Gustavo Figueira Olegário

Um framework para geração de testes automatizados para aplicações mobile

Brasil

2019

Gustavo Figueira Olegário

Um framework para geração de testes automatizados para aplicações mobile

Trabalho de Conclusão de Curso submetido ao Curso de Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.

Universidade Federal de Santa Catarina

Centro Tecnológico - CTC

Departamento de Informática e Estatística

Ciências da Computação

Orientador: Ricardo Pereira e Silva

Brasil

2019

Gustavo Figueira Olegário
Um framework para geração de testes automatizados para aplicações mobile/ Gustavo Figueira Olegário. – Brasil, 2019-
85 p. : il. (algumas color.) ; 30 cm.

Orientador: Ricardo Pereira e Silva

Dissertação (Bacharelado) – Universidade Federal de Santa Catarina
Centro Tecnológico - CTC
Departamento de Informática e Estatística
Ciências da Computação, 2019.

1. Framework. 2. Testes. I. Ricardo Pereira e Silva. II. Universidade Federal de Santa Catarina. III. Bacharelado em Ciências da Computação. IV. Um framework para geração de testes automatizados para aplicações mobile

CDU 02:141:005.7

Gustavo Figueira Olegário

Um framework para geração de testes automatizados para aplicações mobile

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de Bacharel em Ciências da Computação, e aprovado em sua forma final pelo Curso de Ciências da Computação da Universidade Federal de Santa Catarina.

Dr. Prof. **Ricardo Pereira e Silva**
Orientador

Dr. Profa. **Patricia Vilain**
Banca

Mestre **Roberto Silvino da Cunha**
Banca

Brasil
2019

Este trabalho é dedicado a todos que não cansam de testar seus limites.

Agradecimentos

Os agradecimentos principais são direcionados aos meus pais, irmãos, amigos fiéis e à minha namorada, que sempre acreditaram em meu potencial e que me ajudaram, de alguma forma, a concluir minha graduação.

Agradeço também ao professor Ricardo por seu tempo e dedicação neste trabalho e que desde o início esteve disposto a ajudar na orientação.

Por último, mas não menos importante, agradeço a Deus por esta oportunidade. Pois, sem Sua graça, nada disso existiria.

Não se trata de bater duro. Se trata de quanto você aguenta apanhar e seguir em frente. O quanto você é capaz de continuar tentando. É assim que se consegue vencer.

Resumo

No escopo da engenharia de software, durante o processo de desenvolvimento de software, sabe-se que as fases de desenvolvimento da aplicação e testes são as que, costumeiramente, demandam muito esforço. Desta forma, uma ferramenta capaz de gerar testes automatizados para certas aplicações, no mundo de dispositivos *mobile*, pode ser vista como uma forma de acelerar o processo de desenvolvimento e permitir que os desenvolvedores se concentrem em atividades que demandem mais criatividade. Nesse sentido, a proposta desse trabalho é desenvolver um framework orientado a objetos capaz de apoiar tal atividade no contexto dos dispositvos móveis que possuam o sistema operacional Android. A finalidade do framework é diminuir o esforço e tempo investidos durante a fase de desenvolvimento de testes automatizados, com ênfase na elaboração desses.

Palavras-chaves: Engenharia de Software. testes. aplicações *mobile*. framework OO

Abstract

In the software engineering scope, through the development process of software, the stages of application development and tests are the ones that usually require more effort. Therefore, a tool capable to generate automated tests for certain applications, at the context of mobile phones, can be seen as an accelerator during the process of tests allowing the developers to focus on tasks that demand more intelligence than handwork. So, the main objective of this work is to develop a framework capble to support this task in mobile gadgets context that runs Android. The framework aims to decrease the amount of time and efforts invested during the period of development of automated tests.

Key-words: Software Engineering. tests. mobile application. framework OO

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

XP	Extreming Programming
UML	Unified Modeling Language
TDD	Test Development Driven
DRY	Don't Repeat Yourself
AFD	Autômato Finito Determínistico
LTS	Long Term Support
ISA	Instruction Set Architecture
AOT	Ahead-Of-Time
SDK	Software Development Kit
XML	Extensible Markup Language
API	Application Programming Interface

Sumário

1 Introdução e objetivos

1.1 Introdução

Desde o tempo em que o homem começou a programação de computadores, a atividade de teste de software sempre foi uma atividade vista com descaso e que só era executada se sobrasse tempo durante o projeto. Em alguns casos ela costumava ser usada como castigo para programadores que não cumpriam com suas funções (WAZLAWICK, 2012).

Com a chegada da crise do software (DIJKSTRA, 1971), não demorou muito tempo para que desenvolvedores percebessem que a atividade de criação e execução de testes era uma atividade muito importante e que também deveria ser incluída durante o planejamento do projeto.

Nos dias atuais, os testes são tão importantes que eles são incorporados por praticamente todas as metodologias ágeis, as quais, por sua vez, são as mais adotadas no mercado atualmente (JEREMIAH, 2017). A relevância dessa fase no processo de desenvolvimento é tão grande que, dependo da metodologia adotada, o programador deve primeiro programar o teste e depois o módulo o qual lhe foi designado.

Entretanto, devido a essa notória importância que a fase de testes ganhou nos últimos tempos, ela também tem sido uma das partes que mais tem tomado tempo em projetos(ERIKSSON, 2014) e que poderia ser facilmente automatizada em muitas situações já que costuma ser uma atividade que demanda mais trabalho manual do que inteligência. Nesse cenário, a proposta desse trabalho é desenvolver uma ferramenta para geração de testes automatizados para determinadas aplicações móveis, a fim de que os desenvolvedores possam focar em outras tarefas, enquanto o Framework se encarrega de gerar os testes. Os testes gerados de forma automática pela ferramenta são testes de funcionalidade que utilizam a técnica de caixa preta.

Sendo assim, nas próximas partes desse trabalho é, primeiro, apresentado de maneira mais precisa o tema framework orientado a objetos. Em seguida, é feito um estudo de técnicas de testes de software. Na sequência será abordado alguns aspectos no que se refere a desenvolvimento de aplicações *mobile* dentro da plataforma Android. Por último, a proposta de desenvolvimento do framework e uma apresentação de suas capacidades.

1.2 Justificativa

Como dito previamente na introdução, a fase que mais toma tempo durante um projeto de desenvolvimento de software é a parte de testes. Muitos projetos poderiam ter seu tempo reduzido ou incluir mais requisitos implementados por entrega, se os desenvolvedores pudessem investir mais tempo nisso ao invés de investirem esforços em desenvolvimento de testes. Sendo assim, este trabalho se propôs a implementar um framework capaz dar suporte a produção de testes de maneira automática, a fim de que os programadores possam focar naquilo que realmente importa: suas aplicações.

1.3 Método de pesquisa

1.3.1 Natureza: Aplicada

- Visando criar conhecimento a fim de minimizar um problema existente. No caso desse trabalho, reduzir o tempo gasto na implementação manual de testes automatizados para aplicações *mobile* da plataforma Android, utilizando como meio a implementação um framework em Java.

1.3.2 Abordagem: Qualitativo

- As soluções já utilizadas em trabalhos passados similares a esse já são uma fonte de dados para o problema que está sendo abordado, não necessitando de cálculos específicos para capturar as informações necessárias.

1.3.3 Objetivo: exploratório

- Almejando a compreensão do problema através de trabalhos relacionados elicidos nesse trabalho usando estudo de soluções passadas para problemas similares ao que está sendo abordado.

1.3.4 Procedimentos técnicos: Bibliográficos, Estudo de Caso

- O desenvolvimento de um framework automatizado que auxilie na elaboração de testes automazidos no contexto de aplicações para dispositvios *mobile* já foi uma ideia previamente explorada em estudos passados. Busca-se replicar a mesma ideia para a plataforma Android.

1.4 Objetivo geral

Analisar e desenvolver um framework orientado a objetos capaz de dar suporte a produção de testes automatizados para aplicações específicas de dispositivos que possuam como sistema operacional o Android.

1.5 Objetivos específicos

1. Diminuir tempo e esforço investidos durante a fase de testes de um projeto de software de determinadas aplicações Android.

2. Desenvolver uma ferramenta gratuita que permita desenvolvedores Android focarem em tarefas que demandem mais inteligência do que trabalho braçal.

1.6 Estrutura do documento

Este capítulo inicial aborda a justificativa do trabalho, bem como seus objetivos geral e específicos. No capítulo 2, são apresentados os conceitos relevantes ao trabalho para melhor compreensão do leitor. O capítulo 3 apresentará trabalhos relacionados. Em seguida, no capítulo 4 é mostrado como o framework foi implementado e, por último, no capítulo 5 mostram-se os resultados obtidos juntamente com sugestões de trabalhos futuros.

Parte I

Fundamentação teórica

2 Frameworks

Como este trabalho se propõe a implementar um framework de geração automática de testes, é importante que este conceito esteja muito bem definido. No mundo da programação, os desenvolvedores estão acostumados a usar bibliotecas ou código de terceiros. Esse processo geralmente resume-se, basicamente, a executar um comando de inclusão de arquivo e as funções previamente implementadas já podem ser usadas.

O uso de bibliotecas é algo que geralmente traz melhorias significativas para o código que está sendo desenvolvido, visto que o programador está fazendo reuso de módulos que já foram usados no passado e, consequentemente, estão menos suscetíveis a erros. Além disso, algumas bibliotecas estão constantemente recebendo atualizações a fim de fornercer novas facilidades ao programador, bem como minimizar o número de defeitos.

Entrento, o conceito de framework em si, vai muito além do que uma simples coleção de funções que podem ser reusadas entre vários projetos. O framework é responsável por orientar o programador a forma que ele deve desenvolver o projeto, ou pelo menos uma parte dele. Ele geralmente indica ao programador aonde o código deve ser produzido e de que forma.

Esse estilo de programação não só fornece uma estrutura mais sólida ao projeto, pois todos os programadores da equipe devem seguir o mesmo padrão, como também segue a mesma filosofia das bibliotecas, em que o foco principal é o reuso de código entre projetos que compartilham o mesmo domínio de problema.

Nas próximas seções será apresentado como foi o surgimento inicial do conceito de framework, quais os principais pontos positivos e negativos de se usar uma ferramenta como essa e os princípios seguidos por frameworks.

2.1 Origem dos Frameworks

Sabe-se que em seus primórdios, a indústria de software tinha como principal foco de atuação quase que única e exclusivamente a resolução de problemas matemáticos. Com o passar dos anos, as linguagens de programação foram ficando mais robustas e o hardware também foi se desenvolvendo. Esse processo, permitiu que computadores se popularizassem e eis que a necessidade por novos softwares foi surgindo.

Entretanto, os softwares que estavam, e continuam, sendo exigidos pelo mercado eram aplicativos com complexidade cada vez maior. Não bastasse isso, as datas de planejamento de projeto começaram a ficar ainda mais apertadas para poder suprir a necessidade de seus clientes. Esse cenário dantesco que começou a se formar para os progamadores,

começou a levar os desenvolvedores a filosofar sobre qual seria a melhor forma de resolver tal situação.

Analisando diferentes projetos, ficou evidente para muitos programadores que eles estavam sofrendo com o mal de estar o tempo todo "reiventando a roda". Em outras palavras, em diferentes projetos sempre se estava reimplementando o mesmo módulo ou um muito similar. Dessa forma, ficou claro que não havia necessidade de sempre estar se desenvolvendo um mesmo componente para diferentes projetos, bastava que se tirasse proveito do reuso que o problema estaria resolvido (LEACH, 2011).

É claro que esse processo de reuso de diferentes partes de um software, em termos de granularidade, foi crescendo aos poucos através dos tempos (JALENDER, 2014). Inicialmente, começou-se com funções básicas como na linguagem C, por exemplo, foi avançando para estruturas mais complexas como classes até que chegou no que conhecemos por frameworks nos dias de hoje.

Um framework é, então, nada mais que um conjunto bem estruturado de classes relaciodas através de associações e heranças (JOHNSON, 1988). Porém esse conjunto não é totalmente implementado, algumas partes ficam não terminadas, propositalmente, para poder dar uma certa flexibilidade de domínio de problema. Ou seja, nada andiantaria o framework ser totalmente implementado, pois aí ele não daria nenhum grau de liberdade para o usuário e o único problema que ele resolveria seria o que ele mesmo implementa.

No framework FraG, um framework voltado a suportar o desenvolvimento de programas de jogos de tabuleiro (SILVA, 1998), podemos ver um exemplo disso. Existe a classe Board, que representa o tabuleiro de um jogo. Porém, um tabuleiro pode variar muito de jogo para jogo. Dessa forma deixam-se alguns métodos abstratos e o usuário apenas estende essa classe para seu domínio de aplicação e implementa os métodos que necessita e adiciona os atributos que forem necessários, mantendo um código organizado e seguindo boas práticas de orientação a objetos. A modelagem UML (*Unified Modeling Language*) utilizada para esse framework pode ser observada na Figura 1.

Dessa forma, o desenvolvedor apenas precisou desenvolver algumas pequenas partes para ter uma aplicação totalmente pronta, tendo a maior parte do software implementado pelo framework.

A implementação de um framework consiste principalmente em identificar qual domínio de problema ele deve dar suporte. No caso do FraG era um framework voltado a desenvolvimento de jogos de tabuleiro. O processo de modelagem para um framework requer que o desenvolvedor identifique as classes mais comuns de um domínio problema, também conhecidos como "hot spots"(SILVA, 1998). Esse processo não costuma ser trivial, pois o responsável pela modelagem do framework deve estudar ao menos 3 aplicações que compartilham o mesmo domínio (JOHNSON, 1993), porém sem se prender muito aos

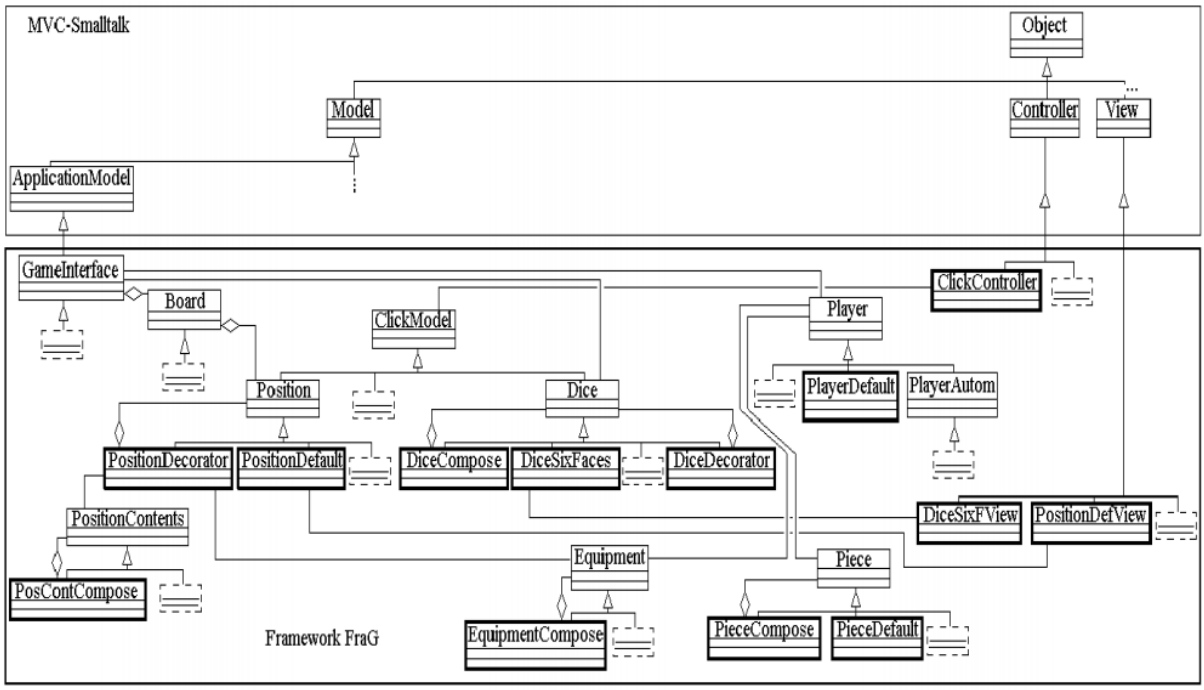

Figura 1 – Modelagem do framework FraG.

(FONTE: Adaptado de SILVA, 1998)

detalhes do problema específico.

Após identificadas as classes que semanticamente são mais utilizadas em um certo domínio, inicia-se o processo de modelagem que deve definir quais classes deverão ser implementadas e quais serão redifinidas pelo desenvolvedor. A partir daí, consegue-se obter resultados de quão eficiente o framework desenvolvido é. Para analisar esse dado, basta verificar em um projeto que utiliza o framework a porcentagem de classes implementadas do zero e quais foram reaproveitadas do framework. Isso é claro desconsiderando classes como de interface com o usuário, por exemplo, visto que usualmente são totalmente dependendes do domínio da aplicação.

