PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA – PPGIa

ADRIANO PIZZINI

SENTINEL: PROCESSO PARA REMOÇÃO AUTOMÁTICA DE TEST SMELLS

Tese apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Doutor em Informática.

Curitiba
2024

ADRIANO PIZZINI

SENTINEL: PROCESSO PARA REMOÇÃO AUTOMÁTICA DE TEST SMELLS

Tese apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Doutor em Informática.

Área de concentração: Ciência da Computação

Orientadora: Profª. Dra. Andreia Malucelli
Coorientadora: Profª. Dra. Sheila Reinehr

Curitiba
2024

Dados da Catalogação na Publicação
Pontifícia Universidade Católica do Paraná
Sistema Integrado de Bibliotecas – SIBI/PUCPR
Biblioteca Central
Gisele Alves – CRB 9/1578

P695s 2024

Pizzini, Adriano
Sentinel : processo para remoção automática de test smells / Adriano Pizzini ; orientadora : Andreia Malucelli ; coorientadora: Sheila Reinehr. – 2024.
202 f. ; il. : 30 cm

Tese (doutorado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2024
Bibliografia: f. 155-167

1. Software – Controle de qualidade. 2. Software – Refatoração. I. Malucelli, Andreia. II. Reinehr, Sheila dos Santos. III. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática. IV. Título.

CDD. 20. ed. – 004

Pontifícia Universidade Católica do Paraná
Escola Politécnica
Programa de Pós-Graduação em Informática

Curitiba, 17 de janeiro de 2025.

06-2025

DECLARAÇÃO

Declaro para os devidos fins, que **ADRIANO PIZZINI** defendeu a tese de Doutorado intitulada "**SENTINEL: PROCESSO PARA REMOÇÃO AUTOMÁTICA DE TEST SMELLS**", na área de concentração Ciência da Computação no dia 02 de dezembro de 2024, no qual foi aprovado.

Declaro ainda, que foram feitas todas as alterações solicitadas pela Banca Examinadora, cumprindo todas as normas de formatação definidas pelo Programa.

Por ser verdade firmo a presente declaração.

gov.br Documento assinado digitalmente
EMERSON CABRERA PARAISO
Data: 17/01/2025 09:52:12-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Emerson Cabrera Paraiso
Coordenador do Programa de Pós-Graduação em Informática

DEDICATÓRIAS

A Deus.

Aos meus pais.

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Agradeço à Deus pelo dom da vida.

Aos meus pais, seu Valdir e dona Ires, pelo amor, carinho e incentivo em todas as etapas da vida.

Aos meus irmãos e minha irmã.

À minha orientadora Andreia Malucelli, e à minha coorientadora Sheila Reinehr, por compartilharem suas experiências, conhecimentos e por me orientarem na execução do trabalho, além de serem exemplos de conduta profissional.

Aos colegas do Grupo de Pesquisa em Engenharia de Software (GEPS), por compartilharam experiências e sugerirem correções e melhorias no trabalho.

Aos colegas, e hoje doutores, Luiz Fernando Puttow Southier e Sheila Cristiana de Freitas, pela parceria durante as disciplinas e outras atividades realizadas durante o doutorado.

Aos professores pesquisadores da Pontifícia Universidade Católica do Paraná (PUCPR).

Ao Instituto Federal Catarinense (IFC), por possibilitar a realização do trabalho em tempo integral.

À Coordenação e Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES), pelo apoio financeiro por meio do Código de Financiamento 001.

Por fim, mas não com menor importância, para uma mulher prendada em especial que eu, carinhosamente, chamo de prenda por suas qualidades.

RESUMO

Os testes de unidade são testes desenvolvidos para identificar erros em partes do código de produção que podem ser testados isoladamente, como métodos e classes. A presença de Test Smells (TS) nos testes de unidade prejudica a legibilidade e manutenção do código do teste. Além disso, a quantidade de ocorrências de TS no código de teste aumenta com decorrer do tempo, sugerindo que a equipe de desenvolvimento desconhece a sua existência ou não tem tempo hábil para remover, contribuindo para deteriorar ainda mais a qualidade dos testes de unidade. O objetivo desta tese é desenvolver um processo para automatizar a remoção de ocorrências de smells de classes de testes de unidade que possa apoiar a equipe de desenvolvimento na melhoria da qualidade dos testes de unidade por meio da remoção de smells. Esta pesquisa utiliza Design Science Research Methodology (DSRM), método de pesquisa é composto por seis etapas: identificação do problema e motivação; definição dos objetivos para uma solução; projetar e desenvolver o processo proposto; demonstrar o processo proposto para especialistas e integrantes de equipes de desenvolvimento; avaliar os resultados qualitativos das avaliações de modo a aperfeiçoar o processo proposto; e comunicar os resultados por meio da publicação de artigos científicos e da escrita da tese. Para realizar as refatorações, foram selecionados os TS Assertion Roulette e Redundant Assert, Exception Catching Throwing, Eager Test e Test Code Duplication por afetarem trechos diferentes do código do teste, como linha de código, blocos de código, código do método de teste e mais de um método de teste, respectivamente. A avaliação do processo proposto baseou-se nos critérios facilidade, utilidade, intenção de uso do Technology Acceptance Model 3. Os resultados da avaliação do processo Sentinel sugere que o processo foi considerado como adequado pelos avaliadores. A implementação baseada em comandos de terminal foi considerada de fácil integração com os processos da indústria, mas de difícil utilização em outros contextos devido à ausência de interface gráfica. Em relação à utilidade, o processo proposto pode ser útil tanto para a academia, gerando pesquisas e comparações entre versões de testes, quanto para a indústria, oferecendo aprendizado e melhorias na qualidade dos testes. No entanto, futuros estudos de caso são necessários para facilitar sua adoção na indústria. Quanto à intenção de uso, os avaliadores consideraram o processo promissor, sugerindo que o processo pode contribuir para a melhoria da qualidade dos testes de unidade e que um aplicativo que

implemente o processo proposto pode ser bem recebido pela indústria. Em relação ao refatorador, as sugestões focaram em melhorias para apoiar a pesquisa e novas funcionalidades, como configuração das refatorações a serem realizadas, geração de log das refatorações e a integração com o backlog. A principal preocupação foi a necessidade de flexibilizar as refatorações e minimizar o tempo de execução, especialmente na validação, para não impactar os prazos de entrega. A remoção automática de TS não é uma tarefa trivial, pois requer a identificação automática de problemas decorrentes da remoção, como erros de compilação, mudança de comportamento dos testes e do software sendo testado. Para apoiar adequadamente às equipes, é necessário que o processo considere aspectos do contexto de atuação de cada equipe, tais como a linguagem de programação e frameworks usados, a experiência em desenvolvimento e o conhecimento sobre TS, além do benefício e prejuízos causados pela remoção automática de TS. Portanto, a priorização das refatorações deve focar nas ocorrências de TS cujas remoções sejam mais benéficas para as equipes e reduzam o esforço da equipe na remoção de TS e, também, reduzam o tempo de refatoração.

Palavras-chaves: Qualidade de Software; Refatoração; Testes de Unidade; Test Smells; Avaliação de qualidade.

ABSTRACT

Unit tests are tests designed to identify errors in parts of the production code that can be tested in isolation, such as methods and classes. The presence of Test Smells (TS) in unit tests harms the readability and maintainability of the test code. In addition, the number of TS occurrences in the test code increases over time, suggesting that the development team is unaware of their existence or does not have enough time to remove them, contributing to further deterioration of the quality of the unit tests. The objective of this thesis is to develop a process to automate the removal of smell occurrences from unit test classes that can support the development team in improving the quality of unit tests by removing smells. This research uses Design Science Research Methodology (DSRM), a research method composed of six steps: problem identification and motivation; definition of objectives for a solution; design and development of the proposed process; demonstration of the proposed process to experts and members of development teams; evaluation of the qualitative results of the evaluations in order to improve the proposed process; and communicate the results through the publication of scientific articles and the writing of the thesis. To perform the refactoring's, the TS Assertion Roulette and Redundant Assert, Exception Catching Throwing, Eager Test and Test Code Duplication were selected because they affect different parts of the test code, such as code line, code blocks, test method code and more than one test method, respectively. The evaluation of the proposed process was based on the criteria easiness, usefulness, and intended use of the Technology Acceptance Model 3. The results of the evaluation of the Sentinel process suggest that the process was considered adequate by the evaluators. The implementation based on terminal commands was considered easy to integrate with industry processes, but difficult to use in other contexts due to the lack of a graphical interface. Regarding usefulness, the proposed process can be useful both for academia, generating research and comparisons between test versions, and for industry, offering learning and improvements in test quality. However, future case studies are needed to facilitate its adoption in industry. Regarding intended use, the evaluators considered the process promising, suggesting that the process can contribute to improving the quality of unit tests and that an application that implements the proposed process can be well received by the industry. Regarding the refactoring, the suggestions focused on improvements to support research and new features, such as configuring the

refactoring’s to be performed, generating a refactoring log, and integrating with the backlog. The main concern was the need to make refactoring’s more flexible and minimize execution time, especially in validation, so as not to impact delivery deadlines. Automatic TS removal is not a trivial task, as it requires the automatic identification of problems resulting from the removal, such as compilation errors, changes in test behavior, and the software being tested. To adequately support teams, the process must consider aspects of each team's operating context, such as the programming language and frameworks used, development experience, and knowledge of TS, in addition to the benefits and drawbacks caused by automatic TS removal. Therefore, the prioritization of refactoring’s should focus on TS occurrences whose removals are more beneficial to the teams and reduce the team's effort in removing TS and, also, reduce refactoring time.

Keywords: Software Quality; Refactoring; Unit Test; Test Smells; Quality assessment.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AL	Assertionless
AR	Assertion Roulette
AUU	Abnormal UTF-Use
BC	Blob Class
BPMN	Business Process Model and Notation
CBO	Coupling between Object Classes
CCC	Client-based class cohesion
CDSBP	Class Data Should Be Private
CEP	Comitê de Ética em Pesquisa
CI	Constructor Initialization
CTL	Conditional Test Logic
CVDS	Ciclo de Vida de Desenvolvimento de Software
DA	Duplicate Assert
DC	Duplicated Code
DepT	Dependent Test
DfT	Default Test
DIT	Depth of Inheritance Tree
DSR	Design Science Research
DSRM	Design Science Research Methodology
DT	Dívida Técnica
ECT	Exception Catching Throwing
EH	Exception Handling
EmT	Empty Test
ET	Eager Test

FGAT	Ferramentas de Geração Automática de Testes
FTO	For Testers Only
GF	General Fixture
ICSE I	International Conference on Software Engineering
ID	Indirect Test
IgT	Ignored Test
InT	Indented Test
LCOM	Lack of Cohesion of Methods
LLOC	Logical Lines Of Code
LT	Lazy Test
MG	Mystery Guest
MI	Maintainability Index
ML	Machine Learning
MNT	Magic Number Test
NOC	Number of Children
NOI	Number of Outgoing Invocations
OO	Orientação a Objetos
PUCPR	Pontifícia Universidade Católica do Paraná
RA	Redundant Assertion
RFC	Response for a Class
RO	Resource Optimism
RP	Redundant Print
SBQS	Simpósio Brasileiro de Qualidade de Software
SC	Spaghetti Code
SE	Sensitive Equality

ST	Sleepy Test
SUT	System Under Test
TAM	Technology Acceptance Model
TCD	Test Code Duplication
TCLE	Termo de Consentimento Livre e Esclarecido
TCUD	Termo de Confidencialidade de Uso de Dados
TLLOC	Total Logical Lines of Code
TNOS	Total Number of Statements
TRW	Test Run War
TS	Test Smells
UT	Unknown Test
VT	Verbose Test
WMC	Weighted Methods per Class

CAPÍTULO 1 - INTRODUÇÃO

O aprimoramento da qualidade do software requer não apenas a melhoria contínua dos processos, mas também dos artefatos produzidos ao longo do desenvolvimento. Entre os artefatos elaborados no desenvolvimento do software, os testes de unidade[1] desempenham um papel importante na identificação de erros no código de software (GRANO, 2019a). Portanto, garantir a qualidade desses testes é crucial, pois eles possibilitam a detecção precoce de problemas de qualidade. No entanto, a presença de *Test Smells* (TS) pode comprometer a capacidade dos testes na identificação de erros no software destacando, assim, a necessidade de identificar e mitigar tais problemas para assegurar a robustez e a confiabilidade do conjunto de testes.

Os TS[2] são problemas nos testes que surgem de decisões de design inadequadas implementadas no código de teste. Esses problemas podem ter um impacto significativo na capacidade dos testes em detectar erros no software e na capacidade dos desenvolvedores de manter o código de teste (GAROUSI et al., 2019; SPADINI et al., 2020). Os problemas de qualidade causados pelos *smells* afetam negativamente a compreensão e a manutenção dos testes (BAVOTA et al., 2015), que são artefatos importantes na detecção precoce de problemas de qualidade de software.

Os *smells* são indicadores ou sintomas de problemas de *design* mais graves (YAMASHITA, 2014; SOUSA, 2016) criados a partir de soluções ruins (KHOMH et al., 2011; ARNAOUDOVA et al., 2013) ou que violam as boas práticas de um determinado domínio (SHARMA et al., 2016).

A presença de *smells* em uma classe de testes prejudica a evolução do software, impactando a qualidade do produto (BOWES et al., 2017; BLESER et al.,

[1] A partir deste ponto do documento, o termo teste será usado no lugar de teste de unidade, incluindo sua escrita no plural. Nos locais onde for necessário, o tipo do teste será escrito para fins de compreensão.

[2] A partir deste ponto do documento, o termo *smell* será usado no lugar de *test smell*, incluindo a escrita no plural, excetuando-se os casos nos quais a utilização não é considerada adequada por comprometer o entendimento do texto.

2019; ALJEDAANI et al., 2021), contribuindo para reduzir a produtividade dos desenvolvedores (BAVOTA et al., 2012) e aumentando o tempo necessário para entender os programas (SPADINI et al., 2018; GAROUSI; KÜÇÜK, 2018) e a possibilidade de falhas (QUSEF et al., 2019). Além disso, a presença de *smells* impacta outros aspectos da qualidade do código, como a arquitetura do software (GARCIA et al., 2009), o código-fonte do software (ARNAOUDOVA et al., 2013), o desempenho do software (SHARMA; ANWER, 2014; WANG et al., 2014), o consumo de energia (VETRO et al., 2013), a usabilidade (ALMEIDA et al.; 2015), entre outros.

A refatoração do código de teste é uma possível solução para lidar com os impactos negativos na qualidade que a presença de *smells* causa. As atividades de refatoração são benéficas porque a refatoração de testes remove falhas de projeto e ajudam a corrigir outros problemas (PALOMBA e ZAIDMAN, 2017). Para refatorar um teste, os desenvolvedores de software podem usar ferramentas automatizadas ou semiautomatizadas para ajudar nas atividades de refatoração de *smells* ou executar a refatoração manual. As ferramentas semiautomatizadas são abordagens humano dependentes, contando com a iniciativa dos desenvolvedores de software para executar a ferramenta e aprovar a refatoração (MARTINEZ et al., 2020; SANTANA et al., 2020; LAMBIASE et al., 2020). Por outro lado, a refatoração manual é prejudicada por problemas como a falta de desenvolvedores qualificados nas organizações (SAMARTHYAM et al., 2017), o que prejudica a melhoria da qualidade dos testes. Outro problema é a negligência na resolução de problemas de teste (DAKA; FRASER, 2014; KIM et al., 2021) aliada à pressão para realizar a entrega de funcionalidades (SAMARTHYAM et al., 2017), o que contribui para a permanência de *smells* no repositório de software e a deterioração da qualidade dos testes. Mesmo quando os desenvolvedores estão cientes da presença de *smells* nos testes, a remoção manual de *smells* geralmente é um efeito secundário das atividades de manutenção e evolução do software (PERUMA et al., 2020a; ALOMAR et al., 2021; HABCHI et al., 2021).

As atividades de refatoração não são suficientes para remover os *smells*, pois a quantidade de *smells* no repositório aumenta à medida que o software envelhece, prejudicando ainda mais a qualidade dos testes (QUSEF et al., 2019). Nesse contexto, a refatoração manual tem mais testes para refatorar, levando a um esforço maior nas atividades de refatoração e, consequentemente, menos recursos disponíveis para manter e evoluir o software. Ao mesmo tempo, os problemas de qualidade continuam

a prejudicar o processo de desenvolvimento. No caso das abordagens semiautomáticas, os problemas que afetam esse tipo de abordagem estão relacionados às dependências humanas, como a refatoração manual, mesmo que em menor gravidade.

Algumas pesquisas têm apresentado propostas para auxiliar no processo de melhoria da qualidade dos testes, fornecendo ferramentas e/ou métodos para a detecção de *smells* (PALOMBA et al., 2018; BLESER et al., 2019; PERUMA et al., 2020a; VIRGÍNIO et al., 2020; FERNANDES et al., 2022; WANG et al., 2022; PAUL et al., 2024; YANG et al., 2024) e para a refatoração de testes (LAMBIASE et al., 2021; MARTINEZ et al., 2021; SANTANA et al., 2021); No entanto, o mapeamento sistemático conduzido por (ALJEDAANI et al., 2021) mostra que há poucos estudos dedicados à melhoria da qualidade dos testes por meio da remoção de *smells*, indicando que há maior ênfase em pesquisas relacionadas à detecção de *smells* em comparação com a refatoração. No estudo de Aljedaani et al., (2021), os autores sugerem a necessidade de investigações futuras voltadas para a refatoração de *smells*, visando assim uma abordagem mais abrangente e eficaz para aprimorar a qualidade dos testes de software.

Conforme será discutido na Seção 2.8, as abordagens atuais para identificação de *smells* são predominantemente semiautomatizadas, exigindo que o desenvolvedor execute um *plugin* ou uma ferramenta e tome decisões quanto à refatoração sugerida, o que ainda está sujeito aos desafios enfrentados pelas abordagens manuais. No entanto, as ferramentas de refatoração automáticas destinadas à remoção de *smells* representam uma alternativa promissora, capaz de contribuir significativamente para melhorar a qualidade do software sem a necessidade de intervenção humana.

Independentemente da abordagem escolhida para melhorar a qualidade dos testes, a remoção de *smells* dos testes de unidade por meio da refatoração deve levar em consideração o comportamento do teste (BLADEL e DEMEYER, 2018; TRAN et al., 2021). Analisar o comportamento do teste é essencial para identificar aspectos relacionados aos estados dos objetos do software sendo testados, como ocorre a interação dos testes com esses objetos e como os próprios objetos interagem entre si. Esta análise é fundamental, uma vez que os desenvolvedores elaboram conjuntos de testes para verificar automaticamente se o software atende aos comportamentos esperados (XUAN et al., 2016), ajudando assim a evitar erros nos testes que poderiam

impactar negativamente o software*,* ao mesmo tempo em que preservam os cenários de teste.

Para mitigar os impactos negativos na qualidade do software causados pela presença de *smells* em classes de testes, bem como enfrentar os desafios associados à refatoração manual e semiautomatizada, esta pesquisa busca responder à seguinte questão: Como apoiar a equipe de desenvolvimento na melhoria da qualidade dos testes de unidade por meio da remoção de *Test Smells*?

Ao investigar essa questão, o estudo pretende não apenas avaliar o impacto das refatorações no cotidiano da equipe de desenvolvimento e na melhoria contínua dos testes, mas também analisar como a refatoração automática de *smells* pode impactar a qualidade desses testes. Além disso, pretende-se explicitar como as remoções de *smells* podem ser realizadas por meio da refatoração automática.

1.1 Objetivos

O objetivo geral desta pesquisa é desenvolver um processo para automatizar a remoção de ocorrências de *smells* de classes de testes de unidade.

Para isso, foram definidos os seguintes objetivos específicos:

I. Determinar critérios para priorizar a remoção automática das ocorrências de TS identificadas;
II. Elaborar um processo para remoção automatizada de *smells*;
III. Avaliar o processo proposto.

1.2 Delimitação de escopo

A implementação do refatorador para o processo proposto é limitado à refatoração de softwares orientados a objetos escritos na linguagem Java, que usem o *framework* JUnit 5 para a execução de testes de unidade e o aplicativo Apache Maven nas etapas de compilação e construção do software.

Como a pesquisa está relacionada à remoção de *smells* por meio da refatoração automática, somente os testes de unidade serão considerados. Por isso, outros tipos de testes, como os de integração ou de *interface*, não estão sendo considerados, ou, quando presente no conjunto de testes a serem refatorados, serão considerados como testes de unidade. A possível falta de documentação sobre o domínio do software para permitir a identificação de elementos como requisitos funcionais/não requisitos e diagramas, também faz com que outros tipos de testes,

mesmo que com maior importância, não sejam considerados na refatoração automática.

Embora a literatura sobre Dívida Técnica (DT) descreva etapas como identificação, priorização, gerenciamento e pagamento da DT, não serão propostas ferramentas para identificar, priorizar e gerenciar a dívida técnica. A identificação da DT não faz parte do escopo desta pesquisa, pois pretende-se incorporar abordagens existentes na literatura para identificar as ocorrências de *smells* nos repositórios do software.

1.3 Estrutura do documento

Este documento está organizado da seguinte forma:

- No CAPÍTULO 1 são apresentados os conceitos relacionados aos testes de unidade e *Test Smells*, fornecendo uma base para situar a presente pesquisa em relação ao estado da arte. Além disso, neste capítulo, são apresentados a questão de pesquisa que norteia o estudo, os objetivos definidos e as limitações da proposta que delineiam o escopo e as restrições da pesquisa.
- No CAPÍTULO 2 é conduzida uma revisão da literatura, explorando estudos relacionados ao teste de software, qualidade de software e avaliação da qualidade dos testes. Além disso, são examinados os *Test Smells*, incluindo suas causas e consequências, além de abordar aspectos relacionados com a refatoração de código. Por fim, são apresentados os trabalhos relacionados organizados de acordo com o artefato utilizado em cada estudo e ordenados em uma progressão crescente de similaridade com o presente trabalho.
- No CAPÍTULO 3 é delineado o posicionamento metodológico da pesquisa, fundamentando-se nas atividades da *Design Science Research* (DSR). Essa abordagem metodológica proporciona um arcabouço robusto para o desenvolvimento e a validação de soluções inovadoras em ambientes de pesquisa aplicada, como é o caso desta investigação.
- No CAPÍTULO 4 é detalhado um método para a remoção do *Eager Test* s*mell* que foi implementado em uma ferramenta de refatoração automática de testes de unidade. O capítulo também descreve dois estudos exploratórios realizados com a ferramenta de refatoração automática

desenvolvida para avaliar a significância da ocorrência de falhas e o impacto no tempo de execução dos testes refatorados, bem como para a identificação das causas de falhas e erros dos testes refatorados para remover ocorrências de *smells*. No primeiro estudo exploratório, somente o *Eager Test* foi removido automaticamente, enquanto no segundo o *Eager Test* e o *Test Code Duplication* foram removidos automaticamente.

- No CAPÍTULO 5 é apresentado o processo elaborado para a remoção automática de *smells* das classes de testes de unidade, bem como aspectos relacionados com a atualização da ferramenta de refatoração desenvolvida para aumentar a quantidade de *smells* removidos automaticamente, o qual foi denominado Sentinel. Além disso, são apresentados dois quase-experimentos realizados e um *survey*. O primeiro quase-experimento teve por objetivo avaliar a viabilidade da remoção automática de *smells* por meio da identificação automática de problemas causados pela refatoração. O segundo quase-experimento teve por objetivo identificar a variação na ocorrência de *smells* após a remoção automática dos *smells* removidos automaticamente. O objetivo do *survey* foi identificar aspectos relacionados à percepção de ocorrência e de impacto da presença de *smells* nas classes de testes, bem como aspectos relacionados com a prioridade da remoção de *smells* de uma classe de testes.
- No CAPÍTULO 6 é descrita a avaliação do processo especificado, realizada junto a especialistas em *smells,* desenvolvedores e analistas de qualidade da indústria. Além disso, são discutidos os resultados da avaliação e são apresentadas as modificações realizadas no processo Sentinel após a avaliação, bem como na árvore de decisão para a realização das refatorações e nas condições a serem consideradas na priorização das refatorações, conforme as interações definidas na DSR.
- O CAPÍTULO 7 conclui o trabalho, resumindo a sua relevância, contribuições e limitações. Além disso, propõe trabalhos futuros de pesquisa.

1.4 Considerações sobre o capítulo

Neste capítulo foram apresentados conceitos relacionados sobre *Test Smells*, seguido de uma breve descrição dos problemas causados pela presença de *smells*

no código dos testes. Em seguida, foram apresentadas as abordagens manual e semiautomatizada de remoção de *smells*, ressaltando os problemas que podem prejudicar a adoção de cada abordagem. Além disso, ressaltou-se a importância da identificação e manutenção do comportamento do software no processo de refatoração para que o cenário de testes possa ser mantido.

Por fim, foram apresentados o objetivo principal e os objetivos específicos, seguido da delimitação do escopo da pesquisa e da estrutura deste documento de tese.

CAPÍTULO 2 - REVISÃO DA LITERATURA

Esse capítulo apresenta a revisão da literatura, explorando aspectos relacionados ao teste de software, qualidade de teste, atributos de teste e métricas de qualidade. Além disso, aborda o problema dos Test Smells, destacando suas causas, consequências e possíveis soluções para sua correção. No final do capítulo são apresentados os trabalhos relacionados a esta pesquisa.

2.1 Teste de software

O processo de desenvolvimento de software está sujeito à ocorrência de problemas que afetam a qualidade do produto. A equipe de desenvolvimento pode usar testes para avaliar a qualidade do produto e evitar que os efeitos nocivos dos problemas de qualidade afetem os usuários finais. Um teste é definido pela ISO/IEC 24765:2010 como "uma atividade na qual um software ou componente é executado sob condições especificadas, os resultados são observados ou registrados e é feita uma avaliação de algum aspecto do software ou componente" (ISO/IEC 24765, 2010) e pode ser usado para revelar falhas o mais rápido possível.

Os testes de unidade são projetados para verificar o funcionamento isolado de elementos de software testáveis separadamente (AMMANN e OFFUT, 2016). No desenvolvimento de testes de unidade, a equipe define casos de teste em um conjunto de testes para avaliar a qualidade do produto. Os casos de teste são desenvolvidos para verificar automaticamente se o software atende aos comportamentos esperados (XUAN et al., 2016). Um caso de teste é definido como "um conjunto de entradas de teste, condições de execução e saídas esperadas, desenvolvido para um objetivo específico, como exercitar um determinado caminho em um programa ou verificar um determinado requisito" (ISO/IEC 24765, 2010).

O teste de unidade é uma prática comum em que os desenvolvedores escrevem casos de teste junto com o código regular (DAKA e FRASER, 2014). Uma ideia fundamental do teste de unidade é que cada parte do código precisa de seus próprios testes (WANG e OFFUTT, 2009). Para executar um teste de unidade em um caso de teste, o software é colocado em um estado e a unidade em teste é estimulada

de modo que o resultado do estímulo possa ser analisado para identificar se o comportamento do software está de acordo com o esperado. As etapas para a execução de testes de unidade foram descritas por Rompaey et al., (2007) e são mostradas na Figura 2-1, sendo compostas por quatro etapas, descritas a seguir:

- Configurar (*setup*): os recursos necessários para realizar um teste são adquiridos;
- Estimular (*stimulate*): um estímulo é enviado do código de teste para a unidade em teste;
- Verificar (*verify*): o código de teste consulta a unidade em teste, obtém o resultado do estímulo e informa o resultado do teste ao testador;
- Destruir ou desmontar (*teardown*): os recursos alocados no teste são liberados.

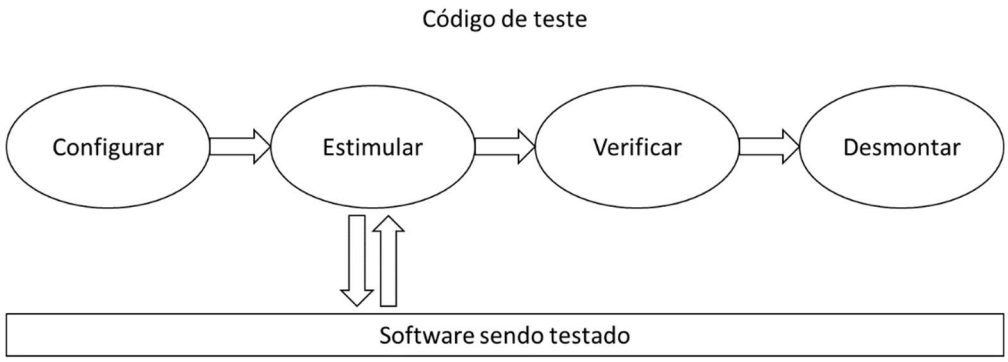

Figura 2-1 Etapas para a execução de testes de unidade (traduzido de Rompaey et al., 2007)

As etapas descritas por Rompaey et al., (2007) são uma referência de como os testes de unidade são projetados e executados, onde os objetos do software são criados na etapa de configuração e os dados necessários para executar o teste são carregados. Em seguida, a etapa de estímulo consiste na execução de um ou mais métodos do software que, a partir do estado inicial, podem produzir alterações no estado do software. Após o estímulo, a etapa de verificação consiste em analisar o resultado do estímulo realizado para identificar se o comportamento do software é adequado para o caso de teste. Entretanto, é importante notar que os recursos alocados no teste de unidade podem ser liberados automaticamente, dependendo da linguagem de programação, sem que exista a necessidade de que o programador implemente qualquer código relacionado à liberação dos recursos alocados.

Embora as etapas sejam apresentadas sequencialmente por Rompaey et al., (2007), os testes podem pular ou executar as etapas alternadamente. Por exemplo, um teste pode pular a etapa de estímulo, testando o(s) estado(s) do(s) recurso(s) criado(s) na etapa de configuração antes da realização dos estímulos.

Em relação à etapa configurar, a criação de instâncias de objetos para execução dos testes pode ocorrer por meio de três formas (Meszaros, 2007):

- *inline*: o código de configuração é inserido no código do método de teste;
- implícita: a configuração está em um método declarado na classe de teste denominado método de configuração;
- delegada utiliza classes auxiliares à classe de teste, separando os detalhes de configurações mais complexas da classe de teste.

No caso da *inline*, a reutilização de código de configuração é dificultada, pois o código de configuração está contido no método de teste podendo causar a duplicação de código em dois ou mais métodos de testes. No caso da implícita e da delegada, o código de configuração pode ser reutilizado devido à separação do código de configuração do código de estímulo e de verificação (MESZAROS, 2007; PERUMA et al, 2019; SILVA e VILAIN, 2020), o que pode favorecer a manutenção e aumentar a reutilização.

2.2 Qualidade de testes

A qualidade do software está relacionada à qualidade do processo de desenvolvimento e dos artefatos relacionados ao desenvolvimento, como documentação, código fonte, código de testes, entre outros. De acordo com a NBR ISO/IEC 25010, "a qualidade de um sistema é o grau em que o sistema satisfaz as necessidades declaradas e implícitas de suas várias partes interessadas e, portanto, fornece valor". A qualidade do software é essencial porque não apenas o software é usado em diversas áreas de vários aplicativos, mas também porque vários eventos históricos indicaram o impacto das falhas de software em todo o mundo (BARRAOOD et al., 2018).

Os atributos de qualidade de software são categorizados em dois tipos: atributos internos e externos. De acordo com Kaur (2020), os atributos internos de qualidade, como acoplamento, herança, coesão, tamanho e complexidade, podem ser medidos usando apenas artefatos de código, ao passo que os atributos externos de qualidade, como legibilidade, capacidade de manutenção, capacidade de teste,

capacidade de alteração etc., não podem ser medidos usando apenas artefatos de código.

A norma NBR ISO/IEC 25000 define oito características de qualidade de produto de alto nível, que são: adequação funcional, eficiência de desempenho, compatibilidade, usabilidade, confiabilidade, segurança, capacidade de manutenção e portabilidade.

De acordo com a ISO/IEC 25010 (2011), a capacidade de manutenção é o grau de eficácia e eficiência com o qual um produto de software pode ser modificado pelos mantenedores e é composto pelos seguintes atributos:

- Modularidade: grau em que um software é composto de componentes discretos, de modo que uma alteração em um componente tem impacto mínimo em outros componentes;
- Reutilização: grau em que um ativo pode ser usado em mais de um software ou na criação de outros ativos;
- Analisabilidade: grau de eficácia e eficiência com o qual é possível avaliar o impacto em um produto ou software de uma alteração pretendida em uma ou mais partes, ou diagnosticar deficiências ou causas de falhas em um produto, ou identificar peças a serem modificadas;
- Modificabilidade: grau em que um produto ou um software pode ser modificado de forma eficaz e eficiente sem introduzir defeitos ou degradar a qualidade do produto existente;
- Testabilidade: grau de eficácia e eficiência com o qual os critérios de teste podem ser estabelecidos para um software, produto ou componente e os testes podem ser realizados para determinar se esses critérios foram atendidos.

Como os testes são essenciais para avaliar a qualidade do software (GRANO et al., 2020), a qualidade dos artefatos relacionados aos testes se torna ainda mais importante, pois as consequências das falhas de software podem resultar em perdas financeiras e humanas (BARRAOOD et al., 2018). Além das consequências das perdas humanas e financeiras, os artefatos de teste de baixa qualidade afetam a evolução e a manutenção do software devido aos requisitos de qualidade semelhantes para manutenção e código de produção (ATHANASIOU et al., 2014).

A similaridade dos requisitos de qualidade entre o código de teste e do software decorre do fato de que o código de teste é escrito em uma linguagem de programação regular para estimular um software, observar seu comportamento e as saídas produzidas (GAROUSI et al., 2015). Com isso, o aumento da qualidade do código de teste pode aumentar o desempenho das equipes de desenvolvimento na correção de *bugs* e na implementação de novos recursos (ATHANASIOU et al., 2014). Além disso, após o desenvolvimento inicial, como qualquer artefato de engenharia de software, o código de teste requer avaliação e manutenção de qualidade (GAROUSI et al., 2015).

Para melhorar a qualidade dos testes, Bowes et al., (2017) afirmaram que o desenvolvimento de testes deve seguir um conjunto de princípios:

- Simplicidade: significa que um teste é mantido simples, com menos chance de cometer erros, e é mais fácil manter a base de testes quando evitamos a complexidade;
- Legibilidade e compreensão: os testes devem ter uma compreensão clara das intenções do código de teste, evitando números mágicos, ramificações e convenções de nomes inexpressivas;
- Responsabilidade única: impor a análise de uma condição por método de teste para facilitar a localização da causa raiz da falha de um teste;
- Evitar excesso de proteção: evitar muitas afirmações ou vários tipos de declarações de afirmação aparecendo juntos, o que pode estar relacionado ao excesso de proteção e pode levar a afirmações redundantes no código de teste;
- Testar o comportamento (não a implementação): manter o foco no comportamento esperado e utilizar a interface pública da classe em teste evitaria problemas relacionados ao código obsoleto e à necessidade de atualizar o código para a nova implementação do software sendo testado (SUT, do inglês *System Under Test*);
- Manutenibilidade (refatoração do código de teste): à medida que o código de teste evolui com o código de produção, evite a duplicação do código de teste e aplique a refatoração para manter os testes simples e ajudar nas atividades de manutenção;

- Um teste deve falhar: um teste deve ter pelo menos uma instrução assert, que não seja assertTrue(true), e um método de teste incompleto, ou seja, não devidamente implementado, deve conter a instrução fail();
- Confiabilidade: os testes devem se comportar como esperado, ou seja, devem passar ou falhar consistentemente em condições determinadas;
- Testes felizes vs. tristes: os testadores devem evitar o fenômeno conhecido como viés de confirmação, quando os testadores tendem a confirmar o comportamento do teste em vez de quebrá-lo;
- Os testes não devem ditar o código: a lógica de teste deve ser separada do código de produção. Portanto, evite códigos de produção que sejam chamados somente a partir do código de teste ou alterações no código de produção que permitam que parte do código seja chamada a partir do código de teste;
- *Feedback* rápido: falhas lógicas na implementação do código de teste ou de produção que introduzam complexidade indesejada ou vazamentos de memória. Além disso, minimize o uso de recursos externos no código de teste para fornecer *feedback* oportuno sobre os resultados dos testes;
- Design de teste de unidade de quatro fases: todo teste deve usar as fases de configuração, estímulo, verificação e desmontagem, mesmo que a execução esteja incorporada no dispositivo;
- Simplicidade dos dispositivos: os dispositivos devem ser mínimos, evitando dispositivos genéricos cujas partes são usadas por um pequeno número de testes;
- (In)dependência de testes: um teste não deve invocar outro teste, mantendo os testes isolados uns dos outros;
- Uso de duplas de teste: use *stub*/*mock* caso ainda não haja recursos externos usados nos testes para evitar atrasos no tempo de execução na conclusão dos testes.

Os princípios descritos por Bowes et al., (2017) podem ser usados para nortear o desenvolvimento de testes, evitando problemas durante as atividades de manutenção, pois as taxas de probabilidade de erro aumentam significativamente devido à ocorrência de antipadrões e clones (JAAFAR et al., 2017). A avaliação da

qualidade do teste, que será discutida na próxima seção, pode ser realizada medindo os atributos do código de teste e por meio da participação da equipe de desenvolvimento.

2.3 Avaliação de qualidade dos testes

De acordo com a ISO/IEC 25010 (2011), "as propriedades mensuráveis relacionadas à qualidade de um sistema são chamadas de propriedades de qualidade, com medidas de qualidade associadas". Para chegar a medidas de características ou subcaracterísticas de qualidade, é necessário identificar um conjunto de propriedades que, juntas, abrangem a característica, obter medidas de qualidade para cada uma delas e combiná-las computacionalmente para chegar a uma medida de qualidade derivada correspondente à característica de qualidade (ISO 25010, 2011).

A medida das características ou subcaracterísticas de qualidade faz parte da avaliação da qualidade do teste. A avaliação da qualidade do teste está relacionada à aspectos da qualidade do código de teste, como integridade, eficácia e capacidade de manutenção (ATHANASIOU et al., 2014). A avaliação da qualidade do teste pode ser realizada usando métricas relacionadas às propriedades internas e externas do teste.

As propriedades de qualidade interna, tais como acoplamento, herança, coesão, tamanho e complexidade, podem ser medidas usando apenas artefatos de código, enquanto as propriedades de qualidade externa, tais como legibilidade, capacidade de manutenção, capacidade de teste, capacidade de mudança etc., não podem ser medidas usando apenas artefatos de código (KAUR, 2020).

A avaliação das propriedades de qualidade externas é afetada por aspectos subjacentes do código, como conhecimento, experiência, tempo disponível e intuição (BAKOTA et al, 2011). Os desenvolvedores concordam, em sua maioria, com um subconjunto de aspectos que são desejáveis (por exemplo, bons identificadores, legibilidade e presença de comentários) ou indesejáveis (por exemplo, muitos pontos de saída) para ter um código compreensível (SCALABRINO et al., 2021).

Os aspectos de avaliação internos e externos da qualidade do código de teste podem ser relacionados aos fatores que afetam a compreensibilidade do código descritas por Börstler e Paech (2016) que descrevem fatores que influenciam na compreensão do código. Conforme mostra a Figura 2-2, Börstler e Paech (2016) descrevem os seguintes fatores que impactam na compreensão do código: pessoais, de software, de projeto e cognitivos. Os fatores pessoais compreendem aspectos

como fluência na linguagem de programação, habilidade/experiência em programação, conhecimento de domínio do software e estresse/motivação, enquanto os fatores de software compreendem o tamanho e a legibilidade do código, a complexidade dos componentes do código, e a coerência que representa a aplicação de padrões e práticas de desenvolvimento de software. Os fatores cognitivos são derivados de teorias cognitivas ou teorias de compreensão de programas e os fatores de projeto descrevem elementos do ambiente do projeto que podem facilitar a compreensão do código.

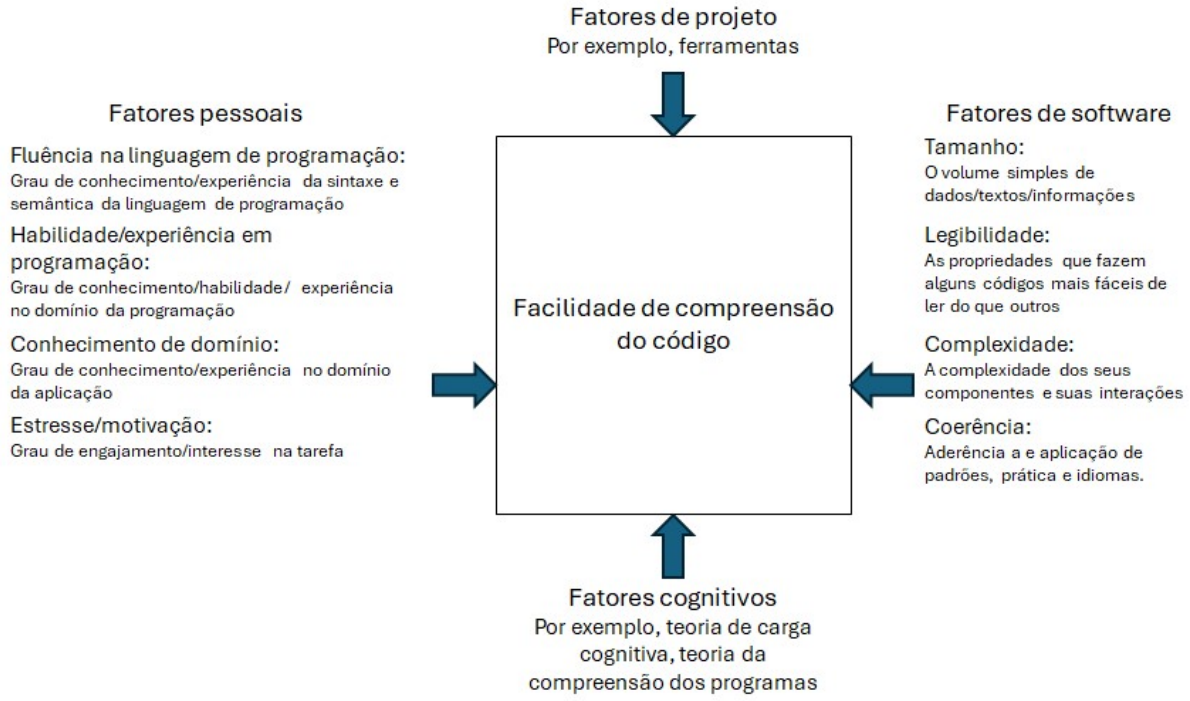

Figura 2-2 Fatores que afetam a facilidade de compreensão do código (Fonte: traduzido de Börstler e Paech, 2016)

A legibilidade é um fator que afeta a manutenção porque influencia a capacidade de compreensão de um trecho de código. Para manter um código, a primeira etapa é a leitura do código, que é uma das atividades mais frequentes na manutenção de software, pois, antes de implementar alterações, é necessário entender completamente o código escrito (SCALABRINO, 2016, SETIANI et al., 2020).

A legibilidade é importante para os desenvolvedores diagnosticarem o resultado dos casos de teste e conduzirem as atividades de depuração (GRANO et al., 2020) e pode ser uma barreira que precisa ser superada pelo programador para entender um código (POSNETT et al., 2011). Ainda, a legibilidade pode ser avaliada medindo-se os atributos internos de qualidade do código-fonte (KAUR, 2020). Quando

somadas a fatores cognitivos, pessoais e de projeto, a legibilidade influencia a compreensibilidade e, consequentemente, a capacidade de manutenção de um código.

Assim, para avaliar a compreensibilidade, é necessário integrar outras formas de avaliação que não sejam baseadas apenas em métricas relacionadas ao código do teste. A avaliação de atributos como legibilidade, compreensibilidade e capacidade de manutenção é discutida a seguir.

2.3.1 Avaliação de legibilidade

De acordo com Posnett et al., (2011), a legibilidade é uma impressão subjetiva que os programadores têm da dificuldade de leitura do código, à medida que tentam entendê-lo. A legibilidade do código é fundamental para a compreensão do código e pode aumentar o esforço de compreensão do programa (SCALABRINO, 2016), pois um trecho de código pode ser fácil de ler, mas difícil de entender por causa dos aspectos semânticos e cognitivos relacionados às tarefas de compreensão (ALQADI e MALETIC, 2020).

A legibilidade pode ser avaliada principalmente em fatores locais considerando métricas de complexidade relacionadas ao tamanho das classes e métodos e à extensão de suas interações (BUSE e WEIMER, 2010). Buse e Weimer (2010) mostraram que há uma correlação significativa entre legibilidade e qualidade ao extrair um conjunto de métricas de partes de código anotado para treinar um algoritmo de classificação. A Tabela 2-1 mostra o conjunto de recursos extraídos por análises sintáticas do código-fonte e usados que medem aspectos relacionados ao tamanho do código, à complexidade e à estrutura de um código (BUSE e WEIMER, 2010).

Tabela 2-1 Conjunto de recursos considerados por Buse e Weimer (Fonte: traduzido de Buse e Weimer, 2010)

Nome do recurso	Considera a média de ocorrências?	Considera o máximo de ocorrências?
Comprimento da linha (quantidade de caracteres)	Sim	Sim
Quantidade de identificadores	Sim	Sim
Comprimento do identificador	Sim	Sim

Recuo	Sim	Sim
Quantidade de chaves	Sim	Sim
Quantidade de comentários	Sim	Sim
Quantidade de períodos	Sim	
Quantidade de vírgulas	Sim	
Quantidade de espaços	Sim	
Quantidade de parênteses	Sim	
Quantidade de operadores aritméticos	Sim	
Quantidade de operadores de comparação	Sim	
Quantidade de atribuições	Sim	
Quantidade de ramos	Sim	
Quantidade de rotações	Sim	
Quantidade de linhas em branco	Sim	
Quantidade de ocorrências de qualquer caractere único		Sim
Quantidade de ocorrências de qualquer identificador único		Sim

O tamanho é um indicador significativo de legibilidade e deve ser incluído em qualquer modelo que busque entender quais fatores afetam a legibilidade (POSNETT et al., 2011). Posnett et al., (2011) criaram um modelo para avaliar a legibilidade usando as métricas de Buse e Weimer (2010), adicionando medidas lexicais de Maurice Howard Halstead e entropia de termos.

As medidas lexicais de Halstead são calculadas da seguinte forma (HALSTEAD, 1977):

- O comprimento do programa (N) é a soma do número total de operadores (N1) e operandos (N2): $N = N1 + N2$;
- O vocabulário do programa é a soma do número de operadores únicos (n1) e operandos únicos (n2): $n = n1 + n2$;
- Volume é o comprimento do programa (N) vezes o log2 do vocabulário do programa (n): $V = N\ log2\ n$;

- A dificuldade é metade dos operadores únicos multiplicados pelo número total de operandos, dividido pelo número de operadores distintos: $\mathrm{D} = \frac{n1}{2} * \frac{N2}{2}$;
- O esforço (E) é a dificuldade (D) multiplicada pelo volume (V): $E = D * V$.

A medida de volume Halstead visa medir o conteúdo informativo do código-fonte, combinando contagens totais com contagens únicas (POSNETT et al., 2011). De acordo com Postnett et al., (2011), a entropia é frequentemente vista como o grau de desordem ou a quantidade de informações em um sinal ou conjunto de dados e é calculada a partir da contagem de termos (*tokens* ou *bytes*), bem como do número de termos e *bytes* exclusivos. De acordo com o modelo especificado por Posnett et al., (2011), dado um documento X, x_i é um termo em X, *count*(x_i) é a quantidade de ocorrências de x_i no documento X e $p(x_i)$ é calculado conforme mostrado em A) na Figura 2-3 e a entropia é calculada conforme mostrado em B) na mesma figura.

A)

$$p(x_i) = \frac{count(x_i)}{\sum_{j=1}^{n} count(x_j)}$$

B)

$$H(X) = -\sum_{i=1}^{n} p(x_i) log_2 p(x_i)$$

Figura 2-3 Fórmula de cálculo de $p(x_i)$ e da entropia (Fonte: Posnett et al., 2011)

O modelo de Posnett et al., (2011) foi usado por Lee et al., (2015) para avaliar a legibilidade do código de produção. Lee et al., (2015) exploraram 210 projetos Java de código aberto do site sourceforge.net, coletando arquivos de código-fonte e metadados sobre a experiência do desenvolvedor, o tamanho da equipe, o tamanho do projeto e a maturidade do projeto. Os autores identificaram que a adesão ou violação das convenções de codificação afeta a legibilidade do código e que existem violações de codificação específicas que têm uma influência relativamente mais forte da legibilidade do código, tais como violações relacionadas aos comentários finais, ao Javadoc e à endentação (LEE et al., 2015). Além disso, os fatores que podem influenciar a extensão das violações de codificação e prejudicar a legibilidade do código estão relacionadas com a experiência do desenvolvedor, o tamanho da equipe, o tamanho do projeto e a maturidade do projeto (LEE et al., 2015).

2.3.2 Avaliação de compreensibilidade

A medição da compreensibilidade é frequentemente associada a critérios de legibilidade porque se supõe que um código fácil de ler será fácil de entender (SETIANI et al., 2020). Por outro lado, a avaliação da compreensibilidade do software é complexa devido à necessidade de considerar aspectos externos ao código-fonte (LIN e WU, 2006), pois a experiência profissional e a experiência em linguagem específica são pontos chave para identificar o quanto um trecho de código é compreensível (TROCKMAN et al., 2018). Assim, enquanto a legibilidade pode ser avaliada usando aspectos internos do código, a avaliação da compreensibilidade precisa considerar o contexto no qual o código avaliado está inserido.

Embora o código legível possa estar relacionado à sua compreensão pelo desenvolvedor, as métricas de legibilidade do código não são suficientes para prever se os desenvolvedores podem compreender a saída do programa, as relações entre as entidades do código, a semântica e a estrutura do código (SETIANI et al., 2020). A compreensibilidade do código só pode ser avaliada a partir da compreensão do contexto do software (KAUR, 2020).

Scalabrino et al., (2017) realizaram um estudo envolvendo 46 participantes para avaliar a correlação das 121 métricas com a compreensibilidade dos trechos de código. Os autores usaram métricas preditoras candidatas para a compreensibilidade do código, agregando medidas de Buse e Weimer (2010), Posnett et al., (2011) e Scalabrino et al., (2016), adicionando recursos relacionados ao código-fonte, à documentação e aos desenvolvedores. Depois de realizar um estudo para avaliar métricas de compreensibilidade de código, Scalabrino et al., (2017) mostraram que nenhuma das métricas que os autores consideraram pode capturar a compreensibilidade do código, nem mesmo as que supostamente avaliam atributos de qualidade fortemente relacionados a ela, como legibilidade e complexidade do código. A complexidade de medir a compreensibilidade sugere que é necessário incluir fatores cognitivos, de design e mais pessoas no processo de avaliação.

Trockman et al., (2018) continuaram o estudo de Scalabrino et al., (2017) combinando métricas do conjunto de dados anterior em recursos para avaliar se a combinação de métricas melhoraria o desempenho dos modelos estatísticos na avaliação da compreensibilidade de um código. De acordo com Trockman et al., (2018), a maioria desses recursos foi calculada a partir da estrutura sintática do

código, tais como: a quantidade de linhas, a quantidade de *tokens*, a quantidade de variáveis etc.

Alguns recursos foram computados a partir da qualidade da documentação do código, por exemplo, prevalência de comentários ou de documentação interna para os métodos usados (TROCKMAN et al., 2018). Além disso, a documentação deve ser considerada com cuidado no processo de avaliação, pois pode estar relacionada a outros assuntos que não a explicação do código ou até mesmo desatualizada. A principal descoberta de Trockman et al., (2018) foi que ser profissional aumenta as chances de compreensão em um fator de 2,75, sinalizando que o fator pessoa (ou seja, nível de experiência) é mais importante na avaliação da compreensibilidade do que fatores relacionados ao software, como o tamanho e a legibilidade do código-fonte.

No entanto, "na maioria das vezes, as métricas de qualidade consideradas não foram capazes de capturar a melhoria da qualidade conforme percebida pelos desenvolvedores" (PANTIUCHINA et al., 2018). Mesmo a combinação de métricas pode não ser adequada para medir a compreensibilidade do código na prática, porque a compreensibilidade depende mais do desenvolvedor do que do trecho que ele está avaliando (SCALABRINO et al., 2021).

2.3.3 Avaliação de manutenibilidade

A capacidade de manutenção está entre os atributos de qualidade externa mais amplamente investigados (KAUR, 2020). Para avaliar a capacidade de manutenção, os autores podem usar métricas relacionadas a softwares orientados a objetos, questionários ou uma combinação de ambos. Por exemplo, Deligiannis et al., (2003) extraíram um conjunto de métricas de um software e combinaram os dados coletados com as respostas de um questionário para analisar a dificuldade de realizar atividades de manutenção em um código.

As métricas baseadas na estrutura do código-fonte foram criadas para avaliar a capacidade de manutenção. Essas métricas estão relacionadas à aspectos estruturais de classes (DELIGIANNIS et al., 2003; BENESTAD et al., 2006; KÁDÁR et al., 2016), linhas de código e quantidade de classes (BRUNTINK e VAN DEURSEN, 2006), conjunto de métricas relacionadas à orientação a objetos, como acoplamento e coesão (ELISH; ALSHAYEB 2009), combinação de métricas, como estruturais e

relacionadas à orientação a objetos (BÁN; FERENC, 2014; BÁN, 2017; PANTIUCHINA et al., 2018).

A análise automática do código-fonte e a extração de métricas levaram a modelos de manutenção baseados em métricas (BÁN, 2017). As primeiras etapas na avaliação da capacidade de manutenção de um código foram realizadas por McCabe (MCCABE, 1976) com a publicação de uma métrica que avalia a complexidade ciclomática do código-fonte com base no número de possíveis fluxos de execução do software.

Coleman et al., (1994) definiram o Índice de Manutenibilidade (MI, do inglês *maintainability index*), uma fórmula que considera as linhas de código-fonte, a complexidade ciclomática e o volume de Halstead para medir a facilidade de suporte e alteração do código-fonte.

Chidamber e Kemerer (1994) propuseram seis métricas para orientação a objetos (OO): Métodos ponderados por classe (WMC), Profundidade da árvore de herança (DIT), Número de filhos (NOC), Acoplamento entre classes de objetos (CBO), Resposta para uma classe (RFC), Falta de coesão nos métodos (LCOM), sendo, respectivamente, traduções do inglês para *Weighted Methods per Class*, *Depth of Inheritance Tree*, *Number of Children*, *Coupling between Object Classes*, *Response for a Class* e *Lack of Cohesion of Methods*. As métricas de coesão de classe são preditoras úteis para a capacidade de manutenção da classe (ALZAHRANI et al., 2019), porque o baixo acoplamento e a complexidade e a alta coesão são desejados para melhorar a capacidade de manutenção (PANTIUCHINA et al., 2018). Como o baixo acoplamento e a complexidade levam à fácil modificação do código-fonte, a alta coesão melhora a reutilização.

