UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COORDENADORIA DO CURSO DE ENGENHARIA DE SOFTWARE

MATHEUS HENRIQUE MOLINETE

UMA ANÁLISE EMPÍRICA DA OCORRÊNCIA DOS TESTES QUEBRADIÇOS EM APLICATIVOS HÍBRIDOS

TRABALHO DE CONCLUSÃO DE CURSO

DOIS VIZINHOS

2019

MATHEUS HENRIQUE MOLINETE

UMA ANÁLISE EMPÍRICA DA OCORRÊNCIA DOS TESTES QUEBRADIÇOS EM APLICATIVOS HÍBRIDOS

Trabalho de conclusão de curso apresentado como requisito parcial à obtenção do título de Bacharel em Engenharia de Software, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Rafael Alves Paes de Oliveira

DOIS VIZINHOS

2019

TERMO DE APROVAÇÃO

Uma análise empírica da ocorrência dos testes quebradiços em aplicativos híbridos

por

Matheus Henrique Molinete

Este Trabalho de Conclusão de Curso foi apresentado em 28 de Novembro de 2019 como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software. O(a) candidato(a) foi arguido(a) pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Rafael Alves Paes de Oliveira
Presidente da Banca

Rodolfo Adamshuk Silva
Membro Titular

Simone de Sousa Borges
Membro Titular

* A Folha de Aprovação assinada encontra-se na Coordenação do Curso

AGRADECIMENTOS

Agradeço primeiramente a Deus, por permitir-me estar aqui e completar essa longa jornada.

Minha eterna gratidão aos meus pais Marino Luiz Molinete e Adriane Regina de Rossi Molinete, meu irmão Marcos Luis Molinete e toda a minha família por sempre estarem ao meu lado e acreditarem nos meus sonhos. Também externo minha eterna gratidão a minha namorada Camila Eduarda de Lima que sempre esteve ao meu lado e nunca mediu esforços para ajudar.

Agradeço também aos professores que participaram dessa longa jornada, em especial ao meu orientador, Prof. Dr. Rafael Alves Paes de Oliveira que contribuiu e auxiliou em todos os momentos necessários. Também agradeço aos professores da banca Prof. Dr. Francisco Carlos Monteiro Souza e Prof^{a}Dr^a Simone de Sousa Borges pelas contribuições e sugestões de melhorias. Por fim, agradeço a Prof^{a}Dr^a Alinne Cristinne Corrêa Souza pelos auxílios essenciais fornecidos durante as aulas de Experimentação em Eng. Software.

Por fim, agradeço imensamente aos meus amigos e colegas, por todos os momentos de apoio, amizade e dificuldades que serão levados para o restante da vida.

RESUMO

Molinete, M. H. M. UMA ANÁLISE EMPÍRICA DA OCORRÊNCIA DOS TESTES QUEBRADIÇOS EM APLICATIVOS HÍBRIDOS. 86 f. Trabalho de conclusão de curso – Coordenadoria do Curso de Engenharia de Software, Universidade Tecnológica Federal do Paraná. Dois Vizinhos, 2019.

Na Engenharia de Software (ES), a área de Teste de software é considerada uma das mais importantes, pois com ela é possível encontrar inconsistências e corrigir problemas antes da entrega. Uma das formas de executar o teste de software é por meio da automatização, na qual a execução dos conjuntos de testes pode ser realizada diversas vezes em diversos momentos. O processo de automatização vem crescendo a cada dia em diversos tipos de projetos. Nesse contexto da automatização, surgiu um problema conhecido como Teste quebradiço (*Flaky test*), que pode ser definido como os testes de software cujo resultado é incerto, ou seja, em algumas execuções é dado como sucesso (*pass*) sem erros, mas em outras execuções ele é rejeitado e interrompido por falhas (*fail*). Para isso, o trabalho realizado visou apresentar a ocorrência de testes quebradiços em aplicativos híbridos, por meio da execução de um estudo de caso. O estudo de caso executado, analisou testes automatizados em diferentes cenários de diversos projetos de aplicativos híbridos. Os resultados obtidos, mostraram a ocorrência dos testes quebradiços em aplicativos híbridos, também auxiliaram na identificação quanto às causas para os testes quebradiços e, por fim, qual o tipo de técnica de teste de software está mais propensa à ocorrência desse problema. Os resultados mostraram que a técnica de teste de software funcional é mais favorável à ocorrência dos testes quebradiços, também possibilitaram identificar que a causa de espera assíncrona ocorre com mais facilidade em projetos de aplicativos híbridos. O trabalho realizado promoveu contribuições para academia e indústria. Para a academia as informações obtidas e sintetizadas ajudam futuros trabalhos e promovem novos estudos sobre os testes quebradiços. Para a indústria, as informações obtidas auxiliam em projetos de software que possam vir a sofrer com os testes quebradiços, fornecendo dados de como ocorrem os testes quebradiços para possíveis prevenções.

Palavras-chave: Engenharia de software, Testes quebradiços, Testes de sofware, Automatização de seste, Testes automatizados, Aplicativos híbridos

ABSTRACT

Molinete, M. H. M. AN EMPIRICIS ANALYSIS OF THE OCCURRENCE OF FLAKY TESTS IN HYBRID APPLICATIONS. 86 f. Trabalho de conclusão de curso – Coordenadoria do Curso de Engenharia de Software, Universidade Tecnológica Federal do Paraná. Dois Vizinhos, 2019.

Considering Software Engineering (ES) activities, Software Testing is considered one of the most important activities because of it one can find inconsistencies, allowing fix problems before delivery. A feasible way to perform software testing is through automation, where testing data can be executed several times at various occasions. The automation process has been growing every day in many types of projects. In this context of automation, a problem that has emerged is known as Flaky Test, which can be broadly defined as software testing whose outcome is uncertain, which means in some executions it is given as pass with success, but in other executions it is rejected and interrupted by fail. For this, in this study, we aimed to present the occurrence of Flaky Tests in hybrid apps, by performing a case study. The case study we performed analyzed automated tests in different scenarios of various hybrid apps projects. The results empirically obtained through the case study, showed the occurrence of Flaky tests in hybrid apps, also it helped to identify the causes for Flaky tests and, finally, which type of software testing technique is more error-prone to this flaky circumstances. The contributions obtained showed that the functional software testing technique is more error-prone to Flaky tests, also made it possible to identify that the cause of asynchronous waiting occurs more easily in hybrid apps projects. The work we conducted has promoted contributions to academia and industry. For the academy, information gathered and synthesized helps on future studies and promotes new studies on Flaky tests. For the industry, the information obtained assists in software projects that may suffer from Flaky testing, providing data on how flaky testing occurs for possible predictions.

Keywords: Software engineering, Flaky test, software test, Test automation, Automated Testing Hybrid applications,

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ACM	*Association for Computing Machinery*
API	*Application Programming Interface*
CE	Critérios de Exclusão
CI	Critérios de Inclusão
CSS	*Cascading Style Sheets*
ES	Engenharia de Software
GFC	Grafo de fluxo de controle
GQM	*Goal Question Metric*
HTML5	*Hypertext Markup Language*
IC	Integração Contínua
IEEE	*Institute of Electrical and Electronics Engineers*
NASA	*National Aeronautics and Space Administration*
NRE	Número de execuções
PWA	*Progressive Web Aps*
RI	Revisão Integrativa
SDKS	Kit de desenvolvimento de software
SWEBOK	*Software Engineering Body of Knowledge*
VV&T	Verificação, Validação e Teste
URL	*Uniform Resource Locator*

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

O presente trabalho consiste de um esforço de pesquisa na área de teste de software, mais especificamente acerca de um problema conhecido como teste quebradiço (do inglês *Flaky test*). A área de teste de software está inserida na Engenharia de Software (ES) e é considerada uma das áreas mais importantes no processo de desenvolvimento de software. Esta área tem como objetivo executar um programa visando averiguar se o comportamento está de acordo com o esperado (DELAMARO, 2016).

A ES é definida como uma disciplina da engenharia tradicional e visa apoiar todas as fases do desenvolvimento de um software por meio de suas áreas de conhecimento, desde a concepção inicial, até a finalização e manutenção do projeto (SOMMERVILLE, 2011). A ES vem ganhando espaço desde os meados dos anos 1970 após problemas relacionados com a crise de software, como o intuito de identificar soluções para os problemas inseridos dentro do desenvolvimento de software (SOMMERVILLE, 2011). No contexto da ES, a área de Teste de Software é considerada essencial, pois com ela é possível encontrar inconsistências e corrigir problemas antes da entrega do produto final.

O teste de software é uma atividade dinâmica que tem como intuito executar um programa com alguns valores de entrada e verificar se o comportamento está de acordo com o esperado e especificado (DELAMARO, 2016). Dentre as formas de executar os testes de software, pode-se destacar a execução automatizada, que tem como objetivo criar conjuntos de testes[1] para verificar a ocorrência de falhas a cada funcionalidade nova disposta no software (SOMMERVILLE, 2011). A vantagem da automatização é que um conjunto de testes pode ser executado diversas vezes de forma rápida e automática. Na indústria, empresas de desenvolvimento têm como garantia o resultado desses testes para a tomada de decisão no momento de liberar novas funcionalidades e também na refatoração de código.

[1] É o agrupamento de diversos casos de testes para uma execução única (DELAMARO, 2016)

Levando em conta os testes automatizados, com a evolução dos software contemporâneos, surgiu um termo conhecido como "Testes quebradiços". Genericamente, esse termo representa aqueles testes cujo resultado é incerto, ou seja, em alguns momentos é dado como sucesso (*pass*), mas em outras execuções, aleatoriamente, ele é interrompido por falhas (*fail*) (THORVE; SRESHTHA; MENG, 2018; LUO et al., 2018). Essa incerteza no resultado, é considerada prejudicial para o projeto, tanto para a equipe de testes quanto para a equipe de desenvolvimento do projeto.

Um dos cenários contemporâneos comuns do aparecimento do teste quebradiço é a Integração Contínua (IC), que é muito usada para ajudar na manutenção e controle de projetos de software. A IC é uma prática adotada para a integração de códigos fontes a cada funcionalidade nova realizada pelos desenvolvedores. Os desenvolvedores juntam os códigos em um repositório central, no qual as funcionalidades aprovadas e mescladas passam por uma bateria de testes automatizados, com o objetivo de verificar se tudo está funcionando corretamente (STÂHL; BOSCH, 2014). Com o aumento da complexidade das aplicações a IC passou a sofrer dificuldades com a execução dos testes de software, pois a quantidade de testes tornou-se grande e muitos deles acabaram ficando desatualizados.

Atualmente, a literatura é carente de estudos realizados para identificar as causas e também possíveis soluções para testes quebradiços em domínios específicos (BELL OWOLABI LEGUNSEN, 2018; THORVE; SRESHTHA; MENG, 2018; LUO et al., 2018), o que demonstra um tema pouco estudado ainda por parte dos pesquisadores. A ocorrência dos testes quebradiços em projetos reais, geralmente acontece de modo distinto e pode ocorrer de diferentes formas em diversos projetos, dependendo do domínio da aplicação e de recursos de infraestrutura consumidos.

Diante do apresentado, procurando contribuir no cenário dos testes quebradiços, o presente estudo realiza uma análise empírica-quantitativa em aplicativos híbridos, buscando uma avaliação na ocorrência dos testes quebradiços e identificando padrões de acontecimento. Os aplicativos híbridos são aqueles que usam tecnologias web padrões e conseguem ser transformados para diversas plataformas como Android e iOS, com somente um pacote de códigos compilados (GOK; KHANNA, 2013). Além disso, é possível utilizar boa parte dos recursos nativos dos celulares se comparado com os aplicativos nativos. Os aplicativos híbridos vêm crescendo demasiadamente no mercado de desenvolvimento mobile no setor privado e tornando-se populares entre os desenvolvedores.

1.2 MOTIVAÇÃO

Grandes organizações do setor de tecnologia como [2]Google®, [3]Netflix® e [4]Microsoft® já relataram (MICCO, 2016; MEMON, 2017; HERZIG; NAGAPPAN, 2015) que sofrem com os testes quebradiços e afirmam que é prejudicial para toda a equipe envolvida no desenvolvimento de software. A validação de algumas funcionalidades por meio dos testes quebradiços podem ser demoradas e ocasionam um encarecimento dos custos do projeto por conta dos atrasos nos cronogramas estabelecidos. A maneira que os projetos crescem, o tamanho do projeto evoluí, a complexidade das funcionalidades também cresce, faz com que a evidência de testes quebradiços se torne cada vez maior.

Diante do apresentado, uma das principais motivações associadas ao estudo realizado é a contribuição com estudos relacionados aos aplicativos híbridos. Até a data do estudo nenhuma pesquisa foi encontrada analisando os testes quebradiços neste tipo de projeto e que usam a mesma linguagem de programação. Os esforços de pesquisas até o momento são executados com base em projetos variados ou então somente um tipo de projeto, como foi realizado no trabalho de Luo et al. (2018). Tal estudo observou especificamente projetos de aplicativos Android. A cada ano as aplicações híbridas crescem, fazendo com que o número de adeptos a soluções híbridas de desenvolvimento cresça também. Pesquisas apontam que desde 2017 até os dias atuais o aumento da criação de aplicativos híbridos pode-se chegar a 30% (IONIC, 2017).

Por fim, é importante destacar que os esforços de pesquisas relacionados aos testes quebradiços podem contribuir, tanto para academia, quanto para indústria, já que muitas organizações sofrem com esse problema em seus projetos e ainda existem poucas soluções no mercado.

1.3 OBJETIVO

Este trabalho de conclusão de curso tem como objetivo a investigação da ocorrência dos testes quebradiços em aplicativos híbridos por meio da análise de testes automatizados em projetos reais, realizando avaliações quantitativas, sistemáticas, e categorizações dos resultados obtidos. Por meio de um objetivo geral, criaram-se alguns objetivos específicos:

- Obter informações sobre testes quebradiços em projetos híbridos, junto com as

[2]acesse: https://about.google/intl/pt-BR/products/
[3]acesse: https://www.netflix.com/
[4]acesse: https://www.microsoft.com/pt-br

possíveis causas e soluções;

- Identificar e encontrar projetos híbridos reais com testes automatizados e identificar padrões nos testes quebradiços para possíveis causas;
- Catalogar os resultados obtidos nos testes automatizados de projeto híbridos e proporcionar uma ajuda para futuras pesquisas nos testes quebradiços;
- Catalogar motivos da ocorrência dos testes quebradiços para aplicativos híbridos; e
- Conduzir investigações de modo a levantar informações relevantes que contribuam para a prevenção da ocorrência de testes quebradiços.

1.4 ORGANIZAÇÃO

Além do presente capítulo introdutório, este trabalho conta com outros seis capítulos organizados da seguinte forma: o Capítulo 2 descreve os aspectos conceituais da ES, Teste de software, IC, Desenvolvimento *mobile* e por fim os Testes quebradiços; no Capítulo 3 é apresentado como foi executado a revisão integrativa, juntamente com os resultados; no Capítulo 4 é apresentado a metodologia utilizada para execução do trabalho, juntamente com seus materiais; no Capítulo 5 são apresentadas as questões de pesquisa, design experimental e os resultados obtidos; no Capítulo 6 s ão apresentadas as respostas para as questões de pesquisa do trabalho, juntamente com análise do resultados obtidos, incluindo trabalhos futuros e ameaças a validade; por fim no Capítulo 7 destacam-se os problemas que o trabalho aborda e também as contribuições do mesmo.

2 ASPECTOS CONCEITUAIS

Neste capítulo, são apresentados os principais conceitos usados no desenvolvimento do presente trabalho. A organização das seções está definida da seguinte forma: Na Seção 2.1 são apresentados os conceitos de engenharia de software e suas áreas de conhecimento, na Seção 2.2 são descritos os conceitos, técnicas e terminologias de teste de software, juntamente com automatização de teste, na Seção 2.3 são descritos os conceitos sobre integração contínua, na Seção 2.4 são descritos os tipos de desenvolvimento *mobile*, por fim na Seção 2.5 são descritos os conceitos relacionados a teste quebradiço.

2.1 ENGENHARIA DE SOFTWARE

Em meados dos anos 1970, após diversos problemas relacionados à crise de software, uma disciplina conhecida como Engenharia de Software veio à tona para identificar soluções para os principais problemas encontrados no desenvolvimento de um software. Em geral, os problemas envolviam atraso nas entregas, funcionalidades solicitadas que não eram desenvolvidas o que resultava em um alto custo de desenvolvimento (SOMMERVILLE, 2011).

A ES, em Sommerville (2011) foi definida como uma disciplina de engenharia tradicional, com o objetivo de conceituar o desenvolvimento de um software. Com ajuda da ES, pode-se dividir o desenvolvimento de software em processos desde o início do projeto até conclusão do mesmo, usando ferramentas e métodos como auxílio para um bom desenvolvimento.

A ES está dividida em áreas do conhecimento, que foram estabelecidas pelo *Software Engineering Body of Knowledge* (SWEBOK), um guia criado pelo Instituto de Engenheiros Eletricistas e Eletrônicos ou do inglês *Institute of Electrical and Electronics Engineers* (IEEE) com objetivo de fornecer uma base de conhecimento para a Engenharia de Software. As áreas de conhecimento são divididas, de acordo com o guia, em: (I) Requisitos de software; (II) Projeto de software; (III) Design de software; (IV) Teste

de software; (*V*) Manutenção de software; (*VI*) Gerência de configuração de software; (*VII*) Gerência de engenharia de software;(*VIII*) Ferramentas e métodos de engenharia de software;(*IX*) Qualidade de software; (*X*) Práticas profissionais em engenharia de software; (*XI*) Economia da engenharia de software; (*XII*) Fundamentos de computação; (*XIII*) Fundamentos de matemática; e (*XIV*) Fundamentos de engenharia (SOCIETY, 2014).

As áreas de conhecimento da ES fizeram com que processos fossem seguidos no decorrer do desenvolvimento de software. Dentro desses processos, uma das etapas mais importantes é o Teste de software que será apresentado na próxima seção.

2.2 TESTE DE SOFTWARE

No desenvolvimento de software profissional, uma série de atividades são distribuídas e realizadas dentro de fases específicas, cada uma delas com sua importância para tentar garantir um projeto sem inconsistências (SOMMERVILLE, 2011). A cada interação com o projeto, seja uma funcionalidade nova, uma manutenção em partes do projeto ou uma correção de defeitos, é aberta a possibilidade de ocorrência de novos defeitos.

Diante disso, dentro da área de conhecimento de teste de software, as técnicas conhecidas como Verificação, Validação e Teste (VV&T) têm sido introduzidas ao processo de desenvolvimento de software com o intuito de demonstrar que o projeto está de acordo com o especificado pelo cliente e também evitar problemas frequentes durante o desenvolvimento, e a cada interação da equipe. Dentro dessas técnicas, uma das principais atividades é o teste de software (DELAMARO, 2016).

O teste de software consiste de uma análise dinâmica e tem como objetivo executar programas utilizando algumas entradas de dados verificando se o funcionamento e comportamento do programa estão de acordo com o esperado (DELAMARO, 2016).

Em um projeto de desenvolvimento de software, tudo o que é gerado durante o processo de desenvolvimento é considerado um artefato testável, isso incluí, desde códigos fontes, documentações até estruturas de dados (NEVES, 1999). Desse modo, quanto mais validações e testes, menos chances de inconsistências.

A atividade de teste de software é considerada uma das mais difíceis no processo de desenvolvimento de software. Diversos problemas podem aparecer durante o desenvolvimento do software, pois os seres humanos envolvidos podem cometer enganos (*Mistake*) que podem produzir defeitos (*Faults*) durante a realização de uma atividade ou processo do desenvolvimento do software. Esses defeitos durante a execução do programa podem

ocasionar erros (*Error*), que são caracterizados por estados inconsistentes ou inesperados do programa em execução. Por fim, esse estado pode gerar falhas (*Failure*), fazendo com que o resultado esperado para o programa seja falho como mostra a Figura 1 (DELAMARO, 2016).

Figura 1: Defeitos, Erros e Falha.

Fonte: Neto (2007)

Devido a importância do teste de software dentro do processo de desenvolvimento de software, esforços estão sendo realizados para criar padrões com algumas terminologias usadas. O padrão IEE 24765-2010 (ISO/IEC/IEEE. . . ,) ,define alguns termos e conceitos:

- defeito (*fault*): passo, processo ou definição de dados incorretos, como uma instrução ou um comando escrito de maneira incorreta;
- engano (*mistake*): ação humana que produz um resultado diferente do esperado, um exemplo disso é um código escrito de modo errôneo pelo programador;
- erro (*error*): diferença entre o valor calculado e o esperado, ou seja, qualquer estado do programa em execução que apresente resultado diferente do esperado;
- falha (*failure*): saída incorreta do programa em relação ao esperado e especificado, ou seja, ocorre uma falha quando o programa não corresponde ao especificado demonstrando um erro;
- dado de teste (*test data*): dados criados ou selecionados de entrada para a execução de um caso de teste;
- caso de teste (*test case*): condições para executar o teste e os resultados esperados;
- conjunto de teste (*test suite*): conjunto de casos de testes usados no produto a ser testado; e

- critério de teste (*test criteria*): determina quais os critérios que os casos de testes devem ter para serem aceitos ou não.

