PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

JOSIAS GOMES LIMA

SCoTUAM: UMA ABORDAGEM PARA SELEÇÃO DE COMPONENTES PARA TESTES UNITÁRIOS EM APLICAÇÕES MÓVEIS

Manaus
2018

Josias Gomes Lima

SCoTUAM: UMA ABORDAGEM PARA SELEÇÃO DE COMPONENTES PARA TESTES UNITÁRIOS EM APLICAÇÕES MÓVEIS

Proposta de Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, PPGI, da Universidade Federal do Amazonas como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador: Prof. Dr. Arilo Claudio Dias Neto

Manaus
2018

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Lima, Josias Gomes
L732s SCoTUAM: Uma Abordagem para Seleção de Componentes para Testes Unitários em Aplicações Móveis / Josias Gomes Lima. 2018
100 f.: il. color; 31 cm.

Orientador: Arilo Claudio Dias Neto
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. teste de aplicações móveis. 2. testes automatizados. 3. teste de unidade. 4. seleção de componentes. I. Dias Neto, Arilo Claudio II. Universidade Federal do Amazonas III. Título

FOLHA DE APROVAÇÃO

"SCoTUAM: Uma Abordagem para Seleção de Componentes para Testes Unitários em Aplicações Móveis"

JOSIAS GOMES LIMA

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituida pelos Professores:

Arilo Claudio Dias Neto
Prof. Arilo Claudio Dias Neto - PRESIDENTE

Prof. José Reginaldo Hughes Carvalho - MEMBRO INTERNO

Prof. Eduardo Noronha de Andrade Freitas - MEMBRO EXTERNO

Manaus, 31 de Agosto de 2018

A Deus, a minha família e amigos.

Agradecimentos

A Deus, pelo dom da vida, e por ter colocado pessoas em minha vida que me ajudaram a concluir mais essa etapa.

A minha família, meus pais Antonia e Felix, meus irmãos David, Allan, Mayra e Nayra, minha noiva Thaynara, minha cunhada Bianca, meus primos Jonas e Jhonatan e meus irmãos de coração Jodi e Moisés.

Aos meus amigos da graduação que me incentivaram na entrada para o mestrado, Verônica, Arcanjo e Francisco.

Aos professores da graduação Sancha e Orlewilson por terem feito uma carta de recomendação para o meu ingresso no mestrado.

Ao professor Arilo, por ter aceitado ser meu orientador, e por suas valiosas orientações no decorrer do mestrado.

Ao professor Eduardo Noronha, por ter me ajudado em muitos momentos de dúvidas durante o mestrado.

Aos meus amigos do grupo de pesquisa ExperTS, Karina, Silvia, Awdren, Laiza, Oswald, Larissa, Kariny, Renata, Ivanilse, Jonathas e PH Munhoz.

A todos os participantes que fizeram parte do estudo experimental, obrigado pelo tempo e contribuição.

Aos professores e corpo administrativo do Instituto de Computação pelo suporte durante o tempo do mestrado.

A FAPEAM pelo apoio financeiro.

E a todos que de alguma forma contribuíram para esta pesquisa.

Resumo de Dissertação apresentada à UFAM/AM como parte dos requisitos necessários para a obtenção do grau de Mestre em Informática (M.Sc.)

SCoTUAM: UMA ABORDAGEM PARA SELEÇÃO DE COMPONENTES PARA TESTES UNITÁRIOS EM APLICAÇÕES MÓVEIS

Josias Gomes Lima

Agosto / 2018

Orientador: Prof. Dr. Arilo Claudio Dias Neto

O teste de unidade é o nível de teste de software pelo qual partes individuais do código fonte são testadas. A realização deste tipo de teste traz alguns benefícios, tais como redução de falhas em recursos já existentes, melhoram a estrutura do código, diminuem os efeitos colaterais (*side effects*) e reduzem o medo da alteração do código (Burke e Coyner, 2017). No entanto, a atividade de teste para aplicações móveis tem o tempo reduzido, fazendo com que alguns desenvolvedores optem por não criar os testes de unidade. O tempo reduzido faz com que a automatização dos testes se torne uma necessidade. Nesse contexto, este trabalho propõe um plugin para auxiliar os desenvolvedores na seleção de componentes que tenham um maior valor em relação ao custo x benefício do teste de unidade em aplicações móveis da plataforma Android. Para medir o valor do custo e benefício dos componentes, foram escolhidas as seguintes métricas: *halstead effort* (HE), custo de manutenção futura (CMF), cheiros de código (CS), frequência de chamadas (FC), risco de falhas (RF), vulnerabilidade de mercado (VM) e valor de negócio VN. O plugin proposto possui três processos principais: (1) Extração de métricas estáticas, (2) Extração de métricas dinâmicas, de mercado e de negócio e (3) Execução do algoritmo genético para seleção dos componentes a serem testados. O plugin chamado SCoTUAM pode ser adicionado à interface de desenvolvimento da IDE Android Studio. Neste trabalho foram realizados dois estudos empíricos para avaliação do plugin proposto. No primeiro estudo, o propósito foi analisar a correlação das métricas, onde o resultado mostrou a possibilidade de usar as métricas CMF, CS, FC, RF, VM e VN combinadas em uma solução multiobjetivo. No segundo estudo, o objetivo foi analisar a eficácia do plugin em selecionar componentes com erro comparado com a seleção manual realizada por especialistas em teste de unidade em aplicações móveis Android, onde o resultado mostrou a viabilidade da proposta em auxiliar o desenvolvedor na seleção de componentes para o teste de unidade.

Palavras-chave: teste de aplicações móveis; testes automatizados; teste de unidade; seleção de componentes.

Abstract of Thesis presented to UFAM/AM as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

SCoTUAM: AN APPROACH FOR COMPONENTS SELECTING FOR UNIT TESTING IN MOBILE APPLICATIONS

Josias Gomes Lima

August / 2018

Advisor: Prof. Dr. Arilo Claudio Dias Neto

The unit test is the level of software testing by which individual parts of the source code are tested. Implementing this type of test brings some benefits such as reducing failures in existing resources, improving code structure, decrease side effects, and reducing fear of code change (Burke and Coyner, 2017). However, the test activity for mobile applications is time-consuming, causing some developers to choose not to create unit tests. Reduced time makes testing automation a necessity. In this context, this work proposes a plugin to assist developers in selecting components that have a greater value in relation to the cost x benefit of the unit test in mobile applications of the Android platform. To measure the value of cost and benefit of components, the following metrics were chosen: halstead effort (HE), future maintenance cost (FMC), code smells (CS), call frequency (CF), risk of failures (RF), market vulnerability (MV) and business value (BV). The proposed plugin has three main processes: (1) Extraction of static metrics; (2) Extraction of dynamic, market and business metrics; and (3) Execution of the genetic algorithm to select the components to be tested. The proposed plugin called SCoTUAM can be added to the development interface of the Android Studio IDE. In this work two empirical studies were carried out. In the first study, the purpose was to analyze the correlation of the metrics, where the result showed the possibility of using the combined FMC, CS, CF, RF, MV and BV metrics in a multiobjective solution. In the second study, the objective was to analyze the plugin's effectiveness in selecting components with error compared to the manual selection performed by unit test specialists in Android mobile applications, where the result showed the feasibility of the proposal in assisting the developer in the selection of components for the unit test.

keywords: mobile testing; automated testing; unit test; components selection.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1 - INTRODUÇÃO

Neste capítulo serão apresentados o contexto, a motivação e o problema que será tratado nesta dissertação. Também serão apresentados a hipótese, os objetivos, a metodologia de pesquisa adotada e a organização deste trabalho.

1.1. Contextualização e Motivação

Com o decorrer dos anos, o crescimento tecnológico é percebido nos dispositivos móveis, ocasionando um grande impacto em várias áreas de aplicação, como bancária, redes sociais, entretenimento e comércio eletrônico, fazendo com que as aplicações móveis se tornassem necessárias e fundamentais no cotidiano das pessoas. Um indicador disto é que as pessoas estão cada vez mais dependentes de dispositivos móveis, seja para realizar uma transação bancária ou para acessar o e-mail (Rubinov e Baresi, 2018).

Além disso, no cenário atual, as aplicações móveis devem ser construídas o mais rápido possível para não se tornarem obsoletas em relação a seus concorrentes. Isso fez com que o tempo do ciclo de desenvolvimento de aplicações móveis diminuísse, impactando consequentemente o tempo de execução do teste, e fazendo com que ele se tornasse mais importante do que no caso de software para desktop (Samuel e Pfahl, 2016). Nesse cenário, por intermédio da automação é possível diminuir o tempo de execução do teste de software em relação ao tempo de execução de um processo de teste manual. Os testes automatizados podem beneficiar um projeto de desenvolvimento, possibilitando a execução de testes 24 horas por dia e 7 dias por semana, acelerações no processo de teste, e permitindo que os testes estejam mais frequentes (Øvergaard e Muller, 2013).

O cenário da automação de teste em plataformas móveis foca, em geral, em teste baseados na interface gráfica do usuário (GUI, do inglês *Graphical User Interface*). Nos últimos anos, várias abordagens surgiram com o propósito de implementar a automação baseada na GUI. A técnica mais eminente é *capture & replay*, que consiste de captura, em que a ferramenta de automação de teste supervisiona e analisa as atividades de execução manual de casos de teste, e repetição, no qual o script de teste baseado na GUI gerado pela técnica é executado sem a intervenção do usuário (Euteneuer *et al.*, 2013). No entanto, testes baseado na GUI são difíceis de serem mantidos atualizados. De acordo com Fowler (2012), teste por meio da interface do usuário é caro para escrever, aumenta o tempo de compilação e, mais importante ainda, tais testes são suscetíveis às mudanças, pois um aprimoramento do sistema pode facilmente acabar quebrando muitos desses testes, que, em seguida, tem

que ser regravado. Além disso, mesmo com boas práticas de escrita, os testes de GUI são mais propensos a problemas de não-determinismo, o que pode diminuir a confiança neles. Os testes de GUI também negligenciam importantes propriedades funcionais dos programas que fazem parte do seu projeto ou implementação e que não são descritas nos requisitos (Freitas *et al.*, 2016). Como alternativa, durante o processo de desenvolvimento, o teste de unidade é realizado antes do teste baseado na GUI. Assim, corrigir erros no nível de unidade é mais barato. Os autores Burke e Coyner (2017) apresentam alguns motivos para se fazer testes de unidade:

- **Redução de falhas em recursos já existentes**, pois se um novo recurso interromper a funcionalidade existente, os testes de unidade existentes falham imediatamente, permitindo que você identifique o problema e conserte;
- **Melhoram a estrutura do código**, pois o desenvolvedor é forçado a implementar um código testável;
- **Diminuem os efeitos colaterais (*side effects*)**, pois alguns erros podem ocorrer somente em cenários peculiares, e um novo programador corrigindo um erro poderá está introduzindo outro. No entanto, com os testes de unidade diminuem-se essa possibilidade;
- **Reduzem o medo**, pois um dos maiores medos que os programadores encontram é fazer uma mudança em um código e não saber o que vai quebrar. Ter um conjunto de testes de unidade permite aos programadores remover o medo de fazer alterações ou adicionar novos recursos.

O teste de unidade em aplicações móveis distingue-se do teste de unidade em algumas outras categorias de software, pois as aplicações móveis devem funcionar a qualquer hora e em qualquer lugar. Além disso, as aplicações devem funcionar adequadamente em plataformas que possuem, por exemplo, diferentes sistemas operacionais, tamanhos de exibição, recursos de computação, canais de entrada (ex: teclado, voz e gestos), diversos contextos de conectividade de rede e duração da bateria (Gao *et al.*, 2014).

1.2. Descrição do Problema

A atividade de teste para aplicações móveis tem o tempo reduzido. Portanto, a automação do teste é uma necessidade (Samuel e Pfahl, 2016). Uma etapa da automatização do teste é a seleção do código a ser testado. No entanto, poucos trabalhos auxiliam o testador

na escolha de um subconjunto de componentes[1] para testes (Freitas *et al.*, 2016). Antes de se criar a suíte de teste é necessário selecionar os componentes que serão testados, pois, em geral, testar todos os possíveis conjuntos de seleção de componentes é computacionalmente intratável, devido a restrições de tempo e recursos (Harman *et al.*, 2006).

A seleção de componentes para teste unitário é um problema de otimização multiobjetivo, dentro da área de teste de software baseado em busca (do inglês, *Search-Based Software Testing*, SBST), pois o testador procura simultaneamente diminuir o custo e maximizar o benefício fazendo a seleção de um subconjunto de componentes (Freitas *et al.*, 2016). O SBST tem atraído muita atenção nos últimos anos (Zhu *et al.*, 2017). Uma área dessa linha de pesquisa é o esforço de pré-teste, no qual seleciona-se o código para testes. Essa área de pesquisa tenta analisar os programas e selecionar o código, com o intuito de orientar a construção do teste para alcançar o efeito de cobertura máxima com base em vários critérios, pois uma dúvida frequente levantada antes da construção do teste é a de quais linhas do código devem ser testadas. Muitos critérios podem ser usados para selecionar o código para teste, como frequência de chamada, complexidade estrutural e cobertura de código em potencial (Li *et al.*, 2005).

Na Figura 1, é apresentada uma ilustração para o problema da seleção de componentes em uma aplicação móvel. Uma aplicação móvel (App) possui *N* componentes, e o desenvolvedor precisa escolher, baseado em sua experiência, quais os componentes que tem o melhor valor em relação ao custo x benefício para a criação do teste de unidade. Além da sua experiência, o desenvolvedor pode usar métricas[2] para escolher os componentes. Porém ele precisa ter o conhecimento prévio sobre essas métricas para selecionar o melhor conjunto de componentes.

Com base na necessidade de selecionar componentes e se ter o conhecimento prévio sobre as métricas, é que se torna oportuna a proposta desta pesquisa em auxiliar o desenvolvedor na escolha de componentes que devem ser testados, levando em consideração o valor de Custo x Benefício dos componentes para a realização do teste de unidade. Para esse problema, a plataforma Android foi escolhida para o desenvolvimento do plugin, pois ela representa 86.1% de todas as vendas de smartphones para usuários finais em todo o mundo no segundo quartil de 2017 (Statista, 2017a).

[1] Um componente, no contexto deste trabalho, será considerado como um método de uma classe de um programa orientado a objeto.

[2] Uma métrica de software é um padrão de medida de um grau ao qual um sistema ou processo de software possui alguma propriedade (Boehm *et al.*, 1976).

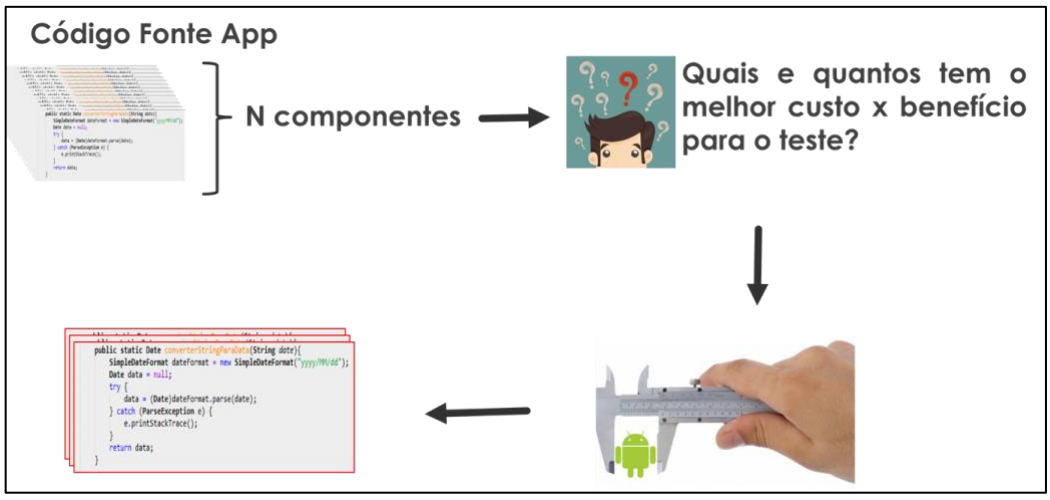


Figura 1. Exemplo do problema de pesquisa.

1.3. Hipótese

A hipótese definida para este trabalho considera o seguinte cenário:

A seleção automática de componentes, com relação ao custo x benefício, contribui para a eficácia (medida pela quantidade de componentes com erro selecionados) na escrita de testes unitários em aplicações móveis na plataforma Android.

1.4. Objetivos

Nesta seção serão apresentados os objetivos desta pesquisa.

1.4.1. Objetivo geral

Desenvolver e avaliar um plugin para a seleção de componentes em aplicações móveis na plataforma Android utilizando um algoritmo baseado em métricas estáticas, dinâmicas, de mercado e de negócio, com a finalidade de se obter uma lista de componentes para guiar a criação de testes unitários.

1.4.2. Objetivos específicos

Para atingir o objetivo geral desta pesquisa, pretende-se alcançar os seguintes resultados intermediários:

- Analisar um conjunto de métricas estáticas, dinâmica e de negócio como suporte para a seleção de componentes para testes unitários em aplicações móveis;

- Desenvolver um algoritmo para a seleção de componentes para testes unitários em aplicações móveis analisando o custo x benefício, medido pelas métricas analisadas;
- Propor um plugin para o Android Studio[3] visando apoiar a seleção multiobjetivo de componentes em uma aplicação na plataforma Android;
- Avaliar a eficácia do plugin SCoTUAM com a seleção realizada por especialistas em relação a quantidade de componentes com erro selecionados.

1.5. Metodologia de Pesquisa

A metodologia de pesquisa usada será baseada na abordagem para desenvolver novas tecnologias de software proposta por Mafra *et al.* (2006). A metodologia é composta por duas fases: **concepção**, cujo o foco é a definição e implementação do plugin, e; **avaliação**, focada em avaliar o plugin e verificar sua viabilidade. Na Figura 2 são mostradas as etapas dentro de cada fase.

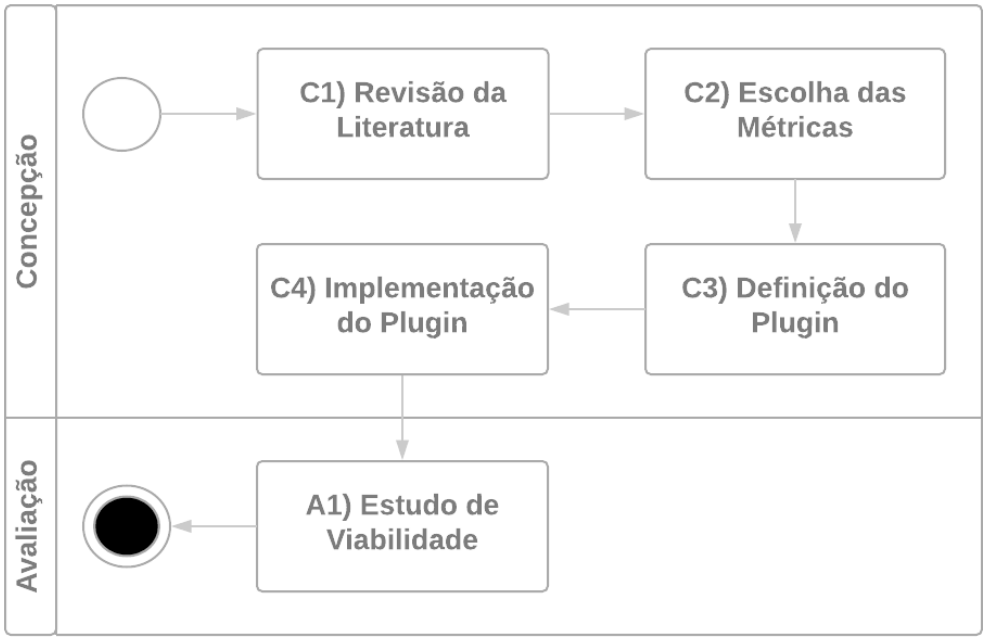


Figura 2. Metodologia de pesquisa.

1.5.1. Fase de Concepção

- **C1)** Revisão da Literatura

A revisão da literatura tem o objetivo de identificar técnicas de seleção e/ou priorização de código (classe, método ou bloco), com a finalidade de fornecer um corpo de

[3] https://developer.android.com/studio/index.html?hl=pt-br

conhecimento de métricas que podem ser utilizadas para a seleção de código proposta nesta pesquisa.

- **C2)** Escolha das Métricas

  Nesta etapa é analisado o corpo de conhecimento de métricas com o intuito de identificar métricas relevantes para serem utilizadas na seleção de componentes para teste de unidade na plataforma Android. O resultado desta etapa será apresentado na seção 2.3.1 deste trabalho.

- **C3)** Definição do Plugin

  Nesta etapa é definida a estrutura do plugin, como os artefatos de entrada, as ferramentas que serão utilizadas para a coleta de métricas, os processos utilizados na seleção dos componentes. O resultado desta etapa será apresentado no capítulo 3 deste trabalho.

- **C4)** Implementação do Plugin

  Nesta etapa o plugin SCoTUAM será implementado para a IDE Android Studio.

1.5.2. Fase de Avaliação

- **A1)** Estudo de Viabilidade

  Planejar e executar um estudo experimental com o objetivo de avaliar o plugin proposto em relação à sua capacidade de selecionar um subconjunto de componentes que possibilite auxiliar com eficácia os desenvolvedores na seleção de componentes para a escrita de testes unitários em aplicações na plataforma Android, visando permitir a transferência dessa tecnologia para o contexto industrial.

1.6. Estrutura do Documento

Este primeiro capítulo apresentou o contexto, no qual este trabalho está inserido, a motivação para realizá-lo, o objetivo a ser atingido e a metodologia a ser seguida. Além deste capítulo, a organização do texto está estruturada em mais cinco capítulos, descritos a seguir:

Capítulo 2: aborda o referencial teórico, no qual esse trabalho é baseado com o seguinte conteúdo: (2.1) Aplicações Móveis, (2.2) Teste de Software, (2.3) Seleção de Componentes de Software para Teste de Unidade, (2.4) Trabalhos Relacionados e (2.5) Considerações Finais.

Capítulo 3: nesse capítulo são apresentadas as seguintes subseções: (3.1) Visão Geral do , (3.2) Selecionar Artefatos de Entrada, (3.3) Executar Análise Estática, (3.4) Executar

Análise Dinâmica, de Mercado e de Negócio, (3.5) Pontuar Métricas, (3.6) Selecionar Componentes, (3.7) Lista de Componentes, (3.8) Análise de Correlação das Métricas e (3.9) Considerações Finais.

Capítulo 4: nesse capítulo é apresentado as tecnologias utilizadas na implementação do plugin SCoTUAM e uma descrição de suas funcionalidades, com as seguintes subseções: (4.1) Tecnologias, (4.2) Interface Gráfica do Usuário, (4.3) Limitações e (4.4) Considerações Finais.

Capítulo 5: nesse capítulo é apresentado o planejamento, execução e resultado do estudo de viabilidade do plugin SCoTUAM, com as seguintes subseções: (5.1) Objetivo do Estudo, (5.2) Planejamento do Estudo, (5.3) Execução do Estudo, (5.4) Resultados do Estudo de Viabilidade e (5.5) Considerações Finais.

Capítulo 6: nesse capítulo são apresentadas as conclusões e trabalhos futuros da dissertação, com as seguintes subseções: (6.1) Considerações Finais, (6.2) Contribuições, (6.3) Limitações e (6.4) Trabalhos Futuros.

CAPÍTULO 2 - REFERENCIAL TEÓRICO

Neste capítulo serão apresentados os conceitos relacionados a Aplicações Móveis, Teste de Software e Seleção de Componentes, que são áreas envolvidas dentro desta pesquisa. Também serão apresentados os trabalhos relacionados que foram classificados em duas categorias: seleção/priorização de código para refatoração e seleção/priorização de código para teste de software.

2.1. Aplicações Móveis

O desenvolvimento de aplicações móveis é o processo pelo qual o software é desenvolvido para dispositivos portáteis de baixo consumo, como *smartphones* ou *tablets* (Okediran *et al.*, 2014). Essas aplicações podem ser pré-instaladas em dispositivos móveis durante a fabricação ou disponibilizadas para download a partir de *App Stores* (lojas de aplicações na Internet), onde a distribuição será gratuita ou por meio de compras. *Google Play* é a loja online para distribuição de aplicações, jogos, filmes, músicas, livros e revistas para a plataforma Android. As aplicações para a plataforma iOS podem ser adquiridas pela *Apple Store*.

Neste trabalho, o foco será a plataforma Android, que é um sistema operacional introduzido pelo Google em 2007 e é o mais popular em smartphones no mundo (Statista, 2017a). Desde o seu lançamento, as vendas de smartphones em execução no Android têm crescido fortemente ao longo dos anos. Foram vendidos 6,8 milhões de smartphones Android em 2009. Somente em 2016 esse número aumentou para mais de 1,27 bilhões. O Android representa 86.1% de todas as vendas de smartphones para usuários finais em todo o mundo no segundo quartil de 2017 (Statista, 2017a).

Este aumento na quantidade de usuários do Android deu um impulso às aplicações móveis desenvolvidas. O número de aplicações disponíveis na *Google Play* ultrapassou 1 milhão em julho de 2013 e foi recentemente colocado em 3,5 milhões em dezembro de 2017 (Statista, 2017b). Com esse rápido crescimento no número de aplicações e levando em consideração a fatia de mercado do Android, pode-se inferir que os dispositivos móveis se tornaram uma parte indispensável da vida diária das pessoas (Ma *et al.*, 2016).

A plataforma Android possui uma estrutura que permite criar aplicações usando um conjunto de componentes reutilizáveis. Esses componentes são os blocos de construção essenciais de uma aplicação para a plataforma Android. Cada componente é um ponto de entrada por meio do qual o sistema ou usuário podem interagir com a aplicação. Alguns

componentes dependem de outros. Há quatro tipos de componentes de aplicações: Atividade, Serviços, Provedores de Conteúdo e Receptores de Transmissão. A seguir, segue uma breve descrição desses componentes segundo Android (2016a).

A Atividade (*Activity*) é um componente de aplicação que fornece uma tela com a qual os usuários podem interagir para fazer algo. Serviços (*Service*) são componentes executados em segundo plano para realizar operações de execução longa ou para realizar trabalho para processos remotos. Os componentes Provedores de Conteúdo (*Content Provider*) gerenciam um conjunto compartilhado de dados da aplicação. Os Receptores de Transmissão (*Broadcast Receiver*) são componentes que respondem a anúncios de transmissão por todo o sistema.

Uma Atividade do Android possui um ciclo de vida, que é definido pelos seguintes métodos: *onCreate(Bundle)*, *onStart()*, *onRestart()*, *onResume()*, *onPause()*, *onStop()* e *onDestroy()*. Desses métodos, pode ser destacado o método *onCreate(Bundle)*, que é chamado quando a atividade é criada pela primeira vez, e o método *onDestroy()*, que é o último a ser chamado antes da Atividade ser destruída (Android, 2017).

2.2. Teste de Software

O Teste de software consiste em uma investigação conduzida para fornecer às partes interessadas informações sobre a qualidade do produto ou serviço sob teste. Os testes de software também podem fornecer uma visão objetiva e independente do software para permitir que a empresa aprecie e compreenda os riscos de implementação do software. As técnicas de teste incluem, mas não estão limitadas ao processo de execução de um programa ou aplicação com a intenção de revelar falhas de software (Yadav *et al.*, 2016).

