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

GUILHERME SILVA DE LACERDA

DR-Tools Code Health: Uma Abordagem para Priorização de *Smells* para apoiar a Manutenção e Evolução de Software

Tese apresentada como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação

Orientador: Prof. Dr. Marcelo Soares Pimenta

Porto Alegre
2024

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Lacerda, Guilherme Silva de

DR-Tools Code Health: Uma Abordagem para Priorização de *Smells* para apoiar a Manutenção e Evolução de Software / Guilherme Silva de Lacerda. – Porto Alegre: PPGC da UFRGS, 2024.

282 f.: il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2024. Orientador: Marcelo Soares Pimenta.

1. Code smells. 2. Design smells. 3. Abordagem de priorização. 4. Qualidade de software. 5. Manutenção e evolução de Software. 6. Problemas de design. I. Pimenta, Marcelo Soares. II. Título.

“Diga-me e eu esquecerei,
ensina-me e eu poderei lembrar,
envolva-me e eu aprenderei.”
— Benjamin Franklin

AGRADECIMENTOS

Gostaria de expressar minha mais profunda gratidão às pessoas que tornaram possível a realização deste trabalho, contribuindo de maneira significativa ao longo dessa jornada. Eu já sabia que esta jornada seria difícil, complexa, cheia de desafios. Ao escolher fazer isso, atuando na indústria e na academia, este caminho se tornou ainda mais complexo. Certamente, tudo que foi feito não seria possível sem o apoio destas pessoas. Por isso, é importante registrar minha gratidão.

Dedico este trabalho, primeiramente, a minha amada esposa, Juliana, e minha querida filha, Isabella. Expresso minha eterna gratidão pelo constante apoio, compreensão e incentivo ao longo deste percurso. Sua presença amorosa foi minha fonte de força nos momentos mais desafiadores, e sua compreensão e paciência foram fundamentais para que eu pudesse me dedicar a esta pesquisa.

Também, dedico este trabalho aos meus pais, Liberta Lacerda e Paulo Lacerda (*in memoriam*), cujo amor, apoio e exemplo de dedicação foram fontes inesgotáveis de inspiração para mim. Suas lições de vida e valores moldaram o meu caráter e me proporcionaram a base sólida necessária para perseguir meus objetivos acadêmicos.

Ao meu estimado orientador, Marcelo Pimenta, e ao co-orientador *de facto*, Fabio Petrillo, sou imensamente grato pela orientação acadêmica excepcional, pelo apoio contínuo e pela inspiração intelectual que proporcionaram durante todo o desenvolvimento deste trabalho. Suas orientações sábias e *insights* valiosos foram cruciais para o sucesso deste projeto. Além disso, são grandes amigos que carrego na vida e no coração.

Expresso também minha sincera gratidão aos sócios da Wildtech, em que sou privilegiado por ser parte, pelo apoio incondicional e compreensão em conciliar minhas responsabilidades acadêmicas com as demandas do mundo corporativo.

Aos meus colegas da empresa Pix Force, onde atuo como Head de Engenharia, agradeço pela colaboração, camaradagem e apoio mútuo ao longo deste ano. Sua dedicação e profissionalismo foram fundamentais para o meu crescimento profissional e pessoal.

Não poderia deixar de mencionar meus alunos, ex-alunos e colegas de todas as universidades onde tive a honra de atuar, nestes quase 20 anos de magistério em nível superior. Suas contribuições, debates estimulantes e parcerias enriqueceram imensamente minha jornada acadêmica, e sou profundamente grato por sua colaboração e amizade.

Por fim, agradeço a todas as outras pessoas que, de uma forma ou de outra, contribuíram para este trabalho, direta ou indiretamente. Suas contribuições foram inestimáveis

e não passaram despercebidas.

Este trabalho é dedicado a todos vocês, cujo apoio, encorajamento e influência moldaram não apenas esta tese, mas também minha jornada como estudante, profissional e ser humano.

Muito obrigado!!

RESUMO

Durante a manutenção e evolução do software, a dívida técnica é uma questão recorrente, já que modificações no software tendem a adicionar problemas aparentes de *design*, como falta de testes, problemas arquiteturais e um grande número de *code smells*, levando à degradação do código. *Smells*, englobando problemas de software a nível de código e falhas de princípios de *design*, impactam negativamente a qualidade e manutenção do software. Para monitorar e acompanhar esses *smells*, a priorização é uma estratégia eficaz, permitindo que desenvolvedores identifiquem e mitiguem problemas críticos de forma tática. Apesar de ser essencial na gestão da dívida técnica, a priorização de *smells* ainda carece de um conjunto sólido, validado e amplamente usado de modelos e ferramentas, permanecendo um campo aberto e relevante de pesquisa.

O objetivo desta tese é investigar a importância da priorização de *smells* para o desenvolvimento e manutenção de software e propor o *DR-Tools Code Health*, uma nova abordagem para priorização de *smells*. Esta abordagem engloba a definição e detecção de mais de *smell* em um mesmo elemento de código, além da classificação e filtragem destes *smells*, permitindo que os desenvolvedores identifiquem no código as partes mais problemáticas e que necessitam de atenção, e as tornem candidatas prioritárias para refatorações, manutenção e evolução de software. O método proposto é inspirado no *Método de Hanlon*, amplamente usado para priorização de problemas de saúde em países em desenvolvimento.

Dois experimentos foram realizados: o primeiro avalia a percepção dos profissionais da indústria, revelando eficácia na identificação de problemas de código com alta concordância nas análises de métodos e classes, embora muitos desenvolvedores não tenham percebido mudanças, sugerindo a necessidade de melhorias na comunicação dos resultados. No segundo experimento, foram investigados 5 projetos *open-source* quanto à priorização e impacto dos *smells* em atributos de qualidade. Observou-se um padrão de aumento gradual ou constante dos *smells*, alinhado a estudos anteriores. O conceito de *smell churn rate* foi destacado, com o projeto *JetUML* mostrando redução significativa de *smells* entre versões. A estratégia de priorização baseou-se em melhorias oportunistas, com modularidade e manutenibilidade sendo os atributos de qualidade mais afetados pelos *smells*.

As principais contribuições deste trabalho incluem a i) abordagem multicritério para priorização de *smells*, ii) um método para contabilizar adições/remoções dos *smells* entre

duas versões e iii) duas ferramentas *open-source*, uma que dá suporte ao uso do método proposto e outra que implementa o método de inserções/remoções dos *smells*.

Palavras-chave: Code smells. design smells. abordagem de priorização. qualidade de software. manutenção e evolução de Software. problemas de design.

DR-Tools Code Health: A Smells Prioritization Approach to support Software Maintenance and Evolution

ABSTRACT

During software maintenance and evolution, technical debt is a recurring issue, as software modifications tend to introduce apparent design problems, such as a lack of tests, architectural issues, and a large number of code smells, leading to code degradation. Smells, encompassing software problems at the code level and violations of design principles, negatively impact the quality and maintainability of the software. To monitor and track these smells, prioritization is an effective strategy, allowing developers to tactically identify and mitigate critical issues. Despite being essential for managing technical debt, the prioritization of smells still lacks a solid, validated, and widely used set of models and tools, remaining an open and relevant field of research.

The objective of this thesis is to investigate the importance of smell prioritization for software development and maintenance and to propose DR-Tools Code Health, a new approach for smell prioritization. This approach encompasses the definition and detection of multiple smells in the same code element, as well as the classification and filtering of these smells, enabling developers to identify the most problematic parts of the code that require attention, making them priority candidates for refactoring, maintenance, and software evolution. The proposed method is inspired by Hanlon's Method, widely used for prioritizing health issues in developing countries.

Two experiments were conducted: the first evaluates the perception of industry professionals, revealing effectiveness in identifying code problems with high agreement in the analysis of methods and classes, although many developers did not perceive changes, suggesting the need for improvements in communicating the results. In the second experiment, 5 open-source projects were investigated regarding the prioritization and impact of smells on quality attributes. A pattern of gradual or consistent increase in smells was observed, aligned with previous studies. The concept of smell churn rate was highlighted, with the JetUML project showing a significant reduction in smells between versions. The prioritization strategy was based on opportunistic improvements, with modularity and maintainability being the quality attributes most affected by smells.

The main contributions of this work include i) a multicriteria approach for smell prioritization, ii) a method for accounting for additions/removals of smells between two versions,

and iii) two open-source tools, one that supports the proposed method and another that implements the method for tracking smell insertions/removals.

Keywords: Code smells. design smells. prioritization approach. software quality. software maintenance and evolution. design issues.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ASAT	*Automatic Static Analysis Tool*
AST	*Abstract Syntax Tree*
BBN	*Bayesian Belief Network*
BPR	*Basic Priority Rating*
CDI	*Code Disease Indicator*
CE	*Code Element*
CK	*Chidamber-Kemerer*
CLI	*Command-line interface*
CNN	*Convolution Neural Networks*
CSV	*Comma-separated Values*
DECOR	*DEtection & CORrection*
DETEX	*DETection EXpert*
DP	Desvio Padrão
DSL	*Domain Specific Language*
DT	Dívida Técnica
EBM	*Evidence-based Medicine*
EBSE	*Evidence-based Software Engineering*
ES	Engenharia de Software
EUA	Estados Unidos da América
FLOSS	*Free-Libre Open Source Software*
GDT	Gestão da Dívida Técnica
GUI	*Graphical User Interface*
HIST	*Historical Information for Smell deTection*
IDE	*Integrated Development Environment*
LDA	*Latent Dirichlet Allocation*
LOC	*Lines of Code*
LSI	*Latent Semantic Indexing*

OO	Orientação a Objetos
MH	Método de Hanlon
ML	*Machine Learning*
MVP	*Minimum Valuable Product*
NACCHO	*National Association of County and City Health Officials*
PAHO	*Pan American Health Organization*
PEM	*Paediatric Emergency Medicine*
QDIR	*Quality Depreciation Index Rule*
QMOOD	*Quality Model for Object Oriented Design*
REST	*Representational State Transfer*
RNN	*Recurrent Neural Networks*
RQ	*Research Question*
RTM	*Relational Topic Models*
RCBHI	*Rwanda Community-Based Health Insurance*
SCI-D	*Spinal Cord Injury or Disease*
SLR	*Systematic Literature Review*
SMO	*Sequential Minimal Optimization*
SO	*Sistema Operacional*
SOAP	*Simple Object Access Protocol*
SVM	*Support Vector Machine*
TACO	*Textual Analysis for Code Smell Detection*
TCC	*Trabalho de Conclusão de Curso*
UHC	*Universal Health Coverage*
UI	*User Interface*
UML	*Unified Modeling Language*
UX	*User eXperience*
VS	Visualização de Software
WHO	*World Health Organization*
WMC	*Weighted Method per Class*

SUMÁRIO

1 INTRODUÇÃO

A Engenharia de Software (ES) é uma subárea da Ciência da Computação que busca definir princípios, métodos, técnicas, metodologias, processos, práticas e ferramentas para a construção de produtos de software, visando sua constante evolução, seguindo padrões de qualidade (IEEE, 2014; PRESSMAN, 2014; SOMMERVILLE, 2016). Embora a ES adote princípios científicos de outras áreas da Engenharia, como *e.g.,* a Engenharia Civil e Mecânica, é fundamental entender que, na sua essência, a forma como o trabalho é realizado necessita ser diferente. O trabalho produzido pela ES é diferenciado: o software.

Diferentemente dos produtos das outras engenharias, o software é um produto virtual, abstrato, complexo e ausente de propriedades físicas (BROOKS, 1987). Portanto, sua construção, evolução e compreensão é uma tarefa desafiadora. Por este motivo, o desenvolvimento de software é um exercício de descobertas, no qual o processo de desenvolvimento deve prover o aprendizado junto ao cliente, produzindo software que agregue valor (POPPENDIECK; POPPENDIECK, 2003; POPPENDIECK; POPPENDIECK, 2006). Esse aprendizado faz com que novas necessidades surjam ao longo do tempo, a partir de novas hipóteses e descobertas. Com isso, é fundamental que equipes de desenvolvimento de software tenham sempre um código saudável, passível de ser modificado, tanto para correções como evoluções. Porém, a indústria de software vem se mostrando carente de ferramentas eficazes e práticas para estruturar o processo da manutenção de software (PARNIN; GORG; NNADI, 2008).

Segundos os estudos (LOWE; PANAS, 2005; TELEA; VOINEA, 2011; WIMALASOORIYA, 2019), 50% a 80% dos custos de um software estão relacionados com atividades de manutenção: remover problemas de projeto e implementação, acrescentar ou modificar funcionalidades e adaptar o software para diferentes ambientes (hardware, SO) são algumas delas. Manutenção de software é difícil por causa da ausência de documentação útil, no qual o código-fonte grande e complexo se torna a principal e muitas vezes a única fonte confiável de informações sobre o sistema (LOWE; PANAS, 2005). Adicionalmente, os desenvolvedores de software gastam cerca de 60% do seu tempo na compreensão de software (ZITZLER et al., 2003). Em um relatório publicado pela Stripe (STRIPE, 2018), cerca de 300 bilhões de dólares são perdidos anualmente, pelo impacto de um código mal escrito implicando em problemas de manutenção e evolução.

A questão é que o código não se degrada com o tempo, mas com as modifica-

ções realizadas ao longo do tempo. Esta degradação, até certo ponto natural, é conhecida por *dívida técnica*[1] (DT) (STERLING, 2011). Este termo foi cunhado por Cunningham (2014), que o define como uma ação comportamental, seja ou não proposital, em que se assume abrir mão de um *design* mais elaborado em prol de uma entrega. Segundo Charalampidou (2019), é um conceito emprestado do domínio financeiro para expressar custos de manutenção extras causados por soluções de curto prazo que comprometem a qualidade para atender demandas comerciais urgentes. Neste caso, esta dívida deve ser paga o quanto antes, pois o seu atraso acarreta em juros que prejudicam a manutenção (SJOBERG et al., 2013; YAMASHITA; MOONEN, 2013b) e evolução do software (ABBES et al., 2011; HALL et al., 2014). Em um relatório recente (SECURITY MAGAZINE, 2022), publicado pela Synopsys com a *Consortium for Information and Software Quality* (CISQ), a DT representa o maior custo de retrabalho no desenvolvimento de software, acumulado em 1,5 trilhão de dólares.

Uma das principais causas do aumento da DT estão relacionadas a violações de boas práticas de desenvolvimento (KRUCHTEN; NORD; OZKAYA, 2012; LUJAN et al., 2020). Estas violações ficaram conhecidas na literatura como armadilhas (WEBSTER, 1995), *anti-patterns* (BROWN et al., 1998) e, o mais conhecido de todos, os *smells* (FOWLER et al., 1999). Eles também contribuem negativamente para a compreensão e potencialmente levam à introdução de falhas (KHOMH; PENTA; GUEHENEUC, 2009; KHOMH et al., 2012). Existem muitos contextos onde os *smells* podem ser encontrados, como modelos (MISBHAUDDIN; ALSHAYEB, 2015), testes (GAROUSI; MANTYLA, 2016), requisitos (ALVES et al., 2016), arquitetura (LI; AVGERIOU; LIANG, 2015; BESKER; MARTINI; BOSCH, 2018; ANICHE et al., 2018; MUMTAZ; SINGH; BLINCOE, 2020) e serviços (SABIR et al., 2018; MUMTAZ; SINGH; BLINCOE, 2020). Porém, o principal foco de estudo na literatura são os *smells* encontrados no código-fonte (FOWLER; BECK, 2019; KAUR, 2020), também conhecidos como *code smells*[2].

Em geral, os desenvolvedores introduzem os *smells* em sistemas de software quando modificações e aprimoramentos são realizados para atender a novos requisitos (SILVA; TSANTALIS; VALENTE, 2016). Os principais motivos que levam a estas introduções estão relacionadas com falta de conhecimento técnico e/ou negócios, mudanças frequentes nos requisitos, restrições de tecnologia, pressão, priorização de funcionalidades ao invés da qualidade, problemas no processo, entre outros (TUFANO et al., 2015; SHARMA, 2019). O código se torna complexo e o *design* original é quebrado, diminuindo a qua-

[1] Do inglês, *technical debt*

[2] neste trabalho, é usado o termo *smell* como sinônimo de *code e design smell*

lidade do software. Os *smells* formam excelentes preditores de propensão a mudanças e estão diretamente relacionados a dívida técnica, não sendo influenciados pelo domínio da aplicação (TSANTALIS; CHAIKALIS; CHATZIGEORGIOU, 2018).

A comunidade tem estudado os *smells* sobre diferentes perspectivas (PALOMBA et al., 2018b). Alguns autores desenvolveram estudos e ferramentas usando diferentes abordagens de detecção como métricas, regras, histórico de mudanças, análise textual e probabilística. Outros, estudaram a relevância, quando e porquê são introduzidos, como evoluem, o impacto na qualidade e priorização. Um aprofundamento destes estudos é apresentado nos Capítulos 2 e 3 deste trabalho.

1.1 Motivação

Embora já se discuta este tema há mais de duas décadas, em um estudo sistemático desenvolvido por Lacerda et al. (2020) levantou-se uma série de temas e desafios relacionado a *smells* que ainda precisam ser explorados. Entre os temas, destacam-se problemas de nomenclatura e definição dos *smells*, falta de abordagens mais efetivas de detecção, a falta de envolvimento dos desenvolvedores no processo de detecção dos *smells*, percepção da indústria em relação a nocividade dos *smells*, como os *smells* tem impactado diferentes aspectos de qualidade, formas de priorização dos *smells*, entre outros.

De fato, segundo Sehgal, Mehrotra and Bala (2018), apenas detectar a presença dos *smells* não é um indicativo de criticidade. É necessário, segundo os tomadores de decisão, avaliar quais impactos na qualidade esses diferentes problemas apresentam. Também, cada problema possui diferentes pontuações de severidade (SEHGAL; MEHROTRA; BALA, 2018). Sem saber sua real severidade, os desenvolvedores sentem-se relutantes em detectá-los e removê-los (KAUR, 2020). Assim, a priorização se torna cada vez mais relevante para apoiar os desenvolvedores neste processo. Segundo Lenarduzzi et al. (2020), a priorização é a atividade mais importante de gestão da dívida técnica, sendo uma área de pesquisa em aberto (PINA; GOLDMAN; TONIN, 2021). Conforme os autores, é evidentemente claro que a priorização pode ser realizada usando diferentes abordagens, com diferentes objetivos e critérios. No entanto, ainda falta um conjunto sólido, validado e amplamente usado de ferramentas para priorização (PINA; GOLDMAN; TONIN, 2021).

Especificamente, a priorização dos *smells* é um campo aberto de pesquisa (PECORELLI et al., 2020), que permite fazer alguns questionamentos: a) ao detectar um

smell, o quanto ele é mais problemático[3] do que seus pares? b) qual sua implicação nos atributos de qualidade e, mais especificamente, na manutenibilidade? c) quais elementos de código e qual granularidade[4] são as mais prioritárias? d) o acúmulo dos problemas em diferentes elementos de código[5] podem ajudar na priorização? e) como envolver o time no processo de detecção e priorização, levando em consideração os objetivos de qualidade e de negócios?

Estes *smells* precisam ser priorizados, filtrados e sintetizados, permitindo que os desenvolvedores possam avaliar sobre ações imediatas ou planejadas ao longo do tempo.

1.2 Objetivos e Contribuições Esperadas

Este trabalho tem por objetivo investigar a importância da priorização de *smells*. Assim, é proposto uma abordagem de priorização de *smells*, permitindo a definição, detecção, classificação e filtragem destes *smells*. Esta abordagem foi inspirada no*Método de Hanlon* (HANLON; PICKETT, 1984), amplamente usado para priorização de problemas de saúde em países em desenvolvimento, sendo também a referência da PAHO/WHO (CHOI et al., 2019). Até onde se sabe, esta abordagem de priorização ainda não foi usada na ES. O método proposto visa avaliar a importância da priorização durante o processo de desenvolvimento e manutenção de software.

As principais contribuições esperadas deste trabalho incluem a i) abordagem multicritério para priorização de *smells*, ii) um método para contabilizar o *churn*[6] dos *smells* e iii) duas ferramentas *open-source*, uma que dá suporte ao uso do método proposto e outra que implementa o *churn* dos *smells*.

1.3 Questões de Pesquisa

A tese visa responder a seguinte questão geral:

Como a priorização de smells *pode melhorar a identificação dos elementos de código*

[3] ao longo do trabalho, será usado o termo severidade para representar o grau de gravidade de um *smell*

[4] granularidade se refere ao nível em que o *smell* é encontrado, podendo ser no nível de pacote, classe ou método

[5] um projeto de software possui estruturas autocontidas - método(s) estão dentro de classe(s). As classe(s) estão dentro de pacote(s). O(s) pacote(s) estão dentro de um projeto

[6] Método desenvolvido para contabilizar inserções/remoções dos *smells* entre duas versões analisadas. Este método foi usado no experimento 2 desta tese. Mais informações podem ser obtidas no Apêndice N.

mais problemáticos de um software?

Entende-se por elementos de código como os principais componentes presentes em um projeto de software, que são autocontidos, como pacotes, classes e métodos. Já o termo problemático refere-se aos elementos de código identificados com diferentes *smells*, afetando vários atributos de qualidade, que impactam na manutenibilidade.

Assim, se os *smells* afetam a qualidade e, na maioria dos casos, o propósito da reestruturação e refatorações tem como intenção e motivação melhorar a qualidade, então associar os problemas com a qualidade e usá-los como um dos critérios de priorização expõem as partes mais problemáticas para os desenvolvedores.

Mais especificamente, com base no problema, definiu-se as seguintes questões de pesquisa (RQs):

RQ1 - A abordagem de priorização proposta ajuda os desenvolvedores a identificar os elementos de código (classes, métodos) mais problemáticos?

RQ2 - Ao confrontar os resultados analisados pela abordagem de priorização proposta, muda a percepção dos desenvolvedores sobre os elementos de código mais problemáticos?

RQ3 - Os desenvolvedores fazem algum tipo de priorização ao remover os *smells* durante a manutenção e evolução de software?

RQ4 - Como os *smells* têm impactado a qualidade durante a evolução de software?

Para avaliar as RQs, foram definidos 2 experimentos para cada par de questões. O primeiro experimento é um estudo misto sobre a percepção dos desenvolvedores da indústria ao considerar a priorização (RQ1 e RQ2). O segundo experimento avalia, de forma longitudinal e exploratória, a ação de priorização dos *smells* e seu respectivo impacto em atributos de qualidade na evolução de software (RQ3 e RQ4). Ambos são detalhados no Capítulo 5.

1.4 Estrutura da Pesquisa

Esta pesquisa foi estruturada em 3 fases: fundamentação, proposta e tese. A Figura 1.1 resume cada uma das fases e etapas.

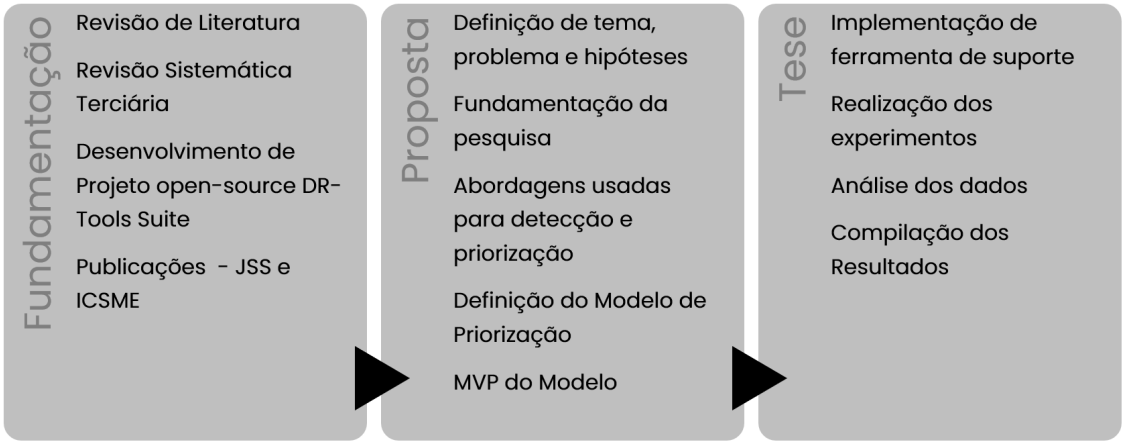
Na primeira fase, foi realizado um estudo amplo sobre a literatura sobre qualidade de software, mais especificamente sobre *smells* e refatorações. A partir deste estudo, foi desenvolvido uma revisão terciária sobre o tema. Também, foi criado um projeto *open-*

source, denominado *DR-Tools Suite*[7], contemplando um conjunto de ferramentas simples para apoiar a análise de código. O resultado de ambos os trabalhos foram publicados no *JSS* e *ICSME 2020*. Mais detalhes sobre publicações e apresentações em eventos da indústria realizadas estão na Seção 6.3.

Na segunda fase, foi definido o tema, tendo como problemática a priorização de *smells*. Assim, foi desenhada a abordagem de priorização de *smells*, contemplando o modelo de detecção e priorização, bem como um método para a estruturação de um MVP do modelo, com o objetivo de validação da ideia e realização de testes iniciais.

Já na terceira fase, descrita nesta tese, são apresentadas a implementação de uma ferramenta que apoie a abordagem, realização de experimentos com profissionais da indústria (Seção 5.1), projetos *open-source* durante a evolução de software (Seção 5.2), além de depoimentos de uso (Seção 5.3), apresentação de resultados e discussão.

Figura 1.1 – Estrutura da Pesquisa



1.5 Organização do Texto

Esta tese está organizada da seguinte forma: o Capítulo 2 descreve os principais fundamentos sobre qualidade e sua relação com *smells*, priorização de problemas no código e *smells*. O Capítulo 3 traz as principais abordagens de detecção, priorização e filtragem, bem como ferramentas descritas nos estudos apresentados recentemente. o *DR-Tools Code Health* é apresentado no Capítulo 4, descrevendo detalhes do modelo e o método agregado de priorização proposto, além da ferramenta de suporte. No Capítulo 5, são apresentados as etapas, experimentos realizados, os principais resultados obtidos e discussões. Finalmente, o Capítulo 6 apresenta as considerações finais, contribuições e

[7]https://drtools.site

publicações realizadas, além de trabalhos futuros que podem ser derivados desta tese.

2 FUNDAMENTAÇÃO

Este capítulo apresenta os principais tópicos relacionados ao tema desta tese, como qualidade e sua relação com *smells* e priorização de problemas de *design*. Grande parte dos fundamentos, abordagens de detecção, priorização e ferramentas foram previamente apresentados em uma revisão sistemática terciária da literatura (LACERDA et al., 2020), de nossa autoria.

2.1 Smells: Definições e Taxonomias

É inegável que o conceito de *smells* foi adotado, primeiramente, pela comunidade ágil de desenvolvimento de software como uma maneira de apontar algo errado ou um ponto de melhoria (BECK; ANDRES, 2004; ELSSAMADISY, 2007). Atualmente, a indústria também adotou este termo para representar anomalias em elementos de software.

O uso do termo *smell* tornou-se popular principalmente devido ao trabalho de Fowler et al. (1999), que o usou para identificar padrões de código que contêm problemas estruturais e, portanto, deveriam ser melhorados. Fowler et al. (1999) foram pioneiros sobretudo em identificar e discutir *smells* de código e prover um guia prático de técnicas para a resolução dos mesmos. Porém, estes problemas de *design* aparecem na literatura com diferentes nomes como *pitfalls* (WEBSTER, 1995), *anti-patterns* (BROWN et al., 1998), *anomalies* (MACIA et al., 2012), *design flaws* (MARINESCU, 2004) e *smells* (FOWLER et al., 1999). Também foram adotadas derivações representando a granularidade, como *code smells*, *design smells* e *architectural smells* (ALKHARABSHEH et al., 2019).

Brown et al. (1998) apresenta o termo *anti-pattern*, que descreve ocorrências comuns para um problema que gera consequências negativas. O *anti-pattern* pode ser o resultado de uma ação de um gerente ou desenvolvedor que não conhece a solução, não tem conhecimento suficiente ou experiência para resolver um tipo de problema particular, ou ter aplicado uma solução em um contexto errado. Nesta obra, são descritos 40 *anti-patterns*, categorizados em desenvolvimento, arquitetura e gestão de projetos. Neste trabalho, um dos objetivos é explorar a definição e identificação destes *anti-patterns* de desenvolvimento, aqui chamados de *design smells* (Tabela 2.1).

Embora não seja o primeiro, certamente o catálogo mais conhecido de *smells* é o proposto por Fowler et al. (1999), sendo o mais citado na literatura (LACERDA et al., 2020). Nesta obra, são descritos 22 *smells* e para cada um deles são associados uma

Tabela 2.1 – Lista de *Design Smells* apresentados por Brown *et al.*

Smell	**Descrição**
Blob	Também conhecido como *God Class*, estilo de *design* procedural que leva um objeto a ter muitas responsabilidades (classes *Controller*) e atributos com baixa coesão, enquanto outros só armazenam dados ou executam processos simples
Lava Flow	Código morto e informações esquecidas de projeto são congeladas em um *design* em constante mudança
Functional Decomposition	Escrita de código procedural em uma tecnologia que implementa o paradigma da OO (geralmente uma função principal que chama muitas outras). Isso acontece por causa da experiência anterior dos desenvolvedores em linguagem procedural e pouca experiência em OO. Alguns indícios deste *anti-pattern* são uma *Main Class*, que usa muito pouco de herança e polimorfismo, associada a pequenas classes, com muitos atributos privados e poucos métodos
Poltergeist	Classes que têm um papel e ciclo de vida muito limitados, frequentemente iniciando processos para outros objetos
Spaghetti Code	Caracterizado pelo pensamento procedural usando linguagens OO. Geralmente, é revelado pelo uso por classes sem estruturas, métodos longos sem parâmetros, uso de variáveis globais, além de não explorar e impedir a aplicação de princípios OO, como herança e polimorfismo
Cut and Paste Programming	Código reusado pela cópia de trechos de código, gerando problemas de manutenção
Swiss Army Knife	Caracterizada por uma classe complexa que oferece um grande número de serviços. Enquanto o *Blob* é um *Controller*, monopolizando o processamento e chamadas de um sistema, o *Swiss Army Knife* expõe a alta complexidade para atender as necessidades previsíveis de uma parte do sistema (geralmente, classes utilitárias com muitas responsabilidades)

Fonte: (BROWN et al., 1998)

série de refatorações que podem ser aplicados para mitigá-los. Este trabalho deu uma importante contribuição ao organizar e catalogar uma lista de *smells*, apresentados na Tabela 2.2.

Recentemente, a lista foi atualizada por Fowler and Beck (2019), porém mantendo a grande maioria dos *smells*. As mudanças estão relacionadas a inclusão dos *smells Mysterious Name, Global Data Mutable Data, Insider Trading* e *Loops* e a troca de nomes de *Long Method* para *Long Function*, *Switch Statements* para *Repeated Switches* e *Lazy Class* para *Lazy Element*. Outros trabalhos estendem esta lista original de Fowler, como (WAKE, 2003) (Tabela 2.3) e (KERIEVSKY, 2004) (Tabela 2.4).

Já Martin (2008) busca identificar possíveis problemas em código sob a ótica de heurísticas de limpeza, ou seja, o que é necessário em termos de regras de estilo, boas práticas e disciplina para manter um código limpo. Os *smells* descritos não estão relacionados apenas à estrutura de código, mas também a uso de comentários, ambiente de construção, tratamento de erros, formatação, nomenclatura de elementos e testes. Zhang et al. (2009) procuraram definir uma melhor forma para representar alguns *smells* como *Data Clumps, Middle Man, Message Chains, Speculative Generality, Switch Statements*, usando a mesma abordagem de catálogo de *patterns*, comparando com a descrição proposta por (FOWLER et al., 1999).

Uma forma interessante de entender os *smells* é através de uma categorização, baseado nas relações entre eles, gerando um maior entendimento (MANTYLA; VANHANEN; LASSENIUS, 2004). Wake (2003) propôs uma categorização dos *smells* apresentados em (FOWLER et al., 1999), conforme Tabela 2.5. Já Mantyla, Vanhanen and Lassenius (2003) propuseram uma taxonomia de *smells*, baseada nestas relações. As categorias são descritas na Tabela 2.6.

Em um trabalho mais recente, Sharma and Spinellis (2018) também propõem uma organização de software *smells*, ampliando as categorias e, consequentemente, a lista de *smells*. Atualmente, constam cerca de 279 *smells* em categorias como *code* (*architecture*, *design*, *energy*, *implementation*, *performance* e *tests*), *configuration*, *database*, entre outros. O catálogo está disponível *online*[1], contendo informações básicas como nome, descrição e referência do *smell*.

[1] http://www.tusharma.in/smells/, acessado em 14/05/2024

Tabela 2.2 – Lista de *Code Smells* apresentados em Fowler *et al.*

Smell	**Descrição**
Duplicated Code	consiste em trechos iguais ou muito similares em diferentes pontos de uma mesma base de código
Long Method	método muito grande e portanto difícil de entender, estender e modificar. É muito provável que este método tenha responsabilidades demais, ferindo um dos princípios de um bom *design* OO (*SRP: Single Responsibility Principle* (MARTIN; MARTIN, 2007))
Large Class	classe que tem muitas responsabilidades e, por consequência, contém muitas variáveis e métodos. O mesmo princípio de *SRP* também se aplica neste caso
Long Parameter List	lista de parâmetros muito extensa, o que a torna de difícil compreensão e geralmente é um indício de que o método tem responsabilidades demais. Este *smell* tem uma forte relação com *long method*
Divergent Change	identificado quando uma única classe precisa ser alterada seguidamente pelas mais diversas razões. Isso é um claro indicativo de que ela não é suficientemente coesa e deve ser dividida. Este *smell* fere outro princípio de *design* OO (*OCP - Open Close Principle (MARTIN; MARTIN, 2007))*
Shotgun Surgery	contrário ao *divergent change*, pois acontece quando, para executar uma modificação, várias classes diferentes precisam ser alteradas
Feature Envy	quando um método se interessa mais por membros de outras classes do que a sua própria, sendo um claro sinal de que o mesmo está na classe errada
Data Clumps	estruturas de dados que sempre aparecem em conjunto e, quando um dos itens não está presente, todo o conjunto perde o sentido. Um exemplo disso são classes que contêm coordenadas, mas ao invés de possuírem um membro "coordenadas", possuem dois inteiros para representar as coordenadas X e Y
Primitive Obsession	representa a situação onde tipos primitivos são utilizados no lugar de classes leves, como por exemplo o uso da classe *String* para o armazenamento de um telefone, no lugar de uma classe que represente especificamente os dados de um telefone
Switch Statements	não são necessariamente *smells* por definição, mas quando são muitos utilizados, geralmente são um sinal de problemas, especialmente quando usados para identificar o comportamento de um objeto baseado em seu tipo. Geralmente nessas situações o uso de polimorfismo é o melhor caminho para a sua eliminação, além do que quase sempre tem-se *duplicated code*
Parallel Inheritance Hierarchies	existência de duas hierarquias de classes que são totalmente conectadas, ou seja, ao adicionar uma subclasse em uma das hierarquias, requer-se que uma subclasse semelhante seja criada na outra
Lazy Class	classe que não tem responsabilidades suficientes e portanto não deveria existir
Speculative Generality	trechos de código são criados para suportar comportamentos futuros do software, que não são ainda necessários
Temporary Field	membro utilizado somente em situações específicas, e que fora delas não possui sentido. O excesso desse *smell* acaba acarretando em um *long method*
Message Chains	um objeto acessa outro, para então acessar outro objeto pertencente a este segundo, e assim por diante, causando um alto acoplamento entre as classes
Middle Man	pode ser identificado quanto uma classe não possui quase nenhuma lógica, pois delega quase tudo para outra classe. Geralmente ela não é mais necessária e referências a ela podem ser feitas diretamente à classe que efetivamente possui a lógica
Inappropriate Intimacy	caso onde duas classes se conhecem demais, caracterizando um alto nível de acoplamento
Alternative Classes with Different Interfaces	uma classe suporta diferentes classes, mas a interface delas é diferente. Por exemplo, uma classe que suporta tanto pessoas físicas como pessoas jurídicas mas para identificar o objeto sendo manipulado chama métodos diferentes, como *getCPF()* ou *getCNPJ()*
Incomplete Class Library	software utiliza uma biblioteca que não é completa, e portanto extensões a essa biblioteca são necessárias
Data Class	classe que serve apenas como um *container* de dados, sem comportamento algum. Geralmente outras classes são responsáveis por manipular seus dados, o que representa um caso de *Feature Envy*
Refused Bequest	indica que uma classe filha não utiliza dados ou comportamentos herdados
Comments	não podem ser considerados um *smell* por definição, mas devem ser utilizados com cuidado, pois normalmente não são necessários. Sempre em que se faz necessário a inserção de um comentário, vale a pena verificar se o código não pode ficar mais expressivo

Fonte: (FOWLER et al., 1999)

Tabela 2.3 – Lista de *Code Smells* apresentados por Wake

Smell	**Descrição**
Type Embedded in Name	são nomes usados, geralmente são definidos com duplicação, como *schedule.addCourse(course)* em vez de *schedule.add(course)*. Entram também nesta categoria o uso da notação húngara e variáveis que refletem seu tipo em contraponto ao seu propósito ou função
Uncommunicative Names	nomes usados em elementos de software (geralmente atributos e variáveis locais) que não comunicam seu nome/intenção o suficiente, como *x* ou *value1*. Isto é ainda mais crítico quando usado em métodos e classes
Inconsistent Names	mesmo nome usado em lugares diferentes, com propósitos distintos
Dead Code	caracterizada por uma variável, atributo ou fragmento de código que não é usado em lugar algum. Geralmente, é resultado de uma mudança no código com uma limpeza inadequada
Null Check	ocorrências que aparecem repetidamente, verificando valores nulos de objetos
Complicated Boolean Expression	trechos de código envolvendo operadores booleanos como *and, or* e *not*
Special Case	são instruções condicionais complexas
Magic Numbers	valores numéricos que aparecem deliberadamente no código e que, invariavelmente, não mudam

Fonte: (WAKE, 2003)

Tabela 2.4 – Lista de *Code Smells* apresentados por Kerievsky

Smell	**Descrição**
Conditional Complexity	descreve que, apesar de estruturas condicionais não serem problemas por si só, o uso exagerado das mesmas trata-se de um *smell* que deve ser combatido
Indecent Exposure	acontece quando clientes têm acesso demais às classes que eles utilizam. Isso aumenta desnecessariamente a complexidade do sistema
Solution Sprawl	idêntica à *shotgun surgery*, onde uma alteração ocasiona mudanças em diversas partes do sistema
Combinatorial Explosion	é uma forma mais sutil, porém muito similar ao *duplicated code*, onde existem diversos trechos de código que executam a mesma função porém em objetos de tipos diferentes
Oddball Solution	ocorre quando existem duas maneiras de resolver o mesmo problema no mesmo sistema, o que geralmente é um sinal mais sutil de *duplicated code*

Fonte: (KERIEVSKY, 2004)

Tabela 2.5 – Taxonomia de *smells* proposta por Wake

Categoria	**Descrição**
Smells within Classes	representa elementos de software que estão dentro das classes. Dentro desta categoria, estão *smells* que podem ser identificados com métricas simples (*comments, long method, large class, long parameter list*), nomes que precisam ser melhorados (*type embedded in name, uncommunicative name, inconsistent names*), complexidade desnecessária (*dead code, speculative generality*), trechos de códigos que precisam ser removidos (*magic numbers, duplicated code, alternative classes with different interfaces*) e problemas em lógicas condicionais (*null check, complicated boolean expression, special case, simulated inheritance*)
Smells between Classes	representa como as classes se relacionam. Estão nesta categoria os *smells* que representam dados como objetos perdidos, com ausência de comportamento adequado (*primitive obsession, data class, data clump, temporary field*), relacionamento entre hierarquias de classes (*refused bequest, inappropriate intimacy, lazy class, combinatorial explosion*), balanceamento de responsabilidades (*feature envy, message chains, middle man*), mudanças no código (*divergent change, shotgun surgery, parallel inheritance hierarchies*) e a falta de uma biblioteca de classes (*incomplete library class*)

Fonte: (WAKE, 2003)

Tabela 2.6 – Taxonomia de *smells* proposta por Mantyla *et al.*

Categoria	**Descrição**
Bloaters	representa qualquer elemento no código que teve um crescimento muito grande e que não pode ser efetivamente tratada. Em geral, são mais difíceis de entender e modificar. Fazem parte desta categoria *long method, large class, primitive obsession, long parameter list* e *data clumps*
Object-Orientation Abusers	relaciona soluções que foram usadas no código, sem explorar princípios de um bom *design* OO (MARTIN; MARTIN, 2007). Entram nesta categoria *switch statements, temporary field, refused bequest, alternative classes with different interfaces* e *parallel inheritance hierarchies*
Change Preventers	estruturas de software que dificultam consideravelmente a modificação, em diversos pontos, seja em um elemento de código ou vários. Nesta categoria estão *divergent change* e *shotgun surgery*
Dispensables	representa os *smells* que são desnecessários e, portanto, devem ser eliminados. Estão nesta categoria *duplicated code, lazy class, data class* e *speculative generality*
Encapsulators	nesta categoria, tem-se dois *smells*: *middle man* e *message chains*, que são opostos. A eliminação de um, incide sobre o aumento do outro
Couplers	os *smells* desta categoria caracterizam um alto acoplamento. É o caso do *feature envy* e *inappropriate intimacy*

Fonte: (MANTYLA; VANHANEN; LASSENIUS, 2003)

2.2 Metáforas, Medicina e o Desenvolvimento de Software

A metáfora é um recurso linguístico muito utilizado, essencial na comunicação humana. Segundo Geary (2012), a metáfora não é só uma figura de linguagem, mas está presente de forma intensa, quase que imperceptível em tudo que as pessoas fazem. Estima-se que uma metáfora seja dita entre 10 e 25 palavras, o que equivale a 6 metáforas por minuto.

As metáforas têm sido utilizadas em diferentes contextos e áreas do conhecimento. Tanto na Ciência da Computação como na ES as metáforas estão constantemente presentes. Brooks (1995) utilizou a metáfora do *"no silver bullet"* para representar que na ES existem muitos desafios e que não existe uma única solução para todos os problemas. Elas têm sido empregadas para representar diversos conceitos, métodos, padrões, práticas e problemas no desenvolvimento de software. Os exemplos são muitos, como processos e métodos de trabalho (*Waterfall*, Espiral, Scrum (SOMMERVILLE, 2016)), arquitetura (*layers, slices, onion/hexagonal, big ball of mud* (MARTIN, 2017)), padrões (*adapter, strategy, visitor, proxy* (GAMMA et al., 1994)), estruturas de dados (mapas-dicionários, conjuntos, listas filas (SCHILDT, 2018)), entre outros. O *eXtreme Programming*, uma das metodologias ágeis, traz a metáfora como uma de suas práticas (BECK; ANDRES, 2004). Elas ampliam o poder de comunicação, trazendo uma visão compartilhada e colaborativa para os times.

