENSIGNAL, 2015) 16
1.4 Mapa da fragmentação do ecossistema *Android* (OPENSIGNAL, 2015) 18

2.1 Pirâmide de Mike Cohn (COHN, 2010). 26
2.2 Esquema de compilação de aplicações nativas (GOOGLE, 2018b). 28
2.3 Interação de aplicações nativas com o SO. 29
2.4 APIs de alto nível e baixo nível do *Android*. 29
2.5 Estado inicial da aplicação 30
2.6 Estado final da aplicação após *scroll down* 31
2.7 Interação de uma *web app* com o dispositivo 32
2.8 Interação de uma app híbrida com o dispositivo. 34
2.9 Pirâmide de Mike Cohn aplicada a testes no *Android* (GOOGLE, 2018c) 34
2.10 Principais frameworks de testes Android. 38
2.11 Esquema de geração de `apks` de testes 38
2.12 Esquema de funcionamento do `adb` 39
2.13 Instalação de testes instrumentados pelo *Android SDK* 40

3.1 Visão geral da plataforma. 48
3.2 Principais componentes do Android Testing Support Library. 49
3.3 Componentes renderizados pelo Android, visualizados através do UiAutomator Viewer. 52
3.4 Distribuição dos testes submetidos a plataforma 54
3.5 Obtenção dos relatórios gerados pelos testes 56

5.1 Participação dos modelos chineses no mercado global (COUNTERPOINTER, 2018). 69
5.2 Detalhamento dos custos de um DH na DBB. 75
5.3 Comparação do custo da DBB com outras soluções. 76
5.4 Redução do custo no uso da DBB em comparação com outras soluções. 76
5.5 Total de *apps* disponíveis na Google Play Store (Statista, 2019) 77

Lista de Tabelas

1.1 Distribuição de artigos baseados na sua classificação 18
1.2 CPI por país (APPBRAIN, 2018) 19

2.1 Fatores Típicos de Qualidade de Software (HETZEL, 1988) 25
2.2 Aplicações Nativas vs Web apps 33
2.3 *Frameworks* para automatização de testes em *apps Android* (SAUCE-LABS, 2019). 37
2.4 Provedores de dispositivos móveis como IaaS 41

3.1 Dependências necessárias ao funcionamento do Espresso 49

4.1 Lista de aplicativos utilizados no experimento A 59
4.2 Casos de testes escritos para a *app Money Balance* 60
4.3 Casos de testes escritos para a *app QuitSmoking* 61
4.4 Casos de testes escritos para a *app Simple File Manager* 62
4.5 Dispositivos utilizados no experimento A 62
4.6 Aplicações utilizadas no experimento B 63
4.7 Dispositivos utilizados no experimento B 64

5.1 Escalabilidade dos players de dispositivos móveis em nuvem 67
5.2 Fatia de mercado (%) por fabricante (COUNTERPOINTER, 2018) 69
5.3 Fatia de mercado (milhões de unidades) por fabricante (COUNTER-POINTER, 2018) 69
5.4 Preços nos principais provedores de nuvem e limite de dispositivos-hora incluídos no preço. 70
5.5 TCO da infraestrutura necessário para um período de 3 anos. 72
5.6 Expectativa de receita a partir da demanda apresentada. 78

CAPÍTULO 1

Introdução

Vários aspectos são considerados em projetos de engenharia, além do escopo das entregas, prazos e controle de qualidade, o custo é quase sempre um fator determinante. No caso da Engenharia de Software, desde estudos passados (KEMERER, 1987) até estudos recentes (MISHRA; MAHANTY, 2016) o custo é sempre colocado como um fator crítico.

Algumas das atividades mais caras do processo de construção do software são os testes. De acordo com Takagi, Furukawa e Yamasaki (2007), cerca de 65% do tempo total de um projeto é consumido pelos processos de testes e validação. Vários trabalhos foram desenvolvidos com o intuito de minimizar o impacto financeiro dos testes no orçamento total de um produto de software, (WEYUKER; OSTRAND; BELL, 2004), (RAY; MOHAPATRA, 2012), (KIM et al., 2007), (OSTRAND; WEYUKER; BELL, 2005), (HASSAN; HOLT, 2005). No entanto, novos mercados e, consequentemente, novos problemas vão surgindo e desafiando ainda mais as equipes envolvidas na construção de softwares, especialmente aqueles que devem ser executados sob diferentes ambientes a todo tempo.

Tal é o impacto dos testes no custo do projeto, que alguns trabalhos têm apresentado diferentes metodologias de mensuração da viabilidade do emprego dos testes dependendo do tipo de aplicação construída (NIKOLIK, 2012), (BASU, 2015). Alguns trabalhos chegam a sugerir que se o custo da correção de um *bug* é menor que o custo para evitá-lo, os testes não são viáveis (EVERETT; JR, 2007), pois a maioria das decisões de projeto são baseadas na comparação do valor de se fazer algo versus o custo envolvido. Esta comparação é tipicamente chamada de Retorno sobre o Investimento (*ROI - Return on Investment*).

Um dos trabalhos pioneiros sobre o custo da correção de um *bug* foi publicado por Barry Boehm em 1976. Neste estudo, Boehm afirma que defeitos são mais caros para serem corrigidos proporcionalmente ao estágio da sua identificação. Boehm representou as fases sucessivas do ciclo de vida do desenvolvimento de software em cascata na escala horizontal, com uma linha diagonal afastando-se na escala vertical, o que corresponde ao custo relativo de corrigir um defeito detectado em uma determinada fase versus corrigindo

o mesmo defeito em uma fase diferente (BOEHM, 1976). As afirmações de Boehm foram aprimoradas ao longo dos anos, e uma segunda relação exponencial foi adicionada com uma inclinação menor designada para projetos de software menores (BOSSAVIT, 2015), (FOWLER, 2012).

Alguns trabalhos promovem uma reflexão acerca da relação custo/benefício da aplicação de testes em determinados cenários e apresentam modelos para otimização do esforço empregado nas atividades de testes e validação (YANG; CHAO, 1995; WEISS et al., 2007). Outros trabalhos abordam a questão do custo relacionado à automatização dos testes (RAMLER; WOLFMAIER, 2006), (BERNER; WEBER; KELLER, 2005). No livro intitulado *"How Google tests software"*, Whittaker afirma que testes automatizados são caros pra serem construídos e facilmente são depreciados com o tempo (WHITTAKER; ARBON; CAROLLO, 2012). No entanto, testar aplicações que endereçam ecossistemas fragmentados[1] de forma manual não é factível, uma vez que os testes necessitam ser executados considerando o maior número de combinações possíveis do ecossistema. Desta forma, o emprego dos testes automatizados deve contribuir para entrega de aplicações livres de *bugs*, minimizando prejuízos.

Além do custo envolvido na automatização dos testes, sua execução em um ecossistema fragmentado é um desafio, pois é necessário que os testes sejam executados em várias configurações possíveis, encarecendo o processo de testes. No caso do ecossistema Android, a execução de testes automatizados em dispositivos reais, exige investimentos na aquisição destes dispositivos, ou a sua locação em provedores de infraestrutura. Considerando a existência de um grande número de dispositivos Android diariamente ociosos em determinadas horas, e que os recursos existentes nestes dispositivos poderiam ser aproveitados para realização de testes de forma a recompensar seus proprietários, esta tese propõe uma arquitetura para execução distribuída de testes de interfaces gráficas (testes UI) realizados em aplicações móveis, viabilizada por meio de uma plataforma baseada no paradigma da Economia Colaborativa (INDEX.CO, 2017). Este paradigma vêm sendo utilizado em vários modelos de negócios dos mais diferentes tipos, desde o transporte e hospedagem de pessoas, até a entrega de mercadorias e locação de veículos. Detalhes acerca deste paradigma bem como sua aplicação na indústria atual são apresentados na seção 2.4.

Na próxima seção são apresentadas os fatores que nortearam a escolha da plataforma Android para o desenvolviemnto desta tese.

[1]ecossistemas fragmentados: São plataformas que permitem um grande número de configurações possíveis. O Android é um ecossistema composto por várias API's que executam em milhares de dispositivos com diferentes configurações

1.1 Motivação

No passado, softwares eram construídos para serem executados em ambientes específicos (sistemas operacionais e hardware específico), após a popularização da Internet e a variedade de navegadores disponíveis, as aplicações web passaram a ser validadas em diferentes navegadores. Mais recentemente, os *smartphones* assumiram uma fatia de mercado correspondente à metade do mercado global de dispositivos. A Figura 1.1 apresenta a fatia do mercado global de cada um desses dispositivos entre novembro de 2017 e novembro de 2018.

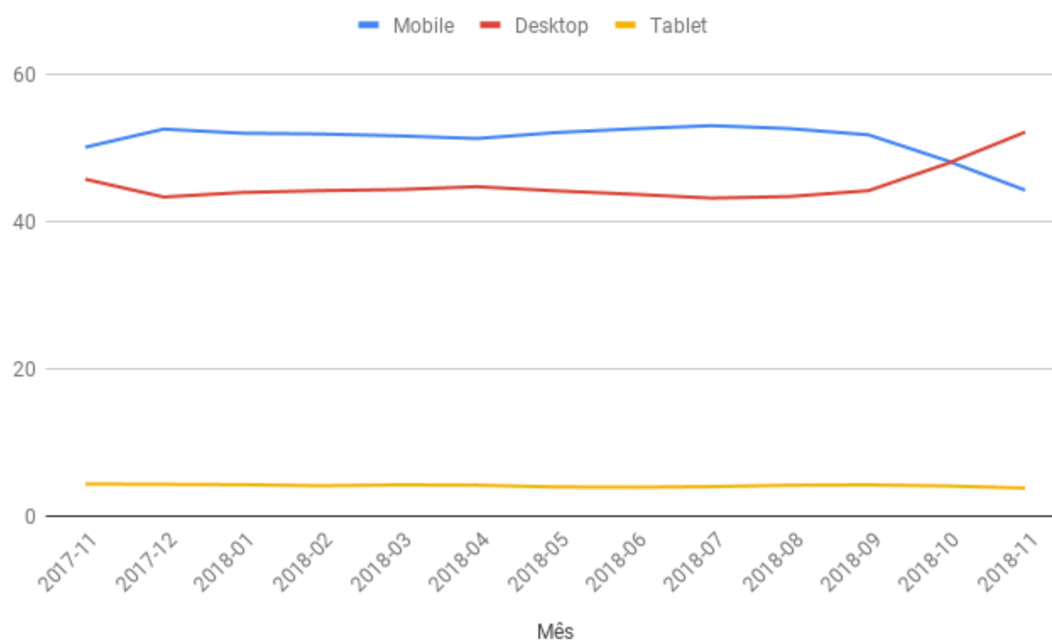


Figura 1.1: Fatia de mercado global: *Desktop vs smartphones vs tablets* (STATCOUNTER, 2018)

Considerando apenas os dispositivos móveis (*smartphones* e *tablets*) temos um domínio absoluto do sistema Android. A Figura 1.2 apresenta a fatia do mercado global dos principais sistemas operacionais utilizados em dispositivos móveis, entre novembro de 2017 e novembro de 2018.

O domínio do sistema Android é dado pelo fato de se tratar de um sistema *Open Source* utilizado por vários fabricantes de dispositivos, fazendo com que uma mesma aplicação se comporte de forma diferente entre dispositivos. A Figura 1.3 apresenta a quantidade de diferentes fabricantes de dispositivos compatíveis com o Android no ano de 2015. De acordo com um levantamento realizado pela *OpenSignal* (OPENSIGNAL, 2015), no ano de 2015 existiam mais de 24 mil diferentes modelos de dispositivos Android, considerando fabricante, configurações de tela e versão do sistema operacional. A esta variedade de configurações possíveis é dado o nome de fragmentação.

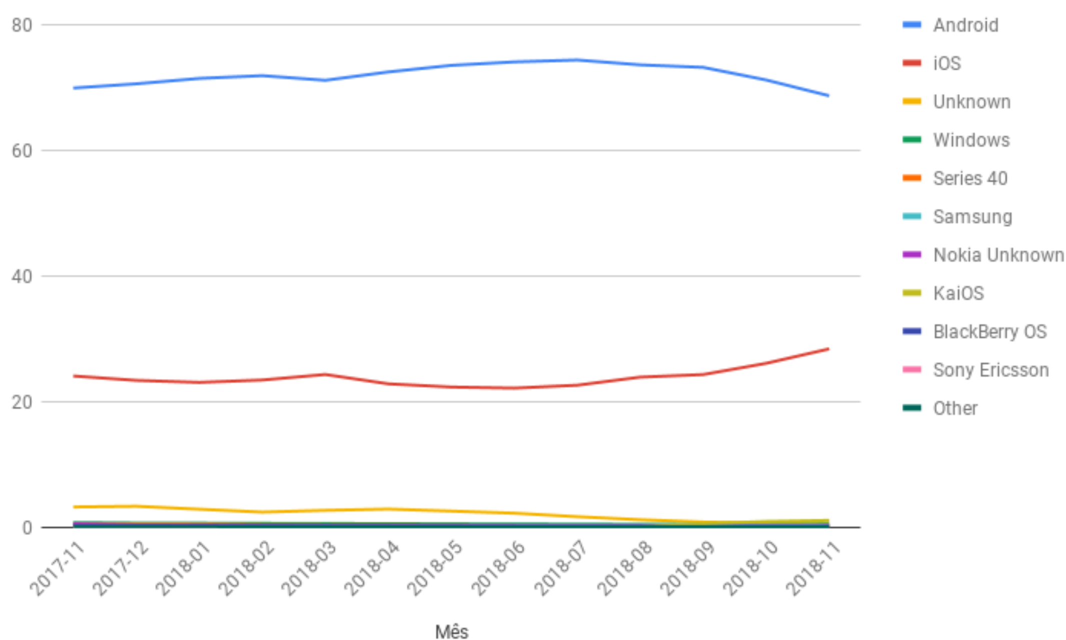


Figura 1.2: *Market Share* dos sistemas operacionais em dispositivos móveis (STATCOUNTER, 2018)

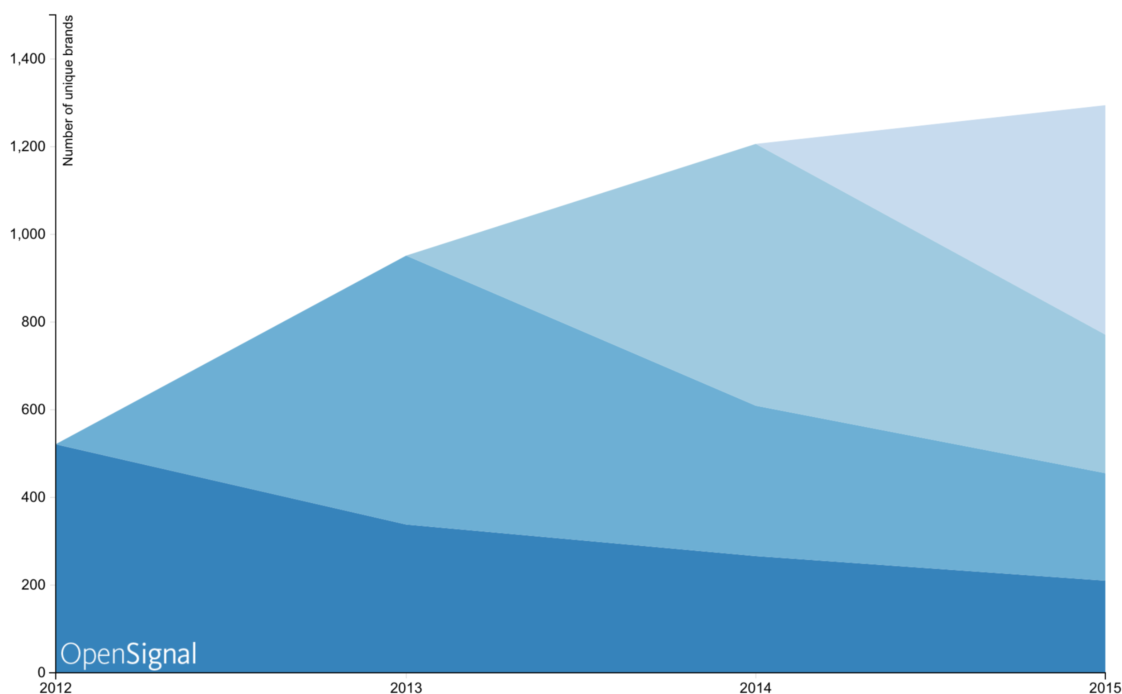


Figura 1.3: Número de fabricantes de dispositivos *Android* (OPENSIGNAL, 2015)

Devido a sua fragmentação, o ecossistema Android é extremamente desafiador aos times de desenvolvimento (HAM; PARK, 2011; VILKOMIR; AMSTUTZ, 2014; FREITAS et al., 2016; FENG et al., 2016), especialmente em atividades de verificação

de compatibilidade[2], pois várias configurações de hardware e diferentes versões de APIs devem ser consideradas durante os testes, exigindo que a validação ocorra em diferentes dispositivos e versões do sistema operacional.

Com o objetivo de otimizar o tempo gasto e, consequentemente o custo, do processo de verificação de compatibilidade, equipes de desenvolvimento/testes utilizam *frameworks* para automatização dos testes, especialmente testes UI: UiAutomator (Google Inc, 2018b), Calabash (Xamarin Inc, 2015), Espresso (Google Inc, 2013) e Appium (JS Foundation, 2012). Por meio destes testes, escritos em linguagem específica, é possível avaliar a compatibilidade de uma aplicação entre diferentes modelos de dispositivos automaticamente, este processo é conhecido como validação *cross device*. Entretanto, devido ao número de diferentes dispositivos a execução dos testes automatizados é um desafio a parte, pois a validação de uma aplicação em um modelo de dispositivo específico, se dá por meio da execução dos testes no referido modelo. A aquisição de todos os dispositivos pretendidos para a verificação de compatibilidade é economicamente inviável, e o uso de emuladores não garante alguns aspectos dos dispositivos reais (BROWSERSTACK, 2019):

1. Suporte para realizar e receber chamadas telefônicas reais;
2. Suporte para enviar e receber mensagens SMS;
3. Suporte para conexões usb;
4. Suporte para câmera;
5. Suporte para fones de ouvidos conectados ao dispositivo;
6. Suporte para detecção do estado de conexão com a Internet;
7. Suporte para determinar o nível de carga da bateria e o estado de carregamento;
8. Suporte para determinar a inserção/remoção de cartão SD;
9. Suporte para *Bluetooth*;
10. Suporte para *Multitouch*.

Vários serviços baseados em nuvem são oferecidos pela indústria a fim de fornecer uma boa variedade de dispositivos a um custo menor se comparado aos custos de aquisição dos dispositivos reais (AMAZON, 2018a), (Google Firebase, 2018), (XAMARIN, 2018), (KOBITON, 2018), (PERFECTO, 2018) e (Sauce Labs, 2018). No entanto, no momento da elaboração desta tese, os preços praticados ainda são altos para uma verificação em larga escala, além disso, vários dispositivos com expressiva fatia de mercado não são encontrados nestes serviços.

Em 2015, a empresa *OpenSignal* realizou um mapeamento acerca da fragmentação do ecossistema Android (OPENSIGNAL, 2015). O resultado é apresentado na Figura

[2]Verificação de compatibilidade de aplicações Android consiste no processo que busca garantir que uma aplicação esteja livre de *bugs* para um espectro de dispositivos pretendidos.

1.4. Os modelos *Galaxy S III*, *Galaxy S5* e *Galaxy S4* representavam a maior fatia de mercado dos dispositivos habilitados até a data do estudo. No mapa, os modelos *Moto E* e *Xperia Z1* representam uma significante fatia de mercado, no entanto, estes modelos não são disponibilizados nos provedores de serviço baseados em nuvem voltados à realização de testes, neste caso faz-se necessária a aquisição destes dispositivos, como se trata de dispositivos relativamente antigos, sua aquisição não é um processo simples.

Figura 1.4: Mapa da fragmentação do ecossistema *Android* (OPENSIGNAL, 2015)

Um estudo conduzido por Elberzhager et al. (2012), evidencia esforços dos pesquisadores em abordagens focadas na redução do custo envolvido nos processos de teste. A Tabela 1.1 apresenta a distribuição (organizada em categorias) dos trabalhos encontrados. Dos 144 trabalhos focados na redução de custo, 71 endereçam a automatização de testes, 41 estão relacionados à predição de defeitos, 22 propõem estratégias para redução do volume de entradas, sete sugerem técnicas para minimizar o volume de testes empregados durante a validação e três sugerem o emprego de estratégias de testes em determinadas fases do ciclo de vida do software.

Tabela 1.1: Distribuição de artigos baseados na sua classificação

categoria	artigos	%
Automação de teste	71	49.3
Predição	41	28.5
Redução da entrada de testes	22	15.2
Garantia da qualidade antes do teste	7	4.9
Estratégias de testes	3	2.0

Outras abordagens foram encontradas em trabalhos mais recentes. Abordagens baseadas em análise estática de código também são utilizadas como forma de verificação de compatibilidade (HAM; PARK, 2011), (WEI; LIU; CHEUNG, 2016). Em Lu et al. (2016), foi proposto um modelo para priorizar dispositivos Android a serem considerados no processo de verificação de compatibilidade de *games* e aplicações de mídia. Neste trabalho são utilizados dados do Wandoujia, ambiente de gerenciamento de aplicativos Android na China (WANDOUJIA, 2018). A abordagem utilizada se basiea na coleta de alguns dados de 3.86 milhões de usuários que utilizam *smartphones* na China. Os dados coletados se referem ao tipo de aplicações geralmente utilizadas em cada dispositivo (tempo das sessões abertas nos aplicativos, tráfego de rede, consumo de bateria e participação dos usuários em feedbacks sobre as aplicações utilizadas). Com base nos dados coletados, o modelo intitulado PRADA, sugere aos desenvolvedores de games e aplicações de mídias quais devem ser os dispositivos com maior chance de receber a aplicação desenvolvida. Em Khalid et al. (2014), é apresentada uma abordagem para auxiliar times de desenvolvimento na priorização de dispositivos com base em avaliações relacionadas a aplicações semelhantes, utilizando como base as bibliotecas em comum.

Outros trabalhos com o objetivo de reduzir os custos inerentes ao processo de testes em dispositivos dispositivos móveis são discutidos no Capítulo 6.

Para ilustrar a importância da verificação de compatibilidade a fim de evitar prejuízos futuros, a Tabela 1.2 apresenta o custo por instalação (CPI) de aplicativo em dispositivos em diferentes mercados. O CPI é calculado dividindo todos os gastos direcionais à propagação de uma aplicação (publicidade e custos para disponibilização de cada versão da *app* nas lojas oficiais) pelo número de instalações realizadas. Este dado é importante, uma vez que um erro ocorrido pode fazer com que o usuário não utilize mais o aplicativo, desperdiçando o CPI empregado. Se considerarmos o custo de um erro ocorrido em aplicativos bancários, de monitoramento, reserva de passagens e outros, as consequências podem ser economicamente severas.

Tabela 1.2: CPI por país (APPBRAIN, 2018)

	Investimento
Estados Unidos	$1.66
Índia	$0.30
Brasil	$0.44
Reino Unido	$2.25
Alemanha	$1.73

Portanto, a principal motivação desta tese é fornecer uma forma de diminuir os custos relacionados à execução de testes UI em ambientes fragmentados, utilizando dispositivos reais. Para isso foi implementada uma arquitetura para execução testes UI no ecossistema Android, capaz de executar testes de forma distribuída, utilizando o

paradigma da Economia Colaborativa (SUNDARARAJAN, 2016), detalhado na seção 2.4.

1.2 Objetivos

Com base na motivação previamente apresentada, o principal objetivo desta pesquisa é apresentar uma plataforma, intitulada DBB (Distributed Bug Buster), capaz de executar testes UI de forma distribuída, aproveitando dispositivos ociosos ao redor do mundo, de forma a maximizar a diversidade de dispositivos disponíveis para testes, minimizando os custos envolvidos na execução de testes em dispositivos reais.

Como objetivos específicos a pesquisa pretende:

1. Identificar limitações existentes na arquitetura que dá suporte à construção e execução de testes de aplicativos Android;
2. Provêr uma arquitetura escalável, capaz de executar testes em dispositivos reais geograficamente distribuídos;
3. Provêr uma arquitetura que permita o aumento da diversidade de dispositivos disponíveis para realização de testes de aplicativos Android;
4. Provêr uma plataforma que minimize o custo envolvido na execução de testes de aplicações Android em dispositivos reais;
5. Promover uma análise da viabilidde econômica da aplicação da Economia Colaborativa no contexto dos testes UI de aplicações móveis.

1.3 Metodologia

Times envolvidos no desenvolvimento de aplicativos para Android enfrentam sérios desafios quanto à garantia de compatibilidade, inerente a ecossistemas fragmentados. Embora várias soluções existentes de fato reduzem o custo da execução dos testes UI, sabe-se que vários modelos de dispositivos com razoável fatia de mercado podem ficar de fora da validação, prejudicando a cobertura e comprometendo a qualidade final do produto. A hipótese de que a economia colaborativa poderia viabilizar a execução massiva de testes UI, de forma a diminuir o custo de infraestrutura e contribuir para exploração da fragmentação foi testada e é a base para a identificação das questões de pesquisa a serem respondidas:

QP1 - É possível executar testes UI de forma eficaz em dispositivos reais geograficamente distribuídos?

QP2 - O uso da Economia Colaborativa promove a escalabilidade de testes UI em ambientes fragmentados?

QP3 - A Economia Colaborativa contribui para a redução do custo financeiro do processo de execução de testes de aplicativos que são executados em ambientes fragmentados?

As questões de pesquisa são respondidas analiticamente e por meio de experimentos envolvendo dispositivos reais disponíveis nas instituições nas quais temos acesso.

Como a pesquisa endereça a verificação de compatibilidade, por meio de testes UI, de aplicações móveis para ambientes fragmentados, foi escolhido o ecossistema Android, devido à sua alta fragmentação e expressiva fatia de mercado, apresentada na Seção 1.1.

1.4 Publicações

Durante o desenvolvimento desta tese, cinco trabalhos foram publicados:

1. FARIA, K. A. C.; FREITAS, E. N. d. A.; MALDONADO, J. C; VINCENZI, A. M. R. Pharos: Uma ferramenta para identificação de defeitos em ni´vel de me´todos a partir de commits e gerenciadores de defeitos. In: VII Congresso Brasileiro de Software. [S.l.]: SBC, 2016.
2. FREITAS, E. N. d. A.; CAMILO-JUNIOR, C. G.; FARIA, K. A. C.; VINCENZI, A. M. R. Amt: An android mirror tool for instant feedback across platform. In: SBC. Congresso Brasileiro de Software (CBSoft), 2016 7th Congresso Brasileiro de Software on. [S.l.], 2016. p. 429–440.
3. FARIA, K. A. C.; FREITAS, E. N. d. A.; VINCENZI, A. M. R. Collaborative economy for testing cost reduction on android ecosystem. In: ACM. Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing. [S.l.], 2017. p. 11–18.
4. MOURA, C. J. M. de; OLIVEIRA, S.; FARIA, K. A. C.; FREITAS, E. N. d. A. A new api for android accessibility testing. In: IEEE. 2017 International Conference on Computational Science and Computational Intelligence (CSCI). [S.l.], 2017. p. 594–598.
5. FARIA, K. A. C.; AQUINO GOMES, R. de; FREITAS, E. N. d. A.; VINCENZI, A. M. R. On using collaborative economy for test cost reduction in high fragmented environments. Future Generation Computer Systems, Elsevier, 2019.

1.5 Organização da Tese

No Capítulo 2 são apresentados os conceitos fundamentais para o entendimento da solução proposta e análise dos resultados. São apresentados os tipos de aplicativos móveis, uma vez que os recursos explorados nos dispositivos variam de acordo com o tipo de aplicativo. Os testes de aplicativos Android e *frameworks* disponíveis para a realização desses testes. As limitações da arquitetura existente para realização de testes UI no Android. O conceito de TaaS (*Test as a Service*) e os provedores desse tipo de serviço. Finalmente, os conceitos da Economia Colaborativa (EC) são apresentados, pois é com base neste modelo que a arquitetura proposta nesta tese pode se viabilizar.

No Capítulo 3 é apresentada a solução proposta para as limitações arquiteturais existentes nos *frameworks* que endereçam testes UI no Android. É apresentada uma visão arquitetural da solução, bem como a forma como esta deve ser utilizada, pois, se trata de um novo paradigma para execução de testes automatizados.

O Capítulo 4 traz os resultados dos experimentos realizados em prol da resposta à questão de pesquisa 1 (QP1 - É possível executar testes UI de forma eficaz em dispositivos reais geograficamente distribuídos?).

Finalmente, no Capítulo 5 é apresentada uma análise da escalabilidade da arquitetura proposta nesta tese, com o objetivo de responder às questões de pesquisa 2 e 3 (QP2 - O uso da Economia Colaborativa promove a escalabilidade de testes UI em ambientes fragmentados?; QP3 - A Economia Colaborativa contribui para a redução do custo financeiro do processo de execução de testes de aplicativos que são executados em ambientes fragmentados?), além de apresentar uma análise da viabilidade econômica do modelo proposto, em comparação com os serviços baseados em nuvem já existentes. Também neste capítulo, são apresentadas limitações técnicas e de mercado relacionadas à solução proposta, bem como considerações acerca de aspectos relacionados à segurança.

Como não foi encontrado nenhum trabalho com o objetivo em otimizar custos relacionados aos testes UI de aplicativos móveis, focado em uma solução arquitetural, no Capítulo 6 é feita uma discussão acerca dos principais trabalhos encontrados, com foco na redução de custos de testes em aplicativos móveis.

Por fim, a conclusão desta tese é apresentada no Capítulo 7.

CAPÍTULO 2

Fundamentação Teórica

Este capítulo traz a fundamentação teórica necessária ao entendimento desta tese. Primeiramente são apresentados alguns conceitos sobre testes de software, os tipos de aplicativos móveis, os testes utilizados nestes tipos de aplicações e suas fases, em seguida o conceito de instrumentação é apresentado e então os teste UI no ecossistema *Android* é descrito. Finalmente o capítulo apresenta conceitos sobre a economia colaborativa, que viabiliza a utilização da arquitetura proposta nesta tese.