2.2.1 ETAPAS E FASES DO TESTE DE SOFTWARE

O teste de software é dividido em quatro etapas: (I) planejamento de testes; (II) projeto de casos de testes; (III) execução; (IV) avaliação dos resultados de teste. Essas etapas durante o desenvolvimento do software são executadas e se concretizam em outras fases de teste: de unidade, de integração, de sistema, aceitação e regressão(PEZZÈ; YOUNG, 2008).

- teste de unidade: tem como principal objetivo averiguar unidades menores do programa, podendo ser funções, métodos, procedimentos com o intuito de identificar erros de lógica na implementação e valores de variáveis;
- teste de integração: é responsável pela verificação da construção do sistema por meio da junção das partes do programa. Ao realizar a integração é necessário verificar se tudo está funcionando de acordo com o esperado;
- teste de sistema: é responsável por verificar as funcionalidades do programa após o seu término. Ao concluir as funcionalidades, devem-se executar os testes, com o intuito de validar se o programa está com as funcionalidades implementadas e funcionando de acordo com os requisitos funcionais e não funcionais;
- teste de aceitação: teste de aceitação verifica o comportamento e aceitação do sistema de acordo com as especificações do cliente; e
- teste de regressão: tem como objetivo verificar o funcionamento das funcionalidades antigas a cada nova versão criada do programa com funcionalidades novas.

2.2.2 TÉCNICAS E CRITÉRIOS DE TESTE

Em busca da minimização dos custos dos testes de software, a aplicação de técnicas de testes de software auxiliam em como realizar a execução do teste e também em como planejá-lo para ter um melhor controle, enquanto os critérios de testes ajudam a técnica a medir em que momento deve-se parar de realizar o teste (DEMILLO, 1980). As diferentes técnicas de teste estão associadas ao recurso utilizado como fonte de informação para a criação dos casos de teste.

A partir das técnicas de teste, derivam-se os critérios. A união dos critérios e técnicas de software é fundamental para um caso de teste com qualidade, eles ajudam a minimizar a quantidade total dos casos de testes e também a identificar os defeitos com mais facilidade. Esses benefícios acabam gerando um custo menor para o projeto.

Os critérios de teste de software são criados com o objetivo de disponibilizar uma maneira sistemática de selecionar os conjuntos de domínio de entradas e ser mais efetivo ao encontrar defeitos. Os critérios genericamente podem ser estabelecidos a partir de três técnicas: (*I*) Técnica funcional; (*II*)Técnica estrutural; e (*III*) Técnica baseada em defeitos (DELAMARO, 2016).

2.2.2.1 TÉCNICA FUNCIONAL

A técnica de teste funcional também conhecida como caixa preta é caracterizada por não levar em conta detalhes do código, somente as funcionalidades em questão impostas pelos requisitos. No teste funcional, os conjuntos de testes são elaborados com base nos requisitos e no momento da execução só é possível visualizar os dados de entrada e avaliar os dados de saída (MYERS; SANDLER; BADGETT, 2011).

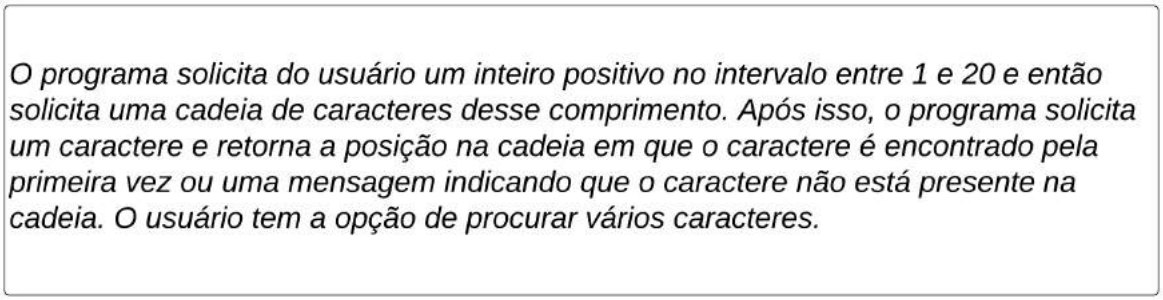
Uma das vantagens da técnica funcional e de seus critérios é de necessitar somente da especificação do produto para a criação dos requisitos de teste. Desse modo é possível aplicá-la em diversos tipos de programas, sejam eles orientado a objetos, procedimental ou componentes do programa.

Em contrapartida, um dos seus problemas é a falta de precisão nos requisitos de testes, devido à informalidade da especificação do programa como um todo, causando dificuldades com a automatização e o uso dos critérios de teste. Outro ponto a ser destacado, é que devido ao uso somente da especificação do programa como base para os critérios de testes, eles não podem assegurar que partes críticas do programa foram executadas durante o teste (ROPER, 1994). Os principais critérios do teste funcional são: Particionamento em Classes de Equivalência, Análise do Valor Limite, Grafo de Causa-Efeito e *Error-Guessing* (DELAMARO, 2016).

A seguir é apresentada e analisada a especificação de um programa chamado "Cadeia de Caracteres", retirada de Roper (1995), para a exemplificação dos critérios Análise de valor limite e Particionamento em classes de equivalência. Na Figura 2 é apresentada a especificação do programa “Cadeia de Caracteres”.

O particionamento de equivalência tem como objetivo tornar os dados de entradas

Figura 2: Cadeia de Caracteres.



Fonte: Roper (1995)

finitos e viáveis, dividindo os domínios de entrada em classes de equivalência. Uma vez que forem definidas as classes, assume-se que qualquer elemento dela que for usado é considerado uma representação dessa classe, pois o resultado dos elementos devem ser similares, ou seja, se um elemento encontra defeitos no programa, os outros devem encontrar também e caso não encontre defeitos os outros elementos também não devem encontrar (DELAMARO, 2016).

A classe de equivalência representa um conjunto válido e um conunto inválido de estados para as condições de entradas. Seguem-se as seguintes diretrizes para a definição das classes de equivalência (MYERS; SANDLER; BADGETT, 2011):

1. se for especificado um intervalo de valores pela condição de entrada, são definidas uma classe válida e duas inválidas;
2. se for especificado uma quantidade de valores pela condição de entrada, são definidas uma classe válida e duas inválidas;
3. se for especificado um conjunto com valores determinados pela condição de entrada, será definido uma classe válida para cada valor determinado e uma classe inválida com com outro valor qualquer; e
4. se for especificada uma situação na qual a entrada seja do tipo "deve ser desta maneira", são definidas uma classe válida e uma inválida.

Após a identificação das classes de equivalência, serão determinados os casos de testes.

Execução do exemplo

De acordo com a análise dos resultados de Delamaro (2016), exemplifica-se o critério de particionamento de equivalência. Para iniciar é necessário identificar as classes de equivalência, nas quais analisam-se as classes de entrada e saída do programa para particioná-las. Após essa primeira etapa de identificação, deve-se gerar os casos de testes com os dados.

As classes de entradas observadas foram:

Q - Quantidade de caracteres da cadeia a ser usada;

CA - *String* de caracteres a ser usado;

C - Caractere a ser procurado dentro da *string*; e

X - Opção para procurar mais de um caractere.

Como especificado anteriormente, as entradas CA e C não podem determinar classes de equivalência, pois podem ter quaisquer valores, enquanto as entradas Q e X conseguem determinar, em que o Q deve estar sempre no intervalo entre 1 e 20 e a entrada X sempre vai ser "Sim"ou "Não". Para a saída pode-se ter duas alternativas, a primeira seria a posição do caractere escolhido dentro da *string* e a segunda uma mensagem informando que o caractere informado não está presente na *string*. Após a análise das entradas e saídas, as classes de equivalência foram geradas conforme a Tabela 1.

Tabela 1: Classes de equivalência geradas a partir das entradas e saídas

Entrada	**Q**	**C**	**X**
Classes válidas de equivalência	1 <= Q <= 20	Caractere encontrado na string	Sim
Classes inválidas de equivalência	Q <1 e Q >20	Caractere não encontrado na string	Não

Fonte: Adaptada de Delamaro (2016)

Por fim, são gerados os casos de testes com os dados de entrada e saída como mostra a Tabela 2:

Tabela 2: Casos de testes gerados com o critério de particionamento de classes em equivalência .

Variáveis Entrada	Q	CA	C	X
Valores	40			
Saída	Quantidade de caracteres da string deve ser entre 1 e 20 .			
Valores	0			
Saída	Quantidade de caracteres da string deve ser entre 1 e 20.			
Valores	4	abcd	d	
Saída	Caractere está na posição 4 da string.			
Valores				s
Saída				
Valores			s	
Saída	Caractere s não está presente na string			
Valores				n
Saída				

Fonte: Adaptada de Delamaro (2016)

Esse critério é recomendado para programas nos quais as variáveis de entrada podem ser identificadas com facilidade e têm valores específicos, por outro lado, não é recomendado para programas com processamento de dados complexos, nos quais os valores das variáveis não são específicos (DELAMARO, 2016).

O critério de análise de valor limite é usado em conjunto com as classes de equivalência e tem como principal objetivo testar os valores limites de cada classe de equivalência. Enquanto o critério de particionamento em classes de equivalência usa valores aleatórios para teste, o critério de análise de valor limite usa valores máximo e mínimos sempre atingindo os limites das classes para o teste (MYERS; SANDLER; BADGETT, 2011). Um exemplo para isso seria uma classe de equivalência na qual é permitido somente valores entre 1 e 255, o critério de análise de valor limite usaria os valores 0 e 256 como entradas para testar os limites máximos e mínimos.

Segundo Myers, Sandler e Badgett (2011) algumas diretrizes podem ser levadas em conta para a criação dos dados de teste:

1. Se for especificada uma entrada com um intervalo de valores, devem ser definidos dados de teste explorando o limite desse intervalo e também dados de teste que

extrapolam o limite máximo e mínimo do intervalo;

2. Se for especificada uma condição de entrada com uma quantidade de valores, devem ser definidos valores únicos de entrada explorando o limite;

3. Usar a Diretriz 1 para condições de saída;

4. Usar a Diretriz 2 para condições de saída;

5. Se o valor da saída e da entrada forem conjuntos de valores ordenados, deve-se observar o primeiro e o último elemento com uma maior atenção; e

6. Usar o palpite para criar outras condições limites.

Execução do exemplo

Para a exemplificação do critério de análise de valor limite, usam-se as classes de equivalência criadas no exemplo anterior na Tabela 1. Após as classes de equivalência observadas foram gerados os casos de testes como mostra a Tabela 3:

Tabela 3: Casos de testes gerados com o critério de análise de valor limite.

Variáveis Entrada	Q	CA	C	X
Valores	0			
Saída	Quantidade de caracteres da string deve ser entre 1 e 20 .			
Valores	21			
Saída	Quantidade de caracteres da string deve ser entre 1 e 20.			
Valores	1	g	g	
Saída	Caractere está na posição 1 da string.			
Valores				s
Saída				
Valores			s	
Saída	Caractere s não está presente na string			
Valores	20	asdfghjkloiuytrewqat	t	
Saída	Caractere está na posição 20 da string.			

Fonte: Adaptada de Delamaro (2016)

2.2.2.2 TÉCNICA ESTRUTURAL

A técnica estrutural é também conhecida como caixa branca e utiliza a estrutura interna de um programa para gerar os casos de testes. A utilização da estrutura interna se dá por meio dos aspectos de implementação, essenciais para a geração dos casos de testes

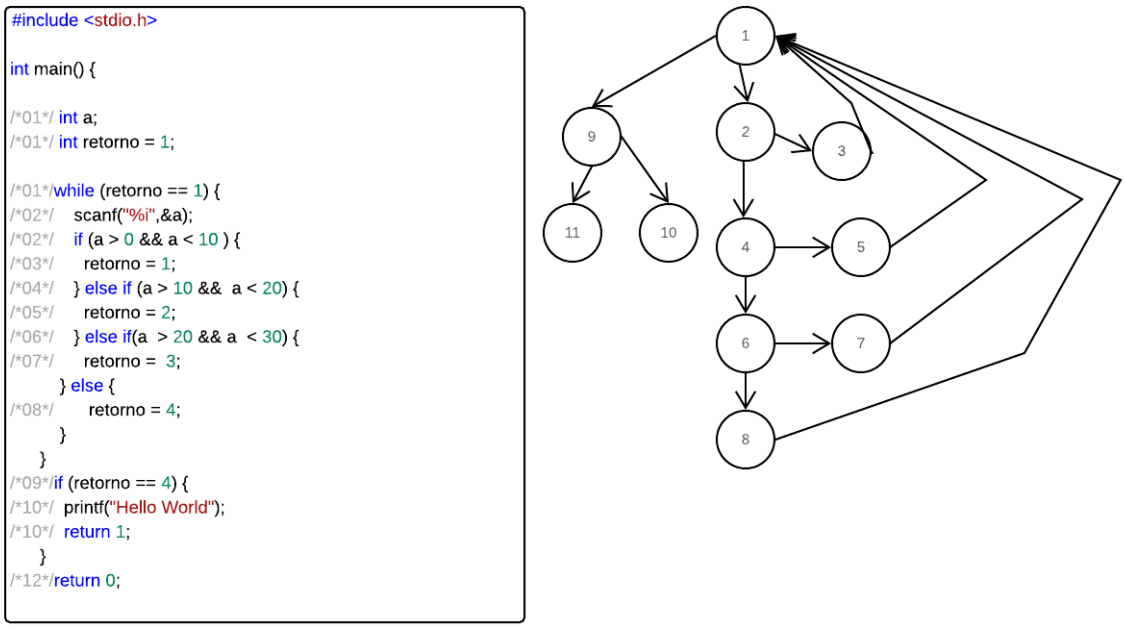
(MYERS; SANDLER; BADGETT, 2011). Nessa técnica os caminhos lógicos do programa são testados, avaliando desde condições, laços até o uso correto de variáveis.

Normalmente, a maioria dos critérios da técnica estrutural usam o GFC para fazer a representação do programa. Na representação, um programa denominado P é formado por um conjunto de blocos de comandos separados, no qual a execução do primeiro bloco resulta na execução de todos os outros blocos na ordem especificada.

Geralmente um programa P pode ser representado por GFC(G = (N,E,s)) que consiste na equivalência entre os vértices (nós) e blocos e também mostrar quais os fluxos de controle entre os blocos através das arestas (arcos). Em um GFC(G = (N,E,s)), N representa o conjunto de nós, E conjunto de arcos e s o nó de entrada (DELAMARO, 2016). GFC é considerado um grafo orientado, no qual cada aresta representa um fluxo alternativo de um bloco para o outro, cada nó representa um bloco único de comandos e existe somente um nó de entrada.

Um exemplo abaixo de um algoritmo genérico para demonstrar o GFC na Figura 3.

Figura 3: GFC exemplo de um algoritmo genérico.



```
#include <stdio.h>

int main() {

/*01*/ int a;
/*01*/ int retorno = 1;

/*01*/while (retorno == 1) {
/*02*/   scanf("%i",&a);
/*02*/   if (a > 0 && a < 10 ) {
/*03*/     retorno = 1;
/*04*/   } else if (a > 10 &&  a < 20) {
/*05*/     retorno = 2;
/*06*/   } else if(a  > 20 && a  < 30) {
/*07*/     retorno =  3;
       } else {
/*08*/      retorno = 4;
       }
   }
/*09*/if (retorno == 4) {
/*10*/  printf("Hello World");
/*10*/  return 1;
   }
/*12*/return 0;
```


Fonte: Autoria própria

No GFC da Figura 3 é possível notar uma sequência de comandos do programa

chamada bloco básico, no qual a execução de qualquer instrução do bloco acarreta na execução de todos os outros blocos. O bloco básico é composto por apenas um ponto de entrada e um ponto de saída (AMMANN; OFFUTT, 2016).

Os critérios de teste usados na técnica de teste estrutural podem ser classificados em: (I) critérios baseados na complexidade, (II) critérios baseados em fluxo de controle e, (III) critérios baseados em fluxo de dados.

Os critérios baseados em fluxo de controle levam em conta somente as informações do fluxo de execução do programa, como desvios e blocos de comandos para determinar a estrutura necessária. Os critérios mais conhecidos são *Todos-Nós* que ordena que todos os comandos sejam executados pelo menos uma vez, o *Todas-Arestas* que obriga a exercitação de todas as arestas pelo menos uma vez e por fim o *Todos-Caminhos* que solicita a execução de todos os possíveis caminhos do programa (DELAMARO, 2016).

Os critérios baseados na complexidade levam em conta a complexidade do programa para a criação dos requisitos de teste e um dos principais critérios dessa classe é o *McCabe* (MCCABE, 1976) que dispõe da complexidade ciclomática. A complexidade ciclomática é uma métrica que proporciona uma medida quantitativa do quanto está complexo a lógica de um programa (PRESSMAN, 2006).

Os critérios baseados em fluxo de dados levam em conta os fluxo de dados para a derivação dos requisitos de teste e uma de suas principais características é que elas necessitam das interações que envolvam definições de variáveis e referencias posteriores a essas definições (DELAMARO, 2016). Um dos motivos para a introdução dos critérios baseados em fluxo de dados, foi que o critério baseado em fluxo de controle não era eficaz o suficiente para demonstrar a presença de defeitos simples. Um dos resultado da introdução desse critério foi o estabelecimento da hierarquia entre os critérios Toda-Aresta e Todos-Caminhos.

Todas as técnicas abordadas, podem ser utilizadas tanto para softwares convencionais, quanto para aplicativos híbridos. A construção e utilização das técnicas não muda de acordo com o tipo de software.

2.2.3 AUTOMATIZAÇÃO DE TESTE

A automatização do teste de software é um termo que surgiu do crescimento das aplicações e sistemas desenvolvidos e também do grau de complexidade dos sistemas. Em alguns programas com testes específicos e complexos, o teste manual acaba se tornando

totalmente inviável devido ao tamanho e crescimento das aplicações, fazendo com que a única opção seja a automatização dos testes (MOLINARI, 2008).

Nesse contexto, em alguns programas o teste manual executado pelo testador não é vantajoso ao levar em conta a demanda e o custo necessário para testar. Um exemplo disso é a simulação do uso simultâneo de um determinado número de usuários em um programa para testar a capacidade de carga. A dificuldade em alocar e fazer com que vários testadores usem o sistema ao mesmo tempo torna o teste manual inviável e a automatização dos testes indispensável (BERNARDO, 2008).

Em geral, os testes automatizados são viabilizados por ferramentas que têm como principal objetivo exercitar o sistema por meio de dados de entrada, testando as funcionalidades do programa e verificando se estão de acordo com o solicitado e especificado (BERNARDO, 2008).

As principais vantagens de automatizar os testes em programas de grande escala é o menor tempo na execução dos testes, uma melhor qualidade para o software e casos de testes mais elaborados. Uma outra grande vantagem é a possibilidade de repetir os testes automatizados a qualquer momento do desenvolvimento do software, tanto em correções de funcionalidades antigas, quanto no momento de inserir novas funcionalidades (BERNARDO, 2008).

2.3 DESENVOLVIMENTO ÁGIL E INTEGRAÇÃO CONTÍNUA

A Integração Contínua (IC) é uma prática usada em equipes de desenvolvimento de software para manter um repositório de código único e correto. Na IC os desenvolvedores mantêm o código em um único repositório, no qual devem ser realizadas integrações de seu trabalho, no mínimo uma vez ao dia. A cada integração feita pelo desenvolvedor, uma bateria de testes é executada com o intuito de verificar se as modificações estão de acordo e o programa continua funcionando corretamente. Muitas equipes acreditam que o desenvolvimento de software usando esta prática reduz consideravelmente a quantidade de problemas no momento da integração, fornecendo mais rapidez e gerando programas mais coesos (FOWLER, 2006).

A IC é uma atividade que visa a qualidade de software dentro do desenvolvimento de software, juntamente com outras atividades, como o teste de software. Os benefícios trazidos ajudam a complementar as demais atividades de qualidade de software e também muitas vezes suprir uma falta delas (STÂHL; BOSCH, 2014).

Entretanto, para realizar a implantação da IC dentro de uma equipe de desenvolvimento, levam-se em conta as características de como a equipe trabalha e também o ambiente em que está inserido. Fowler (2006), com base em suas experiências, apresenta alguns pontos que devem ser levados em conta para a IC se tornar eficaz. Esses pontos são apresentados a seguir.

