UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ROGER RITTER

Reúso de cenários BDD para minimizar o esforço de migração de testes para a plataforma Android

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

Orientadora: Profª. Dra. Érika Cota

Porto Alegre
2018

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Ritter, Roger
Reúso de cenários BDD para minimizar o esforço de migração de testes para a plataforma Android / Roger Ritter -- 2018.
58 f.

Orientadora: Érika Fernandes Cota;

Dissertação (Mestrado) -- Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2018.

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

AGRADECIMENTOS

Primeiramente agradeço a Deus, por estar sempre junto comigo e guiar meus passos, por colocar pessoas sempre tão boas e honestas no meu caminho e ter proporcionado saúde para que chegasse até aqui. Agradeço aos meus pais (Eunice e Sérgio), por a cada momento apoiar minhas decisões e pela paciência e harmonia que ensinaram para viver a cada dia. Agradeço, e sou muito feliz, por ter feito muitos amigos neste caminho em busca desta titulação, pessoas que realmente fizeram e fazem a diferença em minha vida. Um agradecimento em especial a Profª. Érika, minha orientadora, por toda sua dedicação e paciência que teve comigo, desde o início do programa.

Ninguém vence sozinho! Obrigado a todos.

RESUMO

O desenvolvimento de versões móveis de sistemas corporativos que já executam em plataformas Desktop e/ou Web tem se tornado comum. No entanto, o processo de migração tanto da lógica de programação quanto dos testes pode ser bastante complexo, embora muitas funcionalidades permaneçam as mesmas no novo ambiente. Este trabalho propõe o reúso de cenários de teste automatizados como uma alternativa para diminuir este esforço de migração. Para isso, propõe-se uma metodologia para o reúso de cenários de teste suportada por um framework de automação de testes. A metodologia propõe que cenários BDD sejam escritos uma única vez e executados em diferentes plataformas, como Desktop, Web, Móvel ou outra que venha a existir. Para dar suporte à metodologia proposta, o framework dbehave foi estendido para permitir a execução de cenários de teste em plataformas móveis. Uma segunda extensão no framework permite ainda que cenários específicos de uma plataforma possam ser escritos junto aos demais cenários mas executados apenas na plataforma de interesse, permitindo ao desenvolvedor uma maior autonomia na organização e manutenção dos cenários. A metodologia proposta foi utilizada em dois estudos de caso e se mostrou útil, uma vez que uma média de 81.2% dos cenários de aplicações reais foram reutilizados, havendo uma redução considerável no esforço de migração entre plataformas e na escrita de cenários.

Palavras-chaves: Android; Teste de Aceitação; Behavior Driven Development; Teste de Sistema; Teste de Software.

Minimizing the migration effort for the existing Android BDD platform

ABSTRACT

The development of enterprise applications in multiple platforms (Desktop and/or Web and/or Mobile) has become a trend. However, the process of migrating both programming logic and software tests can be very complex, although many functionalities remain the same in the new environment. This work proposes the reuse of automated test scenarios as an alternative to reduce this migration effort. We propose a test methodology that is supported by a test automation framework. The methodology proposes the developer writes BDD scenarios only once and executes such scenarios on different platforms, such as Desktop, Web, Mobile or other that may exist. The dbehave framework was extended to support the execution of test scenarios in mobile platforms. Furthermore, the framework now allows the selection of which scenarios should be executed in which platforms, i.e., platform-specific scenarios can be written next to the other scenarios and run only on the platform of interest. This provides the developer greater autonomy in the organization and maintenance of the scenarios. The proposed methodology was used in two case studies and proved useful, since an average of 81.2% of the real application scenarios were reused, with a considerable reduction in the effort for cross-platform migration and scenario writing.

Keywords: Android; Acceptance Testing; Behavior Driven Development; System Testing; Software Testing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

GUI	Graphical User Interface
BDD	Behavior Driven Development
DSL	Domain-Specific Language
US	User Story
HTTP	Hyper Text Markup Language
TAL	Test Automation Layer
SDK	Software Development Kit
IDE	Integrated Development Environment
API	Application Programming Interface

SUMÁRIO

1 INTRODUÇÃO

Da mesma forma que, há alguns anos, houve uma migração de aplicações da plataforma Desktop para Web, hoje observa-se uma forte tendência de migração da plataforma Web para a plataforma Móvel (Worldwide Mobile App, 2017). Atualmente, muitas organizações já adotam o conceito de *Mobile First,* onde a primeira versão da aplicação é desenvolvida para a plataforma móvel e, posteriormente, a versão Web é disponibilizada. Esta estratégia responde a uma clara demanda do mercado por flexibilidade, alta disponibilidade e acessibilidade de um sistema.

Migrar uma aplicação para outra plataforma de desenvolvimento não é uma tarefa trivial. Embora muitas funcionalidades devam ser mantidas em ambos ambientes, as diferenças tecnológicas exigem considerável retrabalho (O'Brien, 2004). Além disso, novas plataformas trazem novas formas de executar as mesmas funcionalidades. Na plataforma móvel, por exemplo, há uma gama maior de sensores e recursos que podem substituir a entrada manual de dados (por exemplo, o uso da câmera para leitura de código de barras ou o microfone para interpretar um comando). Tais recursos naturalmente exigem a implementação de funcionalidades adicionais. Por outro lado, a expectativa dos usuários em relação à qualidade da aplicação é bem maior, pois esperam encontrar um sistema maduro e que explore bem os recursos disponíveis na nova plataforma. Dessa forma, a etapa de testes de sistema e aceitação tem papel essencial para o sucesso de um processo de migração. No entanto, dificuldades também são encontradas na migração dos testes automatizados.

A migração dos testes automatizados para uma nova plataforma envolve não apenas a decisão de quais testes podem/devem ser reutilizados, mas também a adequação desses testes às características e restrições da nova plataforma. Testes de sistema e aceitação são fortemente baseados na interface com o usuário, tipicamente gráfica (*Graphical User Interface* - GUI) e, portanto, altamente sujeita a mudanças na nova plataforma. Assim, a ferramenta que automatiza a interação dos testes com a aplicação (ou seja, que substitui a ação do usuário durante os testes funcionais) deve ser também adaptada à nova plataforma, de forma a reconhecer e interagir com os novos objetos ou recursos disponíveis. Essa tarefa pode ser bastante custosa, principalmente se não existir padronização tanto de requisitos do software quanto de casos de testes.

Uma técnica que dá ênfase ao comportamento do ponto de vista do usuário e auxilia na

definição dos testes de aceitação é o *Behavior Driven Development*, ou BDD. Agregando mais agilidade e qualidade ao desenvolvimento de software (Dan North, 2017), o BDD é fundamentado nas histórias de usuários e consiste em cenários estruturados de fácil entendimento, sem o uso de jargões técnicos. A linguagem comum do BDD é o Gherkin (Dan North, 2017) que, com sintaxe e semântica simples, define um padrão universal para documentação de cenários de uso de um sistema. No BDD, esta especificação é feita de maneira independentemente da plataforma de execução, focando-se no comportamento do software e sem referência a elementos de interface específicos.

Embora a automação não seja a principal motivação para o uso do BDD, ela está fortemente associada a essa técnica devido à estrutura simples e regular dos cenários que acabam sendo muito propensos à automação. Assim, não é incomum ver cenários BDD com referências a elementos de interface, principalmente quando utilizados como base para testes de aceitação automatizados. Pode-se dizer que esses cenários são de mais baixo-nível, mas ainda agregam valor à especificação por serem menos ambíguos.

Existem ferramentas para a automação de cenários em diferentes linguagens e plataformas, como por exemplo, JBehave (para Java) e Cucumber (para Ruby) e é possível combiná-las com ferramentas que automatizam o papel do usuário durante os testes como Selenium (para aplicações Web) e Fest (para aplicações Desktop). Entretanto, tipicamente, cada ferramenta de automação atende a uma única plataforma de execução. Assim, na prática, há restrições para as combinações possíveis pela limitação do ambiente de desenvolvimento assumido por cada uma.

Atender a mais de uma plataforma significa possibilitar a reutilização de cenários de teste automatizados, ou seja, permitir que um único cenário de uso da aplicação seja executado automaticamente tanto no navegador de um computador, quanto na interface nativa de um celular, sem a necessidade de reescrita do cenário.

No entanto, o cenário automatizado é normalmente descrito em termos de elementos de interface, já prevendo a automação em uma plataforma específica. Desta forma, quando existe a migração do software para uma nova plataforma, a única alternativa para o desenvolvedor é reescrever os cenários usando os elementos de interface da nova plataforma e implementar novamente os testes para a nova plataforma, o que gera considerável re-trabalho. Ainda assim, muitas vezes, desenvolvedores utilizam duas ferramentas (uma para a plataforma Web e outra para a plataforma Móvel, por exemplo).

1.1 OBJETIVOS

Este trabalho propõe o reúso de cenários de teste automatizados entre plataformas de execução de natureza distinta, ou seja, entre plataformas Desktop/Web e plataformas Móveis. A estratégia de reúso é baseada na descrição de cenários BDD, preferencialmente sem referências a elementos específicos de interface. Propõe-se ainda, como parte da abordagem, a extensão de uma ferramenta de automação de cenários BDD para possibilitar a reutilização de cenários entre as plataformas Desktop/Web e Móvel. Através desta abordagem, é possível escrever um cenário da aplicação uma única vez e executá-lo, automaticamente, em três plataformas distintas (Web, Desktop e Android). É possível ainda, definir cenários que se aplicam a apenas uma das plataformas e executá-los, automaticamente, apenas na plataforma de interesse.

A validação da abordagem proposta e sua ferramenta de suporte foi feita através de dois estudos de caso: uma aplicação financeira de uso corporativo que está em processo de migração da plataforma Web para a plataforma Android e a aplicação Wordpress, uma aplicação *open source* para a criação de notícias em Blogs e Websites.

1.2 ORGANIZAÇÃO DO TEXTO

No Capítulo 2 os conceitos básicos como as abordagens *Mobile First*, *Behavior-driven Development* (BDD) e execução automática de cenários são apresentados e discutidos. Ainda neste capítulo, são definidos os conceitos de multiplataforma e reúso de testes usados nesse trabalho. Por fim, são discutidos os trabalhos relacionados.

O Capítulo 3 apresenta um estudo de ferramentas de automação de teste para a plataforma Android. Um estudo de caso é apresentado e usado como base para a seleção da ferramenta que poderia dar suporte à metodologia proposta neste trabalho.

O Capítulo 4 apresenta o MBehavior, o framework de suporte à metodologia proposta. Entre os conceitos abordados neste capítulo está a ferramenta dbehave, que foi expandida até originar o framework proposto. Também é discutido neste capítulo, como foi o processo de evolução do dbehave para o MBehavior, juntamente com a escolha de um *Runner* para implementação de cenários na plataforma Android.

O Capítulo 5 apresenta a validação da metodologia proposta, em dois estudos de caso

utilizando o MBehavior.

Por último, o Capítulo 6 apresenta as conclusões juntamente com os trabalhos futuros que podem ser realizados para dar continuidade a este trabalho.

2 CONCEITOS BÁSICOS

Neste capítulo os conceitos básicos como a abordagem *Mobile First*, *Behavior Driven Development* e a execução automática de cenários são apresentadas e discutidas. São conhecidos também conceitos de multiplataforma relacionados a este trabalho e o reúso de código de automação de testes. Por fim, são discutidos neste capítulo, os trabalhos relacionados.

2.1 ABORDAGEM *MOBILE FIRST*

Em 2020 estima-se que haverá mais de 2.87 bilhões de dispositivos móveis no planeta (Worldwide Mobile App, 2017). Considerando que dispositivos móveis possuem uma capacidade de processamento, armazenamento, tráfego de dados e leiaute específicos, a forma de atendimento dessas demandas necessita de uma estratégia privilegiada, conhecida como *mobile-first*.

Esse conceito de desenvolvimento foca-se em construir a aplicação visando atender, primeiramente, o público de usuários de dispositivos móveis. Esse desenvolvimento é realizado com base em experiência diferenciada, que permite ações que não são comuns em outras plataformas como, por exemplo, a utilização dos polegares para dar *zoom* (efeito pinça e arrasto lateral (*swipe*)) ao invés da navegação por mouse e teclado entre outras. No entanto, a experiência do usuário não é o único foco nesse conceito. O fato das plataformas móveis possuírem infraestrutura própria de hardware (GPS, Wifi, Bluetooth e outros) permite que surjam aplicações que explorem tais recursos, havendo assim um maior grau de competitividade mercadológica quando comparado a plataformas Web e Desktop.

Um aplicativo móvel é um software que foi desenvolvido com o objetivo de ser instalado em um dispositivo móvel, como um *smartphone* ou *tablet*. Os aplicativos móveis são suportados por diferentes aparelhos e sistemas operacionais, como Android, iOS, Windows Phone e Firefox OS. Cada plataforma de execução (dispositivo + sistema operacional) possui características únicas que devem ser levadas em consideração na implementação de um novo aplicativo.

Na construção de uma aplicação para dispositivo móvel, a escolha do tipo do aplicativo é um ponto crucial no processo de desenvolvimento do projeto. Ao iniciar um

projeto é necessário obter a clareza sobre qual plataforma, sistemas, produtos e arquiteturas serão utilizados. Os principais tipos de aplicativos móveis a serem considerados no início de um projeto são:

- Aplicativos Web

Possuem o objetivo de serem utilizados através do navegador do próprio dispositivo, sem a necessidade de instalação de um novo software para o acesso.

- Aplicativos Nativos

São aplicações móveis desenvolvidas exclusivamente para um único sistema operacional móvel, como o Android ou iOS, por exemplo.