Na maioria das organizações de desenvolvimento de programas, o teste de software funciona como a "porta de qualidade" final para uma aplicação, permitindo ou impedindo a entrega do produto para o mundo real. Com este papel, surge uma grande responsabilidade: o sucesso de uma aplicação, e possivelmente da organização, pode repousar sobre a qualidade do produto de software (Dustin, 2002).

Quando a técnica de teste utiliza o processo de execução da aplicação, ele tem dois distintos objetivos: encontrar falhas ou demonstrar a correta execução do programa (Jorgensen, 2016). Os testes são tão bons quanto os casos de teste, mas podem ser inspecionados para garantir que todos os requisitos sejam testados em todas as combinações possíveis de entradas e estados do sistema. No entanto, nem todos os defeitos são detectados durante o teste (Lewis, 2016).

2.2.1. Níveis de Teste

O teste de software geralmente é realizado em diferentes níveis ao longo do processo de desenvolvimento e manutenção. Os níveis podem ser distinguidos com base no objeto de teste, que é chamado de alvo, ou na finalidade, que é chamado de objetivo (do nível de teste) (Bourque *et al.*, 2014). Podem distinguir-se três níveis de teste: unidade, integração e sistema.

- **Teste de Unidade**

É o nível de teste de software pelo qual unidades individuais de código-fonte são testadas (Jorgensen, 2016). De modo intuitivo, pode-se ver uma unidade como a menor parte testável de uma aplicação. Na programação procedural, uma unidade poderia ser um módulo inteiro, mas é mais comumente uma função individual ou procedimento. Na programação orientada a objetos, uma unidade é muitas vezes uma interface inteira, como uma classe, mas pode ser um método individual (Xie *et al.*, 2007).

Os testes de unidade (ou unitários) são fragmentos de código curtos criados por programadores ou ocasionalmente por testadores durante o processo de desenvolvimento. Este teste também é conhecido como teste de componente (Müller *et al.*, 2007).

- **Teste de Integração**

É o tipo de teste de software que procura verificar a corretude das interfaces entre unidades de um projeto de software (Jorgensen, 2016). Estratégias de integração modernas e sistemáticas são normalmente orientadas pela arquitetura, que envolve a integração incremental de componentes de software. Os testes de integração são muitas vezes uma atividade contínua em cada estágio de desenvolvimento durante o qual os engenheiros de software abstraem as perspectivas de nível inferior e concentram-se nas perspectivas do nível em que estão se integrando (Bourque *et al.*, 2014).

- **Teste de Sistema**

O teste de sistema avalia o funcionamento de um sistema completamente integrado para verificar se ele atende aos seus requisitos (Jorgensen, 2016). O teste de sistema é geralmente considerado apropriado para avaliar os requisitos não funcionais do sistema, como segurança, velocidade, precisão e confiabilidade. As interfaces externas para outras aplicações, utilitários, dispositivos de hardware ou os ambientes operacionais também são geralmente avaliadas neste nível (Bourque *et al.*, 2014).

2.2.2. Teste em Aplicações Móveis

O teste de aplicações móveis é um processo pelo qual a aplicação desenvolvida é testada para verificar sua funcionalidade, usabilidade e consistência (Nimbalkar, 2013). O teste de aplicação móvel pode ser realizado de forma automatizada ou manual.

Vários requisitos exclusivos distinguem os testes de aplicações móveis de testes de software convencionais. Primeiro, as aplicações para dispositivos móveis devem funcionar corretamente a qualquer hora e em qualquer lugar. Além disso, como os serviços móveis são frequentemente desenvolvidos para um conjunto de dispositivos selecionados, as aplicações devem funcionar adequadamente em plataformas que possuem, por exemplo, diferentes sistemas operacionais, tamanhos de exibição, recursos de computação e duração da bateria (Gao *et al.*, 2014).

Ainda, para proporcionar a rica experiência que os usuários de dispositivos móveis esperam, as aplicações móveis devem incluir vários canais de entrada (ex: teclado, voz e gestos), suporte multimídia e outros recursos de usabilidade aprimorados. Além disto, simulação em larga escala e virtualização são necessários para manter os custos de hardware baixos. Finalmente, como a maioria dos planos de serviços móveis suporta uma variedade de redes sem fio (2G, 3G, 4G, Wi-Fi, WiMax), as aplicações móveis devem funcionar em diversos contextos de conectividade de rede (Gao *et al.*, 2014).

A fragmentação em dispositivos móveis é um problema que está relacionado com a quantidade de diferentes sistemas operacionais ou diferentes configurações de dispositivos, tais como: memória, tamanho de tela, teclado e resolução de tela (Rajapakse, 2008). Essas características fazem com que o teste para aplicações móveis seja um desafio significativo. Aliado a isso, as estruturas de teste atuais não oferecem o mesmo nível de suporte para diferentes plataformas, como também as ferramentas não suportam recursos importantes para testes móveis, a título de exemplo a mobilidade, serviços de localização, sensores ou gestos e entradas diferentes (Joorabchi *et al.*, 2013). Ademais, existe a dificuldade no reuso dos scripts de teste devido o processo de desenvolvimento sofrer muitas mudanças com a rápida atualização das tecnologias (Ma *et al.*, 2016).

Em consequência a esses desafios, muitas aplicações chegam ao mercado com problemas relevantes, que muitas vezes resultam em falhas durante o uso (Deng *et al.*, 2016).

2.2.3. Automação de Teste

A necessidade de automação no desenvolvimento de software moderno dificilmente precisa de justificativa. Não só a automação facilita a implementação das melhores práticas e

a coleta de dados relacionados ao projeto, mas também reduz a influência humana propensa a erros na implementação do processo. Todas as tarefas repetitivas devem ser automatizadas sempre que possível em qualquer parte do ciclo de vida do desenvolvimento de software (Parasoft, 2009).

A automação é essencial para tornar a prevenção de defeitos uma estratégia sustentável no desenvolvimento de software. Quando as práticas-chave de prevenção de defeitos são automatizadas, as organizações podem garantir que essas práticas sejam implementadas com o mínimo de interrupção para os processos e projetos existentes. Além disso, a automação é a solução para assegurar que tanto as práticas gerais quanto as personalizadas de prevenção de defeitos que a equipe decida implementar sejam aplicadas rigorosamente e consistentemente (Huizinga e Kolawa, 2007).

A execução manual de um caso de teste é rápida e efetiva, porém a execução e repetição de um grande conjunto de testes manualmente é uma tarefa muito custosa e cansativa. Algo normal e compreensivo é que os testadores não verifiquem novamente todos os casos a cada mudança significativa do código. Uma das grandes vantagens dos testes automatizados, é que todos os casos de teste podem ser facilmente e rapidamente repetidos a qualquer momento e com pouco esforço. Sendo que os testes automatizados são programas ou scripts simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos. (Karhu *et al.*, 2008). Uma etapa da automatização dos testes é a automação da seleção dos componentes a serem testados.

2.3. Seleção de Componentes de Software para Teste de Unidade

Um componente de software pode ser qualquer unidade reutilizável e independente de funcionalidade, como uma classe, um método ou uma biblioteca (Haghpanah *et al.*, 2007). Um componente, no contexto deste trabalho, será considerado como um método de uma classe de um programa orientado a objeto.

Em um sistema de software, geralmente não há possibilidade de que apenas um componente atenda todos os requisitos do programa, então cada componente (possuindo suas respectivas funcionalidades) se integra com outros componentes para atender às necessidades do sistema em desenvolvimento (Segundo, 2014).

O trabalho aqui proposto concentra-se na seleção de componentes para a escrita de teste de unidade, onde deve-se escolher a partir do conjunto de todos os componentes um subconjunto que promova o equilíbrio entre o benefício e o custo do teste de unidade. Para sistemas com mais de alguns componentes simples, o espaço de busca é incontrolavelmente grande e complexo, consequentemente nenhum gerente pode esperar para encontrar

escolhas ideais que equilibrem as restrições sem algum suporte automatizado. Este é o "Problema de Seleção de Componentes" (do inglês *Component Selection Problem* – CSP) (Harman *et al.*, 2006).

Para automatizar o processo da seleção de um subconjunto de componentes para a escrita de teste unitário, se faz necessário fazer a medição de algumas métricas. As métricas são funções, enquanto as medições são os números obtidos pela aplicação de métricas.

2.3.1. Métricas Escolhidas

Três tipos de abordagens são amplamente usadas a fim de selecionar componentes para teste de unidade. Elas se baseiam em métricas estáticas, métricas dinâmicas e localização de falhas baseada no espectro (Freitas *et al.*, 2016). Este trabalho utiliza métricas que derivam de quatro categorias: métricas estáticas, dinâmicas, de mercado e de negócio. As seguintes métricas estão sendo utilizadas:

- Estáticas: custo de manutenção futura (CMF) e cheiros de código (CS);
- Dinâmicas: frequência de chamadas (FC) e risco de falha (RF);
- Mercado: vulnerabilidade de mercado (VM);
- Negócio: valor de negócio (VN).

Essas métricas foram escolhidas a partir da revisão da literatura técnica, onde buscou-se trabalhos que utilizavam métricas para a seleção de código-fonte. Sendo que as métricas CMF, FC, RF e VM foram baseadas no trabalho de Freitas *et al.* (2016) que apresenta o método SCOUT para a seleção de componentes para o teste de unidade em aplicações móveis. A métrica cheiro de código foi escolhida devido ela identificar componentes que indicam fraquezas no projeto que podem estar atrasando o desenvolvimento ou aumentando o risco de erros ou falhas no futuro. A métrica valor de negócio foi escolhida devido ela possibilitar a identificação dos componentes que tem uma maior importância para o negócio da aplicação. A premissa básica é que se as áreas críticas do código-fonte da aplicação forem identificadas, o esforço e custo com as atividades de teste podem ser reduzidos. As próximas subseções irão detalhar cada métrica utilizada neste estudo.

A. Custo de Manutenção Futura

A manutenção é parte integrante de um ciclo de vida de software e tem por objetivo modificar a aplicação existente, preservando ao mesmo tempo sua integridade. Ela é necessária para garantir que a aplicação continua a satisfazer os requisitos do usuário, sendo

aplicável ao produto que é desenvolvido usando qualquer modelo de ciclo de vida (Bourque *et al.*, 2014).

As aplicações mudam devido a ações corretivas e não-corretivas (Bourque *et al.*, 2014). Cada componente tem uma propensão ao defeito associada e, em caso de falha, os responsáveis pela manutenção do software gastam tempo para entender o sistema e fazer as mudanças apropriadas.

A métrica Custo de Manutenção Futura (*cfm*) está sendo derivada de duas outras métricas: *Halstead Effort* e *Halstead Bugs* (Halstead, 1979). A composição dessas duas métricas está sendo apresentada na Tabela 1. Elas são baseadas nos números de operadores e operandos do código fonte.

Tabela 1. Seis métricas Halstead.

Métricas	**Símbolo**	**Fórmula**
Component Length	N	$N = N_1 + N_2$
Component Vocabulary	n	$n = n_1 + n_2$
Volume	V	$V = N * (LOG_2\ n)$
Difficulty	D	$D = (n_1 / 2) * N_2 / n_2$
Effort	E	$E = D * V$
Bugs	B	$B = V / 3000$

Onde:

- N_1 = Número total de operadores;
- N_2 = Número total de operandos;
- n_1 = Número distintos de operadores;
- n_2 = Número distintos de operandos.

A fórmula para calcular o Custo de Manutenção Futura, de acordo com Freitas *et al.* (2016):

$$cfm_i = E_i * B_i$$

Equação 1. Custo de manutenção futura.

- E_i : é a quantidade de trabalho em segundos (métrica *Halstead Effort*) para compreender e recodificar o componente *i*;

- B_i : é o número estimado de erros (métrica *Halstead Bugs*) para o componente *i*.

B. Cheiros de Código (*Code Smell - CS*)

Cheiros de código, também conhecido como mau cheiro (em inglês, *bad smell*), geralmente não são defeitos. Eles não são tecnicamente incorretos e não impedem o funcionamento do programa. Em vez disso, eles indicam fraquezas no projeto que podem estar atrasando o desenvolvimento ou aumentando o risco de erros ou falhas no futuro (Martin, 2009).

Alguns cheiros de código apresentados na literatura para o nível de componentes:

- **Método Longo (*Long Method - LM*):** quando o componente contém mais de cem linhas, sem contar as linhas em branco ou comentários.
- **Longa Lista de Parâmetros (*Long Parameter List - LPL*):** quando o componente tem sete parâmetros ou mais.
- **Declarações de Desvio (*Switch Statements - SW*):** quando se tem um operador *switch* complexo, uma sequência de instruções *if* aninhadas ou uma sequência de laços (*for, while, do-while*) de repetição aninhados.
- **Método Cérebro (*Brain Method - BM*):** tendem a centralizar a funcionalidade de uma classe, da mesma forma que uma *God Class* centraliza a funcionalidade de um subsistema inteiro ou, às vezes, até mesmo um sistema inteiro.
- **Complexidade Ciclomática (*Cyclomatic Complexity - CC*):** quando o número de caminhos possíveis através do código fonte for maior ou igual a cinco.

A fórmula abaixo calcula o valor do cheiro de código para cada componente.

$$cs_i = \frac{s_i}{5}$$

Equação 2. Cheiro de código.

s_i : é a quantidade de cheiro de código detectado para o componente *i*.

C. Risco de Falha

Com diferentes graus de formalização, técnicas de teste baseadas em falhas concebem casos de teste especificamente destinados a revelar categorias de falhas prováveis ou predefinidas. Uma boa fonte de informação é a história de falhas descobertas em testes anteriores, bem como a experiência do engenheiro de software. Para descobrir o risco de falha, os casos de teste são projetados especificamente por engenheiros de software que

tentam antecipar as falhas mais plausíveis em uma determinada aplicação (Bourque *et al.*, 2014).

Os dados necessários para calcular essa métrica provêm do registro da execução de um componente em casos de teste que passaram ou falharam. A fórmula abaixo foi extraída de Freitas *et al.* (2016).

$$rf_i = \frac{p_i}{p_i + f_i}$$

Equação 3. Risco de falha.

p_i : é a razão entre o número de casos de teste passados que executaram o componente *i* e o número total de casos de teste passados no conjunto de testes;

f_i : é a razão entre o número de casos de teste que falharam e executaram o componente *i* e o número total de casos de teste falhados no conjunto de testes.

D. Frequência de Chamadas

Durante a execução da aplicação é contado a quantidade de vezes que um método é chamado, pois na análise estática o componente pode ter ficado com uma prioridade alta. No entanto, o impacto dessas métricas estáticas deve estar associado de alguma forma a uma métrica que reflita o nível de requisição de um componente em execução, ou seja, chamadas de método (Freitas *et al.*, 2016). Está métrica irá computar a quantidade de vezes que um componente foi chamado durante a execução dos casos de teste.

E. Vulnerabilidade de Mercado

A métrica de vulnerabilidade de mercado expressa a vulnerabilidade de um componente entre dispositivos de acordo com a distribuição do mercado (Android, 2016b). Em suma, quanto maior for a penetração no mercado de uma determinada configuração de dispositivo, maior será a probabilidade de que falhas em aplicações em execução neste dispositivo afetem muitos usuários. Uma forma de calcular a vulnerabilidade de mercado para um componente é apresentada em Freitas *et al.* (2016) da seguinte forma:

- Para cada dispositivo, é criada uma lista de casos de teste que falharam;
- Para cada dispositivo com falhas, o seu mercado mínimo e máximo associado é computado;
- O mercado de api (*ma*) é calculado como a soma da porcentagem de mercado de API destes dispositivos;

- O mercado da tela (*mt*) é calculado como a soma do tamanho da tela e da densidade de mercado destes dispositivos;
- A vulnerabilidade mínima do mercado pode ser expressa como o valor máximo entre *ma* e *mt*;
- A vulnerabilidade máxima do mercado pode ser expressa como a soma de *ma* e *mt*;
- A vulnerabilidade média do mercado é a média entre o mínimo e o máximo valor da vulnerabilidade do mercado.

F. Valor de Negócio

O domínio da engenharia de software testemunhou o surgimento do conceito de engenharia de software baseado em valor (em inglês, *Value-Based Software Engineering* – VBSE), que pretende mudar as práticas de engenharia de software convencionais para se tornarem centradas no valor e produzir produtos valiosos (Ibrahim *et al.*, 2015). Na qual cada requisito, caso de uso, objeto e defeito não são tratados como igualmente importantes. Portanto os casos de uso com maior valor de negócio devem possuir mais casos de teste.

Para calcular essa métrica é necessário pontuar os casos de uso da aplicação, pois cada vez que um caso de uso executa um componente, o valor desse caso de uso é somado ao componente. Ou seja, o valor dessa métrica para um componente é a soma final de pontos, após a execução dos casos de uso.

2.3.2. Teste de Software Baseado em Busca (SBST)

Um conceito importante para a automatização da seleção de componentes para teste de unidade é a engenharia de software baseada em busca (em inglês, *Search-Based Software Engineering* – SBSE), que tem suas origens a partir de 1970. Inicialmente, SBSE era tratada com pouca atenção pela comunidade acadêmica, até que em 2001 foi mais difundida com a publicação do artigo de (Harman e Jones, 2001). A SBSE é uma subárea da engenharia de software que emprega técnicas e métodos que reformulam e modelam problemas que podem ser resolvidos usando técnicas de otimização baseadas em busca meta-heurística.

Este tipo de busca é utilizado em problemas que não se tem conhecimento de algoritmos eficientes para sua resolução. Além de automatizar a atividade, a busca meta-heurística dispõe-se a obter resultados aceitáveis (soluções próximas da ótima) ou até mesmo pode encontrar a solução ótima (Maia *et al.*, 2009).

Uma subárea de SBSE é testes de software baseado em busca (em inglês, *Search-Based Software Testing* – SBST) que tem atraído muita atenção nos últimos anos. O interesse crescente em SBST pode ser atribuído ao fato de que o teste de software geralmente é considerado um problema indecidível, principalmente devido às muitas combinações possíveis de entrada de um programa (Afzal *et al.*, 2009).

Entre os problemas que são resolvidos em SBST, destacam-se a geração de dados de teste, seleção e/ou priorização de casos de teste, seleção e/ou priorização de código para teste, teste funcionais e testes não funcionais (Maia *et al.*, 2009). Sendo que destes problemas citados, este trabalho se concentra na seleção de código para teste, modelando o problema de seleção de componentes para a escrita de teste unitário como um problema de otimização multi-objetivo.

A otimização multi-objetivo envolve a minimização ou maximização de múltiplas funções de adequação. Este tipo de problema está relacionado a situações que ocorrem compromisso (*trade-off* - dois ou mais objetivos conflitantes), por exemplo, fatores como tempo, custo e qualidade competem entre si. É comum encontrar este tipo de problema na vida real (Segundo, 2014). Por exemplo, para este trabalho os fatores conflitantes são o custo e benefício do teste de unidade de um componente.

Um problema de otimização pode ser solucionado de várias formas: abordagem aleatória, construção de heurística, algoritmo genético (GA), SPEA-II, NSGA-II, NSGA-III. Para este trabalho foi escolhido utilizar o algoritmo multi-objetivos NSGA-II, pois em (Freitas *et al.*, 2016) foi realizado um estudo comparativo dessas soluções citadas verificando qual algoritmo tinha o melhor resultado para o problema da seleção de componentes para teste de unidade e o algoritmo NSGA-II foi o que se mostrou eficaz.

A seleção dos componentes utilizará o algoritmo genético (GA) de classificação não dominado versão 2 (*Non-Dominated Sorting Genetic Algorithm version* 2 - NSGA-II) (Deb *et al.*, 2002). Em um GA, o cromossomo ou indivíduo representa uma solução. Então deve-se mapear o problema para uma representação genética em que seja possível realizar manipulações simbólicas no GA. Sendo que um cromossomo é formado por genes, no contexto deste trabalho cada gene será um componente. Ou seja, um conjunto de componentes é um cromossomo. O NSGA-II utiliza a técnica de seleção por torneio. Nesta técnica, uma determinada quantidade de indivíduos da população é escolhida aleatoriamente. Em seguida, há competição direta destes indivíduos pelo direito de ser pai, de acordo com as funções objetivo de cada um. O melhor indivíduo desse torneio será selecionado para ser pai da próxima geração. Em conjunto com a seleção por torneiro, utiliza-se o elitismo, no qual o NSGA-II garante a manutenção dos melhores indivíduos de uma geração na próxima geração.

O operador de cruzamento que será utilizado será o ponto simples (*single-point*) e o operador de mutação será o *bitwise* para a codificação binária (*binary-coded*).

Neste trabalho há dois objetivos, pois no processo de seleção de componentes é idealizado que os componentes escolhidos para a escrita de teste unitário sejam os melhores possíveis, no qual maximiza-se o benefício e diminua o custo. As duas funções objetivo utilizadas estão descritas abaixo. A função objetivo abaixo maximiza o benefício:

$$max \{ \sum_{i=1}^{N} \frac{pCmf * cfm_i + pRf * rf_i + pFc * fc_i + pVm * vm_i + pCs * cs_i + pVn * vn_i}{(pCmf + pRf + pFc + pVm + pCs + pVn)} . x_i \}$$

Equação 4. Função objetivo de maximização.

- pCmf: porcentagem da métrica cmf, ;
- cfm_i : custo de manutenção futura;
- *pRf*: porcentagem da métrica rf;
- rf_i : risco de falha;
- *pFc*: porcentagem da métrica fc;
- fc_i : frequência de chamadas;
- *pVm*: porcentagem da métrica vm;
- vm_i : vulnerabilidade de mercado;
- *pCs*: porcentagem da métrica cs;
- cs_i : cheiros de código;
- *pVn*: porcentagem da métrica vn;
- vn_i : valor de negócio;
- x_i : o valor é 1 (um) se o componente *i* é selecionado, senão é 0 (zero).

Os valores para *pCmf*, *pRf*, *pFc*, *pVm*, *pCs* é *pVn* pode ser uma das seguintes porcentagens {0; 0,2; 0,4; 0,6; 0,8 e 1,0}, sendo que 0 é onde será utilizado 0% da métrica, ou seja, a métrica não será utilizada, 0,2 é utilizado 20% do valor da métrica e assim sucessivamente.

A função objetivo abaixo para minimizar o custo foi baseada em Freitas *et al.* (2016):

$$min \{ \sum_{i=1}^{N} c_i . x_i \}$$

Equação 5. Função objetivo de minimização.

- c_i : é o custo para desenvolver teste unitário (métrica Halstead Effort) para o componente *i*;
- x_i : o valor é 1 se o componente *i* é selecionado, senão é 0 (zero).

2.4. Trabalhos Relacionados

Os trabalhos relacionados que retornaram da revisão da literatura técnica foram classificados em duas categorias de acordo com a sua utilidade, a saber, seleção/priorização de código para refatoração[4] e seleção/priorização de código para teste de software. Estamos mostrando alguns trabalhos de priorização, mas o foco da pesquisa é a seleção código.

2.4.1. Seleção/Priorização de Código para Refatoração

Os autores em (Liu *et al.*, 2016) demonstram por meio da eficácia uma abordagem para adaptar os limiares de detecção de cheiro de código de forma automática e dinâmica em que os engenheiros definem uma precisão manualmente de acordo com seu tempo de trabalho e requisitos de qualidade. Com o feedback dos engenheiros, a abordagem procura automaticamente uma definição de limite para maximizar a refatoração, tendo simultaneamente a precisão perto da especificada pelo engenheiro. O resultado do experimento mostra uma melhora significativa nos códigos selecionados para a refatoração.

Hecht *et al.* (2016) conduziram um estudo experimental que foca nos impactos de desempenho individuais e combinados de três cheiros de código de desempenho do Android (a saber, *Internal Getter/Setter*, *Member Ignoring Method*, e *HashMap Usage*) em duas aplicações Android de código aberto. Para realizar esse estudo, usou-se o kit de ferramentas *Paprika* para detectar esses três cheiros de código nas aplicações e derivou-se quatro versões das aplicações corrigindo cada cheiro de código detectado de forma independente e depois todos eles. Em seguida, foi feita uma avaliação do desempenho de cada versão em um cenário de uso comum. Em particular, houve a avaliação do desempenho da interface do usuário e da memória usando as seguintes métricas: tempo de quadros, número de quadros atrasados, uso de memória e número de chamadas de coleta de lixo (do inglês, *garbage collection*). Os resultados mostraram que corrigir esses cheiros de código do Android efetivamente melhora a interface do usuário e desempenho de memória.

Amorim *et al.* (2015) estudam a eficácia do algoritmo *Decision Tree* para reconhecer cheiro de código. Para isso, ele foi aplicado em um conjunto de dados contendo 4 projetos de código aberto e os resultados foram comparados com o oráculo manual, com abordagens de detecção existentes e com outros algoritmos de aprendizado de máquina. Os resultados mostraram que a abordagem foi efetivamente capaz de aprender regras para a detecção dos

[4] Uma mudança feita na estrutura interna do software para facilitar a sua compreensão e sua modificação, sem alterar seu comportamento externo (Fowler e Beck, 1999).

cheiros de código estudados. Os resultados foram ainda melhores quando algoritmos genéticos foram usados para pré-selecionar as métricas a serem usadas.

Os autores Techapalokul e Tilevich (2015) argumentam que suporte de refatoração de software de primeira classe deve se tornar uma característica essencial em ambientes de programação para blocos. Apresentando uma análise de programa para detectar cheiro de código e transformações automatizadas para programas baseados em blocos que suportam técnicas comuns de refatoração.

Malhotra *et al.* (2015) propõem uma estrutura para identificar as potenciais classes que imediatamente exigem refatoração com base nos cheiros de código, bem como características de projeto. A abordagem foi avaliada em sistemas de código aberto de tamanho médio *ORDrumbox*. Foram identificados quatro tipos de cheiro de código*: Feature Envy*, *Long Method*, *God Class* e *Type Checking*. Esses são combinados em determinada proporção para calcular a nova métrica proposta de Regra de Índice de Depreciação de Qualidade (em inglês, *Quality Depreciation Index Rule* - QDIR) para cada classe. As classes são organizadas de acordo com seus valores QDIR para identificar as classes gravemente afetadas que requerem tratamento de refatoração imediata. Os resultados refletem que os cheiros de código e métricas de design podem ser usados como uma importante fonte de informação para quantificar os problemas nas classes, assim, ajudam os mantenedores na execução de suas tarefas sob estritas restrições de tempo enquanto mantêm a qualidade geral do software.

Schumacher *et al.* (2010) apresentam resultados para um estudo experimental realizado em ambiente comercial. O estudo investiga a forma como os desenvolvedores de software profissional detectam o cheiro de código *God Class*. Em seguida, ele compara esses resultados com a classificação automática. Os resultados mostram que, embora os sujeitos percebam a detecção de *God Class* como uma tarefa fácil, tem-se um baixo acordo para a classificação. A abordagem de detecção baseada em métricas apresenta um bom desempenho em comparação com a classificação humana. Esses resultados levam à conclusão de que uma pré-seleção automatizada baseada em métricas diminui o esforço gasto em inspeções de código manuais. A detecção automática acompanhada por uma revisão manual aumenta a confiança geral nos resultados dos classificadores baseados em métricas.