As métricas criadas na década de 1990 continuaram a ser usadas, em sua forma original ou adaptadas, nos últimos anos. Kádár et al., (2016) usaram métricas relacionadas a tamanho, como *Total Logical Lines of Code* (TLLOC) e *Total Number of Statements* (TNOS), acoplamento, como RFC e *Number of Outgoing Invocations* (NOI), quantidade de *clones* e complexidade (WMC) para avaliar os efeitos da refatoração na capacidade de manutenção. As descobertas de Kádár et al., (2016) indicam que as classes com baixa capacidade de manutenção estão sujeitas a um número maior de refatorações durante sua vida útil.

A métrica Lack of Cohesion of Methods (LCOM) (CHIDAMBER e KEMERER, 1994) foi adaptada por Alzahrani et al., (2019) para criar a métrica chamada Client-

based class cohesion (CCC). A nova métrica é usada para medir a coesão da classe com base no uso dos atributos ou métodos da classe pelos clientes da classe (ALZAHRANI et al., 2019). Para avaliar a qualidade dos casos de teste, Setiani et al., (2020) descreveram 19 métricas de qualidade, como a contagem do número de construtores, a contagem do número de identificadores exclusivos, a contagem do número de loops, a contagem do número de ramificações e outras.

Até mesmo modelos recentes que empregam algoritmos de aprendizado de máquina (ML, do inglês machine learning) usam métricas clássicas da literatura para criar recursos nos algoritmos para prever a capacidade de manutenção ou falhas de projeto (JHA et al., 2019; SCHNAPPINGER et al., 2019; THONGKUM, MEKRUKSAVANICH, 2020; GUPTA, SHUG, 2021).

As abordagens automáticas atuais foram criticadas porque seus resultados muitas vezes não refletem a opinião de especialistas ou são tendenciosos para um pequeno grupo de especialistas (SCHNAPPINGER et al., 2020). Além disso, assim como a compreensibilidade, a capacidade de manutenção é afetada por fatores externos que não podem ser avaliados apenas pela análise do código-fonte (MISRA, 2005, PANTIUCHINA et al., 2018).

No entanto, os especialistas podem não estar disponíveis para inspecionar e avaliar a capacidade de manutenção de um código porque a experiência, o conhecimento do domínio do aplicativo e as habilidades da linguagem de programação podem contribuir para que as avaliações realizadas pelos profissionais produzam resultados diferentes.

Assim, a medição de atributos externos de qualidade pode envolver o uso de métricas específicas ou a integração da equipe de desenvolvimento no processo de avaliação por meio da interação com artefatos de teste. Nos casos em que as informações disponíveis para medir os atributos externos de qualidade são insuficientes, a medição dos atributos externos de qualidade pode ser feita por meio de modelos ou fórmulas derivadas dos atributos internos como substitutos (KAUR, 2020).

Uma lista de trabalhos encontrados na literatura sobre *smells* e seus respectivos conceitos é apresentada a seguir, seguida de uma discussão sobre as causas e consequências da presença de *smells* no código de teste de unidade.

2.4 Test Smells

Os TS são escolhas de *design* aplicadas pelos desenvolvedores ao implementar casos de teste (PALOMBA et al., 2018) e há autores que definem uma ou mais *smells* em seu trabalho, conforme mostrado a seguir.

A primeira descrição de *smells* foi dada por van Deursen et al., (2001), que descreveram uma lista de onze TS, como segue:

- Mystery Guest (MG): um teste que utiliza recursos externos, como um banco de dados, que contém dados de teste;
- Resource Optimism (RO): o código de teste que faz suposições otimistas sobre a existência (ou ausência) e o estado de recursos externos (como diretórios específicos ou tabelas de banco de dados) pode causar um comportamento não determinístico nos resultados do teste;
- Test Run War (TRW): essas guerras surgem quando os testes são executados sem problemas, desde que você seja o único a testar, mas falham quando mais programadores os executam;
- General Fixture (GF): o método de configuração é muito genérico e os diferentes testes acessam apenas parte do método de configuração;
- Eager Test (ET): quando um método de teste testa vários métodos do objeto a ser testado, sendo difícil de ler e entender e, portanto, mais difícil de usar como documentação;
- Lazy Test (LT): ocorre quando vários métodos de teste testam a mesma unidade, mas, por exemplo, usam valores de diferentes variáveis de instância;
- Assertion Roulette (AR): um método de teste tem várias asserções não documentadas;
- Indirect Test (ID): uma classe de teste contém métodos que realmente executam testes em outros objetos que não são sua contraparte;
- For Testers Only (FTO): quando uma classe de produção contém métodos que são usados somente por métodos de teste, esses métodos (1) não são necessários e podem ser removidos ou (2) são necessários somente para configurar um dispositivo para teste;
- Sensitive Equality (SE): o método *toString* é usado em um método de teste;
- Test Code Duplication (TCD, ou clones): o código de teste pode conter uma duplicação indesejável.

Um exemplo de ocorrência do Eager Test é mostrado na Figura 2-4, na qual é possível identificar que o software é estimulado nas linhas 74, 76, 79, 81, entre outras, por meio da chamada do método add(). Após cada estímulo, um teste é realizado na sequência (linhas 75, 77, 80 e 82, entre outras). Exemplos de ocorrências de outros smells são apresentados no Apêndice C.

```
67      @Test
68      public void testEquals() {
69          BorderArrangement b1 = new BorderArrangement();
70          BorderArrangement b2 = new BorderArrangement();
71          assertTrue( condition: b1.equals( obj: b2));
72          assertTrue( condition: b2.equals( obj: b1));
73
74          b1.add(new EmptyBlock( width: 99.0,  height: 99.0),  key: null);
75          assertFalse( condition: b1.equals( obj: b2));
76          b2.add(new EmptyBlock( width: 99.0,  height: 99.0),  key: null);
77          assertTrue( condition: b1.equals( obj: b2));
78
79          b1.add(new EmptyBlock( width: 1.0,  height: 1.0),  key: RectangleEdge.LEFT);
80          assertFalse( condition: b1.equals( obj: b2));
81          b2.add(new EmptyBlock( width: 1.0,  height: 1.0),  key: RectangleEdge.LEFT);
82          assertTrue( condition: b1.equals( obj: b2));
83
84          b1.add(new EmptyBlock( width: 2.0,  height: 2.0),  key: RectangleEdge.RIGHT);
85          assertFalse( condition: b1.equals( obj: b2));
86          b2.add(new EmptyBlock( width: 2.0,  height: 2.0),  key: RectangleEdge.RIGHT);
87          assertTrue( condition: b1.equals( obj: b2));
88
89          b1.add(new EmptyBlock( width: 3.0,  height: 3.0),  key: RectangleEdge.TOP);
90          assertFalse( condition: b1.equals( obj: b2));
91          b2.add(new EmptyBlock( width: 3.0,  height: 3.0),  key: RectangleEdge.TOP);
92          assertTrue( condition: b1.equals( obj: b2));
```

Figura 2-4 Exemplo de ocorrência do smell Eager Test

Outra descrição de *smells* foi descrito por Breugelmans e van Rompaey (2008), como segue:

- Assertionless (AL): um teste que atua para afirmar dados e funcionalidades, mas não o faz;
- Duplicated Code (DC): um método de teste que possui redundância no código;
- Indented Test (InT): Um método de teste que contém muitos pontos de decisão, loops e instruções condicionais;
- Verbose Test (VT): teste código complexo e não simples ou limpo.

O *smell* denominado *Rotten Green Tests* (testes verdes podres) foi descrito por Delplanque et al., (2019) para indicar testes que são executados sem erros, mas que têm asserções que não são executadas. Esse *smell* ocorre quando a seleção e a repetição de declarações não conseguem realizar o teste lógico, impedindo que a asserção seja executada.

Outra descrição de *smells*, descrita por Peruma et al., (2019), é um conjunto de doze TS, como segue:

- Conditional Test Logic (CTL): o método de teste contém várias instruções de controle;
- Constructor Initialization (CI): o conjunto de testes não deve ter um construtor;
- Default Test (DfT): os testes são criados no modelo de teste gerado automaticamente pelo ambiente de programação;
- Duplicate Assert (DA): um método de teste testa a mesma condição várias vezes, dentro do mesmo método de teste;
- Empty Test (EmT): um método de teste não tem instruções executáveis;
- Exception Handling (EH ou ECT, de Exception Catching Throwing): a aprovação ou reprovação de um método de teste depende explicitamente de o método de produção lançar uma exceção;
- Ignored Test (IT): o teste não será executado devido à anotação @Ignore;
- Magic Number Test (MNT): um método de teste contém literais numéricos inexplicáveis e não documentados como parâmetros ou como valores para identificadores;
- Redundant Print (RP): as instruções de impressão em testes de unidade são redundantes, pois os testes de unidade são executados como parte de um script automatizado;
- Redundant Assertion (RA): os métodos de teste contêm declarações de asserção que são sempre verdadeiras ou sempre falsas;
- Sleepy Test (ST): um método de teste contém instruções de pausa de thread para atrasar a execução;
- Unknown Test (UT): método de teste sem uma declaração de asserção e nome não descritivo.

Como cada TS tem características específicas, as causas e consequências comuns do TS são discutidas a seguir.

2.5 Causas de Test Smells

As causas do *smells* decorrem de vários aspectos. Wiklund et al., (2012) afirmam que as regras para o desenvolvimento de testes são potencialmente menos rigorosas do que as aplicadas ao restante do código-fonte. Outra causa de *smells* é a

falta de padrões de codificação de testes automatizados, conforme descrito por Samarthyam et al., (2017) como um fator que aumenta o esforço para manter o código de teste. Assim, a ausência de parâmetros orientadores e limitadores aumenta o número de possibilidades de criação de testes. Daka e Fraser (2014) identificaram as ações tomadas pela equipe quando houve falhas na execução dos testes. De acordo com os dados coletados na pesquisa realizada pelos autores, em 15,6% das falhas ocorridas, os desenvolvedores eliminam ou comentam sobre o teste falho, enquanto em 11,2% a falha é ignorada pela equipe (DAKA e FRASER, 2014). Embora Daka e Fraser (2014) não tenham qualificado a ação como *smells*, quando a perspectiva humana que causa *smells* é considerada, o comportamento de negligência da equipe pode dar origem ao *smell* de Empty Test (EmT), no qual um método de teste não contém declarações executáveis, ou Unknown Test (UT), no qual não existe ao menos uma afirmação para realizar o teste. Um possível motivo para a negligência da qualidade dos testes é a pressão do cronograma, apontada por Samarthyam et al., (2017) como pressão para entregar valor rapidamente às partes interessadas, em detrimento à qualidade do código de teste de unidade desenvolvido.

A qualidade do código do software pode estar associada à presença de *smells* no código do teste. Por exemplo, Tufano et al., (2016) confirmaram essa possibilidade, identificando que o Eager Test em classes de teste está frequentemente associado a *smells* de código no código de produção, especificamente: Class Data Should Be Private (CDSBP), Spaghetti Code (SC) e Blob Class (BC). De acordo com os autores, é razoável pensar que, ao testar classes grandes e complexas, os desenvolvedores tendem a criar métodos de teste mais complexos, exercitando vários métodos da classe testada (TUFANO et al., 2016).

Outra causa relacionada à perspectiva humana para as causas de *smells* é o viés de confirmação no ciclo de vida de desenvolvimento de software (CVDS), relatado por Calikli e Bener (2015), no qual os desenvolvedores podem preferir apenas os testes de unidade para fazer seu código ser executado em vez de testes de unidade que visam fazer seu código falhar. Calikli e Bener (2015) identificaram que os testadores têm uma predisposição para criar testes que confirmem o comportamento do SUT.

Para evitar o viés de confirmação, o desenvolvimento de testes deve considerar ambos os pontos de vista para equilibrar as tentativas de quebrar e confirmar o comportamento do SUT (BOWES et al., 2017). Uma causa humana relacionada ao

TS é a falta de conhecimento sobre os problemas de qualidade dos testes (CAMPOS et al., 2021). A falta de conhecimento sobre TS torna os desenvolvedores incapazes de identificar TS. Consequentemente, mesmo que a refatoração seja realizada, existe a possibilidade de que um novo TS seja inserido no teste (PALOMBA; ZAIDMAN, 2017; CAMPOS et al., 2021) e que eles prejudiquem a qualidade do código de teste.

As ferramentas de geração automática de testes (FGAT) podem ser mencionadas em termos dos aspectos tecnológicos que causam o TS. De acordo com Grano et al., (2019b), *smells* são difundidos em casos de teste gerados automaticamente por todas as ferramentas experimentadas: 81%, 92% e 98% das classes JUnit geradas pelo EvoSuite, JTExpert e Randoop, respectivamente, podem ser consideradas como sendo afetadas por *smells*. Os resultados publicados por Grano et al., (2019b) foram corroborados por Virgínio et al., (2020), que identificaram que as classes de testes geradas automaticamente apresentaram *smells*. Embora as causas do FGAT sejam problemas de qualidade no código-fonte, esses problemas podem ser identificados e o processo de geração de testes pode ser aprimorado.

2.6 Consequências da presença de Test Smells

As consequências da presença de *smells* estão relacionadas a problemas de qualidade que impactam tanto as atividades de manutenção quanto aquelas que comprometem a qualidade do código do software.

O impacto na compreensão e na manutenção do software foi comprovado por Bavota et al., (2015), que identificaram que os *smells* têm um forte impacto negativo na compreensão e na manutenção dos testes. Os *smells* Eager Test, Sensitive Equality e Test Code Duplication "podem afetar a capacidade de manutenção do teste de unidade, mas não afetam a qualidade do código de produção" (QUSEF et al., 2019). Portanto, embora não afete diretamente o código do software, um desenvolvedor com dificuldade de entender os testes pode inserir erros no código de teste, afetando indiretamente a qualidade do software. Outro impacto negativo da presença de TS é que o código do software é mais propenso a falhar quando testado por um teste que tem algum *smell* (SPADINI et al., 2018, QUSEF et al., 2019).

As falhas do software podem se originar de problemas de qualidade do teste que impedem que os problemas do SUT sejam identificados corretamente. De acordo com Qusef et al., (2019), Mystery Guest e General Fixture implicam um número maior de problemas, como vazamentos de memória no código de produção, e Assertion

Roulette é acompanhado por um aumento no comportamento funcional inesperado no código de produção em teste.

2.7 Refatoração

A refatoração de código consiste em reestruturar um código-fonte para melhorar o seu *design*, sem alterar o seu comportamento externo (FOWLER, 2018), ou seja, alterar as instruções que compõem um trecho de código de modo que o resultado do código refatorado seja o mesmo do código não refatorado preservando, assim, o comportamento do código. A refatoração automática consiste em alterar as estruturas de um código reduzindo necessidade de intervenção humana no processo, desde a detecção de oportunidades de refatoração até a identificação de erros e a modificação do código.

A refatoração usada para remover *smells* pode variar de acordo com as características de cada *smell*. Por exemplo, a refatoração do tipo AR pode ser realizada por meio da inclusão de um parâmetro que serve como explicação do motivo do erro no teste (SANTANA et al., 2020). No caso de ET, DA e CD, a refatoração é realizada extraindo os trechos de código que causam o *smell*. Com isso, são apresentadas algumas refatorações descritas por Fowler (2018) que podem ser usadas na refatoração de testes de unidade:

- Combinar funções: agrupar métodos que tenham alguma semelhança em uma classe, como acesso a dados;
- Descer atributo: mover um atributo para uma subclasse da classe em que ele é declarado;
- Descer função: mover um método para uma subclasse da classe em que foi declarado;
- Remover código morto: remover código não utilizado em classes que prejudicam a compreensão do programa;
- Encapsular variável: alterar o modificador de acesso de um atributo para privado e criar *getters* e *setters* para permitir o acesso externo;
- Extrair classe: criar uma classe a partir de um conjunto de métodos e atributos de uma classe específica;
- Extrair função: criar um método para um trecho específico de código que faz parte de um método existente;

- Extrair superclasse: criar uma superclasse a partir de um conjunto de métodos e atributos de uma classe específica. A classe da qual os métodos e atributos foram extraídos estenderá a superclasse criada.
- Alterar declaração de função: modificar a assinatura de um método de classe, como modificador de acesso, tipo de retorno, nome, parâmetros, anotações ou lançamento de exceção.

2.8 Trabalhos relacionados

Esta seção tem como objetivo apresentar os trabalhos relacionados a esta pesquisa de acordo com o artefato manipulado em cada proposta: casos de teste ou código de teste. Em cada categoria, os trabalhos relacionados são apresentados em ordem crescente de similaridade com esta proposta, destacando as diferenças.

2.8.1 Casos de teste

Hauptmann et al., (2015) apresentaram uma ferramenta para gerar automaticamente propostas de refatoração de casos de teste para ajudar os testadores a entenderem as relações entre os clones e refatorar o conjunto de testes para aumentar a reutilização dos testes. O fluxo de trabalho de refatoração pode ser visto na Figura 2-5, na qual a ferramenta (I) gera a proposta de refatoração, que é (II) compreendida pelo engenheiro de testes e que (III) refatorará o conjunto de testes. Para identificar clones, os autores consideraram um clone como uma sequência consecutiva de pelo menos três etapas dentro de um procedimento de teste que aparece pelo menos duas vezes em um conjunto de testes (HAUPTMANN et al., 2015). A abordagem proposta pelos autores utiliza a modelagem de sequência, o subcampo de inferência de gramática que examina uma única sequência de tokens e forma um modelo dela, como, por exemplo, uma máquina de estado ou gramática (HAUPTMANN et al., 2015). A avaliação da proposta dos autores foi realizada por um engenheiro de testes da empresa Munich Re, que avaliou as propostas de refatoração geradas. Os resultados obtidos pelos autores indicam que, das dezoito propostas de refatoração, em apenas duas os benefícios não justificariam o esforço de refatoração. Apesar de produzir benefícios para a remoção de clones no conjunto de testes, a abordagem proposta por Hauptmann et al., (2015) não utiliza o código de teste como entrada para a geração de propostas. Hauptmann et al., (2015) usaram técnicas

baseadas em processamento de texto que não puderam ser aplicadas no código de teste.

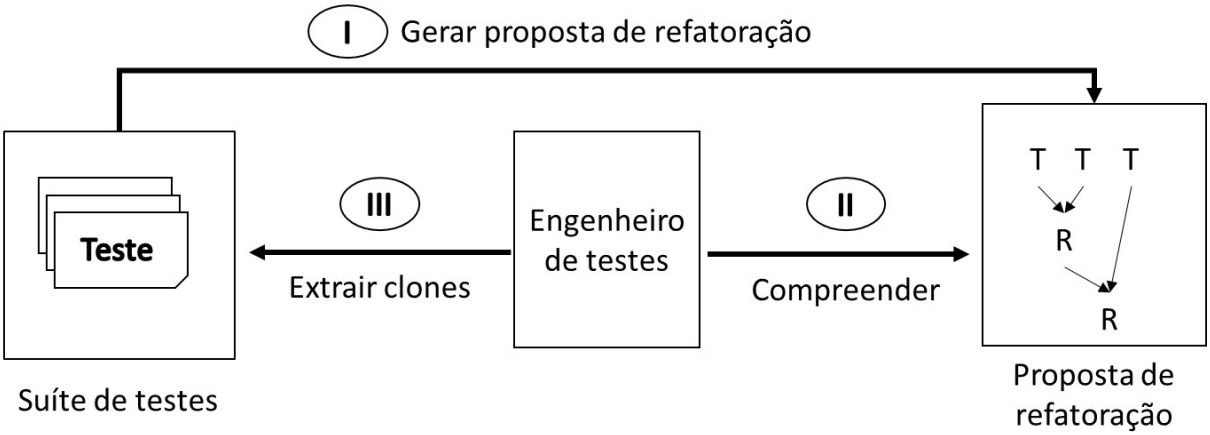

Figura 2-5 Fluxo de trabalho proposto por Hauptmann et al., (2015) (Fonte: traduzido de Hauptmann et al., 2015)

A proposta de Bernard et al., (2020) usa técnicas de processamento de linguagem natural para dar suporte à refatoração manual do conjunto de testes. O objetivo dos autores era melhorar a usabilidade e a capacidade de manutenção dos testes. As técnicas usadas pelos autores compreendem: a) o agrupamento de testes por escopo funcional; b) refatoração, para mesclar "etapas semelhantes tornando-as idênticas e introduzindo etapas parametrizadas para capturar etapas que são idênticas, exceto pelos valores dos dados" (BERNARD et al., 2020). Embora os resultados variem entre os diferentes experimentos, os autores relataram uma redução média de 14% no número de casos de teste e uma redução estimada do tempo de refatoração em uma média de 40% com a ferramenta em comparação com a refatoração manual (BERNARD et al., 2020). O agrupamento de testes é uma etapa necessária para a refatoração, pois permite agrupar os testes de acordo com o contexto de teste, ou seja, o caso de teste. A partir do agrupamento, a refatoração pode identificar quais casos de teste são compatíveis. Embora a proposta de Bernard et al., (2020) tenha como objetivo melhorar a qualidade dos testes, o método de refatoração usado pelos autores (algoritmos de processamento de linguagem natural) não pode ser usado com o código de teste.

2.8.2 Código de teste

Martinez et al., (2020) apresentaram o RTj, um *framework* que analisa casos de teste de projetos Java com o objetivo de detectar casos de teste podres. O RTj é um software que detecta o TS por meio de instrumentação e execução de testes para identificar as asserções que não são executadas devido a erros nas instruções de seleção (instruções *if*). A Figura 2-6 mostra o algoritmo desenvolvido para identificar e analisar testes de um repositório. A proposta de Martinez et al., (2020) foi avaliada em 67 projetos Java de código aberto, detectando 418 testes verdes podres em 26 projetos. A proposta de Martinez et al., (2020) não refatora automaticamente os testes, limitando-se a relatar os testes com problemas. A refatoração descrita pelos autores consiste em duas ações: 1) o analisador de falhas ausentes propõe uma refatoração que substitui as asserções forçadas a falhar e 2) adicionar um comentário TODO logo antes do elemento de código podre (MARTINEZ et al., 2020).

Entrada: programa sob análise P

Saída: Rótulos e testes refatorados

```
1:   M ← criarModeloDoPrograma(P)
2:   T ← encontrarCasosDeTeste(P, M)
3:   Dit ← executarTeste(T)
4:   rotulos ← Ø, refatoradores ← Ø
5:   estados ← Ø, resultadoDinamico ← Ø
6:   para t_i em T faça
7:     para a_i em AnalisadoresDeTestes faça
8:           s_i = a_i.encontreElementos(M, estados, t_i)
9:           estados = estados U < a_i , s_i >
10:          d_i = a_i.analiseDinamica(M, Dit, resultadoDinamico, s_i, t_i)
11:          resultadoDinamico = resultadoDinamico U < a_i , d_i >
12:          l_i = ai.testaRotulo(M, s_i, d_i, t_i)
13:          rotulos = rotulos U < t_i , l_i >
14:          t' = a_i.aplicaRefatoracao(M, s_i, d_i, t_i)
15:          se t' não é nulo entao
16:                refatoradores = refatoradores U < t_i , t' >
     retorne rotulos, refatoradores
```

Figura 2-6 Algoritmo para detecção e refatoração de testes verdes podres (Fonte: traduzido de Martinez et al., 2020)

Santana et al., (2020) propuseram a ferramenta RAIDE para auxiliar os testadores com um ambiente para detecção automatizada de linhas de código afetadas por *smells*, bem como refatoração semiautomatizada para projetos Java usando o *framework* JUnit (SANTANA et al., 2020). A ferramenta proposta pelos autores consiste em um *plugin* capaz de detectar e refatorar TS Duplicated Assert e Assertion Roulette. Conforme mostrado na Figura 2-7, o desenvolvedor precisa executar o *plugin* selecionando o tipo de *smell* a ser identificado. Depois de selecionar o tipo de *smell*, a ferramenta analisa as classes de teste e identifica se elas são afetadas pelo *smell* selecionado, propondo a refatoração que o desenvolvedor avalia e aprova. A refatoração do TS Assertion Roulette é realizada adicionando o texto "*Add Assertion Explanation Here*" para explicar a asserção. Os autores não mencionaram

que a ferramenta proposta analisa a ordem de execução do teste para garantir que a ordem original seja mantida após a refatoração.

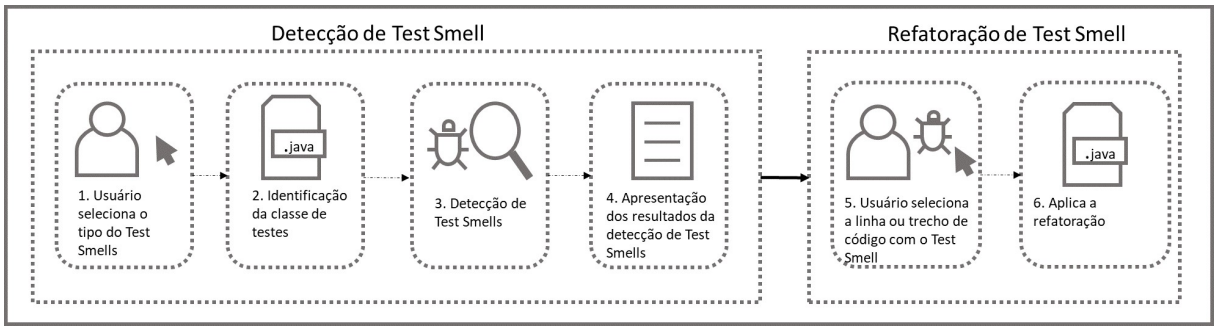

Figura 2-7 Fluxo de trabalho do RAIDE (Fonte: traduzido de Santana et al., 2020)

Lambiase et al., (2020) apresentaram um *plugin* para identificar e refatorar três TS: General Fixture, Eager Test e Lack of Cohesion of Test Methods. O *plugin* funciona no nível do *commit*, alertando o desenvolvedor sobre a presença de TS no código que está sendo enviado ao repositório. Apesar de a refatoração ser automática, a abordagem proposta pelos autores é dependente do ser humano, pois depende de o desenvolvedor ler o alerta do *plugin*, entender o impacto de cada *smell* na qualidade do código, concordar com a refatoração proposta e aplicá-la.

A capacidade de identificação do Eager Test é uma similaridade entre esta tese e a proposta de Lambiase et al., (2020). Com relação à análise do comportamento do teste, os autores argumentam que o DARTS implementa ações de refatoração que devem apenas melhorar os aspectos de qualidade do código de teste sem alterar a forma como ele exerce uma classe de produção (LAMBIASE et al., 2020). Os autores não descrevem o que o *plugin* faz para preservar o comportamento do teste ou como o comportamento é identificado. Além disso, a análise do código-fonte do projeto DARTS permite identificar que a ferramenta proposta extrai apenas o código que leva o Eager Test a um novo método de teste, sem identificar, por exemplo, a ordem de execução dos métodos do teste e como os métodos dos testes extraídos serão executados. A ferramenta de identificação de TS usada por Lambiase et al., (2020) baseia-se na comparação textual da similaridade dos nomes dos métodos executados com um limite de 0,6.

A Tabela 2-2 apresenta a comparação desta proposta de tese com estudos publicados relacionados à refatoração não manual de testes. As propostas de Santana et al., (2020), Lambiase et al., (2020) e Martinez et al., (2020) empregam seus próprios

algoritmos, enquanto esta tese propõe a integração com ferramentas de identificação de *smells* existentes na literatura e a validação automática das refatorações realizadas. Apenas o trabalho de Martinez et al., (2020) usa instrumentação de código para rastrear a execução do SUT. Diferentemente dos outros trabalhos, a identificação do comportamento dos testes e do SUT será realizada por meio da análise estática e dinâmica do código de teste e do SUT, minimizando as chances de que dados ocultos possam interferir na identificação do comportamento. Além disso, esta tese remove automaticamente os *smells* AR, ECT, ET, RA e TCD, enquanto as abordagens existentes na literatura exigem a intervenção do engenheiro de software em algum ponto do processo. A refatoração automática proposta nesta tese e apresentada na seção 5.2 é baseada na instrumentação das classes de produção para identificação do comportamento do software antes e depois das refatorações.

Tabela 2-2 Comparação de trabalhos relacionados com esta tese (Fonte: o autor)

Artefato	Autor (es)	Identificação / Refatoração	Test Smell	Usa instrumentação de código?	Validação da refatoração	Recursos/técnicas utilizadas
Caso de teste	Hauptmann et al., 2015	Reestruturação	NA*	NA*	NA*	Métricas e algoritmos próprios
	Bernard et al., 2020	Reestruturação	NA*	NA*	NA*	Processamento de linguagem natural
Código de teste	Martinez et al., (2021)	Identificação	Rotten green test	Sim	NA*	Aprendizado de máquina
	Santana et al., (2021)	Refatoração (semiautomática)	Assertion roulette Duplicate assertion	Não	Humano-dependente	Análise sintática e dinâmica Algoritmo próprio
	Lambiase et al., (2021)	Refatoração (semiautomática)	General Fixture Eager Test Lack of cohesion	NI**	NI**	Algoritmo próprio
	Este trabalho	Refatoração (automatizada)	Assertion Roulette Exception Catching Throwing Eager Test Redundant Assert Test Code Duplication	Sim	Automática	Análise sintática e dinâmica Algoritmo próprio Refatorações

*NA significa “não aplicável”

**NI significa “não identificado/informado”

2.9 Considerações sobre o capítulo

Este capítulo apresentou uma fundamentação teórica sobre a qualidade dos testes e a avaliação da qualidade. Uma breve lista de TS foi apresentada, discutindo as causas e consequências comuns na qualidade do produto e no processo de desenvolvimento de software.

Além disso, foram abordados os trabalhos relacionados a esta pesquisa, organizados de acordo com os artefatos usados em cada trabalho. Inicialmente, foram apresentados os trabalhos que empregam casos de teste como artefato. Em seguida, foram apresentados os trabalhos relacionados que utilizam o código de teste de unidade como artefato na proposição de ações de melhoria da qualidade por meio da identificação ou remoção semiautomatizada de *smells*.

Por fim, todos os trabalhos foram agrupados em uma tabela, oferecendo uma descrição detalhada das semelhanças e diferenças entre elas.

CAPÍTULO 3 - MÉTODO DE PESQUISA

Este capítulo descreve o método de pesquisa adotado para a execução das atividades desta tese, iniciando com a caracterização da pesquisa, seguido das atividades da *Design Science Research* (DSR).

O projeto de pesquisa foi aprovado pelo Comitê de Ética em Pesquisa (CEP) da Pontifícia Universidade Católica do Paraná (PUCPR) sob Parecer Nº 6.134.910. Conforme descrito no projeto de pesquisa, os dados coletados foram usados unicamente para fins de pesquisa, sendo tratados em forma anônima e agrupada, mantendo a proteção do anonimato dos participantes.

Os documentos relacionados ao projeto de pesquisa estão disponíveis nos Apêndices desta tese, sendo eles: o questionário sobre a priorização da remoção de *smells* (Apêndice A), o convite para participação (Apêndice B), o material de apoio para os participantes (Apêndice C), o Termo de Confidencialidade de Uso de Dados (TCUD) (Apêndice D), o Termo de Consentimento Livre e Esclarecido (TCLE) (Apêndice E). O TCLE foi assinado digitalmente pelos participantes e o TCUD foi assinado digitalmente pela equipe de pesquisa. O projeto de pesquisa pode ser consultado no site da Plataforma Brasil em: https://plataformabrasil.saude.gov.br.

3.1 Caracterização da pesquisa

Gil (2002) define pesquisa como um procedimento racional e sistemático que tem por objetivo proporcionar respostas aos problemas propostos. A caracterização da pesquisa é necessária para situá-la dentro do universo de possíveis métodos, técnicas e procedimentos científicos disponíveis ao pesquisador.

Este trabalho é caracterizado como pesquisa exploratória (GIL, 2002), pois tem por objetivo gerar conhecimento sobre a remoção automática de *smells* pode apoiar a equipe de desenvolvimento na melhoria da qualidade dos testes.

Com relação aos objetivos, esta pesquisa é classificada como exploratória. De acordo com Gil (2002), a pesquisa exploratória tem como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo explícito ou a constituir hipóteses (GIL, 2002). Além disso, a pesquisa exploratória pode servir de base para outros tipos

de investigação, pois os resultados da pesquisa exploratória podem ser explicados em pesquisas teóricas e informar os pesquisadores experimentais sobre os efeitos para os quais os desenhos experimentais devem controlar (BRIGGS; SCHWABE, 2011).

No que se refere à coleta de dados, a presente pesquisa pode ser classificada como quantitativa e qualitativa. Os dados necessários para avaliar a abordagem proposta foram coletados por meio da execução da ferramenta de refatoração desenvolvida e por meio da coleta de dados realizada com especialistas em Test Smells e com membros das equipes de desenvolvimento.

3.2 Método de pesquisa

Com relação aos procedimentos metodológicos, o método de pesquisa escolhido para realizar esta pesquisa foi o Design Science Research Methodology (DSRM) (PEFFERS et al., 2007). O paradigma do *design* científico busca ampliar os limites das capacidades humanas e organizacionais por meio da criação de artefatos novos e inovadores (HEVNER et al., 2004).

O DSRM foi proposto para incorporar princípios, práticas e procedimentos necessários à realização de pesquisas aplicadas envolvendo o desenvolvimento de artefatos. Conforme mostrado na Figura 3-1 e detalhado a seguir, o DSRM inclui seis etapas (PEFFERS et al., 2007): identificação e motivação do problema, definição dos objetivos de uma solução, projeto e desenvolvimento, demonstração, avaliação e comunicação.

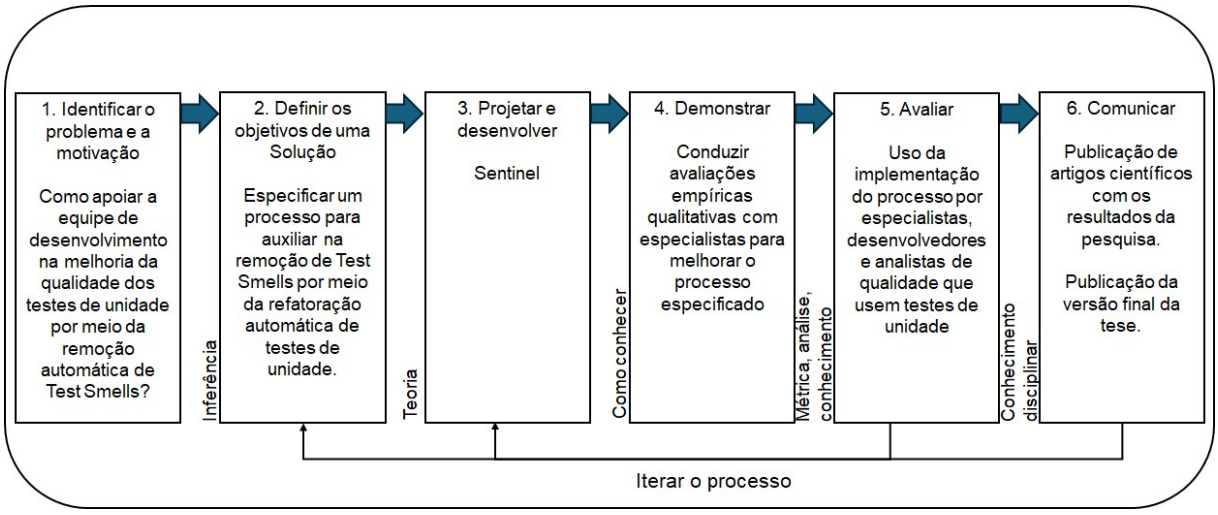

Figura 3-1 Modelo de processo DSRM (Fonte: Peffers et al., 2007)

3.2.1 Definir o problema e a motivação

A motivação para a pesquisa relacionada à remoção automática de TS ocorreu a partir do estudo de quatro artigos: Garousi e Küçük (2018), Bladel e Demeyer (2018), Aljedaani et al., (2021) e Tran et al., (2021).

O artigo publicado por Garousi e Küçük (2018), incentiva os pesquisadores a projetar e propor novas métricas para prevenção, detecção e/ou correção de *smells*. O artigo de Bladel e Demeyer (2018), argumenta que há necessidade de suporte a ferramentas que possam analisar se um conjunto de testes refatorado preserva seu comportamento pré e pós-refatoração. O artigo publicado por Aljedaani et al., (2021), sugeriu a expansão da detecção de *smells* de teste para a refatoração interativa em um estudo de mapeamento sistemático sobre ferramentas de detecção de TS. O artigo publicado por Tran et al., (2021) apresenta uma revisão sistemática da literatura para identificar e analisar estudos secundários existentes sobre aspectos de qualidade de artefatos de teste de software. Esse artigo ajudou a identificar estudos primários e secundários relacionados e avaliações de métricas para aferir aspectos relacionados ao código do software.

3.2.2 Definir os objetivos da solução

Segundo Hevner et al., (2004), a pesquisa seguindo o método DSR deve produzir um artefato criado para resolver um problema. Os objetivos quantitativos e/ou qualitativos do artefato devem ser inferidos racionalmente a partir da especificação do problema e do conhecimento do que é possível e viável (Peffers et al., 2007).

Assim, o objetivo dessa tese é propor um processo para a remoção automática de ocorrências de *smells* a partir da integração com uma ferramenta de identificação de *smells* descrita na literatura. O processo proposto busca realizar a refatoração sem a necessidade de análise e aprovação das refatorações realizadas, ou seja, sem a intervenção de membros da equipe de desenvolvimento na análise e verificação de erros causados pela refatoração. Para realizar a refatoração automática, foram selecionados *smells* que afetam diferentes partes do código do teste: Eager Test, Test Code Duplication (clones), Duplicate Assert, Assertion Roulette e Exception Catching Throwing.

3.2.3 Projeto e desenvolvimento

A terceira atividade da DSR consiste na criação de artefatos que, de acordo com Peffers et al., (2007) "são potencialmente constructos, modelos, métodos ou instanciações". A contribuição da pesquisa está presente no artefato produzido, incluindo aspectos como funcionalidades e arquitetura (PEFFERS et al., 2007).

O desenvolvimento do artefato foi realizado em duas etapas: a especificação de um método para a remoção do *Eager Test smell* e a especificação de um processo de remoção automática de s*mells*. A especificação da versão final do processo, apresentado na seção 6.5, foi resultado da realização de uma sequência de estudos e mudanças no refatorador que são apresentados nas seções 4.2, 4.3, 5.2, 5.3, 5.4, e 5.5 e 6.3.

A especificação do método de remoção do *Eager Test* é descrita no CAPÍTULO 4 -, e correspondeu ao desenvolvimento e avaliação de uma ferramenta de refatoração automática. A avaliação da ferramenta desenvolvida nesta etapa foi realizada por meio da realização de dois estudos exploratórios realizados para identificar o impacto da refatoração em aspectos como tempo de execução dos testes refatorados e erros causados pela refatoração.

A especificação do processo de remoção automática de *smells* é descrita no CAPÍTULO 5 - e refere-se à elaboração do processo denominado Sentinel, nome que foi usado para a ferramenta de refatoração automática desenvolvida. A ferramenta Sentinel foi atualizada na segunda etapa para realizar a validação automática das refatorações por meio da identificação de erros de compilação, erros na execução dos testes e da identificação da mudança de comportamento do software sendo testado. Além disso, a ferramenta foi modificada para identificar e remover ocorrências de TCD (clones) e foi realizado um quase-experimento para avaliar a capacidade de identificação automática de erros de refatoração com 21 projetos *open source*. Os projetos selecionados tiveram as ocorrências de ET e TCD identificadas, removidas e validadas automaticamente. A partir dos resultados do quase-experimento realizado sobre a capacidade de validação automática das refatorações, a ferramenta Sentinel passou por uma nova atualização para remover ocorrências de Assertion Roulette (AR), Exception Catching Throwing (ECT) e Redundant Assert (RA) identificados por meio da integração com a ferramenta de identificação JNose. Em seguida, foi realizado um segundo quase-experimento, no qual 33 projetos *open source* foram

usados para identificar as variações nas ocorrências de *smells* nas classes refatoradas pela ferramenta Sentinel.

Além disso, como parte da evolução do artefato e da ferramenta, foi realizado um *survey* no qual desenvolvedores, analistas de qualidade, demais integrantes de equipes de desenvolvimento e pesquisadores sobre testes de unidade e *smells* foram convidados a responder questões demográficas e relacionados a problemas de qualidade relacionados com *smells*. Os problemas de qualidade das questões do *survey* estão relacionados com a identificação da frequência de ocorrência dos *smells* e respectivo o impacto na legibilidade do código, com o impacto causado nos erros dos testes e do software. Além disso, os participantes foram ainda questionados a responder sobre a prioridade de remoção dos *smells* e o esforço necessário para a remoção de cada *smell*.

A partir dos resultados do *survey*, foram definidas situações que devem ser consideradas na priorização das refatorações, as quais são descritas na seção 5.6 e compreendem aspectos como a possibilidade de refatoração, a remoção de *smells* que afetam o mesmo trecho de código, a prioridade de remoção dos *smells*, a criação/remoção de ocorrências de outros *smells* a partir da remoção de um *smell* específico além da aceitabilidade da remoção de um *smell* ou da criação de ocorrências decorrentes da remoção de um *smell* específico. Além das situações definidas, os resultados do *survey* foram usados para especificar uma árvore de decisão para determinar a aplicabilidade da remoção de ocorrências de um *smells* considerando as situações de priorização.

3.2.4 Demonstração

A atividade de demonstração foi realizada por meio de entrevistas semiestruturadas com especialistas com experiência em *Test Smells* e com integrantes de equipes de desenvolvimento com experiência em testes de unidade para coletar *insights* sobre a adoção do processo proposto no contexto de cada avaliador. Os participantes analisaram o processo proposto, a árvore de decisão das refatorações, as situações para a priorização das refatorações e classes de testes refatoradas para coletar dados sobre a percepção de facilidade de uso, utilidade e intenção de uso futuro com o objetivo de avaliar e aperfeiçoar o artefato.

Para a realização da demonstração, um roteiro para entrevistas semiestruturadas foi elaborado com base no Technology Acceptance Model (TAM) 3 (MARANGUNIĆ; GRANIĆ, 2015).

O perfil dos participantes é mostrado na Tabela 3-1, na qual é possível identificar 8 (oito) avaliadores, sendo 4 (quatro) pesquisadores da área de testes e *Test Smells* e 4 (quatro) da indústria, dos quais 3 (três) são analistas de qualidade e 1 (um) é desenvolvedor. Os participantes foram convidados por e-mail, no qual foi encaminhado o Termo de Consentimento Livre e Esclarecido (TCLE), Termo de Compromisso de Utilização de Dados (TCUD), material de apoio sobre *Test Smells*, o processo proposto, a árvore de decisão das refatorações e as situações para a priorização das refatorações.

Tabela 3-1 Perfil dos participantes da primeira avaliação (Fonte: o autor)

Avaliador	Gênero	Função	Tempo de experiência com testes (em anos)
#01	Masculino	Pesquisador	10
#02	Masculino	Analista de qualidade	8
#03	Feminino	Pesquisador	6
#04	Feminino	Pesquisador	6
#05	Feminino	Pesquisador	5
#06	Masculino	Desenvolvedor	3
#07	Masculino	Analista de qualidade	6
#08	Masculino	Analista de qualidade	20

A partir do aceite dos participantes e do recebimento do TCLE assinado, a demonstração dos elementos acima descritos foi realizada e gravada por meio do aplicativo Zoom. Durante a demonstração, foram tomados cuidados para garantir o anonimato dos participantes e das demais informações que poderiam identificar quaisquer organizações ou outras pessoas.

3.2.5 Avaliação

A avaliação dos resultados da demonstração foi realizada por meio da transcrição das avaliações realizadas. As transcrições foram armazenadas em documentos no Microsoft Word (.docx) e importadas no aplicativo Atlas.Ti versão 24,

que foi usado para codificar manualmente os dados na análise qualitativa dos resultados da demonstração.

A análise qualitativa e codificação dos dados foi realizada por meio da codificação inicial (ou aberta) e da codificação axial (SALDAÑA, 2013). Na codificação inicial, trechos de texto da transcrição de cada participante foram analisados e associados à códigos de acordo com o contexto de cada trecho. Em seguida, os códigos gerados foram analisados e agrupados em categorias na codificação axial.

3.2.6 Comunicação

A sexta atividade da DSR consiste em comunicar os resultados da pesquisa por meio da publicação de artigos científicos e por meio da escrita desta tese. A seguir são apresentados os trabalhos publicados até o momento como resultado da pesquisa realizada nesta tese.

Um resumo da proposta foi publicado e apresentado no Simpósio Doutoral do *International Conference on Software Engineering* (ICSE):

- PIZZINI, Adriano. Behavior-based test smells refactoring: Toward an automatic approach to refactoring Eager Test and Lazy Test smells. 2022. IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2022, pp. 261-263, doi: 10.1109/ICSE-Companion55297.2022.9793823.

Foi publicado um artigo no Simpósio Brasileiro de Qualidade de Software (SQBS) que descreve um método para remoção automática do Eager Test *smell* em um estudo exploratório para identificar o impacto causado pela remoção automática do smell em um repositório público.

- PIZZINI, Adriano; REINEHR, Sheila; MALUCELLI, Andreia. 2022. Automatic Refactoring Method to Remove Eager Test Smell. In XXI Brazilian Symposium on Software Quality (SBQS ’22), November 07–10, 2022, Curitiba, Brazil. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3571473.3571478

A versão inicial do processo Sentinel foi publicada no Simpósio Brasileiro de Qualidade de Software (SBQS):

- PIZZINI, Adriano; REINEHR, Sheila; MALUCELLI, Andreia. 2023. Sentinel: A process for automatic removing of Test Smells. In XXII Brazilian Symposium on Software Quality (SBQS ’23), November 07–10,

2023, Brasília, Brazil. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3629479.3630019

O artigo que apresenta a versão final do processo proposto, assim como os resultados do experimento de validação das refatorações e as avaliações dos especialistas está sendo escrito e será submetido para Journal Transactions on Software Engineering:

- PIZZINI, Adriano; REINEHR, Sheila; MALUCELLI, Andreia. 2025. XXXXX XXXXX XXXXX XXXXX XXXXX XXXXX. IEEE Transactions on Software Engineering, v. ###, p. ######, 2025. DOI: https://doi.org/##.####

3.3 Considerações sobre o capítulo

Neste capítulo foi delineada a estrutura da pesquisa baseada no método *Design Science Research*, na qual foram realizadas as etapas: A) definição do problema e motivação; B) definição dos objetivos da solução; C) Projeto e desenvolvimento; D) Demonstração; E) Avaliação; e F) Comunicação dos resultados obtidos.

O próximo capítulo descreve um método para eliminar o *smell Eager Test* em testes de unidade, abordando dois estudos exploratórios que investigaram as consequências dessa remoção automática.

CAPÍTULO 4 - ESTUDOS EXPLORATÓRIOS SOBRE A REMOÇÃO DE TEST SMELLS

Este capítulo apresenta o método proposto para remover as ocorrências do *smell* Eager Test dos testes de unidade e dois estudos exploratórios realizados para identificar consequências da remoção automática de *smells*, tais como: influência no tempo de execução dos testes de unidade, erros e problemas causados pela refatoração, além de mudanças nas métricas de qualidade relacionadas aos códigos dos testes de unidade. O método foi implementado em uma ferramenta de refatoração automática que foi usada na realização dos estudos exploratórios.

4.1 Método de remoção de Eager Test smell

Conforme mencionado, um método para a remoção de ocorrências do ET foi especificado e foi implementada a primeira versão de uma ferramenta de refatoração automática para suportar o método especificado. A especificação do método foi necessária devido a dois fatores: a característica do ET e a refatoração adotada para remover as ocorrências deste *smells*. O método de remoção do ET adotado foi a extração dos testes para métodos específicos, com base nos achados de Palomba e Zaidman (2017) que identificaram que a operação de extração é a mais usada para a remoção do ET.

O primeiro fator decorre da definição mostrada na seção 2.4 de que o *smell* ET é caracterizado pela existência de vários testes no mesmo método de teste, o que viola o princípio “responsabilidade única” e pode violar os princípios “(in) dependência de testes” e “projeto de teste de quatro fases”, todos definidos por Bowes et al., (2017) e mostrados na seção 2.2. Convém ressaltar que a definição original do ET dada por Deursen et al., (2001) emprega a expressão “vários métodos do objeto a ser testado” como condição que caracteriza a existência do *smell* ET. Porém, como o termo “vários” é subjetivo e pode causar interpretações divergentes. Assim, a regra de que se um método de teste é composto de dois ou mais testes, o método de teste é classificado como sendo afetado pelo *smell* ET. Os testes foram encontrados por meio da identificação de sequências de estímulos e afirmações, ou seja, quando o método

de teste é formado por dois ou mais ciclos de alternância de instruções de não afirmação com instruções de afirmação (ou asserções).

O segundo fator decorre da sugestão de Palomba et al., (2016) que sugerem que uma possível solução é a aplicação de uma refatoração do tipo Extract Method, capaz de dividir o método de teste de forma a especializar suas responsabilidades (FOWLER, 2018).

O método proposto é composto por seis etapas, conforme mostrado na Figura 4-1: 1) identificar as variáveis locais declaradas no método de teste, 2) converter variáveis declaradas em atributos, 3) converter declarações de variáveis locais em atribuições, 4) atualizar referências de variáveis locais para atributos, 5) extrair teste(s) para métodos de testes próprios, e 6) atualizar a ordem de execução dos testes.

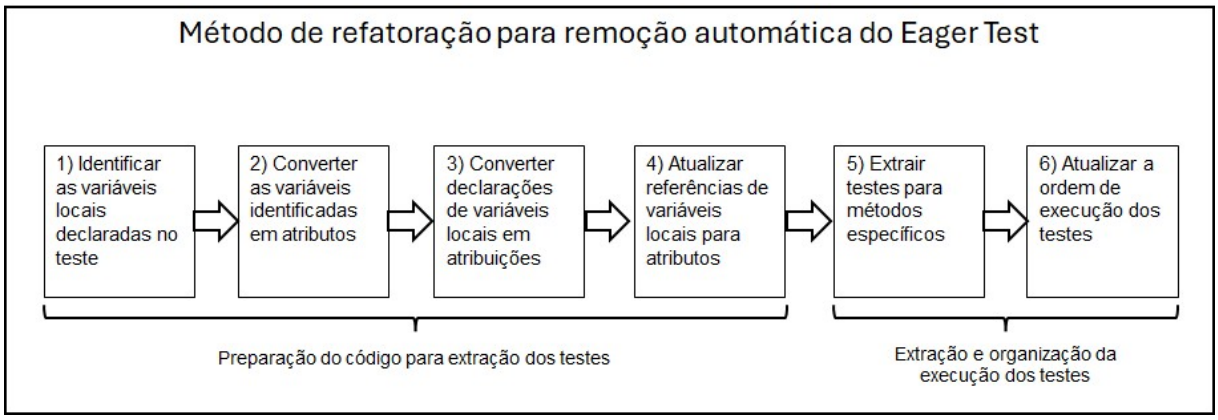

Figura 4-1 Etapas do método de refatoração para remoção automática do Eager Test *smell* (Fonte: o autor)

Enquanto as quatro primeiras etapas do método estão relacionadas com a extração das variáveis locais de um teste afetado pelo ET *smell*, as duas últimas estão associadas com a extração e organização dos testes criados, conforme descrito a seguir:

1. Identificar as variáveis locais declaradas no teste: consiste em identificar as variáveis locais que são declaradas no teste e que são referenciadas nos trechos de código que serão extraídos para comporem novos testes;

2. Converter as variáveis identificadas em atributos: consiste na criação de atributos estáticos com os nomes e tipos de dados usados na declaração das variáveis locais. A criação de atributos estáticos é necessária para que o estado dos objetos possa ser mantido após a execução dos testes;

3. Converter declarações de variáveis locais em atribuições: as declarações de variáveis que possuem inicialização são convertidas em instruções de

atribuição. A inicialização das variáveis é mantida na posição original do método para que a sequência de execução das instruções não seja alterada;

4. Atualizar referências de variáveis locais para atributos: esta etapa é executada somente se um ou mais atributos forem declarados com nomes diferentes das variáveis locais originais. Assim, para evitar erros de sintaxe e de semântica, as referências para os nomes originais das variáveis locais são localizadas e substituídas pelos respectivos nomes de atributos;

5. Extrair testes para métodos específicos: consiste em extrair as instruções, a partir do segundo ciclo de estímulo-afirmação, com as quais serão criados métodos de teste. A extração baseia-se na identificação dos blocos de código delimitados por *assertions*, que são instruções usadas para fazer afirmações. No caso de existirem duas ou mais asserções em sequência, a extração será realizada após a última asserção. As instruções contidas nos ciclos extraídos são removidas do método de teste original, permanecendo nele apenas as instruções do primeiro ciclo;

6. Atualizar a ordem de execução dos testes: consiste em anotar o método original e os métodos que foram extraídos com “Order(x)”, no qual x é um número sequencial, incrementado a cada novo teste extraído e que é usado para determinar a ordem de execução do teste extraído dentro da classe de testes.

A próxima seção descreve o primeiro estudo exploratório para identificar a significância estatística dos erros causados pela refatoração e o impacto no tempo de execução dos testes refatorados.

4.2 Estudo exploratório 1: Significância da ocorrência de erros e impacto no tempo de execução dos testes de unidade refatorados

A partir da especificação do método e da implementação da ferramenta de refatoração automática que implementa o método especificado, um estudo exploratório foi realizado para identificar a significância estatística da ocorrência de erros e o impacto causado pela remoção do ET no tempo de execução dos testes de unidade.

4.2.1 Planejamento

O estudo exploratório foi realizado por meio da remoção automática das ocorrências do ET em todas as classes de testes do projeto *open source* JFreeChart, versão 1.5.3, que foi refatorado automaticamente para avaliar o impacto da remoção

automática do ET das classes de testes. O projeto JFreeChart foi escolhido por ser desenvolvido em Java, usar o framework de execução de testes JUnit 5 e ter sido usado por Vahabzadeh et al (2018) na identificação do tempo de execução dos testes.

A avaliação do método proposto foi realizada por meio da experimentação na qual uma cópia do repositório do projeto foi refatorada automaticamente. As seguintes hipóteses foram definidas:

- Hipótese nula ($H1_0$): a refatoração automática não causa erros significativos nas afirmações realizadas pelos testes;
- Hipótese alternativa ($H1_1$): a refatoração automática causa erros significativos nas afirmações realizadas pelos testes;
- Hipótese nula ($H2_0$): A refatoração automática não influencia significativamente no tempo de execução dos testes;
- Hipótese alternativa ($H2_1$): A refatoração automática influencia significativamente no tempo de execução dos testes.