- Aplicativos Híbridos

É uma mescla de um aplicativo Web e Nativo. Executam em um navegador, porém são compilados para a plataforma alvo. Todavia, o código nativo representa uma pequena fração desta aplicação, que geralmente é utilizado de forma transparente para acessar recursos específicos, o restante é Web (Budiu, 2013).

Segundo a IDC Research (2016), em 2015, 82.8% dos usuários preferiam sistemas Android e, em segundo lugar, o iOS da Apple, representando 13.9% das preferências. Por esse motivo, Android é o sistema operacional móvel escolhido para esse trabalho. Assim, resumidamente, as plataformas de execução consideradas neste trabalho são: Desktop, Web e Android.

Android é o sistema operacional mais utilizado em dispositivos móveis há mais de 5 anos consecutivos, segundo IDC Research (2016). Apesar de inicialmente ter sido desenvolvido para *smartphones*, na atualidade, o sistema operacional Android está presente em vários dispositivos, como relógios, televisores, automóveis e outros (Android, 2017).

Os aplicativos Android são desenvolvidos na linguagem Java. Assim, destacam-se as características que esta linguagem proporciona, como o reúso, encapsulamento e polimorfismo. Um desenvolvedor de aplicativos não necessita de conhecimentos avançados na arquitetura Android para a implementação de um aplicativo. Ele pode fazer o uso do SDK (*Software Development Kit*), que é disponibilizado gratuitamente, para auxiliá-lo. Os SDKs geralmente são distribuídos juntamente com uma IDE (*Integrated Development Environment*).

No caso do Android a IDE é o Android Studio, onde o desenvolvedor escolhe qual é o dispositivo alvo para seu aplicativo, que pode ser um relógio (*Watch*), uma *SmartTV* ou um *Smartphone* (Android, 2017). Esse conjunto de facilidades permite que desenvolvedores iniciantes implementem novos aplicativos e o distribuam nas lojas virtuais de aplicativos.

Juntamente com este SDK, existem muitas APIs (*Application Programming Interface*) para criar um leiaute, para acionar dispositivos como câmera, wifi e *bluetooth*, entre outros. Essas APIs trazem uma maior agilidade e padronização ao desenvolvimento de um aplicativo. Com isso, ao mesmo tempo em que o desenvolvimento é mais ágil, o desenvolvedor é obrigado a seguir uma padronização proposta pela arquitetura Android (Android, 2017). Por outro lado, o não uso do SDK desafia o desenvolvedor a conhecer determinados componentes e recursos da arquitetura Android. Sem seguir a padronização proposta, é provável que este desenvolvedor cometa erros que poderão afetar a qualidade do aplicativo.

Atualmente, os aplicativos móveis podem ser adquiridos, em sua maioria, nas lojas virtuais de aplicativos. Na loja, é possível adquirir um determinado aplicativo, avaliar e realizar comentários que poderão ser vistos por outros usuários. Nesse contexto, aplicativos bem avaliados e recomendados tendem a obter mais *downloads* de que aplicativos com uma avaliação mais baixa. De fato, há uma grande variedade de aplicativos que compreendem as mesmas funcionalidades, portanto, havendo um maior número de aplicativos, os usuários possuem uma exigência maior.

Considerando-se que atualmente existe um grande número de aplicativos, e que os usuários possuem um nível alto de exigência, uma questão importante para o desenvolvedor é se seu aplicativo possui a qualidade desejada pelo usuário. Um dos principais desafios dos aplicativos móveis é a entrega de um determinado software com rapidez, mas sem comprometer a qualidade, ou seja, o tempo para disponibilizar um aplicativo para o mercado (*Time to Market*) deve ser cada vez menor, mas com uma qualidade satisfatória. Uma fábrica de software que utiliza este conceito obtém mais rentabilidade e ainda assim, desenvolve produtos com qualidade, tornando-se mais competitiva.

Obter uma maior velocidade no desenvolvimento de aplicativos, sem abrir mão da qualidade pode não ser uma tarefa trivial. Uma técnica de projeto que visa acelerar e qualificar o desenvolvimento de um aplicativo é o BDD (*Behavior Driven Development*), ou seja, o desenvolvimento de um software dirigido ao comportamento do usuário. Esta técnica será detalhada a seguir.

2.2 BEHAVIOR DRIVEN DEVELOPMENT

A técnica BDD descreve as regras de negócio de forma estruturada, dando ênfase ao comportamento do ponto de vista do usuário e agregando mais agilidade e qualidade ao desenvolvimento (Dan North, 2017). Através dessa técnica, a etapa de elicitação de requisitos busca exemplos de uso do sistema que definem um contexto (papel do usuário e estado do sistema em um determinado momento), uma ação do usuário e uma consequência esperada a partir desta ação naquele contexto. Esses exemplos de uso são descritos em uma linguagem de domínio específico (*Domain-Specific Language* - DSL) de alto nível, estruturada e livre de jargões técnicos.

Utilizando a DSL Gherkin (Gherkin, 2018), a técnica BDD tem como vantagem a facilidade de comunicação entre os diversos atores (pessoal técnico e não-técnico) envolvidos no desenvolvimento do sistema e, por consequência, a geração de uma especificação menos ambígua. Além disso, por gerar uma especificação em linguagem estruturada, a técnica BDD propicia a automação da execução dos cenários descritos, facilitando a definição de testes de aceitação (ou sistema) automatizados.

Uma funcionalidade de sistema, ou *Feature*, quando se utiliza a técnica BDD, corresponde a um conjunto de histórias do usuário (*Stories* ou *User Stories* - *US*) (Soares, 2011). As histórias de usuário fornecem um conjunto de características que serão entregues a um determinado sistema assim que forem concluídas, substituindo, dessa forma, o tradicional documento de requisitos (Soares, 2011). Um modelo de como uma história de usuário e seus cenários podem ser descritos é proposto por Dan North (2017) e exibido na Figura 2.1.

Figura 2.1 – Modelo de história do usuário proposto.

```
1 - [Title of Story] (Only one line)
2 - As a [Role]
3 - I want [Feature]
4 - So that I can [Benefits]
5 -
6 - Scenario 1: [Scenario Title]
7 - Given [Context]
8 - And [Some more contexts]....
9 - When [Event]
10 - Then [Outcome]
```

Fonte: Dan North (2017).

A Figura 2.2, apresenta um exemplo de uma história descrita de acordo com o modelo proposto por Dan North (2017).

Figura 2.2 – Exemplo de história de usuário que pode ser descrita em Gherkin.

```
1 - Verify that information on INF website is correct.
2 - As a guest
3 - I want to get information from INF website
4 - So that I can know the name of under-graduation courses
5 -
6 - Scenario 1: Check the INF website information
7 - Given I am at 'WebSite INF'
8 - When I click in 'Ciência da Computação'
9 - Then I should see the title 'Bacharelado em Ciências de Computação'
```

Fonte: Dan North (2017).

Pode-se observar na Figura 2.2 que não há nenhuma referência a aspectos tecnológicos (plataforma de execução, detalhes da interface com o usuário, etc) nem como o estado inicial do sistema para aquele determinado cenário foi alcançado.

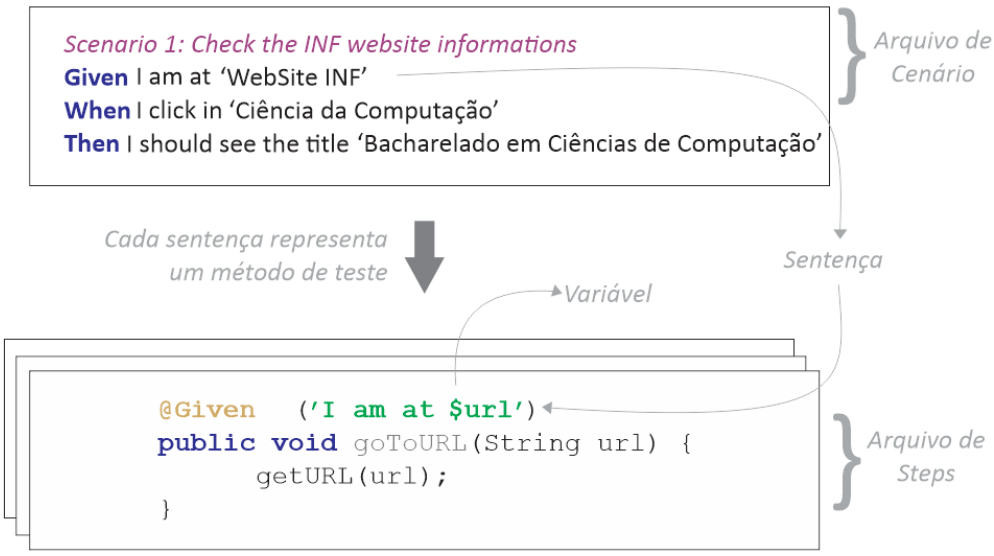
Assim, como um conjunto de cenários descreve a visão do usuário e são considerados critérios de aceitação do software, é interessante que sejam executados automaticamente (Smart, 2015).

Segundo Dan North (2017), é necessário que o solicitante das regras de negócio realize o aceite dessas regras tornando-as assim um importante artefato de entrada para os testes de aceitação (uma vez que as regras já foram 'aceitas'). Quando isso não ocorre, pode-se dizer que os cenários descrevem testes de sistema, e não de aceitação.

2.2.1 Execução Automática de Cenários

Cada linha de um cenário é denominada de sentença e reflete um contexto, ação ou evento (Smart, 2015). Cada sentença descreve um passo (*Step*) da execução de um teste de aceitação. Para que sejam executadas automaticamente, essas sentenças devem corresponder a métodos de teste que simulam a interação do usuário com a aplicação. A Figura 2.3 mostra um trecho de uma implementação em Java do passo que representa o contexto do cenário apresentado na Figura 2.2.

Figura 2.3 – Exemplo de classe em Java de um passo/*Step* do teste de aceitação.



Fonte: O próprio Autor.

Pode-se observar, no exemplo da Figura 2.3, que a anotação *@Given* vem acompanhada de um rótulo idêntico ao que aparece no cenário original ("*I am at*"). Ainda neste exemplo, o rótulo aparece acompanhado de um parâmetro (*$url*) que conterá o endereço de uma página Web. Assim, se o mesmo rótulo "*I am at*" aparecer em outra sentença *@Given* de algum outro cenário, o mesmo método pode ser chamado mesmo que este contexto se refira a uma outra página. Da mesma forma, para cada passo de cada cenário deve existir um método, parametrizado ou não, que implemente aquela ação. O arquivo de *Steps* contém, então, métodos do tipo Contexto (*@given*), Evento (*@when*) ou Resultado (*@then*) que também pode ser exemplificado, para um melhor entendimento, como: 'Dado que [estou em um determinado contexto]', 'Quando [executa-se uma determinada ação]' e 'Então [um determinado resultado deve ser apresentado]'.

Por fim, o uso de E (ou AND), representando que a sentença anterior deve ser continuada, e o uso de MAS (ou BUT), representando uma extensão negativa de um Resultado (*@then*), também está previsto como parte das sentenças.

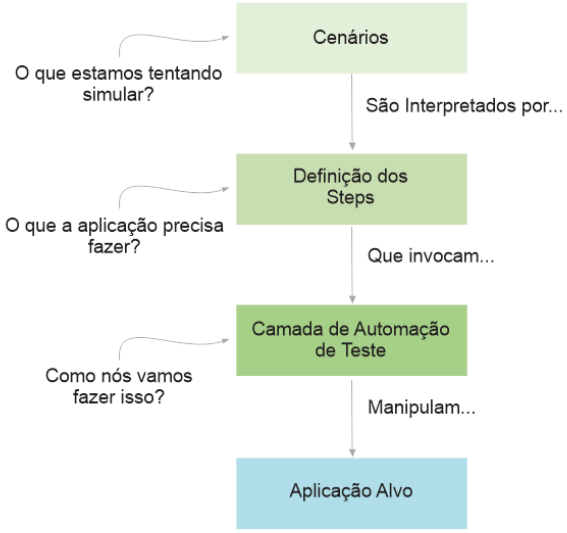
Uma descrição como a apresentada na Figura 2.2 é inserida em um arquivo de cenários, tipicamente nomeado com a extensão *.story* ou *.feature* (dependendo da ferramenta utilizada). Os métodos de teste, por outro lado, estão acondicionados em outro arquivo separado, tipicamente denominado de arquivo de *Steps*. Na execução de um cenário, as

sentenças do arquivo de cenários são executadas de forma sequencial. Assim, para cada sentença original, o método correspondente é buscado no arquivo *Steps* e executado.

Como o método de teste geralmente implementa comandos que simulam ações na interface da aplicação, essa execução deve ocorrer através de alguma ferramenta que realize a interação com a aplicação. Esta mesma ferramenta deve receber a resposta da aplicação ao comando executado e retorná-la para comparação com o resultado esperado descrito no cenário (também implementado no arquivo *Steps* como uma verificação).

Uma ferramenta de apoio à automação de cenários pode atuar em uma ou mais etapas da execução dos cenários, ou seja: i) realizar a conferência se um método de teste existe para uma determinada sentença; e ii) conectar-se à plataforma alvo para executar o método de teste. Conceitualmente, denomina-se *Runner* ou TAL - *Test Automation Layer* a camada responsável por conectar-se à interface da aplicação e executar o método de teste (Smart, 2015). Quando uma determinada sentença existe mas não há o método de teste correspondente, ela será marcada com o status de 'Pendente' em sua execução. A Figura 2.4 mostra o fluxo completo de automação de cenários BDD em uma ferramenta, conforme sugerido por Smart (2015).