Vidal *et al.* (2016) focam em priorizar código para refatoração com o intuito de resolver problemas arquiteturais que afetam a evolução de projetos de software. Quando não adequadamente tratado, esses problemas podem dificultar a longevidade de um sistema de software. No entanto, um projeto de software geralmente contém milhares de anomalias de código e muitos deles não têm relação com problemas arquiteturais. Como consequência

disso, os desenvolvedores buscam determinar efetivamente quais (grupos de) anomalias são arquiteturalmente relevantes. No trabalho foi proposto critérios para priorizar grupos de anomalias de código como indicadores de problemas arquiteturais em sistemas em evolução.

Vidal *et al.* (2015) apresentam uma ferramenta flexível para priorizar a dívida técnica na forma de cheiros de código. A ferramenta é flexível para permitir que desenvolvedores adicionem novas estratégias de detecção e priorização de cheiro de código e grupos de cheiro de código, com base na configuração de seus múltiplos critérios. Para ilustrar essa flexibilidade, foi apresentado um exemplo de aplicação da ferramenta. Os resultados sugerem que a ferramenta pode ser facilmente estendida para ser alinhada com os objetivos do desenvolvedor.

Fontana *et al.* (2015) apresentam um *Índice de Intensidade* para ser utilizado como um estimador para determinar os casos mais críticos, priorizando o exame de cheiro de código e, potencialmente, a sua remoção. O *Índice de Intensidade* foi aplicado na detecção de seis cheiros de código bem conhecidos e comuns e foi relatado sua distribuição de intensidade a partir de uma análise realizada em 74 sistemas do Qualitas Corpus, mostrando como a Intensidade poderia ser usada para priorizar a inspeção de cheiro de código.

2.4.2. Seleção/Priorização de Código para Teste de Software

Usando o conceito de Execução Simbólica Dinâmica (*Dynamic Symbolic Execution* – DSE, em inglês), os autores em (Feist *et al.*, 2016) consideram os caminhos de DSE para enumerar parte de um dado trecho do programa. O critério de seleção de caminho proposto, visa minimizar o número de consultas aos solucionadores SMT (do inglês, *Satisfiability Modulo Theories*). Esse critério é baseado na probabilidade de um caminho sair da fatia do programa. Os experimentos mostraram que essas informações podem ser computadas em um tempo razoável para fins de DSE, que é uma técnica eficiente de enumeração de caminho baseada em SMT usada em testes de software.

Freitas *et al*. (2016) apresentam o *SCOUT (*do inglês, *Selector of Software Components for Unit Testing),* um método que fornece uma extração automática de métricas estáticas, dinâmicas e valor de mercado, seguida de um processo de otimização multiobjetivo. O SCOUT é um método capaz de auxiliar os testadores em diferentes domínios. A plataforma Android foi escolhida para realizar os experimentos, usando nove aplicações de código aberto. O método foi comparado em termos de eficácia com duas outras estratégias bastante utilizadas. Também foi comparado a eficácia e a eficiência de sete algoritmos na resolução de um problema de seleção de componentes multiobjetivo. Os experimentos foram realizados

sob diferentes cenários, e revelam o potencial do SCOUT em reduzir a vulnerabilidade de mercado, em comparação com outras abordagens.

Em (Freitas *et al.*, 2014) é apresentado uma abordagem evolutiva multiobjetivo que procura um subconjunto de artefatos que minimiza o custo ao mesmo tempo que maximiza sua importância estratégica. Métricas estáticas e dinâmicas, como a complexidade ciclomática, a cobertura operacional e a frequência de modificação, foram usadas para definir a importância estratégica. A motivação para usar a abordagem evolutiva multiobjetivo é que o crescimento da propensão à falha é inversamente proporcional ao custo. Os experimentos foram realizados no contexto real em sistema da indústria, e os resultados encontrados confirmaram os benefícios da proposta.

Ray *et al.* (2011) propõem uma métrica de programa chamada métrica de influência para encontrar a influência de um elemento de programa no código-fonte. Primeiro, representa-se o código fonte em um gráfico intermediário chamado gráfico de dependência do sistema estendido. Em seguida, o corte em frente é aplicado em um nó do gráfico para obter a influência desse nó. A métrica de influência para um método *m* num programa mostra o número de declarações do programa que utilizam direta ou indiretamente o resultado produzido pelo método *m*. Calculamos a métrica de influência para uma classe *c* com base na métrica de influência de todos os seus métodos. Foi realizado experimentos para dois estudos de caso bem conhecidos - Sistema de Gerenciamento de Bibliotecas e Sistema de Automação Comercial - e identificou-se elementos críticos no código-fonte de cada estudo de caso. Também se realizou experimentos para comparar esse esquema com um esquema relacionado. Os estudos experimentais justificam que essa abordagem é mais precisa do que as existentes na exposição de elementos críticos no nível de implementação.

Os autores em (Li *et al.*, 2006) apresentam um método que orienta testadores através de gerações de casos de teste. Em vez de automatizar todo o procedimento, esse método tem como objetivo gerar um framework de casos de teste baseados em código fonte, onde caminhos de teste que minimizam a quantidade de casos de teste gerados são selecionados e tem por objetivo prover meios para que os usuários instanciem o framework em casos de teste executáveis. O estudo experimental inicial deste método mostrou a eficácia do mesmo.

Ray e Mohapatra (2014) propõem um método para estimar a criticidade de um componente dentro de um sistema. O método de estimativa é baseado em documentos de projeto. Primeiramente, prioriza-se os componentes para testes de acordo com sua criticidade estimada. Em seguida, é apresentado uma técnica baseada em algoritmos genéticos para selecionar casos de teste de um grande conjunto de casos de teste. No método, a intensidade com que cada componente é testado deve ser proporcional à sua prioridade e a suíte de teste

é ideal sob outras restrições. Foram realizados experimentos para comparar esse esquema com um esquema relacionado. Os resultados experimentais estabelecem que uma maior confiabilidade pode de fato ser alcançada usando esse método.

Os autores em (Kasai *et al.*, 2013) propõem uma abordagem de predição propensa a falhas que combina um modelo de predição e inspeção manual. A inspeção manual é conduzida por uma lista de verificação pré-definida que consiste de perguntas e procedimentos de pontuação. As perguntas captam os sinais de falha ou indicações que são difíceis de serem capturados por métricas de código fonte usadas como entrada por modelos de previsão. Essa abordagem consiste em dois passos. No primeiro, os módulos são priorizados por um modelo de predição propenso a falhas. Na segunda etapa, um inspetor inspeciona e classifica os módulos priorizados. Foi realizado um estudo de caso com módulos de código fonte em software comercial que haviam sido mantidos e evoluídos ao longo de dez anos e comparado os valores de AUC (em inglês, *Area Under the Curve*) do Diagrama de Alberg entre três modelos de previsão: (A) máquinas de vetores de suporte, (B) linhas de código, e (C) preditor aleatório com quatro ordens de priorização. Os resultados indicam que os valores máximos de AUC sob α e o coeficiente do escore de inspeção foram maiores que os valores de AUC dos modelos de predição sem inspeção manual em cada uma das quatro combinações e os três modelos do contexto.

Ray e Mohapatra (2012) propõem uma abordagem que considera cinco fatores de um componente, como o valor de influência, tempo de execução médio, complexidade estrutural, gravidade e valor de mercado como entradas e produz o valor de prioridade do componente como uma saída. A abordagem é eficaz na orientação do esforço de teste, uma vez que está ligada à medida externa da gravidade do defeito e do valor do negócio, medida interna de frequência e complexidade. Foram realizados experimentos para comparar esse esquema com um esquema relacionado. Os resultados estabelecem que essa abordagem que prioriza o esforço de teste dentro do código-fonte é capaz de minimizar os tipos altamente discriminados de falhas e também o número de falhas no tempo de pós-liberação de um sistema de software.

Episkopos *et al.* (2011) apresentam um método para fornecer dicas sobre qual parte do código deve ser testada primeiro para obter a melhor cobertura de código. Esse método tem duas contribuições principais. Primeiro, leva em conta uma "visão global" da execução de um programa que está sendo testado, considerando o impacto da chamada de relacionamento entre métodos/funções de software complexo. Em seguida, relaxa a condição "garantida" da análise do dominador tradicional para ser "pelo menos" a relação entre nós dominantes, o que torna o cálculo do dominador muito mais simples sem perder sua precisão.

Foi implementado duas versões de métodos de priorização de código, uma baseada na análise do dominador original e a outra na análise do dominador relaxado com visão global. O estudo de comparação mostra que esse último é consistentemente melhor em termos de identificação de código para testes com o fim de aumentar a cobertura de código.

Hosseingholizadeh (2010) apresenta uma técnica para a priorização e otimização de testes de componentes de software, que é baseada em uma análise de risco baseada em código-fonte. A técnica de classificação fornece aos desenvolvedores um modelo de risco da aplicação que é projetado especificamente para auxiliar o processo de teste, identificando os componentes mais importantes e sua estimativa de esforço de teste correspondente. Foi desenvolvido uma ferramenta de análise para aplicar essa técnica a um projeto de software de teste e os resultados mostram que a técnica maximiza a quantidade de risco que pode ser eliminada com a utilização da técnica.

Shihab *et al.* (2010) apresentam uma abordagem que auxilia gerentes de desenvolvimento de software e teste, que empregam TDM (no inglês, *Test-Driven Maintenance*), a utilizarem os recursos limitados que têm para testar sistemas legados eficientemente. A abordagem aproveita o histórico de desenvolvimento do projeto para gerar uma lista priorizada de funções que os gerentes devem focar seus recursos para a escrita de teste de unidade. A lista de funções é atualizada dinamicamente à medida que o desenvolvimento do sistema legado avança. Para avaliar essa abordagem, foi realizado um estudo de caso sobre um grande sistema de software comercial legado. Os resultados sugerem que as heurísticas baseadas no tamanho da função, frequência de modificação e frequência de fixação de bugs devem ser usadas para priorizar a escrita de teste de unidade de sistemas legados.

2.5. Considerações Finais

Neste capítulo foram considerados os principais conceitos sobre aplicações móveis, teste de software e suas características. Logo é descrito o teste de aplicações móveis e seus desafios, assim como os critérios que são usados para testá-las, como também o teste de software baseado em busca (em inglês, SBST) e finalmente os trabalhos relacionados.

Os trabalhos apresentados abordam diversas formas de seleção e/ou priorização de código, em diferentes níveis e com propósitos diferentes. A Tabela 2 apresenta um resumo das principais características em comum dos trabalhos relacionados.

As características consideradas para o agrupamento dos trabalhos são as seguintes:

- **Finalidade:** refere-se ao propósito da seleção ou priorização do código;
- **Categoria:** informa se o trabalho é de seleção ou priorização de código;

- **Nível:** especifica a parte do código em que a seleção ou priorização irá focar;
- **Linguagem:** informa a linguagem do código fonte utilizado no experimento;
- **Plataforma:** diz respeito a plataforma em que os experimentos foram executados.

Tabela 2. Resumo dos trabalhos relacionados.

ID	Trabalho	Finalidade	Categoria	Nível	Linguagem	Plataforma
1	Liu *et al.* (2016)	Refatoração	Seleção	Classe, Método e Bloco	Java	Desktop
2	Hecht *et al.* (2016)	Refatoração	Seleção	Método	Java	Mobile
3	Amorim *et al.* (2015)	Refatoração	Seleção	Classe	Java e JavaScript	Desktop
4	Techapalokul e Tilevich (2015)	Refatoração	Seleção	Bloco	Não informado	Não informado
5	Malhotra *et al.* (2015)	Refatoração	Seleção	Classe	Java	Desktop
6	Schumacher *et al.* (2010)	Refatoração	Seleção	Classe e Método	C#	Web
7	Vidal *et al.* (2016)	Refatoração	Priorização	Classe	Java	Web e Mobile
8	Vidal *et al.* (2015)	Refatoração	Priorização	Classe e Método	Java	Não informado
9	Fontana *et al.* (2015)	Refatoração	Priorização	Classe e Método	Java	Desktop
10	Feist *et al.* (2016)	Teste	Seleção	Bloco	Não informado	Não informado
11	Freitas *et al.*, 2016	Teste	Seleção	Método	Java	Mobile
12	Freitas *et al.*, 2014	Teste	Seleção	Método	Java	Web
13	Ray *et al.* (2011)	Teste	Seleção	Classe e Método	Java	Desktop
14	Li e Yee (2006)	Teste	Seleção	Bloco	Java	Web
15	Ray e Mohapatra (2014)	Teste	Priorização	Classe e Método	Não informado	Não informado
16	Kasai *et al.* (2013)	Teste	Priorização	Método	C e C++	Desktop
17	Ray e Mohapatra (2012)	Teste	Priorização	Classe e Método	Java	Não informado
18	Episkopos (2011)	Teste	Priorização	Bloco	Java	Não informado
19	Hosseingholizadeh (2010)	Teste	Priorização	Método	Não informado	Não informado

20	Shihab (2010)	Teste	Priorização	Método	C e C++	Não informado

Neste resumo pode-se observar que a maioria dos trabalhos realizam experimentos com código fonte escrito na linguagem Java. Também se destaca que dos três níveis, o nível que mais apareceu nos trabalhos foi o de método.

Na Tabela 3 é mostrado para cada métrica utilizada nesta pesquisa, quais foram os trabalhos relacionados que também fizeram uso da métrica. Pode-se observar que quatro das seis métricas utilizadas nesta pesquisa estavam no trabalho de Freitas *et al.* (2016).

Tabela 3. Trabalhos agrupados por métricas.

Métrica	**Trabalho**
Custo de Manutenção Futura	Freitas *et al.* (2016)
Cheiros de Código	Liu *et al.* (2016), Hecht *et al.* (2016), Vidal *et al.* (2016), Vidal *et al.* (2015), Amorim *et al.* (2015), Techapalokul e Tilevich (2015), Fontana *et al.* (2015), Malhotra *et al.* (2015), Schumacher *et al.* (2010)
Frequência de Chamadas	Freitas *et al.* (2016), Ray e Mohapatra (2014), Hosseingholizadeh (2010)
Risco de Falha	Freitas *et al.* (2016), Kasai *et al.* (2013), Ray e Mohapatra (2012), Hosseingholizadeh (2010)
Vulnerabilidade de Mercado	Freitas *et al.* (2016)
Valor de Negócio	Ray e Mohapatra (2012), Hosseingholizadeh (2010)

Com as informações apresentadas neste capítulo, percebe-se a importância da automatização da seleção de componentes para o teste de unidade em aplicações móveis. No próximo capítulo será apresentada a definição proposta do plugin para a seleção de componentes, bem como um estudo para analisar a correlação das métricas usadas no plugin.

CAPÍTULO 3 – SELEÇÃO DE COMPONENTES PARA TESTE UNITÁRIO EM APLICAÇÕES MÓVEIS

Neste capítulo será apresentada a definição do plugin para a seleção de componentes para a escrita de teste unitário, em que os três principais processos do plugin são descritos, como também a relação entre eles. Também é apresentado um estudo que faz a análise de correlação das métricas.

3.1. Visão Geral do Plugin

O plugin proposto chamado SCoTUAM pode ser adicionado à interface de desenvolvimento da IDE *Android Studio*. Ele possui três processos principais: (1) Extração de métricas estáticas, (2) Extração de métricas dinâmicas, de mercado e de negócio e (3) Execução do algoritmo genético para seleção dos componentes a serem testados. Estes processos são apresentados na Figura 3.

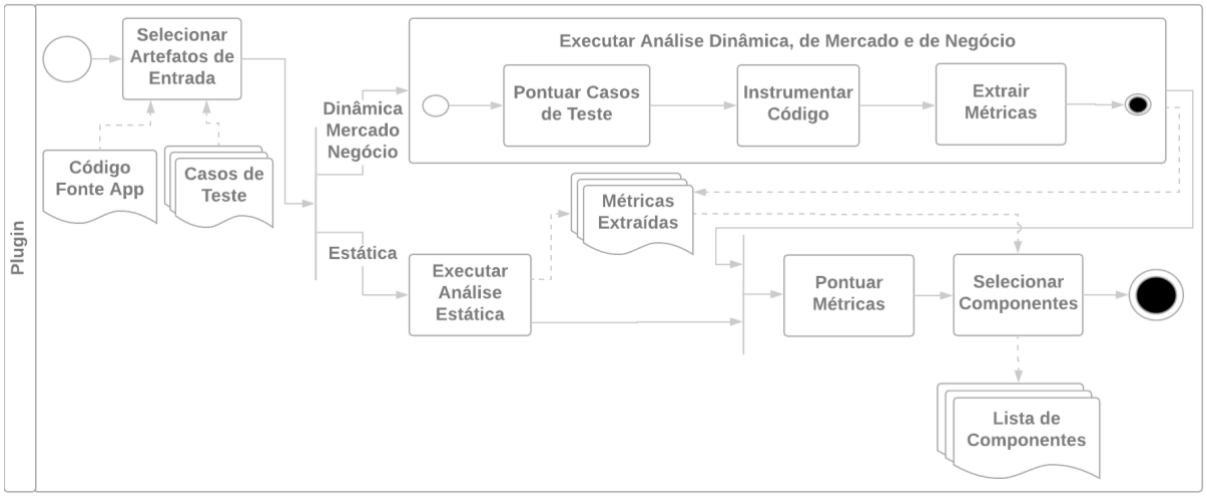


Figura 3. Visão geral do plugin.

A seguir serão detalhadas cada etapa que compõe os processos apresentados na Figura 3.

3.2. Selecionar Artefatos de Entrada

Este plugin tem como artefatos de entrada dois componentes: o código fonte de uma aplicação e os casos de teste projetados para ela.

- **Código Fonte**: será uma aplicação para a plataforma Android implementada na linguagem Java, já disponível no próprio Android Studio;

- **Casos de Teste**: os casos de teste de sistema escritos na linguagem Java e que utilizam o framework *Espresso*[5].

3.3. Executar Análise Estática

Esse processo possibilita a seleção de duas métricas: custo de manutenção futura e cheiros de código.

a) Custo de Manutenção Futura

A ferramenta *JHawk*[6] será usada para extrair as métricas *Halstead*. O valor do custo de manutenção futura é normalizado da seguinte forma: encontra-se o componente com maior valor de *cmf*, e os demais valores são divididos por esse valor. A Tabela 4 mostra um exemplo de valores para essa métrica e a normalização (N.) extraído da análise estática da aplicação *Recurrence*[7] do pacote *com.bleyl.recurrence.activities*. Nesse exemplo, o componente com o valor máximo de *cmf* é o *actionMarkAsDone* da classe *ViewActivity*. Os demais componentes tiveram o valor de *cmf* divido pelo *cmf* do componente *actionMarkAsDone*.

Tabela 4. Custo de manutenção futura.

Classe	Componente	Parâmetros	Retorno	E_i	B_i	cfm_i	N.
MainActivity	onCreate	(Bundle)	void	3678,03	0,18	662,04	0,31
ViewActivity	actionDelete	()	void	2632,24	0,11	289,54	0,13
ViewActivity	actionMarkAsDone	()	void	7999,57	0,26	2079,88	1
ViewActivity	actionShareText	()	Void	1672,95	0,09	150,56	0,07
AboutActivity	launchEmail	(View)	String	1984,83	0,10	198,48	0,09

b) Cheiros de Código (*Code Smell - CS*)

Os cheiros de código que serão detectados são os seguintes: Método Longo (LM), Longa Lista de Parâmetros (LPL), Declarações de Desvio (SS), Método Cérebro (BM), Complexidade Ciclomática (CC). Eles serão extraídos com o auxilio da ferramenta JHawk.

A cada cheiro de código detectado em um componente, será somado 1 ponto para o componente. Em seguida, a quantidade de cheiros de código detectados para cada componente será somado, um exemplo de valores aplicado para a fórmula da Equação 2 dessa métrica é apresentada na Tabela 5. No exemplo, para o componente "*atualizar"* foram

[5] https://google.github.io/android-testing-support-library/docs/espresso/index.html

[6] http://www.virtualmachinery.com/jhawkprod.htm

[7] https://f-droid.org/repository/browse/?fdfilter=recurrence&fdid=com.bleyl.recurrence

detectados 3 cheiros de código: método longo (LM), longa lista de parâmetros (LPL) e método cérebro (BM). O valor de cheiro de código (cs_i) para o componente foi de 0,6, de acordo com a Equação 2.

Tabela 5. Cheiros de código.

Componente	LM	LPL	SW	BM	CC	s_i	cs_i	Normalização
adicionar	1	-	1	-	-	2	0,4	0,66
atualizar	1	1	-	1	-	3	**0,6**	1
excluir	-	1	-	-	-	1	0,2	0,33
editar	1	-	-	1	-	2	0,4	0,66
buscar	-	-	-	-	-	0	0	0
buscarTodos	-	-	-	-	1	1	0,2	0,33

3.4. Executar Análise Dinâmica, de Mercado e de Negócio

Esse processo está dividido em quatro passos: a escolha de dispositivos, a pontuação dos casos de uso, a instrumentação do código e a extração das métricas. Esses passos estão descritos nas seguintes subseções.

3.4.1. Pontuar Casos de Teste

O usuário irá informar um peso de 0 a 5 para cada caso de teste, sendo que 0 é onde o caso de teste não será executado, 1 é pouco importante e 5 é muito importante. Os casos de teste representam os casos de uso do sistema. Ou seja, o usuário irá pontuar cada caso de teste com o peso equivalente do caso de uso que ele representa.

Em geral, em um projeto Android os casos de teste de uma aplicação estão nos seguintes diretórios *~/app/src/test/* ou *~/app/src/androidTest/*. Essa lista de casos de teste será mostrada ao usuário pela interface do plugin SCoTUAM. O usuário então irá fazer apenas a pontuação.

3.4.2. Instrumentar Código

Na etapa de instrumentação, um algoritmo que foi desenvolvido irá interagir com a aplicação e irá modificar a aplicação para inserir os comandos *startMethodTracing* no método *OnCreate* e *stopMethodTracing* no método *OnDestroy* da *Activity* principal da aplicação. Estes dois métodos inseridos como comandos estão contidos na classe *android.os.Debug* e são responsáveis por iniciar e parar a coleta de perfis respectivamente. Isto serve para saber o que está acontecendo durante a execução da aplicação.

3.4.3. Extrair métricas

Para a extração das métricas dinâmicas de mercado e de negócio será necessária a execução dos casos de teste, que será realizada com o auxílio da ferramenta *Spoon*[8], pois está ferramenta gerencia a execução simultânea de um caso de teste em vários dispositivos, como também registra o tempo de execução e se o caso de teste passou ou falhou para cada dispositivo. Serão extraídas quatro métricas: frequência de chamadas, risco de falha, vulnerabilidade de mercado e valor de negócio.

a) Frequência de Chamadas

Após a execução de cada caso de teste, será gerado um arquivo de rastreamento que fornece métricas detalhadas sobre um componente, como o número de chamadas, tempo de execução e tempo gasto executando o componente, esse arquivo será analisado pela ferramenta *traceview*[9] fornecida com o Android SDK e depois será usado uma ferramenta desenvolvida em Freitas *et al.* (2016) para transformar *small code* em código Java. Como resultado, será gerado um conjunto de dados com detalhes dos componentes executados. Essa métrica foi extraída de Freitas *et al.* (2016).

Está métrica irá computar a quantidade de vezes que um componente foi chamado durante a execução dos casos de teste. Para normalizar essa métrica, encontra-se o componente com o maior número de chamadas, e o número de chamadas dos demais componentes será divido por esse valor. Na Tabela 6 é mostrado um exemplo de normalização da frequência de chamadas.

Tabela 6. Frequência de chamada.

Componente	Número de Chamadas	Normalização
adicionar	5	0,5
atualizar	5	0,5
excluir	2	0,2
editar	1	0,1
buscar	**10**	1
buscarTodos	1	0,1

b) Risco de Falha

Os dados necessários para calcular essa métrica provêm do registro da execução de um componente em casos de teste que passaram ou falharam. Foi desenvolvido um algoritmo

[8] http://square.github.io/spoon/

[9] https://developer.android.com/studio/profile/traceview.html

para analisar o resultado da ferramenta *Spoon*. Como resultado dessa análise, será gerado um conjunto de dados de quais casos de teste falharam e passaram para cada componente. Um exemplo de valores para essa métrica pode ser visto na Tabela 7, neste exemplo foi executado 10 casos de teste, sendo que 6 casos de teste passaram e 4 casos de teste falharam. O componente *adicionar* tem 2 casos de teste que passaram de todos os 6 e também falhou 1 caso de teste de todos os 4, este componente ficou com o risco de falha de 0,56, de acordo com a Equação 3.

Tabela 7. Risco de falha.

Componente	p_i	f_i	rf_i	Normalização
adicionar	2/6 = 0,33	1/4 = 0,25	0,56	1
atualizar	3/6 = 0,50	3/4 = 0,75	0,40	0,71
excluir	1/6 = 0,16	1/4 = 0,25	0,39	0,69
editar	3/6 = 0,50	2/4 = 0,50	0,50	0,89
buscar	3/6 = 0,50	3/4 = 0,75	0,40	0,71
buscarTodos	1/6 = 0,16	1/4 = 0,25	0,39	0,69

c) Vulnerabilidade de Mercado

A métrica de vulnerabilidade do mercado é usada para representar a porcentagem do mercado no qual um componente é vulnerável. Na Tabela 8 é mostrado um exemplo onde alguns dispositivos com as seguintes configurações de APIs 19, 21, 22 e 23 tem respectivamente 20,8%; 9,4%; 23,15% e 31,3% de vulnerabilidade média do mercado e o componente *editar* falhou nos dispositivos com as APIs 22 e 23 esse componente ficou com a vulnerabilidade de mercado de 54,4%.

Tabela 8. Vulnerabilidade de Mercado.

Componente	Versão de API				Vulnerabilidade (%)	Normalização
	19	21	22	23		
adicionar	x				20,8	0,38
atualizar	x		x		43,9	0,80
excluir		x			9,4	0,17
editar			x	x	**54,4**	1
buscar	x	x			30,2	0,55
buscarTodos				x	31,3	0,57

d) Valor de Negócio

Foi desenvolvido um algoritmo que analisa o resultado da ferramenta *Spoon* e gera um conjunto de dados de quais casos de teste executaram cada componente. Quando um

caso de teste executar um componente, o peso desse caso de teste será somado ao componente. Essa métrica foi adaptada de Ray e Mohapatra (2012). Na Tabela 9 é mostrado um exemplo de valores para essa métrica. A coluna valor de negócio se refere à soma final de pontos do componente, após a execução dos casos de teste.

Tabela 9. Valor de negócio.

Componente	Valor de Negócio	Normalização
adicionar	10	0,7
atualizar	7	0,5
excluir	3	0,2
editar	9	0,6
buscar	5	0,3
buscarTodos	**13**	1

3.5. Pontuar Métricas

Essa etapa permite ao usuário selecionar uma porcentagem de utilização para cada métrica com uma das seguintes porcentagens {0; 0,2; 0,4; 0,6; 0,8 e 1,0}, sendo que 0 é onde será utilizado 0% da métrica, ou seja, a métrica não será utilizada, 0,2 é utilizado 20% do valor da métrica, e assim sucessivamente.