- A equipe deve manter um único repositório de código, no qual possa fazer o controle de versão dos artefatos do programa para realizar a *build*[1] do sistema. Toda a equipe de desenvolvimento deve ter conhecimento deste repositório;
- A equipe deve automatizar a *build* do programa, para facilitar a manipulação dos arquivos e artefatos para compilação do programa. A automatização dessa *build*, deve ocorrer em ambientes diferentes daqueles usados pelos desenvolvedores no dia a dia;
- Tornar a *build* automatizada no quesito testes de software. Deve-se implementar uma bateria de testes que seja executada automaticamente a cada integração de novas funcionalidade. O *feedback* das falhas deve ser imediato, aumentando assim a rapidez da equipe quanto à correção do problema;
- Enviar modificações novas todos os dias. A equipe deve criar hábitos de envios de modificações várias vezes ao dia. Quanto menor a quantidade de modificações enviadas, mais fácil será a rastreabilidade caso ocorra algum erro com a *build* do programa;
- Cada modificação nova deve ser integrada com o repositório principal. A modificação nova só pode ser integrada caso a *build* seja realizada com sucesso;
- Manter o processo de *build* rápido. O tempo para a execução do *build* do programa deve ser curto, de modo que o foco maior seja na execução dos testes automatizados, visando encontrar *bugs* e ter um *feedback* dos erros mais rapidamente;
- Testar as modificações em um ambiente clone ao de produção. Esse ambiente deve ser configurado igual, se possível, ao ambiente de produção usado. Esses testes são fundamentais para manter a integridade entre ambientes e para que não ocorram problemas inesperados ao tentar integrar as modificações com o ambiente de produção;

[1] Versão compilada do sistema

- Manter todos da equipe com o conhecimento sobre o estado do programa. A equipe deve saber sempre que ocorram falhas com integrações novas e também qual o motivo da falha estar ocorrendo; e
- A implantação do sistema deve ser automatizada. Em qualquer ambiente seja ele o de produção ou de testes, o programa deve ser implantado com facilidade sem ocorrer nenhum problema.

Na IC, uma das principais atividades que se beneficiam desse modelo de integração são os testes automatizados. Os testes automatizados podem ser integrados junto com a *build* do sistema e a cada interação nova de funcionalidades eles são executados para verificar se existem inconsistências. Ao encontrar problemas, a *build* falha e não deixa integrar essas novas funcionalidades com as antigas.

2.4 DESENVOLVIMENTO *MOBILE*

O desenvolvimento de aplicações *mobile* a cada dia torna-se uma oportunidade nova no mercado de negócios, levando em conta o crescimento do uso dos *smartphones* no mundo. As necessidades de aplicações para os *smartphones*, abrem oportunidades para uma diversidade de aplicativos, desde entretenimento até os mais focados em grandes corporações.

Nos aplicativos *mobile* existem diversificações em como desenvolve-los e também em plataformas eles podem ser usados. Sobretudo, existem três tipos de aplicativos *mobile*: Aplicativos nativos, Aplicativos híbridos e os Aplicativos web (CHEDE, 2016). As subseções a seguir visam apresentar cada um desses tipos.

2.4.1 APLICATIVOS NATIVOS

Os aplicativos nativos podem ser encontrados em lojas de aplicativos oficiais, geralmente são desenvolvidas com linguagens de programações nativas e específicas para determinados sistemas operacionais, como Java[2] para Android[3], *Objective-C* para iOS[4] ou C para Windows Phone[5]. As funcionalidades nativas como câmera, *Bluetooth* entre outras, são disponibilizadas por meio de *SDKS* para os desenvolvedores usarem (FOLLOW, 2013; CHEDE, 2016; HEITKÖTTER; HEIERHOFF; MAJCHRZAK, 2013).

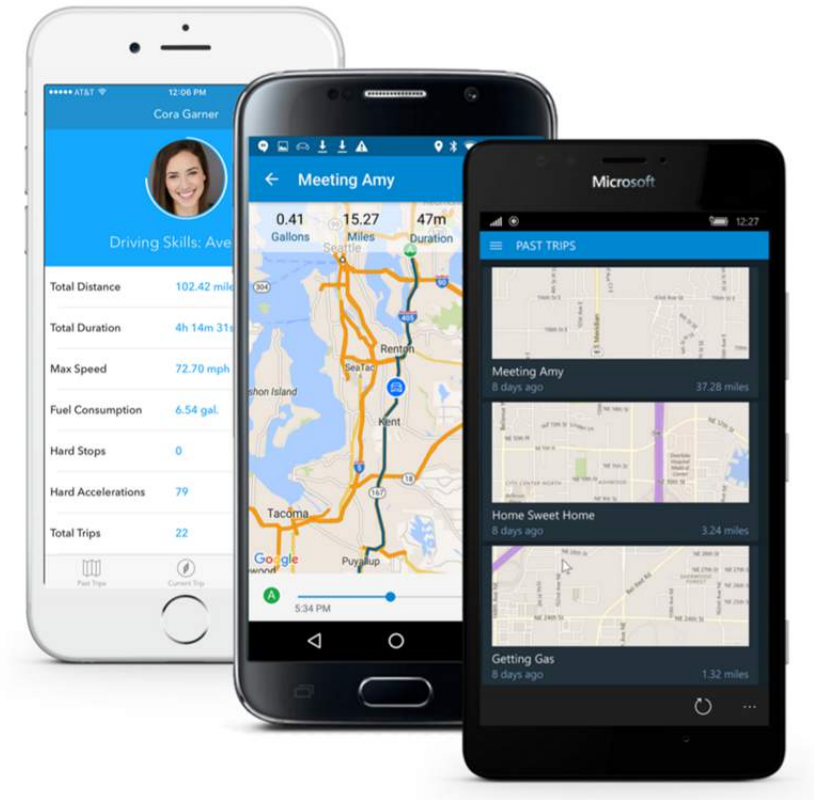
[2]acesse: `https://www.java.com/en/`
[3]acesse: `https://www.android.com/`
[4]acesse: `https://www.apple.com/br/ios/ios-12/`
[5]acesse: `https://www.microsoft.com/pt-br/windows/phones`

Por outro lado, como os aplicativos são desenvolvidos com linguagens específicas para determinados sistemas operacionais, é necessário implementar diversas vezes o mesmo aplicativo para atingir todo o mercado de celulares. Com isso, o desenvolvimento fica demorado e repetitivo, resultando em gastos maiores de recursos humanos e financeiros. Outro ponto a ser destacado é o aprendizado por parte da equipe de programação de diversas linguagens de programação para a implementação do aplicativo em todos os sistemas operacionais desejados (FOLLOW, 2013).

As aplicações nativas são úteis quando trata-se de desempenho e otimização, alguns aplicativos dependem de grandes processamentos, tanto de dados como de imagens. Um exemplo para isso é uma aplicação com processamento de gráficos interativos de ponta (FOLLOW, 2013).

Na Figura 4 pode-se observar três tipos de aplicações, uma no sistema operacional Android, uma no iOS e por fim no Windows Phone. Para cada sistema operacional, necessita-se de uma implementação única em linguagens de programação diferentes.

Figura 4: Aplicativos nativos.



Fonte: aes (2016)

2.4.2 APLICATIVOS HÍBRIDOS

Na criação de aplicativos nativos para diversos sistemas operacionais faz-se necessário o uso de diversos *frameworks* e linguagens de programação. Uma aplicação híbrida consegue suprir essa necessidade apenas com a linguagem Web baseada em padrões nativos. As aplicações do tipo híbridas são denominadas uma categoria especial de aplicações web, pois conseguem ampliar o ambiente de uma aplicação que está executando na web para determinados dispositivos e sistemas operacionais, fazendo com que o mesmo aplicativo consiga executar tanto na web por meio de computadores quanto *mobile* por dispositivos móveis (FOLLOW, 2013; HEITKÖTTER; HEIERHOFF; MAJCHRZAK, 2013).

Os recursos nativos que predominam nos aplicativos nativos, também são usados nos híbridos por meio de *Application Programming Interface* (API) acessada por JavaScript[6]. Os desenvolvedores conseguem beneficiar-se da possibilidade de acessar os recursos nativos dos dispositivos, pois assim existe um equilíbrio maior entre os melhores recursos nativos e os melhores recursos web (FOLLOW, 2013).

No desenvolvimento dos híbridos, usa-se uma combinação de *Hypertext Markup Language*, versão 5 (HTML5), *Cascading Style Sheets* (CSS), JavaScipt e SDKs específicas de cada dispositivo e sistema operacional para a escrita dos códigos fontes. O pacote final dos códigos fontes é composto por todos os recursos necessários (JavaScript, CSS, HTML5) para executar a aplicação instantaneamente como se fosse um aplicativo nativo, sem necessitar de uma resposta de um servidor web.

Para os desenvolvedores, a abordagem híbrida fornece algumas vantagens (FOLLOW, 2013):

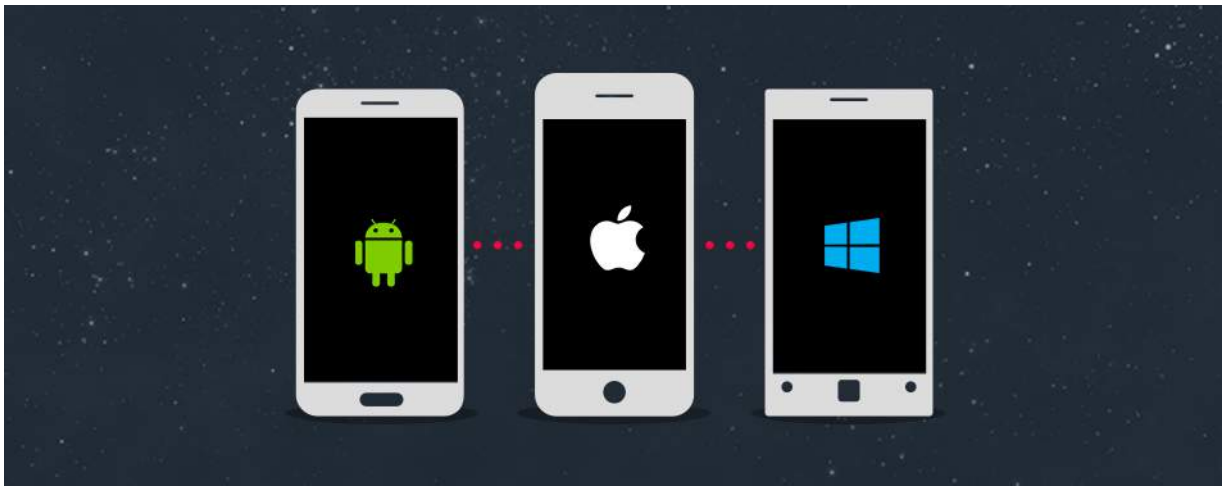
- A atualização, remoção ou adição de conteúdo no aplicativo pode ser feita sem a necessidade de solicitar ao usuário que atualize o aplicativo;
- A parte visual do aplicativo se torna mais genérica, sendo uma só para todas as plataformas, o que faz com que a produtividade dos programadores aumente.

Abaixo a Figura 5 representa que a mesma aplicação com somente um código, pode ser instalada e executada em diversos sistemas operacionais mobile.

Atualmente no mercado de aplicativos, existem diversas empresas que desenvolvem

[6]acesse: `https://www.javascript.com/`

Figura 5: Aplicativos híbridos.



Fonte: aes (2016)

aplicativos híbridos. Um deles muito utilizado é o aplicativo da organização financeira Nubank®.

2.4.3 APLICATIVOS WEB

Aplicativos web são desenvolvidos utilizando tecnologias HTML, CSS e JavaScript e costumam ter a aparência e o comportamento de páginas web comuns. Os aplicativos web não conseguem usar os recursos nativos dos dispositivos, como câmera, GPS, sensores e muitos outros, se comparado com os híbridos e os nativos, o que os tornam mais limitados (HEITKÖTTER; HEIERHOFF; MAJCHRZAK, 2013).

Esse tipo de aplicativo não é considerado nativo e raiz, na realidade são sites que podem ser abertos pelos navegadores dos dispositivos por meio de uma *Uniform Resource Locator*(URL). Ao estabelecer um cenário comparativo dos aplicativos nativos com os híbridos não é necessário nenhuma instalação, tornando-se mais leve e consumindo menos memória do dispositivo (TAVARES, 2016).

Uma forma de desenvolver os aplicativos web é por meio de um conceito novo conhecido como *Progressive Web Aps (PWA)*, que foi introduzido pelo Google[7] em 2015. Essa forma de desenvolver utiliza tecnologias modernas e novas para suprir a limitação dos poucos recursos se comparado com os aplicativos híbridos e nativos. Adicionam-se novos recursos como notificações e a possibilidade de consumir conteúdo enquanto estiver *offline* (WAHLSTRÖM, 2017).

[7] `https://www.google.com.br/`

2.5 TESTES QUEBRADIÇOS

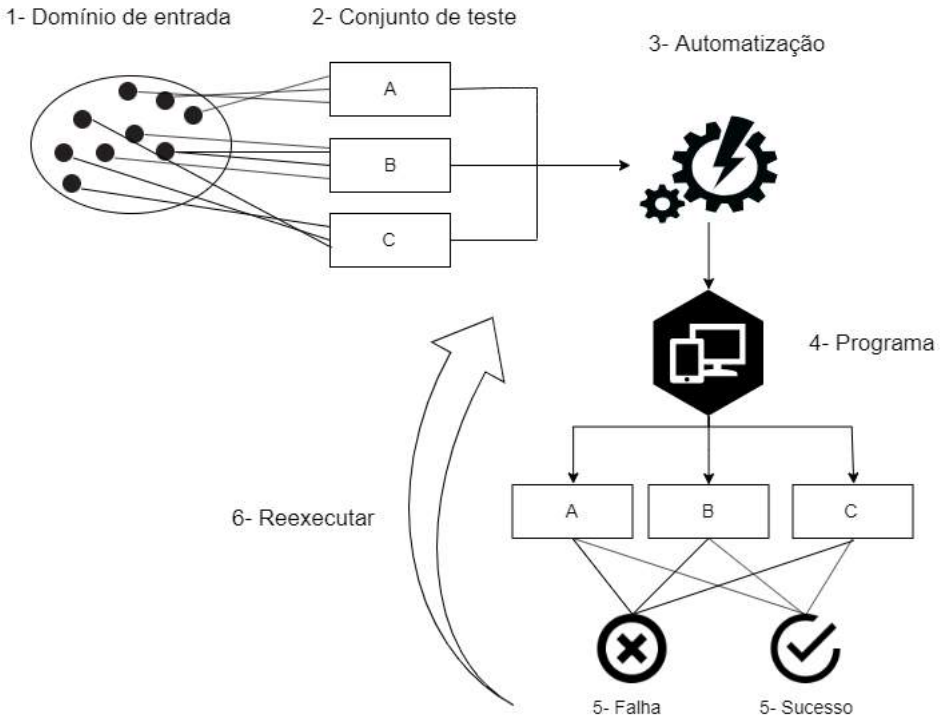
Em muitos projetos de desenvolvimento de software, tarefas novas de desenvolvimento, que são integradas com as antigas, requerem baterias de testes para garantir que a funcionalidade está correta. Diante disso, a aprovação e reprovação de cada teste executado tem um papel importante para a equipe saber se o programa está funcionando perfeitamente (MICCO, 2016). Um ambiente muito utilizado para a execução dessas baterias de testes a cada interação de novas funcionalidade é a prática de IC, abordada na Seção 2.3.

Na execução dos testes, um teste pode se tornar quebradiço cujo resultado seja incerto, ou seja, em alguns momentos é dado como sucesso (*pass*) sem erros, mas em outras execuções aleatoriamente ele é rejeitado e interrompido por falhas (*fail*) (THORVE; SRESHTHA; MENG, 2018; GEORGE, 2014). Geralmente, as causas dessas reprovações podem variar desde lógica de programação e concorrência, até falhas de interação, podendo causar uma dúvida para a equipe do projeto e um retrabalho desnecessário (THORVE; SRESHTHA; MENG, 2018; LUO et al., 2018). Independente do motivo, é senso comum que a maioria dos projetos de software tendam a sofrer com testes quebradiços. Atualmente diversas organizações como [8]Google®, [9]Netflix® e [10]Microsoft® já relataram (MICCO, 2016; MEMON, 2017; HERZIG; NAGAPPAN, 2015) que sofrem com os testes quebradiços. Em muitas delas o termo utilizado para representar esse tipo de problema pode ser representado de outras formas (teste indeterminado, teste inconsistente, testes instáveis, entre outros).

Abaixo, na Figura 6, pode-se ver um exemplo de um fluxo com testes de software automatizados sendo executados dentro de um programa e podendo ter seus resultados incertos. Na Fase 1 têm-se o domínio de entrada que será usado para criar os conjuntos de testes A, B e C. Na Fase 2 têm-se os conjuntos de testes gerados A, B e C que serão automatizados por uma ferramenta na Fase 3. Na Fase 3 os conjuntos de testes são transformados em testes automatizados. Na Fase 4 os testes automatizados são executados dentro de um programa em específico. Por fim, na Fase 5 o resultado dos testes pode ser sucesso ou falha. O fluxo pode ser reexecutado na Fase 6, voltando a partir da Fase 3, já que após a automatização dos testes, eles podem ser executados novamente a qualquer momento. No fluxo apresentado, os testes quebradiços surgem após o resultado de diversas execuções do mesmo teste variar, em determinados momento ele apresenta sucesso e outros apresenta falha.

[8]acesse: `https://about.google/intl/pt-BR/products/`
[9]acesse: `https://www.netflix.com/`
[10]acesse: `https://www.microsoft.com/pt-br`

Figura 6: Fluxograma de execução de testes automatizados.



Fonte: Autoria própria

Durante a ocorrência dos testes quebradiços, várias hipóteses sobre o teste podem ser levantadas. Em alguns casos, os resultados dos testes tornam-se muito confusos o que se torna uma dúvida para os integrantes da equipe do projeto o que leva eles a melhorar a estrutura interna dos testes sem alterar o comportamento externo. Para os desenvolvedores e testadores, testes quebradiços são um problema em potencial, haja vista que o retrabalho pode ser inútil quando a causa do teste é incerta. Por outro lado, essa incerteza pode esconder algum problema grave para o projeto.

Com o surgimento dos testes quebradiços, alguns estudos e pesquisas sobre projetos de variados tipos foram realizados para tentar descobrir as causas e também como resolver essas situações (BELL OWOLABI LEGUNSEN, 2018; LUO et al., 2018; THORVE; SRESHTHA; MENG, 2018; GAO, 2017). Alguns *frameworks* também já relatam os testes quebradiços e criaram suas próprias anotações dentro das ferramentas (TOMLIN, 2019; DEVELOPERS,). Esforços e pesquisas também promoveram uma categorização quanto às causas dos quebradiços (THORVE; SRESHTHA; MENG, 2018; LUO et al., 2018; BELL OWOLABI LEGUNSEN, 2018). São elas: *(I)* espera assíncrona; *(II)* concorrência, *(III)* ordem de execução dos testes, *(IV)* rede, *(V)* tempo,*(VI)* vazamento de dados,*(VII)* operações de ponto flutuante, *(VIII)* lógica de programação, *(IX)* interface de usuário,

(X) operações de entrada e saída, *(XI)* dados gerados aleatoriamente, *(XII)* coleções de dados não ordenadas, *(XIII)* causas não identificadas.

Na Tabela 4 podem ser observados exemplos de causas dos testes quebradiços de acordo com suas categorias e também quais métodos foram usados para resolver a ocorrência do teste quebradiço retirados de (THORVE; SRESHTHA; MENG, 2018). Na primeira linha, por exemplo, podem ser observadas as ocorrências relacionadas a causa de concorrência e também como foi resolvido em cada tipo de categoria de solução.

Tabela 4: Distribuição de 77 commits entre as causas e estratégias de resolução de testes quebradiços.

Estratégia / Causa	Melhorar a implementação (53)	Substituir implementação (6)	Reexecutar (4)	Modificar asserções (4)	Remover teste (10)
Concorrência (28)	26	-	-	2	-
Lógica de programação (9)	4	5	3	1	4
Dependência (17)	8	-	-	1	-
Rede (6)	5	1	-	-	-
Difícil de classificar (11)	6	-	-	-	-
Design (6)	4	-	1	-	6

Fonte: Thorve, Sreshtha e Meng (2018)

Na Figura 7 pode-se observar a resolução de um teste quebradiço originalmente formado por um problema de lógica de programação. No exemplo do código, o teste quebradiço ocorria devido a um lançamento de uma exceção mal posicionada que fazia o sistema para inesperadamente. Para resolver, ao invés de executar uma exceção, o código foi alterado para somente guardar a exceção e não parar o sistema.