Figura 2.4 – Automação de cenários BDD.



Fonte: Adaptado de Smart (2015)

Percebe-se que na linha 8 da Figura 2.2 é descrita a ação de um clique na interface da aplicação. Porém a informação enviada como parâmetro ('Ciência da Computação') não

define qual é o componente exato (botão? Link?) que deverá ser acionado. Para se resolver isso, durante a execução, pode-se usar um arquivo de mapeamento (*Mapping*) que contém essa informação.

O arquivo de *Mapping* contém endereçadores para determinadas sentenças utilizadas nos cenários (dbehave, 2017). Dessa forma, garante-se a utilização de palavras não técnicas na descrição dos cenários mantendo-se também a viabilidade da execução automática dos mesmos. O uso desse arquivo de mapeamento não é obrigatório em uma ferramenta de automação de cenários, mas é uma característica importante que permite manter os benefícios tanto dos cenários de alto nível quanto da automatização.

A Figura 2.5 exibe um trecho do código onde é realizado o mapeamento do cenário apresentado na Figura 2.2, para um botão em uma página específica na Web. O mapeamento é definido através da notação *@ScreenMap*, como uma tela onde serão mapeados os elementos. Esses elementos possuem a notação *@ElementMap*, como a ação de mapear um botão, por exemplo.

Figura 2.5 – Exemplo do arquivo de mapeamento de cenário.

```
@ScreenMap(name = 'WebSite INF', location = 'http://www.inf.ufrgs.br')
public void getClickTarget () {
   @ElementMap(name = 'Ciência da Computação', locatorType =
   ElementLocatorType.ID, locator = 'comp_link')
   private Button button;
}
```

Fonte: O próprio autor.

Existem diversas ferramentas para automação de cenários em diferentes linguagens e plataformas. Por exemplo, JBehave (JBehave, 2017) e Cucumber (Cucumber, 2017) são ferramentas que processam cenários implementados, respectivamente, em Java e Ruby. As ferramentas apresentam ainda outras diferenças, tais como o suporte oferecido (ou não) ao uso de rótulos parametrizados, mapeamento e descrição de cenários similares em formato reduzido, etc.

No entanto, como será detalhado no Capítulo 3, é incomum encontrar ferramentas que suportam mais de uma plataforma de execução, de forma a facilitar o processo de migração dos testes automatizados entre diferentes plataformas.

2.3 MULTIPLATAFORMA E REÚSO DE TESTES

No domínio dos aplicativos móveis o conceito de multiplataforma muitas vezes é entendido como o uso de dispositivos móveis que executam sistemas operacionais distintos. Assim, por exemplo, diz-se que um aplicativo é multiplataforma ou *cross-platform*, quando a criação de um aplicativo é realizada por meio de um único processo de desenvolvimento mas o resultado final serão aplicativos lançados em sistemas operacionais diferentes (Android e iOS, por exemplo).

Isso é possível através de ferramentas e frameworks de desenvolvimento que utilizam linguagens Web como HTML5, CSS3 e Javascript (entre outras linguagens de programação). Os principais sistemas operacionais do mercado mobile suportam essas linguagens e suas APIs, de forma que, ao criar um aplicativo baseado nessas ferramentas, o mesmo código pode ser compilado para sistemas operacionais diferentes (Microsoft, 2017).

Nesse trabalho, porém, usa-se a palavra multiplataforma para indicar plataformas de execução de natureza distinta, ou seja, plataformas móveis e não móveis. Especificamente, são consideradas três plataformas de execução de natureza distintas: Desktop, Web e Móvel. Essa diversidade impacta não apenas o desenvolvimento do aplicativo mas também a automação do teste de software pois cada plataforma possui características específicas tanto para a programação quanto para a interação com o usuário.

Muitas ferramentas não atendem mais que uma configuração de hardware ou sistema operacional (Android e iOS, por exemplo), o que dificulta a reutilização dos códigos de automação do teste de software. Quando se busca ferramentas que atendem a mais que uma plataforma (Web, Desktop e Móvel) o número é ainda mais escasso. No contexto de aplicativos Web (domínio Móvel e Web) tem-se um exemplo de ferramenta de teste multiplataforma que é a ferramenta Selenium (Selenium, 2018). Essa ferramenta atende a sistemas que executam sobre um navegador tanto na plataforma Web quanto na plataforma Móvel (para aplicativos do tipo Móvel Web). Porém, ela não atende às necessidades de teste dos outros tipos de aplicativos móveis (Nativo e Híbrido).

Embora o desenvolvimento de aplicativos móveis multiplataforma (codificados uma única vez para executarem em diferentes plataformas móveis) já conte com o suporte de ferramentas que propiciem o reúso tanto de código quanto de testes, o mesmo não acontece

para aplicativos que visam a execução em plataformas de natureza distinta. Conforme detalhado no Capítulo 3, não foi encontrado nenhuma abordagem que atendesse às três plataformas (Web, Desktop e Móvel) juntamente com os três tipos de aplicativos oferecidos pela recente plataforma móvel (Nativo, Web e Híbrido).

2.4 TRABALHOS RELACIONADOS

Nesta Seção discutem-se outras metodologias, ferramentas e técnicas de reúso de teste no contexto de aplicações multiplataformas.

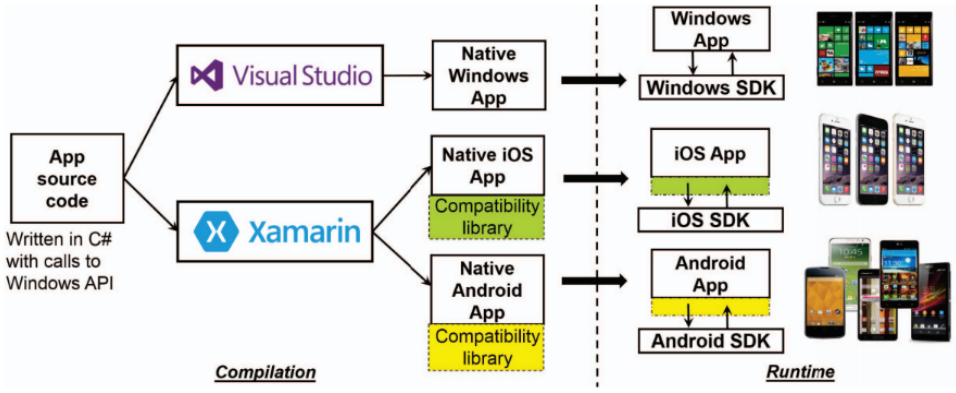
Ganapathy e Nagarakatte (2015) discute frameworks de desenvolvimento de aplicativos móveis multiplataformas (conceito tradicional, multiplataforma móveis). Tais frameworks permitem a especificação da lógica de negócio em um único código fonte, usando linguagem e API de uma plataforma de desenvolvimento e a geração automática de múltiplas versões do aplicativo, uma para cada sistema operacional móvel (Android, iOS e Windows Phone, por exemplo). Um desafio para o desenvolvimento desses frameworks é a correta tradução da API usada no desenvolvimento para a (possivelmente diferente) API da plataforma alvo sem alterar o comportamento da aplicação. Muitas vezes o framework de desenvolvimento não compila corretamente ou gera erros e inconsistências por não prover o mesmo comportamento que é implementado na API da plataforma alvo. Isso ocorre porque as APIs são atualizadas em um curto espaço de tempo e algumas funcionalidades e comportamentos implementados em um sistema operacional móvel podem não funcionar em outro.

Buscando resolver esse problema, Ganapathy e Nagarakatte (2015) propõe uma metodologia de teste para identificar possíveis inconsistências nesses frameworks que geram aplicações. Essa metodologia é suportada através de um protótipo de ferramenta desenvolvido pelo autor, denominada de X-Checker. De maneira resumida, a metodologia consiste em buscar o que a API da plataforma alvo oferece e comparar com a API do framework. Isso é feito para todos os sistemas operacionais móveis que o framework suporta, como mostrado na Figura 2.6.

Na solução proposta por Ganapathy e Nagarakatte (2015) , as regras de negócio e os comportamentos que o aplicativo deverá possuir são implementados em C# em um sistema operacional Windows. Compilando esse código no Visual Studio, uma versão WindowsPhone do aplicativo pode ser gerada. Da mesma forma, o compilador Xamarin permite a geração de

executáveis para iOS e Android. Especificamente, os autores buscam identificar casos de teste onde o comportamento da aplicação na plataforma de desenvolvimento difere do comportamento na plataforma alvo. A geração dos casos de teste é baseada na técnica de teste diferencial (apud MCKEEMAN, 1990) e os testes consistem em aplicativos móveis gerados usando técnicas de geração de teste aleatório com realimentação (apud PACHECO, 2007). A avaliação do comportamento de cada versão do aplicativo é feita através da observação de todas as estruturas de dados alcançáveis a partir das variáveis definidas nos casos de teste. Essas estruturas de dados são serializadas em um formato padrão e comparadas. Assumindo-se que o estado das plataformas de desenvolvimento e de execução é o mesmo antes da execução dos testes, espera-se que o estado final, após a execução da aplicação de teste, também seja idêntico. Se isso não ocorrer, há uma inconsistência no processo de tradução da aplicação.

Figura 2.6 – Exemplo de arquitetura da ferramenta X-Checker.



Fonte: Ganapathy e Nagarakatte 2015.

Ganapathy e Nagarakatte (2015) relata que a aplicação da ferramenta X-Checker no caso do Xamarin detectou 47 inconsistências entre a versão original e a versão móvel do aplicativo. Esse resultado reforça a hipótese dessa dissertação sobre o esforço de migração e a dificuldade de reúso entre plataformas de execução diferentes. No entanto, além de identificar as diferenças nas APIs, o trabalho de Ganapathy e Nagarakatte (2015) não discute os aspectos de automação dos testes de aceitação da aplicação nem como o problema de incompatibilidade pode ser mitigado pelo desenvolvedor.

O trabalho de Choudhary (2013) discute as incompatibilidades entre navegadores

disponíveis em diferentes plataformas de execução sejam elas móveis ou não. Como apresentado na Seção 2.1, aplicações Web têm o objetivo de serem acessadas através do navegador (*browser*) e não possuem a necessidade da instalação de um software para o uso. Assim, é comum a mesma aplicação Web ser acessada tanto da plataforma Desktop quanto da plataforma Móvel. Porém, na plataforma Móvel a mesma deve ser responsiva, ou seja, os elementos da interface devem ser redesenhados de forma a facilitar o uso nos diferentes formatos de dispositivos que respondem a essa plataforma.

Verificando diversas aplicações, Choudhary (2013) entende que não é uma tarefa trivial para os desenvolvedores reformular uma aplicação Web da plataforma desktop para a plataforma móvel, ou vice-versa. Muitas inconsistências podem ser encontradas entre os diferentes tipos de aplicações, fazendo com que se obtenha uma aplicação de baixa qualidade. Os autores definem o conceito de incompatibilidade entre navegadores como as discrepâncias de aparência e/ou comportamento na execução da mesma aplicação em ambientes (navegador e plataforma de execução) diferentes. Segundo Choudhary (2013), embora existam diversas abordagens de teste visando a detecção dessas incompatibilidades, essa tarefa ainda é pouco automatizada e basicamente baseada em dicas e "truques" que o desenvolvedor deve seguir para garantir uma compatibilidade mínima entre diferentes plataformas de execução. Estratégias de teste automatizado para esse problema consistem em: i) identificar um conjunto de características da aplicação a partir da sua execução em cada navegador e modelar a forma como a aplicação reage a um conjunto de estímulos (máquinas de estados são modelos comuns) e ii) comparar as características coletadas de cada navegador em busca de inconsistências de estado, comportamento e/ou visualização.

Mohamed (2015) também considera as inconsistências derivadas das múltiplas versões de um aplicativo móvel quando aplicativos nativos são desenvolvidos separadamente para cada plataforma e propõem uma técnica automática para detecção dessas inconsistências. A solução proposta instrumenta, automaticamente, o código-fonte e captura traços de execução de aplicativos iOS e Android. Os traços de execução são obtidos para um conjunto de cenários de execução pré-definidos. Modelos abstratos de execução são então derivados para cada conjunto de traços e comparados usando métricas e critérios baseados em código e na interface gráfica da aplicação.

No entanto, métodos automatizados muitas vezes identificam um grande número de falsos positivos, falsos negativos e falhas duplicadas, dificultando seu uso efetivo pelos

desenvolvedores. O framework de teste proposto por Choudhary (2013) também se baseia na identificação automática de características da aplicação em cada plataforma de execução, mas busca aprimorar os resultados de detecção usando quatro algoritmos independentes e ortogonais, cada um focando na detecção de um problema de compatibilidade específico: comportamento, estrutura, conteúdo visual e conteúdo textual. Os autores identificaram que a maior causa de incompatibilidade é relativa ao leiaute da aplicação, que não era claramente identificado em técnicas anteriores.