2.1 Testes de Software

Teste de software é o processo de executar um programa ou sistema com a intenção de encontrar erros (MYERS; SANDLER; BADGETT, 1979). Os testes envolvem qualquer atividade que tenha como objetivo avaliar um atributo ou capacidade de um programa ou sistema e determinar se ele atende ou não aos resultados exigidos (GELPERIN; HETZEL, 1988). O software não é diferente de outros processos físicos em que as entradas são recebidas e as saídas são produzidas. A diferença está na maneira como o software falha. A maioria dos sistemas físicos falha em um conjunto fixo (e razoavelmente pequeno) de formas. Por outro lado, o software pode falhar de muitas maneiras. Detectar todos os diferentes modos de falha do software é geralmente inviável. Por isso, defeitos (comunmente chamados de *bugs*), estão sempre presentes em qualquer software não trivial, não por falta de cuidado ou irresponsabilidade dos desenvolvedores, mas porque a complexidade do software está além da capacidade humana de gerenciá-la.

2.1.1 Motivações e limitações dos testes

Como o software não é um sistema contínuo, o uso dos valores dos limites das entradas não são suficientes para garantir a correteza de um programa. Todos os valores possíveis precisam ser testados e verificados, mas testes completos são inviáveis. Testar um programa simples de forma exaustiva para adicionar apenas duas entradas inteiras de 32 bits (produzindo 2^{64} cenários de teste distintos) levaria centenas de anos, mesmo se

os testes fossem realizados a uma taxa de milhares por segundo. Entradas do mundo real são ainda mais desafiadoras, pois além de serem consideradas as possíveis entradas de um programa, existem circustâncias adicionais a serem consideradas nos testes.

De acordo com Everett e Jr (2007), a principal motivação para testar qualquer projeto de software é o mesmo: reduzir o risco de despesas de desenvolvimento não planejadas; e o risco de falhas do projeto. Esse risco de desenvolvimento pode ser quantificado como algum tipo de perda tangível, como a de receita, de clientes e até de vidas. Alguns riscos de desenvolvimento são tão grandes que a empresa aposta todo o seu negócio em um projeto. Para saber o tamanho do risco e sua probabilidade de ocorrer, uma avaliação de risco é realizada. Essa avaliação de riscos é uma série de perguntas estruturadas que investigam e auxiliam no mapeamento as causas mais prováveis de falha, dependendo do tipo de negócio que o projeto deve suportar. Essa motivação de risco é dividida em quatro objetivos de testes:

1. **Identificar a magnitude e as fontes de risco de desenvolvimento redutíveis por testes** - quando um projeto é considerado viável, é necessário identificar se os riscos envolvidos são reduzidos por meio de testes;
2. **Realizar testes para reduzir os riscos identificados** - os testes devem abordar resultados positivos (procurando por *features* que funcionem conforme especificado) e negativos (procurando por *bugs* que podem provocar falhas);
3. **Saber quando o teste está concluído** - o objetivo de atingir 100% das situações no processo de testes em geral é inatingível, portanto é necessário priorizar o que deve ser testado para que esteja claro o momento de finalizar os testes;
4. **Gerenciar os testes como um projeto dentro do projeto de desenvolvimento** - o custo e os benefícios dos testes no processo de desenvolvimento são consideráveis, portanto, é importante que os testes sejam planejados, documentados e gerenciados.

Independente de suas limitações, os testes são parte integrante do desenvolvimento de software e, portanto, devem ser implantados em todas as fases do ciclo de desenvolvimento, com as seguintes finalidades: Melhoria da qualidade; Verificação e validação; Estimativa de confiabilidade.

Hetzel (1988) apresentou as dimensões da qualidade com sendo: funcionalidade (relacionada à qualidade exterior), engenharia (relacionada à qualidade interior) e adaptabilidade (relacionada à qualidade futura). Cada dimensão está ligada a vários fatores, apresentados na Tabela 2.1

Tabela 2.1: Fatores Típicos de Qualidade de Software (HETZEL, 1988)

Funcionalidade	**Engenharia**	**Adaptabilidade**
Correteza	Eficiência	Flexibilidade
Confiabilidade	Testabilidade	Reusabilidade
Usabilidade	Documentação	Manutenibilidade
Integridade	Estrutura	

Bons testes fornecem métricas para todos os fatores relevantes. A importância de um fator em particular varia de acordo com o software. Qualquer sistema em que vidas humanas estejam em jogo, é necessário dar ênfase à confiabilidade e integridade. Em um sistema coorporativo típico, a usabilidade e a facilidade de manutenção são os fatores-chave, enquanto que, para um programa científico, esses dois fatores podem não ser significativos. Os testes, para serem totalmente eficazes, devem ser capazes de medir cada fator relevante e, assim, forçar a qualidade a tornar-se tangível e visível (HETZEL, 1988). Os testes UI, abordados nesta tese, quando bem escritos, fornecem métricas a cerca dos fatores relacionados à funcionalidade.

Testes com a finalidade de validar o software são denominados testes positivos. Portanto a validação tem o objetivo de verificar se o software funciona para os casos de teste especificados.

A confiabilidade do software tem relações importantes com muitos aspectos, incluindo a estrutura e a quantidade de testes a que foi submetido. Com base em um perfil operacional (uma estimativa da frequência relativa de uso de várias entradas para o programa (MICHAEL, 1996)), o teste pode servir como um método de amostragem estatística para obter dados de falha para a estimativa de confiabilidade.

Na próxima subseção, alguns conceitos e termos importantes para o contexto desta tese são apresentados.

2.1.2 Conceitos chave e taxonomia

Várias classificações e agrupamentos acerca dos tipos de teste foram concebidos desde a primeira definição sobre testes de software. Pan (1999) apresenta três agrupamentos dos testes: pelo propósito, pela fase do ciclo de vida, e pelo escopo. Everett e Jr (2007) classifica os testes em quatro abordagens: testes estáticos, testes de caixa branca, testes de caixa preta e testes de performance. As abordagens apresentadas por Everett e Jr (2007) são utilizadas nesta tese. A lista a seguir definie essas abordagens:

- **testes estáticos** - 85% dos defeitos de um software são introduzidos na fase de projeto (EVERETT; JR, 2007). Como o código tende a ser produzido após esta fase, os artefatos a serem testados para reduzir a quantidade de defeitos propagados

durante o projeto, não são artefatos de código. Desta forma, os testes estáticos se baseiam predominantemente em documentos de especificação;

- **testes de caixa branca** - é uma abordagem de teste no qual a estrutura/implementação interna do item que está sendo testado é conhecida pelo testador. O testador escolhe entradas para exercitar caminhos através do código e determina as saídas apropriadas. Esta abordagem pode ser aplicada em testes de unidade, integração e sistema.
- **testes de caixa preta** - é uma abordagem de teste no qual a estrutura/implementação interna do item que está sendo testado não é conhecida pelo testador. Esses testes podem ser funcionais ou não funcionais. Esta abordagem pode ser aplicada em testes de integração e testes de sistema.
- **testes de performance** - é um tipo de teste de software que pretende determinar como um sistema funciona em termos de capacidade de resposta e estabilidade sob uma determinada carga.

A realização de testes de caixa branca e testes de caixa preta está relacionada à criação de casos de teste, que consistem de um conjunto de condições ou variáveis sob as quais um testador determina se um sistema em teste satisfaz os requisitos ou funciona corretamente. Isso ocorre através do fornecimento de entradas e a avaliação das saídas fornecidas pelo programa. A checagem da saída, obtida através da execução de um caso de teste, observando os valores de entrada, se dá por meio de um mecanismo chamado de oráculo. Como o foco desta tese está em testes de interface gráfica com o usuário (testes UI), os testes estáticos bem como os testes de performance não serão abordados.

Os tipos de testes a serem empregados em cada abordagem, seja de caixa branca ou de caixa preta, são apresentados na Figura 2.1.

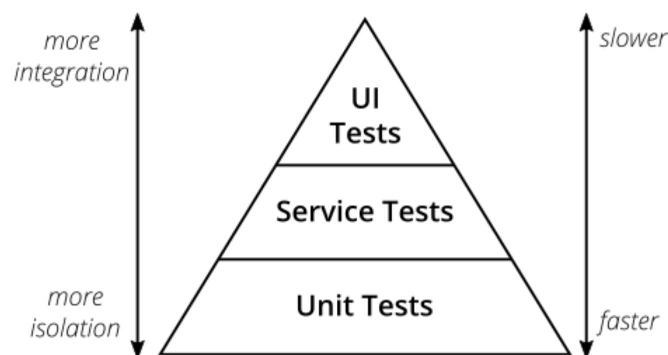


Figura 2.1: Pirâmide de Mike Cohn (COHN, 2010).

Os testes unitários (*Unit Tests*), na base da pirâmide, são constituídos de casos de testes envolvendo as menores unidades testáveis de um programa (em geral métodos, funções e subrotinas). Esses casos de teste são mais rapidamente executados e possuem maior capacidade de isolamento das unidades em teste. Os testes de integração (*service

tests), representados no meio da pirâmide, são constituídos de testes que envolvam a participação de vários componentes do sistema, portanto são executados mais lentamente por envolverem cenários mais complexos de serem configurados. Os testes de sistema, representados no topo da pirâmide pelos *UI Tests*, são constituídos de casos de testes mais complexos com o objetivo de verificar o funcionamento do sistema como um todo. Em geral, os testes de sistema são executados em condições normais de ambiente de produção. No contexto dos testes em dispositivos móveis, os testes UI são executados em dispositivos reais ou emuladores.

De acordo com Maldonado, Delamaro e Vincenzi (2018), é difícil decidir como testar um software de maneira completa e achar a maioria dos defeitos que ele possa conter, além disso, é necessário que a execução dos testes ocorra várias vezes. Este processo, quando executado somente por pessoas, em geral é caro, demanda tempo e é propenso a erros. Com o objetivo de reduzir o custo e o tempo gasto no processo de teste, a automatização de alguns tipos de testes pode ajudar as equipes no desenvolvimento de produtos livres de defeitos. Vários *frameworks* e ferramentas, foram desenvolvidos pela indústria e academia, auxiliam no processo de automatização, alguns destes *frameworks* são discutidos na Subseção 2.3.

Em alguns casos, a construção de testes unitários automatizados requer a utilização de unidades que simulem um determinado comportamento para que o item em teste possa ser isolado, mesmo que os itens que cooperam para a execução do cenário pretendido não estejam prontos ou se quer existam. Estas unidades, que simulam comportamentos, são chamadas de *mocks*. No contexto dos testes em dispositivos móveis, além de auxiliar no isolamento de unidades em teste, os *mocks* são frequentemente utilizados para simulação das respostas fornecidas por sensores e serviços disponíveis no dispositivo, mesmo durante a execução de testes de integração.

Como o foco desta tese está nos testes UI de aplicações Android, o restante desta seção se dedica a conceitos, ferramentas e *frameworks* utilizados neste contexto.

2.2 Tipos de Aplicativos

Como descrito no Capítulo 1, esta tese aborda ecossistemas móveis fragmentados, por isso, esta seção se dedica à apresentação de detalhes inerentes aos diferentes tipos de aplicativos direcionados à plataforma Android.

Três tipos de aplicações estão disponíveis no mercado de dispositivos móveis, aplicações nativas, web, e híbridas. Aplicações nativas constituem o tipo mais convencional e com maior nível de acesso aos dispositivos, são construídas através de linguagem própria e otimizadas para execução em determinado sistema operacional (SO). As aplicações web móveis são executadas a partir de um navegador e possuem um nível limitado

de acesso ao dispositivo, portanto, a experiência do usuário pode ser comprometida. Aplicações híbridas constituem um tipo interessante em que as duas abordagens anteriores são combinadas e por isso têm uso massivo no contexto corporativo. As Subseções 2.2.1, 2.2.2 e 2.2.3 abordam cada um dos tipos supracitados.

2.2.1 Aplicações Nativas

A Figura 2.2 apresenta uma visão de alto nível do processo de obtenção de um arquivo binário capaz de ser executado de forma nativa pelo SO.

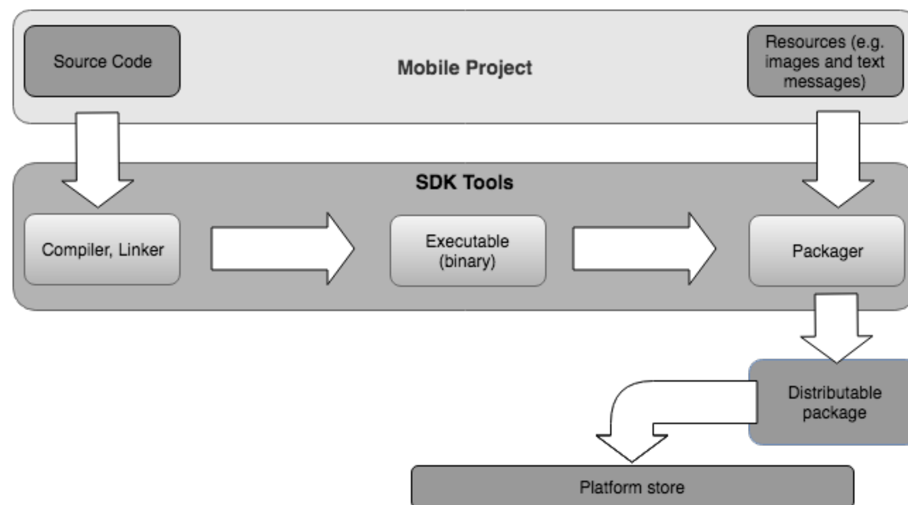


Figura 2.2: Esquema de compilação de aplicações nativas (GOOGLE, 2018b).

Tipicamente, a obtenção de um arquivo executável binário de um aplicativo móvel se dá da seguinte maneira: os desenvolvedores escrevem o código fonte da aplicação utilizando uma linguagem apropriada, criam recursos adicionais (imagens, áudio, configurações diversas, etc), e por meio de um *sdk (standard development kit)*[1] compilam o projeto gerando um executável em forma binária. O arquivo binário obtido é então empacotado junto com os recursos adicionais da aplicação, a fim de obter um pacote que possa ser distribuído, como por exemplo, a partir de uma loja de aplicativos. Como esta tese tem foco em ambientes fragmentados e, como descrito no Capítulo 1, o Android possui a maior fragmentação do mercado de *smartphones* e *tablets*, o restante desta seção se baseia neste SO.

Para criar uma aplicação Android nativa, os desenvolvedores escrevem seu código em Java, C, C++ e mais recentemente Kotlin. Para compilar e empacotar suas aplicações, os desenvolvedores utilizam o *Android SDK* (GOOGLE, 2018a), atualmente integrado às principais IDEs utilizadas para o desenvolvimento de aplicações Android. A

[1]SDK - No contexto de software para dispositivos móveis é um conjunto de ferramentas e utilitários para o desenvolvimento de software, geralmente providos pelo fabricante de um SO específico

saída gerada pelo *sdk* é um arquivo `.apk` que pode ser enviado à loja de aplicativos do Android.

O arquivo `.apk` gerado se trata de uma aplicação nativa, portanto com acesso irrestrito à API disponibilizada pelo SO. A Figura 2.3 ilustra a interação de uma aplicação Android nativa com o dispositivo em que está sendo executada. Todas as interações com o dispositivo são intermediadas por uma API proprietária provida pelo SO, esta API possui acesso direto ao hardware que equipa o dispositivo. Portanto, por meio da API nativa, a aplicação interage diretamente com componentes como *touch screen*, teclado, interfaces de conexão, dispositivos de áudio e vídeo, GPS, acelerômetro, entre outros.

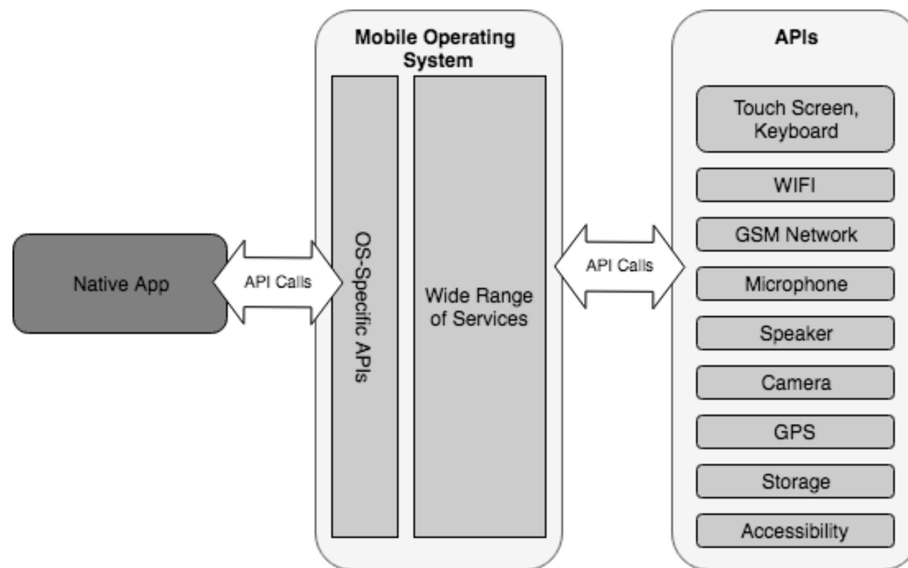


Figura 2.3: Interação de aplicações nativas com o SO.

A fim de otimizar o uso de suas APIs proprietárias, o Android disponibiliza uma API de alto nível, para acesso a recursos corriqueiros, bem como uma API de baixo nível, permitindo um acesso flexível aos recursos de hardware. As APIs mais utilizadas nos dois níveis são apresentadas na Figura 2.4

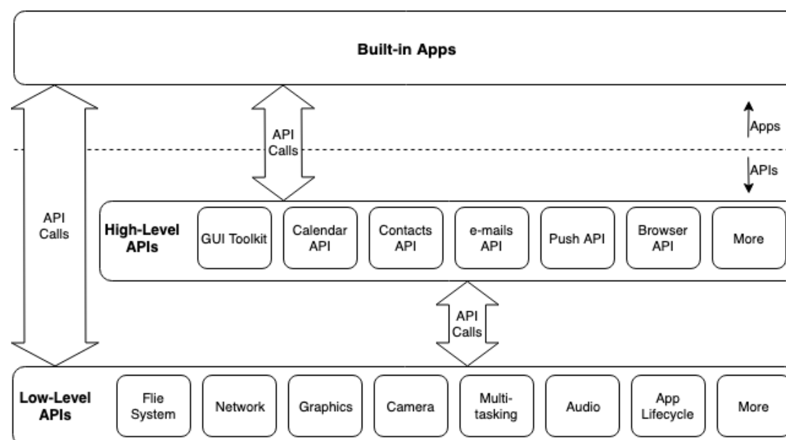


Figura 2.4: APIs de alto nível e baixo nível do *Android*.

Uma das APIs mais utilizadas envolve componentes de tela (*GUI Toolkit*), caixas de texto, listas, botões, *checkboxes*, etc. Além dos componentes puros, as aplicações frequentemente fazem uso da combinação de dois ou mais componentes gráficos, para uma melhor experiência do usuário. Esta flexibilidade no uso de componentes gráficos traz efeitos colaterais acerca do seu comportamento entre dispositivos com diferentes versões do Android, pois cada versão do SO pode utilizar diferentes camadas para renderizar os componentes construídos, causando falhas no momento da exibição e em alguns casos comprometendo a experiência do usuário. Este tipo de falha é geralmente identificada pelos testes UI, considerados nesta tese. As Figuras 2.5 e 2.6 apresentam os estados inicial e final de uma tela de um mesmo aplicativo executada em diferentes versões do SO. Após uma operação de rolagem (*scroll down*), o estado da tela muda de acordo com a versão da API do Android no dispositivo residente. Na Figura 2.6, o dispositivo da direita apresenta mais conteúdo quando comparado ao dispositivo da esquerda, mesmo que as configurações de tela sejam idênticas, a versão da API Android presente em cada um dos dispositivos causa um comportamento diferente da mesma aplicação.

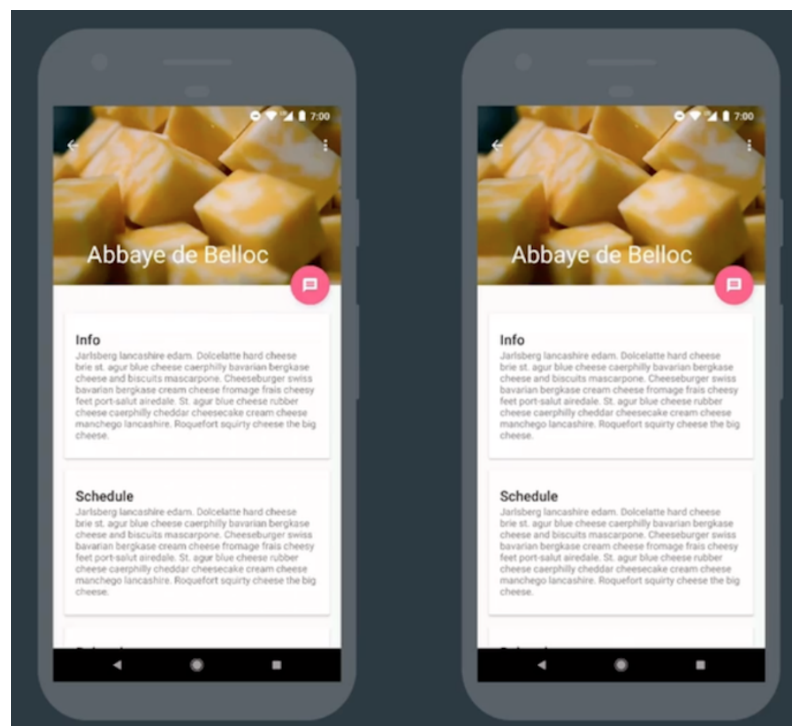


Figura 2.5: Estado inicial da aplicação

Estas diferenças de comportamento durante a execução de uma aplicação nativa, se deve ao fato destas interagirem com o sistema operacional por meio de chamadas nativas à API disponibilizada e então o acesso aos recursos do dispositivo é realizado. Usando as interfaces disponibilizadas pela API, a aplicação acessa os recursos existentes em cada dispositivo (WIFI, câmera, leitura e escrita em arquivos, etc). Como a interação entre a aplicação e o sistema operacional é direta, o nível de liberdade para o desenvolvimento

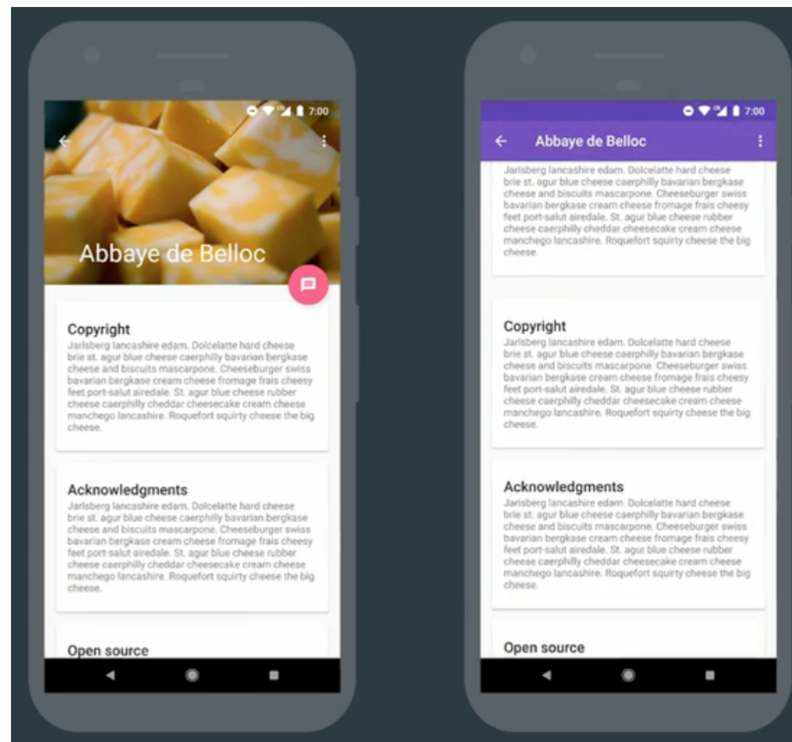


Figura 2.6: Estado final da aplicação após *scroll down*

privilegia a construção de aplicações que permitem aos usuário uma melhor experiência de uso.

Estas APIs evoluem por meio de manutenções corretivas e evolutivas, de forma que funcionalidades são incluídas, modificadas e até excluídas. Novos recursos de hardware e questões de segurança costumam ser os maiores motivadores para modificação das APIs, especialmente no baixo nível. O fato é que essas modificações podem causar impactos enormes nos aplicativos em produção, impossibilitando sua correta execução, comprometendo a experiência do usuário e causando prejuízos de grande magnitude. Este cenário reforça a importância da execução de testes em larga escala para determinados aplicativos que possuem um alto risco em relação às alterações em APIs.

Acerca das características de aplicações nativas, duas se destacam: 1) o acesso é extremamente rápido e flexível; 2) as APIs são proprietárias e cada versão do SO pode fornecer interfaces diferentes para acesso aos componentes de hardware e recursos do sistema. Esta última encoraja desenvolvedores a desenvolverem aplicações web móveis (*web app*) e aplicações híbridas, abordadas nas seções que seguem.

2.2.2 Aplicações Web

Motivada pelo efeito colateral à alta performance e flexibilidade, o elevado custo do desenvolvimento de várias *apps* dependendo do SO hospedeiro, a indústria de aplicações móveis desenvolveu vários *frameworks* para auxiliar desenvolvedores na

construção de *apps* multiplataforma (Corona Labs, 2017), (TheAppBuilder Ltd, 2016), (Adobe Systems, 2016), (IONIC, 2012). Estes *frameworks* são baseados em tecnologias web conhecidas (*HTML*, *JavaScript* e *CSS3*), e prometem entregar portabilidade de aplicativos que utilizam suas APIs. Este tipo de aplicação é chamado de *web app*.

Tipicamente uma *web app* é construída utilizando tecnologias web já existentes em aplicações convencionais baseadas em navegadores. Os *frameworks* utilizados na construção deste tipo de aplicação, empacota um *browser* utilizado para execução da *web app* no momento da compilação, de forma que a aplicação possa ser disponibilizada em forma de um arquivo `.apk` (no caso do Android) e instalado como uma aplicação nativa.

A Figura 2.7 ilustra a interação de uma *web app* com um dispositivo Android. O componente fundamental é o *Rendering Engine*, que traduz as instruções *HTML*, CSS e *JavaScript* em uma interface de usuário real. Este componente é chamado de *Webkit* (WEBKIT, 2018) e se trata de um projeto *Open Source* mantido pelos maiores fabricantes de SO do mercado com o objetivo de expôr uma API mínima às *web apps*.

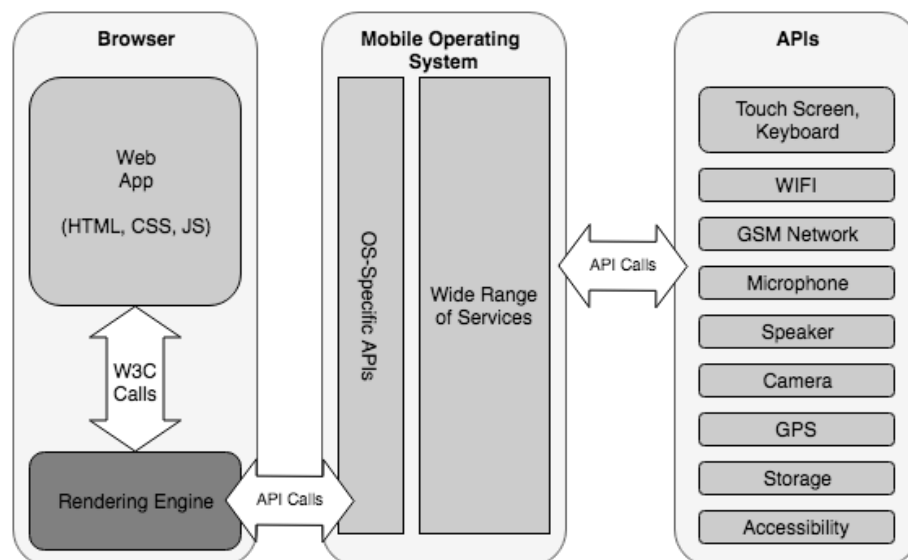


Figura 2.7: Interação de uma *web app* com o dispositivo

Alguns recursos presentes nos dispositivos móveis não são disponibilizados para aplicações *web app*, como é o caso do uso do *bluetooth* ou das câmeras. No entanto, o custo do desenvolvimento de *web apps* multiplataformas é inferior ao custo de aplicações nativas, pois até o momento da escrita desta tese, a construção de aplicações nativas exige um projeto diferente para cada ecossistema pretendido. A Tabela 2.2 apresenta características das aplicações móveis nativas e *web apps*.

Especialmente aplicações corporativas têm sido construídas utilizando o paradigma das *web apps*, em detrimento do alto custo de construção e manutenção de várias *apps* nativas que atendem à mesma demanda. Apesar de não entregar o melhor desempenho e o acesso ao dispositivo ser restrito, o custo razoável parece compensar estas características. No entanto, a obrigatoriedade de conexão com a Internet é um empecilho

Tabela 2.2: Aplicações Nativas vs Web apps

	Nativa	**Web**
Acesso ao dispositivo	total	parcial
Desempenho	excelente	satisfatório
Custo de desenvolvimento	caro	razoável
Necessita aprovação	obrigatório	inexistente

na garantia da manutenção de serviços mínimos ao usuário em casos extremos (FOX; BREWER, 1999), pois o *back-end* da aplicação é consumido como serviço e nada é armazenado no dispositivo.

Outra alternativa é a construção de aplicações híbridas, apresentadas na Subseção 2.2.3.

2.2.3 Aplicações Híbridas

Como apresentado na Subseção 2.2.1, as aplicações nativas entregam excelente performance e flexibilidade, promovendo a experiência do usuário. No entanto, o custo de desenvolvimento da mesma *app* para diferentes plataformas é um problema, especialmente no contexto corporativo. Para este tipo de cenário, aplicações *web app* são uma alternativa razoável, apesar de comprometer a performance e, em alguns casos, experiência do usuário. Considerando este contexto, as aplicações híbridas constituem um interessante tipo de aplicações que possuem tanto elementos baseados em tecnologias web quanto elementos nativos.

As principais características de uma *app* híbrida são:

1. É uma *app* nativa com componentes web embutidos;
2. Apresenta todos os benefícios de acesso ao dispositivo, existentes em uma app nativa;
3. Os elementos web de uma *app* híbrida podem ser obtidos a partir de um download convencional ou empacotado em na *app* disponibilizada na loja virtual do SO.

A Figura 2.8 ilustra a interação de uma *app* híbrida com um dispositivo real. A parte web da aplicação é executada pelo *Rendering Engine* e a parte nativa é executada diretamente pelo SO, de forma que é possível o acesso a todas as APIs.

Na próxima seção são apresentados os testes aplicados às apps Android.