3.6. Selecionar Componentes

As métricas utilizadas correspondem às funções objetivos do NSGA-II, pois na concepção dessas métricas foi planejado que elas fariam parte das funções objetivos. Por esse motivo, todas as métricas são normalizadas no intervalo [0;1] para facilitar a combinação dos objetivos no NSGA-II. Um algoritmo irá unir os resultados da extração das métricas em único arquivo que servirá de entrada para o algoritmo genético. A implementação do NSGA-II será na linguagem Java.

3.7. Lista de Componentes

O algoritmo NSGA-II implementa o conceito de dominância, ou seja, a População Total é classificada em fronteiras de acordo com o grau de dominância. Os indivíduos que estão localizados na primeira fronteira são considerados os melhores, enquanto que os indivíduos

na última fronteira são os piores. Então pode-se encontrar melhores soluções (pontos mais próximos da região de *Pareto*) (Marinho, 2009).

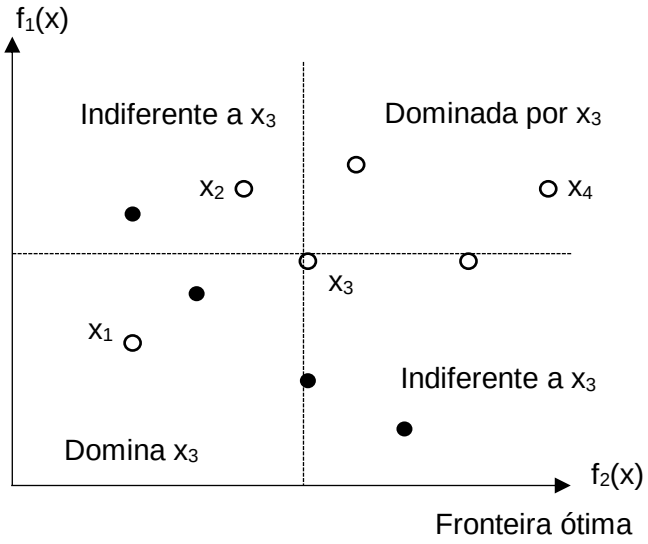


Figura 4. Exemplo de dominância de Pareto (Iannoni e Morabito, 2006).

Por exemplo, na Figura 4, para os pontos x_3 e x_4, a relação é: $f_1(x_3) < f_1(x_4)$ e $f_2(x_3) < f_2(x_4)$. Assim, a solução x_3 domina a solução x_4. Considerando os pontos x_2 e x_3, tem-se $f_1(x_2) > f_1(x_3)$ e $f_2(x_2) < f_2(x_3)$, então, as soluções x_2 e x_3 são indiferentes entre si, ou seja, não há dominância entre elas. A solução x^* é chamada eficiente ou Pareto ótima se não há outra solução $x \in X^*$ que domine x^*. O conjunto de soluções eficientes determina a curva de compromisso (*trade-off*) chamada curva Pareto ótima (linha sobre os pontos em negrito da Figura 4 (Iannoni e Morabito, 2006). Então a lista de componentes da solução Pareto ótima será apresentada ao usuário no formato: pacote, classe, componente, parâmetros e retorno, como demonstrado na Tabela 10.

Tabela 10. Exemplo da lista de componentes.

Pacote	Classe	Componente	Parâmetros	Retorno
Pacote 1	Classe 3	Componente 1	()	void
Pacote 1	Classe 5	Componente 3	()	void
Pacote 1	Classe 10	Componente 6	(int)	string
Pacote 2	Classe 13	Componente 1	()	void
Pacote 2	Classe 1	Componente 2	(String)	void
Pacote 3	Classe 6	Componente 10	()	int
Pacote 3	Classe 9	Componente 7	()	float

3.8. Análise de Correlação das Métricas

Esse estudo tem como propósito analisar a correlação entre as 6 métricas descritas na seção 2.3.1 do capítulo 2. A perspectiva do estudo é do ponto de vista do pesquisador. Nesse contexto, o pesquisador irá executar o experimento e depois fazer a análise estatística de correlação das métricas. Na Tabela 11 é possível visualizar um resumo das métricas.

Tabela 11. Resumo das métricas.

Sigla	Descrição	Fórmula
CMF	Custo de Manutenção Futura	$cfm_i = E_i * B_i$
CS	Cheiro de Código	$cs_i = \frac{s_i}{5}$
FC	Frequência de Chamadas	-
RF	Risco de Falha	$rf_i = \frac{p_i}{p_i + f_i}$
VM	Vulnerabilidade de Mercado	-
VN	Valor de Negócio	-

3.8.1. Objetivo de Pesquisa

Considerando que as métricas custo de manutenção futura, cheiro de código, frequência de chamada, risco de falha, vulnerabilidade de mercado e valor de negócio compõem a função objetivo de maximização do benefício, se faz necessário saber se existe uma correlação entre as mesmas, para uma métrica não possuir o peso dobrado no algoritmo genético.

3.8.2. Planejamento

O estudo experimental possui três fases: seleção das aplicações móveis, seleção dos dispositivos móveis e execução do estudo.

- **Seleção das Aplicações Móveis**: A seleção das aplicações está baseada em três parâmetros, sendo que duas características foram baseadas em Fazzini *et al.* (2017), que são a diversidade (categorias diferentes de aplicações) e autocontenção (sem maiores configurações a serem implementadas), e o outro parâmetro é possuir scripts de teste de sistema. Na Tabela 12 são mostradas as aplicações selecionadas e alguns de seus atributos.
    - **ID** - um número para identificar a aplicação;
    - **Aplicação** - Nome da aplicação;

- **Downloads** - quantidade de downloads na Google Play Store;
- **Código (LOC)** - quantidade de linhas de código da aplicação;
- **Componentes** - quantidade de componentes que os testes chamaram **/** quantidade total de componentes que tem na aplicação;
- **Testes** - quantidade de testes de GUI da aplicação;
- **Categoria** - categoria a qual a aplicação pertence.

Tabela 12. Aplicações móveis para o experimento.

ID	Aplicação	Downloads	Código (LOC)	Componentes	Testes	Categoria
1	Pocket Code (PC)	100k – 500k	83k	496**/**7610	420	Educação
2	Pocket Paint (PP)	100k – 500k	11k	239**/**916	33	Ferramentas

- **Seleção dos Dispositivos Móveis**: Para este estudo, os dispositivos móveis foram selecionados com base nas versões de SDK Android mais usadas por aplicações publicadas na *Google Play Store*[10] (dados atualizados em setembro de 2017), no tamanho e resolução da tela. Na Tabela 13 é possível ver a porcentagem de mercado de algumas das versões de API mais usadas e a quantidade de dispositivos que serão utilizados de cada versão de API. Com base nessa informação e de acordo com o estudo de Vilkomir *et al.* (2015) que avaliou a eficácia de revelação de falhas utilizando várias combinações com quantidade diferentes de dispositivos e encontrou o resultado que 13 dispositivos é uma quantidade boa para a revelação de falhas, então definimos os 13 dispositivos que serão utilizados no experimento, listados na Tabela 13.

Tabela 13. Dispositivos móveis usados no experimento.

ID	Versão	Nome da Versão	API	Porcentagem	Modelo	Tamanho	Densidade
D1	4.4.2	KitKat	19	15,38%	BLU	480x800	230
D2					LG E977	768x1280	320
D3	5.0	Lollipop	21	7,69%	Galaxy Note 3 LTE	1080x1920	480
D4	5.1		22	23,07%	Alcatel PIXI4 (4.0)	320x480	160
D5					Alcatel PIXI4 (6.0)	1280X720	320
D6					Galaxy J1 Mini	480x800	240
D7	6.0	Marshmallow	23	38,46%	ASUS Zenfone Go LTE	720x1280	320
D8					Galaxy A9	1080x1920	420
D9					LG X Power	720x1280	320
D10					Moto E	960x540	240

[10] https://developer.android.com/about/dashboards/index.html

D11					Moto Z Power Edition	1440x2560	640
D12	7.0	Nougat	24	15,38%	Galaxy A5	720x1280	294
D13					Moto G 5	1080x1920	480

- **Execução do Estudo**: Para o processo de execução do estudo, para cada aplicação foi aberto o plugin SCoTUAM no Android Studio, foi selecionado randomicamente um valor entre 1 e 5 para cada script de teste. Então os scripts de teste foram executados ao mesmo tempo em todos os dispositivos móveis usados no experimento. Posteriormente, iniciou-se a extração do valor das métricas para a aplicação.

3.8.3. Resultados e Discussão

Foram extraídas as métricas estáticas, dinâmicas, de mercado e de negócio das 2 (duas) aplicações, sendo que os scripts de teste de cada aplicação foram executados em 13 dispositivos. O valor dessas métricas para cada componente por aplicação está disponível em "https://goo.gl/7YrDk7".

O programa SPSS[11] foi usado como apoio à verificação da existência ou não de correlação entre as métricas, com o objetivo de gerar o coeficiente de correlação de Pearson, esse coeficiente é adequado quando os dados são quantitativos e tem a distribuição normal. A Tabela 14 extraída de Rumsey (2009) ajuda a interpretar o valor gerado pela correlação de Pearson.

Tabela 14. Interpretando o valor de correlação.

Valor	Correlação
0	Nenhuma relação linear
+0,3 ou -0,3	Fraca
+0,5 ou -0,5	Moderada
+0,7 ou -0,7	Forte
+1 ou -1	Perfeita

a) Resultado Aplicação *Pocket Code*

As métricas dinâmicas, de mercado e de negócio dependem do resultado dos scripts de teste de sistema. Na Figura 1 é mostrado o resultado da execução dos 420 scripts de teste da aplicação *Pocket Code* pelo plugin SCoTUAM, onde observa-se que 251 (60%) dos scripts

[11] https://www.ibm.com/analytics/us/en/technology/spss/

de teste falharam em ao menos um dispositivo e 169 (40%) dos scripts passaram em todos os dispositivos.

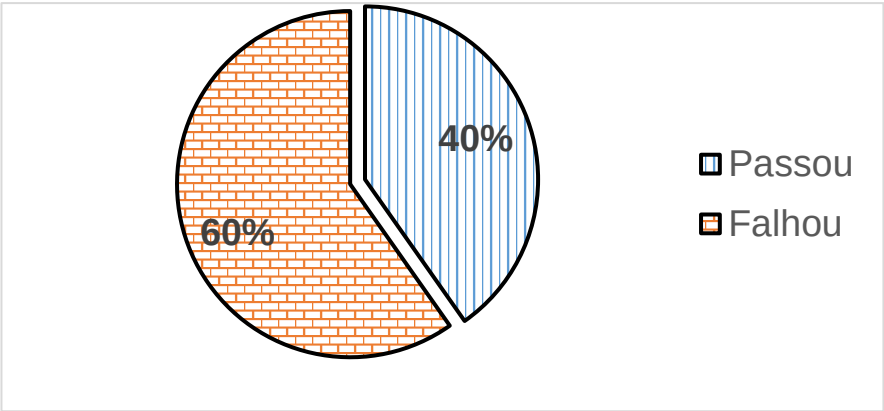


Figura 5. Resultado dos scripts de teste de sistema *Pocket Code*.

A Tabela 15 mostra a quantidade de scripts de teste que falharam por dispositivo. Analisando esses dados é possível observar que a maioria dos scripts de teste falharam em quase todos os dispositivos, pois a mediana da porcentagem de falhas por dispositivo chega a 72,51% do total dos scripts de teste que falharam.

Tabela 15. Quantidade de Falhas por dispositivo *Pocket Code*.

ID Dispositivo	Total de Falhas	Porcentagem de Falhas
D1	184	73,31%
D2	192	76,49%
D3	180	71,71%
D4	196	78,09%
D5	177	70,52%
D6	184	73,31%
D7	182	72,51%
D8	173	68,92%
D9	179	71,31%
D10	185	73,71%
D11	176	70,12%
D12	184	73,31%
D13	178	70,92%
Média	**182,30**	**72,63%**
Mediana	**182**	**72,51%**

Na Tabela 16 está sendo mostrado o valor do coeficiente de correlação de Pearson para a aplicação *Pocket Code*. A correlação entre as métricas risco de falha (RF) e vulnerabilidade de mercado (VM) é forte provavelmente devido 161 (64,14%) scripts de testes ter falhado em todos os dispositivos e a mediana de falha de script de teste por dispositivo ser de 72,51%, fazendo com que a métrica vulnerabilidade de mercado ficasse com seu valor perto de 1 (um) para a maioria dos componentes, pois o valor dessa métrica depende dos

dispositivos em que a falha ocorreu. A métrica risco de falha também ficou com seus valores perto de 1 (um) para cada componente, fazendo com que a correlação entre as duas métricas ficasse forte. No entanto se os testes falharem com uma maior variabilidade entre os dispositivos, o valor da métrica vulnerabilidade de mercado terá uma maior variação, fazendo com que mude o valor de correlação entre as duas métricas.

Tabela 16. Correlação de Pearson *Pocket Code*.

	CFM	CS	VN	FC	VM	RF
CFM	1					
CS	0,19**	1				
VN	0,009	-0,006	1			
FC	0	0,024*	0,238**	1		
VM	-0,001	0	0,528**	0,122**	1	
RF	-0,001	0,002	0,473**	0,119**	0,987**	1

Nota: ** A correlação é significativa no nível 0,01 (2 extremidades);
* A correlação é significativa no nível 0,05 (2 extremidades).

b) Resultado Aplicação *Pocket Paint*

A Figura 6 mostra o resultado da execução dos 33 scripts de teste de sistema da aplicação *Pocket Paint* pelo plugin SCoTUAM. Onde 27 (82%) scripts de teste falharam em pelo menos um dispositivo e 6 (18%) scripts de teste passaram em todos os dispositivos.

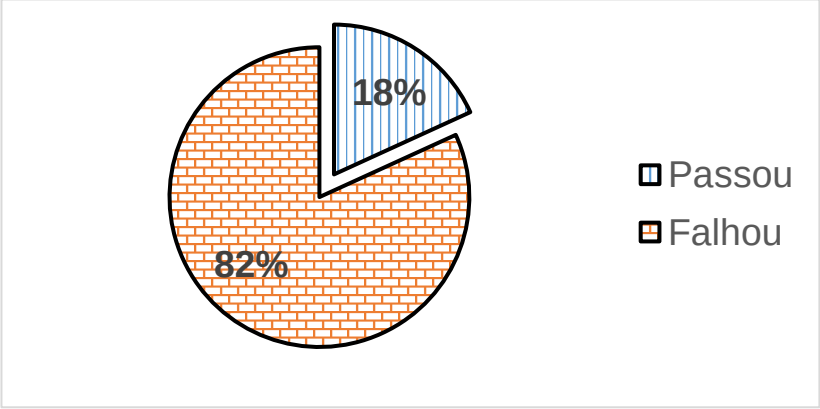


Figura 6. Resultado dos scripts de teste de sistema *Pocket Paint*.

Na Tabela 17 é mostrada a quantidade de scripts de teste que falhou por dispositivo para a aplicação *Pocket Paint*. Nesta aplicação, a mediana da porcentagem de falhas nos dispositivos foi de 74,07%. Isso quer dizer que a maioria dos scripts de teste falhou em quase todos os dispositivos. Sendo que 17 (62,96%) scripts de teste falharam em todos os dispositivos. Esse grande número de falhas tanto na aplicação *Pocket Code* quanto na *Pocket Paint* remete o problema de compatibilidade de scripts de teste.

Tabela 17. Quantidade de Falhas por dispositivo *Pocket Paint*.

ID Dispositivo	Total de Falhas	Porcentagem de Falhas
D1	22	81,48%
D2	20	74,07%
D3	22	81,48%
D4	22	81,48%
D5	20	74,07%
D6	19	70,37%
D7	20	74,07%
D8	19	70,37%
D9	19	70,37%
D10	19	70,37%
D11	24	88,89%
D12	19	70,37%
D13	20	74,07%
Média	**20,38**	**75,50%**
Mediana	**20**	**74,07%**

Na **Tabela 18** está sendo mostrado o valor do coeficiente de correlação de Pearson para a aplicação *Pocket Paint*. Pode-se observar que existe uma correlação forte entre as métricas vulnerabilidade de mercado e valor de negócio, no entanto, o valor de negócio de cada componente depende do peso dado a cada script de teste, com isso a correlação pode mudar de acordo com a pontuação dada a cada script de teste.

A explicação para a forte correlação entre as métricas vulnerabilidade de mercado e risco de falha se aplica a mesma informada na aplicação *Pocket Code*, pois aqui também tem uma pequena variabilidade nos dispositivos em que os scripts de teste falharam.

Tabela 18. Correlação de Pearson *Pocket Paint*.

	CFM	**CS**	**VN**	**FC**	**VM**	**RF**
CFM	1					
CS	0,324**	1				
VN	-0,024	-0,032	1			
FC	-0,009	0,006	0,362**	1		
VM	-0,031	-0,041	0,732**	0,266**	1	
RF	-0,028	-0,053	0,413**	0,159**	0,882**	1

Nota: ** A correlação é significativa no nível 0,01 (2 extremidades).

3.8.4. Ameaças à Validade

Este estudo possui ameaças internas, externas, de construção e de validade de conclusão que devem ser examinadas. Esta seção detalhará cada uma.

- **Validade interna**

  Se o tratamento faz com que o resultado (o efeito) aconteça.

1) **Instrumentação***:* Foi realizada a alteração no código fonte das aplicações para incluir suporte a coleta do perfil de cada aplicação, pois não era possível coletar esses dados sem essa instrumentação. Essa instrumentação não influência no funcionamento da aplicação.

2) **Seleção dos Sujeitos***:* A escolha das aplicações móveis não foi feita aleatoriamente, uma vez que foram selecionados com base na disponibilidade de seu código fonte e seus scipts de teste.

- **Validade externa**

  Está preocupada com a generalização.

1) **Generalização dos Resultados***:* Para diminuir essa ameaça foram utilizadas duas aplicações de diferentes categorias, o estudo pode não ser representativo para outras categorias de aplicações móveis.

2) **Ambiente Experimental***:* Embora, o ambiente experimental fosse um ambiente acadêmico, a infraestrutura computacional utilizada (dispositivos reais) representa o mesmo que na indústria por testadores.

- **Validade do construto**

  Que o tratamento reflete a construção da causa. Que o resultado reflete a construção do efetuar.

1) **Viés Mono-operação**: Neste estudo utilizaram-se duas aplicações móveis de duas categorias diferentes: educação e ferramentas.

- **Validade Conclusão**

  Está preocupado com a relação entre o tratamento e os resultados.

1) **Confiança das Medidas**: Levando em consideração o fato de que os resultados das execuções de testes anteriores poderiam afetar as próximas execuções de teste, depois de executar cada script de teste, o plugin SCoTUAM executa um procedimento de teste para desinstalar e limpar a aplicação sobre teste (em inglês, Application Under Test - AUT). O peso de cada caso de uso foi atribuído randomicamente com o intuito de simular as opções que teria o usuário.

2) **Irrelevâncias Aleatórias na Configuração Experimental**: Elementos fora do ambiente experimental podem interromper os resultados, como uma mensagem ou uma chamada recebida. Portanto, os dispositivos móveis foram colocados em modo avião, já que as AUTs não precisavam de conexão à internet para funcionar. Outra configuração nos dispositivos móveis foi configurar a opção mantenha-se acordado para que a tela não entre no modo de espera durante a execução do teste.

3.9. Considerações Finais

Os resultados mostraram a possibilidade de usar as métricas code smell, custo de manutenção futura, frequência de chamada, risco de falha, vulnerabilidade de mercado e valor de negócio combinadas numa solução para a seleção de componentes. O uso das métricas dinâmicas, de mercado e de negócio ajudam no teste unitário, porém ainda tem a dependência de um problema existente que é a compatibilidade dos scripts de teste. Considerando que esse problema de compatibilidade fosse resolvido, o uso dessas métricas teria mais benefícios para o teste de unidade em aplicações móveis.

Nesse capítulo foram apresentados os detalhes do processo de concepção do plugin para seleção de componentes. Em seguida, foi apresentado um estudo com o objetivo de verificar se existe correlação entre as métricas escolhidas para fazer parte do plugin.

O Próximo capítulo irá apresentar as tecnologias utilizadas para a implementação do plugin chamado SCoTUAM, como também uma descrição de suas funcionalidades.

CAPÍTULO 4 – IMPLEMENTAÇÃO DO PLUGIN SCOTUAM

Neste capítulo serão apresentadas as tecnologias usadas para o desenvolvimento do plugin SCoTUAM, bem como descrever as suas funcionalidades. Lembrando que as partes que compõem o plugin SCoTUAM foram descritas no capítulo anterior.

4.1. Tecnologias

O plugin SCoTUAM foi implementado para ser utilizado no Android Studio, que é um ambiente de desenvolvimento integrado para desenvolver aplicações para a plataforma Android. O *Android Studio* foi escolhido por ele ser o ambiente de desenvolvimento oficial do Android.

Para o desenvolvimento do plugin SCoTUAM, foram utilizadas algumas tecnologias como:

- ***Intellij IDEA*:** Uma interface de desenvolvimento que permite criar plug-ins totalmente compatíveis com o *Android Studio;*
- **Linguagem Java:** A linguagem Java foi escolhida para o desenvolvimento do plugin SCoTUAM, essa é uma linguagem de programação interpretada e orientada a objetos;
- **JHawk:** Essa ferramenta coleta métricas em quatro níveis diferentes. Os níveis de método, classe, pacote e sistema. Ela foi utilizada para auxiliar na extração das métricas estáticas;
- ***Comment Remover*[12]:** É uma biblioteca de remoção de comentários de código-fonte para Java™ 7 e versões acima. Ela também suporta JavaScript, HTML, CSS, Propriedades, JSP e comentários XML. Essa biblioteca foi utilizada para remover comentários de arquivos Java e XML;
- ***Android Debug Bridge* (ADB)[13]:** É uma ferramenta de linha de comando versátil que permite a comunicação com uma instância de emulador ou com um dispositivo Android conectado. Ela foi utilizada para pegar o ID dos dispositivos conectados e para copiar o arquivo no formato ".trace" do dispositivo para o computador;

[12] https://github.com/ertugrulcetin/CommentRemover/blob/master/Readme.md

[13] https://developer.android.com/studio/command-line/adb.html?hl=pt-br

- ***Spoon*:** Essa ferramenta faz a execução de um script de teste em vários dispositivos simultaneamente e exibe os resultados de forma significativa. Ela foi utilizada para a execução dos scripts de teste;
- ***TraceView*:** É uma ferramenta que fornece representações gráficas de registros de rastreamento. Ela foi usada para converter o arquivo de log do formato ".trace" para ".csv";
- ***Framework jMetal*:** Para auxiliar na implementação do algoritmo genético foi escolhido o framework jMetal, a curva de Pareto é gerado pelo framework jMetal. Ele possui código aberto, baseia-se em Java para otimização multiobjetivo com metaheurísticas (Nebro, 2015).

4.2. Interface Gráfica do Usuário

Na **Figura 7** é apresentado o menu para abrir a tela inicial do plugin SCoTUAM, onde é possível observar que o item do menu fica ao lado direito da opção "Help", com o nome "Plugin SCoTUAM" e "SCoTUAM".

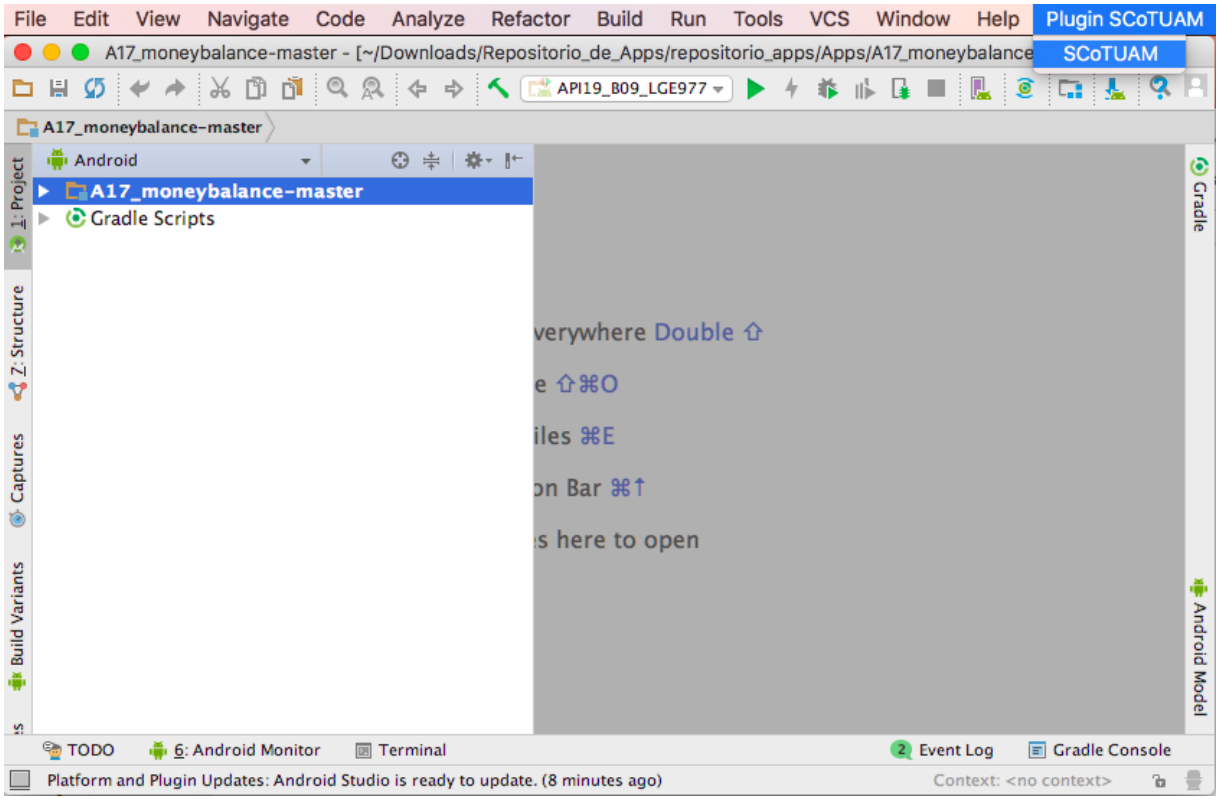


Figura 7. Menu plugin SCoTUAM.

Após clicar na opção SCoTUAM, é aberta a tela principal do plugin, mostrada na **Figura 8**. A tela inicial possui 3 campos. O campo *Source Code Folder* é preenchido automaticamente com o caminho do código fonte da aplicação aberta no *Android Studio* e não pode ser alterado. O campo *SDK Folder* também é preenchido automaticamente com o caminho padrão da pasta

SDK e pode ser alterado caso esteja incorreto. O campo *Choose Type Of Metrics* possibilita escolher se irá utilizar métricas estáticas e/ou dinâmicas na seleção de componentes.