2.2 Aspectos postivos e negativos

Quando um framework é utilizado em um projeto de software, é importante estar atento aos principais benefícios e problemas em termos de implementação. Quando um projeto é feito "from scratch", ou seja, todas as classes são produzidas do zero pelo próprio time de desenvolvimento, os programadores tem total liberdade para implementar a arquitetura como julgarem melhor. Um exemplo de modelagem desse tipo de implementação pode ser melhor observado na Figura 2, onde todas as classes com a borda preta foram implementadas pelo desenvolvedor. Contudo, o problema disso, obviamente, é que

nada é reusado, a aplicação fica muito suscetível a defeitos e demanda muito esforço.

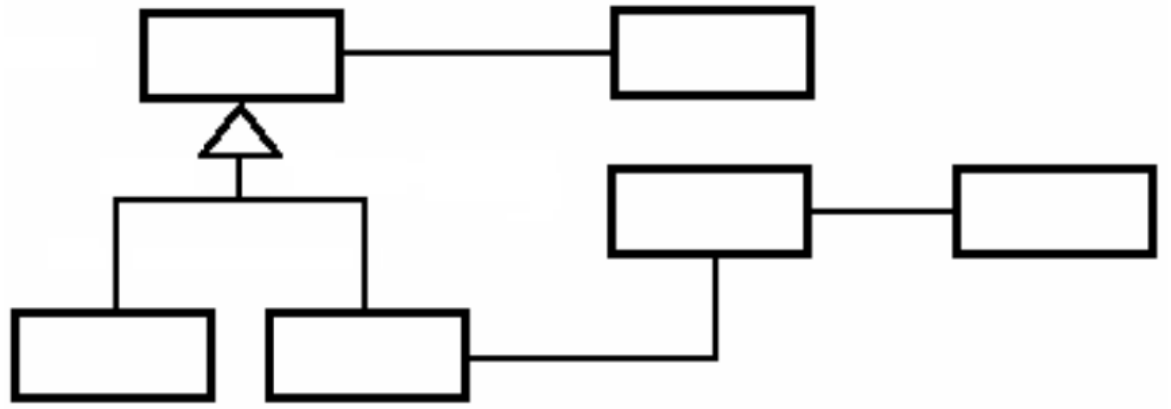

Figura 2 – Um exemplo de aplicação *from scratch*

(FONTE: Adaptado de SILVA, 1998)

Quando um framework é utilizado para suprir as necessidades um projeto, muitas das classes que inicialmente seriam implementadas do zero, já estão praticamente prontas, necessitando, geralmente, apenas de poucas sobrescritas de métodos. Os ganhos nesse processo é que os desenvolvedores têm seus esforços minimizados, uma vez que boa parte das classes já estão totalmente implementadas. Outro aspecto positivo ao se utilizar um framework é que a granularidade do reuso é muito maior do que se comparada a um projeto que não usa frameworks. Em outras palavras, ao utilizar-se um framework dentro de uma aplicação o reuso costuma atingir facilmente dezenas de classes.

Quanto aos aspectos negativos, o mais fácil de ser observado é que os desenvolvedores da aplicação ficam presos à arquitetura e ao controle de fluxo de dados impostos pelo framework. Isso ocorre devido ao fato de que, como a maioria das classes do domínio do problema já estão sendo implementadas pelo framework, tanto a arquitetura como o fluxo de controle já são, por definição, pré-estabelecidos.

Outro grande ponto negativo é que os programadores precisam ter um bom conhecimento de como funciona o framework para poder utilizá-lo corretamente. É necessário ter conhecimento de como o framework é implementado para saber quais classes já foram programadas e acabar não duplicando uma funcionalidade já existente. O problema é que para conseguir tais informações os desenvolvedores terão que conferir a documentação disponível ou o código fonte. A primeira alternativa, apesar de ser a melhor, nem sempre é viável, pois nem sempre é produzida ao final de um projeto. Existem casos em que mesmo a documentação não é capaz de detalhar o que foi implementado, seja pela não

completude ou por não ter sido atualizada durante o desenvolvimento do projeto.

Já a alternativa de analisar o código fonte é, de longe, a maneira mais árdua de aprender como a ferramenta funciona. Exige do programador uma atividade de engenharia reversa e costuma ter um baixo rendimento. Em alguns cenários pode até mesmo deixar o desenvolvedor mais confuso ainda. Isso é claro considerando um framework de código aberto, do contrário nem mesmo acesso ao fonte o desenvolvedor teria.

2.3 Princípios de frameworks

Nesta secção são apresentados alguns conceitos implementados pela maioria dos frameworks, mas isso não significa que caso algum framework não implemente alguma dessas ideias ele não possa ser classificado como tal. Voltando mais uma vez ao exemplo do framework do FraG, percebe-se os frameworks se baseiam fortemente nas caracrísticas de associações e heranças.

Pode-se compreender o motivo pelo qual a herança é tão usada no universo de frameworks. Quando utilizada em classes abstratas, por definição, é exigido que o programador implemente os métodos abstratos definidos pela superclasse. Isso garante que sempre que qualquer objeto que herde a classe abstrata, no momento que for utilizado interna ou externamente pelo framework, os métodos estarão implementados, o que possibilita uma certa customização dentro do framework e é aí que vem a flexibilidade dos frameworks dentro de um domínio de problema.

Tal fato pode ser observado na modelagem do par *Template* e *Hook*. Nesse modelo, o método *Template* é um algoritmo sempre estável que inicializa objetos que sempre serão necessários para o framework. Entretanto, como existem objetos que podem variar de aplicação para aplicação ao final da execução do método *Template* o método *Hook* é executado (GAMMA, 1994). Esse, por sua vez, é um método que deve ser sobrescrito pelo usuário para que ele possa inicialzar demais objetos que necessite para sua aplicação. Na Figura 3, é possível observar alguns métodos, dentro do framework FraG, que são classificados como template ou hook.

Outro fato que leva a relação de herança ser tão explorada no mundo de frameworks, é o que se chama de princípio de Hollywood (*"Don't call us, we'll call you"*) (FOWLER, 2005). Esse princípio decorre do fato de que quando o desenvolvedor estiver usando a ferramenta ele não deve instanciar objetos do framework e chamar métodos. Pelo contrário, ele deve criar novas classes que herdem de outras já definidas pelo framework. Usualmente, são classes abstratas. Dessa forma, quando a aplicação estiver sendo executada, o framework saberá quais classes foram herdadas e quais métodos foram escritos. Dessa forma, ele se encarrega de chamar os objetos executando corretamente os novos métodos sobrescritos. Daí vem o lema: *"Don't call us, we'll call you"*.

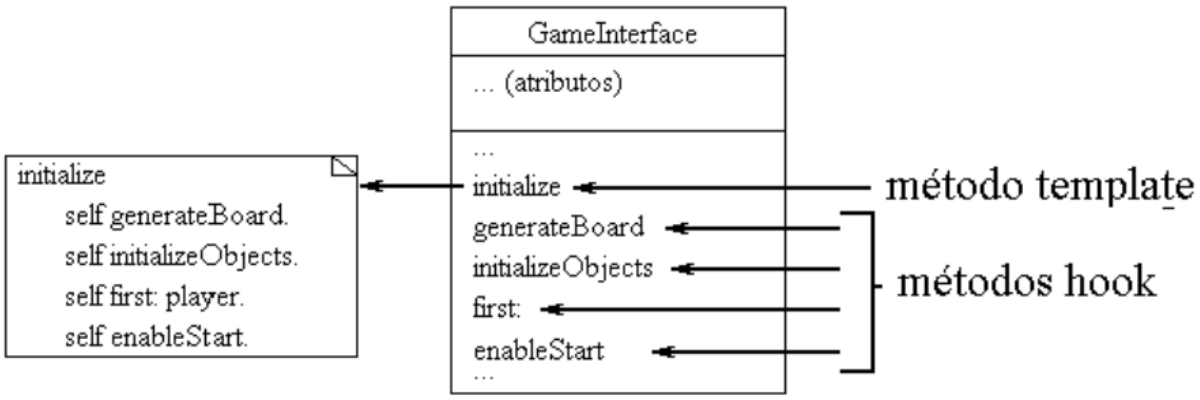

Figura 3 – Modelagem dos métodos Template e Hook

(FONTE: Adaptado de SILVA, 1998)

Já o fato de existir tantas relações de associações no framework, vem do fato de que os *Design Patterns* são amplamente explorados neste universo de soluções. *Design Patterns* (GAMMA, 1994) são modelagens elegantes e que solucionam problemas que já estão muito bem conhecidos na modelagem de software. Na modelagem do FraG, podemos perceber em que há casos em que está sendo usado simultaneamente 3 *Design Patterns*, para tratar um único problema no projeto. São eles: *factory, composite, decorator*. Não cabe aqui nesse escopo explanar cada um desses *Design Patterns*, muito menos todos os existentes, mas é importante deixar claro que cada framework, muito provavelmente, terá padrões reusados conforme a necessidade.

Outro ponto forte, é que o framework deve ser capaz de dar suporte a uma grande quantidade de aplicações com o mesmo domínio que ele. Como foi visto, esse processo na modelagem do framework vem do fato de várias aplicações que compartilham o mesmo domínio terem sido estudas e terem seus módulos semelhantes identificados e implementados de forma genérica na nova ferramenta. Feita essa parte, basta que os desenvolvedores escolham precisamente quais classes devem ser reescritas em cada projeto, e terão boa parte da implementação já realizada.

3 Testes

Nos primórdios da história do desenvolvimento de software, a etapa de testes era geralmente um parte vista com maus olhos pelos programadores e rotineiramente só se realizava quando "sobrava tempo"no projeto. Com o passar do tempo, foi ficando evidente para as empresas que a parte de testes é essencial para assegurar a qualidade do produto (WAZLAWICK, 2012). Ela não só é importante para comprovar o correto funcionamento, como também ajuda a garantir estabilidade durante a refatoração de código, pois se um teste estava obtendo sucesso antes da alteração e passou apresentar falhas depois das mudanças é porque certamente houve algum equívoco por parte de quem alterou o código.

Nessa parte do trabalho é de suma importância deixar esclarecido que um projeto que foi conduzido sem testes, pode atingir um patamar satisfatório de qualidade ao longo do processo, mas isso provavelmente terá um custo alto, em que o programador segue a filosofia do *code and fix*, em que a medida que ele vai codificando, vai arrumando os erros que encontra. Isso faz com que o tempo de projeto aumente, bem como os custos.

É importante deixar claro, desde já, que o teste de software em momento algum supõe que o módulo implementado esteja livre de defeitos. A única coisa que o teste garante para o desenvolvedor é que para as entradas especificadas o componente está se comportando como o programador desejava. O teste costuma ser implementado pensando em como usuário poderia causar erro no módulo, mas também não esquecendo da *happy path*, também conhecido como sequência de estados em que não há ações inválidas (MESZAROS, 2011). O teste é, então, nada mais que um conjunto de código que coloca o módulo que está sendo testado em um determinado estado, ou seja com valores específicos para as variáveis e analisa como o módulo está reagindo.

Pensando em um cenário mais prático, propõe-se um módulo simples em que a única responsabilidade seja verificar se um dado CPF é válido ou não, mesmo que possua uma máscara. Abaixo, podem-se observar, na Figura 4, as assinaturas de método do mesmo na linguagem *Python*.

Apesar de nenhum dos métodos ter sido realmente implementado, é perfeitamente possível desenvolver um arquivo de teste para esse módulo. Basta que o programador tenha conhecimento do algoritmo de validação de CPF. Dessa forma, ele saberá quando um CPF é válido mesmo estando sem a máscara (i.e. formatação). Ou seja, quando o teste for implementado basta ele escrever que os CPFs que obedeçam o algoritmo devam ser classificados como válidos, enquanto que os inválidos devem ser rejeitados. Com o teste desenvolvido, basta o programador executá-lo e obviamente o teste retornará uma mensagem de erro, isso porque o módulo ainda não foi implementado. Caso o teste retorne

```
class CpfValidator(object):
  def __init__(self):
    # TODO
    pass

  def cpf_is_valid(cpf):
    # TODO
    pass

  def cpf_with_mask_is_valid(cpf_with_mask):
    # TODO
    pass
```

Figura 4 – Módulo com assinaturas de método para validar CPF
(FONTE: Imagem feita pelo autor)

somente mensagens de sucesso é porque o teste possui algum equívoco.

Com o teste produzido, o progamador dever seguir aquilo que é conhecido como *baby steps*. O teste, uma vez escrito e executado, dirá qual o erro está acontecendo, seja função não implementada, método retornando valores incorretos, objeto nulo. Dessa forma, o desenvolvedor irá corrigir o erro e irá executar novamente. Provalvelmente na segunda iteração, o teste apontará um novo erro. O programador deve, então, proseguir nesse processo até que o teste retorne somente mensagens de sucesso.

Esse processo é também conhecido como desenvolvimento orientado a testes, cuja a sigla em inglês é TDD. A Figura 5 seguinte ilustra bem esse processo e que também é conhecido como *red, green and refactor*, já que o programdor começa inicialmente na fase de erros (*red*), faz somente o necessário para que o módulo passe no teste (*green*) e melhora no final a qualidade do código a medida do necessário (*refactor*) (BECK, 2004).

Ainda no contexto de desenvolvimento de testes, é importante que os desenvolvedores e o time de qualidade estejam constantemente revisando o código produzido por cada um. Isso é importante, pois quem está desenvolvendo um módulo já sabe como o código funciona e fica mais difícil em perceber os *bugs*. Dessa forma, quando alguém que não escreveu o código faz uma revisão de um módulo, fica muito mais evidente para quem está revisando identificar o erro. Por isso, os cenários de teste devem ser sempre atualizados e refatorados (REID, 2016).

Outro ponto importante a ser ressaltado é a questão das *pré e pós condições* de um dado caso de teste. A pré-condição é uma obrigatoriedade que deve ocorrer antes que um dado método inicialize. Pensando num exemplo mais prático, imagina-se uma outra classe que executa algum procedimento depois de a classe de validaçao de CPF ter comprovado que o dado informado está correto. Nos métodos que essa nova classe

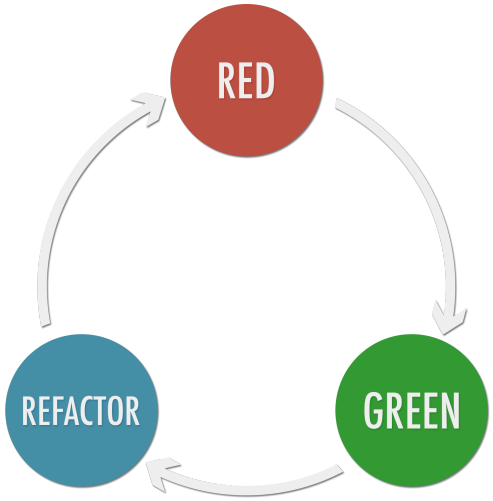

Figura 5 – Diagrama que representa o padrão RGF

(FONTE: Adaptado de FOWLER, 2014)

implementa, o desenvolvedor não precisa se preocupar se o dado está correto ou não, pois a classe anterior já o fez. Isso é o que se chama de pré-condição. Já a pós-condição é o estado que o sistema deve se encontrar após o procedimento ter sido executado. Muitas vezes, o teste deve verificar se o sistema chegou no estado esperado. No mesmo exemplo, supõe-se que a nova classe pesquise no banco o usuário pelo CPF e mande um email para o mesmo. Para ver se a pós-condição foi atendida, o sistema deve verificar se o servidor de emails mandou um email para o usuário com aquele CPF.

Nos dias atuais, o mercado é repleto de ferramentas e frameworks capazes de auxiliar os desenvolvedores durante a implementação de testes, fornecendo objetos falsos, também conhecidos como *mocks*, funções de asserção, execução de testes em ordem aleatória. Como o intuito de trabalho é desenvolver um framework que seja capaz de automatizar a produção de testes, esse capítulo tratará essencialmente de tipos de teste, níveis e técnicas de teste. Neste trabalho, testes relacionados a segurança e desempenho, como de carga e stress, não serão contemplados, pois fogem do escopo do que o trabalho se propõe a fazer.

3.1 Modelagem e implementação de testes

A modelagem de testes é uma fase importante, visto que é responsável por garantir o mínimo de qualidade no módulo que está sendo implementado. O desenvolvimento de uma aplicação pode seguir o processo de TDD em que o programador primeiro implementa o teste e depois desenvolva o módulo que lhe foi designado (BECK, 2002). Porém, nada impede que o desenvolvedor implemente inicialmente o módulo e depois avançe para a

parte de teste.

Entretanto, durante a modelagem de testes, o mínimo que o desenvolvedor precisa conhecer para poder implementar o teste são os métodos do artefato que lhe foi atribuido, bem como a ideia de seus algoritmos. Dessa forma ele consegue separar as entradas para o teste em dois grandes conjuntos: o conjunto das entradas válidas e das inválidas. De posse de tais informações básicas, ele pode escrever testes que seguem o *happy path* e aqueles que devem lançar algum tipo de exceção (MESZAROS, 2011).