Em outro trabalho, Choudhary (2013) considera especificamente as incompatibilidades não intencionais entre versões da mesma aplicação desenvolvidas para diferentes plataformas de execução, especificamente para as versões Desktop e Móvel. O autor propõe usar um conjunto de traços de execução, um para cada plataforma, como base para a comparação das funcionalidades implementadas em cada versão da aplicação. A comparação feita considera apenas a estrutura funcional da aplicação, abstraindo as diferenças de apresentação entre plataformas de execução. A solução consiste em quatro fases: 1) coleção de um conjunto de traços de comandos de comunicação enviados entre cliente e servidor; 2) os traços são processados para a identificação de requisições que são instâncias de uma mesma ação na aplicação (há um mapeamento para um alfabeto de ações pré-definido); 3) os traços abstratos são clusterizados e um conjunto de traços essenciais ou canônicos (um traço por característica/funcionalidade em cada plataforma de execução) é definido; 4) os traços essenciais de diferentes plataformas são comparados buscando-se a correspondência entre as características e funcionalidades. O resultado dessa análise é a identificação de características e/ou funcionalidades de uma plataforma que não estão presentes em outra e que podem indicar uma falha. Embora promissora, a solução proposta por Choudhary (2013) considera apenas aplicações do tipo Web e apresenta um custo extra para coleta e processamento dos traços de execução. Além disso, a precisão e alcance dos problemas detectados estão atrelados à qualidade dos traços, que são gerados a partir de casos de teste.

A abordagem proposta neste trabalho busca o reúso do mesmo cenário de teste para versões de uma aplicação executando em diferentes plataformas (Desktop, Web e Móvel). Acredita-se que a abordagem proposta é relevante no processo de construção ou evolução de uma aplicação que deve executar em plataformas de natureza distinta, sendo complementar às técnicas existentes.

3 FERRAMENTAS DE AUTOMAÇÃO DE TESTE NA PLATAFORMA ANDROID

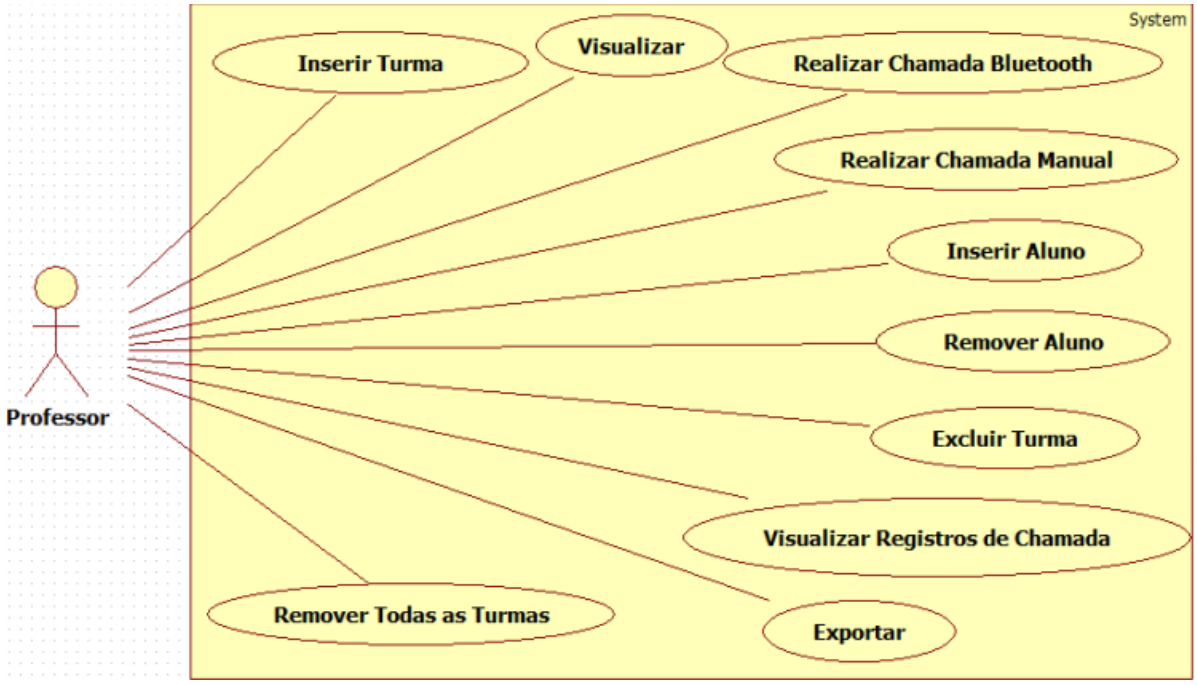
Este capítulo apresenta algumas ferramentas que suportam a técnica BDD no desenvolvimento de aplicações Android com o objetivo de avaliar o suporte disponível para o reúso de cenários de teste em aplicações multiplataforma (de natureza móvel e de naturezas distintas). A análise comparativa das ferramentas é realizada com base no estudo de caso descrito a seguir.

3.1 Estudo de Caso

Para avaliar e comparar as diferentes ferramentas foi utilizado um estudo de caso desenvolvido no Android Studio. Trata-se de uma aplicação para controle de frequência acadêmica, denominada de BlueTApp (Albiero, 2017).

As principais funcionalidades desta aplicação são apresentadas através de um caso de uso, apresentado na Figura 3.1.

Figura 3.1 – Caso de uso da aplicação BlueTApp (funcionalidades do papel de professor).



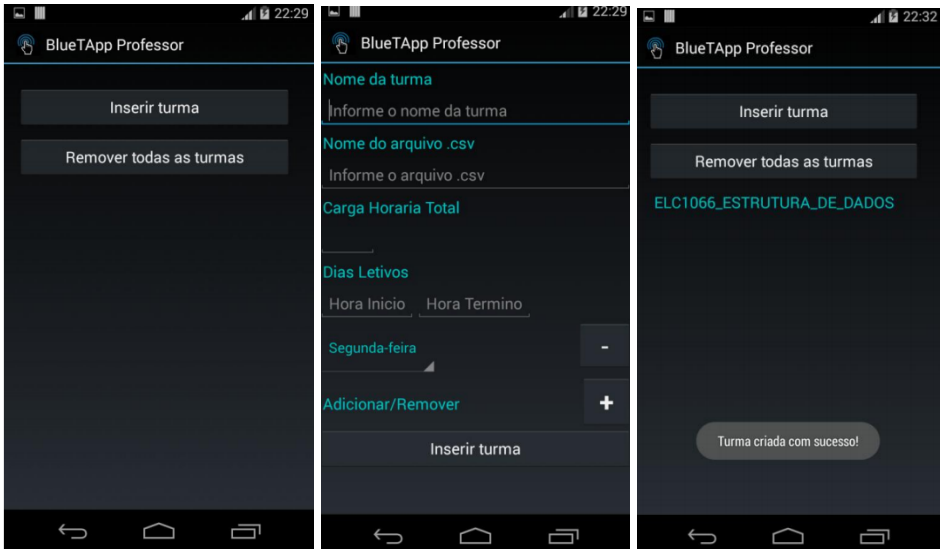
Fonte: Albiero (2017).

Dentre as diferentes funcionalidades, duas foram escolhidas para o desenvolvimento de cenários de teste:

1) Inserir Turma;
2) Realizar chamada manual.

A Figura 3.2 apresenta as telas do fluxo de Inserir Turma na aplicação.

Figura 3.2 – Criação de uma turma no BlueTApp.



Fonte: Albiero (2017).

Para validar a criação de uma turma no BlueTApp foram desenvolvidos dois cenários utilizando a técnica BDD, um válido, no qual a aplicação deverá aceitar a turma inserida, e outro inválido, no qual a aplicação não deverá aceitar a turma inserida, retornando uma mensagem de erro. Ambos cenários são apresentados na Figura 3.3.

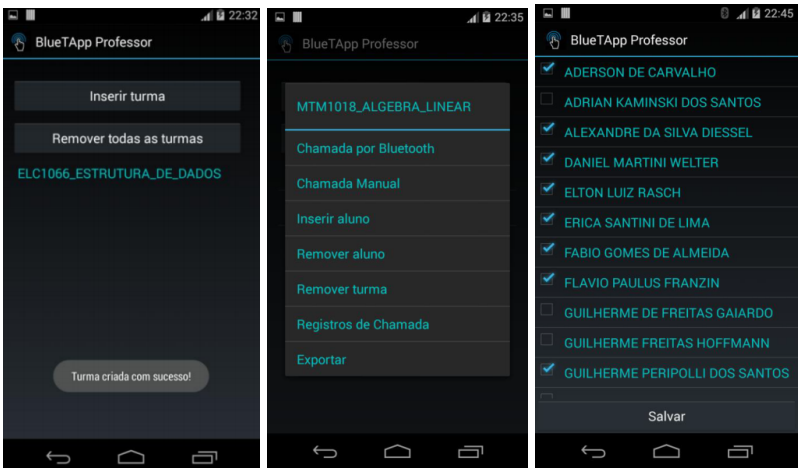
Figura 3.3 – Cenários da técnica BDD executados na aplicação BlueTApp.

```
Cenário: Inserir turma válida de forma manual

DADO que estou na tela inicial
QUANDO clico em inserir turma
E no campo Nome da Turma informo ‘ELC1066 ESTRUTURA DE DADOS’
E no campo Nome do arquivo .csv informo ‘arquivo.csv’
E o arquivo ‘arquivo.csv’ está no diretório padrão da aplicação
E no campo Carga Horária Total informo ‘220’
E em Dias Letivos informo ‘19:20’ em Hora Inicio e ‘22:35’ em Hora Término
E em Dias Letivos seleciono ‘Quarta-feira’
E clico em Inserir Turma
ENTÃO a mensagem de ‘Turma criada com sucesso!’ deve ser exibida

Cenário: Inserir turma inválida de forma manual

DADO que estou na tela inicial
QUANDO clico em inserir turma
E no campo Nome da Turma informo ‘ELC1066 ESTRUTURA DE DADOS’
E no campo Nome do arquivo .csv informo ‘arquivvo.csv’
E o arquivo ‘arquivo.csv’ está no diretório padrão da aplicação
E no campo Carga Horária Total informo ‘220’
E em Dias Letivos informo ‘19:20’ em Hora Inicio e ‘22:35’ em Hora Término
E em Dias Letivos seleciono ‘Quarta-feira’
E clico em Inserir Turma
ENTÃO a mensagem de ‘Não foi possível encontrar o arquivo CSV, por favor verifique.’ deve ser exibida
```

Fonte: O próprio Autor.

A Figura 3.4 apresenta telas do fluxo de realização de chamada de forma manual.

Figura 3.4 – Realização de chamada na forma manual.



Fonte: Albiero (2017).

Para realizar a validação da chamada manual é necessário ter uma turma cadastrada corretamente. Para isso é invocado o cenário de ‘Inserir turma válida de forma manual’ (Figura 3.3) e a partir deste, é validada a chamada manual. Utilizando assim, o reúso de cenários, conforme exemplificado na primeira sentença da Figura 3.5.

Figura 3.5 – Cenário BDD com reúso na aplicação BlueTApp.

```
Cenário: Realizar chamada manual

DADO que inseri uma turma válida de forma manual
E estou na tela principal da aplicação
QUANDO clique longo sobre a turma ‘ELC1066 ESTRUTURA DE DADOS’
E clicar na opção Chamada Manual
E selecionar as opções ‘ELTON LUIZ RASH’, ‘DANIEL MARTINI WELTER’ e ‘ERICA SANTINI DE LIMA’
E clicar em Salvar
ENTÃO a mensagem de ‘Chamada efetuada com sucesso’ deve ser exibida
```

Fonte: O próprio Autor.

3.2 METODOLOGIA DE SELEÇÃO E AVALIAÇÃO DE FERRAMENTAS

Inicialmente, foi realizado um levantamento de possíveis ferramentas de suporte à técnica BDD para a plataforma Móvel, através de buscas na Internet, consulta em blogs, fóruns, listas de emails e acessos originados do Google e Google Scholar. Palavras chave como ‘*Agile Software Testing*’, ‘*Mobile Application Quality*’, ‘*Mobile Applications Testing Automation*’, ‘*Mobile BDD tools*’ e ‘*Mobile Test Automation Framework*’ foram utilizadas nessa busca, resultando um universo de aproximadamente 50 ferramentas. Verificado que muitas dessas ferramentas não possuíam o código fonte para um estudo mais amplo, e outras não contemplavam o sistema operacional Android, foi definido um critério de busca para obter-se uma maior filtragem. O critério de busca foi definido como mostrado a seguir:

Ferramentas Open Source E
Automação de Teste de Software *Mobile* E
Para Android E
Utilização OU AUXÍLIO na aplicação da Técnica de BDD

Resultando menos que 20 ferramentas, em uma primeira avaliação comparativa das ferramentas, os seguintes aspectos foram considerados:

- nível de suporte à técnica BDD (se suporte completo ou como um facilitador);
- suporte a aplicativos móveis do tipo Web;
- necessidade do código fonte da aplicação;
- aderência a uma *BDD engine* (parser dos cenários descritos em linguagem Gherkin).

É importante ressaltar a convergência, nos resultados das buscas, para duas principais ferramentas de apoio à descrição dos cenários de teste: ferramentas Cucumber e JBehave. Por essa razão, o critério de aderência a uma dessas ferramentas foi incluído nesta primeira avaliação.

A Tabela 3.1 apresenta a avaliação das seis ferramentas selecionadas para análise considerando os aspectos relatados. A coluna 'Principal utilização na Aplicação do BDD' mostra o nível de suporte à técnica oferecido pela ferramenta. Tal suporte pode variar desde um suporte mínimo, onde a ferramenta corresponde a um único componente de apoio (*Parser* ou *Runner*), até o suporte completo, onde a ferramenta contém todos os conceitos apresentados na Seção 2.2.1 - Execução Automática de Cenários.

A coluna *'Mobile* Web' define se a ferramenta suporta ou não testes em sites e aplicações móveis que são acessadas via *Browser* do dispositivo. A coluna 'Necessita do Código Fonte' define se a ferramenta requer ou não o código fonte da aplicação alvo.