Alguns termos usados neste trabalho, como dívida técnica (STERLING, 2011) e *smells* (FOWLER et al., 1999) também são exemplos de metáforas. A dívida técnica, com origem na Economia, tem trazido alguns conceitos mais específicos de finanças para o desenvolvimento de software, como *principal* e *juros* (AMPATZOGLOU et al., 2020). O *principal* está relacionado com o esforço necessário para eliminar ineficiências no projeto. Já os *juros* estão relacionados ao esforço de desenvolvimento adicional necessário para modificar o software ao longo do tempo.

Outra área que tem inúmeros exemplos e relações com o desenvolvimento de software é a área da saúde. Metáforas relacionadas com a Medicina (FEATHERS, 2004) tem sido usadas já a algum tempo. A Medicina[2] é uma área do conhecimento que pode trazer vários *insights* interessantes de como gerentes e times podem lidar com situações do cotidiano do desenvolvimento de software. O médico usa sua formação, métodos, técnicas e ferramentas para chegar de forma mais assertiva ao diagnóstico de doenças e tratamentos.

[2] mais informações em https://pt.wikipedia.org/wiki/Medicina

O profissional utiliza inúmeras informações, muitas vezes provenientes de ferramentas de apoio, como exames laboratoriais, diagnóstico por imagem, entre outros. Em situações mais críticas, quando um médico precisa fazer uma intervenção cirúrgica, ele usa testes e ferramentas para minimizar os riscos e o impacto do procedimento ao paciente.

Fazendo associações entre as duas áreas, pode-se considerar o software como um paciente. O software, conforme descrito anteriormente, é um produto virtual, ausente de propriedades físicas, no qual não se sabe por quanto tempo será importante para os negócios. No entanto, ao não ser mais relevante, ou mesmo por problemas (doenças) de manutenção e evolução, o software pode ser descontinuado (falecimento). Parnas (1994) abordou tais aspectos de amadurecimento de software e do cuidado preventivo necessário com *design*, documentação e revisões, no qual denominou de medicina preventiva. Vaucher et al. (2009) desenvolveram um estudo sobre o ciclo de vida de instâncias do *smell god class*, usando como metáfora norteadora o campo da epidemiologia. Mais recentemente, Tornhill and Borg (2022) desenvolveram uma métrica chamada *code health*, usada para medir a complexidade do código e identificar e medir atributos específicos da complexidade. Esta métrica está presente na ferramenta *CodeScene*.

Ao necessitar fazer uma manutenção no software que está em produção, o desenvolvedor precisa lidar com situações semelhantes a uma cirurgia, com uma análise da alteração, diagnóstico prévio dos impactos, avaliações e testes necessários para garantir seu funcionamento. Ao considerar os sintomas das doenças como problemas de *design* de código, é possível usar inúmeros tratamentos, como refatorações (FOWLER et al., 1999), heurísticas de limpeza (MARTIN, 2008; SEEMANN, 2021; MARTIN, 2021) e *tidyings* (BECK, 2023)[3], de forma imediata, planejada ou até considerar conviver com tais problemas. Isso representa a relação causal da medicina (MEILIA et al., 2020). Assim, no desenvolvimento de software, quando o desenvolvedor precisa fazer uma alteração no código, qual é o impacto dessa mudança? Como filtrar e priorizar os problemas? Quais pontos são afetados e por quê? Qual é uma maneira eficaz de tratar esses problemas? Portanto, para que os desenvolvedores tomem uma decisão mais correta, eles precisam do suporte de ferramentas de qualidade. O desenvolvedor de software, assim como médico, deve usar seu treinamento e conhecimento para poder manter e evoluir o software de uma forma sustentável, atendendo as requisitos de negócio (POPPENDIECK; POPPENDIECK, 2006).

A Medicina tem sido um campo rico não apenas pelas metáforas, mas também por

[3]pequenas organizações realizadas no código, em conjunto com as refatorações

emprestar procedimentos técnicos e científicos a outras áreas do conhecimento, inclusive à ES (GOUES et al., 2018). Esta relação entre teoria e prática, ajudando a reconciliar os padrões rigorosos da comunidade de pesquisa com as necessidades pragmáticas dos profissionais tem sido o papel da *Evidence-based Medicine* (EBM) (GUYATT et al., 1992). A EBM é definida pelo uso consciente, explícito e criterioso das melhores evidências atuais dos médicos para tomar decisões oportunas sobre o atendimento de pacientes. A EBM conecta os resultados de pesquisa às necessidades da prática clínica médica, e vice-versa.

A EBM já inspirou as práticas de pesquisa em ES. Kitchenham, Budgen and Brereton (2015) introduziram a *Evidence-based Software Engineering* (EBSE), produzindo recomendações para profissionais e pesquisadores. A EBSE reconhece que o impacto tanto na Medicina quanto na ES surge pela coleção de inúmeras fontes ao redor de uma ideia, sintetizadas por meio de estudos secundários (GOUES et al., 2018). Portanto, entre os métodos científicos oriundos da Medicina e usado na ES está a *Systematic Literature Review* (SLR). Além de outros métodos (WOHLIN et al., 2012), a comunidade de SE tem aceito estudos secundários e terciários como úteis e com grandes contribuições na área (KITCHENHAM et al., 2010).

Esta associação da ES com a Medicina serviu de inspiração para um trabalho anterior (LACERDA; PETRILLO; PIMENTA, 2020) e também para desenvolvimento de um projeto *open-source* (LACERDA, 2024). Seguindo a mesma linha, nesta tese, foi utilizado como inspiração o *Método de Hanlon* (Apêndice A), para criação de uma abordagem de priorização de *smells* no código, denominada *DR-Tools Code Health*.

2.3 Relação Qualidade-*Smell*-Refatoração

A qualidade é uma das questões mais críticas na ES, recebendo atenção de profissionais e pesquisadores. Desenvolver software com qualidade é essencial, mas preservar ou aumentar a qualidade do software durante a atividades de manutenção é ainda mais crítico.

Diferentes modelos de qualidade de software foram discutidos na literatura (LI; AVGERIOU; LIANG, 2015; ALVES et al., 2016; BEHUTIYE et al., 2017; BESKER; MARTINI; BOSCH, 2018). Cada modelo define um conjunto de atributos de qualidade de software, no qual alguns atributos são comuns em diferentes modelos. São exemplos de modelos de qualidade o FURPS [4], *McCalls Factor Model* (MCCALL, 1977) e a ISO/IEC

[4]https://en.wikipedia.org/wiki/FURPS

9126 (International Standards Organisation - ISO, 1991). O modelo ISO/IEC 9126 é o mais abrangente, fornecendo seis principais características classificadas como atributos externos (*Functionality, Reliability, Usability, Efficiency, Maintainability* e *Portability*). Atualmente, o modelo padrão de referência é a ISO/IEC 25010 (International Standards Organisation - ISO, 2011), uma evolução da ISO/IEC 9126.

Conforme apresentado por Lacerda et al. (2020), *smells* e refatorações tem algumas características interessantes, no qual ambos afetam a qualidade do software. Os *smells* apontam problemas de qualidade de software (SANTOS et al., 2018; CAIRO; CARNEIRO; MONTEIRO, 2018; KAUR, 2020; SHARMA; SINGH; SPINELLIS, 2020). Primeiramente, impactam atributos externos de qualidade, como a manutenibilidade, complexidade e desempenho do software, levando o aumento da dívida técnica. Em segundo lugar, atributos internos de qualidade também são afetados pelos *smells*. Alguns *smells* produzem problemas de baixa coesão, alto acoplamento e problemas de encapsulamento que influenciam as decisões de *design* e manutenção. Porém, segundo alguns pesquisadores, este impacto na qualidade e pode ter um efeito oposto em diferentes atributos de qualidade (KAUR, 2020).

As refatorações e os *smells* estão ligados pois a refatoração é a principal estratégia para removê-los ou mitigá-los, melhorando aspectos como clareza, simplicidade e compreensão, sendo a abordagem primária para reduzir a dívida técnica (LI; AVGERIOU; LIANG, 2015; BEHUTIYE et al., 2017; BESKER; MARTINI; BOSCH, 2018). No entanto, se as refatorações forem aplicadas incorretamente, podem gerar novos *smells*, afetando negativamente a qualidade (ALVES et al., 2016). Na Figura 2.1, é apresentado um esquema desse relacionamento, com características específicas dos *smells* e refatorações, ambos conectados com a qualidade.

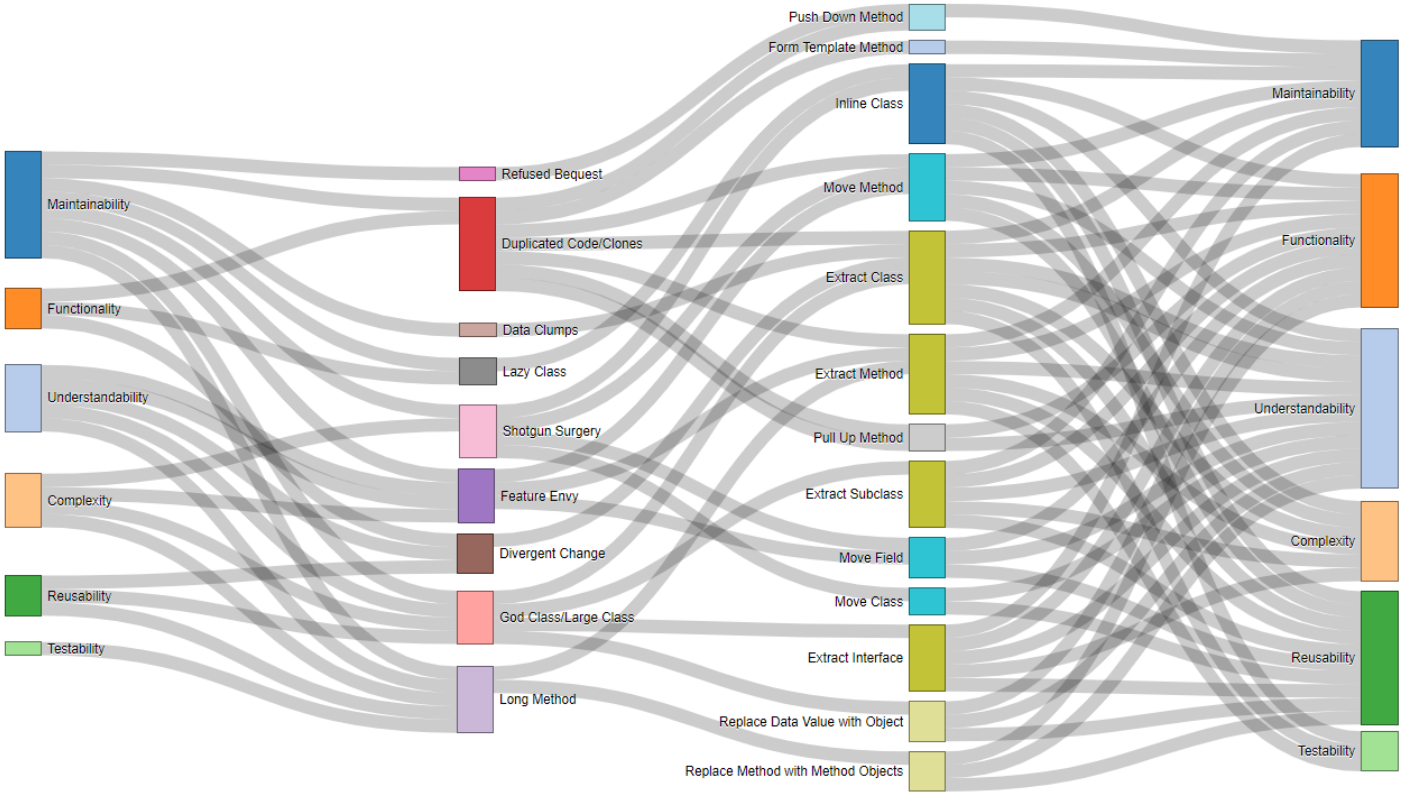
Figura 2.1 – *Code smells* e refatorações possuem algumas características similares. No centro, os aspectos de qualidade que os conectam

Fonte: Lacerda et al. (2020)

Alguns estudos identificaram o relacionamento entre *smells* e atributos de qualidade (SHARMA; SPINELLIS, 2018; CAIRO; CARNEIRO; MONTEIRO, 2018; SANTOS et al., 2018; ALKHARABSHEH et al., 2019). A natureza do relacionamento identificado pelos autores pode variar de um estudo para outro. A maioria dos *smells*, em particular, definidos por Fowler et al. (1999) afetam mais de um atributo de qualidade.

Com base nos dados coletados de forma sistemática, Lacerda et al. (2020) apresentaram a relação entre atributos de qualidade com *smells* e refatorações. Conforme a Figura 2.2, é observado o relacionamento entre atributos de qualidade (coluna 1), os *smells* (coluna 2) mais citados nos estudos que são afetados pelos atributos de qualidade, as refatorações (coluna 3) que podem ser aplicados nos *smells* e o impacto no atributo de qualidade (coluna 4) quando a refatoração é usada. Alguns atributos de qualidade influenciam mais que outros.

Figura 2.2 – Relação dos atributos de qualidade com *smells* e refatorações



Fonte: Lacerda et al. (2020)

Segundo estudos analisados por Lacerda et al. (2020), os atributos de qualidade mais afetados pelos *smells* são *Maintainability, Complexity* e *Understandability*. Eles têm um papel significativo nos custos de manutenção de software. Observando os *smells*, os que mais afetam diferentes atributos de qualidade são *God Class/Large Class*, *Long Method* e *Feature Envy*. Alguns *smells* como, *e.g., Duplicated Code/Clones*, possuem um

grande conjunto de refatorações (NYAMAWE et al., 2018) que podem ser aplicados. E, ao escolher uma determinada refatoração, como *Extract Method*, tem impactos em mais de um atributo da qualidade. O estudo de impacto/efeito dos *smells* na qualidade tem recebido muita atenção até agora. Isso sugere que ainda não há evidências abrangentes e suficientes sobre a extensão dos efeitos adversos associados aos *smells* na manutenção e evolução do software (LACERDA et al., 2020).

Também, muitas razões por trás da aplicação das refatorações e de melhorias no código não estão relacionados somente a *smells* (VASSALLO et al., 2019), mas com melhorias na qualidade ou adaptação por causa das mudanças dos requisitos (SILVA; TSANTALIS; VALENTE, 2016). Pantiuchina et al. (2020) desenvolveram um estudo sobre as motivações para aplicar refatorações, estendendo o estudo proposto por Silva, Tsantalis and Valente (2016) e apresentando uma taxonomia com 67 motivações. Segundo a pesquisa, as principais motivações estão relacionadas a melhoria de entendimento e legibilidade, melhoria de *design* de código e na capacidade de manutenção. Ou seja, muitas das intenções e motivações para melhoria estão relacionadas a qualidade. Embora os *smells* não estejam diretamente relacionado com as motivações, eles estão diretamente relacionados as principais causas destas motivações. Portanto, entender como *smells* estão relacionados a atributos de qualidade e ter mecanismos que permitam mapear esta relação pode ajudar no processo. O conjunto de *smells* relacionados com estes atributos de qualidade são os que tem maior grau de prioridade para serem removidos (LACERDA et al., 2020; ABUHASSAN; ALSHAYEB; GHOUTI, 2020).

Assim, parece fazer sentido levar em consideração na priorização não apenas os *smells* detectados e mais críticos, mas também aspectos de qualidade. O número de atributos de qualidade e suas possíveis combinações com *smells* distintos são altos.

Neste trabalho, o foco é como os atributos de qualidade - mais especificamente com manutenibilidade - podem ser associados aos *smells* para auxiliar na priorização e filtragem dos elementos de código mais problemáticos.

2.4 *Smells* e a Dívida Técnica

O assunto Dívida Técnica (DT) tem recebido muita atenção da comunidade acadêmica[5] nos últimos anos (LACERDA et al., 2020). Isto se deve ao fato da indústria

[5]quase 3.5 milhões de resultados obtidos no Google Scholar, usando o termo *technical debt*, em abril/2024

também ter a mesma preocupação, conduzindo algumas pesquisas nos últimos anos. Em uma pesquisa realizada em 2020, conduzida pela *Code Ahoy* (Code Ahoy, 2020) com mais de 100 desenvolvedores, mais de 60% das pessoas respondentes dizem trabalhar com produtos de software que possuem uma alta/muita alta dívida, no qual 80% estão cientes da dívida, mas afirmam não ter um plano ou não se preocupar com os problemas. Já em outro estudo realizado em 2021, conduzido pela *OutSystems* (OutSystems, 2021), 69% dos líderes de TI identificam a DT como uma grande ameaça à capacidade de inovação de suas empresas. Em 2022, o *Project Management Institute* (Project Management Institute, 2022) apresentou os resultados de uma pesquisa conduzida com pessoas com atuação em desenvolvimento e gestão de projetos sobre DT. Segundo os resultados apresentados, poucas organizações tem lidado com a DT de forma intencional, indicando que a dívida está fora de controle das empresas. No entanto, a liderança geralmente entende a DT e reconhecem que, atualmente, é um problema de gestão.

Tornhill and Borg (2022) discutem uma série de aspectos relacionados a DT, com base em sua pesquisa mais recente. Os autores chegaram a algumas constatações, entre elas: a) o desenvolvimento de software raramente é sustentável. A organização desperdiça em média de 23 a 42% do seu tempo de desenvolvimento por DT; b) a contratação de mais desenvolvedores aumenta custos de coordenação, o que, por sua vez, torna o desenvolvimento menos eficiente, principalmente em bases de código repletas de DT; c) Se sua organização gasta mais de 15% em trabalho não planejado, então isso é um sinal de que o potencial de entrega é desperdiçado. A DT provavelmente será um pedaço desse lixo; d) Com base em dados, muitas organizações pagam por 100 desenvolvedores, mas estão recebendo apenas o equivalente a 75 desenvolvedores; e) A DT é apenas um fator que muitas vezes vem junto com questões de time ou processo que precisam ser compreendidas e abordadas. Ferramentas modernas ajudam a detectar esses gargalos; f) Ao abordar as causas raízes, uma organização tende a aumentar sua capacidade de desenvolvimento eficaz pelo menos 25%, e finalmente g) 25% de capacidade extra significa que você pode fornecer mais funcionalidades e também obter uma visão clara ganhar na satisfação do cliente devido a qualidade melhorada. Assim, a DT é um assunto vasto, conectando vários aspectos (negócio e tecnologia) com partes (requisitos, arquitetura, design, código, documentação, testes, qualidade, entre outros) do desenvolvimento de software.

A metáfora da *dívida técnica* (DT) foi citada pela primeira vez por Ward Cunningham em 1992 no OOPSLA[6] (CUNNINGHAM, 1992). Esse termo foi utilizado para

[6] *Objected Oriented Programming System, Languages & Applications é a principal conferência mundial na área de Sistemas de Software Orientada a Objetos*

retratar a situação em que, para acelerar o desenvolvimento de software, é necessário o desenvolvimento de um código imaturo à ser reescrito futuramente, gerando assim uma dívida. Segundo Cunningham (1992), o problema ocorre quando essa dívida não é reembolsada, pois cada minuto que o código é mantido em inconformidade, juros são acrescidos, tornando-a cada vez mais difícil de ser paga.

Martin Fowler (2023a) faz uma analogia desse acúmulo de problemas que degradam o software, impedindo sua evolução com o acúmulo de lixo.

> *Os sistemas de software são propensos ao acúmulo de lixo - deficiências na qualidade interna que tornam mais difícil do que seria ideal modificar e estender ainda mais o sistema. A dívida técnica é uma metáfora, cunhada por Ward Cunningham, que enquadra como lidar com esse lixo, pensando nele como uma dívida financeira. O esforço extra necessário para adicionar novas funcionalidades são os juros pagos sobre a dívida.*
>
> *O que mais me atrai sobre a metáfora da dívida é como ela enquadra em como eu penso sobre como lidar com esse lixo. Eu poderia levar cinco dias para limpar a estrutura modular, removendo aquela sujeira, metaforicamente pagando o principal. Se eu fizer isso apenas para esse recurso, não há ganho, pois levaria nove dias ao invés de seis. Mas se eu tiver mais dois recursos semelhantes chegando, acabarei mais rápido removendo o lixo primeiro.*
>
> *Pensar nisso como o pagamento de juros versus o pagamento do principal pode ajudar a decidir qual problema enfrentar. Se eu tenho uma área terrível na base de código, uma que é um pesadelo para mudar, não é um problema se eu não tiver que modificá-la. Eu só aciono um pagamento de juros quando tenho que trabalhar com essa parte do software (este é um lugar onde a metáfora se desfaz, já que os pagamentos de juros financeiros são acionados pela passagem do tempo).*

A DT (STERLING, 2011) reflete o compromisso técnico de sacrificar a saúde de longo prazo de um produto de software em termos de benefícios a curto prazo. Esse sacrifício pode acarretar no aumento da complexidade e dificuldades de manutenção (ALLMAN, 2012). Isto se deve as modificações que podem acontecer por diversas razões (POPPENDIECK; POPPENDIECK, 2006), de forma consciente ou inconsciente por parte dos desenvolvedores. E, estas modificações realizadas ao longo do tempo é que causam uma degradação no software. Segundo Allman (2012), a DT é o resultado da la-

cuna entre as boas práticas de desenvolvimento e fatores presentes no projeto, como falta de tempo e custos.

A DT pode se tornar um empecilho que impede o sistema de evoluir de forma eficiente, resultando em menor produtividade e maiores custos de manutenção (EISENBERG, 2012). Assim como os *smells*, a DT é diferente de *bugs* e falhas, podendo ser sintomas da DT (CURTIS; SAPPIDI; SZYNKARSKI, 2012). Como a DT se origina do domínio financeiro (CHARALAMPIDOU, 2019), também envolve um conjunto de conceitos financeiros que ajudam a pensar sobre a qualidade de software em ternos de negócios. A seguir, são apresentados alguns conceitos relacionados ao tema (CURTIS; SAPPIDI; SZYNKARSKI, 2012; LI; AVGERIOU; LIANG, 2015; AMPATZOGLOU et al., 2020):

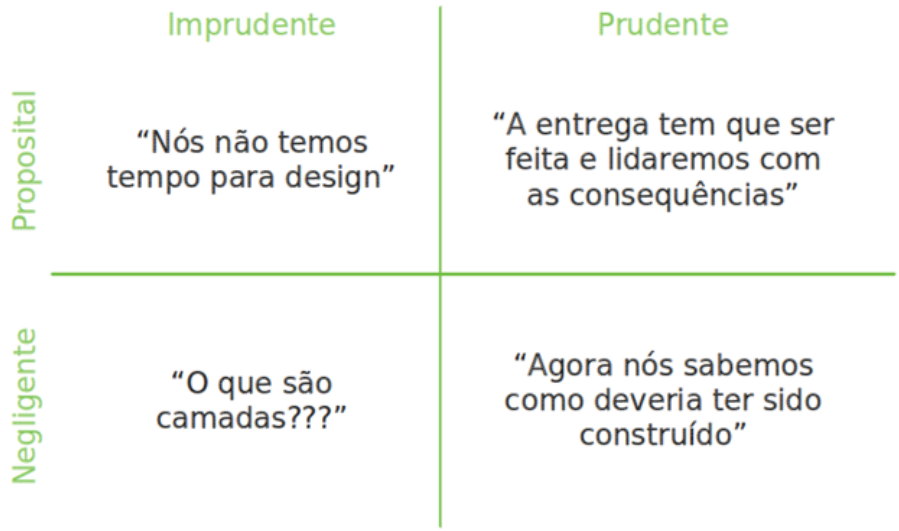
- **Dívida Técnica**: Custos futuros atribuídos a falhas estruturais conhecidas no software que precisam ser consertadas. Um custo pode incluir tanto o *principal* quanto *interest*;
- **Principal**: O custo estimado de resolução de um determinado tipo de DT;
- **Juros**: Esforço extra necessário (juros) para modificar parte do software que contém uma DT;
- **Risco**: DT é um tipo de risco para o projeto de software, uma que vez que pode eventualmente prejudicar a integridade do projeto, caso não seja resolvido;
- **Tipo**: DT pode ser classificada em vários tipos como DT de arquitetura e documentação;
- **Probabilidade de Juros**: Probabilidade que um tipo particular de DT terá, de fato, consequências visíveis;
- **Decisão Técnica**: DT é causada por decisões técnicas, como decisões de arquitetura;
- **Taxa de Juros**: Percentual dos juros de uma parte específica da DT fora do seu *principal* em um dado período;
- **Item de DT**: é uma unidade de DT em um sistema de software como, por exemplo, um *smell* detectado em uma parte do código. A DT em sistema é composto por vários itens de DT;
- **Custo**: Custo de incorrer em DT, ou seja, a soma do *principal* e *interest*;
- **Causa**: A razão para a existência da DT;
- **Sintoma**: Um sinal da manifestação da existência da DT;

- **Consequências**: Influências de incorrer em DT no sistema de software;
- **Valor/Benefício**: Benefício potencial que pode ser obtido imediatamente ou futuramente após o pagamento da DT;
- **Falência**: Ocorre quando a parte do sistema de software que contém a DT não é mais viável para suportar futuras modificações e uma reescrita completa é necessária.

Para uma melhor compreensão da metáfora da DT, alguns autores preferem categorizar as dívidas encontradas em seus estudos. É o caso de McConnell (2008), que divide DT em intencional e sem intenção. Se a dívida for classificada como intencional, o autor ainda faz outra divisão: a dívida à curto e a longo prazo. A DT sem intenção é resultado não-estratégico de um trabalho mal planejado, contraída de forma involuntária (MCCONNELL, 2008). Como esta dívida não é esperada, pode-se dizer que ela causa um impacto negativo no projeto. Já a DT intencional acontece normalmente por priorização de outras dívidas, mas também pode ser resultado de um cronograma muito rígido. A equipe pode contrair essa dívida numa atitude estratégica para otimizar o projeto. A DT à curto prazo é assumida quando não se possui uma solução mas se está próxima dela (MCCONNELL, 2008). A dívida à curto prazo é utilizada para cobrir pequenas lacunas de tempo, por exemplo, a necessidade da entrega de uma funcionalidade em um curto período. O autor ainda vê a DT à longo prazo como uma decisão estratégica e pró-ativa. Assumida com o intuito de suprir necessidades de grande porte, é uma dívida que, quando bem gerenciada, pode sobreviver anos dentro do projeto sem causar problema.

Da mesma forma que McConnell (2008), Martin Fowler (2023b) definiu quadrantes para classificar a DT, ilustrado na Figura 2.3. A dívida imprudente geralmente surpreende os times de desenvolvimento de software, tornando-se um fator significativo para estouros de custo e prazos. É uma sequência de más escolhas que podem resultar em altos juros ou em um longo período para seu respectivo pagamento (ROSSER; NORTON, 2021). Uma DT imprudente intencional é uma dívida assumida de forma negligente pelo time. Caso seja esquecida, se torna um problema, podendo gerar juros impossíveis de pagar levando até mesmo à falência do projeto (Martin Fowler, 2023b). A DT prudente normalmente é gerenciada ao invés de eliminada. É uma dívida pequena e que fica em um local pouco acessado, pois causa pouco impacto e pode ser mantida no projeto (Martin Fowler, 2023b). Rosser and Norton (2021) observam que, quando a DT é aplicada com prudência, será identificada, avaliada como a melhor resposta de curto prazo, registrada e marcada para remediação antes da próxima execução em produção do software.

Figura 2.3 – Quadrantes da Dívida Técnica



Fonte: (Martin Fowler, 2023b)

A DT representa os efeitos de artefatos imaturos na evolução do software que trazem benefícios de curto prazo, mas que precisam ser ajustados posteriormente. O conceito, cujo escopo era inicialmente limitado ao código e artefatos relacionados, foi expandido para considerar diferentes estágios de desenvolvimento de software e produtos de trabalho (ALVES et al., 2016). Rios, Neto and Spínola (2018) fornecem uma taxonomia com 15 tipos de DT, conforme apresentado na Tabela 2.7.

As dívidas relacionadas a *design*, código e arquitetura são as que recebem mais atenção da comunidade, sendo os mais estudados (DETOFENO; MALUCELLI; REINEHR, 2023). Isso provavelmente ocorre porque várias ferramentas de análise de código ajudam a identificar problemas como código complexo, *smells* e outros, que geralmente servem como indicadores de DT. Os autores também definem uma lista de situações em que itens de DT podem ser encontrados no software (DETOFENO; MALUCELLI; REINEHR, 2023). *God Class/Large Class* tem sido o *smell* mais investigado. Este *smell* é conceitualmente fácil de entender, tem 13 vezes mais chances de serem afetados por defeitos e até 7 vezes mais propenso a mudanças, o que os torna bons candidatos para mitigação de DT (LACERDA et al., 2020).

Embora a DT afete todos os envolvidos no projeto, independentemente da causa, o nível de comunicação em relação ao DT varia. Os membros da equipe geralmente discutem DT entre si, mas entendem que há dificuldades em apresentar evidências de gestão da dívida técnica (GDT) ao nível superior de gestão (CODABUX et al., 2017). A GDT inclui identificar, monitorar e pagar itens de DT incorridos em um sistema (GRIFFITH

Tabela 2.7 – Tipos de Dívida Técnica

Tipo	**Descrição**
Architecture Debt	Refere-se aos problemas encontrados na arquitetura do produto, que podem afetar os requisitos de arquitetura. Normalmente, a dívida de arquitetura pode resultar de soluções iniciais sub-ótimas ou soluções sub-ótimas à medida que tecnologias e padrões são substituídos, comprometendo alguns aspectos internos de qualidade, como a manutenibilidade
Test Automation Debt	Refere-se ao trabalho envolvido na automação de testes das funcionalidades previamente desenvolvidas para suportar a integração contínua e ciclos de desenvolvimento mais rápidos
Build Debt	Refere-se a problemas que tornam a tarefa de construção mais difícil e demorada desnecessariamente
Code Debt	Refere-se aos problemas encontrados no código (código mal escrito que viola as melhores práticas de codificação ou regras de codificação) que podem afetar negativamente a legibilidade do código, tornando-o mais difícil de manter
Defect Debt	Refere-se a defeitos conhecidos, geralmente identificados por atividades de teste ou pelo usuário e relatados em sistemas de *bug tracking*, que o time de desenvolvimento concorda que devem ser corrigidos, mas, devido à prioridades concorrentes e recursos limitados, devem ser adiados para uma mais tarde
Design Debt	Refere-se à dívida descoberta pela análise do código-fonte e pela identificação de violações sólidas dos princípios de design orientado a objetos
Documentation Debt	Refere-se aos problemas encontrados na documentação do projeto de software
Infrastructure Debt	Refere-se a problemas de infraestrutura que podem atrasar ou atrapalhar algumas atividades de desenvolvimento se presentes na organização de software. Tais problemas afetam negativamente a capacidade da equipe de produzir um produto de qualidade
People Debt	Refere-se a questões que podem atrasar ou dificultar algumas atividades de desenvolvimento se presentes na organização de software". Isso é representado para contratação tardia, por exemplo
Process Debt	Refere-se a processos ineficientes, por exemplo, o que o processo foi projetado para tratar pode não ser mais adequado
Requirement Debt	Refere-se às compensações feitas em relação a quais requisitos a equipe de desenvolvimento precisa implementar ou como implementá-los. Em outras palavras, refere-se à distância entre a especificação de requisitos ótimos e a implementação real do sistema
Service Debt	Refere-se à seleção e substituição inadequada de serviços web que levam a uma incompatibilidade entre as características do serviço e os requisitos das aplicações. Este tipo de dívida é relevante para sistemas com arquiteturas orientadas a serviços
Test Debt	Refere-se a problemas encontrados em atividades de teste que podem afetar a qualidade dessas atividades
Usability Debt	Refere-se a decisões de usabilidade inadequadas que devem ser ajustadas posteriormente
Versioning Debt	Refere-se a problemas no versionamento do código-fonte, como bifurcações de código desnecessárias

Fonte: (RIOS; NETO; SPÍNOLA, 2018)

et al., 2014). Li, Avgeriou and Liang (2015) desenvolveram um estudo sistemático que apresentou uma visão mais abrangente sobre o conceito de DT e uma visão geral sobre a gestão da dívida técnica (GDT). A GDT é composta por um conjunto de atividades (Tabela 2.8) que impedem ou lidam com a ocorrência de DT em potencial, mantendo abaixo de um nível razoável.

Tabela 2.8 – Atividades de Gestão da Dívida Técnica

Atividade	**Descrição**
Identificação	Detecta a DT causada por decisões técnicas intencionais ou não intencionais em sistema, utilizando técnicas específicas de detecção (*e.g.,* análise estática de código)
Medição	Quantifica o benefício e o custo da DT conhecida por meio de técnicas de estimativa
Priorização	Classifica os itens de DT identificados de acordo com certas regras predefinidas para tomada de decisão (quais itens de DT devem ser reembolsados primeiro e quais podem ser tolerados)
Prevenção	Tem por objetivo evitar que possíveis DTs ocorram
Monitoramento	Observa as mudanças de custo e benefício da DT não resolvidos ao longo do tempo
Reembolso/ Pagamento	resolve ou atenua a DT de um sistema de software por meio de técnicas específicas, como reengenharia ou refatoração
Representação/ Documentação	Fornece uma maneira de representar ou codificar a DT de maneira uniforme
Comunicação	Torna a DT identificada visível para todas as pessoas interessadas, para que possa ser discutida e gerenciada *a posteriori*

Fonte: (LI; AVGERIOU; LIANG, 2015)

A GDT possibilita tomar essas decisões sobre a eliminação do DT e o momento mais adequado para fazê-lo (GUO; SPÍNOLA; SEAMAN, 2016). Portanto, ela deve ser baseada em uma abordagem racional para a tomada de decisão, considerando o desenvolvimento futuro planejado e potencial (SCHMID, 2013). Algumas atividades, como identificação, medição e pagamento recebem mais atenção com o suporte de ferramentas apropriadas e abordagens (LI; AVGERIOU; LIANG, 2015). A atividade de pagamento refere-se às atividades realizadas para apoiar a tomada de decisão sobre o momento mais adequado para eliminar os itens de DT. As refatorações têm sido a abordagem primária para minimizar estes efeitos (LACERDA et al., 2020). Nesse ponto, é feita a priorização de qual item da DT deve ser eliminado (RIOS; NETO; SPÍNOLA, 2018).

Entretanto, a DT é muitas vezes confundida com mau código em geral. Esta é uma falácia perigosa que leva as organizações a perder meses em melhorias que não têm um claro resultado do negócio ou não são urgentes (TORNHILL; BORG, 2022). Por isso, a priorização é considerada a atividade mais importante da DT. Como aponta Li,

Avgeriou and Liang (2015), mais estudos industriais são necessários para mostrar como priorizar uma lista de itens de DT para maximizar os benefícios de um projeto de software. Também, é necessário avaliar quais fatores devem ser considerados durante a priorização de DT no contexto do desenvolvimento de software comercial. O DT relacionado ao código e seu gerenciamento ganharam mais atenção (LI; AVGERIOU; LIANG, 2015).

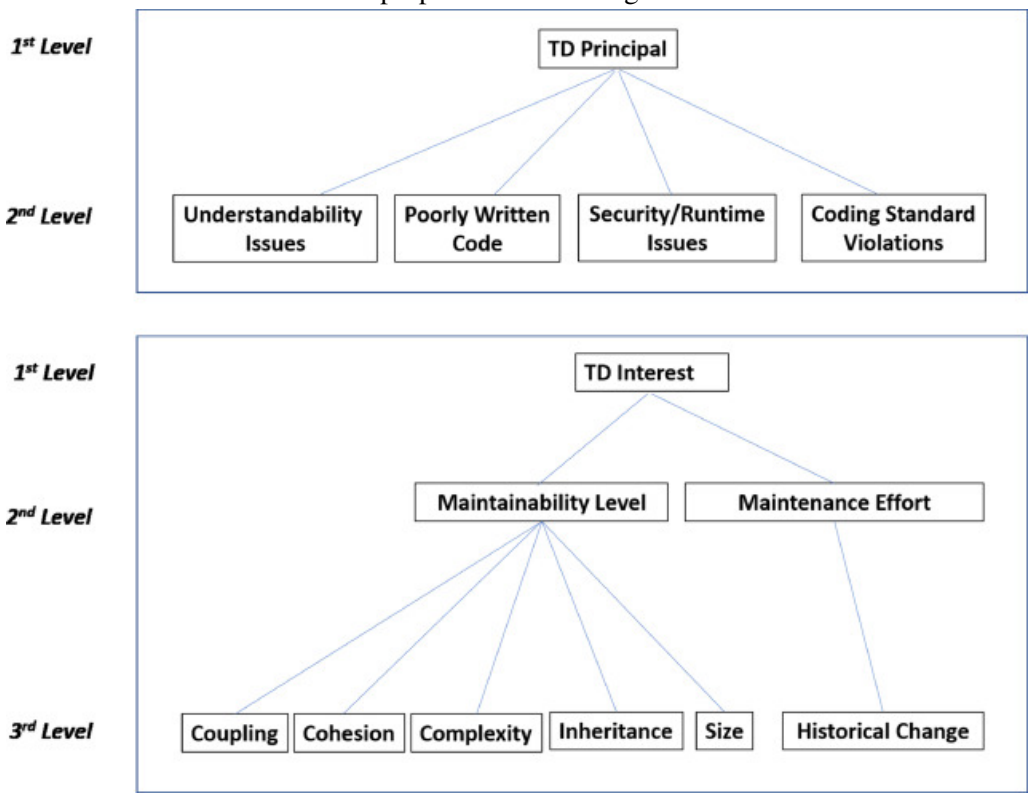
Ampatzoglou et al. (2020) desenvolveram uma pesquisa sobre DT, investigando a relação da DT principal com a DT juros em código, a partir de duas perspectivas: entender as relações subjacentes entre os conceitos e fornecer um caminho eficiente para gestão da DT. Para isso, os pesquisadores realizaram um estudo de caso com 3600 classes de 10 projetos *Apache*. Conforme Figura 2.4, é apresentado uma visão geral do principal e do juros. Dada a estrutura hierárquica dos conceitos, nota-se que o cálculo de cada aspecto em um nível superior é uma agregação do nível anterior. O principal é composto por 4 aspectos que correspondem a categorias de *smells*. Já o juros tem dois aspectos no segundo nível, relacionados à manutenção: nível de manutenibilidade e esforço de manutenção. O terceiro nível é formado por propriedades estruturais que compõem a manutenibilidade e as mudanças históricas que compõe o esforço de manutenção. Os resultados apresentados neste estudo sugerem que o principal está mais relacionado aos aspectos do juros tamanho e acoplamento, seguidos por coesão e complexidade. Em relação ao aspecto principal, aquele que pare e estar mais fortemente inter-relacionado a níveis mais altos de juros são os *smells*, dificultando a compreensão e, portanto são os mais urgentes a serem resolvidos. Os pesquisadores sugerem aprofundar estratégias para prevenção, reembolso e priorização da DT com base nestes aspectos.

Conforme abordado anteriormente, a DT é um assunto bastante amplo. Neste trabalho, explora-se um subconjunto desta temática, através da detecção do tipo de dívida relacionadas a código e *design*, que são os *smells*. Também, entende-se que a priorização é uma etapa essencial para apoiar o time na tomada de decisão em trabalhar nas partes do projeto que realmente merecem sua atenção.

2.5 Priorização de *Smells*

Geralmente, os desenvolvedores sabem o que está ruim no código (LEPPANEN et al., 2015b). Mesmo assim, há casos que os desenvolvedores não são capazes de identificar as oportunidades de melhoria, seja por falta de conhecimento ou falta de compreensão do código que afetam seu julgamento (YAMASHITA; MOONEN, 2013a; HOZANO et al.,

Figura 2.4 – Aspectos da DT (*Principal* e *Juros (Interest)*) e suas relações com características e propriedades do código



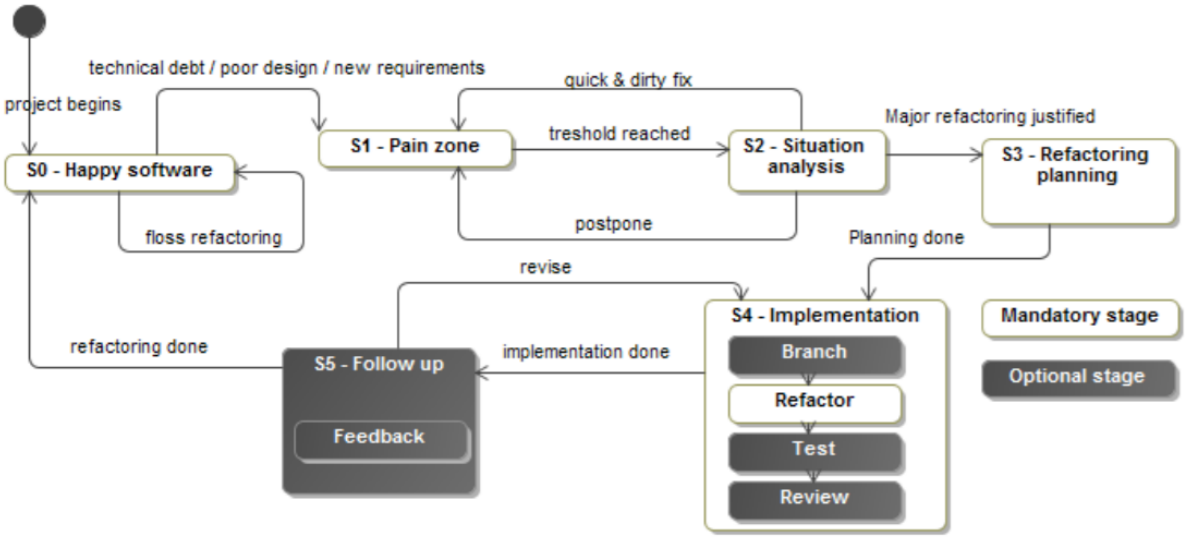
Fonte: (AMPATZOGLOU et al., 2020)

2018).

Segundo Taibi, Janes and Lenarduzzi (2017), os desenvolvedores estão preocupados com *smells* na teoria (nocivos e prejudiciais), mas quando foram solicitados para analisar trechos de código, apenas alguns foram considerados prejudiciais. A partir de um grande número de descobertas de problemas de *design* no projeto, os desenvolvedores não têm um ponto de partida para um processo de melhoria a longo prazo (STEIDL; EDER, 2014). Aversano, Carpenito and Iammarino (2020) confirmam que poucas classes são responsáveis por grande parte das violações de *design* em projeto. Muitas vezes, eles não corrigem nenhuma descoberta porque não sabem em quais descobertas vale a pena gastar esforço. Para realmente começar a removê-las, os desenvolvedores precisam de um mecanismo de priorização para poder diferenciá-las. Frequentemente, a decisão de remover um problema não depende apenas de encontrá-lo, mas também de informações contextuais adicionais como, se a descoberta está localizada em código problemático ou em contraste, em um código que será excluído. Em outra pesquisa, apenas os *smells* graves foram considerados para refatoração (SIMMONS, 2020). Porém, o que torna uma refatoração boa ou ruim é uma boa detecção dos problemas de *design*, evitando falsos positivos (SZOKE, 2017). Por isso a importância da priorização.