Para deixar mais claro, retorna-se ao exemplo do módulo de validação de CPF escrito em *Python*. Como pode-se perceber pelo código fornecido na Figura 4, é facilmente perceptível que a linguagem é **fracamente tipada**, ou seja, é deixado ao interpretador inferir a tipagem de cada variável. Para efeitos práticos, será considerado que os parâmetros fornecidos para os dois métodos de validação sejam do tipo *string*. Um possível cenário de teste que segue o *happy path* seria fornecer um CPF válido e o teste iria retornar dizendo que tudo ocorreu como esperado. Por outro lado, se em algum cenário de teste em que seja fornecida uma sequência inválida de inteiros, o correto seria que o módulo lançasse alguma exceção do tipo *InvalidType* e o teste detectasse que essa exceção foi lançada. Ao perceber que houve uma exceção o teste deve informar, novamente, tudo ocorreu como esperado. Do contrário, ele deve trazer um relatório escrito dizendo que estava sendo esperado uma exceção, porém a mesma não ocorreu.

Na modelagem de testes é importante também que durante a implementação exista o teste de entradas aleatórias para simular uma possível simulação com o usuário. Nos dias atuais, existem várias bibliotecas capazes de gerar dados falsos que cumprem com este objetivo e são capazes de criar informações falsas, porém válidas, como: nomes, emails, senhas, cartões de crédito, CPF. Sendo assim, é importante também que durante a modelagem de teste o desenvolvedor preveja ao menos um cenário em que ele possa tirar proveito dessas bibliotecas.

A modelagem de teste em si pode ser abordada de inúmeras formas (Petrenko, 2012). Dentro da ciência da computação, o mais comum é enxergar o teste do módulo como sendo um **autômato finito** (WAZLAWICK, 2012). Os estados representam o objeto em si e as transições seriam os métodos e os possíveis valores de retorno, ou exceções, dependendo dos valores passados para o método. O objetivo do teste, vendo a modelagem por essa ótica, seria verificar se a máquina chegou num estado de aceitação, ou não. Nesse sentido, é possível ver o autômato representando o teste para a validação de um CPF, do módulo representado anteriormente, na Figura 6.

Olhando para o autômato descrito, o desenvolvedor se quisesse garantir o máximo de qualidade possível, teria que testar todos os caminhos possíveis partindo do estado inicial. Nesse exemplo, a tarefa de testar o módulo é perfeitamente realizável, uma vez que existem poucas transições. Porém, pensando em casos mais reais em que os módulos

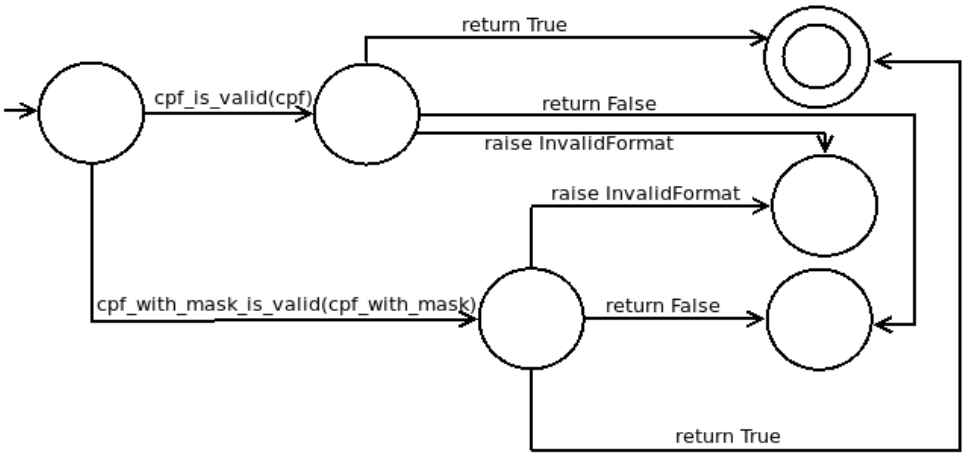

Figura 6 – Diagrama que representa o teste do módulo de CPF na ótica de AFD

(FONTE: Imagem feita pelo autor)

possuem pelo menos de 6 a 8 procedimentos e que podem ter suas próprias exceções, a tarefa de testar completamente um artefato começa a ficar inviável. Nesse sentido, o programador deve priorizar as entradas mais comuns e algumas exóticas para garantir uma certa qualidade.

Ainda dentro do contexto do autômato, é importante lembrar que não foram representadas mais transições dos estados mais à direita em direção a outros estados por uma mera questão de legibilidade. Entretanto, dentro de um contexto de um objeto que é utilizado várias vezes, a chamada repetitiva de vários métodos sucessivamente pode acarretar, eventualmente, na mudança de algum atributo que poderia encadear alguma mudança no comportamente do método, dependendo do algoritmo implementado. Sendo assim, é importante que o programador tenha em mente essas possibilidades também.

Já no que se refere à parte de implementação, a implementação do padrão de projeto *factory* costuma ser muito comum, visto que é uma solução elegante para os testes (FOWLER, 2003). Esse padrão se faz essencial para manter um código mais limpo, visto que em cada cenário de teste o desenvolvedor precisará de objetos com valores de atributos diferentes. Dessa forma, ele pode usar uma biblioteca que implemente o padrão para os objetos necessários para o teste e para cada cenário ele solicita objetos diferentes para a fábrica. Isso ajuda a manter um código mais legível, visto que se esse padrão não fosse utilizado, para cada teste o programador teria que sempre estar instanciando um objeto da mesma classe e configurando os valores manualmente, o que, por sua vez, não segue o príncipio básico do DRY (*Don't repeat youself*).

3.2 Níveis de teste

Dentro do universo de teste de software, pode-se classificar os testes pelos seus níveis ou fases. A classificação decorre do fato de que a medida que a aplicação vai crescendo os testes também precisam ter uma complexidade maior, pois o número de combinações de entradas possíveis do usuário vai aumentando. Segundo Wazlawick (2012), os testes, quando divididos dessa forma, podem pertencer alguma dessas classes: unidade, integração, sistema, aceitação e o de regressão. O conceito de níveis deriva daquilo que é conhecido por *Pirâmide de testes*, ilustrada na Figura 7. A pirâmide de testes correlaciona a quantidade de cada nível de teste junto com seus custos e tempo de execução. Na base da pirâmide, portanto, identificam-se os testes com maior expressividade, em número, e que exigem pouco esforço para serem programados e que executam mais rapidamente, enquanto que no topo é justamente o inverso. Nas próximas seções serão apresentadas brevemente os níveis de: unidade, integração, funcionalidade, aceitação e regressão, de acordo com Myers (2004).

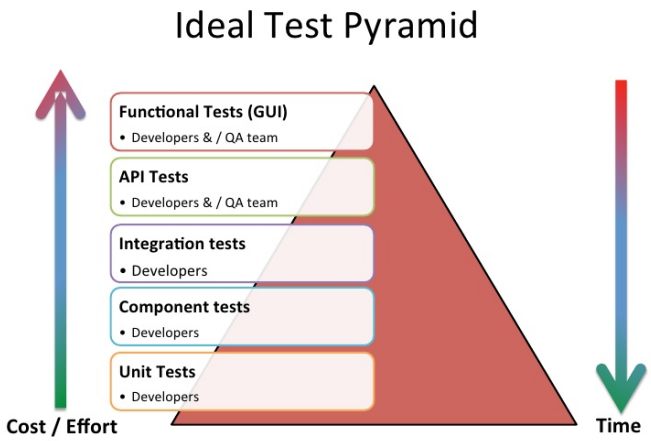

Figura 7 – Pirâmide que correlaciona quantidade de cada nível de teste

(FONTE: Adaptado de BAGMAR, 2012)

3.2.1 Teste de unidade

O teste de unidade é o teste mais básico de um software. O intuito dele é basicamente testar a classe que está para ser implementada ou que recém foi programada. Boa parte dos testes de uma aplicação são classificados dessa forma, uma vez que são os mais baratos de serem desenvolvidos e os mais rápidos de serem executados (FOWLER, 2012). Este nível de teste visa somente fazer o teste de uma classe isolada. Caso a classe que está sendo testada interaja de alguma forma com outras classes, seja por atributos ou parâ-

metros de métodos, o programador, deve, então, usar objetos falsos, também conhecidos como *mocks*, para simular a classe que possui relação com a classe que está sendo testada. Essa metodologia garante que caso qualquer defeito detectado pelo teste, o defeito vem única e exclusivamente da classe que está sendo testada, e não de classes terceiras.

3.2.2 Teste de integração

Como o nome já sugere, o objetivo desse teste é garantir a qualidade de dois ou mais módulos que trabalham em conjunto. Nesse nível deve-se testar todos os módulos que possuem alguma relação do tipo: associação, composição ou agregação. Ao invés de usar objetos falsos, como era feito no nível anterior, usam-se objetos verdadeiros e testam-se os métodos pelos quais estes objetos trocam mensagens. Em casos em que um dos módulos estava funcionando no teste de unidade e passou a não funcionar no teste de integração, é porque muito provalmente o módulo que foi integrado está com algum problema. Este tipo de teste não deve cobrir integrações com sistemas de terceiros, APIs por exemplo.

3.2.3 Teste de funcionalidade

O teste de funcionalidade é o teste em que os desenvolvedores utilizam o sistema no modo de produção fingindo ser um usuário comum e simulando suas respectivas ações. Esse teste, geralmente usa como base algum caso de uso descrito no processo de engenharia de requisitos e também busca achar eventuais defeitos na interface de usuário. Esse teste atualmente pode ser facilmente automatizado, mas como a interface é um componente que usulamente está sendo alterada, algumas empresas ainda preferem realizá-lo de maneira manual e automatizar somente quando a interface já está bem estabelecida.

3.2.4 Teste de aceitação

O teste aceitação é o teste mais importante de todos, pois é nesse teste em que o usuário utiliza o sistema sem auxílio dos desenvolvedores. Ele é o mais importante, pois o cliente dirá se as funcionalidades requisitadas foram cumpridas e valida se a coleta de requisitos foi feita da forma correta.

3.2.5 Teste de regressão

O teste de regressão é um nível de teste que busca executar toda a base de testes para um sistema que foi recém refatorado ou que teve uma nova versão lançada. O nome vem do fato de que caso o sistema que foi atualizado estava sendo aprovado nos testes e passou a ter erros, é dito, então, que o sistema regrediu (WAZLAWICK, 2012).

3.3 Técnicas de teste

As técnicas de teste(MYERS, 2004) indicam a forma como os casos de teste serão implementados. Todas elas possuem em comum o objetivo de encontrar falhas no artefato desenvolvido. As principais técnicas que podem ser aplicadas são: caixa-branca, caixa-preta e caixa-cinza. Os próximos tópicos detalharão cada uma dessas técnicas.

3.3.1 Técnica da caixa preta

É também conhecida como teste comportamental e, usualmente, é a técnica mais aplicada. O desenvolvedor fornece um conjunto de entradas e utiliza uma função de asserção para avaliar o resultado retornado, sem se preocupar com a execução interna do código. Essa técnica pode ser aplicada em qualquer nível de teste. Obviamente, quanto mais entradas sejam testadas, melhor o teste é. Como já foi dito anteriormente, é impossível testar todas as entradas possíveis, para a maior parte das situações práticas. Nesse caso, o programador pode utilizar classes de equivalência para maximizar os casos de teste coberto. As classes de equivalência nada mais são do que dois subconjuntos do universo do conjunto de entradas possíveis, em que um subconjunto representa as entradas válidas e o outro as entradas inválidas. Uma abstração dessa técnica pode ser contemplada na Figura 8.

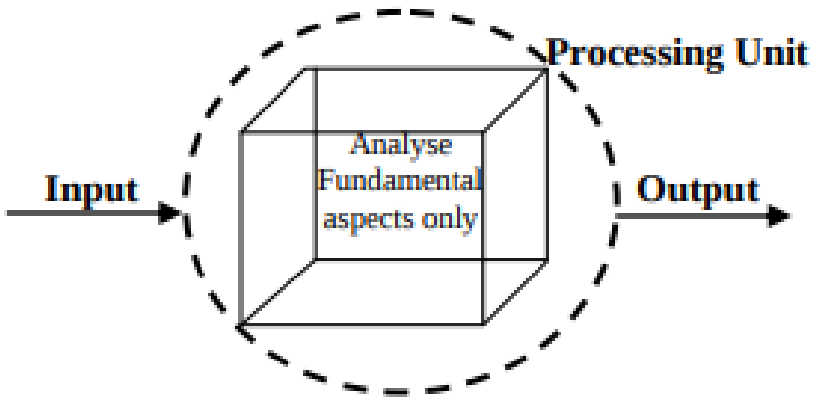

Figura 8 – Visão da técnica da caixa preta

(FONTE: Adaptado de KHAN, 2012)

A filosofia por trás dessa técnica, encontra-se na ideia de que basta o programador saber o que um determinado método deveria retornar para um certo parâmetro X. Para melhor exemplificar essa ideia, propõe-se mais adiante um módulo escrito na linguagem *Python* capaz de fazer cálculo da função fatorial. O módulo pode ser observado na Figura 9. Mesmo que, por um acaso, o progamador não soubesse como a função fatorial funcionasse ou se ainda ela fosse muito complexa envolvendo cálculos matemáticos avançados, basta

que ele saiba que fornecendo um valor *X*, o método deve retornar obrigatoriamente *X*(X-1)*(X-2)*...*1*.

Dessa forma, ao criar um teste para esse método basta que o programador escreva que quando o algoritmo for executado com o valor 4, por exemplo, deve ser retornado o valor 24. Caso seja retornado qualquer coisa diferente desse valor ou uma mensagem de erro é porque, então, existe um defeito no módulo implementado. É claro que a a função de cálculo de fatorial é um algoritmo trivial de ser testado. Porém, em projetos mais complexos de software há a necessidade de se explicitar o que cada método deve retornar a fim de que quem escrever um teste baseado na técnica da caixa preta possa fazê-lo sem saber como o método é realmente implementado.

```
class FactorialCalculator(object):
  def __init__(self):
    pass

  def factorial(n):
    result = 1
    while (n != 0):
      result *= n
      n -= 1
    return result
```

Figura 9 – Módulo com implementação de método que calcula fatorial

(FONTE: Imagem feita pelo autor)

Usualmente, nesse tipo de técnica, costuma-se buscar valores limites para verificar se o módulo está se comportando como deveria, uma vez que os defeitos de um software costumam ficar em suas frestas (WAZLAWICK, 2012). Considerando que o método recebe somente valores inteiros, o algoritmo possui uma grande falha no que se refere às frestas. Ele, apesar de calcular corretamente os valores para qualquer inteiro não negativo, se eventualmente o método for executado usando como parâmetro um inteiro negativo, então o módulo ficará computando eternamente.

Aplicando o princípio dos conjuntos de equivalência para um possível arquivo de teste para este módulo, o desenvolvedor poderia criar testes usando os seguintes valores como parâmetro: -2, -1, 0, 1 e 2. Para os valores negativos, o programa, necessariamente deveria lançar uma exceção e o teste, ao receber a exceção, deveria afirmar que tudo ocorreu como esperado. Nesse caso, para arrumar o módulo implementado, ao em vez de usar o operador de diferença, basta utilizar o operador de >. Para os demais valores, basta criar os casos de teste utilizando a técnica da caixa preta como foi descrita anteriormente.

3.3.2 Técnica da caixa branca

A técnica da caixa branca, por sua vez, é uma técnica muito mais robusta que se comparada à da caixa preta. A ideia consiste em testar qual fluxo o código está sendo executado no componente. Esse tipo de teste visa eliminar trechos que nunca são alcançados dentro de um método ou que sejam reduntantes. Quando esta técnica é aplicada todas as possibilidades de fluxo de um método devem ser testados (i.e. *while, for, if, else, try, catch, finally*).

Esse tipo de teste não necessariamente procura erros dentro de um módulo, mas busca otimizá-lo, em termos de processamento. Isso decorre do fato de que, podem haver estruturas de controle de fluxo que estão sendo declaradas, mas nunca são alcançadas. Ou seja, o módulo está funcionando corretamente, justamente por tais estruturas não estarem sendo executadas. Dessa forma, a finalidade do teste é diminuir as linhas de código e deixar ele com um desempenho ainda maior.

Nessa subsecção, cabe um comentário para complementar o que foi apresentado quanto à questão de modelagem de testes baseado em autômatos. Conforme Wazlawick (2012 em p. 400) é apresentado uma forma de modelar testes, baseados nessa técnica. A ideia consiste em modelar um autômato finito em que cada nodo representa uma linha ou um conjunto de linhas sequenciais que contenham no máximo uma estrutura de controle e as arestas representam o atendimento ou não da condição imposta pela estrutura. A modelagem com esse tipo de autômato para o método de cálculo de fatorial pode ser visto na Figura 10.