Figura 7: Exemplo de resolução para a categoria Lógica de programação.

```
1 } catch (IOException e) {
2     Util.closeQuietly(MockSpdyPeer.this);
3     throw new RuntimeException(e);
4     e.printStackTrace();
5 }
```

Fonte: aes (2016)

2.5.1 RELATOS DA INDÚSTRIA

Em muitos projetos, já relatam-se os testes quebradiços de forma direta ou indireta e confirmam-se que eles são prejudiciais para toda a equipe (MICCO, 2016; FOWLER, 2011; ZHANG et al., 2014). Principalmente para os desenvolvedores no âmbito de refatoração de código e também para os testadores validarem se os casos de testes são válidos e as funcionalidades realmente estão funcionando de acordo com o esperado. Os testes quebradiços muitas vezes são reportados como falhas e requerem novas execuções, o que significa mais recursos gasto. A identificação de testes quebradiços em ambientes de desenvolvimento encarece custos de projeto e atrasa cronogramas previamente estabelecidos.

Na organização Google, os casos de testes que falham são executados pelo menos dez vezes, e se após a falha ele for aprovado, rotula-se o caso de teste como teste quebradiço (GUPTA; PENIX, 2011). Alguns relatos como o de Micco (2016), afirmam que 1,5% de todas as execuções dos casos de testes elaborados por eles têm um resultado como sendo teste quebradiço. Afirma-se também que 16% dos testes de software internos do Google tem falhas que não envolvem códigos de teste e tão pouco códigos de programas, sendo considerados como testes quebradiços. Outras organizações renomadas do setor de TI como, por exemplo, a Netflix® [11](MEMON, 2017) e Microsoft® [12](HERZIG; NAGAPPAN, 2015) também sofrem com os testes quebradiços.

No Capítulo 3 são apresentados os esforços de pesquisas encontrados por meio de uma Revisão Integrativa. A revisão integrativa utilizada visa reunir todas as pesquisas relacionadas ao tema do trabalho e também apontar os principais resultados e sínteses obtidas por meio da execução da revisão.

[11] acesse:https://www.netflix.com
[12] aesse:https://www.microsoft.com/pt-br

3 REVISÃO INTEGRATIVA

Este capítulo apresenta a Revisão Integrativa (RI) da literatura que foi elaborada para a criação do presente trabalho. Optou-se por fazer uma RI devido à falta de tempo para conduzir uma revisão sistemática. A RI é considerada uma revisão mais criteriosa se comparada com uma revisão narrativa e pode ser replicada para diversos setores de pesquisas.

A RI tem como principal objetivo resumir o conhecimento científico construindo uma análise ampla da literatura e contribuindo com discussões e resultados para futuros trabalhos de um determinado tópico de pesquisa (SASSO et al., 2008).

A RI consiste na divisão da pesquisa em seis etapas (SASSO et al., 2008):

- Primeira etapa: Identificação do tema e seleção da hipótese ou questão de pesquisa para a elaboração da revisão integrativa;
- Segunda etapa: Estabelecimento de critérios para inclusão e exclusão de estudos/ amostragem ou busca na literatura juntamente com a *string* de busca;
- Terceira etapa: Definição e absorção das informações a serem extraídas dos estudos selecionados;
- Quarta etapa: Avaliação e categorização dos estudos incluídos na RI;
- Quinta etapa: Interpretação dos resultados; e
- Sexta etapa: Apresentação da revisão e síntese do conhecimento.

O objetivo dessa revisão é identificar os principais esforços de estudos relacionados aos testes quebradiços, desde o surgimento dos testes quebradiços em projetos de software, suas causas, soluções possíveis e também ferramentas criadas para tratar os testes quebradiços. Um objetivo secundário é buscar o máximo de trabalhos possíveis sobre os testes quebradiços, possibilitando uma análise histórica dos trabalhos já elaborados.

3.1 IDENTIFICAÇÃO DO TEMA

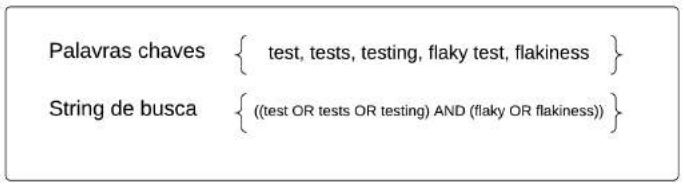
O trabalho proposto analisa um problema conhecido na área de teste de software como teste quebradiço. Para identificar os estudos nessa área, analisam-se não somente os testes quebradiços nos aplicativos híbridos, mas também qualquer outra contribuição relacionada com esse tema.

A primeira etapa é considerada uma das mais importantes para a RI devido a sua importância para a execução e finalização com sucesso da revisão (SASSO et al., 2008). Para realizar a revisão, primeiramente, foi criada uma questão de pesquisa que é usada para nortear a pesquisa quanto aos trabalhos pesquisados. No presente trabalho, a questão de pesquisa foi definida como:

QP: Quais as contribuições existentes na literatura para a identificação dos testes quebradiços e também identificação de suas causas e/ou soluções dos testes quebradiços .

Após a definição da questão de pesquisa, as palavras chaves foram identificadas e usadas para compor uma *string* de busca usada para buscar os trabalhos relacionados. As palavras chaves identificadas foram traduzidas para o idioma inglês devido à insuficiência de trabalhos no idioma português. As palavras chaves identificadas foram *test*, *tests*, *testing*, *flaky test*, *flakiness*. Por fim, a string de busca criada foi "((*test* OR *tests* OR *testing*) AND (*flaky* OR *flakiness*))"conforme a Figura 8.

Figura 8: *String* de busca.



Fonte: Autoria própria

3.2 ESTABELECIMENTO DE CRITÉRIOS

As fontes de informações escolhidas para a busca dos trabalhos relacionados foram: [1]IEEE Xplore e [2]ACM Digital Library.

[1] acesse:https://ieeexplore.ieee.org/Xplore/home.jsp
[2] acesse:https://dl.acm.org/

A seleção dos estudos retornados é uma etapa essencial para uma revisão ser considerada válida e consistente. Dessa forma para identificar a relevância e selecionar os estudos foram definidos os seguintes Critérios de Inclusão e Exclusão:

- **Critérios de Inclusão (CI)**

  1. Estudos que forneçam dados sobre a ocorrência dos testes quebradiços;
  2. Estudos que forneçam dados sobre as causas dos testes quebradiços;
  3. Estudos que forneçam dados sobre as possíveis soluções dos testes quebradiços; e
  4. Estudos sobre as ferramentas criadas para prevenir e corrigir os testes quebradiços.

- **Critérios de Exclusão (CE)**

  1. Estudos que são duplicados;
  2. Estudos disponíveis apenas sob a forma de resumos;
  3. Estudos não relacionados às questões de pesquisa; e
  4. Estudos escritos em linguagem diferente do inglês.

A amostra inicial de resultados obtidos constituiu-se de 31 artigos identificados na busca sendo: 21 artigos (IEEE); 10 artigos (ACM). As bases de dados e a *string* de busca usadas estão descritas abaixo na Tabela 5 juntamente com a quantidade de trabalhos retornados.

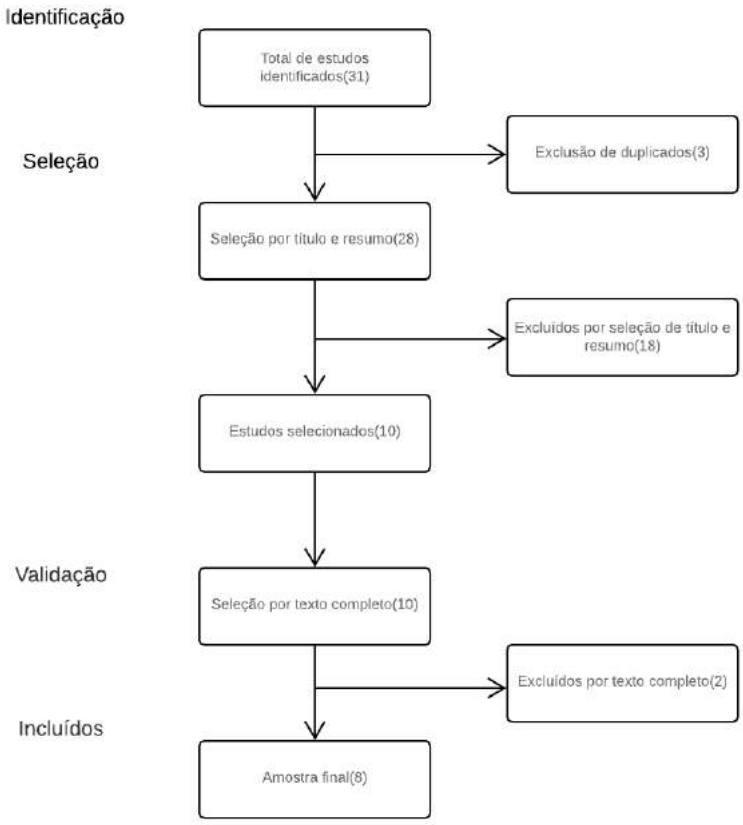
Tabela 5: Resultados obtidos durante a busca nas fontes de informações.

Fontes de informação	String de busca	Resultados
IEEE	((test OR tests OR testing) AND (flaky OR flakiness))	21
ACM	((test OR tests OR testing) AND (flaky OR flakiness))	10

Fonte: Autoria própria

O processo de leitura dos dados ocorreu primeiramente pela leitura textual do título e resumo de cada trabalho retornado nas buscas. Por meio dos critérios de inclusão e exclusão realizou-se a seleção dos estudos primários, que incluiu-se dez estudos. Por fim realizou-se uma leitura completa de todos os trabalhos e apenas oito trabalhos foram

Figura 9: Processo de seleção dos trabalhos.



Fonte: Autoria própria

incluídos e definidos como primários. O processo de seleção dos artigos é apresentado na Figura 9 .

Durante a análise dos resumos dos trabalhos da amostragem inicial foram excluídos 21 trabalhos que não eram relacionados com o tema ou só citavam as palavras chaves no meio do texto. Após a leitura na íntegra desses estudos somente oito deles atenderam aos critérios de inclusão e responderam a questão de pesquisa norteadora do presente trabalho.

Tabela 6: Trabalhos selecionados no processo da RI

Fontes de informação	Artigos encontrados	Artigos selecionados
IEEE	21	6
ACM	10	2

Fonte: Autoria própria

3.3 EXTRAÇÃO DE DADOS

Para analisar os estudos selecionados, elaborou-se um quadro informativo para a coleta das informações principais para identificação de cada trabalho. Os dados foram sintetizados de forma organizada com base na leitura completa dos artigos primários. Os itens identificados para o quadro foram: identificação do estudo, autores, base de dados, ano publicação e país de filiação do primeiro autor. Após o levantamento de dados inicias, coletaram-se as informações de objetivos, metodologias e resultados de cada trabalho.

Abaixo são apresentadas as informações sintetizadas de cada trabalho avaliado de acordo com o seu identificador na Tabela 7.

O trabalho de Palomba e Zaidman (2017) tem como objetivo verificar se os tipos de *tests smells*(mal cheiro em testes) podem representar a causa dos testes quebradiços e também verificar quais medidas de refatoração são necessárias para corrigir esse problema. Os objetivos foram divididos em três etapas, no qual a primeira etapa consiste em identificar as causas para os testes quebradiços, a segunda visa estudar a relação dos *tests smells* com os testes quebradiços e por fim a última etapa mede capacidade de refatorar os *tests smells* corrigindo os testes quebradiços. Foram avaliados commits de vários projetos Apache com testes em JUnit para a análise dos *tests smells* e a relação com os testes quebradiços. Para a Etapa 1 foram encontrados 8.829 testes quebradiços sobre o total de 19.532 métodos de teste JUnit analisados. Assim, 45% dos testes sofrem de testes quebradiços. Para a Etapa 2 observou-se que 61% dos métodos que são considerados *tests smells* são quebradiços também, e que também 54% dos casos, foram os *tests smells* que ocasionaram os testes quebradiços, sendo que as principais causas são concorrência, dependência entre testes e I/O (entrada e saída) 8. Para a última etapa aplicou-se a refatoração dos 54% de mal cheiro de testes que ocasionaram em testes quebradiços e com isso conseguiu-se resolver todos os testes quebradiços, provando que a refatoração dos mal cheiro de testes pode ajudar a resolver os testes quebradiços.

O trabalho de King Dionny Santiago (2018) tem como objetivo avaliar quais são os fatores relacionados para a ocorrência dos testes quebradiços. Para isso um aprendizado de máquina foi elaborado, com uma rede *bayesiana* baseada nos fatores das causas dos testes quebradiços. Caso seja bem sucedido, esse aprendizado de máquina serviu como um classificador de testes quebradiços e pode ser usado em vários projetos, principalmente aqueles que usam a integração contínua. Para isso foram criadas três etapas, a primeira é a avaliação de quais são os sintomas que causam os testes quebradiços, depois é a criação

Tabela 7: Informações primárias identificadas de cada artigo.

Id	Título	Autores	Base	Ano	País
1	Does Refactoring of Test Smells Induce Fixing Flaky Tests?	Fabio Palomba, Andy Zaidman	IEEE XPLORE	2017	Holanda
2	Towards a Bayesian Network Model for Predicting Flaky Automated Tests	Tariq M. King, Dionny Santiago, Justin Phillips, Peter J. Clarke	IEEE XPLORE	2018	Estados Unidos
3	DeFlaker: Automatically Detecting Flaky Tests	Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, Darko Marinov	IEEE XPLORE	2018	Estados Unidos
4	An Empirical Study of Flaky Tests in Android Apps	Swapna Thorve, Chandani Sreshtha, Na Meng	IEEE XPLORE	2018	Estados Unidos
5	Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications	August Shi, Alex Gyori, Owolabi Legunsen, Darko Marinov	IEEE XPLORE	2016	Estados Unidos
6	Which of My Failures are Real? Using Relevance Ranking to Raise True Failures to the Top	Zebao Gao, Atif M. Memon	IEEE XPLORE	2015	Estados Unidos
7	An empirical analysis of flaky tests	Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, Darko Marinov	ACM	2014	Estados Unidos
8	Measuring the cost of regression testing in practice: a study of Java projects using continuous integration	Adriaan Labuschagne, Laura Inozemtseva, Reid Holmes	ACM	2017	Canadá

Fonte: Autoria própria

de uma ferramenta para coletar os dados com base em métricas e por fim a criação um conjunto de dados de exemplos com testes que são quebradiços e com outros que não são para treinar o classificador. Para o experimento foram usadas cinco equipes de testes, onde cada uma teve entre 150 a 1000 casos de testes e uma taxa de aprovação nas execuções dos testes entre 20% a 95%. Essas equipes usaram as ferramentas de *TeamCity* para integração contínua e o Eco para desenvolver os testes de sistemas. Dados de uma equipe foram usadas para análise dentro de um período de execução de teste de um mês. Foi analisado uma base de testes dessa equipe com 714 casos de testes e com 26.298 execuções de testes. Os resultados permitiram observar que 65,7% de sucesso quanto à previsão dos testes quebradiços usando o aprendizado de máquina, através da identificação e remoção dos casos de testes para a quarentena.

O trabalho de Bell Owolabi Legunsen (2018) tem como objetivo a criação de uma ferramenta denominada *Deflaker* que tem como função o *Rerun*, bastante conhecido no cenário dos testes quebradiços, pois executa todos os testes novamente para verificar se realmente é testes quebradiço. O diferencial do *Deflaker* para o *Rerun* é a cobertura de testes usadas, enquanto o *Rerun* leva em conta toda a cobertura de código que está sendo testada, o Deflaker analisa somente a parte do código que foi alterada pela última vez. O experimento com o Deflaker foi realizado em dois ambientes separados. O primeiro é um ambiente conhecido como histórico, no qual leva-se em conta projetos que já foram realizados, avaliando os resultados caso fossem avaliados os commits com o Deflaker durante o desenvolvimento. O outro ambiente de projeto usado é um ambiente real, em que o projeto está sendo implementado e a ferramenta proposta é usada no mesmo ambiente que os desenvolvedores do projeto. O ambiente real pode ter um alcance maior em questão de quantidade de projetos, em contrapartida o ambiente histórico consegue alcançar menos pelo fato de ter que procurar os projetos em bases de dados e configurá-los para os testes. O Deflaker em sua execução relatou 4,846 testes quebradiços dentro dos testes executados no ambiente real e no ambiente histórico foram 87 testes quebradiços.

O trabalho de Thorve, Sreshtha e Meng (2018) tem como foco principal o estudo sobre os testes quebradiços ocorridos e quais estratégias de correção foram usadas em aplicativos *android*, por meio da avaliação dos *commits* de alguns projetos do [3]Github. Essa avaliação foi segmentada em duas partes: Filtragem e Análise. Para a filtragem primeiramente realizou-se uma busca de projetos *android* em várias linguagens de programação, posteriormente usou-se os projetos para encontrar os *commits* com o parâmetro “flaky” e “intermittent”. Após isso uma busca e separação manual foi realizada pela equipe com

[3]`acesse:https://github.com/`

um resultado de 77 *commits* de 29 projetos coletados para a análise. Para o processo de análise filtraram-se os *commits* por meio de dois parâmetros: Se era irrelevante para testes quebradiços ou, então, se ocorria testes quebradiços mas não continham nenhuma alteração de correção do mesmo. Os resultados obtidos pela equipe de pesquisa com base nas causa dos testes quebradiços foram classificados em seis categorias: Concorrência, Dependência, Lógica de programação, Rede, UI (do inglês User Interface), outros. Foram encontrados 36% (22 de 77) dos *commits* com a causa de concorrência, 22% (17 de 77) de dependência, 12% (9 de 77) de lógica de programação, 8% (6 de 77) de rede e UI, e por fim 11 dos 17 *commits* não foram identificadas as causas.

O trabalho de Shi Alex Gyori (2016) tem como objetivo propor uma técnica chamada *NONDEX* para detectar os testes quebradiços com o código ADINS que refere-se ao código que pressupõe uma implementação determinística de um método com uma especificação não determinística. Esta técnica foi desenvolvida em Java e pode ser generalizada para outras linguagens de programação. Para o teste do *NONDEX* foram usados dois conjuntos de programas, no qual 195 projetos são de código aberto do *Github* e 72 submissões de códigos de alunos de Engenharia de Software. Os resultados obtidos pelo artigo foram a detecção de 60 testes quebradiços em 21 dos 195 projetos de código aberto enquanto nas submissões de alunos foram encontrados 110 testes quebradiços.

O trabalho de Gao (2015) tem como objetivo desenvolver uma abordagem baseada em cálculo de entropia para classificar as propriedades gráficas de cada componente dentro de uma interface de usuário. Essa classificação serve para determinar se um *bug* é real ou é derivado de um falso positivo que muitas vezes ocasiona em um teste quebradiço. Em um cenário de teste normal o trabalho compara os estados do componente após execuções de versões distintas para identificar erros de layout. O trabalho usou três aplicativos como base para encontrar os defeitos, o [4]jEdit4 que é um editor de texto para programadores, o [5]Jmol5 que é um virtualizador molecular de estruturas químicas e o [6]JabRef6 que é um gerenciador de referências bibliográficas. Após a avaliação de entropia nos componentes dos aplicativos, o trabalho conseguiu reunir as principais propriedades dos componentes que podem ser a razão dos falsos positivos. As propriedades foram divididas em dois grupos: (I) propriedades de renderização que incluem ícones, largura e altura, coordenadas, fonte, cor de fundo entre outras propriedades; e (II) propriedades funcionais no código que geram ações na tela, textos, localidade, entre outros. Os resultados obtidos mostram que

[4]`acesse:http://www.artman21.com/old_product/Jedit4/index_E.html`
[5]`acesse:http://jmol.sourceforge.net/`
[6]`acesse:https://docs.jabref.org/`

as propriedades de renderização são responsáveis por 93% de todos os falsos positivos e são consideradas mais causadoras de testes quebradiços se comparada com as propriedades funcionais.

O trabalho de Luo et al. (2018) tem como objetivo fornecer informações sobre como evitar os testes quebradiços por meio da detecção e correção deles. Para isso, o trabalho concentrou-se na identificação e avaliação dos testes quebradiços analisando *logs* e comentários de *commits* que dizem corrigir o teste quebradiço. Os processos de avaliação e identificação ocorreram com projetos da *Apache Software Foundation* e somente os *logs* e as mensagens de *commits* que citavam as palavras-chave "intermit" e "flaky" foram escolhidos para a avaliação. O trabalho foi dividido em duas fases, (1) filtragem e (2) análise. No processo de filtragem foram analisados todos os códigos de *commits* e também os *logs* dos controles de versões. Nessa fase foram identificados 486 *commits* com testes quebradiços em pelo menos 51 dos 153 projetos analisados. Na fase de análise, os projetos usados foram divididos em dois grupos, pequenos e grandes com base na quantidade de *commits*. Para a avaliação final foram selecionados todos os projetos agrupados como pequenos e um terço dos projetos considerado grandes, totalizando 201 *commits*. Os resultados obtidos foram de grande ajuda, principalmente a identificação de 10 categorias de causas dos testes quebradiços e também algumas formas de corrigir esses testes quebradiços. Dentre as categorias pode-se destacar a de espera assíncrona com 45% das causas e concorrência com 20% das causas.