A partir da definição das hipóteses, foram executados os testes nos repositórios refatorados e não refatorados. A avaliação das hipóteses H1 e H2 foi realizada usando o teste de postos sinalizados de Wilcoxon com nível de significância de 5%, pois não foi possível assumir a normalidade dos dados das duas amostras relacionadas. Para a hipótese H1, foi atribuído o valor 0 (zero) para as classes que foram executadas sem erros e o valor 1 (um) para as classes que executaram com erros. Para a hipótese H2, os tempos de execução de cada classe de testes foram coletados do arquivo de *log* da construção de cada projeto pelo Maven.

4.2.2 Resultados

Das 350 classes de testes encontradas, 38 não eram afetadas pelo ET, 312 classes foram refatoradas por conterem pelo menos uma ocorrência ET e 17 apresentaram erros após a refatoração.

Para a avaliação das hipóteses definidas, todas as classes de testes foram consideradas (N=350). A quantidade de erros na execução dos testes refatorados foi superior à quantidade de erros na execução dos testes não refatorados. O teste de sinais de postos de Wilcoxon indica que a quantidade de erros é estatisticamente significativa (V = 153, *p-value* = 4.201e-05), resultando na rejeição da hipótese nula $H1_0$ e na aceitação da hipótese alternativa $H1_1$.

O teste de desempenho de refatoração foi realizado em um computador com Windows 10, CPU Intel(R) Core (TM) i7-7700HQ 2,80 GHz, 16,0 GB de RAM e SSD Sandisk de 240 GB, resultando em uma média de 56 segundos. A comparação do tempo de execução dos testes não refatorados com os refatorados indicou que o tempo de execução dos últimos foi superior aos primeiros. O teste de sinais de postos de Wilcoxon indicou que a diferença no tempo de execução é estatisticamente significativa (V = 14672, *p-value* = 1.157e-15), resultando na rejeição da hipótese nula $H2_0$ e na aceitação da hipótese alternativa $H2_1$.

4.2.3 Discussão dos resultados

O tempo de execução dos testes no repositório original passou de 3,9 segundos para 4,8 segundos no repositório refatorado, sendo superior aos 2,752s identificados por Vahabzadeh et al (2018). O fato de o tempo de execução no repositório não refatorado neste experimento ser diferente do encontrado por Vahabzadeh et al., (2018) pode ser atribuído ao aumento da quantidade de testes devido à evolução do software. Além disso, a conversão de variáveis locais em atributos altera o modo como a memória é alocada e gerenciada, enquanto o aumento da quantidade de métodos de testes aumenta o tempo necessário para a troca de contexto do teste (DEURSEN et al., 2001), ou seja, aumenta o tempo de processamento que o *framework* de testes precisa para executar todos os métodos de uma classe de testes, incluindo os métodos de configuração e desmonte, e código de classes de testes ancestrais, caso existam.

Em relação à quantidade de código, o aumento da quantidade de linhas lógicas de código é ocasionado pela declaração dos atributos e criação de métodos de testes específicos.

No que se refere ao método de refatoração proposto, mesmo que a ordem de execução das instruções dos métodos de testes após a extração permaneça inalterada, não é possível garantir que os testes continuam sendo executados sem a ocorrência de erros nas afirmações realizadas. Os problemas com os testes refatorados ocorreram em classes que possuem estrutura e comportamento semelhantes, nas quais as classes de testes são o código de teste e o dado do teste. Nestas classes, os atributos declarados são usados para armazenar o estado do objeto em teste e a execução da versão refatorada causa a reinicialização dos valores

destes atributos, causando problemas na execução dos testes refatorados devido à mudança de comportamento.

Os resultados deste primeiro estudo exploratório sobre as consequências da remoção automática do *smell* ET sugerem que existe significância estatística nos erros decorrentes da refatoração automática. Como consequência dos resultados identificados, considerou-se necessário identificar quais foram as causas dos erros dos testes de unidades refatorados. Assim, a próxima seção descreve um estudo exploratório para identificar quais foram as causas dos erros dos testes de unidade refatorados.

4.3 Estudo exploratório 2: Identificação de causas de erros nos testes refatorados

A partir dos resultados identificados na seção anterior sobre a significância estatística dos erros e sobre impacto no tempo de execução dos testes refatorados, foi realizado um segundo estudo exploratório para identificar quais são as causas dos erros causados pela refatoração automática.

Neste segundo estudo exploratório, o refatorador foi modificado para ter a capacidade de detectar e remover as ocorrências do *smell* Test Code Duplication (TCD). Os *smells* ET e TCD foram selecionados devido à sua maior ocorrência nos repositórios (ALJEDAANI et al., 2021), juntamente com o Assertion Roulette (AR). No entanto, o AR foi descartado porque sua refatoração não exigia alterações na estrutura da classe de teste, como a inclusão de métodos e atributos, e não era esperado que a sua remoção causasse a alteração do comportamento do teste. Além da maior ocorrência desses *smells*, ET e TCD foram escolhidos por causa das diferentes modificações na estrutura da classe de testes causadas pelas refatorações desses *smells*.

4.3.1 Planejamento

Para realizar o estudo, foram selecionados trinta e quatro projetos de código aberto do GitHub escritos em Java e que utilizavam o *framework* JUnit 5. No GitHub, os projetos foram listados em ordem decrescente de número de estrelas, sendo analisados até a página 50 do GitHub, durante o primeiro semestre de 2022.

As configurações dos projetos foram modificadas para remover *plugins* de implantação automática e adequação de estilo de código. Após a modificação da

configuração, os projetos selecionados foram construídos repetidamente para remover todos os testes que apresentaram erros. Após a construção de cada projeto, os testes foram considerados determinísticos e a refatoração automática dos testes foi realizada, com os erros sendo analisados e documentados durante a construção dos projetos.

As regras apresentadas na Tabela 4-1 foram estabelecidas para determinar se a refatoração de um método de teste que apresentasse os *smells* ET e CD seria realizada. A identificação do CD foi realizada ao identificar a igualdade nas linhas de código correspondentes às etapas de configuração e estímulo dos testes de unidade propostos por Silva e Vilain (2020), para encontrar a maior sequência de instruções compatíveis entre dois testes. Essa comparação não considerou as instruções de afirmação, pois o código de afirmação deveria estar contido em um método de teste separado (MESZAROS, 2007). Para a identificação do ET, foi utilizado o método proposto por Pizzini et al., (2022), que extrai os testes a partir da identificação dos blocos de instruções de afirmação contidos em cada método de teste.

Tabela 4-1 Regras definidas para refatoração automática de ET (Fonte: o autor)

Regra	Descrição
1	O método deve ser um teste ativo
2	O método não pode ser um teste repetido
3	O método não pode ser um teste parametrizado
4	O método não pode conter suposições
5	O método não pode ser um teste condicional
6	O método não pode sobrecarregar um método da classe ancestral (se existir)

Os projetos escolhidos para passarem por refatoração, juntamente com as métricas coletadas, estão detalhados na Tabela 4-2. Junto a cada projeto, são apresentados: a versão; o número de configurações de tipo implícito criadas durante a refatoração (A); a quantidade de métodos de teste que utilizam a configuração implícita após a refatoração (B); a taxa de reutilização da configuração implícita criada (A/B); a complexidade média ponderada (WMC) antes e depois da refatoração, incluindo a variação percentual (%); e o tamanho médio das classes de teste em linhas lógicas de código (LLOC, do inglês *Logical Lines Of Code*) antes e depois da

refatoração, também apresentando a variação percentual (%). As métricas WMC e LLOC foram obtidas utilizando a ferramenta CK (ANICHE, 2015).

Tabela 4-2 Projetos selecionados para refatoração automática (Fonte: o autor)

Projeto	Versão	Configurações implícitas criadas (A)	Testes que usam as configurações criadas (B)	Taxa de reutilização de configurações (A/B)	Média WMC			Média LLOC		
					Antes	Depois	Var. (%)	Antes	Depois	Var. (%)
Avro	1.11.1	83	204	2,5	13,4	10,9	18,7	116,6	94,1	19,3
Classgraph	4.8.155	5	17	3,4	2,5	2,5	0,0	21,3	21,0	1,4
Cucumber-jvm	7.11.1	155	424	2,7	6,4	5,3	17,2	44,5	37,5	15,7
Disruptor	4.0.0	19	52	2,7	9,6	8,2	14,6	60,5	51,5	14,9
Dropwizard	2.1.4	95	231	2,4	5,5	5,4	1,8	42,0	41,3	1,7
Fastjson2	2.0.24	292	860	2,9	6,6	5,7	13,6	58,3	49,4	15,3
Graphhopper	6.2	78	191	2,4	11,2	6,0	46,4	115,0	54,6	52,5
Jasypt	3.0.5	4	12	3,0	12,0	12,0	0,0	92,8	92,8	0,0
Javaparser	3.25.1	270	762	2,8	27,4	19,3	29,6	102,8	72,5	29,5
JDA	5.0.0	1	2	2,0	7,3	5,0	31,5	53,2	34,1	35,9
Jetcache	2.7.3	12	33	2,7	6,1	5,3	13,1	64,5	53,7	16,7
Jfreechart	1.5.3	299	785	2,6	10,4	5,5	47,1	91,1	31,3	65,6
Jodd	5.3.0	71	180	2,5	4,3	4,0	7,0	36,0	27,4	23,9
Jsoup	1.15.3	46	106	2,3	21,4	4,6	78,5	165,9	32,4	80,5
Liquibase	4.19.1	65	155	2,4	5,7	5,0	12,3	36,6	32,2	12,0
Logback	1.4.5	77	191	2,5	8,0	7,3	8,8	51,5	47,2	8,3
OpenPDF	1.3.30	15	34	2,3	3,7	3,5	5,4	40,4	37,6	6,9
Optaplanner	9.35.0	230	631	2,7	6,6	6,1	7,6	53,3	49,2	7,7
Poitl	1.12.1	6	18	3,0	6,5	6,0	7,7	45,0	41,2	8,4

Continua na próxima página

Continuação

Projeto	Versão	Configurações implícitas criadas (A)	Testes que usam as configurações criadas (B)	Taxa de reutilização de configurações (A/B)	Média WMC			Média LLOC		
					Antes	Depois	Var. (%)	Antes	Depois	Var. (%)
Redisson	3.19.3	424	1128	2,7	41,9	5,4	87,1	153,2	43,8	71,4
Ripme	1.7.95	9	19	2,1	3,0	2,9	3,3	15,5	14,7	5,2
RoaringBitmap	0.9.39	221	709	3,2	34,6	6,7	80,6	218,3	40,8	81,3
Rsocket-java	1.1.3	47	119	2,5	13,1	9,9	24,4	91,2	68,0	25,4
Simple binary encoding	1.27.0	21	48	2,3	7,0	4,9	30,0	66,7	45,3	32,1
Simplify	1.3.0	4	8	2,0	9,5	8,5	10,5	69,4	61,8	11,0
Spring-batch	5.0.1	289	717	2,5	6,5	5,6	13,8	54,2	45,2	16,6
Spring-cloud-netflix	4.0.0	17	42	2,5	7,3	7,1	2,7	47,1	45,7	3,0
Spring-data-jpa	3.0.3	86	360	4,2	7,6	6,6	13,2	47,6	40,2	15,5
Springdoc-openapi	2.0.2	1	2	2,0	2,6	2,6	0,0	12,9	12,9	0,0
Sqlite-jdbc	3.41.0.0	40	176	4,4	14,5	10,7	26,2	153,4	110,7	27,8
Tablesaw	0.43.1	139	429	3,1	9,7	5,4	44,3	70,1	36,7	47,6
Thymeleaf	3.1.1	33	131	3,9	6,4	6,2	3,1	48,9	46,6	4,7
Unirest-java	3.14.2	50	125	2,5	8,6	5,5	36,0	51,5	31,9	38,1
Wiremock	3.0.0	172	459	2,7	9,7	7,0	27,8	64,6	45,5	29,6

4.3.2 Resultados da remoção automática de smells

Os resultados do estudo exploratório para identificar os impactos da refatoração automática nos testes são divididos em duas categorias: o impacto nas métricas das classes refatoradas e os tipos de erros nos testes causados pela refatoração.

Em relação ao impacto nas métricas das classes de testes, a redução média da complexidade (WMC) foi observada em 31 dos 34 projetos submetidos à refatoração, com exceção dos projetos Classgraph, Jasypt e SpringDoc. O número médio de linhas lógicas de código (LLOC) diminuiu em 32 dos 34 projetos refatorados, com exceção dos projetos Jasypt e SpringDoc. Projetos como Graphhopper, JFreeChart, Jsoup, Redisson e RoaringBitmap apresentaram reduções percentuais nas métricas de complexidade (WMC) e tamanho (LLOC), todas superiores a 45% em ambas as métricas. Metade dos projetos analisados revelou uma diminuição percentual entre 10% e 40% em pelo menos uma das métricas. Nos casos em que não houve alteração nas métricas investigadas, a taxa de reutilização da configuração criada foi equivalente à dos demais projetos, embora esses projetos estejam entre os cinco que geraram a menor quantidade de configurações.

Em relação aos tipos de erros causados pela refatoração nos testes de unidade, foram identificados três tipos mostrados na Figura 4-2: a) erros de compilação, b) erros na execução dos testes após a refatoração e c) mudança de comportamento do software por causa da refatoração. Estes problemas podem ocorrer no contexto de cada método/classe de teste refatorado ou em outros métodos/classes que referenciam o método de teste refatorado. Convém ressaltar que mesmo que os tipos de problemas “a” e “b” não sejam identificados, é possível que o tipo de erro “c” tenha sido gerado, o que somente pode ser identificado por meio da comparação das instruções do software que foram executadas antes e após a refatoração. Esta identificação depende da compilação, construção, execução dos testes e da análise do comportamento dos testes nas duas versões do software, ou seja, na versão não refatorada e versão refatorada.

Os erros de compilação tipo (a), podem ser identificados por meio da construção do software após a refatoração e da análise do *log* do processo de construção gerado pela ferramenta de construção do software refatorado. A simples análise da compilação isolada da classe de teste refatorada pode não ser suficiente para identificar se a refatoração causou erros de compilação, pois o erro pode ocorrer

em uma classe de testes que, de alguma forma, referencie algum elemento da classe refatorada. Como a compilação precede a execução dos testes, este tipo de erro é identificado com maior rapidez quando comparado aos demais tipos de problemas, pois somente após a compilação das classes de produção e de testes é que os testes serão executados. Conforme descrito em Pizzini et al., (2022), alguns erros de compilação podem ocorrer devido à existência de código morto que faz referência a métodos de testes extraídos para subclasses e à extração de métodos de teste sobrepostos de uma classe de testes abstrata.

Os erros do tipo (b) na execução dos testes refatorados são um problema que pode ser identificado por meio da análise do resultado da execução dos testes e da análise do *log* de construção do software refatorado. Neste tipo de problema, os erros estão relacionados com a mudança na forma como o código de teste é executado, o que faz com que os objetos criados em cada teste possuam estados diferentes dos esperados, causando a mudança de comportamento dos testes de unidade executados na construção do software (PIZZINI et al., 2022).

As causas dos erros nos testes de unidade após a refatoração automática foram identificadas em três subcondições: b.1) eliminação dos recursos necessários para a execução dos testes; b.2) mudança de estado dos objetos na transição da execução do teste; b.3) referência de classes armazenadas em constantes e arquivos de configuração.

Na subcondição (b.1), a extração de testes para métodos específicos aumenta a quantidade de vezes que os métodos das etapas de configuração e desmonte são executados para fazer a transição entre os testes, causando a reinicialização do estado dos objetos usados nos testes refatorados devido à eliminação de recursos necessários para executar os testes.

A subcondição (b.2), relacionada à mudança de estado dos objetos, foi apontada em Pizzini et al., (2022) como uma condição em que os testes apresentam erros com a transição dos métodos de teste após a refatoração, mesmo que sejam testes não refatorados.

A subcondição (b.3), relacionada à erros nas referências das classes refatoradas, foi causada por testes que referenciaram classes incorretas em três situações: anotações que estavam na classe de teste e, após a refatoração, foram colocadas na classe de configuração criada; referências às classes de teste que eram

constantes no código de teste; e referência às classes de teste que estavam em arquivos externos ao código, como configuração de injeção de dependência.

A alteração do comportamento do software sendo testado (c) pode ser ocasionada pela mudança do comportamento dos testes devido à refatoração. Um exemplo que pode ser citado é o caso de testes repetidos afetados pelo ET, no qual a extração dos testes para métodos de testes separados remove a ligação entre a repetição do método original com os extraídos. Deste modo, o método original continuará sendo executado inúmeras vezes, enquanto os métodos extraídos serão executados uma única vez, ocasionando a redução da cobertura do código do software devido à redução da quantidade de repetições executadas.

Figura 4-2 Tipos de erros causados pela refatoração automática e como podem ser identificados (Fonte: o autor)

4.3.3 Discussão dos resultados

A remoção dos *smells* ET e CD favorece a redução do tamanho e da complexidade das classes de teste, pois as classes e os testes passam a ser compostos por um conjunto menor de instruções com responsabilidades mais específicas e maior possibilidade de reutilização do código de teste.

Os erros causados pela refatoração automática sugerem a necessidade de uma etapa de validação automática no processo definido para evitar que a qualidade da suíte de testes seja prejudicada pela implantação de testes com erros. A etapa de validação tende a ser a etapa mais demorada do processo de refatoração, pois implica a execução de ações pela ferramenta de construção de projetos, como compilar as

classes alteradas, identificar dependências, executar os testes refatorados, entre outras.

As alterações de comportamento que não causam erros na execução dos testes refatorados são o tipo mais complexo de problema que deve ser detectado na validação. Essa complexidade decorre do fato de que a identificação da mudança de comportamento depende da análise do comportamento do software que está sendo testado e não apenas do resultado da execução do teste.

A priorização e a validação das refatorações podem desempenhar um papel significativo no desempenho geral do processo de refatoração, pois quanto maior a granularidade da validação, maior tende a ser o custo global associado e maior a precisão da validação, conforme pode ser observado na Figura 4-3. Por exemplo, a menor granularidade corresponde a refatorar e validar cada ocorrência de TS identificadas em uma classe de testes, o que aumenta a quantidade de vezes que a construção do software é realizada. Com isso, todos os testes do software são executados, o que aumenta a precisão na identificação de refatoração problemática e o custo global de refatoração de todas as classes de testes de um software. Por outro lado, o maior nível de granularidade por software possui o menor custo global de validação quando comparado aos demais níveis, mas possui a menor precisão na identificação de qual refatoração é problemática. Isso ocorre porque um mesmo método de teste pode ser refatorado para remover mais de uma ocorrência de *smells,* e problemas podem ser encontrados em classes ou testes não refatorados que, de alguma forma, referenciam o método de teste refatorado.

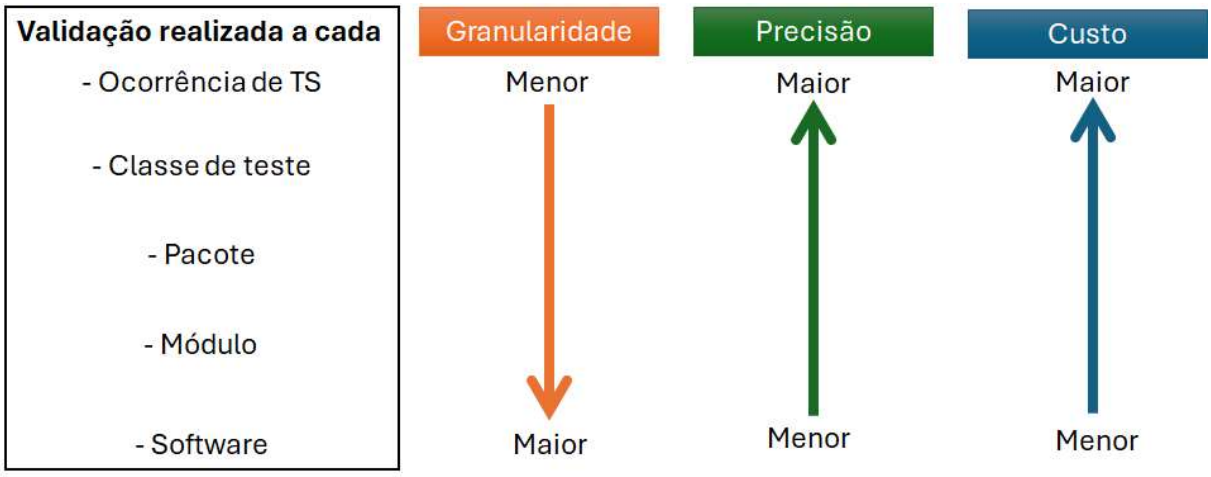

Figura 4-3 Custo e precisão da refatoração de acordo com a granularidade das validações (Fonte: o autor)

Uma possibilidade de redução do custo de validação é executar refatorações e validações em blocos, agrupando ocorrências de *smells* com baixa possibilidade de problemas na validação devido às características dos *smells*. Por exemplo, refatorar e validar em conjunto os *smells* que não alteram a estrutura da classe de testes, como AR, RA e RP. Portanto, com o aperfeiçoamento da priorização das refatorações, é esperada a redução da possibilidade de descarte de refatorações não problemáticas por causa de problemas causados por uma ou mais refatorações.

Considerando que o tempo necessário para identificar e remover manualmente as ocorrências de *smells* tende a ser maior do que o da remoção automática, a equipe de desenvolvimento pode ser beneficiada pela remoção automática de smells, por meio da redução da possibilidade de erros de refatoração e aumento da qualidade do código de teste.

Outro ponto importante a ser ressaltado é a preservação dos estados dos objetos usados nos testes que pode ser alcançada convertendo as declarações de variáveis locais em atributos e ordenando a execução dos métodos de testes extraídos para métodos específicos. A ordenação da execução dos testes extraídos tem por objetivo manter a ordem da execução dos testes de modo que a sequência original de instruções não seja alterada pela refatoração. Dependendo do conjunto de instruções existentes no primeiro teste extraído, as instruções relacionadas com a criação das instâncias de objetos podem ser movidas para o método de configuração da classe de testes.

Além da criação de um método de configuração na classe de testes, pode ser necessária a mudança do ciclo de vida da execução para preservar o estado dos objetos durante os testes. Isso pode levar à ocorrência do *smell abnormal UTF-Use* (AUU), onde o comportamento do *framework* de testes é modificado nas classes de testes (REICHHART et al., 2007). Assim, a possibilidade de surgir um novo *smell* devido à aplicação de uma abordagem para remover um *smell* existente não pode ser descartada.

4.4 Considerações sobre o Capítulo

Esse capítulo apresentou um método para remoção automática de ocorrências do *smell* ET das classes de testes. Além do método, foram apresentados os resultados de dois estudos exploratórios sobre a remoção automática de *smells*. No primeiro

estudo, foi analisado o impacto da remoção automática do ET no tempo de execução dos testes de unidade e na ocorrência de erros nos testes refatorados. No segundo estudo, o refatorador foi modificado para remover ocorrências de TCD e teve por objetivo identificar as causas dos erros na execução dos testes refatorados automaticamente.

A constatação, no segundo estudo, de que a refatoração automática pode causar erros de compilação, mudança de comportamento dos testes refatorados ou mudança de comportamento do software sendo testado fez com que fosse necessário incluir uma etapa de validação automática das refatorações realizadas. Com isso, foi especificado um processo para a remoção automática de *smells* que é apresentado no próximo capítulo. Além do processo especificado, também serão apresentados detalhes sobre a ferramenta que implementa o processo, aspectos relacionados à priorização das refatorações, um quase-experimento para avaliar a capacidade de validação automática das refatorações e um *survey* sobre a percepção de Test Smells e priorização das refatorações.

CAPÍTULO 5 - PROCESSO PARA REMOÇÃO AUTOMÁTICA DE TEST SMELLS

Este capítulo apresenta o processo para remoção automática de *smells*, denominado Sentinel, detalhes sobre a ferramenta de remoção automática que segue o processo especificado, a instrumentação de código dos projetos e as refatorações implementadas. Além disso, são apresentados os resultados de um quase-experimento para avaliar a capacidade de validação automática das refatorações realizadas, de um quase-experimento para identificar a alteração na quantidade de ocorrências de *smells* após a refatoração automática, e de um *survey* para identificar a percepção de aspectos relacionados à frequência de ocorrência e impacto da presença de *smells*, assim como sobre a priorização das refatorações.

5.1 Processo para remoção automática de smells

A partir dos resultados das pesquisas exploratórias descritas nas seções 4.2 e 4.3, foi especificado um processo para remoção automática de Test Smells por meio da refatoração. O processo foi denominado Sentinel e é apresentado na Figura 5-1.

O processo definido é composto por 14 (quatorze) atividades, sendo elas: Criar repositório de validação, Carregar arquivos do projeto, Identificar classes de testes, Instrumentar classes de código do software, Selecionar uma classe de teste a ser refatorada, Analisar classe de teste, Identificar ocorrências de Test Smells, Priorizar refatorações, Identificar o comportamento pré-refatoração, Refatorar a classe de testes, Identificar o comportamento pós-refatoração, Comparar comportamentos pré e pós-refatoração, Descartar refatorações e Implantar refatorações.

A primeira atividade, **Criar repositório de validação**, é necessária para criar duas réplicas de cada projeto que será refatorado evitando que as modificações de código realizadas pelo refatorador sejam aplicadas acidentalmente no repositório de produção sem passarem pela validação. O primeiro repositório será usado para identificar o comportamento do software antes da refatoração e o segundo para identificar o comportamento após a refatoração.

Na segunda atividade, **Carregar arquivos do projeto**, o refatorador identifica todas as classes que contenham algum tipo de código, seja do software ou de testes. Em seguida, a terceira atividade é executada, **Identificar classes de testes**, para identificar cada uma das classes de testes e classes auxiliares dentre os arquivos do projeto identificados na segunda atividade. A identificação de uma classe como sendo de testes se deve pela presença da anotação que define ao menos um método da classe como sendo um teste, o que é realizado por meio da anotação “@test”. Os arquivos de projeto identificados na segunda atividade e que não foram classificados como classes de testes na terceira atividade serão considerados como arquivos de código do software e serão instrumentados na quarta atividade, **Instrumentar classes de código do software**. A instrumentação consiste em inserir instruções para registrar o fluxo de execução de cada um dos métodos e registrar o estado do objeto em cada ponto de análise. São inseridas instruções de registro do fluxo de execução no início e final do método, bem como no início de cada um dos caminhos de execução que podem ser definidos por meio de estruturas de seleção, repetição, tratamento de exceções, entre outras.

A quinta atividade, **Selecionar uma classe de teste a ser refatorada**, consiste em identificar uma classe de testes que ainda não tenha sido refatorada na lista de classes de testes identificadas na terceira atividade, **Identificar classes de testes**.

A sexta atividade, **Analisar classe de teste**, é executada se houver uma classe de testes para ser refatorada. Nesta atividade são identificadas características da classe de testes, tais como:

1. Os elementos declarados, como tipos, métodos e atributos, construtores, classes internas, extensão de classe e implementação de *interfaces*;
2. O tipo de cada método declarado, como métodos repetidos e parametrizados, métodos com suposições, métodos de configuração e desmonte, métodos de teste, métodos auxiliares, entre outros;
3. Os *frameworks* de execução de teste usados na classe de teste;
4. A existência de uma classe de código associada.

A identificação dos *frameworks* de execução de testes é necessária para a correta identificação dos métodos invocados no código dos testes e pode impedir que uma refatoração seja realizada. Por exemplo, o uso combinado das anotações de testes do JUnit 5 com as afirmações AssertJ, como *assertThat*, é uma situação na

qual a remoção do *smell* AR não pode ser realizada por meio da inclusão do parâmetro que representa a explicação da afirmação para o caso de erro na execução do teste.

Após a análise da classe de testes selecionada será executada a sétima atividade, **Identificar ocorrências de Test Smells**, que tem como entradas as regras de identificação de ocorrências de *smells* e a classe de testes selecionada na quinta atividade. A saída desta atividade é a lista de ocorrências de TS identificadas na classe de testes selecionada.

A lista de ocorrências de TS identificadas e a classe de testes selecionada são entradas para a oitava atividade, **Priorizar refatorações**, que tem como entradas a classe de testes selecionada na quinta atividade, as ocorrências de TS identificadas na sétima atividade pelo aplicativo JNose e as regras de priorização definidas na ferramenta desenvolvida, descritas na seção 5.6. A saída da execução desta atividade é o conjunto de ocorrências de TS que podem ser refatorados. Nessa versão do processo, a priorização das refatorações é realizada seguindo a seguinte ordem: 1) *smells* que afetam trechos de código do método de teste, como AR e DA; 2) *smells* que reduzem a complexidade ciclomática dos testes, como ECT; 3) *smells* que afetam trechos de código do método de teste, como ET; 4) *smells* que podem afetar mais de um método de teste, como TCD.

A próxima (nona) atividade a ser executada é **Identificar o comportamento pré-refatoração**, na qual os testes da classe de testes são executados e o comportamento do software sendo testado é registrado por meio da identificação do fluxo de execução das instruções executadas pelo software e do respectivo estado dos objetos em cada parte do fluxo de execução.

A décima atividade consiste na execução da atividade **Refatorar a classe de testes**, na qual as refatorações são aplicadas na classe de teste selecionada a partir da lista de ocorrências de TS identificadas que podem ser refatoradas, que são resultado da execução da oitava atividade.

Após a execução da atividade de refatoração, é executada a décima primeira atividade, **Identificar o comportamento pós-refatoração**, na qual ocorre a identificação de problemas causados pela refatoração, como os descritos na seção 4.3.2. A primeira ação a ser executada é a construção do projeto para identificar se existem erros de compilação, seguida da execução de toda a suíte de testes para identificar se existem testes que passaram a ter erros após a refatoração. Após a execução de todos os testes da suíte, os testes refatorados são executados para

registrar a o fluxo de execução do código do software com os respectivos estados dos objetos criados.

A décima segunda atividade, **Comparar comportamentos pré e pós-refatoração**, é executada para comparar as instruções executadas e o estado dos objetos na versão não refatorada e refatorada dos testes.

A décima terceira atividade, **Descartar refatorações**, representa o retorno da classe de teste ao estado anterior à refatoração e a sinalização de que as refatorações realizadas não serão realizadas novamente na próxima sequência de ações de refatoração.

A décima quarta atividade, **Implantar refatorações**, representa a substituição da classe de teste do repositório de produção com a versão refatorada da classe de testes e com as subclasses extraídas na atividade de refatoração e a sinalização de que as refatorações realizadas foram devidamente validadas.

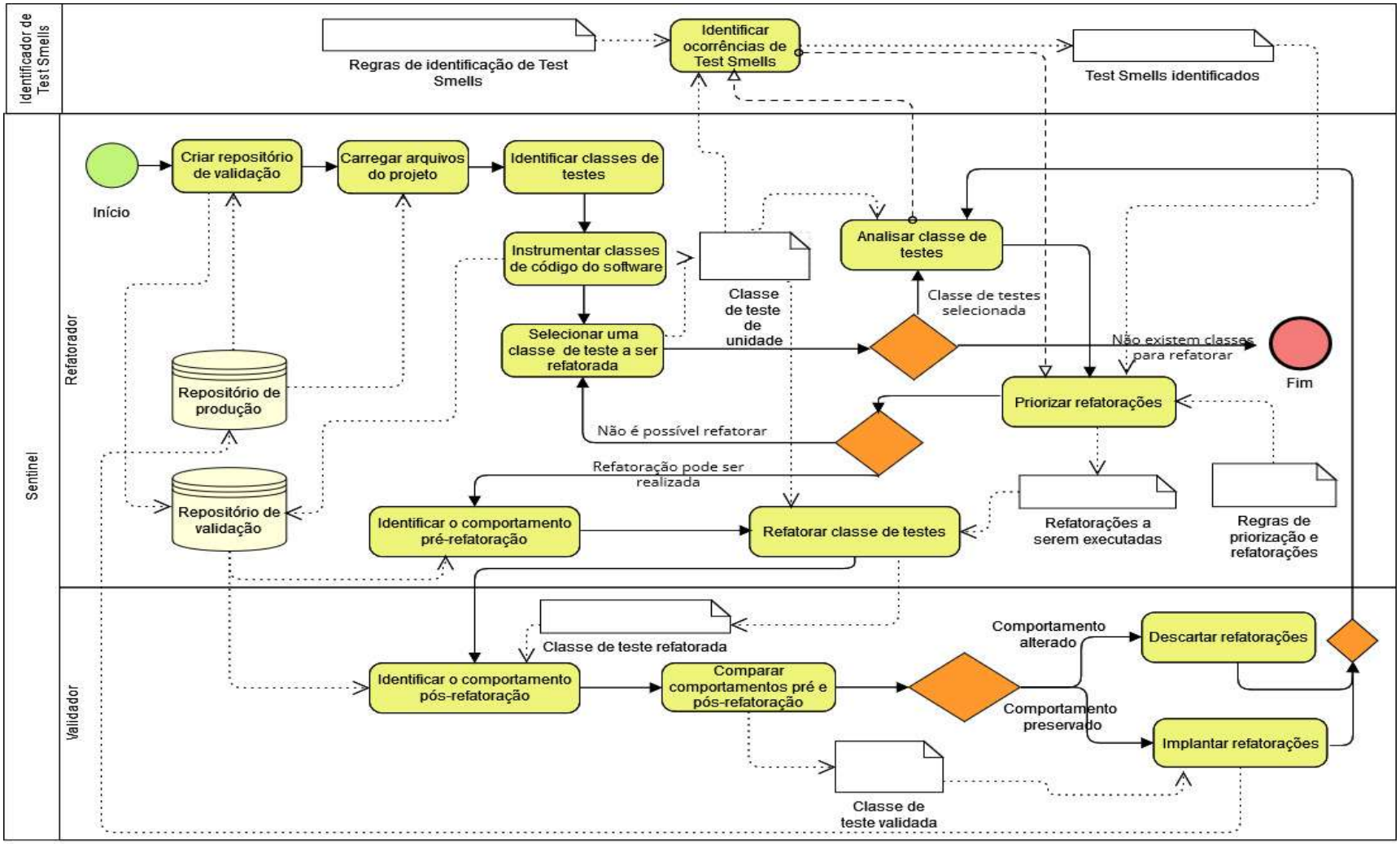

Figura 5-1 Processo proposto para refatoração automática de testes de unidade (Fonte: o autor)

A próxima seção apresenta aspectos relacionados com a implementação da ferramenta de remoção automática de *smells* que implementa o processo especificado nesta seção.

5.2 Atualizações do refatorador Sentinel

Esta seção apresenta aspectos sobre as atualizações da ferramenta denominada Sentinel que implementa o processo de refatoração proposto na seção anterior para remover automaticamente ocorrências de *smells*. Conforme mencionado na seção 4.2, a primeira versão do refatorador foi implementada para realizar a remoção automática do *smell Eager Test*.

A primeira atualização do refatorador foi realizada para a validação automática das refatorações, por meio da instrumentação das classes do software sendo testado, que é apresentada na seção 5.2.2. Esta versão do refatorador foi usada na condução do quase-experimento descrito na seção 5.3. A segunda atualização do refatorador foi realizada para aumentar a quantidade de *smells* removidos automaticamente, incorporando a remoção de TCD (clones), conforme as refatorações descritas na seção 5.2.3. A versão resultante do refatorador foi usada na realização do quase-experimento descrito na seção 0 e na avaliação do processo de refatoração proposto que foi descrito no CAPÍTULO 6 -.

5.2.1 Integração com a ferramenta de detecção de Test Smells JNose

Seguindo a sugestão de Aljedaani et al., (2021) sobre o desenvolvimento de novas ferramentas ou a integração das existentes, a ferramenta de identificação JNose foi integrada à ferramenta desenvolvida devido à sua capacidade de indicar a ocorrência de cada *smell* com a informação sobre as linhas de código envolvidas em cada ocorrência. A indicação dos métodos de testes e das respectivas linhas afetadas por determinado *smell* é um recurso importante para delimitar o trecho de código afetado por um *smell*.

A integração foi realizada por meio da criação de uma classe de conversão de objetos internos do JNose para o Sentinel e usou alguns trechos de código da classe principal do JNose para permitir a identificação de *smells* na classe selecionada na ferramenta de refatoração desenvolvida. Assim, o contexto de cada classe de testes de unidade pode ser analisado em conjunto com as ocorrências de TS identificadas pelo JNose.

5.2.2 Instrumentação das classes do software

A instrumentação do código do software consiste em inserir instruções que atendam a um objetivo específico em pontos específicos do código alvo. No caso da ferramenta Sentinel, o objetivo da instrumentação é identificar o estado do objeto na entrada e saída de cada método executado, além do início de cada caminho (*branch*) do fluxo de execução das instruções do método instrumentado, simulando a etapa de depuração que seria realizada pelos membros da equipe de desenvolvimento. Um ponto que merece atenção é a saída do método, pois é possível que ocorra o lançamento de uma exceção que interrompa o fluxo de execução das instruções fazendo com que não seja possível identificar o estado do objeto.

As instruções do método a ser instrumentado podem ser movidas para o bloco de proteção de uma instrução do tipo *try*/*finally*. Antes da execução do *try* é realizado o registro da entrada do método em questão com o respectivo estado. No bloco *try*, as instruções do método serão executadas e, mesmo que uma exceção seja lançada por uma destas instruções, as instruções do bloco *finally* serão executadas para registrar a saída do método com o respectivo estado. No bloco *finally*, é inserida a instrução para registrar a saída do método em questão com o respectivo estado do objeto.

Na Figura 5-2 é apresentado um trecho da classe StaxEventItemReader do projeto Spring Batch, versão 5.0.1. Nas linhas 239, 240, 258, 259 e 260 pode ser observado o bloco *try*/*finally* para registrar o início e término da execução do método e o respectivo estado do objeto. Nas linhas 243, 245, 249 e 253 foi inserida uma instrução para registrar a execução das instruções em cada um dos caminhos existentes na classe de produção.

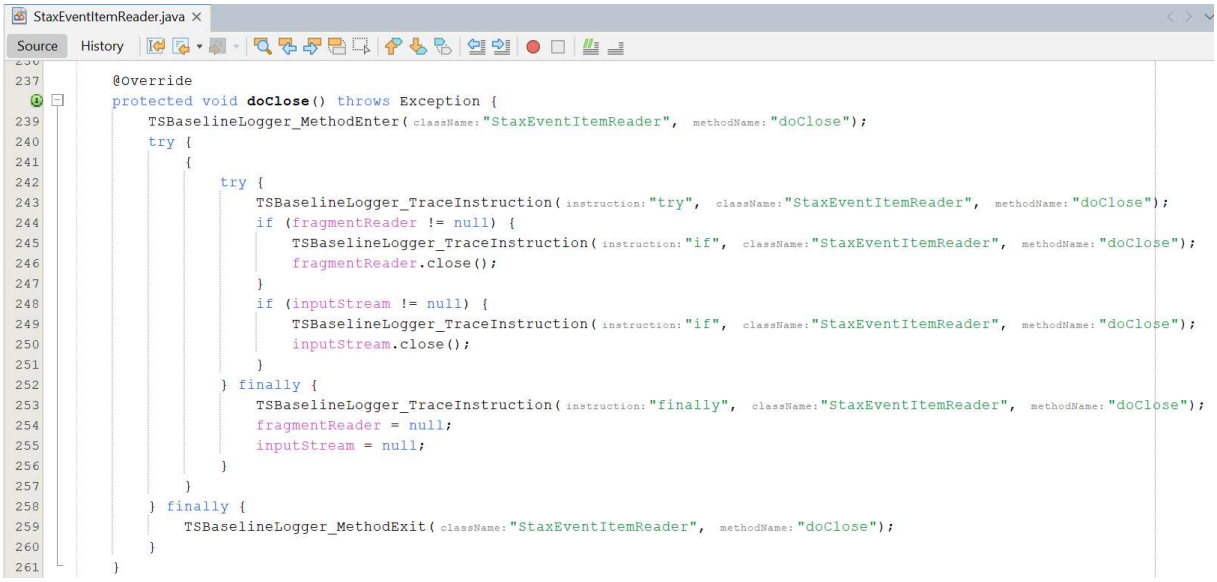

Figura 5-2 Exemplo de código de produção instrumentado para gerar o trace da execução (Fonte: o autor)

Outra consideração é não ser necessário instrumentar *getters* e construtores. No caso de um *getter*, a finalidade deste método é retornar o valor de um atributo declarado no objeto (DAKA e FRASER, 2004), ou seja, não é um método que altera o estado do objeto, e, no caso dos construtores, eles são métodos que são executados na instanciação do objeto, ou seja, na definição do estado inicial de cada objeto específico (DAKA e FRASER, 2004).

5.2.2.1 Problemas causados pela instrumentação

Como mencionado anteriormente, a instrumentação do código foi usada para determinar quais instruções do software foram executadas e o respectivo estado de cada objeto que estão relacionados com a execução de cada teste de unidade. Como a instrumentação altera o código da classe instrumentada, a inclusão de instruções na classe instrumentada pode causar problemas como a mudança de comportamento da classe.

Um exemplo de situação em que a instrumentação causa problemas é a instrumentação de classes de testes que referenciem elementos de si próprias, como atributos e métodos, seja direta ou por meio de constantes, e a quantidade de linhas de código. A mudança de comportamento dos testes instrumentados é uma situação que pode ser observada na linha 24 do código mostrado na Figura 5-3, onde é realizada uma afirmação sobre uma exceção lançada pelo objeto criado na linha 14 da própria classe de testes.

```
1  package io.dropwizard.logging.common;
2
3  import ch.qos.logback.classic.spi.ThrowableProxy;
4  import org.junit.jupiter.api.BeforeEach;
5  import org.junit.jupiter.api.Test;
6
7  import java.io.IOException;
8  import java.util.Collections;
9
10 import static org.assertj.core.api.Assertions.assertThat;
11
12 class PrefixedExtendedThrowableProxyConverterTest {
13     private final PrefixedExtendedThrowableProxyConverter converter = new
       PrefixedExtendedThrowableProxyConverter();
14     private final ThrowableProxy proxy = new ThrowableProxy(new IOException("noo"));
15
16     @BeforeEach
17     void setup() {
18         converter.setOptionList(Collections.singletonList("full"));
19         converter.start();
20     }
21
22     @Test
23     void prefixesExceptionsWithExclamationMarks() throws Exception {
24         assertThat(converter.throwableProxyToString(proxy))
25                 .startsWith(String.format("! java.io.IOException: noo%n" +
26                                           "! at
                                           io.dropwizard.logging.common.PrefixedExtendedThrowableProxyConverterTest.<init>(
                                           PrefixedExtendedThrowableProxyConverterTest.java:14)%n"));
27     }
28 }
```

Figura 5-3 Referência à linha de código da própria classe na linha 26 (Fonte: o autor)

A identificação do estado dos objetos por meio da instrumentação pode ser prejudicada em alguns casos, como a sobreposição de métodos usados no cálculo do código *hash* do objeto. Por exemplo, no projeto Spring Batch, na linha 248 da classe *AbstractResource* mostrada na Figura 5-4 é possível identificar que o método *hashCode* retorna o hash do método *getDescription*. A classe *AbstractResource* é estendida na classe de testes *StaxEventItemReaderTests* na qual o método *getDescription* é sobreposto, mas retorna o valor *null*, conforme pode ser observado na linha 481 da Figura 5-5. O retorno do valor *null* retornado impede o cálculo do *hashCode* do objeto quando o teste é executado. Com isso, não é mais possível fazer a construção do projeto após a instrumentação, pois durante a execução dos testes será lançada uma *NullPointerException* quando o estado do objeto é identificado por meio do cálculo do *hashCode* de cada objeto. Assim, o erro na construção do projeto causado pela instrumentação pode fazer com que todas as refatorações realizadas sejam identificadas como problemáticas, pois erros de execução podem ocorrer nos testes refatorados e nos testes não refatorados.

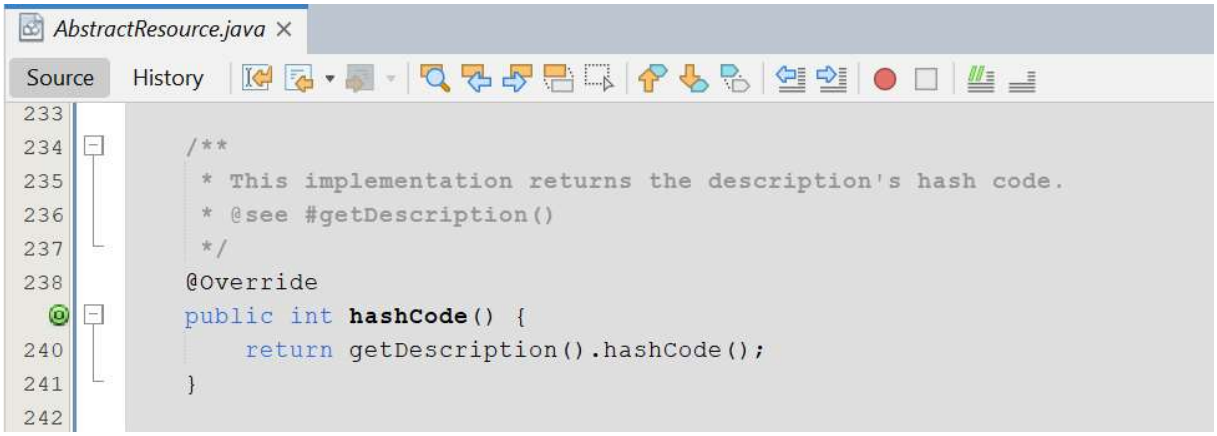

Figura 5-4 Classe abstrata cujo método hashCode retorna o hashCode de um método (Fonte: o autor)

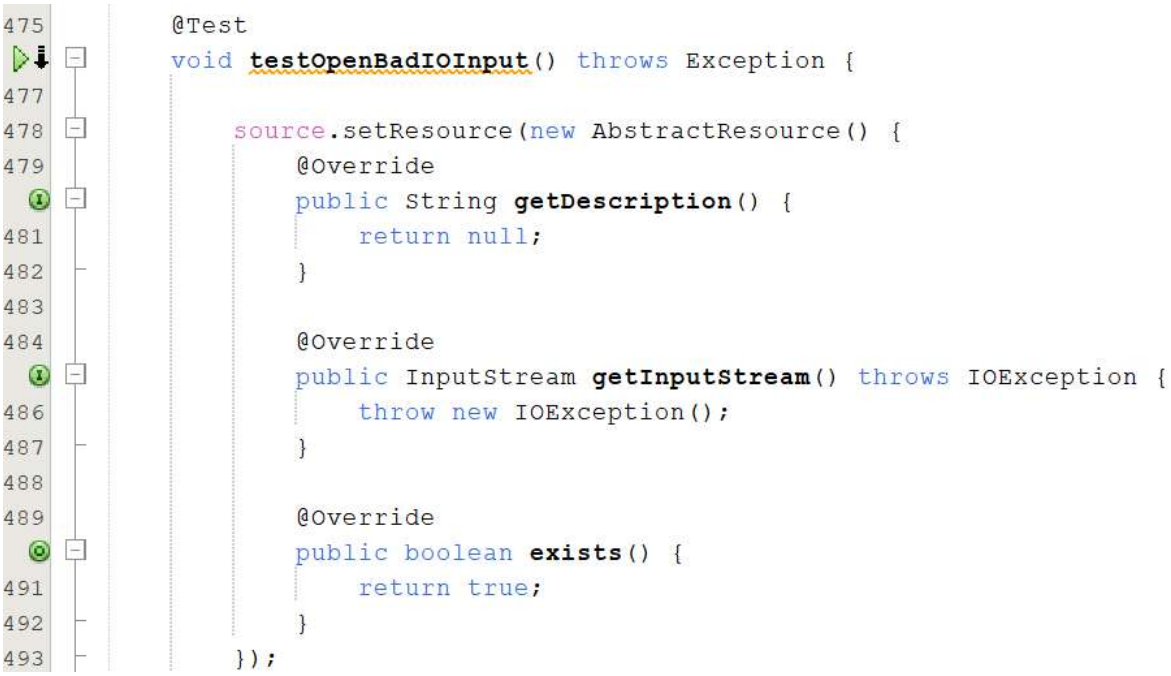

Figura 5-5 Classe concreta que estende a classe abstrata e retorna null no método usado para calcular o hashCode na linha 481 (Fonte: o autor)

Os casos apresentados nas Figuras 5-4 e 5-5 mostram que a instrumentação do código das classes de unidade pode ocasionar erros nos testes e no software. Para mitigar os problemas causados, foram adotadas duas soluções: instrumentar somente as classes de produção e criar uma função de cálculo de *hash* para representar o estado de modo que as classes de testes que estendam classes de produção não sejam consideradas na análise de mudança de comportamento do software.

A instrumentação das classes de produção fez com que os testes tivessem que ser executados individualmente, antes e após a refatoração, para identificar o

comportamento do software, pois não é possível identificar qual o teste que estimulou o sistema e causou a execução das instruções do código.

A criação de uma função de cálculo de *hash* para representar o estado do objeto foi realizada para evitar erros com as funções de cálculo de *hash* implementadas, mas também para prover uma implementação de cálculo de *hash* para as classes nas quais a função não tinha sido implementada. A implementação da função está baseada na identificação de todos os atributos não transientes declarados em cada classe de produção, sejam eles estáticos ou dinâmicos, que são passados por parâmetro para a função estática “hash ()” da classe “Objects” do Java[3], conforme pode ser observado na Figura 5-6.

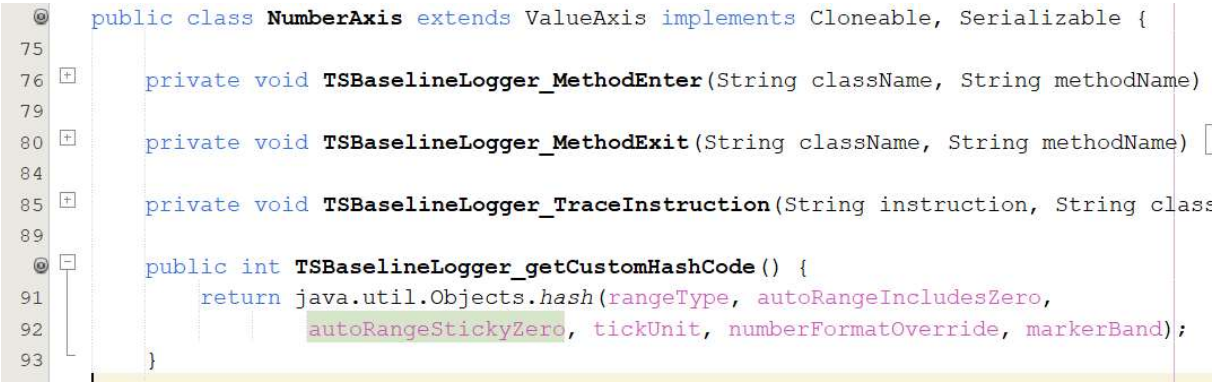

Figura 5-6 Exemplo de função de cálculo do estado do objeto gerada automaticamente (linhas 90 a 92) (Fonte: o autor)

5.2.3 Test smells removidos automaticamente

A implementação do aplicativo de refatoração automática que suporta o processo de refatoração descrito na seção 5.1 remove as ocorrências dos seguintes *smells*: Assertion Roulette (AR), Duplicate Assertion (DA), Eager Test (ET), Exception Catching Throwing (ECT) e Test Code Duplication (TCD).

A remoção do AR é realizada por meio da inclusão de uma constante que é usada como explicação nas afirmações em caso de erro na execução do teste descrita por Santana et al., 2020 e implementada na ferramenta RAIDE. Na refatoração do AR com a ferramenta RAIDE, conforme mencionado na Seção 2.8, o processo de remoção do *smell* é semiautomatizado, ou seja, necessita da intervenção de um

[3] A função hash da classe Object foi implementada na versão 1.7 do Java e, em outros contextos, pode ser substituída por outra função equivalente.

membro da equipe de desenvolvimento para executar a ferramenta e aprovar a refatoração.

Na Figura 5-7(a) é mostrado um teste afetado pelo *smell* AR e na Figura 5-7(b), um teste com a remoção do *smell*, por meio da adição da explicação conforme descrito por Santana et al., (2020).

Teste afetado pelo Assertion Roulette (A)

```
@Test
void primeiroTeste(){
    Calculadora c = new Calculadora()
    int resultadoEsperado = 10;
    int primeiraParcela = 11;
    int segundaParcela = -1;
    int resultado = c.somar(primeiraParcela, segundaParcela);
    assertEsquals(resultado, resultadoEsperado);
}
```

Teste com o *smell* removido (B)

```
@Test
void primeiroTeste(){
    Calculadora c = new Calculadora()
    int resultadoEsperado = 10;
    int primeiraParcela = 11;
    int segundaParcela = -1;
    int resultado = c.somar(primeiraParcela, segundaParcela, "Add assertion explanation");
    assertEsquals(resultado, resultadoEsperado);
}
```

Figura 5-7 Exemplos de testes afetados pelo smell AR (a) e com o smell removido (b)

A remoção do *smell* Redundant Assertion implementada na ferramenta Sentinel consiste na remoção das afirmações que foram identificadas como redundantes pelo JNose. Um exemplo de código afetado pelo *smell* RA pode ser observado na Figura 5-8 (a) e a versão do teste com as afirmações removidas na Figura 5-8 (b).

Teste afetado pelos smells Redudant Assert e Assertion Roulette (A)

```
@Test
public void testAfirmacoesRedundantes(){
    int a = 10;
    assertEquals(a, a);
    assertEquals(a, 10);
    int b = 10;
    assertEquals(a, b);
    assertEquals(b, b);
    assertEquals(b, 10);
}
```

Teste com os smells Redudant Assert e Assertion Roulette removidos (B)

```
@Test
public void testAfirmacoesRedundantes(){
    int a = 10;
    assertEquals(a, 10, "Add assertion explanation here");
    int b = 10;
    assertEquals(a, b, "Add assertion explanation here");
    assertEquals(b, 10, "Add assertion explanation here");
}
```

Figura 5-8 Exemplo de teste afetado *pelos* smells Redundant Assertion e Assertion Roulette (a) e com os *smells* removidos (b)

Na Figura 5-9 (a) é mostrado um teste que tem ocorrências dos *smells* Exception Catching Throwing (ECT) e Assertion Roulette, enquanto na Figura 5-9 (b) é apresentada a versão refatorada. Como pode ser observado, existem dois blocos de proteção de código, definidos por meio de instruções *try*/*catch*. No primeiro bloco, Figura 5-9 (a), é esperado que o estímulo realizado cause o lançamento de uma exceção a partir da execução da instrução "int resultado = calculadora2.dividir(dividendo, divisor1);". No segundo bloco *try*/*catch* mostrado na

Figura 5-9 (b), não é esperado que o estímulo realizado na mesma instrução lance uma exceção, ou seja, o teste foi projetado esperando que o estímulo não lance uma exceção e não se caracteriza como sendo uma ocorrência do *smell* ECT. Assim, as ocorrências do *smell* ECT identificadas precisam ser analisadas para identificar se é esperado ou não que a exceção seja lançada pelo conjunto de instruções do bloco *try*, pois a refatoração do segundo bloco *try* pode ocasionar erro no teste devido ao não lançamento da exceção esperada.