Muitas vezes, a motivação da priorização está relacionada com a aplicação de refatorações. Leppanen et al. (2015a) investigaram como os desenvolvedores tomam decisões acerca das refatorações. No trabalho, os autores propuseram um *framework* para tomada de decisões (Figura 2.5). Os autores identificaram, através de entrevistas com profissionais da indústria, os estágios, transições entre os estágios e fatores que contribuem para tomada de decisão. Segundo os autores, as decisões tendem a ser mais subjetivas e não necessariamente racional. No *framework*, os estágios S1 (*pain zone*) e S2 (*situation analysis*) são chaves para esta tomada de decisão, avaliando trechos de código que necessitam de manutenção e evolução. Embora os desenvolvedores sejam os especialistas com base em suas percepções e experiência, segundo os autores, ao invés de tomar decisões subjetivas, o uso de algum método de apoio para auxiliar na avaliação das partes problemáticas ajudaria essencialmente o processo de tomada de decisão.

Figura 2.5 – *Framework* para tomada de decisões para refatorações, no qual os estágios S1 e S2 são os locais para considerar a priorização e filtragem dos problemas



Fonte: Leppanen et al. (2015a)

Embora os *smells* sejam considerados uma ameaça à qualidade, não está claro seu impacto real (KAUR, 2020). Assim, os desenvolvedores e gerentes sentem-se relutantes em detectá-los e removê-los. Apenas a ação de detectá-los não indica, se de fato, são prejudiciais na sustentabilidade de um projeto de software (CATOLINO, 2020). Por outro lado, de acordo com Khomh et al. (2012), a presença dos *smells* aumentam a propensão a mudanças e falhas, em relação a elementos de código não afetados por qualquer *smell*. Tufano et al. (2017), em um estudo que contemplou mais de 200 projetos *open-source*, identificaram que cerca de 80% dos *smells* sobrevivem no sistema e 20% são removidos (9% através de refatorações). Além disso, os *smells* geralmente são introduzidos

pelos desenvolvedores ao alterar funcionalidades existentes ou implementar novas funcionalidades (PALOMBA et al., 2018b). Assim, é essencial os desenvolvedores utilizarem mecanismos de filtragem e priorização para planejar suas ações.

Sae-Lim, Hayashi and Saeki (2018a) desenvolveram o primeiro estudo empírico investigando fatores considerados por profissionais ao filtrar e priorizar *smells*. O estudo contou com 10 profissionais, pedindo que selecionassem e priorizassem *smells* de um projeto *open-source*, com a condição de que completassem uma lista de 5 tarefas. Assim, foi possível identificar as motivações usadas para selecionar e priorizar os *smells*. No total, obtiveram 69 respostas sobre filtragem e 50 respostas sobre priorização de *smells*. Entre os fatores estão severidade (indicativo da gravidade de um *smell*), acoplamento entre *smells* (um *smell* relacionado a outro e precisa ser resolvido em conjunto), *smells* co-localizados (múltiplos *smells* no mesmo módulo e devem ser resolvidos ao mesmo tempo), falsos positivos (*smells* não considerados ou não detectados, mas são um *smell*), relevância/importância/custo e risco de implementação de tarefas, atributos de qualidade (testabilidade, legibilidade, manutenibilidade), importância e dependência dos módulos e, finalmente, custos de refatoração. A pesquisa identificou como os mais usados a importância do módulo e a relevância das tarefas e os mais relevantes a severidade dos *smells* e a relevância das tarefas (SAE-LIM; HAYASHI; SAEKI, 2018a).

No estudo realizado por Amanatidis et al. (2018), os pesquisadores estudaram os fatores que afetam a decisão dos desenvolvedores de concordar com a remoção de um passivo de dívida técnica. Foram utilizados projetos escritos em PHP, com o uso do *Sonarqube* para análise. Com base nos resultados, 36% dos casos concordaram com a necessidade de aplicar refatorações para remover um problema, com nível de concordância alto ou muito alto. Os desenvolvedores parecem ser amplamente influenciados pela gravidade de um problema. A decisão dos desenvolvedores parece também não ser afetadas por fatores como frequência das modificações nos arquivos. A descoberta pode estar relacionada a uma crença latente de que a análise automatizada da qualidade tende a superestimar os problemas. No entanto, os desenvolvedores devem se concentrar no problema em si e não no contexto. Ainda, segundo o estudo, problemas de testabilidade são 3,9 mais propensos a serem considerados como necessitando de resolução do que um problema relacionado a manutenibilidade.

Outro estudo sistemático desenvolvido por Lenarduzzi et al. (2020) discutiu estratégias, processos, fatores e ferramentas para priorização no âmbito da dívida técnica. Foram considerados 557 artigos, nos quais 44 estudos primários foram selecionados. Se-

gundo as evidências identificadas, os principais tipos de dívida técnica são de código, arquitetura e *design*. A priorização é considerada a principal atividade de gestão da dívida técnica, usada para definir a ordem e programação das iniciativas do time. Há, também, uma falta de um conjunto de ferramentas empíricas e validadas para priorização. Também não há um consenso sobre quais são os fatores importantes a serem usados e como medi-los. Ao adicionar mais um elemento na análise como, *e.g.,* o tempo, pode-se ampliar os aspectos de priorização. Ainda em relação a dívida técnica, Charalampidou (2019) considera a probabilidade da mudança nas classes como um dos aspectos para priorização.

Em outro estudo recente, Pina, Goldman and Tonin (2021) desenvolveram um mapeamento sistemático para identificar métodos de priorização de dívida técnica, fornecendo uma classificação para estes métodos. A partir dos 51 estudos selecionados, os autores definiram uma taxonomia de 2 níveis, englobando 10 categorias. Também, identificaram características técnicas, práticas e requisitos para os métodos de priorização em projetos reais. Os principais achados, segundo os autores, é que embora tenham vários métodos na literatura, nenhum deles é adaptado ao contexto, que sejam independentes de linguagem de programação e que cobrem vários tipos de dívida técnica. Eles ainda enfatizam que há uma falta de ferramentas para usar estes modelos de priorização.

Um dos principais fatores que afetam a decisão dos desenvolvedores em concordância e remoção da dívida técnica é a severidade (AMANATIDIS et al., 2018). A severidade descreve o quão problemático é um *smell*, sendo um dos fatores mais populares usados para priorizá-los. Marinescu definiu severidade como uma métrica que apresenta quantas vezes o valor de uma métrica excede determinado *threshold* (MARINESCU, 2012). A classificação da severidade dos *smells* pode ser subjetiva, uma vez que a sua definição é informal e diferentes desenvolvedores podem considerar diferentes aspectos dos *smells* ao avaliar manualmente suas instâncias (FONTANA; ZANONI, 2017). Por isso, seria importante usar técnicas para apoiar a tomada de decisões. Cada potencial *smell* também tem diferentes pontuações de severidade, afetando ou contribuindo negativamente em diferentes aspectos de qualidade (SEHGAL; MEHROTRA; BALA, 2018).

Sjoberg et al. (2013) desenvolveram um estudo para quantificar os efeitos dos *smells* no esforço de manutenção. Neste trabalho, foi utilizada a densidade, definida pela quantidade de *smells* em relação ao tamanho de código[7]. Este também tem sido um fator usado para a priorização (OIZUMI et al., 2019).

Oizumi et al. (2019) desenvolveram um estudo sobre sintomas de degradação es-

[7] geralmente, usa-se a referência de tamanho em LOC

trutural, com foco na aplicação de *root canal refactorings*[8]. A degradação estrutural está relacionada a ocorrência de decisões que afetam atributos relacionados a dependência, de forma negativa. Os autores levaram em consideração informações como densidade e diversidade para priorização. Até então, segundo os autores, a maioria dos estudos eram mais focados em densidade. A densidade está relacionada a várias instâncias de um sintoma por LOC e diversidade a vários tipos de sintomas em um trecho de código. Os autores observaram que os desenvolvedores tendem a aplicar refatorações em classes com alta densidade e alta diversidade de sintomas. No entanto, a maioria das técnicas é limitada a um atributo de qualidade. Foram analisados 8 projetos de software. Inclusive, esse baixo número de projetos é uma das limitações identificadas pelos autores, que pode não ser suficiente para generalizar os resultados.

Outro fator considerado nas pesquisas sobre priorização é a co-ocorrência (FONTANA; FERME; ZANONI, 2015b). A co-ocorrência está relacionada com entidades que são afetadas por mais de um *smell*. Alguns autores usam o termo diversidade (OIZUMI et al., 2019) ou co-localização (SHARMA; SINGH; SPINELLIS, 2020). Um dos primeiros estudos a definir tipos de relações entre os *smells* foi de Pietrzak e Walter (PIETRZAK; WALTER, 2006). Ainda, alguns pesquisadores exploraram mais a fundo como são estas relações de co-ocorrência, chamadas de aglomerações (OIZUMI et al., 2016). A aglomeração é um grupo de *smells* inter-relacionados que afetam a mesma parte do código como um mesmo método, classe, hierarquia ou pacote (ou componente) (OIZUMI et al., 2017). A aglomeração pode ser categorizada em intra-método, intra-classe, hierárquico e intra-componente (OIZUMI et al., 2016).

Palomba et al. (2018a) apresentou um estudo em que 59% das classes que continham problemas tinham mais de um *smell*, sendo estas mais propensas a mudanças e falhas. Já Oizumi et al. (2017) desenvolveram um estudo de método misto para analisar se e como os desenvolvedores podem efetivamente encontrar problemas de *design* ao refletir sobre *stinky code*[9]. Os autores fizeram um experimento e uma entrevista com 11 profissionais, identificando que 36,36% dos desenvolvedores encontraram mais problemas de *design* ao raciocinar explicitamente sobre múltiplos *smells* em comparação com *smells* únicos.

Sharma, Fragkoulis and Spinellis (2017) estudaram a co-ocorrência de *smells*, investigando as correlações de inter-categorias e intra-categorias no nível de método e

[8] *Root canal refactoring*, assim como *floss refactoring* são táticas adotadas para implementar refatorações (MURPHY-HILL; PARNIN; BLACK, 2012)

[9] termo usado no trabalho para indicar o local de um programa afetado por vários *smells*

classes. O estudo analisou 19 *smells* de classe e 11 *smells* de método em 1988 repositórios de código C#, contendo mais de 49 milhões de LOC. Os resultados apresentaram a maior frequência dos *smells Unutilized Abstraction* e *Magic Number*, mostrando também que os *smells* de método e de classe estão fortemente correlacionados.

Abbes et al. (2011) estudou a co-ocorrência de *smells* como *Blob* e *Spaghetti Code* na compreensão de programas. Os resultados obtidos apontam que a presença de mais de um *smell* diminui fortemente a capacidade dos desenvolvedores em lidar com tarefas de compreensão. A interação entre diferentes instâncias de *smells* também foi estudada por Yamashita and Moonen (2013b), que confirmou que os desenvolvedores têm mais dificuldades para trabalhar em classes afetadas com mais de um *smell*. Os mesmos autores analisaram este mesmo fenômeno e como ele impacta a manutenibilidade (YAMASHITA; MOONEN, 2012).

Martins et al. (2020) investigaram se a remoção de co-ocorrências através de refatorações tem impacto positivo nos atributos de qualidade. Na avaliação, foram considerados 11 versões de 3 sistemas *open-source*, observando o impacto da remoção (antes/depois) e quais co-ocorrências são as mais difíceis. Segundo os autores, a remoção de co-ocorrência reduz significativamente a complexidade. Os *smells* mais difíceis e mais frequentes de aplicar refatorações são *God Class com Long Method e Dispersed Coupling com Long Method*, indicando que estas anomalias precisam ser monitoradas.

Baabad et al. (2020) desenvolveram um estudo sistemático sobre a degradação da arquitetura em projetos *open-source*. No trabalho, foram considerados 8 bases de dados e 74 estudos primários, reportando como principais sintomas e razões de degradação a rápida evolução do software, mudanças frequentes e a falta de conscientização dos desenvolvedores, nos quais os principais indicadores são os *smells* de código e arquitetura. Embora *smells* individuais sejam estudados mais frequentemente como indicadores de degradação, a co-ocorrência de *smells* apresenta uma correlação significativa com a arquitetura do que problemas individuais. Ainda, uma das estratégias apontadas pelo estudo para lidar com a erosão arquitetural são as priorizações de *smells* de arquitetura e anomalias de código. A questão aqui é que se um *smell* estiver relacionado com outros *smells* pode ser mais problemático que um *smell* isolado (SOUSA et al., 2020). Este fator também pode ser usado na priorização e filtragem.

Por todos estes pontos levantados, pouco se sabe em como apoiar os desenvolvedores na priorização dos *smells*, sendo um campo aberto de pesquisa (PECORELLI et al., 2020). Assim, recursos de priorização e filtragem se tornam cada mais relevantes

(SADOWSKI; STOLEE; ELBAUM, 2015).

2.6 Ampliando a discussão sobre os *Smells*

O entendimento pelos desenvolvedores de pontos que estão ruins e precisam de modificações na maioria das vezes são motivados por aspectos de qualidade e não somente pela detecção dos *smells* (VASSALLO et al., 2019). Segundo Yamashita (2015), um atributo de qualidade está relacionado a um problema se o trecho de código identificado contém um problema mapeado ao atributo de qualidade. Assim, parece fazer sentido associar os problemas no código com atributos de qualidade na sua definição, ampliando a forma de detecção e filtragem.

Contudo, os *smells* também apresentam os problemas relacionados à nomenclatura e definição (LACERDA et al., 2020), independente do nível de granularidade. Por exemplo, *Blob* (BROWN et al., 1998) é usado em alguns estudos como sinônimo de *Large Class* (FOWLER et al., 1999) ou *God Class* (RIEL, 1996). Um outro caso, *Copy and Paste Programming* foi usado como sinônimo para *Duplicated Code/Clones*. Foram identificados estudos distintos (SHARMA; SPINELLIS, 2018; SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019) usando nomes de *smells* diferentes para descrever o mesmo problema no *design* e de código. Ou seja, o uso de termos e classificações adotados por diferentes autores não são sólidos. Por um lado, encontrou-se definições distintas para o mesmo nome de *smell*. Por outro lado, a mesma definição de *smell* é apresentada com nomes diferentes. De acordo com Sobrinho, Lucia and Maia (2018), essa fragmentação de definições é devido à falta de uma taxonomia mais sistemática ou formal para anomalias de código.

Um *smell* representa um sintoma, um indicador de problemas e violações recorrentes de princípios de *design* do código que impactam um ou mais atributos de qualidade e, especificamente, a manutenibilidade. O *design* envolve qualquer decisão tomada na estrutura do código, desde nomenclaturas (variáveis, métodos, atributos, classes, pacotes), estruturas de dados, decisões e iterações, bem como algoritmos usados. Os *smells* são adotados para representar problemas, independente de granularidade. Um *smell* é diferente de um *bug*, pois ele não necessariamente causa um erro na aplicação, mas pode implicar em outras consequências negativas na manutenção e evolução do software.

Neste trabalho, é usado o termo *smell* para representar sintomas em diferentes granularidades, mas que também estejam associados a atributos de qualidade. Isso se deve

a natureza subjetiva e informal das definições dos *smells*, sensíveis a percepção dos desenvolvedores, necessitando diretamente da eficiência de um processo automatizado de personalização (HOZANO et al., 2017b). Outro motivo, também, é que os desenvolvedores usam os termos *smells* e *anti-pattern* de maneira liberal, onde grande parte do código designado como *smell* ou *anti-pattern* é apresentado sem mencionar nomes específicos (TAHIR et al., 2020). Finalmente, se as principais motivações para aplicar refatorações e limpezas no código (PANTIUCHINA et al., 2020; ALOMAR et al., 2021) e, junto com a preocupação do time ao fazer modificações no código (PALOMBA et al., 2018b), envolvem aspectos de qualidade, então associar atributos de qualidade aos *smells* podem ajudar no processo de gestão destes problemas de *design*, bem como em sua filtragem.

Alguns pesquisadores acreditam que os *smells* devem ser precisamente identificados e abordados. No entanto, é complicado detectá-los por questões como inconsistência e informalidade na especificação, gerando altas taxas de falsos positivos no processo, afetando consequentemente seu sucesso (VASSALLO et al., 2020; MAYVAN; RASOOLZADEGAN; JAFARI, 2020). Então, o interesse associar os *smells* a atributos de qualidade está mais relacionado no entendimento de que algo está afetando negativamente a qualidade e o *design* e que, portanto, precisa ser detectado, priorizado e filtrado para que o time considere ações imediatas ou planejadas no desenvolvimento.

3 TRABALHOS RELACIONADOS: ABORDAGENS E FERRAMENTAS

Neste capítulo, é apresentado as principais abordagens de detecção encontradas na literatura, bem como as ferramentas que implementam estas abordagens. Também, são apresentados as principais abordagens de priorização encontradas até o momento na literatura.

3.1 Abordagens de Detecção

Segundo Lacerda et al. (2020), o tópico mais discutido em estudos secundários selecionados são as abordagens de detecção. Dentre eles, o uso de percepção humana, métricas, regras/heurísticas, análise textual, histórico de código, visualização de software e análise probabilística/*search-based* são as abordagens mais citadas. Estas abordagens são apresentadas, de forma resumida, a seguir.

3.1.1 Percepção Humana

A abordagem baseada na percepção humana (LI; AVGERIOU; LIANG, 2015; SANTOS et al., 2018; SOBRINHO; LUCIA; MAIA, 2018) é uma abordagem manual fundamental para identificar os *smells*, geralmente com base em diferentes orientações seguidas por desenvolvedores para detectar defeitos de *design* manualmente (RASOOL; ARSHAD, 2015; ALKHARABSHEH et al., 2019).

Esta abordagem de detecção usa como base a experiência dos desenvolvedores (MANTYLA; LASSENIUS, 2006; LIU et al., 2007). Este tipo de análise é subjetiva, já que é realizada por humanos em vez do uso de métricas automáticas. Todavia, como estas anomalias possuem definições às vezes complexas e não sendo formalmente descritas, seria plausível imaginar que o entendimento desses problemas por parte de desenvolvedores pode variar, por conta de fatores como experiência, ambiente de desenvolvimento, interpretações diversas, entre outros (MURPHY-HILL; BLACK, 2008).

Travassos et al. (1999) apresentou um processo baseado em inspeções manuais e técnicas de leitura para identificação de *smells*, sem se preocupar com a especificação das anomalias. Porém, não são apresentados tentativas de automatizar o processo e, assim, o modelo não é escalável para grandes aplicações.

Na linha de detecção de *smells*, Fontana, Braione and Zanoni (2012) estudou diferentes ferramentas para detecção automática de *smells* e suas diferenças, especialmente tentando minimizar o efeito das diferenças de interpretação humana que podem definir a existência ou não de um *smell*.

Técnicas manuais são centradas nas pessoas, demoradas e propensa a erros. Essas técnicas eliminam incertezas no processo de detecção devido ao envolvimento humano, mas não são úteis para examinar *smells* em grandes bases de código. No entanto, é importante considerar a participação humana do processo de detecção.

3.1.2 Métricas

A abordagem baseada em métricas (SHARMA; SPINELLIS, 2018; SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019) é usada para detectar inúmeros *smells*, sendo a mais utilizada (LACERDA et al., 2020). As métricas medem propriedades específicas de um software mapeados por números e indicadores, de acordo com regras bem definidas para os objetivos de medição. A medição é usada para descrever, julgar ou mesmo predizer características de um software (LANZA, 2003).

Vários estudos exploraram as métricas em diferentes contextos como complexidade (MCCABE, 1976), orientação a objetos (CHIDAMBER; KEMERER, 1994; LORENZ; KIDD, 1994; SIKET; FERENC, 2004; LANZA; MARINESCU, 2010), coesão/acoplamento (BRIAND; DALY; WÜST, 1997; MARTIN; MARTIN, 2007), arquitetura (KOZIOLEK, 2011), entre outros. As métricas também são usadas como uma abordagem para detecção de *smells* (FENSKE et al., 2015; BIGONHA et al., 2019; LACERDA et al., 2020), predição de falhas (RADJENOVIC et al., 2013) e gestão de dívida técnica (CHARALAMPIDOU, 2019).

As métricas merecem um destaque no apoio a identificação de problemas de *design* do código, pois estão relacionadas e apoiam qualquer uma das formas de detecção descritas. Contudo, quando as métricas são consideradas de forma isolada, elas fornecem informações não muito precisas para detecção *smells* de forma direta e objetiva (MANTYLA; VANHANEN; LASSENIUS, 2004; LACERDA; PETRILLO; PIMENTA, 2020). As métricas por si só não oferecem uma compreensão muito aprofundada do software, nem sugerem alternativas imediatas aos desafios do código. Entretanto, mesmo sem apontar precisamente problemas ou soluções de implementação, as métricas podem servir como um ponto de partida para análises mais apuradas dos fatos extraídos a fim de elevar

o nível de compreensão do software (PARNIN; GORG; NNADI, 2008).

Embora existam inúmeras métricas (SARAIVA et al., 2015) e várias ferramentas que automatizam a coleta das métricas (KAYARVIZHY, 2016; MSHELIA; APEH; EDOGHOGHO, 2017), ainda existem alguns problemas de entendimento e interpretação no que as métricas podem ajudar. Algumas considerações que podem levar a essas interpretações são inconsistências de nomenclaturas (mesmos nomes e diferentes significados para a mesma métrica) e significados similares (diferentes nomes para a mesma métrica), discutidas em mais detalhes por (SARAIVA et al., 2015). Outro fator relevante a considerar é quais métricas são efetivamente utilizadas em situações reais (BOUWERS; DEURSEN; VISSER, 2013).

A precisão das abordagens baseadas em métricas também depende da seleção adequada de *thresholds*[1], que geralmente são empíricos e não muito confiáveis (BANDI; WILLIAMS; ALLEN, 2013; RASOOL; ARSHAD, 2015; ALKHARABSHEH et al., 2019). Ainda não existe um consenso sobre estes *thresholds* para detecção de *smells* e, conseqüentemente, há muita disparidade entre os resultados das diferentes técnicas. Um dos fatores que podem contribuir para esse achado é a falta de padronização e formalização da definição do *smells*.

3.1.3 Regras/Heurísticas

As regras/heurísticas (SHARMA; SPINELLIS, 2018; SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019) combinam regras, expressões lógicas e métricas usados tipicamente para detecção de *smells*. Elas são uma alternativa para solucionar estes problemas com métricas. Também são conhecidas como *estratégias de detecção* (MARINESCU, 2004; LANZA; MARINESCU, 2010). Cada regra é específica para cada *smell* e que podem ser definidas manualmente ou automaticamente usando diferentes técnicas (ALKHARABSHEH et al., 2019).

A *estratégia de detecção* sugerida por Marinescu (2004) formula regras baseadas em métricas que capturam desvios de bons princípios de *design* e heurísticas. A estratégia é baseada em dois mecanismos de filtragem e composição. A filtragem visa detectar fragmentos de *design* com propriedades específicas capturadas por uma métrica. No entanto, a composição é baseada em um conjunto de operadores lógicos usados para associar as métricas uma regra articulada.

[1] valore limite usado na métrica

O principal objetivo dessas estratégias é facilitar o trabalho do desenvolvedor na localização direta destes *smells* em componentes de software. Moha et al. (2010) e Jaafar et al. (2016) propuseram um método chamado DECOR que contempla todos os passos necessários para a especificação e detecção de *smells*, bem como uma ferramenta que implementa este método (DETEX), através do uso de DSL.

Segundo Sharma (2019), esta abordagem expande os horizontes das métricas, oferecendo mecanismos de detecção que podem revelar alta proporção dos *smells*.

No entanto, a abordagem apresenta algumas limitações. A conversão de sintomas em regras de detecção requer análise e esforço de interpretação para selecionar os *thresholds* adequados (FONTANA et al., 2015a), bem como a subjetividade com que as saídas são interpretadas pelos desenvolvedores (FONTANA et al., 2016). Também, ainda não há acordo sobre a definição de sintomas padrão com as mesmas interpretações e, portanto, a precisão da abordagem é baixa (FONTANA; BRAIONE; ZANONI, 2012).

3.1.4 Textual/*Token-based*

Técnicas baseadas em texto tentam encontrar *smells* por meio da análise léxica do código (MARINESCU et al., 2005).

Bavota et al. (2013) apresentaram o *MethodBook*, o primeiro trabalho a combinar informações estruturais e semânticas (ou seja, textuais) para identificar oportunidades da refatoração *Move Method* para o *smell Feature Envy*. Além disso, foi também o primeiro trabalho baseado no *Relational Topic Models* (RTM), um modelo probabilístico hierárquico para representar e modelar tópicos, documentos e relações conhecidas entre eles. A vantagem do RTM é que ele considera o contexto do documento e os *links* entre os documentos, enquanto outras técnicas de modelagem de tópicos, como *Latent Dirichlet Allocation* (LDA) ou *Latent Semantic Indexing* (LSI), consideram apenas as informações textuais dos documentos a ser modelado.

Palomba et al. (2016) apresentaram o TACO (*Textual Analysis for Code Smell Detection*), uma técnica que explora a análise textual para detectar uma família de *smells* (*Long Method, Feature Envy, Blob, Promiscuous Package* e *Misplaced Class*) de diferentes naturezas e níveis de granularidade. Segundo os autores, o TACO apresentou um *recall* de 72% a 84% com *precision* entre 67% e 77%.

Ao invés de manipular textos ou palavras-chave exatas, a abordagem baseada em *tokens* lida com sequência de *tokens* extraídas dos dados de entrada (ABUHASSAN;

ALSHAYEB; GHOUTI, 2020). Em geral, os *tokens* têm mais poder descritivo do que texto simples, obtendo um processo mais preciso. A partir desta sequência formada, diferentes algoritmos como *similarity scoring, fingerprinting* e expressões regulares podem ser aplicadas (RATTAN; BHATIA; SINGH, 2013).

Uma abordagem proposta por Kamiya, Kusumoto and Inoue (2002) usa uma técnica que divide os lexemas e as sequências de *tokens* são comparadas para detectar *Duplicated Code*. Esta técnica é implementada no *CCFinder*, descrito na Subseção 3.2.2.

3.1.5 Histórico de Código

Alguns autores detectaram *smells* usando informações de evolução do código. Esses métodos extraem informações estruturais do código e de mudanças ao longo do tempo, usando estas informações para inferir os *smells* (SHARMA, 2019).

Palomba et al. (2015) propuseram o *Historical Information for Smell deTection* (HIST), uma abordagem para detectar *smells* com base nas informações de histórico de mudanças extraídas de sistemas de controle de versão e, especificamente, analisando alterações que ocorrem no código. A abordagem detecta os *smells Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob* e *Feature Envy*. O HIST depende da descoberta da regra de associação para encontrar classes e métodos que mudam frequentemente no histórico de *commits* do controle de versão. Os autores avaliaram a abordagem em 9 projetos pertencentes ao ecossistema *Apache*, 5 projetos pertencentes as APIs do *Android* e 6 outros sistemas *open-source*, incluindo *Eclipse Core*, *Google Guava, jEdit* e *MongoDB*. Os resultados reportados indicaram que o HIST apresenta *precision* entre 72% e 86% e *recall* entre 58% e 100%.

Fu and Shen (2015) apresentaram uma abordagem usando *evolutionary data mining* para detectar *Duplicated Code, Shotgun Surgery* e *Divergent Change*. Os autores exploraram as regras de associação extraídos do histórico de mudanças, usando algoritmos heurísticos para detecção dos *smells*. O estudo avaliou 5 projetos *open-source* (*Eclipse, Google Guava, jEdit, Clousure Compiler* e *Maven*). Os resultados obtidos foram comparados com o *SonarQube*, obtendo *precision* entre 50% e 100%, *recall* entre 60% e 100% e *F-measure* entre 58% e 100%.

Segundo Sharma (2019), esta abordagem tem aplicabilidade limitada, pois apenas alguns *smells* podem ser detectados. No entanto, a abordagem se torna interessante se combinada com outras, sendo essencial para a gestão da dívida técnica.

3.1.6 Visualização de Software

A abordagem baseada em Visualização de Software (VS) integra a capacidade da experiência humana com o processo de detecção automatizado (GUPTA; SURI; MISRA, 2017; SINGH; KAUR, 2017; SHARMA; SPINELLIS, 2018). A VS está preocupada com a visualização da estrutura, comportamento e evolução do software, utilizando elementos estáticos (extraídos diretamente do código) e/ou dinâmicos (extraídos da execução do software) visando entender como ele está construído e como evolui (DIEHL, 2007). Algumas técnicas usadas para representação das estruturas de código compreendem metáfora de cidades (WETTEL; LANZA, 2008), técnicas de visualização 3D (DHAMBRI; SAHRAOUI; POULIN, 2008), ambientes interativos (MURPHY-HILL; BARIK; BLACK, 2013) e modelo de grafos (PETRILLO et al., 2015). Os dados visualmente representados são complementados com métricas através de metáforas visuais específicas, com o intuito de reduzir a complexidade de entendimento (REIS; CARNEIRO; ANSLOW, 2020).

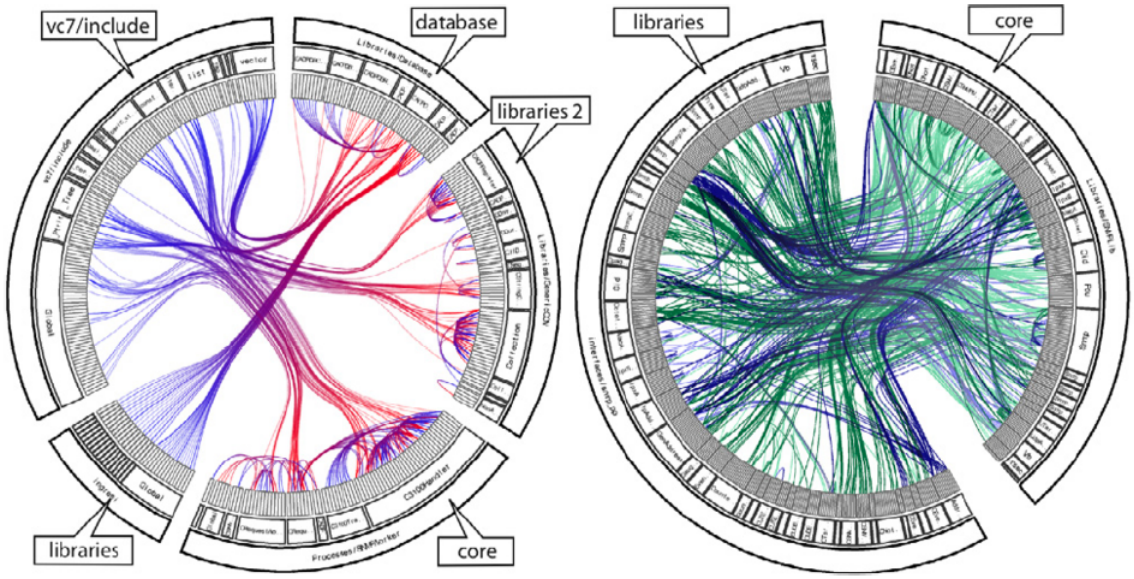
Conforme Storey et al. (2008) e Petrillo, Pimenta and Freitas (2012), existem várias ferramentas de manutenção do código que usam da visualização para revelar informações do software ao desenvolvedor. Estas informações não são obviamente percebidas durante o exame dos artefatos, como é o caso da Figura 3.1, um grafo de chamadas que evidencia o grau de dependência entre diferentes módulos de um sistema. Exemplos mais triviais desse conceito podem ser encontrados em ferramentas de depuração de código, cobertura de testes, marcação de erros e *warnings*, que apresentam ao desenvolvedor formas visuais alternativas, para a interação com o software durante o desenvolvimento.

Emden and Moonen (2002) apresentaram uma abordagem para detecção automática e visualização de *smells* e discutem como esta abordagem pode ser usada no *design* de software, sobretudo, como uma ferramenta de inspeção. Neste trabalho é utilizado a ferramenta *jCosmo*, protótipo proposto pelos autores para detecção automática de anomalias e o software *Rigi* para gerar a visualização.

Parnin, Gorg and Nnadi (2008) destacaram a importância da visualização, principalmente quando o uso de métricas produzem resultados volumosos e imprecisos. Neste cenário, descrevem várias técnicas de visualização de possíveis defeitos reportados por ferramentas de inspeção. Além disso, propuseram um catálogo *lightweight*[2] de visuali-

[2]neste trabalho, os autores procuraram representar alguns *smells* passíveis de visualização. Por exemplo, *Duplicated Code* é um *smell* que ficou fora da lista por necessitar de uma análise mais profunda

Figura 3.1 – Visualização de um grafo de chamadas - Sistema modular (esquerda) *versus* sistema mal projetado (direita)



Fonte: Telea, Byelas and Voinea (2009)

zação para auxiliar revisores na identificação dos problemas. A visualização foi gerada usando uma ferramenta 2D denominada *NosePrints*.

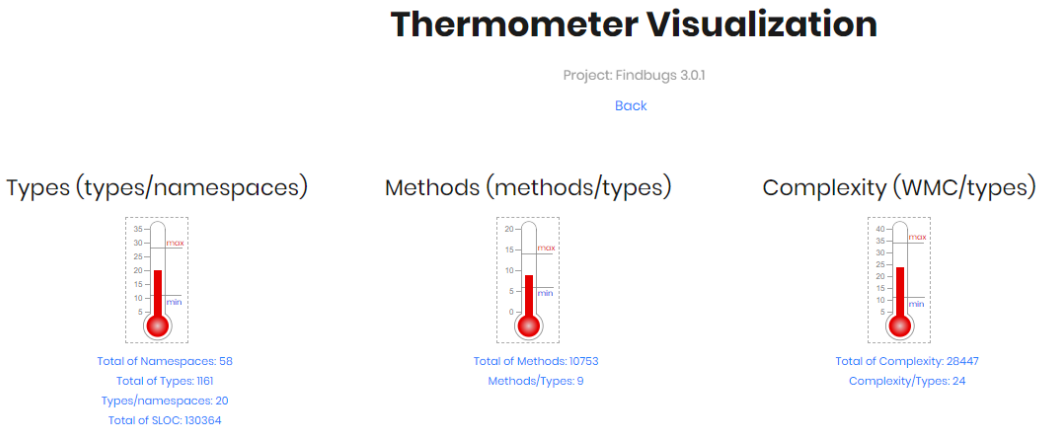
Já Cassell et al. (2010) apresentaram o *ExtC Visualizer*, ferramenta que usa a visualização e a técnica de *clustering* para apoiar os desenvolvedores no entendimento da estruturas de classes, bem como seus respectivos membros e relacionamentos. Com a ferramenta, é possível representar as relações de acoplamento (como a classe usa outras classes ou é usada por outras). Atualmente está limitado a aplicação de *Extract Class*. Outro trabalho (JEFFERSON; WAINER, 2008) usa a técnica de *dotplot* para representar de forma visual alguns *smells* como *Duplicated Code, Long Method, God Class, Shotgun Surgery, Switch Statements, Feature Envy* e *Innapropriate Intimacy*.

No trabalho de Murphy-Hill and Black (2010), é apresentado uma ferramenta de detecção de *smells* denominada *Stench Blossom*, usando um ambiente interativo de visualização para apoiar os desenvolvedores na investigação das anomalias fornecendo subsídios para aplicar as técnicas corretas.

Ghezzi and Gall (2013) também propuseram uma alternativa de análise da evolução da arquitetura do software, que aborda três níveis de análise, integrando a extração de fatos com a identificação de medidas de código e problemas de *design* sinalizados através dos *smells*. Entretanto o grande desafio atualmente é encontrar propostas visuais que melhor representem os problemas de código identificados.

Baseado na metáfora da medicina, Lacerda, Petrillo and Pimenta (2020) introduziram o *DR-Tools Suite*, um conjunto de ferramentas que ajudam os desenvolvedores na análise e entendimento do software. Atualmente, o *DR-Tools Suite* possui duas ferramentas: (1) *DR-Tools Metric*, uma ferramenta CLI[3] que analisa e apresenta métricas contextualizadas e ordenadas (sumário do projeto, pacotes, classes, métodos, dependências e acoplamento) do código, e (2) *DR-Tools Metric Visualization*, uma ferramenta que usa os dados gerados pelo *DR-Tools Metric*, permitindo a visualização em diferentes gráficos contextualizados. O *DR-Tools Suite* usa métricas utilizadas em situações reais e que também possam ser combinadas em cada contexto, permitindo que o desenvolvedores filtrem as informações e façam inferências sobre o código. As ferramentas desenvolvidas recuperam informações do código, fornecendo subsídios para que os desenvolvedores reflitam e aprendam sobre complexidade de software e como direcionar atividades de manutenção e evolução do software. O objetivo é reduzir a sobrecarga cognitiva dos desenvolvedores para analisar os resultados da ferramenta. Na Figura 3.2, é apresentado o termômetro do projeto, com a distribuição de classes, métodos e complexidade, considerando os *thresholds* usados. A Figura 3.3 é apresentada a ressonância do código. Com esta visualização, é possível ver a distribuição das classes dentro dos pacotes e o tamanho das classes (tamanho da bolha). As bolhas em vermelho representam classes que possuem um alto WMC (de acordo com os *thresholds* usados, $WMC > 100$).

Figura 3.2 – Visualização do termômetro do projeto, apresentando informações gerais sobre distribuição média de classes, métodos e complexidade

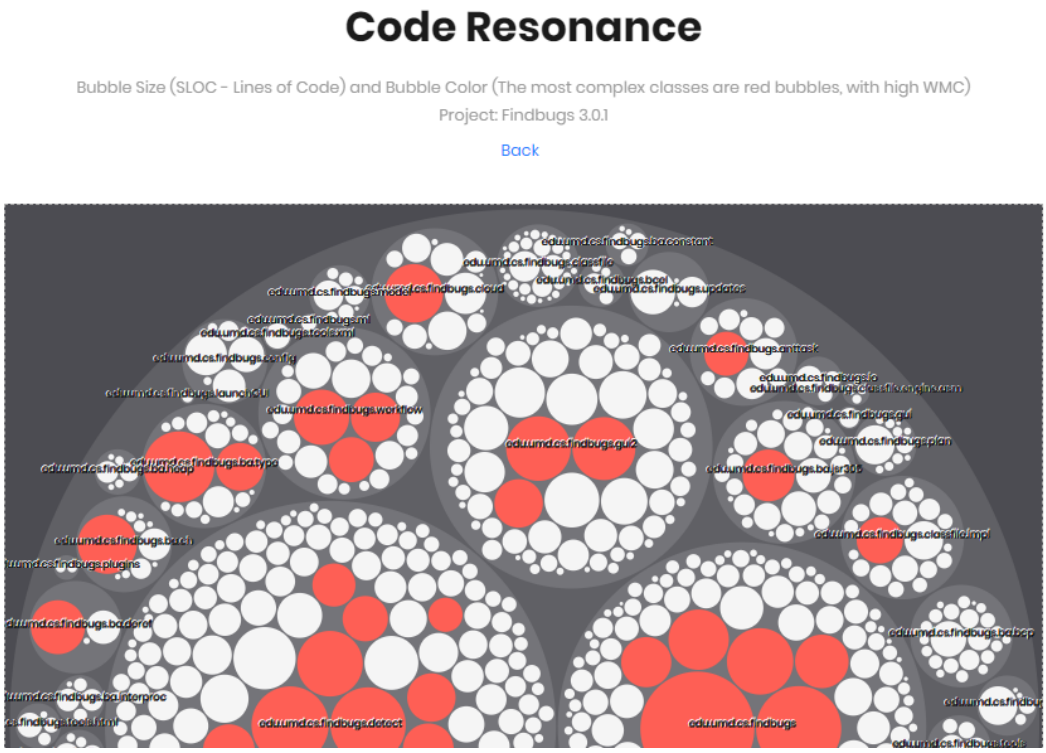


Fonte: Lacerda, Petrillo and Pimenta (2020)

As visualizações podem ser usadas tanto pelo desenvolvedor, individualmente, como pelo time em sessões de discussão técnica ou revisão de código, visando planejar ações de manutenção.

[3] *command-line interface*

Figura 3.3 – Ressonância de código usando *DR-Tools Metric Visualization*



Fonte: Lacerda, Petrillo and Pimenta (2020)

A VS pode apresentar problemas de escalabilidade para grandes sistemas e é sujeita a erros devido ao julgamento humano errado, dependendo do tipo de visualização. No entanto, essa abordagem pode ajudar os desenvolvedores combinada com outras abordagens de detecção, para apoiar os pontos do código a serem melhorados, reduzindo a dívida técnica.

3.1.7 Probabilísticas/*Search-based*

As abordagens probabilísticas/*search-based* (GUPTA; SURI; MISRA, 2017; SINGH; KAUR, 2017; SHARMA; SPINELLIS, 2018) aplicam diferentes algoritmos e regras para detecção de *smells* diretamente do código. A maioria das técnicas nesta categoria aplica algoritmos de *machine learning* (ML) e lógica *fuzzy*. O estudo de Alkharabsheh et al. (2019) reforça o uso da abordagem *search-based* para detectar diferentes tipos de *smells*. Diversas técnicas e algoritmos propostos para extrair regras especificadas para detectar *smells* com técnicas baseadas em algoritmos genéticos e busca heurística.

Khomh, Penta and Gueheneuc (2009) propuseram o uso de um algoritmo *Bayesian Belief Network* (BBN) para detectar instâncias de *God Class* em dois projetos *open-*

source. Os autores envolveram 4 alunos de pós-graduação para validar manualmente um conjunto de classes, relatando se cada classe contém uma instância de *God Class*. A partir de um conjunto contendo 15 instâncias de *smells*, os autores realizaram um procedimento de validação cruzada de três vezes para calibrar a BBN e avaliar seu desempenho na detecção do *God Class*. A BBN foi capaz de detectar todas as instâncias, resultando em um *recall* de 100%. No entanto, o algoritmo classificou incorretamente as classes que não continham *smells* e atingiu uma precisão de 68%. Os mesmos autores estenderam o estudo anterior, aplicando BBNs para detectar três tipos de *smells* (KHOMH et al., 2011). Neste trabalho, os autores seguiram o mesmo procedimento para produzir o conjunto de treinamento que foi utilizado para calibrar e avaliar o BBN. Mais uma vez, as BBNs conseguiram atingir um *recall* de 100%. Por outro lado, apresentaram pior *precision* (quase 33%) na detecção dos *smells* analisados.

O trabalho descrito por Maiga et al. (2012) avaliou a eficiência de uma abordagem baseada em *Support Vector Machine* (SVM) para detectar *smells*. A abordagem proposta foi avaliada por meio de um conjunto de treinamento contendo 250 exemplos validados manualmente por diferentes desenvolvedores. De acordo com os resultados, a abordagem baseada em SVM foi capaz de atingir, em média, um *recall* e um *precision* igual a 70% e 74%, respectivamente.

Amorim et al. (2015) usaram um algoritmo de árvore de decisão para detectar *smells*. Semelhante ao trabalho anterior, os autores usaram um conjunto de treinamento contendo um grande número de exemplos e validado por diferentes desenvolvedores. O trabalho informou que o algoritmo foi capaz de atingir, em média, um *recall* de 71% e uma *precision* de até 78%.