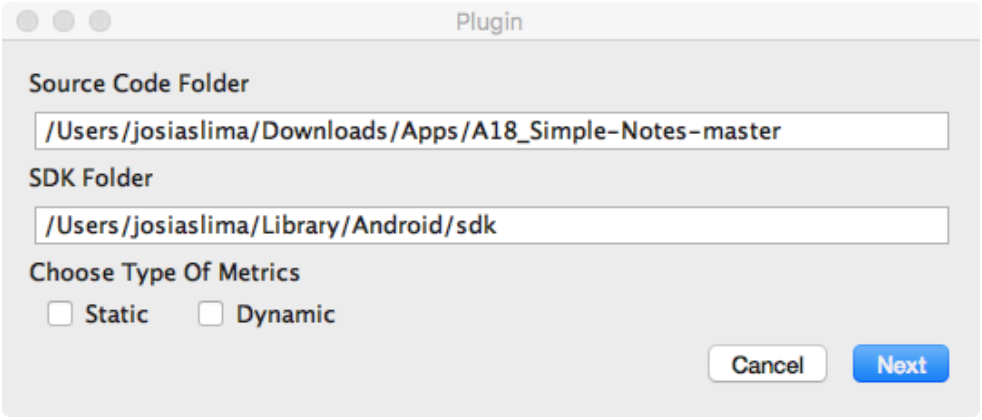


Figura 8. Tela inicial do plugin SCoTUAM.

Quando escolhe-se utilizar métricas dinâmicas, é mostrada uma tela com todos os casos de teste de sistema que a aplicação possui, conforme mostrado na **Figura 9**. Nesta tela, o usuário irá pontuar os casos de teste de acordo com o valor do caso de uso que eles exercitam. A pontuação será em uma escala ordinal de 0 a 5, sendo que 0 indica que o caso de teste não será executado, 1, que ele é pouco importante e 5, muito importante.

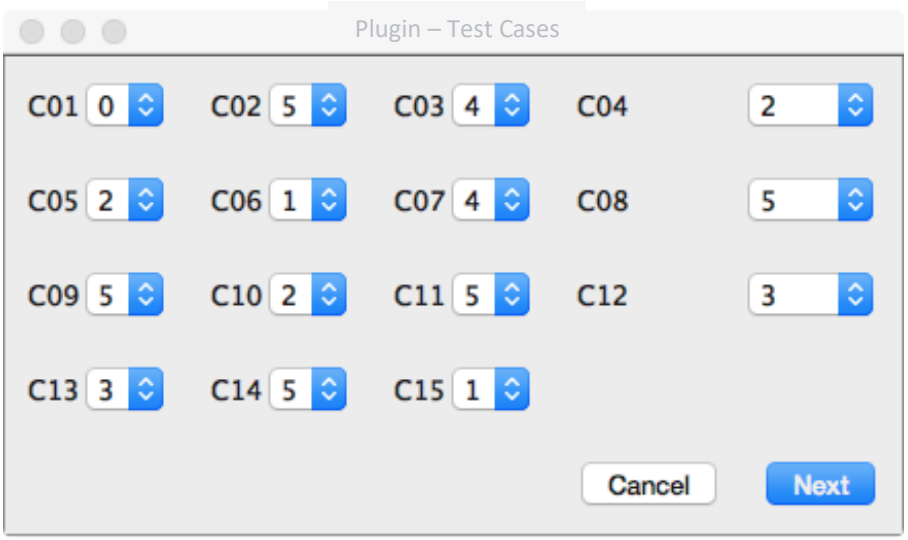


Figura 9. Tela casos de teste da aplicação.

Após o usuário escolher os tipos de métricas, e no caso de métricas dinâmicas pontuar os casos de teste, será mostrada uma tela de espera enquanto se executa os casos de teste e são extraídas as métricas estáticas e dinâmicas, conforme mostrado na **Figura 10**.

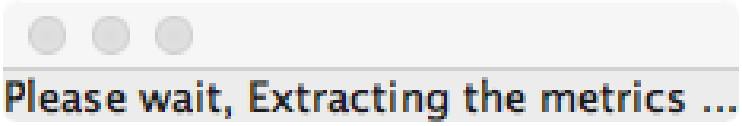


Figura 10. Tela de espera, extração das métricas.

Na **Figura 11** é mostrada a tela com os parâmetros a serem informados ao algoritmo genético. Na primeira parte o usuário irá selecionar uma porcentagem de utilização para cada métrica com uma das seguintes porcentagens {0; 0.2; 0.4; 0.6; 0.8 e 1.0}, sendo que 0 é onde será utilizado 0% da métrica, ou seja, a métrica não será utilizada, 0.2 é utilizado 20% do valor da métrica, e assim sucessivamente.

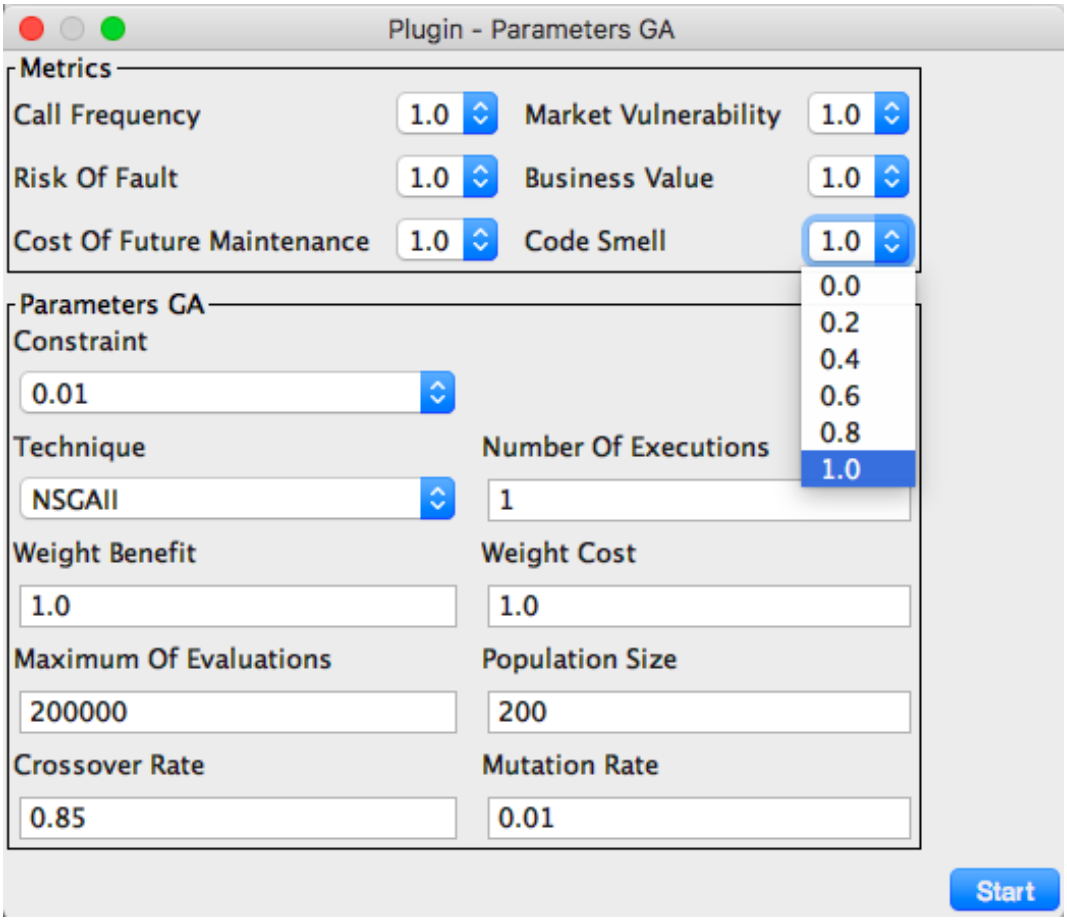


Figura 11. Parâmetros do algoritmo genético.

Na segunda parte, o usuário pode informar os seguintes parâmetros:

- ***Constraint*:** essa restrição está relacionada à quantidade total de componentes que se pretende testar (1% = 0.01; 5% = 0.05; 10% = 0.1; 15% = 0.15; 20% = 0.20; 50% = 0.50; 70% = 0.70; 100% = 1.00);
- ***Technique*:** por enquanto está disponível apenas o algoritmo genético NSGAII;
- ***Number Of Executions*:** a quantidade de execuções do algoritmo genético;
- ***Weight Benefit*:** o peso da função de benefício;
- ***Weight Cost*:** o peso da função de custo;

- ***Maximun Of Evaluations*:** a quantidade máxima de avaliações;
- ***Population Size*:** o tamanho da população;
- ***Crossover Rate*:** a taxa de crossover;
- ***Mutation Rate*:** a taxa de mutação.

A **Figura 12** mostra a mensagem exibida ao usuário enquanto o plugin SCoTUAM faz a seleção dos componentes para o teste de unidade. Essa tela é mostrada após o usuário clicar no botão “Start” da **Figura 11**.

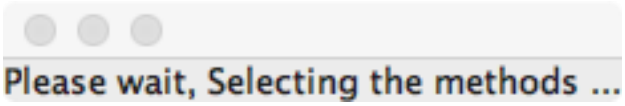
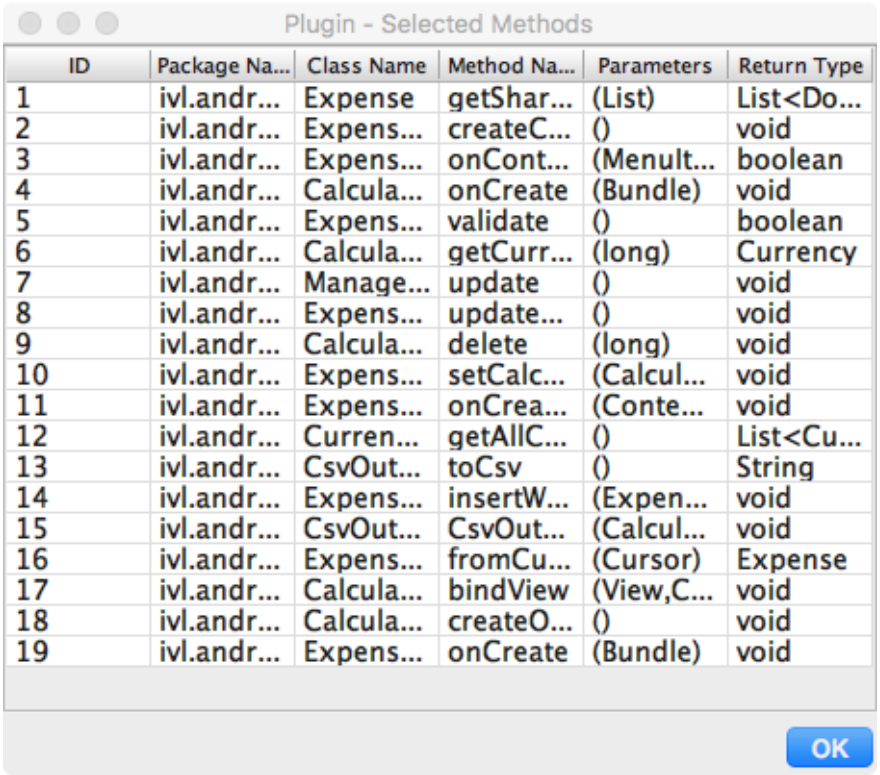


Figura 12. Tela de espera, seleção dos componentes.

A **Figura 13** é apresenta a tela exibida ao usuário com os componentes selecionados pelo algoritmo genético. Esses são os componentes com o melhor valor em relação aos critérios usados para avaliar o valor do custo x benefício do teste de unidade dos componentes da aplicação móvel.



ID	Package Na...	Class Name	Method Na...	Parameters	Return Type
1	ivl.andr...	Expense	getShar...	(List)	List<Do...
2	ivl.andr...	Expens...	createC...	()	void
3	ivl.andr...	Expens...	onCont...	(Menult...	boolean
4	ivl.andr...	Calcula...	onCreate	(Bundle)	void
5	ivl.andr...	Expens...	validate	()	boolean
6	ivl.andr...	Calcula...	getCurr...	(long)	Currency
7	ivl.andr...	Manage...	update	()	void
8	ivl.andr...	Expens...	update...	()	void
9	ivl.andr...	Calcula...	delete	(long)	void
10	ivl.andr...	Expens...	setCalc...	(Calcul...	void
11	ivl.andr...	Expens...	onCrea...	(Conte...	void
12	ivl.andr...	Curren...	getAllC...	()	List<Cu...
13	ivl.andr...	CsvOut...	toCsv	()	String
14	ivl.andr...	Expens...	insertW...	(Expen...	void
15	ivl.andr...	CsvOut...	CsvOut...	(Calcul...	void
16	ivl.andr...	Expens...	fromCu...	(Cursor)	Expense
17	ivl.andr...	Calcula...	bindView	(View,C...	void
18	ivl.andr...	Calcula...	createO...	()	void
19	ivl.andr...	Expens...	onCreate	(Bundle)	void

Figura 13. Tela que mostra os componentes selecionados.

4.3. Limitações

Algumas limitações do plugin SCoTUAM são:

- Compatível com os sistemas operacionais Windows 7 ou versão superior e macOS Sierra 10.12.15 ou versão superior;

- Compatível com o Android Studio 2.2.3 ou versão superior;
- O código fonte da aplicação Android precisa ser a linguagem Java;
- Para a utilização das métricas dinâmicas, de mercado e negócio, a aplicação precisa possuir scripts de teste de sistema utilizando o framework espresso (versão 2.2 em diante), sendo um caso de teste por arquivo Java.

4.4. Considerações Finais

Neste capítulo foram apresentadas as tecnologias que foram utilizadas para a implementação do plugin SCoTUAM, bem como a sua interface gráfica. O plugin apresentado tem como função auxiliar o desenvolvedor na escolha dos componentes para a escrita de teste de unidade.

O plugin SCoTUAM é uma primeira versão de um protótipo que ainda carece de evolução para poder entrar em funcionamento de forma estável em ambientes de desenvolvimentos reais. No entanto, o mesmo pode ser usado em ambiente acadêmico para futuros projetos de pesquisas. O próximo capítulo irá explicar em detalhes o estudo de viabilidade que foi planejado e executado, assim como os resultados obtidos.

CAPÍTULO 5 – ESTUDO DE VIABILIDADE

Neste capítulo será apresentado um estudo experimental para verificar a viabilidade do plugin SCoTUAM, avaliado do ponto de vista da eficácia em relação à seleção de componentes realizada manualmente no que diz respeito à quantidade de componentes com erros selecionados.

5.1. Objetivo do Estudo

O propósito deste estudo é responder à questão: "A utilização do plugin SCoTUAM é viável, analisando a sua eficácia comparado com a seleção manual dos componentes para teste de unidade em aplicações móveis na plataforma Android?".

Por eficácia, entende-se neste estudo como a quantidade de componentes com erros selecionados. A seleção manual dos componentes refere-se à seleção que o especialista[14] faz dos componentes que serão criados os testes de unidade.

A perspectiva é do ponto de vista do pesquisador. Nesse contexto, o pesquisador irá comparar a lista de componentes selecionados pelo plugin SCoTUAM com a lista gerada a partir da seleção manual realizada por especialistas.

5.1.1. Objetivos Específicos

- **Analisar** a seleção automatizada de componentes utilizando o plugin SCoTUAM;
- **Com o propósito de** caracterizá-lo;
- **Com relação à** eficácia;
- **Do ponto de vista do** pesquisador;
- **No contexto** da seleção de componentes para a criação de teste de unidade em aplicações móveis.

5.1.2. *Questão de Pesquisa e Métrica*

- Q1. Qual é a eficácia do plugin SCoTUAM na seleção de componentes em relação à seleção realizada manualmente no que diz respeito à quantidade de componentes com erros selecionados?
    - Métrica: quantidade de componentes com erro selecionados.

[14] Profissionais com experiência na realização do teste de unidade em aplicações móveis Android.

5.2. Planejamento do Estudo

Este estudo tem como objetivo a seleção de componentes por meio da utilização do plugin SCoTUAM, no qual serão usadas aplicações móveis da plataforma Android que contenham scripts de testes automatizados. Com os resultados obtidos a partir da realização desse estudo, será possível caracterizar o plugin SCoTUAM em relação à seleção manual de componentes. Esse estudo de viabilidade busca construir um corpo de conhecimento que permita avaliar a viabilidade do plugin proposto. Com isso, espera-se que os resultados obtidos e o corpo de conhecimento decorrente de sua condução forneçam informações que permitam evoluir o plugin e melhorar seu uso na seleção de componentes para teste de unidade em aplicações móveis.

5.2.1. Identificação dos Componentes com Erro

A identificação dos componentes com erro ocorrerá por meio da análise dos *commits* de correção de erros. Para identificar esses *commits*, será realizada uma busca manual nas mensagens dos *commits* utilizando as palavras-chaves *fix*, *problem, incorrect, correct*, conforme (Mockus e Votta, 2000). Para amenizar as ameaças à validade deste estudo, serão analisados somente os *commits* da versão da aplicação utilizada nos experimentos.

Tendo em vista não ter sido encontrada uma lista de componentes selecionados por especialistas com o propósito de criar teste de unidade para aplicações móveis na plataforma Android, será necessário que especialistas realizem a seleção de componentes de forma manual, que no caso deste estudo ocorrerá da seguinte forma: cada especialista irá receber um documento explicando sobre as principais funcionalidades e características da aplicação móvel em que ele irá realizar a seleção de componentes. Após isso, o especialista irá selecionar os componentes para teste de unidade e irá descrever quais critérios foram utilizados para a seleção. Depois de se obter as listas de componentes selecionados manualmente para cada aplicação, o pesquisador irá avaliar a eficácia da seleção de cada especialista e posteriormente irá comparar à seleção do melhor especialista de cada aplicação com a seleção do plugin SCoTUAM.

5.2.2. Cenários Avaliados

Para a comparação da seleção de especialistas com a seleção do plugin SCoTUAM, 63 distintos cenários de priorização foram construídos para simular o amplo espectro de realidades diversas presentes na indústria de software. Os cenários baseiam-se em seis critérios: Custo de manutenção futura (CMF), Propensão ao defeito (PD), Frequência de chamada (FC), Risco de falha (RF), Vulnerabilidade de mercado (VM) e Valor de negócio (VN).

No primeiro cenário (C01), a prioridade é selecionar componentes com alta taxa de custo de manutenção futura para testes de unidade. No segundo (C02), seleciona-se componentes com alto grau de prontidão a defeito (cheiros de código). Esta regra é seguida até o sexto cenário (C06), de acordo com os outros critérios. Do sétimo cenário (C07) em diante, é gerado um arranjo que incorpora todas as combinações de critério entre eles, conforme apresentado na Tabela 19.

Por exemplo, no décimo terceiro cenário (C13), os componentes priorizados para a seleção são definidos como aqueles com uma alta taxa de propensão de defeito e risco de falha. O valor normalizado dos critérios considerados foi usado para construir os cenários.

Tabela 19. Cenários de priorização da seleção de componentes.

ID	CRITÉRIOS	ID	CRITÉRIOS	ID	CRITÉRIOS	ID	CRITÉRIOS
C01	CMF	C17	FC x VM	C33	PD x FC x VM	C49	CMF x FC x RF x VN
C02	PD	C18	FC x VN	C34	PD x FC x VN	C50	CMF x FC x VM x VN
C03	FC	C19	RF x VM	C35	PD x RF x VM	C51	CMF x RF x VM x VN
C04	RF	C20	RF x VN	C36	PD x RF x VN	C52	PD x FC x RF x VM
C05	VM	C21	VM x VN	C37	PD x VM x VN	C53	PD x FC x RF x VN
C06	VN	C22	CMF x PD x FC	C38	FC x RF x VM	C54	PD x FC x VM x VN
C07	CMF x PD	C23	CMF x PD x RF	C39	FC x RF x VN	C55	PD x RF x VM x VN
C08	CMF x FC	C24	CMF x PD x VM	C40	FC x VM x VN	C56	FC x RF x VM x VN
C09	CMF x RF	C25	CMF x PD x VN	C41	RF x VM x VN	C57	CMF x PD x FC x RF x VM
C10	CMF x VM	C26	CMF x FC x RF	C42	CMF x PD x FC x RF	C58	CMF x PD x FC x RF x VN
C11	CMF x VN	C27	CMF x FC x VM	C43	CMF x PD x FC x VM	C59	CMF x PD x FC x VM x VN
C12	PD x FC	C28	CMF x FC x VN	C44	CMF x PD x FC x VN	C60	CMF x PD x RF x VM x VN
C13	PD x RF	C29	CMF x RF x VM	C45	CMF x PD x RF x VM	C61	CMF x FC x RF x VM x VN
C14	PD x VM	C30	CMF x RF x VN	C46	CMF x PD x RF x VN	C62	PD x FC x RF x VM x VN
C15	PD x VN	C31	CMF x VM x VN	C47	CMF x PD x VM x VN	C63	CMF x PD x FC x RF x VM x VN
C16	FC x RF	C32	PD x FC x RF	C48	CMF x FC x RF x VM		

Devido à natureza aleatória das abordagens evolutivas, cada cenário foi executado 30 vezes com o algoritmo evolutivo NSGA-II, e a média entre os melhores resultados foi utilizada para comparação de acordo com os parâmetros habitualmente citados na literatura técnica para algoritmos evolutivos.

Os dados que foram usados para a execução do algoritmo foram:

- Tamanho da população: 200;
- Número máximo de avaliações: 200.000;
- Taxa de cruzamento: 0,85;

- Taxa de mutação: 0,01.

A fórmula abaixo será utilizada na análise dos resultados.

$$f(x) = \frac{TCE}{TC}$$

Equação 6. Relação dos componentes com erro com o total de componentes.

Onde:

- *x* é a solução encontrada na curva de Pareto;
- *TCE* é o total de componentes com erro que tem na solução *x;*
- *TC* é o total de componentes que tem na solução x.

5.2.3. Formulação das Hipóteses

Uma premissa por trás de um programa que auxilia o testador na seleção de componentes para teste de unidade em aplicações móveis é que a lista de componentes seja melhor ou igual à seleção manual realizada por especialistas. Assim, foi definida a seguinte hipótese nula para este estudo:

- **Hipótese nula (H0):** Não há diferença entre os resultados obtidos usando o plugin SCoTUAM e a seleção manual realizada por especialistas em relação à eficácia na seleção de componentes com erro.
- **Hipótese Alternativa (H1):** Há diferença entre os resultados obtidos usando o plugin SCoTUAM e a seleção manual realizada por especialistas em relação à eficácia na seleção de componentes com erro.

5.2.4. Seleção de Participantes

Para participar desse estudo, foram convidados profissionais da indústria e academia com experiência em teste de unidade em aplicações móveis na plataforma Android.

A base de desenvolvedores/pesquisadores foi obtida a partir de uma busca na rede social *Linkedin*[15], como também na comunidade Android Dev BR[16]. Para participar do estudo, os profissionais tiveram que manifestar interesse em participar do estudo, concordando com o Termo de Consentimento Livre e Esclarecido (TCLE) e preenchendo um formulário de caracterização. Isso foi com o objetivo de ter o conhecimento do grau de experiência de cada profissional e assim direcionar a seleção para a seguinte fase do experimento. Os instrumentos estão disponíveis em https://goo.gl/71TvEQ.

O formulário de caracterização foi enviado para mais de 100 profissionais. Desse total, 31 participantes passaram para próxima fase de seleção de grupos. Os 31 participantes foram

[15] http://linkedin.com

[16] http://androiddevbr.slack.com

divididos em dois grupos, Grupo A e Grupo B, levando em consideração as informações que foram preenchidas no formulário de caracterização, sobre a experiência em desenvolvimento e o tempo de experiência em teste de unidade em aplicações Android.

Do total de participantes, somente 7 concluíram o estudo. Destes, 2 deles têm 2 anos de experiência (29%); 2 deles têm 3 anos de experiência (29%); 3 deles tem 1, 4 e 5 anos de experiência, respectivamente, e cada um representa (14%).

5.2.5. Materiais

Devido ao experimento envolver a seleção de componentes de forma manual e automatizada, foi necessário o uso dos seguintes instrumentos:

- **Aplicações Móveis**: A seleção das aplicações está baseada em três parâmetros, sendo que duas características foram baseadas em Fazzini *et al.* (2017), que são a diversidade (categorias diferentes de aplicações) e autocontenção (sem maiores configurações a serem implementadas), e o outro parâmetro é possuir scripts de teste escritos utilizando o framework Espresso[17]. Para cada aplicação, foi criado um documento descrevendo suas principais funcionalidades e características. Na Tabela 20 é mostrada a descrição de algumas características das aplicações móveis. A informação mais detalhada sobre essas aplicações está disponível em https://goo.gl/71TvEQ.
  - **ID** - um número para identificar a aplicação;
  - **Aplicação** - nome da aplicação;
  - **Código (LOC)** - quantidade de linhas de código da aplicação;
  - **Componentes** - quantidade de componentes que os testes chamaram **/** quantidade total de componentes da aplicação;
  - **Testes** - quantidade de testes de GUI da aplicação;
  - **Categoria** - categoria a qual a aplicação pertence.

Tabela 20. Descrição das aplicações.

ID	Aplicação	Código (LOC)	Componentes	Testes	Categoria
1	Budget Watch (BW)	946	36**/**141	15	Finanças
2	Recurrence (R)	2.4k	27**/**194	15	Produtividade

- **Dispositivos**: Para este estudo, os dispositivos móveis foram selecionados com base nas versões de SDK Android mais usadas por aplicações publicadas na Google Play Store[18]

[17]https://google.github.io/android-testing-support-library/docs/espresso/index.html

[18] https://developer.android.com/about/dashboards/index.html

(dados atualizados em setembro de 2017), no tamanho e resolução da tela. Na Tabela 21 é possível ver a porcentagem de mercado de algumas das versões de API mais usadas e a quantidade de dispositivos que serão utilizados de cada versão de API. Com base nessa informação e de acordo com o estudo de Vilkomir *et al.* (2015) que avaliou a eficácia de revelação de falhas utilizando várias combinações com quantidade diferentes de dispositivos e encontrou o resultado que 13 dispositivos é uma quantidade boa para a revelação de falhas, então definimos os 13 dispositivos que serão utilizados no experimento, listados na Tabela 21.

Tabela 21. Dispositivos móveis usados no experimento.

Versão	Nome da Versão	API	Porcen-tagem	Modelo	Tamanho	Densi-dade
4.4.2	KitKat	19	15,38%	BLU	480x800	230
				LG E977	768x1280	320
5.0	Lollipop	21	7,69%	Galaxy Note 3 LTE	1080x1920	480
5.1		22	23,07%	Alcatel PIXI4 (4.0)	320x480	160
				Alcatel PIXI4 (6.0)	1280X720	320
				Galaxy J1 Mini	480x800	240
6.0	Marshmallow	23	38,46%	ASUS Zenfone Go LTE	720x1280	320
				Galaxy A9	1080x1920	420
				LG X Power	720x1280	320
				Moto E	960x540	240
				Moto Z Power Edition	1440x2560	640
7.0	Nougat	24	15,38%	Galaxy A5	720x1280	294
				Moto G 5	1080x1920	480

5.3. Execução do Estudo

O estudo de viabilidade foi conduzido de forma online[19] e executado em duas partes. Na primeira parte, os participantes foram divididos nos grupos A e B, depois foi enviada por e-mail uma descrição das características da aplicação para cada participante. O grupo A ficou com a aplicação *Recurrence* e o grupo B, com a aplicação *Budget Watch*. Para cada participante foi solicitado que eles realizassem a seleção de componentes para a escrita de teste de unidade e enviassem a lista de componentes selecionados por e-mail.