Teste afetado pelos *smells* Exception Catching Throwing e Assertion Roulette (A)

```
@Test
public void testaDivisoes(){
    final int dividendo = 10;
    final int divisor1 = 0;
    try{
        int resultado = calculadora2.dividir(dividendo, divisor1);
        fail("Deveria lançar ArithmeticException");
    } catch (ArithmeticException e) {
    }

    int divisor2 = 1;
    try{
        int resultado = calculadora2.dividir(dividendo, divisor2);
        assertEquals(resultado, dividendo);
    } catch (NumberFormatException e) {
        fail("Não deveria lançar a exceção");
    }
}
```

Teste com as ocorrências de Exception Catching Throwing e Assertion Roulette removidas (B)

```
@Test
public void testaDivisoes(){
    final int dividendo = 10;
    final int divisor1 = 0;
    assertThrows(ArithmeticException.class, () -> {
      int resultado = calculadora2.dividir(dividendo, divisor1);
    }, "Deveria lançar ArithmeticException");

    int divisor2 = 1;
    try{
        int resultado = calculadora2.dividir(dividendo, divisor2);
        assertEquals(resultado, dividendo, "Add assertion explanation here");
    } catch (NumberFormatException e) {
        fail("Não deveria lançar a exceção");
    }
}
```

Figura 5-9 Exemplo de teste afetado pelos *smells* Exception Catching Throwing e Assertion Roulette (a) e com os *smells* removidos (b)

A remoção dos *smells* Eager Test (ET) e Test Code Duplication (TCD) foi realizada por meio da extração das ocorrências dos *smells* para subclasses da classe de teste afetada pelos dois *smells*. A remoção do ET ocasiona a criação de uma classe de testes cujo nome é composto pelo nome da classe de testes original associada ao nome do método de teste afetado pelo ET. A remoção do TCD ocasiona a criação de uma subclasse na qual o nome da subclasse é composto do nome da classe original concatenado com um número sequencial para identificar o grupo de clones, como “clone1”, “clone2” e, assim, sucessivamente. Na implementação da remoção do TCD, os clones são agrupados de acordo com a quantidade de linhas de código que foram identificadas como clones entre os métodos de testes. Caso um método de teste faça parte de dois ou mais clones, ele será refatorado no grupo de clones que ele tenha a maior quantidade de linhas clonadas. Em ambos os casos, caso a classe original tenha um construtor, um construtor será criado nas subclasses para invocar o construtor da classe ancestral sempre que os testes extraídos forem executados.

No caso do *smell* Eager Test (ET), os testes são extraídos para métodos de testes próprios e a execução dos testes é ordenada por meio da adição da anotação “@Order” nos métodos de testes. Na Figura 5-10(a) é apresentado um teste afetado pelo ET e, na Figura 5-10(b), a versão refatorada, na qual as variáveis locais foram convertidas em atributos para serem acessíveis a todos os métodos de testes criados.

Teste afetado pelo Eager Test *smell* (A)

```
@Test
public void testSoma (){
    /*
    somas de números positivos
    */
    Calculadora calculadora1 = new Calculadora();
    int resultado = calculadora1.somar(10, 5);
    assertEquals(resultado, 15);

    resultado = calculadora1.somar(10, 3);
    assertEquals(resultado, 13, "O resultado da soma deveria ser 13");
}
```

Testes extraídos para métodos específicos (B)

```
protected Calculadora calculadora1;
protected int resultado;

@Test
@org.junit.jupiter.api.Order(1)
public void testSomaSubtracao_1(){
    /*
    somas de números positivos
    */
    Calculadora calculadora1 = new Calculadora();
    int resultado = calculadora1.somar(10, 5);
    assertEquals(resultado, 15);
}

@Test
@org.junit.jupiter.api.Order(2)
public void testSomaSubtracao_2(){
    resultado = calculadora1.somar(10, 3);
    assertEquals(resultado, 13, "O resultado da soma deveria ser 13");
}
```

Figura 5-10 Exemplo de teste afetado pelo Eager Test (a) e a respetiva versão refatorada (b)

No caso da remoção do Test Code Duplication (TCD ou clones), os métodos de testes identificados como contendo linhas de código repetidas são agrupados de acordo com a quantidade de linhas de código repetidas. Deste modo, os testes serão extraídos para a subclasse de testes na qual eles possuem a maior quantidade de linhas de código repetidas. Nas classes extraídas é criado um método de configuração (@BeforeEach) com as linhas de código repetidas entre os métodos de testes.

Na Figura 5-11 (a) é mostrado um conjunto de métodos de testes afetados por clones. Os métodos de testes “testDivisao1Clone” e “testDivisao2Clone” foram extraídos para a subclasse mostrada na Figura 5-11 (b). Os métodos de testes “testDivisao2Clone” e “testDivisao4Clone” foram extraídos para a subclasse mostrada em Figura 5-11 (c). Como pode ser observado na Figura 5-11 (a), a classe original contém um método de configuração (“testMethodSetup”). Por esse motivo, é criado um método de configuração nas subclasses extraídas (Figura 5-11, a e b) o qual invoca o método de configuração da superclasse e executa as instruções de configuração identificadas como clones.

Teste afetado pelo Test Code Duplication (clones)

```
@BeforeEach                                          (A)
public void testMethodSetup(){
    calculadora2 = new Calculadora();
}

@Test
public void testDivisao1Clone(){
    int dividendo = 10;
    int divisor = 2;
    assertEquals(calculadora2.dividir(dividendo, divisor), 5);
}

@Test
public void testDivisao2Clone(){
    int dividendo = 10;
    int divisor = -2;
    assertEquals(calculadora2.dividir(dividendo, divisor), -5);
}

@Test
public void testDivisao3Clone(){
    int dividendo = -10;
    int divisor = 2;
    assertEquals(calculadora2.dividir(dividendo, divisor), -5);
}

@Test
public void testDivisao4Clone(){
    int dividendo = -10;
    int divisor = -2;
    assertEquals(calculadora2.dividir(dividendo, divisor), 5);
}
```

Testes agrupados de acordo com o código duplicado

```
protected int dividendo;  (B)

@BeforeEach()
public void CalculadoraTest_clone1TestSetup() {
    testMethodSetup();
    dividendo = 10;
}

@Test
public void testDivisao1Clone() {
    int divisor = 2;
    assertEquals(calculadora2.dividir(dividendo, divisor), 5, "Add assertion explanation here");
}

@Test
public void testDivisao2Clone() {
    int divisor = -2;
    assertEquals(calculadora2.dividir(dividendo, divisor), -5, "Add assertion explanation here");
}
```

```
protected int dividendo;  (C)

@BeforeEach()
public void CalculadoraTest_clone2TestSetup() {
    testMethodSetup();
    dividendo = -10;
}

@Test
public void testDivisao3Clone() {
    int divisor = 2;
    assertEquals(calculadora2.dividir(dividendo, divisor), -5, "Add assertion explanation here");
}

@Test
public void testDivisao4Clone() {
    int divisor = -2;
    assertEquals(calculadora2.dividir(dividendo, divisor), 5, "Add assertion explanation here");
}
```

Figura 5-11 Exemplo de testes com código duplicado (a) e a versão refatorada (b e c)

5.3 Quase-experimento: capacidade da remoção automática de smells

Nesta seção, são apresentados os resultados de um quase-experimento para avaliar a viabilidade de validar automaticamente remoções automáticas de TS por meio de refatorações. Para orientar o quase-experimento mostrado nesta seção, foi criada a seguinte questão de pesquisa: Qual a capacidade de manutenção do comportamento dos testes após a remoção de *smells* por meio da refatoração automática?

Para responder à questão de pesquisa foram definidas as seguintes hipóteses:

- Hipótese nula ($H1_0$): o comportamento do teste após a remoção automática do ET é mantido em menos de 49% dos testes refatorados.
- Hipótese alternativa ($H1_1$): o comportamento do teste após a remoção automática do ET é mantido em 49% ou mais dos testes refatorados.
- Hipótese nula ($H2_0$): o comportamento do teste após a remoção automática do TCD é mantido em menos de 49% dos testes refatorados.

- Hipótese alternativa ($H2_1$): o comportamento do teste após a remoção automática do TCD é mantido em 49% ou mais dos testes refatorados.

As hipóteses foram avaliadas a partir da quantidade média de refatorações realizadas e validadas, ou seja, refatorações que não causaram erros de compilação, erro na execução de testes ou alterações no comportamento do software. O percentual de 49% foi definido com base na pesquisa de Alcocer et al., (2020) que identificaram que refatorações manuais que são baseadas na extração de código apresentam taxa de erro de 49%. A extração de código do método de teste é uma operação de refatoração que pode ser empregada tanto na remoção do ET quanto na do TCD. No caso da remoção do ET, os trechos extraídos são convertidos em métodos de teste, enquanto na remoção do TCD os trechos extraídos são convertidos em métodos de configuração, em consonância com as etapas para a criação dos testes de unidade descritos por Rompaey et al., (2007) e apresentados na seção 2.1.

5.3.1 Planejamento

Foram selecionados vinte projetos de código aberto, escritos na linguagem Java, utilizando o *framework* de testes JUnit 5 e o gerenciador de *build* Maven. As configurações do projeto foram alteradas para remover *plugins* e configurações relacionadas à implantação automática e realização de checagens durante a construção do projeto, como estilo de código e informações de licenciamento. Os projetos foram compilados e os testes que apresentaram erros de execução foram removidos, sendo que após a compilação dos testes sem erros, os testes foram considerados determinísticos.

Após compilação sem erros, os *smells* foram identificados nos projetos utilizando o detector JNose (VIRGINIO et al., 2020), a implementação de Pizzini et al., (2022) para o *smell* ET e a proposta de Silva et al., (2020) para o *smell* TCD. As ocorrências de ET e CD foram refatoradas automaticamente nos métodos de teste não afetados por Mystery Guest, Resource Optimism e Sleepy Test, os quais foram identificados pelo JNose e que caracterizam os métodos de testes como não determinísticos.

A validação automática da refatoração foi realizada por meio de: a) identificação de erros na compilação das classes de teste refatoradas; b) identificação de erros na

execução dos testes refatorados; e c) comparação do comportamento e dos estados do software nas versões não refatorada e refatorada dos testes de unidade.

5.3.2 Resultados

A Tabela 5-1 apresenta os projetos selecionados e, para os *smells* ET e TCD, a respectiva quantidade de ocorrências identificadas (Ident.), a quantidade de ocorrências de *smells* refatoradas (Refat.), a quantidade e o percentual de ocorrências de *smells* validadas com sucesso (Validados), o tempo médio de refatoração (Tempo Refat.) e o tempo médio de validação (Tempo Valid.), ambos em segundos.

Em relação ao ET, das 3.529 ocorrências identificadas, 3.290 foram refatoradas, das quais 2.064 não causaram erros de compilação, erros na execução dos testes e nem mudança de comportamento do software após a refatoração, o que corresponde a uma taxa de 62,7%. Assim, a hipótese nula $H1_0$ foi rejeitada e a hipótese alternativa $H1_1$ foi aceita.

No caso do TCD, das 1.589 ocorrências identificadas, 1.458 foram refatoradas, das quais 1.347 não causaram erros de compilação, erros na execução dos testes e nem a mudança de comportamento do software após a refatoração, o que corresponde a uma taxa de 92,1%. Assim, a hipótese nula $H2_0$ foi rejeitada e a hipótese alternativa $H2_1$ foi aceita.

Tabela 5-1 Projetos selecionados para o quase-experimento e dados coletados (Fonte: o autor)

Projeto	Eager Test						Test Code Duplication					
	Ident.	Refat.	Validados		Tempo Refat.	Tempo Valid.	Ident.	Refat.	Validados		Tempo Refat.	Tempo Valid.
			Quanti.	%					Quanti	%		
Avro	225	179	172	96,1	7,1	8,1	85	71	67	94,4	5,4	20,0
Classgraph	3	1	0	0,0	3,0	9,1	6	5	0	0,0	3,2	32,5
Dropwizard							54	52	47	90,4	3,7	13,9
Jasypt												
JavaParser	245	241	224	92,9	4,8	4,8	220	213	210	98,6	4,2	13,4
Jetcache	47	46	43	93,5	6,0	4,5	5	5	5	100,0	3,6	9,9
Jfreechart	1047	1047	252	24,1	5,1	7,2	278	278	232	83,5	2,6	23,8
Jsoup	300	286	2	0,7	6,4	13,6	44	43	2	4,7	4,1	32,6
Logback	191	184	163	88,6	5,7	7,1	27	26	24	92,3	3,9	18,1
OpenPDF	14	7	7	100,0	5,6	10,6	14	13	13	100,0	4,3	23,7
Poi-tl	41	36	2	5,6	4,5	12,4						
Recaf	12	12	2	16,7	4,6	9,2	1	1	1	100,0	3,0	22
Redisson	395	332	332	100,0	5,5	18,4	479	430	430	100,0	3,2	50,4
Spring-batch	726	702	652	92,9	4,1	3,2	134	120	118	98,3	2,3	8,5
Spring-cloud-netflix							9	9	9	100,0	4,3	8,0
Spring-data-jpa							35	35	35	100,0	2,8	9,9
Springdoc-openapi												
Tablesaw	132	129	125	96,9	7,0	5,5	75	74	71	95,9	5,6	17,7
Thymeleaf	83	24	24	100,0	2,4	3,6	66	29	29	100.0	3.0	22,8
Unirest-java	68	64	64	100,0	3,3	4,1	49	47	44	93.6	1.9	9,4
Total	3529	3290	2064	62,7	-	-	1589	1458	1347	92.1	-	-

5.3.3 Discussão dos resultados

Os resultados deste quase-experimento sugerem que a remoção automática de *smells* das classes de testes de unidade é viável quando são comparadas às taxas de erros, pois as taxas de erros para a remoção automática dos *smells* ET e TCD foram inferiores às das refatorações manuais baseadas na extração de código identificadas por Alcocer et al., (2020).

No que diz respeito à validação automática, o tempo necessário para validar as refatorações é maior que o tempo necessário para executá-las, devido à necessidade de construir o projeto para cada validação, o que envolve compilação e execução de todos os testes do projeto, além da execução dos testes refatorados para identificar alterações de comportamento do software. Assim, balancear a quantidade de refatorações que precisam ser validadas pode melhorar o desempenho geral da refatoração automática. Mesmo assim, a remoção automática de *smells* tende a ser mais rápida do que as remoções manuais ou semiautomáticas, pois a identificação de mudança de comportamento do software depende da análise das instruções executadas a partir de um estímulo realizado por um teste (que depende da compilação e/ou construção do projeto), bem como da identificação da mudança de estado do software, a qual depende da análise das instruções executadas para cada estímulo realizado. A remoção manual de *smells* pode ser realizada sem a devida identificação da mudança de comportamento do software. Nessa situação, a não identificação do comportamento após a refatoração pode ser causada por fatores como a *expertise* do membro da equipe responsável por remover a ocorrência do *smell* que é capaz de refatorar o código sem causar mudanças de comportamento, falta de conhecimento, negligência, entre outros.

Os dados coletados sugerem que a remoção de clones produz menos mudanças no comportamento do teste do que as remoções de TS baseadas na extração de código, como Eager Test (ET) e possivelmente Duplicate Assert (DA), cuja remoção é baseada na extração dos trechos de código que causam o *smell* (SANTANA et al., 2020). As refatorações baseadas na extração de código podem analisar a dependência das seções extraídas para a) manter a ordem de execução das instruções dos métodos criados; e b) remover dependências entre testes. No entanto, remover dependências de testes é complexo e apenas cerca de 12,5% das ocorrências podem ser removidas automaticamente (LAM, 2016).

A próxima etapa da pesquisa consiste na realização de um quase-experimento para avaliar a variação das ocorrências de *smells* após a remoção do conjunto de *smells* implementados no Sentinel e descritos na seção 5.2.3.

5.4 Quase-experimento: identificação da variação nas ocorrências de smells após a remoção automática

Esta seção apresenta um estudo sobre a variação na quantidade de ocorrências de *smells* após a remoção automática dos *smells* que são removidos automaticamente pelo Sentinel. Nesse estudo, a validação das refatorações não foi realizada, pois foi adotada a premissa que a remoção dos *smells* em questão podem ser realizadas, manual ou automaticamente, sem causar os erros identificados no estudo exploratório descrito na Seção 4.3.

5.4.1 Planejamento

Para realizar a refatoração, foram selecionados trinta e três (33) projetos de código aberto do repositório público GitHub, escritos na linguagem Java e que usam o *framework* de testes JUnit 5. Os projetos foram limpos, ou seja, os resultados de construções anteriores foram eliminados, e os seguintes TS foram refatorados individualmente: Assertion Roulette (AR), Exception Catching Throwing (ECT), Eager Test (ET), Redundant Assertion (RA) e Test Code Duplication (TCD).

As seguintes ações foram realizadas para coletar os dados desse quase-experimento: a) identificar as ocorrências de TS antes da refatoração; b) refatorar as classes de testes sem realizar a validação das refatorações realizadas; e c) identificar as ocorrências de *smells* após a refatoração. Os métodos de testes afetados por MG, RO ou ST não foram refatorados por serem considerados não determinísticos.

A coleta de dados da quantidade de ocorrências de cada *smell* nas classes de testes foi realizada usando o detector JNose (VIRGINIO et al., 2020), a implementação de Pizzini et al., (2022) para o ET e a proposta de Silva e Vilain (2020) para o TCD, a qual compara as linhas de código dos métodos de testes para identificar a maior sequência duplicada de linhas de código dos testes.

Os projetos e as quantidades de ocorrências de cada *smell* são mostrados na Tabela 5-2. A refatoração foi realizada em uma cópia do projeto original, e os projetos não foram compilados antes ou depois da refatoração.

Tabela 5-2 Quantidade de ocorrências identificadas de cada *smell* em cada projeto antes da refatoração (Fonte: o autor)

Projeto	AR	CI	DA	ET	EmT	ECT	GF	LT	MNT	MG	OS	RA	RO	SE	ST	TCD	VT
Avro	1.928	3	76	225	4	174	32	78	527	94	21		155	148	6	121	112
Classgraph	13		31	2	1	169		14	9	2	1		3	37		7	18
Cucumber	609		20	46	3	10	5	438	116	4	1	4	17	42		137	66
Disruptor	135	1	5	21		46	7	140	70						2	13	5
Dropwizard	31	1	200			100	34	413	1160	9			12	53	11	54	34
Fastjson2	19.750	7	954	1198	5	252	22	1104	5682	16	348	47	16	1137		675	503
Graphhopper	4.984	11	430	524		17	14	367	3431	2	2	2	21	246		105	270
Jasypt	58					1		2			15			4			
Javaparser	4.436	6	145	245		38	7	419	610	93	10	1	22	195		261	97
JDA	379		3	8				7	29					36		1	5
Jetcache	469		52	50		6	1	34	67			42			4	6	10
Jfreechart	9.319		972	1047		361		1167	2669			53		5		553	220
Jodd	4.101		427	317		72	4	403	1422	9	9	2	9	135		113	110
Jsoup	3.159		90	312		34	1	186	627	36		9	36	95	2	73	31
Liquibase	1.089	16	33	80	9	46	15	188	175	2	2	17	4	42		75	47
Logback	1.537	1	70	196	2	54	53	233	339	46	55		36	41	11	32	36
OpenPDF	189	2	4	14		50		24	101	15		2	15	11		14	15

Continua na próxima página

Continuação

Projeto	AR	CI	DA	ET	EmT	ECT	GF	LT	MNT	MG	OS	RA	RO	SE	ST	TCD	VT
Poi	417		31	43	1	5		15	95	12	3		18	20		3	35
Recaf	434		23	12		23		15	101		7					1	6
Redisson	2217		655	395		153	84	882	1406	4	44		4	60	489	578	239
RoaringBitmap	6410		435	466		12		546	2900	4	140		4	21		376	183
Rsocket	150		115	37		8	7	156	154		22			7	3	39	89
Simple	317		21	23		10	2	27	298	1			1	21		19	27
Simplify	409		6	28			24	51	72	3	1		1	30		5	4
Spring-batch	6159	19	158	725	2	61	104	1614	1767	18	4	11	11	244	20	173	174
Spring-cloud			24		1	2	1	41						2		9	2
Spring-data			46		31	7	55	16	54					3	1	35	15
Springdoc	43		8		1	8	1	5	1			6		2	2		3
Sqlite			285		2	58	6	23	610	19	16		12	3	3	13	47
Tablesaw	2779		54	136		16	14	286	1928	12	20		21	43		92	47
Thymeleaf	5048	8	706	83		93		108	368	324	1	226	2	897		80	105
Unirest	903		17	68		14	3	162	336	21	8	18	13	46		53	1
Wiremock	548	1	29	60	1	49	2	363	638	21	49	2	4	94		110	44
Total	78020	76	6125	6361	63	1949	498	9527	27762	767	779	442	437	3720	554	3826	2600

5.4.2 Avaliação e Resultados

A avaliação do impacto da remoção de TS nas ocorrências de outros *smells* foi realizada por meio de um quase-experimento para avaliar as seguintes hipóteses:

- Hipótese nula ($H1_0$): A variação na quantidade dos TS_s (TS_{x1}, TS_{x2}, TS_{x3}, ... TS_{xn}) independe da quantidade removida de um TS_y;
- Hipótese alternativa ($H1_1$): Existe uma correlação positiva ou negativa entre a variação na quantidade do TS_x e a quantidade removida do TS_y.

Para avaliar as hipóteses, a classe de teste foi escolhida como variável independente. Foram definidas duas variáveis quantitativas discretas como variáveis dependentes: a quantidade removida de cada um dos TS removidos pela ferramenta Sentinel (R_{AR}, R_{ECT}, R_{ET}, E_{RA}, R_{TCD}) e a variação na quantidade de ocorrências dos demais TS no repositório refatorado (TS_{AR}, TS_{CI}, TS_{CTL}, TS_{DA}, ... TS_{VT}).

Para avaliar as hipóteses, foi executado o teste de postos de Spearman com coeficiente de significância de 5%, bicaudal. A correlação positiva indica que a remoção de TS_y causa a remoção de ocorrências de outro TS (TS_{xn}), enquanto a correlação negativa indica que a remoção do TS_y causa a criação de ocorrências de outro TS (TS_{xn}). A correlação foi realizada analisando todas as classes de testes que tiveram o TS_y removido.

Na Tabela 5-3 são apresentadas as quantidades identificadas de cada *smell* por projeto antes da refatoração, e os *smells* que não foram identificados em nenhum projeto foram omitidos. O TS com maior ocorrência foi o AR, com 76.807 ocorrências, seguido de MNT, com 27.762, LT, com 9.527, ET, com 6.361, DA, com 6.125, TCD, com 3.826. A quantidade de ocorrências de AR confirma os achados descritos na literatura, de que o AR é um dos três mais difusos. Entretanto, nos repositórios considerados, a quantidade identificada de ocorrências de ET e TCD não confirmaram os achados da literatura.

Na Tabela 5-4 são apresentados o coeficiente de correlação de Spearman (ρ) e o nível de significância da correlação (*p-value*) dos TS refatorados. Na primeira coluna, são apresentados os TS refatorados com a indicação do total de remoções realizadas (N=número). Para não prejudicar a visualização da tabela, os TS cujas quantidades não variaram com a refatoração foram omitidos. Além disso, somente as casas decimais são mostradas, omitindo-se o zero antes da vírgula, com o sinal – para indicar números negativos ou com o símbolo < para representar valores menores que

o exibido à direita do símbolo. As células com cor de fundo cinza indicam as correlações com nível de significância (*p-value*) dentro da margem estabelecida de 95% (*p-value* menor que 0,05).

Considerando os resultados apresentados na Tabela 5-4, a hipótese nula ($H1_0$) foi rejeitada, sendo aceita a hipótese alternativa ($H1_1$) para as seguintes situações:

- Na remoção de AR (R_{AR}), para as variações dos *smells* MNT (TS_{MNT}) e VT (TS_{VT});
- Na remoção de ECT (R_{ECT}), para as variações dos *smells* DA ($TS_{D}A$), ET (TS_{ET}) e VT (TS_{VT});
- Na remoção de ET (R_{ET}), para as variações dos *smells* DA (TS_{DA}), LT (TS_{LT}), TCD (TS_{TCD}) e VT(T_{SVT});
- Na remoção de RA (R_{RA}), para as variações dos *smells* AR (TS_{AR}) e DA (TS_{DA});
- Na remoção de TCD (R_{TCD}), para as variações dos *smells* DA (TS_{DA}), LT (TS_{LT}), MNT (TS_{MNT}) e VT (TS_{VT}).

A interpretação dos valores de correlação obtidos para identificar a força da correlação foi realizada usando a escala definida por Hinkle et al., (2003), conforme mostrado Tabela 5-3.

Tabela 5-3 Interpretação dos valores de correlação (Fonte: Traduzido de Hinkle et al., 2003, p.109)

Valor de correlação	Força da correlação
De 0,90 até 1,00 (-0,90 até -1,00)	Correlação positiva (negativa) muito alta
De 0,70 até 0,90 (-0,70 até -0,90)	Correlação positiva (negativa) alta
De 0,50 até 0,70 (-0,50 até -0,70)	Correlação positiva (negativa) moderada
De 0,30 até 0,50 (-0,30 até -0,50)	Correlação positiva baixa (negativa)
De 0,00 até 0,30 (0,00 até -0,30)	Correlação insignificante

Tabela 5-4 Coeficiente de correlação (*ρ*) e significância (*p-value*) do teste Spearman (destaque para as correlações significantes encontradas) (Fonte: o autor)

TS removido	Dado coletado	TS_{AR}	TS_{CI}	TS_{DA}	TS_{EmT}	TS_{ET}	TS_{GF}	TS_{LT}	TS_{MNT}	TS_{PS}	TS_{SE}	TS_{TCD}	TS_{VT}
R_{AR} (N=4398)	ρ	-							0,053				0,279
	p-value	-							<0,001				<0,001
R_{ECT} (N=84)	ρ	0,089		0,618		-0,469			0,089				0,252
	p-value	0,422		<0,001		<0,001			0,419				0,021
R_{ET} (N=2023)	ρ		-0,014	0,362		-	-0,017	-0,182	0,040			0,348	0,269
	p-value		0,538	<0,001		-	0,440	<0,001	0,069			<0,001	<0,001
R_{RA} (N=76)	ρ	0,932		0,247	0,050	0,078	-0,042		0,078				-0,014
	p-value	<0,001		0,031	0,667	0,503	0,717		0,505				0,907
R_{TCD} (N=1648)	ρ	0,047	-0,018	0,086	-0,103		0,019	-0,104	0,089	-0,020	0,038	-	0,250
	p-value	0,057	0,476	<0,001	<0,001		0,450	<0,001	<0,001	0,409	0,121	-	<0,001

A seguir são apresentados os gráficos de dispersão para as correlações, sejam positivas ou negativas, de força baixa, moderada, alta e muito alta. Os gráficos de correlação de força insignificante são apresentados no Apêndice G.

Na Figura 5-12 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ECT e a variação na quantidade de ocorrências de DA. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva moderada encontrada (*p-value*=<,001 e ρ=,618), indicando que a remoção de ECT causou a remoção de ocorrências de DA.

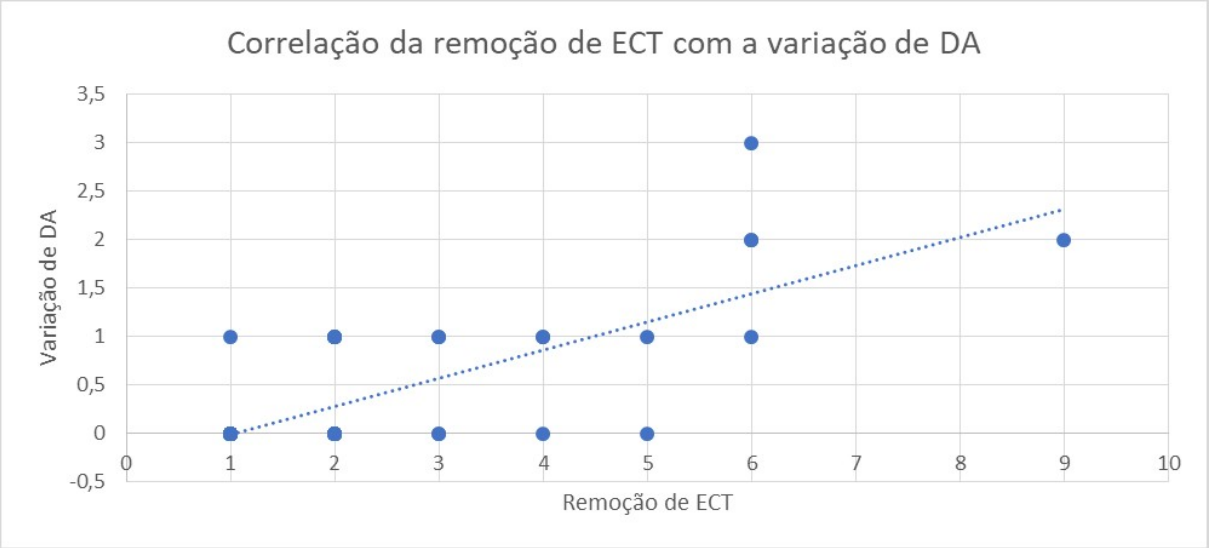

Figura 5-12 Gráfico de dispersão entre a remoção de ECT com a variação de DA (Fonte: o autor)

Na Figura 5-13 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ECT e a variação na quantidade de ocorrências de DA. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa baixa causada pela remoção do ECT (*p-value*=<,001 e ρ=,469), na qual a remoção de ECT causa a criação de ocorrências de ET, indicando que a remoção de ECT causou a criação de ocorrências de ET.

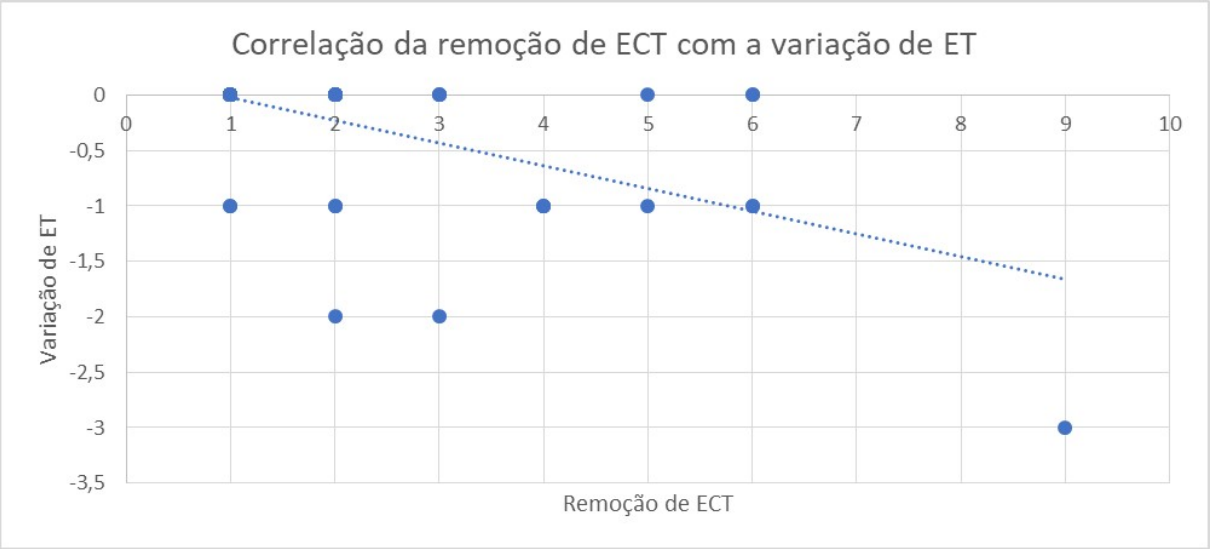

Figura 5-13 Gráfico de dispersão entre a remoção de ECT com a variação de ET (Fonte: o autor)

Na Figura 5-14 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ET e a variação na quantidade de ocorrências de DA. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva baixa causada pela remoção do ET em relação à variação da quantidade de ocorrências de DA (*p-value*=<,001 e ρ=,362), indicando que a remoção de ET causou a remoção de ocorrências de DA.

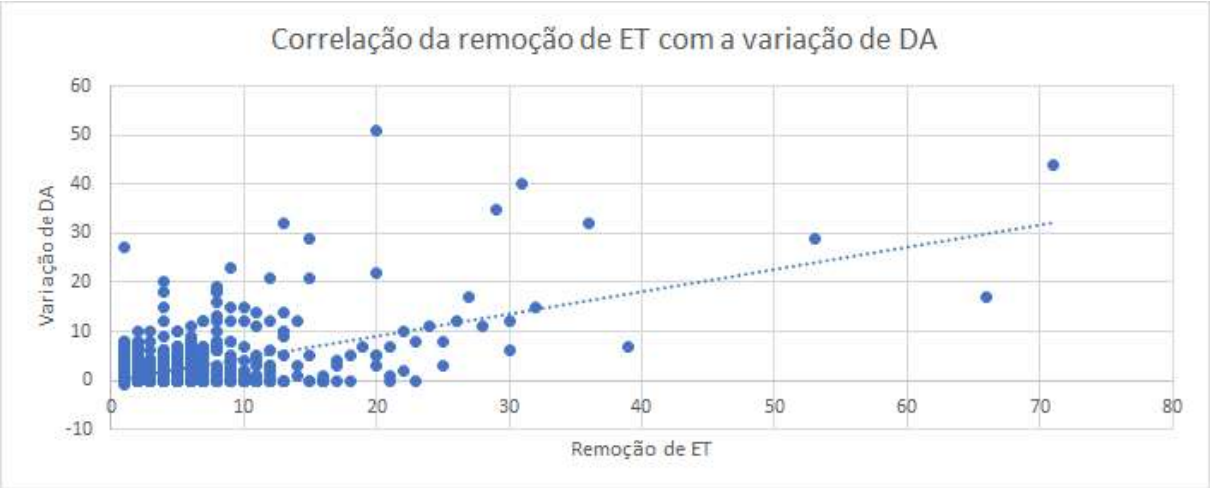

Figura 5-14 Gráfico de dispersão entre a remoção de ET com a variação de DA (Fonte: o autor)

Na Figura 5-15 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ET e a variação na quantidade de ocorrências de TCD. O agrupamento dos pontos da correlação e a linha linear de

tendência corroboram com a correlação positiva baixa entre a remoção do ET e a variação da quantidade de ocorrências de TCD (*p-value*=<,001 e ρ=,348), o que mostra que a remoção de ET causa a remoção de ocorrências de TCD.

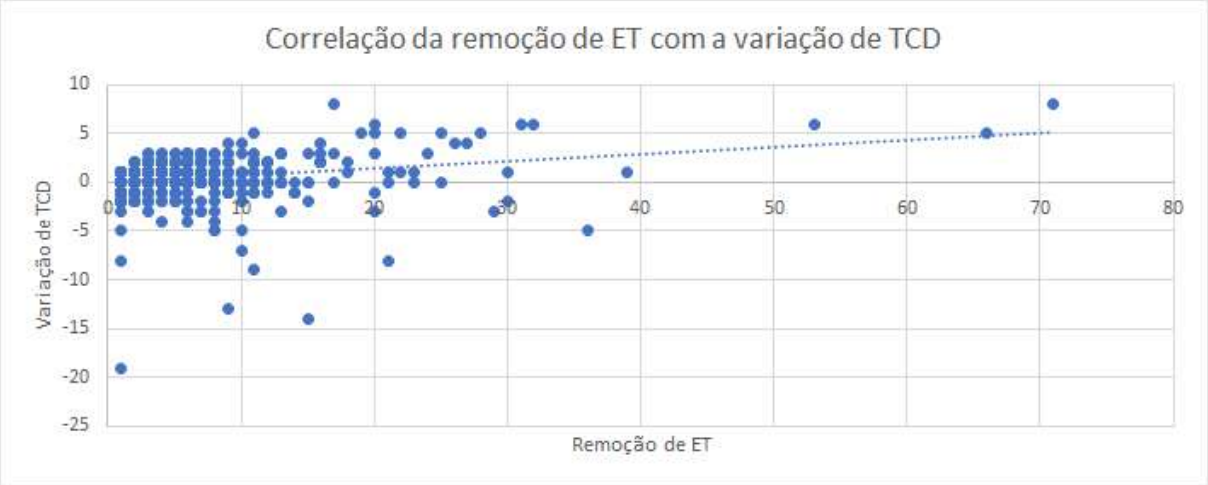

Figura 5-15 Gráfico de dispersão entre a remoção de ET com a variação de DA (Fonte: o autor)

A Figura 5-16 mostra o gráfico de dispersão da correlação entre a quantidade de RA removidos e a variação na quantidade do AR, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva alta entre a remoção do AR e a variação da quantidade de ocorrências de RA (*p-value*=<,001 e ρ=,932) e mostra que a remoção de RA causou a remoção de ocorrências de AR.

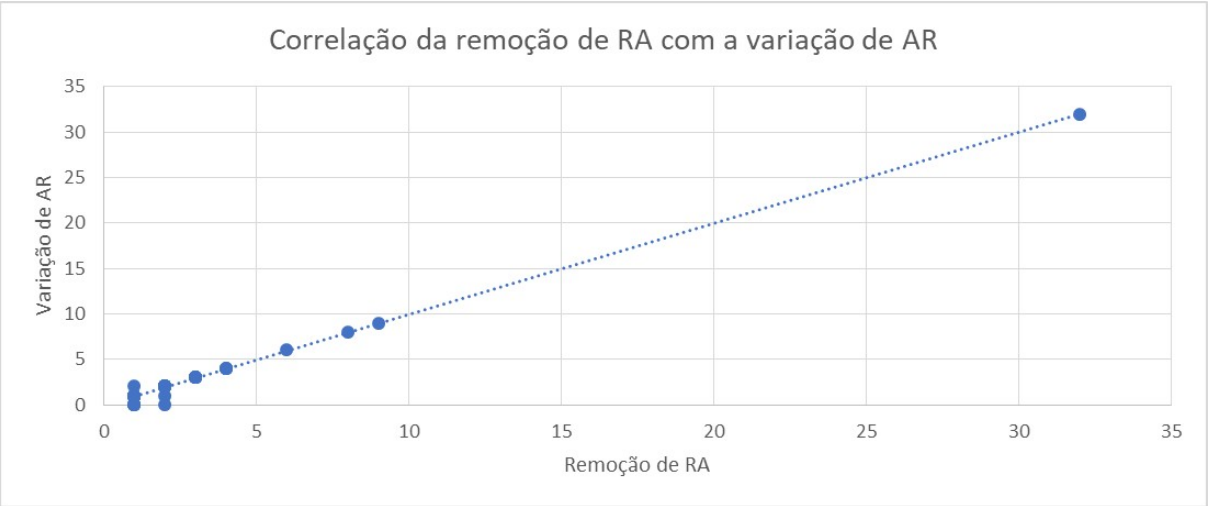

Figura 5-16 Gráfico de dispersão entre a remoção de RA e variação de AR (Fonte: o autor)

Na próxima seção, as correlações identificadas e o impacto da refatoração nos demais TS são discutidos.

5.4.3 Discussão dos resultados do quase-experimento

O primeiro ponto de discussão dos resultados é uma análise da remoção dos TS selecionados, considerando, especificamente, a variação de MNT e VT identificadas na remoção de todos os *smells* selecionados neste quase-experimento. A identificação de MNT e de VT é influenciada pela posição e quantidade de linhas de código das afirmações, no caso do MNT, e de todo o método de teste, no caso do VT. A remoção automática de TS pode alterar o formato do arquivo de teste, removendo linhas em branco e colocando os argumentos de afirmações na mesma linha de código.

A remoção do MNT não está associada à remoção de AR, ECT, ET e TCD. A remoção de AR implica na adição de uma explicação para o erro da execução do teste, sem remover ou alterar trechos do código existente, o que, em tese, poderia criar ocorrências de VT por causa da adição de código ao teste. No caso das remoções de ECT, ocorre a transformação do código em uma afirmação, sem alterar as instruções executadas no código não refatorado. A remoção de ET por meio da extração dos trechos de código de cada teste para métodos específicos não impacta nas afirmações dos testes e a remoção de TCD não considera as afirmações como código duplicado, por estar restrita a identificação de igualdades entre as instruções de configuração e estímulos. A redução de MNT pode estar correlacionada à remoção de RA devido à remoção da afirmação redundante do método de teste, a qual remove a afirmação e as contantes referenciadas na afirmação removida.

Em relação ao VT, a correlação com a remoção de AR não faz sentido, pelo fato de, como mencionado anteriormente, a remoção do AR implicar no aumento da quantidade de código no método de teste. Assim, a remoção do VT somente pode ocorrer por meio da remoção de ECT, ET, RA e TCD, as quais reduzem a quantidade de linhas de código dos métodos de teste.

Em relação às correlações identificadas, a remoção do AR não está correlacionada com a variação na quantidade de ocorrências de MNT e de VT, conforme descrito acima, pois as correlações identificadas decorrem da mudança de formatação das classes de testes, como a remoção de linhas em branco, na mudança de formato das instruções de uma declaração, entre outras.

No caso da remoção do ECT, além do benefício de reduzir a complexidade ciclomática do teste, ocorre a redução do tamanho do método de teste, o que contribui

para a remoção do VT. A redução de tamanho do teste é resultado da substituição de parte das instruções do trecho de código afetado pelo ECT por uma afirmação.

Outro benefício da remoção do ECT é a remoção do DA, devido à substituição de partes do trecho do código original por uma afirmação na qual as partes do código removido estão implícitas, como a identificação da exceção lançada pelo código sendo testado e a falha do teste. Esta substituição de código também contribui para a remoção do AR, por remover afirmações que são usadas na detecção do AR, como a instrução fail. Entretanto, a remoção do ECT pode estar associada à criação de AR nos casos em que o teste a ser refatorado não possui uma afirmação que verifique o retorno. Ainda sobre a remoção do ECT, a substituição do trecho de código afetado pelo *smell* por uma afirmação está correlacionada com a criação de ET, cuja detecção se baseia na existência de afirmações para determinar o término das instruções de um teste.

A extração de testes afetados pelo ET em classes de testes causa a criação de ocorrências de CI e LT. As ocorrências de CI são criadas por causa da necessidade de inserir construtores nas subclasses extraídas quando existirem construtores na classe não refatorada. A criação de ocorrências de LT é ocasionada pela existência de testes da mesma unidade que fazem parte do método de teste refatorado, ou seja, os testes de uma unidade estão agrupados em um único método. Entretanto, convém ressaltar que a detecção do LT é baseada na identificação das unidades referenciadas/invocadas pelos testes, sem considerar o caminho das instruções executadas na unidade.

A extração dos testes para métodos específicos na remoção de ET está correlacionada com a remoção de DA e VT, além de remover ocorrências de GF. A remoção de DA e VT sugere que estes *smells* são, em parte dos casos, causados pela existência de mais de um teste devido a criação de testes para a mesma unidade dentro do mesmo método de teste. Assim, a correlação da remoção de ET com a remoção de DA e de VT pode ser incorporada como uma regra na priorização da refatoração, pois ambas as remoções podem ser baseadas na extração dos trechos de código para métodos específicos. A remoção de GF ocorre nas situações em que o método que causa o TS é extraído para uma subclasse, na qual as variáveis declaradas nos testes são transformadas em atributos e fazendo com que a *fixture* seja usada na totalidade pelos testes.

A remoção de ET está correlacionada com a variação de TCD não apenas por causa da remoção de TCD, mas também por causa da sua criação. A extração dos trechos de código de cada teste para métodos específicos em uma subclasse faz com que a similaridade dos testes contidos no método refatorado possam ser analisadas na subclasse, favorecendo a detecção de TCD, enquanto reduz a quantidade de TCD identificados por não realizar a comparação entre subclasse criada com as demais classes de testes.

A remoção de RA por meio da retirada da afirmação redundante do código pode causar a mudança de comportamento dos testes, como nos casos em que a redundância ocorre por causa da invocação de métodos que alteram o estado de objetos a cada invocação.

A remoção de RA está correlacionada com a remoção de DA e ET, além de impactar nas ocorrências de outros TS, como remover AR e criar EmT. Em relação ao AR, DA e ET, a detecção deles considera as afirmações, que são o enfoque da refatoração do RA. Em relação ao EmT, foram criadas ocorrências deste *smell* em 17 classes de testes, das quais 16 tiveram a remoção de ET, sugerindo que os métodos de testes refatorados tinham somente a afirmação que foi retirada do código.

A variação na quantidade de ocorrências de AR e EmT por causa da remoção de TCD ocorre devido ao uso de afirmações de *frameworks* de testes diferentes do JUnit, como Hamcrest, que foram identificadas como invocações de métodos e não como afirmações. Com isso, as afirmações do método de teste não foram identificadas e foram junto com o código de configuração para a classe de teste extraída, reduzindo a quantidade de ocorrências e AR e causando a criação de ocorrências de EmT.

Do mesmo modo que ocorreu com a remoção de ET, o impacto causado pela remoção de TCD na variação de CI ocorreu com a criação de ocorrências do *smell* devido a necessidade de criar construtores nas subclasses extraídas quando a classe não refatorada tem um ou mais construtores.

O impacto causado nas ocorrências de GF, PS e SE foi causado pela remoção de ocorrências destes *smells*, tendo sido removidas, respectivamente, 4, 1 e 1 ocorrências. A remoção de GF foi causada pela criação da subclasse, a extração das variáveis declaradas no método de teste para atributos e a inicialização dos atributos no método de configuração. As ocorrências de PS e SE foram removidas por serem duplicações de código nos métodos de testes que passaram a fazer parte do método de configuração. Ou seja, no caso de PE e SE, o código afetado permanece na classe

de teste, mas em menor quantidade de ocorrências quando o *smell* estiver contido no trecho de código que também é afetado por TCD.

A remoção de TCD está correlacionada com a variação de DA e de LT. No caso de DA, identificou-se a remoção de ocorrências de DA por causa de afirmações de outros frameworks não contidos no escopo deste trabalho e que, por isso, não foram identificadas corretamente. Com isso, as afirmações passaram para o método de configuração indicando que os testes refatorados tinham, além da configuração e estímulo, duplicações nas instruções da etapa de verificação. No caso do LT, identificou-se que ocorre a remoção e a criação de ocorrências do *smell* com a remoção de TCD. A remoção de LT é resultado da mudança das instruções de configuração e estímulos duplicados para o método de configuração na classe de teste extraída. Entretanto, não são criadas ocorrências de LT durante a remoção de TCD, pois os clones são extraídos para uma subclasse. Nestes casos, a identificação de LT gera uma quantidade de ocorrências diferente nas versões não refatorada e refatorada da classe de testes.

Na literatura podem ser encontrados estudos que identificam a relação entre operações de refatoração e os *smells* (PERUMA et al., 2020b), que propõem refatorações para remover ocorrências de *smells* (SOARES et al., 2022), e investigam o impacto da refatoração dos *smells* a partir das perspectivas dos desenvolvedores e dos atributos internos de qualidade (DAMASCENO et al., 2022). Neste estudo, identificou-se que a remoção de *smells* pode ocasionar a criação de ocorrências de outros *smells*, bem como a remoção de *smells* que afetem o mesmo trecho de código que foi refatorado.

A remoção de um TS de um nível de granularidade específico (como uma linha de código do teste) pode impactar nas ocorrências de TS do mesmo nível de granularidade, como uma afirmação, ou dos níveis maiores (como um teste). Por exemplo, a remoção do ECT reduz a quantidade de linhas de código do método de teste e está correlacionada com a remoção de VT, pois a quantidade de linhas no método de teste é uma métrica usada na identificação do VT. Outro exemplo é a remoção de DA que está correlacionada com a remoção do AR, pois ambos os *smells* afetam as afirmações do método de teste e podem estar associados à mesma afirmação.

Assim, a remoção de um *smell* pode impactar no processo de refatoração. A primeira etapa do processo que é afetada é a priorização das refatorações, na qual é

necessário definir a sequência de refatorações que serão executadas na classe de teste. Assim, identificar quais *smells* afetam os mesmos trechos de código pode contribuir para evitar que refatorações sejam executadas sem necessidade, ou que a execução de uma refatoração seja realizada em uma versão do código desatualizada, o que pode ocasionar problemas como erros de compilação, erro na execução dos testes ou mudança de comportamento do software. Portanto, para evitar que as refatorações sejam executadas em versões desatualizadas do código, pode ser necessário executar a identificação de *smells* por mais de uma vez durante a refatoração de uma classe de testes.

Na Figura 5-17 são mostrados os relacionamentos entre *smells* causados por causa da remoção ou criação de ocorrências a partir da remoção automática descritos anteriormente nesta seção. Em verde, estão destacados os *smells* removidos automaticamente, enquanto as setas descrevem os efeitos da remoção nos demais TS.

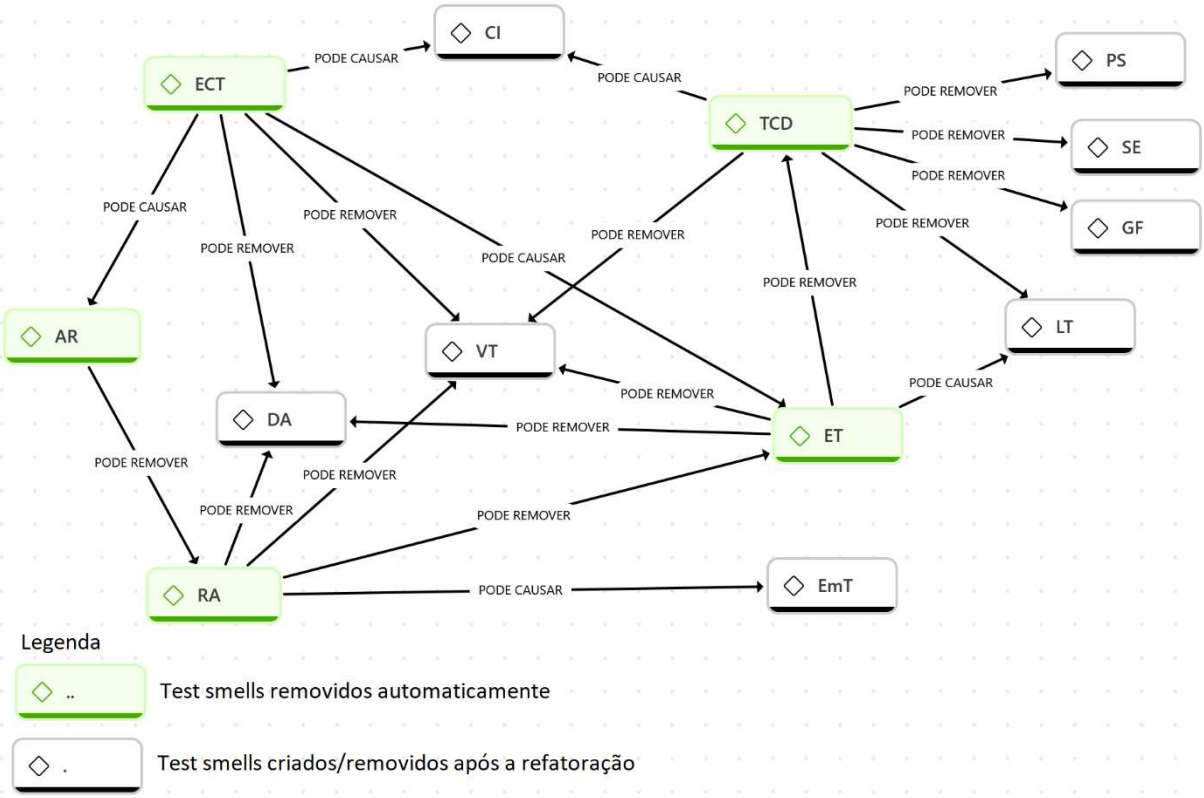

Figura 5-17 Remoção e criação de ocorrências de TS identificadas na refatoração automática (Fonte: o autor)

A próxima seção descreve um *survey* realizado para identificar como as equipes de desenvolvimento priorizam a remoção de *smells*.

5.5 Survey com membros de equipes de desenvolvimento sobre a remoção de Test Smells

O processo de refatoração proposto pode ser influenciado por diversos fatores, como: a) a granularidade do trecho de código afetado pelo *smell*, como uma instrução/linha de código, blocos de código, um método teste ou múltiplos métodos de testes; b) a criação de ocorrências de *smell* após uma refatoração; c) a remoção secundária de ocorrências de um *smell*; d) a necessidade de identificar as ocorrências de *smell* após a remoção de um de *smell* específico; e) os benefícios percebidos pela equipe de desenvolvimento com a remoção de ocorrências de *smells*; entre outras.

Assim, para identificar como as equipes de desenvolvimento priorizam a remoção de ocorrências de *smells*, foi conduzido um *survey* para coletar dados que possam nortear a priorização das refatorações.

O *survey* é um método de pesquisa executado com o objetivo de obter, descrever ou explicar conhecimentos das pessoas, onde se busca o entendimento de determinado aspecto de uma população (WOHLIN et al., 2012).

Para a coleta de dados, foram considerados no estudo 19 (dezenove) *smells* que são identificados pelo JNose e o TCD, que, conforme já mencionado, possui elevada ocorrência nos repositórios e está incluído no conjunto de *smells* removidos automaticamente na ferramenta desenvolvida. O público-alvo para a realização do *survey* foram desenvolvedores, analistas de qualidade, demais integrantes de equipes de desenvolvimento e pesquisadores sobre testes de unidade e *smells*.

5.5.1 Planejamento

Para nortear o estudo, foi definida a seguinte questão de pesquisa: Como os profissionais da área de desenvolvimento de software priorizam a remoção de *smells*?