A coluna 'BDD Engine com maior aderência' define a qual *Parser*, dentre os dois principais (JBehave e Cucumber), uma dada ferramenta teria mais facilidade de integração. Assim, por exemplo, maior aderência ao *Cucumber* significa que a ferramenta foi construída seguindo as premissas do framework *Cucumber*.

	Principal utilização na Aplicação do BDD	***Mobile* Web**	**Necessita do Código Fonte**	**BDD engine com maior aderência**
Espresso	Facilitador na comunicação entre ações no Android. (Cliques, Ações, Comparações, etc.)	Não	Sim	Cucumber
Calabash	Ferramenta completa	Não	Não	Cucumber

Cucumber-JVM	Ferramenta completa	Sim	Sim	Cucumber
Selendroid	Facilitador na comunicação com o Android utilizando webdriver	Sim	Não	JBehave
Appium	Facilitador na comunicação com o Android utilizando webdriver	Sim	Não	JBehave
Robotium	Facilitador na comunicação entre ações no Android. (Cliques, Ações, Comparações, etc.)	Não	Sim/Não	Cucumber

Tabela 3.1. Ferramentas inicialmente selecionadas para o estudo.

A partir desses critérios, três conjuntos de ferramentas foram selecionados para uma avaliação mais detalhada: *Calabash* (Calabash, 2018), *JBehave* com *Selendroid* (Jbehave, 2017 e Selendroid, 2017) e *Cucumber* com *Espresso* (Cucumber, 2017 e Espresso, 2018):

Calabash: Uma ferramenta completa na aplicação da técnica BDD, ou seja, é independente de outras ferramentas;

JBehave com Selendroid: Ferramenta integrada que não possui necessidade do código fonte;

Cucumber com Espresso: Ferramenta integrada que possui a necessidade do código fonte do aplicativo.

Variações nesses conjuntos seriam possíveis, tais como: a utilização do *Appium* (Appium, 2018) em substituição ao *Selendroid*, ou a utilização do *Robotium* (Robotium, 2018) ao invés do *Espresso*. Porém, como são ferramentas semelhantes, optou-se pela utilização apenas do *Selendroid* e do *Espresso* devido ao critério de disponibilidade de documentação e facilidade de uso.

A avaliação detalhada foi feita a partir da implementação dos cenários de teste escolhidos para o projeto BlueTApp (representados nas Figuras 3.3 e 3.5) em cada um dos três conjuntos de ferramentas selecionados. Ao longo da implementação, os seguintes aspectos foram considerados para a avaliação comparativa:

- possibilidade de uso da ferramenta em outras plataformas, como a Web e desktop;
- volume de documentação disponível (Abrangente quando há grande volume ou Razoável quando há o mínimo necessário);
- suporte aos diferentes tipos de aplicações móveis (Web, Híbridas ou Nativas);
- necessidade de alterações na aplicação para aplicação do teste (por exemplo, concessão de uma permissão extra no aplicativo);
- linguagem de implementação;
- ambiente de execução do teste (Emulador, Dispositivo ou ambos).

A Tabela 3.2 resume os resultados da avaliação comparativa para os três conjuntos de ferramentas selecionados.

	Cucumber + Espresso	**Calabash**	**JBehave + Selendroid**
Aderência da ferramenta em outras plataformas como Web e Desktop.	Não possui	Não possui	Não possui
Documentação	Razoável	Abrangente	Razoável
Necessita do código fonte	Sim	Não	Não
Suporte a aplicações Web, Híbridas e Nativas	Nativas	Híbridas e Nativas	Híbridas, Nativas e Web
Necessário mudanças na aplicação ou .apk	Sim	Sim	Sim
Linguagem de Implementação	Java	Ruby	Java
Execução dos Testes	Ambos	Ambos	Ambos

Tabela 3.2. Principais características das ferramentas selecionadas.

Uma desvantagem das ferramentas que não necessitam do código fonte é que a aplicação a ser testada requer pelo menos uma alteração em seu código antes de ser compilada. Esta alteração se faz necessária para 'abrir' a comunicação com a aplicação através do protocolo HTTP.

Segue uma avaliação crítica para cada conjunto de ferramentas estudado:

- Calabash
    - **Vantagem**: Uso em múltiplos sistemas operacionais móveis (iOS e Android) e um grande acervo de documentação existente.
    - **Desvantagem:** Sem integração com a plataforma de desenvolvimento e grande dificuldade de organização de uma suíte de teste (dificulta Testes de Regressão e Features *Toggle*).
- Cucumber com Espresso
    - **Vantagem:** Fácil instalação.
    - **Desvantagem:** Necessita do código fonte e somente é possível executar o teste em aplicações nativas.
- JBehave com Selendroid
    - **Vantagem:** Atender todos os tipos de aplicações, não necessitar do código fonte e atender múltiplos sistemas operacionais móveis.
    - **Desvantagem:** Necessitar uma mudança no código fonte para se obter uma comunicação com o aplicativo através do protocolo HTTP.

A ferramenta *JBehave* com *Selendroid* apresenta vantagens no sentido de instalação, organização da suíte de teste (facilitando testes de Regressão e Features *Toggle*) e relatórios.

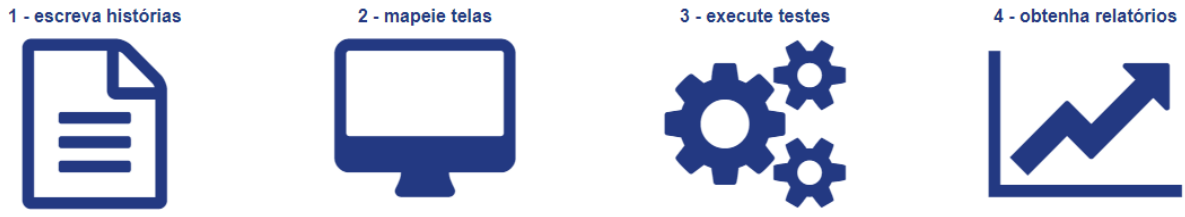
Pode-se notar que não foram encontradas, nessa pesquisa inicial, ferramentas que possam ser utilizadas em todas as plataformas de execução de natureza distinta (Móveis, Web e Desktop). Encontrou-se, como será discutido no Capítulo 4, um framework baseado em JBehave que atende às plataformas Web e Desktop, mas não suporta plataformas móveis. Assim, a partir dessa análise comparativa, entende-se que a combinação de JBehave com Selendroid é a mais propícia para dar suporte ao desenvolvimento deste trabalho.

3.3 REÚSO DE CENÁRIOS DE TESTE EM APLICATIVOS MULTIPLATAFORMA

Em estudo recente, Mesbah (2013) destaca que o maior desafio da indústria de desenvolvimento de software é manter uma aplicação consistente, alternando-se entre plataformas distintas. São muitas as variáveis que podem influenciar propiciando erros que podem se tornar defeitos ou falhas, como: sistema operacional, tipo de aplicativo, linguagem de programação, camadas de software e outras. Em outro estudo (Mohamed, 2015), destaca que desenvolvedores tratam as aplicações (principalmente as de plataformas móveis) de forma separada e manual, fazendo implementações tela a tela e que muitas vezes causam diversas inconsistências, além de tornar o processo tedioso, oneroso e bastante propício a erros.

Entretanto, este problema não está somente ligado ao desenvolvimento ou migração de uma aplicação, mas sim também ao teste de software. Quando trata-se da execução automática de cenários BDD um desenvolvedor que está utilizando um novo framework para a automação de cenários obtém um fluxo semelhante ao representado na Figura 3.6. Primeiramente, é necessário obter os cenários de uso/teste (*user stories*) que podem ser representadas por um arquivo *.story* ou *.feature*. Após, é necessário realizar o mapeamento dos elementos gráficos que serão utilizados na interface, para cliques, seleções, *inputs* e etc. Na etapa 3, a descrição de 'execute testes', refere-se aos métodos de teste implementados que estão associados a cada sentença do cenário de teste, e que serão executados durante a aplicação automatizada. Por último, obtém relatórios de execução.

Figura 3.6 – Exemplo resumido de um novo framework para automação de cenários.



Fonte: Adaptado de Dbehave, 2017.

A problemática ligada ao reúso neste contexto está em que desenvolvedores, muitas

vezes, não possuem o hábito de se precaver contra uma alteração de plataforma, ou seja, o reúso dos métodos de teste automatizados entre plataformas distintas não é considerado durante a implementação dos mesmos. Quando uma nova plataforma passa a ser considerada, comumente, uma nova ferramenta é implantada, fazendo com que somente as histórias (cenários de teste) sejam reutilizadas e os métodos de teste sejam reescritos gerando, assim, um retrabalho considerável. Além disso, a manutenibilidade dos próprios testes é prejudicada pela coexistência de ao menos duas ferramentas para suporte e manutenção.

Esta dissertação propõe uma metodologia em que a estrutura de programação dos testes (e não apenas as histórias) seja reutilizada em aplicações que precisam executar em plataformas de natureza distinta. Para isso, ferramentas de automação de cenários foram estudadas e classificadas de acordo com o suporte oferecido à técnica BDD, resultando na Tabela 3.2. Dessa forma, foram identificados os conceitos de *Runners*, *Parsers*, *Steps* e *Mapping* em conjunto com as ferramentas associadas a cada um desses conceitos. Para exemplificar, algumas ferramentas *open source* mais familiares à comunidade de teste de software podem ser citadas como exemplos de *Runner*: Selenium (Automação de testes funcionais para aplicações Web), Espresso (aplicativos Android), FEST (aplicações Desktop) e Appium/Selendroid (aplicativos Android e iOS).

Essa separação e estudo foi essencial para viabilizar uma metodologia que permitisse a reutilização de códigos de teste, independentemente da plataforma alvo.

A metodologia proposta permite desenvolver uma solução mais econômica, na questão de tempo, na automação de cenários BDD, com uma manutenção fácil e de grande suporte a uma nova plataforma que poderá existir, sem a necessidade de reescrita de cenários ou uma nova implementação de automação de cenários BDD.

De maneira geral, a metodologia proposta consiste na descrição dos cenários de teste sem o uso de referências a elementos da interface do aplicativo específicos de uma plataforma de execução. Além disso, é necessária e a integração de diversos *Runners* compatíveis com a linguagem de cenários em alto-nível, para se obter um único cenário que, de fato, execute em diferentes plataformas e preserve a estrutura funcional do método de teste.

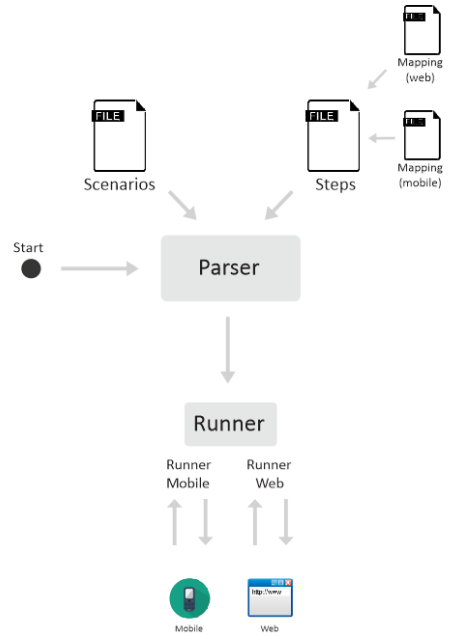
Sabe-se, porém, que determinadas sentenças em cenários utilizam referências a determinados elementos da interface da plataforma alvo do teste. Portanto, a metodologia se baseia na existência de um mapeamento separado para esses elementos de interface. Logo, o mapeamento (*Mapping*) descrito na Seção 2.2.1, é o único conceito que não permite a

reutilização total, uma vez que o endereçador de elementos (de um botão, por exemplo) muda em diferentes plataformas, mesmo permanecendo com a mesma funcionalidade.

No entanto, o *Runner* (ferramenta de automação de testes funcionais) pode ser acoplado a uma estrutura que permita automação de cenários BDD desde que a ferramenta seja compatível com a linguagem de programação dos *Steps*, transformando o conjunto de ferramentas em um framework multiplataforma (Web e Móvel, por exemplo). Porém, na prática, poucas ferramentas que automatizam cenários atendem múltiplos *Runners*. Assim, existem frameworks de suporte completo ao BDD baseados na combinação de JBehave com Selendroid (Saikri – (Saikri, 2017)) para atender unicamente a plataforma Móvel, outro framework que combina Cucumber com MsTest (SpecFlow – (Specflow, 2017)) para atender a plataforma Web, e assim por diante.

Em resumo, para que o suporte a plataformas de natureza distinta seja efetivo, deve ser possível usar conceitos e nomenclatura genéricos na implementação dos métodos de teste (que representam as sentenças de alto nível) resumindo a adaptação apenas ao arquivo de mapeamento, que descreve os detalhes de cada interface e precisa, de fato, ser refeito. A Figura 3.7 mostra uma visão geral de uma ferramenta com essas características.

Figura 3.7 – Diagrama de uma ferramenta de suporte ao reúso de cenários BDD.



Fonte: O próprio Autor.

Especificamente na plataforma Móvel, os diferentes tipos de aplicativos (Web, Nativo e Híbrido) devem ser considerados. Dessa forma, é necessário que o *Runner* que atende a plataforma Móvel, ofereça suporte a estes diferentes tipos de aplicativos.

Durante o desenvolvimento deste trabalho, encontrou-se uma única ferramenta, chamada dbehave, que oferece suporte às plataformas Desktop e Web mas que não suporta a automação de testes em plataformas móveis. Ainda assim, o suporte ao reúso de cenários oferecido por essa ferramenta é deficiente. Essa ferramenta é descrita no próximo capítulo juntamente com sua extensão, denominada MBehavior, implementada nesse trabalho como prova de conceito da metodologia proposta.