Fontana et al. (2016) apresentaram um estudo maior que compara e experimenta diferentes configurações de seis algoritmos de ML para detecção de *smells*. Para o treinamento, os autores consideraram um conjunto composto por mais de 1900 exemplos de *smells* validados manualmente por diferentes desenvolvedores. Segundo os autores, todas as técnicas avaliadas apresentaram alta acurácia, no qual a maior delas foi obtida por dois algoritmos baseados em árvores de decisão (*J48* e *Random Forest*). Além disso, os autores também relataram que as técnicas necessitavam de 100 exemplos de treinamento para atingir uma acurácia de, pelo menos, 95%.

Hozano et al. (2017a) avaliaram 6 algoritmos de ML na detecção de *smells* para diferentes desenvolvedores, considerando sua percepção individual sobre os *smells*. O estudo comparou experimentalmente os algoritmos *J48, JRip, Random Forest, Naive Bayes,*

Sequential Minimal Optimization (SMO) e *SVM* com os *smells God Class, Data Class, Long Method* e *Feature Envy*, para 40 desenvolvedores. Foi relatado no estudo a falta de capacidade de aprendizado, em razão de um conjunto de dados limitados. Segundo os autores, os resultados mostram uma baixa precisão para os desenvolvedores, evidenciando sua sensibilidade ao tipo de *smell* e ao desenvolvedor.

Em outro estudo, Hozano et al. (2017b) realizaram uma pesquisa sobre o uso de ML para personalizar a detecção de *smells* de acordo com a percepção dos desenvolvedores. O ML foi usado para personalizar as detecções *smells* de acordo com a percepção dos desenvolvedores, de forma não guiada. No entanto, a personalização não guiada pode não ser eficiente, exigindo um esforço considerável para alcançar alta eficácia. O trabalho, através de uma técnica guiada chamada *Histrategy*, busca melhorar a eficiência na detecção de *smells*. Os autores usaram heurísticas que permitiram a personalização e, assim, a abordagem superou 6 algoritmos de ML usados em abordagens não guiadas, tanto em eficácia quanto em eficiência.

Sharma et al. (2019) apresentou um estudo explorando a viabilidade de modelos de *deep learning* para detectar *smells*, investigando a possibilidade de aplicação de *transfer-learning* no contexto de detecção de *smells* em diferentes linguagens de programação. Para isso, foram treinados modelos de detecção baseado em redes neurais, como *Convolution Neural Networks* (CNN) e *Recurrent Neural Networks* (RNN), que são métodos de aprendizado supervisionado de última geração. Foram estudados os *smells Empty Catch Block, Magic Number, Complex Method* e *Multifaceted Abstraction*, porém com diferentes características que misturam abordagens de detecção. Segundo os autores, ambos os modelos podem ser usados para detecção, com desempenho variável, dependendo do *smell* detectado.

Al-Shaaby, Aljamaan and Alshayeb (2020) apresentaram um estudo sistemático do uso de ML para detecção de *smells*. No estudo, foram discutidos 17 trabalhos, abordando 27 *smells* (*God Class, Long Method, Feature Envy* e *Data Class* sendo os mais investigados). No total, 16 algoritmos de ML foram empregados, com destaque para o SVM, como mais investigado e com pior desempenho. No entanto, segundo os autores, *J48* e *Random Forest* superam os demais na detecção de *smells*.

Essas abordagens são capazes de personalizar a detecção *smells* de um conjunto de exemplos usados para treinamento (KHOMH et al., 2011; MAIGA et al., 2012; FONTANA et al., 2016). No entanto, existem algumas limitações como a falta de avaliação da eficiência das técnicas de detecção de *smells* de acordo com a visão de cada desen-

volvedor. Em primeiro lugar, alguns estudos envolveram diferentes desenvolvedores em experimentar as abordagens propostas. Porém, a precisão de tais abordagens não é avaliada e apresentada individualmente para cada desenvolvedor. A precisão é calculada de acordo com um conjunto restrito de exemplos criados por desenvolvedores que foram forçados a compartilhar a mesma percepção sobre alguns *smells*, considerando que os desenvolvedores muitas vezes têm diferentes percepções da presença dos *smells*. Assim, não se sabe se as abordagens propostas são eficientes para detectar *smells* individualmente para desenvolvedores que detectam *smells* de forma diferente. Portanto, as abordagens de ML apresentam baixa precisão (LUJAN et al., 2020). Em segundo lugar, as abordagens propostas exigiram um número alto de exemplos, necessitado um grande esforço e tempo dos desenvolvedores, o que dificulta sua aplicabilidade em projetos reais.

O sucesso destas técnicas depende da qualidade e do treinamento do conjunto de dados (FERNANDES et al., 2016; HOZANO et al., 2017a; SHARMA, 2019). Essas técnicas são limitadas para lidar com definições desconhecidas e variáveis de *smells* de código. Porém, é uma das abordagens a ser mais exploradas em trabalhos futuros, não apenas para detecção de *smells*, mas também para apoiar recomendações de refatorações (LACERDA et al., 2020).

3.1.8 Discussão

A adoção das abordagens de detecção de *smells* ainda é limitada, principalmente ao considerar a priorização (PECORELLI et al., 2020). Um motivo está relacionado a grande quantidade de instâncias combinadas a pouco conhecimento em como priorizar e apoiar nas estratégias de limpeza, como as refatorações. Um outro motivo são as evidências empíricas que mostram como as abordagens de detecção disponíveis identificam *smells* que os desenvolvedores não percebem ou não consideram problemáticos (FONTANA et al., 2016; PALOMBA et al., 2018).

Embora já exista trabalhos sobre taxonomia (MANTYLA; VANHANEN; LASSENIUS, 2003) e categorizações de *smells* (WAKE, 2003), há uma falta de definições padronizadas para detecção de *smells* na comunidade de pesquisa (FERNANDES et al., 2016; GUPTA; SURI; MISRA, 2017; SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019).

O processo de padronização é necessário para permitir a unificação da terminologia e sua definição mais precisa (ABUHASSAN; ALSHAYEB; GHOUTI, 2020). Uma

alternativa possível seria uma norma catalogando todos os *smells* (*design*/código) definidos até o momento, determinando aqueles que se referem ao mesmo *smell* com nomes diferentes. Alkharabsheh et al. (2019) também sugerem a criação de um catálogo único (similar ao *catálogo de Design Patterns*) com uma entrada única no catálogo incluindo *"outros nomes"* ou *" também conhecido como"*. É importante aumentar os esforços para a padronização dos conceitos, o que também permitiria um aumento na consistência na detecção de *smells* (LACERDA et al., 2020).

De acordo com alguns estudos (FERNANDES et al., 2016; SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019), abordagens de detecção de *smells* e seus resultados produzidos correspondentes são altamente inconsistentes. Em geral, abordagens genéricas são usadas para todos os tipos de *smells*, enquanto algumas abordagens específicas são usadas para *smells* mais específicos. Esse é o caso de abordagens como baseadas em histórico e probabilísticas/*search-based*. Sobrinho, Lucia and Maia (2018) relata que diferentes detectores para o mesmo *smell* produzem respostas diferentes, o que é coerente com a necessidade de novas estratégias para identificar *smells* (ou mesmo abordagens) de forma mais eficiente/eficaz do que as abordagens atuais.

Também é necessário explorar se as abordagens podem ser combinadas ou usadas individualmente para a detecção de um conjunto de *smells*. Mumtaz, Singh and Blincoe (2020) relata, *e.g.,* o uso da abordagem baseada em regras combinadas com outras abordagens como *search-based* e visualização.

Diversos motivos justificam a prevalência mais alta de alguns *smells* do que outros (LACERDA et al., 2020): ferramentas disponíveis para sua detecção, a frequência de ocorrência dos *smells*, popularidade entre os profissionais, representatividade nos problemas de código e a incidência de um *smell* em outro.

Muitos trabalhos geralmente consistem em avaliar um determinado *smell* ou mesmo sua relação com outros *smells* (LACERDA et al., 2020). Alguns *smells* aparecem juntos em vários estudos. Por exemplo, os estudos (MISBHAUDDIN; ALSHAYEB, 2015; SINGH; KAUR, 2017; SOUSA; BIGONHA; FERREIRA, 2018; SANTOS et al., 2018) discutem o *Blob* com *Spaghetti Code, Swiss Army Knife, Lava Flow, Functional Decomposition* e *Poltergeist*. Os *smells* de diferentes granularidades também foram estudados em conjunto. Os estudos (CARDOSO; FIGUEIREDO, 2015; CAIRO; CARNEIRO; MONTEIRO, 2018; GRADISNIK; HERICKO, 2018; ALKHARABSHEH et al., 2019) relataram relações entre os seguintes pares de *smells Blob-Data Class* e *Blob-Large Class*. Outros pares de *smells* também foram encontrados: *God Class-God Method*, *God Class-*

Feature Envy, *God Class-Data Class*, *God Class-Duplicated Code*, *Data Class-Data Clumps*, *Divergent Change-Shotgun Surgery* e *Divergent Change-Shotgun Surgery, Feature Envy-Long Method* (CARDOSO; FIGUEIREDO, 2015; ALKHARABSHEH et al., 2019).

Estas avaliações de pares levaram também ao estudo de co-ocorrências dos *smells*. A ausência de abordagens em identificar co-ocorrências entre os *smells* são relatadas em alguns estudos (SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019). Segundo estudos como (CARDOSO; FIGUEIREDO, 2015; SOUSA; BIGONHA; FERREIRA, 2018; SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019), observou-se a co-ocorrência dos *smells God Class/Large Class, Long Method, Feature Envy* e *Duplicated Code/Clones*. Outros *smells* como *Lazy Class, Refused Bequest, Shotgun Surgery, Long Parameter List, Divergent Change* e *Data Clumps* são mencionados nos estudos, mas a relação entre eles não é discutida, sugerindo que ainda é um tópico que merece mais atenção. Os estudos atuais sobre a coexistência dos *smells* indicam uma associação com problemas de manutenção e *design*.

As co-ocorrências podem ser mais exploradas, como o aparecimento de *smells* em consequência de outro *smell*, *smells* que estão sempre próximos (a presença de um implica a presença de outro), entre outros (LACERDA et al., 2020). Isto pode acontecer, inclusive, em diferentes níveis de granularidade (MUMTAZ; SINGH; BLINCOE, 2020). Por exemplo, a presença de um *Long Parameter List* pode resultar em um *Long Method*. A presença do *Long Method*, por sua característica, pode indicar uma *God Class/Large Class*. Portanto, é necessário o desenvolvimento de abordagens de detecção com reconhecimento de co-ocorrência (PALOMBA et al., 2018a). Estas identificações também podem ser úteis para priorização de problemas no código, principalmente considerando não apenas a co-ocorrência dos *smells*, mas também trechos de código mais afetados por diferentes *smells*, independente da granularidade. A Tabela 3.1 resume as principais abordagens e limitações identificadas.

3.2 Ferramentas de Apoio

Atualmente, a detecção manual de *smells* é muito demorada, propensa a erros e cara (LACERDA et al., 2020). Para resolver esses problemas, os pesquisadores desenvolveram muitas ferramentas de detecção de *smells* semi-automatizadas e automatizadas, implementando várias técnicas baseadas nas abordagens apresentadas anteriormente.

Tabela 3.1 – Resumo das abordagens de detecção de *smells*

Abordagem	**Resumo**	**Limitações**
Percepção Humana	considera a experiência dos desenvolvedores e ambientes de desenvolvimento, eliminando incertezas	subjetivo, podendo apresentar diferenças de interpretações; propenso a erros; demorado
Métricas	abordagem mais utilizada; considerada relativamente fácil de implementação, utilizando propriedades identificadas do código	precisão depende dos *thresholds* associados; inúmeras inconsistências, como interpretação e nomenclatura
Regras/Heurísticas	expande a capacidade das métricas; combina operações lógicas e regras; flexível	precisão depende dos *thresholds* associados; subjetivo, passível de diferentes interpretações
Textual/*Token-based*	uso de análise léxica e *tokens* para formar sequências a serem analisadas	aplicável a um número pequeno de problemas de *design*
Histórico de Código	usa o aspecto temporal para analisar mudanças no código	aplicável a um número pequeno de problemas de *design*
Visualização de Software	abordagem semi-automática, que combina percepção humana com visualização	problemas de escalabilidade, dependendo da visualização utilizada; subjetivo, passível de diferentes interpretações
Probabilísticas/*Search-based*	aplicação de diferentes algoritmos ML e lógica *fuzzy*	apresentam baixa precisão para detecção; dependência da qualidade do conjuntos de dados para treinamento dos modelos

Fonte: O Autor

Em geral, as ferramentas investigadas nos estudos (RASOOL; ARSHAD, 2015; FERNANDES et al., 2016; SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019) apresentaram muitas características, resumidas a seguir: a) gratuitas ou comerciais; b) *open-source* ou proprietárias; c) linguagens e plataformas suportadas; d) grau de automação; e) a capacidade de também realizar refatorações; f) maneiras de executar a ferramenta; g) capacidade de gerar métricas; h) formato de entrada e saída; i) formato de interação (CLI, GUI, conectadas a IDEs como plugins) e j) a lista de *smells* que a ferramenta pode detectar.

Lacerda et al. (2020) investigaram as principais ferramentas de detecção de *smells* citadas em estudos secundários, bem como plataformas/linguagens de programação usadas. A seguir, é apresentado um breve relato desses achados.

3.2.1 Plataformas e Linguagens de Programação

De acordo com Lacerda et al. (2020), *Java* é plataforma/linguagem de programação mais usada, seguida de *C++* e *C#*. Segundo o *GitHub*[4], Java é a quarta linguagem de programação mais popular usada nos projetos enquanto de acordo com o *Tiobe Index*[5], *Java* é a segunda linguagem de programação mais popular.

O *Java* tem sido a plataforma mais amplamente utilizada para o desenvolvimento de ferramentas e também como linguagem alvo para os experimentos. Poucas ferramentas (*PMD, Borland Together, CCFinder, inFusion, inCode* e *iPlasma*) listadas nos estudos (RASOOL; ARSHAD, 2015; FERNANDES et al., 2016; SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019) suportam mais de uma linguagem de programação. Portanto, existe a oportunidade de desenvolver ferramentas que suportam mais de uma linguagem de programação. Por exemplo, o *Javascript* é cada vez mais adotado pela indústria e já é líder em projetos FLOSS no *GitHub*. No entanto, ela aparece apenas como a quarta tecnologia mencionada nos estudos referenciados em (LACERDA et al., 2020).

3.2.2 Destaques

Foram identificadas 162 ferramentas para detecção de *smells*, no qual Lacerda et al. (2020) selecionaram as 5 mais citadas para fazer uma breve discussão.

O *CCFinder* é a ferramenta mais citada para detectar *Code Clones*[6] (*e.g., Type-1* e *Type-2*, que são os mais comuns), usando a abordagem baseada em *tokens*. Essa técnica apresenta novos desafios para manutenção de software, refatorações e gerenciamento de *clones* (RATTAN; KAUR, 2016; CAIRO; CARNEIRO; MONTEIRO, 2018; GRADISNIK; HERICKO, 2018) .

O *PMD* é uma ferramenta de análise estática usada para detectar violações de código e más práticas de desenvolvimento. Devido ao seu amplo espectro de ação, detecta também alguns *smells* como *Duplicated Code, Large Class, Long Method* e *Long Parameter List*. O *PMD* aparece entre as ferramentas mais citadas devido ao seu desempenho mais geral, embora, não seja muito precisa (RASOOL; ARSHAD, 2015; FERNANDES et al., 2016).

[4]mais informações em https://octoverse.github.com/, acessada em 15/04/2024

[5]mais informações em https://www.tiobe.com/tiobe-index/, acessada em 15/04/2024

[6]mais informações sobre diferentes tipos de *code clones* estão disponíveis em (SHENEAMER; KALITA, 2016)

O *inCode* é um *plugin* para *Eclipse* que ajuda na detecção de *smells*, identificando *Duplicated Code, Large Class, Feature Envy, Data Clumps* e *Refused Bequest*, com suporte de visualização. Além dos múltiplos *smells*, a ferramenta suporta várias linguagens de programação (FERNANDES et al., 2016; SOBRINHO; LUCIA; MAIA, 2018; ALKHARABSHEH et al., 2019), estando entre as ferramentas de detecção mais comumente citadas.

O *inFusion* é outra ferramenta que detecta *Duplicated Code, Large Class, Feature Envy, Long Method, Data Clumps* e *Refused Bequest*. O *inFusion* foi uma das primeiras ferramentas a ter algum mecanismo de severidade de *smells*. O *inFusion* tem uma versão *open-source* chamada *iPlasma*, que também é citada, mas com funcionalidades mais limitadas. No entanto, infelizmente tanto a *inFusion* quanto a *iPlasma* não estão mais disponíveis.

O *DECOR/DETEX*, proposto por Moha *et al.* (MOHA et al., 2010), é a ferramenta capaz de detectar mais de 10 *smells* (*Large Class/God Class, Lazy Class, Long Method, Long Parameter List, Refused Bequest, Speculative Generality, Message Chains, Shotgun Surgery, Duplicated Code, Comments, Data Class*) identificados por Fowler (FOWLER et al., 1999). Além disso, *DECOR/DETEX* detecta *smells* de *design* propostos por Brown (BROWN et al., 1998), como *Swiss Army Knife, Blob* e *Functional Decomposition*. *DECOR* é um método e *DETEX* é uma ferramenta que nos permite especificar e detectar *smells* de *design* e código, usando uma DSL (SINGH; KAUR, 2017). A ferramenta permite também que os desenvolvedores façam configurações de *thresholds* de métricas para detectar *smells*. O *DECOR/DETEX* está entre as ferramentas de detecção mais citadas, provavelmente devido à natureza leve que permite aos pesquisadores empregá-la para detectar vários tipos de *smells* sem a necessidade de compilar nada a cada vez. É considerado um detector de última geração (PANTIUCHINA et al., 2020).

Poucas ferramentas apresentam recursos de priorização e filtragem. Afinal, não é viável corrigir todos os problemas que surgem no código. Embora existam várias ferramentas de detecção (LACERDA et al., 2020), elas não permitem que os desenvolvedores descubram quais são os mais urgentes de acordo com os seus objetivos. Neste sentido, destacam-se duas ferramentas com recursos de priorização: *JCodeOdor* e *JSpIRIT*.

O *JCodeOdor* é uma ferramenta de detecção de *smells* que usa estratégias de detecção aplicadas em métricas (FONTANA et al., 2015b). Segundo os autores, a ferramenta é capaz de detectar, filtrar e priorizar instâncias dos *smells God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling* e *Message Chains*. O *JCodeOdor*

usa o *Intensity Index*, uma estimativa de severidade de um *smell*, com intervalo entre 1 e 10. A estimativa é calculada com base nas métricas que ultrapassam os *thresholds* das métricas usadas nas estratégias de detecção.

O *JSpIRIT*, ferramenta proposta por Vidal *et al.* (VIDAL et al., 2015) suporta a identificação dos *smells Brain Class, Brain Method, Data Class, Dispersed Coupling, Feature Envy, God Class, Intensive Coupling, Refused Parent Bequest, Shotgun Surgery* e *Traditional Breaker*. A ferramenta também detecta aglomerações do tipo *intra-component, cross-component* e *hierarchical*. O recurso de priorização necessita de algumas informações prévias como definições de módulos e responsabilidades, usadas também na detecções e aglomerações.

3.2.3 Discussão

A maioria das ferramentas apresentadas se concentra na recuperação de *smells* de uma única linguagens, ou seja, na maioria dos casos, a linguagem *Java* (LACERDA et al., 2020).

Um grande número de ferramentas foram desenvolvidas para a detecção de *smells*. No entanto, faltam estruturas de avaliação que poderiam ajudar o usuário na seleção apropriada de qualquer ferramenta para um determinado contexto (RASOOL; ARSHAD, 2015; ALKHARABSHEH et al., 2019). Do ponto de vista da avaliação de ferramentas, a indisponibilidade de implementações dificulta a reprodutibilidade e impõe barreiras aos estudos empíricos subjacentes, em particular para aqueles que objetivam comparar novas abordagens com o estado da arte.

Especialistas da indústria e pesquisadores precisam avaliar os resultados das ferramentas em relação à detecção de falsos positivos e falsos negativos, resultando em um número alto de violações (VASSALLO et al., 2020; LENARDUZZI et al., 2021). Também é essencial ter opiniões dos especialistas sobre priorização de *smells*, impacto dos *smells* na qualidade do produto e dívida técnica (LACERDA et al., 2020). Em geral, de acordo com (RASOOL; ARSHAD, 2015; ALKHARABSHEH et al., 2019), a falta de maturidade das ferramentas e muitas limitações restringem seu uso e adoção pela indústria.

As ferramentas atualmente disponíveis (FERNANDES et al., 2016; SHARMA; SPINELLIS, 2018) podem detectar apenas um número muito pequeno de *smells*. Segundo Lacerda et al. (2020), nenhuma das ferramentas relacionadas detecta todos os 22 *smells*

identificados por Fowler (FOWLER et al., 1999). Em média, as ferramentas cobrem de três a quatro *smells* para detecção (RASOOL; ARSHAD, 2015; FERNANDES et al., 2016). A ferramenta *inFusion* afirma detectar todos os *smells* de Fowler, porém é uma ferramenta comercial, sem versão de avaliação e não estava disponível para experimentos realizados em (RASOOL; ARSHAD, 2015; ALKHARABSHEH et al., 2019). Ainda é um grande problema determinar qual *smell* é eficaz para indicar a necessidade e tipo de refatoração, e o envolvimento do desenvolvedor ainda é necessário (FERNANDES et al., 2016).

A precisão de uma ferramenta de detecção de *smells* é um aspecto fundamental de sua validade. Alguns estudos como (RASOOL; ARSHAD, 2015; FERNANDES et al., 2016; SHARMA; SPINELLIS, 2018; SOBRINHO; LUCIA; MAIA, 2018) revelam que aproximadamente 30% das ferramentas destacam a acurácia, isto é, *precision* e *recall*, de sua técnica ou ferramenta. Além disso, os estudos (RASOOL; ARSHAD, 2015; SOBRINHO; LUCIA; MAIA, 2018) descrevem que os autores de ferramentas de detecção de *smells* realizam experimentos em sistemas diferentes, e a comparação dos resultados publicados torna-se difícil quando as ferramentas não estão disponíveis. Em um estudo sistemático recente, Trindade, Bigonha and Ferreira (2020) fizeram uma pesquisa sobre *datasets* disponíveis sobre *smells*. Segundo os pesquisadores, apenas 12 deles estão disponíveis online. Portanto, os pesquisadores podem se beneficiar destes dados para produzir novos estudos, cruzar dados e avaliar as formas de detecção. No entanto, o número de sistemas presentes nestes *datasets* ainda é considerado pequeno (no máximo, 29 sistemas com projetos variando entre 86 e 17 mil classes). Com isso, este é tópico muito relevante e que requer a atenção da comunidade de pesquisa.

Murphy-Hill, Parnin and Black (2012) apresentaram uma lista de diretrizes como fatores de sucesso relacionados à usabilidade para detectores de *smells*. Alguns estudos (FERNANDES et al., 2016; SOBRINHO; LUCIA; MAIA, 2018) apresentaram uma discussão sobre usabilidade e ferramentas de detecção. Eles definem seis características de análise de ferramentas: a) exportação fácil: os resultados dos *smells* detectados eram facilmente exportáveis, *e.g.,* para texto, CSV ou outros formatos de arquivos; b) destaque de ocorrências de *smells*; c) configuração: permitindo configurações de detecção; d) visualizações de gráficos; e) filtragem de *smells* detectados; f) Análise de múltiplas versões. De acordo com Fernandes et al. (2016), *inFusion* é a única ferramenta que suporta cinco recursos (*a* a *e*), embora dois deles estejam disponíveis apenas na versão comercial completa da ferramenta. Além dos recursos mencionados anteriormente, os estudos

(FERNANDES et al., 2016; SOBRINHO; LUCIA; MAIA, 2018) descrevem que alguns problemas de usabilidade podem prejudicar a experiência do usuário da ferramenta. Entre eles, está a dificuldade de navegar entre as ocorrências de *smells* (em geral, resultados apresentados em longas listas, sem resumo), dificuldade de identificar o código-fonte relacionado à detecção de *smells* e a falta de filtros avançados para detecção de *smells* específicos.

Uma vez que há uma grande variedade de ferramentas e discrepâncias nos resultados das ferramentas, não se pode descartar resultados discrepantes em diferentes estudos porque os estudos estavam usando ferramentas diferentes (LACERDA et al., 2020). Ainda assim, de acordo com Alkharabsheh et al. (2019), apenas algumas ferramentas podem analisar projetos de grande porte, com milhões de linhas de código. A maioria das ferramentas não levou em consideração o *feedback* dos especialistas ou outras características como a influência do contexto, domínio e status do projeto.

Os resultados apontados por Oizumi et al. (2017) indicam que os desenvolvedores precisam de um melhor suporte de ferramentas para analisar códigos com múltiplos *smells* no mesmo trecho de código. Martins et al. (2020) comentam que apenas ocorrências individuais de *smells* podem ser detectadas automaticamente, enquanto as co-ocorrências são detectadas manualmente. Estes resultados estão também de acordo com Palomba et al. (2018a), indicando que a monitoria de co-ocorrências que acontecem frequentemente podem ser exploradas para construir ferramentas de detecção mais precisas, capazes de localizar e classificar a gravidade dos problemas de *design* que afetam o código.

Em geral, as ferramentas não fornecem visualizações de dados por meio de análises estatísticas, contadores de resultados de detecção ou apresentação de resultados em formato gráfico. Os desenvolvedores de ferramentas precisam levar em consideração, na construção das ferramentas, sua adoção dentro do contexto e conectadas ao fluxo de trabalho de desenvolvimento dos times (MARCILIO et al., 2020). Além disso, no geral, a maioria não apresenta recursos de priorização para os *smells*. Identifica-se, portanto, uma oportunidade para o desenvolvimento de ferramentas que contemple tais características.

3.3 Priorização de *Smells* no Código

Muitas vezes o foco da priorização está relacionado com a aplicação de refatorações (MAYVAN; RASOOLZADEGAN; JAFARI, 2020). Como discutido por vários autores (SZOSKE et al., 2016), decidir quando e onde aplicar refatorações não é algo

trivial, justamente porque a base de código de um sistema, frequentemente, sofre grandes mudanças fazendo com que os profissionais percam o controle sobre problemas de *design* existentes. Neste sentido, há a necessidade de técnicas que efetivamente apoiem os desenvolvedores em tarefas como identificação, priorização, gestão e remoção destes problemas (LIM; TAKSANDE; SEAMAN, 2012; ERNST et al., 2015).

Como o grande o número de *smells* detectados por ferramentas é um dos desafios cruciais (MARCILIO et al., 2020), vários estudos foram realizados para priorizar e filtrar os resultados de detecção de *smells*. Além disso, algumas abordagens que foram propostas usando não apenas um, mas vários critérios para priorização de *smells*.

Fontana et al. (2015b) introduziram o *Intensity Index* como um critério para priorização de *smells*. O índice é calculado pela distribuição de métricas de software, usado para quantificar a importância de cada instância de *smell*. Eles representaram o índice como um valor numérico entre 1 a 10, com cinco níveis de intensidade correspondentes a cada intervalo: *very low, low, mean, high* e *very high*. A abordagem dedica mais atenção aos casos mais graves, limitadas a 6 tipos de *smells* (*God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling* e *Message Chains*). A técnica foi implementada na ferramenta *JCodeOdor*. Os autores usaram as estratégias de detecção, definindo algumas novas métricas para identificar as características dos *smells*.

Choudhary and Singh (2016) forneceram uma abordagem para identificar classificações de classe com base em *smells* existentes e correção máxima estimada de *smells*, dada a solução mínima de refatorações. Primeiramente, são removidas as classes que sofreram menos refatoração e, assim, descobriram a relevância da arquitetura entre as classes restantes. Com esta informação, é definida a classificação para realizar as refatorações nas classes altamente propensas. A abordagem resultou na economia considerável de esforço de refatoração, mas exigiria replicação em diferentes sistemas e linguagens para confiabilidade.

Outra abordagem proposta para priorizar *smells* foi o uso do *Context Relevance Index* (SAE-LIM; HAYASHI; SAEKI, 2016). Os autores usaram um conjunto de problemas no contexto do desenvolvedor coletados de um sistema de *issue tracking* para priorizar *smells*. Na avaliação, usaram 4 sistemas *open-source* (*ArgoUML, JabRef, jEdit* e *muCommander*). A técnica apresentou uma melhor classificação para nível de classe, mas falhou em 2 sistemas no nível de método, sem relevância estatística. Em uma evolução deste trabalho (SAE-LIM; HAYASHI; SAEKI, 2018b), os autores também propuseram uma técnica de priorização usando a combinação de severidade dos *smells* e o *Context*

Relevance Index. No entanto, neste trabalho, não foi considerado o contexto dos desenvolvedores.

Arcoverde et al. (2013) propuseram uma técnica para priorizar *smells* com base em seu potencial de degradação da arquitetura. Eles usaram quatro heurísticas: densidade de mudança, densidade de erro, densidade da anomalia e função de arquitetura. Os autores concluíram que sua técnica é útil principalmente em cenários como problemas de arquitetura em classes que sofreram modificações juntas, comunicação entre classes diferentes, mudanças que não são predominantemente perfectivas, entre outras.

Vidal, Marcos and Díaz-Pace (2016) apresentaram uma abordagem semi-automatizada para priorizar *smells*, com base em três critérios: informações históricas de modificações de componentes, a relevância do tipo *smell* da perspectiva do desenvolvedor e cenários de modificabilidade do sistema. A técnica prioriza *smells* com base no contexto de desenvolvedores usando cenários de modificabilidade. No entanto, tais informações requerem experiência e conhecimento do sistema, também como um trabalho manual especificando cenários e componentes de código relacionados. A técnica também requer algumas informações relacionadas a histórico de alterações do sistema.

Codabux and Williams (2016) apresentaram uma técnica para priorizar dívida técnica usando análise preditiva. Os autores primeiro consideraram as classes que são mais propensas a defeitos e a mudanças como um item de dívida técnica. Em seguida, eles calcularam a propensão da dívida técnica de cada classe usando um modelo de previsão que foi construído com uma abordagem *bayesiana*. Os itens são categorizados em grupos baixo, médio e alto usando o modelo de previsão. Finalmente, com a decisão dos gerentes de projeto ou desenvolvedores, a técnica gera itens de dívida técnica com a probabilidade de ser mais crítica, exigindo uma decisão manual dos desenvolvedores para apresentar os resultados.

Malhotra and Singh (2020) realizaram um estudo focado na aplicação de refatorações, baseado em classes severamente afetadas para melhorar a qualidade. Assim, propuseram uma estrutura que detecta um subconjunto de classes que necessitam de refatorações. Para isso, usaram a gravidade dos *smells* e características OO para compor o *Quality Depreciation Index Rule* (QDIR). Usaram para avaliação 10 projetos *open-source*, 10 *smells* (*Long Method, God Class, Feature Envy, Type Checking, Nested Try Statement, Empty Catch Block, Shotgun Surgery, Refused Bequest, Brain Method* e *Intensive Coupling*) e as 6 métricas CK.

Pecorelli et al. (2020) desenvolveram um estudo preliminar com a priorização ori-

entada à percepção dos desenvolvedores, usando ML para classificar os *smells* com a crítica atribuída pelos desenvolvedores. Foram estudados *Blob, Complex Class, Spaghetti Code* e *Shotgun Surgery*. Para coletar as instâncias, os pesquisadores usaram o DECOR (MOHA et al., 2010) e o HIST (PALOMBA et al., 2015). Embora eles tenham testado vários algoritmos de ML, o *Random Forest* foi o que apresentou melhor resultado, com um *F-measure* entre 72% e 85%, ou seja, preciso na classificação percebida dos *smells*. Alguns trabalhos tentaram fornecer soluções que podem ser classificadas como instâncias de *smells* avaliando sua severidade, calculada com base em métricas. Porém, isso pode não ser suficiente, pois não leva em consideração a percepção dos desenvolvedores (PECORELLI et al., 2020).

3.3.1 Filtragem

Além da priorização dos *smells*, outros estudos foram conduzidos para filtrar os resultados da detecção dos *smells* para aumentar a precisão das ferramentas existentes.

Rapu et al. (2004) usou informações históricas para medir a estabilidade de cada *smell* e filtrar as instâncias que podem não ser prejudiciais ao sistema. A estabilidade está relacionada a frequência com que uma classe é alterada (pelo menos um método adicionado ou removido) e a persistência, que indica por quanto tempo a classe foi afetada por *smells*. Em seguida, os resultados foram usados para filtrar as classes que podem não afetar adversamente os resultados originais detectados usando uma estratégia de versão única. A técnica é limitada a *God Class* e *Data Class*.

Fontana, Ferme and Zanoni (2015a) consideraram o domínio de aplicação do sistema para calcular filtros fortes e fracos. Eles usaram um filtro forte para remover *smells* que são falsos positivos dos resultados de detecção e um filtro fraco para identificar instâncias de *smells* que provavelmente não seriam problemáticos.

Sadowski, Stolee and Elbaum (2015) desenvolveram um estudo de caso sobre como os desenvolvedores filtram e fazem pesquisas no código. Em função do crescente tamanho dos repositórios de código, atividades de pesquisa e filtragem estão cada vez mais comuns no desenvolvimento de software. Segundo o estudo, os desenvolvedores utilizaram este recurso, pelo menos, 5 vezes por dia. Entre as razões destacadas, o uso de filtros para localizar código ajuda nas atividades de rastreamento de trechos problemáticos, avaliações de conexões/dependências e impacto em mudanças futuras. Assim, os recursos de filtragem parecem afetar diretamente a produtividade dos desenvolvedores.

Em outro trabalho, Sadowski et al. (2018) apresentaram um estudo sobre adoção de ferramentas de análise de código no *Google*. Segundo os autores, o grande número de falsos positivos relacionados aos problemas, juntamente com a falta de recursos de filtragem, torna o processo insustentável em larga escala. Assim, a filtragem deve apoiar a priorização para eliminar possíveis sujeiras de informação.

Marcilio et al. (2019) realizaram um estudo abrangente sobre violações reportadas pelo *SonarQube* com regras customizadas. Foram minerados quase 422 mil *issues* reportadas em 246 projetos em 4 instâncias do *SonarQube*: dois de ecossistemas *open-source* (*Eclipse* e *Apache*) e 2 projetos reais (Tribunal Contas da União e Polícia Federal, ambos no Brasil). O estudo apresentou uma baixa taxa (em torno de 13%) de resolução de problemas. Em média, os desenvolvedores corrigem os problemas em 19 dias, sendo que quase 1/3 delas acontecem após um ano do seu relato. Assim, os autores conjecturaram que apenas um subconjunto dos verificadores revelaram falhas reais de *design*, o que reforça a necessidade de priorização e filtragem.

Vassallo et al. (2020) desenvolveram um estudo para investigar o uso de ferramentas de análise estática (ASATs) em diferentes contextos como atividades de desenvolvimento regulares, revisão de código ou vinculados no processo de integração contínua. Segundo os pesquisadores, essas perspectivas são essenciais para melhorar a priorização dos *warnings* identificados, seja na detecção de defeitos ou de problemas de *design*. Foi realizado, inicialmente, um *survey* com 56 desenvolvedores, compostos de 66% da indústria e 34% de projetos *open-source*, além de entrevistar 11 especialistas industriais para compreender como é usada as ASATs nestes cenários diferentes. Além do *survey*, foram inspecionados manualmente contribuições de 176 projetos *open-source*. Dentre os vários resultados do trabalho, é destacado a severidade como fator mais importante e que deve ser levado em conta na seleção dos *warnings*. Outro ponto destacado é a necessidade de uma nova geração de ASATs capazes de melhorar a experiência de uso dos desenvolvedores, permitindo a seleção dos *warnings* mais dependente do contexto, principalmente pela definição de filtros e mecanismos de priorização.

3.3.2 Discussão

Faltam estudos empíricos de como os desenvolvedores filtram e priorizam os *smells* (SAE-LIM; HAYASHI; SAEKI, 2018a).

Existem inúmeras limitações relacionadas como falta de alta precisão para en-

contrar esses problemas de *design*. Entre elas estão o alto número de falsos positivos, a ausência de estratégias para filtrar e classificar os problemas não apresentando um resultado consistente em diferentes projetos de software (VIDAL et al., 2019). Embora muitas abordagens tenham sido propostas, elas usam fatores amplamente variáveis para filtrar e priorizar os *smells*.

Outro ponto levantado por Oizumi et al. (2017) indica que os desenvolvedores precisam priorizar co-ocorrências com maior probabilidade de indicar um problema de projeto. Os algoritmos de priorização são necessários porque a análise de código com múltiplos *smells* é difícil e demorada. Assim, os desenvolvedores devem se concentrar nas co-ocorrências que provavelmente indicam mais problemas de *design*. Já Verma, Kumar and Verma (2021) afirma que é necessário levar em consideração o conhecimento dos desenvolvedores, tanto nos parâmetros considerados quanto nos efeitos da qualidade do software.

Em estudos secundários recentes sobre o tema, foram listadas algumas considerações importantes a cerca da priorização de *smells*. Kaur et al. (2021) analisaram 23 estudos sobre priorização de *smells* publicados até maio/2020, sendo o primeiro trabalho secundário sobre o tema. Já Verma, Kumar and Verma (2023) selecionaram 39 estudos publicados entre 2011 e 2022 e os classificaram em sete fatores: (i) *code smells*, (ii) fatores, (iii) técnicas de avaliação, (iv) ferramentas, (v) fórmulas de classificação, (vi) conjunto de dados utilizados e (vii) medições usadas para validação. Ambos os trabalhos apontam vários desafios em relação a este campo de pesquisa. A seguir, é apresentado uma lista combinada destes desafios:

- A maioria das abordagens de priorização implementadas não estão disponíveis de forma gratuita (são pagos ou não estão disponíveis para compra). Assim, os pesquisadores precisam propor técnicas disponíveis gratuitamente para que outros pesquisadores avaliem os resultados com mais facilidade;
- Alguns *smells* têm ganho mais atenção dos pesquisadores. Porém, outros *smells* também podem apresentar impacto negativo na qualidade e poderiam ser explorados em pesquisas futuras;
- Os pesquisadores se concentraram mais em projetos FLOSS para avaliação das abordagens de priorização. Embora estes sistemas estejam disponíveis gratuitamente permitindo que se replique os estudos, ainda é necessário prestar a atenção em projetos industriais, visando testar estas abordagens em um conjunto de dados real;

- Conjunto de dados de pequeno e médio porte tem chamado mais a atenção dos pesquisadores. Assim, a precisão dos resultados também pode ser testada em grandes conjuntos de dados;
- Apenas alguns pesquisadores usaram testes estatísticos para avaliar suas abordagens;
- A maioria dos estudos estão relacionados a tecnologia *Java*, tanto para desenvolvimento de ferramentas quanto para avaliações de projetos;
- Pesquisadores se concentraram mais nos *smells* definidos por Fowler et al. (1999) do que em outros *smells* para estudos relacionados a priorização, sendo um ponto a ser explorado;
- Pesquisadores e desenvolvedores de ferramentas não percebem realmente os fatores ou critérios que devem ser considerados na filtragem e priorização de *smells*. Os fatores usados pelos profissionais precisam ser considerados ao propor novas técnicas.

Portanto, apresentam-se vários desafios que podem direcionar esta área de pesquisa. Assim, as evidências empíricas dos estudos devem ser capazes de fornecer uma oportunidade de apoiar este avanço. Além disso, há uma carência de validação das abordagens no contexto industrial (ALFAYEZ et al., 2020). A Tabela 3.2 apresenta um resumo dos estudos específicos sobre priorização e suas limitações.

Tabela 3.2 – Resumo das abordagens de priorização propostas

Autores	**Critérios**	**Abordagem**	**Ferramentas**	**Limitações**
Fontana et al. (2015b)	*Intensity Index*, baseado na importância do *smell* (1)	dedica atenção aos mais graves, considerando 6 *smells*. Usaram estratégias de detecção	desenvolveram o *JCodeOdor*	não consideraram a percepção dos desenvolvedores; não consideraram co-ocorrências; experimentos não alcançaram uma validação experimental adequada
Choudhary and Singh (2016)	classificação com base na correção máxima dos *smells* (1)	identifica classificações de classes com base nos *smells* existentes, considerando a correção máxima estimada deles com a solução mínima de refatorações	usaram *Organic, RefFinder, JSpIRIT, inFusion*	falta de testes de maior magnitude; falta de testes em projetos industriais; poucos *smells* considerados; sem participação dos desenvolvedores; não é uma abordagem automatizada
Sae-Lim, Hayashi and Saeki (2016)	*Context Relevance Index*, considerando o contexto dos desenvolvedores (1)	classificação baseada na relevância das tarefas, extraídas de um sistema de *issue tracking*, com uma análise de impacto nos problemas, antes de qualquer refatoração	*inFusion* e ferramenta construída no trabalho (sem maiores detalhes)	falta de avaliação da técnica de priorização e utilidade para os desenvolvedores; não foi considerado a severidade e a importância dos problemas
Arcoverde et al. (2013)	densidade de mudança, densidade de erro, densidade da anomalia e função arquitetural (4)	técnica para priorizar *smells* com base na degradação arquitetural	*Findbugs, SCOOP*	não foi testada em projetos industriais; técnica com aplicabilidade em cenários que envolvam mudanças; os critérios foram analisados individualmente e não combinados
Vidal, Marcos and Díaz-Pace (2016)	informação histórica das modificações dos componentes, relevância do *smell* e cenários de modificabilidade (3)	técnica que usa os cenários de modificabilidade para priorização	desenvolveram o *JSpIRIT*	requer trabalho manual para especificação dos cenários e histórico de alterações; não é escalável; aborda aglomeração sem mostrar a relação da mesma
Codabux and Williams (2016)	classes que são mais propensas a efeitos e mudanças (2)	técnica que usa a propensão da dívida técnica de cada classe usando um modelo *bayesiano*	*SCITool Understand, Jira, Apache Hive*	requer decisão manual dos resultados; não escalável
Malhotra and Singh (2020)	*Quality Depreciation Index Rule* (QDIR) baseado em *smells* graves e métricas OO (2)	abordagem focada em refatoração, priorizando classes afetadas por *smells*	*JSpIRIT, JDeodorant, SCI Understand, Robusta Metric*	não leva em consideração co-ocorrências; falta de avaliação industrial; não escalável
Pecorelli et al. (2020)	criticidade conforme percepção dos desenvolvedores (1)	uso de ML para priorização usando a criticidade percebida pelos desenvolvedores, coletado através de feedback	várias ferramentas como *DECOR, HIST, WEKA, JDeodorant, PyDriller*	consideraram um único tipo de granularidade (classe); conjunto pequeno de *smells*; grande conjunto de dados para treino; falta de avaliação industrial

Fonte: O Autor

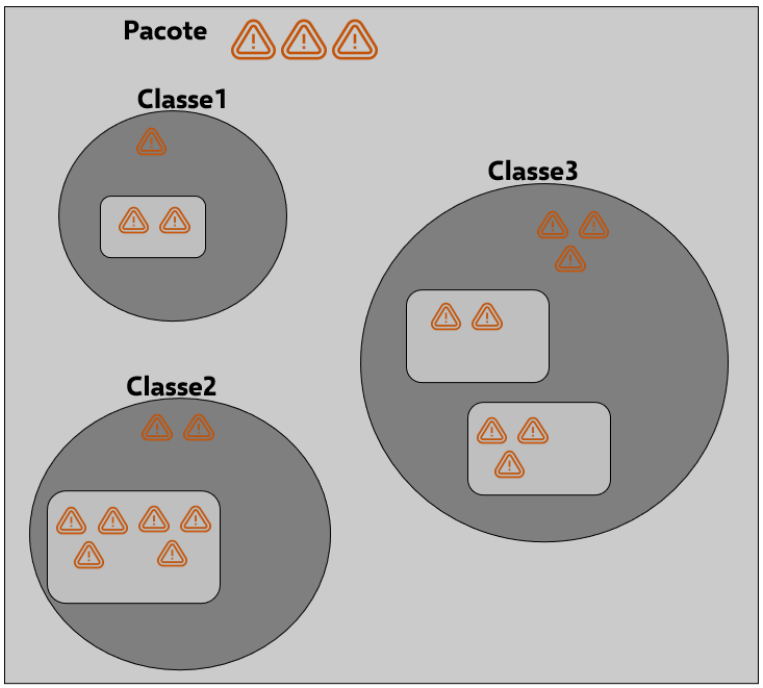
4 *DR-TOOLS CODE HEALTH* - ABORDAGEM PROPOSTA PARA PRIORIZAÇÃO DE *SMELLS*

Este capítulo apresenta a abordagem proposta, contextualizando o modelo e suas implicações. Também é apresentado mais detalhes de sua implementação em uma ferramenta *open-source*, bem como sua aplicabilidade.