Através do diagrama, é percebido que, então, o princípio do teste é passar por todas as arestas do automâto e caso alguma aresta não seja utilizada em nenhuma hipótese é porque a estrutura jamais é alcançada ou possui uma condição impossível de ser atendida.

Percebe-se pela figura que quanto mais arestas o autômato possui (i.e. estruturas condicionais), fica mais complexo de se testar o módulo. Isso mostra a importância clara de que todo método deve ter no máximo 20 linhas, pois além de ser mais trivial de ser testado, mantém uma maior legibilidade.

3.3.3 Técnica da caixa cinza

Essa técnica tem por finalidade combinar as duas técnicas apresentadas anteriormente em uma só. A caixa cinza baseia-se em testar quais são as saídas para cada uma das entradas fornecidas, bem como rastrear quais foram os trechos que foram executados para gerar aquele resultado e avaliar se faz sentido, ou não. Utilizando o exemplo do fatorial, imaginando um cenário que o desenvolvedor fornece como parâmetro o valor 4 ele deveria obrigatoriamente retornar o valor 24. Mas, mais do que isso, a estrutura de controle *while* deveria ser executada 4 vezes e depois retornar o valor esperado.

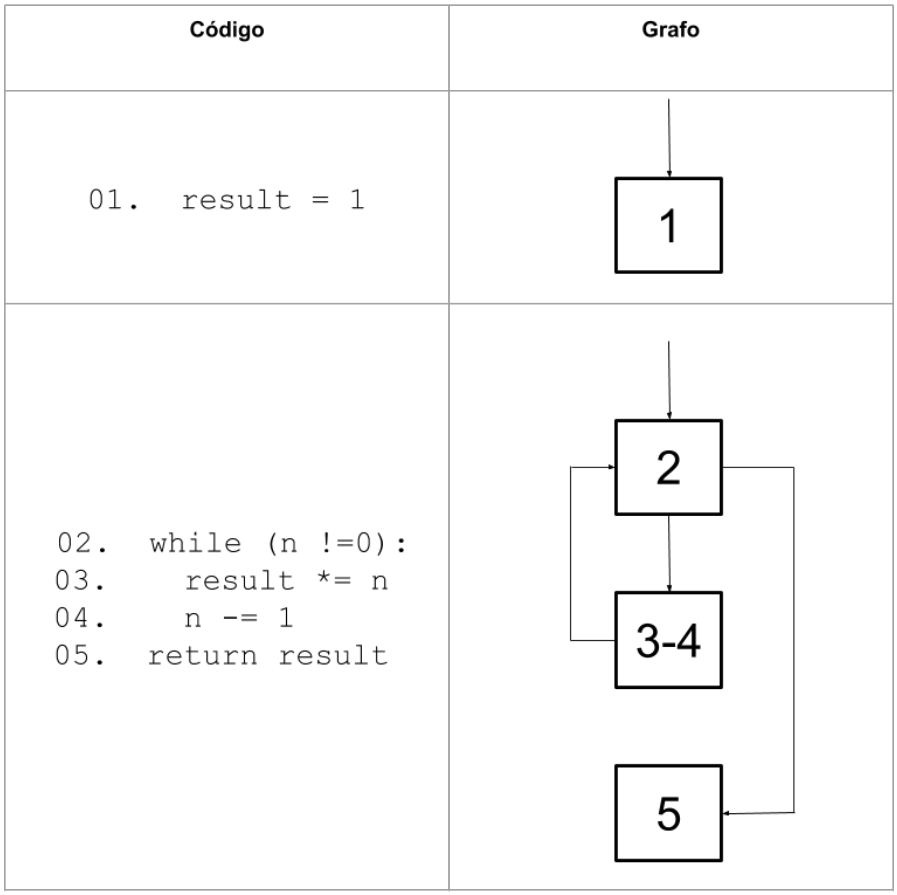

Figura 10 – Visão da técnica da caixa branca usando AFD.

(FONTE: Imagem feita pelo autor)

3.3.4 Conclusão

Considerando o conteúdo abordado nessa seção, é evidente que existem muitas formas de se testar um software. Nesse sentido, a principal contribuição dos conceitos aqui apresentados, além do teste de software em si, são os testes de aceitação. O framework desenvolvido ao longo desse trabalho é capaz de capturar as ações executadas pelo usuário e a partir delas criar um teste em nível de aceitação. Mais detalhes sobre a implementação do framework serão apresentados mais adiante.

4 Desenvolvimento Mobile

O desenvolvimento de aplicações móveis, é uma área que existe desde o início dos primeiros dispositivos móveis, mas que ganhou mais destaque nos últimos anos, como visto na Figura 11. Nas primeiras gerações de celulares com interface gráficas e com capacidade de processamento mais robusta, uma séries de aplicativos era disponibilizada ao usuário como: agenda, calendário, email, jogos e até mesmo acesso à internet. Porém, não era possível que um usuário leigo pudesse instalar aplicações de terceiros de maneira trivial. Geralmente, o processo envolvia baixar a aplicação desejada no computador e fazer instalação manualmente no dispositivo.

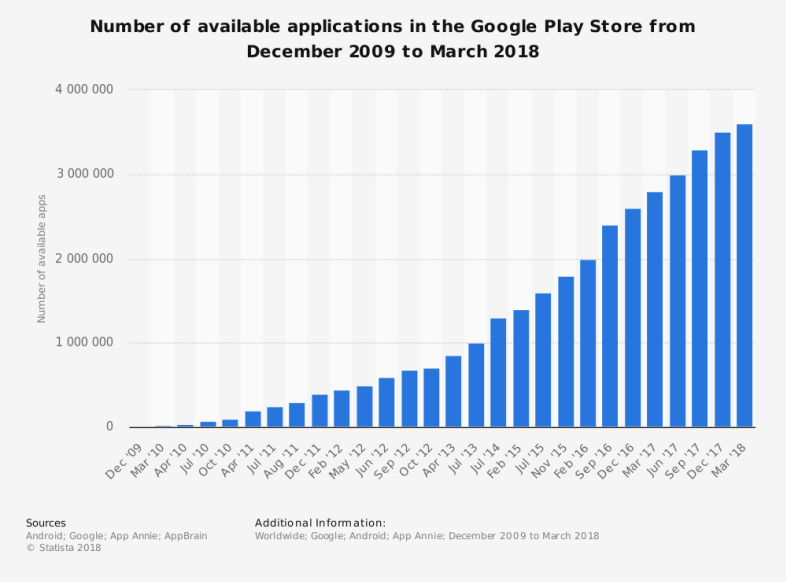

Figura 11 – Número de aplicações disponível na Google Play ao passar dos anos

(FONTE: Statista, 2018)

Com o passar dos anos, a indústria de dispositivos *mobile* passou por um processo muito semelhante ao que houve no mundo de computadores desktop, tanto no quesito hardware quanto software. Os componentes eletrônicos para essas plataformas foram ficando cada vez mais poderosos e baratos ao longo dos anos (HALPERN, 2016). Já no que se refere aos sistemas operacionais, o mercado, aos poucos, foi convergindo para um número menor de sistemas. Dessa forma, surgiram sistemas operacionais que hoje, praticamente, dominam totalmente o mercado de dispositivos móveis, como pode ser observado na Figura 12. São esses: Android, iOS e Microsoft Phone, produzidos respectivamente pela Google, Apple e Microsoft.

Com um mercado renovado, os desenvolvedores passaram a ter uma maior liberdade para poder desenvolver suas aplicações. Porém, o que fez com que esse mercado ficasse tão em voga como está nos dias de hoje foi o fato de que as empresas responsáveis pelos sistemas operacionais disponibilizassem aos seus usuários uma loja virtual de aplicativos. Isso fez com que os usuários não ficassem mais limitados única e exclusivamente às aplicações que vinham de fábrica. Outro fator que fez com que o mercado criasse essa grande necessidade por aplicações *mobile* foi a facilidade que muitas das empresas dão aos desenvolvedores em disponibilizar seus aplicativos nessas mesmas lojas, fosse de forma gratuita ou não.

Porém, semelhante aos softwares para as plataformas desktop, o desenvolvimento de aplicações *mobile* não é absolutamente perfeito. O maior problema para desenvolvedores que almejam ter suas aplicações disponíveis em todas as plataformas, é o da linguagem de programação. Enquanto o ambiente da Google suporta apenas Java e Kotlin, o iOS era uma plataforma que exigia que os programadores programassem em Objective-C, mas que agora usa como linguagem nativa o Swift. Por último, a Microsoft possui como linguagem padrão o Visual C++ e o C#. Dessa forma, por mais que uma aplicação móvel seja bem projetada, do ponto de vista de orientação a objetos, sempre será necessário replicar a aplicação para outra linguagem, seja através de um transpilador ou de trabalho puramente manual. Por tal razão, o trabalho focará exclusivamente na plataforma Android.

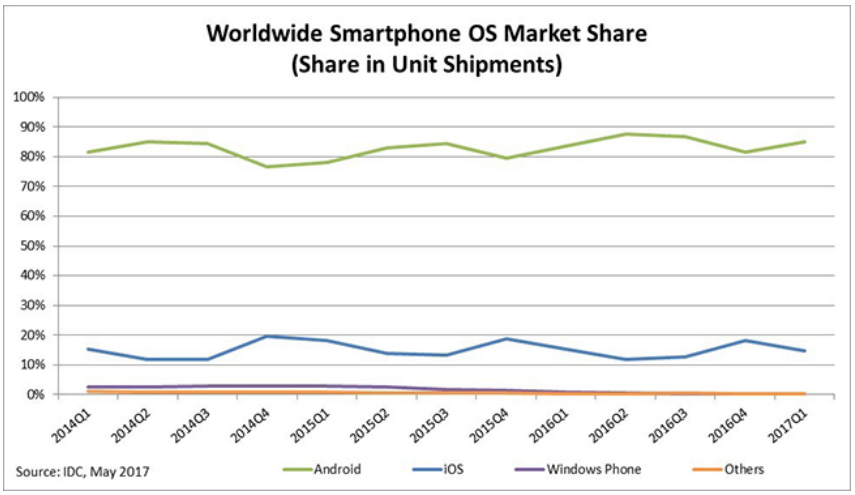

Figura 12 – Porcentagem de sistema operacional por dispositivo móvel

(FONTE: IDC, 2017)

Nesse sentido, nas próximas secções é apresentada uma breve descrição técnica do Android, seguido de uma seção que tratará do desenvolvimento de aplicações nessa mesma plataforma, trazendo os principais aspectos e conceitos

4.1 Especificações técnicas do Android

O Android foi uma plataforma que teve seu primeiro lançamento, oficial, em 2008. Na época, a divisão que desenvolveu o sistema já pertencia à Google. O objetivo inicial do projeto, era desenvolver um sistema operacional para celulares capaz de identificar a localização do usuário e suas principais preferências. No que se refere a termos de licença, o Android possui uma licença própria criada pela Google conhecida como AOSP (Android Open Source Project), ou seja uma licença de código aberto.

O Android, atualmente, é baseado em uma das versões do kernel do Linux, mais especificamente as versões: 3.18 e 4.4, dependendo do aparelho. Apesar de o Android ter o kernel baseado em versões LTS (Long Term Support), muitas modificações tiveram que ser feitas para atender aos requisitos da Google. Uma das principais mudanças foi que a Google teve que desenvolver uma biblioteca alternativa à GNU C, visto que os processadores que iriam executar o sistema eram CPU's com baixas frequências. A biblioteca ficou conhecida como *Bionic*.

As principais arquiteturas suportadas atualmente pelo Android são a ARM, x86 e MIPS. Como pode ser visto no gráfico seguinte, a arquitetura ARM é a dominante, visto que é uma ISA (Instruction Set Architecture) que preza pela eficiência energética. Já as arquiteturas x86 e MIPS foram arquiteturas que foram ganhar suporte somente mais tarde, visto que essas eram as arquiteturas utilizadas nos computadores dos desenvolvedores que emulavam as aplicações Android. Mais recentemente, as versões 64 bits dessas mesmas ISA estão, gradativamente, ganhando suporte da plataforma.

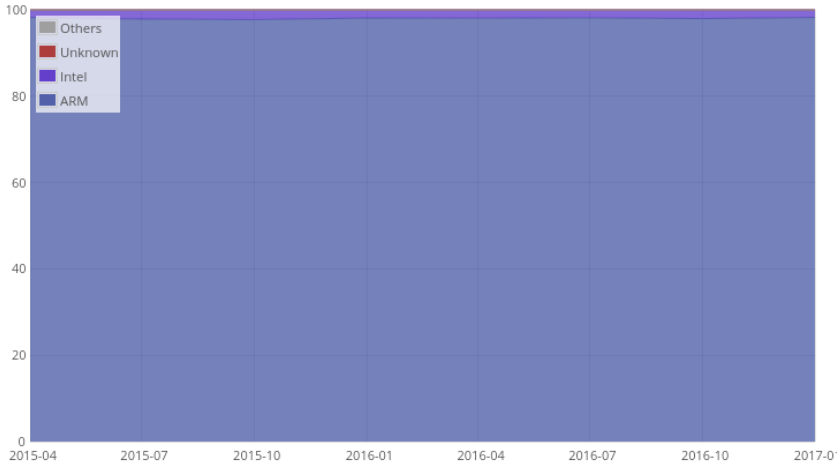

Figura 13 – Porcentagem de ISA's que atualmente estão executando Android.

(FONTE: Unity - Mobile, 2018)

No que se refere à execução de processos dentro do o sistema operacional, a plataforma utiliza o Dalvik, uma máquina virtual de processos, que recompila o Java bytecode

para rodar nativamente. Na versão 4.4, o Android introduziu o ambiente *Android Runtime* que usava a técnica de compilação AOT (*ahead-of-time*). A ideia consistia basicamente em recompilar o bytecode inteiro pra código de máquina nativo.

Sobre a gerência de memória, a plataforma utiliza um sistema de gerenciamento bem simples. Quando o usuário interrompe o uso de um aplicativo, o SO apenas suspende a aplicação para que pare de consumir recursos da CPU, mas continua mantendo os dados na memória. Porém, caso a memória comece a ficar escassa, então o sistema começa a matar os processos que estão há mais tempo sem serem utilizaods.

Já quanto às aplicações que são desenvolvidas para o Android, estas utilizam como suporte o Android SDK (*Software Development Kit*). O SDK possui ferramentas como: debugger, bibliotecas (API's), emulador baseado no QEMU, documentação. O SDK possui suporte nativo à linguagem Java que pode ser combinado com trechos de código escritos em C/C++ e, mais recentemente, a Google anunciou que a linguagem Kotlin também passou a ser suportada pelo SDK. Outra *feature* interessante do SDK é que ele permite que os desenvolvedores testem seus aplicativos em versões mais antigas do Android, a fim de que possam analisar o desempenho de suas aplicações em plataformas mais antigas.

4.2 Desenvolvimento de aplicações mobile Android

O desenvolvimento de aplicações *mobile* para a plataforma Android usualmente se dá através da ferramenta oficial da Google, o Android Studio. Para iniciar um novo projeto, o desenvolvedor deve escolher, entre outros aspectos, qual o alvo de dispositivos que a aplicação visa (i.e. tablets, celulares, TV's ou *SmartWatches*), se a aplicação terá suporte à C++ e qual a versão da API do Android será utilizada no projeto. Essa é uma decisão muito importante para o projeto, visto que caso o desenvolvedor escolha uma versão muito recente, poucos dispositivos poderão ter acesso ao aplicativo que está sendo desenvolvido. Por outro lado, caso o programador opte por escolher por uma versão muito antiga, pensando em atender 100% do mercado, ele fica sem muitas facilidades e atualizações que ele teria em versões mais atuais. Para facilitar o processo, o próprio Android Studio fornece uma porcentagem, aproximada, de quantos dispositivos serão contemplados ao escolher uma dada versão da API.

Na sequência, o programador pode escolher com qual modelo de tela inicial o usuário será recebido, ou simplesmente nenhuma, podendo alterar a decisão mais tarde. Após feita essa configuração inicial, o projeto está criado. Ao finalizar a criação de um projeto novo, o próprio Android Studio já irá criar, automaticamente, uma estrutura de diretórios padrão, sendo os principais diretórios: *java* e o *res*. Na pasta *java*, fica todo o código fonte da aplicação que será executado. Já dentro do diretório *res* é onde ficam os arquivos *XML* de configuração de interface e outros arquivos importantes como o de

internacionalização de strings.

A Google adotou como padrão que sempre que o desenvolvedor quiser criar uma nova interface para aplicação, ele deve criar o que é chamado de **activity**. Uma *activity*, nada mais é do que um template de tela como foi mostrado para o usuário quando ele inicou o projeto. Ao escolher uma atividade, o Android Studio irá gerar o código XML e Java padrão para o template escolhido. O arquivo XML serve exclusivamente para definir como cada componente será disposto na tela, incluindo propriedades como textos, cor, layout. Sempre que o usuário quiser desenvolver a interface da atividade ele pode alterar diretamente o código fonte do arquivo XML ou ele pode utilizar um módulo *built-in* da própria IDE que permite a edição de interface, graficamente, como pode ser visto na Figura 14.