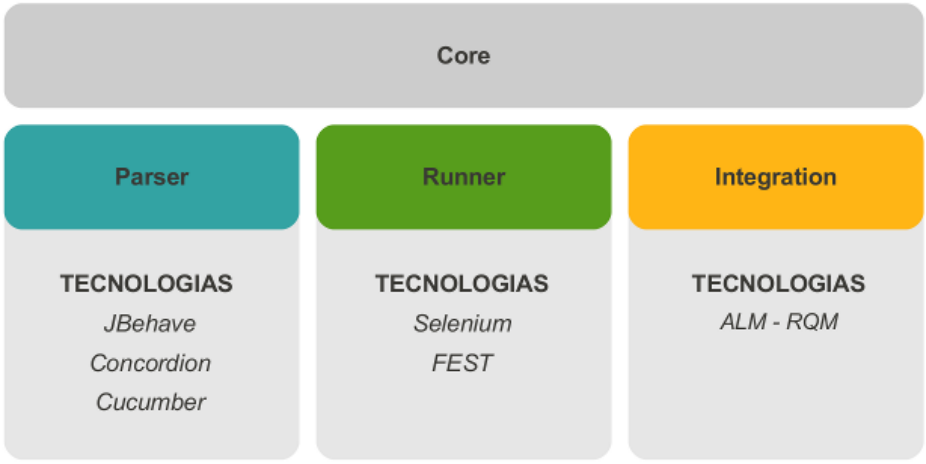
4 MBEHAVIOR

Neste capítulo o framework de suporte à metodologia proposta é apresentado e discutido, entre outros conceitos apresentados neste capítulo está a ferramenta dbehave, que foi expandida até originar o MBehavior, o processo de evolução juntamente com a escolha de um Runner para a implementação de cenários na plataforma Android.

4.1 DBEHAVE

O dbehave (Demoiselle Behave) é uma ferramenta *open source,* publicada em 2013, que realiza a automação de cenários utilizando a linguagem Java. A Figura 4.1 exibe a arquitetura do dbehave, formada por quatro componentes principais (DBehave, 2017).

Figura 4.1 – Arquitetura dbehave.



Fonte: (DBehave, 2017)

O Core é responsável pelas principais interfaces do framework, possuindo uma classe *Controller* responsável pela interação com os outros três principais módulos. O *Parser* é o componente responsável pela conexão entre os cenários e métodos de teste e envia os comandos (*Steps*) ao *Runner* por meio de sentenças padrões ou específicas da plataforma alvo. O *Runner* é responsável pela comunicação com a aplicação, fazendo as manipulações de tela e recebendo as respostas do sistema em teste. O módulo *Integration* é o componente que realiza as integrações entre o framework e ferramentas externas como o ALM (RQM), por exemplo (DBehave, 2017).

Uma das principais contribuições do dbehave é a existência de um conjunto de

sentenças e seus métodos de teste correspondentes que estão pré-definidos e prontos para uso. De fato, no contexto de aplicações gráficas ou Web, há uma série de ações muito comuns, tais como: “Given I am at ‘X’”, “When I click in ‘Y’”, “Then I should see the title ‘Z’”, e assim por diante. A ferramenta provê mais de 45 sentenças com ações diversas que já estão implementadas e podem simplesmente ser utilizadas nos cenários, sem maiores dificuldades. Isto faz com que um novo usuário já possa automatizar um teste de aceitação sem ao menos ter escrito um único método de teste, somente utilizando os métodos existentes.

Tanto as sentenças quanto os métodos de teste pré-definidos assumem uma nomenclatura comum. Assim, por exemplo, o framework define a classe *Runner* que contém a definição de uma série de ações comuns (tais como clicar) e os métodos de teste são implementados com base nesta classe. Durante a execução, um *Runner* específico é instanciado e associado a essa classe e sua API é mapeada para as ações definidas na classe. O dbehave também permite o uso de arquivos de mapeamento. Dessa forma, o dbehave oferece o suporte mínimo necessário para o uso de múltiplos *Runners*. Em sua versão atual, a ferramenta permite a automação de cenários em plataformas Desktop (usando o FEST) e Web (usando o Selenium).

Embora o dbehave use uma nomenclatura comum na definição dos cenários e métodos de teste, o reúso dos cenários em uma mesma aplicação executando em plataformas distintas ainda não é tão simples quanto poderia. Por exemplo, não é possível ter, no mesmo arquivo, cenários relacionados à mesma funcionalidade da aplicação mas que operam de maneira ligeiramente diferente em cada plataforma alvo. Para que isso seja possível, o usuário deve comentar os cenários específicos da plataforma que não é alvo da execução, sob pena de ter a execução suspensa por incompatibilidade com o *Runner*. Dessa forma, o usuário da ferramenta é induzido a criar uma cópia do arquivo de cenários e editá-lo manualmente quando altera o *Runner*, criando, ao final, duas estruturas de teste que precisam ser mantidas daquele momento em diante. Assim, embora a alteração de plataformas (Web → Desktop / Desktop → Web) seja simples de ser feita, assume-se que o arquivo de cenários seja ou reusado completamente ou editado a cada chaveamento de plataforma alvo (bastante sujeito a erros) ou ainda clonado e editado manualmente (operação também sujeita a erros e que aumenta os custos de manutenção dos testes). Além disso, a ferramenta ainda não oferece suporte a *Runners* voltados para a plataforma Móvel.

4.2 EVOLUÇÃO DO DBEHAVE PARA O MBEHAVIOR

A abordagem proposta nesta dissertação é obter um cenário que possa ser utilizado em mais de uma plataforma, com pouco, ou nenhum, retrabalho. Desta forma, desenvolvedores poderão testar o software mais cedo, garantindo que funcionalidades que antes funcionavam na plataforma antiga, ainda funcionem na nova plataforma, agregando assim, qualidade ao software.

Devido à forte tendência de migração para a plataforma Móvel, decidiu-se neste trabalho desenvolver uma extensão de uma ferramenta já existente (dbehave), para que reutilize seus cenários (que atualmente já executam nas plataformas Web e Desktop) na nova plataforma Móvel. Desta forma, as seguintes adaptações ao framework dbehave, são propostas neste trabalho: 1) suporte a um *Runner* para a plataforma Android e 2) suporte à definição, no próprio cenário, da(s) plataforma(s) à(s) qual(is) aquele cenário se aplica. Com essas adaptações, os desenvolvedores têm mais liberdade para organizarem os cenários em torno das funcionalidades ao invés de o fazerem em torno da plataforma alvo, facilitando a manutenção e evolução desses cenários juntamente com a aplicação. A essa nova versão do dbehave, é proposto através deste trabalho o nome de MBehavior.

4.2.1 Definição de um *Runner* para Android

A implementação do MBehavior fundamentou-se na definição de um *Runner* que atendesse a plataforma Móvel e fosse também compatível com os *Runners* disponibilizados pelo framework.

Uma questão importante, no contexto de aplicativos móveis, é que há diferentes tipos de aplicações móveis e cada tipo pode exigir um *Runner* distinto. Deve-se considerar, portanto, a execução de aplicativos do tipo Nativos, Web e Híbridos. Outra questão que impacta a definição do *Runner* é se o desenvolvedor deve possuir ou não acesso ao código fonte da aplicação. Por fim, é importante conhecer como o *Runner* irá conectar-se ao dispositivo móvel, se através de um dispositivo presente fisicamente conectando-se através de um cabo USB ou através de um emulador, e se o *Runner* suporta ambos os formatos.

Como apresentado no Capítulo 3, as ferramentas open source *Cucumber*, *Espresso*, *Calabash*, *JBehave* e *Selendroid*, foram analisadas em relação aos aspectos comentados acima. Esta análise deu suporte à escolha do novo *Runner* do framework MBehavior.

Assim, dentre as diversas ferramentas *open source* analisadas, a ferramenta *Selendroid* mostrou-se a mais adequada, pelas seguintes razões: 1) suporta aplicativos do tipo Nativos, Web e Híbridos; 2) não exige acesso ao código fonte da aplicação; 3) permite o uso tanto de emuladores quanto de dispositivos reais. Além disso, como o dbehave já suporta o software *Selenium* como *Runner*, a compatibilidade do novo *Runner* com o dbehave é maior.

4.2.2 Selendroid

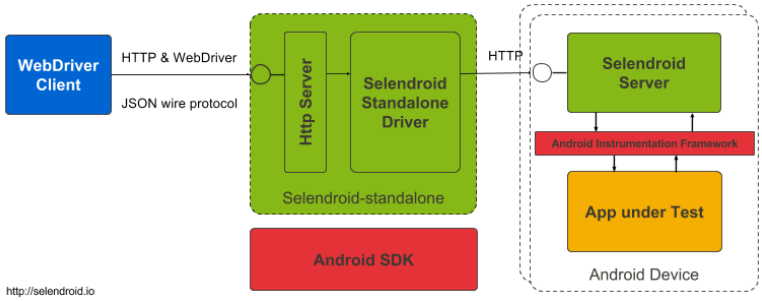
O Selendroid é um framework *open source* para a automação de testes de interface em aplicativos Android e que suporta aplicações do tipo Híbridas, Web e Nativas (Selendroid, 2017). Para executar testes em aplicações Nativas utilizando o Selendroid, além de obter um arquivo .apk, deve-se adicionar uma linha de comando neste arquivo, que permita à ferramenta obter acesso à interface da aplicação, utilizando o protocolo HTTP.

Baseado no *Android instrumentation*, uma característica do framework Selendroid é a possibilidade de utilização tanto do dispositivo quanto de um emulador para executar os métodos de teste (Selendroid, 2017). O Selendroid utiliza quatro componentes principais em sua arquitetura:

- *Selendroid-Client* - Biblioteca em java, baseada no cliente do selenium;
- *Selendroid-Server* - Executado diretamente no sistema operacional Android;
- *AndroidDriver-App* - Componente Android para realizar testes de aplicações Web;
- *Selendroid-Standalone* - Gerenciador de diferentes aplicativos Android, no qual instala, versão por versão, o Selendroid-Server para manipular objetos da interface.

A Figura 4.2 apresenta a arquitetura Selendroid na sua versão 0.17.0 e que foi utilizada neste trabalho.

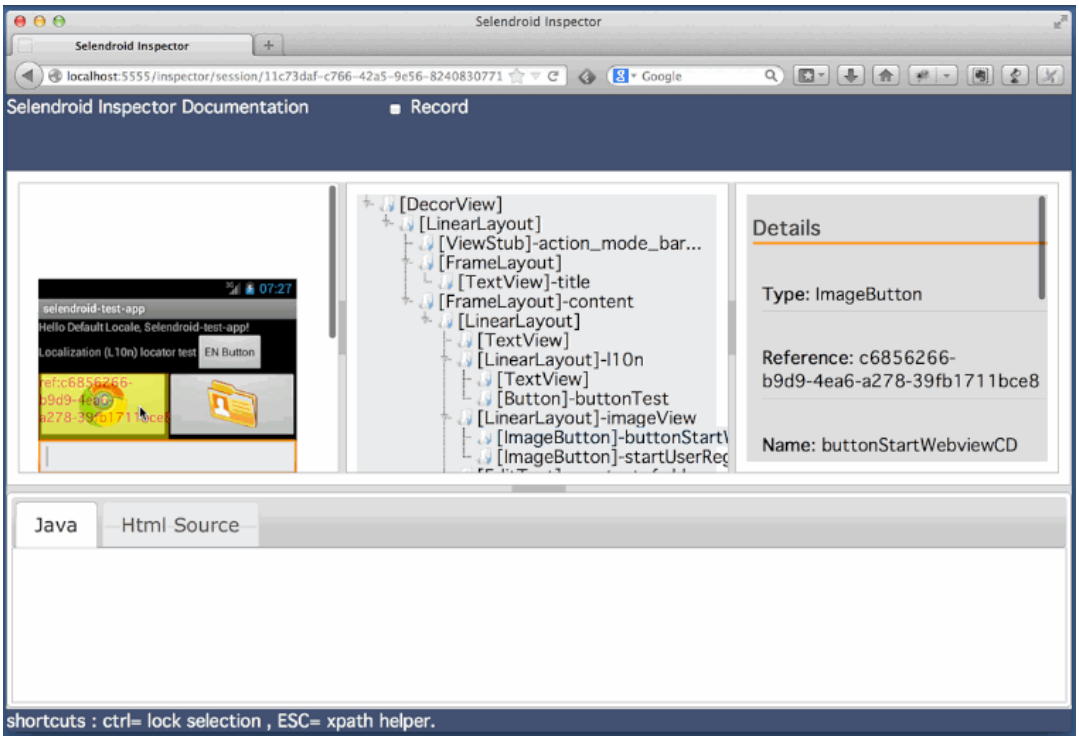
Figura 4.2 – Arquitetura Selendroid.



Fonte: Selendroid, 2017.

O Selendroid-Server é o principal componente para a automação de testes nas aplicações (Selendroid, 2017). Como o código fonte de uma aplicação alvo de teste, não está disponível, é impossível conhecer determinados endereçadores para mapear ações. Para resolver isso, a ferramenta disponibiliza um framework Web, denominado de Selendroid Inspector, que simula através do navegador uma tela de um aplicativo nativo, com os devidos endereçadores.

Figura 4.3 – Selendroid Inspector.



Fonte: Selendroid, 2017.

O Selendroid Inspector, conforme apresentado na Figura 4.3, é fundamental para conhecer quais são os endereçadores das ações que podem ser efetuadas no aplicativo do tipo Nativo e Híbrido. Sem conhecer estes endereçadores, torna-se impossível a automação de um cenário.