2.3 Testes de apps Android

Mike Cohn apresentou o diagrama da pirâmide em seu livro *Succeding with Agile* (COHN, 2010), e embora existam críticas (FOWLER, 2018), este diagrama é utilizado nos conceitos de testes de aplicativos Android (GOOGLE, 2018c). A Figura 2.9 apresenta a

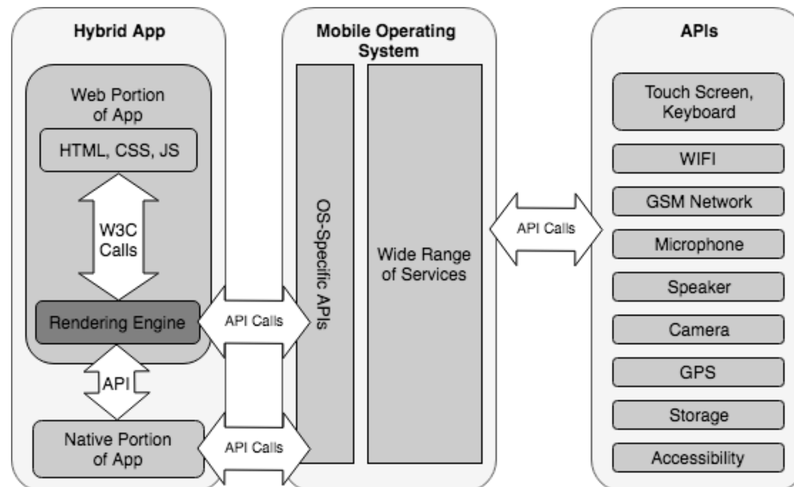


Figura 2.8: Interação de uma app híbrida com o dispositivo.

pirâmide de Mike Cohn mostrando os três tipos de testes funcionais a serem aplicados em suítes de testes de aplicativos Android. Estes testes podem ser divididos em três categorias:

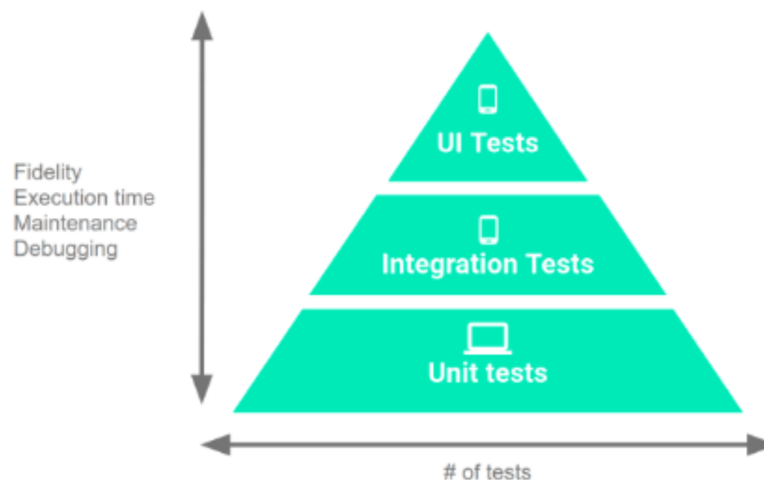


Figura 2.9: Pirâmide de Mike Cohn aplicada a testes no *Android* (GOOGLE, 2018c)

Testes pequenos - são testes unitários que podem ser executados de forma isolada e não necessariamente de dentro do dispositivo;
Testes médios - são testes de integração que integram vários componentes e embora possam fazer o uso de *mocks* são em geral executados de dentro do dispositivo;
Testes grandes - são testes de integração e interface que executam todo o fluxo que um usuário executaria. Por meio destes testes busca-se garantir que as interfaces gráficas se comportem de acordo com o esperado em dispositivos reais e emuladores.

Embora os testes unitários (testes pequenos) sejam rápidos de serem executados por lidarem com um domínio restrito e permitirem a identificação de falhas de forma ante-

cipada, este conjunto de testes proporciona baixa fidelidade resultando em baixa confiabilidade acerca do comportamento esperado da aplicação quando executada diretamente no dispositivo. O oposto ocorre com testes UI (testes grandes), pois sua fidelidade é maior, bem como seu domínio e dificuldade de manutenção.

Em função das diferenças existentes em cada um dos testes mencionados, recomenda-se a inclusão dos três tipos de testes apresentados no diagrama da pirâmide no processo de testes de aplicativos Android. Embora a proporção de testes de cada categoria possa variar em função das características funcionais da aplicação testada, a recomendação para garantia mínima de qualidade e custo moderado é a seguinte: 70% dos testes compostos por testes de unidade, 20% dos testes compostos por testes de integração e 10% dos testes compostos por testes UI (GOOGLE, 2018c).

A escrita e execução de testes unitários, bem como testes de integração são contemplados por qualquer *framework* que utilize uma máquina virtual para execução dos testes, no caso de testes que dispensem forte interação com o *Android framework*. Para testes unitários que exijam forte interação com o *Android framework*, ferramentas adicionais são recomendadas (ROBOELETRIC, 2018).

Em função da alta fragmentação inerente ao ecossistema Android, os Testes UI são um desafio a parte, pois focam em testar interações do usuário com as interfaces gráficas, por meio do reconhecimento e reação acerca das entradas do usuário a partir das interfaces gráficas.

Cada componente gráfico, tais como botões, menus, listas e caixas de texto, possui um conjunto de propriedades e API disponível e para testá-los recomenda-se (Google Inc, 2019):

1. Exercitar cada componente existente em cada uma das telas da aplicação;
2. Examinar o estado de cada componente utilizado, em momentos diferentes durante a execução da interface gráfica;
3. Fornecer entradas para todos os componentes das interfaces gráficas;
4. Verificar as saídas providas por cada componente de interface gráfica, a fim de encontrar inconsistências com base nas saídas esperadas.

De acordo com Joorabchi, Mesbah e Kruchten (2013) e Vasquez et al. (2018), os desenvolvedores de aplicativos testam manualmente cada interface gráfica na medida em que estas são construídas, a fim de garantir que as funcionalidades implementadas sejam contempladas, para isso são executadas operações sob a perspectiva do usuário e o comportamento da interface é verificado. Entretanto, esta estratégia manual é onerosa e propensa a erros. Além disso, em uma interface complexa é muito difícil cobrir todas as combinações possíveis às interações do usuário, sempre que alguma modificação é feita no código da interface gráfica. Considerando ambientes com alta fragmentação, as tarefas

manuais deveriam ser repetidas em todas as configurações pretendidas para validação do aplicativo. Portanto pode-se categorizar os problemas acerca da estratégia manual de testes em:

Domínio - Em uma interface gráfica existem muitas operações que precisam ser testadas. Mesmo uma aplicação pequena pode ter centenas de possibilidades de operações UI.
Sequência - Algumas funcionalidades do aplicativo podem depender de uma sequência de eventos UI.

A necessidade de cobrir o domínio relacionado aos componentes, em conjunto com a sequência de uso necessária para validação por meio dos testes, motivou o desenvolvimento de trabalhos e produtos voltados à automação dos testes UI (MAO; HARMAN; JIA, 2016), (MAO; HARMAN; JIA, 2017), (Google Inc, 2018c), (Google Inc, 2013), (JS Foundation, 2012), (JUNIT, 1998), (Spoon, 2015), (FAZZINI et al., 2017a).

A automatização dos testes UI, uma vez adotada, permite que os problemas de domínio e sequência sejam minimizados, possibilitando menos tempo gasto com testes já automatizados, apesar do custo inicial envolvido na automatização.

Dois tipos básicos de testes UI de aplicações Android norteiam a utilização de *frameworks* e ferramentas:

Testes UI envolvendo uma única *app* - Verifica se a *app* se comporta como esperado quando um usuário realiza ações ou fornece entradas específicas. Permite a verificação de saídas obtidas para entradas fornecidas em resposta às interações do usuário com interfaces específicas.
Testes UI envolvendo múltiplas *apps* - Verifica o comportamento correto de interações entre *apps* diferentes. Por exemplo, testes envolvendo uma *app* que faz uso do *Google Maps* (GOOGLE, 2018) ou mesmo acione a abertura da câmera fotográfica do dispositivo e execute uma fotografia.

Vários *frameworks* foram desenvolvidos a fim de apoiar a automação de testes UI capazes de explorar cenário com uma única *app* bem como cenários envolvendo múltiplas *apps*, Espresso (Google Inc, 2013), Appium (JS Foundation, 2012), UiAutomator (Google Inc, 2018b), entre outros.

A Tabela 2.3 apresenta os principais *frameworks* para a realização de testes em *apps* Android. Todos os *frameworks* que fazem uso da técnica de instrumentação são baseados no JUnit (JUNIT, 1998), como apresentado na Figura 2.10. No contexto dos testes de aplicativos móveis, a instrumentação é a técnica que permite que os testes sejam

executados em dispositivos reais e acessem todos os recursos da *app* alvo[2], no entanto, os testes instrumentados tem acesso apenas à *app* em que a instrumentação ocorreu. Desta forma, *frameworks* que fazem uso da instrumentação, auxiliam na criação de testes envolvendo uma única *app*, já os *frameworks* que não fazem uso da instrumentação auxiliam na criação de ambos os tipos de teste, envolvendo uma única *app* e envolvendo múltiplas *apps*.

Tabela 2.3: *Frameworks* para automatização de testes em *apps Android* (SAUCELABS, 2019).

	Linguagem	API's suportadas	Manutenção	Utiliza instrumentação
Appium	principais	todas	Active	sim
Calabash	Ruby	todas	descontinuado	sim
Espresso	Java	a partir da api 9	Google	sim
UIAutomator	Java	a partir da api 16	Google	não
Robotium	Java	todas	Comunidade	sim

Os *frameworks* apresentados na Tabela 2.3 trazem diferenças sensíveis quanto ao tipo de teste UI a ser empregado em uma app. *Appium* (JS Foundation, 2012) se trata de um *framework open source* para automatização de testes UI entre diferentes plataformas móveis, auxiliando na geração de testes de caixa preta (BEIZER, 1995). *UI Automator* é um *framework* disponível na biblioteca de suporte a teste do Android, capaz de auxiliar na geração de testes de caixa preta, foi construído com o objetivo de verificar comportamento da interação de múltiplas *apps*, pois permite que o teste possa interagir com qualquer elemento existente na tela (Google Inc, 2018b). *Robotium* (ROBOTIUM, 2019) e *Espresso* (Google Inc, 2013) são *frameworks* muito similares, pois funcionam com base na instrumentação e auxiliam na construção de testes UI (NIDHRA; DONDETI, 2012). Além de trabalharem com tipos de testes diferentes, *Robotium* apoiando testes de caixa preta e *Espresso* apoiando testes de caixa branca, outra diferença sensível é que o *Espresso* implementa funcionalidades de monitoramento à renderização de componentes de tela, permitindo ao testador maior abstração no controle do ciclo de vida de cada interface gráfica. Considerado como um *framework* para automatização de testes BDD (*Business-driven development*), *Calabash* (Xamarin Inc, 2015) auxilia testadores na geração de testes em linguagem natural (Cucumber, 2015), que posteriormente são convertidos em testes compatíveis com o *Robotium*.

Como o *Espresso* é o *framework* oficial de testes UI do Android, a proposta arquitetural desta tese foi desenvolvida com base neste *framework*, pois além de proporcionar vantagens técnicas em testes envolvendo uma única *app*, acredita-se que este irá evoluir de acordo com as mudanças no sistema Android. Na próxima seção é descrito como se dá a geração e execução de testes com o *Espresso*.

[2] *app* alvo - app que está sendo testada/validada

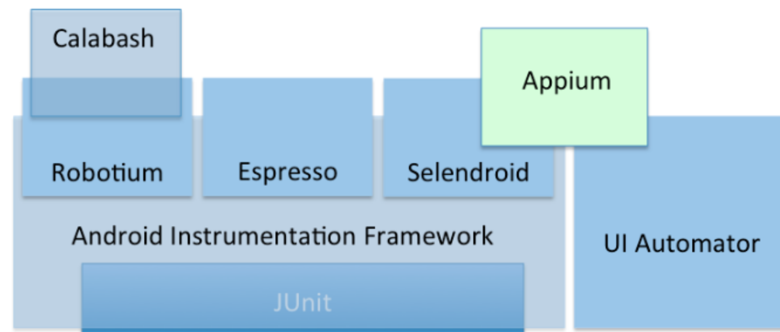


Figura 2.10: Principais frameworks de testes Android.

2.3.1 Testes usando o Espresso

Como descrito no início da Seção 2.3, o *Espresso* baseia-se na construção de testes instrumentados, desta forma os testes são executados em um mesmo processo criado pelo sistema operacional para execução do aplicativo testado, permitindo que os testes tenham acesso a todos os recursos do aplicativo. Os testes escritos são executados em um dispositivo ou emulador Android, para isso o *sdk* do Android gera dois arquivos com extensão `.apk` para serem instalados no Android, um arquivo correspondente à *app* alvo e outro correspondente aos testes construídos. Este processo é ilustrado na Figura 2.11.

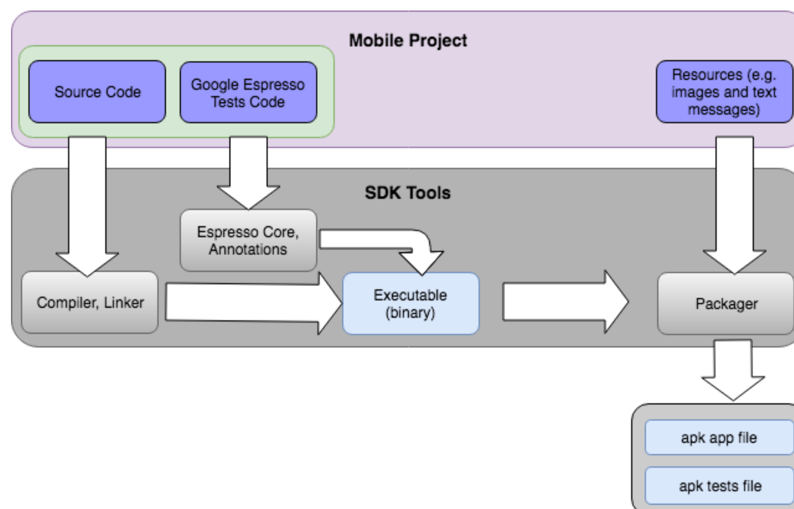


Figura 2.11: Esquema de geração de `apks` de testes

No ambiente de desenvolvimento, os arquivos com extensão `.apk` gerados pelo *sdk* são transferidos e instalados no dispositivo, resultando em duas *apps* instaladas, então a *app* relacionada aos testes é invocada pelo `adb` *(Android Debug Bridge)* (GOOGLE, 2010). Este componente está presente em todos os dispositivos Android e é fundamental para a execução de aplicações e testes em ambiente de desenvolvimento. A Figura 2.12 apresenta uma visão de alto nível dos componentes do `adb` envolvidos no processo de instalação e execução de aplicações e seus testes.

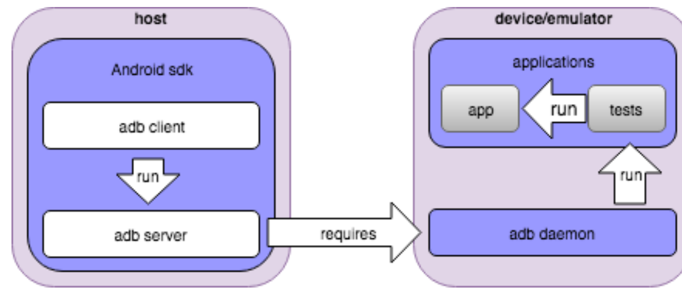


Figura 2.12: Esquema de funcionamento do `adb`

O `adb client` é executado no *host* de desenvolvimento e é responsável pelo envio de comandos ao `adb server`, este executa os comandos nos dispositivos presentes no *pool* de conexões abertos a partir do `adb daemon`, presente nos dispositivos e emuladores Android. Durante a inicialização do `adb client` é verificado se existe um processo `adb server` em execução, caso não exista, o `adb client` inicia um processo `adb server`, que permanece monitorando a porta 5037, utilizada para a comunicação com o `adb client`, além do intervalo de portas utilizadas para abertura de conexões oriundas do `adb daemon` (da porta 5555 até a porta 5585). Para que os componentes `adb` funcionem é necessário a configuração individual em cada dispositivo, habilitando a opção de depuração USB. Uma vez conectado ao *host* de desenvolvimento, um dispositivo é acessível por meio da API de comandos disponível no `adb client`, esta API é utilizada pelo *Android sdk* no momento de realizar a instalação dos arquivos com extensão `.apk` (do aplicativo e dos testes) em todos os dispositivos que possuírem conexão com o `adb server` a partir do `adb daemon`. Esta arquitetura foi desenvolvida para que a instalação e desinstalação de uma *app* em um dispositivo conectado ao *host* de desenvolvimento, ocorra de forma silenciosa, o que não é possível caso o dispositivo esteja desconectado de um *host* configurado com o *sdk Android*. Graças a este modelo, não é necessário que o dispositivo seja desbloqueado para acesso como superusuário. Esta exigência do Android é contornada pela arquitetura proposta nesta tese a partir da autorização voluntária do usuário. No Capítulo 3 são apresentados detalhes sobre a abordagem utilizada.

A Figura 2.13 apresenta o conjunto de passos realizados pelo *Android sdk* durante a instalação e execução do aplicativo e seus testes. Primeiramente o aplicativo em desenvolvimento é instalado, em seguida seus testes são instalados e por último o orquestrador. Finalmente os testes instrumentados são iniciados pelo `adb`. O orquestrador de testes Android é um componente presente no *AndroidJUnitRunner* e permite que cada um dos testes execute de forma independente e isolada, provendo os seguintes benefícios:

Estado compartilhado mínimo - Cada teste executa em sua própria instância de instrumentação. Portanto, se os testes compartilham algum estado da app (por exemplo, conexões, variáveis estáticas, etc), estes são removidos da CPU ou memória após a execução de cada teste.

As falhas são isoladas - Mesmo que um teste falhe, apenas sua instância de instrumentação é atingida, de forma que os outros testes continuam sendo executados.

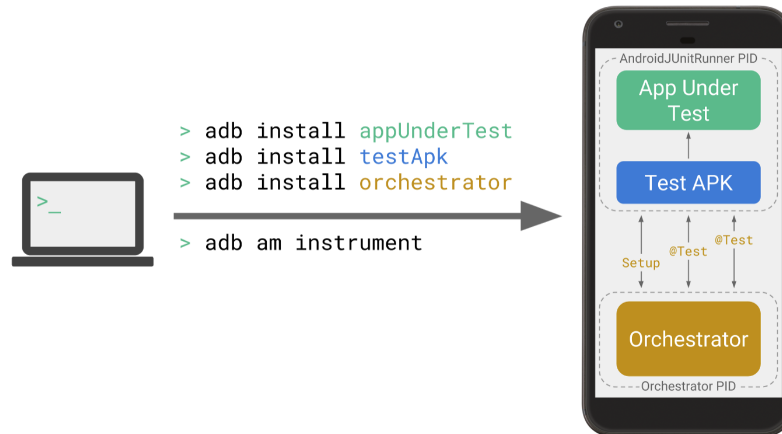


Figura 2.13: Instalação de testes instrumentados pelo *Android SDK*

Todo o processo apresentado na Figura 2.13 é possível por meio de uma arquitetura totalmente dependente de uma conexão física (via *socket*), o que traz limitações importantes considerando a quantidade de dispositivos contra os quais um testador pode realizar testes simultaneamente. Além disso, o próprio `adb` traz limitações quanto ao número de conexões simultâneas gerenciadas pelo `adb server`, ao todo 35 dispositivos simultâneos. Neste cenário, o testador precisa iniciar a suíte de testes em cada um dispositivos conectados ao *host* de desenvolvimento ou utilizar alguma solução de distribuição de testes (Spoon, 2015). No entanto, apesar de eliminar a necessidade da inicialização dos testes em cada dispositivo, a quantidade de dispositivos possíveis ainda continua limitada à quantidade de portas disponíveis para o `adb`.

Além das limitações técnicas inerentes ao uso do `adb` por parte do *Android sdk*, ainda existem limitações relacionadas ao custo, pois de acordo com o modelo atual é necessário a aquisição de dispositivos reais ou a contratação de serviços no modelo *TaaS (Test as a Service)*. Na próxima seção são abordados os principais provedores do mercado que oferecem dispositivos em nuvem para serem usados em testes de aplicativos para dispositivos móveis.

2.3.2 Provedores de dispositivos em nuvem

Na Subseção 2.3.1 foi apresentada a forma convencional da execução de testes *Espresso* para validação de aplicativos Android. Nesta subseção, serão abordados os principais provedores do mercado que oferecem dispositivos reais para serem utilizados como *IaaS (Infrastructure as a Service)* a fim de atender atividades de testes apoiadas no paradigma *TaaS*.

Tabela 2.4: Provedores de dispositivos móveis como IaaS

	Dispositivos	Diversidade	Contrato	Dispositivos simultâneos
Google Firebase Test Cloud	nd	69 modelos	sob demanda	nd
AWS Device Farm	1000	191 modelos	sob demanda e reserva	5
Xamarin Test Cloud	2000	250	reserva	30
Perfecto	10000	62	reserva	3
Kobiton	300	95	reserva	1

Vários autores consideram o *TaaS* como um novo modelo de negógio (ZHANG et al., 2015; AMALFITANO et al., 2012; ANAND et al., 2012a; BAI et al., 2013; BO; XIANG; XIAOPENG, 2007; BUYYA; RANJAN; CALHEIROS, 2009; GAO et al., 2012; GAO et al., 2014; GAO; BAI; TSAI, 2011; GAO et al., 2014; HARGASSNER et al., 2008; MUCCINI; FRANCESCO; ESPOSITO, 2012; RIDENE; BARBIER, 2011; RIUNGU; TAIPALE; SMOLANDER, 2010; SATOH, 2004; SATOH, 2003; TAO; GAO, 2014; TSAI; HUANG; SHAO, 2011; YANG et al., 2011). Um provedor de *TaaS* realiza atividades e tarefas de testes em software utilizando infraestrutura em nuvem e as fornece como serviço.

No caso de aplicações móveis, os *TaaS* são disponibilizados como serviços que oferecem diversos dispositivos para serem utilizados no processo de validação, diminuindo o custo por meio do compartilhamento de recursos.

A Tabela 2.4 apresenta os principais provedores de *TaaS* que disputam um mercado de desenvolvedores/testadores que apresentam demandas de validações de aplicações móveis, especialmente envolvendo o ecossistema Android.

Normalmente, há duas maneiras de se contratar infraestrutura de teste de nuvem, com base na demanda, em que os usuários pagam de acordo com seu uso, e por reserva, em que os usuários pagam por um recurso previamente dimensionado a ser usado em um determinado período de tempo. No modelo de demanda, os desenvolvedores correm o risco de pagarem muito caro devido à alta utilização de recursos; por outro lado, no modelo de reserva, o risco inerente é superdimensionar a alocação de dispositivos. Até o momento da escrita desta tese, a disponibilidade das *IaaS* destinadas aos *TaaS* de cada um dos provedores apresentados na Tabela 2.4 é a seguinte:

Google Firebase Test Cloud - oferece 69 modelos diferentes de dispositivos que podem ser utilizados sob demanda (Google Firebase, 2018).

Aws Device Farm - oferece 1000 dispositivos de 191 modelos diferentes, podendo ser contratados sob demanda ou por reserva. Um limite de 5 dispositivos simultâneos é colocado para evitar filas em dispositivos com alta demanda (AMAZON, 2018a).

Xamarin Test Cloud - oferece 2000 dispositivos de 250 modelos diferentes, podendo ser contratados por reserva. Um limite de 30 dispositivos simultâneos é colocado para evitar filas em dispositivos com alta demanda (XAMARIN, 2018).

Perfecto - oferece 10 mil dispositivos de 62 modelos diferentes. O formato de alocação é por reserva e é possível a utilização de até 3 dispositivos simultaneamente (PERFECTO, 2018).

Kobiton - oferece 300 dispositivos de 95 modelos diferentes. O formato de alocação é por reserva e é possível a utilização de apenas 1 dispositivo para execução de testes (KOBITON, 2018).

No Capítulo 5 é apresentada uma comparação da plataforma proposta nesta tese e estes provedores. Na próxima seção são abordados conceitos fundamentais da economia colaborativa, paradigma que inspirou o desenvolvimento desta arquitetura e sua disponibilização em uma plataforma.

2.4 Economia Colaborativa

Nesta seção, são apresentados conceitos fundamentais do paradigma da Economia Colaborativa (EC) e como o processo de execução de testes de software poderia ser aprimorado, a partir do compartilhamento de recursos, reduzindo custo e promovendo a escalabilidade e diversidade de modelos de dispositivos móveis.

Durante a primeira fase do capitalismo industrial, os produtos que antes eram feitos pelas pessoas, de forma manual, em suas próprias casas, gradualmente começaram a ser produzidos em grande escala pelas fábricas (RIFKIN, 2001), em atendimento às demandas existentes. Os trabalhadores consumiam produtos, levando a produção maciça desses produtos ao domínio da economia capitalista até meados do século XX. A partir deste período, iniciou-se a comercialização de informações e a uma vasta quantidade de novos serviços, dando origem a era dos serviços (TÉBOUL, 1999), em que ao invés de pensar em produtos como itens fixos, com determinadas especificações, e um valor de venda em determinado momento, as empresas passaram a oferecer serviços e acreditar neles como plataforma de melhoria de atendimento de valor agregado focado em estabelecer um relacionamento de longo prazo com seus clientes. Com a chegada da Internet, a indústria passou a ter mais uma ferramenta apoiando o provimento de serviços de alto valor agregado, chegando a mudar a forma como as pessoas faziam negócio, através do e-commerce e modelos B2B (Business to Business).

A partir do século XXI, o mercado de mídias se redefiniu, de forma que os usuários pudessem ter acesso a um conteúdo desejado a partir de serviços de *streaming*, sem necessariamente adquirir um produto físico. Portanto, várias transições econômicas

significativas ocorreram, uma delas é a transição da posse, de produtos e serviços como ativos para o acesso *just-in-time* a bens e serviços. Em que as pessoas, cada vez mais, pagam pelo acesso a seus produtos e serviços, por informações, entretenimento, hardware, software, etc.

Atualizações, inovações e personalizações são liberadas para a população cada vez mais rapidamente; o ciclo de vida do produto foi encurtado, tornando os produtos obsoletos mais rapidamente. Dessa forma, confiar em ativos durante a era do acesso (RIFKIN; PEREIRA, 2001) pode ser caro e ineficiente, e a prática de pagar para acessar alguns tipos de produtos e serviços é quase sempre mais barata.

Com a popularização da Internet e a evolução tecnológica, a indústria de software pôde fornecer produtos como um serviço por meio de acesso remoto, para que os consumidores pudessem acessar seu conteúdo e usá-lo determinado tempo. Vários exemplos podem ser mencionados, entre eles o *Apple music* (Apple, 2015) e os filmes do *Google Play Movies & TV* (Google, 2015).

Outro exemplo sobre a economia baseada em acesso é a computação em nuvem, em que é possível fornecer a infraestrutura como um serviço. Na indústria de software, esse modelo ganha popularidade porque as empresas, especializadas no desenvolvimento de software, podem se concentrar em seu problema real e usar toda a infraestrutura externamente e de acordo com a demanda por seus serviços.

A era do acesso trouxe algumas facilidades, simplificou diversas áreas e suportou diversos mercados. No entanto, construir uma infra-estrutura para fornecer serviços por meio de acesso ainda, pode ser um desafio e implica em alto custo e alta complexidade para manter. Assim, outras mudanças significativas ocorreram nas relações de mercado, em que as pessoas tiraram proveito da capacidade instalada disponível na sociedade como um todo e construíram um ecossistema socioeconômico que compartilha recursos humanos, físicos e intelectuais (KOTLER; KARTAJAYA; SETIAWAN, 2010).

A Economia Colaborativa (EC) reflete esta mudança, hoje existem vários serviços oferecidos com base neste paradigma. Por exemplo, Uber (Uber, 2008), Lyft (Lyft, 2012), Getaround e Turo (Turo, 2009) redefiniram como as pessoas se locomovem pelas cidades, a DoorDash (DoorDash, 2013) inovou a forma como empresas podem utilizar o serviço de entrega de comida e a Airbnb (Airbnb, 2008) revolucionou a forma como as pessoas se hospedam. Seguramente, a capacidade instalada para atender às demandas é aumentada pelo aproveitamento e compartilhamento da infraestrutura existente.

Graças a este novo paradigma, para alguém oferecer um serviço de hospedagem não precisa construir um hotel, basta compartilhar algum espaço ocioso em sua casa para prestar o serviço, desta forma tanto o proprietário do imóvel quanto o hóspede ganham neste relacionamento, satisfazendo um importante pilar do paradigma da Economia Colaborativa, a relação ganha-ganha. Esse tipo de serviço, baseado em colaboração, tem

atraído o interesse de empresários (INDEX.CO, 2017), mesmo assim, várias áreas da economia ainda não são afetadas por este modelo.

Um fato interessante é que até hoje não há aplicação da economia colaborativa na engenharia de software e tão pouco em testes de software. Portanto, com o objetivo de estabelecer uma relação ganha-ganha entre desenvolvedores de aplicativos e proprietários de dispositivos móveis em todo o mundo, esta tese traz esta reflexão e a validamos por meio de uma plataforma capaz de minimizar o custo da validação de aplicações utilizando dispositivos reais.

Na próxima seção é descrito como a EC pode ser aplicada para redução dos custos envolvidos na execução de testes de software.

2.4.1 Economia Colaborativa Aplicada ao Teste de Software

Apesar do alto custo envolvido nos testes de qualquer tipo de software, esta seção se dedica a apresentar uma reflexão acerca da aplicação da EC na redução do custo de testes de aplicações Android em dispositivos reais.

Desenvolvedores e testadores do setor de aplicativos móveis precisam validar seus aplicativos para fornecer qualidade e confiabilidade a seus usuários, mas ao mesmo tempo o orçamento e o tempo são limitados. Desta forma, minimizar o custo e o tempo inerentes ao processo de validação de aplicações em larga escala pode ajudar os envolvidos a fornecer aplicativos com melhor qualidade.

O paradigma da EC permite enxergar os dispositivos móveis (sobretudo *smartphones* e *tablets*) habilitados ao redor do mundo, como uma enorme infraestrutura disponível, especialmente aqueles dispositivos que executam o Android. Desta forma, é possível a criação de uma grande *IaaS* composta por dispositivos espalhados ao redor do mundo, tornando possível a cobertura de quase 100% do mercado de dispositivos Android, sem a necessidade da aquisição destes dispositivos pelos envolvidos no processo de validação. Pessoas anônimas poderiam se beneficiar com isso, oferecendo seus dispositivos como infraestrutura de testes barata para desenvolvedores e testadores validarem suas aplicações, estabelecendo uma relação ganha-ganha e gerando assim um novo mercado.