O trabalho de Labuschagne, Inozemtseva e Holmes (2017) tem como objetivo investigar os custos e benefícios do teste de regressão automatizada. Foram observados as falhas de 61 projetos que usavam Travis CI como ferramenta de IC. Também observou-se como os desenvolvedores resolveram essas falhas para realizar uma classificação quanto à causa, podendo classificá-las: (I) um teste quebradiço; (II) defeito no sistema; (III) teste mal implementado. Descobriu-se que 18% das execuções de conjuntos de testes falham e que 13% dessas falhas são testes quebradiços. Das falhas que não são consideradas como testes quebradiços , apenas 74% foram causadas por um defeito no sistema em teste, enquanto os 26% restantes foram devidos a testes incorretos ou obsoletos. Além disso, descobriu-se que, nas compilações com falha, apenas 0,38% das execuções de casos de teste falharam e 64% das construções com falha continham mais de um teste com falha, o que é considerado dependência entre testes.

3.4 AVALIAÇÃO DOS ESTUDOS

A avaliação dos estudos nessa etapa baseou-se no levantamento de dados da Seção 3.3. Os dados extraídos foram sintetizados e categorizados de acordo com as informações expostas por cada trabalho.

Todos os estudos encontrados abordam os testes quebradiços de uma forma, seja ela uma análise acerca da ocorrência dos testes quebradiços ou a criação de técnicas e ferramentas para prevenir ou solucionar os testes quebradiços. Para um melhor entendimento e sintetização dos dados, separaram-se os trabalhos em categorias que respondem a questão norteadora da presente revisão. As categorias abordadas nos próximos tópicos são: Identificação das causas dos testes quebradiços; Identificação de possíveis soluções para os testes quebradiços; e Criação de técnicas e ferramentas para prevenir ou solucionar os testes quebradiços.

3.4.1 IDENTIFICAÇÃO DAS CAUSAS DOS TESTES QUEBRADIÇOS

O primeiro trabalho a ser destacado é o de Luo et al. (2018). Os autores são considerados os primeiros a realizarem um estudo extensivo sobre as possíveis causas dos testes quebradiços. Nesse trabalho, o esforço dos autores ajudou a categorizar as causas dos testes quebradiços em dez categorias não conhecidas pelos pesquisadores do assunto. As categorias encontradas foram: espera assíncrona, concorrência, dependência da ordem de execução do teste, vazamento de dados, rede, hora do sistema, operações de entrada e saída, dados gerados aleatoriamente, operações de ponto flutuante, coleções de dados não ordenadas e por fim as causas não identificadas.

A causa dos testes quebradiços pode ser decorrente de um teste mal planejado conhecido como mal cheiro de testes, que pode afetar os resultados dos testes, como é abordado no trabalho Palomba e Zaidman (2017). Os autores desse trabalho usam as dez categorias encontradas por Luo et al. (2018) para identificar os testes quebradiços e depois verificar se é decorrente de um teste mal planejado.

O aprendizado de máquina também está presente na busca de identificar as principais causas dos testes quebradiços. King Dionny Santiago (2018) propuseram uma abordagem para identificar os fatores da ocorrência dos testes quebradiços através de uma rede *bayesiana*. Essa rede será baseada nos fatores encontrados para a ocorrência do teste quebradiço. Os fatores que o estudo aponta como causadores dos testes quebradiços são: alta complexidade do código de teste, acoplamento de implementação de testes, teste não

determinístico e configuração inapropriada do teste.

Novas categorias de causas foram estabelecidas por meio do estudo Thorve, Sreshtha e Meng (2018), os autores levaram como base as categorias identificadas por Luo et al. (2018) para identificar os testes quebradiços nos aplicativos android. Além disso o trabalho realizado é o primeiro focado nos aplicativos android. Como resultado, surgiram novas categorias que não haviam sido identificadas por outros pesquisadores da área. As novas categorias são: Lógica de programação e interface de usuário.

3.4.2 IDENTIFICAÇÃO DE POSSÍVEIS SOLUÇÕES PARA OS TESTES QUEBRADIÇOS

Os autores de Thorve, Sreshtha e Meng (2018) identificaram as possíveis soluções dos testes quebradiços em aplicativos android, são elas: melhora na implementação, reescrever a implementação, retestar até o limite ou sucesso, modificar o resultado esperado do teste e remover o teste por completo.

As soluções também foram descobertas de acordo com as categorias de causas (LUO et al., 2018). Nesse trabalho os autores conseguiram classificas as soluções de acordo com cada categoria descoberta de causa. Para o problema de espera assíncrona podem-se destacar as soluções: bloqueio da thread até que uma condição seja satisfeita, adicionar funções de espera dentro do código (Ex: *sleep*) e alterar a ordem de execução de partes do código. Para concorrência, pode-se destacar: adicionar bloqueios para concorrência de threads, eliminar simultaneidade dos códigos, aperfeiçoar condições em determinadas funções que aceitam vários acessos ao mesmo tempo e correção de asserções nos códigos. Para dependência da ordem de execução do teste, pode-se destacar: limpeza do estado dos testes antes de iniciar outro, removendo dependências de atributos entre códigos e mesclar códigos de testes em outros testes. Para as outras categorias eles destacam soluções como: controle de acesso de dados, fechar sempre que abrir conexões de entrada e saída, tratamento de limites para geradores de números aleatórios,

3.4.3 CRIAÇÃO DE TÉCNICAS E FERRAMENTAS PARA PREVENIR OS TESTES QUEBRADIÇOS

No trabalho King Dionny Santiago (2018) os autores criaram uma técnica com aprendizado de máquina para identificar os testes quebradiços por meio dos fatores das causas identificadas. Os autores do trabalho (SHI ALEX GYORI, 2016) criaram uma técnica chamada de NONDEX que detecta testes quebradiços com códigos conhecidos como ADINS. Esses códigos referem-se à interpretação determinística de um método com

uma especificação não determinística. A técnica foi desenvolvida na linguagem Java.

Os testes quebradiços também estão presentes em testes envolvendo interface do usuário. Gao (2015) usou uma técnica baseada em entropia para classificar as propriedades gráficas de cada componente dento da interface do usuário. A classificação serve para averiguar se o teste falha é um bug real ou deriva-se de um falso positivo que pode ocasionar os testes quebradiços.

Dentre as ferramentas criadas para o problema dos testes quebradiços, pode-se destacar o Deflaker (BELL OWOLABI LEGUNSEN, 2018). Essa ferramenta tem como função reexecutar os testes para averiguar se realmente é um teste quebradiço.

3.5 INTERPRETAÇÃO E APRESENTAÇÃO DOS RESULTADOS

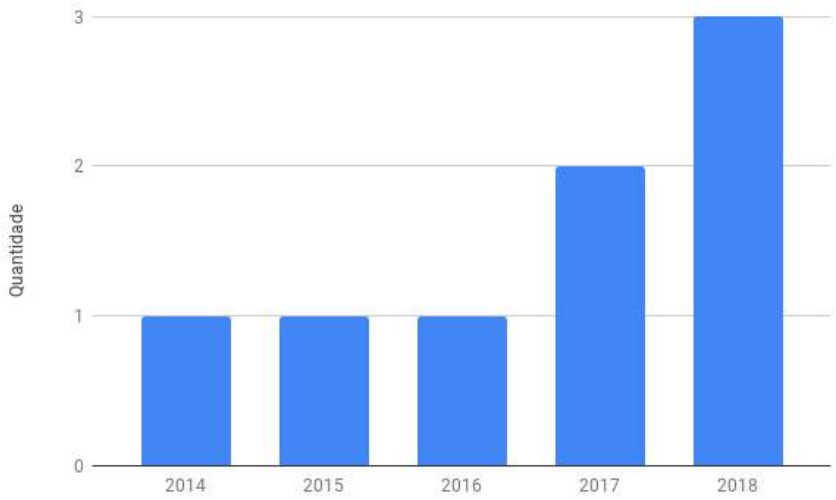
Por fim, apresentam-se os principais resultados coletados dos trabalhos selecionados e avaliados. Para, isso o trabalho de pesquisa foi executado em quatro etapas, desde a identificação do tema de pesquisa juntamente com a questão de pesquisa a síntese dos resultados.

Os trabalhos selecionados foram lidos na íntegra e avaliados quanto aos seus resultados e conclusões. A Tabela 7 apresenta dados identificadores dos trabalhos juntamente com uma explicação sobre cada trabalho. Cada estudo foi observado de acordo com seus objetivos, metodologia e conclusão.

Por fim, realizou-se uma síntese e comparação dos resultados obtidos de cada trabalho. As comparações apontaram melhorias e diferenças entre os trabalhos, como também apontaram semelhanças. A síntese dos dados, possibilitou a criação de três categorias para classificar os trabalhos de acordo com o seu tipo de contribuição para os testes quebradiços, levando-se em conta a questão norteadora da pesquisa.

Os estudos encontrados e selecionados são considerados recentes como mostra a Figura 10, afirmando que o tema é atual e de interesse de vários pesquisadores.

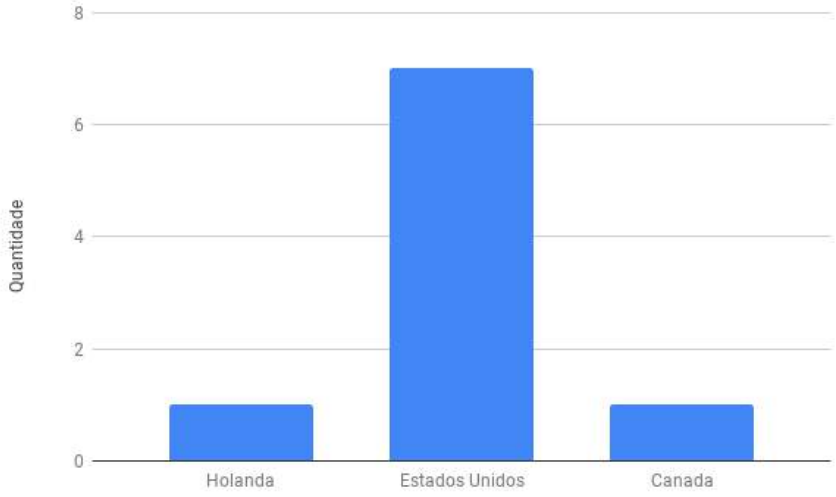
Figura 10: Quantidade de trabalhos encontrados na RI de acordo com seu ano.



Fonte: Autoria própria

Também pode-se destacar os países de filiação dos primeiros autores que publicaram os trabalhos selecionados na Figura 11, o Estados Unidos é o destaque com a maioria dos trabalhos e são acompanhados pelo Canadá e a Holanda com somente um trabalho cada. Assim, afirma-se que no Brasil os testes quebradiços têm sido pouco estudados o que valoriza a contribuição do presente estudo.

Figura 11: Quantidade de trabalhos encontrados na RI de acordo com seu país.

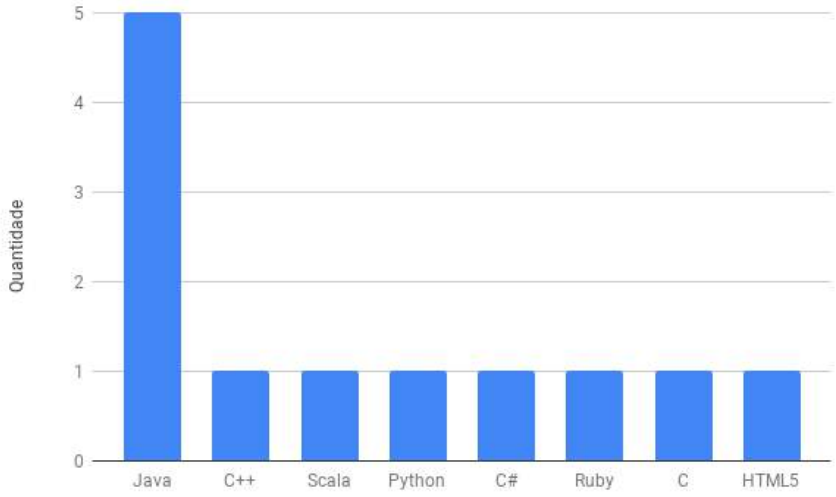


Fonte: Autoria própria

Entre os tipos de linguagens de programação analisados pelos projetos, a linguagem Java é a destaque com a maioria dos trabalhos, como mostra a Figura 12. O HTML5

aparece apenas com um trabalho, o que demonstra um campo pouco explorado nas linguagens de programação web, possibilitando estudos sobre os aplicativos híbridos.

Figura 12: Quantidade de trabalhos encontrados na RI de acordo com sua linguagem de programação.



Fonte: Autoria própria

O aprendizado de máquina também está presente nos testes quebradiços, King Dionny Santiago (2018) propôs uma rede bayesiana para identificar os testes quebradiços por meio dos seus fatores de causas identificados. Como pouco têm se estudado e criado abordagens novas podem ser criadas usando a inteligência artificial como auxiliadora na identificação dos testes quebradiços.

Após a análise dos trabalhos selecionados, conseguiu-se reunir informações suficientes sobre as categorias de causas dos testes quebradiços como mostra a Tabela 8.

Tabela 8: Categorias de causas dos testes quebradiços

Causa	Motivo
Espera assíncrona	Os problemas envolvendo métodos assíncronos são aqueles que ocorrem através da espera do programa por algo que ainda está por vir, ou seja, a execução do teste depende de algo carregar ou renderizar na tela, mas acaba continuando a execução sem antes aparecer. Um exemplo para isso é uma requisição para algum servidor, onde espera-se o retorno com dados. Caso haja alguma falha com a rede causando lentidão na busca, o programa acaba ficando sem o retorno de dados, o que pode gerar erros na execução.
Concorrência	Os testes de software que entram na categoria de concorrência são aqueles que podem ser classificados como uma interação do programa não desejável com outros serviços onde podem ocorrer disputa de dados. Um exemplo disso é o deadlock.

Continuação na próxima página

Tabela 8 – Continuação da página anterior

Causa	Motivo
Ordem da execução dos testes	Os testes de software sempre devem ser únicos e independentes de outros testes de software. Nesta categoria podemos incluir os testes de software que dependem um do outro e também da ordem em que são executados os testes de software.
Rede	A rede é uma causa dos testes quebradiços onde pode-se ter menos controle. Muitas vezes, a conexão torna-se instável e lenta, o que pode levar um teste de software a falhar durante a sua execução, causando uma certa dúvida para os testadores e desenvolvedores.
Hora do sistema	Em alguns casos os testes podem falhar devido ao fuso horário do sistema. Muitas plataformas tratam a hora e seu fuso de diferentes formas, o que ocasiona teste não determinísticos em determinadas situações.
Vazamento de dados	Algumas falhas podem ocorrer quando o programa não gerencia seus recursos corretamente, podendo ocorrer vazamentos de recursos. Dentro disso temos alguns exemplos que ocorrem dentro de banco de dados com buscas que excedem o limite de dados, alocações de memória incorretas e também o próprio envio de muitos dados através da rede.
Operações de ponto flutuante	Operações com ponto flutuante são comuns em sistemas onde é necessário alto desempenho. As operações devem ser realizadas corretamente e com precisão para que não hajam problemas como o de *underflow*, adições não associativas entre outros.
Lógica de programação	Em alguns casos os testes quebradiços são causados devido a erros de programação por parte dos desenvolvedores. Muitas vezes uma interpretação errada de como implementar a funcionalidade acaba ocasionando em um teste quebradiço.
Interface de usuário	Problemas com interface de usuário também acabam gerando testes quebradiços. Esses problemas acontecem quando os desenvolvedores projetam um layout do programa de forma incorreta, ou seja, componentes que não renderizam adequadamente e componentes que não tem seu comportamento executado corretamente(abrir e fechar).
Operações de entrada e saída	Nessa categoria podemos classificar os problemas relacionados com operações que podem usufruir de entrada e saída de dados. Um exemplo para isso é a leitura de dados por parte do programa em cima de um arquivo de texto. Alguns casos pode-se inserir arquivos com textos longos, o que ocasiona uma certa demora no processo, abrindo oportunidades para um teste se tornar quebradiço.
Dados gerados aleatoriamente	Alguns programas usam geradores de dados aleatórios que são usados em diversas funcionalidades. Nesse contexto os geradores devem ser implementados de forma cuidadosa, pois em alguns casos os números gerados podem não serem aceitos pelo programa e ocasionarem erros. Esse tipo de problema pode ser considerado um teste quebradiço, pois em alguns momentos os dados podem ser corretos e o programa funcionar perfeitamente, mas em outros momentos os dados podem ser incorretos para o programa.
Coleções de dados não ordenadas	Nessa categoria podemos classificar os problemas relacionadas com algoritmos que tratam coleções de dados não ordenados e esperam retorno de dados ordenados. Um teste pode falhar caso deseja-se retornar os dados de uma coleção em uma ordem específica, mas a coleção de dados é iterada de forma não ordenada.
Causas não identificadas	Os estudos mostraram alguns problemas de teste quebradiço sem causas relacionadas, não conseguindo identificar o problema e nem resolvê-lo.

Continuação na próxima página

Fonte: Autoria própria

Na Tabela 9 foi possível reunir informações sobre as soluções encontradas na literatura para resolver os testes quebradiços.

Tabela 9: Soluções encontradas para testes quebradiços na literatura

Soluções	Definição
Melhora na implementação	Essa é uma das soluções mais comuns dentro dos testes quebradiços, em muitos casos a correção do código adicionado novas validações, ou removendo validações que já existiam, é o suficiente para corrigir os problemas como o de lógica de programação e o de interface de usuário. Outros problemas que podem ser removidos através de melhoria de código são aqueles que envolvem coleções de dados não ordenadas e dados gerados aleatoriamente
Atualização ou substituição	Uma das soluções apontadas pelos autores é a atualização de serviços e bibliotecas usadas no programa por uma versão melhorada ou alguma versão antiga. Também apontam como solução a substituição de uma biblioteca que realiza determinada funcionalidade, por outra biblioteca com o mesmo intuito.
Reexecução	A reexecução do teste de software é uma solução para os quebradiços quando a equipe de desenvolvimento não sabe qual a possível causa do problema. Alguns testes podem falhar simplesmente devido a conexão lenta envolvendo grandes processamentos, espera assíncrona e consumo alto de dados de serviços externos. Devido a esses motivos, realizar o teste de software novamente é importante para alguns casos do teste
Alterar asserções dos testes de software	Em alguns testes de software, as asserções esperadas podem ser inválidas ou mal interpretadas pelos desenvolvedores e testadores, o que pode acabar gerando um teste quebradiço. Para isso os pesquisadores citam que a alteração e reavaliação pode ser fundamental para a correção do teste quebradiço.
Remoção do teste de software	Essa medida para correção dos testes quebradiços não é considerada uma solução correta para o problema. Ela é considerada uma solução somente quando os desenvolvedores e testadores não conseguem encontrar a causa. Um problema que envolve essa solução é que os testes que serão removidos podem esconder problemas reais no programa.

Fonte: Autoria própria

No Capítulo 4 são apresentados os materiais utilizados para a execução do estudo, os projetos encontrados, a metodologia utilizada, as ferramentas de apoio e os *scripts* criados.

4 MATERIAIS E MÉTODOS

Este capítulo apresenta os materiais, métodos e recursos utilizados durante a execução do estudo de caso. Inicialmente, apresenta-se qual a estratégia utilizada para a identificação e coleta dos projetos de aplicativos híbridos, as ferramentas de apoio usadas, os hardwares utilizados e por fim os *Scripts* de apoio criados.

4.1 CONTEXTO GERAL

Atividades de teste de software dentro do desenvolvimento de software são consideradas medidas de segurança para avaliar se o funcionamento do software está correto ou não (DELAMARO, 2016). Em muitos casos, esses testes são automatizados para obter uma produtividade maior, por meio do grande número de execuções e também para conseguir uma maior abrangência na absorção de erros (BERNARDO, 2008). Contudo, nem sempre os testes de software podem trazer confiança, eles podem ser confusos e indeterminados. Diante disso, em algumas execuções um caso de teste tem um resultado de sucesso, mas em outras ocasiões ele acaba falhando, o que pode causar dúvida para os envolvidos devido à incerteza dos resultados e a confiabilidade do conjunto de teste. Esses testes duvidosos são conhecidos na academia e na indústria como testes quebradiços (THORVE; SRESHTHA; MENG, 2018; GEORGE, 2014).