Na segunda parte, para o processo de execução do estudo, cada aplicação foi aberta pelo plugin SCoTUAM no Android Studio, foi selecionado randomicamente um valor entre 1 e 5 para cada script de teste, então os scripts de teste foram executados ao mesmo tempo em

[19] O estudo foi realizado com especialistas da indústria e academia (público alvo).

todos os dispositivos móveis usados no experimento. Posteriormente, iniciou-se a extração do valor das métricas para a aplicação. Finalmente, ainda no plugin SCoTUAM foram selecionados componentes para a escrita de teste de unidade, sendo que foram feitas 63 seleções distintas utilizando os 63 cenários para a seleção dos componentes. A lista de componentes selecionados de cada cenário foi comparada com a lista montada a partir da seleção realizada pelos especialistas.

5.4. Resultados do Estudo de Viabilidade

A seguir serão apresentados os resultados deste estudo para cada uma das aplicações, iniciando com a aplicação *Budget Watch* e depois o resultado para a aplicação *Recurrence*.

Primeiro será mostrada a lista de componentes com erro, em seguida a avaliação da eficácia da seleção manual realizada pelos especialistas, continuando com a avaliação da eficácia da seleção automatizada realizada pelo plugin SCoTUAM, por fim a comparação da seleção manual com a seleção automatizada.

5.4.1. Resultados Aplicação Budget Watch

A Tabela 22 mostra a lista de componentes que foram identificados com erro (CE) para a aplicação *Budget Watch*. Essa lista de componentes será usada para avaliar a seleção dos especialistas e do plugin SCoTUAM.

Tabela 22. Lista de componentes com erros da aplicação *Budget Watch*.

ID	Classe	ASSINATURA DO COMPONENTE
CE1	BudgetActivity	protected void onCreate(Bundle savedInstanceState)
CE2	BudgetActivity	public boolean onContextItemSelected(MenuItem item)
CE3	BudgetActivity	public void onResume()
CE4	BudgetActivity	protected void onDestroy()
CE5	BudgetViewActivity	public boolean onOptionsItemSelected(MenuItem item)
CE6	BudgetViewActivity	public void onResume()
CE7	BudgetViewActivity	protected void onDestroy()
CE8	BudgetViewActivity	protected void onCreate(Bundle savedInstanceState)
CE9	CsvDatabaseExporter	public void exportData(DBHelper db, OutputStreamWriter output)
CE10	DatabaseCleanupTask	protected Void doInBackground(Void... nothing)
CE11	MainActivity	protected void onCreate(Bundle savedInstanceState)
CE12	MainActivity	private void displayAboutDialog()

5.4.1.1. Resultado da Avaliação dos Especialistas

O primeiro especialista (E1) selecionou oito componentes usando o critério "*Métodos públicos, que são possíveis de ser testados, métodos que são usados dentro de*

funcionalidades", que representa 5,67% do total de componentes da aplicação. Desses, um deles (ID CE9) está na lista de doze componentes com erros conhecidos, apresentada na Tabela 22, representando 8,33%.

O segundo especialista (E2) selecionou dezesseis componentes usando o critério "*Classes que tratam dos métodos de ciclo de vida*", que representam 11,35% do total de componentes da aplicação. Desses, oito (Ids CE1, CE2, CE3, CE4, CE6, CE8, CE11 e CE12) estão na lista de doze componentes com erros conhecidos Tabela 22, representando 66,67%.

A Figura 14 mostra o gráfico que apresenta a porcentagem de cobertura de cada um dos dois especialistas, para os componentes que foram identificados com erros na aplicação *Budget Watch*.

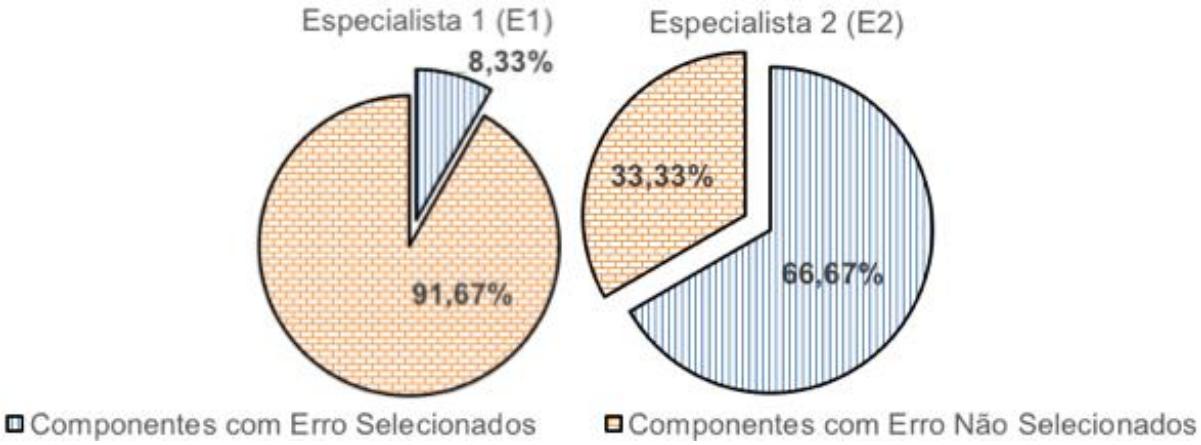


Figura 14. Cobertura de componentes com erros pelos especialistas para a aplicação *Budget Watch*.

5.4.1.2. Resultado da Avaliação do Plugin SCoTUAM

Para uma melhor compreensão da cobertura de componentes com erros pelo plugin SCoTUAM, a Figura 15 apresenta a quantidade de componentes com erro (apresentados na Tabela 22) que poderia ser selecionado por cada métrica utilizada nos 63 cenários.

Os dois componentes (IDs CE1 e CE3 da Tabela 22) foram executados durante a execução dos casos de teste. Portanto, eles têm uma maior probabilidade de serem selecionados nos cenários que utilizam as métricas VN, VM, RF e FC. Os quatro componentes com IDs CE2, CE6, CE9 e CE12 da Tabela 22 têm uma maior probabilidade de serem selecionados nos cenários que utiliza a métrica PD, pois foi identificado cheiros de código nesses componentes. Finalmente, todos os componentes da aplicação, incluindo os doze componentes com erros da Tabela 22, podem ser selecionados nos cenários que utilizam a métrica CMF, pois todos os componentes possuem esta métrica.

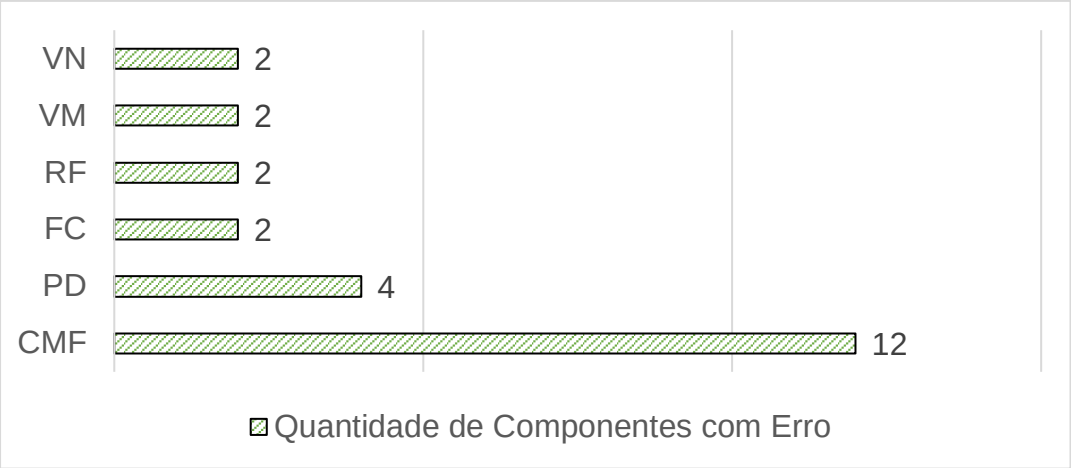


Figura 15. Quantidade de componentes com erro que que poderia ser selecionado por cada métrica para a aplicação *Budget Watch.*

5.4.1.3. Componentes com Erro

A Tabela 23 apresenta a cobertura de componentes com erro de cada um dos 63 cenários para a aplicação *Budget Watch*, onde observa-se que as soluções de cada um dos cenários conseguiram no máximo encontrar 10 componentes (83,33%) com erro. Vale ressaltar que cada um dos cenários obteve uma quantidade diferente de soluções na fronteira de Pareto. A fronteira de Pareto foi gerada a partir das 30 execuções do algoritmo genético para cada um dos cenários, e na Tabela 23 estão apenas as soluções da fronteira de Pareto. As características consideradas na Tabela 23 são as seguintes:

- **C01 a C63** são os cenários;
- **0 a 12** é a quantidade de componentes com erro;
- **Quantidade** é o total de soluções que encontraram um número específico de componentes com erro, citado no item acima;
- **Maior f(x)** refere-se à solução que teve o maior valor para a Equação 6;
- **Menor f(x)** refere-se à solução que teve o menor valor para a Equação 6.

Tabela 23. Cobertura de componentes com erro de cada um dos 63 cenários para a aplicação *Budget Watch*.

		Componentes com Erro												
		0	1	2	3	4	5	6	7	8	9	10	11	12
C1	Quantidade	489	619	500	578	282	198	570	264	69	7	0	0	0
	Maior f(x)	0,00%	**100,00%**	**100,00%**	**75,00%**	**66,67%**	**62,50%**	**46,15%**	**53,85%**	**34,78%**	25,71%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	12,50%	20,00%	20,00%	25,00%	25,00%	13,95%	14,58%	15,69%	17,65%	0,00%	0,00%	0,00%
C2	Quantidade	28	18	2	5	6	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	20,00%	10,53%	14,29%	17,39%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	10,00%	12,50%	15,38%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C3	Quantidade	6	31	19	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	25,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	7,14%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C4	Quantidade	8	41	4	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	7,14%	12,50%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C5	Quantidade	7	19	2	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	13,33%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	7,14%	12,50%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C6	Quantidade	3	34	28	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	40,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	7,14%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C7	Quantidade	112	208	170	615	653	387	251	298	48	5	0	0	0
	Maior f(x)	0,00%	**100,00%**	15,38%	16,67%	20,00%	20,83%	23,08%	25,00%	25,81%	25,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	8,33%	10,71%	12,90%	14,71%	16,67%	14,29%	16,67%	20,93%	0,00%	0,00%	0,00%
C8	Quantidade	16	379	974	538	373	204	143	344	85	12	1	0	0
	Maior f(x)	0,00%	**100,00%**	40,00%	30,00%	23,53%	25,00%	26,09%	26,92%	27,59%	27,27%	22,22%	0,00%	0,00%
	Menor f(x)	0,00%	5,88%	8,70%	12,00%	14,29%	16,13%	17,14%	14,89%	15,09%	18,75%	22,22%	0,00%	0,00%
C9	Quantidade	15	169	506	683	692	381	183	412	161	23	4	0	0
	Maior f(x)	0,00%	50,00%	16,67%	20,00%	23,53%	26,32%	28,57%	28,00%	27,59%	29,03%	23,26%	0,00%	0,00%
	Menor f(x)	0,00%	5,56%	9,52%	13,04%	14,81%	17,24%	18,75%	14,58%	15,38%	16,07%	21,28%	0,00%	0,00%
C10	Quantidade	34	357	489	606	684	414	368	588	195	31	3	0	0
	Maior f(x)	0,00%	50,00%	50,00%	20,00%	22,22%	26,32%	27,27%	29,17%	29,63%	26,47%	23,26%	0,00%	0,00%

	Menor f(x)	0,00%	5,26%	9,09%	13,04%	15,38%	16,67%	17,14%	14,00%	16,00%	16,67%	20,00%	0,00%	0,00%
C11	Quantidade	8	149	934	609	560	252	150	350	77	18	0	0	0
	Maior f(x)	0,00%	**100,00%**	40,00%	37,50%	21,05%	25,00%	26,09%	26,92%	27,59%	28,13%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	6,25%	8,70%	12,50%	13,79%	16,67%	18,75%	14,00%	16,33%	18,75%	0,00%	0,00%	0,00%
C12	Quantidade	42	108	424	92	188	185	63	1	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	20,00%	18,75%	16,67%	18,52%	19,35%	20,59%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	7,14%	9,68%	11,11%	12,82%	15,00%	20,59%	0,00%	0,00%	0,00%	0,00%	0,00%
C13	Quantidade	8	79	22	11	11	5	8	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	10,53%	10,34%	12,50%	15,15%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,76%	6,25%	9,09%	11,43%	13,51%	15,38%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C14	Quantidade	7	68	20	19	6	5	6	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	9,09%	11,11%	12,90%	14,29%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,76%	6,45%	9,09%	11,43%	13,16%	15,38%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C15	Quantidade	12	107	256	51	58	87	87	1	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	15,38%	12,50%	14,81%	17,86%	19,35%	20,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,55%	6,67%	9,68%	12,12%	13,16%	15,00%	20,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C16	Quantidade	8	54	8	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	6,67%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C17	Quantidade	6	35	8	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	6,67%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C18	Quantidade	6	32	26	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	40,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	7,14%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C19	Quantidade	8	39	3	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	7,14%	12,50%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C20	Quantidade	11	51	10	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	6,67%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C21	Quantidade	6	33	8	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

	Menor f(x)	0,00%	6,67%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C22	Quantidade	92	238	641	199	770	790	537	184	137	30	6	0	0
	Maior f(x)	0,00%	**100,00%**	33,33%	20,00%	17,39%	18,52%	20,69%	21,88%	19,05%	20,45%	21,28%	0,00%	0,00%
	Menor f(x)	0,00%	4,76%	6,90%	9,09%	10,26%	11,90%	12,24%	13,73%	14,55%	15,79%	17,86%	0,00%	0,00%
C23	Quantidade	18	224	119	162	209	213	391	192	307	18	0	0	0
	Maior f(x)	0,00%	50,00%	50,00%	10,71%	13,33%	15,15%	17,14%	18,42%	20,00%	20,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,00%	**5,88%**	8,11%	10,53%	12,50%	12,77%	13,73%	14,04%	16,07%	0,00%	0,00%	0,00%
C24	Quantidade	29	458	294	121	179	243	477	191	328	43	2	0	0
	Maior f(x)	0,00%	50,00%	40,00%	17,65%	14,81%	16,13%	17,65%	18,92%	20,00%	21,43%	17,54%	0,00%	0,00%
	Menor f(x)	0,00%	3,45%	6,25%	8,11%	10,00%	12,20%	13,04%	14,29%	13,33%	15,25%	17,54%	0,00%	0,00%
C25	Quantidade	14	323	562	160	207	614	817	286	204	43	2	0	0
	Maior f(x)	0,00%	**100,00%**	50,00%	16,67%	15,38%	18,52%	20,00%	21,21%	19,51%	20,45%	18,87%	0,00%	0,00%
	Menor f(x)	0,00%	4,76%	6,45%	9,38%	11,11%	12,50%	12,77%	14,29%	14,04%	16,07%	18,87%	0,00%	0,00%
C26	Quantidade	25	153	491	672	613	348	163	365	101	11	0	0	0
	Maior f(x)	0,00%	50,00%	18,18%	17,65%	22,22%	25,00%	25,00%	25,93%	27,59%	28,13%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,00%	9,09%	12,50%	12,90%	16,67%	16,67%	13,73%	15,69%	18,00%	0,00%	0,00%	0,00%
C27	Quantidade	16	243	439	584	639	413	326	604	161	20	1	0	0
	Maior f(x)	0,00%	**100,00%**	50,00%	18,75%	21,05%	25,00%	26,09%	28,00%	29,63%	27,27%	20,83%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	8,70%	12,50%	13,79%	17,24%	17,14%	13,73%	14,81%	16,98%	20,83%	0,00%	0,00%
C28	Quantidade	11	135	840	562	572	223	159	340	80	13	1	0	0
	Maior f(x)	0,00%	**100,00%**	40,00%	21,43%	21,05%	23,81%	28,57%	28,00%	27,59%	29,03%	20,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,88%	8,70%	12,50%	13,79%	16,67%	17,65%	16,28%	16,33%	18,00%	20,00%	0,00%	0,00%
C29	Quantidade	13	161	419	597	657	413	337	523	159	9	2	0	0
	Maior f(x)	0,00%	**100,00%**	14,29%	18,75%	23,53%	26,32%	28,57%	29,17%	29,63%	25,00%	25,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,56%	9,09%	13,04%	14,81%	17,24%	17,65%	14,89%	15,09%	19,57%	24,39%	0,00%	0,00%
C30	Quantidade	19	198	570	724	728	386	187	393	124	30	0	0	0
	Maior f(x)	0,00%	**100,00%**	50,00%	18,75%	22,22%	25,00%	26,09%	28,00%	26,67%	29,03%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	8,70%	11,54%	14,29%	15,63%	16,67%	14,58%	15,38%	16,98%	0,00%	0,00%	0,00%
C31	Quantidade	16	240	457	644	675	455	305	556	158	11	0	0	0
	Maior f(x)	0,00%	**100,00%**	50,00%	18,75%	22,22%	23,81%	28,57%	28,00%	28,57%	28,13%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,00%	8,33%	12,00%	14,29%	17,24%	18,75%	13,46%	14,04%	17,65%	0,00%	0,00%	0,00%
C32	Quantidade	9	91	43	20	10	8	15	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	9,09%	10,00%	12,50%	14,29%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

	Menor f(x)	0,00%	4,55%	6,06%	8,82%	11,43%	13,16%	15,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C33	Quantidade	7	97	32	39	8	9	14	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	9,09%	12,00%	12,50%	14,29%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,55%	6,45%	8,82%	11,11%	13,16%	15,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C34	Quantidade	8	118	294	78	74	90	69	3	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	16,67%	15,00%	15,38%	17,86%	18,18%	20,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,55%	6,45%	9,68%	11,76%	12,82%	15,00%	18,42%	0,00%	0,00%	0,00%	0,00%	0,00%
C35	Quantidade	6	65	14	34	5	7	9	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	9,52%	13,04%	12,90%	15,15%	17,14%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,76%	7,14%	9,09%	11,76%	13,51%	15,38%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C36	Quantidade	11	102	22	33	14	9	13	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	9,52%	12,00%	12,90%	14,29%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,55%	6,67%	8,82%	11,43%	13,16%	15,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C37	Quantidade	8	95	23	56	9	13	15	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	9,52%	13,04%	12,50%	14,71%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,55%	7,41%	8,82%	11,43%	13,16%	15,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C38	Quantidade	9	50	8	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	6,67%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C39	Quantidade	4	53	10	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	6,67%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C40	Quantidade	3	40	8	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	6,67%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C41	Quantidade	6	55	10	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	6,67%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C42	Quantidade	20	440	310	196	273	306	495	215	261	89	0	0	0
	Maior f(x)	0,00%	50,00%	40,00%	13,04%	12,90%	15,15%	17,14%	17,95%	19,05%	20,45%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	**3,23%**	**5,88%**	**7,89%**	10,00%	11,90%	12,50%	14,00%	14,29%	15,52%	0,00%	0,00%	0,00%
C43	Quantidade	17	514	277	273	215	342	562	264	315	67	4	0	0
	Maior f(x)	0,00%	50,00%	28,57%	12,50%	13,33%	15,63%	17,14%	19,44%	19,51%	20,93%	19,23%	0,00%	0,00%

	Menor f(x)	0,00%	3,70%	6,06%	8,57%	10,00%	11,36%	12,24%	14,00%	14,04%	15,00%	17,24%	0,00%	0,00%
C44	Quantidade	33	317	498	166	176	457	693	265	269	54	2	0	0
	Maior f(x)	0,00%	**100,00%**	40,00%	14,29%	14,81%	17,86%	19,35%	20,59%	19,51%	20,93%	21,74%	0,00%	0,00%
	Menor f(x)	0,00%	4,00%	6,45%	9,38%	11,11%	12,50%	12,77%	14,00%	14,29%	16,07%	21,28%	0,00%	0,00%
C45	Quantidade	16	436	307	414	233	316	538	255	383	56	3	0	0
	Maior f(x)	0,00%	50,00%	16,67%	14,29%	16,67%	16,13%	18,18%	19,44%	20,00%	20,93%	19,23%	0,00%	0,00%
	Menor f(x)	0,00%	3,70%	6,67%	**7,89%**	10,53%	11,90%	13,04%	13,73%	14,29%	15,25%	18,52%	0,00%	0,00%
C46	Quantidade	19	452	286	242	302	351	574	241	318	56	2	0	0
	Maior f(x)	0,00%	**100,00%**	40,00%	12,50%	13,79%	15,15%	17,14%	18,42%	19,05%	20,00%	21,74%	0,00%	0,00%
	Menor f(x)	0,00%	3,57%	6,06%	8,11%	**9,76%**	11,63%	12,77%	14,00%	13,33%	**14,75%**	18,18%	0,00%	0,00%
C47	Quantidade	17	458	276	413	259	351	596	238	339	52	1	0	0
	Maior f(x)	0,00%	**100,00%**	50,00%	14,29%	15,38%	15,63%	17,14%	18,92%	19,51%	20,45%	20,00%	0,00%	0,00%
	Menor f(x)	0,00%	3,70%	6,45%	8,33%	10,26%	12,20%	12,77%	14,00%	**13,11%**	15,25%	20,00%	0,00%	0,00%
C48	Quantidade	18	199	412	617	633	393	291	532	109	27	1	0	0
	Maior f(x)	0,00%	50,00%	50,00%	17,65%	22,22%	25,00%	27,27%	28,00%	29,63%	25,71%	22,73%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	8,33%	12,00%	14,29%	16,67%	18,18%	14,58%	15,38%	15,52%	22,73%	0,00%	0,00%
C49	Quantidade	13	175	545	653	657	342	184	377	85	13	0	0	0
	Maior f(x)	0,00%	**100,00%**	40,00%	20,00%	22,22%	25,00%	26,09%	28,00%	27,59%	23,08%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	8,00%	12,00%	13,79%	16,67%	17,14%	13,73%	16,00%	17,31%	0,00%	0,00%	0,00%
C50	Quantidade	21	242	383	550	652	364	305	575	134	12	0	0	0
	Maior f(x)	0,00%	50,00%	28,57%	20,00%	22,22%	25,00%	27,27%	26,92%	27,59%	**30,00%**	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	8,70%	12,00%	14,29%	17,24%	18,18%	**12,96%**	14,81%	17,65%	0,00%	0,00%	0,00%
C51	Quantidade	14	201	460	690	749	444	303	533	121	12	8	0	0
	Maior f(x)	0,00%	**100,00%**	40,00%	18,75%	22,22%	25,00%	27,27%	29,17%	28,57%	21,43%	**26,32%**	0,00%	0,00%
	Menor f(x)	0,00%	5,00%	8,70%	12,00%	13,33%	16,67%	18,18%	13,73%	16,33%	18,37%	17,54%	0,00%	0,00%
C52	Quantidade	12	92	30	50	5	8	16	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	9,52%	13,04%	12,12%	14,29%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,55%	7,14%	8,82%	11,43%	13,16%	15,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C53	Quantidade	4	111	20	37	8	7	12	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	9,09%	13,04%	12,50%	14,29%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,55%	6,90%	8,82%	11,43%	12,82%	15,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C54	Quantidade	3	87	25	52	8	11	12	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	10,00%	13,04%	12,50%	14,29%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

	Menor f(x)	0,00%	4,55%	7,41%	8,82%	11,11%	13,16%	15,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C55	Quantidade	9	96	27	41	7	8	12	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	11,76%	13,04%	12,12%	14,29%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,55%	8,00%	8,82%	11,43%	13,16%	15,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C56	Quantidade	4	47	10	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	6,67%	11,76%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C57	Quantidade	16	404	276	442	261	373	616	275	357	81	7	0	0
	Maior f(x)	0,00%	**100,00%**	18,18%	14,29%	17,39%	15,63%	17,14%	19,44%	19,05%	20,45%	21,28%	0,00%	0,00%
	Menor f(x)	0,00%	3,70%	6,67%	8,33%	10,00%	11,90%	12,77%	13,73%	13,56%	15,00%	18,52%	0,00%	0,00%
C58	Quantidade	20	427	243	329	345	359	506	239	321	76	3	0	0
	Maior f(x)	0,00%	**100,00%**	18,18%	13,04%	14,81%	15,63%	16,67%	18,42%	20,00%	20,93%	19,23%	0,00%	0,00%
	Menor f(x)	0,00%	3,57%	6,45%	8,33%	10,26%	11,90%	12,50%	14,00%	14,04%	**14,75%**	17,54%	0,00%	0,00%
C59	Quantidade	18	453	236	446	246	360	564	259	353	87	6	0	0
	Maior f(x)	0,00%	**100,00%**	28,57%	15,79%	17,39%	15,63%	17,65%	18,42%	19,05%	20,45%	20,00%	0,00%	0,00%
	Menor f(x)	0,00%	3,85%	6,45%	**7,89%**	10,26%	11,90%	12,50%	14,58%	13,56%	15,25%	17,24%	0,00%	0,00%
C60	Quantidade	17	398	326	482	278	354	592	252	351	92	0	0	0
	Maior f(x)	0,00%	**100,00%**	25,00%	15,79%	19,05%	15,63%	17,65%	18,92%	19,05%	20,93%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,00%	6,45%	**7,89%**	10,26%	11,90%	13,33%	14,00%	13,56%	15,00%	0,00%	0,00%	0,00%
C61	Quantidade	19	194	443	634	705	394	311	491	153	22	0	0	0
	Maior f(x)	0,00%	**100,00%**	28,57%	20,00%	22,22%	25,00%	27,27%	29,17%	29,63%	29,03%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	8,70%	11,54%	13,33%	16,13%	17,65%	15,22%	15,09%	17,31%	0,00%	0,00%	0,00%
C62	Quantidade	8	107	32	42	8	11	12	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	12,50%	13,04%	12,12%	14,29%	16,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	4,55%	8,00%	8,82%	11,11%	12,82%	15,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C63	Quantidade	17	385	323	492	262	374	624	270	382	50	6	0	0
	Maior f(x)	0,00%	**100,00%**	18,18%	15,00%	16,67%	15,63%	17,14%	18,92%	19,51%	20,45%	18,87%	0,00%	0,00%
	Menor f(x)	0,00%	4,00%	6,25%	8,11%	10,00%	**11,63%**	**12,00%**	14,00%	13,56%	15,00%	**15,63%**	0,00%	0,00%

Ainda analisando a Tabela 23, pode-se verificar que o cenário 1 (C01) obteve a solução com o maior valor para a relação quantidade de componentes com erro divido pela quantidade total de componentes (Equação 6), quando encontrou de 1 a 8 componentes com erro, isso se deve ao fato de todos os componentes terem um custo de manutenção futura (CMF).