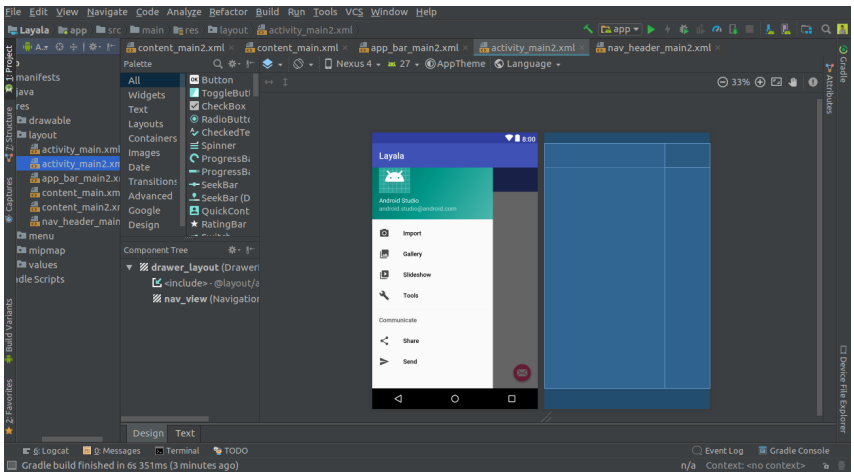

Figura 14 – Edição de uma atividade do Android.

(FONTE: Imagem feita pelo autor)

Já o arquivo Java da atividade criada serve como uma controladora para a interface. Esse arquivo Java conterá uma classe que herda, indiretamente, da classe *Activity*. O objetivo dessa classe é unicamente ser a controladora da atividade que está sendo desenvolvida, ou seja, atualizar a interface e invocar métodos de outros objetos que tratem do modelo. Cada *activity* tem sua própria controladora e não deve gerenciar mais nenhuma outra interface, além da que foi designada. Outro ponto importante, é que alguns métodos herdados da classe *Activity* tratam de eventos importantes como: atividade inicializada, resumida, suspensa, etc. Esses métodos são importantes, pois, por exemplo, caso o usuário venha a suspender o aplicativo que esteja executando para utilizar outro, o desenvolvedor pode utilizar o método *onPause* para salvar dados importantes que o usuário tenha inserido até aquele momento.

A aplicação desenvolvida também pode se comunicar com algum servidor ou ser

uma aplicação que funciona apenas com dados locais. No primeiro caso, a aplicação funciona como um sistema web comum que segue o modelo de implementação cliente/servidor. Sempre que for necessário utilizar dados do servidor para executar alguma ação, a aplicação fará uma requisição para o servidor através do protocolo HTTP para poder ter acesso às endpoints das APIS necessárias. Esse processo requer uma troca de dados estruturados. Ultimamente, muitas das API's que vem sendo implementadas têm utilizado apenas dados em formato JSON, mas já houve épocas em que o formato padrão era XML.

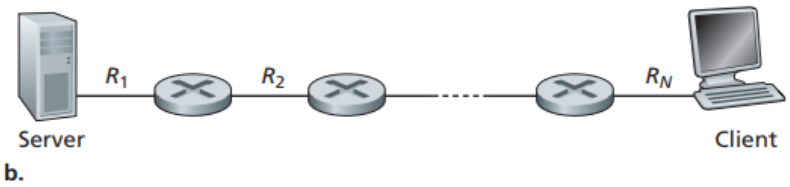

Figura 15 – Modelo Cliente servidor

(FONTE: KUROSE, J., KEITH, R. - Mobile, 2013)

Já no que se refere a dados locais, o Android utiliza, como ferramenta nativa, o banco de dados relacional *SQLite*. Esse banco foi escolhido, visto que o propósito dele é unicamente ser um banco para sistemas embarcados e que utiliza apenas um único nodo computacional para fazer o acesso e atualização dos dados.

Outro aspecto importante no desenvolvimento das aplicações para Android, é que o desenvolvedor pode testar o software sem precisar baixar em um aparelho físico. O Android Studio possui uma ferramenta nativa chamada Android Virtual Device (AVD), que permite emular os mais diversos aparelhos disponíveis no mercado, independente da marca ou segmento (i.e. tablets, celulares, TV's, *Smartwatches*). Com essa plataforma, o desenvolvedor pode verificar como o seu software se comporta nos mais variados tamanhos de telas, bem como diferente tipos de configurações de hardware. Isso permite ao desenvolvedor garantir uma boa responsividade sem precisar pagar por novos aparelhos. Mesmo assim, é importante que quem esteja testando a aplicação faça os testes finais em dispositivos físicos para ver se a aplicação está se comportando da mesma forma como nos testes em ambiente de emulação, uma vez que os emuladores também estão abertos a falhas.

4.3 Ciclo de vida de uma aplicação Android

Uma aplicação Android, passa por diversos estados, desde sua execução até sua finalização. Os estados e as respsectivas transições de uma aplicação podem ser vistos na Figura 16. Os métodos representados no diagrama, como estados, são métodos im-

plementados em toda classe do tipo *Activity*. Nesse métodos, o Android utiliza váriaveis internas para poder controlar o estado e recursos que a aplicação está utilizando. Entretanto, quando um desenvolvedor está criando sua aplicação ele pode sobrescrever esses métodos para adaptar para necessidades específicas.

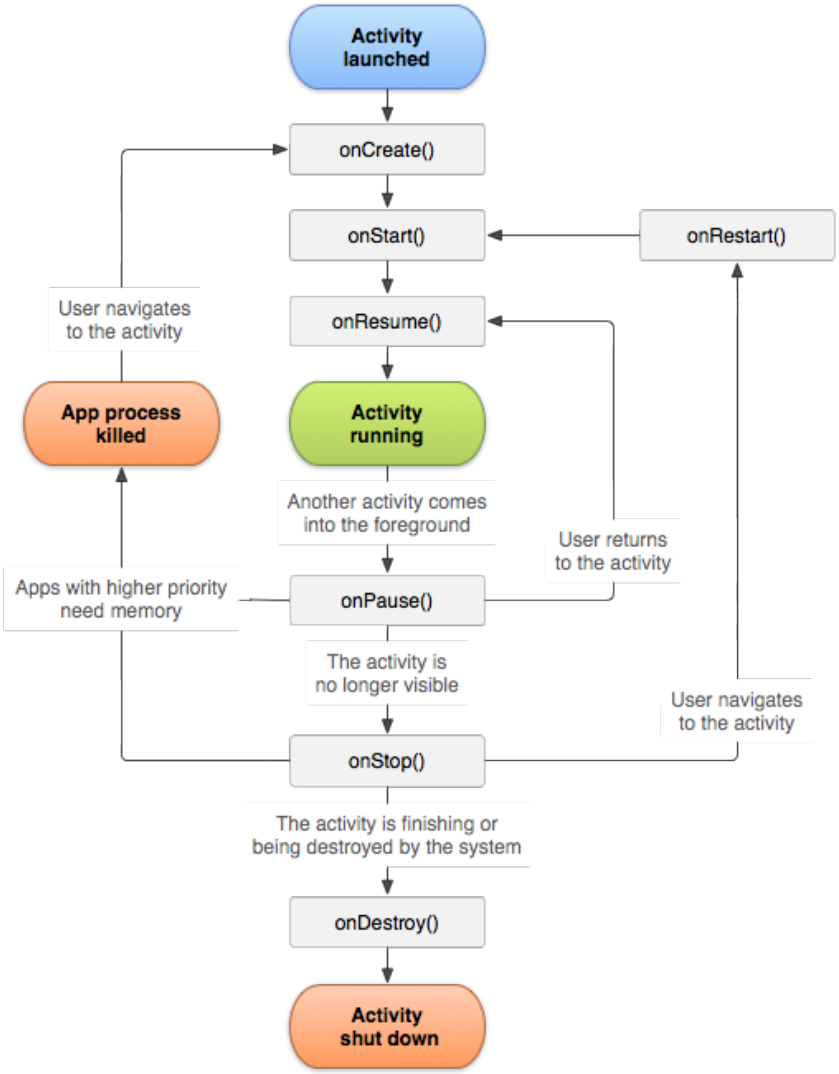

Figura 16 – Ciclo de vida de uma aplicação

(FONTE: Android, 2019)

Propõe-se o seguinte cenário: um usuário está utilizando uma determinada aplicação e está preenchendo um formulário. Se o usuário trocar de aplicação enquanto o formulário é preenchido, os dados, se não forem salvos pelo usuário, serão perdidos na troca de contexto entre as aplicações. Nesse sentido, caso o desenvolvedor queira garantir que o usuário não perca seus dados, basta que sobrescreva o método *onPause()* e persistir os dados em algum local, seja no aparelho ou em um servidor web.

Sendo assim, quando o usuário retornar a aplicação, o método *onResume()* será evocado pelo Android, e o desenvolvedor pode então preencher os campos com os dados que o usuário já havia informado previamente. Esse tipo de implementação garante uma melhor experiência para o usuário e permite ao desenvolvedor entender qual estado que sua aplicação se encontra em cada momento.

É importante ressaltar que quem faz a execução e chamda desses métodos é o próprio Sistema Operacional e nunca o programador. Como explicado na secção anterior, caso o aplicativo fique muito tempo sem ser utilizado pelo usuário, ou seja permaneça por muito tempo no estado *onStop()* e a memória comece a ficar escassa, o Android irá então finalizar a aplicação para liberar recursos para outros processos que estão necessitando. Por isso, qualquer dado que o desenvolvedor julgue como importante, deve ser salvo, na pior das hipóteses, quando este método for evocado.

4.4 Espresso

O *espresso* é um framework oficial da API do Android que permite fazer testes em nível de usuário, ou seja executar cliques na tela, preencher formulários, encerrar aplicação. Além disso, o *espresso* fornece funções de asserção e uma sintaxe específica para verificar estados de objetos na tela baseado em seus identificadores globais.

O *espresso* além de suportar, nativamente, eventos assíncronos, ele é uma ferramenta voltada para desenvolvedores que querem testar sua aplicação utilizando tanto a técnica de caixa preta quanto a técnica de caixa branca, como pode ser observado na documentação oficial do Android.

Nesse sentido, o *espresso* permite que o desenvolvedor crie testes que implementam ações de usuário que normalmente seriam feitas de forma manual. Dessa forma, as ações de um usuário são automatizadas e quando o teste é executado, a aplicação vai sendo controlada por essas ações previamente programadas.

Essa ferramenta foi utilizada internamente, como uma dependência, para implementar o framework desse trabalho.

4.5 Conclusão

Assim como qualquer outro software, as aplicações Android precisam ser testadas durante o processo de desenvolvimento. O Android SDK oferece alguns recursos nativos para testar aplicações como: framework para testes unitários, *debugger*, funções de asserção. Dessa forma, é importante compreender como as aplicações Android são desenvolvidas e quais ferramentas para teste estão disponíveis para que, então, um framework capaz de gerar testes nativos para aplicação seja possível de ser implementado.

Parte II

Trabalhos relacionados

5 Trabalhos relacionados

Nesse capítulo são apresentados trabalhos relacionados que envolvem tanto o desenvolvimento de frameworks, como ferramentas que ajudem na elaboração automática de testes automatizados. Os trabalhos apresentados ajudaram, além de justificar a existência desse trabalho, a reusar ideias já existentes e aplicá-las em um contexto específico, no caso aplicações para a plataforma Android. Sendo assim, são apresentados artigos que, além de consolidar os conceitos já apresentados, permitam reaproveitar ideais passadas em um projeto atual.

5.1 A busca de generalidade, flexibilidade e extensibilidade no processo de desenvolvimento de frameworks orientados a objetos

Nesse artigo de Ricardo Pereira e Silva e Roberto Tom Price (1998) é feita uma abordagem, resumida, dos principais conceitos envolvendo frameworks, a maioria deles já explicados no capítulo de Frameworks desse trabalho. Porém, um ponto importante discutido dentro desse artigo são os tópicos de padrões de projeto e metapadrões relacionados ao desenvolvimento de frameworks. Os padrões de projetos e metapadrões são contribuições importantíssimas , no que se refere ao reuso de software. O trabalho comenta de um catálogo com 23 padrões a serem seguidos pelos mais variados tipos de projetos e qual modelagem os projetos devem seguir. Já os metapadrões são mais independentes de domínio e visam a flexibilidade. Os templates são o principal exemplo disso dentro do artigo. Uma vez que os padrões de projetos são voltados a domínios específicos eles acabam sendo usados para o desenvolvimento de frameworks, porém os metapadrões não são completamente descartados, podendo ser utilizados quando for necessário dar alguma flexibilidade no algoritmo.

O estudo comenta o caminho que o desenvolvedor deve seguir para desenvolver um framework, o qual seria: generalização, flexibilização, aplicação de metapadrões, aplicações de padrões de projeto e aplicações de boas práticas dentro da orientação a objetos. Na etapa de generalização, busca-se estudar as aplicações implementadas do domínio que será abordado e observar suas características compartilhadas, visando principalmente elementos comuns de domínio. A etapa de flexibilização seria identificar as pequenas variações de uma aplicação para outra. No caso do framework do artigo, foi generalizado o *controller* responsável por processar a ação do usuário. Nos metapadrões, todos os jogos precisam ter tabuleiro e demais peças instanciadas. Para isso usou-se templates e hooks. Já para os padrões de projeto, utilizou-se os padrões observador e observável e *abstract* e *factory*.

Também se utilizou o padrão *decorator* no qual uma classe é responsável por realizar ações sobre o objeto que está sendo "decorado". Sobre as práticas de OO é comentado que a herança deve ser usada principalmente quando busca-se generalidade e concretizar classes abstratas.

A principal contribuição do artigo para o trabalho são os conceitos de framework em si

5.2 An Object-Oriented Framework for Improving Software Reuse on Automated Testing of Mobile Phones

Frameworks capazes de auxiliar os desenvolvedores na elaboração de testes automatizados para aplicações móveis é um conceito que já foi aplicado no passado, como pode ser observado nesse artigo. Esse trabalho (KAWAKAMI, L., KNABBEN A. Et al., 2007) possui como principal foco o desenvolvimento de testes automatizados baseado em casos de uso, usando o reuso como principal forma de alcançar esse objetivo. No framework desenvolvido, ainda era necessário que o programador implementasse manualmente os testes. Porém, a grande vantagem do uso desse framework era que ele possuia grandes abstrações de hardware e se comunicava com o baixo nível através da API disponibilizada pela empresa que fabricava o telefone, no caso a Motorola, uma espécie de*syscall*. A grande vantagem, era que a portabilidade entre dispositivos físicos podia ser feito sem necessidade de alterar o teste.

Dos resultados obtidos do artigo, pode-se verificar que 10 modelos de celulares foram utilizados no estudo. À medida que os testes foram sendo portados para cada plataforma, o reuso aumentava consideravelmente, alcançando uma média de 84% de reaproveitamento de testes. Fica nítido, através do estudo, também que o custo para automatizar testes é grande, visto que é o preço que se paga pelo reuso. Entretanto, ao portar testes de um modelo de celular para outro o esforço necessário é 1/4 do necessário se comparado ao esforço para escrever os testes, levando 1/3 do tempo. O artigo ainda aborda um estudo ao se utilizar 60 casos de teste passíveis de serem automatizados numa família de 15 modelos de celulares. Neste cenário, iniciou-se automatizando os testes mais simples. Enquanto certo teste não ficava pronto, ele era executado totalmente de forma manual. Além disso, baseado em quanto tempo levou-se para executar determinado teste manualmente, esse valor foi multiplicado pelo número de vezes que ele seria executado ao longo do projeto, a fim de melhor ajustar a estimativa. Neste estudo, o ganho de produtividade, no período de um ano, foi de três vezes, sendo que no terceiro mês de projeto, a automatização de testes já começou a se pagar.

Nesse sentido, a principal contribuição do artigo para esse trabalho é, além de validar a importância do desenvolvimento de uma ferramenta similar para os dipositivos

atuais, é utilizar o conceito de caso de uso para gerar testes automaticamente. A atual API do Android já é muito consolidada e fornece abstrações suficientes para que o programador não precise reescrever várias vezes a mesma aplicação para cada modelo de celular.

5.3 DroidMate: A Robust and Extensible Test Generator for Android

Nesse artigo (JAMROZIK, K., ZELLER, A., 2016) é apresentado o DroidMate: uma ferramenta capaz de gerar testes baseado na interface de usuário combinado com testes exploratórios. O artigo inicia comentando a principal dificuldade de se escrever testes, que é manter a base de testes sincronizada com a versão mais atual da aplicação. Isso acontece, visto que qualquer refatoração ou implementação de nova feature no sistema, ainda que seja mínima, necessita de uma revisão na base de teste. Dado esse cenário, é apresentada uma ferramenta capaz de monitorar o bytecode executado dentro da aplicação durante a utilização por um usuário. O framework implementado se propõe a criar um log utilizando como base as ações do usuário como cliques e texto digitado. Esse monitoramento é guardado em um arquivo do próprio framework. Após uma condição ser atendida, geralmente um *timeout*, toda a ação monitorada é transformada em um teste para a aplicação.