Portanto, para que a EC possa ser utilizada para minimizar custos inerentes à execução de testes, é necessário uma plataforma que estabeleça o relacionamento entre pessoas anônimas (proprietários de dispositivos) e desenvolvedores/testadores, de forma que testes possam ser enviados a dispositivos que atendam determinadas configurações, os relatórios de execução sejam entregues aos interessados e os proprietários dos dispositivos possam ser recompensados pelo compartilhamento do recurso.

Assim como em todas as plataformas de serviço baseadas em EC, quem remunera o sistema é o tomador do serviço. No caso dos serviços de transporte, o tomador do

transporte é quem paga pela viagem, remunerando tanto o prestador do serviço quanto a plataforma que viabiliza o serviço. No contexto dos testes de software, desenvolvedores/-testadores (considerados os tomadores do serviço) iriam remunerar tanto o proprietário do dispositivo quanto a plataforma que viabiliza a execução remota e distribuída dos testes de interface.

Para que testes UI escritos usando o *Espresso* (Google Inc, 2013) possam ser executados, é necessário que tanto a aplicação quanto os testes sejam instalados em todos os dispositivos pretendidos, como descrito na Subseção 2.3.1. Como o Android, por questões de segurança, limita as atividades que podem ser executadas de forma silenciosa, ou seja, sem que o usuário interaja com o dispositivo, o próprio proprietário do dispositivo precisa executar o processo de instalação no seu aparelho. Portanto, existe o risco de testes enviados a um dispositivo não serem executados, pois no momento da instalação podem ocorrer erros fazendo com que o proprietário do dispositivo aborte e não permita a inicialização dos testes.

Como a execução de testes UI em ambientes remotos são assíncronas, o desenvolvedor/testador precisa de um relatório contendo o resultado dos testes executados em cada dispositivo. Para que estes relatórios sejam enviados aos interessados, o dispositivo precisa estar conectado à internet, trazendo mais um risco para os testes executados remotamente, impossibilitando os interessados no testes de receber o relatório de execução e o proprietário do dispositivo de receber sua recompensa.

Embora existam cenários de integração que possam apresentar incompatibilidades entre dispositivos que executam diferentes versões do Android e são equipados com componentes e *drivers* que possuem algum *bug* explorado pela aplicação, testes unitários e de integração podem ser executados em ambiente de desenvolvimento usando emuladores, pois o uso de dispositivos físicos para a execução deste tipo de teste, pode não apresentar uma boa relação custo/benefício. Portanto a execução de testes em diferentes dispositivos físicos tem o objetivo de encontrar problemas que afetam o comportamento da aplicação quando executada em ambientes diferentes, este processo é conhecido como validação *cross-device*. No caso de aplicações Android, para garantia da qualidade, é necessário que os testes UI sejam executados em diferentes ambientes, a fim de identificar problemas de compatibilidade.

Outra característica importante apresentada por um ambiente de testes UI baseado em EC, é o fato de que os testes não são manualmente iniciados pelos desenvolvedores/testadores, estes apenas definem os dispositivos desejados e enviam os testes a serem executados. Neste ambiente, o próprio proprietário do dispositivo permite o inicio dos testes quando lhe for conveniente, desde que o prazo máximo para retorno do relatório de execução não seja excedido.

Em prol da qualidade e da variedade dos testes que podem ser executados em dispositivos remotos, a plataforma apresentada nesta tese não coloca limitações de acesso ao dispositivo, senão as já impostas pelo Android. Desta forma, se a aplicação testada acessa um arquivo armazenado no dispositivo e o algum teste explora esta funcionalidade, é necessário que esta solicitação apenas dependa das notificações de *runtime* já previstas pelo sistema operacional. Embora esta medida traga problemas de segurança ao proprietário do dispositivo, entende-se que este está assumindo um risco com base na recompensa definida.

2.5 Considerações Finais

A fim de apresentar os problemas relacionados à execução de testes automatizados, objeto desta tese, este capítulo apresentou como ocorre a interação de cada tipo de aplicativo com dispositivos Android. Foram abordados os tipos de testes aplicados aos aplicativos móveis, os frameworks que endereçam cada tipo de teste e suas limitações arquiteturais.

Foi descrito como ocorre a compilação e execução de testes instrumentados utilizando o principal *framework* de testes para Android, o Espresso, além de apresentar os principais provedores do mercado que disponibilizam dispositivos reais para execução de testes automatizados. Por fim, foram apresentados conceitos sobre Economia Colaborativa e como esta poderia ser utilizada para minimizar os custos inerentes à execução de testes automatizados em dispositivos reais.

No próximo capítulo, é apresentada a solução proposta nesta tese. São apresentados detalhes arquiteturais sobre a construção da plataforma proposta.

CAPÍTULO 3

DBB: Uma proposta de solução para execução de testes UI distribuídos

Neste capítulo é apresentada a proposta de solução para o problema da execução de testes UI sem a necessidade de conexão física estabelecida pelo `adb`. Intitulada DBB (*Distributed Bug Buster*), a solução tem o objetivo de executar de testes UI (escritos utilizando o *Espresso* (Google Inc, 2013)) de forma distribuída.

De acordo com (GAO et al., 2013), o termo *Testing as a Service* (TaaS) foi inicialmente introduzido pela empresa Tieto[1] na Dinamarca em 2009. *TaaS* se trata de um modelo de serviço em testes de software sob demanda, no qual os testes são executados em um ambiente baseado em nuvem, escalável e usando um Acordo de Nível de Serviço (ANS). Além disso, os *TaaS* permitem que usuários possam obter diferentes tipos de serviços de testes por meio do uso do *pay-as-you-test billing*, de forma que o custo possa ser compartilhado e consequentemente reduzido (GAO et al., 2013).

De acordo com esta definição, a DBB é uma plataforma de *TaaS* que viabiliza a execução de testes UI utilizando dispositivos reais, reduzindo os custos envolvidos na execução e aumentando a diversidade de dispositivos disponíveis para realização de testes. Como a abordagem se baseia no paradigma da EC, viabilizando o uso de dispositivos já habilitados, a redução de custos pode ser mais efetiva quando comparada com outras abordagens.

A Figura 3.1, apresenta uma visão geral da plataforma proposta. Dispositivos reais, conectados à Internet e cadastrados na plataforma, são convidados a aceitarem a execução de testes enviados por desenvolvedores/testadores. Uma vez aceitos, os testes são iniciados pelo proprietário do dispositivo de acordo com sua conveniência. Após a execução dos testes, relatórios de execução são enviados aos interessados. No ato do recebimento do relatório, os desenvolvedores/testadores são bilhetados, remunerando os usuários que cederam os dispositivos para que os testes fossem executados.

[1] Tieto - http://www.tieto.com/

Esta tese propõe uma plataforma de execução de testes distribuída utilizando a EC como meio de viabilizar a utilização de recursos pré-existentes, no entanto, o foco está em possibilitar que os testes possam ser executados de forma distribuída em dispositivos espalhados ao redor do mundo, sem a necessidade de conexão física entre os dispositivos e os *hosts* de desenvolvimento. Portanto, políticas de reembolso, faturamento, tipo de moeda, meios de pagamento e outros detalhes de operação ligadas aos mercados não são abordados nesta tese.

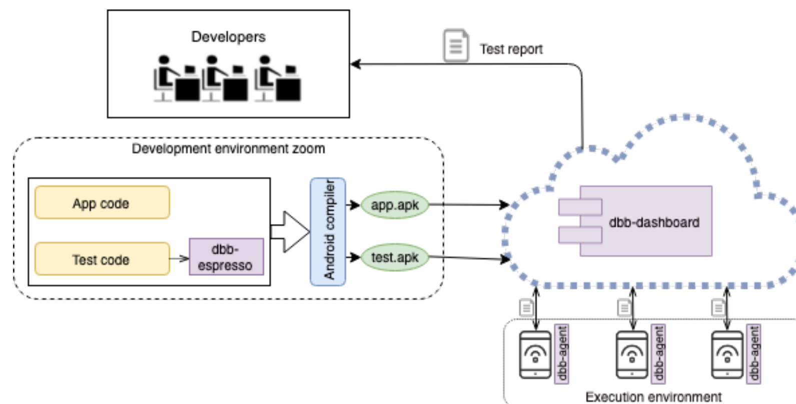


Figura 3.1: Visão geral da plataforma.

A plataforma é divida em três componentes principais, `dbb-dashboard`, `dbb-agent` e `dbb-espresso`. O papel de cada componente é descrito a seguir:

- **dbb-dashboard** - É por meio deste componente que o desenvolvedor/testador submete seus testes para serem executados nos dispositivos desejados. Além disso, é possível o acompanhamento do estado dos testes enviados: aceito, negado e reportado.
- **dbb-agent** - Este componente é um aplicativo Android, instalado pelos proprietários dos dispositivos e responsável por receber e executar os testes submetidos por meio do `dbb-dashboard`.
- **dbb-espresso** - Este componente é uma versão customizada do *Espresso* (Google Inc, 2013) que deve ser utilizada para compilação dos testes UI. Portanto, para que o desenvolvedor/testador possa utilizar a plataforma, os testes escritos precisam ser compilados por este componente.

3.1 Compilação dos testes

Para escrita de testes usando o *Espresso*, o desenvolvedor/testador necessita declarar, como dependência do projeto em questão, as bibliotecas necessárias para o funcionamento do *framework*. A Tabela 3.1 apresenta a relação de componentes necessários ao funcionamento do *Espresso*.

Tabela 3.1: Dependências necessárias ao funcionamento do Espresso

Dependência	Objetivo
espresso-core	APIs de acesso aos componentes de tela e checagem de resultados.
espresso-web	API de acesso a recursos WebView.
Android test rules	API para acesso, através de anotações, a Activity testada.
Android test runner	Componente a ser utilizado na execução dos testes.

Desde 2017, o *Espresso* passou a compôr a nova biblioteca de testes para Android. Denominada *Android Testing Support Library*[2] (Google, 2017), tal biblioteca é composta pelos componentes apresentados na Figura 3.2. Os componentes `espresso-intents`, `espresso-idling-resources`, `espresso-web`, `espresso-contrib` e `espresso-core` integram o *Espresso*, por meio deles, é possível a escrita de testes UI gerenciados pelo *framework*. Os demais componentes que integram a biblioteca, são utilizados para escrita de testes de unidade e integração, além de interagirem com o JUnit (por meio do *runner*) [3] para a execução dos testes. A partir da versão 2 do *Espresso*, o *runner* padrão utilizado nos projetos de testes é o `android.support.test.runner.AndroidJUnitRunner`.

Em função das demandas da plataforma proposta: 1) Geração de um relatório ao final da execução de cada caso de teste; e 2) Executar testes em dispositivos reais remotamente; foi necessário a reconstrução do *runner* padrão. As modificações foram realizadas de forma a contornar limitações arquiteturais do *framework* original. As limitações relacionadas à execução de testes automatizados em dispositivos físicos, se deve às questões consideradas na Seção 2.3.1, pois originalmente o *Espresso* é dependente de conexões `adb` para implantação e execução de testes em dispositivos reais.

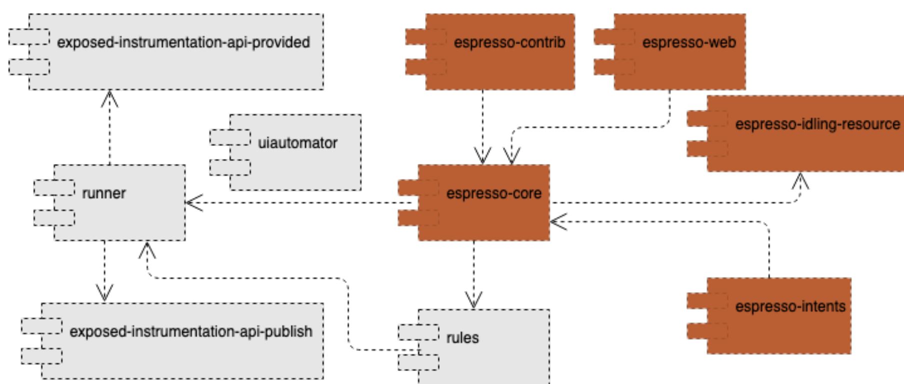


Figura 3.2: Principais componentes do Android Testing Support Library.

Esta evolução arquitetural no `android.support.test.runner.AndroidJUnitRunner` exige que os desenvolvedores/testadores, usuários da plataforma, compilem seus testes utilizando este novo componente. No momento da escrita desta tese, este se trata de

[2] Android Testing Support - código fonte disponível em: https://android.googlesource.com/platform/frameworks/testing
[3] Componente de execução de testes instrumentados

um arquivo `.jar` contendo todos os componentes da *Android Testing Support Library*. Apenas com a inclusão deste componente como dependência no projeto a ser testado, é possível executar testes localmente e remotamente, descartando a necessidade de alteração de configurações para realização de testes em emuladores e dispositivos reais conectados diretamente ao *host* de desenvolvimento.

Na próxima seção são apresentados os detalhes de como a plataforma executa remotamente os testes compilados.

3.2 Execução distribuída dos testes

Uma vez submetidos à plataforma, a partir do `dbb-dashboard`, os testes são encaminhados a todos os dispositivos que se encaixam nos modelos configurados pelo desenvolvedor/testador. Ao receber um teste, o proprietário do dispositivo deve sinalizar se aceita a execução do teste ou não. Para aceitar a execução dos testes, o proprietário do dispositivo deve instalar os dois pacotes, submetidos pelo desenvolvedor/testador, o `.apk` gerado para a aplicação a ser testada e o `.apk` gerado para os testes escritos.

Uma vez aceito o teste, o proprietário do dispositivo pode solicitar o início da execução de acordo com sua conveniência. Este procedimento é possível por meio do componente `dbb-agent`, instalado no dispositivo pelo seu proprietário.

Além de prover ao proprietário do dispositivo uma interface que permita a aceitação, instalação e acompanhamento de execuções já realizadas (para fins de bilhetagem), o `dbb-agent` é responsável por três papéis fundamentais:

Inicialização do testes - o `dbb-agent` invoca o *runner* existente no arquivo de testes compilado;

Recebimento dos resultados - o `dbb-agent` recebe o relatório de execuções gerado pelo *runner*;

Confirmação de solicitações de *RunTime* - o `dbb-agent` confirma todas as solicitações de *Runtime* exigidas pelo Android.

Ao solicitar a execução dos testes, o proprietário do dispositivo é avisado quanto a necessidade da não utilização do dispositivo durante o processo de execução dos testes, por isso o `dbb-agent` ativa o modo avião, para que chamadas de voz não ocorram durante o processo de execução. Finalmente o `dbb-agent` aciona o `runner` por meio da API disponível para inicialização de testes instrumentados.

Como a autorização de execução (realizada pelo proprietário do dispositivo) pode envolver vários testes de aplicativos diferentes, e os testes de um aplicativo não devem interferir nos testes de outro aplicativo, o `dbb-agent` organiza os testes a serem executados em uma fila, de acordo com a ordem de chegada dos testes à plataforma. Cada

teste é executado pelo `runner` e seu relatório é capturado pelo `dbb-agent` que o envia aos interessados.

A partir da API 23, o Android passou a exigir confirmações acerca da utilização de determinados recursos no momento em que o recurso é efetivamente utilizado, o que não ocorria em versões anteriores. Até a API 22, o Android observava o arquivo `AndroidManifest.xml` no momento de iniciar a aplicação invocada e então solicitava ao utilizador a autorização para acesso a todos os recursos declarados neste arquivo. A Listagem 3.1 apresenta um arquivo `AndroidManifest.xml` de uma aplicação disponível como exemplo para desenvolvedores na documentação do Android.

Listing 3.1: Exemplo de um arquivo Manifest.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3            package="com.example.android.testing.espresso.CustomMatcherSample">
4      <uses-sdk android:minSdkVersion="9" android:targetSdkVersion="26" />
5      <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
6      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
7      <application
8              android:icon="@drawable/ic_launcher"
9              android:label="@string/app_name"
10             android:theme="@style/AppTheme">
11         <activity
12                 android:name="com.example.android.testing.espresso.CustomMatcherSample.MainActivity"
13                 android:label="@string/app_name">
14             <intent-filter>
15                 <action android:name="android.intent.action.MAIN"/>
16                 <category android:name="android.intent.category.LAUNCHER"/>
17             </intent-filter>
18         </activity>
19     </application>
20 </manifest>
```

A habilidade de monitorar o estado da tela apresentada no dispositivo fora do contexto da aplicação instrumentada com testes não compete ao *Espresso*, pois como os testes são instrumentados, estes têm acesso apenas aos componentes da aplicação. Notificações de *runtime* renderizam componentes do próprio sistema operacional e portanto estão fora do alcance do *Espresso*. Para que a DBB seja capaz de responder a estas notificações, foi construído um mecanismo que faz uso do componente de Acessibilidade do Android[4]. Este mecanismo percorre todos os componentes renderizados na tela do dispositivo a cada mudança de estado detectado na tela. O mecanismo busca por componentes que satisfazem os seguintes critérios: pertencer a classe `android.widget.TextView` e propriedade `resource-id` igual a `com.android.packageinstaller:id/permission_message`. Estas propriedades juntas identificam quando o Android apresenta alguma notificação de *runtime* e podem ser observadas na Figura 3.3. Esta figura apresenta o momento em que o Android renderiza a tela de solicitação de permissão para leitura e escrita no sistema de armazenamento do dispositivo. Para que a aplicação continue funcionando normalmente, é necessário a

[4]Accessibility - Implementa recursos de acessibilidade para usuários portadores de deficiência

confirmação, no caso da DBB esta confirmação ocorre de forma automática, a partir do mecanismo que faz uso dos recursos de acessibilidade do Android.

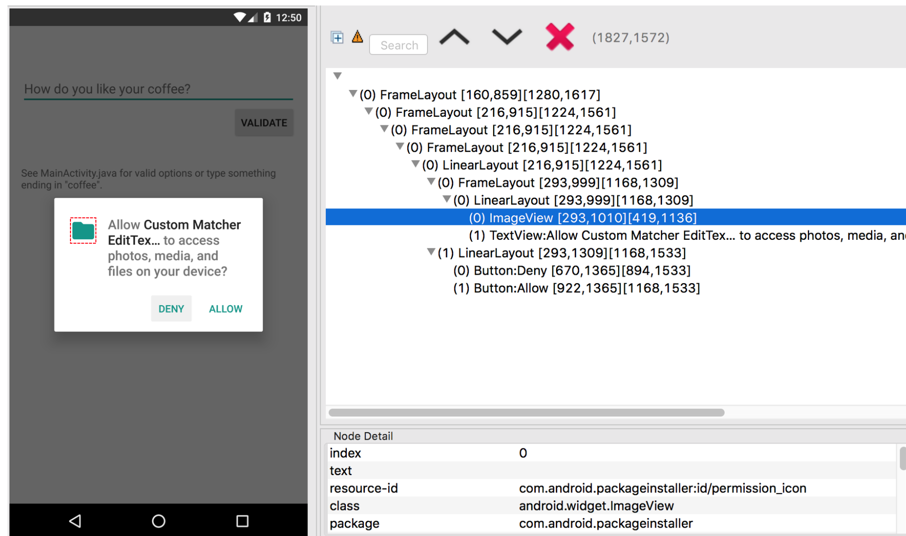


Figura 3.3: Componentes renderizados pelo Android, visualizados através do UiAutomator Viewer.

Portanto, para que o `dbb-agent` funcione normalmente, é necessário que o usuário autorize o uso dos recursos de acessibilidade explicitamente, esta permissão é solicitada pelo `dbb-agent` no momento da sua inicialização. O processo de inicialização do `dbb-agent` ocorre com descrito no Algoritmo 1.

Algoritmo 1 Inicialização do `dbb-agent`

1: **procedure** SUBSCRIBING
2: initializeDataBase()
3: $api \leftarrow getApiVersion()$ ▷ *obtendo a versão da api do dispositivo*
4: **if** $api \geq 23$ **then**
5: $am \leftarrow requireAccessibility()$
6: **if** $am == true$ **then**
7: exit();
8: loadMainActivity();

Primeiramente a base de dados é criada, em seguida a versão corrente da API do Android em execução no dispositivo é obtida, e caso a versão seja maior ou igual a versão 23 o `dbb-agent` solicita o acesso aos recursos de acessibilidade presentes no sistema operacional. Caso a permissão de acesso aos recursos de acessibilidade seja concedida, a tela inicial da aplicação é apresentada, possibilitando o login do usuário. Na seção 3.2.1 são apresentados os detalhes do registro de dispositivos na plataforma, habilitando-os para execução de testes.

3.2.1 Registro de dispositivos na plataforma

Para que dispositivos possam ser oferecidos na plataforma proposta, estes precisam ser registrados em alguma conta de usuário existente na plataforma. Desta forma,

além da criação de um usuário, o interessado em ceder o dispositivo para execução de testes deve instalar o componente `dbb-agent` e habilitá-lo em sua conta.

A habilitação do dispositivo, ocorre por meio da realização de login a partir do `dbb-agent`. Ao efetuar o login, o dispositivo é associado ao usuário, e passa a receber testes endereçados ao seu modelo. Um aspecto importante, é que um mesmo usuário pode habilitar vários dispositivos em sua conta. Isto se deve ao fato de que vários usuários, em especial de *smartphones*, possuem mais de um dispositivo, para uso em diferentes ambientes.

O Algomritmo 2 apresenta os passos executados para a realização de registros de dispositivos a serem disponibilizados pela plataforma.

Algoritmo 2 Registro de dispositivos na plataforma

```
1: procedure INIT(user, pass)
2:     active ← authenticate(user,pass)                    ▷ consumindo webservice
3:     if active == true then
4:         a ← getDeviceBrand()
5:         b ← getDeviceModel()
6:         c ← getBuildNumber()
7:         device ← registerDevice(a, b, c, user)          ▷ consumindo webservice
8:         persistDevice(device)
9:         openDashboard()
10:    else
11:        exit()
```

Para efetuar a autenticação do usuário, o `dbb-agent` consome uma interface *restfull* passando as credenciais do usuário (linha 2 do algoritmo). Visando a garantia da identidade de cada dispositivo, sua identificação é realizada por meio dos seguintes dados: fabricante (variável **a** do algoritmo), modelo (variável **b** do algoritmo), número de compilação (variável **c** do algoritmo) e o usuário proprietário. Esta identificação é devido à necessidade de troca de mensagens entre os componentes `dbb-dashboard` e `dbb-agent`. Após a obtenção das variáveis de identificação do dispositivo, estas são enviadas (linha 7 do algoritmo), por meio de uma interface *restfull*, para que o registro possa ser realizado pelo `dbb-dashboard`.

Após o registro, o `dbb-agent` grava a identidade do dispositivo em uma base de dados local. Esta identidade é utilizada para a criação de estruturas assíncronas de mensageiria utilizadas durante a comunicação entre o dispositivo e a plataforma. Essas estruturas são detalhadas na Subseção 3.2.2, que detalha o processo de distribuição dos testes submetidos à plataforma.

3.2.2 Distribuição dos testes

Após a compilação dos testes, os desenvolvedores/testadores submetem, por meio do `dbb-dashboard`, os arquivos `.apk` (da aplicação e dos testes) para serem executados em dispositivos previamente selecionados. Esta solicitação é capturada pelo

dbb-dashboard e encaminhada a todos os dispositivos que se encaixam nos modelos previamente configurados pelos desenvolvedores/testadores. A Figura 3.4 apresenta uma visão dos componentes envolvidos neste processo.

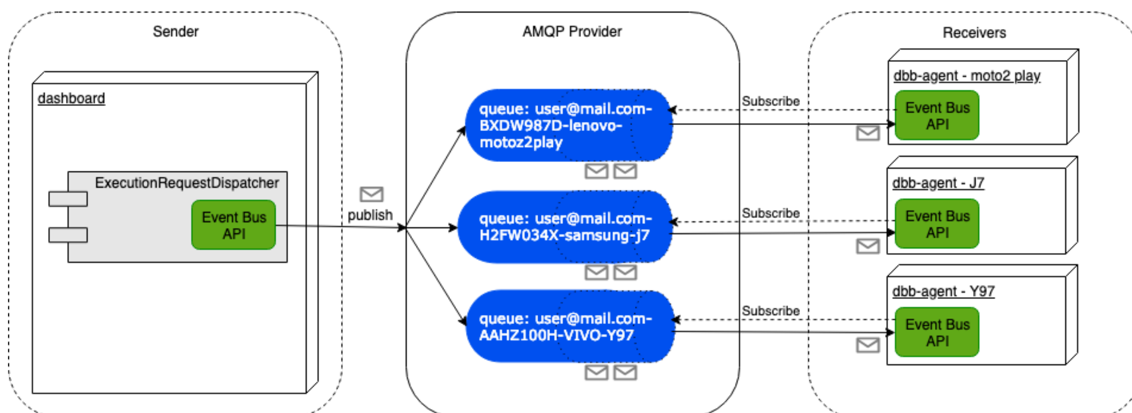


Figura 3.4: Distribuição dos testes submetidos a plataforma

O componente ExecutionRequestDispatcher, presente no dbb-dashboard, recebe as requisições de execução e se conecta como produtor de mensagens (*Message Sender*) ao provedor das filas AMQP [5]. Para cada dispositivo autenticado, uma fila AMQP é criada utilizando a identidade do dispositivo (apresentada na Subseção 3.2.1), isso ocorre por meio do dbb-agent instalado no dispositivo. Desta forma, requisições de execução de testes são enviadas como mensagens individuais para cada dispositivo, na medida em que os usuários iniciam o dbb-agent, estas mensagens são consumidas e o proprietário do dispositivo toma conhecimento das solicitações, podendo atendê-las ou negá-las.

Foi adotada a criação de uma fila AMQP para cada dispositivo para que os testes possam ser enviados a vários dispositivos do mesmo modelo, pois como se trata de uma plataforma baseada em EC, não existe a garantia de que os proprietários dos dispositivos endereçados na solicitação vão aceitar os testes enviados. O fato da solicitação chegar para vários dispositivos do mesmo modelo, aumenta a chance do teste ser executado nos dispositivos pretendidos.

Outros fatores que podem comprometer a atividade de testes são: **a) aceitação tardia dos testes -** os testes podem ser iniciados tardiamente pelo proprietário do dispositivo, fazendo com que o *deadline* para finalização dos testes seja excedido; **b) impossibilidade de envio do relatório -** neste caso os testes foram aceitos e finalizaram dentro do prazo, mas por problemas de comunicação não foi possível o envio do relatório aos interessados.

[5] Advanced Message Queuing Protocol - O AMQP é um protocolo de camada de aplicação, aberto, para *middlewares* orientados a mensagens

Estes riscos também são amenizados por meio do envio da solicitação de execução de testes para vários dispositivos do mesmo modelo, justificando a adoção de uma fila AMQP assíncrona para cada dispositivo. A escolha do protocolo AMQP foi motivada pelo fato de se tratar de um protocolo aberto, desenvolvido como alternativa ao MQTT[6].

Na Seção 3.3 são apresentados detalhes da geração dos relatórios de testes.

3.3 Relatório dos testes executados

Nesta seção são apresentados detalhes acerca dos relatórios gerados durante a execução dos testes enviados aos dispositivos. Foi reservada uma seção desta tese para o detalhamento do relatório, uma vez que é com base nele que o proprietário do dispositivo recebe sua recompensa e o desenvolvedor/testador têm o *feedback* que pretende, com custo reduzido.

Como descrito na Seção 3.2 as solicitações para execução de testes chegam aos dispositivos e precisam ser confirmadas pelo usuário. Esta confirmação é possível após a instalação da aplicação alvo e dos respectivos testes no dispositivo. Considerando que várias solicitações, envolvendo diferentes aplicativos e desenvolvedores/testadores, chegam ao mesmo tempo, a execução dos testes é realizada obedecendo a ordem de chagada das solicitações ao dispositivo. De acordo com a conveniência do usuário, os testes são iniciados e, para cada aplicação, o `dbb-agent` inicia um *listener* que recebe o relatório de execução dos testes. A Figura 3.5 ilustra este mecanismo.

O `TestRunner` cria um `listener` (que receberá o relatório de testes de uma *app* alvo) para cada solicitação de teste e executa cada um dos testes existentes para uma aplicação. Após o término da execução o `TestRunner` publica o relatório de testes que é consumido pelo `listener` correspondente à solicitação atendida. O `listener` armazena o relatório em uma base de dados local, para que este possa ser enviado aos interessados, tão logo exista conexão ativa do dispositivo com a Internet.

[6]Message Queue Telemetry Transport - O MQTT é um protocolo desenvolvido pela IBM, para middlewares orientados a mensagens

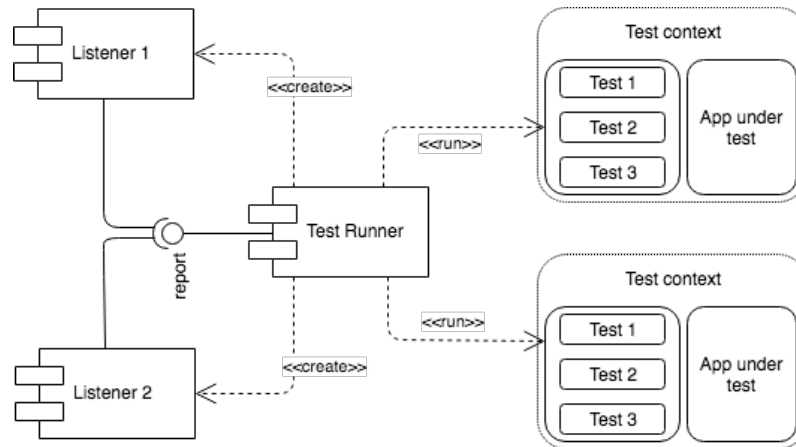


Figura 3.5: Obtenção dos relatórios gerados pelos testes

O relatório gerado se trata de um arquivo `JSON`, apresentado na Listagem 3.2, contendo os seguintes dados:

- modelo do dispositivo;
- número de casos de testes que falharam;
- lista contendo todos os casos de testes executados;
- tempo total para execução dos testes.

A lista contendo todos os casos de testes executados é composta pelo nome do caso de teste, um status (que informa se o teste passou ou falhou) e o *stacktrace* contendo informações técnicas sobre a falha.

Com base neste relatório, os interessados no resultado dos testes conseguem informações relevantes para identificação e reparação do *bug* que gerou a falha. No momento da escrita desta tese, a DBB não captura o estado da tela onde o erro foi gerado.

Listing 3.2: Relatório de execução dos testes