4.1 Priorização de *Smells*: desafios

Conforme discutido anteriormente nos Capítulos 2 e 3, os *smells* em um projeto de software podem se manisfestar de diferentes formas e em diferentes momentos. A Figura 4.1 apresenta uma estrutura de um projeto, com potenciais *smells* (representados pelos marcadores em laranja), localizados em diferentes níveis de granularidade.

Figura 4.1 – Potenciais *smells* (em laranja) em um projeto de software, em diferentes granularidades - pacotes (retângulo mais externo), classes (círculos internos) e os métodos (retângulos internos no círculo)



Fonte: O Autor

Assim, é necessário apresentar algumas considerações:

- ***Smells* têm diferentes severidades:** Em um projeto de software, é possível encontrar diferentes tipos de problemas de *design*, com diferentes níveis de severidades.

A severidade está relacionada a criticidade de um elemento de código[1], influenciando negativamente a manutenção e evolução. Alguns elementos de código são mais problemáticos que outros, inclusive com *smells* de mesma instância;

- ***Smells* têm diferentes granularidades:** Os problemas de *design* também são localizados em diferentes níveis de granularidade. Os problemas podem estar presentes nos métodos, classes e pacotes/módulos que compõe o sistema. Ainda, os problemas em um determinado nível podem influenciar o outro nível. Segundo Palomba et al. (2018a) e Sharma, Fragkoulis and Spinellis (2017), problemas de *design* no nível de método pode ser a causa raiz da inserção de problemas no nível de classes. Consequentemente, os problemas no nível de classe tem uma forte correlação positiva com problemas no nível de pacotes (SHARMA, 2019; SHARMA; SINGH; SPINELLIS, 2020);
- ***Smells* são acumulativos:** Um projeto de software é formado por estruturas autocontidas (módulos/pacotes, que contém classes, que contém atributos e métodos). Por este motivo, um aspecto a ser considerado é a presença de mais de um *smell* em um elemento específico de código (*e.g.,* em um método), conhecido como co-ocorrência. Ainda, além da co-ocorrência, é importante considerar o acúmulo destes problemas nos diferentes níveis de granularidade, ou a composição das co-ocorrências;
- ***Smells* afetam diferentes atributos de qualidade:** Conforme discutido na Seção 2.3, não só os atributos de qualidade afetam vários problemas de *design*, mas também os problemas de *design* podem afetar mais de um atributo de qualidade. Um atributo de qualidade está relacionado a um problema se a estrutura de definição do problema contém o mapeamento entre o problema e os atributos de qualidade (YAMASHITA, 2015; LACERDA et al., 2020);
- **Dependendo da granularidade observada, muda a percepção de severidade:** Modelos em diferentes níveis de granularidade devem ser o foco de novas investigações (ABUHASSAN; ALSHAYEB; GHOUTI, 2020). Assim, ao observar a granularidade, é possível que a percepção de severidade mude. Por exemplo, ao olhar na granularidade de métodos, o método mais problemático e, portanto, prioritário para avaliação é um. Ao mudar a granularidade para classes e, assim, ter uma composição de todos os *smells* dos métodos e os *smells* presentes na classe, esta

[1] são os principais componentes presentes em um projeto de software, autocontidos, como pacotes, classes e métodos

percepção tende a mudar;

- ***Smells* possuem diferentes níveis de intervenções:** Ao analisar os *smells*, é importante ter uma noção daqueles que exigem uma intervenção imediata, intervenção planejada ou até mesmo que possam continuar no repositório, pois não afetam o projeto.

Visando contemplar todas estas considerações, o presente trabalho propõe um modelo de priorização para *smells* para apoiar atividades de desenvolvimento e manutenção de software, ajudando os desenvolvedores na classificação e seleção dos elementos de código mais afetados pelos *smells*.

4.2 Modelo Proposto

O modelo proposto visa integrar informações de definição, identificação, priorização e filtro dos *smells*. A forma de priorização proposta é inspirada no *Método de Hanlon* (MH), sobretudo pelos critérios severidade, representatividade e intervenção[2] (NEIGER; THACKERAY; FAGEN, 2011). Na Tabela 4.1, é apresentado este mapeamento dos critérios adotados pelo MH e os critérios adotados pelo modelo proposto para os *smells*.

Além dos critérios adaptados, também é necessário descrever em mais detalhes os termos adotados na definição do modelo (Tabela 4.2). Seguindo a nomenclatura de um trabalho anterior (LACERDA; PETRILLO; PIMENTA, 2020) para elementos de código, é adotado o termo *namespace* para representar pacote, *type* para representar classe e *method* representando método.

A estrutura do modelo permite que atributos de qualidade sejam associados com os *smells*, para apoiar a priorização e filtragem. Além disso, o modelo permite que pesos sejam atribuídos aos atributos de qualidade como indicado por Wagey, Hendradjaya and Mardiyanto (2015). Outros pesos também foram inseridos para refletir a percepção dos desenvolvedores em relação ao seu contexto de projeto. A Tabela 4.3 apresenta os pesos que podem ser definidos pelos desenvolvedores e utilizados na computação da priorização. Para a definição dos pesos utilizados como valores *default*, foram utilizados 3 critérios: análise empírica (SEDGEWICK; WAYNE, 2011), de sensibilidade (SALTELLI et al., 2008) e experiência prática/intuição (BROOKS, 1995). Também, para se chegar

[2]A WHO propõe um trabalho coletivo de intervenções em doenças de caráter global, considerando fins estatísticos. Mais informações em https://www.who.int/standards/classifications/international-classification-of-health-interventions. Mais informações sobre o MH podem ser obtidas no Apêndice A.

Tabela 4.1 – Mapeamento dos critérios definidos para software e o *Método de Hanlon*

Critérios Adaptados para Software	**Critérios do Método de Hanlon**
Severidade - define o quão grave é um *smell* e, consequentemente, um elemento de código afetado por vários *smells*	*Severidade* - A severidade no contexto da saúde é referente a criticidade da doença para a população (o problema é considerado grave? Qual o impacto na vida das pessoas? É uma demanda pública? Exige um alto grau de hospitalização?)
Representatividade - instâncias de um mesmo *smell*, recorrente em diferentes elementos de código, representando seu impacto na granularidade considerada	*Magnitude* - dimensão do problema frente a uma comunidade (Afeta grande parte da população ou alguma comunidade específica?)
Intervenção - define o grau de envolvimento dos desenvolvedores para o *smell* que afetam os elementos de código, considerando o alvo (*e.g.,* uma classe) e a ação necessária (imediata, planejada ou futura)	*Eficácia/Intervenção* - Ato a ser considerado para avaliação, melhoria, manutenção, promoção ou modificação do funcionamento e condições da saúde (o problema é de fácil resolução? Existem os meios necessários para a ação?)

Fonte: O Autor

nos valores *default*, foi utilizado um *dataset* de 50 projetos (Apêndice B), de diferentes tamanhos e propósitos.

Assim, o modelo combina aspectos quantitativos (*e.g.,* quantidade de *smells* em um elemento de código), com aspectos qualitativos (efeito de um *smell* na qualidade) e a percepção dos desenvolvedores, com a definição de pesos em outros dos critérios considerados. A Figura 4.2 apresenta os elementos integrantes deste processo, contemplados pelo modelo. De forma mais geral, a partir dos elementos de código que possuem *smells* (granularidades de pacote, classe e método), o modelo utiliza os critérios como severidade, representatividade, impacto na qualidade e intervenção, combinando pesos (importância do *smell*, grau de intervenção, impacto nos atributos de qualidade) definidos pelo time para priorizar os elementos de código mais problemáticos.

4.2.1 Detalhamento do Modelo

Uma visão resumida do modelo proposto é apresentada na Figura 4.3. Algumas regras foram usadas para a definição do modelo, levando em consideração os termos

Tabela 4.2 – Termos adotados no modelo

Termo	**Descrição**
Smell	Termo adotado para representar problemas de *design* no código, independente da granularidade. Mais detalhes sobre sua definição podem ser obtidas na Seção 2.1
Heurística do *Smell*	Definição da forma de detecção do *smell*, representando uma regra de detecção ou algo mais elaborado, como um algoritmo para sua detecção
Elemento de Código	Pode representar pacote(s), classe(s) e/ou método(s) de um sistema
Granularidade	Representa o nível do elemento de código, considerando todas as instâncias de *smell* do contexto (*Namespace, Type* ou *Method*)
Diversidade	Representa a quantidade de *smells* em um mesmo elemento de código
Severidade	Representa a criticidade de um *smell*, descrita em mais detalhes na Seção 4.2.2
Representatividade	Representa o quão significativo é um *smell* na granularidade correspondente (mais detalhes na Seção 4.2.2)
Threshold	Limite definido para a métrica que pode compor a heurística do *smell*
Atributos de Qualidade	São os atributos (internos e externos) comumente utilizados no campo da Qualidade de Software, mais especificamente relacionados a manutenibilidade e associados na definição um *smell*
Intervenção	Define a importância a ser considerada na priorização de um *smell*
Code Disease Indicator (CDI)	É um valor agregado que define a criticidade total de um elemento de código
Composição	Representa o acúmulo de CDIs dos *smells* nos elementos de código, considerando o contexto atual e granularidades inferiores

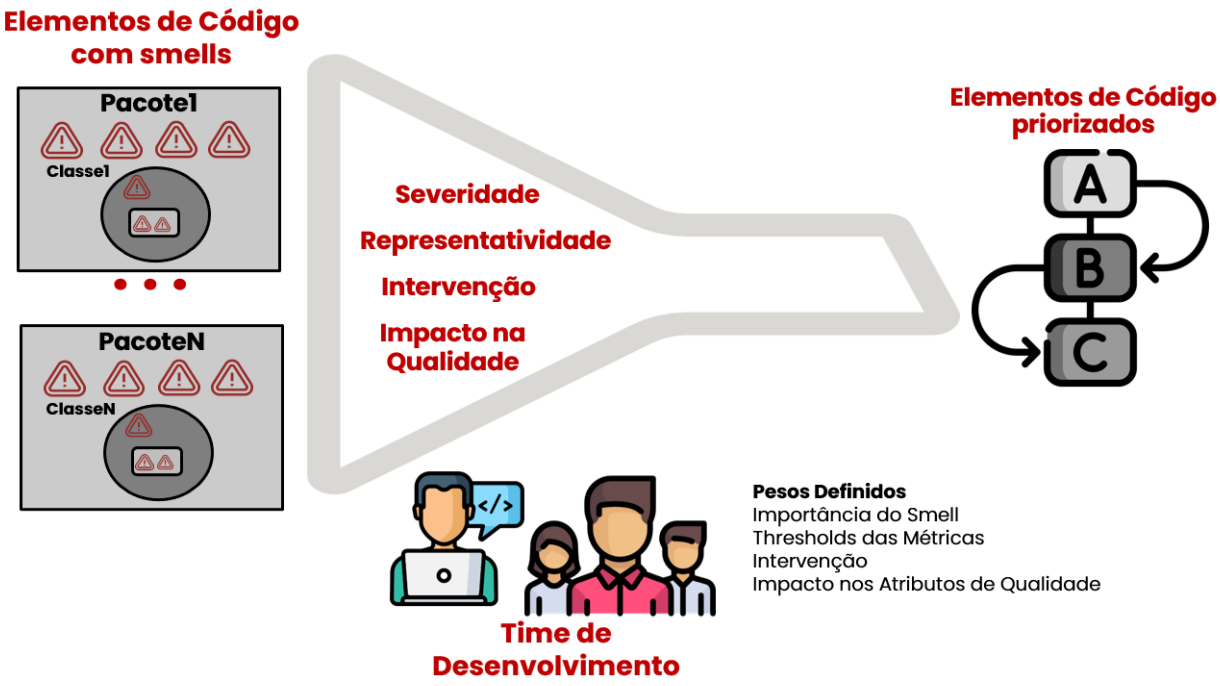
Fonte: O Autor

Tabela 4.3 – Pesos definidos pelos desenvolvedores

Peso	Descrição
θ (*Theta*)	*Threshold* definido para a métrica, segundo referências da literatura (SARAIVA et al., 2015; BIGONHA et al., 2019) e que pode ser personalizado pelos desenvolvedores levando em consideração o contexto do projeto
α (*Alpha*)	Peso dado pelo desenvolvedor a um *smell* em relação a sua importância (*default*=1)
β (*Beta*)	Peso definido pelo desenvolvedor para o impacto em dado atributo de qualidade (*default*=1)
γ (*Gamma*)	Peso definido pelo desenvolvedor em relação a intervenção ao tratar um dado *smell* (*default*=1)

Fonte: O Autor

Figura 4.2 – Elementos integrantes e contemplados pelo modelo de priorização



Fonte: O Autor

definidos na Tabela 4.2. Elas foram escritas em forma de sentenças simples para um mapeamento direto ao modelo. São elas:

- Um *smell (classe Smell)* é especificado através de uma heurística que define como detectar um determinado problema *design* no código (*classes SmellHeuristic, NamespaceSmellHeuristic, TypeSmellHeuristic e MethodSmellHeuristic*);
- A granularidade (*classe Granularity*) define o nível, como pacote (*Namespace*), classe (*Type*) ou método (*Method*). Um *smell* (*classe Smell*) é definido em uma granularidade específica (*classes PackageSmell, TypeSmell e MethodSmell*);
- A intervenção (*classe Intervention*) define o nível de importância de um *smell*, indicando se ele deve ser tratado imediatamente. Pode ser definido pelo desenvolvedor, através de um peso;
- Um *smell* pode estar associado a mais de um atributo de qualidade (*classe QualityAttribute*);
- Os atributos de qualidade (*classe QualityImpact*) podem ter associados pesos, indicando sua importância no projeto. Pode ser definido pelo desenvolvedor;
- O modelo gera um indicador referente ao(s) problema(s) detectados no elemento de código, denominado *Code Disease Indicator* (CDI). Para este cálculo, são considerados os critérios de severidade (*classe SeverityCalculation*), representatividade (*classe RepresentativityCalculation*), qualidade (*classe QualityCalculation*) e intervenção (*classe InterventionCalculation*);
- Como o CDI é por elemento de código e os elementos podem ser auto-contidos, o modelo contempla a composição dos CDIs referentes a cada granularidade (*classe CDIExecutor*). Ou seja, para priorização, é possível considerar os *smells* no nível de classe e no nível de método, levando em conta métodos que fazem parte da classe;
- Ao identificar os *smells* nos elementos de código, o modelo armazena e classifica as co-ocorrências (*classe SmellCoOccurrence*) no mesmo nível de granularidade (intra-métodos, intra-classes, intra-pacotes) e entre granularidades (pacotes-classes, classes-métodos), representadas pela *classe CoOccurrenceClassification*;
- O método de priorização usa o CDI para fazer classificação dos elementos mais problemáticos do software.

Também, procurou-se contemplar a parte de coleta de dados e instrumentação do código (representada por classes de sufixo *Element*). A estruturação dos smells são tratadas por classes de sufixo *Smell*, conectadas a granularidade, importância, impacto em atri-

butos de qualidade projetado e co-ocorrências. As regras de detecção são representadas pelas classes *Heuristic* e a computação dos critérios e ranking pelas classes *Computation*.

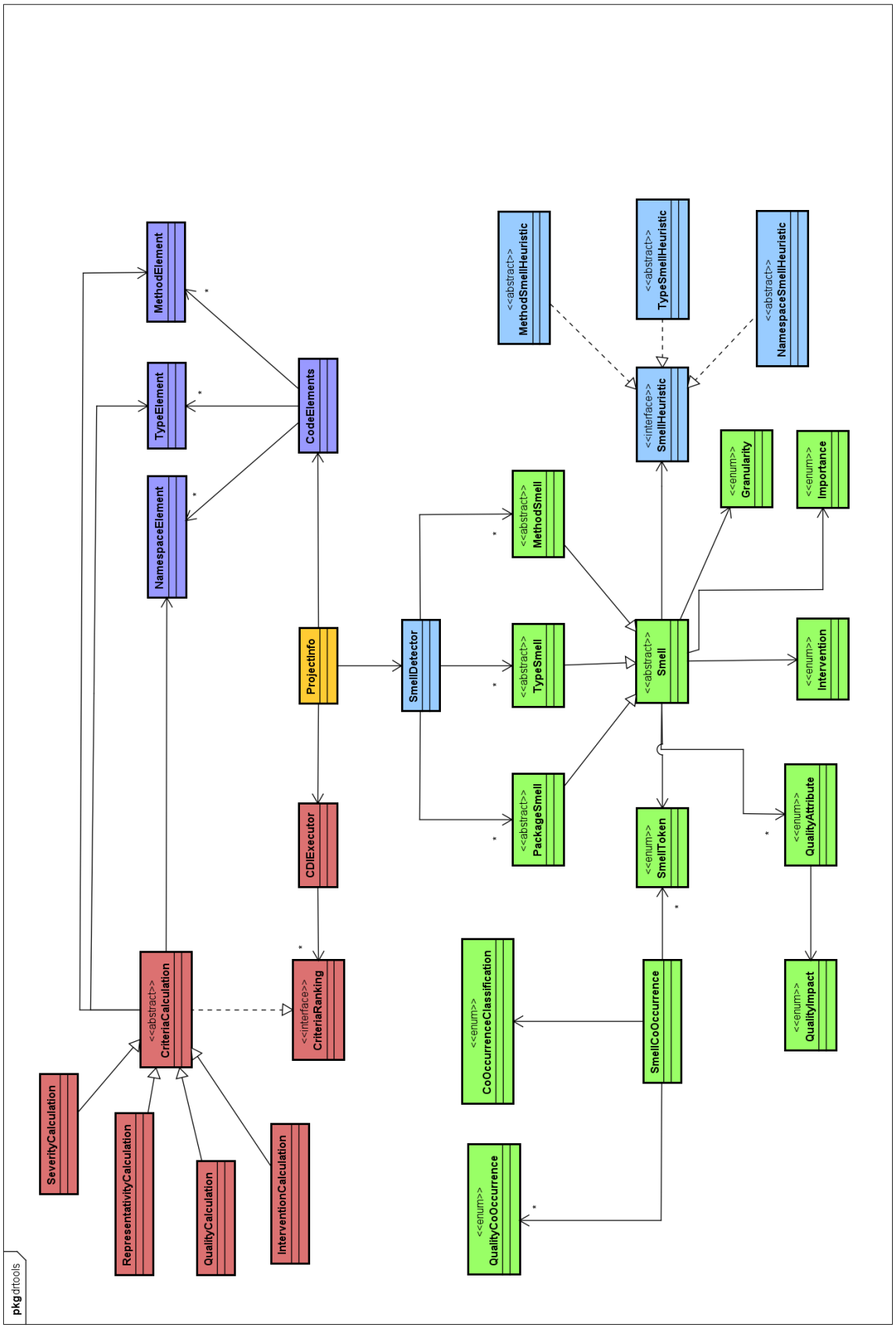
Outro aspecto considerado ao projetar o modelo e a infraestrutura que o suporta é considerar estratégias personalizáveis para definição dos *smells* e customizações do modelo. Em projetos reais, estratégias não personalizadas são ineficientes (HOZANO et al., 2017b). Hozano et al. (2017b) ainda confirmaram que a maioria dos desenvolvedores se beneficia da abordagem baseada em heurísticas, porque permite a construção de detecções personalizadas, sendo eficiente na detecção de *smells*. A definição da customização é realizada em duas etapas: 1) formulação da heurística; 2) refinamento da heurística geral, incluindo métricas específicas, *thresholds* e operadores lógicos. O sucesso da detecção depende diretamente da eficiência de um processo automatizado de personalização. Há, também, a necessidade de modelos de detecção generalizados que possam ser facilmente adaptados para novos *smells* (ABUHASSAN; ALSHAYEB; GHOUTI, 2020) e que permita a detecção de mais de um problema no mesmo elemento de código (PALOMBA et al., 2018a). Assim, desenvolveu-se uma estrutura de alto nível que apoie a definição e identificação dos *smells*, do que puramente o uso de AST[3] (TSANTALIS; CHAIKALIS; CHATZIGEORGIOU, 2018) ou métricas (MARINESCU, 2004).

Esta estrutura de suporte à detecção é uma extensão e adaptação da infraestrutura construída para o *DR-Tools Suite* (LACERDA; PETRILLO; PIMENTA, 2020), que já contempla componentes de estrutura de código e métricas OO. O objetivo é contemplar a instanciação do modelo e o método para dar suporte a priorização.

Outro ponto considerado na estruturação do modelo é o vínculo de atributos de qualidade, mais especificamente, sobre manutenibilidade, com os *smells*. Foram estudados modelos como o QMOOD (BANSIYA; DAVIS, 2002), que ajudam no mapeamento entre atributos externos e internos. Outros estudos, como (IBRAHIM; KAMEL; HASSAN, 2016) e (MOLNAR; MOTOGNA, 2020), que mapeiam as métricas com os atributos internos e externos à manutenibilidade também foram analisados. Ainda, outros modelos como *ADG Maintainability model* (WAGEY; HENDRADJAYA; MARDIYANTO, 2015) e *Quamoco Model* (WAGNER et al., 2015) foram estudados para ter um mapeamento mais concreto entre o elementos de código com atributos de qualidade. Finalmente, para fazer o mapeamento de atributos de qualidade com os problemas de *design*, foram utilizados trabalhos como (ALKHARABSHEH et al., 2019) e (KAUR, 2020). Assim, foram definidos 15 *smells* de diferentes granularidades para implementação no modelo, descritos

[3] *abstract syntax tree*

Figura 4.3 – Modelo resumido proposto representado por um diagrama de classes da UML



Fonte: O Autor

em detalhes na Tabela 4.4.

Tabela 4.4 – *Smells* considerados para detecção e definidos no modelo

Granularidade	Smell	Descrição	Impacto em	Referência
Namespace	*Cyclic Dependency*	refere-se a um subsistema que é envolvido em uma cadeia de relações que rompem a desejável natureza acíclica da estrutura de dependência de um subsistema, dificultando sua liberação	acoplamento, modularidade	(FONTANA et al., 2017; LIPPERT; ROOCK, 2006)
Namespace	*Too Large Package*	Pacotes/subsistemas com alto número de classes/pacotes, indicando que eles servem a mais de uma responsabilidade específica	modularidade, entendimento	(LIPPERT; ROOCK, 2006)
Type	*God Class*	tendem a fazer muito trabalho por conta própria e, por isso, concentra muito código, tende a ser complexo e implementar várias funcionalidades com propósitos diferentes. Também conhecido como *God Class*	manutenibilidade, entendimento	(FOWLER et al., 1999; MAYVAN; RASOOLZADEGAN; JAFARI, 2020)
Type	*Broken Modularization*	surge quando os dados e/ou métodos que idealmente deveriam ter sido localizados em uma única abstração são separados e distribuídos por várias abstrações	modularidade, manutenibilidade	(SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)
Type	*Cyclically Dependent Modularization*	surge quando duas ou mais abstrações dependem uma da outra diretamente ou indiretamente	modularidade	(SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)
Type	*Insufficient Modularization*	uma abstração que não foi completamente decomposta e sua decomposição posterior poderia reduzir tamanho e/ou complexidade	tamanho, complexidade	(SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)
Type	*Deep Hierarchy*	este *smell* surge quando uma hierarquia de herança é 'excessivamente' profunda	herança	(SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)
Type	*Deficient Encapsulation*	ocorre quando a acessibilidade declarada de um ou mais membros de uma abstração é mais permissiva do que realmente necessário	encapsulamento	(SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)
Type	*Hub-Like Modularization*	surge quando uma abstração tem dependências (tanto de entrada quanto de saída) com um grande número de outras abstrações	modularidade	(SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)
Type	*Multifaceted Abstraction*	implica quando uma abstração tem mais de uma responsabilidade atribuída a ela	coesão	(SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)
Type	*Wide Hierarchy*	surge quando uma hierarquia de herança é 'muito' ampla, indicando que tipos intermediários podem estar ausentes	herança	(SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)
Method	*Long Parameter List*	método que possui muitos parâmetros	manutenibilidade	(FOWLER et al., 1999)
Method	*Long Method*	método muito grande e portanto difícil de entender, estender e modificar. É muito provável que este método tenha responsabilidades demais, ferindo um dos princípios de um bom *design* OO (*Single Responsibility Principle*. Também conhecido como *God Method*	entendimento, manutenibilidade	(FOWLER et al., 1999)
Method	*Complex Method*	método que possui uma estrutura complexa. Também conhecido como *Brain Method*	entendimento, manutenibilidade, complexidade	(SHARMA; FRAGKOULIS; SPINELLIS, 2017)
Method	*Bumpy Road*	método que possui alto número de blocos aninhados	manutenibilidade, entendimento, complexidade	(TORNHILL, Adam, 2023)

Fonte: O Autor

4.2.2 Critérios de Priorização

O modelo de priorização proposto inclui os critérios de severidade, representatividade dos *smells*, impacto na qualidade e grau de intervenção, além da composição destes critérios em diferentes granularidades. Diferentemente da abordagem original do MH, baseada em informações fornecidas por partes interessadas, algumas informações são extraídas diretamente do código para compor os critérios juntamente com os pesos

fornecidos pelos desenvolvedores. A seguir, são descritos mais detalhadamente estes critérios e formato de computação.

Severidade

A severidade é um critério que considera o quão grave está um elemento de código. Para compor a pontuação de severidade, é considerado os fatores excedentes das métricas que compõem um dado *smell*, a média aritmética destes fatores vinculados aos pesos atribuídos e, finalmente, o acúmulo destes fatores presentes em vários *smells* presentes no mesmo elemento de código.

O fator da métrica (*MetricFactor*, Equação 4.1) é dado pela razão do valor métrico (*MetricValue*) pelo *threshold* (θ) correspondente que compõe a definição do *smell*. A lista de métricas e *thresholds* considerados no modelo estão disponíveis no Apêndice D. O modelo considera também situações em que outras métricas façam parte da definição do *smell*.

$$MetricFactor_i = \frac{MetricValue_i}{\theta} \tag{4.1}$$

O θ $\{\forall\theta \in \mathbb{R}|\theta > 0\}$ pode ser redefinido pelo desenvolvedor e *i* representa a iteração das métricas usadas na heurística do *smell*. Esta operação de divisão tem dois propósitos: obter o valor excedente ao *threshold* e a normalização da métrica.

Após a computação de cada $MetricFactor_i$, é calculado o fator médio para o *smell* detectado, considerando todos os fatores das métricas que compõe o *smell* (Equação 4.2).

$$MeanFactor_{Smell} = \frac{\left(\sum_{i=1}^{n} MetricFactor_i\right)}{n} \tag{4.2}$$

Onde *n* $\{\forall n \in \mathbb{Z}|n \geq 1\}$ representa o número de métricas presentes na heurística do *smell*. O Algoritmo 1 descreve o processo de computação do $MetricFactor$ e do $MeanFactor_{Smell}$.

Assim, a severidade do *smell* detectado é dada pelo produto do fator médio do *smell* pelo α (Equação 4.3), que representa o peso dado a importância do *smell* pelos desenvolvedores. Caso o peso para este *smell* não seja definido, é considerado o valor *default Normal* ($\alpha = 1$).

$$Severity_{Smell} = MeanFactor_{Smell} \cdot \alpha \tag{4.3}$$

Algoritmo 1: Algoritmo do fator médio do *smell* ($MeanFactor_{Smell}$)

input : Lista das métricas que compõe o *smell* ($MetricValue$) e lista dos thresholds (θ) das métricas correspondentes, ambas de tamanho n
output: Média dos fatores métricos do *smell* ($MeanFactor$)

for $i \leftarrow 1$ **to** n **do**
    $MetricFactor \leftarrow MetricValue[i]/\theta[i]$;
    $SumFactor \leftarrow SumFactor + MetricFactor$;
end
$MeanFactor_{Smell} \leftarrow SumFactor/n$;

As referências de importância a serem atribuídas e seus respectivos pesos estão definidos na Tabela 4.5, podendo ser redefinidos pelos desenvolvedores. Formalmente, o peso α é definido por $\{\forall\alpha \in \mathbb{N} | \alpha \geq 1\}$.

Tabela 4.5 – Pesos (α) atribuídos a importância de um *Smell*

Importância	**Peso**
Critical	4
High	3
Normal	1

Fonte: O Autor

Já a severidade do elemento de código ($Severity_{CE}$, Equação 4.4) é dada pelo somatório das severidades dos *smells* presentes no elemento dividido pela soma dos pesos α dados a cada *smell*.

$$Severity_{CE} = \frac{\left(\sum_{i=1}^{n} Severity_{Smell(i)}\right)}{\sum \alpha} \tag{4.4}$$

O *CE* representa o elemento de código analisado na granularidade de pacote, classe ou método. Assim, o cálculo da severidade contempla não apenas a criticidade de um *smell* mas também a presença de mais de um *smell* em um mesmo elemento de código. Ambas as considerações foram identificadas na literatura como relevantes para priorização (SAE-LIM; HAYASHI; SAEKI, 2018a; PALOMBA et al., 2018a).

O Algoritmo 2 descreve os passos para cálculo da severidade do elemento de código. Na estrutura do algoritmo, são consideradas duas funções: *GetMetricsOfSmell*, responsável por retornar uma lista de métricas consideradas na definição do *smell* e *GetTh-*

resholdsOf, responsável por retornar uma lista dos *thresholds* referente as métricas definidas no *smell*. Estas duas funções fornecem as informações necessárias para a computação do $MeanFactor_{Smell}$ (Algoritmo 1). A severidade é dada pelo produto da soma das severidades dos smells encontrados pela soma das importâncias atribuídas para cada *smell*.

Algoritmo 2: Cálculo da severidade do elemento de código ($Severity_{CE}$)

input : Lista dos *smells* detectados no elemento de código ($SmellList$), lista dos pesos vinculados a importância do *smells* ($ImportanceList$), ambos de tamanho n
output: Severidade do elemento de código ($Severity_{CE}$)

for $i \leftarrow 1$ **to** n **do**
    $MetricValue \leftarrow$ `GetMetricsOfSmell` ($SmellList[i]$) ;
    $\theta \leftarrow$ `GetThresholdsOf` ($MetricValue$) ;
    $Severity_{Smell} \leftarrow$ `MeanFactorDI` ($MetricValue$, θ) ;
    $SumSeverity_{Smell} \leftarrow SumSeverity_{Smell} + Severity_{Smell}$;
    $SumAlpha \leftarrow SumAlpha + ImportanceList[i]$;
end
$Severity_{CE} \leftarrow SumSeverity_{Smell} * SumAlpha$;

Representatividade

A representatividade define quão representativos são os *smells* presentes em um elemento de código frente a outros *smells* na mesma granularidade. Com base na representatividade, é possível avaliar quais elementos de código são mais afetados por *smells* em uma dada granularidade.

A representatividade de um elemento de código (Equação 4.5) é dada pela razão da diversidade ($Diversity_{CE}$), ou seja, quantidade de *smells* diferentes presentes no elemento de código pela quantidade de ocorrências dos *smells* em todos os elementos de código da granularidade ($TotalAmountSmell_{granularity}$, com $TotalAmountSmell_{granularity} \geq 1$).

$$Representativity_{CE} = \frac{Diversity_{CE}}{TotalAmountSmell_{granularity}} \tag{4.5}$$

O Algoritmo 3 descreve os passos para a computação da representatividade de um elemento de código. Em sua estrutura, é considerado duas funções (*GetSetSmellsOf*) que

devolvem um conjunto de *smells* conforme o parâmetro fornecido.

Algoritmo 3: Algoritmo da computação da representatividade referente a um elemento de código ($Representativity_{CE}$)

input : Elemento de código ($CodeElement$)
output: Representatividade do elemento de código ($Representativity_{CE}$)

$Diversity_{CE} \leftarrow$ `GetSetSmellsOf` ($CodeElement$).`Size`;
$TotalAmountSmell_{granularity} \leftarrow$ `GetSetSmellsOf` ($CodeElement.GetGranularity$).`Size`;
$Representativity_{CE} \leftarrow Diversity_{CE}/TotalAmountSmell_{granularity}$;

Impacto na Qualidade

A qualidade é um dos aspectos mais importantes no desenvolvimento de software (SOMMERVILLE, 2016). Conforme Yamashita (2015), os atributos de qualidade estão presentes em um problema se existe um mapeamento na definição do problema com estes atributos. Também, conforme apresentado anteriormente, um *smell* no código pode ter mais de um atributo de qualidade impactado (LACERDA et al., 2020). Assim, no modelo proposto, os atributos de qualidade são vinculados na definição de um *smell*. Portanto, é possível avaliar não apenas os *smells* que mais impactam a qualidade, mas também avaliar quais atributos de qualidade são mais impactados pelos *smells* detectados no projeto.

A importância dos atributos de qualidade pode ser definida pelo desenvolvedor, através de um peso β $\{\forall\beta \in \mathbb{R}|\beta > 0\}$, conforme sugerido na Tabela 4.6.

Tabela 4.6 – Pesos (β) atribuídos aos atributos de qualidade e vinculados a um *Smell*

Importância	**Peso**
Critical	3
High	2
Normal	1
Low	0,5

Fonte: O Autor

O impacto da qualidade em um *smell* ($Quality_{Smell}$, Equação 4.6) é dada pela soma dos pesos atribuídos aos atributos de qualidade e vinculados a um *smell*.

$$Quality_{Smell} = \sum_{i=1}^{n} \beta_{Smell(i)} \tag{4.6}$$

O *n* representa o número de atributos de qualidade associados ao *smell* na definição da heurística. Assim, o impacto da qualidade no elemento de código ($Quality_{CE}$, Equação 4.7) é dada pelo produto da soma dos pesos atribuídos aos atributos de qualidade vinculados aos *smells* pelo número de atributos de qualidade associado ($AmountOfAttributes$, com $AmountOfAttributes \geq 1$).

$$Quality_{CE} = \Big(\sum_{i=1}^{n} Quality_{Smell(i)} \Big) \cdot AmountOfAttributes \tag{4.7}$$

O processo de computação do impacto da qualidade de um elemento de código é melhor detalhado no Algoritmo 4. Na estrutura do algoritmo, a função *GetQualityOf* é responsável por fornecer uma lista dos atributos de qualidade vinculados a um *smell*.

Algoritmo 4: Cálculo no impacto na qualidade do elemento de código ($Quality_{CE}$)

input : Elemento de código ($CodeElement$)
output: Impacto na Qualidade do elemento de código ($Quality_{CE}$)

```
SmellList ← GetListOfSmells (CodeElement) ;
SumOfAttributes ← 0 ;
AmountOfAttributes ← 0 ;
for i ← 1 to SmellList.Size do
    Quality_Smell ← 0;
    β ← GetQualityOf (SmellList[i]) ;
    for j ← 1 to β.Size do
        Quality_Smell ← Quality_Smell + β[j] ;
    end
    SumOfAttributes ← SumOfAttributes + Quality_Smell;
    AmountOfAttributes ← AmountOfAttributes + β.Size ;
end
Quality_CE ← SumOfAttributes * AmountOfAttributes ;
```

Intervenção

A intervenção define o grau de importância em tratar um dado elemento de código (Equação 4.8) que possui *smells*. Para tanto, considera-se também na computação deste critério a diversidade de *smells*, ou seja, a presença de vários *smells* em um mesmo

elemento de código. A intervenção é definida pelo produto do somatório de pesos da intervenção atribuídas aos *smells* do elemento de código pela diversidade.

$$Intervention_{CE} = \left(\sum_{i=1}^{n} \gamma_{Smell(i)} \right) \cdot Diversity_{CE} \tag{4.8}$$

O peso da intervenção γ $\{\forall \gamma \in \mathbb{R} | \gamma > 0\}$ de um *smell* pode ser definida pelo desenvolvedor (Tabela 4.7). O *CE* representa o elemento de código analisado (pacote, classe, método). O $Diversity_{CE}$ é a diversidade, ou seja, a quantidade de *smells* presentes no elemento de código.

Tabela 4.7 – Pesos (γ) atribuídos à intervenção e vinculados a um *smell*

Intervenção	**Peso**
Immediate	4
Planned	3
Normal	1
Low	0,5

Fonte: O Autor

Algoritmo 5: Cálculo da pontuação da intervenção no elemento de código ($Intervention_{CE}$)

input : Elemento de código ($CodeElement$)
output: Pontuação da intervenção no elemento de código ($Intervention_{CE}$)

$SmellList \leftarrow$ `GetListOfSmells` ($CodeElement$) ;
$Diversity_{CE} \leftarrow$ `GetSetSmellsOf` ($CodeElement$).`Size`;
$\gamma \leftarrow 0$;
for $i \leftarrow 1$ **to** $SmellList$.`Size` **do**
  $\gamma \leftarrow \gamma + SmellList[i]$.`Intervention`;
end
$Intervention_{CE} \leftarrow \gamma * Diversity_{CE}$;

Normalização de Valores

Como os valores computados nos critérios tem diferentes origens (métricas de diferentes contextos e pesos), é necessário realizar a normalização destes valores, tanto

para computação do CDI quanto para apresentação dos resultados ao desenvolvedores. Portanto, após a computação dos critérios, eles são normalizados.

No modelo, considera-se a função de normalização *Min-Max*[4] com intervalo entre 1 (menos representativo) e 10 (mais representativo), conforme Equação 4.9.

$$Normalization_{10} = 1 + \frac{(x' - Min(x)) \cdot (10 - 1)}{Max(x) - Min(x)} \tag{4.9}$$

Indicador de Doenças no Código

O indicador de doenças no código, ou *Code Disease Indicator* (CDI), considera os critérios anteriormente definidos para calcular a classificação da priorização de *smells* em um dado elemento de código (Equação 4.10).

$$CDI_{CE} = \frac{\Big(Representativity_{CE} + Quality_{CE} + Intervention_{CE}\Big) \cdot Severity_{CE}}{3} \tag{4.10}$$

Os critérios de representatividade, impacto na qualidade e grau de intervenção são normalizados (Equação 4.9) e agregados. Assim, é possível indicar o acúmulo de *smells*, o reflexo desses *smells* em relação à qualidade e a intervenção definida para atuação em relação ao *smells* presentes no elemento de código. O CDI é dado pelo produto destes critérios agregados pela severidade. A severidade é usada como fator que contempla a criticidade dos *smells*, um dos critérios mais relevantes para priorização (OIZUMI et al., 2019).

Os valores são normalizados, obtendo-se um número no intervalo de 1 a 10 para cada critério. Já para o CDI, a divisão pela constante 3 se deve à obtenção de um valor entre 1 (menos representativo) e 100 (mais representativo), indicando o CDI do elemento de código analisado.

Composições do Indicador de Doenças no Código

A composição compreende no acúmulo dos indicadores de *smells* nos elementos de código. Se o elemento de código for uma classe, a composição (Equação 4.11) se dará

[4]Mais informações em <http://en.wikipedia.org/wiki/Feature_scaling>

pela média aritmética do CDI da classe e a média dos CDIs dos métodos (*i* representa cada método) que compõe a mesma.

$$CDIComposition_{Type} = \frac{\left(CDI_{Type} + \frac{\left(\sum_{i=1}^{n} CDI_{Method(i)}\right)}{n}\right)}{2} \tag{4.11}$$

Da mesma forma, a composição para o pacote compreende na média aritmética do CDI do pacote com a média dos CDIs de todas as classes do respectivo pacote, onde *m* representa o número de classes presentes no respectivo pacote (Equação 4.12).

$$CDIComposition_{Namespace} = \frac{\left(CDI_{Namespace} + \frac{\left(\sum_{j=1}^{m} CDIComposition_{Type(j)}\right)}{m}\right)}{2} \tag{4.12}$$

Para o cálculo do CDI geral do projeto, pode-se obter esse valor pela média aritmética da composição dos CDIs de todos os *o* pacotes pertencentes, representado por *k* (Equação 4.13).

$$CDIComposition_{Project} = \frac{\left(\sum_{k=1}^{o} CDIComposition_{Namespace(k)}\right)}{o} \tag{4.13}$$

A média aritmética das composições dos CDIs de diferentes granularidades é necessária para manter o valor intervalar do indicador entre 1 e 100. Ou seja, o CDI em uma granularidade maior (*e.g.,* pacote) vai envolver a computação de CDIs de granularidades inferiores (classes, métodos).

No Apêndice C, é apresentado um método para computar o modelo de priorização, considerando os critérios e CDIs definidos pelo *DR-Tools Code Health*.

4.3 Suporte automatizado ao método: a ferramenta *DR-Tools Code Health*

Para testar o método e automatizar análises e experimentos, foi desenvolvida uma ferramenta *open-source* de mesmo nome[5]. O *DR-Tools Code Health* é uma ferramenta que faz parte do *DR-Tools Suite* (LACERDA, 2024), um conjunto de ferramentas simples e leves usadas para auxiliar os desenvolvedores em suas tarefas diárias relacionadas a *smells*, expandindo as capacidades de análise por meio de um ambiente interativo baseado em *prompt* e CLI.

Embora ferramentas visuais ofereçam uma variedade de capacidades para diferentes contextos, os desenvolvedores ainda preferem ferramentas CLI para inúmeras atividades (COLEMAN; GRISWOLD; MITCHELL, 2022). Ainda, o domínio dos *prompts* tem sido parte da rotina diária dos desenvolvedores por muito tempo, especialmente como ferramentas de administração de banco de dados (CHAMBERLIN; GILBERT; YOST, 1981). A ferramenta permite avaliar métricas e estatísticas dos elementos de código, *smells* de código, co-ocorrências desses *smells* e priorizar os elementos de código mais afetados pelos *smells*.