Para implementar esse teste gerado pela ferramenta, ele reusa o UIAutomator, uma ferramenta oficial do próprio Android voltada para o uso de testes de interface de usuário. O framework é capaz de gerar esse teste, já que durante o monitoramento de bytecode ele consegue ter acesso a *stacktrace* de execução, chamada de funções e valores de parâmetros. Nesse sentido, ele combina os valores monitorados junto com métodos disponibilizados no UIAutomator para criar os testes para o desenvolvedor.

Sendo assim, a contribuição desse artigo para o esse trabalho se dá no fato de usar a interação do usuário com a aplicação, monitorada programaticamente, para gerar os testes de forma automática. Porém, diferente do DroidMate, o framework que foi implementado buscar monitorar chamadas em mais alto nível, sem precisar fazer a verificação de cógido a nível de bytecode.

Parte III

O framework

6 Análise e desenvolvimento do framework

6.1 Visão geral do framework

O *capuccino*, nome do framework desenvolvido durante esse trabalho, é uma ferramenta capaz de gerar testes de interface, ou seja em nível de usuário, baseado em execução da aplicação. O framework utiliza a ideia de capturar ações do usuário enquanto ele utiliza a aplicação para construir um caso de teste. À medida que o usuário utiliza o aplicativo, em um dispositivo emulado, as suas ações de clique e preenchimento de formulários vão sendo capturadas. Ao longo desse processo, o *capuccino* vai mantendo esses eventos em uma lista ordenada por horário em que o evento ocorreu. Quando o usuário terminar de executar todas as suas ações, ele deve encerrar a aplicação para indicar ao framework que nenhuma interação será mais executada e que o teste já pode ser gerado.

Após o framework ser notificado de que a aplicação foi finalizada, ele irá gerar o arquivo de teste com todas ações do usuário de forma programática. Para criar esse teste, o *capuccino* utiliza algumas informações que foram previamente fornecidas por quem está executando as ações como: qual deve ser a asserção executada ao final do teste, nome do arquivo do teste, caminho do diretório raiz da aplicação. Com essas informações, o teste é criado, utilizando bibliotecas oficiais do Android que simulam ações do usuário. Uma vez escrito, o arquivo gerado pelo framework pode, então, ser exportado para o pacote de testes de aplicação e ser executado normalmente.

Nesse sentido, o caso de teste gerado é responsável por inicializar a aplicação e implementar as ações do usuário através de uma biblioteca que permite simular operações de usuário como: clique em uma posição da tela, tecla digitada e movimento de rolagem. Ao final dessas ações, é executada, então, a asserção definida previamente pelo usuário.

É importante deixar claro, que em nenhum momento o framework assume qual deve ser a asserção executada ao final do teste. Isso se deve ao fato de que esse tipo de informação é totalmente dependente do domínio do problema, ou seja, a aplicação que está sendo testada.

Outro ponto a ser ressaltado é que para definir a asserção que será executada ao final do teste, a pessoa que está a gerar o caso de teste deverá seguir uma sintaxe específica do framework *espresso*. Dessa forma, antes de iniciar a produção do teste, o usuário deverá definir através de uma classe fornecida pelo próprio framework, qual a asserção será executada ao final do teste, usando a sintaxe do framework indicada. Caso a asserção não seja definida, o caso de teste criado pela ferramenta ficará incompleto.

6.2 Concepção

Considerando os trabalhos apresentados no capítulo anterior, fica evidente que uma implementação capaz de gerar testes automaticamente para dispositivos *mobile* é perfeitamente possível. O framework desenvolvido nesse trabalho foi concebido sob a óptica de capturar as ações do usuário e gerar um teste no nível de interface de usuário. Porém, diferente do DroidMate, ao invés de configurar um *timeout*, é requisitado ao usuário configurar uma asserção para ser executada no final do teste, para verificar se houve sucesso ou não durante a ação do usuário.

Além disso, o framework foi idealizado com a ideia de que o desenvolvedor não precise reescrever sua aplicação para se adequar à ferramenta. Nesse sentido, o framework foi desenvolvido com a ideia de que o desenvolvedor necessite apenas herdar classes do framework implementado e escrever algumas informações no arquivo de configuração. É importante deixar claro, que mesmo herdando as classes do framework, a aplicação desenvolvida não terá seu comportamento alterado de nenhuma maneira.

Atualmente, a API do Android possui diversas chamadas de método que permitem detectar ações do usuário com a aplicação. Dentre todos os métodos disponíveis, destacam-se aqui os dois principais que permitiram a implementação do framework:

- *public boolean dispatchTouchEvent(MotionEvent me)*
- *public boolean dispatchKeyEvent(KeyEvent kEvent)*

Como os nomes já sugerem, eles permitem detectar ações de clique do usuário e texto digitado, respectivamente. Esses métodos são evocados pelo Android toda vez que um evento de clique ou de digitação ocorre. Para o caso em que um clique é detectado, o objeto passado como parâmetro, *MotionEvent*, permite coletar informações como posição em que o clique ocorreu (i.e. X e Y). Infelizmente essa classe não traz informações para detectar se o clique foi do tipo *click and hold*, mas com uma lógica simples para verificar o tempo que levou para o clique iniciar e finalizar, é perfeitamente possível coletar esse tipo de informação. Uma lógica similar a essa é também utilizada para verificar se o clique foi do tipo *scroll*. Para esse tipo de detecção, além de calcular o tempo de clique, é comparado a posição inicial com a posição final do clique.

Para os eventos de digitação do usuário, o objeto passado como parâmetro no método, permite coletar qual foi a tecla digitada pelo usuário. Assim como o evento de clique, esse método é invocado pelo sistema operacional toda vez que um evento de digitação ocorre. É importante salientar que esses métodos são invocados duas vezes por evento: uma vez quando o evento é iniciado e outra quando o evento é finalizado. Dessa forma, a deteção de ações como *scroll* e *click and hold* é perfeitamente implementável do jeito como foi explicado anteriormente.

Após o usuário finalizar as suas ações, o usuário deverá encerrar a aplicação para indicar que suas ações terminaram. Após isso, o framework gera o teste com todos os atos feitos pelo usuário junto com a asserção definida previamente. Assim como o DroidMate, o framework utiliza a biblioteca UIAutomator para poder simular as ações do usuário. Escolheu-se essa biblioteca, pois além de ser a oficial da plataforma, ela é capaz de gerar cliques em posições X e Y. Para a função de asserção, o usuário deverá seguir as funções disponibilizadas no framework *espresso*. Dessa forma, por utilizar esse framework internamente, a ferramenta desenvolvida foi batizada de *capuccino*.

Nesse sentido, o framework concebido acaba funcionando como uma espécie de proxy, em que as ações capturadas do usuário são guardadas em uma estrutura de dados da própria biblioteca, e o evento é então repassado para aplicação, sem alterar nenhum valor ou comportamento original. Essas chamadas de método da API do Android estão disponíveis em toda *Activity* e, como explicado no capítulo de desenvolvimento *mobile*, todo o desenvolvimento de uma aplicação Android se baseia em *Activities*. Por isso, a principal função do *capuccino* é reescrever esse métodos e solicitar ao desenvolvedor que suas *Activities* herdem daquela sobrescrita pelo framework.

6.3 Modelagem e Implementação

Na Figura 17 é possível visualizar a modelagem UML do *capuccino*. Como pode ser visto no diagrama, o *capuccino* possui várias estruturas de dados (i.e. classes) para poder armazernar informações durante o processo de captura de ações do usuário. Para entender como o framework funciona, existem duas entidades-chave para compreender seu correto funcionamento. São elas:

- *CapuccinoBasicActivity*
- *LauncherAppActivity*

A classe *LauncherAppActivity* é utilizada para iniciar a aplicação do usuário. Além disso ela também serve para criar e configurar as informações que serão utilizadas para criar o teste automaticamente. Essa classe deve ser herdada pelo desenvolvedor da aplicação e no arquivo *AndroidManifest* de seu projeto deve configurar para que essa *Activity* seja o ponto de entrada da aplicação. Além de implementar essa herança, o usuário deve sobrescrever o método *onCreate* usando os métodos *setters* disponíveis a fim de informar os dados para poder gerar o teste automaticamente de forma correta. Explicando brevemente cada um desses dados e respectiva importância:

- *packagePath*: indica o *package* da aplicação do usuário.

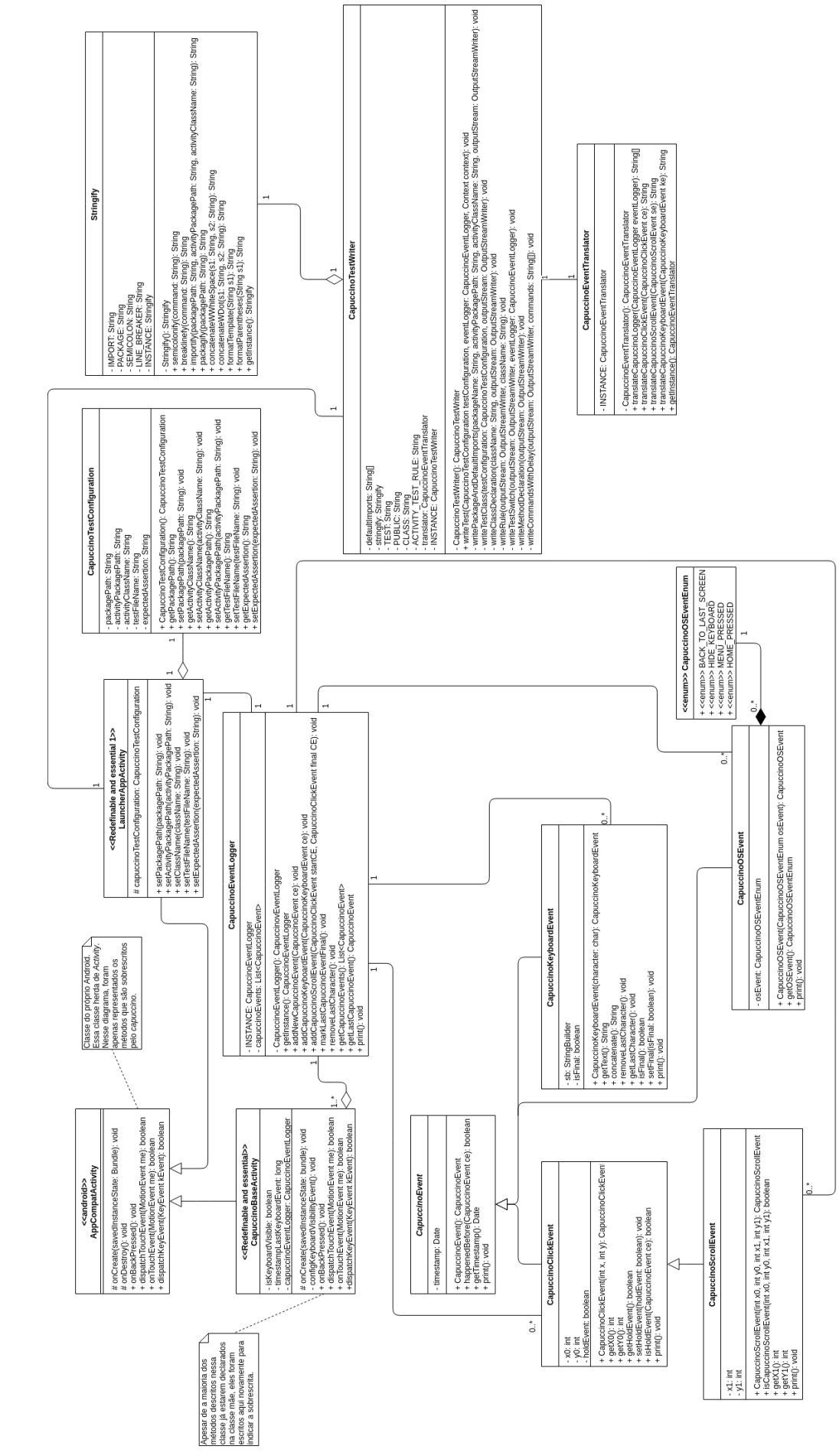

Figura 17 – Diagrama de classes UML do framework Capuccino

(FONTE: Android, 2019)

- *activityPackagePath*: indica o *package* onde a atividade a ser testada se encontra.
- *activityClassName*: nome da classe da atividade que será testada.
- *testFileName*: nome do arquivo de teste que o desenvolvedor deseja que seja gerado.
- *expectedAssertion*: asserção que o usuário quer que seja executada ao final do do teste.

Já a classe *CapuccinoBasicActiviy* deve ser herdada por todas as atividades do usuário, ou pelo menos aquelas que serão testadas. Como dito anteriormente, o *capuccino* funciona como uma proxy em que ele capta as ações do usuário e repassa para aplicação original. Quando o usuário herda suas atividades dessa classe é exatamente isso o que acontece. A *CapuccinoBasicActiviy* está associada ao *CapuccinoEventLogger* que é responsável por manter os eventos ordenados por horário em que ocorreu, em uma lista.

A classe *CapuccinoEvent* é responsável por abstrair qualquer ação do usuário. Ela é uma classe, nativamente, abstrata. A partir dela, são herdadas classes concretas que servem para indicar qual foi o tipo de ação que o usuário executou. Todos esses eventos possuem como atributo o horário em que ocorreram, pois dessa forma é possível deixar os mesmos ordenados na lista de eventos do *CapuccinoEventLogger*. É importante também destacar aqui, que na classe *CapuccinoOSEvent*, que serve para indicar ações do usuário que envolvam chamadas do Sistema Operacional, como voltar para a tela inicial do *smartphone* ou bloquear a tela, somente algumas dessas ações são abstraídas.

Assim que o usuário finalizar suas ações e chegar na condição que desejava para criar o teste, ele deve encerrar sua aplicação. A partir disso, o *LauncherAppActivity*, imediatamente, invocará *onDestroy* que está configurado para chamar a classe que escreve os testes antes de eliminar a aplicação. A classe *CapuccinoTestWriter* irá escrever o arquivo de teste usando como auxiliar a classe *CapuccinoEventTranslator* que pega as estruturas de dados utilizadas pelo *capuccino* e traduz para os comandos equivalentes do *UIAutomator*. É interessante ressaltar que o arquivo escrito ficará salvo no sistema de arquivos do smartphone emulado e não do computador do usuário. Nesse sentido, ao terminar de criar o teste ele deve abrir o sistema de arquivos do dispositivo emulado e deve ir até o diretório onde estão os arquivos da aplicação.

Ao final desse processo, o usuário terá criado um único caso de teste descrito em um arquivo com a asserção que ele definiu previamente antes de iniciar a aplicação. Esse teste, ao ser exportado para o diretório de testes de aplicação, pode ser executado normalmente. Durante a execução do teste criado, o teste irá iniciar a aplicação no dispositivo previamente emulado e irá executar as mesmas ações que o usuário executou, na mesma ordem de forma automática. Após executar a última ação, a asserção definida pelo usuário será executada e seu retorno será exibida no *Logger* do Android Studio.

Para criar mais casos testes, basta o usuário redefinir os dados necessários na classe de configuração de teste, reiniciar a aplicação e ir executando um caso de uso normalmente e encerrá-la, quando não houver mais ações para executar. Novamente, o framework realiza os mesmos passos em que escreve as ações do usuário em um arquivo de teste no sistema de arquivos do dispositivo emulado.

6.4 Uso do framework

Para melhor entendimento, do ponto de vista de um usuário, o intuito dessa seção é apresentar como um desenvolvedor pode integrar sua aplicação ao *capuccino*. Inicialmente, o projeto com código do framework deve ser copiado para o diretório raiz da aplicação que está sendo desenvolvida. Isso pode ser feito através de uma clonagem do repositório do projeto que está disponível no *Github*. Feito esse processo, o usuário deve então instalar as dependências do *capuccino* através do *gradle*, gerenciador de pacotes oficial do Android Studio.

Depois de instalar os pacotes necessários para o correto funcionamento do framework, o desenvolvedor deve indicar no arquivo *settings.gradle* que o diretório do *capuccino* deve ser incluído durante a geração do binário da aplicação. Dessa forma, quando a aplicação estiver sendo compilada, não haverá erros de *import*, nem de dependendências não encontradas.

Feito essa configuração inicial, o usuário da ferramenta deve, então, fazer com que cada uma de suas classes que herdam da classe *AppCompatActivity* passem a herdar da classe *CapuccinoBasicActiviy*. Como visto anteriormente no diagrama do framework, essa classe é a base para que a captura de ações do usuário seja possível. Nesse sentido, é necessário que as classes de *activities* do projeto devam herdar dessa classe, ou pelo menos aquelas que o usuário quer que casos de teste sejam gerados.