O surgimento dos testes quebradiços abriu portas para novas pesquisas na área de teste de software. Alguns estudos foram realizados visando descobrir as causas e também propostas de como resolver os testes quebradiços (BELL OWOLABI LEGUNSEN, 2018; THORVE; SRESHTHA; MENG, 2018; LUO et al., 2018). Almejando a oportunidade de novas pesquisas, esse trabalho de conclusão de curso investiga a ocorrência dos testes quebradiços em aplicativos híbridos por meio da análise e execuções sistemáticas dos testes automatizados de projetos reais, realizando avaliações quantitativas, sistemáticas e também categorizações dos resultados obtidos.

Diante do apresentado, fez-se necessário dividir o estudo em três grandes etapas:

1. Identificação e seleção de projetos híbridos reais (Seção 4.2);
2. Seleção de ferramentas e desenvolvimento de instrumentos experimentais (Seção 4.3); e
3. Condução de avaliações empíricas (Capítulo 5).

4.2 IDENTIFICAÇÃO DE PROJETOS DE APLICATIVOS HÍBRIDOS

Para a execução do estudo de caso, diversos projetos foram selecionados em diferentes repositórios online de códigos fonte. Os repositórios utilizados foram:

- [1]*Github*: é uma plataforma de desenvolvimento onde você pode hospedar e revisar códigos, gerenciar projetos e criar software (GITHUB, 2019); e
- [2]*Gitlab*: é um gerenciador de repositório de software baseado em *git*, que realiza gerenciamento de tarefas e também IC (GITLAB, 2019).

Algumas palavras chaves foram definidas para a busca manual dos projetos dentro dos repositórios de código, com o objetivo de encontrar aplicativos híbridos com testes automatizados. As palavras chaves usadas foram: '*React native*', '*Ionic*' e '*Flutter*'.

Para a coleta inicial dos projetos, algumas características foram levadas em conta:

1. Ser caracterizado como um aplicativo híbrido;
2. Conter testes automatizados implementados; e
3. Ser um aplicativo de código aberto.

Após a conclusão da busca dos projetos, obteve-se uma amostra inicial de 26 projetos de aplicativos híbridos com testes automatizados, como mostra a Tabela 10. As 26 amostras foram avaliadas e testadas para averiguar o funcionamento correto, tanto do aplicativo rodando quanto dos testes automatizados. Após essa avaliação, obteve-se uma amostra final de doze projetos que foram utilizados para a execução do estudo de caso como mostra a Tabela 11. Os projetos que foram descartados se encontravam com erros na execução dos testes e também com erros de configuração de projeto.

[1]acesse: `https://github.com/`
[2]acesse: `https://about.gitlab.com/`

Tabela 10: Amostra inicial de projetos para o estudo de caso

Nome	Link	Repositório	Framework
inKino	https://gitlab.com/xsahil03x/inKino	Gitlab	Flutter
inKino mobile	https://gitlab.com/alexisvt/inKino	Gitlab	Flutter
Al-Quran app	https://gitlab.com/5yunus2efendi/quran_app	Gitlab	Flutter
Sapawarga	https://gitlab.com/jdsteam/sapa-warga/sapawarga-flutter	Gitlab	Flutter
Spartathlon	https://gitlab.com/thgoebel/spartathlon	Gitlab	Flutter
Flutter study	https://gitlab.com/good-good-study/Flutter_Study_App	Gitlab	Flutter
Travel rates	https://gitlab.com/greycastle/travel-rates	Gitlab	Flutter
Stencil	https://gitlab.com/shipsahoy/m/js/stencil	Gitlab	Ionic
Carbon Footprint	https://github.com/AOSSIE-Org/CarbonFootprint-Mobile	Gitlab	React native
Minds mobile	https://gitlab.com/minds/mobile-native	Gitlab	React native
Jet	https://github.com/invertase/jet	Github	React native
RocketChat	https://github.com/RocketChat/Rocket.Chat.ReactNative	Github	React native
RNTester	https://github.com/facebook/react-native/tree/master/RNTester	Github	React native
Pride in London App	https://github.com/redbadger/pride-london-app	Github	React native
Bristol Pound	https://gitlab.com/TownPound/Cyclos/ScottLogic.mobile.react-native/BristolPound	Gitlab	React native
Beep	https://github.com/moduscreateorg/beep	Github	Ionic
Perfi	https://github.com/apiko-dev/Perfi	Github	React native
WeightTracker	https://github.com/MSzalek-Mobile/weight_tracker	Github	Flutter
Invoice Ninja	https://github.com/invoiceninja/flutter-mobile	Github	Flutter
Cine reel	https://github.com/kserko/CineReel	Github	Flutter
Slide puzzle	https://github.com/kevmoo/slide_puzzle	Github	Flutter
Flitter	https://github.com/dart-flitter/flitter	Github	Flutter
Rxdb	https://github.com/pubkey/rxdb	Github	React native
Soundboard	https://github.com/Mokkapps/parents-soundboard	Github	React native
Words Reminder	https://github.com/akiver/wordsreminder	Github	React native
GitPoint	https://github.com/gitpoint/git-point	Github	React native

Fonte: Autoria própria

4.3 FERRAMENTAS DE APOIO

Durante a execução do estudo de caso, algumas ferramentas foram utilizadas para apoiar a captação e análise dos resultados. Primeiramente definiram-se as *IDEs*, que foram utilizadas para analisar a estrutura dos projetos de aplicativos híbridos, a criação dos *Scripts* de apoio e executar os testes automatizados dos mesmos.

- [3]Android studio: é um ambiente de desenvolvimento integrado para desenvolver para a plataforma *Android*. Foi baseado na Intellij IDEA e conta com um editor de layout para arrastar os componentes, refatoração de código, ferramentas *lint* entre outras funcionalidades (GOOGLE, 2019a);

- [4]Visual studio code: editor de código-fonte criado pela Microsoft, com capacidade

[3]acesse: `https://developer.android.com/studio`
[4]acesse: `https://code.visualstudio.com/`

de depuração, controle *git*, ajuste inteligente de código, *snippets* e refatoração de código (MICROSOFT, 2019); e

- [5]Intellij IDEA: é um ambiente de desenvolvimento integrado desenvolvido pela *JetBrains* na linguagem de programação Java, para o desenvolvimento de software, oferecendo diversos recursos, como refatoração de código, conclusão de código, entre outras (JETBRAINS, 2019).

Para armazenar os resultados obtidos nas execuções dos testes automatizados e também para armazenar informações gerais sobre cada projeto utilizou-se da ferramenta de planilhas do *Google*, conhecida como *Google Sheets*.

- [6]Google sheets: é um programa de manuseio de planilhas criado pelo *Google*®, que fornece suporte a integração com diversas ferramentas da empresa(Google docs, Google drive) (GOOGLE, 2019b).

4.4 RECURSOS COMPUTACIONAIS UTILIZADOS

Durante a execução do estudo de caso, duas configurações de recursos computacionais distintos foram utilizados para a execução dos testes automatizados em cada um dos 12 projetos selecionados. Em alguns projetos a execução foi demorada, por necessitar de um hardware com melhores configurações para obter uma melhor desempenho. Para isso, o Macbook Pro® foi utilizado por contar com uma boa configuração e desempenho. Assim, foram utilizados dois notebooks, um sendo da marca Samsung® e o outro da Apple®:

- Samsung: Notebook Samsung 8G Ram,processador Intel Core i5; e
- Apple: Macbook Pro 2018 8G Ram.

4.5 *SCRIPTS* DE APOIO

Dois *Scripts* distintos foram criados para apoiar na coleta de dados durante a execução dos testes automatizados.

- *Script* de execução: tem como objetivo executar os mesmos testes automatizados diversas vezes com apenas um comando, substituindo uma execução manual, no qual teria que executar o mesmo comando diversas vezes

[5]acesse: `https://www.jetbrains.com/idea/`
[6]acesse: `https://www.google.com/sheets/about/`

- Script de análise de dados: tem como objetivo analisar os dados gerados em cada execução dos testes automatizados, onde basicamente o *Script* lê os dados de um arquivo que foi gerado nas execuções e cria um outro arquivo no formato *csv*, com informações relevantes das execuções.

.

4.5.1 SCRIPT DE EXECUÇÃO

O *Script* de execução foi desenvolvido na linguagem de programação *Javascript*, com o objetivo de automatizar a execução dos testes.

Na Figura 13 é possível notar a estrutura do *Script* criado, promovendo a execução do comando "yarn test" 200 vezes. A estrutura se dá por meio de três funções "executeCommand", "executeScript" e "geraTxt" nomeadas em f1, f2 e f3 respectivamente. A função f1 tem como objetivo executar o comando escolhido e armazenar sua saída para gerar um arquivo no formato *txt* com as informações. A função f2 tem como objetivo iterar o número de vezes escolhido dentro da estrutura de *loop* e para cada iteração chamar a função f1. Por fim a função f3, tem como objetivo gerar um arquivo de texto com as informações recebidas e guardar em uma pasta o arquivo.

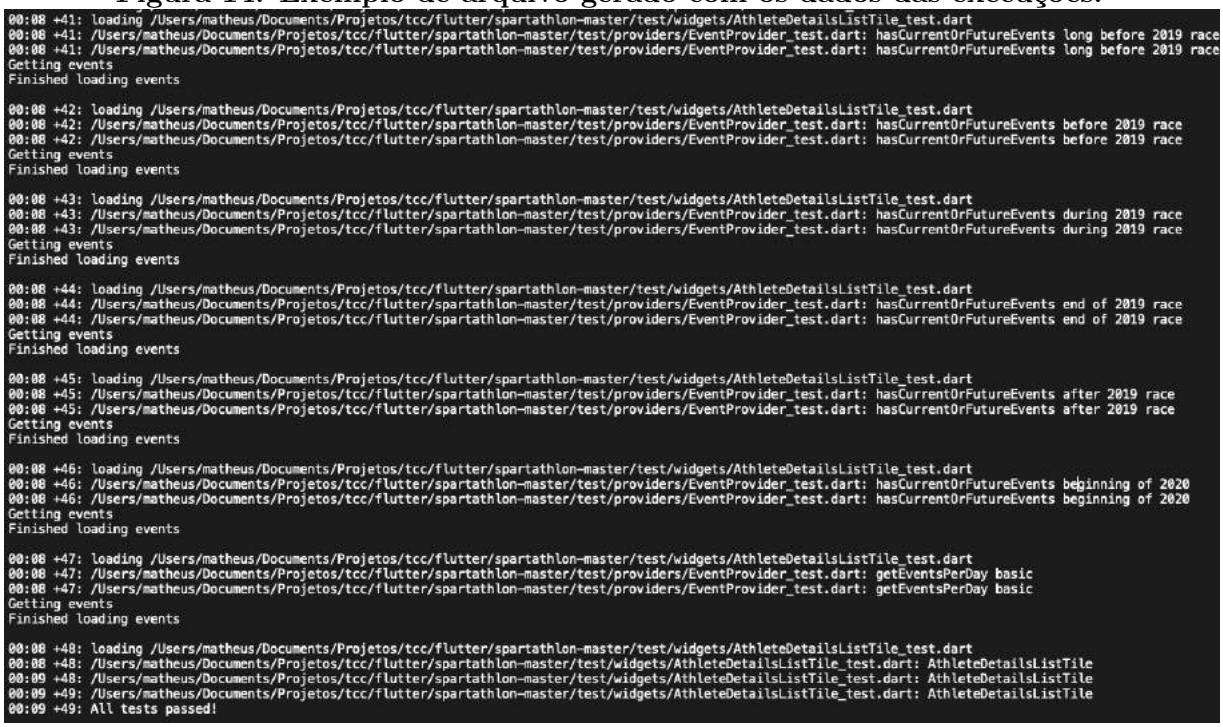
Figura 13: Script para automatizar execuções.

```
const util = require('util');
const fs = require('fs')
const exec = util.promisify(require('child_process').exec);
async function executeCommand(i) {
  try {
      const { stdout, stderr } = await exec('yarn test');
       await geraTxt(stdout,i);
       await geraTxt(stderr,i);
  }catch (err){
       console.error(err);
  };
};
let cont = 0;
async function executeScript() {
   for (let i = 0; i < 200; i ++) {
       cont ++;
       await executeCommand(i);
   }
}
async function geraTxt(data,i) {
   await fs.writeFile('resultadosteste200/Output' + i + '.txt',
   data, (err) => {
       if (err) throw err;
   })
}
executeScript();
execute();
```

Fonte: Autoria própria

A Figura 14 apresenta uma amostra parcial de dados gerados pela execução do *Script* armazenados em um arquivo no formato *.txt*. Esses dados retornos da execução de cada teste automatizado de um projeto, informando se o teste passou ou falhou.

Figura 14: Exemplo de arquivo gerado com os dados das execuções.



Fonte: Autoria própria

4.5.2 SCRIPT DE ANÁLISE DE DADOS

O *Script* de análise de dados foi criado na linguagem de programação Java com o objetivo de automatizar o processo de coletar dados dos resultados obtidos nas execuções dos testes automatizados. Para cada tipo de projeto criou-se uma versão diferente do *Script*, mapeando as informações necessárias como mostram as Figuras 15, 16 e 17.

A versão destinada para *Ionic* e para o Flutter são parecidas, contam com uma iteração de acordo com a quantidade informada no *loop*, seguidamente realiza-se a leitura do arquivo de dentro de uma pasta específica. Por fim, aplica-se uma validação na última linha dos arquivos gerados, verificando se todos os testes passaram. Caso todos os testes tenham passado, algumas informações do arquivo são coletadas e guardadas. Caso em algum arquivo não passem todos os testes, um *log* com o nome do arquivo será disparado para uma averiguação.

Figura 15: Script para automatizar coleta de resultados em projetos Ionic.

```java
for (int i = 0; i < 98; i++) {
    final Stream<String> lines = Files.lines(
            Paths.get("/Users/matheus/tcc-workspace/tcc/src/main
            /java/arquivos/Output" + i + ".txt"),
            Charset.defaultCharset());
    final List<Object> retorno = lines.collect(Collectors.toList());
    List<String> strings = new ArrayList<String>();
    for (Object object : retorno) {
        strings.add(Objects.toString(object, null));
    }
    int size = strings.size();
    Dados obj = new Dados();
    obj.setIdExecucao(i);
    if (strings.get(size - 2).split(" ")[1].equals("passing")) {
        obj.setTestPassed(strings.get(size - 2).split(" ")[0]);
        obj.setTestTotal(strings.get(size - 2).split(" ")[0]);
        String aux = strings.get(size - 2).split(" ")[2];
        String replaced = aux.replaceAll("[()]", "");
        obj.setTime(replaced);
        obj.setTestFail(Double.parseDouble(
        obj.getTestTotal()) - Double.parseDouble(obj.getTestPassed()));
        list.add(obj);
    } else {
        System.out.print("Erro no arquivo Output" + i);
    }
}
```

Fonte: Autoria própria

Figura 16: Script para automatizar coleta de resultados em projetos Flutter.

```java
for (int i = 0; i < 200; i++) {
    final Stream<String> lines = Files.lines(
        Paths.get("/Users/matheus/tcc-workspace/tcc/src/main
        /java/arquivos/Output" + i + ".txt"),
        Charset.defaultCharset());
    final List<Object> retorno = lines.collect(Collectors.toList());
    List<String> strings = new ArrayList<String>();
    for (Object object : retorno) {
        strings.add(Objects.toString(object, null));
    }
    int size = strings.size();
    Dados obj = new Dados();
    obj.setIdExecucao(i);
    if (strings.get(size - 1).split(":")[2].trim().compareTo
    ("All tests passed!") == 0) {
        obj.setTestPassed(strings.get(size - 1)
        .split(":")[1].split("")[1].replace("+", "").trim());
        obj.setTestTotal(strings.get(size - 1).split(":")[1]
        .split(" ")[1].replace("+", "").trim());
        obj.setTestFail(0.0);
        obj.setTime(strings.get(size - 1).split(":")[1].split(" ")[0]);
        list.add(obj);
    } else {
        System.out.print("Erro no arquivo Output" + i);
    }
}
```

Fonte: Autoria própria

Para os projetos de React native, mantém-se a base da lógica das versões de Ionic e Flutter, porém sem a validação que verifica se todos os testes passaram. Essa validação não é necessária, haja vista que dentro dos arquivos desse projeto constam informações detalhadas dizendo quantos testes passaram e quantos falharam.

Figura 17: Script para automatizar coleta de resultados em projetos React native.

```java
for (int i = 0; i < 200; i++) {
   final Stream<String> lines = Files.lines(
           Paths.get("/Users/matheus/tcc-workspace/tcc/src/main
           /java/arquivos/Output" + i + ".txt"),
           Charset.defaultCharset());

   final List<Object> retorno = lines.collect(Collectors.toList());

   List<String> strings = new ArrayList<String>();
   for (Object object : retorno) {
       strings.add(Objects.toString(object, null));
   }
   int size = strings.size();
   Dados obj = new Dados();
   obj.setIdExecucao(i);
   obj.setSuitsPassed(strings.get(size - 5).
   split(":")[1].split(",")[0].split(" ")[1]);
   obj.setSuitsTotal(strings.get(size - 5).
   split(":")[1].split(",")[1].split(" ")[1]);
   obj.setTestPassed(strings.get(size - 4).
   split(":")[1].split(",")[0].split(" ")[7]);
   obj.setTestTotal(strings.get(size - 4).
   split(":")[1].split(",")[1].split(" ")[1]);
   obj.setTime(strings.get(size - 2).split(":")[1].trim());
   obj.setTestFail(Double.parseDouble
   (obj.getTestTotal()) - Double.parseDouble(obj.getTestPassed()));
   list.add(obj);
}
```

Fonte: Autoria própria

Por fim, na Figura 18 é apresentada a última parte dos *scripts* criado para gerar um arquivo com o formato csv com todos os dados coletados em cada execução.

Figura 18: Script para gerar arquivo csv.

```
PrintWriter pw = null;
try {
  pw = new PrintWriter(new File("/Users/matheus/tcc-workspace/tcc/src/main
  /java/csv/Avaliacoes.csv"));
} catch (FileNotFoundException e) {
  e.printStackTrace();
}

StringBuilder builder = new StringBuilder();
String ColumnNamesList = "Execucao,Quantidade de testes
+ "que passaram,Quantidade de testes que falharam,"
+ "Quantidade testes,Tempo total";
builder.append(ColumnNamesList + "\n");
for (int x = 0; x < list.size(); x++) {

  builder.append(list.get(x).getIdExecucao() + ",");
  builder.append(list.get(x).getTestPassed() + ",");
  builder.append(list.get(x).getTestFail() + ",");
  builder.append(list.get(x).getTestTotal() + ",");
  builder.append(list.get(x).getTime());
  builder.append('\n');

}
pw.write(builder.toString());
pw.close();
System.out.println("done!");
```

Fonte: Autoria própria

Na Figura 19, vê-se uma amostra parcial com 14 execuções do script e também o arquivo gerado no formato csv com as informações coletadas das execuções. Os resultados coletados em csv foram armazenados em planilhas no *Google Drive*, facilitando assim a organização para consultas aos dados. Os arquivos de resultados de cada projeto podem ser consultados [7]online.

[7]acesse: `https://drive.google.com/open?id=1MlEPDMhm22VRztBL-Ei_WYNcTTueRNXC`

Figura 19: Resultados coletados após a análise dos dados com o script de análise.

	A	B	C	D	E
1	Execucao	Quantidade de testes que passaram	Quantidade de testes que falharam	Quantidade testes	Tempo total
2	0	38	0.0	38	575.75
3	1	38	0.0	38	568.71
4	2	38	0.0	38	616.95
5	3	38	0.0	38	500.18
6	4	38	0.0	38	599.81
7	5	38	0.0	38	690.65
8	6	38	0.0	38	633.23
9	7	38	0.0	38	633.23
10	8	38	0.0	38	565.07
11	9	38	0.0	38	567.43
12	10	38	0.0	38	557.82
13	11	38	0.0	38	552.91
14	12	37	1.0	38	543.70
15	13	38	0.0	38	558.63
16	14	38	0.0	38	551.16

Fonte: Autoria própria

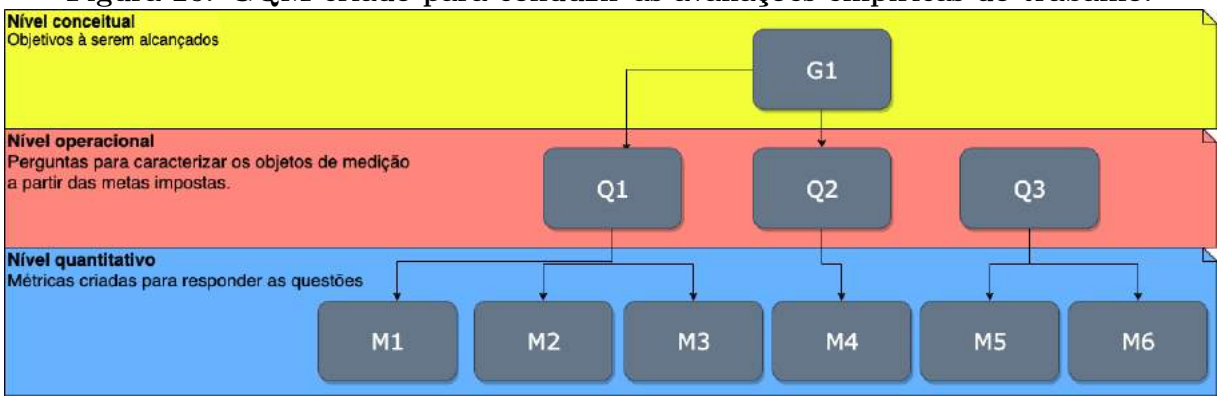
5 AVALIAÇÕES EMPÍRICAS

Este capítulo apresenta a estratégia experimental criada, os projetos utilizados para a execução do estudo de caso e o *design* experimental criado com seus procedimentos para atingir os resultados finais.