4.3.1 Estrutura Interna

Para implementação da ferramenta, foram elencados os seguintes recursos:

- permitir a definição de heurísticas para detecção de *smells*, de forma configurável e extensível;
- permitir o envolvimento dos desenvolvedores no processo de detecção de *smells*;
- considerar como os *smells* impactam a qualidade do software;
- trazer um formato de priorização, multicritério, considerando diferentes níveis de granularidade e suas respectivas composições para identificar os elementos de código mais problemáticos.

Como os problemas de *design* são priorizados e filtrados, os desenvolvedores podem avaliar ações imediatas ou planejadas ao longo do tempo. O *DR-Tools Code Health* tem como objetivo fornecer recursos de suporte e informações úteis para ajudar os desenvolvedores a tomarem boas decisões.

[5]Mais detalhes sobre documentação, instruções de instalação, formato de uso e recursos podem ser encontradas em <https://drtools.site>

Para a implementação do *DR-Tools Code Health*, considerou-se várias características importantes da análise de código. Foi levado em conta diferentes níveis de gravidade de problemas de *design*, diferentes granularidades onde esses problemas podem ocorrer e sua natureza cumulativa (PALOMBA et al., 2018a). Além disso, também foram considerados os atributos de qualidade afetados por *smells* e a percepção de gravidade em diferentes granularidades (ABUHASSAN; ALSHAYEB; GHOUTI, 2020). O método de priorização implementado no *DR-Tools Code Health* permite que os desenvolvedores identifiquem e selecionem os elementos de código mais problemáticos, fornecendo suporte para atividades de desenvolvimento e manutenção de software.

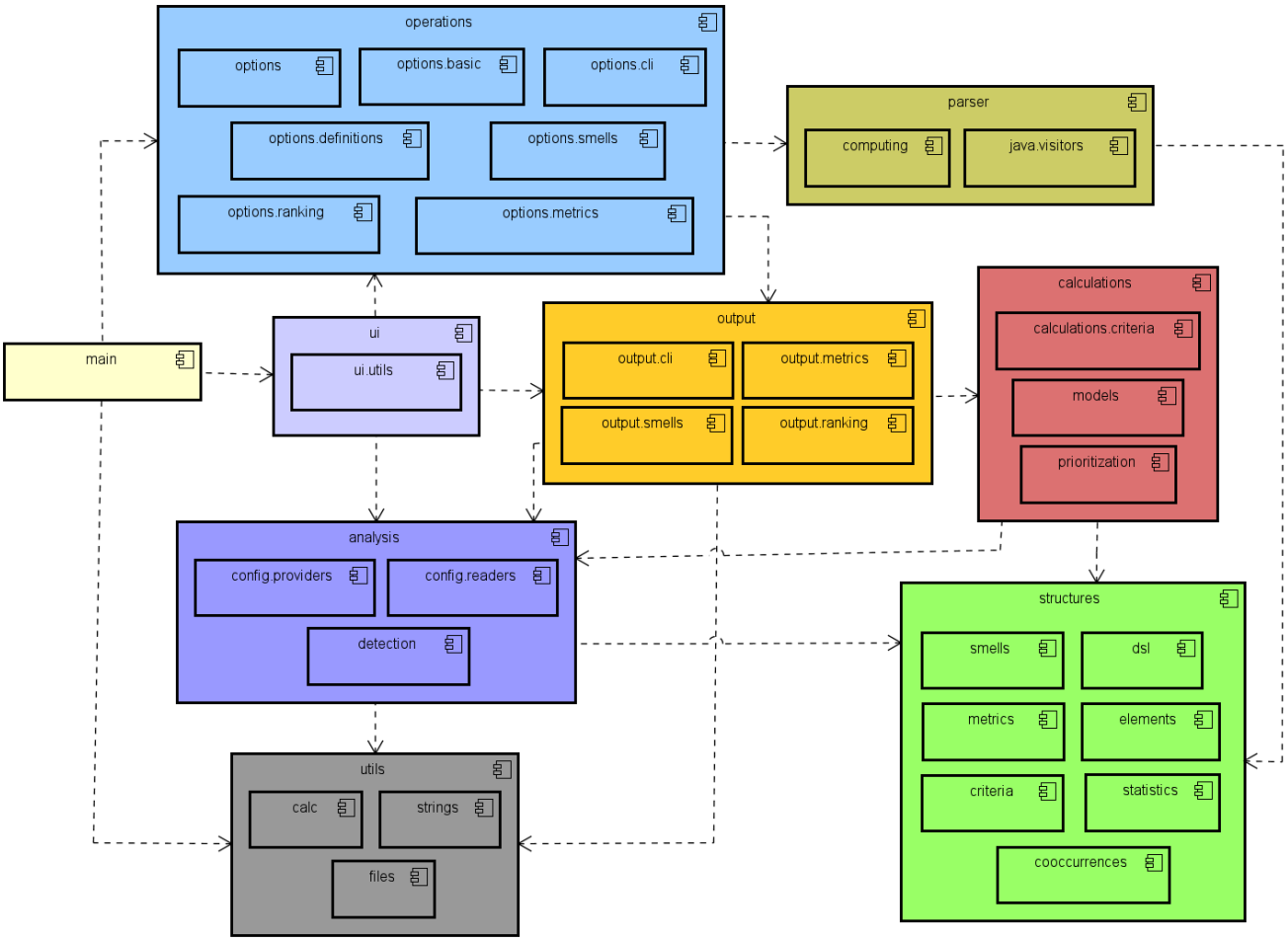
Na Figura 4.4, são apresentados os principais componentes implementados no *DR-Tools Code Health*. O componente *main* é responsável por receber informações de entrada (código, arquivos de configuração); o componente das *operations* define o conjunto completo de comandos disponíveis; o componente *parser* fornece funcionalidades para ler elementos de código via AST. O *DR-Tools Code Health* está projetado para extensão a outras linguagens de programação; o componente *ui* descreve todos os elementos e interações da GUI. A *analysis* é responsável por carregar configurações pré-existentes e realizar a detecção de *smells*. As *structures* representam os componentes de elementos de código, definições de *smells*, co-ocorrências, métricas, critérios usados e API fluente. O componente *calculations* realizam o cálculo de classificação e priorização dos elementos de código. O componente *utils* descrevem os componentes básicos usados em cálculos estatísticos, manipulação de arquivos e operações com *strings*. Por fim, o componente *output* contém os componentes responsáveis por apresentar os resultados aos desenvolvedores, em formato console, CSV e JSON.

4.3.2 Modos de Uso

O *DR-Tools Code Health* possui dois modos de execução: CLI e interativo (Figura 4.5).

No modo *CLI*, os desenvolvedores podem personalizar os parâmetros para seus projetos e obter os resultados da análise do *DR-Tools Code Health*. São duas opções de uso: *–init* e *–analyze*. Com a opção *–init*, o *DR-Tools Code Health* criará um diretório `.drtools/` dentro da estrutura de diretórios do repositório analisado, para armazenar os arquivos de configuração e, também, outros dois subdiretórios (`config` e `templates`, respectivamente). Dentro de `config`, o arquivo `drtools-config.properties` con-

Figura 4.4 – Estrutura dos componentes do *DR-Tools Code Health*

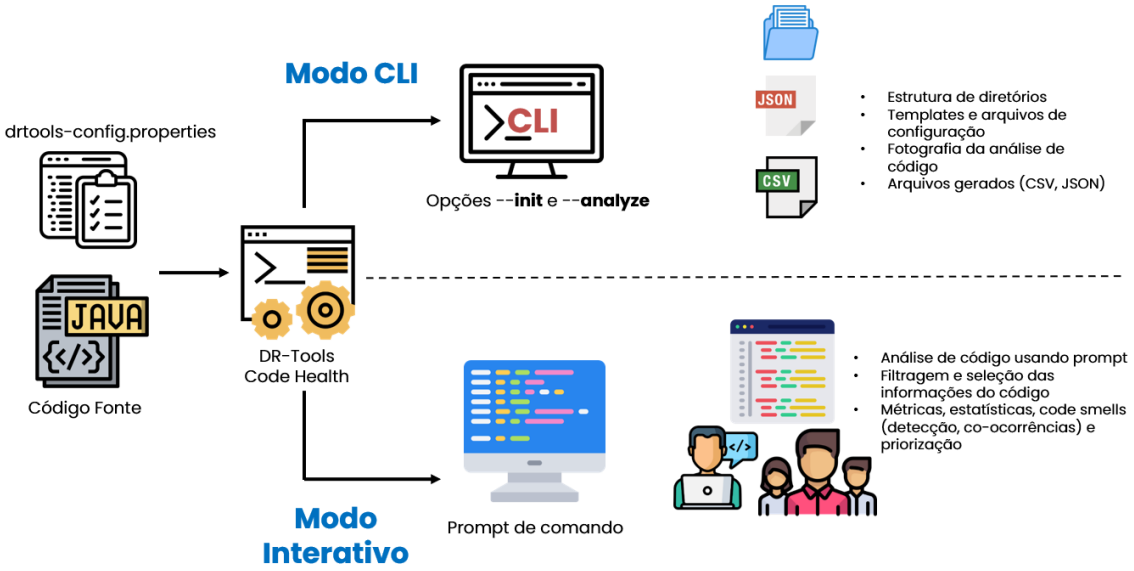


Fonte: O Autor

tém os parâmetros usados para analisar o código do projeto. O diretório `templates` possui configurações alternativas que também podem ser usadas durante a análise, variando os *thresholds* das métricas (parâmetros definidos pelo usuário, valores considerando soma da média com o desvio padrão das respectivas métricas e considerando informações estatísticas do código). Estes arquivos são identificados na ferramenta a partir do conceito de *Profile*. O *Profile* ajuda a customizar e definir nomes para estas configurações, permitindo que os desenvolvedores gerenciem melhor o uso destes parâmetros no *DR-Tools Code Health*. Para utilizar as configurações que estão no diretório `templates`, basta mover o arquivo para o diretório `config` e renomear o arquivo para `drtools-config.properties`.

A opção *–analyze* gera os resultados da análise de código, organizados por métricas, *smells* e *ranking*. Os resultados estão disponíveis em arquivos no formato CSV e JSON. Estes resultados são gerados por padrão, no diretório `analysis`, identificados por data e hora de geração, dentro do `.drtools/`. Também, é possível gerar os

Figura 4.5 – Modos de Execução do *DR-Tools Code Health*. *Entrada:* código-fonte e arquivo de configuração do projeto (opcional). *Processamento:* podem ser utilizados dois modos de uso - *CLI*, através das opções *–init* e *–analyze*, criando diretórios, arquivos de configuração e resultados de análise em formatos CSV e JSON; *Interativo*, permitindo aos desenvolvedores analisar o projeto, observar métricas, estatísticas, *smells* e priorização via *prompt de comando*



Fonte: O Autor

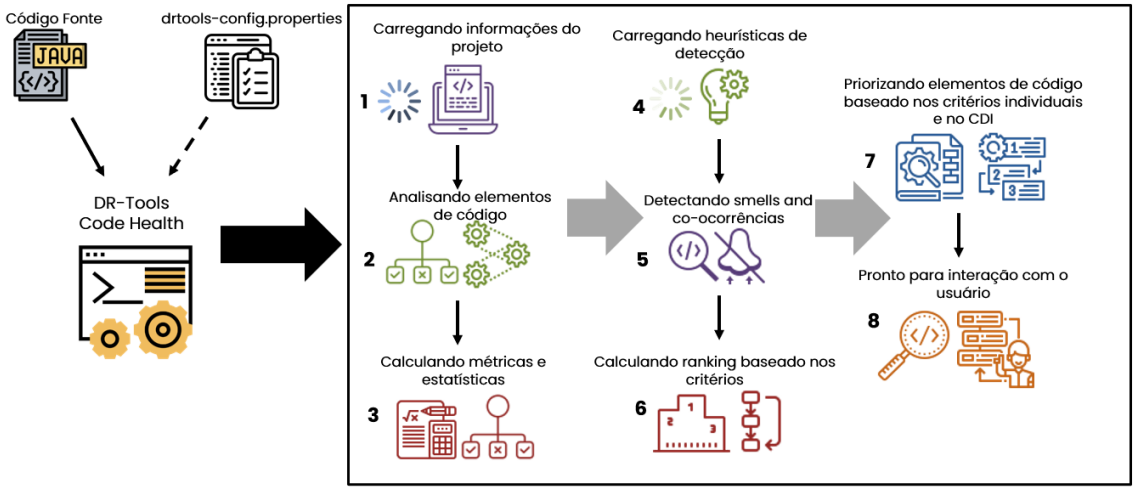
resultados em um diretório específico, conforme o comando `drtools-code-health <diretório do projeto> -analyze /temp`. Assim, toda a estrutura de diretórios (`.drtools/analysis/`) estará dentro do diretório `/temp`. Este tipo de recurso é útil, pois as análises não ficariam dentro da estrutura de diretórios do repositório do projeto, podendo ser usada como uma *task* de um processo de *build pipeline*[6] do projeto.

Além do modo CLI, é possível executar o *DR-Tools Code Health* no modo interativo. Nesta opção, após a análise prévia do código do projeto, é fornecido um *prompt* com vários comandos de consulta a métricas, *smells* e *ranking* de problemas no código. Independente do modo de uso, o *DR-Tools Code Health* executa uma série de etapas para análise do código, resumidas na Figura 4.6.

No modo interativo (Figura 4.7), um *prompt* é fornecido com mais de 60 comandos de análise e consulta sob a perspectiva de métricas, *smells* e classificação de problemas de código a partir da análise do código do projeto. É possível usar tanto o comando completo (*command*) quanto o atalho (*shortcut*).

[6]Mais informações sobre este processo podem ser obtidas em https://www.redhat.com/en/topics/devops/what-cicd-pipeline

Figura 4.6 – Estrutura interna do *DR-Tools Code Health*: Partindo da entrada do código do projeto e do arquivo de configuração (opcional), uma série de ações são executadas - (1) carregamento das informações gerais do projeto; (2) análise dos elementos de código e suas respectivas informações (3) cálculo de métricas e estatísticas considerando o estado atual; (4) carregamento das heurísticas de detecção para *smells* e co-ocorrências com base nas informações anteriores; (5) detecção de *smells* e co-ocorrências com base nas heurísticas; (6) cálculo de classificação considerando os critérios utilizados (por exemplo, gravidade, impacto na qualidade); (7) priorização dos elementos de código usando os critérios individuais e o CDI agregado (Indicador de Doença no Código); e finalmente, (8) dados do projeto prontos para serem gerados em formato de arquivo (modo CLI) ou manipulados e consultados via *prompt* (modo interativo)



Fonte: O Autor

4.3.3 Módulos

O *DR-Tools Code Health* possui 3 módulos: métricas, *smells* e *ranking* dos problemas nos elementos de código.

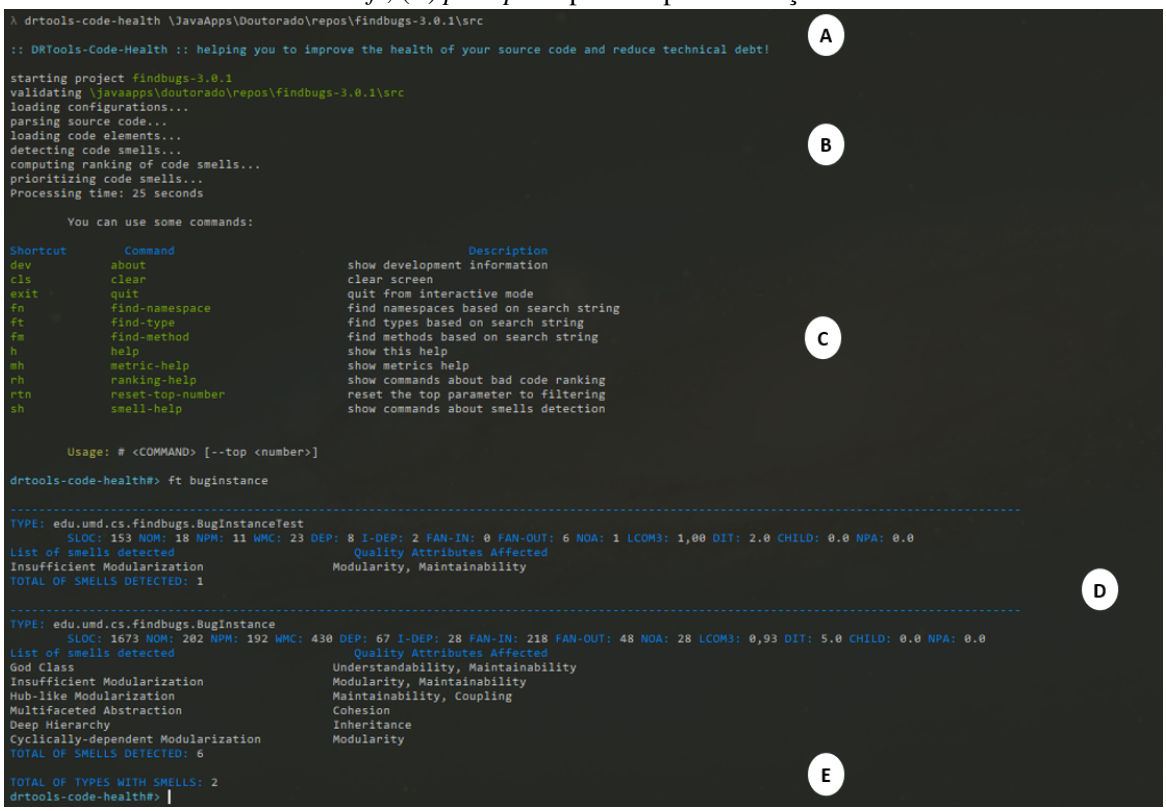
No **módulo de métricas**, são apresentadas 48 métricas contextualizadas por sumário do projeto, pacote, classe, método, dependências e acoplamento. A lista completa das métricas pode ser consultada no Apêndice D.

Também, o *DR-Tools Code Health* fornece uma análise estatística com quartis, média, mediana, valores mínimo e máximo, desvio padrão, amplitude e *outlier* para as métricas analisadas, além, é claro, do *threshold* dado pela literatura. Assim, é possível comparar os dados coletados do código com os dados estatísticos do projeto. Os *thresholds* podem ser customizados pelos desenvolvedores, considerando as características do projeto.

Como exemplo, conforme apresentado na Figura 4.8, é possível analisar aspectos mais gerais do projeto, em diferentes níveis de granularidade.

Na granularidade de *types*, também é possível ver inúmeras informações impor-

Figura 4.7 – *DR-Tools Code Health* em ação - (A) executando *DR-Tools Code Health* no *prompt* do sistema (iniciando o modo interativo), (B) processando dados do projeto - arquivos de configuração, analisando código, detectando *smells* e gerando classificação, (C) exibindo opções básicas de comando e ajuda de módulo, (D) procurando por uma classe chamada *"buginstance"* usando o comando *ft*, (E) *prompt* disponível para interação com o usuário

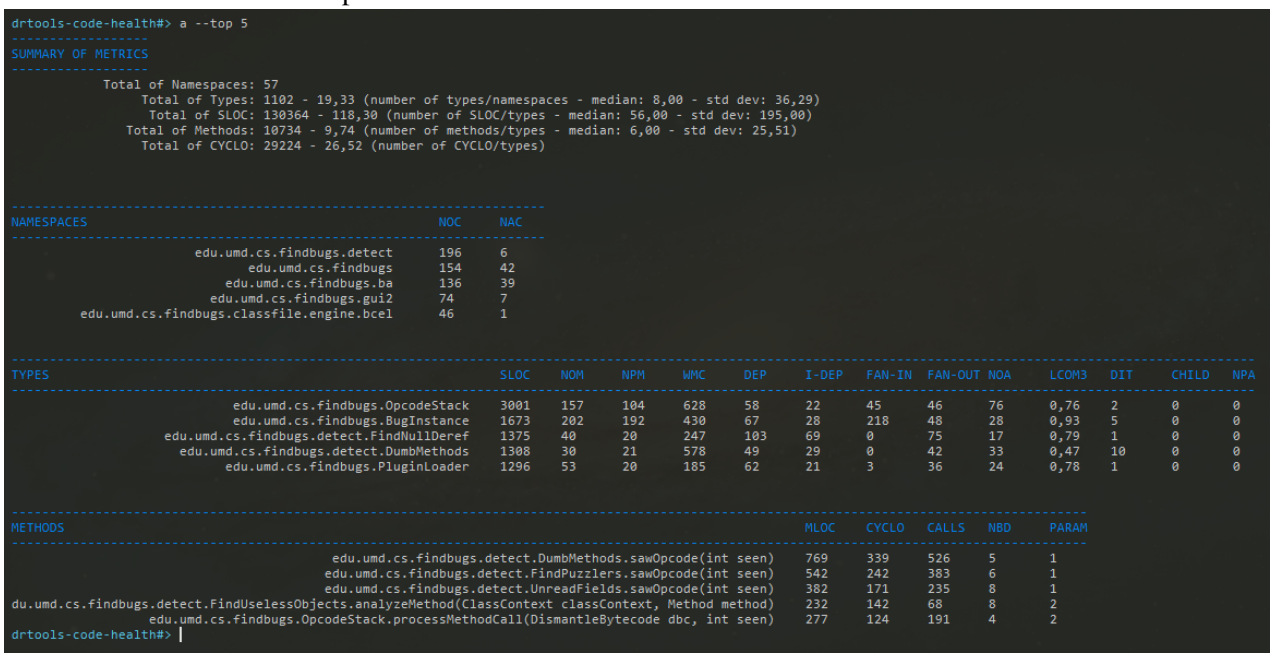


Fonte: O Autor

tantes, com foco na complexidade (métrica WMC) e tamanho (métrica SLOC) das classes, além de informações sobre dependências (DEP, I-DEP), tamanho da API (NPM) e acoplamentos (FAN-IN, FAN-OUT). Passando para a granularidade de *methods*, observa-se a lista de 5 métodos considerados mais complexos. Note que os primeiros 3 métodos listados têm a mesma assinatura (`sawOpcode`) em classes distintas, o que poderia indicar que um método está definido em uma hierarquia. É possível avaliar diferentes aspectos como tamanho ou complexidade. No entanto, caso o desenvolvedor queira considerar problemas de compreensão de código, pode observar a métrica NBD. É possível observar um método em destaque, com um alto valor de NBD. Este método não é o maior, mas certamente se destaca com problemas de compreensão de código.

No **módulo de *smells***, são considerados 15 tipos de *smells* (Tabela 4.4), nas granularidades de *namespace*, *type* e *method*. Os *smells* afetam a qualidade e, na maioria dos casos, o propósito da reestruturação e refatorações tem como intenção e motivação melhorar a qualidade (PANTIUCHINA et al., 2020; ALOMAR et al., 2021). Portanto,

Figura 4.8 – Com o comando *a –top 5*, é possível ter uma visão geral em diferentes níveis do projeto. Primeiramente, um resumo geral com o número de pacotes, tipos e métodos, valores médios, medianas e desvios padrão. Note que se tem uma média de 19 *types* por *namespace*. No entanto, quando é observado os dados específicos no segundo nível (*namespace*), pode-se ver que os 5 pacotes listados têm valores acima da média



Fonte: O Autor

é importante associar os problemas com a qualidade e usá-los como um dos critérios de priorização, expondo as partes mais problemáticas para os desenvolvedores do time.

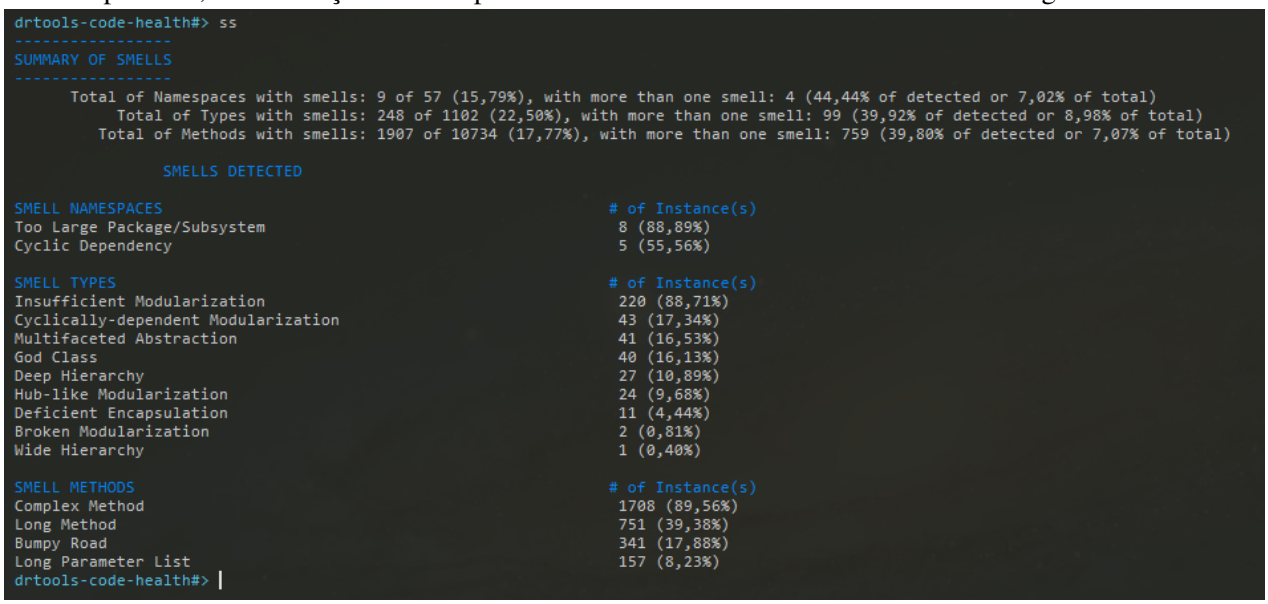
Como exemplo, pode-se analisar o código observando como os *smells* estão distribuídos em diferentes granularidades. Para isso, o desenvolvedor pode usar o comando `ss` (Figura 4.9), que fornece um resumo dos *smells* identificados.

Ao analisar os dados apresentados no nível do método, mais de 17% dos métodos apresentam pelo menos um sintoma, e dos métodos detectados, mais de 39% têm mais de um sintoma. Isso é um forte indicador para investigação, pois elementos de código que têm múltiplos *smells* são mais propensos a *bugs* e mudanças.

No Apêndice E, é apresentada a lista de configurações e informações dos *smells* com respectivas informações adicionais como importância, impacto na qualidade e intervenção.

Outro recurso encontrado no *DR-Tools Code Health* é a detecção e análise de co-ocorrências dos *smells*. Este recurso é importante, pois permite que os desenvolvedores possam analisar quais *smells* mais aparecem juntos, sendo um *anti-pattern* que precisa ser discutido com o time. Afinal, possivelmente isso seja uma ação recorrente ao longo do tempo, propagado pelas pessoas que interagiram com o código. Como exemplo, apresen-

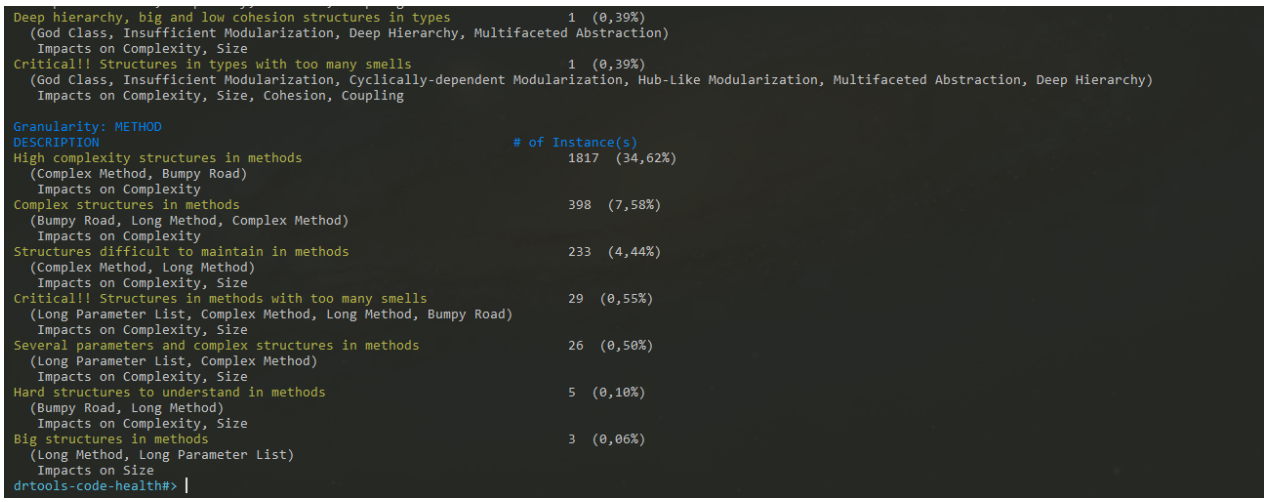
Figura 4.9 – Comando *ss* permite avaliar como os *smells* estão distribuídos em diferentes níveis de granularidade (*namespace, type, method*), quais *smells* aparecem com mais frequência e, mais importante, a distribuição de múltiplos *smells* dentro de um único elemento de código



Fonte: O Autor

tado na Figura 4.10, é possível observar que *complex method* e *bumpy road* se repetem muito. Com esta informação, o time pode gerar ações para reduzir essas estruturas com complexidade excessiva.

Figura 4.10 – Resultado da execução do comando *sco*, que lista todas as co-ocorrências encontradas. Nesta figura, é possível observar as co-ocorrências definidas na granularidade de método



Fonte: O Autor

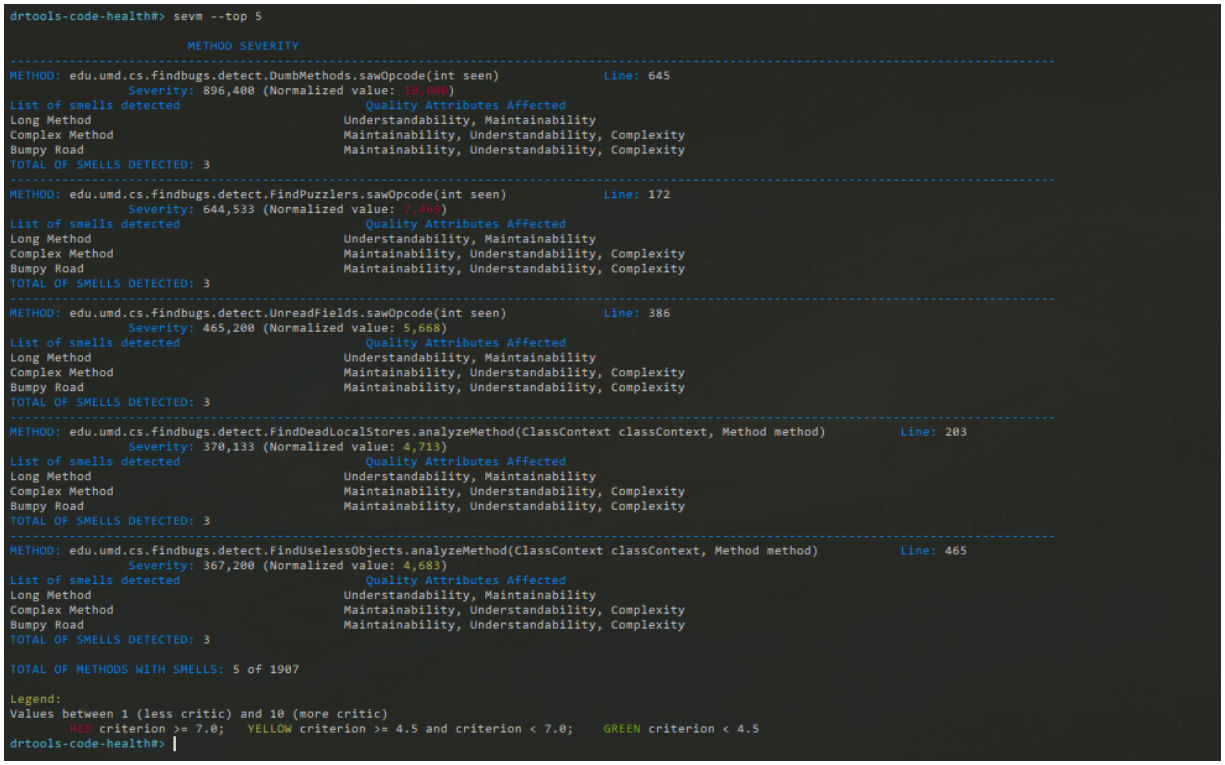
A lista completa de todas as co-ocorrências configuradas inicialmente no *DR-Tools Code Health* estão disponíveis no Apêndice F. Esta lista pode ser estendida pelos desenvolvedores, através dos recursos de customizações.

O **módulo de *ranking*** implementa o modelo de priorização com os critérios de

severidade, representatividade, impacto na qualidade e grau de intervenção do time, identificando elementos de código críticos que demandam mais atenção (Seção 4.2.2). A severidade de um elemento de código é determinada pela gravidade de todos os seus *smells*, com seus respectivos pesos definidos pelo desenvolvedor. A representatividade indica que um determinado *smell* de código em um elemento de código é representativo (ou seja, o número de instâncias desse *smell*) em comparação com outros no mesmo nível de granularidade. O impacto na qualidade é avaliado atribuindo pesos aos atributos de qualidade associados a um *smell*. A intervenção representa a importância de abordar um elemento de código que possui *smells*, também podendo ser personalizado desenvolvedores.

Com a ferramenta, pode-se fazer uma análise individual de cada critério (por exemplo, severidade) em seus respectivos níveis de granularidade. Os resultados para um critério individual são apresentados em um valor normalizado de 1 a 10 (Figura 4.11).

Figura 4.11 – Resultado da execução do comando *sevm –top 5*, listando os 5 métodos priorizados por gravidade, juntamente com os respectivos *smells* detectados e impacto na qualidade

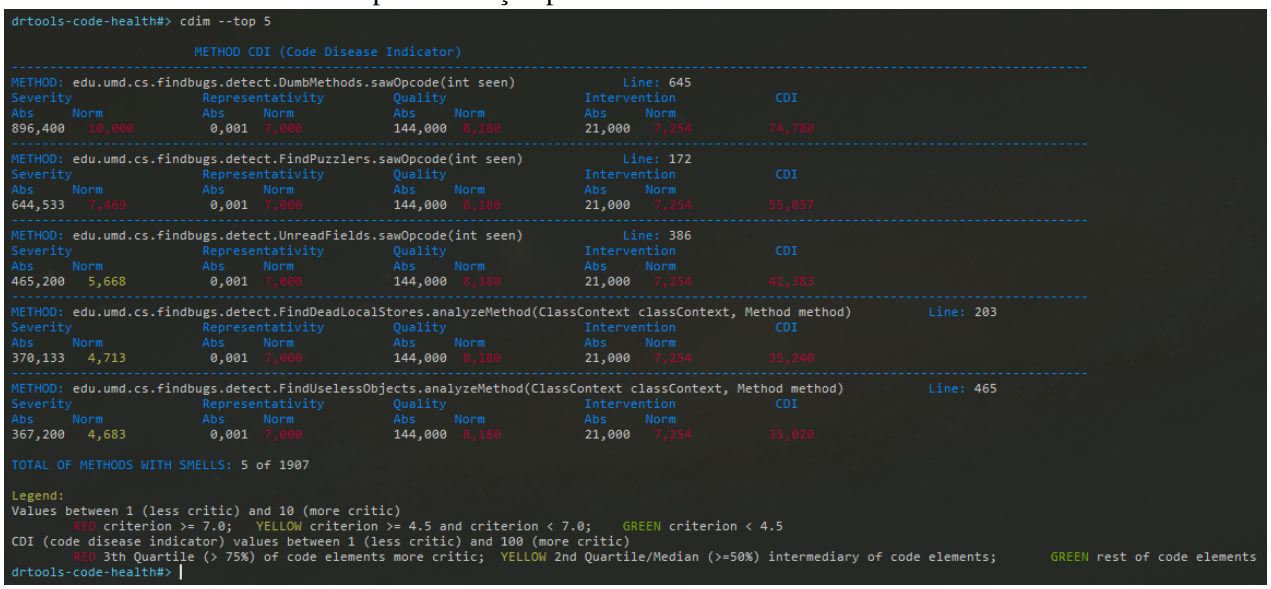


Fonte: O Autor

Também, é possível agregar os critérios em um indicador global (Figura 4.12), que apresenta os elementos de código mais problemáticos em um determinado nível de granularidade. Este indicador é denominado CDI, variando de 1 (menos crítico) a 100 (mais crítico). O indicador pode ser gerado em outros níveis de granularidade (Figura 4.13).

Ainda, a ferramenta calcula a composição do Indicador de Doença do Código (CCDI) combinando os CDIs de elementos de granularidade mais baixa (por exemplo, métodos) com o CDI de elementos de granularidade mais alta (por exemplo, classes), seguindo a mesma escala de CDI (Figura 4.14).

Figura 4.12 – Comando *cdim –top 5*, apresentando os 5 métodos mais críticos, de acordo com o CDI. O CDI leva em consideração todos os critérios para criar um *ranking* dos métodos sugeridos para avaliação pelo desenvolvedor



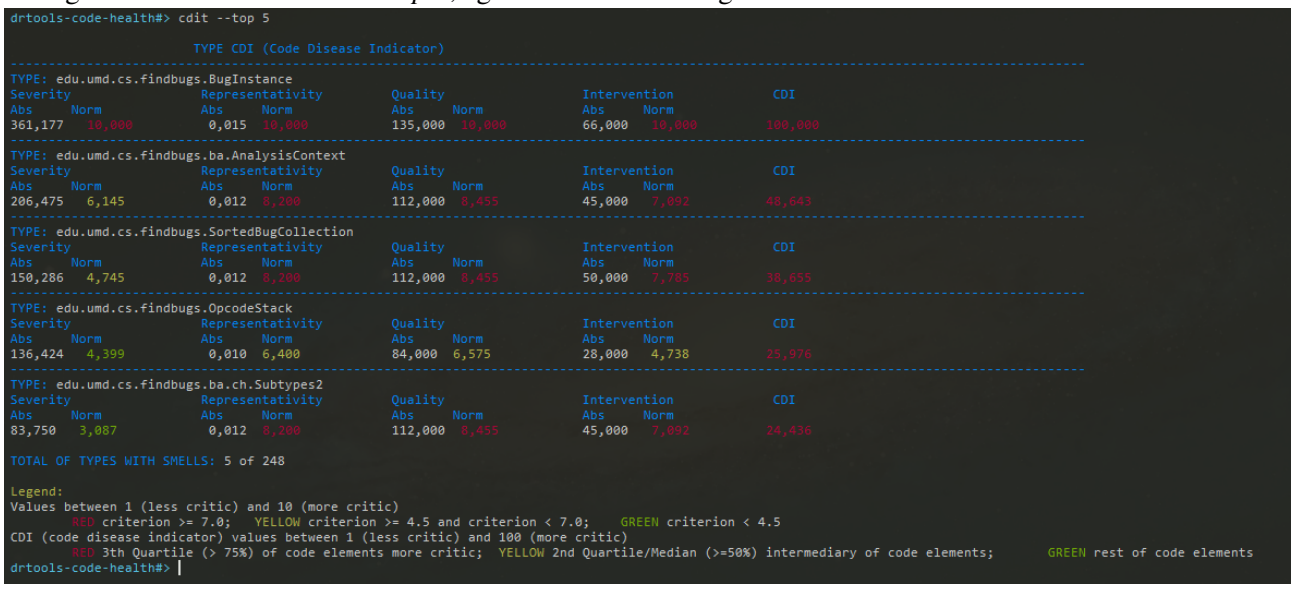
Fonte: O Autor

4.3.4 Customizações

É possível realizar vários tipos de personalizações no *DR-Tools Code Health*, levando em consideração o contexto do projeto. Dependendo do tipo de projeto, o time pode ter a intenção de investigar certos aspectos como, por exemplo, elementos de código que estão impactando mais certos atributos de qualidade, como complexidade e manutenibilidade. Estas customizações são relacionadas as definições dos *smells*, co-ocorrências e pesos utilizados na configuração da ferramenta. Algumas customizações podem envolver alteração de código e outras apenas alterando parâmetros no arquivo de configuração.

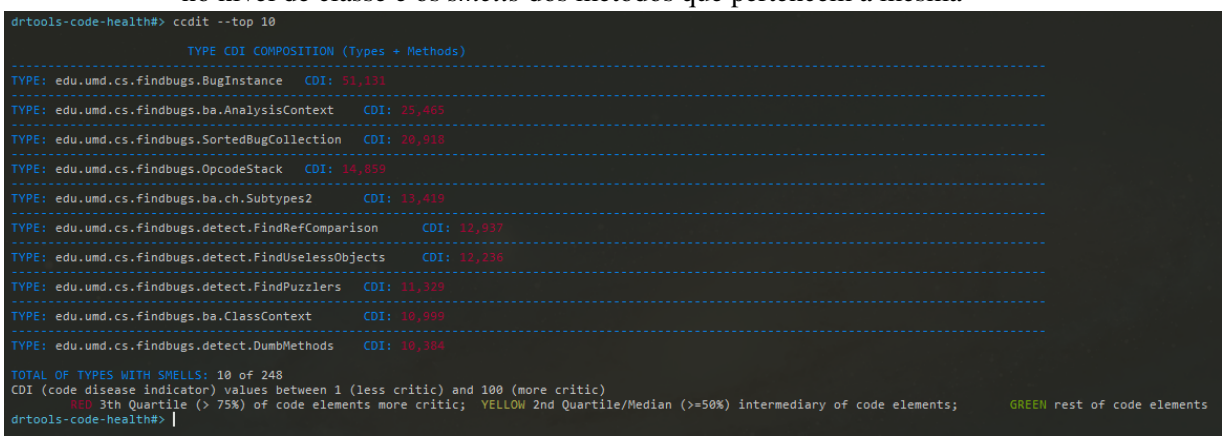
As **customizações via código** estão relacionadas com a inserção ou modificação das heurísticas de detecção dos *smells* e co-ocorrências. A maioria dos desenvolvedores pode se beneficiar da abordagem baseada em heurísticas, porque permite a construção de detecções personalizadas, sendo eficiente na detecção dos *smells*. Também, a implementação realizada possibilita facilmente a inserção de novos *smells* e a detecção de mais de

Figura 4.13 – Comando *cdit –top 5*, agora considerando a granularidade no nível de classes



Fonte: O Autor

Figura 4.14 – Comando *ccdit –top 10*, listando as 10 classes mais críticas, considerando os *smells* no nível de classe e os *smells* dos métodos que pertencem a mesma



Fonte: O Autor

um problema no mesmo elemento de código.

Assim, o *DR-Tools Code Health* possui uma estrutura de alto nível que apoia a definição e identificação dos *smells*, do que puramente o uso de ASTs ou métricas. Esta estrutura de suporte à detecção é uma extensão e adaptação da infraestrutura construída para o *DR-Tools Suite*, que já contempla componentes de código e métricas OO. O objetivo é permitir a instanciação do modelo e do método de priorização através de uma API Fluente. Portanto, foi construída uma DSL interna (FOWLER, 2010) para definição dos *smells* e *co-ocorrências* (Figuras 4.15 e 4.16, respectivamente).