```
1 {"deviceModel": "motorola G5 plus",
2  "elapsedTime:":"28987"
3  "numFailures":"1",
4  "failures":
5  [{"testCase":"test_TC4(ivl.android.moneybalance.TC4)",
6    "failureMessage" : "android.support.test.espresso.NoMatchingViewException: No views in hierarchy found matching:
       (with id: ivl.android.moneybalance:id/new_calculation and with content description: is "New Calc" and is
       displayed on the screen to the user)
7    If the target view is not part of the view hierarchy, you may need to use Espresso.onData to load it from one of
       the following AdapterViews:android.widget.ListView{f5fc014 VFED.VCL. .F...... 0,0-1080,1680 #7f0b0057
       app:id/calculation_list}
8
9    View Hierarchy:
10       +>DecorView{id=-1, visibility=VISIBLE, width=1080, height=1920, has-focus=true, has-focusable=true, has-
           window-focus=true, is-clickable=false, is-enabled=true, is-focused=false, is-focusable=false, is-layout-
           requested=false, is-selected=false, root-is-layout-requested=false, has-input-connection=false, x=0.0, y
           =0.0, child-count=2}
11       |
12   +->LinearLayout{id=-1, visibility=VISIBLE, width=1080, height=1920, has-focus=true, has-focusable=true, has-
       window-focus=true, is-clickable=false, is-enabled=true, is-focused=false, is-focusable=false, is-layout-
       requested=false, is-selected=false, root-is-layout-requested=false, has-input-connection=false, x=0.0, y
       =0.0, child-count=2}
13   |
14   +-->ViewStub{id=16909450, res-name=action_mode_bar_stub, visibility=GONE, width=0, height=0, has-focus=false,
       has-focusable=false, has-window-focus=true, is-clickable=false, is-enabled=true, is-focused=false, is-
       focusable=false, is-layout-requested=true, is-selected=false, root-is-layout-requested=false, has-input-
       connection=false, x=0.0, y=0.0}
15   |
```

3.4 Considerações Finais

Este capítulo apresentou as modificações arquiteturais, realizadas no *Espresso*, necessárias para o funcionamento do modelo proposto, bem como os componentes construídos para o funcionamento da DBB (`dbb-dashboard`, `dbb-agent` e `dbb-espresso`). O modelo de envio de requisições de testes baseado no protocolo AMQP viabilizou a criação de filas individuais para dispositivos autenticados na plataforma, permitindo que mais de um dispositivo do mesmo modelo receba a mesma requisição de testes, maximizando sua chance de retorno.

O componente de confirmação de notificações de *runtime* é fundamental para que os testes possam ser executados remotamente, sem a intervenção do usuário, o que não ocorre na implementação original do Espresso, que exige a intervenção do testador mesmo que o dispositivo de testes esteja fisicamente conectado ao *host* de desenvolvimento.

No próximo capítulo, são apresentados os experimentos realizados em prol da validação da arquitetura proposta com o objetivo de verificar sua robustez na execução remota de testes UI.

CAPÍTULO 4

Experimentos e Resultados

Neste capítulo são apresentados os experimentos realizados em ambiente controlado utilizando a arquitetura proposta, a fim de provar sua robustez e efetividade na execução de testes UI de forma distribuída, em resposta à Questão de Pesquisa 1 (QP1 - É possível executar testes UI de forma eficaz em dispositivos reais geograficamente distribuídos?), apresentada na Seção 1.3.

Os experimentos envolveram voluntários que se cadastraram na plataforma e, seguindo um roteiro, registraram seus dispositivos Android na plataforma, portanto se colocando como proprietários de dispositivos.

Foram realizados dois experimentos envolvendo 23 diferentes modelos de dispositivos. No primeiro experimento foram utilizadas aplicações reais disponíveis em repositório de código aberto, com o intuito de validar a arquitetura acerca da construção e execução distribuída de testes UI utilizando a versão do *Espresso* gerada a partir das customizações arquiteturais realizadas neste estudo. No segundo experimento foram utilizados exemplos disponíveis no ambiente de treinamento do *Espresso* fornecido pela Google, com o objetivo de explorar diferentes componentes de tela e, desta forma, validando o modelo contra os componentes recomendados pelo fabricante em seus treinamentos.

Na próxima seção são apresentadas as aplicações utilizadas nos experimentos bem como os dispositivos utilizados e casos de teste construídos.

4.1 Detalhamento dos experimentos

Esta tese apresenta uma arquitetura para execução de testes de forma distribuída. Para que a validação da proposta pudesse ocorrer, foi necessária a escrita de testes automatizados utilizando a versão original do *Espresso*. Os experimentos foram divididos em dois: A) experimentos envolvendo aplicações reais; B) experimentos envolvendo exemplos do ambiente de treinamento do *Espresso*.

4.1.1 Experimento A

Para a realização do experimento **A** foram utilizadas as aplicações relacionadas na Tabela 4.1. A escolha destas aplicações foi baseada nos seguintes critérios:

- **Aplicações que sejam utilizadas -** foram consideradas aplicações com pelo menos 1000 downloads realizados;
- **Aplicações *open source* -** a escrita de testes instrumentados exige acesso ao código fonte da aplicação;
- **Aplicações nativas -** como se tratava do primeiro experimento, para validação da arquitetura, apenas este tipo de aplicação foi considerada;
- **Utilizam diferentes componentes UI -** Estas aplicações utilizam diferentes componentes e também diferentes notificações de *runtime*;
- **Compatibilidade com a IDE oficial -** Para simplificar a montagem do ambiente de experimentação, optou-se por projetos que foram construídos a partir do *Android Studio* [1].

Tabela 4.1: Lista de aplicativos utilizados no experimento A

	descrição	versão	avaliação	downloads
Money Balance [2]	Monitoramento de despesas pessoais	1.5	4.7	1K+
Quit Smoking [3]	Aplicativo para ajudar pessoas a pararem de fumar	1.5	4.7	1K+
Simple File Manager [4]	Explorador de arquivos	1.5	4.4	100K+

Foram escritos quatro casos de testes para cada uma das aplicações selecionadas. A definição do oráculo para a automatização dos testes se deu por meio da utilização prévia, dos participantes, de cada aplicação escolhida. Cada teste foi escrito em uma classe separada e recebeu um nome de acordo com a seguinte sintaxe: TC*i*, onde *i={1...4}*. Portanto cada aplicação possui os seguintes testes escritos TC1, TC2, TC3 e TC4. Todos estes testes podem ser acessados por meio de um repositório GIT [5] utilizado para compartilhamento dos experimentos.

As Tabelas 4.2, 4.3 e 4.4 apresentam os casos de testes escritos para cada uma das aplicações. Estes casos de testes foram construídos com base no *Espresso* 2 e seus objetivos são a verificação do comportamento de algumas interfaces gráficas existentes nas aplicações selecionadas, em prol de responder à questão de pesquisa 1 (QP1).

[1] *Android Studio* - Se trata da IDE oficial da Google para desenvolvimento de aplicativos para Android
[5] Uri dos projetos utilizados: https://github.com/kenyofaria/a-test2017

Tabela 4.2: Casos de testes escritos para a *app Money Balance*

Caso de teste	Objetivo	Pré-requisito	Passos	Resultado esperado
TC1	Verificar inicialização da app	A *app* deve ser iniciada		1. A *String MoneyBalance* deve constar na tela
TC2	Verificar a tela de cadastro de cálculos	A tela inicial da app deve estar ativa	2. Clicar no botão adicionar; 5. Clicar na caixa de texto *title*; 6. Digitar o texto *"test"* na caixa de texto *title*;	1. O botão de adicionar deve estar visível e habilitado; 3. A tela *New Calculation* é apresentada; 4. O campo *title* deve estar visível e habilitado; 7. O texto *"test"* deve ser apresentado no campo *title*;
TC3	Adicionar um novo cálculo	A tela inicial da app deve estar ativa	1. Clicar no botão para adicionar um cálculo; 3. Atribuir o título "calc 1"ao cálculo; 4. Escolher a moeda Vanuatu Vatu (VUV); 5. Atribuir duas pessoas aos cálculos; 6. Clicar no botão para salvar o cálculo.	2. A tela *New Calculation* deve ser apresentada; 7. Verificar se a *String "title 1"* é apresentada; 8. Duas pessoas adicionadas ao cálculo devem constar na lista.
TC4	Verificar a obrigatoriedade do registro de duas pessoas ao adicionar um cálculo	A tela inicial da app deve estar ativa	1. Clicar no botão para adicionar um cálculo; 3. Atribuir um título ao cálculo; 4. Escolher a moeda Vanuatu Vatu (VUV); 5. Clicar no botão para salvar o cálculo.	2. A tela *New Calculation* deve ser apresentada; 6. A mensagem "*Please enter at least 2 names.*"deve ser apresentada no campo para registro da segunda pessoa associada ao cálculo.

Tabela 4.3: Casos de testes escritos para a *app QuitSmoking*

Caso de teste	Objetivo	Pré-requisito	Passos	Resultado esperado
TC1	Verificar, na primeira inicialização da app, se todas as opções constam na tela *Settings*	A *app* deve ser iniciada		1. A *String Settings* deve estar visível; 2. As seguintes opções devem estar visíveis: *Overview tab, Health tab, Goal tab, Diary tab, Cigarettes, Date format, Date, Time, Money, Duration, Currency, Goal, Costs of your goal, Next goal, License, Changelog.*
TC2	Verificar o estado da configuração da opção *Overview tab*	A configuração inicial da aplicação deve ter sido feita (TC1)	1. Abrir a tela de configurações (*Settings*); 3. Desabilitar o item *Overview tab*; 5. Fechar a tela de configurações (*Settings*) 7. Abrir a tela de configurações (*Settings*) 9. Desabilitar o item *Overview tab*; 10. Fechar a tela de configurações (*Settings*);	2. O item Overview tab deve estar estar habilitado; 4. O item deve passar para estado desabilitado; 6. A tela deve omitir o conteúdo da tab *Overview*; 8. O item *Overview tab* deve estar desabilitado; 11. O conteúdo apresentado deve estar contido na tab *Health*
TC3	Realizar o resumo do dia	A configuração inicial da aplicação deve ter sido feita (TC1)	1. Abrir a aba *Diary*; 3. Incluir o resumo "Summary of day 1"; 5. Abrir a aba *Health*; 7. Voltar a aba *Diary*;	2. O conteúdo de *Diary* deve estar vazio; 4. O conteúdo *"Summary of day"* deve ser incluído; 6. O conteúdo da aba *Health* deve ser apresentado: 8. O resumo *"Summary of day"* deve continuar armazenado.
TC4	Criar um objetivo	A configuração inicial da aplicação deve ter sido feita (TC1)	1. Abrir a aba *Goal*; 3. Abrir a tela *Settings*; 4. Selecionar o item *Goal*; 6. Entre com a String *"goal 1"*; 7. Clique no botão OK; 9. Saia da tela *Settings*; 11. Seleciona a aba *Goal*.	2. Apresentar a aba *Goal* contendo a *String* "Configure in settings"; 5. Uma caixa de texto deve ser apresentada; 8. A tela *Settings* deve ser apresentada; 10. As abas devem ser apresentadas; 12. A *String "goal 1"* deve ser apresentada como conteúdo da aba *Goal*.

Tabela 4.4: Casos de testes escritos para a *app Simple File Manager*

Caso de teste	Objetivo	Pré-requisito	Passos	Resultado esperado
TC1	Verificar opções de menu	A tela principal deve estar aberta	1. Clique no menu de opções.	2. O menu de opções deve apresentar as opções: "Settings"e "About".
TC2	Verificar se o diretório DCIM do dispositivo foi mapeado	A tela principal deve estar aberta		1. O item "DCIM"deve estar visível.
TC3	Checar a navegabilidade	A tela principal deve estar aberta	2. Clique no item *DCIM*; 4. Pressionar *back button*;	1. A *String Home* deve constar na barra de migalhas; 3. A *String DCIM* deve constar na barra de migalhas; 5. Apenas a *String Home* deve constar na barra de migalhas.
TC4	Testar a criação de diretórios no DCIM	A tela principal deve estar aberta	1. Clique no item *DCIM*; 3. Clique no botão de adicionar itens; 6. Digite "*test directory one*"no nome do item; 7. Clique no botão "OK".	2. O botão de adicionar itens deve estar habilitado; 4. A caixa de diálogo deve ser apresentada; 5. O *Radio Button* deve estar indicando "*Directory*"; 8. O item "*test directory one*"deve constar na lista.

Como o objetivo da arquitetura proposta é a verificação da compatibilidade de aplicativos entre dispositivos utilizando testes UI, foram selecionados dispositivos de diferentes fabricantes, com diferentes configurações de tela e executando diferentes versões do Android. A relevância dessas variáveis no contexto dos testes UI é detalhada na Seção 1.1. Os dispositivos utilizados nos experimentos são apresentados na Tabela 4.5. Cada variável considerada nos dispositivos pode ser lida da seguinte forma:

API - 24 (Android 7 - Nougat); 23 (Android 6 - Marshmallow); 22 (Android 5.1 - Lollipop); 19 (Android 4.4 - KitKat);
Tamanho de tela - grande ($\geq$ 5.5 polegadas) e normal (4.5 polegadas);
Densidade de tela - hdpi (240 dpi), xhdpi (320 dpi);
Coordenadas - localização dos dispositivos no momento do experimento;
Tipo de conexão - tipo de conexão utilizado para realização do experimento.

Tabela 4.5: Dispositivos utilizados no experimento A

Fabricante	Modelo	API	Tamanho de tela	Densidade de tela	Coordenadas	Tipo de conexão
lenovo	A7010A48	23	grande	xhdpi	-16.662444, -49.366865	4G
motorola	MOTO G5 plus	24	grande	xhdpi	-16.662444, -49.366865	WIFI
motorola	MOTO G	22	normal	hdpi	-16.680987, -49.267629	4G
asus	ZENDFONE	19	normal	hdpi	-16.680987, -49.267629	WIFI
motorola	MOTO G4	24	grande	xxhdpi	-21.983190, -47.880267	4G
samsung	SM-A520F	24	grande	xhdpi	-21.983190, -47.880267	4G
lenovo	A2016B30	23	normal	hdpi	-21.983190, -47.880267	4G

A variável Coordenadas foi utilizada para demonstrar que os dispositivos estavam geograficamente distribuídos, cenário básico para funcionamento da arquitetura. A variável Tipo de conexão foi apresentada para verificação do funcionamento (recebimento de testes e envio de relatório de testes) do componente `dbb-agent`, apresentado no Capítulo 3, independentemente do tipo de conexão utilizada.

Os casos de teste foram escritos por três perfis diferentes de desenvolvedores, júnior, pleno e sênior, sendo que apenas o perfil sênior tinha conhecimento da automatização de testes utilizando o *Espresso*. Após a construção dos testes e sua verificação de funcionamento utilizando emuladores, os testes foram submetidos à plataforma para serem executados nos dispositivos disponíveis. Cada dispositivo recebeu testes envolvendo todas as três aplicações, portanto, foram executados 12 casos de teste em cada dispositivo e os resultados são apresentados na Seção 4.2.

A fim de aumentar a diversidade de dispositivos e cenários envolvidos em casos de testes reais, foi realizado o experimento **B** com a participação de voluntários (como proprietários de dispositivos) durante uma palestra. Este experimento é detalhado na próxima seção.

4.1.2 Experimento B

Neste experimento foram utilizadas aplicações existentes no ambiente de treinamento oficial do *Espresso*[6], essas aplicações estão relacionadas na Tabela 4.6. A utilização desses exemplos foi motivada pela necessidade de explorar diferentes cenários para validação da arquitetura proposta, especialmente no que concerne o `TestRunner`, pois diferentes componentes são exercitados.

Tabela 4.6: Aplicações utilizadas no experimento B

Aplicação	Descrição
Basic Sample	Exemplo básico contendo operação em um *input* de texto
Custom Matcher Sample	Exemplo que explora a identificação de propriedades *hint* de um input
Data Adapter Sample	Exemplo que explora operações em `AdapterViews`
Intents Advanced Sample	Exemplo que simula a busca de um bitmap usando a câmera do dispositivo
Intents Basic Sample	Exemplo que explora cenários baseados em intenções
Web Basic Sample	Exemplo de um teste que explora uma aplicação WebView

Os casos de teste desse experimento não são descritos nesta tese, pois estão disponíveis no ambiente de treinamento oficial do *Espresso*, e tem o objetivo de apresentar como o *framework* deve ser utilizado com cada um dos componentes de interface gráfica disponível na API do Android. Os dispositivos utilizados são apresentados na Tabela 4.7.

[6] samples - https://developer.android.com/training/testing/samples

Tabela 4.7: Dispositivos utilizados no experimento B

Fabricante	Modelo	API	Tamanho de tela	Densidade de tela	Coordenadas	Tipo de conexão
motorola	XT1225	23	Normal	xxhdpi	-16.666074, -49.255043	4G
motorola	Moto G3	23	Normal	hdpi	-16.666074, -49.255043	4G
motorola	moto G4	24	Grande	xhdpi	-16.666074, -49.255043	4G
motorola	moto G5	24	Normal	xhdpi	-16.666074, -49.255043	4G
motorola	moto Z2	26	Grande	xxhdpi	-16.666074, -49.255043	4G
LG	LG-H879I	24	Normal	xhdpi	-16.666074, -49.255043	4G
LG	LG-K130	22	Normal	mdpi	-16.666074, -49.255043	4G
LG	LG-M250	24	Normal	hdpi	-16.666074, -49.255043	4G
LG	Nexus 5	23	Normal	xxhdpi	-16.666074, -49.255043	4G
SAMSUNG	SM-J500M	23	Normal	hdpi	-16.666074, -49.255043	4G
SAMSUNG	SM-J700M	23	Grande	hdpi	-16.666074, -49.255043	4G
SAMSUNG	SM-N9005	21	Grande	xxhdpi	-16.666074, -49.255043	4G
SAMSUNG	SM-G925I	24	Normal	xxxhdpi	-16.666074, -49.255043	4G
SAMSUNG	SM-G610M	24	Grande	xhdpi	-16.666074, -49.255043	4G
SAMSUNG	SM-G935F	24	Grande	xxxhdpi	-16.666074, -49.255043	4G
ASUS	ASUS_Z017DC	24	Normal	xhdpi	-16.666074, -49.255043	4G

Os casos de teste foram obtidos e compilados utilizando a versão customizada do Espresso (`dbb-espresso`). Então os testes foram submetidos, por meio do `dbb-dashboard`, aos dispositivos disponíveis (36 dispositivos mas com 16 modelos diferentes), de forma que todos os casos de teste fossem executados.

Na próxima seção são apresentados os resultados obtidos nos experimentos **A** e **B**, evidenciando a efetividade da arquitetura proposta, respondendo a QP1.

4.2 Resultados

Dois experimentos foram realizados a fim de responder a QP1 (É possível executar testes UI de forma eficaz em dispositivos reais geograficamente distribuídos?). O experimento **A**, baseado em aplicações reais disponíveis no repositório F-Droid[7], foi concebido para validar a arquitetura acerca da construção de testes UI. Os testes executados neste experimento foram organizados de forma que cada caso de teste fosse escrito em uma classe separada e explorasse recursos reais das aplicações especialmente as confirmações das notificações de *runtime*, presentes em algumas das aplicações em diferentes momentos.

As aplicações *Money Balance* e *Simple File Manager* solicitam acesso de leitura e escrita no dispositivo no momento em que são iniciadas, ou seja, quando a principal `Activity` da *app* alvo é aberta pelo Android, e o *QuitSmoking* realiza a solicitação de leitura e escrita somente após o usuário realizar a primeira configuração, a partir de uma das telas da aplicação. Em todos os dispositivos cuja versão da API era 23 ou

[7] Uri do repositório F-Droid: https://f-droid.org/en/

superior, o componente `dbb-agent` realizou a confirmação da permissão, viabilizando a execução dos testes sem a intervenção humana, fundamental para execução remota. Nos dispositivos configurados com as versões 19 ou 22, a solicitação de acesso aos recursos de armazenamento do Android foram dadas no ato da instalação de cada aplicação, sendo necessária apenas a inicialização dos testes pelo componente `dbb-agent`.

Os resultados do experimento foram apresentados no *11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (FARIA; FREITAS; VINCENZI, 2017).

O experimento **A** foi elaborado a partir de aplicações reais com testes escritos por colaboradores. A fim de eliminar o viés das limitações (relacionadas ao uso do *Espresso*) dos colaboradores do experimento **A**, o experimento **B** foi elaborado para verificar o comportamento do componente `dbb-expresso`, frente aos testes disponíveis no ambiente de treinamento oficial do *Espresso*, disponibilizado pelo seu fabricante. Neste experimento, os casos de teste de cada aplicação foram escritos em uma só classe de testes, demonstrando a capacidade da arquitetura em lidar com este tipo de organização. Embora a quantidade e diversidade de dispositivos tenha sido maior do que no experimento **A**, todos os testes passaram.

Além da execução dos testes em dispositivos reais de forma remota, os mesmos testes foram executados utilizando a arquitetura padrão do *Espresso*, ou seja, utilizando os quatro primeiros dispositivos da Tabela 4.5 conectados diretamente ao *host* de desenvolvimento. Foram utilizados apenas os quatro primeiros dispositivos devido ao fato de serem os únicos aos quais tivemos acesso físico. Os relatórios de execução utilizando a arquitetura padrão do *Espresso*, bem como a execução da DBB estão disponíveis no nosso ambiente no GitHub[8].

4.3 Considerações Finais

Embora nenhum problema de compatibilidade das aplicações testadas tenha sido identificado nos experimentos, pois a busca por incompatibilidades requer uma grande diversidade de dispositivos e em geral uma grande quantidade de casos de testes escritos, pôde-se verificar a efetividade da arquitetura construída, em especial o componente `Test Runner`, responsável por executar os testes a partir de um dispositivo sem a necessidade do `adb`.

No próximo capítulo, é apresentada de forma analítica a robustez acerca da escalabilidade proporcionada pelo modelo apresentado, além de comparar o custo desta solução com os principais provedores de *TaaS* existentes na indústria.

[8]https://github.com/kenyofaria/dbb

CAPÍTULO 5

Análise da plataforma proposta

Este capítulo é dedicado a uma análise da arquitetura proposta comparando-a com as soluções existentes no mercado que têm o objetivo de prover dispositivos reais para serem utilizados na execução de testes de aplicações móveis. Esta análise foi realizada com o intuito de responder às questões de pesquisa 2 e 3 (QP 2: O uso da Economia Colaborativa promove a escalabilidade de testes UI em ambientes fragmentados?; QP 3: A Economia Colaborativa contribui para a redução do custo financeiro do processo de execução de testes de aplicativos que são executados em ambientes fragmentados?), introduzidas na Seção 1.3. Primeiramente foi realizada uma análise comparativa acerca da escalabilidade da solução proposta e em seguida uma comparação dos custos com abordagens existentes.

5.1 Análise comparativa da escalabilidade e cobertura

Nesta análise foram considerados os maiores provedores em nuvem destinados à realização de testes em aplicações móveis: Google Firebase Test Cloud (Google Firebase, 2018), AWS Device Farm (AMAZON, 2018a), Xamarin Test Cloud (XAMARIN, 2018), Kobiton (KOBITON, 2018), e Perfecto (PERFECTO, 2018). Ao obter informações sobre a disponibilidade de dispositivos nessas plataformas, pode-se concluir que os provedores de nuvem de teste não fornecem uma plataforma escalável, pois fornecem uma baixa quantidade de dispositivos que podem ser utilizados ao mesmo tempo, e não fornecem uma boa cobertura em relação à diversidade de dispositivos Android existentes. A Tabela 5.1 mostra um resumo da infraestrutura disponibilizada por cada provedor.

Para utilização destes dispositivos, os provedores disponibilizam dois modelos de contrato, sob demanda, em que o usuário paga de acordo com o seu uso, e por reserva, em que o usuário paga por um recurso previamente definido. No modelo sob demanda, o custo varia conforme a demanda, de forma que não é possível um controle dos gastos com infraestrutura, por outro lado, no modelo baseado em reserva existe o risco dos recursos serem superestimados, causando impacto negativo para execução dos testes.

Tabela 5.1: Escalabilidade dos players de dispositivos móveis em nuvem

Platforma	Número de dispositivos disponíveis	Diversidade de dispositivos	Modelo de contrato	Dispositivos concorrentes
Google Firebase Test Cloud	nd	69	on-demand only	devices on catalog
AWS Device Farm	1000	191	on-demand and reservation	5 devices
Xamarin Test Cloud	2000	250	reservation	30 devices
Perfecto	10000	62	reservation	3 device
Kobiton	300	95	reservation	1 device

Buscando responder a QP2, esta análise foi realizada considerando quatro variáveis que impactam diretamente na escalabilidade dos serviços baseados em nuvem que oferecem dispositivos móveis, e na cobertura de dispositivos que possuem maior participação no mercado: número de dispositivos disponíveis, diversidade de modelos, modelo de contrato e o uso concorrente de dispositivos. O número de dispositivos disponíveis afeta diretamente a escalabilidade, uma vez que um dispositivo não pode ser utilizado por mais de um usuário simultaneamente. A diversidade de dispositivos é considerada porque quanto maior, maior tende a ser a cobertura do mercado. A limitação do uso simultâneo de dispositivos é usado por alguns provedores para evitar a indisponibilidade de dispositivos devido à baixa diversidade provida e, em alguns casos, a um número incipiente de dispositivos físicos existentes. O modelo de contrato é considerado pelo seu impacto direto no custo (abordado em detalhes na próxima seção). Os resultados são apresentados na Tabela 5.1. Os dados foram recuperados do catálogo de dispositivos fornecido por cada um dos provedores no momento da escrita desta tese.

O fato dos provedores de testes de aplicativos móveis baseados em nuvem precisarem adquirir os dispositivos para então provê-los em sua infraestrutura, deve-se ao fato de os dispositivos móveis não poderem ser usados em uma base compartilhada devido a limitações técnicas, impostas pelo próprio sistema operacional, além da dependência do `adb` descrita no Capítulo 2. Dessa forma, para que esses provedores possam suportar, por exemplo, 10 desenvolvedores/testadores rodando testes simultaneamente, é necessário possuir pelo menos 10 dispositivos físicos respondendo às solicitações de cada usuário. Este modelo não é escalável. Tomando como exemplo a plataforma *Kobiton cloud* (KOBITON, 2018), que no momento da escrita desta tese não suportava mais do que 300 testadores executando testes simultaneamente devido à quantidade de dispositivos disponíveis.

Mesmo quando o provedor possui um grande número de dispositivos, ele pode entregar baixa diversidade. Por exemplo, a plataforma *Perfecto test cloud* (PERFECTO, 2018) que possui o maior número de dispositivos. Apesar de existirem 10 mil dispositivos

disponíveis para utilização, apenas 62 diferentes modelos são oferecidos. Esta característica contribui para a formação de filas para uso de modelos com grande fatia de mercado como é o caso do modelo *Samsung S6*, que neste momento possui a maior fatia de mercado de dispositivos Android. Essa limitação é contornada por meio de um controle no número de dispositivos simultâneos permitidos, forçando o desenvolvedor/testador a executar seus testes em um número limitado de diferentes modelos ao mesmo tempo. Esta política pode ser vista nas plataformas *AWS Device Farm* (AMAZON, 2018a), no *Xamarin* (XAMARIN, 2018), *Perfecto* (PERFECTO, 2018) e *Kobiton* (KOBITON, 2018).

Mesmo que os provedores apresentados suportem a demanda existente gerada por desenvolvedores/testadores com relação à disponibilidade de dispositivos, outro problema a ser abordado pela questão de pesquisa 2 (QP2) é a cobertura. Considerando um relatório fornecido pelo Open Signal em 2015 (OPENSIGNAL, 2015), naquele momento havia mais de 24 milhões de dispositivos Android diferentes e ativos ao redor do mundo, ou seja, o catálogo fornecido pelos provedores na Tabela 5.1 não proporciona aos desenvolvedores/testadores uma boa verificação de compatibilidade, mesmo que tenham dispositivos com a maior participação de mercado em seus catálogos, pois, existem aplicativos que exigem validação em dispositivos com baixa participação de mercado.