O método de pesquisa selecionado para a realização do estudo foi um *survey*, no qual cada participante foi convidado a responder questões relacionadas sobre a percepção deles a respeito de propriedades dos TS, tais como:

- Ocorrência: identificar qual é a frequência de ocorrência percebida dos *smells* selecionados;
- Impacto na legibilidade: identificar como a equipe de desenvolvimento considera a presença de cada *smell* selecionado prejudicial à legibilidade do teste de unidade;

- Relação com a ocorrência de erros nos testes de unidade: identificar qual a percepção da equipe a respeito dos erros dos testes de unidade serem causadas pela presença de *smells* nos testes de unidade;
- Relação com a ocorrência de erros no software: identificar a percepção da equipe a respeito dos erros do software serem causadas pela presença de *smells* nos testes de unidade;
- Benefício de refatoração: para identificar qual é o benefício que a remoção de cada *smell* selecionado proporciona para a equipe de desenvolvimento;
- Esforço para remover as ocorrências: para identificar o esforço necessário (em tempo) para remover cada ocorrência de cada *smell* selecionado;
- Prioridade de remoção: para identificar qual a ordem de remoção de ocorrências de *smell* que o participante considera adequada.

5.5.2 Público-alvo, instrumento e execução da pesquisa

O instrumento de coleta dos dados é apresentado no Apêndice A. A validação do instrumento foi realizada por meio da apresentação, discussão e coleta de respostas de participantes do curso de mestrado e doutorado da PUCPR.

O questionário foi organizado em três partes: dados demográficos, avaliação de aspectos relacionados aos *smells* e a priorização da remoção de ocorrências de *smells*. Os dados demográficos referem-se à função do respondente, experiência profissional com testes de unidade e o nível de conhecimento sobre *smells*. A avaliação dos aspectos relacionados aos *smells* envolveu a avaliação sobre a frequência de ocorrência, o prejuízo à legibilidade do código de teste, a relação entre a presença de *smells* estar associada a erros na execução do teste, a relação entre a presença de *smells* estar associada a erros no software, o benefício e o esforço de remoção de ocorrências de smells do código de teste. Para as questões relacionadas com a avaliação de aspectos de *smells* e da priorização da remoção de ocorrências de *smells*, os respondentes poderiam informar, para cada uma das questões referentes à avaliação, que não sabiam responder, ou responder com base em 5 (cinco) níveis de concordância em uma escala Likert (LIKERT, 1932).

O questionário foi enviado para empresas, desenvolvedores e pesquisadores da área de Engenharia de Software, além de grupos de discussão sobre testes em redes sociais para a coleta de respostas e redistribuição do questionário. As respostas foram coletadas no período de 15/09/2023 até 17/10/2023.

5.5.3 Resultados

Esta Seção apresenta os resultados do *survey* realizado, mostrando os aspectos demográficos dos respondentes, a definição e o resultado da avaliação das hipóteses definidas, os resultados obtidos por categoria de respondentes, além das associações positivas por *smell* que foram identificadas na pesquisa.

5.5.3.1 Demografia dos respondentes

Esta seção apresenta os dados demográficos dos respondentes da pesquisa sobre a priorização da remoção de ocorrências de *smells*.

Na Figura 5-18 é mostrada a distribuição dos respondentes de acordo com a experiência com testes de unidade, dos quais: 3 (três) responderam não possuir experiência, 1 (um) tem menos de um ano de experiência, 4 (quatro) têm de 1 (um) até 3 (três) anos de experiência, 4 (quatro) têm mais de 3 (três) até 5 (cinco) anos de experiência, 2 (dois) têm mais de 5 (cinco) até 10 (dez) anos de experiência, e 3 (três) têm mais de 10 (dez) anos de experiência.

Figura 5-18 Distribuição dos respondentes de acordo com a experiência com testes de unidade (Fonte: o autor)

Na Figura 5-19 é mostrada a distribuição dos respondentes por nível de conhecimento sobre *smells*, dos quais 1 (um) respondeu que não possuía nenhum conhecimento, 7 (sete) responderam que conheciam pouco, 6 (seis) responderam que têm conhecimento razoável, e 3 (três) responderam que têm muito conhecimento sobre *smells*.

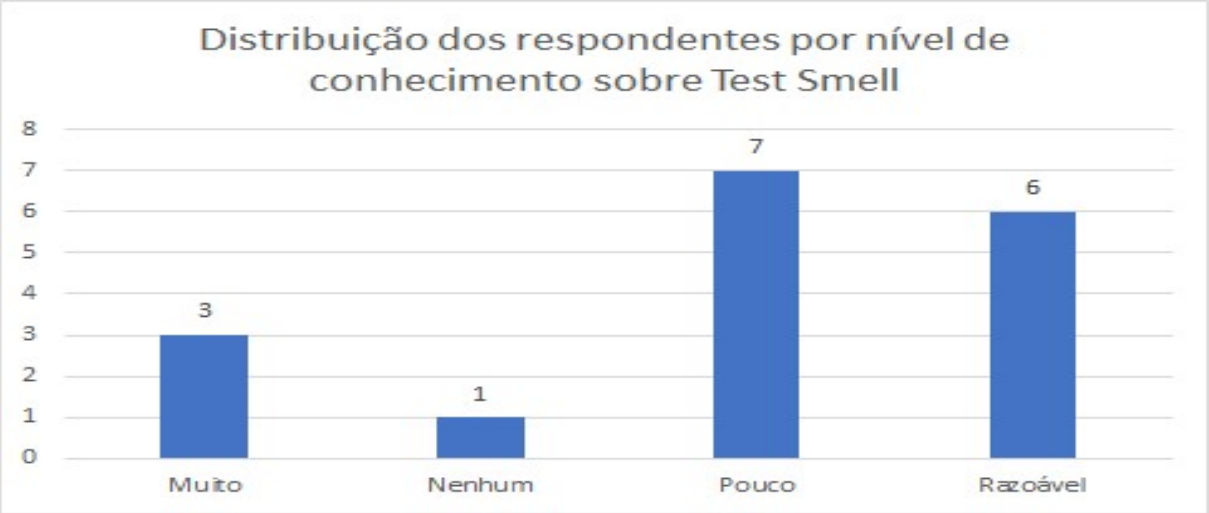

Figura 5-19 Distribuição dos respondentes por nível de conhecimento sobre Test Smells (Fonte: o autor)

Na Tabela 5-5 é apresentada a distribuição dos respondentes por função e experiência com testes de unidade. Do total de entrevistados, 4 (quatro) eram analistas de qualidade, dos quais 3 (três) não tinham experiência e 1 (um) possuía experiência de 1 (um) até 3 (três) anos, 1 (um) arquiteto de software com mais de 10 (dez) anos de experiência, 8 (oito) desenvolvedores, dos quais 4 (quatro) têm de 1 (um) até 3 (três) anos de experiência, 2 (dois) possuíam mais de 3 (três) até 5 (cinco) anos, e outros 2 (dois) possuíam mais de 5 (cinco) até 10 (dez) anos de experiência, 1 (um) estudante de doutorado possuía mais de 3 (três) até 5 (cinco) anos de experiência, 1 (um) gerente de projetos possuía menos de 1 (um) ano de experiência, 1 (um) gerente de qualidade e 1 (um) pesquisador, ambos com mais de 10 (dez) anos de experiência cada.

Tabela 5-5 Quantidade de respondentes por função e experiência com testes de unidade (Fonte: o autor)

Função x Experiência com testes de unidade	Não tem	Menos de um ano	De 1 até 3 anos	Mais de 3 até 5 anos	Mais de 5 até 10 anos	Mais de 10 anos
Analista de Qualidade	3		1			
Arquiteto de software						1
Desenvolvedor			4	2	2	
Estudante de doutorado				1		
Gerente de Projetos		1				
Gerente de qualidade						1
Pesquisador						1

Em relação à comparação do nível de conhecimento sobre *smells* e a função exercida, na Tabela 5-6 é mostrado que os 4 (quatro) analistas de qualidade possuíam níveis distintos de conhecimento, ou seja, 1 (um) não tem conhecimento, 1 (um) tem pouco conhecimento, 1 (um) tem conhecimento razoável, e outro tem muito conhecimento sobre *smells*. No caso da função arquiteto de software, o único respondente afirmou ter pouco conhecimento. Na função desenvolvedor, dos 8 (oito) respondentes, metade dos respondentes afirmou ter pouco conhecimento e a outra metade afirmou ter conhecimento razoável sobre *smells*. O estudante de doutorado afirmou ter muito conhecimento sobre *smells*, enquanto o gerente de projetos afirmou ter pouco conhecimento, o gerente de qualidade, nível de conhecimento razoável, e o pesquisador, nível de conhecimento muito elevado sobre *smells*.

Tabela 5-6 Quantidade de respondentes por função e nível de conhecimento sobre *smells* (Fonte: o autor)

Função x nível de conhecimento	Nenhum, não tenho nenhum conhecimento sobre Test Smells	Pouco, tenho conhecimento teórico sobre Test Smells	Razoável, consigo identificar Test Smells no código dos testes de unidade	Muito, a identificação e/ou remoção de Test Smells fazem parte na minha rotina de trabalho
Analista de Qualidade	1	1	1	1
Arquiteto de software		1		
Desenvolvedor		4	4	
Estudante de doutorado				1
Gerente de Projetos		1		
Gerente de qualidade			1	
Pesquisador				1

Na próxima seção são descritas as hipóteses definidas para a realização da pesquisa e são apresentados os resultados da sua avaliação.

5.5.3.2 Teste de hipóteses

Para realizar a avaliação dos dados coletados de todos os respondentes, as seguintes hipóteses foram definidas:

- Hipótese nula ($H1_0$): A prioridade da remoção de *smells* não está associada à percepção da frequência de ocorrência dos *smells*;
- Hipótese alternativa ($H1_1$): A prioridade da remoção de *smells* está associada à percepção da frequência de ocorrência dos *smells*;
- Hipótese nula ($H2_0$): A prioridade da remoção de *smells* não está associada à percepção do impacto causado na legibilidade do teste de unidade;
- Hipótese alternativa ($H2_1$): A prioridade da remoção de *smells* está associada à percepção do impacto causado na legibilidade do teste de unidade;
- Hipótese nula ($H3_0$): A prioridade da remoção de *smells* não está associada à percepção do impacto causado na ocorrência de erros nos testes de unidade;
- Hipótese alternativa ($H3_1$): A prioridade da remoção de *smells* está associada à percepção do impacto causado na ocorrência de erros nos testes de unidade;
- Hipótese nula ($H4_0$): A prioridade da remoção de *smells* não está associada à percepção do impacto causado na ocorrência de erros no software;
- Hipótese alternativa ($H4_1$): A prioridade da remoção de *smells* está associada à percepção do impacto causado na ocorrência de erros no software;
- Hipótese nula ($H5_0$): A prioridade da remoção de *smells* não está associada à percepção do benefício de remoção das ocorrências de *smells*;
- Hipótese alternativa ($H5_1$): A prioridade da remoção de *smells* está associada à percepção do benefício de remoção das ocorrências de *smells*;

- Hipótese nula ($H6_0$): A prioridade da remoção de *smells* não está associada à percepção do esforço necessário para remover as ocorrências de *smells*;
- Hipótese alternativa ($H6_1$): A prioridade da remoção de *smells* está associada à percepção do esforço necessário para remover as ocorrências de *smells*;

O teste de qui-quadrado (χ^2) foi aplicado para identificar a existência de associação entre a prioridade de remoção de *smells* definida pelos respondentes com: a frequência de ocorrência, o benefício de remoção de *smells*, a relação do *smell* com a ocorrência de erros na execução do teste, a relação do *smell* com a ocorrência de erro no software, o benefício da remoção de *smells*, e o esforço de remoção.

A combinação dos valores das variáveis sendo analisadas no teste do qui-quadrado resulta em uma tabela “n” por “m”, na qual “n” e “m” são a quantidade de valores possíveis para as duas variáveis sendo testadas. Cada célula resultante da tabela deve ter uma contagem mínima de 5 casos para a realização do teste. Porém, na análise dos dados coletados, identificou-se que a quantidade mínima de casos não era atendida para os valores que representavam as duas maiores intensidades das variáveis, como, por exemplo, “Sempre ocorrem” e “Muitas vezes ocorrem” para a questão de medição das ocorrências de *smells*. Com isso, os valores das respostas que representam as duas maiores intensidades foram combinados na que representa a segunda maior intensidade, como, por exemplo, em “Muitas vezes ocorrem”.

A respeito da prioridade das refatorações, os respondentes indicaram a ordem de remoção de um conjunto de 20 *smells*. Assim, cada *smell* poderia assumir a prioridade variando de 1 até 20. Para reduzir a granularidade das possíveis prioridades, a prioridade de remoção de 1 até 4 foram categorizadas como “Alta”, de 5 até 10 como “Média” e de 11 a 20 como “Baixa”. A prioridade “Alta” foi determinada com base da adaptação do Princípio de Pareto, supondo que os primeiros 20% das priorizações estão associadas a 80% dos benefícios para a equipe de desenvolvimento. As demais categorias foram distribuídas aleatoriamente, sendo as últimas 50% prioridades definidas como categoria “Baixa”, restando as 30% intermediárias para a categoria “Média”.

Os casos válidos, omissos e total de casos (válidos + omissos) são mostrados na Tabela 5-7, sendo exibidos para cada caso a quantidade (coluna N) e o respectivo

percentual (coluna %). Os casos omissos apresentados na tabela são provenientes das respostas em que os participantes responderam que não sabiam responder à questão.

Tabela 5-7 Casos válidos, omissos e total de casos (Fonte: o autor)

Análise realizada	Casos válidos		Casos omissos		Total de casos	
(Prioridade de refatoração versus ...)	N	%	N	%	N	%
Ocorrência	293	100	0	0,0	293	100
Legibilidade	290	99	3	1,0	293	100
Percepção de erros nos testes	289	98,6	4	1,4	293	100
Percepção de erros no software	283	96,6	10	3,4	293	100
Benefício da refatoração	293	100	0	0,0	293	100
Esforço de refatoração	292	99,7	1	0,3	293	100

A avaliação das hipóteses $H1_0$, $H2_0$, $H3_0$, $H4_0$, $H5_0$ e $H6_0$ foi realizada usando o teste do qui-quadrado (χ^2), bicaudal e com nível de significância de 5%, para identificar a associação, e o teste Tau-b de Kendall (τ) para identificar a intensidade da associação. As associações das linhas (X) e colunas (Y) identificadas podem ser positivas ou negativas. Associações positivas indicam que o aumento de X está associado ao aumento de Y, e vice-versa, enquanto na associação negativa o aumento de X está associado à diminuição de Y, e vice-versa.

O resultado do teste qui-quadrado para a hipótese $H1_0$ foi $\chi^2 = 6,847$ (*p-value* = 0,335), que é menor que o valor crítico ($\chi^2_{0,05;6} = 12,592$), não sendo possível rejeitar a hipótese nula $H1_0$, indicando que não existe associação entre a prioridade de refatoração com a frequência de ocorrência de *smells*.

O resultado do teste qui-quadrado para a hipótese $H2_0$ foi $\chi^2 = 16,259$ (*p-value* = 0,012), que é maior que o valor crítico ($\chi^2_{0,05;6} = 12,592$), resultando na rejeição da hipótese nula $H2_0$ e na aceitação da hipótese alternativa $H2_1$, indicando que existe associação entre o prejuízo para a legibilidade do código de teste e a prioridade de refatoração. O resultado do teste Tau-b de Kendall indica que a intensidade da associação é fraca ($\tau = -0,161$, *p-value* = 0,002). A análise dos resíduos do teste qui-quadrado aponta que a associação ocorre nas seguintes combinações de categorias de variáveis: a) “Muitas vezes prejudicam” está associado positivamente à “Alta” (resíduo = 3,3), e b) “Muitas vezes prejudicam” está associado negativamente à “Baixa” (resíduo = -3,6).

O resultado do teste qui-quadrado para a hipótese $H3_0$ foi $\chi^2 = 10,644$ (*p-value* = 0,100), que é menor que o valor crítico ($\chi^2_{0,05;6} = 12,592$), não sendo possível rejeitar a hipótese nula $H3_0$, indicando que a prioridade de refatoração não está associada à percepção dos respondentes sobre a presença de *smells* no código do teste de unidade estar com a ocorrência de erros nos testes de unidade.

O resultado do teste qui-quadrado para a hipótese $H4_0$ foi $\chi^2 = 15,228$ (*p-value* = 0,019), que é maior que o valor crítico ($\chi^2_{0,05;6} = 12,592$), resultando na rejeição da hipótese nula $H2_0$ e na aceitação da hipótese alternativa $H4_1$, indicando que a prioridade de refatoração está associada à percepção dos respondentes sobre a presença de *smells* no código do teste de unidade estar com a ocorrência de erros no software. O resultado do teste Tau-b de Kendall indica que a intensidade da associação é fraca ($\tau = -0,167$, *p-value* < 0,001). A análise dos resíduos do teste qui-quadrado aponta que a associação ocorre em dois casos: a) “Nunca está relacionada” está positivamente associada à “Baixa” (resíduo = 2,8) e b) “Muitas vezes está relacionada” está associada negativamente à “Baixa” (resíduo = -3,1).

O resultado do teste qui-quadrado para a hipótese $H5_0$ foi $\chi^2 = 36,165$ (*p-value* < 0,001), que é maior que o valor crítico ($\chi^2_{0,05;6} = 12,592$), resultando na rejeição da hipótese nula $H5_0$ e na aceitação da hipótese alternativa $H5_1$, indicando que existe associação entre a percepção do benefício de remoção e a prioridade de remoção de *smells*. O resultado do teste Tau-b de Kendall indica que a intensidade da associação é fraca ($\tau = -0,292$, *p-value* < 0,001). A análise dos resíduos do teste qui-quadrado aponta que a associação ocorre nas seguintes combinações de categorias de variáveis: a) “Não é benéfica” está associada positivamente à “Baixa” (resíduo = 2,2), b) “Pouco benéfica” está associada negativamente à “Alta” (resíduo = -3,4), c) “Pouco benéfica” está associada positivamente à “Baixa” (resíduo = 3,7), d) “Muito benéfica” está associada positivamente à “Alta” (resíduo = 4,7), e e) “Muito benéfica” está associada negativamente à “Baixa” (resíduo = -4,8).

O resultado do teste qui-quadrado para a hipótese $H6_0$ foi $\chi^2 = 27,554$ (*p-value* < 0,001), que é maior que o valor crítico ($\chi^2_{0,05;6} = 12,592$), resultando na rejeição da hipótese nula $H6_0$ e na aceitação da hipótese alternativa $H6_1$, indicando que existe associação entre o esforço de remoção e a prioridade de remoção de *smells*. O resultado do teste Tau-b de Kendall indica que a intensidade da associação é fraca ($\tau = -0,252$, *p-value* < 0,001). A análise dos resíduos do teste qui-quadrado aponta que a associação ocorre nas seguintes combinações de categorias de variáveis: a)

“Esforço muito baixo” está associado negativamente a “Alta” (resíduo = -3,3), b) “Esforço muito baixo” está associado positivamente com “Baixa” (resíduo = 2,6), c) “Esforço baixo” está associado positivamente a “Baixa” (resíduo = 2,4), d) “Esforço elevado” está associado positivamente a “Alta” (resíduo = 3,2), e e) “Esforço elevado” está associado negativamente a “Baixa” (resíduo = -3,6).

5.5.3.3 Resultados por categoria de respondente

As respostas foram analisadas categorizando os respondentes em “Especialistas (E)”, “Não especialistas (N)” e “Todos (T)” para identificar as associações por categoria de respondente. Foram considerados como “Especialistas” os respondentes que afirmaram ter muito conhecimento sobre *smells*, onde a identificação e/ou remoção de Test Smells fazem parte da rotina. Os demais foram categorizados como “Não especialistas” e “Todos” corresponde a união dos respondentes das duas categorias anteriores.

Na Tabela 5-8 são apresentadas as correlações identificadas no *survey* por categoria de respondente. Os respondentes categorizados como “Especialistas” são mostrados com a letra “E”, os categorizados como “Não especialistas”, são mostrados com a letra “N”, e todos os respondentes são mostrados com a letra “T”. As associações positivas de cada categoria de respondente são mostradas por meio do sinal “+” após a categoria do respondente, enquanto o sinal “-”, corresponde às associações negativas.

Em relação à ocorrência, somente as respostas da categoria “Especialistas” produziram associações. Os *smells* de ocorrência “Às vezes” tiveram associação positiva com a prioridade “Baixa” e associação negativa com a prioridade “Alta”, enquanto *smells* de ocorrência “Muitas vezes” foram associados negativamente à prioridade “Baixa” e positivamente à prioridade “Alta”, sugerindo que os especialistas priorizariam a remoção de *smells* de alta prioridade e ocorrência elevada (i.e., “Muitas vezes”).

A associação entre impacto na legibilidade e prioridade de refatoração foi encontrada somente na categoria de respondentes “Todos” em duas combinações de valores: na primeira, os entrevistados consideraram que os *smells* que muitas vezes prejudicam não devem ter baixa prioridade de remoção; e na segunda, os entrevistados consideraram que os *smells* que muitas vezes prejudicam a legibilidade devem ter prioridade alta de remoção.

Não foram encontradas associações positivas significativas ou negativas entre a prioridade de remoção e a presença de *smells* estar associada aos erros nos testes de unidade.

A associação entre a percepção sobre a presença de cada um dos *smells* no código do teste de unidade estar relacionada com a ocorrência de erros no software com a prioridade foi encontrada nas seguintes combinações de categorias: a) associação positiva entre “Nunca está relacionada” com prioridade “Baixa” para as categorias de respondentes “Não especialistas” e “Todos”, b) associação negativa entre “Às vezes está relacionada” com prioridade “Média” para os respondentes “Não especialistas”, c) associação negativa entre “Muitas vezes está relacionada” com a prioridade “Baixa” para as categorias de respondentes “Não especialistas” e “Todos”, d) associação positiva entre “Muitas vezes está relacionada” com a prioridade “Média” para a categoria de respondentes “Todos”.

A associação entre benefício e prioridade foram encontradas para 5 (cinco) combinações de níveis das variáveis. A primeira, a associação positiva entre benefício “Não é benéfico” com a prioridade “Baixa” para as categorias de respondentes “Não especialistas” e “Todos”. A segunda, a associação positiva entre benefício “Pouco benéfico” com prioridade “Baixa” para as categorias de respondentes “Não especialistas” e “Todos”. A terceira, associação negativa entre benefício “Pouco benéfico” com prioridade “Alta” para as categorias de respondentes “Não especialistas” e “Todos”. A quarta, associação negativa entre benefício “Muito benéfica” com prioridade “Alta” para a categoria de respondentes “Não especialistas”. A quinta, associação positiva entre benefício “Muito benéfico” com a prioridade “Alta” para as categorias de respondentes “Não especialistas” e “Todos”.

Em relação ao esforço, foram encontradas associações para 7 (sete) combinações de níveis de esforço com prioridade. A primeira, associação positiva entre esforço “Muito baixo” e prioridade “Baixa” para as categorias de respondentes “Especialistas” e “Todos”. A segunda, associação negativa entre esforço “Muito baixo” com prioridade “Alta” para as categorias de respondentes “Não especialistas” e “Todos”. A terceira, associação positiva entre esforço “Baixo” com prioridade “Baixa” para as categorias de respondentes “Especialistas” e “Todos”. A quarta, associação negativa entre esforço “Baixo” e prioridade “Média” para a categoria de respondentes “Especialistas”. A quinta, associação negativa entre esforço “Razoável” com prioridade “Baixa” para a categoria de respondentes “Especialistas”. A sexta,

associação negativa entre esforço “Baixo” e prioridade “Baixa” para as três categorias de respondentes. A sétima, associação positiva entre esforço “Elevado” com prioridade “Alta” para as categorias “Não especialistas” e “Todos”.

Tabela 5-8 Correlações identificadas por categoria de respondente (Fonte: o autor)

		Prioridade		
		Baixa	Média	Alta
Ocorrência	Nunca			
	Raramente			
	Às vezes	E+		E-
	Muitas vezes	E-		E+
Legibilidade	Nunca prejudica			
	Raramente prejudica			
	Às vezes prejudica			
	Muitas vezes prejudica	T-		T+
Erros nos testes	Nunca está relacionada			
	Raramente está relacionada			
	Às vezes está relacionada			
	Muitas vezes está relacionada			
Erros no software	Nunca está relacionada	N+T+		
	Raramente está relacionada			
	Às vezes está relacionada		N-	
	Muitas vezes está relacionada	N-T-	T+	
Benefício	Não é benéfica	N+T+		
	Pouco benéfica	N+T+		N-T-
	Relativamente benéfica			
	Muito benéfica	N-		N+T+
Esforço	Muito baixo	E+T+		N-T-
	Baixo	E+T+	E-	
	Razoável	E-		
	Elevado	E-N-T-		N+T+

5.5.3.4 Test Smells com associações positivas

As associações positivas encontradas foram analisadas para identificar quais foram os *smells* e as respectivas quantidades de respostas que produziram combinações relevantes para a associação.

A associação entre frequência de ocorrência com a prioridade de refatoração foram encontradas para duas combinações de categorias, conforme mostrado na Tabela 5-9. Na associação entre a categoria “Às vezes” de ocorrência e a categoria “Baixa” de prioridade nas respostas dos especialistas, foram encontrados os seguintes *smells*: GF e SE, com 2 (duas) respostas cada; DA, ECT, EmT, RA, ST, TCD e VT, com 1 (uma) resposta cada. Na associação entre a categoria “Muitas vezes” de ocorrência e a categoria “Alta” de prioridade nas respostas dos especialistas, foram encontrados os seguintes *smells*: CTL e ECT, com duas 2 respostas cada, e AR, DA, MNT, RA, TCD e UT, com uma resposta cada.

Tabela 5-9 Associações encontradas com a combinação dos níveis das variáveis ocorrência e prioridade (Fonte: o autor)

Nível de ocorrência	Nível de prioridade	Respondentes	Test Smell	Quantidade
Às vezes	Baixa	Especialistas	GF	2
			SE	2
			DA	1
			ECT	1
			EmT	1
			RA	1
			ST	1
			TCD	1
			VT	1
Muitas vezes	Alta	Especialistas	CTL	2
			ECT	2
			AR	1
			DA	1
			MNT	1
			RA	1
			TCD	1
			UT	1

A associação entre prejuízo à legibilidade e prioridade de refatoração foi encontrada para uma combinação de categorias, mostradas na Tabela 5-10, na qual “Muitas vezes” está associada com “Alta”, com as seguintes quantidades de respostas: CTL, com 6 (seis), AR e ECT, com 4 (quatro) cada, VT com 3 (três), DA e ET, com 2 (duas) cada, CI, EmT, GF, MNT, RA, RP, ST e TCD, com 1 (uma) cada.

Tabela 5-10 Associação encontrada com a combinação dos níveis das variáveis legibilidade e prioridade (Fonte: o autor)

Nível de legibilidade	Nível de prioridade	Respondentes	Test Smell	Quantidade
Muitas vezes	Alta	Todos	CTL	6
			AR	4
			ECT	4
			VT	3
			DA	2
			ET	2
			CI	1
			EmT	1
			GF	1
			MNT	1
			RA	1
			RP	1
			ST	1
			TCD	1

A associação entre a presença de *smells* nos testes causarem erros no software com a prioridade de refatoração foi encontrada para duas combinações de categorias, mostradas na Tabela 5-11, na qual “Nunca está relacionada” está associada com “Baixa” e "Muitas vezes" está associada com "Média". Na primeira combinação, as seguintes quantidades de respostas foram encontradas: RP, com 9 (nove), SE, com 7 (sete), CI e RA, com 5 (cinco) cada, TCD e UT, com 4 (quatro) cada, MNT e VT, com 3 (três) cada, AR, DA, IT, LT, MG e ST, com 2 (duas) cada, CTL e GF, com 1 (uma) cada. Na segunda combinação, foram encontradas as seguintes quantidades de respostas: IT, com 7 (sete), EmT, com 5 (cinco), LT, com 4 (quatro), ECT, ET, GF, MG e VT, com 2 (duas) cada, MNT, RA e ST, com 1 (uma) cada.

Tabela 5-11 Associações encontradas com a combinação dos níveis das variáveis falha no software e prioridade (Fonte: o autor)

Nível de erro	Nível de prioridade	Respondentes	Test Smell	Quantidade
Nunca está relacionada	Baixa	Todos	RP	9
			SE	7
			CI	5
			RA	5
			TCD	4

			UT	4
			MNT	3
			VT	3
			AR	2
			DA	2
			IT	2
			LT	2
			MG	2
			ST	2
			CTL	1
			GF	1
Muitas vezes	Média	Todos	IT	7
			EmT	5
			LT	4
			ECT	2
			ET	2
			GF	2
			MG	2
			VT	2
			MNT	1
			RA	1
			ST	1

A associação entre benefício e a prioridade de refatoração foi encontrada para 3 (três) combinações, mostradas na Tabela 5-12: “Não benéfico” e “Baixa”, “Pouco benéfico” e “Baixa”, e “Muito benéfico” e “Alta”. Na primeira combinação, foram identificadas as seguintes quantidades de respostas: MG, com 4 (quatro), MNT, RP, SE e ST, com 3 (três) cada, IT e LT, com 2 (duas) cada, RA, RO, UT e VT, com 1 (uma) cada. Na segunda combinação, foram encontradas as seguintes quantidades de respostas: RP, com 5 (cinco), SE, com 4 (quatro), CI e UT, com 3 (três) cada, AR, RA e VT, com 2 (duas) cada, DA, ECT, GF, LT, MNT, ST e TCD, com 1 (uma) cada. Na terceira combinação, foram encontradas as seguintes quantidades de respostas: CTL, com 10 (dez), ECT, com 6 (seis), AR, com 5 (cinco), ET e VT, com 3 (três), CI, DA, RA, RO, ST, TCD e UT, com 2 (duas) cada, EmT, GF, IT, MNT e RP, com 1 (uma).

Tabela 5-12 Associações encontradas com a combinação dos níveis das variáveis benefício e prioridade (Fonte: o autor)

Nível de benefício	Nível de prioridade	Respondentes	Test Smell	Quantidade
Não é benéfica	Baixa	Todos	MG	4
			MNT	3
			RP	3
			SE	3
			ST	3
			IT	2
			LT	2
			RA	1
			RO	1
			UT	1
			VT	1
Pouco benéfica	Baixa	Todos	RP	5
			SE	4
			CI	3
			UT	3
			AR	2
			RA	2
			VT	2
			DA	1
			ECT	1
			GF	1
			LT	1
			MNT	1
			ST	1
			TCD	1
Muito benéfica	Alta	Todos	CTL	10
			ECT	6
			AR	5
			ET	3
			VT	3
			CI	2
			DA	2
			RA	2
			RO	2
			ST	2
			TCD	2

			UT	2
			EmT	1
			GF	1
			IT	1
			MNT	1
			RP	1

A associação entre esforço de remoção e a prioridade de refatoração foi encontrada para 3 (três) combinações, conforme mostrado na Tabela 5-13: “Muito baixo” e “Baixa”, “Baixo” e “Baixa”, e “Elevado” e “Alta”. Na primeira combinação, foram encontradas as seguintes quantidades de respostas: SE, com 8 (oito), RP, com 7 (sete), RA, com 5 (cinco), CI e UT, com 4 (quatro) cada, DA e MNT, com 3 (três) cada, AR, IT, ST e TCD, com 2 (duas) cada, e LT e MG, com 1 (uma) cada. Na segunda combinação, foram encontradas as seguintes quantidades de respostas: GF, SE e VT, com 3 (três) cada, EmT, IT, MG, RO, TCD e UT, com 2 (duas) cada, CI, ECT, LT, MNT, RA, RP e ST, com 1 (uma) cada. Na terceira combinação, foram encontradas as seguintes quantidades de respostas: CTL, com 5 (cinco), VT, com 3 (três), ET e ST, com 2 (duas) cada, AR, CI, ECT, GF, MG, MNT, RA, RO, RP e UT, com 1 (uma) cada.

Tabela 5-13 Associações encontradas com a combinação dos níveis das variáveis esforço e prioridade (Fonte: o autor)

Nível de esforço	Nível de prioridade	Respondentes	Test Smell	Quantidade
Muito baixo	Baixa	Todos	SE	8
			RP	7
			RA	5
			CI	4
			UT	4
			DA	3
			MNT	3
			AR	2
			IT	2
			ST	2
			TCD	2
			LT	1
			MG	1
Baixo	Baixa	Todos	GF	3
			SE	3

			VT	3
			EmT	2
			IT	2
			MG	2
			RO	2
			TCD	2
			UT	2
			CI	1
			ECT	1
			LT	1
			MNT	1
			RA	1
			RP	1
			ST	1
Elevado	Alta	Todos	CTL	5
			VT	3
			ET	2
			ST	2
			AR	1
			CI	1
			ECT	1
			GF	1
			MG	1
			MNT	1
			RA	1
			RO	1
			RP	1
			UT	1

Na Tabela 5-14 são apresentadas as quantidades de respostas que produziram as associações entre as variáveis categóricas definidas. Na segunda linha são apresentadas as variáveis categóricas. Na quarta linha, os níveis das categorias de cada variável em que foram identificadas as associações. Na sexta linha, são apresentadas as categorias de prioridade das associações identificadas. Na primeira coluna são apresentados os *smells* selecionados para a realização do *survey*. Os números destacados em negrito referem-se às três maiores quantidades de cada coluna.

Tabela 5-14 Quantidade de associações por Test Smell para as variáveis definidas (Fonte: o autor)

*Smell*s / associação identificada	1) Variáveis categóricas										
	Ocorrência		Legibilidade	Falha no software		Benefício			Esforço		
	2) Níveis de cada variável categórica										
	Às vezes	Muitas vezes	Muitas vezes	Nunca está relacionada	Muitas vezes	Não é benéfica	Pouco benéfica	Muito benéfica	Muito baixo	Baixo	Elevado
	3) Prioridades com associação aos níveis de cada variável categórica										
	Baixa	Alta	Alta	Baixa	Média	Baixa	Baixa	Alta	Baixa	Baixa	Alta
AR		**1**	**4**	2			2	**5**	2		1
CI			1	**5**			**3**	2	4	**1**	1
CTL		**2**	**6**	1				**10**			**5**
DA	**1**	**1**	2	2			1	2	3		
ECT	**1**	**2**	**4**		2		1	**6**		**1**	1
EmT	**1**		1		**5**			1		**2**	
ET			2		2			3			**2**
GF	**2**		1	1	2		1	1		**3**	1
IT				2	**7**	2		1	2	**2**	
LT				2	**4**	2	1		1	**1**	
MG				2	2	**4**			1	**2**	1
MNT		**1**	1	3	1	**3**	1	1	3	**1**	1
RA	**1**	**1**	1	**5**	1	1	2	2	**5**	**1**	1
RO						1		2		**2**	1

Continua na próxima página

Continuação

Smells / associação identificada	1) Variáveis categóricas										
	Ocorrência		Legibilidade	Falha no software		Benefício			Esforço		
	2) Níveis de cada variável categórica										
	Às vezes	Muitas vezes	Muitas vezes	Nunca está relacionada	Muitas vezes	Não é benéfica	Pouco benéfica	Muito benéfica	Muito baixo	Baixo	Elevado
	3) Prioridades com associação aos níveis de cada variável categórica										
	Baixa	Alta	Alta	Baixa	Média	Baixa	Baixa	Alta	Baixa	Baixa	Alta
RP			1	**9**		**3**	**5**	1	**7**	**1**	1
SE	**2**			**7**		**3**	**4**		**8**	**3**	
ST	**1**		1	2	1	**3**	1	2	2	**1**	**2**
TCD	**1**	**1**	1	4			1	2	2	**2**	
UT		**1**		4		1	**3**	2	4	**2**	1
VT	1		3	3	2	1	2	3		3	**3**

5.5.4 Discussão dos resultados

A análise das associações identificadas por categoria de respondente (especialistas, não especialistas e todos) não mostrou concordância entre os grupos em todos os níveis das categorias comparadas. Por outro lado, não foram encontradas associações divergentes nos mesmos níveis das duas categorias. Do mesmo modo, para um mesmo nível de uma categoria, como “Muitas vezes” para a ocorrência por exemplo, não foram encontradas associações divergentes, ou seja, a intensidade de uma categoria ou não apresentou associações com os níveis de prioridade, ou as associações identificadas foram coerentes com o nível de prioridade. Assim, supõe-se que existiu certo grau de coerência nas respostas coletadas na pesquisa.

Um caso no qual o conhecimento sobre os *smells* pesquisados pode ter influenciado os resultados da pesquisa está na ausência de associações positivas entre a prioridade de refatoração e a percepção dos respondentes de que a presença de Test Smells está associada com a falha nos testes. Por exemplo, o Sleepy Test é um *smell* associado a testes não determinísticos, presente no questionário e que não apresentou associações entre os níveis de prioridade e de falha nos testes estipulados, sugerindo que os respondentes não conseguiram relacionar adequadamente a presença dos *smells* com as possibilidades de falhas dos testes de unidade.

Por outro lado, a associação positiva para o *smell* IT entre a prioridade e a presença de *smells* estar associada com a existência de falhas no software sugere que os respondentes consideraram o *smell* como um indicativo de dívida técnica auto admitida. Neste caso, a experiência e conhecimento dos respondentes pode indicar que o teste de unidade afetado pelo IT apresenta algum problema e, por algum motivo, teve sua execução desabilitada até que o problema seja corrigido. Com isso, o trecho de código do software não tem a devida cobertura de testes e falhas no código ficam sem ser identificadas.

Em relação às seis categorias definidas para identificar associações com a prioridade de refatoração, as duas que apresentaram a maior quantidade de associações foram benefício e esforço de remoção de *smells*. Nestas duas categorias destacaram-se os *smells* CTL e ECT, cuja remoção foi considerada “Muito benéfica” e de “Alta” prioridade de remoção. Além disso, CTL apresentou associação positiva para o nível “Muitas vezes” de legibilidade.

A partir dos resultados identificados nesta seção sobre a priorização da remoção de *smells* dos testes de unidade, foram definidas situações a serem consideradas no processo de refatoração, que são apresentadas na próxima seção.

5.6 Situações para a priorização das refatorações

A partir dos resultados do estudo exploratório descrito na seção 4.3 e do *survey* sobre a priorização das refatorações descrito na seção 5.5, foram definidos 5 (cinco) tipos de situações para serem consideradas na priorização da remoção de ocorrências dos *smells* hipotéticos A e B (quando necessário) de uma classe.

O primeiro tipo de situação está relacionado com a possibilidade de refatoração de uma determinada ocorrência de um *smell* hipotético (A), tais como:

- **A remoção de A pode ser realizada:** uma ocorrência de um *smell* em questão pode ser removida;
- **A remoção de A não pode ser realizada:** uma ocorrência do *smell* em questão não pode ser realizada. Um exemplo de impedimento à remoção é um método de teste afetado por um *smell* refatorável, como o Eager Test, e por um *smell* não determinístico, como o Sleepy Test.

O segundo tipo de situações está relacionado aos trechos de código afetados por duas ocorrências de *smells*, tais como:

- **A é coocorrente com B:** a ocorrência do *smell* em questão é coocorrente com outro *smell*, ou seja, pelo menos uma ocorrência de outro *smell* afeta o mesmo trecho de código que será refatorado do *smell* em questão;
- **A não é coocorrente com B:** o *smell* em questão não é coocorrente com uma ocorrência de outro *smell*, ou seja, as ocorrências afetam trechos distintos do código de teste.

O terceiro tipo de situações está relacionado com a intenção de remover determinado *smell* do repositório, tais como:

- **A remoção de A é permitida:** a priorização da refatoração deve incluir as ocorrências do *smell* em questão na lista de refatorações a serem realizadas;

- **A remoção de A não é permitida:** a priorização da refatoração não deve incluir as ocorrências do smell em questão na lista de refatorações a serem realizadas.

O quarto tipo de situações está relacionado com a prioridade de remoção de dois *smells* coocorrentes, tais como:

- **A prioridade de remoção de A é maior que a de B:** a prioridade de remoção do *smell* em questão é maior que a prioridade de remoção de outro *smell* coocorrente;
- **A prioridade de remoção de A é menor que a de B:** a prioridade de remoção do *smell* em questão é menor que a prioridade e remoção de outro *smell* coocorrente.

O quinto tipo de situações está relacionado com a consequência da remoção de um *smell* hipotético (A), tais como:

- **A remoção de A pode criar ocorrência de B:** a remoção do *smell* em questão pode criar uma ou mais ocorrências de outro *smell*;
- **A remoção de A pode remover ocorrência de B:** a remoção do *smell* em questão contribui para remover ou reduzir a quantidade de ocorrências de outro *smell*. Um exemplo a ser citado nesta condição é a remoção de AR e DA que podem afetar a mesma afirmação no código do teste;
- **A criação de B a partir da remoção de A não é permitida:** a remoção da ocorrência do *smell* em questão pode causar a criação de uma ou mais ocorrências de outro *smell*, o que não é aceitável.

A partir das situações acima descritas, a árvore de decisão mostrada na Figura 5-20 foi criada para determinar se uma ocorrência identificada para um *smell* em questão deve ser ou não refatorada.

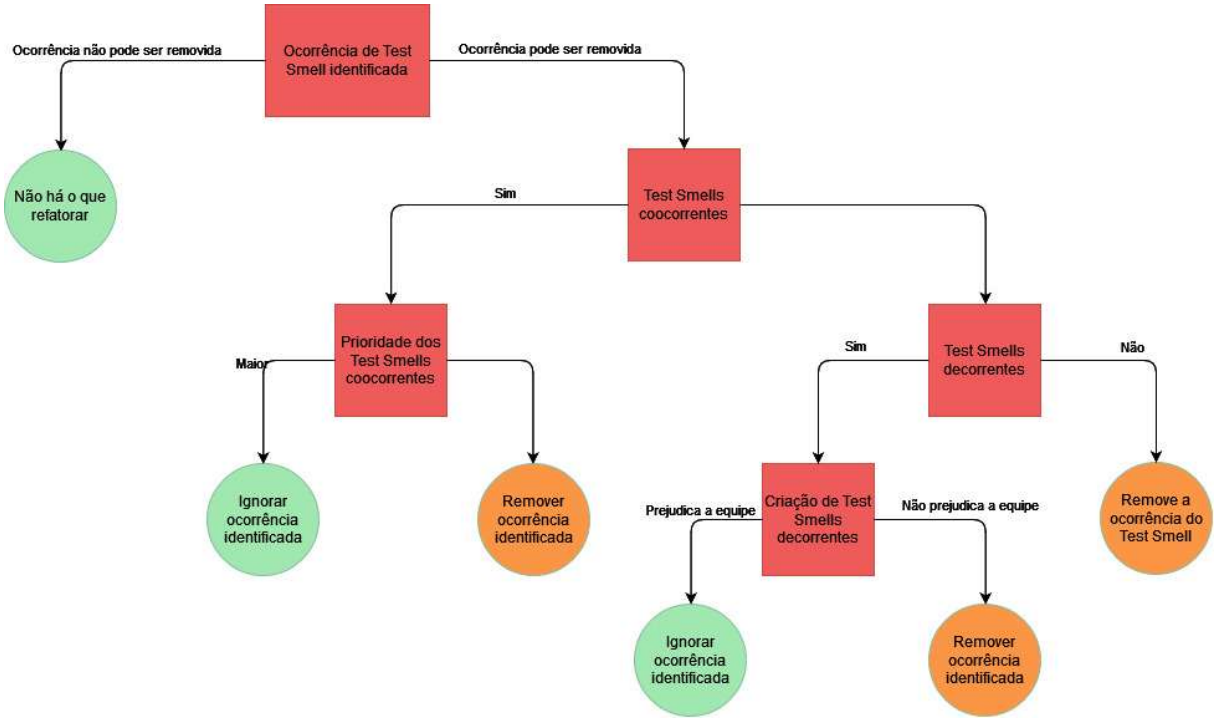

Figura 5-20 Árvore de decisão sobre refatorações (Fonte: o autor)

5.7 Considerações sobre o Capítulo

Este capítulo apresentou o processo de refatoração automática especificado para remover ocorrências de *smells* identificados em classes de testes de unidade que utiliza cópias do repositório de produção para a realização da validação das refatorações realizadas. A validação proposta tem por objetivo remover a necessidade de intervenção humana no processo de refatoração por meio da identificação e comparação do comportamento dos testes antes e após a refatoração. Além disso, foram apresentadas situações que podem ser usadas na priorização das refatorações e apresentadas as refatorações implementadas no aplicativo que instancia o processo especificado. Em seguida, foram apresentados os detalhes sobre como foi realizada a implementação dos *smells* selecionados para a realização da pesquisa.

No próximo Capítulo é apresentado como foi realizada a avaliação e os resultados da avaliação do processo de refatoração proposto e descrito na seção 5.1.

CAPÍTULO 6 - AVALIAÇÃO DO PROCESSO

Neste capítulo são apresentados os resultados da avaliação do processo que foi apresentado na seção 5.1, denominado Sentinel, da árvore de decisão para a realização das refatorações, das situações que precisam ser consideradas na realização das refatorações, e do refatorador. A avaliação do artefato foi realizada em duas etapas que possibilitaram a melhoria do artefato descrito no CAPÍTULO 5 -.

Na seção 6.1 é apresentada a avaliação da facilidade, intenção de uso e utilidade do processo em relação aos critérios facilidade, intenção de uso e utilidade do Technology Acceptance Model (TAM) 3. Na seção 6.2 é apresentada a avaliação da árvore de decisão e das situações a serem consideradas na priorização das refatorações automáticas. Na seção 6.3 é apresentada a avaliação do refatorador. Na seção 6.4, é apresentada a discussão dos resultados da avaliação. As mudanças realizadas são apresentadas na seção 6.5. Por fim, na seção 6.6 são apresentadas as ameaças à validade da pesquisa.

6.1 Avaliação da facilidade, intenção de uso e utilidade do processo

Conforme apresentado na seção 3.2.5, a análise dos dados das avaliações do processo foi realizada por meio da análise qualitativa. Na Figura 6-1 é mostrado o resultado da avaliação do critério facilidade de uso do Technology Acceptance Model (TAM) 3, na qual pode ser observado que: a) o processo pode ser integrado ao processo de desenvolvimento da organização; b) o processo foi considerado claro; e c) houve contradição no entendimento dos avaliadores sobre a compreensão do processo, pois o relato de ausência de legenda e a notação usada na especificação do processo contribuíram para o não entendimento do processo.

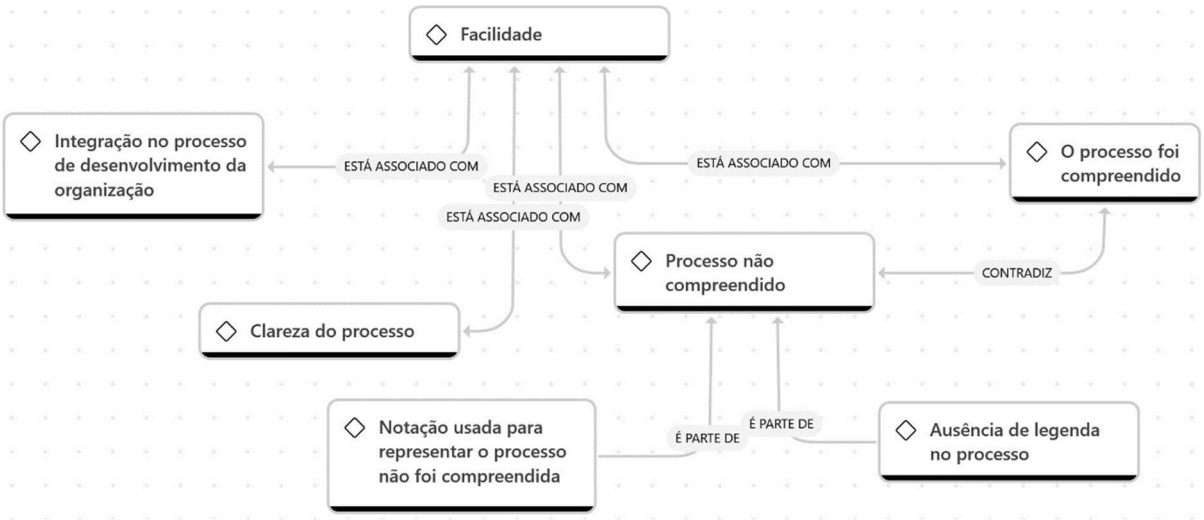

Figura 6-1 Rede criada sobre a avaliação da facilidade de uso do processo

A avaliação do critério intenção de uso do Technology Acceptance Model (TAM) 3 é mostrada na Figura 6-2, na qual é possível identificar que houve citações relacionadas com a espera da disponibilidade, a intenção de integrar a validação automática das refatorações à ferramenta própria, a melhoria da qualidade dos testes de unidade na organização e sobre o uso do processo para a comparação das refatorações realizadas.

Figura 6-2 Rede criada sobre a avaliação da intenção de uso

Em relação ao critério utilidade do Technology Acceptance Model (TAM) 3, na Figura 6-3 pode ser observado que a utilidade está relacionada aos contextos de uso na indústria e na academia. Em relação ao uso na academia, a utilidade está relacionada a avaliação das refatorações e na comparação das versões dos testes de

unidade, além da possibilidade de derivação de pesquisadas relacionadas com a validação automática das refatorações. Em relação ao uso na indústria, a melhoria da qualidade dos testes de unidade e o auxílio à equipe de desenvolvimento foram aspectos apontados como utilidade do processo de refatoração proposto. Além disso, a remoção de *smells* foi considerada benéfica, mas os benefícios decorrentes da adoção do processo podem ser subestimados na indústria.

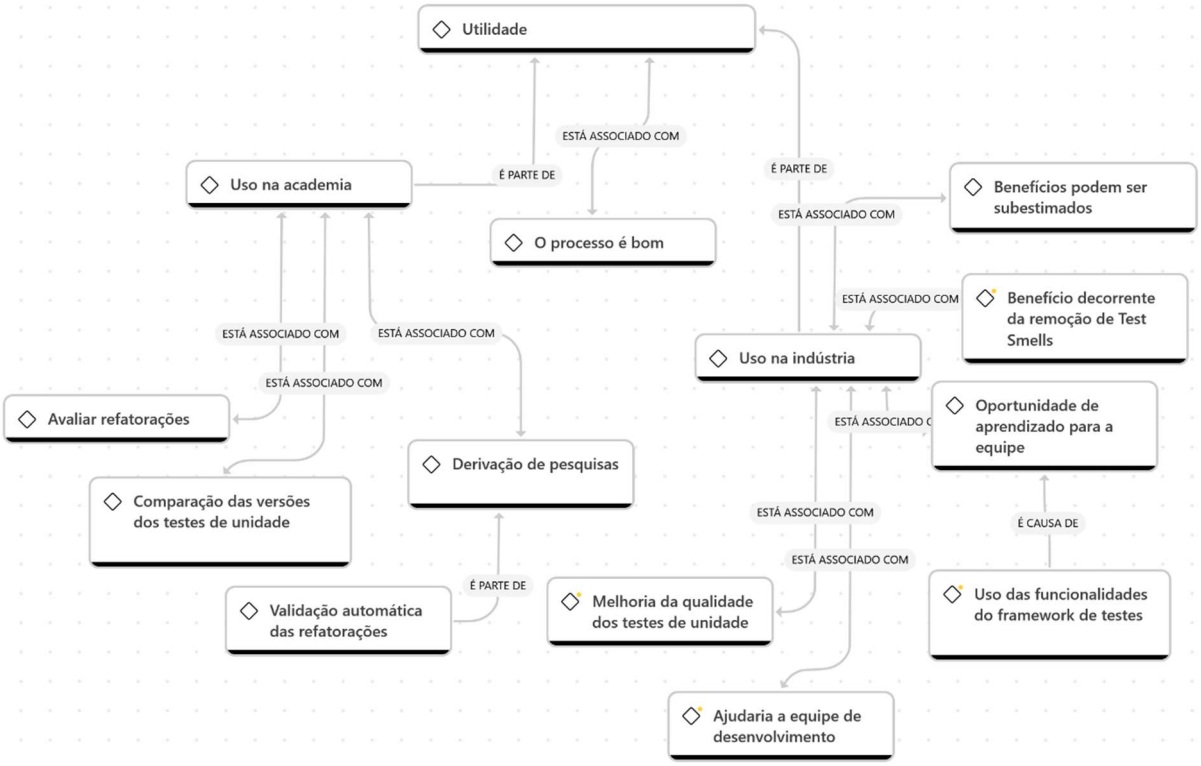

Figura 6-3 Rede criada sobre a avaliação da utilidade do processo Sentinel

6.2 Avaliação da árvore de decisão e das situações para a priorização das refatorações

Na Figura 6-4 é mostrada a rede criada sobre a avaliação da árvore de decisão, na qual podem ser observados códigos relacionados à falta de legenda e de informações na árvore, além das sugestões de mudar os termos usados e o padrão de cores na árvore.

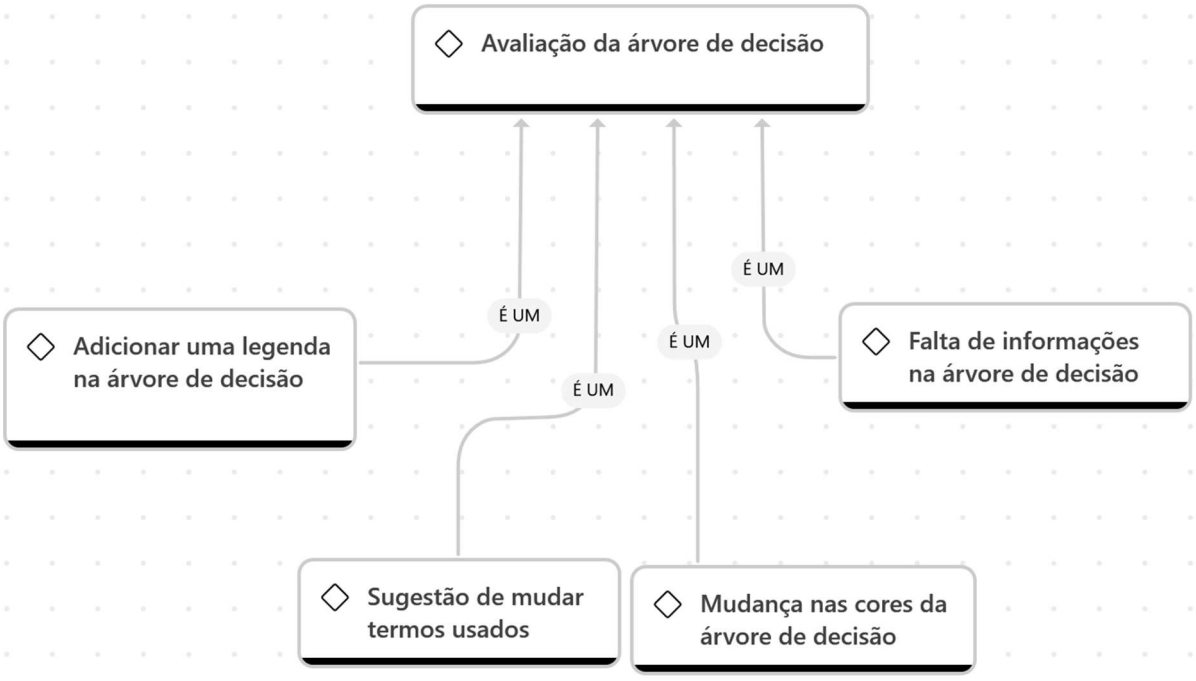

Figura 6-4 Rede criada sobre a avaliação da árvore de decisão

Na Figura 6-5 é mostrada a rede criada sobre a avaliação das situações para a priorização das refatorações, na qual podem ser observados dois códigos contraditórios sobre o entendimento das situações.