5.1 GQM

GQM que significa Objetivo, Questão e Métrica (do Inglês *Goal, Question and Metric*), é definido para avaliar os defeitos presentes em projetos. Inicialmente a estratégia surgiu para atender um projeto da NASA *Goddard Space Flight Center* e foi expandido para atender diversas áreas experimentais. Atualmente é usado para a definição de metas para a melhoria da qualidade dentro de organizações de software (BASILI; CALDIERA; ROMBACH, 1994). A Figura 20 ilustra o GQM criado para direcionar as avaliações empíricas do presente trabalho.

Figura 20: GQM criado para conduzir as avaliações empíricas do trabalho.



Fonte: Autoria própria

O GQM é dividido em três camadas: (i) conceitual; (ii) operacional; e (iii) quantitativo. Considerando o escopo do estudo de caso, foi definido o objetivo da aplicação do GQM:

- **G1**(*Goal* 1): Investigar e caracterizar testes quebradiços em projetos reais de aplicativos híbridos reais.

Na segunda camada foram elaboradas as questões de pesquisas a serem respondidas no intuito de levantar informações para atingir o objetivo almejado:

- **Q1**: Existem sinais da ocorrência de testes quebradiços em projetos reais de aplicativos híbridos reais?;
- **Q2**: Quais as possíveis causas de testes quebradiços em projetos de aplicativos híbridos; e
- **Q3**: Casos de teste baseados em qual técnica de teste (funcional ou estrutural) são mais propensos a gerar testes quebradiços em projetos híbridos?.

Por fim, na terceira camada, criaram-se métricas na busca de mensurar as respostas das perguntas para responder o objetivo:

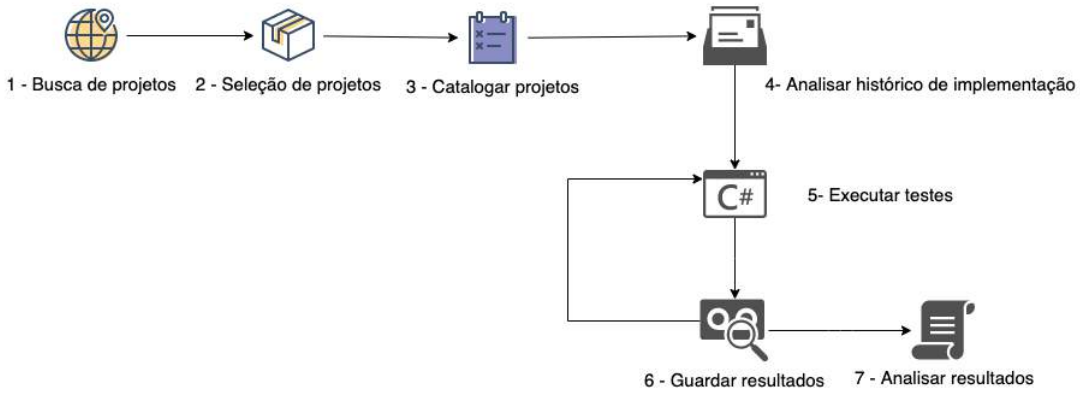
- **M1**: porcentagem dos testes que passam e falham invariavelmente;
- **M2**: porcentagem dos testes que passam;
- **M3**: porcentagem dos testes que falham;
- **M4**: porcentagem dos tipos de causas;
- **M5**: porcentagem de testes quebradiços ocorridos na técnica estrutural; e
- **M6**: porcentagem de testes quebradiços ocorridos na técnica funcional.

5.2 DESIGN EXPERIMENTAL E PROCEDIMENTOS

Para a realização do presente trabalho foi usada a abordagem quantitativa para análise e representação de dados estatísticos recuperados das execuções dos testes automatizados de cada projeto. Para a coleta de dados utilizou-se de uma metodologia baseada em métricas. Por meio da abordagem GQM, métricas foram utilizadas para saber quais dados coletar (WOHLIN et al., 2012). Diante disso, uma avaliação empírica foi conduzida por meio de um estudo de caso.

A execução do estudo de caso foi efetuada por meio de uma sequência de atividades, visando responder as questões e o objetivo definido no GQM criado, conforme ilustrado na Figura 21.

Figura 21: Procedimento experimental de estudo de caso.



Fonte: Autoria própria

Na Atividade 1 foi realizada a busca dos projetos de aplicativos híbridos em repositórios de códigos online. Os repositórios escolhidos foram [1]Github e [2]Gitlab, por serem os mais usados pela comunidade.

Na Atividade 2 foi realizada a avaliação dos projetos, para averiguar os mais aptos a serem usados no estudo de caso. A seleção baseou-se nas características definidas na Seção 4.2.

Na Atividade 3 foi realizada a catalogação dos projetos selecionados quanto a métricas de linhas de código e total de testes automatizados implementados. Essas informações coletadas são usadas para a Atividade 6, como um complemento para o resultado final.

Na Atividade 4 analisou-se o histórico de versionamento dos artefatos da implementação dos projetos dentro dos repositórios online. Essa análise tinha como objetivo, buscar sinais de testes quebradiços em *logs* de *commits*.

Na Atividade 5 foi realizada a execução dos testes automatizados dos projetos, sejam eles funcionais ou estruturais. A execução foi efetuada por meio de um *script* criado conforme apresentado na Seção 4.5, que visa automatizar o comando manual de executar os testes. Esse *script*, tem como benefício efetuar diversas vezes o comando responsável

[1] `https://github.com/`
[2] `https://about.gitlab.com/`

por executar os testes automatizados de cada projeto com apenas um comando.

Na Atividade 6, as informações dos resultados obtidos nas execuções da Atividade 5 foram guardados em um repositório de informações para consultas posteriores. O repositório utilizado para armazenar as informações foi a ferramenta Google planilhas.

Por fim, na Atividade 7 os resultados foram analisados a fim de responder as questões definidas no GQM. A análise realizada é baseada no resultado das execuções dos testes automatizados de cada projeto. Para definir se existem testes quebradiços, avaliou-se o resultado de cada teste automatizado, se o teste passou em algumas execuções e falhou em outras, ele é considerado um teste quebradiço.

Também foi definida uma variável dependente, com o objetivo de criar diferentes cenários de execuções em busca de melhores resultados. A variável criada controla o número de execuções dos testes automatizados de cada projeto, possibilitando uma alternância de execuções e criação de dois cenários diferentes,seguindo recomendações experimentais do Wohlin et al. (2012). O principal intuito de realizar diversas vezes a execução dos testes automatizados é tentar forçar e estimular a ocorrência dos testes quebradiços. Existem outras formas de tentar estimular essa ocorrência, porém as mesmas não puderam ser abordadas por limites associados ao tempo para a realização do trabalho de conclusão de curso e, por isso, serão consideradas em pesquisas futuras.

- NRE(Número de execuções): 100 e 200.

Por meio da criação de dois valores para a variável NRE, foi possível criar dois cenários de execuções:

- **C1**: Foram executadas 100 vezes todos os testes automatizados de cada projeto; e
- **C2**: Foram executadas 200 vezes todos os testes automatizados de cada projeto.

5.3 PROJETOS (APPS) UTILIZADOS

Para a execução do estudo de caso, foi utilizado uma amostra de doze projetos selecionados após uma triagem inicial. Na Tabela 11 é possível observar os projetos, considerando seus nomes, links, repositórios e *frameworks*.

Tabela 11: Projetos utilizados no estudo de caso

Código	Nome	Link	Repositório	Framework
1	inKino	https://gitlab.com/xsahil03x/inKino	Gitlab	Flutter
2	inKino mobile	https://gitlab.com/alexisvt/inKino	Gitlab	Flutter
3	Al-Quran app	https://gitlab.com/5yunus2efendi/quran_app	Gitlab	Flutter
4	Sapawarga	https://gitlab.com/jdsteam/sapa-warga/sapawarga-flutter	Gitlab	Flutter
5	Spartathlon	https://gitlab.com/thgoebel/spartathlon	Gitlab	Flutter
6	Travel rates	https://gitlab.com/greycastle/travel-rates	Gitlab	Flutter
7	Stencil	https://gitlab.com/shipsahoy/m/js/stencil	Gitlab	Ionic
8	Minds mobile	https://gitlab.com/minds/mobile-native	Gitlab	React native
9	Beep	https://github.com/moduscreateorg/beep	Github	Ionic
10	WeightTracker	https://github.com/MSzalek-Mobile/weight_tracker	Github	Flutter
11	Flitter	https://github.com/dart-flitter/flitter	Github	Flutter
12	Words Reminder	https://github.com/akiver/wordsreminder	Github	React native

Fonte: Autoria própria

Os projetos foram classificados quanto a técnica de teste funcional de software implementada, como mostra a Tabela 12. A classificação foi feita, por meio de uma execução inicial de cada teste automatizado, visando analisar o código implementa e identificar qual técnica estava implementada.

Tabela 12: Projetos coletados da técnica de teste estrutural

Código	Nome	Técnica	Quantidade de testes
1	inKino	Estrutural	31
2	inKino mobile	Estrutural	19
3	Al-Quran app	Estrutural	11
4	Sapawarga	Estrutural	230
5	Spartathlon	Estrutural	49
6	Travel rates	Estrutural	68
7	Stencil	Estrutural	1337
8	Minds mobile	Estrutural	399
9	WeightTracker	Estrutural	68
10	Flitter	Estrutural	16

Fonte: Autoria própria

Os projetos também foram classificados quanto a técnica de teste estrutural

implementada, como mostra a Tabela 13.

Tabela 13: Projetos coletados da técnica de teste funcional

Código	Nome	Técnica	Quantidade de testes
1	Beep	Funcional	16
2	Words Reminder	Funcional	38

Fonte: Autoria própria

6 ANÁLISE DE RESULTADOS E DISCUSSÕES

Neste capítulo são apresentados os resultados obtidos por meio das execuções do estudo de caso descrito na seção anterior, as respostas relacionadas as questões de pesquisa criadas no GQM, as descobertas e recomendações, ameaças à validade e, por fim, os trabalhos futuros.

O estudo de caso foi executado e avaliado em dois cenários de execuções C1 e C2 descritos na Seção 5.2.

6.1 CENÁRIO 1 (C1): 100 EXECUÇÕES REPETIDAS

No C1 foram executados todos os testes automatizados de cada projeto, utilizando a variável independente NRE, com o valor de 100 execuções.

Para cada um dos 12 projetos, foi executado o *Script* de execução (Figura 13), que realizou 100 vezes o processo de executar todos os testes automatizados. Diante das pré-condições estabelecidas, dentre os 12 projetos, somente um deles teve a ocorrência de testes quebradiços, enquanto nos outros projetos, todos os testes passaram nas 100 execuções.

6.2 CENÁRIO 2 (C2): 200 EXECUÇÕES REPETIDAS

No C2 foram executados todos os testes automatizados de cada projeto, utilizando a variável independente NRE, com o valor de 200 execuções.

Para cada um dos 12 projetos, foi executado o *Script de execução*(Figura 13), que realizou 200 vezes a execução de todos os testes automatizados.Diante das pré-condições estabelecidas, dentre os 12 projetos, somente um deles teve a ocorrência de testes quebradiços similar ao modo como ocorreu no C1.

6.2.1 ANÁLISE DOS RESULTADOS

Nos dois cenários, diante dos ambientes elaborados, somente o projeto 12 mostrou a ocorrência de testes quebradiços. Este projeto conta com 38 testes automatizado e todos da técnica funcional. Na Tabela 14 e 15 pode-se visualizar uma amostra parcial de dados das 100 execuções e 200 execuções respectivamente. As referidas tabelas contém dados sobre a quantidade de testes que passaram, falharam e também a quantidade de testes total, por fim o tempo total em segundos gasto de cada execução. As planilhas completas estão disponíveis [1]online. .

Tabela 14: Resultados das 100 execuções do C1.

Execução	Quantidade de testes que passaram	Quantidade de testes que falharam	Quantidade testes	Tempo total (segundos)
0	38	0.0	38	575.75
1	38	0.0	38	568.71
2	38	0.0	38	616.95
3	38	0.0	38	500.18
4	38	0.0	38	599.81
5	38	0.0	38	690.65
6	38	0.0	38	633.23
7	38	0.0	38	633.23
8	38	0.0	38	565.07
9	38	0.0	38	567.43
10	38	0.0	38	557.82
11	38	0.0	38	552.91
12	37	1.0	38	543.70
13	38	0.0	38	558.63
14	38	0.0	38	551.16
15	38	0.0	38	556.12
16	38	0.0	38	602.75
17	38	0.0	38	563.87
18	38	0.0	38	520.69
19	38	0.0	38	533.36
20	38	0.0	38	712.56

Fonte: Autoria própria

[1] acesse:https://drive.google.com/open?id=1I1ba2qYwrTf2N7qtHQqn-iVysTt-NWDu

Tabela 15: Resultados das 200 execuções do C2.

Execução	Quantidade de testes que passaram	Quantidade de testes que falharam	Quantidade testes	Tempo total (segundos)
0	38	0.0	38	509.17
1	37	1.0	38	585.58
2	37	1.0	38	571.76
3	38	0.0	38	522.19
4	38	0.0	38	523.41
5	33	5.0	38	436.20
6	38	0.0	38	518.62
7	36	2.0	38	512.72
8	35	3.0	38	465.63
9	37	1.0	38	569.00
10	37	1.0	38	542.88
11	38	0.0	38	541.17
12	38	0.0	38	561.28
13	38	0.0	38	584.79
14	36	2.0	38	581.36
15	37	1.0	38	494.10
16	38	0.0	38	533.31
17	38	0.0	38	582.66
18	37	1.0	38	638.42
19	37	1.0	38	541.71
20	38	0.0	38	532.04

Fonte: Autoria própria

Na Tabela 14 pode-se notar um exemplo parcial, mostrando que em uma ocasião das 100 execuções realizadas, foi identificada como teste quebradiço. Por outro lado na Tabela 15, também mostra um exemplo parcial, 19 execuções de um total de 200 foram identificadas como testes quebradiços. As amostras completas disponíveis online e mostram que no C1 22 execuções de 100 são consideradas testes quebradiços e para o C2 75 execuções de 200 são consideradas testes quebradiços.

Os testes automatizados do C1 e C2 que falharam foram selecionados e analisados quanto ao seu código implementado. Também foram identificadas quantas vezes cada teste falhou dentre os dois cenários de execuções, como mostram as Tabelas 16 e 17.

Tabela 16: Testes automatizados classificados como quebradiços do C1: projeto 12.

Código	Teste	Quantidade de falhas
1	*Should navigate to the screen to create a word*	8
2	*Should display an error for email already in use*	4
3	*Should sign up a new user*	3
4	*Should navigate to the screen to edit the word*	1
5	*Should not submit the form if value is empty*	1
6	*Should display a message*	1
7	*Should delete the word*	7
8	*Should navigate to the screen to create a dictionary*	1
9	*Should navigate to the screen to edit the dictionary*	1
10	*Should delete the dictionary*	1
11	*Should display the words list*	1

Fonte: Autoria própria

Tabela 17: Testes automatizados classificados como quebradiços do C2: projeto 12.

Código	Teste	Quantidade de falhas
1	*Should delete the word*	65
2	*Should display an error for email already in use*	5
3	*Should navigate to the screen to create a word*	9
4	*Should navigate to the screen to edit the word*	2
5	*Should redirect to signin*	3
6	*Should display an error for invalid email*	3
7	*Should display an error for weak password*	3
8	Should *navigate to the screen to edit the dictionary*	1
9	*Should delete the dictionary*	2

Fonte: Autoria própria

Comparando os testes quebradiços das Tabelas 16 e 17, pode-se observar que no C2 surgiram três novos testes quebradiços (Testes dos códigos 5, 6 e 7 da Tabela 17) que não haviam ocorrido no C1. Por outro lado, na execução do C1 alguns testes quebradiços (Teste dos códigos, 11, 6, 5 e 8 da Tabela 16) que haviam ocorridos não foram evidenciados

novamente no C2.

Na Figura 22, pode-se analisar a mensagem emitida no exemplo de uma execução com erro do Teste Quebradiço 1 da Tabela 16. Pode-se observar que no *log* de erro gerado ao executar o teste, é informado que não foi possível acessar o elemento esperado na tela. Esse problema ocorre, devido ao fato de que o elemento esperado ainda não carregou na tela, ocasionando erro ao tentar acessar esse elemento. No próprio *log* de erro, uma sugestão de correção é dada, informando para verificar a existência do elemento antes de tentar acessá-lo. Por meio da mensagem de erro, observou-se que o teste automatizado que falhou pode ser categorizado como teste quebradiço ocasionado por espera assíncrona.

Figura 22: Exemplo do log de erro para espera assíncrona: projeto 12.

```
WordsScreen screen
  ✕ should navigate to the screen to create a word (3097ms)
  with words
    ✓ should display the words list (4339ms)
    ✓ should navigate to the screen to edit the word (5700ms)
    ✓ should delete the word (7667ms)
  without words
    ✓ should display a message (2687ms)

• WordsScreen screen › should navigate to the screen to create a word

  Cannot find UI element.
  Exception with Action: {
    "Action Name":  "Tap",
    "Element Matcher":  "((!(kindOfClass('RCTScrollView')) && ((respondsToSelector(accessibilityIdentifier) &&
accessibilityID('dictionary-OAt7inSmgcCDxIECajr9')) && !(kindOfClass('UIAccessibilityTextFieldElement')))) ||
(((kindOfClass('UIView') || respondsToSelector(accessibilityContainer)) && parentThatMatches(kindOfClass('RCTScrollView'))) &&
((kindOfClass('UIView') || respondsToSelector(accessibilityContainer)) &&
parentThatMatches(((respondsToSelector(accessibilityIdentifier) && accessibilityID('dictionary-OAt7inSmgcCDxIECajr9')) && !
(kindOfClass('UIAccessibilityTextFieldElement')))))))",
    "Recovery Suggestion":  "Check if the element exists in the UI hierarchy printed below. If it exists, adjust the matcher so
that it accurately matches element."
  }

  Error Trace: (
    {
      "Description":  "Interaction cannot continue because the desired element was not found.",
      "Error Domain":  "com.google.earlgrey.ElementInteractionErrorDomain",
      "Error Code":  "0",
      "File Name":  "GREYElementInteraction.m"
```

Fonte: Autoria própria

Todas as falhas dos testes quebradiços encontrados no C1 e C2 foram classificados em categorias quanto às causas como mostra a Tabela 18. As categorias utilizadas, foram as mesmas identificadas na Seção 3.5, por meio das pesquisas realizadas na literatura.

Tabela 18: Testes quebradiços com suas causas para o C1 e C2: projeto 12.

Código	Causas encontradas	Quantidade de falhas
1	Espera assíncrona	19
2	Causas não identificadas	109

Fonte: Autoria própria

Dentro das avaliações realizadas nos testes quebradiços encontrados, somente duas categorias de causas foram identificadas: (1) espera assíncrona e (2) causas não

identificadas. Para a primeira categoria, 19 falhas dos testes foram identificados, enquanto para a segunda categoria, 109 falhas dos testes não tiveram uma causa aparentemente identificada.

Em uma análise sucinta acerca dos testes quebradiços ocasionados por motivos desconhecidos, nota-se que uma análise minuciosa por parte da equipe de desenvolvimento pode ser suficiente para elucidar as causas/motivos de ocorrência. Isso promoveria ações para prevenção e correção de testes quebradiços.

Na Figura 23 pode-se observar o código implementado do Teste Quebradiço 1 categorizado como espera assíncrona da Tabela 16. Esse código conta com um método que recebe dois parâmetros (*"dictionaryId" e "wordId"*). Na Linha 11 o código navega até a tela de *WordScreen*, após isso na Linha 12 aguarda-se o elemento com o valor do parâmetro *wordId* sofrer a execução de um *swipe*. Após a execução da Linha 12, na Linha 13 aguarda-se a execução do processo de *Tap*(pressionar um botão) na tela e, por fim, na Linha 14 aguarda-se o elemento de edição da tela ficar visível para liberar a edição na tela.