De acordo com o IDC, os dispositivos chineses representavam 40% do mercado global no terceiro trimestre de 2018 (IDC Inc, 2018), estes dados são apresentados na Tabela 5.2 em valores percentuais e na Tabela 5.3 em valores absolutos. Os principais fabricantes chineses são representados por Huwaei (Huawei Inc, 2018), Xiaomi (Xiaomi Inc, 2018), OPPO (OPPO Inc, 2018) e VIVO (Vivo Inc, 2018). A Figura 5.1 mostra a fatia de mercado dos principais modelos de dispositivos chineses. Apesar dessa importante participação de mercado, no momento da escrita desta tese, apenas o modelo XIOMI RedMI Note 4X é fornecido por um dos provedores apresentados na Tabela 5.1. A baixa diversidade de dispositivos é um importante problema causado por um modelo baseado em aquisições que não escala a um custo razoável, afetando diretamente a cobertura do mercado de dispositivos Android, impondo aos testadores a necessidade de adquirir dispositivos não encontrados nas plataformas ou mesmo validar seus aplicativos usando emuladores.

A arquitetura proposta nesta tese, validada por meio da plataforma DBB, apresenta condições para fornecer uma plataforma escalável mesmo considerando ambientes com alta fragmentação. Contudo, a arquitetura proposta requer maiores testes de escalabiliade. A estratégia adotada faz uso de dispositivos ociosos ao redor do mundo, a fim de fornecer, para desenvolvedores/testadores, um grande número de dispositivos bem como uma grande variedade de modelos. Na próxima seção, é apresentada uma análise do modelo proposto nesta tese, em uma perspectiva de custo, em conjunto com os grandes provedores apresentados nesta seção.

Tabela 5.2: Fatia de mercado (%) por fabricante (COUNTERPOINTER, 2018)

	2017 Q3	2017 Q4	2018 Q1	2018 Q2	2018 Q3
Samsung	21%	18%	22%	20%	19%
Huawei	10%	10%	11%	15%	14%
Apple	12%	18%	14%	11%	12%
Xiaomi	7%	7%	8%	9%	9%
Oppo	8%	7%	7%	8%	9%
vivo	7%	6%	5%	7%	8%
Motorola	3%	3%	2%	2%	3%
Outros	32%	31%	31%	28%	26%

Tabela 5.3: Fatia de mercado (milhões de unidades) por fabricante (COUNTERPOINTER, 2018)

	2017 Q3	2017 Q4	2018 Q1	2018 Q2	2018 Q3
Samsung	83.3	74.4	78.2	71.5	72.3
Huawei	39.1	41	39.3	54.2	52
Apple	46.7	77.3	52.2	41.3	46.9
Xiaomi	28.5	31	28.1	32	33.3
Oppo	32.5	30.7	24.2	29.6	33.9
vivo	28.6	24	18.9	26.5	30.5
Motorola	12.5	10.9	7.6	8.3	10.6
Outros	124.8	133.3	112.1	100.3	99.9

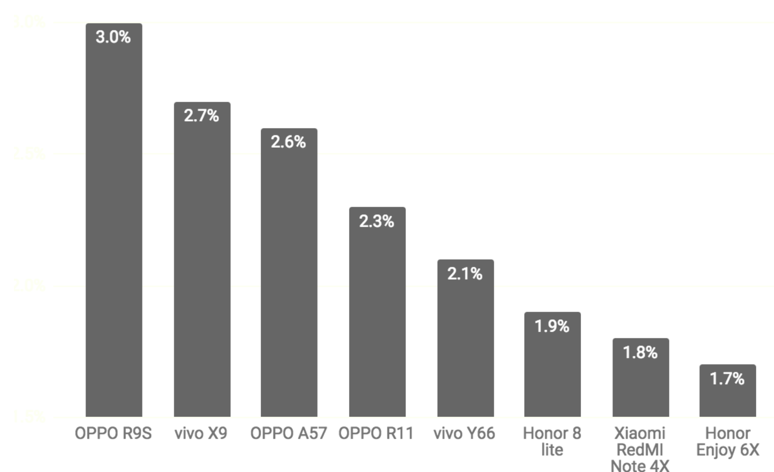


Figura 5.1: Participação dos modelos chineses no mercado global (COUNTERPOINTER, 2018).

5.2 Avaliação do custo

Na seção anterior, destacou-se a vantagem competitiva gerada pelo número potencialmente grande de dispositivos acessíveis por meio da DBB, que incorpora o uso do paradigma da Economia Colaborativa. No entanto, é necessário avaliar essa vantagem

Tabela 5.4: Preços nos principais provedores de nuvem e limite de dispositivos-hora incluídos no preço.

Plataforma	Categoria	Modelo de faturamento	Preço	DH
AWS Device Farm	Sob demanda	$0.17 por minuto	$10.2	1
	Reserva	$250 mensal	$250	730
Firebase Test Lab	Sob demanda	$5 por hora	$5	1
Xamarin Test Cloud	Reserva	$99 mensal	$99	30
Kobiton	Sob demanda	$0.1 por minuto	$6	1
Perfecto	Reserva	$129 mensal	$129	5

do ponto de vista econômico, conforme destacado pela questão de pesquisa 3 (QP3 - A Economia Colaborativa contribui para a redução do custo financeiro do processo de execução de testes de aplicativos que são executados em ambientes fragmentados?). Com isso em mente, realiza-se um estudo de viabilidade econômica sobre o uso da DBB em comparação com soluções alternativas, disponibilizadas por grandes plataformas baseadas em nuvem voltadas à execução de testes de aplicações móveis.

Esta análise foi conduzida presumindo que um usuário testador deseja realizar testes UI em diversos dispositivos diferentes. Estes testes duram uma hora. O custo esperado para realizar essa tarefa nas plataformas baseadas em nuvem consideradas neste trabalho, bem como o limite de dispositivos-hora (DH) incluído em tal preço, são descritos na Tabela 5.4.

Para manter a simplicidade da análise, foi usado apenas a categoria de modelo de faturamento mais adequada para pequenos testadores (exceto AWS devido à sua alta adoção). Na Tabela 5.4 é definido o preço como a despesa mínima que o usuário deve pagar para usar 1 DH, embora possa usar mais dispositivos sem aumentos de preços em alguns casos (conforme mostrado na tabela).

Como pode ser visto, não há padrão no modelo de faturamento e o usuário tem que enfrentar uma diferença significativa no preço. Além disso, mesmo alguns provedores adotando um modelo de cobrança mensal, há um limite de horas a ser usado, significando que, na verdade, o contrato não é para um mês completo (exceto AWS[1]). Outro problema é que os modelos de faturamento adotados por esses provedores dificultam a realização de testes em pequena escala. Por exemplo, o uso de provedores com cobrança mensal está sujeito ao pagamento do preço total. Essa política torna as alternativas atraentes apenas quando o número de DH está próximo do limite permitido.

Diante das limitações destacadas, foi estimado o preço do uso de um dispositivo por meio da solução proposta nesta tese. É usado um modelo de faturamento por hora, que é calculado pela equação:

[1] O número de DH é estimado em 730 horas. A mesma suposição é usada no restante deste capítulo.

$$\text{DBB pricing} = \frac{I}{\sum r} + d + p \qquad (5\text{-}1)$$

Onde I é o custo de implantação da plataforma para cada hora, r é o número de solicitações simultâneas (número de dispositivos entregando relatórios de testes), d é o custo de recompensa do dispositivo e p é a receita obtida pela plataforma para que esta se torne viável. Para definir I, são assumidas as despesas para um ciclo de vida de três anos usando a abordagem intitulada *Total Cost of Ownership* (TCO) (MARTENS; WALTERBUSCH; TEUTEBERG, 2012), pois fornece suporte de decisão confiável (ELLRAM; SIFERD, 1998). Além disso, foram incluídos os seguintes componentes de custo:

- **Despesas de capital (CapEx)** - Novas compras de infraestrutura e alocação de novos *data centers*;
- **Despesas operacionais (OpEx)** - Atividades necessárias para instalar, configurar e manter a plataforma em execução;
- **Custo indireto do negócio (Ind)** - O potencial impacto do tempo de inatividade na produtividade, mais os benefícios de tempo de mercado de maior agilidade.

A análise desses componentes de custo foi realizada considerando cinco alternativas de ambiente:

1. *On-premises* (ONP), os equipamentos são próprios e se localizam em local próprio;
2. *Colocatio*n (COL), os equipamentos são próprios e se localizam em espaço alugado;
3. Terceirização da estrutura necessária ao funcionamento dos serviços para a nuvem da Amazon (AWS) (Amazon, 2018);
4. Terceirização para a nuvem Azure (AZU) (Microsoft, 2018);
5. Terceirização para a nuvem da Google (GCP) (Google, 2018).

Estes cenários foram definimos visando à análise de modelos de infraestrutura privada e modelos de terceirização usando os fornecedores de nuvem líderes de mercado, de acordo com o último relatório do Gartner[2] sobre infraestrutura de nuvem como um serviço (LYDIA et al., 2017). A Tabela 5.5 descreve os componentes de custo e apresenta os resultados.

Para calcular as despesas de capital, primeiro foram definidos requisitos de hardware com base nas recomendações de planejamento de capacidade (ORACLE, 2007), uma vez que todos os serviços construídos para o funcionamento do modelo proposto são baseados em Java. Para o atendimento de 1000 solicitações simultâneas são recomendadas quatro máquinas com *CPU dual core*, com *2GB* de memória RAM na camada de aplicação, respondendo à requisições recebidas por um *load balancer* e uma máquina com

[2]Gartner: https://www.gartner.com

Tabela 5.5: TCO da infraestrutura necessário para um período de 3 anos.

Expense	gastos em 3 anos				
	ONP	**COL**	**AWS**	**AZU**	**GCP**
Capital Expenses					
Server Infra.	$48,623.62	$48,623.62	-	-	-
Storage	$1,814.13	$1,814.13	-	-	-
Backup	$38.16	$38.16	-	-	-
Networking/Security	$13,127.81	$28,629.04	-	-	-
Total: 3 anos	$63,603.72	$79,104.95	$0	$0	$0
Operating Expenses					
Personnel	$403,491	$403,491	$100,872.75	$100,872.75	$100,872.75
Server Maintenance	$58,113.91	$58,113.91			
Software Licensing	$10,255.39	$10,255.39	$55,143.67	$46,183.56	$52,059.91
Space/Power	$102,553.96	$203,323.75			
Total: 3 anos	$574,417.28	$675,184.05	$156,016.42	$147,056.31	$152,932.66
Indirect Costs					
Estimated loss due to productivity	$16,293.6	$16,293.6	$162.94	$814.68	$162.94
Estimated revenue lost due to delays	$287,784	$287,784	$82,224	$82,224	$82,224
Total: 3 anos	$304,077.6	$304,077.6	$82,836.94	$83,038.68	$82,836.94
TCO	$942,098.6	$1,058,366.6	$238,853.36	$230,095.29	$235,769.6

CPU contendo 16 núcleos, com *16GB* de memória RAM e *256GB* de armazenamento, para camada de banco de dados. Para as soluções ONP e COL, foram realizadas suposições utilizando a calculadora do AWS TCO (AMAZON, 2018b). O preço é baseado nos custos estimados de equipamentos de fornecedores de infraestrutura global usando São Paulo, Brasil como local. As opções de nuvem não exigem despesas de capital, pois nenhum equipamento precisa ser comprado.

As despesas operacionais não relacionadas a pessoal, para os cenários ONP e COL, também foram estimadas usando a calculadora AWS TCO, usando um modelo baseado em serviço para licenciamento de software. Para os cenários baseado em nuvem, foram utilizados os tipos *VM c3.large* e *m4.xlarge*, para AWS; e *Standard_A2* e *Standard_F16*, para o AZU, uma vez que correspondem de perto às necessidades impostas pela demanda estimada para desenvolvimento da comparação de custos. Para o GCP, utilizam-se tipos de VM personalizados com as configurações sugeridas pela Oracle. A fim de evitar o favorecimento da DBB, foi usado o preço sob demanda para estes recursos de VM, que é maior se comparado com o modelo de reserva.

Os custos com pessoal são a maior despesa operacional. Estes custos foram calculados para os cenários ONP e COL, incluindo salários e benefícios de um gerente de infraestrutura em tempo integral. Para estimar esse custo, foi utilizado o salário médio de profissionais de infraestrutura no Brasil (Robert Half, 2018). Um dos principais benefícios dos cenários de nuvem é a menor equipe interna de pessoal necessária para a administração da plataforma. Por esse motivo, assumiu-se que o funcionário é alocado apenas 25% do tempo no gerenciamento da infraestrutura da DBB.

Despesas de capital e despesas operacionais não são os únicos aspectos a considerar ao avaliar o TCO. Vários custos indiretos afetam os negócios quando a infraestrutura é submetida a um tempo de inatividade ou leva mais tempo para trazer

os recursos para suportar uma nova oportunidade de receita. Embora estes custos sejam difíceis de medir, foi incluído um conjunto de suposições para demonstrar o impacto. A DBB assume 99% de tempo de atividade para os cenários ONP e COL (87.6 horas de tempo de inatividade anual não planejado). Para AWS e GCP, os níveis de serviço de disponibilidade do sistema são garantidos para 99.99% (0.876 horas de tempo de inatividade anual não planejado) (AWS, 2018; Google Cloud, 2018). Para a AZU, os níveis de serviço de disponibilidade do sistema são garantidos a 99.95% (tempo de inatividade não planejado de 4.38 horas) (Microsoft Azure, 2018). Estes períodos de indisponibilidade foram calculados considerando o período de 1 ano. Para modelar a perca de produtividade no trabalho, estimou-se que 20 funcionários, com um salário médio de $62 por hora, poderiam ser impactados pelo tempo de inatividade da solução diretamente como equipe de TI ou indiretamente em outras funções operacionais exigidas para o funcionamento da plataforma. Assumiu-se que não há impacto significativo sobre a produtividade da mão-de-obra decorrente do tempo de inatividade não planejado, estabelecendo um fator de ponderação de 5%.

Como este modelo se baseia no paradigma da EC, existe um risco de alta adesão por parte de interessados em ceder seus dispositivos para realização de testes. Portanto a redução dos riscos de mercado oriundos dos picos de utilização é uma preocupação. Este modelo adota uma receita anual bruta de $1M e, de maneira conservadora um crescimento anual de 5%. Para os cenários ONP e COL, foi adotado um tempo médio para obtenção de infraestrutura adicional de 45 dias por ano e uma média de 25 dias por ano para implanta-la. Nos cenários baseados em *cloud*, não existe este *overhead* pela infraestrutura, uma vez que uma série de responsabilidades são terceirizadas para o provedor contratado. No entanto, foi definido um tempo médio para construção e implantação dos recursos em nuvem de 20 dias por ano.

Como pode ser visto, a melhor alternativa para disponibilização da DBB (`dbb-dashboard`) é a infraestrutura Azure. Portanto, foi usada esta opção para estimativa do custo desta plataforma. O segundo componente na estimativa de custo é o custo de recompensa do dispositivo. Nesta análise, a recompensa do dispositivo é proporcional ao tempo de execução de testes. Mais precisamente, a recompensa foi configurada como custo para aquisição de um dispositivo dividido pelo número de horas em um ano. Desta forma, se o proprietário do dispositivo disponibiliza-o para ser utilizado na DBB 24 horas por dia, sete dias por semana, após um ano o proprietário é capaz de adquirir outro dispositivo do mesmo preço[3]. No entanto, embora isso pareça ser persuasivo o suficiente, foi realizado uma análise que aponta resultados ainda mais atraentes, estes resultados indicam o pagamento de uma recompensa três vezes maior. Fazendo isso, o proprietário do

[3]Está sendo considerado o modelo topo de linha fornecido com Android, no valor $800

dispositivo pode comprar outro dispositivo do mesmo preço apenas provendo seu dispositivo na DBB durante oito horas diárias. É uma estratégia atraente, pois o proprietário do dispositivo pode optar por ingressar na DBB somente em períodos ociosos, como durante a noite. Nesta análise, foi definido o preço do dispositivo em US$800, no momento da escrita desta tese este é o preço dos modelos mais caros equipados com o Android. Este tipo de dispositivo foi considerado por ser caro o suficiente para inviabilizar o modelo proposto, caso o preço do dispositivo se apresente alto demais para ser oferecido com infraestrutura colaborativa. Desta forma, o custo da recompensa do dispositivo é calculado usando $\frac{\$800}{365\times h}$, onde h é o número de horas dedicadas à DBB por dia: 24 (para a execução de testes 24 horas por dia) e 8 (para execução de testes durante 8 horas diárias).

Por fim, para a receita (terceiro componente na Equação 5-1), foram analisados cenários com Retorno do Investimento (ROI) de 20%, 30% e 40%, pois representam forte relação com o *market share* (BUZZELL; GALE; SULTAN, 1975). Usando os três componentes foi estimado o preço da DBB em cada cenário. A Figura 5.2 detalha o custo da DBB.

Os gráficos apresentam dados considerando o ROI em 20% (com recompensas 100% e 300%), em 30% e 40% (ambos com recompensa de 300%), além do custo do *dashboard* em cada alternativa.

Para comparar a abordagem apresentada nesta tese com as soluções alternativas voltadas à execução de testes de aplicações móveis, variou-se o número de dispositivos conectados de 1 a 6000, que é suficiente para cobrir certa de 25% dos modelos de dispositivos atualmente habilitados ao redor do mundo. A Figura 5.3 apresenta o custo para cada caso, considerando os quatro cenários para estimativa de preço da DBB. O eixo vertical está em escala logarítmica para melhor visualização.

Nos quatro cenários de estimativa de preço da DBB, o custo final vence os custos das outras soluções consideradas em praticamente todos os casos. Utilizando o cenário com retorno de 20% a DBB é sempre a melhor escolha. Por outro lado, quando o retorno é definido como em 30% ou 40%, o custo da DBB é mais alto do que o AWS no modelo de alocação por reserva, em alguns casos individuais (3,24% de todos os casos). No entanto, defende-se que essa diferença pode ser atenuada pelos outros benefícios de usar a DBB na execução de testes UI. Como já discutido, mesmo usando grandes números de dispositivos como parâmetro de comparação, a AWS não permite um número tão grande de dispositivos simultâneos sem solicitação prévia. Outra desvantagem é o número de modelos diferentes que este provedor (bem como os outros) oferece, que é significativamente menor do que a maioria dos cenários dados. Para reforçar estes argumentos, também comparou-se a redução de custos no uso da DBB em relação às outras soluções. A Figura 5.4 apresenta os resultados da comparação. Valores negativos indicam um aumento no custo da DBB em relação a outros provedores.

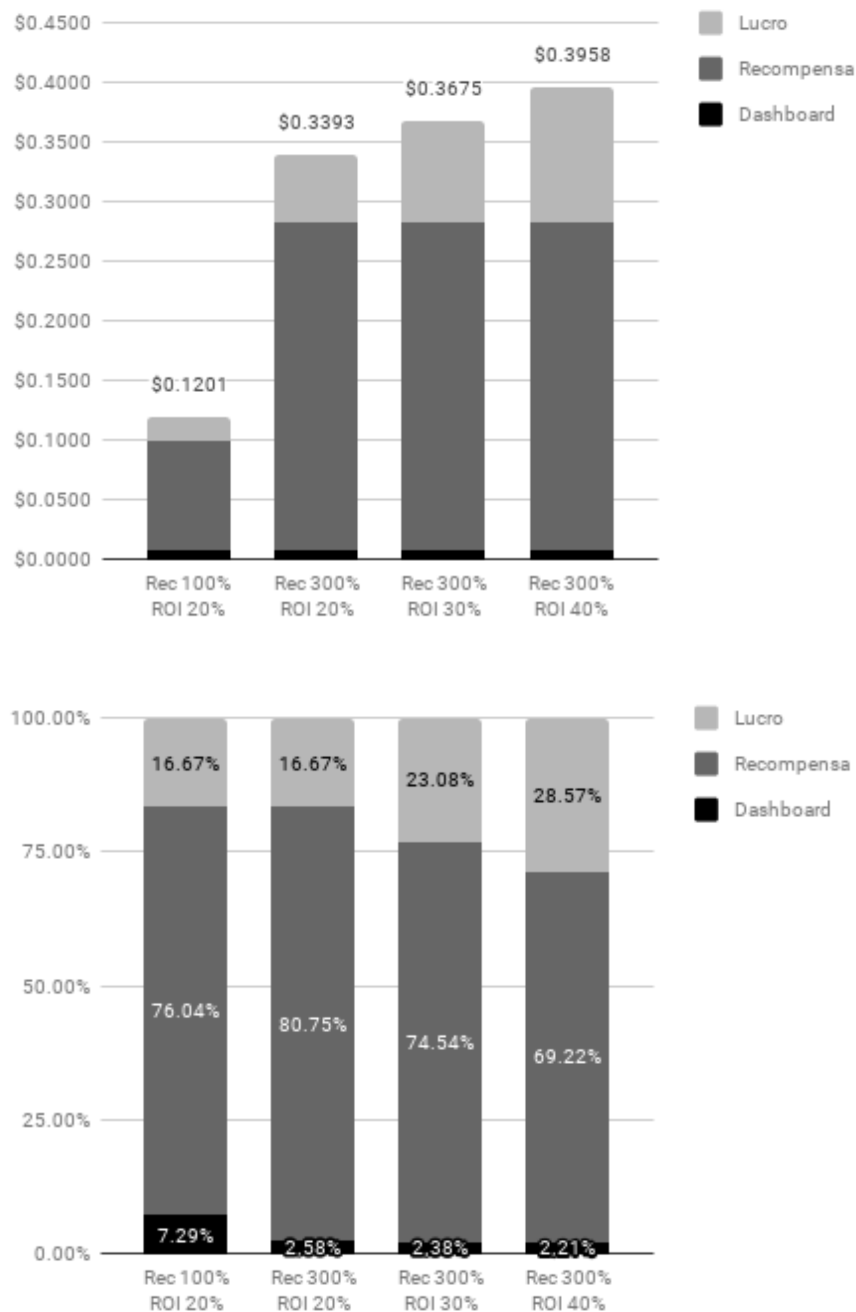


Figura 5.2: Detalhamento dos custos de um DH na DBB.

O uso da DBB pode promover uma economia média de 85.67% no custo total para execução de testes UI quando comparado com os maiores provedores em nuvem destinados ao mesmo fim. Além disso, para 75.67% dos casos analisados, a redução do custo é maior que 90%. Para reforçar a insignificância dos casos onde o custo da DBB é mais alto, a diferença do custo é de apenas 6.41% em média, com um máximo de 15.58%.

Considerando os resultados apresentados, pode-se defender que nossa abordagem é uma solução muito mais efetiva do ponto de vista econômico. Essa afirmação pode

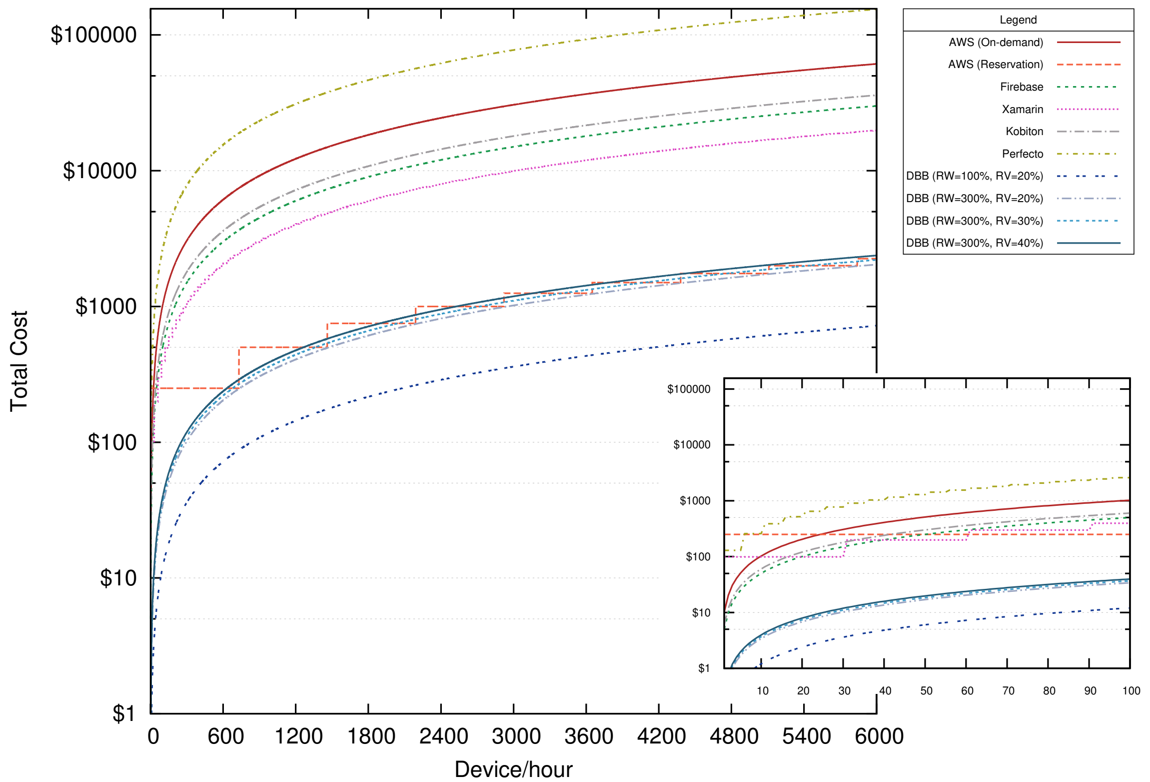


Figura 5.3: Comparação do custo da DBB com outras soluções.

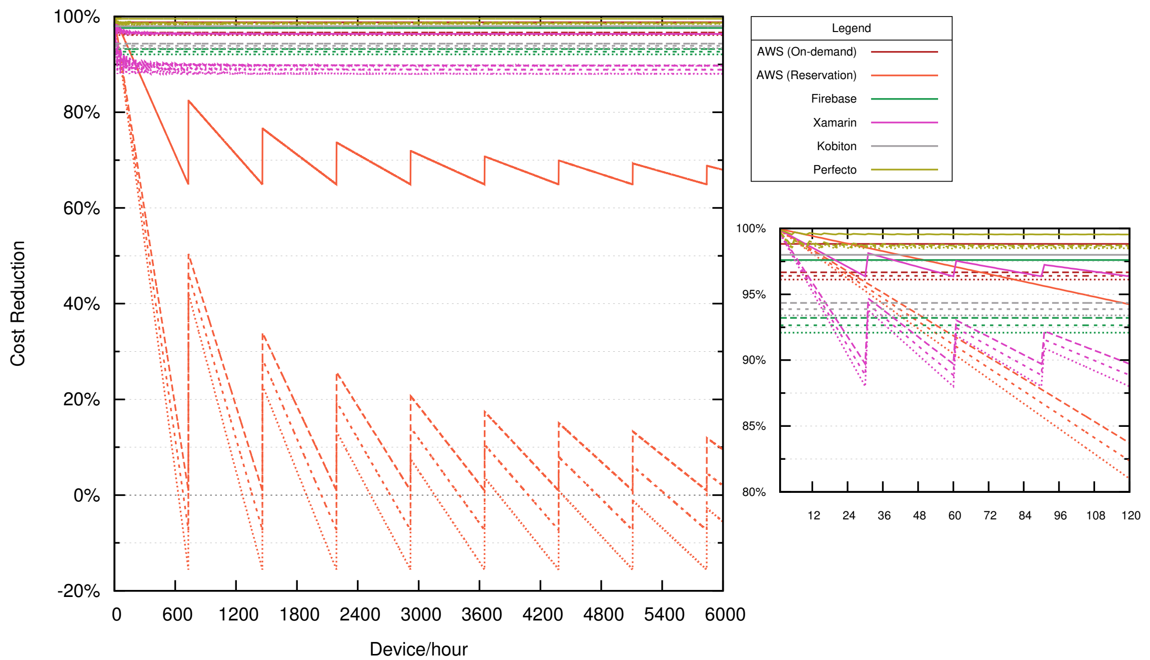


Figura 5.4: Redução do custo no uso da DBB em comparação com outras soluções.

ser fortalecida pelo potencial de melhores resultados se as premissas adotamos nessa análise forem alteradas. Por exemplo, um preço mais atraente pode ser alcançado se for obtida uma recompensa menos dispendiosa pela provisão de dispositivos ou se adotarmos o modelo de reserva na terceirização de recursos em nuvem. Dito isto, pode-se afirmar que

a solução proposta nesta tese é uma opção muito atraente, não só do ponto de vista da disponibilidade do dispositivo, mas também de uma visão econômica.

Isso é possível pelo fato de que a DBB é baseada no modelo de EC, que viabiliza a utilização de recursos de terceiros existentes para fornecer serviços baratos e escaláveis.

5.3 Demanda existente

Até o momento foram apresentados argumentos acerca da capacidade de cobertura, escalabilidade e redução de custos inerentes ao processo de validação de aplicativos entre diferentes dispositivos. Embora os resultados apresentados na seção anterior sejam atraentes do ponto de vista do desenvolvedor/testador, é necessário a verificação da demanda existente e, como consequência, o potencial de receita da DBB.

Ao invés de efetuar um levantamento com desenvolvedores/testadores de aplicativos Android, optou-se por uma análise considerando o número de *apps* existentes na loja virtual do Android. De acordo com dados divulgados, existem 2.6M de *apps* na Google Play Store (Google Play Store, 2018). A Figura 5.5 apresenta um histórico contendo a quantidade de *apps* em determinados períodos, desde a implantação deste modelo distribuição de aplicativos.

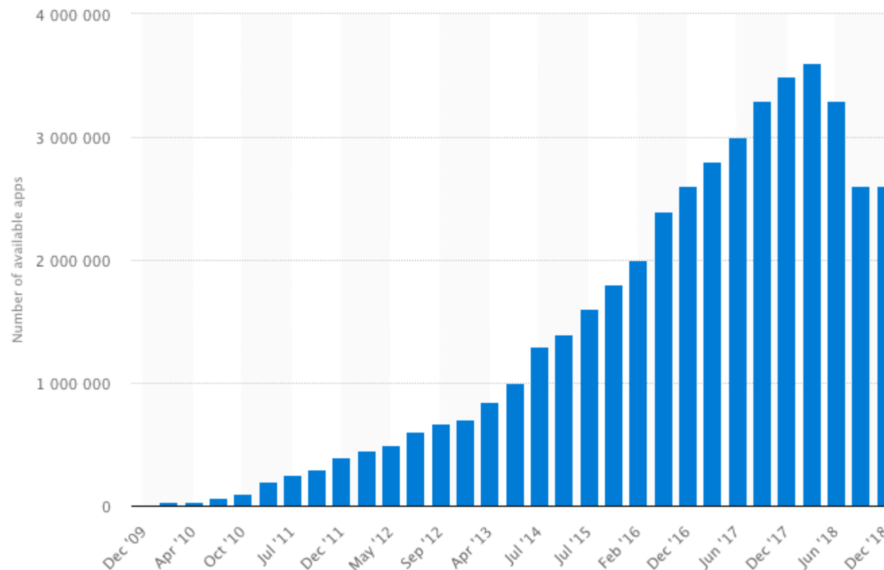


Figura 5.5: Total de *apps* disponíveis na Google Play Store (Statista, 2019)

Assumindo uma expectativa conservadora, em que somente 1% das *apps* existentes seriam testadas na DBB, ou seja 26k no total, e para cada uma destas fosse executado 1 min de testes UI para a liberação de uma *release*, a receita obtida pela DBB atinge os valores expressos na Tabela 5.6.

A execução de testes durante 1 min em 26k *apps*, geram 433.33 horas de faturamento, considerando o preço da DBB calculado na seção anterior de U$0.12 (ROI

Tabela 5.6: Expectativa de receita a partir da demanda apresentada.

	Recompensa de 100% e ROI de 20%	**Recompensa de 100% e ROI de 30%**	**Recompensa de 300% e ROI de 30%**	**Recompensa de 300% e ROI de 40%**
Número de horas (execuções de 1min)	433.33	433.33	433.33	433.33
Receita (DH)	U$52.04	U$142.99	U$155	U$168.99
Receita (cobertura de 25%)	U$312.240,00	U$857.940,00	U$930.000,00	U$1.013.940,00

de 20% e recompensa de 100%), temos uma receita de U$52.04 (por DH). Se a cobertura utilizada nos testes for de 25% do mercado (aproximadamente 6 mil dispositivos), a receita bruta para 433.33 horas de execução seria de U$312.240,00. Esta análise também é realizada para as recompensas de 100% (com ROI de 30%), 300% (com ROI de 30%) e 300% (com ROI de 40%). Assumindo que a grande maioria das *apps* existentes são construídas em ambientes ágeis, e que a recomendação das iterações para liberação de novas *releases* não devem ultrapassar 30 dias (SUTHERLAND; SUTHERLAND, 2014), a expectativa da receita expressa pela Tabela 5.6 é mensal.

Lembrando que a atratividade de validar *apps* com 25% de cobertura é verificada pela Equação 5-1, pois, executar 1 hora de testes em 6 mil dispositivos resulta em um custo de U$ 720.57, considerando o modelo de recompensa de 100% com ROI de 20%.

Na próxima seção são apresentadas ameaças ao modelo proposto, especialmente acerca de sua aplicação na indústria e algumas limitações identificadas.

5.4 Ameaças a Validade

No Capítulo 2 foram apresentados conceitos acerca da Economia Colaborativa (EC), apresentando-a como um modelo de compartilhamento de recursos em prol da redução de custos em diversas áreas do mercado. No entanto, para que um modelo desse tipo seja efetivo, é necessário que a comunidade interessada faça uso do dispositivo que opera o modelo. No contexto desta tese, é necessário que os desenvolvedores/testadores de aplicativos *Android* sejam convencidos da efetividade da arquitetura disponibilizada pela DBB, pois são eles que remunerariam uma plataforma desse tipo. A verificação desta ameaça pode ser realizada por meio de um levantamento envolvendo desenvolvedores e testadores de aplicativos móveis ao redor do mundo, em especial, aqueles envolvidos com desenvolvimento de *apps* com alto impacto no caso da ocorrência de falhas.

Do outro lado, existem os colaboradores do modelo, ou seja, os proprietários de dispositivos, que cederiam seus recursos para que terceiros pudessem fazer uso. Para via-

bilização dessa proposta, é necessário que os proprietários de dispositivos sejam convencidos das vantagens em disponibilizar seus recursos para outras pessoas. Considerando que os *smartphones* armazenam dados importantes de seus proprietários, é natural que haja alguma resistência por parte de usuários comuns a aderirem e colaborarem com o modelo. Apesar de, a recompensa financeira representar a contrapartida ao risco, esta ameaça pode ser minimizada por meio de experimentos que provoquem o envio de testes (mesmo que *fakes*) para execução em dispositivos reais, e a devida remuneração aos usuários que aceitarem as solicitações de execução.

Outro aspecto importante que deve ser colocado, é o fato de que a DBB funciona, no momento da escrita desta tese, apenas executando testes instrumentados escritos a partir da API do Espresso. Portanto, não é possível a execução de testes que envolvam interação entre diferentes aplicativos. Esta limitação pode ser eliminada através da integração da DBB com *frameworks* que endereçam testes UI não instrumentados, como é o caso do UiAutomator (Google Inc, 2018b).

No momento, a DBB não oferece um ambiente para que dispositivos disponíveis ao redor do mundo possam ser catalogados a partir de um modelo de priorização, de forma que o testador possa escolher entre a realização de testes em todos os dispositivos disponíveis ou apenas dispositivos que garantam uma boa cobertura do mercado.

Um aspecto importante mencionado no Capítulo 5 é sobre o problema da diversidade de dispositivos oferecidos pelos provedores em nuvem. No caso da DBB, não existe a garantia de que haja alta diversidade, uma vez que não é vetado o ingresso de nenhum dispositivo Android habilitado, permitindo o ingresso de um grande número de dispositivos do mesmo modelo. No entanto, o modelo é muito mais escalável quando comparado com o modelo existente e utilizado pelos provedores atuais, aumentando as chances do crescimento do parque de dispositivos com investimento mínimo.

Como mencionado no Capítulo 3, existe o risco de relatórios de execução de testes nunca serem reportados aos interessados, por questões de conectividade ou mesmo indisponibilidade do dispositivo em que o relatório foi gerado mas não entregue. O fato de mais de um dispositivo do mesmo modelo receber a solicitação de execução do teste, ameniza esta situação, embora não resolva em todos os cenários.

Outro aspecto que deve ser considerado são as interrupções, pois mesmo que um usuário (proprietário de dispositivo) tenha aceito e autorizado a execução de vários testes, é possível que algo aconteça e o usuário necessite utilizar seu *smartphone* para se comunicar com alguém, ou mesmo para desligar o despertador. Esses casos são colocados como objeto de trabalhos futuros.

5.5 Considerações Finais

Este capítulo apresentou uma análise da escalabilidade e cobertura de dispositivos proporcionadas pela plataforma proposta, bem como uma avaliação do custo e precificação do dispositivo/hora, além de trazer uma reflexão acerca do tamanho do mercado disponível e o potencial de exploração deste mercado pelo modelo proposto.

A análise realizada apresenta uma economia substancial (acima de 85% em média) em comparação com os modelos oferecidos pelos provedores de *TaaS* existentes. Como o modelo implementado pela DBB não depende da aquisição de dispositivos, dependendo apenas da parceria com proprietários de dispositivos, a diversidade de modelos pode aumentar, permitindo que validações possam ocorrer até em dispositivos antigos, não disponíveis para compra ou mesmo locação em nuvem. No entanto, várias limitações importantes precisam ser mitigadas ou eliminadas, principalmente aquelas relacionadas à segurança dos dados dos proprietários de dispositivos que, concordarem em ceder seus dispositivos para execução de testes. Além disso, algumas melhorias são sensíveis para os desenvolvedores/testadores, como a captura do estado das telas em que ocorreram falhas.

No próximo capítulo, são apresentados os principais trabalhos da literatura voltados a redução dos custos de testes de aplicativos móveis.

CAPÍTULO 6

Trabalhos Correlatos

Neste capítulo são apresentados alguns dos principais trabalhos relacionados à redução de custos do processo de testes de aplicações móveis. Apesar de haver vários outros trabalhos acerca deste assunto, este capítulo é dedicado a trabalhos que possuem maior alinhamento com a abordagem proposta nesta tese, pois endereçam problemas relacionados aos testes UI.

Os trabalhos foram divididos em 7 categorias: A) Gravação e replicação de casos de teste; B) Análise estática e dinâmica; C) Geração de dados de entrada; D) Problema do oráculo; E) Priorização de casos de teste, artefatos para testes e dispositivos; E) Plataformas em nuvem.