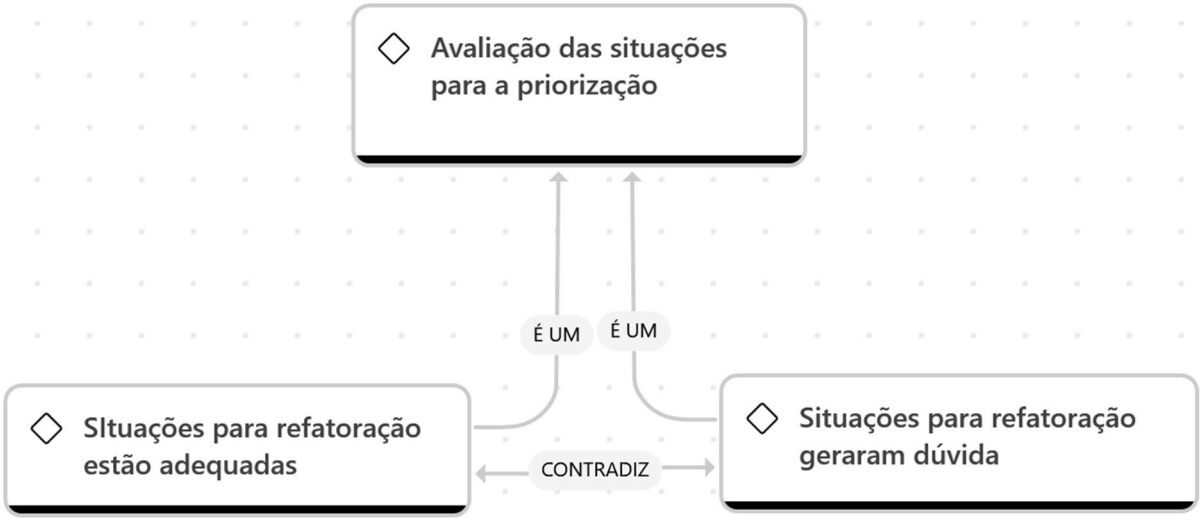

Figura 6-5 Rede criada sobre a avaliação das situações de priorização das refatorações

6.3 Avaliação do refatorador

Em adição aos resultados das avaliações do processo, mostrado na seção 6.1, e da árvore de decisão e das situações para a priorização das refatorações, mostrado na seção 6.2, nesta seção é apresentado o resultado da análise das avaliações do refatorador que implementa o processo especificado.

Na Figura 6-6 é mostrada a rede criada sobre a avaliação do refatorador Sentinel, na qual podem ser observadas sugestões de melhoria e preocupações com a refatoração. As sugestões de melhoria compreendem a geração de *log* das refatorações, a configuração das refatorações a serem realizadas, a possibilidade de comparar as versões do código refatorado e a integração com o *backlog* para registro das refatorações não realizadas. As preocupações com a refatoração incluem dificuldade de compreensão das refatorações realizadas, como nomes de classes e métodos de testes extraídos, a dificuldade de uso causada pela ausência de interface gráfica e o tempo necessário para realizar as refatorações.

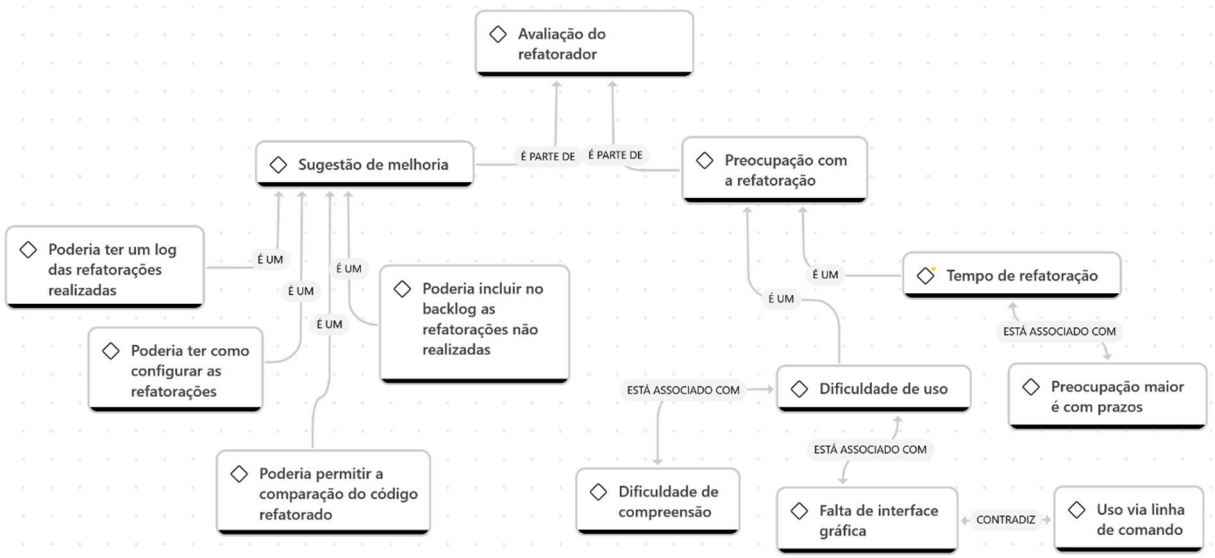

Figura 6-6 Rede criada sobre a avaliação do refatorador implementado para o processo Sentinel

6.4 Discussão dos resultados

A avaliação da automatização de tarefas realizadas por seres humanos não é uma tarefa trivial, pois o processo de automatização, o programa que automatiza uma tarefa e os resultados da automatização podem ser avaliados em conjunto.

Os resultados da avaliação do processo em relação aos aspectos do TAM 3, como facilidade, intenção de uso e utilidade, sugerem que o processo foi considerado adequado pelos avaliadores. Em relação à facilidade, a avaliação mostrou que o processo não foi adequadamente compreendido por causas como a falta de legenda e a notação usada no processo, o que contrasta com avaliações de que o processo está claro e que pode ser, facilmente, incorporado no processo de desenvolvimento das organizações. Esse contraste pode estar relacionado com o diferente nível de conhecimento dos avaliadores sobre a especificação de processos usando a notação do Business Process Model and Notation (BPMN), tais como tarefas, objetos de dados, fluxo de sequência, entre outros, que afetou a capacidade dos avaliadores de compreenderem os signos usados na elaboração do processo Sentinel.

Em relação à intenção de uso, as avaliações realizadas foram consideradas favoráveis ao processo Sentinel, pois as avaliações sugeriram que o processo pode contribuir com a melhoria da qualidade dos testes de unidade e na comparação das refatorações realizadas. A possibilidade de integração do processo proposto a uma abordagem existente, mesmo que parcialmente, sugere que o processo Sentinel apresenta aspectos de inovação que podem ser incorporados a pesquisas posteriores. Além disso, de acordo com os participantes do contexto da indústria, a disponibilidade do aplicativo de refatoração que implementa o processo proposto é aguardada, pois pode contribuir com a melhoria da qualidade dos testes de unidade por meio da remoção de *smells*.

Em relação à utilidade, os resultados sugerem que o processo pode ser útil para a academia e para a indústria. No contexto da academia, a utilidade pode estar relacionada com a derivação de pesquisas decorrentes da avaliação automática das refatorações proposta nesta tese, bem como por meio da avaliação das refatorações realizadas automaticamente e da comparação entre as versões refatoradas e não refatoradas dos testes. No contexto da indústria, a utilidade pode estar relacionada com a oportunidade de aprendizado decorrente das refatorações realizadas automaticamente, as quais podem ser exemplos para a equipe de desenvolvimento. Além disso, os possíveis benefícios da remoção de *smells* podem beneficiar a equipe de desenvolvimento por meio da melhoria da qualidade dos testes. Entretanto, os resultados da avaliação apontaram que futuros estudos de caso podem contribuir para facilitar a adoção do processo no contexto da indústria.

Em relação à avaliação da árvore de decisão para a realização das refatorações, os resultados da avaliação sugerem que as avaliações realizadas focaram em aspectos superficiais, pois ficaram restritas a aspectos estéticos tais como a falta de legenda e informações, além de sugestões de mudança de cores e de termos usados.

Além da árvore de decisão para a realização das refatorações, as situações a serem consideradas na refatoração foram avaliadas. Neste ponto, foi identificada uma contradição relacionada com o entendimento das situações especificadas, que sugere que, as situações quando foram avaliadas, foram realizadas de modo superficial.

Em relação à avaliação do refatorador, os resultados da avaliação apontaram enfoque em sugestões de melhorias e na preocupação com a execução das refatorações. As sugestões de melhoria apresentadas estão relacionadas com funcionalidades que possam auxiliar na realização de atividades de pesquisa e com funcionalidades que não foram previstas na versão inicial do refatorador, como configuração e integração com *backlog*. A dificuldade de uso do refatorador devido à ausência de *interface* gráfica pode ser considerado como uma consequência do enfoque da pesquisa que esteve focada nas equipes de desenvolvimento de software, que podem ser beneficiadas devido a remoção de ocorrências de *smells* dos testes de unidades do (s) repositório (s) da organização, e que podem estar mais habituadas a execução de programas por meio da linha de comando (terminal) do sistema operacional. Além disso, a ausência de um treinamento prévio ou elaboração de material de treinamento pode ter contribuído com a dificuldade de uso do aplicativo de refatoração Sentinel.

Em relação à avaliação do refatorador, os principais pontos a serem considerados foram a sugestão de melhoria de permitir a configuração das refatorações a serem realizadas e a preocupação com o tempo necessário para a realização das refatorações. A configuração das refatorações em um processo de remoção automática de ocorrências de *smells* é necessário para flexibilizar o comportamento do refatorador, pois mesmo que seja possível remover todas as ocorrências de *smells* de uma classe de testes, os resultados de uma ou mais refatorações podem prejudicar a equipe de desenvolvimento.

No caso do tempo necessário para a realização das refatorações, a preocupação com este aspecto está relacionada com a preocupação das equipes estar focada no cumprimento dos prazos de entrega de produtos e atualizações.

Assim, o tempo necessário para a execução das refatorações, em especial da etapa de validação das refatorações, deve ser minimizado para reduzir o impacto causado no processo de desenvolvimento por causa da demora na realização das refatorações.

A avaliação do processo Sentinel finaliza a primeira interação na aplicação do método de pesquisa DSMR, na qual o artefato foi proposto e avaliado. Seguindo o método DSMR, a partir dos resultados da avaliação realizada, o artefato proposto deve ser revisado e pode ser aperfeiçoado. Assim, o processo Sentinel, a árvore de decisão e o conjunto de situações a serem consideradas na refatoração foram revisados e modificados após a realização das avaliações, conforme apresentado na próxima seção.

6.5 Mudanças realizadas após a avaliação

Esta seção apresenta as mudanças realizadas na árvore de decisão, nas situações a serem consideradas na refatoração e no processo Sentinel a partir dos resultados das avaliações realizadas.

Em relação à árvore de decisão, os nós das decisões relacionadas com a não realização da refatoração, como “Ignorar ocorrência identificada” foram destacadas na cor laranja, quando serão somente ignoradas, ou na cor amarela, quando serão ignoradas e a possibilidade de refatoração deve ser integrada ao sistema de registro de atividades da organização (i.e., *backlog*). Além disso, os nós das decisões relacionadas com a realização das refatorações foram destacados na cor verde. Por fim, foi incluída uma legenda para descrever os elementos presentes na árvore de decisão.

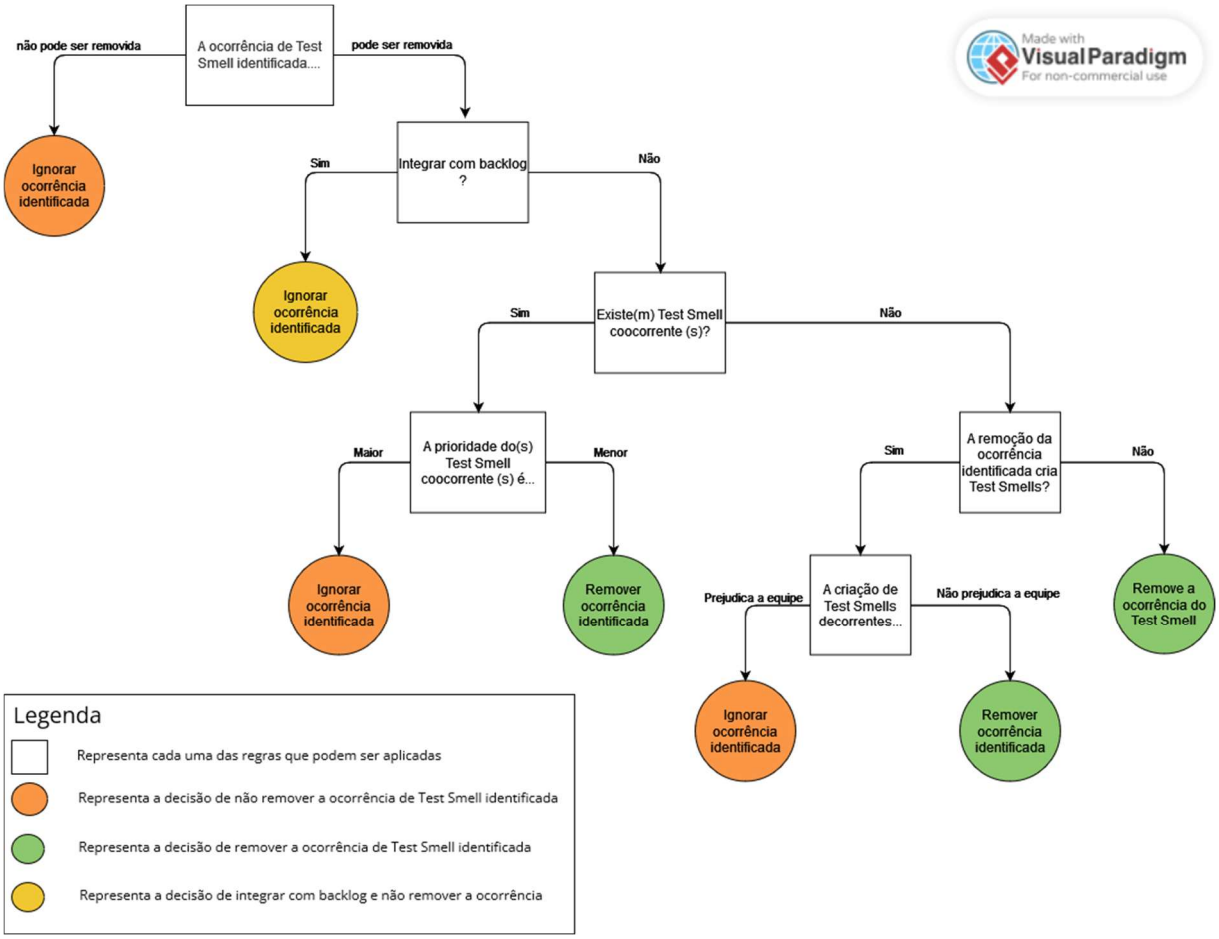

Figura 6-7 Árvore de decisão modificada após a realização das avaliações

Em relação às situações a serem consideradas na refatoração, a mudança realizada está relacionada com a intenção de remover determinado *smell* do repositório. Na versão inicial, essas intenções usavam o termo desejada, o qual foi substituído por permitida. Assim, as situações “A remoção de A é desejada” e “A remoção de A não é desejada” foram modificadas para “A remoção de A é permitida” e “A remoção de A não é permitida”, respectivamente.

Em relação ao processo Sentinel, após a realização das avaliações e da reavaliação do processo proposto, as seguintes mudanças foram realizadas:

- As atividades “Carregar arquivos do projeto” e “Instrumentar classes de código do software” foram agrupadas e a atividade resultante foi renomeada para “Identificar e instrumentar classes de código do software”;

- Na atividade “Identificar e instrumentar classes de código do software”, foi adicionada uma ligação de saída da atividade com o repositório de validação;
- Na atividade “Priorizar refatorações”, a entrada “Regras de priorização e refatorações” foi renomeada para “Regras de priorização e de refatorações”;
- Na atividade “Priorizar refatorações”, a saída “Refatorações a serem executadas” foi renomeada para “Ocorrências de Test Smells a serem refatoradas”;
- Na atividade “Refatorar classe de testes”, foi adicionada a entrada “Ocorrências de Test Smells a serem refatoradas”;
- Na atividade “Identificar o comportamento pré e pós-refatoração”, foi adicionada a saída “Comportamento pré e pós-refatoração”;
- Na atividade “Comparar comportamentos pré e pós-refatoração”, foi adicionada a entrada “Comportamento pré e pós-refatoração”;
- Foi adicionada a atividade “Gerar log da refatoração” após a atividade “Implantar refatorações”;
- Foi adicionado o *pool* Backlog, na qual foi criada a atividade “Atualizar lista de tarefas de refatoração” e o repositório “Backlog”.

A versão atualizada do processo Sentinel com as modificações apresentadas acima é mostrada na Figura 6-8. Convém ressaltar que o aplicativo que faz as refatorações não foi atualizado para incorporar as sugestões de melhoria identificadas na avaliação do processo.

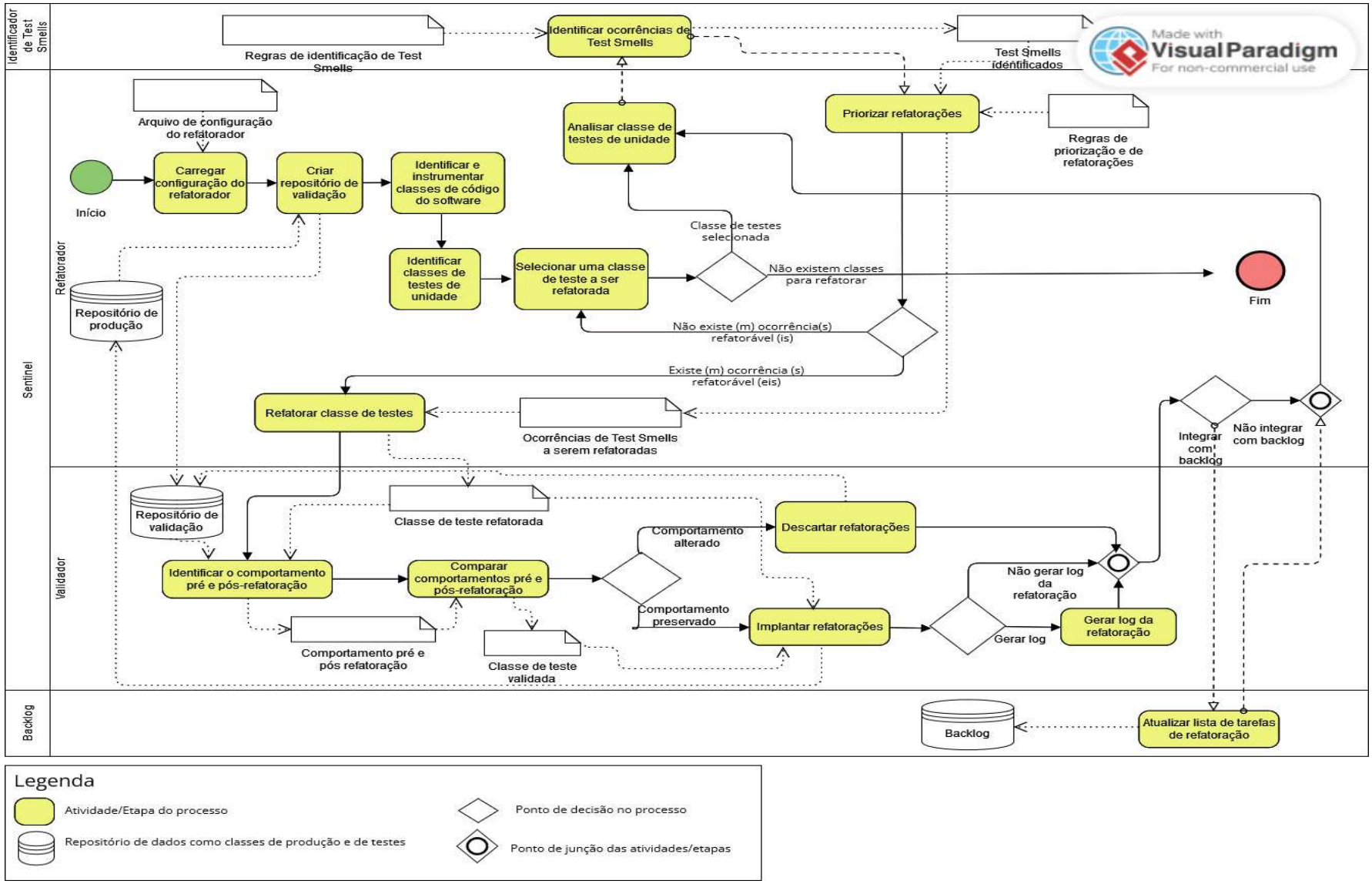

Figura 6-8 Versão modificada do processo proposto após a realização das avaliações

6.6 Ameaças à validade da avaliação

Nesta seção são descritas as ameaças relacionadas à validade do construto, à validade interna, à validade externa e à validade da resposta do instrumento usado na avaliação do processo, da árvore de decisão e das situações a serem consideradas na refatoração que foram propostos no contexto desta tese.

Em relação à validade do construto, a avaliação qualitativa dos aspectos facilidade, intenção de uso e utilidade do processo do TAM 3 pode produzir um viés na avaliação. De modo a mitigar o viés na avaliação, foi estabelecido um roteiro de avaliação, o qual é mostrado no Apêndice H, que foi definido para permitir que as avaliações pudessem coletar dados de acordo com o andamento da avaliação.

Em relação à validade interna, a amostra de avaliadores foi reduzida e composta por especialistas com diferentes perfis, tais como desenvolvedor, pesquisadores e analistas de qualidade, cujas identidades foram mantidas anônimas durante a avaliação de modo que os avaliadores se sentissem à vontade para se expressarem sem se preocuparem com possíveis inconvenientes. Além disso, a identidade de pessoas ou instituições relacionadas aos avaliadores foram mantidas em sigilo por meio da anonimização dos dados coletados.

Em relação à validade externa, a realização da pesquisa foi prejudicada pela falta de adesão aos convites para participar da avaliação, resultando em um pequeno número de avaliadores (N = 8) que não permite assegurar a generalização dos resultados obtidos. Para mitigar a limitação relacionada ao pequeno número de avaliadores, foram convidados avaliadores com experiência acadêmica e na indústria.

No que se refere às ameaças à validade de conclusão, a quantidade de avaliadores (N = 8) é a principal limitação desta pesquisa. Entretanto, apesar da quantidade pequena de avaliadores, as contribuições nas avaliações foram realizadas por pessoas ligadas a academia e a indústria não limitando, assim, o contexto de uso nos quais o processo foi avaliado.

6.7 Considerações sobre o Capítulo

Neste capítulo foram apresentados os resultados da avaliação do processo Sentinel com participantes ligados a academia e a indústria por meio de um estudo qualitativo. Na seção 6.1 foram apresentadas as redes criadas a partir dos dados da avaliação dos aspectos facilidade, intenção de uso e utilidade do TAM 3. Na seção

6.2 foram apresentadas as redes criadas a partir dos dados da avaliação da árvore de decisão e das situações a serem consideradas na refatoração. Em seguida, na seção 6.3 foi apresentada a rede criada com base nos dados da avaliação do refatorador que implementa o processo proposto. Na seção 6.4 são discutidos os resultados da avaliação. As mudanças realizadas no processo, árvore de decisão e situações a serem consideradas na priorização das refatorações são mostradas na seção 6.5. Por fim, as ameaças à validade são apresentadas na seção 6.6.

CAPÍTULO 7 - CONSIDERAÇÕES FINAIS

Neste capítulo são apresentadas as considerações finais da pesquisa. Na seção 7.1 é apresentada a relevância do estudo. Na seção 7.2 são apresentadas as contribuições da pesquisa. Na seção 7.3 são apresentadas as limitações da pesquisa. Por fim, na seção 7.4 são apresentadas as possibilidades de trabalhos futuros.

7.1 Relevância do estudo

As pesquisas sobre *Test Smells* estão relacionadas à identificação de ocorrências, com poucos estudos que abordam a remoção dessas. Os estudos existentes sobre a remoção de *Test Smells* geralmente abordam métodos semiautomatizados, nos quais a equipe de desenvolvimento é responsável por avaliar e aprovar as refatorações realizadas.

Nesse contexto, torna-se essencial investigar como a refatoração automática pode auxiliar a equipe de desenvolvimento na eliminação de *Test Smells* dos testes de unidade, permitindo melhorar a qualidade desses testes sem sobrecarregar a equipe.

Além disso, é necessário identificar como a refatoração automática pode ser realizada, de modo a subsidiar novas pesquisas voltadas para o aperfeiçoamento desse processo, incluindo a identificação de mudanças no comportamento dos testes de unidade refatorados e do software em teste.

Adicionalmente, compreender como as ocorrências de *smells* variam após a remoção de determinados *smells* é fundamental para avaliar o impacto de cada remoção. Isso possibilita priorizar refatorações que realmente contribuam para a melhoria da qualidade dos testes de unidade, evitando esforços em refatorações que, segundo o entendimento da equipe de desenvolvimento, não agreguem valor significativo ao processo.

7.2 Contribuições da pesquisa

A presente pesquisa apresenta as seguintes contribuições:

- A especificação do processo para a remoção automática de *smells*, apoiado por uma árvore de decisão e por um conjunto de situações a serem consideradas na priorização das refatorações, que propõe a validação automática das refatorações realizadas por meio da comparação do comportamento do SUT antes e após a refatoração para identificar mudanças de comportamento no SUT decorrentes da remoção automática de *smells*. Além disso, o processo prevê a reidentificação de ocorrências de *smells* após a aplicação das refatorações validadas de modo a refatorar a maior quantidade de ocorrências de smells durante a execução do refatorador;
- A especificação de uma abordagem para identificar o comportamento do software sendo testado antes e após a refatoração dos testes de unidade para identificar se houve mudança de comportamento por causa remoção de uma ocorrência de *smell*, por meio da identificação do estado de cada objeto do SUT em cada caminho de execução das instruções (ramificações);
- Um conjunto de situações que podem ser consideradas na etapa de priorização das refatorações de ocorrências de *smells* e que envolvem aspectos relacionados com a possibilidade de remoção dessas ocorrências, os trechos do código do teste que são afetadas por elas, a intenção de remover ou manter determinados *smells* no repositório, a prioridade de remoção e a aceitabilidade da criação de ocorrências de *smells* por meio da remoção automática;
- Uma árvore de decisão para a priorização das refatorações de ocorrências de *smells* que podem ser removidas na qual são consideradas a coocorrência de *smells* e a criação de *smells* decorrente da refatoração automática;
- Resultados de um quase-experimento sobre a variação na quantidade de ocorrências de *smells* decorrentes da remoção de um conjunto de *smells* escolhidos por afetarem partes diferentes do código de teste, como uma linha de teste, um trecho de código do método de teste ou mesmo mais de um método de teste. A variação na quantidade de ocorrências de *smells* após a remoção de um *smell* específico deve ser

considerada na refatoração (manual e automática) para evitar que a equipe de desenvolvimento seja prejudicada devido a criação de *smells* cuja remoção seja mais custosa que a não remoção do *smell* específico;

- Um método para a remoção automática de ocorrências do *smell Eager Test* no qual os testes são extraídos para métodos próprios, cuja execução é ordenada para manter o estado do SUT durante a transição dos métodos de testes extraídos.

7.3 Limitações da pesquisa

As principais limitações identificadas nesta pesquisa são:

- O processo para a remoção automática foi avaliado por um conjunto limitado de especialistas em Test Smells e integrantes de equipes da indústria. Apesar disso, foi realizada a avaliação com integrantes de equipes de desenvolvimento de software que apresentou alguns resultados relacionados com o uso do processo nas organizações de desenvolvimento de software;
- A quantidade de *smells* considerados na pesquisa corresponde a uma pequena porção em relação à quantidade de *smells* descritos na literatura. Entretanto, os *smells* escolhidos para remoção automática afetam porções diferentes do código de testes, como que afeta linha de código (Assertion Roulette), bloco de código (Exception Catching Throwing e Duplicate Assert), método de teste (Eager Test) e multimétodos de teste (TCD).

7.4 Trabalhos futuros

A partir das contribuições da presente pesquisa apresentados na seção 7.2, os seguintes trabalhos futuros são propostos:

- Avaliar o processo Sentinel com maior quantidade de especialistas e integrantes de equipes de desenvolvimento;
- Realizar estudos de caso em organizações de desenvolvimento de software para identificar o impacto da remoção automática de *smells* nas equipes de desenvolvimento de software;

- Investigar como aperfeiçoar a validação automática das refatorações de modo a reduzir o tempo necessário para a identificação da mudança de comportamento do software após a refatoração;
- Investigar como as ocorrências de Test Smells podem ser priorizadas de modo a reduzir o custo computacional relacionado com a validação automática das refatorações.

REFERÊNCIAS BIBLIOGRÁFICAS

(ALCOCER 2020) ALCOCER, Juan Pablo Sandoval; ANTEZANA, Alejandra Siles; SANTOS, Gustavo; BERGEL, Alexandre. Improving the success rate of applying the extract method refactoring. Science of Computer Programming, Volume 195, 2020, 102475, https://doi.org/10.1016/j.scico.2020.102475.

(ALJEDAANI 2021) ALJEDAANI, Wajdi; PERUMA, Anthony; ALJOHANI, Ahmed; ALOTAIBI, Mazen; MKAOUER, Mohamed Wiem; OUNI, Ali; NEWMAN, Christian D.; GHALLAB, Abdullatif; LUDI, Stephanie. 2021. Test Smell Detection Tools: A Systematic Mapping Study. In: Proceedings of Evaluation and Assessment in Software Engineering (EASE 2021). Association for Computing Machinery, New York, NY, USA, 170–180. https://doi.org/10.1145/3463274.3463335

(ALMEIDA 2015) ALMEIDA, Diogo; CAMPOS, José Creissac; SARAIVA, João; SILVA, João Carlos. 2015. Towards a catalog of usability smells. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15). Association for Computing Machinery, New York, NY, USA, 175–181. https://doi.org/10.1145/2695664.2695670

(ALOMAR 2021) ALOMAR, Eman Abdullah; PERUMA, Anthony; MKAOUER, Mohamed Wiem; NEWMAN, Christian; OUNI, Ali; KESSENTINI, Marouane. How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation. Expert Systems with Applications, volume 167, 2021, https://doi.org/10.1016/j.eswa.2020.114176

(ALQADI 2020) ALQADI; Basma S.; MALETIC, Jonathan I. Integration of Program Slicing with Cognitive Complexity for Defect Prediction. In: Proceedings of 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 839-843. https://doi.org/10.1109/ICSME46990.2020.00106

(ALZAHRANI 2019) ALZAHRANI, Musaad; ALQITHAMI, Saad; MELTON, Austin. Using Client-Based Class Cohesion Metrics to Predict Class Maintainability. 2019 In: IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), 2019, pp. 72-80, https://doi.org/10.1109/COMPSAC.2019.00020

(AMMANN 2016) AMMANN, Paul; OFFUTT, Jeff. Introduction to Software Testing. Cambridge University Press; 2nd edition (December 13, 2016) 364 pages. ISBN-10 : 9781107172012

(ANICHE 2015) ANICHE, Maurício. 2015. Java code metrics calculator (CK). Available in https: //github.com/mauricioaniche/ck/

(ARNAOUDOVA 2013) ARNAOUDOVA, Venera; DI PENTA, Massimiliano; ANTONIO, Giuliano; GUEHENEUC, Yann-Gael. 2013. A New Family of Software Anti-Patterns: Linguistic Anti-Patterns. In: Proceedings of 17th European Conference on Software Maintenance and Reengineering. https://doi.org/10.1109/CSMR.2013.28

(ATHANASIOU 2014) ATHANASIOU, Dimitrios; NUGROHO, Ariadi; VISSER, Joost; ZAIDMAN, Andy. Test Code Quality and Its Relation to Issue Handling Performance. IEEE Transactions on Software Engineering, vol. 40, no. 11, November 2014. https://doi.org/10.1109/TSE.2014.2342227

(BÁN 2017) BÁN, Dénes. The connection between antipatterns and maintainability in Firefox. Acta Cybernetica. Vol 23 No 2 (2017)

(BÁN 2014) BÁN, Dénes; FERENC, Rudolf. (2014) Recognizing Antipatterns and Analyzing Their Effects on Software Maintainability. In: Murgante B. et al., (eds) Computational Science and Its Applications – ICCSA 2014. ICCSA 2014. Lecture Notes in Computer Science, vol 8583. Springer, Cham. https://doi.org/10.1007/978-3-319-09156-3_25

(BAVOTA 2012) BAVOTA, Gabriele; QUSEF, Abdallah; OLIVETO, Rocco; DE LUCIA, Andrea; BINKLEY, David. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. 2012 In: Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 56-65, https://doi.org/10.1109/ICSM.2012.6405253

(BAVOTA 2015) BAVOTA, Gabriele; QUSEF, Abdallah; OLIVETO, Rocco; DE LUCIA, Andrea; BINKLEY, Dave. Are test smells really harmful? An empirical study. Empirical Software Engineering, pg1052–1094 (2015). https://doi.org/10.1007/s10664-014-9313-0

(BAKOTA 2011) BAKOTA, Tibor; HEGEDŰS, Péter; KÖRTVÉLYESI, Péter; FERENC, Rudolf; GYIMÓTHY, Tibor. A probabilistic software quality model. 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 243-252, https://doi.org/10.1109/ICSM.2011.6080791

(BARRAOOD 2018) BARRAOOD, Samera Obaid; MOHD, Haslina; BAHAROM, Fauziah; OMAR, Mazni. (2018). Quality factors of test cases: A systematic literature review. In: Proceedings of the Knowledge Management International Conference, KMICE, 355-362

(BENESTAD 2006) BENESTAD H.C., ANDA B., ARISHOLM E. (2006) Assessing Software Product Maintainability Based on Class-Level Structural Measures. In: Münch J., Vierimaa M. (eds) Product-Focused Software Process Improvement. PROFES 2006. Lecture Notes in Computer Science, vol 4034. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11767718_11

(BERNARD 2020) BERNARD, Elodie; BOTELLA, Julien, AMBERT, Fabrice; LEGEARD, Bruno; UTTING, Mark. Tool Support for Refactoring Manual Tests. 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 332-342, https://doi.org/10.1109/ICST46399.2020.00041

(BLADEL 2018) BLADEL, Brent; DEMEYER, Serge. 2018. Test behaviour detection as a test refactoring safety. In Proceedings of the 2nd International Workshop on Refactoring (IWoR 2018). Association for Computing Machinery, New York, NY, USA, 22–25. https://doi.org/10.1145/3242163.3242168

(BLADEL 2021) BLADEL, Brent; DEMEYER, Serge. A comparative study of test code clones and production code clones. The Journal of Systems & Software, vol 176, June 2021. https://doi.org/10.1016/j.jss.2021.110940

(BLESER 2019) BLESER, Jonas; DI NUCCI, Dario; DE ROOVER, Coen. Assessing Diffusion and Perception of Test Smells in Scala Projects. In: IEEE/ACM 16th

International Conference on Mining Software Repositories (MSR), 2019. pp. 457-467. https://doi.org/10.1109/MSR.2019.00072

BOWES, David; HALL, Tracy; PETRIC, Jean; SHIPPEY, Thomas; TURHAN, Burak. How Good Are My Tests? 2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM), Buenos Aires, Argentina, 2017, pp. 9-14, doi: 10.1109/WETSoM.2017.2.

(BÖRSTLER 2016) BÖRSTLER, Jürgen; PAECH, Barbara. The Role of Method Chains and Comments in Software Readability and Comprehension - An Experiment. In IEEE Transactions on Software Engineering, vol. 42, no. 9, pp. 886-898, 1 Sept. 2016. https://doi.org/10.1109/TSE.2016.2527791

(BREUGELMANS 2008) BREUGELMANS, Manuel; VAN ROMPAEY, Bart. 2008. TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites. In WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques

(BRIGGS 2011) BRIGGS, Robert O.; SCHWABE, Gerard. (2011) On Expanding the Scope of Design Science in IS Research. In: Jain H., Sinha A.P., Vitharana P. (eds) Service-Oriented Perspectives in Design Science Research. DESRIST 2011. Lecture Notes in Computer Science, vol 6629. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-20633-7_7

(BRUNTINK 2006) BRUNTINK, Magiel; VAN DEURSEN, Arie. An empirical study into class testability. Journal of Systems and Software, Volume 79, Issue 9, 2006, Pages 1219-1232, https://doi.org/10.1016/j.jss.2006.02.036

(BOWES 2017) BOWES, David; HALL, Tracy; PETRIC, Jean; SHIPPEY, Thomas; TURHAN, Burak. How Good Are My Tests? In: IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM), 2017, pp. 9-14. https://doi.org/10.1109/WETSoM.2017.2

(BUSE 2010) BUSE; Raymond P.L.; WEIMER, Westley R. Learning a Metric for Code Readability. IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 546-558, July-Aug. 2010. https://doi.org/10.1109/TSE.2009.70

(CALIKLI 2015) CALIKLI, Gul; BENER, Ayse. Empirical analysis of factors affecting confirmation bias levels of software engineers. Software Quality Journal 23, 695–722 (2015). https://doi.org/10.1007/s11219-014-9250-6

(CAMPOS 2021) CAMPOS, Denivan; ROCHA, Larissa; MACHADO, Ivan. 2021. Developer’s perception on the severity of test smells: an empirical study. In: Proceedings of XXIV Congresso Ibero-Americano em Engenharia de Software (CibSE'2021).

(CHIDAMBER 1994) CHIDAMBER, Shyam R.; KEMERER, Chris F. A metrics suite for object-oriented design. IEEE Transactions on Software Engineering (TSE), vol. 20, no. 6, pp. 476–493, June 1994. https://doi.org/10.1109/32.295895

(COHEN 1992) COHEN, Jacob. (1992). Statistical Power Analysis. Current Directions in Psychological Science, 1(3), 98-101. https://doi.org/10.1111/1467-8721.ep10768783

(COLEMAN 1994) COLEMAN, Don; ASH, Dan; LOWTHER, Bruce; OMAN, Paul. Using metrics to evaluate software system maintainability. Computer, vol. 27, no. 8, pp. 44-49, Aug. 1994. https://doi.org/10.1109/2.303623

(DAKA 2014) DAKA, Ermira; FRASER, Gordon. A Survey on Unit Testing Practices and Problems. In: IEEE 25th International Symposium on Software Reliability Engineering, 2014, pp. 201-211. https://doi.org/10.1109/ISSRE.2014.11

(DAMASCENO 2022) DAMASCENO, Humberto; BEZERRA, Carla; COUTINHO, Emanuel; MACHADO, Ivan. 2022. Analyzing Test Smells Refactoring from a Developers Perspective. In XXI Brazilian Symposium on Software Quality (SBQS ’22), November 7–10, 2022, Curitiba, Brazil. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3571473.3571487

(DELIGIANNIS 2003) DELIGIANNIS, Ignatios; SHEPPERD, Martin; ROUMELIOTIS, Manos; STAMELOS, Ioannis. An empirical investigation of an object-oriented design heuristic for maintainability. The Journal of Systems and Software 65 (2003) 127–139. https://doi.org/10.1016/S0164-1212(02)00054-7

(DELUCCHI 1993) DELUCCHI, K. L. (1993). On the use and misuse of chisquare. In G. Keren & C. Lewis (Eds.), A handbook for data analysis in the behavioral sciences (pp. 294-319). Hillsdale, NJ: Lawrence Erlbaum.

(DELPLANQUE 2019) DELPLANQUE, Julien; DUCASSE, Stéphane; POLITO, Guillermo; BLACK, Andrew P.; ETIEN, Anne. Rotten Green Tests. 2019 In: IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 500-511, https://doi.org/10.1109/ICSE.2019.00062

(ELISH 2009) ELISH, Karim O.; ALSHAYEB, Mohammad. Investigating the Effect of Refactoring on Software Testing Effort. 2009 16th Asia-Pacific Software Engineering Conference, 2009, pp. 29-34. https://doi.org/10.1109/APSEC.2009.14

(FERNANDES 2022) FERNANDES, Daniel; MACHADO, Ivan; MACIEL, Rita. 2022. TEMPY: Test Smell Detector for Python. In Proceedings of the XXXVI Brazilian Symposium on Software Engineering (SBES '22). Association for Computing Machinery, New York, NY, USA, 214–219. https://doi.org/10.1145/3555228.3555280

(FOWLER 2018) FOWLER, Martin. 2018. Refactoring: improving the design of existing code. Addison Wesley Professional.

(GARCIA 2009) GARCIA, Joshua; POPESCU, Daniel; EDWARDS, George; MEDVIDOVIC, Nenad. 2009. Identifying Architectural Bad Smells. In Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (CSMR '09). IEEE Computer Society, USA, 255–258. https://doi.org/10.1109/CSMR.2009.59

(GAROUSI 2015) GAROUSI, Vahid; AMANNEJAD, Yasaman; CAN, Aysu Betin. Software test-code engineering: A systematic mapping. Information and Software Technology 58 (2015) 123–147. https://doi.org/10.1016/j.infsof.2014.06.009

(GAROUSI 2019) GAROUSI, Vahid; KÜÇÜK, Barıs; FELDERER, M. What We Know About Smells in Software Test Code. in IEEE Software, vol. 36, no. 3, pp. 61-73, May-June 2019, https://doi.org/10.1109/MS.2018.2875843

(GAROUSI 2018) GAROUSI, Vahid; KÜÇÜK, Barıs. Smells in software test code: A survey of knowledge in industry and academia. The Journal of Systems and Software 138 (2018). pp52–81. https://doi.org/10.1016/j.jss.2017.12.013

(GIL 2002) GIL, Antonio Carlos. Como elaborar projeto de pesquisa. 4ª edição. São Paulo: Editora Atlas, 2002

(GLASER 1967) GLASER, B.; STRAUSS, A. The discovery of Grounded Theory: strategies for qualitative research. New York: Aldine de Gruyter, 1967.

(GRANO 2019a) GRANO, Giovanni. 2019. A New Dimension of Test Quality: Assessing and Generating Higher Quality Unit Test Cases. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3293882.3338984

(GRANO 2019b) GRANO, Giovanni; PALOMBA, Fabio; DI NUCCI, Dario; DE LUCIA, Andrea; GALL. Harald C. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. The Journal of Systems and Software 156 (2019) pp312–327. https://doi.org/10.1016/j.jss.2019.07.016

(GRANO 2020) GRANO, Giovanni; DE IACO, Cristian; PALOMBA, Fabio; GALL, Harald C. Pizza versus pinsa: On the perception and measurability of unit test code quality. In: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution, IEEE, 2020, pp. 336–347. https://doi.org/10.1109/ICSME46990.2020.00040

(GUPTA 2021) GUPTA, Shikha; CHUG, Anuradha. An Optimized Extreme Learning Machine Algorithm for Improving Software Maintainability Prediction. 2021 11th International Conference on Cloud Computing, Data Science & Engineering (Confluence), 2021, pp. 829-836. https://doi.org/10.1109/Confluence51648.2021.9377196.

(HABCHI 2021) HABCHI, Sarra; MOHA, Naouel; ROUVOY, Romain. Android code smells: From introduction to refactoring. The Journal of Systems & Software 177 (2021) 110964. https://doi.org/10.1016/j.jss.2021.110964

(HALSTEAD 1977) HALSTEAD, Maurice. Elements of software science. Elsevier, New York, 1977.

(HAUPTMANN 2015) HAUPTMANN, Benedikt; EDER, Sebastian; JUNKER, Maximilian; JUERGENS, Elmar; WOINKE, Volkmar. Generating Refactoring Proposals to Remove Clones from Automated System Tests. 2015 IEEE 23rd International Conference on Program Comprehension. https://doi.org/10.1109/ICPC.2015.20

(HEVNER 2004) HEVNER, Alan R.; MARCH, Salvatore T.; PARK, Jinsoo; RAM, Sudha. MIS Quarterly, Vol. 28, No. 1 (Mar., 2004), pp. 75-105 (31 pages) https://doi.org/10.2307/25148625

(HEVNER 2010) HEVNER A., CHATTERJEE S. (2010) Design Science Research in Information Systems. In: Design Research in Information Systems. Integrated Series in Information Systems, vol 22. Springer, Boston, MA. https://doi.org/10.1007/978-1-4419-5653-8_2

(HEVNER 2004) HEVNER, Alan R.; MARCH, Salvatore T.; PARK, Jinsoo; RAM, Sudha. Design Science in Information Systems Research. MIS Quarterly. vol. 28, No. 1 (Mar. 2004), pp. 75-105 (31 pages)

(HINKLE 2003) HINKLE, Dennis E.; WIERSMA, William; JURS, Stephen G. (2003). Applied Statistics for the Behavioral Sciences 5th ed. Boston: Houghton Mifflin, 2003

(ISO/IEC 24765 2010) INTERNATIONAL ORGANIZATION FOR STANDARDIZATION/ INTERNATIONAL ELECTROTECHNICAL COMISSION. NBR ISO/IEC 24765 - Systems and software engineering — Vocabulary. Geneve, 2010, 410p.

(ISO/IEC 25000 2014) INTERNATIONAL ORGANIZATION FOR STANDARDIZATION/ INTERNATIONAL ELECTROTECHNICAL COMISSION. NBR ISO/IEC 25000 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Geneve, 2014. 34p.

(ISO/IEC 25010 2011) INTERNATIONAL ORGANIZATION FOR STANDARDIZATION/ INTERNATIONAL ELECTROTECHNICAL COMISSION. NBR ISO/IEC 25010 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE. Geneve, 2011. 27p.

(JAAFAR 2017) JAAFAR, Fehmi; LOZANO, Angela; GUÉHÉNEUC, Yann-Gaël; MENS, Kim. Analyzing software evolution and quality by extracting Asynchrony change patterns. Journal of Systems and Software, Volume 131, 2017, Pages 311-322, https://doi.org/10.1016/j.jss.2017.05.047

(JHA 2019) JHA, Sudan; KUMAR, Raghvendra; SON, Le Hoang; ABDEL-BASSET, Mohamed; PRIYADARSHINI, Ishaani; SHARMA, Rohit; LONG, Hoang Viet. Deep Learning Approach for Software Maintainability Metrics Prediction. IEEE Access, vol. 7, pp. 61840-61855, 2019. https://doi.org/10.1109/ACCESS.2019.2913349

(KÁDÁR 2016) KÁDÁR, István; HEGEDUS, Péter; FERENC, Rudolf; GYIMÓTHY, Tibor. A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 599-603, https://doi.org/10.1109/SANER.2016.42

(KAUR 2020) KAUR, Amandeep. A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes.

Archives of Computational Methods in Engineering (2020) no. 27, pg. 1267–1296. https://doi.org/10.1007/s11831-019-09348-6

(KHOMH 2011) KHOMH, Foutse; VAUCHER, Stephane; GUÉHÉNEUC, Yann-Gaël; SAHRAOUI, Houari. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. The Journal of Systems and Software 84 (2011) 559–572. https://doi.org/10.1016/j.jss.2010.11.921

(KIM 2021) KIM, Dong Jae; CHEN, Tse-Hsun; YANG, Jinqiu. The secret life of test smells - an empirical study on test smell evolution and maintenance. Empirical Software Engineering (2021) 26: 100. https://doi.org/10.1007/s10664-021-09969-1

(LAM 2016) LAM, Wing. 2016. Repairing test dependence. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). Association for Computing Machinery, New York, NY, USA, 1121–1123. https://doi.org/10.1145/2950290.2983969

(LAMBIASE 2020) LAMBIASE, Stefano; CUPITO, Andrea; PECORELLI, Fabiano; DE LUCIA, Andrea; PALOMBA, Fabio. 2020. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In Proceedings of the 28th International Conference on Program Comprehension (ICPC '20). Association for Computing Machinery, New York, NY, USA, 441–445. https://doi.org/10.1145/3387904.3389296

(LEE 2015) LEE, Taek; LEE, Jung-Been; IN, Hoh Peter. Effect Analysis of Coding Convention Violations on Readability of Post-Delivered Code. IEICE Transactions on Information and Systems, 2015, Volume E98.D, Issue 7, Pages 1286-1296, Released July 01, 2015, https://doi.org/10.1587/transinf.2014EDP7327

(LIKERT 1932) Likert, Rensis. (1932). A technique for the measurement of attitudes. Archives of Psychology, 22 140, 55.

(LIN 2006) LIN, Jin-cherng; WU, Kuo-chiang. A Model for Measuring Software Understandability. In: The Sixth IEEE International Conference on Computer and Information Technology (CIT'06), 2006, pp. 192-192. https://doi.org/10.1109/CIT.2006.13

(MCCABE 1976) MCCABE, T. J. A Complexity Measure. In IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320, Dec. 1976, https://doi.org/10.1109/TSE.1976.233837

(MARANGUNIĆ; GRANIĆ, 2015) MARANGUNIĆ, N.; GRANIĆ, A. Technology acceptance model: a literature review from 1986 to 2013. Universal access in the information society, v. 14, n. 1, p. 81-95, 2015.

(MARTINEZ 2020) MARTINEZ, Matias; ETIEN, Anne; DUCASSE, Stéphane; FUHRMAN, Christopher. 2020. RTj: a Java framework for detecting and refactoring rotten green test cases. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings (ICSE '20). Association for Computing Machinery, New York, NY, USA, 69–72. https://doi.org/10.1145/3377812.3382151

(MESZAROS 2007) MESZAROS, Gerard. xUnit Test Patterns: Refactoring Test Code. Addison Wesley, 2007

(MISRA 2005) MISRA, Subhas Chandra. Modeling Design/Coding Factors That Drive Maintainability of Software Systems. Software Quality Journal 13, 297–320 (2005). https://doi.org/10.1007/s11219-005-1754-7

(MOHA 2007) MOHA, Naouel; GUÉHÉNEUC, Yann-Gael. 2007. Decor: a tool for the detection of design defects. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07). Association for Computing Machinery, New York, NY, USA, 527–528. https://doi.org/10.1145/1321631.1321727

(MOHA 2010) MOHA, Naouel; GUEHENEUC, Yann-Gael; DUCHIEN, Laurence; LE MEUR, Anne-Francoise. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering, vol. 36, no. 1, January/february 2010. https://doi.org/10.1109/TSE.2009.50

(PALOMBA 2017) PALOMBA, F.; ZAIDMAN, A. Does Refactoring of Test Smells Induce Fixing Flaky Tests? 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 1-12, https://doi.org/10.1109/ICSME.2017.12

(PALOMBA 2018) PALOMBA, Fabio; ZAIDMAN, Andy; DE LUCIA, Andrea. Automatic Test Smell Detection using Information Retrieval Techniques. 2018 In: Proceedings of IEEE International Conference on Software Maintenance and Evolution. https://doi.org/10.1109/ICSME.2018.00040

(PANICHELLA 2016) PANICHELLA, S., PANICHELLA, A., BELLER, M., ZAIDMAN, A., Gall, H.C., 2016. The impact of test case summaries on bug fixing performance: An empirical investigation. In: Proceedings of the Thirt-Eighth International Conference on Software Engineering. ACM, New York, NY, USA, pp. 547–558. https://doi.org/10.1145/2884781.2884847

(PANICHELLA 2020) PANICHELLA, Annibale; PANICHELLA, Sebastiano; FRASER, Gordon; SAWANT, Anand Ashok; HELLENDOORN, Vincent J. Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 523-533. https://doi.org/10.1109/ICSME46990.2020.00056

(PANTIUCHINA 2018) PANTIUCHINA, Jevgenija; LANZA, Michele; BAVOTA, Gabriele. Improving Code: The (Mis) Perception of Quality Metrics. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 80-91. https://doi.org/10.1109/ICSME.2018.00017

(PAUL 2024) PAUL, Partha Protim; AKANDA, Md Tonoy; ULLAH, Mohammed Raihan; MONDAL, Dipto; CHOWDHURY, Nazia Sultana; TAWSIF, Fazle Mohammed. 2024. XNose: A Test Smell Detector for C#. In Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24). Association for Computing Machinery, New York, NY, USA, 370–371. https://doi.org/10.1145/3639478.3643116

(PEFFERS 2007) PEFFERS, K.; TUUNANEN, T.; ROTHENBERGER, M. A.; CHATTERJEE, S. A design science research methodology for information systems research. Journal of Management Information Systems, v. 24, n. 3, p. 45-77, 2007. https://doi.org/10.2753/MIS0742-1222240302

(PERUMA 2019) PERUMA, Anthony; ALMALKI, Khalid; NEWMAN, Christian D.; MKAOUER, Mohamed Wiem; OUNI, Ali; PALOMBA, Fabio. 2019. On the Distribution of Test Smells in Open-Source Android Applications: An Exploratory Study. In: 29th Annual International Conference on Computer Science and Software Engineering, November 4–6, 2019.