O cenário C02 que usa a métrica propensão a defeito (PD) selecionou quatro componentes com erro, sendo que esses componentes tinham cheiros de código identificados (Figura 15). Os quatro cenários C03, C04, C05 e C06 que usam as métricas frequência de chamadas (FC), risco de falha (RF), vulnerabilidade de mercado (VM) e valor de negócio (VN), respectivamente, selecionaram dois componentes com erro, os quais foram executados pelos casos de teste (Figura 15).

Com relação ao melhor caso, podem ser citados os cenários C51 e C50, pois esses foram o melhor e segundo melhor cenário para a Equação 6, respectivamente. A melhor solução para o cenário C50 obteve 21,27% do total de componentes da aplicação, sendo que com essa quantidade de componentes essa solução cobriu 75% dos componentes com erro identificados e ficou com 70% de componentes a mais. Para o cenário C51, a melhor solução ficou com 26,95% do total de componentes da aplicação, cobrindo 83,33% dos componentes com erro identificados e com 73,68% de componentes a mais. Isso indica que no melhor caso o SCoTUAM está alcançando uma boa cobertura dos componentes com erro, no entanto, precisa de um refinamento para diminuir a quantidade de componentes a mais que compõem a solução.

No que se refere ao pior caso, pode ser citado o cenário C42, pois ele teve o menor valor para a Equação 6. A pior solução para o cenário C42 obteve 21,98% do total de componentes da aplicação, sendo que com essa quantidade de componentes essa solução cobriu 8,33% dos componentes com erro identificados e ficou com 96,77% de componentes a mais.

5.4.1.4. Componentes com Erro Geral

A Figura 16 apresenta a porcentagem de soluções para cada quantidade de componentes com erro identificados nas soluções da fronteira de Pareto. Onde observa-se que a maioria das soluções da fronteira de Pareto identificou três componentes com erro e apenas 1,43% das soluções identificou zero componentes com erro. Isso indica que o SCoTUAM possui uma boa cobertura de erros.

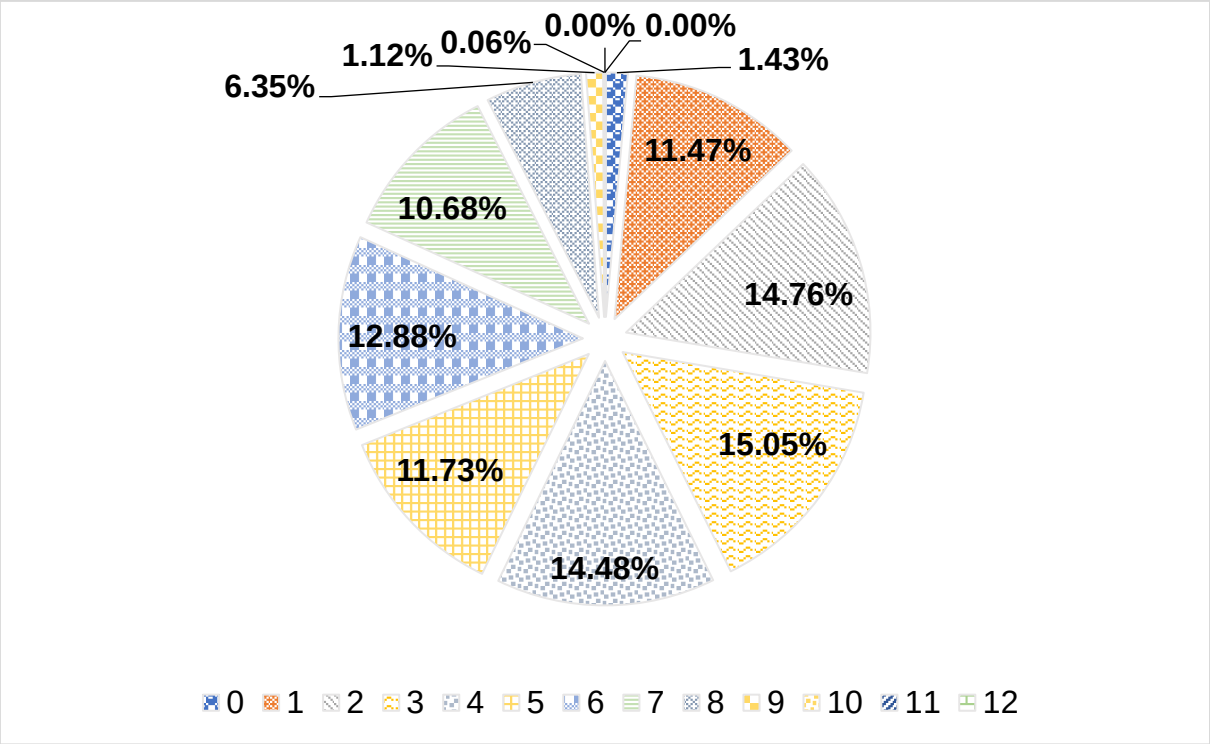


Figura 16. Porcentagem de soluções do SCoTUAM para a aplicação *Budget Watch*.

No entanto, a Figura 17 mostra que 84,61% dos componentes nas soluções não continham erros. Por este motivo, percebe-se a necessidade de um refinamento do plugin para diminuir a quantidade de componentes que não contém erros (falsos positivos), direcionando a componentes que possuem erros reais, em aplicações reais.

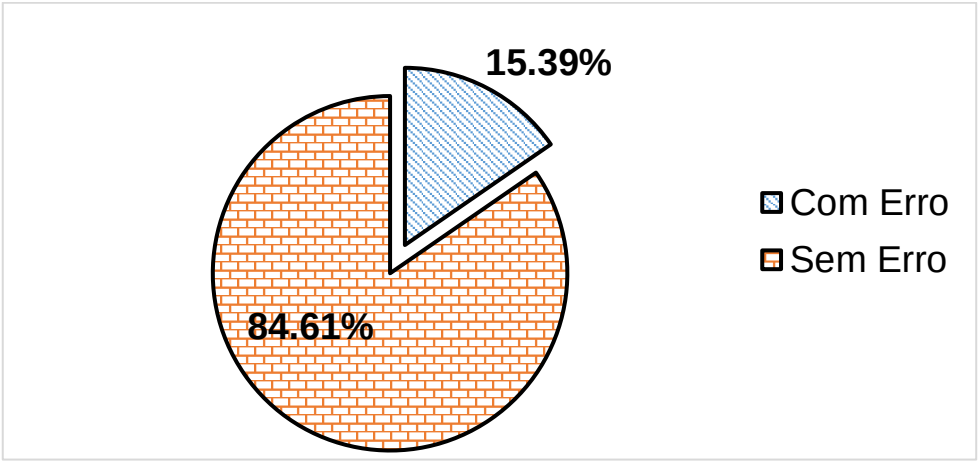


Figura 17. Porcentagem de componentes com e sem erro das soluções do SCoTUAM para a aplicação *Budget Watch*.

5.4.1.5. Métricas e Componentes com Erros

No que diz respeito às métricas que compõem os cenários de priorização da seleção de componentes, a Tabela 24 apresenta a quantidade de componentes com erro (CE) e sem erro (SE) para cada uma das métricas. A métrica custo de manutenção futura tem a maior quantidade de componentes com erro e sem erro. Isso deve-se ao fato de todos os

componentes terem um valor para essa métrica, fazendo com que os cenários que utilizam essa métrica possuam mais soluções e consequentemente mais componentes. Observa-se ainda que a relação entre componentes com erro e sem erro foi similar a todas as métricas, ficando entre 0,15 e 0,18.

Tabela 24. Métricas e componentes com erro para a aplicação *Budget Watch*.

MÉTRICA	CE	SE	CE / SE
Custo de Manutenção Futura	422511	2296622	0,18
Vulnerabilidade de Mercado	230023	1263927	0,18
Valor de Negócio	224797	1265740	0,17
Code Smell	220903	1398426	0,15
Frequência de Chamadas	220808	1252222	0,17
Risco de Falha	216164	1206150	0,18

5.4.1.6. Comparação entre os Especialistas e o Plugin SCoTUAM

Para a comparação da seleção automatizada utilizando o plugin SCoTUAM com a seleção manual realizada por especialistas, foi utilizado o resultado do especialista E2, pois entre os dois especialistas ele teve uma maior cobertura (8; 66,67%) dos componentes com erros. Para o SCoTUAM, foram selecionados os cenários C01, C50 e C51, pois eles tiveram o melhor resultado entre os 63 cenários.

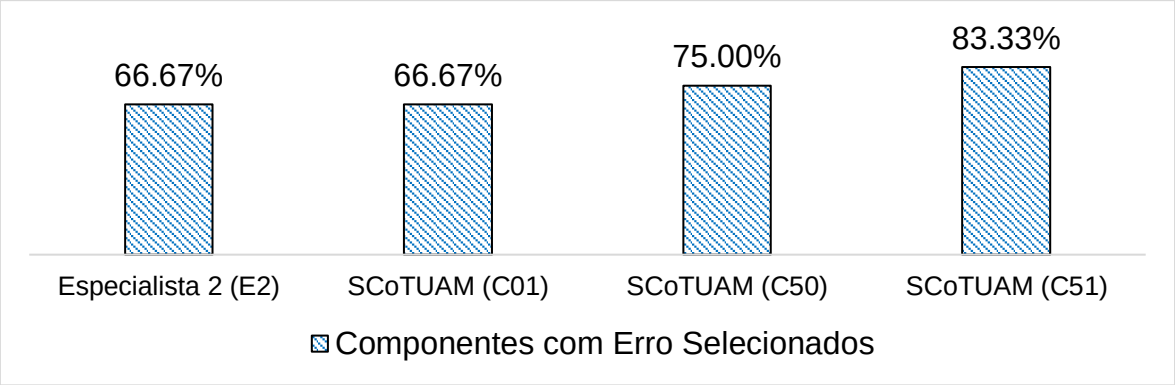


Figura 18. Comparação entre especialistas e SCoTUAM para a aplicação *Budget Watch*.

A Figura 18 apresenta uma comparação entre a porcentagem de componentes com erro selecionados pelo especialista E2 e pelos três melhores cenários (C01, C50 e C51) do plugin SCoTUAM. Observa-se que o especialista E2 obteve uma menor cobertura dos componentes com erro comparando com os cenários C50 e C51 de SCoTUAM. Embora a quantidade de componentes com erro selecionados pelo SCoTUAM seja maior que a do especialista E2, observou-se que as soluções do SCoTUAM trouxeram uma quantidade maior de componentes que não foram identificados com erro, conforme mostrado na Tabela 25. Isso indica que o plugin SCoTUAM precisa melhorar nesse aspecto, de acordo com o resultado dessa aplicação.

Tabela 25. Resultado da Equação 6 para a aplicação *Budget Watch*.

	Equação 6 (fx)
Especialista 2 (E2)	50,00%
SCoTUAM (C01)	34,78%
SCoTUAM (C50)	30,00%
SCoTUAM (C51)	26,32%

5.4.2. Resultados Aplicação *Recurrence*

Na Tabela 26 é apresentada a lista de 15 componentes que foram identificados com erro (CE) para a aplicação *Recurrence*. Essa lista de componentes será usada para avaliar a seleção dos especialistas e do plugin SCoTUAM.

Tabela 26. Lista de componentes com erros da aplicação *Recurrence*.

ID	CLASSE	ASSINATURA DO COMPONENTE
CE1	AboutActivity	void launchAppURL()
CE2	AboutActivity	void launchEmail()
CE3	AboutActivity	void showContributorsDialog(View view)
CE4	AboutActivity	void showLibrariesDialog()
CE5	AlarmReceiver	public void onReceive(Context context, Intent intent)
CE6	AlarmUtil	public static void setNextAlarm(Context context, Reminder reminder, DatabaseHelper database)
CE7	CreateEditActivity	void colourSelector()
CE8	CreateEditActivity	void datePicker(View view)
CE9	CreateEditActivity	void iconSelector()
CE10	CreateEditActivity	void repeatSelector()
CE11	CreateEditActivity	void switchClicked()
CE12	CreateEditActivity	void timePicker()
CE13	CreateEditActivity	void toggleSwitch()
CE14	MainActivity	void fabClicked()
CE15	ViewActivity	public void actionMarkAsDone()

5.4.2.1. Resultado da Avaliação dos Especialistas

O primeiro especialista (E1) selecionou 24 componentes usando o critério "componentes representativos para o negócio, sem dependências do Android framework", que representam 12,37% do total de componentes da aplicação. Desses, dois deles (componentes com ids CE5 e CE6) estão na lista de quinze componentes com erros conhecidos, apresentada na Tabela 26, representando 13,33%.

O segundo especialista (E2) selecionou 37 componentes usando o critério "componentes de classes mais complexas, de classes onde erros bobos costumam acontecer", que representam 19,07% do total de componentes da aplicação. Desses, um (componente com id CE6) está na lista de componentes com erros conhecidos da Tabela 26, representando 6,66%.

O terceiro especialista (E3) selecionou 21 componentes usando o critério “principais funcionalidades: 1 - criar lembrete; 2 - visualizar lembrete; 3 - CRUD Database”, que representam 10,82% do total de componentes da aplicação. Desses, um (componente com id CE15) está na lista de componentes com erro, representando 6,66%.

O quarto especialista (E4) selecionou 33 componentes usando o critério “componentes públicos de classes de negócio e utilitários”, que representam 17,01% do total de componentes. Desses, dois (componentes com ids CE5 e CE6) estão na lista de componentes com erros, representando 13,33%.

O quinto especialista (E5) selecionou 34 componentes usando o critério “componentes que são independentes do SDK do Android”, que representam 17,52% do total de componentes. Desses, nenhum componente está na lista de componentes com erro.

Na Figura 19 é apresentado o gráfico que mostra a porcentagem de cobertura de cada um dos cincos especialistas, para os componentes que foram identificados com erros na aplicação *Recurrence*.

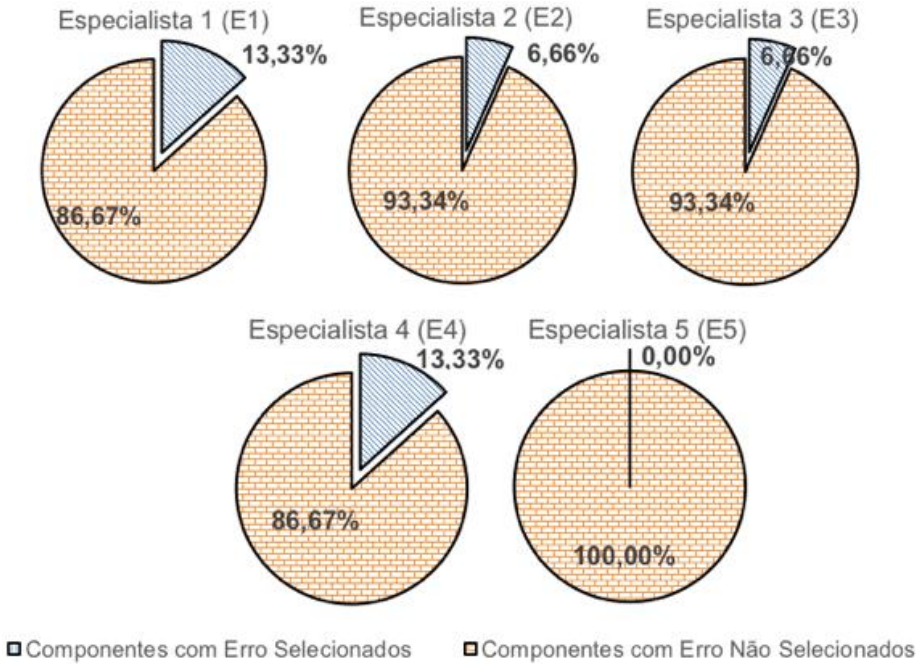


Figura 19. Cobertura de componentes com erros pelos especialistas para a aplicação *Recurrence*.

5.4.2.2. Resultado da Avaliação do Plugin SCoTUAM

A Figura 20 mostra a quantidade de componentes com erro (apresentados na Tabela 26) que poderiam ser selecionados por cada métrica utilizada nos 63 cenários. O componente com id CE14 da Tabela 26 foi executado durante a execução dos casos de teste. Portanto, ele tem uma maior probabilidade de ser selecionado nos cenários que utilizam as métricas VN, VM, RF e FC. O componente com id CE6 tem uma maior probabilidade de ser selecionado nos cenários que utilizam a métrica PD, pois foi identificado cheiros de código nesse componente. Por último, todos os componentes da aplicação, incluindo os quinze

componentes com erros da Tabela 8, podem ser selecionados nos cenários que utilizam a métrica CMF, pois todos os componentes possuem um custo de manutenção futura.

Figura 20. Quantidade de componentes com erro que que poderia ser selecionado por cada métrica para a aplicação *Recurrence*.

5.4.2.3. Componentes com Erro

A Tabela 27 mostra a cobertura de componentes com erro de cada um dos 63 cenários para a aplicação *Recurrence*, onde observa-se que as soluções de cada um dos cenários conseguiram no máximo encontrar 5 componentes (33,33%) com erro. Vale ressaltar que cada um dos cenários obteve uma quantidade diferente de soluções na fronteira de Pareto. A fronteira de Pareto foi gerada a partir das 30 execuções do algoritmo genético para cada um dos cenários e na Tabela 27 estão apenas as soluções da fronteira de Pareto. As características consideradas na Tabela 27 são as seguintes:

- **C01 a C63** são os cenários;
- **0 a 15** é a quantidade de componentes com erro;
- **Quantidade** é o total de soluções que encontraram um número específico de componentes com erro, citado no item acima;
- **Maior f(x)** refere-se à solução que teve o maior valor para a Equação 6;
- **Menor f(x)** refere-se à solução que teve o menor valor para a Equação 6.

Tabela 27. Cobertura de componentes com erro de cada um dos 63 cenários para a aplicação *Recurrence*.

		Componentes com Erro															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C01	Quantidade	1031	799	346	398	37	4	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	**66,67%**	**27,27%**	**15,38%**	**13,16%**	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	3,70%	5,26%	6,25%	8,70%	11,11%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C02	Quantidade	7	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	12,50%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C03	Quantidade	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C04	Quantidade	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C05	Quantidade	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C06	Quantidade	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C07	Quantidade	530	1370	342	384	39	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	33,33%	28,57%	17,65%	13,79%	**13,16%**	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	3,45%	5,13%	6,52%	8,33%	13,16%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C08	Quantidade	650	635	355	208	20	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	33,33%	12,50%	13,64%	12,90%	10,64%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,70%	4,65%	6,00%	7,84%	10,64%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C09	Quantidade	876	616	352	238	41	4	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	15,38%	13,64%	12,50%	11,90%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,78%	4,35%	6,25%	8,00%	9,62%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C10	Quantidade	958	649	338	325	35	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	14,29%	12,00%	11,11%	11,36%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

	Menor f(x)	0,00%	2,86%	4,26%	**5,77%**	8,33%	11,36%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C11	Quantidade	926	606	314	262	36	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	16,67%	13,04%	11,11%	10,64%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,94%	4,44%	6,25%	8,33%	10,64%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C12	Quantidade	107	58	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	14,29%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C13	Quantidade	54	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	6,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,56%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C14	Quantidade	25	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	5,88%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,56%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C15	Quantidade	81	26	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	7,14%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C16	Quantidade	33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C17	Quantidade	38	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C18	Quantidade	34	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C19	Quantidade	19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C20	Quantidade	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C21	Quantidade	31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C22	Quantidade	475	1060	336	148	19	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	25,00%	20,00%	25,00%	12,50%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,63%	4,55%	6,38%	8,51%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C23	Quantidade	418	1277	345	136	15	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	11,76%	10,00%	11,76%	12,82%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	**2,22%**	4,55%	6,12%	7,84%	12,82%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C24	Quantidade	503	1057	307	250	48	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	11,11%	12,50%	11,43%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,63%	4,65%	6,00%	8,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C25	Quantidade	453	1108	331	172	28	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	13,33%	11,11%	11,43%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,56%	4,26%	**5,77%**	8,16%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C26	Quantidade	990	750	358	280	21	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	33,33%	14,29%	11,11%	12,90%	11,90%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,86%	4,44%	6,00%	8,33%	11,90%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C27	Quantidade	981	681	355	297	30	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	16,67%	15,38%	14,29%	13,79%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,78%	4,65%	6,00%	8,51%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C28	Quantidade	816	582	316	292	25	3	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	16,67%	14,29%	10,71%	12,50%	**13,16%**	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,56%	4,00%	6,12%	8,16%	11,36%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C29	Quantidade	848	603	332	321	41	4	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	14,29%	10,71%	12,50%	**13,16%**	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,78%	4,44%	**5,77%**	7,84%	11,11%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C30	Quantidade	795	594	362	205	26	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	13,33%	10,00%	11,43%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,70%	4,17%	6,25%	8,16%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C31	Quantidade	835	628	349	280	45	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	10,00%	10,00%	11,43%	11,63%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,70%	4,44%	6,12%	7,84%	11,63%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C32	Quantidade	87	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	6,67%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C33	Quantidade	59	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	5,88%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C34	Quantidade	86	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C35	Quantidade	32	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	5,88%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,56%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C36	Quantidade	63	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	6,25%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C37	Quantidade	53	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	5,88%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C38	Quantidade	33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C39	Quantidade	32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C40	Quantidade	37	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C41	Quantidade	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C42	Quantidade	462	1251	337	123	19	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	33,33%	11,76%	13,64%	12,12%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,50%	4,17%	6,12%	8,16%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C43	Quantidade	545	1205	350	154	29	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	**100,00%**	11,11%	13,64%	10,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

	Menor f(x)	0,00%	2,63%	4,44%	5,88%	**7,55%**	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C44	Quantidade	439	1180	358	145	23	2	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	20,00%	12,50%	11,11%	11,11%	11,63%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,50%	**3,92%**	6,00%	7,84%	10,87%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C45	Quantidade	420	1156	359	197	16	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	11,11%	13,64%	11,43%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,50%	4,26%	6,12%	8,33%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C46	Quantidade	413	1277	400	148	32	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	11,76%	15,00%	11,11%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,56%	4,17%	5,88%	7,84%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C47	Quantidade	562	1284	379	136	26	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	11,76%	16,67%	11,11%	10,64%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,44%	4,17%	6,00%	**7,55%**	10,64%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C48	Quantidade	959	700	373	295	35	5	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	25,00%	50,00%	14,29%	10,81%	12,50%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,70%	4,65%	6,12%	8,16%	10,64%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C49	Quantidade	877	595	337	301	28	2	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	20,00%	13,33%	12,00%	11,11%	9,62%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,86%	4,44%	6,12%	8,00%	**9,26%**	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C50	Quantidade	927	616	336	358	37	4	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	16,67%	14,29%	10,71%	12,50%	12,20%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,70%	4,55%	**5,77%**	7,84%	10,42%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C51	Quantidade	966	649	349	321	25	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	33,33%	13,33%	10,34%	11,76%	11,36%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,56%	4,35%	6,00%	8,16%	11,36%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C52	Quantidade	47	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	5,88%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C53	Quantidade	61	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	6,25%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C54	Quantidade	53	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	5,88%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C55	Quantidade	55	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	5,88%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C56	Quantidade	33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C57	Quantidade	473	1322	394	143	18	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	20,00%	11,76%	11,54%	10,81%	9,80%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,56%	4,55%	6,38%	8,51%	9,80%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C58	Quantidade	440	1249	423	178	28	2	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	20,00%	11,76%	15,00%	12,50%	**13,16%**	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,56%	4,08%	6,00%	7,84%	11,11%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C59	Quantidade	570	1254	341	190	25	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	20,00%	10,53%	13,04%	11,11%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,50%	4,55%	6,00%	8,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C60	Quantidade	528	1245	393	147	26	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	50,00%	10,53%	14,29%	10,53%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,44%	4,44%	6,12%	**7,55%**	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C61	Quantidade	1053	708	342	308	42	4	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	11,11%	10,00%	13,64%	11,76%	12,20%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,70%	4,55%	5,88%	8,51%	11,11%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C62	Quantidade	53	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	5,88%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	5,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
C63	Quantidade	507	1225	421	158	20	1	0	0	0	0	0	0	0	0	0	0
	Maior f(x)	0,00%	20,00%	11,76%	13,64%	10,81%	11,11%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	Menor f(x)	0,00%	2,38%	4,17%	5,88%	8,33%	11,11%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

Na Tabela 27 pode-se verificar que o cenário 1 (C01) obteve a solução com o maior valor para a relação quantidade de componentes com erro dividido pela quantidade total de componentes (Equação 6), quando encontrou de 2 a 5 componentes com erro. Isso se deve ao fato de todos os componentes terem um custo de manutenção futura (CMF).

O cenário 2 (C02) que usa a métrica propensão a defeito selecionou um componente com erro, sendo que este componente possui cheiros de código identificados (Figura 20). Os quatro cenários C03, C04, C05 e C06 que usam as métricas frequência de chamadas (FC), risco de falha (RF), vulnerabilidade de mercado (VM) e valor de negócio (VN), respectivamente, não selecionaram nenhum componente com erro, pois para diminuir o espaço de busca do algoritmo genético foram removidos os componentes com complexidade ciclomática menor que dois, e isso inclua o componente CE14 que possui complexidade ciclomática igual a um.

Com relação ao melhor caso vamos analisar o cenário C01, pois esse foi o melhor cenário para a Equação 6. A melhor solução para o cenário C01 com 5 componentes com erro obteve 19,58% do total de componentes da aplicação, sendo que com essa quantidade de componentes essa solução cobriu 33,33% dos componentes com erro identificados e ficou com 86,84% de componentes a mais. Isso indica que no melhor caso o SCoTUAM está alcançando uma cobertura baixa dos componentes com erro, sendo que, a maioria (10; 66,67%) dos componentes com erro para essa aplicação possui complexidade ciclomática igual a um, impossibilitando sua seleção pelo plugin.

No que se refere ao pior caso vamos analisar o cenário C23, pois ele teve o menor valor para a Equação 6. A pior solução para o cenário C23 obteve 23,19% do total de componentes da aplicação, sendo que com essa quantidade de componentes essa solução cobriu 6,66% dos componentes com erro identificados e ficou com 97,77% de componentes a mais.

5.4.2.4. Componentes com Erro Geral

Na Figura 21 é mostrada a porcentagem de soluções para cada quantidade de componentes com erro identificados nas soluções da fronteira de Pareto. Observa-se que a maioria das soluções da fronteira de Pareto identificou um componente com erro e 32,09% das soluções identificaram zero componentes com erro. Isso indica que o SCoTUAM possui uma baixa cobertura de erros, quando os erros se concentram em componentes com complexidade igual a um.