Figura 23: Exemplo de código de um teste automatizado de espera assíncrona.

```
import { element, waitFor, by } from 'detox'
import { navigateToWordsScreen }
from '@e2e/navigation/navigate-to-words-screen'
import { WORDS_ROW, WORDS_ROW_EDIT,
WORD_EDIT_SCREEN } from '@e2e/ids'

const navigateToEditWordScreen = async (
  dictionaryId: string,
  wordId: string
) => {
  await navigateToWordsScreen(dictionaryId)
  await element(by.id(WORDS_ROW(wordId))).swipe('left')
  await element(by.id(WORDS_ROW_EDIT(wordId))).tap()
  await waitFor(element(by.id(WORD_EDIT_SCREEN)))
    .toBeVisible()
    .withTimeout(1000)
}
export { navigateToEditWordScreen }
```

Fonte: Autoria própria

6.3 RESPOSTAS DAS QUESTÕES DE PESQUISA

Nesta seção são apresentadas as considerações acerca das respostas das questões de pesquisa criadas no GQM proposto. As respostas foram obtidas por meio da coleta e análise dos resultados dos Cenários de execuções C1 e Cenário C2 do estudo de caso.

Na Tabela 19 os resultados coletados foram classificados de acordo com as métricas criadas no GQM para o Cenário C1 com o objetivo de responder as questões de pesquisas.

Tabela 19: Métricas do GQM coletadas para o C1.

Projeto	% M1	% M2	% M3	M4		% M5	% M6
				% Espera assíncrona	% Causa não identificada		
Inkino	0	100	0	0	0	0	0
Inkino mobile	0	100	0	0	0	0	0
Quran app	0	100	0	0	0	0	0
Sapawarga	0	100	0	0	0	0	0
Spartathlon	0	100	0	0	0	0	0
Words reminder	28.9	100	28.9	9	91	0	100
Travel rates	0	100	0	0	0	0	0
Stencil	0	100	0	0	0	0	0
Minds mobile	0	100	0	0	0	0	0
WeightTracker	0	100	0	0	0	0	0
Beep	0	100	0	0	0	0	0

Fonte: Autoria própria

Na Tabela 20 os resultados coletados foram classificados de acordo com as métricas criadas no GQM para o Cenário C2 com o objetivo de responder as questões de pesquisas.

Tabela 20: Métricas do GQM coletadas para o C2.

Projeto	% M1	% M2	% M3	M4		% M5	% M6
				% Espera assíncrona	% Causa não identificada		
Inkino	0	100	0	0	0	0	0
Inkino mobile	0	100	0	0	0	0	0
Quran app	0	100	0	0	0	0	0
Sapawarga	0	100	0	0	0	0	0
Spartathlon	0	100	0	0	0	0	0
Words reminder	23.6	100	23.6	11.2	88.8	0	100
Travel rates	0	100	0	0	0	0	0
Stencil	0	100	0	0	0	0	0
Minds mobile	0	100	0	0	0	0	0
WeightTracker	0	100	0	0	0	0	0
Beep	0	100	0	0	0	0	0

Fonte: Autoria própria

Q1: Existem sinais da ocorrência de testes quebradiços em projetos reais de aplicativos híbridos?

Foi concluído que essa questão de pesquisa foi respondida totalmente por meio da análise dos resultados coletados na execução do estudo de caso. No Cenário C1 e C2 os resultados coletados mostraram a presença de testes quebradiços em aplicativos híbridos.

Em ambos os Cenários foram analisados 12 projetos e dentre eles um projeto mostrou a ocorrência de testes quebradiços, tanto no C1 quanto no C2. Por meio das Tabelas 19 e 20 observa-se que do total de 38 testes automatizados 11 deles (28.9%) foram classificados como teste quebradiço de acordo com a Métrica M1 para o C1 e 9 (23.6%) para o C2. Além disso, a Métrica M2 mostra que 100% dos testes passaram em pelo menos uma execução tanto para o C1, quanto para o C2 e também a Métrica M3 nos informa que 28.9% para o C1 e 23.6% para o C2 dos testes em pelo menos uma das execuções falhou.

Q2: Quais às possíveis causas de testes quebradiços em projetos de aplicativos híbridos?

Foi concluído que esta questão de pesquisa foi respondida parcialmente, uma vez que poucas categorias de causas de testes quebradiços foram identificadas se comparado com as pesquisas da literatura. Nos Cenários C1 e C2 foi possível identificar somente duas categorias de causas para os testes quebradiços identificados, sendo elas espera assíncrona e causas não identificadas (THORVE; SRESHTHA; MENG, 2018; LUO et al., 2018; BELL OWOLABI LEGUNSEN, 2018).

No Cenário C1, por meio da Métrica M4, foi possível identificar um (9%) teste quebradiço relacionado a espera assíncrona e dez (91%) para causas não identificadas em um total de 11 testes quebradiços identificados. No Cenário C2 por meio da Métrica M4 foi possível identificar 1 (11.2%) teste quebradiço relacionado a espera assíncrona e 8 (88.8%) para causas não identificadas em um total de 9 testes quebradiços identificados. Por fim, destaca-se que mais estudos são necessários para que a questão possa ser respondida de modo mais completo.

Q3: Casos de teste baseados em qual técnica de teste (funcional ou estrutural) são mais propensos a gerar testes quebradiços em projetos híbridos?

Foi concluído que esta questão de pesquisa foi respondida totalmente, uma vez que todos os testes quebradiços identificados foram encontrados em projetos que contemplavam a técnica de teste funcional.

No Cenário C1 e Cenário C2, por meio das Métricas M5 e M6, mostram que o único projeto em que ocorreram testes quebradiços contava com a técnica funcional implementada. Dentre todos os projetos analisados, a maioria deles contava com implementação da técnica estrutural, fazendo com que existissem mais análises voltadas para testes estruturais, do que para testes funcionais.

6.4 CONTRIBUIÇÕES, DESCOBERTAS E RECOMENDAÇÕES

A partir da condução dos estudos empíricos executados durante esse trabalho, diversas contribuições, descobertas e recomendações puderam ser consideradas.

Por meio da execução do estudo de caso e a coleta de seus resultados, permitiu-se descobrir que a técnica funcional é mais favorável a ocorrer testes quebradiços em aplicativos híbridos, se comparada com a técnica estrutural. Também foi descoberto, que por meio da dependência entre elementos na técnica funcional, os testes quebradiços encontrados e identificados eram, na sua maioria, de causas não identificadas, mas aqueles que puderam ser identificados, todos eles foram classificados como espera assíncrona.

Uma das principais dificuldades encontradas para a realização do estudo de caso, foi encontrar projetos de aplicativos híbridos reais com testes de software automatizados implementados com código aberto para uso durante as avaliações e execuções. Considerando os 26 projetos encontrados inicialmente, surgiram muitos problemas, tanto nas execuções dos testes quanto na instalação de dependências, o que justifica a escolha final de 12 projetos selecionados para o estudo de caso.

Outro problema identificado durante o estudo de caso, foi a demora da execução dos testes automatizados. Nos dois Cenários (C1 e C2) criados, houve uma certa demora para executar os testes e também avaliar todos os resultados obtidos para 100 e 200 execuções.

O presente estudo contribui também, como o primeiro esforço de pesquisa relacionado a testes quebradiços em aplicativos híbridos. Essa abordagem focada em um domínio específico, pode ser replicada para qualquer outro contexto de testes quebradiços. Além disso, os resultados coletados e analisados, podem auxiliar como *baseline* para futuras pesquisas, ferramentas criadas entre outros estudos. Não somente para estudos, mas o estudo também pode auxiliar organizações que possam a vir sofrer com os testes quebradiços em aplicativos híbridos e outros contextos parecidos.

6.5 AMEAÇAS À VALIDADE

Considerando o modelo de análise de ameaças à validade sugerido em Wohlin et al. (2012), a seguir é apresentada uma análise acerca do estudo realizado.

Validade de conclusão: (análise associada à certeza das conclusões do estudo) considera-se que essa validade gera baixa ameaça aos estudos conduzidos, uma vez que um estudo de caso foi executado e validações estatísticas não foram realizadas;

Validade interna: (análise associada entre procedimentos experimentais e descobertas) considera-se que esse nível de ameaças é alto em relação às descobertas. Essa análise é feita, principalmente, por conta dos seguintes motivos:

1. replicações de apenas um cenário para gerar testes quebradiços. Acredita-se que a implementação de cenários simuladores de possíveis motivos de geração de quebradiços podem alterar os resultados apresentados. Devido à limitação de tempo e recurso para simular mais cenários, pouco foi feito para aliviar essa ameaça. Estudos futuros irão considerar esses aspecto;
2. alguns dos projetos analisados não tinham cenários favoráveis para a criação de testes quebradiços; e
3. baixo número de projetos reais analisados devido às limitações de tempo.
4. utilização de ambientes diferentes, com máquinas diferentes.

Validade de construção: (análise associada entre a hipótese e as descobertas) as ameças associadas a esse tópico são baixas devido ao fato de que o estudo foi conduzido de modo observacional. Portanto, nenhuma ação por parte dos pesquisadores precisou ser feita; e

Validade externa: (análise associada à generalização do estudo para outros escopos) as ameaças nessa modalidade são consideradas de nível médio. Para diminuir essas ameaças, os pesquisadores documentaram de modo amplo o estudo. Assim, pesquisadores podem replicar o planejamento empírico realizado em projetos com outros escopos.

6.6 TRABALHOS FUTUROS

Após a execução do estudo proposto, alguns trabalhos futuros surgiram como proposta de continuação do estudo realizado. Tais propostas de investigações são em torno de esclarecer e encontrar mais evidências dos testes quebradiços nos aplicativos híbridos e também em outros tipos de software.

Uma forma de abordagem para a identificação dos testes quebradiços é criar ambientes controlados na tentativa de forçar a ocorrer diversos testes quebradiços de

diversas causas. Essa abordagem pode ser aplicada tanto para aplicativos híbridos, assim como, qualquer outro tipo de projeto. Sendo assim, um trabalho futuro seria a realização do estudo em outros escopos.

Por fim, outra abordagem futura a ser feita, é criar um ambiente de IC com algum projeto de grande porte e simular um ambiente real de grandes empresas. Nesse ambiente realizar simulações com diversas implementações de funcionalidades novas, para averiguar o comportamento da execução dos testes automatizados a cada *deploy* realizado.

7 CONSIDERAÇÕES FINAIS

O presente estudo concentra-se na área de Teste de Software, contribuindo com o problema dos testes quebradiços em testes automatizados. Os testes quebradiços são considerados problemas frequentes em projetos de desenvolvimento software devido ao resultado incerto dos testes e podem gerar problemas para toda equipe envolvida no desenvolvimento. Os problemas mais graves envolvem atrasos no cronograma e também aumento do custo final do projeto. Atualmente, poucos estudos focados nos testes quebradiços foram realizados como mostra o Capítulo 3.

Diante disso, o estudo realizado gera informações para contribuir com a identificação e alívio dos testes quebradiços em aplicativos híbridos. Para isso, um estudo de caso foi realizado, executando testes automatizados em diversos projetos de aplicativos híbridos. Após a execução, os testes quebradiços identificados foram classificados quanto à causa e os projetos de acordo com a técnica de testes de software utilizada.

Por fim, visando contribuições, os resultados obtidos durante a execução do estudo de caso mostram que os testes quebradiços ocorrem em aplicativos híbridos. Também afirmou-se que os testes quebradiços estão mais associadas a projetos que utilizam a técnica de teste funcional, se comparada com a técnica estrutural. As contribuições possibilitam uma transferência tecnológica do conteúdo da academia para a indústria, por meio do consumo das informações levantadas. Para a academia, as informações obtidas e sintetizadas ajudam futuros trabalhos e promovem novos estudos sobre os testes quebradiços. Para a indústria, as informações obtidas auxiliam em projetos de software que possam vir a sofrer com os testes quebradiços, fornecendo dados de como ocorrem os testes quebradiços para possíveis prevenções.

REFERÊNCIAS

AES, R. D. **Alaska Airlines usando Visual Studio para aplicativos Android / IOS**. 2016. Disponível em: <http://www.ramonduraes.net/2016/05/31/alaska-airlines-usando-visual-studio-para-aplicativos-android-ios/>. Acesso em: 20 de junho de 2019.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 2^{th}. ed. Cambridge: Cambridge University Press, 2016.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. In: **Encyclopedia of Software Engineering**. [S.l.]: Wiley, 1994.

BELL OWOLABI LEGUNSEN, M. H. L. E. T. Y. D. M. J. DeFlaker: automatically detecting flaky tests. **ICSE '18 Proceedings of the 40th International Conference on Software Engineering**, p. 433–444, 2018.

BERNARDO, F. K. P. C. A importância dos testes automatizados. **Engenharia de Software Magazine**, n. 3, p. 54–57, 2008.

CHEDE, C. **Desenvolvimento de apps – Parte 2: híbrido, nativo ou web?** 2016. Disponível em: <https://www.ibm.com/developerworks/community/blogs/ctaurion/entry/desenvolvimento-_de_apps-parte_2_hibrido_nativo_ou_web?lang=en>. Acesso em: 13 de maio de 2019.

DELAMARO, M. **Introdução Ao Teste De Software**. 2^{th}. ed. Brasil, Rio de Janeiro: Elsevier Editora, 2016. 448 p.

DEMILLO, R. Mutation analysis as a tool for software quality assurance. p. 7, 10 1980.

DEVELOPERS, G. **FlakyTest**. Disponível em: <https://developer.android.com/reference/-android/test/FlakyTest.html>. Acesso em: 26 de maio de 2019.

FOLLOW, N. K. N. G. **Building Hybrid Android Apps with Java and JavaScript**. 1^{th}. ed. Sebastopol, Califórnia, EUA: O'Reilly Media, 2013. 156 p.

FOWLER, M. **Continuous Integration**. 2006. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 25 de maio de 2019.

FOWLER, M. **Eradicating Non-Determinism in Tests**. 2011. Disponível em: <https://martinfowler.com/articles/nonDeterminism.html>. Acesso em: 26 de maio de 2019.

GAO, A. M. M. Z. Which of my failures are real? using relevance ranking to raise true failures to the top. In: . Lincoln, NE, USA: [s.n.], 2015. p. 62–69.

GAO, Z. **Quantifying flakiness and minimizing its effects on softwre testing**. Tese (Doutorado) — University of Maryland, Washington, DC, 2017.

GEORGE, M. **RSpec: crie especificações executáveis em Ruby**. 1^{th}. ed. Los Alamitos, CA: Editora Casa do Código, 2014. 160 p.

GITHUB. **GitHub**. 2019. Disponível em: <https://github.com/>. Acesso em: 28 de outubro de 2019.

GITLAB. **GitLab**. 2019. Disponível em: <https://about.gitlab.com/>. Acesso em: 28 de outubro de 2019.

GOK, N.; KHANNA, N. **Building Hybrid Android Apps with Java and JavaScript**. Sebastopol: O'Reilly Media, 2013.

GOOGLE. **Androis Studio**. 2019. Disponível em: <https://developer.android.com/studio>. Acesso em: 28 de outubro de 2019.

GOOGLE. **Google Sheets**. 2019. Disponível em: <https://www.google.com/sheets/about/>. Acesso em: 28 de outubro de 2019.

GUPTA, M. I. P.; PENIX, J. **Testing at the speed and scale of Google**. 2011. Disponível em: <http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html>. Acesso em: 18 de maio de 2019.

HEITKÖTTER, H.; HEIERHOFF, S.; MAJCHRZAK, T. A. Evaluating cross-platform development approaches for mobile applications. In: . Porto, Portugal: [s.n.], 2013. p. 120–138.

HERZIG, K.; NAGAPPAN, N. Empirically detecting false test alarms using association rules. In: PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. Piscataway, NJ, USA, 2015. p. 39–48.

IONIC, F. **Insights, trends, and perspectives from the worldwide Ionic Community**. 2017. Disponível em: <https://ionicframework.com/survey/2017trends>. Acesso em: 20 de junho de 2019.

ISO/IEC/IEEE 24765: 2010(E): Systems and Software Engineering - Vocabulary. 1^{th}. ed. [S.l.]: IEEE.

JETBRAINS. **IntelliJ IDEA**. 2019. Disponível em: <https://www.jetbrains.com/idea/>. Acesso em: 28 de outubro de 2019.

KING DIONNY SANTIAGO, J. P. P. J. C. T. M. Towards a bayesian network model for predicting flaky automated tests. In: . Lisboa, Portugal: [s.n.], 2018. p. 100–107.

LABUSCHAGNE, A.; INOZEMTSEVA, L.; HOLMES, R. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In: . New York, NY, USA: [s.n.], 2017. p. 821–830.

LUO, Q. et al. An empirical analysis of flaky tests. **FSE 2014 Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**, p. 643–653, 2018.

MCCABE, T. J. A complexity measure. **IEEE Transactions on Software Engineering**, n. 4, p. 308–320, 1976.

MEMON, A. **Get the Most Out of Test Automation at Scale: "Listen to Your Data"**. 2017. Disponível em: <https://www.youtube.com/watch?v=gJYPS40q2G4>. Acesso em: 26 de maio de 2019.

MICCO, J. **Flaky Tests at Google and How We Mitigate Them**. 2016. Disponível em: <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html>. Acesso em: 21 de maio de 2019.

MICROSOFT. **Visual Studio Code**. 2019. Disponível em: <https://code.visualstudio.com/>. Acesso em: 28 de outubro de 2019.

MOLINARI, L. **Testes Funcionais De Software**. 1^{th}. ed. Florianópolis, Brasil: Visual Books, 2008. 250 p.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3^{th}. ed. USA: Wiley Publishing, 2011. 256 p.

NETO, A. C. D. Introdução a teste de software. **Engenharia de Software Magazine**, n. 1, p. 55–59, 2007.

NEVES, L. **PROJETO DE UMA METODOLOGIA DE TESTES PARA SISTEMAS DE INFORMAÇÃO CORPORATIVA**. Monografia (Pós-Graduação) — ICSP, Curitiba, PR, Brasil, 1999.

PALOMBA, F.; ZAIDMAN, A. Does Refactoring of Test Smells Induce Fixing Flaky Tests?. **2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)**, p. 1–12, 2017.

PEZZÈ, M.; YOUNG, M. **Teste e Análise de Software: Processos, Princípios e Técnicas**. 1^{th}. ed. [S.l.]: Grupo A - Bookman, 2008.

PRESSMAN, R. **Engenharia de software**. 6^{th}. ed. New York City: McGraw-Hill, 2006. 720 p.

ROPER, M. **Software testing**. 1^{th}. ed. [S.l.]: McGraw-Hill Ryerson, Limited, 1994. 149 p.

ROPER, M. **Software Testing**. 1^{th}. ed. New York, NY, USA: McGraw-Hill, Inc., 1995. 146 p.

SASSO, K. et al. Revisão integrativa: Metodo de pesquisa para a incorporação de evidências na saude e na enfermagem integrative literature review: A research method to incorporate evidence in health care and nursing revision integradora: Metodo de investigacion para la incorporacin de evidencias en la salud y la enfermeria. **Texto Contexto - Enfermagem**, p. 758–764, 2008.

SHI ALEX GYORI, O. L. D. M. A. Detecting assumptions on deterministic implementations of non-deterministic specifications. In: . Chicago, USA: [s.n.], 2016. p. 80–90.

SOCIETY, I. C. **SWEBOK: Guide to the Software Engineering Body of Knowledge**. 3^{th}. ed. Los Alamitos, CA: IEEE Computer Society, 2014.

SOMMERVILLE, I. **Software Engineering**. 9^{th}. ed. São Paulo: Pearson, 2011. 529 p.

STÂHL, D.; BOSCH, J. Continuous integration flows. **Continuous software engineering**, p. 107–115, 2014.

TAVARES, H. L. Introdução a desenvolvimento de aplicações hÍbridas. **Revista Eletrônica e-F@tec**, v. 6, n. 1, 2016.

THORVE, S.; SRESHTHA, C.; MENG, A. N. An Empirical Study of Flaky Tests in Android Apps. **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**, p. 534–538, 2018.

TOMLIN, N. **Protractor Flake**. 2019. Disponível em: <https://www.npmjs.com/package/protractor-flake>. Acesso em: 20 de maio de 2019.

WAHLSTRÖM, M. **Exploring progressive web applications for health care : Developing a PWA to gather patients' self assessments**. Dissertação (Mestrado) — Umea University, Department of Applied Physics and Electronics, Umeå, Suécia, 2017.

WOHLIN, C. et al. **Experimentation in Software Engineering**. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435.

ZHANG, S. et al. Empirically revisiting the test independence assumption. In: . New York, NY, USA: [s.n.], 2014. p. 385–396.