(PERUMA 2020a) PERUMA, Anthony; ALMALKI, Khalid; NEWMAN, Christian D.; MKAOUER, Mohamed Wiem; OUNI, Ali; PALOMBA, Fabio. 2020. tsDetect: An Open-Source Test Smells Detection Tool. In: Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’20), November 8–13, 2020, Virtual Event, USA. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417921

(PERUMA 2020b) Peruma, Anthony; Newman, Christian D.; Mkaouer, Mohamed Wiem; Ouni, Ali; Palomba, Fabio. 2020. An Exploratory Study on the Refactoring of Unit Test Files in Android Applications. In IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW’20), May 23–29, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3387940.3392189

(PIZZINI 2022) PIZZINI, Adriano; REINEHR, Sheila; MALUCELLI, Andreia. 2023. Automatic Refactoring Method to Remove Eager Test Smell. In Proceedings of the XXI Brazilian Symposium on Software Quality (SBQS '22). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3571473.3571478

(POSNETT 2011) POSNETT, Daryl; HINDLE, Abram; DEVANBU, Premkumar. 2011. A simpler model of software readability. In: Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11). Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/1985441.1985454

(QUSEF 2019) QUSEF, Abdallah, ELISH, Mahmoud O.; BINKLEY, David. Exploratory Study of the Relationship Between Software Test Smells and Fault-Proneness. IEEE Access. vol 7, 2019. https://doi.org/10.1109/ACCESS.2019.2943488

(REICHHART 2007) REICHHART, Stefan; GÎRBA, Tudor; DUCASSE, Stéphane. 2007. Rule-based Assessment of Test Quality. Journal of Object Technology 6, 9 (2007), 231–251

(ROMPAEY 2007) ROMPAEY, B., DU BOIS, B., DEMEYER, S., RIEGER, M., 2007. On the detection of test smells: A Metrics-Based approach for general fixture and eager test. IEEE Transaction on Software Engineering. 33 (12), pp.800–817. https://doi.org/10.1109/TSE.2007.70745

(SALDAÑA 2013) SALDAÑA, Johnny. The coding manual for qualitative researchers. 2nd Edition. SAGE Publications Inc, 2013. ISBN 978-1-44624-736-5

(SAMARTHYAM 2017) SAMARTHYAM G.; MURALIDHARAN M.; ANNA R.K. 2017. Understanding Test Debt. In: Mohanty H., Mohanty J., Balakrishnan A. (eds) Trends in Software Testing. Springer, Singapore. https://doi.org/10.1007/978-981-10-1415-4_1

(SANTANA 2020) SANTANA, Railana; MARTINS, Luana; ROCHA, Larissa; VIRGÍNIO, Tássio; CRUZ, Adriana; COSTA, Heitor; MACHADO, Ivan. 2020. RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring. In Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES '20). Association for Computing Machinery, New York, NY, USA, 374–379. https://doi.org/10.1145/3422392.3422510

SHARMA, Tushar; FRAGKOULIS, Marios; SPINELLIS, Diomidis. Does Your Configuration Code Smell? 2016. In: IEEE/ACM 13th Working Conference on Mining Software Repositories, Austin, TX, 2016 pp. 189-200.

(SCHNAPPINGER 2019) SCHNAPPINGER, Markus; OSMAN, Mohd Hafeez; PRETSCHNER, Alexander; FIETZKE, Arnaud. Learning a Classifier for Prediction of Maintainability Based on Static Analysis Tools. 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019, pp. 243-248. https://doi.org/10.1109/ICPC.2019.00043

(SCALABRINO 2021) SCALABRINO, Simone; BAVOTA, Gabriele; VENDOME, Christopher; LINARES-VÁSQUEZ, Mario; POSHYVANYK, Denys; OLIVETO, Rocco. Automatically Assessing Code Understandability. IEEE Transactions on Software Engineering, vol. 47, no. 3, pp. 595-613, 1 March 2021, https://doi.org/10.1109/TSE.2019.2901468

(SCALABRINO 2017) SCALABRINO, Simone; BAVOTA, Gabriele; VENDOME, Christopher; LINARES-VÁSQUEZ, Mario; POSHYVANYK, Denys; OLIVETO, Rocco. Automatically assessing code understandability: How far are we? In: Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 417-427, https://doi.org/10.1109/ASE.2017.8115654

(SCALABRINO 2016) SCALABRINO, Simone; LINARES-VÁSQUEZ, Mario; POSHYVANYK, Denys; OLIVETO, Rocco. Improving code readability models with textual features. IEEE 24th International Conference on Program Comprehension (ICPC), 2016, pp. 1-10, https://doi.org/10.1109/ICPC.2016.7503707

(SETIANI 2020) SETIANI, Novi; FERDIANA, Ridi; HARTANTO, Rudy. Test Case Understandability Model. in IEEE Access, vol. 8, pp. 169036-169046, 2020, https://doi.org/10.1109/ACCESS.2020.3022876

(SHARMA 2016) SHARMA, Tushar; FRAGKOULIS, Marios; SPINELLIS, Diomidis. Does Your Configuration Code Smell? 2016. In: IEEE/ACM 13th Working Conference on Mining Software Repositories, Austin, TX, 2016 pp. 189-200. https://doi.org/10.1145/2901739.2901761

(SHARMA 2014) SHARMA, Vibhu Saujanya, ANWER, Samit, 2014. Performance antipatterns: detection and evaluation of their effects in the cloud. In: Proceedings - 2014 IEEE International Conference on Services Computing, SCC 2014. IEEE,

Accenture Services Pvt Ltd., Bangalore, India, pp. 758–765. https://doi.org/10.1109/SCC.2014.103

(SILVA 2020) SILVA, Lucas Pereira; VILAIN, Patrícia. 2020. LCCSS: A Similarity Metric for Identifying Similar Test Code. In Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '20). Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/3425269.3425283

(SOARES 2022) SOARES, Elvys; RIBEIRO, Márcio; GHEYI, Rohit; AMARAL, Guilherme; SANTOS, André. Refactoring Test Smells with JUnit 5: Why Should Developers Keep Up to Date? In: IEEE Transactions on Software Engineering, vol. 49, no. 3, pp. 1152-1170, 1 March 2023, doi: 10.1109/TSE.2022.3172654

(SPADINI 2018) SPADINI, Davide; PALOMBA, Fabio; ZAIDMAN, Andy; BRUNTINK, Magiel, BACCHELLI, Alberto. On The Relation of Test Smells to Software Code Quality. 2018 In: IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 1-12. https://doi.org/10.1109/ICSME.2018.00010

(SPADINI 2020) SPADINI, Davide; SCHVARCBACHER, Martin; OPRESCU, Ana-Maria; BRUNTINK, Ana-Maria; BACCHELLI, Alberto. 2020. Investigating Severity Thresholds for Test Smells. In: 17th International Conference on Mining Software Repositories (MSR ’20), October 5–6, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387453

(SOUZA 2016) SOUSA, Leonardo da Silva. Spotting Design Problems with Smell Agglomerations. In: IEEE/ACM 38th IEEE International Conference on Software Engineering Companion, 2016. http://dx.doi.org/10.1145/2889160.2889273

(STRAUSS 1998) STRAUSS, A.; CORBIN, J. Basics of qualitative research: techniques and procedures for developing Grounded Theory. 2. ed. Sage Publications, 1998.

(THONGKUM 2020) THONGKUM, Patcharapan; MEKRUKSAVANICH, Sakorn. Design Flaws Prediction for Impact on Software Maintainability using Extreme Learning Machine. 2020 In: Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunications Engineering (ECTI DAMT & NCON), 2020, pp. 79-82. https://doi.org/10.1109/ECTIDAMTNCON48261.2020.9090717

(TROCKMAN 2018) TROCKMAN, Asher; CATES, Keenen; MOZINA, Mark; NGUYEN, Tuan; KÄSTNER, Christian; VASILESCU, Bogdan. Automatically Assessing Code Understandability Reanalyzed: Combined Metrics Matter. 2018. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18). Association for Computing Machinery, New York, NY, USA, 314–318. https://doi.org/10.1145/3196398.3196441

(TUFANO 2016) TUFANO, Michele; PALOMBA, Fabio; BAVOTA, Gabriele; DI PENTA, Massimiliano; OLIVETO, Rocco; DE LUCIA, Andrea; POSHYVANYK, Denys. An Empirical Investigation into the Nature of Test Smells. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).

Association for Computing Machinery, New York, NY, USA, 4–15. https://doi.org/10.1145/2970276.2970340

(TRAN 2021) TRAN, Huynh Khanh Vi; UNTERKALMSTEINER, Michael; BÖRSTLER, Jürgen; BIN ALI, Nauman. Assessing test artifact quality: A tertiary study. Information and Software Technology 139 (2021). https://doi.org/10.1016/j.infsof.2021.106620

(VAHABZADEH 2018) VAHABZADEH, Arash; STOCCO, Andrea; MESBAH, Ali. 2018. Fine-grained test minimization. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18). Association for Computing Machinery, New York, NY, USA, 210–221. https://doi.org/10.1145/3180155.3180203

(VAN DEURSEN 2001) VAN DEURSEN, A.V., MOONEN, L., BERGH, A.V.D., KOK, G., 2001. Refactoring test code. In: Marchesi, M. (Ed.), Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001), University of Cagliari, pp. 92–95.

(VETRO 2013) VETRO; Antonio, ARDITO, Luca; PROCACCIANTI, Giuseppe; MORISIO, Maurizio. Definition, implementation, and validation of energy code smells: an exploratory study on an embedded system. March 2013. Conference: ENERGY 2013: The Third International Conference on Smart Grids, Green Communications, and IT Energy-aware Technologies.

(VIRGÍNIO 2021) VIRGÍNIO, Tássio; MARTINS, Luana; ROCHA, Larissa; SANTANA, Railana; CRUZ, Adriana; COSTA, Heitor; MACHADO, Ivan. 2020. JNose: Java Test Smell Detector. In: Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES '20). Association for Computing Machinery, New York, NY, USA, 564–569. https://doi.org/10.1145/3422392.3422499

(XUAN 2016) XUAN, Jifeng; CORNU, Benoit; MARTINEZ, Matias; BAUDRY, Benoit; SEINTURIER, Lionel; MONPERRUS, Martin. B-Refactoring: Automatic test code refactoring to improve dynamic analysis. Information and Software Technology 76 (2016) 65–80. https://doi.org/10.1016/j.infsof.2016.04.016

(WAGEY 2015) WAGEY; Billy C.; HENDRADJAYA, Bayu; MARDIYANTO, M. Sukrisno. A proposal of software maintainability model using code smell measurement. In: Proceedings of International Conference on Data and Software Engineering (ICoDSE), 2015, pp. 25-30. https://doi.org/10.1109/ICODSE.2015.7436966

(WANG 2009) WANG, Shuang; OFFUTT, Jeff. Comparison of Unit-Level Automated Test Generation Tools. 2009. IEEE International Conference on Software Testing Verification and Validation Workshops. https://doi.org/10.1109/ICSTW.2009.36

(WANG 2014) WANG, C., HIRASAWA, S., TAKIZAWA, H., KOBAYASHI, H., 2014. A platform-specific code smell alert system for high performance computing applications. In: Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW'14). IEEE Computer Society, pp. 652–661. https://doi.org/10.1109/IPDPSW.2014.76

(WANG 2022) WANG, Tongjie; GOLUBEV, Yaroslav; SMIRNOV, Oleg; LI, Jiawei; BRYKSIN, Timofey; AHMED, Iftekhar. 2022. PyNose: a test smell detector for python. In Proceedings of the 36th IEEE/ACM International Conference on Automated

Software Engineering (ASE '21). IEEE Press, 593–605. https://doi.org/10.1109/ASE51524.2021.9678615

(WIKLUND 2012) WIKLUND, Kristian; ELDH, Sigrid; SUNDMARK, Daniel; LUNDQVIST, Kristina. Technical Debt in Test Automation. In: Proceedings of IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012. https://doi.org/10.1109/ICST.2012.192

(WOHLIN 2012) WOHLIN, Claes; RUNESON, Per; HST, Martin; OHLSSON, Magnus C.; REGNELL, Bjrn; WESSLN, Anders. 2012. Experimentation in Software Engineering. Springer Publishing Company, Incorporated.

(YAMASHITA 2014) YAMASHITA, Aiko. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. Empirical Software Engineering (2014) 19:1111–1143. https://doi.org/10.1007/s10664-013-9250-3

(YANG 2024) YANG, Yanming; HU, Xing; XIA, Xin; YANG, Xiaohu. 2024. The Lost World: Characterizing and Detecting Undiscovered Test Smells. ACM Transaction on Software Engineering and Methodology. Volume 33, Issue 3, Article No.: 59, Pages 1 - 32. https://doi.org/10.1145/3631973

APÊNDICE A – QUESTIONÁRIO SOBRE A PRIORIZAÇÃO DA REMOÇÃO DE TEST SMELLS

TERMO DE CONSENTIMENTO LIVRE E ESCLARECIDO

Você está sendo convidado (a) como voluntário (a) a participar de uma pesquisa sobre a percepção dos desenvolvedores de software acerca de Test Smells.

Os resultados desta pesquisa contribuirão para a construção de uma ferramenta de refatoração automática para remoção de Test Smells dos testes de unidade.

O tempo estimado para responder o questionário é de 10 (dez) minutos.

Caso tenha alguma dúvida sobre o questionário, você poderá contatar Adriano Pizzini pelo e-mail adriano.pizzini@ppgia.pucpr.br.

Para acessar o texto completo do Termo de Consentimento Livre e Esclarecido (TCLE) aprovado, clique aqui.

Para acessar o material de apoio com detalhes sobre Test Smells, clique aqui.

Clicando em aceitar, você declara que concorda com o Termo de Consentimento Livre e Esclarecido (TCLE).

( ) Aceito e sou maior de 18 anos.
( ) Não aceito e/ou sou menor de 18 anos.

1) Qual a sua função?
a) Desenvolvedor
b) Analista de qualidade
Outra
1.1) Por favor, especifique a sua função:

2) Quanto tempo você tem de experiência no desenvolvimento de testes de unidade?
a) Não tenho experiência
b) Menos de 1 ano
c) De 1 a 3 anos
d) Mais de 3 até 5 anos
e) Mais de 5 até 10 anos
f) Mais de 10 anos

3) Qual o seu grau de conhecimento sobre Test Smells?
a) Nenhum, não tenho conhecimento sobre Test Smells
b) Pouco, tenho conhecimento teórico sobre Test Smells
c) Razoável, consigo identificar Test Smells no código de teste de unidade
d) Muito, a identificação/remoção de Test Smells fazem parte da minha rotina de trabalho

Observação: as questões 4 até 11 pedem que o participante responda à pergunta para cada Test Smell. Para evitar a replicação do mesmo conteúdo, os Test Smells serão incluídos apenas na primeira pergunta.

4) Qual a sua percepção sobre a frequência de ocorrência dos Test Smells abaixo?

Test Smell x Resposta	Não sei responder	Nunca ocorrem	Raramente ocorrem	Às vezes ocorrem	Muitas vezes ocorrem	Sempre ocorrem
Assertion Roulette						
Conditional Test Logic						
Constructor Initialization						
Duplicate Assertion						
Eager Test						
Empty Test						
Exception Handling						
General Fixture						
Ignored Test						
Lazy Test						
Magic Number Test						
Mystery Guest						
Redundant Assertion						
Redundant Print						
Resource Optimism						
Sensitive Equality						
Sleepy Test						
Test Code Dupplication						
Unknown Test						
Verbose Test						

5) Qual a sua percepção sobre a presença de cada um dos Test Smells prejudicar a legibilidade do código de um teste de unidade?

Test Smell x Resposta	Não sei responder	Nunca prejudica	Raramente prejudica	Às vezes prejudica	Muitas vezes prejudica	Sempre prejudica
Assertion Roulette						
. . .						
Verbose Test						

6) Qual a sua percepção sobre a presença de cada um dos Test Smells estar relacionada com a ocorrência de erros/falhas nos testes de unidade?

Test Smell x Resposta	Não sei responder	Nunca está relacionada	Raramente está relacionada	Às vezes está relacionada	Muitas vezes está relacionada	Sempre está relacionada
Assertion Roulette						
. .						

.						
Verbose Test						

7) Qual a sua percepção sobre a presença de cada um dos Test Smells no código do teste de unidade estar relacionada com a ocorrência de erros no sistema sendo testado?

Test Smell x Resposta	Não sei responder	Nunca prejudica	Raramente prejudica	Às vezes prejudica	Muitas vezes prejudica	Sempre prejudica
Assertion Roulette						
. . .						
Verbose Test						

a) Não sei responder
b) Nunca está relacionada
c) Raramente está relacionada
d) Às vezes está relacionada
e) Muitas vezes está relacionada
f) Sempre está relacionada

8) Qual a sua percepção sobre o benefício da remoção de cada um dos Test Smells abaixo?
a) Não sei responder
b) Não é benéfica
c) Pouco benéfica
d) Relativamente benéfica
e) Muito benéfica
f) Extremamente benéfica

9) Qual a sua percepção sobre o esforço necessário para a remoção de cada um dos Test Smells abaixo?
Considere o esforço para remover cada ocorrência de cada Test Smell, incluindo a mudança do código do teste, compilação e testes.

a) Não sei responder
b) Esforço muito baixo (1 min)
c) Esforço baixo (10 min)
d) Esforço razoável (30 min)
e) Esforço elevado (1h)
f) Esforço muito elevado (mais de 2h)

10) Com base nos Test Smells listados abaixo, qual é a sequência que representa a prioridade de remoção dos Test Smells das classes de teste de unidade?
Mova um Test Smell para cima para indicar que ele será removido antes de outro Test Smell, ou para baixo para indicar que ele será removido depois de outro Test Smell.

11) A respeito da ordem de remoção dos Test Smells da questão anterior, utilize o espaço abaixo caso queira deixar algum comentário/explicação.

APÊNDICE B – CONVITE PARA PARTICIPAR DA SIMULAÇÃO DE USO DO REFATORADOR

Prezado (a), tudo bem?

Estou convidando-o (a) para participar da avaliação de um refatorador automático de código de testes de unidade por meio de uma simulação de uso do aplicativo.

O objetivo da minha pesquisa é identificar como a remoção automática de Test Smells pode apoiar a equipe de desenvolvimento na melhoria da qualidade dos testes de unidade, tendo em vista que eles serão impactados diretamente pelas mudanças nos testes de unidade realizadas pelo refatorador desenvolvido.

Assim, é importante que o (a) participante tenha a experiência com testes de unidade e, preferencialmente, sobre Test Smells.

O (a) participante executará o removedor automático de Test Smells e avaliará os resultados da remoção por meio da comparação das versões original e refatorada do código do teste de unidade.

Encaminho anexo os seguintes documentos:
- O Termo de Consentimento Livre e Esclarecido (TCLE), o qual preciso que seja devolvido assinado pelos interessados em colaborar. A assinatura pode ser digital;
- O Termo de Confidencialidade do Uso de Dados (TCUD), no qual informamos (eu e minhas orientadoras) de que os dados serão usados anonimamente;
- Material informativo sobre Test Smells, com algumas descrições e exemplos de smells que podem ser encontrados nos testes de unidade;
- O processo definido para a refatoração automática, que precisa ser analisado antes da realização da simulação de uso.

Sobre o processo encaminhado anexo, como o material ainda não está publicado, peço sua colaboração no sentido de não difundir o mesmo.

Desde já, conforme informado no TCLE, informo que a participação na simulação depende da concordância do (a) participante de que a simulação seja gravada. O (a) participante pode desistir de colaborar a qualquer momento, sem precisar explicar os motivos para tal, conforme descrito no TCLE, de modo que a gravação (se tiver sido iniciada) será descartada.

Caso tenha(m) interesse, os resultados da pesquisa poderão ser posteriormente acessados na seção de teses do PPGIa (https://www.ppgia.pucpr.br/pt/).

Se houver alguma dúvida sobre o convite ou sobre o material encaminhado, por favor, contate-me por e-mail ou pelo telefone (99) 99999-9999.

Caso tenha interesse em participar, peço que devolva o TCLE assinado (pode ser a assinatura digital do GOV.BR)

Fico no aguardo do seu aceite para que possamos combinar a data da simulação.

Adriano Pizzini

Doutorando no Programa de Pós-Graduação em Informática (PPGIa)
PUCPR

APÊNDICE C – MATERIAL DE APOIO SOBRE TEST SMELLS

1. Assertion Roulette (AR)

Definição: um método de teste com múltiplas afirmações sem mensagens explicativas.
Exemplo: na imagem abaixo, as afirmações das linhas 56 e 57 não possuem explicação.

```
49      @Test
        public void testCodec() throws IOException {
51        int inputSize = 500_000;
52
53        byte[] input = generateTestData(inputSize);
54
55        Codec codecInstance = CodecFactory.fromString( s: codec).createInstance();
56        assertTrue( condition: codecClass.isInstance( obj: codecInstance));
57        assertTrue( condition: codecInstance.getName().equals( anObject: codec));
58
59        ByteBuffer inputByteBuffer = ByteBuffer.wrap( array: input);
60        ByteBuffer compressedBuffer = codecInstance.compress( uncompressedData: inputByteBuffer);
```

Figura 7-1: Classe TestAllCodecs.java do projeto Apache Avro, versão 1.11.1

2. Conditional Test Logic (CTL)

Definição: um método de teste que contém uma instrução condicional como pré-requisito para executar a instrução de teste.
Exemplo: nas linhas 283 e 286 da figura abaixo podem ser observadas uma estrutura de repetição e uma de seleção, respectivamente.

```
274     @Test
        public void testSettersCreatedByDefault() throws IOException {
276         SpecificCompiler compiler = createCompiler();
277         assertTrue( condition: compiler.isCreateSetters());
278         compiler.compileToDestination( src: this.src,  dst: this.OUTPUT_DIR.getRoot());
279         assertTrue( condition: this.outputFile.exists());
280         int foundSetters = 0;
281         try (BufferedReader reader = new BufferedReader(new FileReader( file: this.outputFile))) {
282             String line;
283             while ((line = reader.readLine()) != null) {
284                 // We should find the setter in the main class
285                 line = line.trim();
286                 if (line.startsWith( prefix: "public void setValue(")) {
287                     foundSetters++;
288                 }
289             }
290         }
291         assertEquals( message: "Found the wrong number of setters",  expected: 1,  actual: foundSetters);
292     }
```

Figura 7-2: Classe de teste TestSpecificCompiler.java do projeto Apache Avro, versão 1.11.1

3. Constructor Initialization (CI)

Definição: uma classe de teste que contém um construtor.
Exemplo: na figura abaixo pode ser observado o construtor declarado na linha 31.

```
29  class MemoryTypeSolverTest extends AbstractTypeSolverTest<MemoryTypeSolver> {
30
31      public MemoryTypeSolverTest() {
32          super(MemoryTypeSolver::new);
33      }
34
35      /**
36       * When solving a type that isn't registered in the memory should fail, while
37       * a existing type should be solved.
38       */
39      @Test
        void solveNonExistentShouldFailAndExistentTypeShouldSolve() {
```

Figura 7-3: Classe de teste MemoryTypeSolverTest.java do projeto JavaParser, versão 3.25.1

4. Duplicate Assert (DA)

Definição: ocorre quando um método de teste tem a afirmação exata diversas vezes no mesmo método de teste.

Exemplo: na figura abaixo, as afirmações das linhas 71, 77 e 82 são iguais. O mesmo ocorre nas linhas 75, 80 e 85.

```
67      @Test
        public void testEquals() {
69          BorderArrangement b1 = new BorderArrangement();
70          BorderArrangement b2 = new BorderArrangement();
71          assertTrue( condition: b1.equals( obj: b2));
72          assertTrue( condition: b2.equals( obj: b1));
73
74          b1.add(new EmptyBlock( width: 99.0,  height: 99.0),  key: null);
75          assertFalse( condition: b1.equals( obj: b2));
76          b2.add(new EmptyBlock( width: 99.0,  height: 99.0),  key: null);
77          assertTrue( condition: b1.equals( obj: b2));
78
79          b1.add(new EmptyBlock( width: 1.0,  height: 1.0),  key: RectangleEdge.LEFT);
80          assertFalse( condition: b1.equals( obj: b2));
81          b2.add(new EmptyBlock( width: 1.0,  height: 1.0),  key: RectangleEdge.LEFT);
82          assertTrue( condition: b1.equals( obj: b2));
83
84          b1.add(new EmptyBlock( width: 2.0,  height: 2.0),  key: RectangleEdge.RIGHT);
85          assertFalse( condition: b1.equals( obj: b2));
86          b2.add(new EmptyBlock( width: 2.0,  height: 2.0),  key: RectangleEdge.RIGHT);
87          assertTrue( condition: b1.equals( obj: b2));
88
89          b1.add(new EmptyBlock( width: 3.0,  height: 3.0),  key: RectangleEdge.TOP);
90          assertFalse( condition: b1.equals( obj: b2));
91          b2.add(new EmptyBlock( width: 3.0,  height: 3.0),  key: RectangleEdge.TOP);
92          assertTrue( condition: b1.equals( obj: b2));
```

Figura 7-4: Classe de teste BorderArrangementTest.java do projeto JFreeChart, versão 1.5.3, afetada pelo Duplicate Assert.

5. Eager Test (ET)

Definição: um método de teste que chama vários métodos do objeto a ser testado.

Exemplo: na figura abaixo, é possível identificar que o software é estimulado nas linhas 74, 76, 79, 81, entre outras, por meio da chamada do método add(). Após cada estímulo, um teste é realizado na sequência (linhas 75, 77, 80 e 82, entre outras).

```
67      @Test
69      public void testEquals() {
70          BorderArrangement b1 = new BorderArrangement();
71          BorderArrangement b2 = new BorderArrangement();
72          assertTrue( condition:b1.equals( obj:b2));
73          assertTrue( condition:b2.equals( obj:b1));
74
75          b1.add(new EmptyBlock( width:99.0,  height:99.0),  key:null);
76          assertFalse( condition:b1.equals( obj:b2));
77          b2.add(new EmptyBlock( width:99.0,  height:99.0),  key:null);
78          assertTrue( condition:b1.equals( obj:b2));
79
80          b1.add(new EmptyBlock( width:1.0,  height:1.0),  key:RectangleEdge.LEFT);
81          assertFalse( condition:b1.equals( obj:b2));
82          b2.add(new EmptyBlock( width:1.0,  height:1.0),  key:RectangleEdge.LEFT);
83          assertTrue( condition:b1.equals( obj:b2));
84
85          b1.add(new EmptyBlock( width:2.0,  height:2.0),  key:RectangleEdge.RIGHT);
86          assertFalse( condition:b1.equals( obj:b2));
87          b2.add(new EmptyBlock( width:2.0,  height:2.0),  key:RectangleEdge.RIGHT);
88          assertTrue( condition:b1.equals( obj:b2));
89
90          b1.add(new EmptyBlock( width:3.0,  height:3.0),  key:RectangleEdge.TOP);
91          assertFalse( condition:b1.equals( obj:b2));
92          b2.add(new EmptyBlock( width:3.0,  height:3.0),  key:RectangleEdge.TOP);
            assertTrue( condition:b1.equals( obj:b2));
```

Figura 7-5: Figura 4: Classe de teste BorderArrangementTest.java do projeto JFreeChart, versão 1.5.3, afetada pelo Eager Test.

6. Empty Test (EmT)

Definição: um método de teste vazio ou que não possui instruções executáveis.
Exemplo: na figura abaixo pode ser observado um método com a anotação @Test que não contém instruções.

```
18  public class OptionTokenizerTest {
19
20      @Test
        public void testEmpty() {
22
23      }
24
25      //
26      // @Test
27      // public void testEmpty() throws ScanException {
```

Figura 7-6: Classe OptionTokenizerTest.java do projeto Logback, versão 1.4.5, afetada pelo Eager Test

7. Exception Handling (EH ou ECT, de Exception Catching Throwing)

Definição: ocorre quando o tratamento de exceções personalizado é utilizado em vez do recurso de tratamento de exceções do JUnit.
Exemplo: na figura abaixo, pode ser observado o tratamento de exceção das linhas 92 (try) até 102 (catch).

```
86      /**
87       * Draws the chart with a null info object to make sure that no exceptions
88       * are thrown (a problem that was occurring at one point).
89       */
90      @Test
92      public void testDrawWithNullInfo() {
93          try {
94              BufferedImage image = new BufferedImage( width: 200 ,  height: 100,
95                      imageType: BufferedImage.TYPE_INT_RGB);
96              Graphics2D g2 = image.createGraphics();
97              this.chart.draw(g2, new Rectangle2D.Double( x: 0,  y: 0,  w: 200,  h: 100),  anchor: null,
98                      info: null);
99              g2.dispose();
100         }
101         catch (Exception e) {
102             fail( message: "There should be no exception.");
103         }
104     }
```

Figura 7-7: Classe de teste BarChartTest.java do projeto JFreeChart, versão, 1.5.3, afetada pelo Exception Handling

8. General Fixture (GF)

Definição: esse smell surge quando o fixture setUp() cria muitos objetos e os métodos de teste usam apenas um subconjunto.

Exemplo: na figura abaixo, o método "setUp" instancia dois objetos nas linhas 38 e 39. Porém, somente o objeto instanciado na linha 39 é referenciado no teste da linha 53.

```
36      @BeforeEach
37      public void setUp() throws Exception {
38          lc = new LoggerContext();
39          converter = new MDCConverter();
40          converter.start();
41          MDC.clear();
42      }
43
44      @AfterEach
45      public void tearDown() throws Exception {...6 lines }
51
52      @Test
54      public void testConvertWithOneEntry() {
55          String k = "MDCConverterTest_k" + diff;
56          String v = "MDCConverterTest_v" + diff;
57
58          MDC.put( key: k,  val: v);
59          ILoggingEvent le = createLoggingEvent();
60          String result = converter.convert( event: le);
61          assertEquals(k + "=" + v,  actual: result);
      }
```

Figura 7-8: MDCConverterTest.java do projeto Logback, versão 1.4.5, afetado pelo General Fixture

9. Ignored Test (IT)

Definição: um método de teste que usa uma anotação de ignorar que impede a execução do método de teste.

Exemplo: na imagem abaixo é possível identificar a anotação @Ignore no método de teste "testBuilderPerformance" na linha 177.

```
177     @Ignore
178     @Test
        public void testBuilderPerformance() {
180         int count = 1000000;
            List<Person> friends = new ArrayList<>( initialCapacity: 0);
182         List<String> languages = new ArrayList<>( c: Arrays.asList( a: "English",  a: "Java"));
183         long startTimeNanos = System.nanoTime();
184         for (int ii = 0; ii < count; ii++) {
                Person.newBuilder().setName("James Gosling").setYearOfBirth(1955).setCountry("US").setState("CA")
186                 .setFriends(friends).setLanguages(languages).build();
187         }
188         long durationNanos = System.nanoTime() - startTimeNanos;
189         double durationMillis = durationNanos / 1e6d;
190         System.out.println("Built " + count + " records in " + durationMillis + "ms (" + (count / (durationMillis / 1000d
191             + " records/sec, " + (durationMillis / count) + "ms/record");
192     }
```

Figura 7-9: A classe TestSpecificRecordBuilder.java do projeto Apache Avro, versão 1.11.1, afetada pelo Ignored Test.

10. Lazy Test (LT)

Definição: ocorre quando vários métodos de teste testam o mesmo método de objeto de produção.
Exemplo: na figura abaixo, o método "equals" é invocado no método de teste "testEquals", linhas 62, 66 e 68, e no método de teste "testCloning", na linha 81.

```
55      /** Confirm that the equals method can distinguish all the required fields ...3 lines */
58      @Test
        public void testEquals() {
60          DialCap c1 = new DialCap();
61          DialCap c2 = new DialCap();
62          assertTrue( condition: c1.equals( obj: c2));
63
64          // visible
65          c1.setVisible( visible: false);
66          assertFalse( condition: c1.equals( obj: c2));
67          c2.setVisible( visible: false);
68          assertTrue( condition: c1.equals( obj: c2));
69      }
70
71      /** Confirm that cloning works ...3 lines */
74      @Test
        public void testCloning() throws CloneNotSupportedException {
76          // test a default instance
77          DialCap c1 = new DialCap();
78          DialCap c2 = (DialCap) c1.clone();
79          assertTrue(c1 != c2);
80          assertTrue(c1.getClass() == c2.getClass());
81          assertTrue( condition: c1.equals( obj: c2));
```

Figura 7-10: Na classe AbstractDialLayerTest.java do projeto JFreeChart, versão 1.5.3, afetada pelo Lazy Test.

11. Magic Number Test (MNT)

Definição: um método de teste que contém valores numéricos não documentados.
Exemplo: na figura abaixo é possível identificar constantes numéricas nas linhas 136, 147 e 150.

```
134     @Test
135     public void testReplaceDataset() {
136         Number[][] data = new Integer[][]{{-30, -20}, {-10, 10}, {20, 30}};
137
138         CategoryDataset newData = DatasetUtils.createCategoryDataset(
139             rowKeyPrefix: "S",  columnKeyPrefix: "C", data);
140         LocalListener l = new LocalListener();
141         this.chart.addChangeListener( listener: l);
142         CategoryPlot plot = (CategoryPlot) this.chart.getPlot();
143         plot.setDataset( dataset: newData);
144         assertEquals( expected: true,  actual: l.flag);
145         ValueAxis axis = plot.getRangeAxis();
146         Range range = axis.getRange();
147         assertTrue(range.getLowerBound() <= -30,
148             "Expecting the lower bound of the range to be around -30: "
149             + range.getLowerBound());
150         assertTrue(range.getUpperBound() >= 30,
151             "Expecting the upper bound of the range to be around 30: "
152             + range.getUpperBound());
153     }
```

Figura 7-11: A classe AreaChartTest.java do projeto JFreeChart, versão 1.5.3, afetada pelo Magic Number Test

12. Mystery Guest (MG)

Definição: um teste que utiliza recursos externos, como um banco de dados, que contém dados de teste.

Exemplo: na figura abaixo é possível identificar a criação de um arquivo na linha 118.

```
110     @Test
111     public void testDirConcat() throws Exception {
112         Map<String, String> metadata = new HashMap<>();
113
114         for (int i = 0; i < 3; i++) {
115             generateData(name.getMethodName() + "-" + i + ".avro",  type: Type.STRING, metadata,  codec: DEFLATE);
116         }
117
118         File output = new File( parent: OUTPUT_DIR.getRoot(), name.getMethodName() + ".avro");
119
120         List<String> args = asList( a: INPUT_DIR.getRoot().getAbsolutePath(),  a: output.getAbsolutePath());
121         int returnCode = new ConcatTool().run( in: System.in,  out: System.out,  err: System.err, args);
122
123         assertEquals( expected: 0,  actual: returnCode);
124         assertEquals(ROWS_IN_INPUT_FILES * 3,  actual: numRowsInFile(output));
125     }
```

Figura 7-12: A classe TestConcatTool.java do projeto Apache Avro, versão 1.11.1, afetada pelo Mystery Guest.

13. Redundant Assertion (RA)

Definição: um método de teste que possui uma afirmação que é permanentemente verdadeira ou falsa.

Exemplo: na figura abaixo é possível identificar que as afirmações usam as mesmas constantes na comparação de igualdade, o que sempre produz o valor lógico verdadeiro.

```
55      @Test
56      public void testEquals() {
57          assertEquals( expected: AreaRendererEndType.LEVEL,  actual: AreaRendererEndType.LEVEL);
58          assertEquals( expected: AreaRendererEndType.TAPER,  actual: AreaRendererEndType.TAPER);
59          assertEquals( expected: AreaRendererEndType.TRUNCATE,  actual: AreaRendererEndType.TRUNCATE);
60      }
```

Figura 7-13: A classe AreaRendererEndTypeTest.java do projeto JFreeChart, versão 1.5.3, afetada pelo Redundant Assertion.

14. Redundant Print (RP)

Definição: um método de teste que possui instrução print.
Exemplo: na figura abaixo, é possível identificar o uso da instrução print nas linhas 270, 274 e 277.

```
267     @Test
        public void testSort1() {
269         DoubleColumn numberColumn = DoubleColumn.create( name: "test",  initialSize: 1_000_000_000);
270         System.out.println( x: "Adding doubles to column");
271         for (int i = 0; i < 100_000_000; i++) {
272             numberColumn.append( d: Math.random());
273         }
274         System.out.println( x: "Sorting");
275         Stopwatch stopwatch = Stopwatch.createStarted();
276         numberColumn.sortAscending();
277         System.out.println("Sort time in ms = " + stopwatch.elapsed( desiredUnit: TimeUnit.MILLISECONDS));
278     }
```

Figura 7-14: A classe NumberColumnTest.java do projeto TableSaw, versão 0.43.1, afetada pelo Redundant Print

15. Resource Optimism (RO)

Definição: um teste que faz uma suposição sobre a existência de recursos externos.
Exemplo: na figura abaixo, ocorre a instanciação de um objeto da classe "File" na linha 118. Porém, os métodos do objeto são invocados sem identificar se o arquivo existe.

```
110     @Test
        public void testDirConcat() throws Exception {
112         Map<String, String> metadata = new HashMap<>();
113
114         for (int i = 0; i < 3; i++) {
115             generateData(name.getMethodName() + "-" + i + ".avro",  type: Type.STRING, metadata,  codec: DEFLATE);
116         }
117
118         File output = new File( parent: OUTPUT_DIR.getRoot(), name.getMethodName() + ".avro");
119
120         List<String> args = asList( a: INPUT_DIR.getRoot().getAbsolutePath(),  a: output.getAbsolutePath());
121         int returnCode = new ConcatTool().run( in: System.in,  out: System.out,  err: System.err, args);
122
123         assertEquals( expected: 0,  actual: returnCode);
124         assertEquals(ROWS_IN_INPUT_FILES * 3,  actual: numRowsInFile(output));
125     }
```

Figura 7-15: A classe TestConcatTool.java do projeto Apache Avro, versão 1.11.1, afetada pelo Resource Optimism.

16. Sensitive Equality (SE)

Definição: ocorre quando uma asserção tem uma afirmação de igualdade usando o método toString
Exemplo: na figura abaixo, é possível identificar o método toString sendo invocado nas linhas em destaque.

```
124     @Test
        public void declaredVsNonDeclaredMethods() {
126         try (ScanResult scanResult = new ClassGraph().enableAllInfo()
127             .acceptPackages( packageNames: DeclaredVsNonDeclaredTest.class.getPackage().getName()).scan()) {
128             final ClassInfo A = scanResult.getClassInfo( className: A.class.getName());
129             final ClassInfo B = scanResult.getClassInfo( className: B.class.getName());
130             assertThat( actual: B.getFieldInfo( fieldName: "x").getClassInfo().getName()).isEqualTo( expected: B.class.getName()
131             assertThat( actual: B.getFieldInfo( fieldName: "z").getClassInfo().getName()).isEqualTo( expected: A.class.getName()
132             assertThat( actual: A.getFieldInfo().get( index: 0).getTypeDescriptor().toString()).isEqualTo( expected: "float");
133             assertThat( actual: B.getFieldInfo().get( index: 0).getTypeDescriptor().toString()).isEqualTo( expected: "int");
134             assertThat( actual: B.getMethodInfo().toString())
135                 .isEqualTo("[void y(final int x, final int y), void w(), abstract void y(java.lang.String x), "
136                     + "abstract void y(java.lang.Integer x)]");
137             assertThat( actual: B.getDeclaredMethodInfo().toString())
138                 .isEqualTo( expected: "[void y(final int x, final int y), void w()]");
139         }
140     }
```

Figura 7-16: A classe CompositeTitleTest.java do JFreeChart, versão 1.5.3, afetada pelo Sensitive Equality (em destaque)

17. Sleepy Test (ST)

Definição: ocorre quando um método de teste possui uma espera explícita (Thread.sleep()).
Exemplo: na figura abaixo, é possível identificar a invocação do método “sleep” na linha 163.

```
156     @Test
        public void testTryLock_Unlock5() throws Exception {
158         when( methodCall: cache.PUT_IF_ABSENT( key: any(), value: any(), expireAfterWrite: anyLong(), timeUnit: any())).then(i ->
159                 //change expire time
160             concreteCache.PUT_IF_ABSENT( key: "key", value: i.getArgument( i: 1).toString(), expireAfterWrite: 100, timeUnit: TimeUnit.HOU
161         );
            AutoReleaseLock lock = cache.tryLock( key: "key", expire: 1, timeUnit: TimeUnit.MILLISECONDS);
163         Thread.sleep( millis: 2);
164         lock.close();
165         assertNotNull( actual: concreteCache.GET( key: "key"));
166     }
```

Figura 7-17: A classe CacheTest.java do projeto JetCache, versão 2.7.3, afetada pelo Sleepy Test.

18. Test Code Duplication (TCD, ou clones)

Definição: ocorre quando os clones de código estão contidos no teste.
Exemplo: na figura abaixo, os dois métodos de teste apresentados apresentam diferença entre as linhas 236 e 254.

```
226     @Test
        public void test2502355_zoomInDomain() {
228         DefaultXYDataset dataset = new DefaultXYDataset();
229         JFreeChart chart = ChartFactory.createXYLineChart( title: "TestChart", xAxisLabel: "X",
230                 yAxisLabel: "Y", dataset, orientation: PlotOrientation.VERTICAL, legend: false, tooltips: false, urls: false);
231         XYPlot plot = (XYPlot) chart.getPlot();
232         plot.setDomainAxis( index: 1, new NumberAxis( label: "X2"));
233         ChartPanel panel = new ChartPanel(chart);
234         chart.addChangeListener( listener: this);
235         this.chartChangeEvents.clear();
236         panel.zoomInDomain( x: 1.0, y: 2.0);
237         assertEquals( expected: 1, actual: this.chartChangeEvents.size());
238     }
239
240     /** Checks that a call to the zoomOutDomain() method, for a plot with more ...4 lines */
244     @Test
        public void test2502355_zoomOutDomain() {
246         DefaultXYDataset dataset = new DefaultXYDataset();
247         JFreeChart chart = ChartFactory.createXYLineChart( title: "TestChart", xAxisLabel: "X",
248                 yAxisLabel: "Y", dataset, orientation: PlotOrientation.VERTICAL, legend: false, tooltips: false, urls: false);
249         XYPlot plot = (XYPlot) chart.getPlot();
250         plot.setDomainAxis( index: 1, new NumberAxis( label: "X2"));
251         ChartPanel panel = new ChartPanel(chart);
252         chart.addChangeListener( listener: this);
253         this.chartChangeEvents.clear();
254         panel.zoomOutDomain( x: 1.0, y: 2.0);
255         assertEquals( expected: 1, actual: this.chartChangeEvents.size());
256     }
```

Figura 7-18: A classe ChartPanelTest.java do projeto JFreeChart, versão 1.5.3, afetada pelo Test Code Duplication

19. Unknown Test (UT)

Definição: um método de teste sem declaração de afirmação e nome não descritivo.
Exemplo: na figura abaixo, pode ser observado um método que não contém nenhuma afirmação.

```
62      @Test
63      public void testDrawWithNullInfo() {
64          MeterPlot plot = new MeterPlot(new DefaultValueDataset( value: 60.0));
65          plot.addInterval(new MeterInterval( label: "Normal", new Range( lower: 0.0,  upper: 80.0)));
66          JFreeChart chart = new JFreeChart(plot);
67          BufferedImage image = new BufferedImage( width: 200,  height: 100,
68                  imageType: BufferedImage.TYPE_INT_RGB);
69          Graphics2D g2 = image.createGraphics();
70          chart.draw(g2, new Rectangle2D.Double( x: 0,  y: 0,  w: 200,  h: 100),  anchor: null,  info: null);
71          g2.dispose();
72          //FIXME we should really assert a value here
73      }
```

Figura 7-19: A classe MeterChartTest.java do projeto JFreeChart, versão 1.5.3, afetada pelo Unknown Test

1. Verbose Test (VT)

Definição: teste código complexo e não simples ou limpo. Por padrão, o JNose considera um teste com 30 linhas ou mais como afetado pelo Verbose Test smell.
Exemplo: na figura abaixo, pode ser observada uma ocorrência do Verbose Test smell na qual o método de teste tem mais de 30 linhas.

```
103     @Test
104     public void testGetListeners() {
105         ChartPanel p = new ChartPanel( chart: null);
106         p.addChartMouseListener( listener: this);
107         EventListener[] listeners = p.getListeners( listenerType: ChartMouseListener.class);
108         assertEquals( expected: 1,  actual: listeners.length);
109         assertEquals( expected: this, listeners[0]);
110         // try a listener type that isn't registered
111         listeners = p.getListeners( listenerType: CaretListener.class);
112         assertEquals( expected: 0,  actual: listeners.length);
113         p.removeChartMouseListener( listener: this);
114         listeners = p.getListeners( listenerType: ChartMouseListener.class);
115         assertEquals( expected: 0,  actual: listeners.length);
116
117         // try a null argument
118         boolean pass = false;
119         try {
120             listeners = p.getListeners((Class) null);
121         }
122         catch (NullPointerException e) {
123             pass = true;
124         }
125         assertTrue( condition: pass);
126
127         // try a class that isn't a listener
128         pass = false;
129         try {
130             listeners = p.getListeners( listenerType: Integer.class);
131         }
132         catch (ClassCastException e) {
133             pass = true;
134         }
135         assertTrue( condition: pass);
136     }
```

Figura 7-20: A classe ChartPanelTest.java do projeto JFreeChart, versão 1.5.3, afetada pelo Verbose Test.

APÊNDICE D – TERMO DE CONFIDENCIALIDADE DE USO DOS DADOS (TCUD)

Nós, abaixo assinados, pesquisadores envolvidos no projeto de título Test Smells: Um estudo sobre problemas de qualidade em testes de unidade, nos comprometemos a manter a confidencialidade sobre os dados coletados sobre os projetos das organizações participantes e dos respondentes, bem como a privacidade de seus conteúdos, como preconizam os Documentos Internacionais e as Resoluções 466/12 e 510/16, do Conselho Nacional de Saúde.

Informamos que os dados a serem coletados dizem respeito à experiência e conhecimento dos entrevistados, bem como dados coletados do repositório de código da organização.

Curitiba, 01 de junho de 2023.

Envolvidos na manipulação e coleta dos dados:

Nome completo	CPF	Assinatura
Andreia Malucelli		
Sheila dos Santos Reinehr		
Adriano Pessini		

APÊNDICE E – TERMO DE CONSENTIMENTO LIVRE E ESCLARECIDO

TERMO DE CONSENTIMENTO LIVRE E ESCLARECIDO

Você está sendo convidado(a) como voluntário(a) a participar do estudo **TEST SMELLS: UM ESTUDO SOBRE PROBLEMAS DE QUALIDADE EM TESTES DE UNIDADE**, que tem como objetivo compreender como a remoção automática de Test Smells pode apoiar a equipe na melhoria da qualidade dos testes de unidade. Acreditamos que esta pesquisa seja importante porque visamos analisar as ações realizadas por desenvolvedores para melhorar a qualidade do software.
Esta pesquisa foi aprovada pelo Comitê de Ética em Pesquisa da PUCPR no processo número 70362323.0.0000.0020.

PARTICIPAÇÃO NO ESTUDO

A sua participação no referido estudo será de responder ao questionário elaborado pelos pesquisadores para compreender o impacto que problemas de qualidade causam na manutenção de testes de unidade, bem como entender como a refatoração pode ser usada para auxiliar na resolução destes problemas.

RISCOS E BENEFÍCIOS

Por meio deste Termo de Consentimento Livre e Esclarecido você está sendo alertado de que, da pesquisa a se realizar, pode esperar alguns benefícios, tais como: compreender problemas de qualidade que afetam testes de unidade, identificar oportunidades e promover a melhoria da qualidade dos testes de unidade por meio da refatoração. É possível que aconteçam os seguintes desconfortos ou riscos em sua participação, tais como sentir-se desconfortável com alguma questão que lhe seja feita. Para minimizar tais riscos, nós pesquisadores tomaremos as seguintes medidas: nenhuma informação será divulgada de forma individualizada ou atribuindo sua identidade, você poderá pedir para se retirar do estudo a qualquer momento.

SIGILO E PRIVACIDADE

Nós pesquisadores garantiremos a você que sua privacidade será respeitada, ou seja, seu nome ou qualquer outro dado ou elemento que possa, de qualquer forma, lhe identificar, será mantido em sigilo. Nós pesquisadores nos responsabilizaremos pela guarda e confidencialidade dos dados, bem como a não exposição dos dados de pesquisa.

AUTONOMIA

Nós lhe asseguramos assistência durante toda pesquisa, bem como garantiremos seu livre acesso a todas as informações e esclarecimentos adicionais sobre o estudo e suas consequências, enfim, tudo o que você queira saber antes, durante e depois de sua participação. Também informamos que você pode se recusar a participar do estudo, ou retirar seu consentimento a qualquer momento, sem precisar justificar, e de, por desejar sair da pesquisa, não sofrerá qualquer prejuízo à assistência que vem recebendo.

RESSARCIMENTO E INDENIZAÇÃO

No entanto, caso tenha qualquer despesa decorrente da participação nesta pesquisa, tais como transporte, alimentação entre outros, bem como de seu acompanhante, haverá ressarcimento dos valores gastos na forma seguinte: depósito em conta corrente.
De igual maneira, caso ocorra algum dano decorrente de sua participação no estudo, você será devidamente indenizado, conforme determina a lei.

CONTATO

Os pesquisadores envolvidos com o referido projeto são Andreia Malucelli, Sheila Reinehr e Adriano Pizzini, todos da Pontifícia Universidade Católica do Paraná (PUCPR) e com o último você poderá manter contato pelo telefone (47) 99789-7812 ou por e-mail no endereço adriano.pizzini@ppgia.pucpr.edu.br.

O Comitê de Ética em Pesquisa em Seres Humanos (CEP) é composto por um grupo de pessoas que estão trabalhando para garantir que seus direitos como participante de pesquisa sejam respeitados. Ele tem a obrigação de avaliar se a pesquisa foi planejada e se está sendo executada de forma ética. Se você achar que a pesquisa não

está sendo realizada da forma como você imaginou ou que está sendo prejudicado de alguma forma, você pode entrar em contato com o Comitê de Ética em Pesquisa da PUCPR (CEP) pelo telefone (41) 3271-2103 entre segunda e sexta-feira das 08h00 às 17h30 ou pelo e-mail nep@pucpr.br.

DECLARAÇÂO

Declaro que li e entendi todas as informações presentes neste Termo de Consentimento Livre e Esclarecido e tive a oportunidade de discutir as informações deste termo. Todas as minhas perguntas foram respondidas e eu estou satisfeito com as respostas. Entendo que receberei uma via assinada e datada deste documento e que outra via assinada e datada será arquivada nos pelo pesquisador responsável do estudo.

Enfim, tendo sido orientado quanto ao teor de todo o aqui mencionado e compreendido a natureza e o objetivo do já referido estudo, manifesto meu livre consentimento em participar, estando totalmente ciente de que não há nenhum valor econômico, a receber ou a pagar, por minha participação.

Dados do participante da pesquisa
Nome:
Telefone:
e-mail:

_____, _____ de _____ de _____.

_____ _____

Assinatura do participante da pesquisa | Assinatura do Pesquisador

USO DE IMAGEM E/OU ÁUDIO

Autorizo o uso de minha imagem e do meu áudio para fins da pesquisa, sendo seu uso restrito à análise a ser realizada no âmbito dessa pesquisa e que serão descartadas após a sua conclusão.

_____ _____

Assinatura do participante da pesquisa | Assinatura do Pesquisador

APÊNDICE F – EXEMPLO DE CLASSE DE TESTES DE UNIDADE FICTÍCIA

```java
package org.jfree.chart;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class DummyTest {

    @Test
    public void testExceptionCatchingThrowingSucess(){
        try{
            int x = Integer.parseInt("A");
            fail("Deve lançar a exceção");
        } catch (NumberFormatException e) {
            assertEquals("For input string: \"A\"", e.getMessage());
        }
    }

    @Test
    public void testExceptionCatchingThrowingFail(){
        try{
            int x = Integer.parseInt("1");
            assertEquals(x, 1);
        } catch (NumberFormatException e) {
            fail("Não deveria lançar a exceção");
        }
    }


    @Test
    public void testFirstClone(){
        int x = 10;
        int y = 20;
        assertTrue(x != y);
    }

    @Test
    public void testSecondClone(){
        int x = 10;
        int y = 20;
        assertFalse(x == y);
    }

    @Test
    public void testThirdClone(){
        int x = 10;
        int y = 20;
        assertFalse(x > y);
    }

    @Test
    public void testRedundantAssert(){
        int z = 10;
        assertTrue(true);
        assertEquals(z, 10);
        assertFalse(false);
        assertEquals(z, z);
    }
```

```
    @Test
    public void testEagerTestSmell(){
        int a = 10;
        int b = 20;
        assertTrue(a < b);

        int c = 20;
        int d = 10;
        assertTrue(c > d);
    }


    @Test
    public void testSimpleTest(){
        String a = "x";
        assertTrue(a.equals("x"));
    }
}
```

APÊNDICE G – GRÁFICOS DE CORRELAÇÃO COM FORÇA DE CORRELAÇÃO INSIGNIFICANTE SOBRE A VARIAÇÃO DA QUANTIDADE DE SMELLS APÓS A REMOÇÃO AUTOMÁTICA

Na Figura 1 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de AR e a variação na quantidade de ocorrências de MNT. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a insignificância da força da correlação positiva encontrada (*p-value*=<,001 e ρ=,053), indicando que a remoção de AR causou a remoção de ocorrências de MNT.

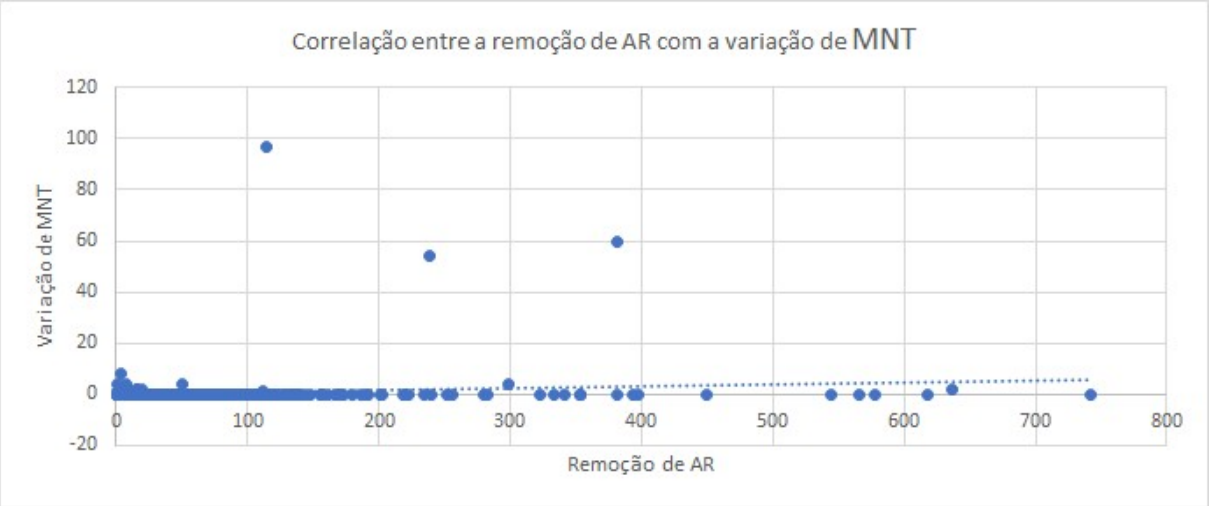

Figura 1 Gráfico de dispersão entre a remoção de AR e a variação de MnT (Fonte: o autor)

Na Figura 2 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de AR e a variação na quantidade de ocorrências de VT. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a insignificância da força da correlação positiva encontrada (*p-value*=<,001 e ρ=,279), indicando que a remoção de AR causou a remoção de ocorrências de VT.