Estas e outras características indicam a ferramenta Selendroid como uma integração viável ao dbehave, de forma a habilitá-lo a efetuar a automação de cenários na plataforma Móvel.

4.2.3 Inclusão do Selendroid como *Runner* no dbehave

Partiu-se da versão 1.5.1 da ferramenta dbehave para implementação do MBehavior. Essa versão possui dois *Runner* habilitados: Selenium (Web) e Fest (Desktop). Um dos desafios na implementação do novo *Runner* Móvel é a compatibilidade com os *Runner* existentes, uma vez que o Selendroid utiliza bibliotecas do Selenium, que já são utilizadas originalmente no dbehave. As questões de compatibilidade foram resolvidas com o Maven, que é o gerenciador de dependências usado no dbehave.

Para executar na plataforma Móvel uma aplicação Nativa, é necessário a inclusão de uma linha de código no arquivo de *properties* indicando o caminho do arquivo .apk da aplicação alvo.

Com essa alteração, o dbehave passa a permitir a automação de cenários em aplicativos Android do tipo Web, Híbrido ou Nativo (sem a utilização do código fonte), utilizando emuladores ou o próprio dispositivo. O novo framework integra ainda a ferramenta Selendroid Inspector, que permite a identificação de endereçadores para que sejam usados no arquivo de mapeamento. Uma possível migração das plataformas Web ou Desktop para a plataforma Móvel pode ser atendida agora sem que exista qualquer reestruturação dos cenários ou necessidade de alguma outra ferramenta.

Além disso, recentemente o Selendroid anunciou o uso do *ios-driver*, que permite ao MBehavior realizar testes também em sistemas operacionais iOS.

4.2.4 Implementação de cenários visando o reúso em multiplataformas

A inclusão do *Runner* para Android não resolve a questão de reúso dos cenários. O objetivo, nesta questão, é permitir ao desenvolvedor incluir, em um único arquivo, cenários que devem ser aplicados tanto em uma plataforma específica quanto nas diversas plataformas onde a aplicação em teste é executada.

Para isso, propõem-se a criação de uma nova sentença de contexto. Essa sentença é denominada de ‘Given it is an execution in “X”’, sendo ‘X’ um parâmetro associado à plataforma alvo que pode assumir um dos seguintes valores: web, desktop, mobile-web ou mobile-nativo. A Figura 4.1 apresenta um exemplo de uso desta sentença em um cenário que

deve ser executado apenas na versão Móvel (com interface web) de uma dada aplicação.

Figura 4.4 – Exemplo de sentença informando a plataforma de execução.

```
1 - Verify if the information on INF website is correct.
2 - As a guest
3 - I want correct presentation on INF website
4 - So that I can utilize the INF website without making any mistake and get a good experience.
5 -
6 - Scenario 1: Check the INF website information
7 - Given it is an execution in 'mobile-web'
7 - And go to 'WebSite INF'
8 - When click in 'Ciência da Computação'
9 - Then should be shown 'Bacharelado em Ciências de Computação'
```

Fonte: O próprio Autor.

É importante ressaltar que todos os comportamentos originais do dbehave foram mantidos. Dessa forma, esta sentença não se tornou obrigatória, mas permitiu obter cenários executáveis ou não em uma determinada plataforma. No dbehave, se um determinado cenário for executado em uma plataforma diferente do configurado, um erro é exibido e a execução deste e dos próximos cenários é suspensa. Com a nova sentença, os cenários que não pertencem à plataforma configurada serão marcados como 'Pendentes'. Através desta nova sentença, o framework verifica qual plataforma está habilitada, ignorando cenários de uso que estão com a sentença diferente da plataforma que está habilitada. Por outro lado, cenários de uso sem a nova sentença implementada, são executados normalmente.

4.2.5 Exemplo de uso do MBehavior

A ferramenta MBehavior, é distribuída com cinco exemplos de uso para que seja mais fácil a compreensão por parte do usuário. Estes exemplos são denominados de *Archetypes*.

Todos os *Archetypes* possuem uma estrutura completa, que permite ao usuário realizar o download de um projeto modelo e partir deste projeto para implementar seus testes. São exemplos de alguns *Archetypes* que a ferramenta disponibiliza:

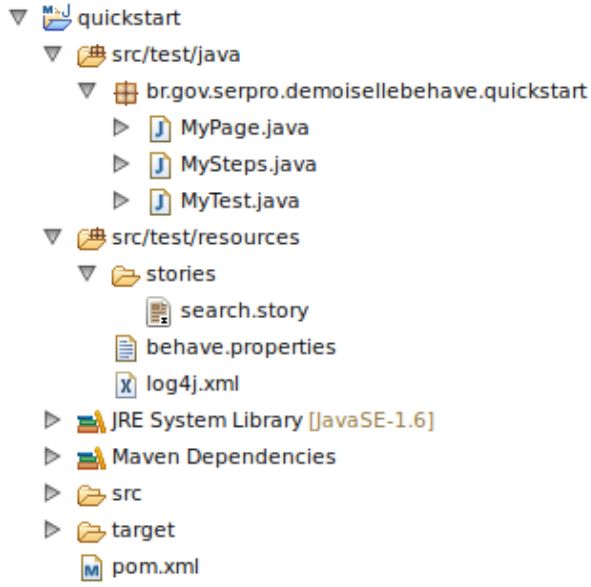
- Archetype Reúso de Cenário

Este pacote reúne exemplos da implementação apresentada na Seção 4.2.4, e consiste

em um arquivo de cenários com extensão *.story*. Neste arquivo existem cenários que são executados em mais de uma plataforma, permitindo a validação da implementação da feita.

A Figura 4.5 mostra a estrutura de um novo projeto baseado no *Archetype* de Reúso de Cenário, onde, basicamente, tem-se o diretório principal src/test/java onde estão contidos os arquivos de *Steps* (sentenças de teste) e o MyTest, que é o responsável por iniciar o método de teste.

Figura 4.5 – Modelo da estrutura de um projeto no archetype de reúso.



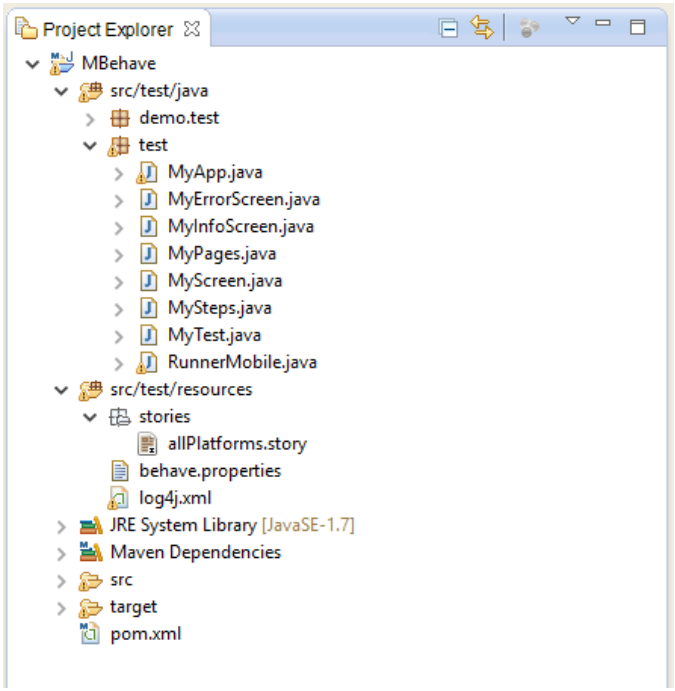
Fonte: O próprio Autor.

- Archetype de Aplicação Desktop

O *Archetype* da aplicação Desktop possui uma aplicação modelo, juntamente com o projeto de teste, que simula uma aplicação *desktop* na linguagem Java com a funcionalidade de Login em um sistema. O projeto de teste, possui pelo menos três cenários automatizados, juntamente com um mapeamento de tela. Para isso, utiliza-se bibliotecas do framework Fest.

A Figura 4.6 apresenta a estrutura do Archetype de uma Aplicação Desktop, onde o arquivo ‘MyPages’ representa o *Mapping* das telas da aplicação Desktop, juntamente com os arquivos ‘MyErrorScreen’ e ‘MyInfoScreen’.

Figura 4.6 – Archetype de uma Aplicação Desktop.



Fonte: O próprio Autor.

- Archetype de Aplicação Web

Para a aplicação Web, são distribuídos alguns cenários que simulam acessos a um determinado website, fazendo algumas ações e verificando resultados conforme descrito nos cenários. Para isso, utiliza-se bibliotecas do framework Selenium.

A Figura 4.7 apresenta o arquivo de ‘Steps’ onde as sentenças de um cenário BDD são executados através de métodos de teste.

Figura 4.7 – Modelo do arquivo Steps.java encontrado no Archetype de Aplicação Web

```
1   WebDriver driver = null;
2   driver = new FirefoxDriver();
3   driver.manage().window().maximize();
4
5   @Given("vou para \"$url_destino\"")
6   public void VouParaURL(String url) throws InterruptedException {
7       driver.get(url);
8   }
9
10  @When("clico em \"$link_click\"")
11  public void ClicoEm(String link) throws InterruptedException {
12      driver.findElement(By.Text(link)).click();
13      Thread.sleep(4000);
14  }
15
16  @Then("será exibido \"$texto\"")
17  public void Exibido(String texto) throws InterruptedException {
18      driver.findElement(By.Text(texto));
19      Thread.sleep(4000);
20  }
21
```

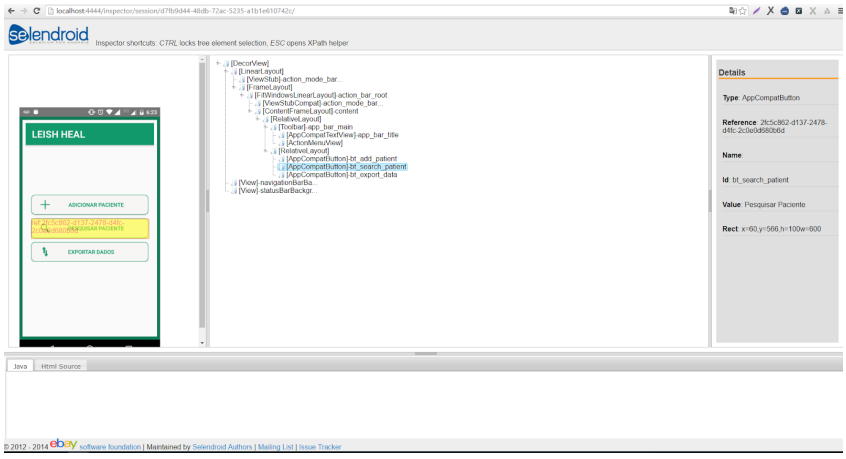
Fonte: O próprio Autor.

- Archetype de Aplicativo Mobile Nativo

Aplicativos Nativos são executados no MBehavior com o auxílio do framework Selendroid. Este *Archetype* consta com um arquivo .apk (Android) que é o instalador de um aplicativo denominado de Leish Heal (Albiero, 2017), desenvolvido através da IDE Android Studio, que tem por objetivo auxiliar no tratamento da doença de *Leishmaniose*.

Arquivos .apk não trazem consigo o código fonte editável do aplicativo, portanto é disponibilizado através deste *Archetype* o Selendroid Inspector que auxilia na busca de elementos em aplicativos onde o código fonte não está disponível, conforme apresentado na Seção 4.2.2

Figura 4.8 – Inspector Archetype de Aplicativo Nativo.



Fonte: O próprio Autor.

- Archetype de Aplicação Mobile Web

Aplicativos Web podem ser executados no próprio celular, caso esteja conectado, ou em um emulador. São execuções através do navegador (*Browser*) do próprio celular, que simulam os cenários.

Além dos *Archetypes* apresentados o MBehavior também conta com um estudo de caso, onde obteve resultados significativos quanto a sua agilidade de implementação e migração entre plataformas.

5 RESULTADOS EXPERIMENTAIS

Para validar a extensão proposta neste trabalho, o MBehavior foi aplicado em dois estudos de caso.

O primeiro estudo de caso, trata-se de uma aplicação móvel *open source* para postagens de notícias em Blogs e Website denominado de Wordpress, já com um grande número de usuários (entre 5.000.000 e 10.000.000, segundo Google Play, 2017) e suporte multilíngue a técnica BDD em sua aplicação na plataforma Web.

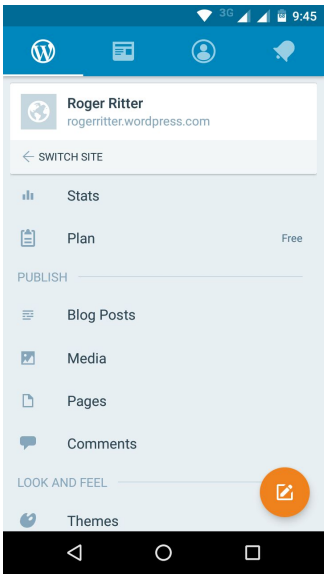
O segundo estudo de caso, trata-se de uma aplicação móvel não *open source* direcionada ao suporte financeiro de clientes de uma determinada instituição bancária presente no Brasil. Estima-se que a aplicação está instalada em mais de 100.000 dispositivos e acessada diariamente por mais de 5.000 usuários. Este estudo de caso também possui uma versão na plataforma web com cenários BDD e, atualmente, estão sendo migrados seus testes (e funcionalidades) para a plataforma móvel. O estudo de caso número dois foi denominado de aplicação financeira.

Ao final, obtém-se uma tabela comparativa entre os dois estudos de caso, comparando a uma média da economia de tempo entre os experimentos.