6.1 Gravação e execução de casos de teste UI

Nesta seção são apresentados os principais trabalhos que buscam contribuir para a redução do custo durante a realização de testes UI em dispositivos móveis, através da técnica de gravação e replicação de testes (LIU et al., 2014; LIN et al., 2014a).

Em Kaasila et al. (2012) foi desenvolvido uma plataforma, intitulada *TestDroid*, que permite a gravação de testes UI por meio do uso da aplicação alvo. Ao final da utilização, um arquivo contendo um *script* (da sequência de passos executados) é enviado para um ambiente online e executado em dispositivos físicos alocados para atender à plataforma desenvolvida. O *script* gerado a partir da utilização da aplicação alvo é baseado no *framework* Robotium (ZADGAONKAR, 2013), portanto não é necessário o acesso ao código fonte da aplicação para que os testes sejam escritos, bastando apenas que o testador tenha o arquivo `.APK` da aplicação alvo. No entanto, a abordagem se baseia nas coordenadas de tela em que ocorrem as interações do usuário com a aplicação, inutilizando os testes entre dispositivos com diferenças significativas de densidade e tamanho de tela. Esta limitação não ocorre na ferramenta MoQuality (MoQuality Inc, 2019), que gera casos de testes para API do Espresso (Google Inc, 2013) e também para API do Appium (JS Foundation, 2012). Neste caso, as interações com o usuário são convertidas para casos de teste instrumentados utilizando API específica. Após a gravação

dos testes, estes podem ser executados em uma infraestrutura de dispositivos reais em nuvem.

Uma ferramenta semelhante ao *TestDroid* é apresentada em (Google Inc, 2018a). Esta ferramenta consiste de um *plugin* disponível para as IDEs *Android Studio* e Eclipse, que permite a desenvolvedores/testadores gerarem testes de caixa branca baseados no Espresso (Google Inc, 2013) a partir da utilização da aplicação. Os casos de testes gerados podem ser executados de acordo com a disponibilidade de dispositivos e emuladores.

Outros trabalhos que geram *scripts* de execução e se baseiam em coordenadas de tela foram desenvolvidos (GOMEZ et al., 2013; HU; AZIM; NEAMTIU, 2015; HALPERN et al., 2015; FREITAS et al., 2016). Gomez et al. (2013) apresenta uma abordagem que registra eventos de baixo nível utilizando o utilitário *Android Getevents* (GOOGLE, 2017) e gera um *script* de replicação para o mesmo dispositivo. A abordagem de baixo nível apresentada é eficaz na gravação e reprodução de gestos complexos como por exemplo *multi-touch*.

Algumas fraquezas apresentadas em Gomez et al. (2013) foram tratadas em Hu, Azim e Neamtiu (2015). Este trabalho traz uma abordagem de gravação e replicação baseada em *stream*, de forma que eventos concorrentes são registrados de acordo com sua ordem de ocorrência. Desta forma, as entradas oriundas da interface de rede, GPS, câmera, microfone, *touchscreen*, acelerômetro, bússola e outros aplicativos via *IPC* (*Inter-process communication*) são replicadas exatamente na ordem em que ocorrem, evitando perturbações durante a replicação do *script* gerado na gravação.

Gao et al. (2014) apresenta uma nova versão do *MobileTest* (BO; XIANG; XIAOPENG, 2007), que seria fornecida através de um serviço. Originalmente, o *MobileTest*, assim como o TestDroid (KAASILA et al., 2012), foi desenvolvido para auxiliar testadores na geração e replicação de ações de usuários baseando-se em *scripts* de alto nível.

Após a ascensão do *framework* oficial para testes UI no Android (Espresso), Fazzini et al. (2017b) propôs uma ferramenta chamada *Barista*, capaz de gravar testes a partir de eventos de tela, gerando testes compatíveis com a versão 1 do Espresso. Atualmente, a ferramenta está disponível no *Google Play* e gera testes compatíveis com a versão 2 do Espresso, trazendo vantagens em relação a outras ferramentas de gravação e replicação que se baseiam em scripts ou *frameworks* mais antigos.

6.2 Análise estática e dinâmica

Alguns trabalhos abordam o custo por meio da diminiução de artefatos a serem testados. Algumas abordagens elencam os artefatos a serem testados por meio de modelos que indiquem a propensão a defeitos, através de análise estática. Wei, Liu e

Cheung (2016) propuseram uma ferramenta chamada *FicFinder*, capaz de indicar componentes propensos a defeitos com base em análise estática de código. Ham e Park (2011) apresenta um sistema de teste de compatibilidade de aplicativos móveis, abordando a fragmentação do *Android*. Com base no resultado da análise do código e do uso de APIs, essa ferramenta é capaz de verificar estaticamente a compatibilidade de um aplicativo com os dispositivos desejados. Assim, ao comparar a fragmentação no nível de código (código construído no projeto) e no nível da API (APIs utilizadas no projeto), o tempo e o custo necessários à execução dos testes poderiam ser reduzidos.

No âmbito dos testes UI, as técnicas de análise estática são aplicadas para extração de grafos que representam caminhos possíveis em softwares desktop já há algum tempo (AMALFITANO et al., 2012). Considerando que a validação de aplicativos móveis passa principalmente pela validação das interfaces gráficas, por meio da interação do usuário, essas técnicas chamadas de *GUI Ripping* são amplamente utilizadas por desenvolvedores/testadores de aplicativos móveis (MEMON et al., 2013; AMALFITANO et al., 2013).

Alguns trabalhos acrescentam a análise dinâmica como ferramenta para validação de aplicativos móveis. Hu e Neamtiu (2011) fazem uma análise dinâmica da execução de testes com dados gerados a partir da ferramenta *UI/Application Exerciser Monkey* (Google Inc, 2018c) e testes gerados a partir do JUnit (JUNIT, 1998). Após a execução de cada caso de teste, a análise ocorre nos arquivos de *log* gerados durante a execução dos testes, em busca de erros ocorridos. Zheng et al. (2012), apresentou um método de análise estática e dinâmica capaz de revelar condições de acionamento baseadas em interface do usuário em aplicativos *Android*. Primeiramente é utilizada análise estática para extrair os caminhos esperados, montando um grafo de chamadas e, em seguida, a análise dinâmica é utilizada para percorrer cada elemento da interface do usuário e explorar os caminhos de interação com as APIs utilizadas. Para validar o modelo proposto, foi implementado uma ferramenta chamada *SmartDroid* que gera testes capazes de explorar os caminhos possíveis extraídos pela análise estática.

6.3 Geração de dados de entrada

Existem várias abordagens para geração automática de entrada de dados de testes de aplicativos móveis. Nesta seção são considerados trabalhos que envolvam ferramentas e técnicas aplicadas na geração de entradas para testes de aplicativos *Android*. O principal objetivo destas ferramentas é a detecção de *bugs* existentes.

O processo de compilação de aplicativos *Android*, faz uso pesado de ofuscação de código, impondo limitações para as ferramentas que empregam análise estática diretamente em arquivos `.APK`. Portanto, para explorar o comportamento de aplicativos *Android*

e contornar as limitações dessas técnicas baseadas em análise estática, é comum o uso da análise dinâmica e o uso de ferramentas para geração automática de dados de entrada.

Estas ferramentas de geração de entradas podem exercitar a aplicação tanto de forma isolada quanto integrada com outros aplicativos. O desafio de qualquer uma destas ferramentas é a geração de entradas relevantes capazes de exercitar ao máximo o comportamento do aplicativo testado.

Como aplicativos *Android* são orientados a evento, entradas são representadas em forma de eventos, que podem ser interações de usuário (eventos UI), como cliques, rolagem, e digitação de texto, ou eventos de sistemas, como uma notificação recebida por SMS. A geração destas entradas é aleatória ou através de uma estratégia de exploração sistemática, neste caso, a exploração pode ser orientada por um modelo do aplicativo, construído estaticamente ou dinamicamente, ou explorando técnicas de cobertura.

Os seguintes trabalhos empregam a estratégia de geração aleatória:

- **Monkey** (Google Inc, 2018c) - é a ferramenta mais usada para testar aplicativos Android, pois faz parte do kit de ferramentas de desenvolvedores do Android e, portanto, não exige nenhum esforço adicional de instalação. O Monkey implementa a estratégia aleatória mais básica, utilizando testes de caixa preta gera apenas eventos de interface com o usuário.
- **Dynodroid** (MACHIRY; TAHILIANI; NAIK, 2013) - também é baseado em exploração aleatória, mas tem vários recursos que tornam sua exploração mais eficiente em comparação com o Monkey (Google Inc, 2018c). Em primeiro lugar, ele pode gerar eventos do sistema, e faz isso verificando quais são os eventos de sistema utilizados pelo aplicativo. O Dynodroid obtém essas informações monitorando quando um aplicativo registra um ouvinte no do Android, por exemplo quando o aplicativo faz uso do mecanismo de acessibilidade do Android. Por esse motivo, é necessário instrumentar o *framework* na aplicação a ser testada.
- **Intent Fuzzer** (SASNAUSKAS; REGEHR, 2014) - testa principalmente como um aplicativo pode interagir com outros aplicativos instalados no mesmo dispositivo. Ele inclui um componente de análise estática, que é construído sobre o FlowDroid (ARZT et al., 2014), para identificar a estrutura esperada de `intents`, de modo que o *Intent Fuzzer* possa gerá-las corretamente. Esta ferramenta mostrou-se eficaz na revelação de problemas de segurança.
- **DroidFuzzer** (YE et al., 2013) - é diferente de outras ferramentas de geração de eventos ou `intents` de interfaces com o usuário. Ele gera apenas entradas para atividades que aceitam tipos de dados `MIME`, como arquivos `AVI`, `MP3` e `HTML`. Os autores do artigo mostram como essa ferramenta pode encontrar falhas em alguns aplicativos de vídeo.

Alguns trabalhos aplicaram estratégias de exploração baseadas em modelo já utilizados, em aplicações web e desktop, em aplicativos Android. Estes modelos são máquinas de estado finita que representam `Activities` como estado e os eventos como transição. A maioria das ferramentas constrói esse modelo dinamicamente, finalizando a máquina de estados quando todos os eventos, que podem ser acionados a partir de todos os estados descobertos, levam a estados já explorados. Os trabalhos a seguir empregam estratégias de exploração baseadas em modelo:

- **MobiGUITAR** (AMALFITANO et al., 2015) - se trata de um *GUIRipper* que cria dinamicamente um modelo (baseado em uma máquina de estados) de testes para um aplicativo a partir de seu estado inicial. Quando um novo estado é visitado, é mantida uma lista de eventos que podem ser gerados a partir do estado atual da tela, e os lança sistematicamente. O *GUIRipper* implementa uma estratégia *DFS* (*Depth-First Search*) (TARJAN, 1972) e retoma a varredura de estado inicial quando não consegue detectar novos estados durante a varredura. São gerados apenas eventos de interface do usuário, portanto, não pode expor o comportamento do aplicativo do qual depende de eventos do sistema. O *MobiGUITAR* tem duas características que o tornam único entre as ferramentas baseadas em modelos. Primeiro, permite explorar um aplicativo a partir de diferentes estados iniciais, além de permitir que os desenvolvedores/testadores forneçam um conjunto de valores de entrada que podem ser usados durante a exploração.
- **ORBIT** (YANG; PRASAD; XIE, 2013) - implementa a mesma estratégia de exploração do *MobiGUITAR*, mas analisa estaticamente o código-fonte do aplicativo para entender quais eventos de UI são relevantes para uma atividade específica. Isso o torna mais eficiente do que o *MobiGUITAR*, uma vez que apenas entradas relevantes são geradas.
- **A^{3}E-Depth-first** (AZIM; NEAMTIU, 2013a) - é uma ferramenta de código aberto que implementa duas estratégias totalmente distintas e complementares. Primeiramente é implementa uma busca de profundidade no modelo da aplicação, gerado dinamicamente. Em essência, ela implementa exatamente a mesma estratégia de exploração das ferramentas anteriores. Sua representação de modelo é mais abstrata, pois representa cada tela como um único estado, sem considerar estados diferentes dos elementos da tela. Essa abstração não permite que a ferramenta distinga alguns estados que são diferentes e pode levar à falta de algum comportamento que seria fácil de exercitar se um modelo mais preciso fosse usado.
- **SwiftHand** (CHOI; NECULA; SEN, 2013) - usa um modelo dinâmico de máquina de estados finita do aplicativo, e uma de suas principais características é que ele otimiza a estratégia de exploração para minimizar as reinicializações do aplicativo

durante o rastreamento. O *SwiftHand* gera apenas toques e rolagens de eventos da interface do usuário e não pode gerar eventos do sistema.

- **PUMA** - (HAO et al., 2014) - fornece a exploração aleatória também implementada pelo *Monkey*. A novidade desta ferramenta não está na estratégia de exploração, mas sim no seu *design*. O *PUMA* é um *framework* que pode ser facilmente estendido para implementar qualquer análise dinâmica em aplicativos Android usando a estratégia básica de exploração utilizada pelo *Monkey*. Além disso, permite a fácil implementação de diferentes estratégias de exploração, pois o *framework* fornece uma representação finita da máquina de estados do aplicativo. Também permite redefinir facilmente a representação de estado e a lógica para gerar eventos.

Outra estratégia de exploração utilizada é a exploração sistemática, utilizada quando um comportamento é revelado apenas com entradas específicas. Neste contexto, técnicas como execução simbólica e algoritmos evolucionários guiam a exploração do código sem cobertura.

- **A^{3}E-Targeted** (AZIM; NEAMTIU, 2013b) - fornece uma estratégia de exploração alternativa que complementa a ferramenta A^{3}E-Depth-first. A abordagem direcionada depende de um componente que, por meio de análises de contaminação, pode construir o grafo de transição de atividade estática do aplicativo. Esse grafo é uma alternativa ao modelo de máquina de estados finitos dinâmicos da exploração de pesquisa em profundidade e permite que a ferramenta cubra atividades de maneira mais eficiente, gerando intenções (`intent`).
- **EvoDroid** (MAHMOOD; MIRZAEI; MALEK, 2014) - faz uso de algoritmos evolutivos para gerar entradas relevantes para uma *app*. Na estrutura de algoritmos evolutivos, o EvoDroid representa os indivíduos como sequências de entradas de teste e implementa a função de adequação para maximizar a cobertura.
- **ACTEve** (ANAND et al., 2012b) - lida com os eventos de sistema e das interfaces gráficas do aplicativo, empregando a técnica de verificação *concolic*, que rastreia, de forma simbólica, os eventos gerados por um *framework* e a manipulação desses eventos no aplicativo. Por esse motivo, o *ACTEve* precisa instrumentar o aplicativo em teste.
- **JPF-Android** (MERWE; MERWE; VISSER, 2014) - estende o Java PathFinder (JPF) (HAVELUND; PRESSBURGER, 2000), uma popular ferramenta de verificação de modelos para aplicações Java, para suportar aplicativos Android. Isso permitiria verificar aplicativos em propriedades específicas. O *JPF-Android* visa explorar todos os caminhos em um aplicativo Android e pode identificar impasses e exceções de tempo de execução.
- **SAPIENZ** (MAO; HARMAN; JIA, 2016) - emprega a otimização multiobjetivo utilizando técnicas baseadas em pesquisa, para explorar e otimizar sequencias de

testes, minimizando a quantidade de casos de testes e aumentando a cobertura. SAPIENZ apresenta melhorias nos resultados, quantidade de testes gerados e total de falhas identificadas,quando comparado com ferramentas como *Dynodroid* e *Monkey*. A ferramenta permite que a validação ocorra mesmo quando o testador tem acesso apenas ao arquivo `APK`.

6.4 Trabalhos que abordam o problema do oráculo

As ferramentas tradicionais de teste de gravação-replicação concentram-se principalmente em facilitar o processo de escrita do caso de teste, mas não no processo de repetição e verificação. A precisão das ferramentas de teste degrada-se significativamente quando o dispositivo sob teste (DUT) está sob carga pesada. A fim de melhorar a precisão, Lin et al. (2014a) propõe uma abordagem validada através de uma ferramenta chamada SPAG. Esta ferramenta usa o *batch* de eventos e uma função de espera inteligente para eliminar a incerteza do processo de reprodução e adota as informações de layout da interface gráfica (GUI) para verificar os resultados do teste. Posteriormente, Lin et al. (2014b) apresenta uma ferramenta de teste UI automatizado para *smartphones* com câmera (SPAG-C), uma extensão do SPAG, para testar aplicativos *Android*, com o objetivo de reduzir o tempo necessário no registro dos casos de teste, aumentando a capacidade de reutilização do oráculo sem comprometer a precisão dos testes.

6.5 Trabalhos que abordam o custo por meio da priorização

No contexto dos testes de aplicativos móveis, a priorização de artefatos, casos de teste e dispositivos, trazem impacto significativo no custo dos testes, especialmente quando a priorização foca em testes UI, pois, de acordo com a pirâmide de testes apresentada no Capítulo 2, os testes UI representam o maior custo do processo de testes.

As técnicas de priorização de casos de teste são amplamente usadas para atingir determinados objetivos de desempenho durante testes de regressão. Uma meta comumente usada é a alta taxa de detecção de falhas, em que os casos de teste são ordenados de forma a permitir a detecção de falhas mais rapidamente. No entanto, para um teste de regressão ideal, é necessário levar em consideração vários indicadores de desempenho. Marijan (2015) apresenta uma abordagem para a priorização de casos de teste de regressão. A abordagem foi desenvolvida para otimizar os testes de regressão possibilitando a detecção de falhas mais rapidamente, integrando três perspectivas diferentes: negócio, desempenho e técnica.

Em ambientes de testes baseados em *crowdsourcing*, inspecionar grandes volumes de relatórios de teste é uma tarefa árdua, mas inevitável. Várias técnicas de classificação e priorização de relatórios de testes tem sido adotadas para auxiliar nesta tarefa. No entanto, no domínio de testes de aplicativos móveis, os relatórios de teste geralmente consistem em capturas de tela e pequenos *logs*, e, portanto, as técnicas baseadas em texto podem ser ineficazes ou inaplicáveis. A escassez e a ambiguidade das informações nos *logs* motivaram trabalhos como o desenvolvido por Feng et al. (2016). A abordagem apresentada neste trabalho utiliza otimização multiobjetivo para realizar a priorização de artefatos a serem exercitados nos testes de regressão com base na similaridade de capturas de tela.

As técnicas de priorização de casos de teste focam em testes de regressão que são conduzidos em uma suíte de teste. No entanto, a priorização do caso de teste para novos testes também é necessária. Yoon et al. (2012) propõe um método para priorizar novos casos de teste calculando o valor da exposição ao risco para requisitos e analisando itens de risco com base no cálculo para avaliar casos de teste relevantes e, assim, determinar a prioridade do caso de teste através dos valores avaliados.

No contexto dos testes em dispositivos *Android*, a seleção, bem como a priorização de dispositivos passaram a ser utilizadas como forma de minimizar o impacto da fragmentação do ecossistema *Android*. Em (VILKOMIR; AMSTUTZ, 2014; VILKOMIR et al., 2015) foi utilizada uma abordagem combinatorial na seleção de dispositivos e os resultados apontam 90% de cobertura do mercado. O método emprega as técnicas *Each Choice (EC)* e *pair-wise* considerando o conjunto de características dos dispositivos que mais influenciam em falhas específicas.

6.6 Plataformas em nuvem

Vários trabalhos apontam vantagens, problemas e desafios a serem enfrentados durante a execução de testes em infraestruturas baseadas em nuvem. Surge então o conceito *Test as a Service (TaaS)*, abordado inicialmente nos trabalhos de Gao, Bai e Tsai (2011), Gao et al. (2012) e Yu et al. (2010).

Tao e Gao (2017) e Katherine e Alagarsamy (2012) apresentam os desafios e sugestões de abordagens para realização de testes envolvendo dispositivos móveis em nuvem. Em 2014 começaram a surgir os ambientes baseados em nuvem com o objetivo de oferecer dispositivos reais para realização de testes. Os provedores Amazon (AMAZON, 2018a), Google (Google Firebase, 2018), Xamarin (XAMARIN, 2018), Perfecto (PERFECTO, 2018) e Kobiton (KOBITON, 2018), iniciaram a corrida para o provimento do *TaaS* em prol da redução dos custos envolvidos na execução massiva de testes de aplicativos móveis.

No capítulo 5 são apresentados detalhes acerca de cada provedor e uma comparação com a plataforma proposta nesta tese.

Schneider e Cheung (2013) apresenta à comunidade científica a utilização do *crowdsourcing* aplicado no contexto de testes. Então várias plataformas voltadas a este paradigma passaram a ser utilizadas por times de desenvolvimento de software interessados na diminuição do custo de alguns tipos de teste. Mao et al. (2015) apresenta uma revisão acerca das plataformas que abordam a adoção do *crowdsourcing* na Engenharia de Software. No momento da escrita desta tese as seguintes plataformas suportam este modelo para a validação de aplicações móveis:

- Passbrains (PASSBRAINS, 2018);
- 99Tests (99TESTS, 2018);
- TestBirds (TESTBIRDS, 2018);
- Pay4Bugs (PAY4BUGS, 2018);
- CrowdSourcedTesting (CROWDSOURCEDTESTING, 2018);
- TestFlight (APPLE, 2018);
- uTest (APPLAUSE, 2018);
- BugCrowd (BUGCROWD, 2018).

Essas plataformas se diferenciam pelo modelo de remuneração dos testadores, tipos de testes possíveis de serem contratados pelos times de desenvolvimento e o ecossistema disponível (no caso do *TestFlight* apenas o IOS é atendido).

6.7 Considerações Finais

Este capítulo apresentou alguns dos principais trabalhos que contribuem para a redução de custos dos testes de aplicativos móveis. Foram considerados trabalhos focados na execução de testes UI, análise estática e dinâmica, geração de dados de entrada, oráculo de testes, priorização de artefatos e dispositivos, além de trabalhos voltados ao provimento do *TaaS* em nuvem.

Embora os trabalhos considerados neste capítulo tenham contribuído para a área de testes em dispositivos móveis, trazendo significativas melhorias, os trabalhos relacionados à execução de testes utilizam a arquitetura padrão dos *frameworks* existentes, de forma que limitações inerentes ao uso destes *frameworks* permaneçam mesmo em grandes provedores de *TaaS*.

A seguir são apresentadas as conclusões bem como os trabalhos futuros e considerações finais realizados a partir do desenvolvimento desta tese.

CAPÍTULO 7

Conclusão

A era do acesso tem revolucionado a forma como as pessoas utilizam recursos, reduzindo custos, viabilizando o acesso e melhorando a vida das pessoas, permitindo que estas se preocupem com o que é mais importante em cada projeto. Como plataforma para suportar este novo paradigma, a Economia Colaborativa viabiliza o acesso das pessoas a vários tipos de infraestrutura, de carros a computadores, de hospedagem a *smartphones*. Graças a esta plataforma, novos mercados são criados em atendimento às demandas pelo acesso.

Os testes de aplicativos em dispositivos móveis oferecem desafios adicionais em relação aos testes de aplicações web e desktop, pois em dispositivos móveis, vários recursos adicionais são frequentemente utilizados pelos aplicativos: acelerômetro, diferentes interfaces de conexão, gestos, localização em tempo real, entre outros. No contexto da execução de testes UI, destaca-se a alta fragmentação existente no ecossistema *Android*, que torna ainda mais desafiador a validação de aplicações em função de diferentes configurações de tela, versão do sistema operacional e *drivers* utilizados pelos aplicativos.

Inspirada neste contexto e no fenômeno da era do acesso, esta tese apresentou uma abordagem disruptiva para execução distribuída de testes UI em dispositivos reais, de forma que o custo possa ser reduzido e com possibilidade de criar um novo mercado, como apontado no Capítulo 5. A DBB respondeu de forma satisfatória a dois experimentos envolvendo 12 aplicativos: um experimento envolvendo três aplicativos, com o objetivo de verificar a execução distribuída de testes UI em dispositivo reais; outro experimento envolvendo nove aplicativos, com o objetivo de explorar mais cenários de utilização de componentes de tela. Alguns aplicativos exigiam confirmação de *runtime* em dispositivos executando versões superiores a API 22 do *Android*, e a DBB confirmou com sucesso a todas as requisições, permitindo que os testes pudessem ocorrer sem a intervenção humana.

A fim de responder as questões de pesquisa 1 e 2 (QP1 e QP2) foram realizadas duas análises, uma relacionada a escalabilidade e diversidade de dispositivos e outra relacionada a precificação e impacto econômico da DBB. Alguns modelos de dispositivos (que possuem uma importante fatia de mercado) não foram encontrados nos provedores

de *TaaS* considerados no estudo, comprometendo a cobertura de dispositivos no processo de testes UI. Uma plataforma baseada em EC aumentaria as chances desses dispositivos estarem disponíveis para testes, além de dispositivos considerados antigos, inexistentes em todos os provedores analisados. Na perspectiva econômica, as análises indicam uma redução média de 85.67% no custo com infraestrutura de testes, podendo chegar a 90%, considerando um ROI e 20%.

Vários problemas, inerentes a operação de um ambiente de testes de software baseado em EC devem ser considerados em trabalhos futuros, enumerados a seguir:

1. A segurança de dados dos proprietários de dispositivos é um desafio, portanto, investigações acerca de dados possíveis de serem acessados por testes automatizados é fundamental para a aceitação do modelo proposto;
2. Aplicativos multiplataformas devem ser exercitados em todas as plataformas em que devem funcionar, no entanto, a DBB executa testes usando a implementação do Espresso e portanto, apenas validações no *Android* são possíveis. Explorar as possibilidades de execução distribuída em outros sistemas operacionais colabora para a melhoria da cobertura do mercado global de dispositivos;
3. Os relatórios de testes de aplicativos móveis devem anexar o estado das telas envolvidas nos casos de teste quando um *bug* é revelado. A DBB não captura o estado da tela que revelou o *bug*. Este incremento é de grande valor no processo de depuração;
4. Cenários de execução imediata, em atendimento à urgências de times de desenvolvimento de aplicativos agregam valor e podem agregar diferentes cenários contribuindo para redução do preço do dispositivo/hora, ou mesmo permitindo que proprietários de dispositivos possam receber uma recompensa maior por atender prontamente a solicitações de execução de testes;
5. Experimentos envolvendo um número massivo de dispositivos reais com a devida remuneração a seus proprietários, validando a proposta em cenários reais de negociação;