Após essa modificação nas classes, deve ser criada uma nova classe, também para controlar uma *Activity*. Porém, essa nova classe não deverá herdar de *CapuccinoBasicActiviy*, e sim de *LauncherAppActivity*. Essa classe, será responsável por inicializar a aplicação do usuário e onde será configurada algumas variáveis como: qual deve ser a asserção executada ao final do teste, nome do arquivo do teste, caminho do diretório raiz da aplicação. Um exemplo dessa classe pode ser visto na Figura 18:

É possível perceber que além dessa classe herdar de *LauncherAppActivity*, no método *onCreate*, uma série de variáveis são configuradas, através de métodos *setters*. A semântica de cada uma dessas variáveis já foi explicada anteriormente. É importante deixar claro que caso alguma dessas variáveis não seja configurada no método *onCreate*, uma exceção será lançada indicando que alguma variável não foi corretamente configurada. Ressalta-se que a variável *expectedAssertion* é a responsável por definir a execução que

```java
package com.example.olegario.escamboapp.activity;

import android.os.Bundle;

import com.getmore.olegario.capuccino.activity.LauncherAppActivity;

public class TestActivity extends LauncherAppActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        final String packagePath = "com.example.olegario.escamboapp";
        final String activityPackagePath = "activity";
        final String className = "HomeWithoutAuthentication";
        final String testFileName = "CapuccinoTest";
        final String expectedAssertion = "Espresso.onView(ViewMatchers.withId(R.id.passwordLoginEditText)).check(ViewAssertions
        this.setPackagePath(packagePath);
        this.setActiviyPackagePath(activityPackagePath);
        this.setClassName(className);
        this.setTestFileName(testFileName);
        this.setExpectedAssertion(expectedAssertion);
        super.onCreate(savedInstanceState);
    }
}
```

Figura 18 – Exemplo da classe responsável por incializar e configurar o projeto do usuário.

(FONTE: Imagem feita pelo autor)

será executada ao final do teste. Por isso, toda vez que o teste for gerado o usuário deverá alterar, a medida do necessário, essa asserção para cada caso de teste.

Após implementar essa classe, o usuário deve então configurar o projeto para que essa nova classe desenvolvida seja o ponto de entrada do aplicativo. Para fazer essa configuração, basta atualizar o arquivo *AndroidManifest.xml*. O trecho do código que faz essa configuração pode ser visualizado na Figura 19.

```xml
<activity
    android:name=".activity.TestActivity"
    android:label="@string/app_name"
    android:theme="@style/AppTheme.NoActionBar">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Figura 19 – Código para configurar novo ponto de entrada da aplicação.

(FONTE: Imagem feita pelo autor)

Feita essa configuração, basta o usuário, então, inicializar a aplicação normalmente em um dispositivo emulado. A aplicação será iniciada e quem estiver criando o teste

não verá nenhum tipo de alteração no comportamento original da aplicação. A partir desse momento, o usuário pode executar os casos de uso da aplicação, a fim de gerar os testes. Durante a execução dos casos de uso, as ações executadas pelo usuário vão sendo capturadas internamente pelo framework. Nesse sentido quando usuário terminar de executar algum caso de uso, ele precisa encerrar a aplicação para indicar ao *capuccino* que o caso de teste já pode ser gerado.

O framework será notificado pela própria API do Android através da sequência de métodos que o Android utiliza quando uma aplicação é encerrada. Mais especificamente, o método que notifica que a aplicação foi encerrada é o *onDestroy*. A partir disso a classe que escreve o teste será chamada e irá escrever o teste. Essa chamada em cadeia de métodos, também poderia ser implementada através de um *timeout*, mas optou-se implementar dessa outra forma, uma vez que o Android oferece uma estrutura para impleemntar através do *onDestroy*. O diagrama de atividades para gerar um teste pode ser observado na Figura 20

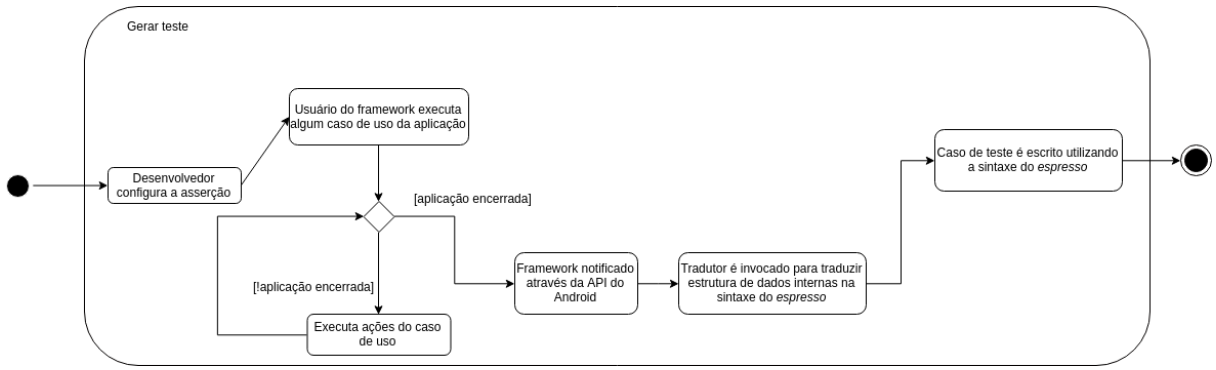

Figura 20 – Diagrama de atividades para o caso de uso de gerar um caso de teste.

(FONTE: Imagem feita pelo autor)

Assim que a aplicação for encerrada, o caso de teste será escrito no sistema de arquivos do dispositivo emulado. Para ter acesso ao arquivo criado pelo framework o usuário deve acessar a aba *Device File Explorer*. No sistema de arquivos, deve-se seguir o seguinte caminho: */data/user/0/pacote.raiz.da.aplicacao/files*. Nesse diretório é onde são guardados todos os arquivos de teste criado pelo framework. Ao abrir o arquivo de teste, o usuário deve copiar o conteúdo para um arquivo equivalente no pacote de testes da aplicação.

Ao final desse processo, o usuário terá então disponível em seu pacote de testes, um arquivo com caso de teste descrito e automatizado, em nível de usuário, responsável por testar o caso de uso que ele acabou de executar. Para criar mais casos de teste, basta que o usuário continue seguindo esse processo até estar satisfeito com a quantidade de

casos de teste criados. Quando quiser executar os casos de teste, basta executar o pacote que contém os testes da aplicação.

Ao executar o pacote com os testes da aplicação, o Android Studio executará todos os testes e sempre indicando o sucesso ou falha de cada asserção definida pelo usuário. Caso uma asserção obtenha sucesso, o relatório no final da execução irá mostrar apenas mensagens de êxito. Do contrário, caso ocorra algum erro, ele indicará qual e onde ocorreu o erro.

6.5 Limitações da ferramenta

Apesar de a ferramenta ser projetada para detectar as ações do usuário, durante a implementação do framework, houve alguns problemas técnicos que causaram algumas limitações no *capuccino*. A primeira a ser destacada é que quando o usuário digita algo errado, para cada letra que ele precisa apagar é necessário que antes de ele apagar ele pressione a tecla *enter* para depois efetivamente apagar a letra. Isso é necessário, porque como é possível observar em vários fóruns de discussões sobre a API do Android, o método utilizado para capturar a digitação do usuário dentro do framework, possui alguns bugs especificamente com a ação de apagar caracteres. Sendo asssim, é preciso que quem estiver gerando os testes se atente a esse fato.

Outra limitação da ferramenta desenvolvida, é que ela foi projetada para gerar um caso de teste por arquivo. Isso significa dizer que se uma determinada tela tiver 5 testes, a ferramenta irá gerar 5 arquivos diferentes. Para a gerência de código e da própria base de testes, acaba sendo um problema considerável, visto que a longo prazo, a medida que o projeto vai crescendo, seria gerado um arquivo para cada caso de teste, sendo que esses testes poderiam estar integrados em um único arquivo de teste para a tela em questão.

Mais uma restrição do *capuccino* são os eventos de rolagem (i.e. *scroll*). Apesar de a ferramenta conseguir dar suporte a esse tipo de ação, os aplicativos, por padrão, são configurados para manterem a inércia do movimento quando percebem um movimento de *scroll*. Infelizmente, esse tipo de comportamento, não é suportado pelo *UIAutomator*. Dessa forma, para que o framework possa ser utilizado corretamente, o testador deve desativar a inércia em movimentos de rolagem em sua aplicação, ou quando estiver usando a aplicação para gerar o teste, deve se certificar que quando fizer um movimento de *scroll*, o movimento não continue.

Por último, um problema da ferramenta é lidar com conexões. Dependendo do tipo da aplicação e das requisições HTTP executadas, o teste gerado, quando executado, pode acusar erros, visto que ele não está programado para aguardar a resposta da requisição. Uma solução paliativa, porém não definitiva para esse problema, é permitir que quando o usuário for gerar o teste, ele configure no framework, uma variável indicando que o

teste que ele irá criar envolve conexões. Dessa forma, sempre que o framework executar operações de clique, ele adiciona um delay na *thread*, para simular o aguardo da resposta.

Parte IV

Resultados obtidos

7 Exemplo de uso do framework

Para verificar a capacidade do framework em gerar os testes, desenvolveu-se uma aplicação simples que simula a compra e venda de produtos. Nessa aplicação, é possível cadastrar usuários, produtos e fazer operações envolvendo autenticação. O código desta aplicação está disponível junto com o framework desenvolvido em um repositório do GitHub do autor. A tela de autenticação dessa aplicação pode ser visualizada na Figura 21.

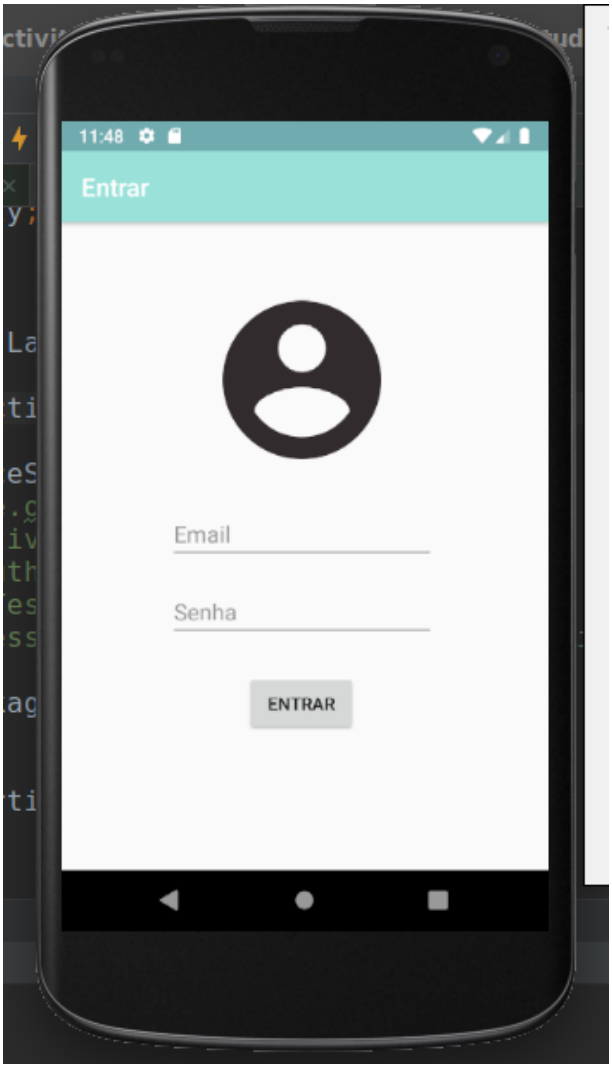

Figura 21 – Tela de autenticação da aplicação de teste

(FONTE: Imagem feita pelo autor)

Utilizando essa aplicação, foram criados testes para autenticar e cadastrar usuário. O código gerado pelo capuccino para o caso de se autenticar está disponível na Figura

22. Como pode ser observado, esse código se encontra no diretório de testes de aplicação. Porém, para que o teste ficasse disponível nesse pacote, foi necessário mover o arquivo do sistema de arquivos do dispositivo emulado para o sistema de arquivos do dispositivo hospedeiro. Os testes criados pelo framework sempre ficam disponíveis no diretório */data/user/0/pacote.raiz.da.aplicacao/files*.

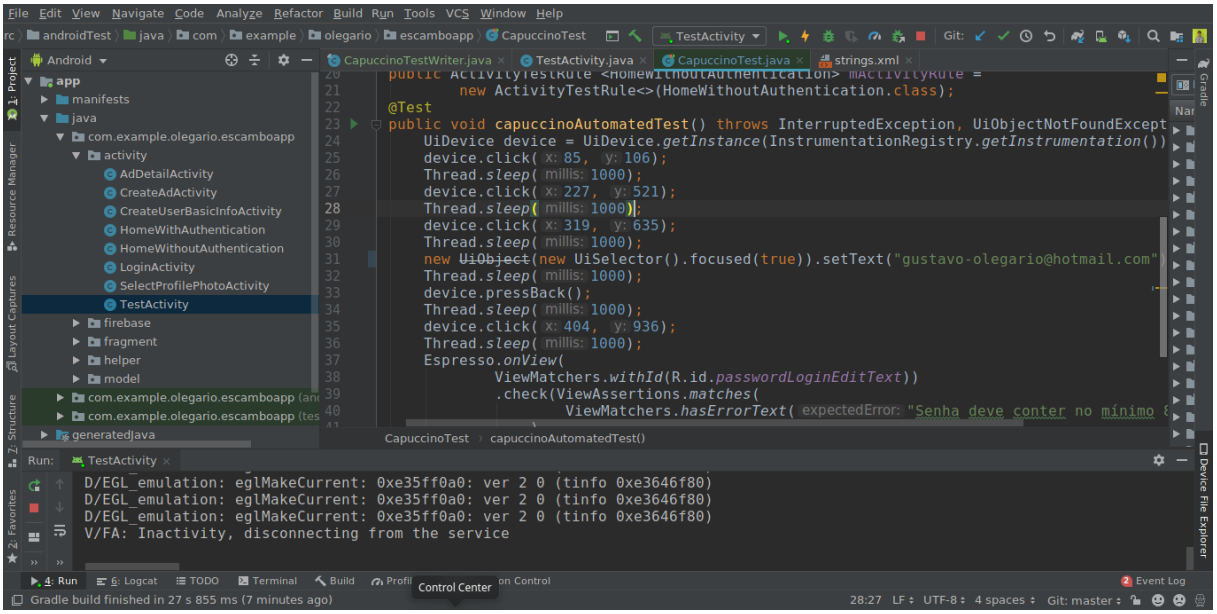

Figura 22 – Caso de teste gerado pelo *capuccino*.

(FONTE: Imagem feita pelo autor)

O teste em questão, é uma tentativa de autenticação em que o usuário não digita a senha. Nesse sentido a asserção definida previamente pelo usuário, verifica se uma mensagem de erro relacionada a senha é exibida. Ao executar o mesmo teste, um log acusando sucesso é exibido, como pode ser visto na Figura 23. Como explicado anteriormente, a asserção executada deve seguir a sintaxe do framework *espresso*. Além disso, essa mesma asserção foi previamente definida pelo usuário ao implementar a classe que herda *LauncherAppActivity*, como visto na Figura 18.

A execução de testes, dentro do Android Studio, mostra um *log* detalhado, no caso de alguma asserção estar incorreta, indicando o valor que estava esperando e qual foi o valor obtido. No caso do *log* exibido na Figura 23, a asserção executada pelo teste gerado obteve sucesso.

Ainda nessa mesma tela de autenticação apresentada, executou-se o fluxo de se autenticar com um usuário inválido, ou seja um usuário que não está cadastrado no banco de dados. Nesse sentido, a asserção definida previamente foi uma asserção que procura por uma mensagem que informe ao usuário que as credenciais informadas são inválidas. O código desse caso de teste pode ser visualizado na Figura 24.

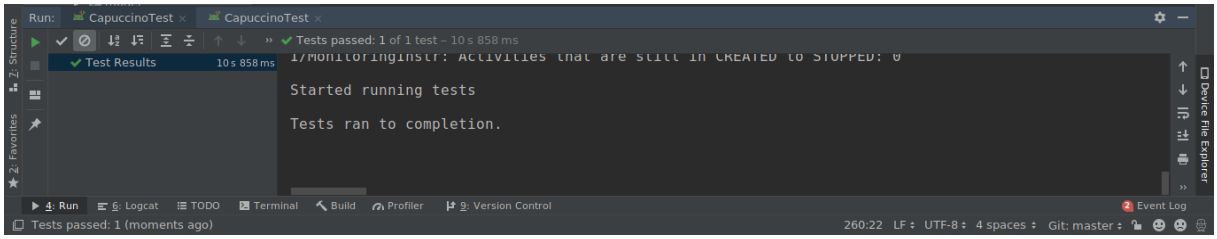

Figura 23 – Log indicando sucesso na execução do teste gerado pelo capuccino.

(FONTE: Imagem feita pelo autor)

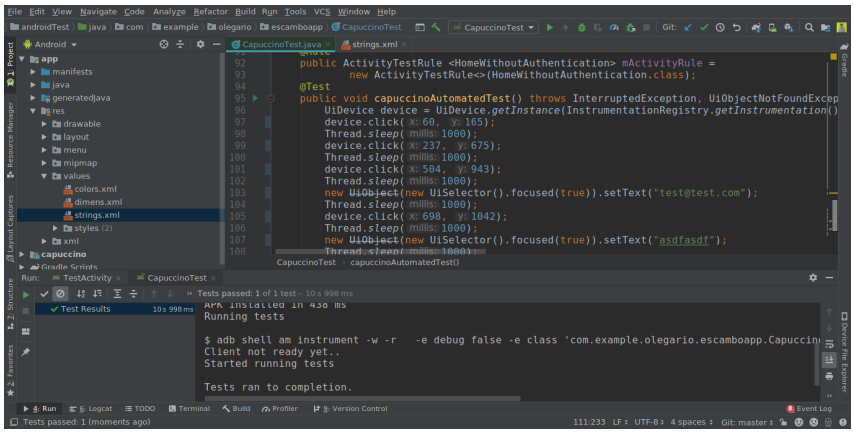

Figura 24 – Teste produzido pelo framework para autenticar um usuário inexistente.

(FONTE: Imagem feita pelo autor)

Apesar de ambos os casos de teste serem muito similares eles testam comportamentos bem distintos da aplicação. No primeiro caso o usuário tenta se autenticar sem fornecer todas as informações. Nesse sentido, é esperado que o sistema mostre um erro informando que todos os campos devem ser preenchidos para que o usuário possa proseguir. Já no segundo caso, o usuário tenta se autenticar com um usuário inválido (i.e. tanto campo de email e senha foram preenchidos, mas esse par não se encontra na base de dados da aplicação). Da mesma forma, a aplicação retorna uma mensagem de erro dizendo que usuário ou senha estão incorretos.

Ainda no que se refere aos testes gerados pela ferramenta, destaca-se as duas asserções criadas pelo usuário. Ambas podem ser visualizadas na Figura 25. Ambas foram escritas utilizando a sintaxe do *espresso*. Na primeira asserção o framework avalia se o campo de senha está exibindo a mensagem "E-mail invalido". Já na segunda asserção, está sendo consultado se o decorador foi exibido (i.e. o *toast* foi mostrado). Nesse segundo caso, o conteúdo da mensagem não foi verificado.

Somando todas as linhas de código geradas pelo framework para esses casos de