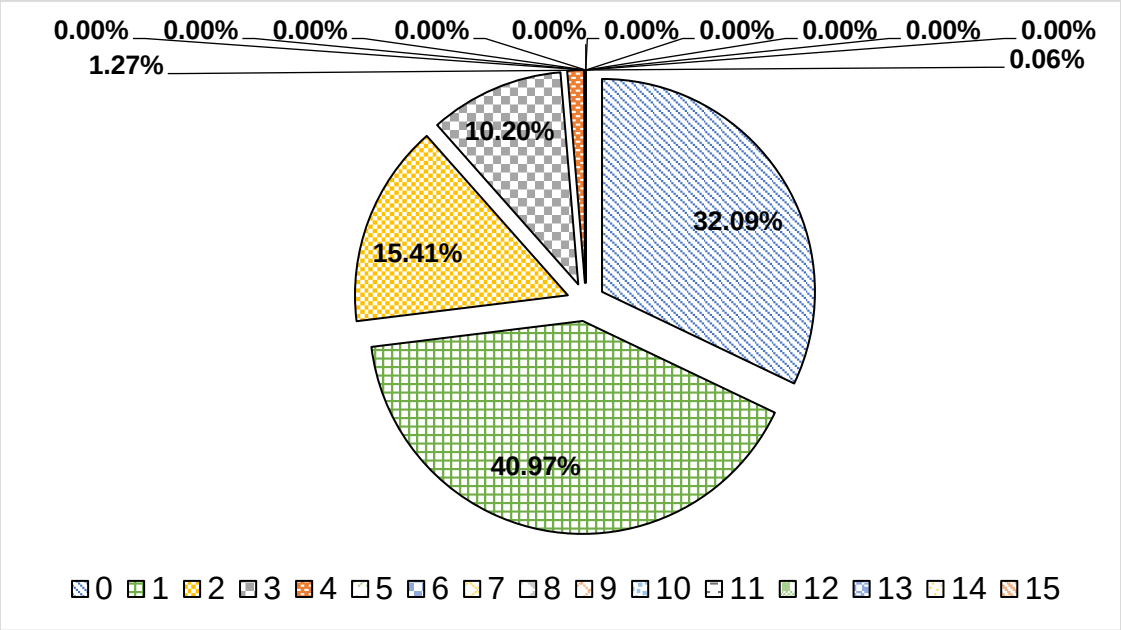


Figura 21. Porcentagem de soluções do SCoTUAM para a aplicação *Recurrence*.

Para um melhor entendimento, a Figura 22 nos mostra que 95,31% dos componentes nas soluções não continham erros. Por este motivo, percebe-se que o plugin SCoTUAM não possui uma boa cobertura de erros quando os erros se concentram em componentes com complexidade ciclomática igual a um.

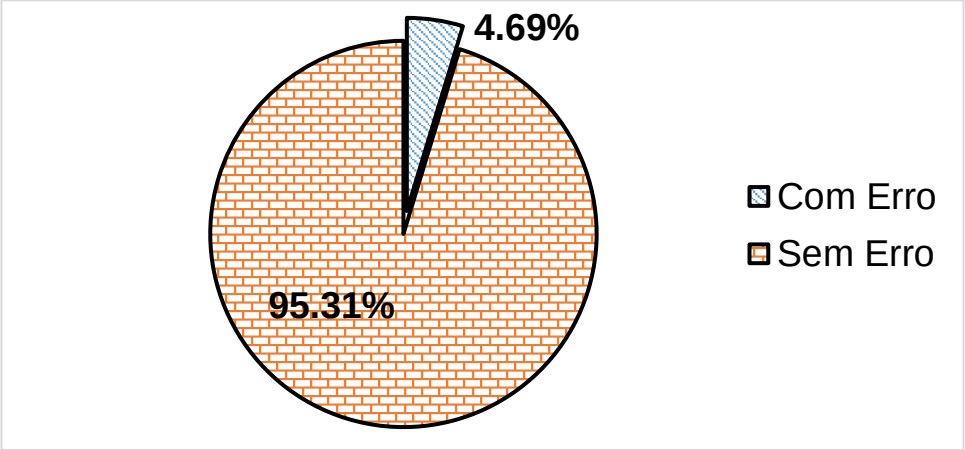


Figura 22. Porcentagem de componentes com e sem erro das soluções do SCoTUAM para a aplicação *Recurrence*.

5.4.2.5. Métricas e Componentes com Erros

No que diz respeito às métricas que compõem os cenários de priorização da seleção de componentes, a Tabela 28 apresenta a quantidade de componentes com erro (CE) e sem erro (SE) para cada uma das métricas. A métrica custo de manutenção futura tem a maior quantidade de componentes com erro e sem erro, devido ao fato de todos os componentes terem um valor para essa métrica, fazendo com que os cenários que utilizam essa métrica

possuam mais soluções e consequentemente mais componentes. Percebe-se ainda que a relação entre componentes com erro e sem erro foi igual a todas as métricas, ficando com o valor de 0,04.

Tabela 28. Métricas e componentes com erro para a aplicação *Recurrence*.

MÉTRICA	CE	SE	CE / SE
Custo de Manutenção Futura	79050	1596576	0,04
Vulnerabilidade de Mercado	40194	850033	0,04
Valor de Negócio	39193	835852	0,04
Code Smell	41456	842526	0,04
Frequência de Chamadas	39145	823446	0,04
Risco de Falha	39389	832566	0,04

5.4.2.6. Comparação entre os Especialistas e o Plugin SCoTUAM

Para a comparação da seleção automatizada utilizando o plugin SCoTUAM com a seleção manual realizada por especialistas, foi utilizado o resultado dos especialistas E1 e E4, pois foram os especialistas entre os cinco participantes que obtiveram uma maior cobertura (2; 13,33%) dos componentes com erros. Para o SCoTUAM, foi selecionado os cenários C01, C07, C28, C29 e C58 pois eles tiveram o melhor resultado entre os 63 cenários.

A Figura 23 apresenta uma comparação entre a porcentagem de componentes com erro selecionados pelos especialistas (E1 e E4) e pelos cinco melhores cenários (C01, C07, C28, C29 e C58) do plugin SCoTUAM. Observa-se que os especialistas E1 e E4 obtiveram uma menor cobertura dos componentes com erro que o SCoTUAM.

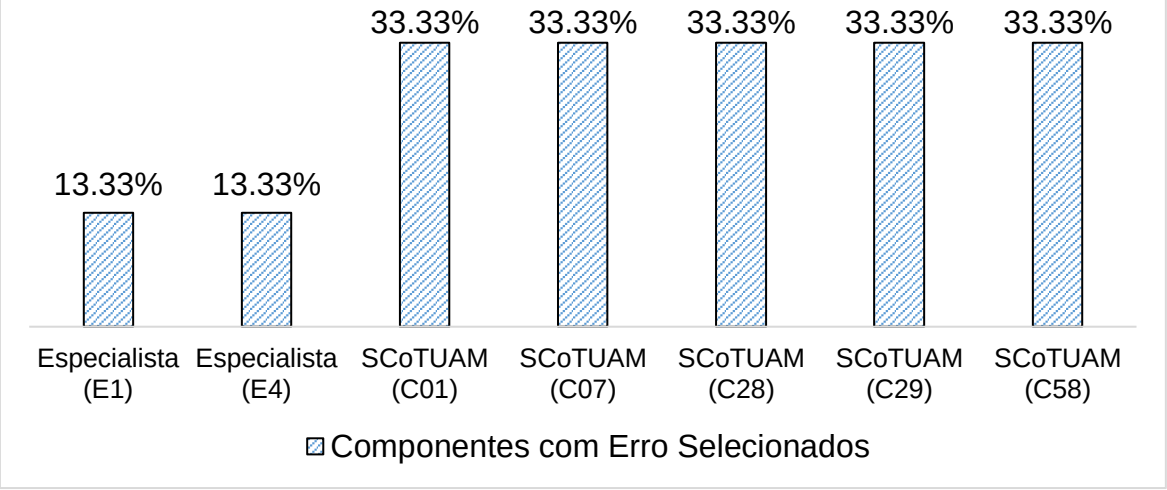


Figura 23. Comparação entre especialistas e SCoTUAM para a aplicação *Recurrence*.

Pode-se observar também que os especialistas E1 e E4 trouxeram uma quantidade maior de componentes que não foram identificados com erro do que o SCoTUAM, conforme mostrado na Tabela 29. Isso mostra que o plugin SCoTUAM obteve um melhor resultado que os especialistas para essa aplicação.

Tabela 29. Resultado da Equação 6 para a aplicação *Recurrence*.

	Equação 6 (fx)
Especialista 1 (E1)	8,33%
Especialista 4 (E4)	6,06%
SCoTUAM (C01)	13,16%
SCoTUAM (C07)	13,16%
SCoTUAM (C28)	13,16%
SCoTUAM (C29)	13,16%
SCoTUAM (C58)	13,16%

5.4.3. Ameaças à Validade

Validade Interna

Mortalidade: Este efeito é devido aos diferentes tipos de pessoas que abandonam o experimento. Para diminuir essa ameaça o tempo de envio da segunda parte do estudo com os especialistas foi no máximo 48 horas após a resposta do formulário de caracterização.

Seleção dos estudos: A amostra dos projetos não foi aleatória, visto que era proveniente da disponibilidade de aplicações de código aberto do f-droid que continham scripts de teste em espresso dentro do grupo de pesquisa no qual o trabalho foi realizado. Como este é um estudo de viabilidade, acredita-se que essa questão não representa uma ameaça significativa.

Validade de Construção

Viés mono-operação: O estudo de viabilidade utilizou duas aplicações da plataforma Android de diferentes categorias desenvolvidas na linguagem Java. O estudo pode não ser representativo para outras categorias de aplicações.

Validade Externa

Interação de seleção e tratamento: Foi aplicado um questionário para a caracterização do perfil, para verificar se o participante se enquadrava no perfil esperado para o estudo.

Generalização dos resultados: Para diminuir essa ameaça foi utilizado duas aplicações de diferentes categorias, o estudo pode não ser representativo para outras categorias de aplicações móveis.. Futuramente, pretende-se aumentar o número de aplicações de diferentes categorias.

Validade de Conclusão

Confiança das Medidas: Na seleção dos componentes utilizando o plugin SCoTUAM foi realizando 30 execuções do algoritmo genético para diminuir o efeito da aleatoriedade.

5.5. Considerações Finais

Este capítulo apresentou o estudo de viabilidade para o plugin SCoTUAM, correspondente a fase de avaliação da metodologia adotada nesta pesquisa. Com o objetivo de analisar a viabilidade, por meio de um experimento controlado, com respeito a eficácia em selecionar componentes com erro comparando com a lista gerada por meio da seleção de componentes realizada por especialistas.

Os resultados confirmaram a viabilidade da proposta em auxiliar o desenvolvedor na seleção de componentes para o teste de unidade, no entanto, percebeu-se a necessidade de refinamento para melhorar o resultado em alguns cenários de priorização da seleção de componentes. No próximo capítulo, serão apresentadas as considerações finais e contribuições deste trabalho, assim como as perspectivas de trabalhos futuros para a continuação desta pesquisa.

CAPÍTULO 6 – CONCLUSÕES E TRABALHOS FUTUROS

Neste capítulo serão apresentadas as considerações finais e conclusões sobre este trabalho de pesquisa, também são indicados os trabalhos futuros como oportunidades de investigação para dar continuidade a esta pesquisa.

6.1. Considerações Finais

A realização do teste de unidade traz alguns benefícios como redução de falhas em recursos já existentes, melhoram a estrutura do código, diminuem os efeitos colaterais (side effects) e reduzem o medo da alteração do código (Burke e Coyner, 2017). No entanto, a atividade de teste para aplicações móveis tem o tempo reduzido, fazendo com que alguns desenvolvedores optem por não criar os testes de unidade. Portanto, a automação do teste é uma necessidade.

Tendo em vista o cenário apresentado, este trabalhou apresentou um plugin para a seleção de componentes para a criação de teste de unidade em aplicações móveis, com o intuito de aumentar o valor de custo x benefício dos componentes selecionados. Para o plugin SCoTUAM foi selecionado um conjunto de métricas para medir o valor do custo e benefício dos componentes, a saber, *halstead effort* (HE), custo de manutenção futura (CMF), cheiro de código (CS), frequência de chamadas (FC), risco de falha (FC), vulnerabilidade de mercado (VM) e valor de negócio (VN).

Foi realizado um primeiro estudo (seção 3.8) com o propósito de analisar a correlação entre as métricas. O resultado deste estudo empírico mostrou a possibilidade de usar as métricas custo de manutenção futura (CMF), cheiro de código (CS), frequência de chamada (FC), risco de falha (RF), vulnerabilidade de mercado (VM) e valor de negócio (VN) combinadas numa solução para a seleção de componentes.

Os resultados obtidos do segundo estudo empírico, apresentado no Capítulo 5, permitiram aceitar a hipótese definida para está pesquisa. Porquanto o plugin proposto ajuda na seleção de componentes que tenham um maior valor em relação ao custo x benefício para a criação dos testes de unidade em aplicações móveis Android.

6.2. Contribuições

As principais contribuições desta pesquisa oferecidas à comunidade acadêmica e à indústria são:

1. Definição de um plugin para a seleção de componentes em aplicações móveis na plataforma Android utilizando um algoritmo evolutivo baseado em métricas estáticas, dinâmicas, de mercado e de negócio.
2. A análise de correlação das métricas custo de manutenção futura (CMF), cheiro de código (CS), frequência de chamadas (FC), risco de falha (RF), vulnerabilidade de mercado (VM) e valor de negócio (VN).
3. A implementação do plugin chamado SCoTUAM para a IDE *Android Studio*;
4. A análise da seleção automatizada de componentes utilizando o plugin SCoTUAM em relação a seleção manual realizada por especialistas em testes de unidades em aplicações móveis Android;
5. Artigo com co-autoria (Publicado):
    a. What are Software Engineers asking about Android Testing on Stack Overflow? – SBES 2016.

6.3. Limitações

Algumas limitações foram observadas no decorrer da pesquisa:

- Os estudos foram realizados apenas para a plataforma Android, então há a necessidade de realizar estudos na plataforma iOS;
- Os participantes do estudo de viabilidade são representativos, porém é importante realizar um estudo com um maior número de participantes.

6.4. Trabalhos Futuros

Como trabalhos futuros, está prevista a realização de um estudo utilizando o modelo de aceitação de tecnologia (em inglês, *Technology Acceptance Model - TAM*), com o intuito de avaliar a usabilidade e aceitação do plugin SCoTUAM. Outra possibilidade seria um estudo que avalie a eficiência da utilização do plugin SCoTUAM na plataforma Android.

A atual versão do plugin SCoTUAM faz a seleção dos componentes para o teste unitário, seria interessante a inclusão da geração automática dos testes unitários a partir dos componentes selecionados pelo SCoTUAM.

Outra pesquisa futura interessante é analisar como se comporta a seleção dos componentes por meio de outras soluções multi-objetivos. Por fim, há a possibilidade de adaptar o plugin e realizar um estudo de viabilidade para a plataforma iOS.

REFERÊNCIAS BIBLIOGRÁFICAS

Afzal, Wasif; Torkar, Richard; Feldt, Robert. A systematic review of search-based testing for non-functional system properties. Information and Software Technology, v. 51, n. 6, p. 957-976, 2009.

Amorim, Lucas; Costa, Evandro; Antunes, Nuno; Fonseca, Baldoino; Ribeiro, Márcio. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In: 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE. p. 261-269, 2015.

Android. Application Fundamentals. Disponível em: <https://goo.gl/59sibi>. Acesso em: 10 dez. 2016a.

Android. Mercado. Disponível em: <https://goo.gl/8d974X>. Acesso em: 10 dez. 2016b.

Android. Activity. Disponível em: <https://goo.gl/xGtrkK>. Acesso em: 20 dez. 2017.

Burke, Eric M.; Coyner, Brian M. Top 12 Reasons to Write Unit Tests. Disponível em: <https://goo.gl/XqaU7k>. Acesso em: 20 dez. 2017.

Boehm, Barry W.; Brown, John R.; Lipow, Mlity. Quantitative evaluation of software quality. In: Proceedings of the 2nd international conference on Software engineering (ICSE). IEEE Computer Society Press. p. 592-605, 1976.

Bourque, Pierre *et al*. Guide to the software engineering body of knowledge (SWEBOK): Version 3.0. IEEE Computer Society Press, 2014.

Deb, Kalyanmoy; Pratap, Amrit; Agarwal, Sameer. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE transactions on evolutionary computation, v. 6, n. 2, p. 182-197, 2002.

Deng, Lin; Offutt, Jeff; Ammann, Paul; Mirzaei, Nariman. Mutation operators for testing Android apps. Information and Software Technology, V. 81, p. 154-168, 2016.

Dustin, Elfriede. Effective Software Testing: 50 Ways to Improve Your Software Testing. Addison-Wesley Longman Publishing Co., Inc. Boston, 2002.

Euteneuer, Sven; Padwal, Abhijeet; Chandra, Himanshu; Kulkarini, Parag; Sarma, Nandini. Mobile Test Automation. Solutions to the Mobile Test Efficiency Challenge. 2013.

Episkopos, Dennis C.; Li, J. Jenny; Yee, Howell S.; Weiss, David M. Prioritize code for testing to improve code coverage of complex software. U.S. Patent n. 7,886,272, 8 fev. 2011.

Feist, Josselin; Mounier, Laurent; Potet, Marie-Laure. Guided dynamic symbolic execution using subgraph control-flow information. In: International Conference on Software Engineering and Formal Methods (SEFM). Springer International Publishing, v. 9763, p. 76-81, 2016.

Fowler, M. TestPyramid. 2012. Disponível em: <http://goo.gl/VbrNqF>. Acesso em: 26 jan. 2016.

Fowler, Martin; Beck, Kent; Brant, John; Opdyke, William; Roberts, Don. Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

Fontana, Francesca Arcelli; Ferme, Vincenzo; Zanoni, Marco; Roveda, Riccardo. Towards a prioritization of code debt: A code smell Intensity Index. In: 7th International Workshop on Managing Technical Debt (MTD). IEEE. p. 16-24, 2015.

Freitas, Eduardo Noronha de Andrade; Camilo-Junior, Celso Gonçalves; Vincenzi, Auri Marcelo Rizzo. SCOUT: A Multi-objective Method to Select Components in Designing Unit Testing. In: 27th International Symposium on Software Reliability Engineering (ISSRE). IEEE. p. 36-46, 2016.

Freitas, Eduardo Noronha de Andrade; Vincenzi, Auri Marcelo Rizzo; Camilo-Junior, Celso Gonçalves. Prioritization of Artifacts for Unit Testing Using Genetic Algorithm Multi-objective Non Pareto. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP). 2014.

Gao, Jerry; Bai, Xiaoying; Tsai, Wey-Tek; Uehara, T. Mobile application testing. Computer, v. 47, n. 2, p. 46-55, 2014.

Haghpanah, Nima; Moaven, Shahrouz; Habibi, Jafar; Kargar, Mehdi; Yeganeh, Soheil Hassas. Approximation algorithms for software component selection problem. In: 14th Software Engineering Conference (APSEC). Asia-Pacific. IEEE. p. 159-166, 2007.

Halstead, Maurice H. Advances in software science. Advances in Computers, v. 18. Elsevier, p. 119-172, 1979.

Harman, Mark; Jones, Bryan F. Search-based software engineering. Information and software Technology, v. 43, n. 14, p. 833-839, 2001.

Harman, Mark *et al*. Search--based approaches to the component selection and prioritization problem. In: Proceedings of the 8th annual conference on Genetic and evolutionary computation (GECCO). ACM. p. 1951-1952, 2006.

Hecht, Geoffrey; Moha, Naouel; Rouvoy, Romain. An Empirical Study of the Performance Impacts of Android Code Smells. In: IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE, 2016.

Hosseingholizadeh, Ahmad. A source-based risk analysis approach for software test optimization. In: International Conference on Computer Engineering and Technology (ICCET). IEEE. V. 2, p. 601-604, 2010.

Huizinga, Dorota; Kolawa, Adam. Automated defect prevention: best practices in software management. Wiley-IEEE Computer Society Press, 2007.

Iannoni, Ana Paula; Morabito, Reinaldo. Modelo hipercubo integrado a um algoritmo genético para análise de sistemas médicos emergenciais em rodovias. Gestão & Produção, v. 13, n. 1, p. 93-104, 2006.

Joorabchi, Mona Erfani; Mesbah, Ali; Kruchten, Philippe. Real challenges in mobile app development. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE. p. 15-24, 2013.

Jorgensen, Paul C. Software testing: a craftsman's approach. CRC press, 2016.

Karhu, K.; Repo, T.; Taipale, O.; Smolander, K. Empirical observations on software testing automation. In: International Conference on Software Testing Verification and Validation (ICST). IEEE, p. 201-209, 2009.

Kasai, Norimitsu; Morisaki, Shuji; Matsumoto, Kenichi. Fault-Prone Module Prediction Using a Prediction Model and Manual Inspection. In: 20th Asia-Pacific Software Engineering Conference (APSEC). IEEE. p. 106-115, 2013.

Lewis, William E. Software testing and continuous quality improvement. CRC press, Ed. 4 revised, 2016.

Li, J. Jenny. Prioritize code for testing to improve code coverage of complex software. In: 16th IEEE International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2005.

Li, J. Jenny; Weiss, David; Yee, Howell. Code-coverage guided prioritized test generation. Information and Software Technology, v. 48, n. 12, p. 1187-1198, 2006.

Liu, Hui; Liu, Qiurong; Niu, Zhendong; Liu, Yang. Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection. IEEE Transactions on Software Engineering. v. 42, n. 6, 2016.

Ma, Xin; Wang, Ning; Xie, Peizhang; Zhou, Jungui; Zhang, Xiaofang; Fang, Churong. An Automated Testing Platform for Mobile Applications. In: IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE. p. 159-162, 2016.

Mafra, S.N., Barcelos, R.F., Travassos, G.H. Aplicando uma Metodologia Baseada em Evidência na Definição de Novas Tecnologias de Software. Proceedings of the 20th Brazilian Symposium on Software Engineering (SBES), v. 1, p. 239-254, 2006.

Maia, Camila; Freitas, Fabrício Gomes de; Maia, Brito; Souza, Jefferson; Coutinho, Daniel Pinto; Campos, Gustavo. Aplicação de metaheurísticas em problemas da engenharia de software: Revisão de literatura. In: II Congresso Tecnológico Infobrasil. 2009.

Malhotra, Ruchika; Chug, Anuradha; Khosla, Priyanka. Prioritization of Classes for Refactoring: A Step towards Improvement in Software Quality. In: Proceedings of the Third International Symposium on Women in Computing and Informatics. ACM. p. 228-234, 2015.

Marinho, Davi. Uma Aplicação do Algoritmo Genético Multiobjetivo NSGA II Para Seleção de Imagens de Satélite de Trechos de Mata Atlântica. Recife, PE, 2009.

Martin, Robert C. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009.

Mockus, Audris; Votta, Lawrence G. Identifying Reasons for Software Changes using Historic Databases. In: Proceedings of the International Conference on Software Maintenance (ICSM). p. 120-130, 2000.

Müller, Thomas *et al*. Certified Tester-Foundation Level Syllabus-Version 2007. International Software Testing Qualifications Board (ISTQB), Möhrendorf, Germany, 2007.

Nebro, Antonio J.; Durillo, Juan J.; Vergne, Matthieu. Redesigning the jMetal multi-objective optimization framework. In: Proceedings of the Companion Publication of the Annual Conference on Genetic and Evolutionary Computation. ACM. p. 1093-1100, 2015.

Nimbalkar, Ravi Ramchandra. Mobile application testing and challenges. International Journal of Science and Research, v. 2, n. 7, p. 56-58, 2013.

Okediran, O. O.; Arulogun, O. T.; Ganiyu, R. A.; & Oyeleye, C. A. Mobile operating systems and application development platforms: A survey. International Journal of Advanced Networking and Applications, v. 6, n. 1, p. 2195, 2014.

Øvergaard, Asgeir; Muller, Gerrit. System Verification by Automatic Testing. In: INCOSE International Symposium. p. 356-367, 2013.

Parasoft. Automated Defect Prevention for Embedded Systems Software Development. 2009.

Rajapakse, Damith C. Fragmentation of mobile applications. In: Handbook of Research on Mobile Software Engineering: Design, Implementation, and Emergent Applications. IGI Global. p. 317-335. 2008.

Ray, Mitrabinda; Lal Kumawat, Kanhaiya; Mohapatra, Durga Prasad. Source code prioritization using forward slicing for exposing critical elements in a program. Journal of Computer Science and Technology, v. 26, n. 2, p. 314-327, 2011.

Ray, Mitrabinda; Mohapatra, Durga Prasad. Code-based prioritization: a pre-testing effort to minimize post-release failures. Innovations in Systems and Software Engineering, v. 8, n. 4, p. 279-292, 2012.

Ray, Mitrabinda; Mohapatra, Durga Prasad. Multi-objective test prioritization via a genetic algorithm. Innovations in Systems and Software Engineering, v. 10, n. 4, p. 261-270, 2014.

Rumsey, Deborah. Statistics II for dummies. Wiley Press, Hoboken, NJ, USA, 2009.

Samuel, Triin; Pfahl, Dietmar. Problems and solutions in mobile application testing. In: 17th International Conference on Product-Focused Software Process Improvement (PROFES). Springer International Publishing. p. 249-267, 2016.

Schumacher, Jan; Nico, Zazworka; Shull, Forrest; Seaman, Carolyn; Shaw, Michele. Building empirical support for automated code smell detection. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ACM. n. 8, 2010.

Segundo, Alexandre*; Maciel, Jailton*. Seleção automatizada de componentes de software orientada por métricas estruturais e informações de reuso. 2014.

Shihab, Emad; Jiang, Zhen Ming; Adams, Bram; Hassan, Ahmed E.; Bowerman, Robert. Prioritizing unit test creation for test-driven maintenance of legacy systems. In: 10th International Conference on Quality Software. IEEE. p. 132-141, 2010.

Statista, Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017. Disponível em: <https://goo.gl/ptwwuN>. Acesso em: 20 dez. 2017a.

Statista, Number of available applications in the Google Play Store from December 2009 to December 2017. Disponível em: <https://goo.gl/mjRzx1>. Acesso em: 20 dez. 2017b.

Techapalokul, Peeratham; Tilevich, Eli. Programming environments for blocks need first-class software refactoring support: A position paper. In: IEEE Blocks and Beyond Workshop (Blocks and Beyond). IEEE. p. 109-111, 2015.

Vidal, Santiago; Vazquez, Hernan; Diaz-Pace, J. Andres; Marcos, Claudia; Garcia, Alessandro; Oizumi, William. JSpIRIT: a flexible tool for the analysis of code smells. In: 34th International Conference of the Chilean Computer Science Society (SCCC). IEEE. p. 1-6, 2015.

Vidal, Santiago, Guimaraes, Everton; Oizumi, William; Garcia, Alessandro; Diaz-Pace, J. Andres; Marcos, Claudia. On the criteria for prioritizing code anomalies to identify architectural problems. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM. p. 1812-1814, 2016.

Xie, T.; Taneja, K.; Kale, S.; Marinov, D. Towards a framework for differential unit testing of object-oriented programs. In: Proceedings of the Second International Workshop on Automation of Software Test. IEEE Computer Society. p. 5, 2007.

Yadav, Pankaj; Yadav, Umesh Kr; Verma, Surbhi. Software Testing: Approach to Identify Software Bugs. Architecture, v. 1, n. 10, p. 15, 2012.

Zakaria, Noor Azura; Ibrahim, Suhaimi; Mahrin, Mohd Naz'ri. A proposed value-based software process tailoring framework. In: 9th Malaysian Software Engineering Conference (MySEC). IEEE. p. 149-153, 2015.