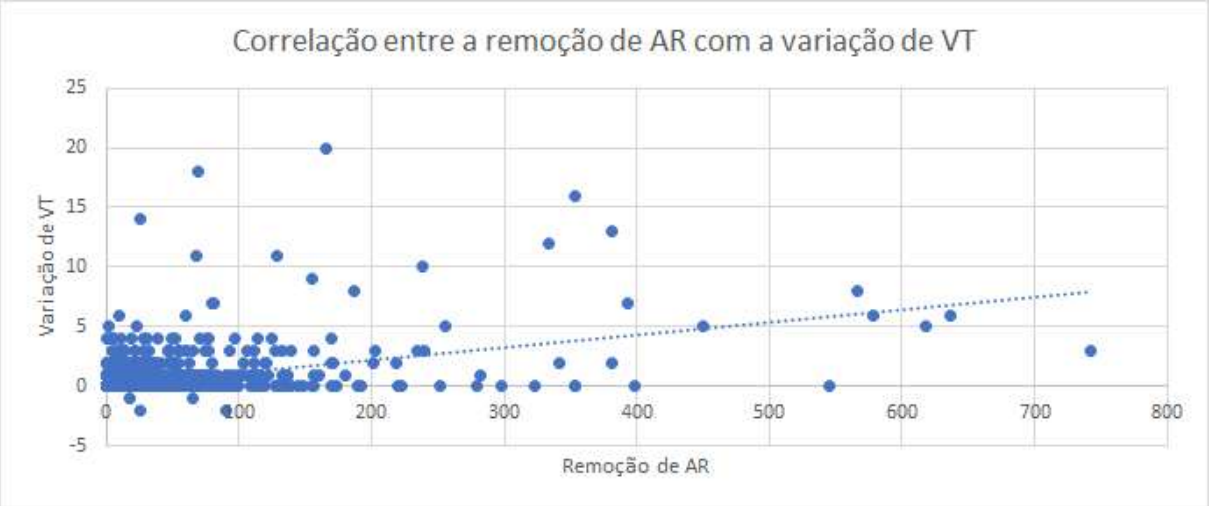

Figura 2 Gráfico de dispersão entre a remoção de AR e a variação de VT (Fonte: o autor)

Na Figura 3 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ECT e a variação na quantidade de ocorrências de AR. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a insignificância da força da correlação encontrada (*p-value*=<,422 e ρ=,089), indicando que a remoção de ECT causou a remoção de ocorrências de AR.

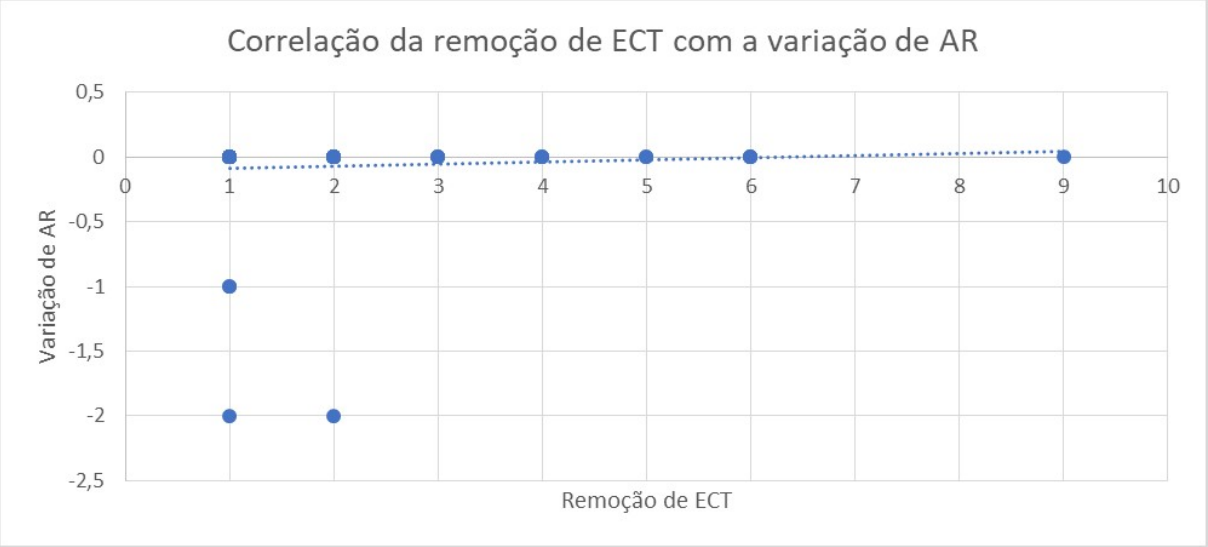

Figura 3 Gráfico de dispersão entre a remoção de ECT com a variação de AR (Fonte: o autor)

Na Figura 4 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ECT e a variação na quantidade de ocorrências de MNT. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante causada pela remoção do ECT

em relação à variação da quantidade de ocorrências de MNT (*p-value*=,419 e ρ=,089), indicando que a remoção de ECT causou a remoção de ocorrências de MNT.

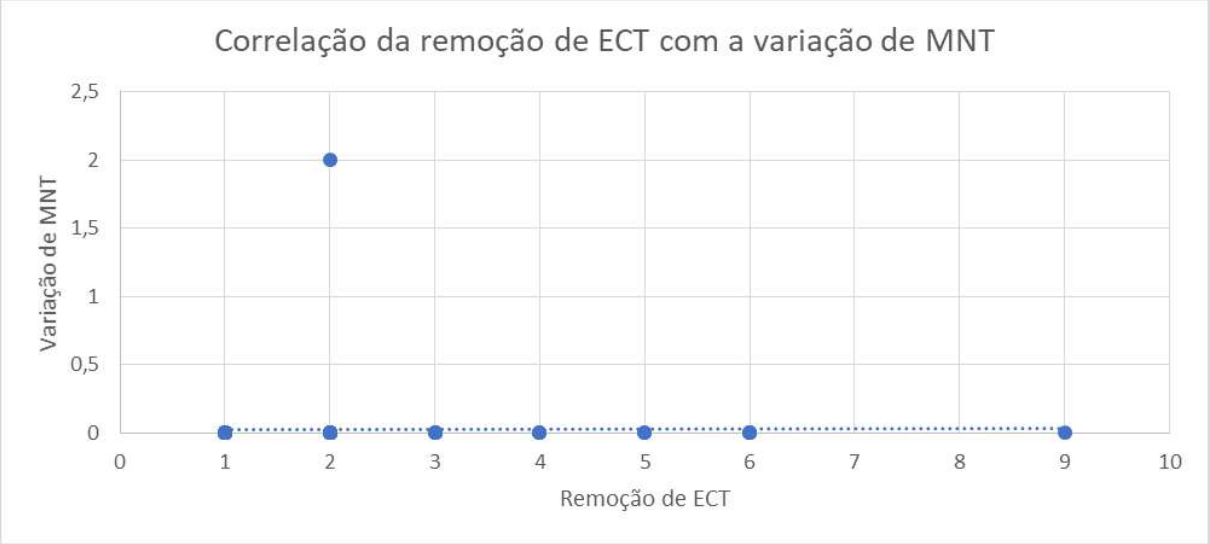

Figura 4 Gráfico de dispersão entre a remoção de ECT com a variação de MNT (Fonte: o autor)

Na Figura 5 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ECT e a variação na quantidade de ocorrências de VT. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante causada pela remoção do ECT em relação à variação da quantidade de ocorrências de VT (*p-value*=,021 e ρ=,252), indicando que a remoção de ECT causou a remoção de ocorrências de VT.

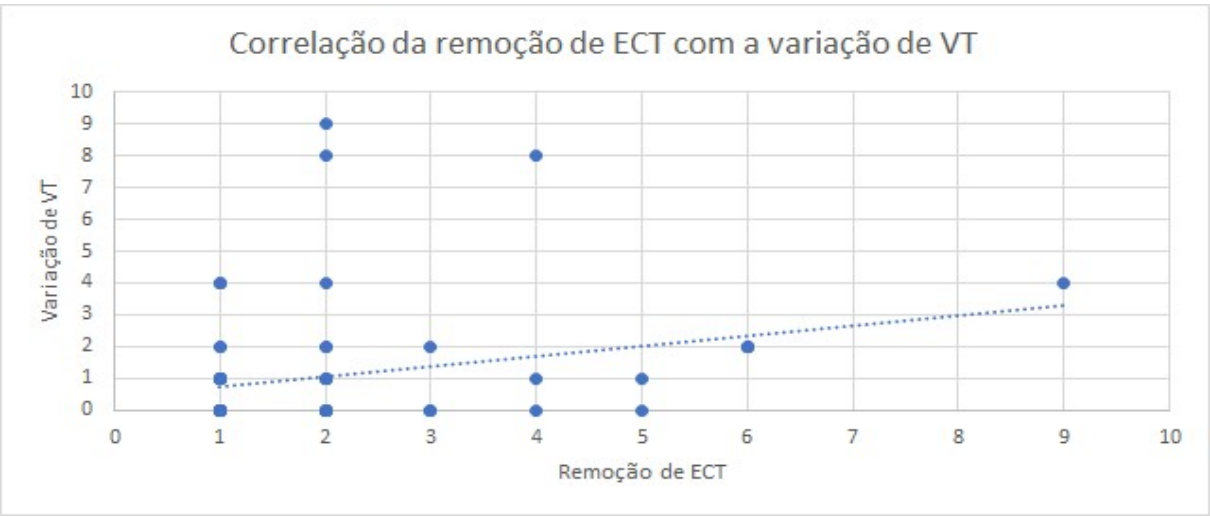

Figura 5 Gráfico de dispersão entre a remoção de ECT com a variação de VT (Fonte: o autor)

Na Figura 6 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ET e a variação na quantidade de ocorrências de

CI. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa insignificante causada pela remoção do ET em relação à variação da quantidade de ocorrências de CI (*p-value*=,538 e ρ=,014), o que indica que a remoção de ET causou a criação de ocorrências de CI.

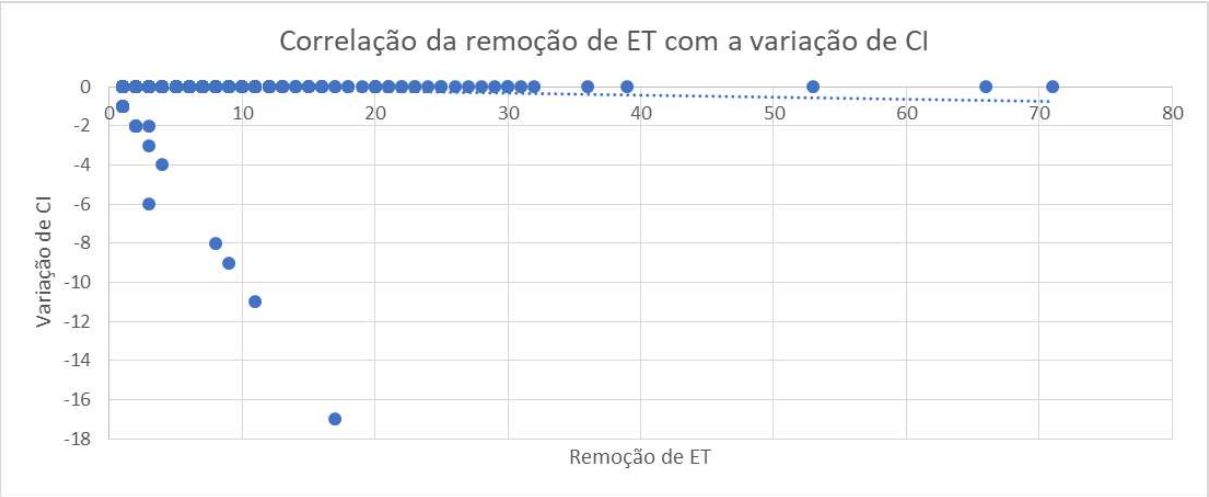

Figura 6 Gráfico de dispersão entre a remoção de ET com a variação de CI (Fonte: o autor)

Na Figura 7 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ET e a variação na quantidade de ocorrências de GF. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa insignificante entre a remoção do ET e a variação da quantidade de ocorrências de GF (*p-value*=,440 e ρ=-,017), indicando que a remoção de ET causou a criação de ocorrências de GF.

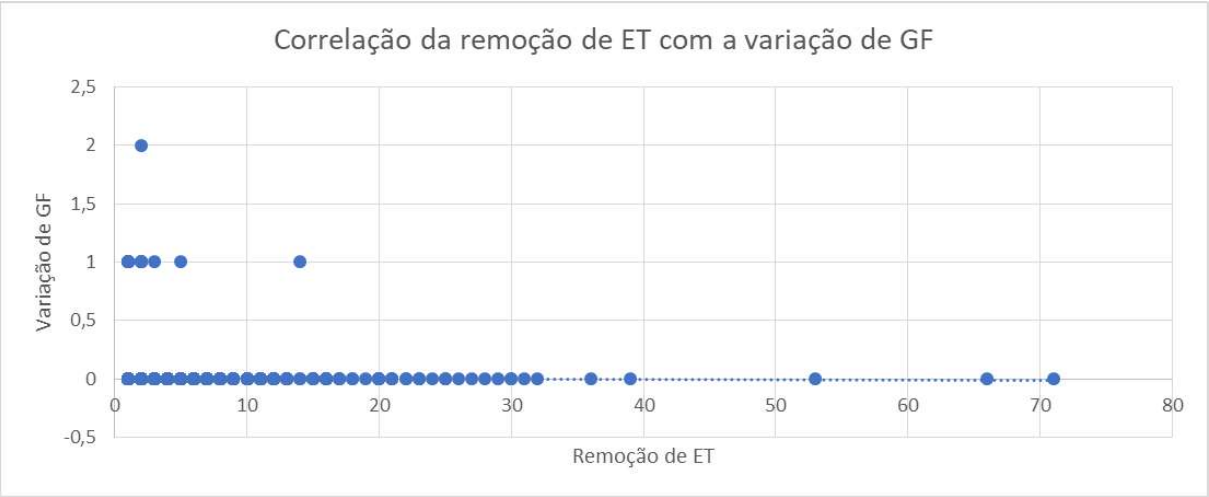

Figura 7 Gráfico de dispersão entre a remoção de ET com a variação de GF (Fonte: o autor)

Na Figura 8 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ET e a variação na quantidade de ocorrências de LT. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa insignificante entre a remoção do ET e a variação da quantidade de ocorrências de LT (*p-value*=<,001 e ρ=-,182), o que mostra que a remoção de ET causou a criação de ocorrências de LT.

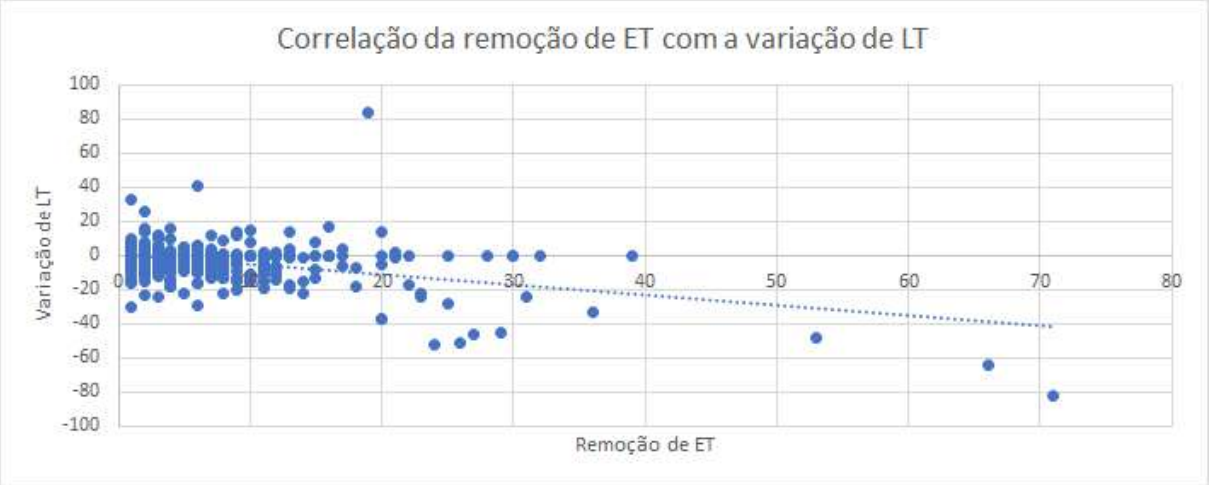

Figura 8 Gráfico de dispersão entre a remoção de ET com a variação de LT (Fonte: o autor)

Na Figura 9 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ET e a variação na quantidade de ocorrências de MNT. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do ET e a variação da quantidade de ocorrências de MNT (*p-value*=,069 e ρ=,040), o que indica que a remoção de ET causou a remoção de ocorrências de MNT.

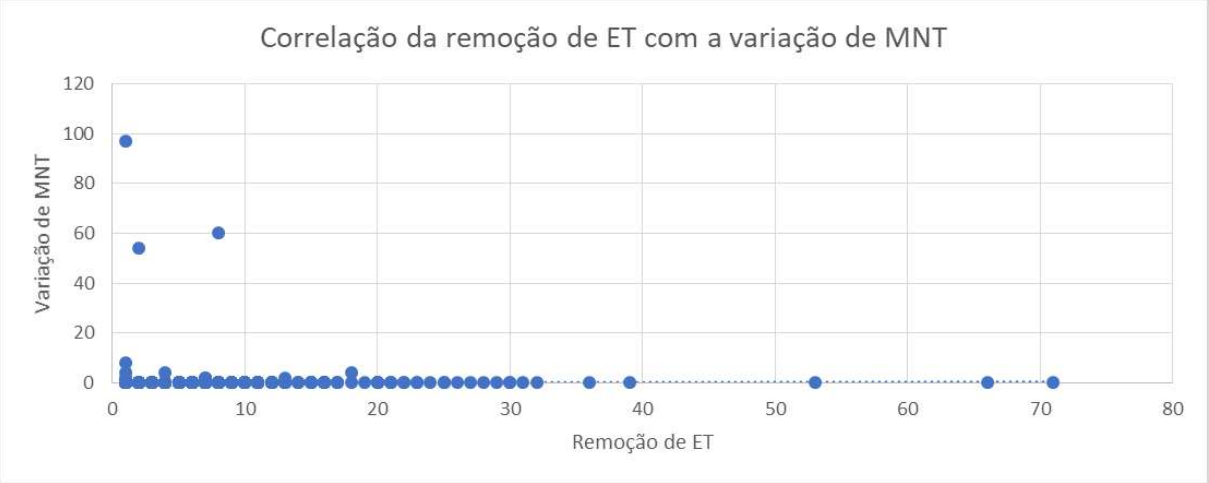

Figura 9 Gráfico de dispersão entre a remoção de ET com a variação de MNT (Fonte: o autor)

Na Figura 10 são mostrados o gráfico de dispersão e a linha linear de tendência entre a quantidade de remoções de ET e a variação na quantidade de ocorrências de VT. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do ET e a variação da quantidade de ocorrências de VT (*p-value*=<,001 e ρ=,269) e mostra que a remoção de ET causou a remoção de ocorrências de VT.

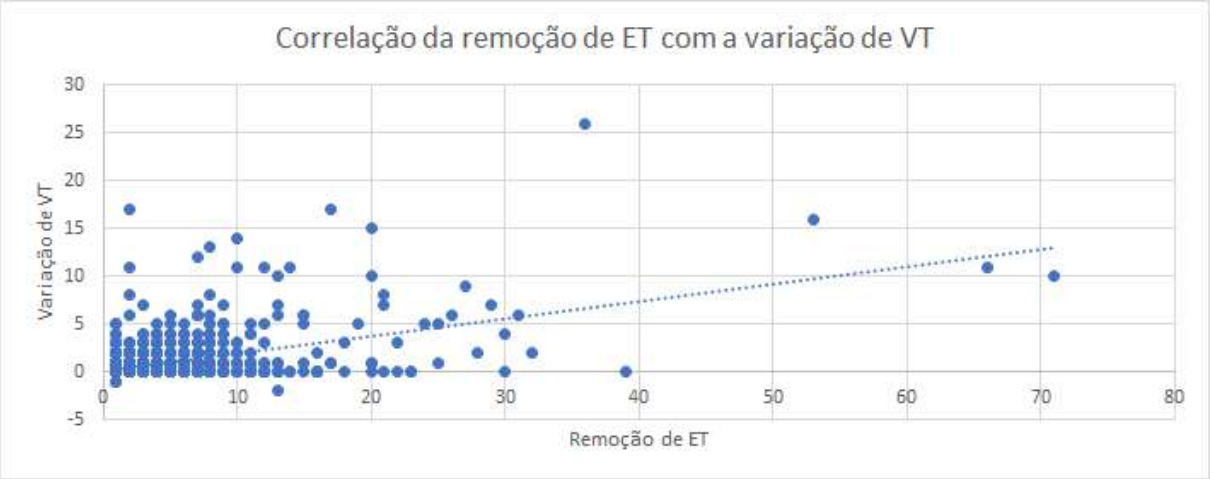

Figura 10 Gráfico de dispersão entre a remoção de ET e variação de ECT (Fonte: o autor)

A Figura 11 mostra o gráfico de dispersão da correlação entre a quantidade de RA removidos e a variação na quantidade do DA, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do AR e a variação da quantidade de ocorrências de DA (*p-value*=<,031 e ρ=,247) e mostra que a remoção de RA causou a remoção de ocorrências de DA.

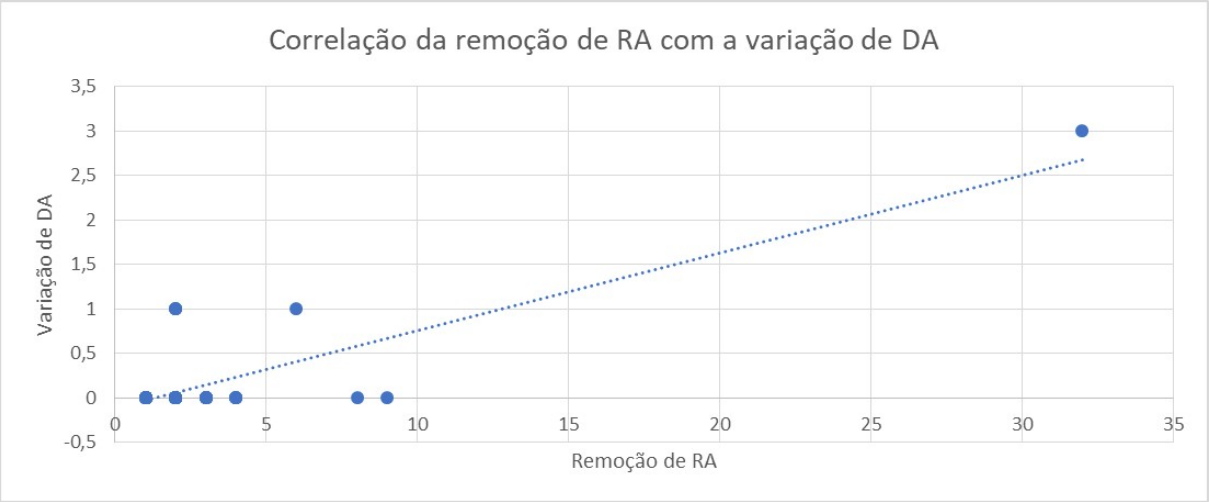

Figura 11 Gráfico de dispersão entre a remoção de RA e variação de DA (Fonte: o autor)

A Figura 12 mostra o gráfico de dispersão da correlação entre a quantidade de RA removidos e a variação na quantidade do EmT, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do AR e a variação da quantidade de ocorrências de EmT (*p-value*=,667 e ρ=,050) e mostra que a remoção de RA causou a remoção de ocorrências de EmT.

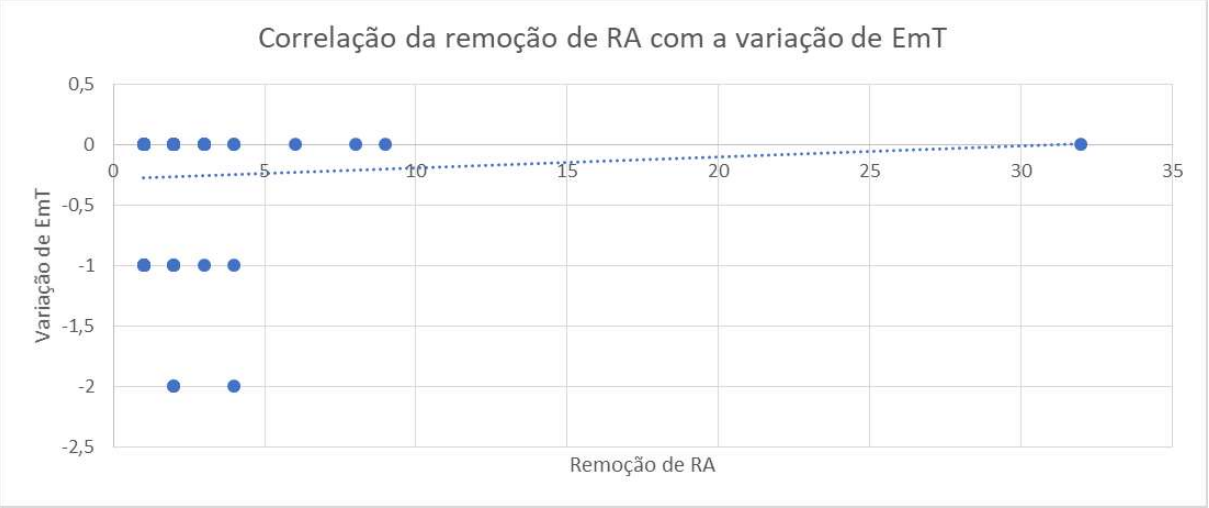

Figura 12 Gráfico de dispersão entre a remoção de RA com a variação de EmT (Fonte: o autor)

A Figura 13 mostra o gráfico de dispersão da correlação entre a quantidade de RA removidos e a variação na quantidade do ET, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do AR e a variação da quantidade

de ocorrências de ET (*p-value*=,503 e ρ=,078) e mostra que a remoção de RA causou a remoção de ocorrências de ET.

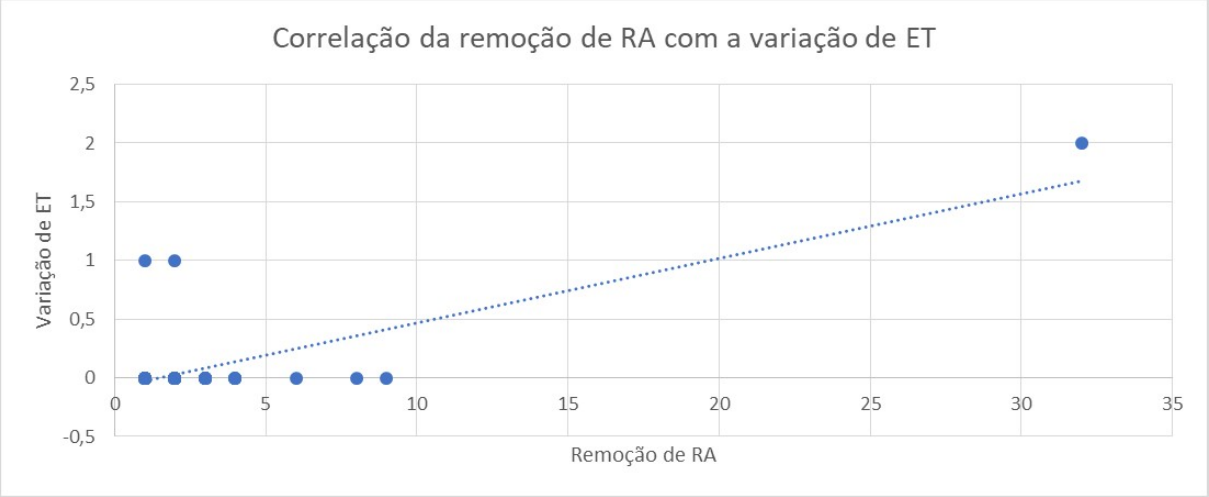

Figura 13 Gráfico de dispersão entre a remoção de RA com a variação de ET (Fonte: o autor)

A Figura 14 mostra o gráfico de dispersão da correlação entre a quantidade de RA removidos e a variação na quantidade do GF, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa insignificante entre a remoção do AR e a variação da quantidade de ocorrências de GF (*p-value*=,717 e ρ=-,042) e mostra que a remoção de RA causou a criação de ocorrências de GF.

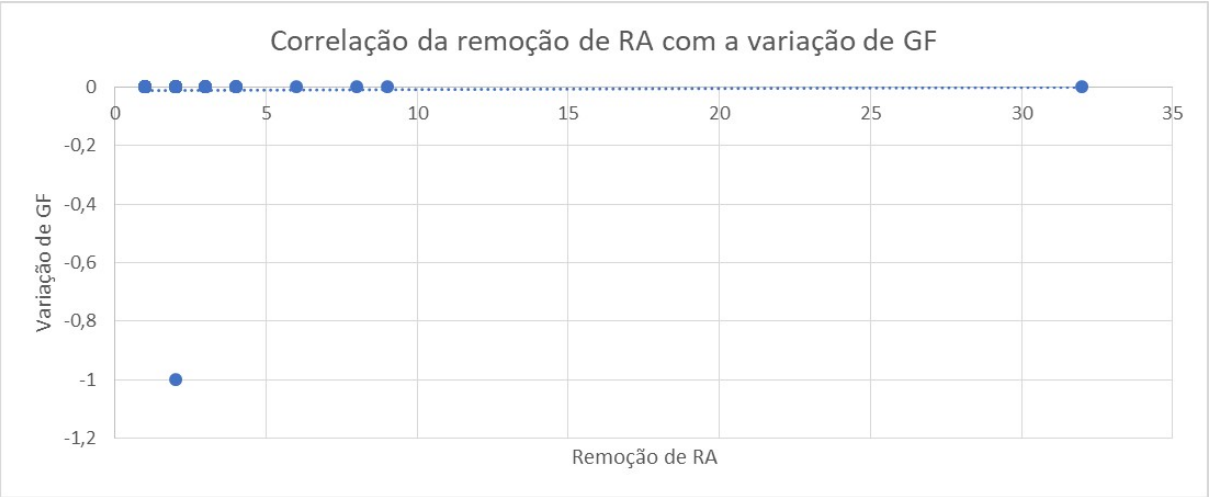

Figura 14 Gráfico de dispersão entre a remoção de RA com a variação de GF (Fonte: o autor)

A Figura 15 mostra o gráfico de dispersão da correlação entre a quantidade de RA removidos e a variação na quantidade do MNT, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com

a correlação positiva insignificante entre a remoção do AR e a variação da quantidade de ocorrências de MNT (*p-value*=,505 e ρ=,078) e mostra que a remoção de RA causou a remoção de ocorrências de MNT.

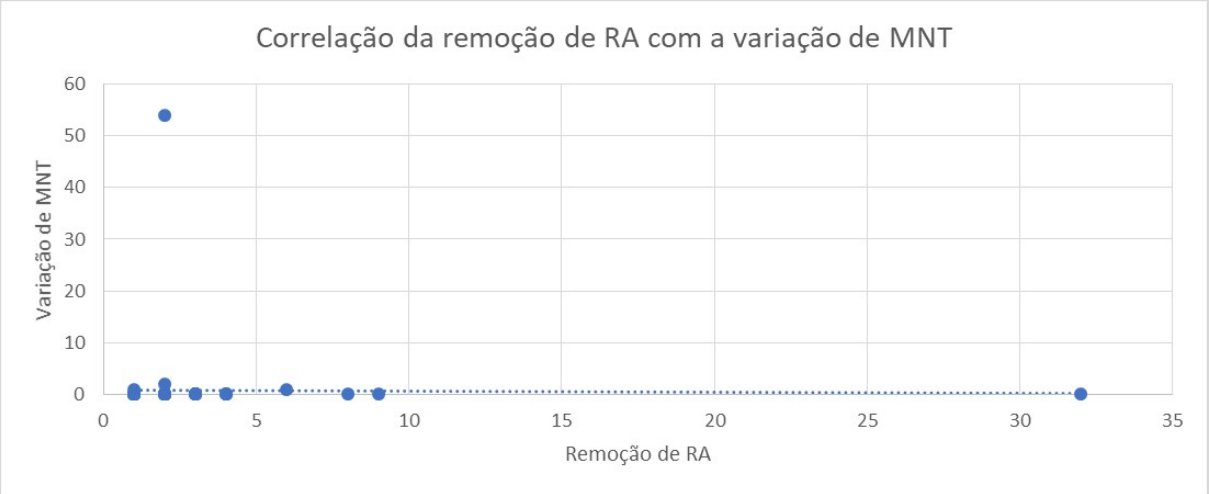

Figura 15 Gráfico de dispersão entre a remoção de RA com a variação de MNT (Fonte: o autor)

A Figura 16 mostra o gráfico de dispersão da correlação entre a quantidade de RA removidos e a variação na quantidade do VT, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa insignificante entre a remoção do AR e a variação da quantidade de ocorrências de VT (*p-value*=,907 e ρ=-,014) e mostra que a remoção de RA causou a criação de ocorrências de VT.

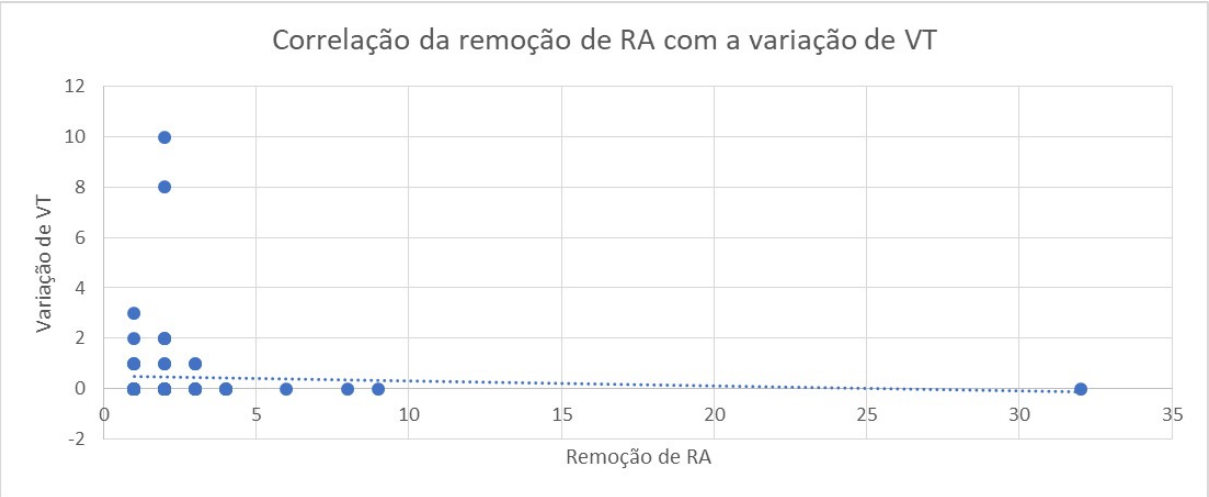

Figura 16 Gráfico de dispersão entre a remoção de RA com a variação de VT (Fonte: o autor)

A Figura 17 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do AR, além da linha de tendência. O

agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do TCD e a variação da quantidade de ocorrências de AR (*p-value*=,057 e ρ=,047) e mostra que a remoção de TCD causou a remoção de ocorrências de AR.

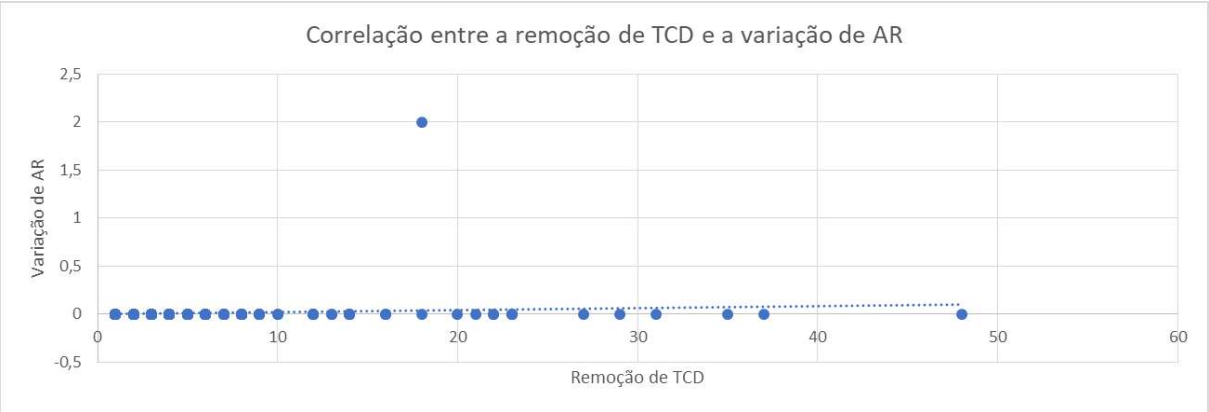

Figura 17 Gráfico de dispersão entre a remoção de TCD com a variação de AR (Fonte: o autor)

A Figura 18 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do CI, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa insignificante entre a remoção do TCD e a variação da quantidade de ocorrências de CI (*p-value*=,476 e ρ=-,018) e mostra que a remoção de TCD causou a criação de ocorrências de CI.

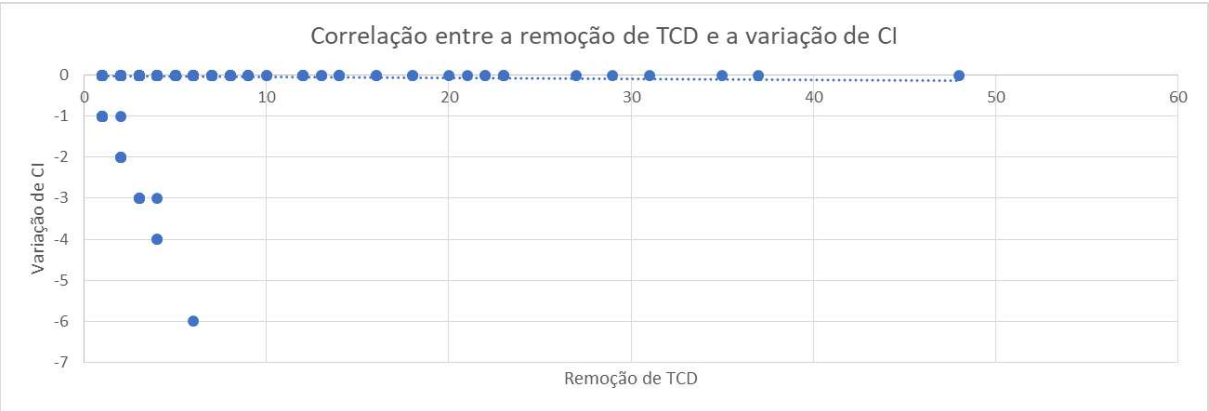

Figura 18 Gráfico de dispersão entre a remoção de TCD com a variação de CI (Fonte: o autor)

A Figura 19 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do DA, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do TCD e a variação da

quantidade de ocorrências de DA (*p-value*=<,001 e ρ=,086) e mostra que a remoção de TCD causou a remoção de ocorrências de DA.

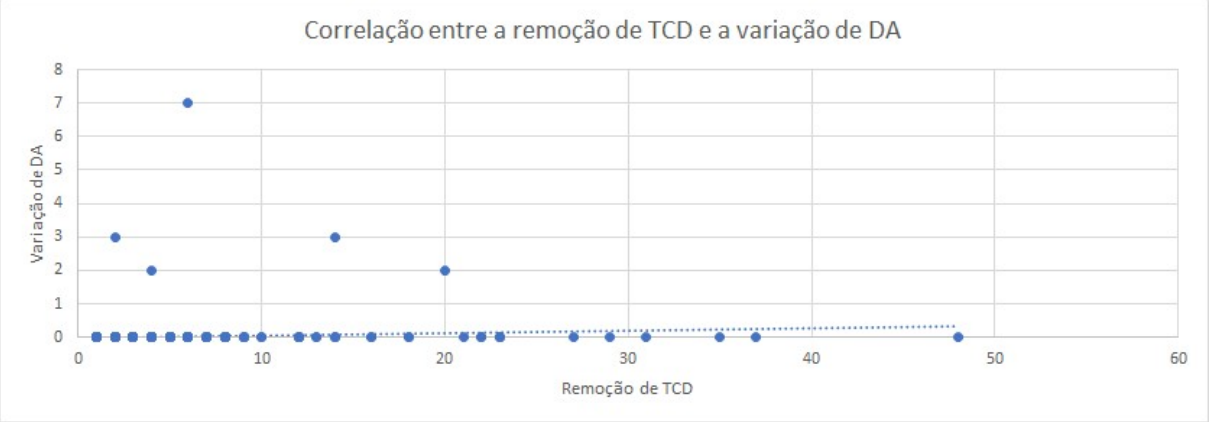

Figura 19 Gráfico de dispersão entre a remoção de TCD com a variação de DA (Fonte: o autor)

A Figura 20 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do EmT e a linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa insignificante entre a remoção do TCD e a variação da quantidade de ocorrências de EmT (*p-value*=<,001 e ρ=-,103) e mostra que a remoção de TCD causou a criação de ocorrências de EmT.

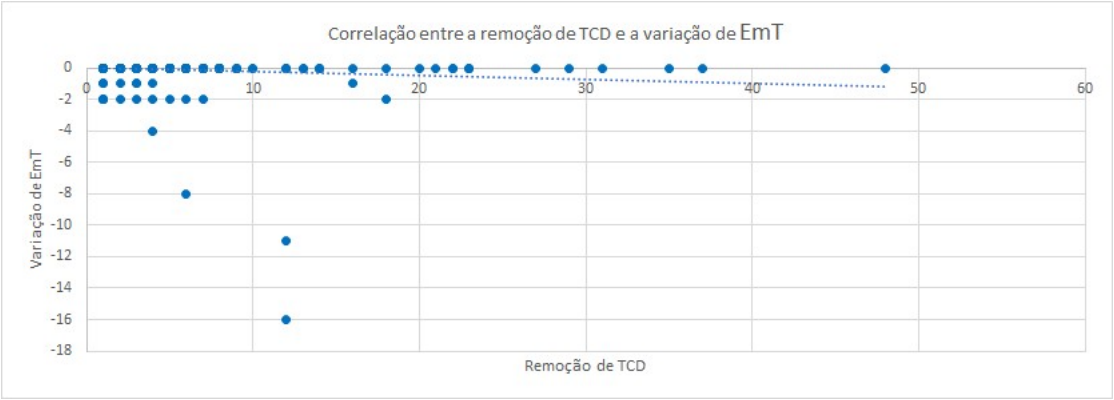

Figura 20 Gráfico de dispersão entre a remoção de TCD com a variação de EmT (Fonte: o autor)

A Figura 21 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do GF e a linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do TCD e a variação da

quantidade de ocorrências de GF (*p-value*=,450 e ρ=,019) e mostra que a remoção de TCD causou a remoção de ocorrências de GF.

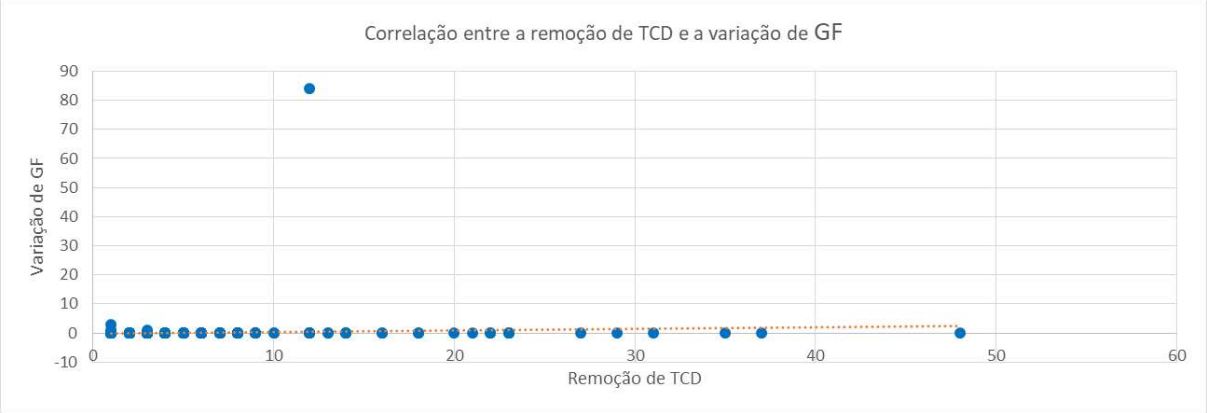

Figura 21 Gráfico de dispersão entre a remoção de TCD com a variação de GF (Fonte: o autor)

A Figura 22 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do LT com a linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa insignificante entre a remoção do TCD e a variação da quantidade de ocorrências de LT (*p-value*=<,001 e ρ=-,104) e mostra que a remoção de TCD causou a criação de ocorrências de LT.

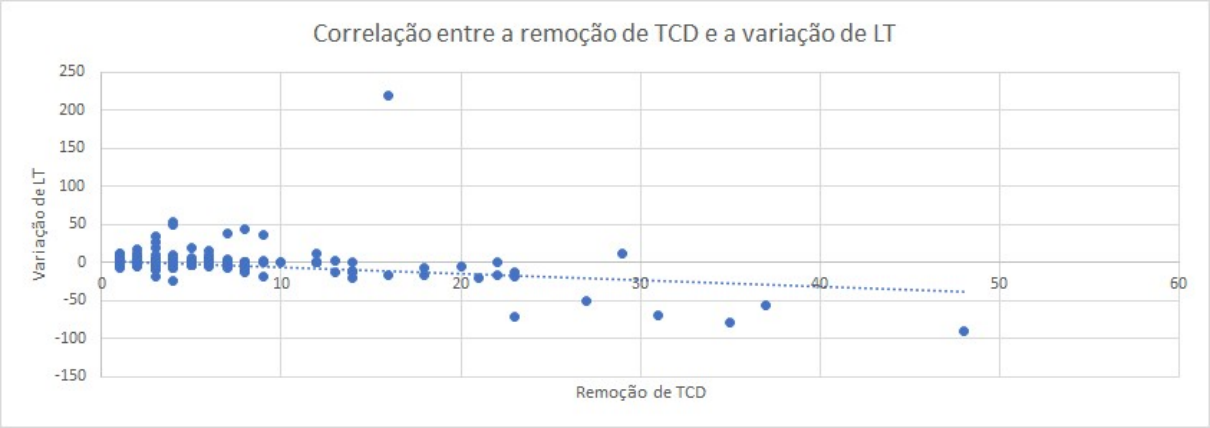

Figura 22 Gráfico de dispersão entre a remoção de TCD com a variação de LT (Fonte: o autor)

A Figura 23 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do MNT com a linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do TCD e a variação da

quantidade de ocorrências de MNT (*p-value*=<,001 e ρ=,089) e mostra que a remoção de TCD causou a remoção de ocorrências de MNT.

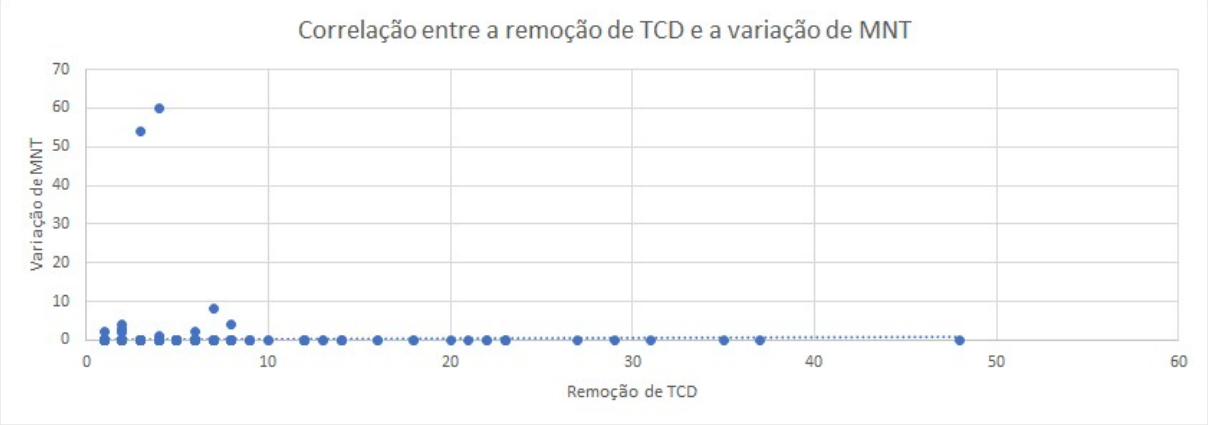

Figura 23 Gráfico de dispersão entre a remoção de TCD com a variação de MNT (Fonte: o autor)

.

A Figura 24 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do PS com a linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação negativa insignificante entre a remoção do TCD e a variação da quantidade de ocorrências de PS (*p-value*=,409 e ρ=-,020) e mostra que a remoção de TCD causou a criação de ocorrências de PS.

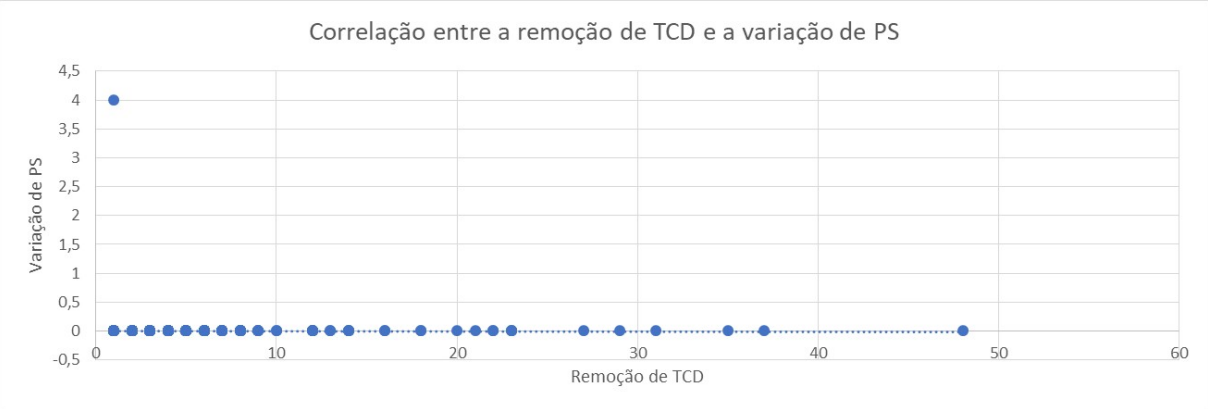

Figura 24 Gráfico de dispersão entre a remoção de TCD com a variação de PS (Fonte: o autor)

A Figura 25 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do SE com a linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do TCD e a variação da

quantidade de ocorrências de SE (*p-value*=,121 e ρ=,038) e mostra que a remoção de TCD causou a remoção de ocorrências de SE.

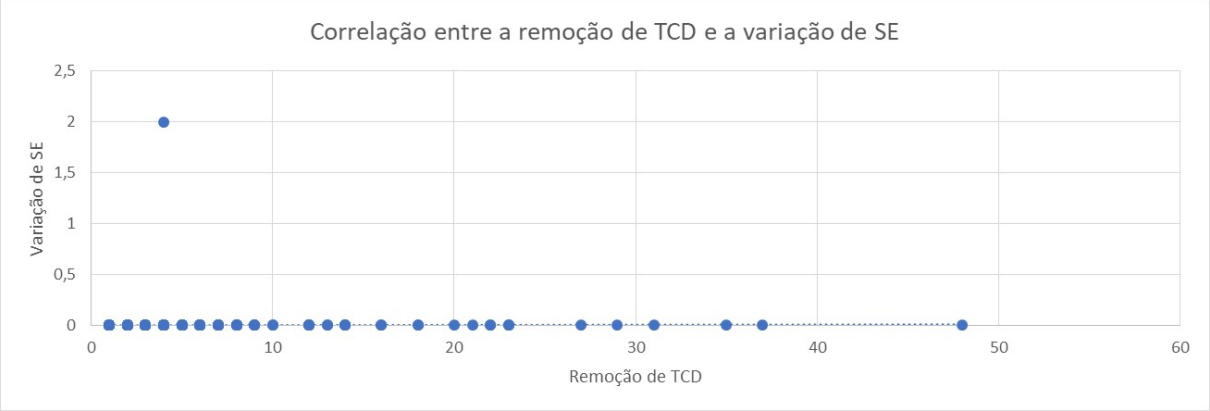

Figura 25 Gráfico de dispersão entre a remoção de TCD com a variação de SE (Fonte: o autor)

A Figura 26 mostra o gráfico de dispersão da correlação entre a quantidade de TCD removidos e a variação na quantidade do VT, além da linha de tendência. O agrupamento dos pontos da correlação e a linha linear de tendência corroboram com a correlação positiva insignificante entre a remoção do TCD e a variação da quantidade de ocorrências de VT (*p-value*=<,001 e ρ=,250) e mostra que a remoção de TCD causou a remoção de ocorrências de VT.

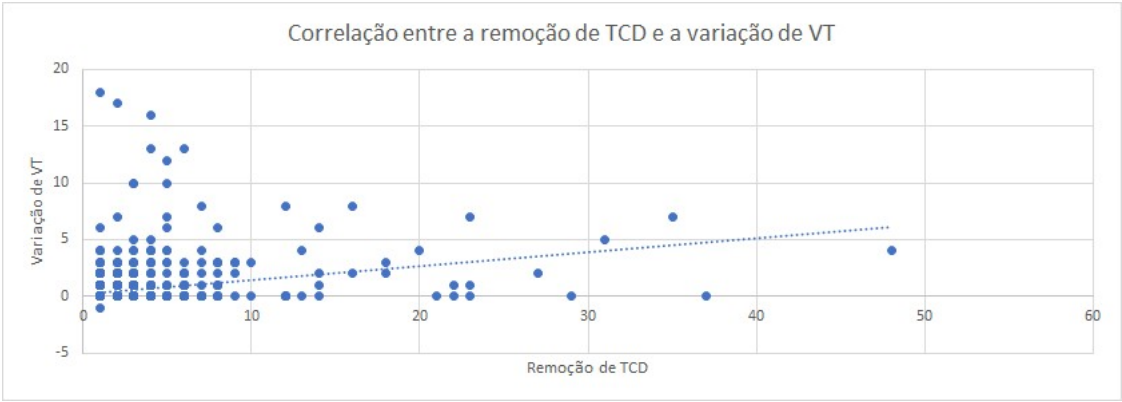

Figura 26 Gráfico de dispersão entre a remoção de TCD com a variação de VT (Fonte: o autor)

APÊNDICE H – PROTOCOLO DE AVALIAÇÃO

1. Agradecer a participação.
2. Iniciar a gravação com o consentimento do avaliador.
3. Auxiliar na execução do refatorador.
4. Investigar aspectos relacionados ao uso do processo e do refatorador no contexto do avaliador.
5. Agradecer a participação e finalizar a gravação.