```
Espresso.onView(
    ViewMatchers.withId(
        R.id.passwordLoginEditText
    )
).check(
    ViewAssertions.matches(
        ViewMatchers.hasErrorText(
          "E-mail invalido"
        )
    )
);

Espresso.onView(
    ViewMatchers.withText(
        R.string.signInError
    )
).inRoot(
    RootMatchers.withDecorView(
        Matchers.not(
            mActivityRule.getActivity().getWindow().getDecorView()
        )
    )
).check(
    ViewAssertions.matches(
        ViewMatchers.isDisplayed()
    )
);
```

Figura 25 – Asserções escritas para cada um dos testes
(FONTE: Imagem feita pelo autor)

teste foram 150 linhas de código geradas em menos de uma hora. Infelizmente, não existe nenhuma métrica que seja capaz de correlacionar número de linhas produzidas por unidade de tempo. Entretanto, é nítido que o framework traz produtividade para um projeto por diversos fatores. O primeiro é que a pessoa que está testando a aplicação não precisa ser um programador ou ser um testador com longos anos de carreira. Um usuário comum utilizando a aplicação em um ambiente previamente configurado, seria perfeitamente possível de gerar os testes, desde que tivesse sido disponibilizado a ele a lista de asserções, o que acaba barateando os custos de um projeto.

A produtividade também cresce dentro do projeto, visto que os programadores, ou o time de QA, não necessitam mais aprender a como implementar um teste de interface de usuário nesse tipo de projeto. Basta utilizar a ferramenta e conhecer como o capuccino faz as asserções que o teste será criado. Isso poupa o tempo dos desenvolvedores que seria gasto para aprender a como implementar testes de interface de usuário em dispositivos Android.

Outro ganho considerável na produtividade é quando o código, da aplicação, passa por algum tipo de refatoração e os testes de interface de usuário passam a não funcionar mais. Em vez do time de QA despender tempo revisando o código de teste e tentando encontrar o erro, basta utilizar a aplicação normalmente, que um novo teste, adaptado as novas refatorações, será gerado por completo.

8 Conclusão

Revendo os objetivos apresentados no início do trabalho, tanto geral quanto específicos, fica claro que o framework implementado cumpre com o que foi proposto. Analisando, inicialmente, o objetivo geral do trabalho, o framework cumpre o que havia sido ambicionado: implementou-se um framework orientado objetos para a plataforma Android, capaz de gerar testes automaticamente através da captura de ações do usuário. O teste é gerado automaticamente de tal forma que as ações do usuário, capturadas da execução da aplicação, são descritas, programaticamente, em um caso de teste.

Quanto ao primeiro objetivo específico, que se refere à diminuição de esforço e tempo durante a fase de testes, também foi alcançado, visto que não é necessário mais implementar testes no nível do código. Basta que o testador, ou um usuário real, utilizem a aplicação alvo usando como guia algum caso de uso, que ao final da execução terão disponível um teste em nível de usuário. Consequentemente, acaba não sendo mais necessário que os membros da equipe de teste de um projeto aprendam a usar uma biblioteca de teste, basta apenas entenderem como definir uma asserção para o teste.

Já quanto ao segundo objetivo específico, que fala sobre uma ferramenta gratuita capaz de diminuir o esforço dos desenvolvedores na implementação de testes também foi completado, visto que a ferramenta é de código aberta, disponível em um repositório do Github, sem licença comercial. Além disso, a ferramenta automatiza toda a parte de simulação de interação de um usuário com a aplicação e permite que desenvolvedores concentrem seus esforços em outras tarefas que demandem mais criatividade.

Isso significa dizer que para o projeto não é mais necessário que haja alguém que domine uma biblioteca específica de testes em nível de usuário ou que o time necessite aprender uma nova tecnologia. Basta que os testadores utilizem o software como um usuário comum, o que implica dizer que o projeto tem redução de tempo e custos.

Como sugestão de trabalhos futuros, indica-se o estudo, análise e correção de alguma das limitações que o capuccino apresenta. Para resolver o problema de gerar um arquivo para cada caso de teste, uma possível solução seria o framework analisar se já existe um arquivo gerado com o mesmo nome. Em caso positivo, em vez de ele sobrescrever o arquivo com os dados atuais, ele pode gerenciar o código internamente para adicionar um novo caso de teste. Para resolver a limitação de rede, pode-se implementar uma emulação, via *proxy*, para responder as requisições localmente sem ter a necessidade de se comunicar com o *backend*.

Uma outra possível sugestão de trabalho seria verificar como o framework se comporta com aplicações como jogos eletrônicos. Esse é um caso interessante de ser testado,

visto que a interface desse tipo de aplicação costuma ser muito dinâmica. Nesse sentido, seria possível avaliar como capuccino reage com programas desse tipo e refatorar o que fosse necessário para que ele se adaptasse e pudesse gerar os testes corretamente. Outra possibilidade seria estudar alguma maneira de fazer com que o framework aguarde corretamente o retorno das requisições feitas na rede. Sendo assim, seria possível desenvolver um módulo que monitorasse o consumo de rede da aplicação e que indicasse para o framework quais ações envolvem operação com a rede ou não.

Outra possibilidade de sugestão de trabalho futuro seria implementar um método que disponibilizasse o arquivo de teste direto no sistema de arquivos do computador hospedeiro, dentro do pacote de teste da aplicação. Como explicado anteriormente, os arquivos gerados pelo framework são escritos no sistema de arquivos do dispositivo emulado. Isso requer que toda vez que um teste é gerado, o usuário exporte manualmente o arquivo para o diretório de teste do aplicativo. Seria muito mais eficiente quando o framework fosse escrever o teste, ele escrevesse diretamente no diretório de testes real da aplicação.

Uma adição interessante a esse trabalho seria, também, reimplementar o módulo que detecta cliques na tela para fazer a detecção através dos identificadores globais do componente e não de posições X e Y. Dessa forma quando uma tela da aplicação fosse restruturada, o teste continuaria funcionando perfeitamente, desde que os identifica dos componentes da tela fosse mantido.

No geral, o framework cumpre aquilo que se propôs a fazer e automatiza a produção de testes mediante a utilização de uma aplicação Android por parte de um usuário. Nesse sentido, há uma diminuição de esforço no que se refere à implementação de testes, visto que o framework é capaz de gerar casos de teste automaticamente.

Referências

WAZLAWICK, R. S. *Engenharia de Software para Sistemas de Informações: Conceitos e práticas que fazem sentido*. Florianópolis [s.n.], 2012.

DIJKSTRA, E. W. *The Humble Programmer*. [S.I], 1971. Disponível em: <https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html>. Acesso em 2 de Setembro de 2017.

JEREMIAH, J. *Is agile the new norm?*. [S.I.:s.n.], 2017 Disponível em: <https://techbeacon.com/survey-agile-new-norm>. Acesso em 2 de Setembro de 2017.

KELLY, A. *Programmers Without TDD Will be Unemployable by 2022*. [S.I.:s.n.], 2014 Disponível em: <https://dzone.com/articles/programmers-without-tdd-will>. Acesso em 22 de Abril de 2018.

ERIKSSON, U. *How much time should you spend on testing?*. [S.I.:s.n.], 2014 Disponível em: <https://goo.gl/bTfQsa>. Acesso em 2 de Setembro de 2017.

SILVA, Ricardo P. e, PRICE, R. T. *A busca de generalidade, flexibilidade e extensibilidade no processo de desenvolvimento de frameworks orientados a objetos*. [S.I.:s.n.], 1998 Disponível em: <https://www.inf.ufsc.br/~ricardo.silva/publications/Ideas98.PDF>. Acesso em 4 de Março de 2018.

Software design pattern. In: Wikipédia, a enciclopédia livre. Flórida: Wikimedia Foundation, 2018. Disponível em: <https://en.wikipedia.org/w/index.php?title=Software_design_pattern&oldid=834346932>. Acesso em 19 de Abril 2018.

JOHNSON, R. E. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. [S.I.:s.n.], 1993 Disponível em: <https://link-springer-com.ez46.periodicos.capes.gov.br/chapter/10.1007%2F3-540-47910-4_21>. Acesso em 26 de Março de 2018.

KENT, B., ANDRES, C. *Extreme Programming Explained*. [S.I.:s.n.], 2004.

ERIKSSON, U. *3 Reasons Why It's Important to Refactor Tests*. [S.I.:s.n.], 2016 Disponível em: <https://qualitycoding.org/why-refactor-tests/#comments>. Acesso em 4 de Maio de 2018.

FOWLER, M. *Making Stubs*. [S.I.:s.n.], 2003 Disponível em: <https://martinfowler.com/bliki/MakingStubs.html>. Acesso em 4 de Maio de 2018.

FOWLER, M. *Test Pyramid*. [S.I.:s.n.], 2012 Disponível em: <https://martinfowler.com/bliki/TestPyramid.html>. Acesso em 4 de Maio de 2018.

BAGMAR, A. *Behavior Driven Testing (BDT) in Agile*. [S.I.:s.n.], 2012 Disponível em: <https://www.slideshare.net/abagmar/anand-bagmar-behavior-driven-testing-bdt-in-a

gile>. Acesso em 2 de Junho de 2018.

KHAN, M. E., KHAN, F. *A Comparative Study of White Box, Black Box and Grey Box Testing Techniques*. [S.I.:s.n.], 2012 Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.261.1758&rep=rep1&type=pdf>. Acesso em 2 de Junho de 2018.

Number of available applications in the Google Play Store from December 2009 to March 2018. [S.I.:s.n.], 2018 Disponível em: <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>. Acesso em 2 de Junho de 2018.

MOBILE (ANDROID) HARDWARE STATS 2017-03. [S.I.:s.n.], 2018 Disponível em: <https://web.archive.org/web/20171113032047/https://hwstats.unity3d.com/mobile/cpu-android.html>. Acesso em 2 de Junho de 2018.

HALPERN, M., ZHU, Y., REDDI, V. J. *Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction*. [S.I.:s.n.], 2016 Disponível em: <http://matthewhalpern.com/publications/mobile-cpus-hpca-2016.pdf>. Acesso em 1 de Junho de 2018.

Android (operating system). In: Wikipédia, a enciclopédia livre. Flórida: Wikimedia Foundation, 2018. Disponível em: <https://en.wikipedia.org/wiki/Android_(operating_system)>. Accesso em 2 de Junho 2018.

Smartphone OS. [S.I.:s.n.], 2018 Disponível em: <https://www.idc.com/promo/smartphone-market-share/os>. Acesso em 2 de Junho de 2018.

KUROSE, J., KEITH, R.*Computer Networking A Top-Down Approach*. [S.I.:s.n.], 2013 Disponível em: <http://www.bau.edu.jo/UserPortal/UserProfile/PostsAttach/10617_1870_1.pdf>. Acesso em 14 de Junho de 2018.

KAWAKAMI, L., KNABBEN A., RECHIA, D., BASTOS, D., PEREIRA, O., SILVA, R. P., SANTOS, L. *An Object-Oriented Framework for Improving Software Reuse on Automated Testing of Mobile Phones*. IFIP 19th TESTCOM - 7th FATES. Tallinn, 2007

SILVA, Ricardo P. e, PRICE, R. T. *A busca de generalidade, flexibilidade e extensibilidade no processo de desenvolvimento de frameworks orientados a objetos*. In: Proceedings of Workshop Iberoamericano de Engenharia de Requisitos e Ambientes de Software (IDEAS'98). Torres: apr. 1998. v.2, p.298-309.

JAMROZIK, K., ZELLER, A. *DroidMate: A Robust and Extensible Test Generator for Android*. ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2016 Disponível em: <http://www.boxmate.org/files/DroidMate_MOBILESoft_2016.pdf>. Acesso em 1 de Junho de 2019.

STACK OVERFLOW *Android - cannot capture backspace/delete press in soft. keyboard*. [S.I.:s.n.], 2014 Disponível em: <https://stackoverflow.com/questions/18581636/andro

id-cannot-capture-backspace-delete-press-in-soft-keyboard>. Acesso em 2 de Junho de 2019.

MESZAROS, G. *XUnit Test Patterns: Refactoring Test Code*. [S.I.:s.n.], 2011 Disponível em: <https://github.com/oolee/software-development-ebooks/blob/master/%5BxUnit%20Test%20Patterns%20Refactoring%20Test%20Code%20(Addison-Wesley%20Signature%20Series%20(Fowler))%20Kindle%20Edition%20by%20Gerard%20Meszaros%20-%202007%5D.pdf>

BECK, K. *Test-Driven Development By Example*. [S.I.:s.n.], 2002. Disponível em: <http://docs.ludost.net/Programming%20and%20Software%20Development/Test-Driven%20Development/Test-Driven%20Development%20By%20Example%20-%20Kent%20Beck%20(2002).pdf>

MYERS, G. J. *The Art of Software Testing*. [S.I.:s.n.], 2004. Disponível em: <http://barbie.uta.edu/~mehra/Book1_The%20Art%20of%20Software%20Testing.pdf>

Petrenko, Alexandre & Simão, Adenilso & Maldonado, José. *Model-based testing of software and systems: Recent advances and challenges*. [S.I.:s.n.], 2012. Disponível em: <https://www.researchgate.net/publication/257468201_Model-based_testing_of_software_and_systems_Recent_advances_and_challenges> LEACH, R. J. *SOFTWARE REUSE: METHODS, MODELS, AND COSTS*. [S.I.:s.n.], 2011 Disponível em: <https://pdfs.semanticscholar.org/700b/83bc8d4a2e4c1d1f4395a4c8fb78462c9f5a.pdf>

JALENDER, B. Et al. *A PRAGMATIC APPROACH TO SOFTWARE REUSE* Journal of Theoretical and Applied Information Technology, 2014 Disponível em: <https://www.researchgate.net/publication/266287893_A_pragmatic_approach_to_software_reuse/download>

JOHNSON, R. E. Et al. *Designing Reusable Classes* Journal of Object-Oriented Programming, Junho/Julho 1988, Volume 1. Disponível em: <http://www.laputan.org/drc.html>

GAMMA, E., HELM, R., JOHNSON, R., VISSIDES, J. *Design Patterns Elements of Reusable Object-Oriented Software* [S.I.:S.N.], 1994. Disponível em: <http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf>

FOWLER, M. *UML Distilled Second Edition A Brief Guide to the Standard Object Modeling Language* [S.I.:S.N.], 1999. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.183.2984&rep=rep1&type=pdf>