Figura 4.15 – Trecho da DSL utilizada para definir a estrutura do *smell*

```
private static void defineModularityTypeSmells() {
    Smell godClass = new SmellBuilder()
            .type()
            .heuristic(new GodClass())
            .withName("God Class", SmellToken.GOD_CLASS,
                    "This smell is huge, complex, and has an extremely high number of fields and methods")
            .importance(Importance.HIGH)
            .withImpact(QualityAttribute.UNDERSTANDABILITY, QualityAttribute.MAINTAINABILITY)
            .withAction(Intervention.PLANNED)
            .createInstance();

    Smell bumpyRoad = new SmellBuilder()
            .method()
            .heuristic(new BumpyRoad())
            .withName("Bumpy Road", SmellToken.BUMPY_ROAD,
                    "This smell occurs when a method has high nested blocks")
            .importance(Importance.CRITICAL)
            .withImpact(QualityAttribute.MAINTAINABILITY, QualityAttribute.UNDERSTANDABILITY,
                    QualityAttribute.COMPLEXITY)
            .withAction(Intervention.IMMEDIATE)
            .createInstance();
```

Fonte: O Autor

Figura 4.16 – Trecho da DSL utilizada para definir as co-ocorrências dos *smells*

```
SmellCoOccurrence bigAndComplexStructuresIC = new SmellCoOccurrenceBuilder()
        .withDescription("Big and complex structures inter-components")
        .addSmells(SmellToken.GOD_CLASS, SmellToken.INSUFFICIENT_MODULARIZATION, SmellToken.LONG_METHOD, SmellToken.COMPLEX_METHOD)
        .category(CoOccurrenceClassification.INTER_COMPONENT)
        .impactOn(QualityCoOccurrence.SIZE, QualityCoOccurrence.COMPLEXITY)
        .createInstance();

SmellCoOccurrence complexStructuresIC = new SmellCoOccurrenceBuilder()
        .withDescription("Complex structures inter-components")
        .addSmells(SmellToken.INSUFFICIENT_MODULARIZATION, SmellToken.COMPLEX_METHOD)
        .category(CoOccurrenceClassification.INTER_COMPONENT)
        .impactOn(QualityCoOccurrence.COMPLEXITY)
        .createInstance();
```

Fonte: O Autor

Outras customizações, sem necessidade de alteração no código, podem ser feitas para ajustar parâmetros dos critérios usados na detecção, priorização e *ranking* dos elementos de código mais problemáticos. Basta editar o arquivo (Figura 4.17) e adaptar os parâmetros para o contexto do seu projeto, redefinindo o *profile*.

As informações específicas de pesos, definidas pelos desenvolvedores, são armazenadas junto ao repositório de código do projeto. Caso estas definições não tenham sidos

feitas para o projeto, é considerado os parâmetros *default*, previamente configurados na ferramenta. No Apêndice G, podem ser consultados os valores pré-definidos e utilizados.

Figura 4.17 – Trecho do arquivo de configuração *drtools-config.properties*

```
#
# -- Intervention, used to smells ranking
#
intervention.immediate=4.0
intervention.planned=2.0
intervention.normal=1.0
intervention.low=0.5

#
# -- Quality Attributes, linked to smells and describing its impact
#    Parameters: CRITICAL, HIGH, NORMAL, and LOW
#
quality.attribute.cohesion=HIGH
quality.attribute.complexity=CRITICAL
quality.attribute.coupling=HIGH
quality.attribute.encapsulation=NORMAL
quality.attribute.inheritance=NORMAL
quality.attribute.maintainability=HIGH
```

Fonte: O Autor

Por ser um projeto *open-source*, para facilitar o processo de alterações específicas no código e futuras contribuições, o projeto possui o processo de *build*[7] configurados através de ferramentas como *Maven*[8] e *Gradle*[9].

O *DR-Tools Code Health* é uma ferramenta flexível e adaptável. Com ela, é possível analisar, cruzar, avaliar os dados do código de diferentes formas, além de poder customizar e parametrizar conforme contexto do projeto. O propósito da ferramenta não é julgar, mas apoiar e ajudar os desenvolvedores na tomada de decisão em relação a decisões técnicas.

4.3.5 Aplicabilidade

De acordo com alguns autores (RASOOL; ARSHAD, 2015; SINGH; KAUR, 2017; BESKER; MARTINI; BOSCH, 2018) a detecção de problemas no código reduz o custo de manutenção se as falhas forem encontradas nos estágios iniciais de desenvolvimento de software. Portanto, ao desenvolver o *DR-Tools Code Health*, pensou-se também em como ele poderia ser utilizado no fluxo de desenvolvimento e como envolver os desenvolvedores neste processo. Afinal, os desenvolvedores são importantes no processo

[7] compilação do projeto e execução dos testes automatizados
[8] Mais informações em https://maven.apache.org/
[9] Mais informações em https://gradle.org/

(SIMONS; SINGER; WHITE, 2015) e sua participação na adaptação de técnicas semi-automatizadas tornam-as mais eficazes (PANTIUCHINA; LANZA; BAVOTA, 2018).

Além disso, é importante que as ferramentas sejam simples, com mecanismos de filtragem e estejam conectadas ao fluxo de desenvolvimento, permitindo ações pontuais e contínuas (SADOWSKI et al., 2018). Assim, o *DR-Tools Code Health* implementa o método proposto, considerando também a participação dos desenvolvedores (VERMA; KUMAR; VERMA, 2021) na definição prévia de informações. Considera-se que a ferramenta possa ser adotada durante as atividades programação, sessões de revisão de código, ou mesmo em sessões mais complexas de análise e investigação do projeto, com objetivo de apoiar a tomada de decisões (HAENDLER; FRYSAK, 2018).

A seguir, são considerados estes cenários de uso:

Atividades de Programação: As atividades de programação consideradas vão desde implementações de novas funcionalidades até manutenções perfectivas no código. Segundo Tufano et al. (2017), a maior parte do código com *smells* (entre 60% e 66%) são introduzidos durante o aprimoramento do código. Afinal, segundo Aversano, Carpenito and Iammarino (2020), as classes alteradas frequentemente são aquelas em que os problemas estão ocorrendo. Assim como as refatorações, que deveriam ser considerados como parte de sua atividade diária, sendo um hábito (LAHTINEN; LEPPANEN, 2016), a análise de código também deveria fazer parte do cotidiano dos desenvolvedores.

O *DR-Tools Code Health* permite priorizar e filtrar os *smells* em diferentes granularidades, considerando aspectos de qualidade durante as atividades de programação pode ajudar na identificação de problemas. Antes mesmo de realizar um *commit*, o seu uso pode ajudar a minimizar e/ou evitar possíveis problemas que seriam identificados *a posteriori* e já presentes no repositório do projeto.

Revisões de Código: As atividades de revisão de código podem acontecer individualmente, em pares ou mesmo em reuniões do time. Estas revisões, quando realizadas com mais membros do time, podem trazer aprendizados, principalmente avaliando os aspectos negativos dos *smells* e que não devem ser seguidos (TAHIR et al., 2020). Digkas et al. (2020) estudaram se há relação entre práticas de qualidade de código e a limpeza de um novo código. Para isso, avaliaram 57 projetos do ecossistema *Apache*, em um total de 66.661 classes e 56.890 *commits*. Os resultados sugerem que escrever um novo código limpo (ou pelo menos mais limpo) pode ser uma estratégia eficiente para reduzir a densidade de dívida técnica, evitando a degradação do software ao longo do tempo. Os resultados também sugerem que os projetos que adotam políticas explícitas de melhoria

da qualidade (*e.g.,* discussões sobre a qualidade nas reuniões do time) estão associados a uma maior frequência de novos *commits* de códigos mais limpos. Os autores também defendem processos que monitorem a dívida técnica do novo código para controlar o acúmulo de dívida em um sistema. Estes resultados estão de acordo também com Aversano, Carpenito and Iammarino (2020), que sugere monitoramento contínuo da presença dos problemas para evitar um aumento crítico nas atividades de mudanças.

O *DR-Tools Code Health* pode ajudar o time a fazer estas investigações técnicas com base no código e planejar suas ações, sejam elas imediatas, planejadas ou até mesmo, ignorá-las.

Análise de Código na Evolução do Software: Conforme os sistemas de software evoluem, novos requisitos são identificados, *bugs* são corrigidos e novas funcionalidades são adicionadas. Este processo de evolução pode resultar em mudanças significativas no código, tornando-o mais complexo e difícil de manter ao longo do tempo (DIGKAS et al., 2018). Um dos principais desafios é acompanhar e compreender as mudanças no código ao longo do tempo, especialmente em projetos de grande escala com múltiplos desenvolvedores contribuindo simultaneamente (VALE et al., 2021). Outro desafio importante é garantir a qualidade e a segurança do código ao longo do tempo, à medida que novas funcionalidades são adicionadas e alterações são feitas, o que pode introduzir novos *bugs* (WANG; CHO; SONG, 2022), vulnerabilidades (JÜRJENS et al., 2019) e *smells* (HALEPMOLLASI; TOSUN, 2024).

Pela sua própria natureza de ser uma ferramenta *open-source* e que gera resultados em formato aberto, o *DR-Tools Code Health* pode ser usado em conjunto com outras ferramentas, como *git*, fornecendo *insights* valiosos sobre métricas, *smells* e elementos de código em diferentes versões, gerados dados para uma posterior análise da evolução do projeto.

Assim, a aplicação prática do *DR-Tools Code Health* visa apoiar necessidades recentes da comunidade de pesquisa e da indústria, apontados por Catolino (2020). Os pesquisadores podem usar e adaptar a ferramenta para avaliar e estudar os problemas no código que são críticos e, consequentemente, os que não são, na sustentabilidade dos projetos. Já os profissionais podem usar o modelo para apoiar na monitoria e evolução dos problemas no código, avaliando de forma sistemática a necessidade de atuação.

5 EXPERIMENTOS

Nos capítulos anteriores, foram apresentados e discutidos vários aspectos relacionados a qualidade, problemas de *design* de código, abordagens de detecção, priorização e ferramentas. Também foi descrito o *DR-Tools Code Health*, abordagem proposta para priorização dos *smells* em elementos de código. Contudo, para validar as questões de pesquisa apresentadas na Seção 1.3, foram definidos dois experimentos. Também, são apresentados e discutidos os resultados e ameaças à validade de cada um dos experimentos, além de depoimentos de uso do *DR-Tools Code Health*.

Qualquer abordagem recém-proposta deve seguir um forte processo de validação para garantir sua confiabilidade e validade (ABUHASSAN; ALSHAYEB; GHOUTI, 2020). Existe uma grande lacuna em relação a pesquisas acadêmicas de ES e a indústria, conforme vários estudos (GAROUSI et al., 2019; GAROUSI; SHEPHERD; HERKILOGLU, 2019; GAROUSI et al., 2019), incluindo áreas como qualidade e *design* de software (ALFAYEZ et al., 2020). Assim, este processo de validação deve ser realizado em colaboração com a indústria de software e profissionais para aumentar sua aceitação.

O processo experimental proposto contempla experimentos usando projetos da indústria com participação de profissionais e projetos *open-source*, visando minimizar o impacto de tamanho, domínio e tipo do conjunto de dados (WOHLIN et al., 2012). No primeiro experimento, é realizado um estudo misto para verificar a percepção sobre o método de priorização proposto ao apoiar desenvolvedores de software. Este experimento foi inspirado em outros experimentos conduzidos em trabalhos como (FONTANA; BRAIONE; ZANONI, 2012; YAMASHITA; MOONEN, 2013a; ARCOVERDE et al., 2013; TAIBI; JANES; LENARDUZZI, 2017; OIZUMI et al., 2017; PECORELLI et al., 2020; BÖRSTLER et al., 2023). No segundo experimento, inspirado em outros trabalhos (CEDRIM et al., 2017; BESSGHAIER; OUNI; MKAOUER, 2021; SAS; AVGERIOU; UYUMAZ, 2022), é desenvolvido um estudo longitudinal exploratório sobre a prática de priorização, além de como os *smells* têm impactado a qualidade durante a manutenção e evolução de software.

Os experimentos foram realizados entre outubro de 2023 a abril de 2024.

5.1 Experimento 1: Pesquisa com Profissionais da Indústria

O experimento 1 tem por objetivo avaliar se o método de priorização proposto auxilia os desenvolvedores a identificar os elementos de código mais problemáticos de um projeto. Afinal, é fundamental considerar o conhecimento dos desenvolvedores, aproveitando seu conhecimento sobre o projeto (LEPPANEN et al., 2015b). Também, avaliou-se a precisão do método para apoiar a priorização e o grau de concordância das respostas dos desenvolvedores com o resultado apresentado pelo método.

Este experimento busca responder a seguinte questão de pesquisa:

Questões de Pesquisa do Experimento 1

RQ1 - A abordagem de priorização proposta ajuda os desenvolvedores a identificar os elementos de código (classes, métodos) mais problemáticos?
RQ2 - Ao confrontar os resultados analisados pela abordagem de priorização proposta, muda a percepção dos desenvolvedores sobre os elementos de código mais problemáticos?

5.1.1 Protocolo

O experimento é apresentado, de forma resumida, na Figura 5.1.

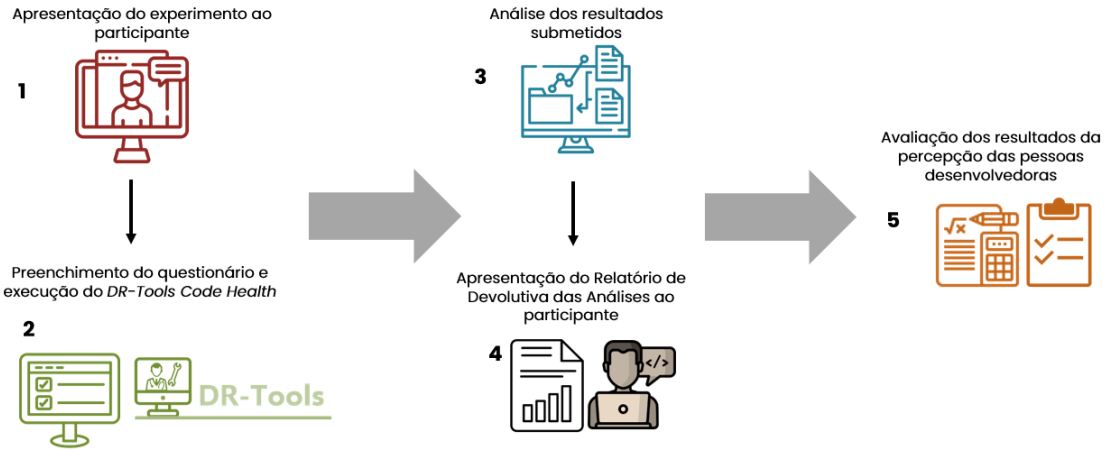
A população analisada engloba a seleção de elementos de código dos projetos escolhidos pelos profissionais para submissão e análise, considerando até 15 classes e 15 métodos, apontados como mais problemáticos, de acordo com a opinião dos desenvolvedores.

Para este experimento, considerou-se pessoas experientes em desenvolvimento de software e com conhecimento profundo nos projetos submetidos. Cada participante pode submeter um projeto. Em linha gerais, tem-se as seguintes etapas:

1. **Apresentação do experimento para o participante**. Após a apresentação e concordância na participação, o contato é feito via e-mail (Apêndice J), liberando o acesso ao questionário (Apêndice I) e termo de consentimento (Apêndice K). Foi considerada a participação do desenvolvedor mais experiente no projeto, com experiência de carreira e conhecimento profundo do projeto.

   Para elaboração do *ground truth ranking*, ficou definido que o desenvolvedor que

Figura 5.1 – Resumo do Experimento 1



Fonte: O Autor

fosse preencher o questionário discutiria previamente com outros membros do time, para chegar em um consenso sobre as classes e métodos mais problemáticos a serem indicados;

2. **Preenchimento do questionário e execução do** *DR-Tools Code Health* (opção – *analyze*). No questionário, considerou-se a análise do perfil dos participantes, informações gerais do projeto, conhecimentos sobre qualidade e *smells* e identificação dos elementos de código mais problemáticos[1] segundo suas percepções;
3. **Análise das respostas**. De posse do preenchimento, o pesquisador analisa os resultados submetidos e elabora um relatório (Apêndice L) que contempla as informações passadas pelo participante e cruza as informações coletadas pela execução *DR-Tools Code Health*.

   O pesquisador avalia a *precision*[2] (BAEZA-YATES et al., 1999) das respostas fornecidas com os dados coletados com o *DR-Tools Code Health* (*TP* é um elemento indicado como problemático e confirmado pela análise das respostas; consequentemente, um *FP* é um elemento indicado como problemático, mas não confirmado pela análise), conforme Equação 5.1;

$$Precision_{GT} = \frac{TP}{TP + FP} \tag{5.1}$$

4. **Entrega do relatório das análises**. O relatório de devolutiva das análises é apresentado para o participante. O participante é convidado novamente a analisar os

[1] estes elementos são considerados o *ground truth ranking*, ou seja, a lista de elementos mais problemáticos do código

[2] indica o quão confiável ou exato é o valor de uma estimativa, medição ou previsão

dados. O objetivo é verificar se há concordância com os resultados apresentados e se eles mudaram sua percepção em relação aos elementos mais problemáticos do projeto;

5. **Análise e verificações**. Após essa segunda avaliação, os dados são compilados e analisados.

   Contudo, conforme etapa 5, ainda pretende-se avaliar novamente a *precision* (Equação 5.2) dos resultados obtidos após a discussão da devolutiva das análises, ao cruzar com a percepção dos desenvolvedores. Assim, considerou-se um *TP* um elemento de código identificado pelo *DR-Tools Code Health* e confirmado pelos desenvolvedores. Já um *FP* representa um elemento de código identificado pelo *DR-Tools Code Health*, mas não confirmado pelos desenvolvedores.

$$Precision_{Final} = \frac{TP}{TP + FP} \tag{5.2}$$

   Ainda na etapa 5, além da *precision*, é usado o coeficiente *Kappa* (LANDIS; KOCH, 1977) para medir o grau de concordância entre os resultados obtidos pelo *DR-Tools Code Health* a percepção do participante.

   A Equação 5.3 considerou p_0 como a taxa de aceitação relativa e p_e é a taxa hipotética de aceitação.

$$Kappa = \frac{p_0 - p_e}{1 - p_e} \tag{5.3}$$

   O resultado *Kappa* tem um valor máximo de 1, quando existe concordância quase perfeita, e tende a zero ou menos quando não há concordância entre eles. A Tabela 5.1 apresenta os valores considerados para a interpretação da métrica.

5.1.2 Desenvolvimento

Durante o processo de elaboração do questionário, visando aumentar a validade do construto e delimitar o escopo de investigação, foi definido que seriam avaliados *smells* na granularidade de *methods* e *types*. Portanto, *smells* no nível de *namespace* bem como investigações das co-ocorrências de *smells* não foram explorados neste estudo.

Ao definir uma versão inicial do questionário, foi executado um piloto com dois profissionais experientes, com mais de 20 anos em desenvolvimento de software. Eles

Tabela 5.1 – Valores *Kappa* e interpretação

Valor	**Interpretação**
< 0	Ausência de concordância
0.00 - 0.20	Concordância pobre
0.21 - 0.40	Concordância leve
0.41 - 0.60	Concordância moderada
0.61 - 0.80	Concordância substancial
0.81 - 1.00	Concordância quase perfeita

Fonte: (LANDIS; KOCH, 1977)

sugeriram alguns ajustes em relação a algumas perguntas e na modificação da ordem do questionário. Os participantes que validaram o questionário piloto concordaram que apresentar o questionário com a maioria das questões fechadas é uma boa estratégia para explorar o assunto. Portanto, ter opções para seleção auxilia os participantes para responder à pesquisa, ou seja, muitas questões abertas poderiam dificultar a obtenção de dados.

O questionário final foi composto por 22 questões, sendo todas obrigatórias. Em algumas questões (*e.g.,* questões 3 e 7) foram incluídas opções para que os participantes fornecessem respostas diferentes das listadas. Em geral, as questões são curtas, de múltipla escolha, seleção única, numéricas, texto livre e escala *Likert*. Nas questões 17 e 18, foi solicitado para os participantes indicarem métodos e classes consideradas mais problemáticas, segundo suas percepções. Na questão 22, é solicitado o envio do *dataset* coletado do projeto com o *DR-Tools Code Health*.

Diferentemente do que tem sido adotado como recomendação para tratar os participantes da pesquisa de forma oculta (BALTES; RALPH, 2022), neste experimento, foi necessário solicitar os dados de identificação, pois uma das etapas é a devolução das análises para posterior verificação.

Considerando isso, foram usadas três técnicas para contato, aplicação do questionário e coleta dos dados dos participantes. São elas (BALTES; RALPH, 2022):

- *Convenience Sampling:* seleção de participantes com base na disponibilidade. Seguindo esta técnica, utilizou-se uma grande rede de contatos de profissionais de

desenvolvimento de software para compartilhar o estudo e solicitar sua participação, dependendo de sua disponibilidade;

- *Purposive Sampling:* seleção de participantes de uma empresa específica, convidando-os para contribuírem com o estudo. Seguindo esta técnica, por exemplo, foi contatado um diretor de desenvolvimento de uma grande empresa de software do Brasil, que encaminhou internamente o questionário para os times de desenvolvimento de sua organização. Além disso, foi divulgado o estudo e solicitado a participação em comunidades online de desenvolvedores[3];
- *Snowballing Sampling:* identificação de alguns participantes da população e solicitação para identificar outros potenciais participantes que poderiam contribuir com estudo. Conforme esta técnica, foi solicitado a alguns participantes da amostragem por conveniência que encaminhassem o estudo para colegas e colaboradores.

Todas as técnicas empregadas foram de amostragens não probabilísticas, ou seja, métodos de amostragem que não empregam aleatoriedade. Também, para as informações quantitativas, é aplicado técnicas de estatística descritiva, visando analisar e resumir os resultados. Optou-se, durante o contato os participantes, de selecionar desenvolvedores com mais experiência em relação projeto. Assim, é possível obter as informações mais importantes relacionadas ao projeto. No caso de dúvidas, o participante poderia consultar outros membros do time.

Após o preenchimento e submissão do questionário pelo participante, os dados são analisados e o *dataset* é baixado. Então, é desenvolvido o relatório de análise do projeto, cruzando as informações submetidas pelo participante com os dados coletados pelo *DR-Tools Code Health*. Em relação as métricas, são considerados os dados de sumário do projeto e dados estatísticos relacionados a métricas de *types* e *methods*. Em relação aos *smells*, foram considerados os dados sumarizados dos *smells* e quantidade de instâncias detectadas por tipo de *smell*. Já para apresentar o *ranking*, foi selecionados dados referentes aos CDIs dos *methods*, *types* e *type composition*, ou seja, a composição dos *smells* da granularidade de *methods* com os *smells* de granularidade de *types* encontrados. Para fins de análise, são considerados apenas elementos de código referentes ao domínio do negócio, ignorando qualquer elemento de código referente a testes. Estes dados são incluídos no relatório de devolutiva das análises.

Os dados são tabulados e registrados em uma planilha do *Google Sheets*. Além

[3]Comunidades como *RSJug* (Comunidade de Desenvolvedores Java do RS), *TDC* (The Developers Conference) e contatos no *LinkedIn*

disso, já é calculada a *precision* inicial, antes da verificação. Todos os dados podem ser acessados no Apêndice H.

O relatório de devolutiva das análises é apresentado ao participante. É solicitado para o participante confrontar a sua percepção com os dados gerados pelo *DR-Tools Code Health* para métodos, classes e composição deles. Também, o participante pode incluir comentários extras relacionados a sua percepção. Após a conferência e verificação do participante, os dados são tabulados na mesma planilha. Agora, além da *precision* final, também é avaliado a concordância com o coeficiente *Kappa*. Para fazer esse cálculo, foi escrito um programa em *Python* usando a biblioteca *sklearn*[4].

5.1.3 Resultados e Discussão

A seguir, são apresentados os resultados obtidos após a análise, seguindo o protocolo descrito na Subseção 5.1.1. Os resultados foram agrupados para fins de organização em informações gerais/dados demográficos, qualidade/*smells*, dados dos projetos e, finalmente, a percepção dos desenvolvedores.

Informações Gerais e Dados Demográficos: O experimento teve início em dezembro de 2023, com finalização em abril de 2024. Foram contatados 55 pessoas atuantes em desenvolvimento de software. Destas 55 pessoas, 48 (87%) responderam ao contato, sendo que 25 (52%) se predispuseram a participar inicialmente do experimento. No entanto, foram obtidas 15 (60%) respostas e submissões de projetos para análise.

A Tabela 5.2 apresenta as principais características dos participantes do experimento, descrevendo a distribuição de cargos/posição atual, tempo de carreira e formação. Com base nos dados, os **profissionais são desenvolvedores experientes** (se for considerado o ponto de corte de 10 anos de experiência, compreende em 93.3% dos participantes) com **atuação direta em desenvolvimento de software** (93.3%, se desconsiderar um dos participantes com posição de *C-level*). Duas posições foram enquadradas como outros, que são *open source contributor* e engenheiro de software. Também, é possível notar que a **formação acadêmica entre os participantes é balanceada entre graduação e especialização/MBA**. Nenhum participante do experimento possui formação acadêmica mais avançada, como mestrado ou doutorado.

Em relação ao **tempo de atuação na empresa, tem-se uma variação de tempo considerável** entre os participantes. A pessoa com mais tempo de empresa tem 16 anos

[4]Mais informações em <https://scikit-learn.org/stable/>

Tabela 5.2 – Perfil dos participantes

Característica	Descrição	Quantidade	%
Cargo/Posição	CTO/CIO	1	6.67
	Pessoa Arquiteta	2	13.33
	Pessoa Desenvolvedora Líder	2	13.33
	Pessoa Líder Técnica	2	13.33
	Pessoa Desenvolvedora Sênior	6	40.00
	Outra	2	13.33
Tempo de Carreira	Entre 5 e 10 anos	1	6.67
	Entre 10 e 15 anos	7	46.67
	Entre 15 e 20 anos	4	6.67
	Mais de 20 anos	3	20.00
Formação	Graduação	7	46.67
	Especialização/MBA	8	53.33

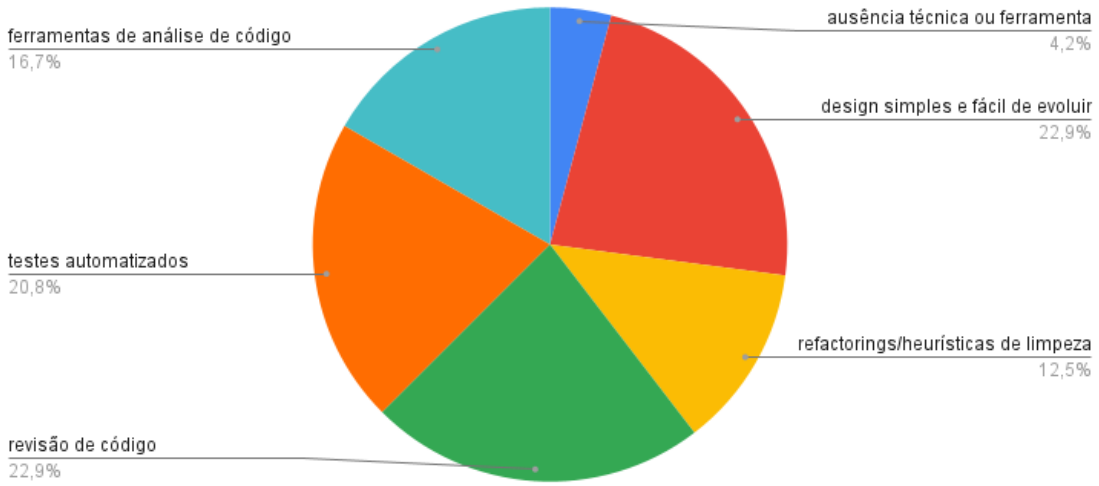
Fonte: O Autor

de serviço enquanto a pessoa com menos tempo, possui 1 ano de serviço na empresa (média=6.73; mediana=5; DP=5.42). **A maioria das empresas dos participantes é enquadrada como empresa tradicional** (12 empresas; 80%) com mais de 10 anos de mercado, enquanto apenas 3 (20%) são *startups*. O **maior foco de atuação é no desenvolvimento de produtos** (8; 53.33%), seguido dos serviços de *outsourcing* (3; 20.00%) e outros (3; 20.00%). Dentre os outros serviços descritos estão os serviços financeiros e bancários. Apenas uma empresa atua como fábrica de software (6.67%).

Qualidade e Smells: Os participantes foram questionados sobre suas percepções sobre qualidade e *smells*. Em relação a qualidade, o primeiro questionamento foi sobre quais técnicas eles utilizam em conjunto com o time para a qualidade do código. De acordo com a Figura 5.2, **os participantes buscam aplicar e disseminar práticas como *code review* e princípios de *design* simples** que propiciem, de forma saudável, a evolução do código (ambos com 22.9%), **seguido de testes automatizados** (20.8%).

A prática de *code review* é uma prática sistemática e amplamente utilizada (DAVILA; NUNES, 2021), realizada por pares ou até mais pessoas, possibilitam identificar e corrigir problemas de qualidade, promovendo a colaboração e aprendizado no time. Busca-se incentivar *commits* curtos, com *pull requests* pequenos, facilitando este processo (SEEMANN, 2021). Ainda, esta prática pode ser potencializada tanto pelo uso do *design*

Figura 5.2 – Técnicas e ferramentas utilizadas para qualidade de código



Fonte: O Autor

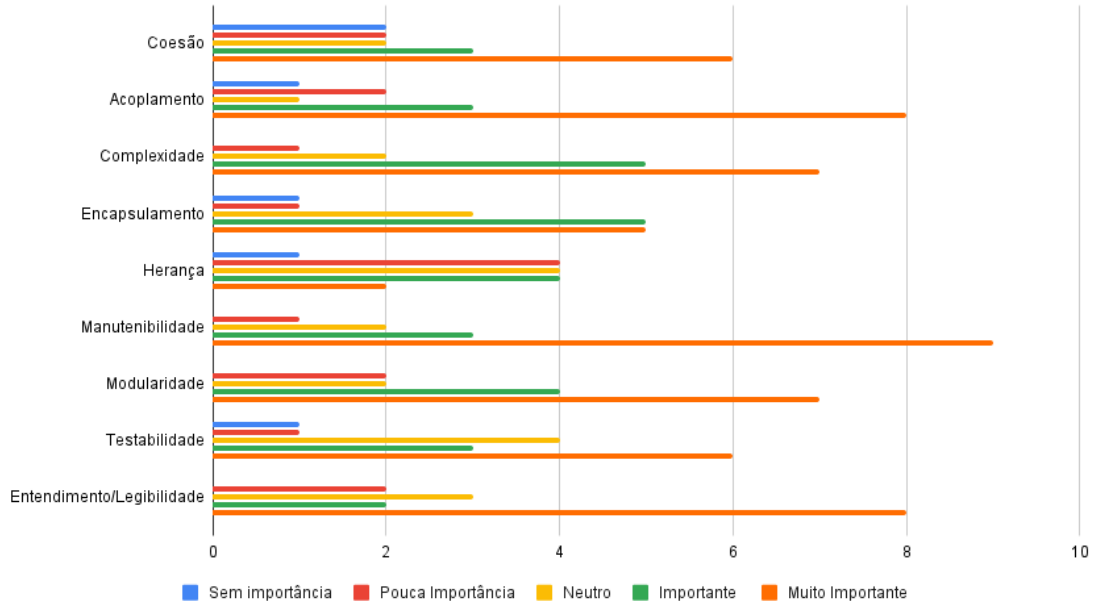
simples quanto pela presença de testes automatizados. *Design* simples é uma prática que consiste em aplicar princípios e técnicas, como SOLID (LAPLANTE; KASSAB, 2022), visando uma melhor organização e distribuição do código, além de manter um estilo de código que seja adotado por todo o time, facilitando a legibilidade e consistência no código (AGHAJANI et al., 2020). Os testes automatizados em diferentes níveis auxiliam neste processo, ao verificar de forma contínua o funcionamento do código, garantindo sua integridade e fornecendo *feedback* rapidamente sobre seu estado (KEMPPAINEN, 2022). Também, cabe destacar **o uso de ferramentas de análise de código** (16.7%) **e técnicas de refatoração e heurísticas de limpeza** (12.5%), consideradas fundamentais para a manutenção e evolução do software. Apenas 4.2% dos participantes dizem não utilizar qualquer técnica ou ferramenta para apoiar práticas de qualidade.

No segundo questionamento sobre qualidade, foi solicitado aos participantes que classificassem do *menos ao mais significativo* os atributos de qualidade para o seu projeto (Figura 5.3).

Manutenibilidade (considerado o *mais significativo*), acoplamento e entendimento/legibilidade (empatados na segunda posição) foram os classificados como *mais significativos*, respectivamente. Na terceira posição, aparecem complexidade e modularidade. **Complexidade, manutenibilidade, modularidade e entendimento/legibilidade não receberam classificação como *sem importância***, o que denota sua relevância para a qualidade. **Herança e encapsulamento foram os atributos de qualidade classificados como *menos significativos***.

Para melhor avaliar as respostas sobre os atributos de qualidade, agrupou-se os

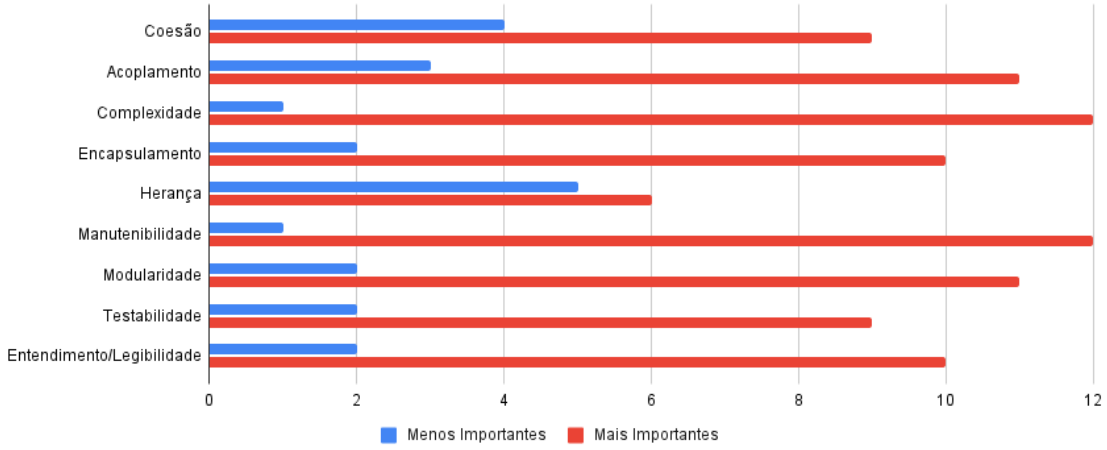
Figura 5.3 – Classificação dos atributos de qualidade, de acordo com sua importância segundo os participantes



Fonte: O Autor

extremos das respostas, somando os valores, descartando a parte intermediária (*neutro*). Portanto, os valores relacionados a *sem importância* e *pouca importância* foram agregados em uma nova categoria (*menos importantes*), assim como os valores *importante* e *muito importante* também (categoria *mais importantes*). Esta nova classificação pode ser observada na Figura 5.4.

Figura 5.4 – Classificação dos atributos de qualidade, agrupados em menos e mais importantes



Fonte: O Autor

Conforme pode ser observado, os destaques ficam para **manutenibilidade e complexidade, considerados os atributos de qualidade *mais importantes***. Para esta obser-

vação, considerou-se não somente a maior classificação, mas também a menor. Ambos tiveram as mesmas classificações nos dois agrupamentos. A **herança foi considerada a *menos importante***.

Finalmente, uma última questão foi apresentada aos participantes para que eles pudessem indicar, segundo suas percepções, como eles classificariam a qualidade geral do projeto analisado. **A maioria dos projetos receberam uma classificação intermediária, de qualidade aceitável (6; 40.00%)**. Outros 5 (33.33%) projetos foram classificados com baixa qualidade e 2 (13.33%) com péssima qualidade. Apenas 2 projetos foram classificados com boa e ótima qualidade (6.67%, respectivamente). Se for considerado como ponto de corte o nível intermediário para baixo (qualidade aceitável para péssima qualidade), 86.66% dos projetos se enquadram nesta faixa.

Na primeira questão relacionada a *smells*, foi perguntado aos participantes em relação ao seu conhecimento sobre o tema. **A maioria (9; 60.00%) considerou ter conhecimento intermediário do tema**, conhecendo seu conceito e sendo capaz de identificá-lo no código. A outra fatia ficou dividida igualmente: 20% dos participantes se identificaram com conhecimento mais avançado sobre o tema, estudando e sendo capaz de ajudar o time nas detecções e priorizações. Os outros 20% dos participantes julgaram seu conhecimento básico, apenas conceitual. Conforme apresentado por Yamashita and Moonen (2013a), estes resultados de certa forma já eram esperados, uma vez que, na seleção dos participantes, buscou-se pessoas mais experientes dentro dos projetos submetidos. Já se sabe que, mesmo com o conhecimento dos desenvolvedores em relação aos *smells*, esta percepção pode mudar na prática (TAIBI; JANES; LENARDUZZI, 2017), sobremaneira levando em consideração o contexto e priorização.

A segunda questão sobre *smells*, os participantes responderam, segundo suas percepções, quais *smells* são considerados mais problemáticos. Como era uma questão aberta de resposta, os participantes responderam de formas diferentes. Os dados completos obtidos podem ser acessados no Apêndice H. A Tabela 5.3 ilustra alguns exemplos de resposta e como foram codificados para análise.

Após esta codificação, os *smells* foram contabilizados para identificar os mais citados pelos participantes. Com isso, na Figura 5.5, é ilustrada uma nuvem de palavras para representar os *smells* mais citados.

Também, conforme outros trabalhos já apontaram (LACERDA et al., 2020), ***duplicated code*, *long method* e *large class/god class* são os *smells* mais citados** e, segundo os participantes, **considerados os mais problemáticos**.

Tabela 5.3 – Exemplos de respostas sobre *smells* e as respectivas codificações

ID	Resposta	Codificação
E	Blocos de códigos muito longos, nome de métodos que dão a entender que fazem X mas fazem Y, acredito que esses são os piores para mim	LongMethod, BadNames
N	Código idêntico ou semelhante aparece em vários lugares, uso direto de valores numéricos sem explicação ou constantes nomeadas, funções que são excessivamente longas e realizam muitas tarefas, uma classe que é muito grande e centraliza muitas responsabilidades	DuplicatedCode, MagicNumber, LongMethod, LargeClass
O	Para mim, os smells mais problemáticos são os que dificultam o entendimento do código, impactando na manutenção, por exemplo, long method, large class, bumpy road	LongMethod, LargeClass, BumpyRoad

Fonte: O Autor

Figura 5.5 – Nuvem de palavras dos *smells* citados pelos participantes

Fonte: O Autor

Finalmente, foi perguntado para os participantes o que eles consideraram para priorizar os elementos de código (classes e métodos) indicados no projeto. As respostas foram em formato textual aberto. Assim, foi necessário fazer uma análise temática (BRAUN; CLARKE, 2012). Após a identificação das respostas e entendimento dos dados, eles foram codificados e categorizados em dois temas: ***baseado em experiência*** **e** ***baseado em princípios de design***. No tema *baseado em experiência*, os **participantes relataram o uso do seu conhecimento prévio em projetos anteriores e práticas do dia a dia**. O tema *baseado em princípios de design*, os **participantes utilizaram princípios do *clean code*** (MARTIN; MARTIN, 2007) e **heurísticas conhecidas como considerar o tamanho das estruturas ou encadeamento interno do código**. **Apenas um participante relatou o uso de ferramenta de análise de código** para apoiar a priorização. O baixo uso de ferramentas de análise de código pode ser explicado por alguns fatores: os desenvolvedores, em sua maioria, confiam no seu entendimento do projeto, o grande

número de falsos positivos e violações apresentadas (VASSALLO et al., 2020; LENARDUZZI et al., 2021) e a dificuldade de conectá-las ao fluxo de trabalho (MARCILIO et al., 2020).

Projetos: Os projetos submetidos para análise são de diferentes domínios e tamanhos. Na Tabela 5.4, são apresentados os seguintes dados por projeto: nome[5], número de linha de código (SLOC), número de pacotes (*namespaces*), número de classes (*types*), número de métodos (*methods*) e descrição geral do projeto. Já na Tabela 5.5, são exibidos os seguintes dados sobre os *smells*: quantidade total de elementos de código com *smells* (*smells*), quantidade total de elementos de código com mais de um *smell* (*2+ smells*), número de *namespaces* com *smell* (*SN*), número de *namespaces* com mais de um *smell* (*2+ SN*), número de *types* com *smell* (*ST*), número de *types* com mais de um *smell* (*2+ ST*), número de *methods* com *smell* (*SM*) e número de *methods* com mais de um *smell* (*2+ SM*).

Considerando a classificação para tamanho de projetos de software de Radjenovic et al. (2013), **3 (20%) projetos são considerados grandes, 5 (33%) projetos considerados médios e 7 (47%) projetos definidos como pequenos**. Em geral, foram submetidos para análise projetos de tamanhos diferentes, o que é ponderado como positivo em experimentos desta natureza (WOHLIN et al., 2012). O **maior projeto (identificado como projeto *E*) apresentou 1.65 milhões de LOC**, enquanto o **menor projeto (projeto *J*) tem LOC de 900** (média=169 KLOC; mediana=16.8 KLOC; DP=421.1 KLOC).

Dentre os projetos analisados, a **maioria são produtos ou conjunto de serviços (4 projetos; 26.67%, ambos)**, seguido de 3 projetos (20%) que são soluções internas da empresa. Ainda, 2 (13.33%) são bibliotecas/componentes e 2 (13.33%) foram enquadrados como outros (ferramenta utilitária para desenvolvedores e sistema principal da empresa).

Conforme a Tabela 5.6, é apresentado a listagem dos projetos analisados e a densidade dos *smells* nos elementos de código. No total, **foram identificados 34280 instâncias de *smells* considerando todas as granularidades**, com **13061 elementos de código que possuem mais de um *smell***.