6. Introdução de capacidade colaborativa de avaliação de provedores de infraestrutura, bem como tomadores de serviço, aumentando a credibilidade do modelo proposto.

Para finalizar, também como trabalho futuro, pretende-se expandir o volume de aplicativos a serem testados, utilizando casos de testes escritos por diferentes testadores, utilizando apenas recursos que diferenciam aplicativos móveis dos demais aplicativos, como gestos, acelerômetros e eventos assíncronos originados de outros aplicativos, com o intuito de mapear a ocorrência de *bugs* relacionados a esses recursos e as APIs utilizadas.

Investigações envolvendo a aplicação dos critérios de testes aos testes UI em dispositivos móveis, em conjunto com a técnica *GUI-Ripping*, podem contribuir para evo-

lução da plataforma, auxiliando desenvolvedores/testadores na construção de conjuntos menores e mais eficazes de testes.

A integração da DBB com ferramentas de geração automática de testes UI, bem como ferramentas de geração de testes baseadas no uso de aplicativos móveis, contribuiria para a utilização em larga escala da plataforma, uma vez que o domínio do *framework* Espresso não seria obrigatório ao desenvolvedor/testador.

A partir do uso massivo da DBB, seria possível a aplicação de modelos de priorização de dispositivos, de forma a auxiliar desenvolvedores/testadores na escolha dos dispositivos a serem considerados no processo de validação de aplicativos, permitindo a redução do custo da execução dos testes UI em dispositivos reais.

Bibliografia

99TESTS. *99Tests Crowd Testing*. 2018. Page. Disponível em: http://www.99tests.com/crodtesting. Acesso em: 23/03/2017.

Adobe Systems. *PhoneGap Framework*. 2016. Project Web Page. https://phonegap.com/. Accessed on: 03/20/2018.

Airbnb. *Airbnb*. 2008. Project Web Page. Available at: https://www.airbnb.com. Accessed on: 01/04/2017.

AMALFITANO, D. et al. Using gui ripping for automated testing of android applications. In: ACM. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.], 2012. p. 258–261.

AMALFITANO, D. et al. Considering context events in event-based testing of mobile applications. In: IEEE. *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. [S.l.], 2013. p. 126–133.

AMALFITANO, D. et al. Mobiguitar: Automated model-based testing of mobile apps. *IEEE software*, IEEE, v. 32, n. 5, p. 53–59, 2015.

Amazon. *Amazon Web Services*. 2018. https://aws.amazon.com. Accessed on: 04/27/2018.

AMAZON. *AWS Device Farm*. 2018. Project Web Page. https://aws.amazon.com/pt/device-farm/. Accessed on: 02/28/2018.

AMAZON. *AWS Total Cost of Ownership (TCO) Calculator*. 2018. http://awstcocalculator.com/. Accessed on: 04/20/2018.

ANAND, S. et al. Automated concolic testing of smartphone apps. In: ACM. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. [S.l.], 2012. p. 59.

ANAND, S. et al. Automated Concolic Testing of Smartphone Apps. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2012.

APPBRAIN. *Average CPI per country*. 2018. Project Web Page. https://www.appbrain.com/stats/android-cpi-per-country. Accessed on: 03/05/2018.

APPLAUSE. *uTest*. 2018. Page. Disponível em: https://www.utest.com/. Acesso em: 23/03/2017.

Apple. *Apple Music*. 2015. Project Web Page. Available at: https://www.apple.com/music/. Accessed on: 01/04/2017.

APPLE. *TestFlight*. 2018. Page. Disponível em: https://developer.apple.com/testflight/. Acesso em: 23/03/2017.

ARZT, S. et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, ACM, v. 49, n. 6, p. 259–269, 2014.

AWS. *Amazon Compute Service Level Agreement*. 2018. https://aws.amazon.com/compute/sla/. Accessed on: 04/27/2018.

AZIM, T.; NEAMTIU, I. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. [S.l.: s.n.], 2013.

AZIM, T.; NEAMTIU, I. Targeted and depth-first exploration for systematic testing of android apps. In: ACM. *Acm Sigplan Notices*. [S.l.], 2013. v. 48, n. 10, p. 641–660.

BAI, X. et al. Vee@ cloud: The virtual test lab on the cloud. In: IEEE. *Automation of Software Test (AST), 2013 8th International Workshop on*. [S.l.], 2013. p. 15–18.

BASU, A. *Software quality assurance, testing and metrics*. [S.l.]: PHI Learning Pvt. Ltd., 2015.

BEIZER, B. *Black-box testing: techniques for functional testing of software and systems*. [S.l.]: Wiley New York, 1995.

BERNER, S.; WEBER, R.; KELLER, R. K. Observations and lessons learned from automated testing. In: ACM. *Proceedings of the 27th international conference on Software engineering*. [S.l.], 2005. p. 571–579.

BO, J.; XIANG, L.; XIAOPENG, G. Mobiletest: A tool supporting automatic black box test for software on smart mobile devices. In: IEEE COMPUTER SOCIETY. *Proceedings of the Second International Workshop on Automation of Software Test*. [S.l.], 2007. p. 8.

BOEHM, B. W. Software engineering. *xyz*, ACM, v. 30, n. 5, p. 416–429, 1976.

BOSSAVIT, L. *The Leprechauns of Software Engineering*. [S.l.]: Lulu. com, 2015.

BROWSERSTACK. *Android Emulators vs Real Devices*. 2019. Project Web Page. Available at: https://www.browserstack.com/test-on-android-emulator. Accessed on: 04/04/2019.

BUGCROWD. *BugCrowd*. 2018. Page. Disponível em: https://www.bugcrowd.com/. Acesso em: 23/03/2017.

BUYYA, R.; RANJAN, R.; CALHEIROS, R. N. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In: IEEE. *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*. [S.l.], 2009. p. 1–11.

BUZZELL, R. D.; GALE, B. T.; SULTAN, R. G. Market share-a key to profitability. *Harvard business review*, January–February, v. 53, n. 1, p. 97–106, 1975.

CHOI, W.; NECULA, G.; SEN, K. Guided gui testing of android apps with minimal restart and approximate learning. In: ACM. *Acm Sigplan Notices*. [S.l.], 2013. v. 48, n. 10, p. 623–640.

COHN, M. *Succeeding with agile: software development using Scrum*. [S.l.]: Pearson Education, 2010.

Corona Labs. *Corona Framework*. 2017. Project Web Page. `https://coronalabs.com/`. Accessed on: 03/20/2018.

COUNTERPOINTER. *CounterPointer*. 2018. Project Web Page. `https://goo.gl/XZsbZF`. Accessed on: 02/28/2018.

CROWDSOURCEDTESTING. *CrowdSourcedTesting*. 2018. Page. Disponível em: `https://crowdsourcedtesting.com/en`. Acesso em: 23/03/2017.

Cucumber. 2015. `https://cukes.info/`.

DoorDash. *DoorDash*. 2013. Project Web Page. Available at: `https://www.doordash.com/`. Accessed on: 10/04/2017.

ELBERZHAGER, F. et al. Reducing test effort: A systematic mapping study on existing approaches. *Information and Software Technology*, Elsevier, v. 54, n. 10, p. 1092–1106, 2012.

ELLRAM, L. M.; SIFERD, S. P. Total cost of ownership: a key concept in strategic cost management decisions. *Journal of business logistics*, Blackwell Publishing Ltd., v. 19, n. 1, p. 55, 1998.

EVERETT, G. D.; JR, R. M. *Software testing: testing across the entire software development life cycle*. [S.l.]: John Wiley & Sons, 2007.

FARIA, K. A. C.; FREITAS, E. N. d. A.; VINCENZI, A. M. R. Collaborative economy for testing cost reduction on android ecosystem. In: ACM. *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*. [S.l.], 2017. p. 11–18.

FAZZINI, M. et al. Barista: A technique for recording, encoding, and running platform independent android tests. In: IEEE. *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*. [S.l.], 2017. p. 01–11.

FAZZINI, M. et al. Barista: A technique for recording, encoding, and running platform independent android tests. In: IEEE. *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. [S.l.], 2017. p. 149–160.

FENG, Y. et al. Multi-objective test report prioritization using image understanding. In: IEEE. *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. [S.l.], 2016. p. 202–213.

FOWLER, M. *TestPyramid*. 2012. Web Page. Available at: http://goo.gl/VbrNqF. Accessed on: 05/15/2015.

FOWLER, M. *The Practical Test Pyramid*. 2018. Project Web Page. https://martinfowler.com/articles/practical-test-pyramid.html. Accessed on: 03/20/2018.

FOX, A.; BREWER, E. A. Harvest, yield, and scalable tolerant systems. In: IEEE. *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*. [S.l.], 1999. p. 174–178.

FREITAS, E. N. et al. Amt: An android mirror tool for instant feedback across platform. In: SBC. *Congresso Brasileiro de Software (CBSoft), 2016 7th Congresso Brasileiro de Software on*. [S.l.], 2016. p. 429–440.

GAO, J.; BAI, X.; TSAI, W.-T. Cloud testing-issues, challenges, needs and practice. *Software Engineering: An International Journal*, v. 1, n. 1, p. 9–23, 2011.

GAO, J. et al. Testing as a service (taas) on clouds. In: IEEE. *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*. [S.l.], 2013. p. 212–223.

GAO, J. et al. Mobile application testing: a tutorial. *Computer*, IEEE, n. 2, p. 46–55, 2014.

GAO, J. et al. A cloud-based taas infrastructure with tools for saas validation, performance and scalability evaluation. In: IEEE. *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. [S.l.], 2012. p. 464–471.

GELPERIN, D.; HETZEL, B. The growth of software testing. *Communications of the ACM*, Citeseer, v. 31, n. 6, p. 687–695, 1988.

GOMEZ, L. et al. Reran: Timing-and touch-sensitive record and replay for android. In: IEEE PRESS. *Proceedings of the 2013 International Conference on Software Engineering*. [S.l.], 2013. p. 72–81.

GOOGLE. *Android Debug Bridge*. 2010. Project Web Page. https://goo.gl/ysZHAC. Accessed on: 03/20/2018.

Google. *Google Play Films*. 2015. Project Web Page. Available at: https://play.google.com/store/movies. Accessed on: 01/04/2017.

GOOGLE. *Getevent*. 2017. Page. Disponível em: https://source.android.com/devices/input/getevent. Acesso em: 23/03/2017.

Google. *Support Testing Library*. 2017. Project Web Page. Available at: https://developer.android.com/topic/libraries/testing-support-library/?hl=pt-br. Accessed on: 25/07/2017.

GOOGLE. *Android SDK*. 2018. Project Web Page. https://developer.android.com/studio/releases/sdk-tools.html. Accessed on: 03/20/2018.

GOOGLE. *Android Studio User Guide*. 2018. Project Web Page. https://developer.android.com/studio/build. Accessed on: 03/12/2018.

GOOGLE. *Fundamentals of Testing*. 2018. Project Web Page. https://developer.android.com/training/testing/fundamentals. Accessed on: 03/20/2018.

Google. *Google Cloud Platform*. 2018. https://cloud.google.com. Accessed on: 04/27/2018.

GOOGLE. *Google Maps*. 2018. Project Web Page. https://www.google.com/maps/. Accessed on: 03/20/2018.

Google Cloud. *Google Compute Engine SLA*. 2018. https://cloud.google.com/compute/sla. Accessed on: 04/27/2018.

Google Firebase. *Firebase Test Lab*. 2018. Project Web Page. https://goo.gl/7GK9G9. Accessed on: 02/28/2018.

Google Inc. *Espresso*. 2013. Project Web Page. https://goo.gl/pzjgQS. Accessed on: 02/28/2018.

Google Inc. *Espresso Test Recorder*. 2018. https://developer.android.com/studio/test/espresso-test-recorder. Accessed on: 04/27/2018.

Google Inc. *UI Automator*. 2018. Project Web Page. https://goo.gl/fw4bJV. Accessed on: 02/28/2018.

Google Inc. *UI/Application Exerciser Monkey*. 2018. https://developer.android.com/studio/test/monkey. Accessed on: 04/04/2019.

Google Inc. *Test UI for a single app*. 2019. Project Web Page. https://developer.android.com/training/testing/ui-testing/espresso-testing.html. Accessed on: 04/04/2019.

Google Play Store. *Google Play*. 2018. Project Web Page. Available at: https://play.google.com/store. Accessed on: 12/20/2018.

HALPERN, M. et al. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In: IEEE. *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. [S.l.], 2015. p. 215–224.

HAM, H. K.; PARK, Y. B. Mobile application compatibility test system design for android fragmentation. In: SPRINGER. *International Conference on Advanced Software Engineering and Its Applications*. [S.l.], 2011. p. 314–320.

HAO, S. et al. Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In: ACM. *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. [S.l.], 2014. p. 204–217.

HARGASSNER, W. et al. A script-based testbed for mobile software frameworks. In: IEEE. *Software Testing, Verification, and Validation, 2008 1st International Conference on*. [S.l.], 2008. p. 448–457.

HASSAN, A. E.; HOLT, R. C. The top ten list: Dynamic fault prediction. In: IEEE. *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. [S.l.], 2005. p. 263–272.

HAVELUND, K.; PRESSBURGER, T. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, Springer, v. 2, n. 4, p. 366–381, 2000.

HETZEL, B. The complete guide to software testing, qed information sciences. *Inc., Wellesley, MA*, 1988.

HU, C.; NEAMTIU, I. Automating gui testing for android applications. In: ACM. *Proceedings of the 6th International Workshop on Automation of Software Test*. [S.l.], 2011. p. 77–83.

HU, Y.; AZIM, T.; NEAMTIU, I. Versatile yet lightweight record-and-replay for android. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2015. v. 50, n. 10, p. 349–366.

Huawei Inc. *Huawei*. 2018. Project Web Page. `http://www.huawei.com/`. Accessed on: 02/28/2018.

IDC Inc. *IDC*. 2018. Project Web Page. `https://www.idc.com/promo/smartphone-market-share/vendor`. Accessed on: 02/28/2018.

INDEX.CO. *Sharing Economy*. 2017. Page. Disponível em: `https://index.co/market/sharing-economy/investors`. Acesso em: 23/03/2017.

IONIC. *Ionic Framework*. 2012. Project Web Page. `https://ionicframework.com/`. Accessed on: 03/20/2018.

JOORABCHI, M. E.; MESBAH, A.; KRUCHTEN, P. Real challenges in mobile app development. In: IEEE. *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.], 2013. p. 15–24.

JS Foundation. *Appium*. 2012. Project Web Page. `http://appium.io/`. Accessed on: 02/28/2018.

JUNIT. *JUnit*. 1998. Page. Disponível em: `https://junit.org`. Acesso em: 10/04/2019.

KAASILA, J. et al. Testdroid: automated remote UI testing on Android. In: *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*. [S.l.: s.n.], 2012.

KATHERINE, A. V.; ALAGARSAMY, K. Software testing in cloud platform: a survey. *International Journal of computer applications*, International Journal of Computer Applications, 244 5 th Avenue,# 1526, New . . . , v. 46, n. 6, p. 21–25, 2012.

KEMERER, C. F. An empirical validation of software cost estimation models. *Communications of the ACM*, ACM, v. 30, n. 5, p. 416–429, 1987.

KHALID, H. et al. Prioritizing the devices to test your app on: A case study of android game apps. In: ACM. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2014. p. 610–620.

KIM, S. et al. Predicting faults from cached history. In: IEEE COMPUTER SOCIETY. *Proceedings of the 29th international conference on Software Engineering*. [S.l.], 2007. p. 489–498.

KOBITON. *Kobiton*. 2018. Project Web Page. `https://kobiton.com/`. Accessed on: 02/28/2018.

KOTLER, P.; KARTAJAYA, H.; SETIAWAN, I. *Marketing 3.0: From products to customers to the human spirit*. [S.l.]: John Wiley & Sons, 2010.

LIN, Y.-D. et al. Improving the accuracy of automated gui testing for embedded systems. *Software, IEEE*, IEEE, v. 31, n. 1, p. 39–45, 2014.

LIN, Y.-D. et al. On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering*, IEEE, v. 40, n. 10, p. 957–970, 2014.

LIU, C.-H. et al. Capture-replay testing for android applications. In: IEEE. *Computer, Consumer and Control (IS3C), 2014 International Symposium on*. [S.l.], 2014. p. 1129–1132.

LU, X. et al. Prada: Prioritizing android devices for apps by mining large-scale usage data. In: ACM. *Proceedings of the 38th International Conference on Software Engineering*. [S.l.], 2016. p. 3–13.

LYDIA, L. et al. *Magic Quadrant for Cloud Infrastructure as a Service, Worldwide*. 2017. `https://www.gartner.com/doc/reprints?id=1-2G2O5FC&ct=150519`. Accessed on: 03/28/2018.

Lyft. *Lyft*. 2012. Project Web Page. Available at: `https://www.lyft.com/`. Accessed on: 10/04/2017.

MACHIRY, A.; TAHILIANI, R.; NAIK, M. Dynodroid: An input generation system for android apps. In: ACM. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. [S.l.], 2013. p. 224–234.

MAHMOOD, R.; MIRZAEI, N.; MALEK, S. EvoDroid: Segmented Evolutionary Testing of Android Apps. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2014.

MALDONADO, J.; DELAMARO, M.; VINCENZI, A. M. R. *Automatização de teste de software com ferramentas de software livre*. [S.l.]: Elsevier Brasil, 2018.

MAO, K. et al. A survey of the use of crowdsourcing in software engineering. *Rn*, v. 15, n. 01, 2015.

MAO, K.; HARMAN, M.; JIA, Y. Sapienz: Multi-objective automated testing for android applications. In: ACM. *Proceedings of the 25th International Symposium on Software Testing and Analysis*. [S.l.], 2016. p. 94–105.

MAO, K.; HARMAN, M.; JIA, Y. Robotic testing of mobile apps for truly black-box automation. *IEEE Software*, IEEE, v. 34, n. 2, p. 11–16, 2017.

MARIJAN, D. Multi-perspective regression test prioritization for time-constrained environments. In: IEEE. *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. [S.l.], 2015. p. 157–162.

MARTENS, B.; WALTERBUSCH, M.; TEUTEBERG, F. Costing of cloud computing services: A total cost of ownership approach. In: IEEE. *System Science (HICSS), 2012 45th Hawaii International Conference on*. [S.l.], 2012. p. 1563–1572.

MEMON, A. et al. The first decade of gui ripping: Extensions, applications, and broader impacts. In: IEEE. *Reverse Engineering (WCRE), 2013 20th Working Conference on*. [S.l.], 2013. p. 11–20.

MERWE, H. van der; MERWE, B. van der; VISSER, W. Execution and property specifications for jpf-android. *ACM SIGSOFT Software Engineering Notes*, ACM, v. 39, n. 1, p. 1–5, 2014.

MICHAEL, R. L. Handbook of software reliability engineering. *McGraw Hill and IEEE Society Press*, 1996.

Microsoft. *Microsoft Azure*. 2018. `https://azure.microsoft.com`. Accessed on: 04/27/2018.

Microsoft Azure. *SLA for Virtual Machines*. 2018. `https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1_8/`. Accessed on: 04/27/2018.

MISHRA, D.; MAHANTY, B. A study of software development project cost, schedule and quality by outsourcing to low cost destination. *Journal of Enterprise Information Management*, Emerald Group Publishing Limited, v. 29, n. 3, p. 454–478, 2016.

MoQuality Inc. *MoQuality*. 2019. Project Web Page. `https://www.moquality.com`. Accessed on: 04/04/2019.

MUCCINI, H.; FRANCESCO, A. D.; ESPOSITO, P. Software testing of mobile applications: Challenges and future research directions. In: IEEE PRESS. *Proceedings of the 7th International Workshop on Automation of Software Test*. [S.l.], 2012. p. 29–35.

MYERS, G. J.; SANDLER, C.; BADGETT, T. The art of software testing, john wiley& sons. *Inc, Canada*, 1979.

NIDHRA, S.; DONDETI, J. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, v. 2, n. 2, p. 29–50, 2012.

NIKOLIK, B. Software quality assurance economics. *Information and Software Technology*, Elsevier, v. 54, n. 11, p. 1229–1238, 2012.

OPENSIGNAL. *Android Fragmentation*. 2015. Project Web Page. `https://opensignal.com/reports/2015/08/android-fragmentation/`. Accessed on: 03/05/2018.

OPPO Inc. *OPPO*. 2018. Project Web Page. `https://www.oppo.com`. Accessed on: 02/28/2018.

ORACLE. *Capacity Planning and Deployment Guide*. [S.l.], 2007. https://docs.oracle.com/cd/E11116_04/otn/pdf/install/E11130_01.pdf. Accessed on: 03/28/2018.

OSTRAND, T. J.; WEYUKER, E. J.; BELL, R. M. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, IEEE, v. 31, n. 4, p. 340–355, 2005.

PAN, J. Software testing. *Dependable Embedded Systems*, v. 5, p. 2006, 1999.

PASSBRAINS. *passbrains Crowd Testing*. 2018. Page. Disponível em: https://www.passbrains.com/. Acesso em: 23/03/2017.

PAY4BUGS. *Pay4Bugs*. 2018. Page. Disponível em: https://blog.pay4bugs.com/. Acesso em: 23/03/2017.

PERFECTO. *Perfecto*. 2018. Project Web Page. https://www.perfectomobile.com. Accessed on: 02/28/2018.

RAMLER, R.; WOLFMAIER, K. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In: ACM. *Proceedings of the 2006 international workshop on Automation of software test*. [S.l.], 2006. p. 85–91.

RAY, M.; MOHAPATRA, D. P. Code-based prioritization: a pre-testing effort to minimize post-release failures. *Innovations in Systems and Software Engineering*, Springer, v. 8, n. 4, p. 279–292, 2012.

RIDENE, Y.; BARBIER, F. A model-driven approach for automating mobile applications testing. In: ACM. *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. [S.l.], 2011. p. 9.

RIFKIN, J. *The age of access: The new culture of hypercapitalism*. [S.l.]: Penguin, 2001.

RIFKIN, J.; PEREIRA, M. S. *A era do acesso: a revolução da nova economia*. [S.l.: s.n.], 2001.

RIUNGU, L. M.; TAIPALE, O.; SMOLANDER, K. Software testing as an online service: Observations from practice. In: IEEE. *Third International Conference on Software Testing, Verification, and Validation Workshops*. [S.l.], 2010. p. 418–423.

Robert Half. *Salary Guide 2018*. [S.l.], 2018. https://www.roberthalf.com.br/guia-salarial. Accessed on: 02/28/2018.

ROBOELETRIC. *Robolectric*. 2018. Project Web Page. http://robolectric.org/. Accessed on: 03/20/2018.

ROBOTIUM. *Robotium Framework*. 2019. Project Web Page. http://www.robotium.org. Accessed on: 04/04/2019.

SASNAUSKAS, R.; REGEHR, J. Intent fuzzer: crafting intents of death. In: ACM. *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. [S.l.], 2014. p. 1–5.

SATOH, I. A testing framework for mobile computing software. *IEEE Transactions on Software Engineering*, IEEE, v. 29, n. 12, p. 1112–1121, 2003.

SATOH, I. Software testing for wireless mobile computing. *IEEE Wireless Communications*, IEEE, v. 11, n. 5, p. 58–64, 2004.

Sauce Labs. *Sauce Labs*. 2018. Project Web Page. `https://saucelabs.com/enterprise#mobile-testing`. Accessed on: 02/28/2018.

SAUCELABS. *The Top 5 Android UI Frameworks for Automated Testing*. 2019. Project Web Page. `https://www.wandoujia.com/`. Accessed on: 04/04/2019.

SCHNEIDER, C.; CHEUNG, T. The power of the crowd: Performing usability testing using an on-demand workforce. In: *Information Systems Development*. [S.l.]: Springer, 2013. p. 551–560.

Spoon. 2015. `http://square.github.io/spoon`.

STATCOUNTER. *StatCounter GlobalStats*. 2018. Project Web Page. `http://gs.statcounter.com/os-market-share/mobile/worldwide`. Accessed on: 03/12/2018.

Statista. *Number of available applications in the Google Play Store from December 2009 to December 2018*. 2019. Web Page. Available at: `http://goo.gl/ul6G08`. Accessed on: 09/15/2015.

SUNDARARAJAN, A. *The sharing economy: The end of employment and the rise of crowd-based capitalism*. [S.l.]: Mit Press, 2016.

SUTHERLAND, J.; SUTHERLAND, J. *Scrum: the art of doing twice the work in half the time*. [S.l.]: Currency, 2014.

TAKAGI, T.; FURUKAWA, Z.; YAMASAKI, T. An overview and case study of a statistical regression testing method for software maintenance. *Electronics and Communications in Japan (Part II: Electronics)*, Wiley Online Library, v. 90, n. 12, p. 23–34, 2007.

TAO, C.; GAO, J. Modeling mobile application test platform and environment: testing criteria and complexity analysis. In: ACM. *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*. [S.l.], 2014. p. 28–33.

TAO, C.; GAO, J. On building a cloud-based mobile testing infrastructure service system. *Journal of Systems and Software*, Elsevier, v. 124, p. 39–55, 2017.

TARJAN, R. Depth-first search and linear graph algorithms. *SIAM journal on computing*, SIAM, v. 1, n. 2, p. 146–160, 1972.

TÉBOUL, J. *A era dos serviços: uma nova abordagem de gerenciamento*. [S.l.]: Qualitymark Editora Ltda, 1999.

TESTBIRDS. *TestBirds Crowd Testing*. 2018. Page. Disponível em: `https://www.testbirds.com/`. Acesso em: 23/03/2017.

TheAppBuilder Ltd. *The App Builder Framework*. 2016. Project Web Page. https://www.theappbuilder.com/. Accessed on: 03/20/2018.

TSAI, W.-T.; HUANG, Y.; SHAO, Q. Testing the scalability of saas applications. In: IEEE. *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*. [S.l.], 2011. p. 1–4.

Turo. *Turo*. 2009. Project Web Page. Available at: https://turo.com/. Accessed on: 10/04/2017.

Uber. *Uber*. 2008. Project Web Page. Available at: https://www.uber.com. Accessed on: 01/04/2017.

VASQUEZ, M. L. et al. How do developers test android applications? *arXiv preprint arXiv:1801.06268*, 2018.

VILKOMIR, S.; AMSTUTZ, B. Using combinatorial approaches for testing mobile applications. In: IEEE. *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*. [S.l.], 2014. p. 78–83.

VILKOMIR, S. et al. Effectiveness of multi-device testing mobile applications. In: IEEE. *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on*. [S.l.], 2015. p. 44–47.

Vivo Inc. *Vivo*. 2018. Project Web Page. https://www.vivo.com/en/. Accessed on: 02/28/2018.

WANDOUJIA. *Chinese Android Apps Management System*. 2018. Project Web Page. https://www.wandoujia.com/. Accessed on: 04/04/2019.

WEBKIT. *Webkit*. 2018. Project Web Page. https://webkit.org/. Accessed on: 03/20/2018.

WEI, L.; LIU, Y.; CHEUNG, S.-C. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In: IEEE. *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. [S.l.], 2016. p. 226–237.

WEISS, C. et al. How long will it take to fix this bug? In: IEEE. *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*. [S.l.], 2007. p. 1–1.

WEYUKER, E.; OSTRAND, T.; BELL, R. Using static analysis to determine where to focus dynamic testing effort. In: *Proceedings of the IEEE Second International Workshop on Dynamic Analysis*. [S.l.: s.n.], 2004. p. 1–8.

WHITTAKER, J. A.; ARBON, J.; CAROLLO, J. *How Google tests software*. [S.l.]: Addison-Wesley, 2012.

XAMARIN. *Xamarin Test Cloud*. 2018. Project Web Page. https://www.xamarin.com/test-cloud. Accessed on: 02/28/2018.

Xamarin Inc. *Xamarin*. 2015. Project Web Page. http://calaba.sh/. Accessed on: 02/28/2018.

Xiaomi Inc. *Xiaomi*. 2018. Project Web Page. http://www.mi.com/. Accessed on: 02/28/2018.

YANG, M. C.; CHAO, A. Reliability-estimation and stopping-rules for software testing, based on repeated appearances of bugs. *IEEE Transactions on Reliability*, IEEE, v. 44, n. 2, p. 315–321, 1995.

YANG, W.; PRASAD, M. R.; XIE, T. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In: *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*. [S.l.: s.n.], 2013.

YANG, Y. et al. Testqual: conceptualizing software testing as a service. *e-Service Journal: A Journal of Electronic Services in the Public and Private Sectors*, JSTOR, v. 7, n. 2, p. 46–65, 2011.

YE, H. et al. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In: ACM. *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. [S.l.], 2013. p. 68.

YOON, M. et al. A test case prioritization through correlation of requirement and risk. *Journal of Software Engineering and Applications*, Scientific Research Publishing, v. 5, n. 10, p. 823, 2012.

YU, L. et al. Testing as a service over cloud. In: IEEE. *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*. [S.l.], 2010. p. 181–188.

ZADGAONKAR, H. *Robotium Automated Testing for Android*. [S.l.]: Packt Publishing Ltd, 2013.

ZHANG, T. et al. Testing location-based function services for mobile applications. In: IEEE. *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*. [S.l.], 2015. p. 308–314.

ZHENG, C. et al. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In: ACM. *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. [S.l.], 2012. p. 93–104.