5.1 WORDPRESS

Este estudo de caso consiste em usar o MBehavior para realizar testes de aceitação automáticos no WordPress (Wordpress, 2017), um sistema para publicação de notícias e gerenciamento de websites que representa 27% do uso da web ao redor do mundo (Wordpress, 2017). O WordPress possui versões para a plataforma Web e, recentemente, para a plataforma Móvel.

Figura 5.1 – Versão Móvel da aplicação Wordpress.



Fonte: Wordpress, 2017.

A versão original da aplicação possui mais de 85 funcionalidades (sem considerar a utilização de plugins). Dessas, 12 (relacionadas com a funcionalidade de fazer login e publicar um conteúdo Web) foram escolhidas para serem descritas no formato de cenários (BDD) com o auxílio do MBehavior. A escolha deste subconjunto de cenário, seguiu os seguintes critérios: 1) requisitos chave da aplicação; 2) requisitos com documentação mais ampla.

Inicialmente, os cenários foram descritos visando o teste da versão Web do sistema. Em um segundo momento, os mesmos cenários foram executados na versão Móvel do WordPress.

Dentre os cenários descritos no MBehavior, 10 representam funcionalidades que são iguais nas duas plataformas alvo, diferenciando-se apenas pelos detalhes de identificação dos componentes da interface definidos no arquivo de mapeamento. Outras duas funcionalidades, representavam caminhos diferentes para obter-se o mesmo resultado. Por exemplo, na plataforma Web, o login é solicitado através do usuário e a senha na mesma tela, mas na plataforma Móvel o login primeiro solicitava o usuário e, somente na próxima tela, a senha era solicitada.

No processo de descrição dos cenários Web, o único esforço do desenvolvedor foi o de escrever os cenários, utilizando sentenças pré-definidas e disponibilizadas pelo dbehave, além de definir o mapeamento dos elementos de cada interface. Da mesma forma, no processo de execução dos cenários na plataforma Móvel. Portanto, nenhuma implementação de método de

teste adicional foi necessária. Apenas o arquivo de mapeamento foi alterado.

Após a descrição dos cenários no MBehavior, os mesmos foram executados sobre a aplicação original (Web), utilizando o Selenium como *Runner*. Em seguida, alternou-se o *Runner* para a versão Móvel (Selendroid). O que o(s) arquivo(s) de mapeamento já havia sido alterado(s) para incluir as referências corretas aos elementos da interface móvel.

Em ambos os casos, os cenários foram executados a contento sem maiores esforços por parte do desenvolvimento. Obtendo, em um software com 12 funcionalidades (100%), sendo que, 83% de cenários (10 funcionalidades) foram reusados sem alterações e 17% de cenários (2 funcionalidades) tiveram que ser alterados, devido ao fluxo de implementação diferenciado, entre as plataformas.

5.2 APLICAÇÃO FINANCEIRA

O estudo de caso da aplicação financeira objetivou-se em utilizar o MBehavior para realizar a migração entre cenários BDD já existentes na plataforma Web para a plataforma Móvel, por um desenvolvedor sem nenhum contato anteriormente com a ferramenta dbehave ou a ferramenta extensão tema deste trabalho, MBehavior.

Com base na documentação formal do projeto, 21 cenários foram escolhidos para serem descritos e executados com o auxílio do MBehavior. A escolha deste subconjunto seguiu os seguintes critérios: 1) requisitos chave da aplicação; 2) cenários com pouca (ou nenhuma) integração com outros softwares;

Inicialmente, o esforço principal relatado, foi de entender o mecanismo herdado do dbehave de sentenças pré-definidas e o mapeamento de botões, campos e etc. da aplicação web. Destacou-se que nenhuma nova sentença adicional foi necessária, além daquelas que já estavam implementadas, o que, promoveu uma grande otimização de tempo, uma vez que os métodos de testes já estavam implementados no MBehavior.

Com 21 cenários executando na aplicação web através do *Runner* Selenium, alternou-se o *Runner* para o Selendroid, buscando iniciar-se os testes na aplicação móvel. Relata-se que, dentre os cenários descritos no MBehavior, 19 representavam funcionalidades iguais nas duas plataformas e outras 3 precisaram ser criadas, devido a divergências de comportamento. Para não alterar os 3 cenários que funcionavam na plataforma Web, mas não na plataforma Móvel, criou-se mais 3 cenários que seriam utilizados apenas na plataforma

Móvel, fazendo uso da nova característica implementada no MBehavior: permitir a escolha de qual plataforma um determinado cenário será executado.

Tanto na plataforma Móvel, quanto na plataforma Web os 24 cenários foram implementados sem maiores esforços por parte do desenvolvimento. Obtendo, em um software com 21 cenários (100%), sendo que, 79,1% dos cenários foram reutilizados sem alterações e 20,9% destes cenários verificou-se uma divergência de fluxos/comportamento que necessitou na criação de mais 3 cenários para uso exclusivo na plataforma Móvel.

Em ambos os estudos de caso (wordpress e aplicação financeira), destaca-se a otimização do tempo quando utiliza-se sentenças pré-definidas e também a facilidade do reúso destes cenários (implementados ou reusados) entre plataformas, principalmente para a nova plataforma Móvel.

Conclui-se através dos dois estudos de caso que o wordpress representou 83,3% de reusabilidade, enquanto a aplicação financeira representou 79,1%. Aderindo assim uma média de reusabilidade entre os estudos de caso de 81,2%.

5.3 TABELA COMPARATIVA

Através de Tabela 5.1 é possível verificar a economia de tempo entre os diferentes estudos de casos. O tempo médio de desenvolvimento de cada cenário (representado através da coluna 3) levou em consideração quanto é o tempo de um desenvolvedor para desenvolver um único cenário para a aplicação, no MBehavior. Levando em consideração que existem cenários mais complexos que necessitam de mais tempo para serem desenvolvidos e cenários mais simples que necessitam de menos tempo, por isto uma média histórica foi realizada.

A coluna de Cenários Reutilizados representa quantos cenários não necessitaram ser implementados novamente em outra plataforma. Ao lado, a porcentagem considerando quantos por cento dos cenários foram reutilizados.

A média história de tempo da construção do cenário foi multiplicada pelo número de cenários que foram reutilizados, considerando outra ferramenta, estes cenários deveriam ser reescritos novamente, e muitas vezes, em uma outra linguagem de programação ou sintaxe, o que poderia demasiar mais tempo.

Experimento	Número de Cenários Implementados	Tempo Médio de Desenvolvimento de cada Cenário	Cenários Reutilizados / Porcentagem	Média da economia de tempo (Cenários Reutilizados x Tempo Médio)
Aplicação Financeira	24	~39m	19 / 79,1%	~12:30h
Wordpress	12	~26m	10 / 83,3%	~4:30h

Tabela 5.1. Tabela comparativa dos resultados experimentais.

Podemos perceber em totais que uma média da economia de tempo seriam de 17 horas, um tempo bastante considerável e que tende a ser uma linha crescente de economia cada vez que mais cenários são implementados utilizando o MBehavior.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho propôs uma metodologia que prevê o reúso de cenários de teste automatizados entre plataformas de execução de natureza distinta, ou seja, entre plataformas Desktop/Web e plataformas Móveis. Essa metodologia é suportada através do desenvolvimento de um framework *open source* denominado de MBehavior, sendo uma extensão do framework dbehave que permite a automação de cenários na plataforma Android. Esta extensão possibilita ao framework suportar as três plataformas mais conhecidas (Desktop, Web e Android) sem descentralizar a documentação dos cenários de uso. A ferramenta Selendroid foi escolhida para ser integrada como *Runner* para a plataforma Android pois permite a execução de aplicativos Móveis do tipo Nativo, Web e Híbrido. Além do *Runner*, a solução proposta implementou uma nova sentença de contexto que pode ser usada em cada cenário para definir em qual(is) plataforma(s) aquele cenário deve ser executado. O MBehavior foi utilizado em duas aplicações reais na plataforma Web e Android mostrando-se uma ferramenta eficaz em seu propósito de minimizar o esforço de migração dos testes de aceitação automatizados, apresentando uma economia de tempo e esforço.

Apesar dos resultados promissores, o MBehavior ainda pode evoluir em diversos aspectos. Por exemplo, algumas outras funcionalidades que o Selendroid nativamente permite, como gestos em aplicações nativas, podem ser disponibilizados como sentenças e métodos pré-definidos. Da mesma forma, simplificar ainda mais a troca da plataforma de execução dos testes.

Além disto, recentemente o Selendroid anunciou uma nova API denominada de *ios-driver* que permite ao Selendroid executar testes no sistema operacional iOS. Desta forma, sugere-se ainda a evolução do framework MBehavior de forma a executar testes neste novo sistema operacional.

REFERÊNCIAS

ALBIERO, Um Aplicativo Móvel para Registro Automático da Presença Acadêmica via Bluetooth. Disponível em: <http://www-app.inf.ufsm.br/bdtg/arquivo.php?id=196>. Acesso em: 19/11/2017.

ANBUNATHAN, AnirbanBasu. Automation Framework for Testing Android Mobiles. **International Journal of Computer Applications**, v. 106, n.1, november, 2014, pages 4-17.

ANDROID, Android Developers - 2017. Disponível em <https://developer.android.com/index.html>. Acesso em 19/11/2017.

APPIUM, Appium: Mobile App Automation Made Awesome.. Disponível em <http://appium.io/>. Acesso em 15/04/2018.

CALABASH, Calaba.sh - Automated Acceptance Testing for iOS and Android Apps. Disponível em: <https://calaba.sh/>. Acesso em 15/04/2018.

CUCUMBER, Cucumber. Disponível em <https://cucumber.io/>. Acesso em: 20/02/2017.

CUCUMBER JVM, Disponível em <https://cucumber.io/docs/reference/jvm>. Acesso em: 15/04/2018.

DAN NORTH, Dan North & Associates. Disponível em: <https://dannorth.net/introducing-bdd/>. Acesso em: 20/02/2017.

DBEHAVE, Framework de Testes Funcionais Automatizados com BDD. Disponível em: <http://dbehave.com/>. Acesso em: 20/02/2017.

ESPRESSO, Espresso | Android Developers. Disponível em: <https://developer.android.com/training/testing/espresso/index.html>. Acesso em: 15/04/2018.

GANAPATHY, B. Nader, V. Nagarakatte, S. Iftode, L.. Testing Cross-Platform Mobile App Development Frameworks. **30th IEEE/ACM International Conference on Automated Software Engineering**, 2015, pages 5-16.

GHERKIN, Gherkin · cucumber/cucumber Wiki · GitHub. Disponível em: <https://github.com/cucumber/cucumber/wiki/Gherkin>. Acesso em: 15/04/2018.

GOOGLE PLAY, WordPress – Apps para Android no Google Play. Disponível em: <https://play.google.com/store/apps/details?id=org.wordpress.android&hl=pt_BR>. Acesso em: 20/02/2017.

JBEHAVE, What is JBehave? Disponível em: <http://jbehave.org/>. Acesso em: 20/02/2017.

MESBAH, A. E. e P. Kruchten. Real challenges in mobile app development. In Proceeding of

the ACM/IEEE **International Symposium on Empirical Software Engineering and Measurement**, ESEM'13, pages 15-34. ACM, 2013.

MOHAMED, A., Ali. M., Joorabchi E. M. Detecting Inconsistencies in Multi-Platform Mobile Apps, **30th IEEE/ACM International Conference on Automated Software Engineering**, 2015, pages 12-32. @2015 IEEE.

MICROSOFT, Desenvolvimento Móvel entre plataformas no Visual Studio. Disponível em: <https://msdn.microsoft.com/pt-br/library/dn771552.aspx> Acesso em: 19/11/2017.

MCKEEMAN W. M. Differential testing for software. **Digital Technical Journal**, 10(1), December 1990, pages 12-24.

O'BRIEN, J.A. Sistemas de informação e as decisões gerenciais na era da internet. 2. ed. São Paulo: Saraiva, 2004.

PACHECO C., S. K. Lahiri, M. Ernst and T. Ball. Feedback-directed random test generation. **International Conference on Software Engineering (ICSE)**, 2007.

ROBOTIUM, GitHub - RobotiumTech/robotium: Android UI Testing. Disponível em <https://github.com/RobotiumTech/robotium>. Acesso em 15/04/2018.

SELENDROID, Selendroid: Selenium for Android. Disponível em <http://selendroid.io/>. Acesso em: 20/02/2017.

SELENIUM, Selenium - Web Browser Automation. Disponível em <https://www.seleniumhq.org/>. Acesso em 15/04/2018.

SOARES, I.. Desenvolvimento orientado por comportamento (BDD) - Um novo olhar sobre o TDD., pp. 91. Java Magazine, Rio de Janeiro (2011).

SMART, F. J.. Bdd in Action - Behavior Driven Development for the whole software lifecycle, **30th IEEE/ACM International Conference on Automated Software Engineering**, 2015, pages 12-16.

SAIKRI, Saikri - JBehave-Selendroid-Android-Automation. Disponível em <https://github.com/saikrishna321/JBehave-Selendroid-Android-Automation>. Acesso em: 20/02/2017.

SPECFLOW, SpecFlow - Binding Business Requirements to .NET Code. Disponível em <http://specflow.org/>. Acesso em: 20/02/2017.

WORLDWIDE MOBILE APP, Worldwide Mobile APP Revenue Forecast. Disponível em: <https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/>. Acesso em: 20/2/2017.

WORDPRESS, Blog Tool, Publishing Platform, and CMS. Disponível em <https://wordpress.org/>. Acesso em: 20/02/2017.