São destacados os 3 projetos com maior percentual de densidade do *smells*, com uma ou mais instâncias presentes. A densidade dos *smells* tem sido descrita de diferentes formas na literatura. Fontana, Braione and Zanoni (2012) apresenta a densidade dos *smells* pela razão do número de *smells* por versão de um mesmo projeto. A forma mais

[5]por razões de confidencialidade, os projetos foram identificados por letras, sendo suprimido o nome original do projeto

Tabela 5.4 – Lista de Projetos considerados no Experimento 1, com destaque para o maior e menor projeto

Projeto	SLOC	Namespaces	Types	Methods	Descrição do Projeto
A	184362	118	2328	23902	Gestão do ambiente no-code do cliente, manutenção de cadastros, gestão das tarefas de campo e criação de atividades (abstração da solução da empresa sobre o que é feito em campo)
B	13573	55	296	1775	O projeto é uma integração entre APIs, o que esta sendo analisado é apenas uma parte do processo, atualmente para realizar a integração existem pelo menos 20 microsserviços
C	255607	229	3179	31017	Plataforma full-stack do Postgres as a service (cloud e on premise)
D	9137	22	128	9137	Driver de comunicação com sistema SAM(*Senior Access Management*) para controladores de acesso e ponto
E	**1653783**	**586**	**9156**	**145108**	O projeto é o sistema principal da empresa, onde é possível contratar diversos produtos oferecidos, conta bancaria, empréstimos cartão, etc. É usado em diversas cooperativas do Brasil.
F	16800	7	123	929	Projeto voltado para o setor logístico para gerenciar todo o ciclo de chegada, pesagem, carregamento e saída de caminhões
G	272031	479	2763	27726	É um módulo que faz parte de um ERP especializado para a indústria da construção. Esse módulo é usado para a gestão financeira da empresa, mais especificamente das contas a pagar da empresa. Esse módulo existe a cerca de 18 anos, e combina tecnologias Java EE como EJB e também Hibernate. Em produção, ele faz parte de um produto maior que é uma aplicação Web Java EE, que roda no servidor de aplicação JBoss. É usado por mais de 4000 construtoras e incorporadoras de todo o Brasil, com dezenas de milhares de usuários no total.
H	1171	2	11	55	O objetivo do projeto é realizar a integração entre 2 sistemas através uma ponte entre 2 bancos de dados para troca de informação.
I	2452	13	67	236	O projeto tem por objetivo extrair métricas de custo financeiro apartir de relatórios CSV disponibilizados pela Amazon Web Services
J	**900**	**11**	**23**	**52**	O sistema consulta o valor atual de várias moedas estrangeiras e cria um relatório resumido com o valor que a moeda estava em alta, em baixa, e o valor do fechamento no dia
K	29214	202	551	1778	Componente de comunicação com a B3 trabalhando com mensageria kafka, bancos de dados oracle e mongo
L	15131	34	218	1348	Este serviço expõe todos os meta dados de um System of Record, para fins de execução de relatório. É uma aplicação em GraphQL, com dois níveis de cache (container/redis), que serve todos os outros componentes do ecossistema da aplicação.
M	68882	132	869	9083	O objetivo do projeto é um sistema de gestão acadêmica que é utilizado por milhares de alunos. Ele foi criado utilizando arquitetura monolítica e utiliza JAVA EE 6 e servidor de aplicação JBOSS.
N	17821	48	389	1061	A aplicação tem como foco o envio de informações sobre apontamentos de Riscos Socioambientais (RSA) aos quais empresas ou clientes de instituições financeiras podem estar expostos. Esses dados serão utilizados pelo Banco Central do Brasil (Bacen) como parâmetros na análise de crédito de empresas e pessoas físicas.
O	5171	27	116	226	O projeto analisado será responsável pelo streaming de dados de uma aplicação para outra, onde a primeira possui dados de colaboradores dos clientes e a outra é responsável pela geração de relatórios customizados para cada cliente.
Total	**2546035**	**1965**	**20217**	**253465**	

Fonte: O Autor

Tabela 5.5 – Lista de Projetos com *smells* detectados nas granularidades no Experimento 1

Projeto	**Smells**	**2+ Smells**	**SN**	**2+ SN**	**ST**	**2+ ST**	**SM**	**2+ SM**
A	1634	267	17	6	536	106	1081	155
B	128	17	1	1	30	6	97	10
C	2764	305	50	3	606	86	2108	216
D	136	65	5	0	23	6	108	59
E	24986	11048	86	21	3450	1028	21450	9999
F	339	165	3	1	18	6	318	158
G	2835	817	12	3	586	101	2237	713
H	24	12	0	0	1	0	23	12
I	11	0	1	0	2	0	8	0
J	8	2	0	0	1	0	7	2
K	141	5	1	0	63	3	77	2
L	117	27	1	0	24	3	92	24
M	945	276	11	1	220	65	714	210
N	135	31	4	0	36	3	95	28
O	77	24	2	0	29	3	46	21
Total	**34280**	**13061**	**194**	**36**	**5625**	**1416**	**28461**	**11609**

Fonte: O Autor

utilizada e relatada em diferentes trabalhos tem sido a razão entre o número de *smells* pelo LOC do projeto (SHARMA; SPINELLIS, 2018; OIZUMI et al., 2019). No entanto, é importante considerar esta métrica quando se analisa projetos considerados de mesmo tamanho (YAMASHITA; COUNSELL, 2013). Ao usar esta medida em projetos de diferentes tamanhos, isso pode dar uma maior vantagem a projetos maiores, levando a uma avaliação ingênua.

Então, para este experimento, considerou-se o percentual da densidade dos *smells*, computado pela razão do número dos *smells* pela a soma da quantidade dos elementos de código do projeto, multiplicado por 100. Também, usou-se a mesma métrica considerando elementos de código que possuem mais de um *smell*. Assim, é possível avaliar que o **projeto *H* foi o que apresentou maior densidade de *smells*, considerando elementos de código com um único *smell***, ultrapassando em quase 3 (2.95) vezes a média e 4 (3.91) vezes a mediana. Se considerar os **elementos de código que apresentaram mais de um *smell*, o mesmo projeto** também ultrapassou em mais de 4 (4.28) vezes a média e mais de 8 (8.56) vezes a mediana. Ainda, é possível observar que o projeto *E* foi o terceiro em densidade de *smells*, considerando mais de um *smell* no mesmo elemento de código. Esta informação é relevante porque pode indicar onde e quais elementos de código apresentam mais possíveis problemas de *design*. Conforme Palomba et al. (2018a), estas estruturas de código que contém mais de um *smell* são mais propensos a mudanças e falhas. Portanto, merecem atenção do time ao considerar as melhorias de código visando a manutenibilidade.

A **maioria dos sistemas submetidos foi definida como serviços** (9; 52.94%), isto é, aplicações orientada a serviços, serviços SOAP e REST. Na sequência, **as aplicações WEB aparecem em segundo** (5; 29.41%), seguida da classificação outros (3; 17.65%), incluindo *middleware*, serviço de integração de sistemas e aplicação CLI. Nenhum dos sistemas avaliados foi classificado como aplicação *desktop* ou *mobile*. Em dois casos, os sistemas foram classificados como aplicação WEB e serviços (projetos *G* e *M*).

Também, foi questionado sobre o tamanho das equipes de desenvolvimento e manutenção destes projetos. Foi considerado um time pequeno com até 3 pessoas. O time de tamanho médio possui entre 4 a 6 pessoas. O time considerado grande tem mais de 6 pessoas. A **maioria dos projetos foi desenvolvida e mantida por times de tamanho médio** (7; 46.67%), seguido de times pequenos (5; 33.33%) e times de grandes (3; 20.00%).

Dos **5 projetos de tamanho médio, 3 tinham times de tamanho médio** (ou seja, 60%). Os outros dois intercalaram entre time grande e pequeno. Nos **projetos classifica-**

Tabela 5.6 – Projetos e o respectivo percentual de densidades dos *smells* por elemento de código

Projeto	**Densidade de *smells* por Elemento de Código**	**Densidade de mais de *smell* por Elemento de Código**
A	6.20	1.01
B	6.02	0.79
C	8.02	0.88
D	1.46	0.69
E	16.13	**7.13**
F	**32.01**	**15.58**
G	9.15	2.63
H	**35.29**	**17.64**
I	3.48	0
J	9.30	2.32
K	5.57	0.19
L	7.13	1.68
M	9.37	2.73
N	9.01	2.06
O	**20.86**	6.50
Média	**11.94**	**4.12**
Mediana	**9.01**	**2.06**
Desvio Padrão	**20.00**	**5.48**

Fonte: O Autor

dos como pequenos, **4 (57.14%) também foram desenvolvidos e mantidos por times pequenos**. Os outros 3 projetos pequenos foram desenvolvidos por times médios. Um indicativo deste cenário é que possivelmente esses times médios mantenham outros sistemas e projetos. Já nos **projetos considerados grandes, 2/3 (66.67%) são mantidos por times grandes.** Assim, é possível observar com base nos projetos analisados neste experimento que **existe uma certa relação entre tamanho dos projetos com os tamanho dos times** (entre 57.14 e 66.67%).

Essa **relação entre o tamanho do time de software e o tamanho do projeto é complexa e multifacetada**. O tamanho do time de software geralmente reflete a complexidade do projeto, a quantidade de trabalho a ser realizado e os recursos disponíveis. Projetos maiores e mais complexos geralmente exigem times maiores para lidar com a carga de trabalho e garantir que todas as áreas do projeto sejam adequadamente atendidas. Além disso, o tamanho da equipe pode ser influenciado pela diversidade de habilidades necessárias para o desenvolvimento e manutenção, incluindo desenvolvedores, testadores, *designers* de UX/UI, entre outros. Por outro lado, o tamanho do projeto é uma medida quantitativa da extensão do software. Projetos maiores tendem a ter mais elementos de código devido à complexidade e ao escopo das funcionalidades implementadas.

No entanto, a relação entre o tamanho do projeto e a eficácia do desenvolvimento nem sempre é linear. Projetos com muito LOC podem ser mais difíceis de manter e gerenciar, especialmente se a estrutura do código não for bem organizada (SCHOLTES; MAVRODIEV; SCHWEITZER, 2016). A eficácia do desenvolvimento de software depende de uma série de fatores, incluindo a comunicação e a colaboração dentro do time, a clareza dos requisitos do projeto, a qualidade do código produzido e a capacidade de gerenciamento de mudanças (SCOTT; CHARKIE; PFAHL, 2020). Outro aspecto a ser considerado é a escala do projeto e a distribuição geográfica do time também desempenham um papel importante (SKELTON; PAIS, 2022). Projetos com times remotos e atuando globalmente podem exigir equipes maiores para coordenar diferentes fusos horários e culturas de trabalho, enquanto projetos menores podem ser mais ágeis e adaptáveis com times pequenos e multifuncionais (SKELTON; PAIS, 2019).

Percepção dos Desenvolvedores: Nesta parte, são agrupados os dados referentes a percepção dos participantes. Foram realizadas duas etapas: i) análise prévia das classes e métodos fornecidos como mais críticos; ii) validação dos resultados apresentados pelo *DR-Tools Code Health* e avaliação da mudança de percepção dos problemas nos elementos de código. Portanto, o objetivo é trazer subsídios para responder as RQs.

Na primeira etapa, as classes e métodos considerados críticos pelos participantes são comparados com os dados coletados pelo *DR-Tools Code Health* e submetidos junto com o questionário. Estes dados são registrados em uma planilha e analisados (Apêndice H). Depois, é calculado o *precision* antes (denominada de *precision before*) da devolutiva dos resultados para os participantes. O relatório de devolutiva das análises (Apêndice L) apresenta os dados coletados do questionário e do projeto como métricas, *smells* e priorização dos elementos de código (*methods, types, types + methods*) informados pelo participante e identificados pelo *DR-Tools Code Health*.

Na segunda etapa, os participantes indicam se confirmam os dados identificados pelo *DR-Tools Code Health* deveriam estar entre os elementos de código priorizados. Também, foi solicitado ao participante responder se os resultados apresentados pelo *DR-Tools Code Health* mudaram a sua percepção sobre os elementos de código mais problemáticos em cada granularidade. Finalmente, o participante também pode tecer comentários sobre os resultados apresentados. Assim, foi calculado a *precision* depois da confirmação (denominada de *precision after*), juntamente com o coeficiente *kappa* de concordância.

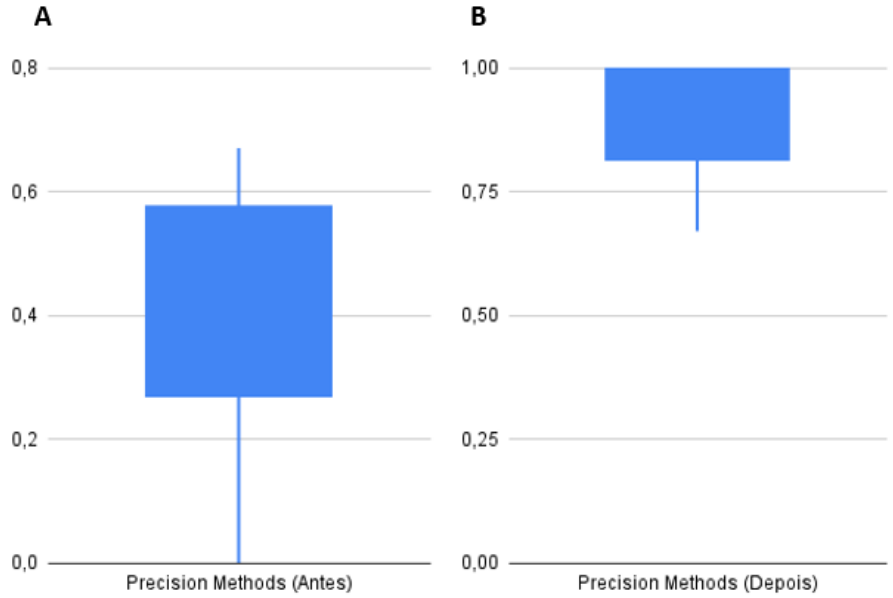
Em relação a *granularidade de métodos* (Figura 5.6), foi obtida uma *precision before* média da confirmação de 0.38 (mediana=0.40; DP=0.22). **Os projetos *H* e *J* apresentaram a maior *precision before* (0.67)**. **Dois projetos** (projetos *C* e *N*) **apresentaram 0.0 de *precision before***. A *precision* aqui foi utilizada como linha de base para uma futura comparação para confirmação da avaliação posterior.

Após a confirmação, **obte-se uma *precision after* médio de 0.90** (mediana=1.00; DP=0.12), refletindo uma uniformidade considerável nos dados. **Oito projetos apresentaram *precision after* máxima** (projetos *A, B, C, E, L, M, N* e *O*). Estes resultados sugerem uma eficácia consistente da abordagem utilizada nesses projetos, fortalecendo a confiança na qualidade e confiabilidade dos resultados produzidos. **O projeto que apresentou menor *precision after* foi o *H*, mantendo o mesmo valor (*precision before* e *after* de 0.67)**. Apesar disso, é interessante notar que o participante deste projeto relatou que a ferramenta poderia contribuir no desenvolvimento[6]. Esse *feedback* indica que, apesar da *precision after* relativamente baixa, ainda há percepção de valor e utilidade *DR-Tools Code Health*. Observa-se, portanto, que outros aspectos além da *precision* podem estar contribuindo para a avaliação da ferramenta, como funcionalidades adicionais de priorização ou integração com o processo de desenvolvimento. A Figura 5.6 apresenta

[6]A lista com alguns comentários dos participantes estão disponíveis na Tabela 5.8

um diagrama *box plot* que traz a distribuição dos dados dos projetos (antes e depois).

Figura 5.6 – *Box plot* da *precision* dos métodos: antes da avaliação (A) e depois da avaliação (B)

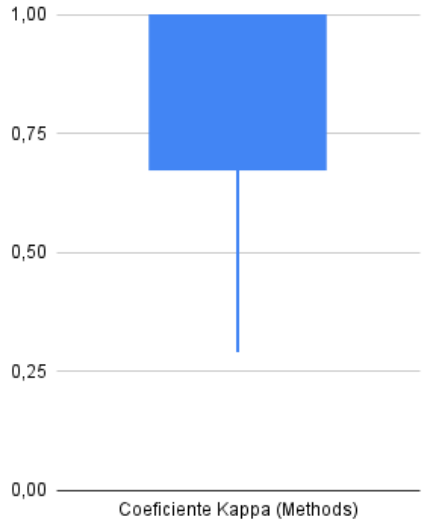


Fonte: O Autor

Em relação a **concordância com os resultados apresentados nos métodos** (Figura 5.7), encontrou-se um **coeficiente *kappa* médio de 0.82, considerada quase perfeita** (mediana=1.00; DP=0.23). Isto sugere um alto grau de concordância global nos resultados obtidos. Este valor é reforçado pela mediana de 1.00, indicando uma consistência notável nos dados, e pelo desvio padrão de 0.23, que revela uma variabilidade baixa em torno da média. Os **projetos *D* e *H* apresentaram os menores valores *kappa*, ou seja, concordância pobre** (valores 0.29 e 0.48, respectivamente). Ao examinar os resultados específicos de cada projeto, é possível identificar nuances importantes. Esses valores indicam uma concordância pobre entre os resultados desses projetos, sugerindo inconsistências significativas ou divergências substanciais. Já **os projetos *A, B, C, E, G, L, M, N* e *O* apresentaram *kappa* acima de 0.81**, considerada **concordância quase perfeita**. Isso indica uma consistência alta nos resultados desses projetos para métodos, com uma concordância quase completa. Esses resultados sugerem uma confiança substancial na precisão e consistência dos dados gerados por esses projetos, o que fortalece sua validade e confiabilidade. Portanto, essas variações sugerem a necessidade de investigar e corrigir as possíveis fontes de divergência nos projetos com baixa concordância, enquanto reforçam a confiabilidade dos dados nos projetos com alta concordância.

A análise dos resultados das classes (Figura 5.8) revela uma **melhoria significativa após a confirmação dos participantes**, com uma ***precision* média aumentando de 0.54** (*before*) **para 0.73** (*after*). Isto sugere uma maior precisão nos resultados obtidos,

Figura 5.7 – *Box plot* do coeficiente *kappa* para os métodos



Fonte: O Autor

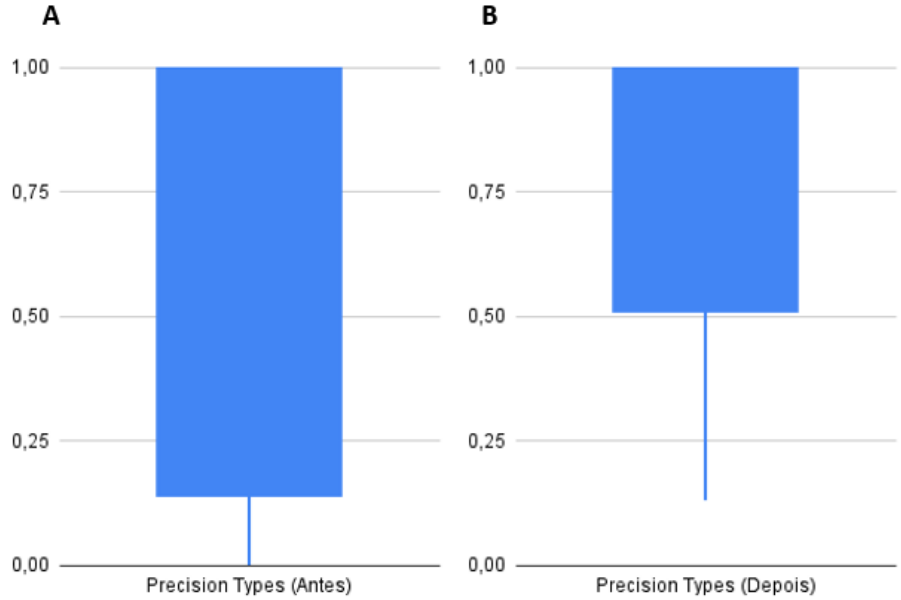
refletindo uma contribuição positiva do *DR-Tools Code Health* no processo de avaliação das classes dos projetos. Além disso, se **obteve uma mediana da *precision* com variação de 0.67 (DP=0.40) para 0.80 (DP=0.31) após a confirmação**, indicando uma consistência observável nos dados.

É possível observar que **os projetos B, D e N apresentaram uma *precision after* menor dos projetos analisados**, sugerindo que **o *DR-Tools Code Health* pode não ser igualmente eficaz em todos os contextos** ou pode haver questões específicas que afetaram a *precision after* nesses projetos. **Por outro lado, oito projetos (*A, C, E, F, G, L, M* e *O*) demonstraram uma *precision after* maior que 0.80**, destacando-se pela eficácia do *DR-Tools Code Health* em melhorar a precisão das medições para classes. Portanto, os resultados indicam que o *DR-Tools Code Health* tem um impacto positivo na *precision* das medições das classes dos projetos de software, resultando na exibição das classes mais problemáticas. No entanto, é importante investigar as razões por trás da baixa *precision* em alguns projetos específicos para identificar possíveis áreas de melhoria e garantir a eficácia contínua do *DR-Tools Code Health*.

A análise dos resultados do coeficiente *kappa* (Figura 5.9) revela uma variedade de níveis de concordância entre os participantes em relação às classes mais problemáticas dos projetos de software investigados. Com um **coeficiente *kappa* médio de 0.65 (mediana=0.74; DP=0.36), observa-se uma concordância moderada a substancial entre os participantes em suas avaliações**.

É interessante observar que **os projetos *B*, *D* e *N* apresentaram um coeficiente**

Figura 5.8 – *Box plot* da *precision* das classes: antes da avaliação (A) e depois da avaliação (B)



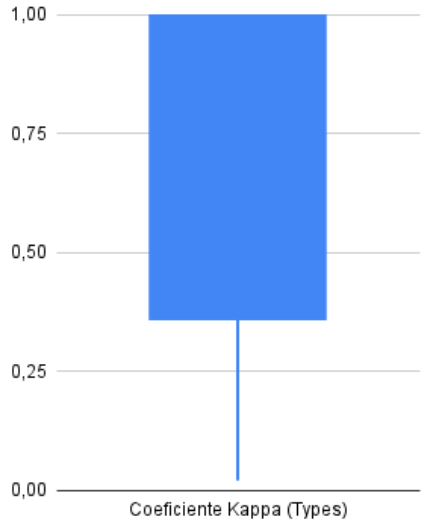
Fonte: O Autor

***kappa* com níveis mais baixos de concordância dos participantes em relação as classes específicas**. Isso pode sugerir que esses projetos enfrentaram desafios particulares em relação à identificação e avaliação das classes mais problemáticas. Por outro lado, **sete projetos (*A, C, E, G, L, M* e *O*) obtiveram um coeficiente *kappa* maior que 0.80**, o que sugere uma concordância substancialmente alta entre os participantes em relação às classes problemáticas desses projetos. De certa forma, é um indicativo de consistência notável nas avaliações dos participantes e uma compreensão compartilhada das classes mais problemáticas em questão. Embora alguns projetos tenham apresentado níveis mais baixos de concordância, a maioria demonstrou uma concordância substancial entre os participantes, fornecendo uma base sólida para análises futuras e melhoria contínua. Assim, resultados destacam a importância do *DR-Tools Code Health* para facilitar a identificação e avaliação das classes mais problemáticas nos projetos de software.

Antes do uso do *DR-Tools Code Health*, a *precision* média para classes + métodos era de 0.57 (mediana=0.50; DP=0.38), indicando uma precisão moderada, porém com uma variabilidade considerável nos dados. No entanto, **após seu uso, houve um aumento significativo na *precision* média para 0.80 (mediana=0.93; DP=0.26), evidenciando uma precisão mais consistente e confiável** nos resultados (Figura 5.10). A análise dos resultados da *precision* revela uma melhoria substancial após a execução do *DR-Tools Code Health* para investigar a classes + métodos nos projetos de software.

Ainda, é relevante destacar que **os projetos *B, D* e *H* apresentaram uma *preci-***

Figura 5.9 – *Box plot* do coeficiente *kappa* para as classes



Fonte: O Autor

***sion after* menor** após a execução do *DR-Tools Code Health*, sugerindo possíveis desafios ou limitações específicas na aplicação da ferramenta nessas instâncias. Por outro lado, **nove projetos (*A, C, E, F, G, L, M, N* e *O*) alcançaram uma *precision after* maior que 0.80**, indicando uma eficácia notável da ferramenta em melhorar a precisão das medições nessas situações.

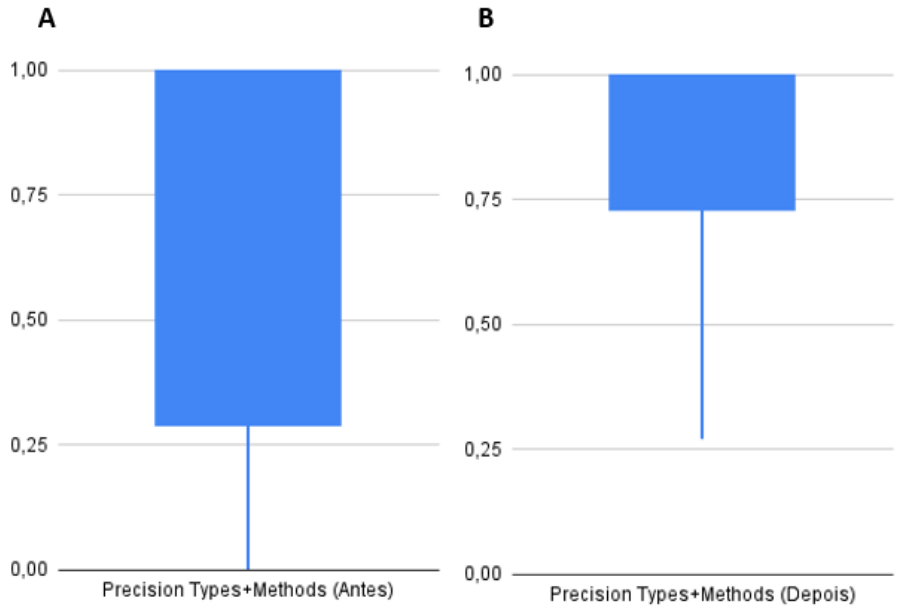
Em conclusão, os resultados apontam para o impacto positivo do *DR-Tools Code Health* na melhoria da *precision* na composição de classes e métodos nos projetos de software. Embora alguns projetos tenham apresentado uma redução na *precision*, a maioria demonstrou uma melhoria substancial, fornecendo uma base sólida para aprimoramentos contínuos na qualidade e confiabilidade dos resultados obtidos.

Na **concordância com os resultados de classes + métodos** (Figura 5.11), que visa juntar *smells* detectados e priorizados nas granularidades de classe e método, encontrou-se um **coeficiente *kappa* médio de 0.73, considerada substancial** (mediana=0.81; DP=0.34). Os **projetos *B* e *D* apresentaram os menores valores *kappa*, ou seja, concordância pobre** (valores 0.09 e 0.12, respectivamente). Já **os projetos *A, C, E, G, L, M, N* e *O* apresentaram *kappa* acima de 0.81**, considerada **concordância quase perfeita**.

Finalmente, a Tabela 5.7 apresenta um agrupamento dos projetos que receberam maior e menor *score* por granularidade analisada, conforme o método estatístico empregado.

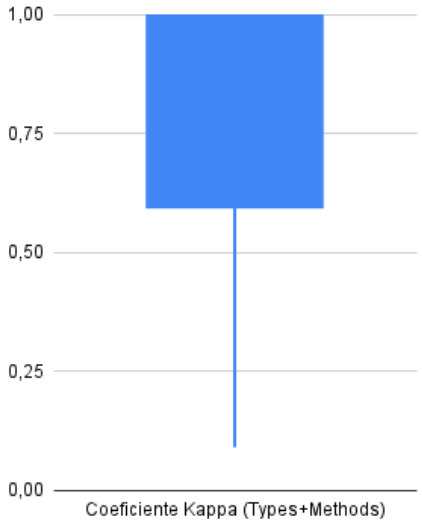
Como é possível observar, **os projetos *A, C, E, L, M* e *O* apresentaram maior *precision* em todas as granularidades analisadas**. Em **relação à concordância**, observa-

Figura 5.10 – *Box plot* da *precision* das classes + métodos: antes da avaliação (A) e depois da avaliação (B)



Fonte: O Autor

Figura 5.11 – *Box plot* do coeficiente *kappa* para os classes + métodos



Fonte: O Autor

se que **os projetos *A, C, E, G, L, M* e *O* estão presentes nas granularidades**. **Ao cruzar *precision* com *kappa*, nota-se que os projetos *A, C, E, L, M* e *O* se destacaram**. Estes projetos são de propósitos e tamanhos diferentes, o que pode indicar um caminho a ser explorado para uma maior generalização dos dados resultantes.

Contudo, também é possível notar que alguns projetos tiveram baixa *precision* e concordância. Os projetos *B, D* e *H* foram os que apresentaram menor *precision* em pelo menos duas das três granularidades analisadas. Enquanto em relação à concordância, o

Tabela 5.7 – Relação de projetos e estatísticas utilizadas para avaliação.Em destaque, os projetos que mais apareceram nas avaliações, independente de granularidade

Estatística	Method	Type	Type+Method
precision(+)	**A**, B, **C**, **E**, **L**, **M**, N, **O** (8)	**A**, **C**, **E**, F, G, **L**, **M**, **O** (8)	**A**, **C**, **E**, F, G, **L**, **M**, N, **O** (9)
kappa(+)	**A**, B, **C**, **E**, **G**, **L**, **M**, N, **O** (9)	**A**, **C**, **E**, **G**, **L**, **M**, **O** (7)	**A**, **C**, **E**, **G**, **L**, **M**, N, **O** (8)
precision(-)	**H** (1)	**B**, **D**, N (3)	**B**, **D**, **H** (3)
kappa(-)	**D**, H (2)	B, **D**, N (3)	B, **D** (2)

Fonte: O Autor

projeto *D* aparece em destaque em todas as granularidades. **Quando se avalia as duas estatísticas e todas as granularidades, de fato, o projeto *D* é o que apresenta menor *score***. Portanto, é importante avaliar estes casos específicos para identificar pontos de melhoria no *DR-Tools Code Health*. Outro aspecto importante para posterior avaliação é o uso da composição de classes + métodos como ponto a ser considerado na avaliação de elementos problemáticos. Conforme resultados, esta composição se mostrou mais relevante para os participantes do que simplesmente analisar os *smells* na granularidade de classe. Isso pode indicar que aquele elemento de código merece mais atenção ao olhar os *smells* de métodos e classes juntos.

Resposta resumida a RQ1

Em geral, o modelo de priorização proposto e implementado pelo *DR-Tools Code Health* auxilia os desenvolvedores na identificação dos elementos de código mais problemáticos.
Nos métodos, obteve-se uma *precision after* média de 0.90 e mediana de 1.00 além de coeficiente *kappa* médio de 0.82 (mediana de 1.00), indicando uma concordância quase perfeita.
Quando analisada as classes, a *precision after* média ficou de 0.73 e mediana de 0.80, além de coeficiente *kappa* médio de 0.65 (mediana de 0.74), indicando uma concordância substancial.
Na composição de classes + métodos, obteve-se uma *precision after* média de 0.80 e mediana de 0.93, com coeficiente *kappa* médio de 0.73 com mediana de 0.81, indicando uma concordância substancial.

A RQ2 é relacionada a avaliação da percepção dos participantes sobre os elementos mais problemáticos. Os participantes foram convidados a comentar suas respostas. Os dados completos obtidos podem ser acessados no Apêndice H. A Tabela 5.8 ilustra alguns exemplos de comentários das percepções dos participantes.

Também, obteve-se casos em que, **mesmo o participante confirmando os ele-**

Tabela 5.8 – Comentários dos participantes sobre a percepção dos resultados

Projeto	Resposta	Confirmado?
E	Não esperava que uma classe VO estivesse na lista, porém já vi diversas má práticas dentro desse tipo de classe.	Sim
F	A maioria dos métodos listados pelo *DR-Tools* realmente são prioridade pelos problemas apresentados. Os 3 métodos marcados como não prioritário possuem refatorações, mas no geral apresentam um código menos acoplado e com uma única responsabilidade, então por isso não seria considerado uma prioridade frente a outros métodos mais problemáticos. No geral as primeiras posições de prioridades fecharam corretamente e no restante organizou em diferentes posições que ficaram melhor ordenadas. As classes marcadas como não prioritárias vale a pena frisar que a classe *br.com.sjc.service.DashboardFilaServiceImp* parece estar duplicada e as demais tem um objetivo muito especifico, então não seriam prioridade.	Sim
G	No sistema em estudo, os maiores problemas que temos ao dar manutenção são nas classes e métodos grandes. São difíceis de entender, de alterar e de testar. Dessa forma, quando buscamos por métricas de tamanho pelo Sonar (LOC e CYCLO), acabamos chegando a resultados bem semelhantes aos sugeridos pelo DR-Tools Code Health. Nos exemplos que não confirmamos, são casos onde, apesar da grande quantidade de linhas de código e de estruturas condicionais, não representam lógicas intrincadas, nem com muitos blocos aninhados. Por isso achamos que não deveriam ser priorizados, considerando outros métodos que temos no sistema que de fato são mais complexos de serem compreendidos.	Sim
H	entendo que nesse contexto os problemas eram conhecidos, porém, na época que o código foi desenvolvido, certamente haveria uma boa contribuição da ferramenta	Não
I	O *DR-Tools* trouxe além de alguns métodos que eu já havia identificado manualmente, alguns novos que realmente fazem sentido estar em uma priorização. O único detalhe que na minha visão não fez muito sentido, foi a priorização de um método *equals*. Este tipo de método pode ser um pouco complexo pela sua natureza, mas no caso apontado, é uma implementação padrão gerada pela IDE e que compara alguns campos entre os objetos envolvidos. Além disso, o *DR-Tools* trouxe além de algumas classes que eu já havia identificado manualmente, alguns novos que realmente fazem sentido estar em uma priorização. As duas classes que marquei como não, na minha visão são relativamente simples e coesas. A classe *fixture* é um pouco grande, mas seus métodos fazem bastante sentido para o contexto, e não são complexos.	Sim
J	A maioria dos métodos problemáticos eu tinha a visão mas não de todos. A ferramenta me ajudou a ampliar minha visão sobre o projeto. Haviam mais classe com *code smells* do que parecia, mas nada muito grave.	Sim
K	Parte dos itens apontados realmente correspondem a métodos mais complexos, apesar de listar alguns métodos os quais não possuem complexidade, fazendo o uso de anotações do *spring*. Algumas classes novas foram listadas, que realmente não haviam sido percebidas como complexas, principalmente classes denominadas como utilitárias, possuindo diversas responsabilidades. Todavia, foram listadas muitas classes de DTO que não possuem implementações concretas, fazendo o uso apenas de anotações do Lombok. A nova listagem de classes + métodos trouxe uma priorização mais relevante, apontando algumas classes que são conhecidas pelo time por sua complexidade, tomando por exemplo as classes de factory, que possuem grande responsabilidade neste projeto pela composição dos modelos de dados.	Sim
L	O resultado me ajudou a enxergar melhor todos os detalhes. Com isso, consigo avaliar complexidades de forma mais inteligente e eficaz. Combinar os resultados facilita do ponto de vista de análise geral da classe, no entanto, é possível que sobrecarregue a experiência do usuário que esteja focado em algo mais especifico (apenas classe ou método) . Oferecê-la com uma opção avançada, ou de *drill-down*, pode ajudar a melhorar essa experiência.	Sim
M	A análise fornecida pelo *Dr-Tools* trouxe uma visão dos métodos mais críticos e que podem evidenciar para uma possível refatoração e melhoria de código. Também, mudou a forma que visualizava a qualidade das classes no geral, fornecendo uma análise baseado em métricas o que ajuda a evidenciar estes débitos técnicos no código.	Sim
N	Mudou minha prioridade para refatorar esses métodos. Já tinha conhecimento que estes métodos deveriam ser refatorados para diminuir a complexidade. Marquei com "Não faz sentido estar nesta lista dos elementos priorizados" para alguns Beans. São classes de Mapeamento OR e alguns POJOs. Os getters e setters são construídos de forma dinâmica utilizando as *annotations* do *framework Lombok*.	Sim

Fonte: O Autor

mentos de código apresentados pelo *DR-Tools Code Health*, alguns participantes selecionaram a opção de que não foi mudada sua percepção. Este foi o caso do projeto *C*, que também não registrou comentários extras, como vários outros projetos.

Em relação a mudança da percepção, o participante pode escolher, para cada granularidade apresentada, se houve ou não mudança ou até mesmo se não observou diferenças. **Na granularidade de métodos, 87% dos participantes afirmaram ter mudado sua percepção, enquanto 13% não mudaram sua percepção.** Observa-se que, na granularidade de métodos, a grande maioria dos participantes afirmou ter mudado sua percepção, indicando uma resposta positiva ao *DR-Tools Code Health*. Já **nas granula-**

ridades de classes e classes + métodos, se obteve os mesmos percentuais. Cerca de **60% dos participantes indicaram mudança na percepção, enquanto 33% indicaram não ter mudança e 7% não observaram diferenças**. Similarmente, nas granularidades de classes e classes + métodos, embora em menor proporção, ainda houve uma porcentagem considerável de participantes que indicaram uma mudança em sua percepção. É interessante notar que uma parte dos participantes (7-33%) relatou não observar diferenças, mesmo após a análise dos resultados apresentados pelo *DR-Tools Code Health*. Isto sugere que, apesar do *DR-Tools Code Health*, alguns participantes podem não ter percebido alterações significativas nos elementos de código ou podem ter tido dificuldades em interpretar as informações fornecidas. Portanto, é possível notar que o uso do *DR-Tools Code Health* parece ter tido um impacto significativo na percepção dos participantes em relação aos elementos de código analisados, embora ainda haja espaço para melhorias na comunicação e na interpretação dos resultados, especialmente para aqueles que relataram não observar diferenças após o uso da ferramenta.

Resposta resumida a RQ2

Os dados apresentados indicam que a utilização do *DR-Tools Code Health* teve um impacto significativo na percepção dos participantes em relação aos elementos de código analisados nos projetos de software.

A maioria dos participantes relatou uma mudança em sua percepção, especialmente na granularidade de métodos, onde 87% afirmaram ter observado diferenças. No entanto, uma parcela considerável dos participantes (7-33%) não percebeu mudanças ou não observou diferenças, destacando a necessidade de uma investigação mais detalhada sobre a eficácia da ferramenta e possíveis áreas de melhoria na comunicação dos resultados fornecidos.

Além disso, ainda houve casos em que os participantes não registraram comentários extras e indicaram não ter mudado sua percepção, mesmo confirmando os resultados apresentados pelo *DR-Tools Code Health*.

5.1.4 Ameaças à Validade

Nesta subseção, discute-se as ameaças à validade, incluindo a validade interna, externa, de construto e de conclusão, e as diferentes táticas adotadas para mitigá-las.

A validade interna refere-se aos fatores que podem ter influenciado este estudo. Considerando os respondentes, eles eram desenvolvedores seniores, com larga experiência em desenvolvimento de software e nos projetos apresentados. Foram utilizadas algumas técnicas como *convenience sampling, purposive sampling* e *snowballing sampling*.

No entanto, o fato de ter que identificar os profissionais pode ter contribuído para inibir a participação da pesquisa. Ainda, em alguns casos, foi necessário solicitar autorização para a empresa do participante, mesmo não tendo acesso direto ao código.

A validação externa trata da generalização dos resultados da pesquisa. Certamente, é necessário ampliar a pesquisa para mais profissionais de desenvolvimento de software. Embora o *DR-Tools Code Health* tenha a função de customização implementada, justamente para adequar os parâmetros e pesos ao contexto o projeto, este recurso não foi explorado. Para minimizar este impacto, os parâmetros utilizados na configuração *default* foram definidos a partir de uma variedade de projetos de contextos e tamanhos diferentes (Apêndice B). Assim, os participantes do estudo atual utilizaram esta configuração. Também, os *smells* especificados no *DR-Tools Code Health* abrangem um conjunto pequeno, porém relevante de *smells* conhecidos. Seria interessante ampliar esse conjunto de *smells*. Por fim, os critérios escolhidos para o modelo de priorização foram baseados em trabalhos citados nesta tese. Certamente, é possível incluir outros ou até mesmo combiná-los para melhorar os resultados.

A validade de construto trata da identificação correta das medidas adotadas no procedimento de medição, como o questionário utilizado no experimento. Assim, conforme comentado anteriormente, para fins de delimitação de escopo, foram avaliadas somente as granularidades *methods* e *types*. Para confirmar que as perguntas correspondem à causa real objeto de interesse deste estudo, primeiro se analisou o mapeamento das perguntas de pesquisa e as perguntas nos questionários usados em estudos anteriores relacionados a percepção *smells* e priorização (YAMASHITA; MOONEN, 2013a; TAIBI; JANES; LENARDUZZI, 2017; OIZUMI et al., 2017; PECORELLI et al., 2020). Em seguida, verificou-se cada pergunta para evitar possíveis discrepâncias, perguntas negativas e ameaças devido ao viés de perguntas previamente respondidas. Neste sentido, o questionário foi avaliado por dois profissionais experientes na área de desenvolvimento de software, com mais de 20 anos de atuação. Algumas razões foram consideradas para o desenho do experimento neste formato: i) seria inviável solicitar aos desenvolvedores para classificar uma extensa lista de elemento de código; ii) a necessidade de avaliar se a abordagem proposta fornece informações relevantes para que os desenvolvedores possam planejar suas ações imediatas ou planejadas. Assim, os primeiros elementos de código problemáticos representam uma amostra significativa de elementos que podem causar problemas; e iii) a análise dos principais elementos de código para avaliar se eles representam um subconjunto útil de fontes de problemas relevantes no código.

A validade de conclusão está relacionada com a análise dos dados para os resultados finais. Foi feito uso da estatística descritiva, muito utilizada em trabalhos desta natureza. Além disso, foram utilizados duas medidas estatísticas adicionais: *precision* e coeficiente *kappa*. Geralmente a *precision* é utilizada em conjunto com outras métricas componentes da *confusion matrix*[7]. No caso deste estudo, esta medida foi utilizada para fornecer uma medida de proporção de instâncias classificadas corretamente como positivas em relação ao total. O coeficiente *kappa*, neste caso, foi utilizado para avaliar a concordância entre um único avaliador e a abordagem desenvolvida. Neste caso, considerou-se cada método e classe como um item a ser classificado. O avaliador indicou a concordância em ser um elemento problemático e passível de ser priorizado. Então, comparou-se as classificações do participante com as classificações resultantes *DR-Tools Code Health*. É importante ressaltar que, ao usar o coeficiente *kappa* com apenas um avaliador (neste caso, o participante), está se avaliando basicamente a consistência do método de classificação em si mesmo. Isso pode ser útil para verificar a confiabilidade do método, mas não fornece informações sobre a concordância entre diferentes avaliadores.

5.2 Experimento 2: Priorização e Impacto na Qualidade dos *Smells* em Projetos *Open-Source* durante a Evolução de Software

O experimento 2 tem por objetivo avaliar, de forma exploratória se há, mesmo que *ad hoc*, a ação de priorização dos *smells* durante a evolução de software. Além disso, investiga-se também o impacto dos *smells* em atributos de qualidade durante a evolução do software.

Neste experimento, é desenvolvido um estudo longitudinal com 5 projetos *open-source*. Um estudo longitudinal (CEDRIM et al., 2017; BESSGHAIER; OUNI; MKAOUER, 2021) é um tipo de pesquisa em que os dados são coletados ao longo de um período de tempo prolongado, geralmente com o objetivo de observar e analisar mudanças, padrões ou tendências ao longo do tempo.

Assim, foram definidas as seguintes questões de pesquisa:

[7]Mais informações em <https://en.wikipedia.org/wiki/Confusion_matrix>

Questões de Pesquisa do Experimento 2

RQ3 - Os desenvolvedores fazem algum tipo de priorização ao remover os *smells* durante a manutenção e evolução de software?

RQ4 - Como os *smells* têm impactado a qualidade durante a evolução de software?

5.2.1 Protocolo

O experimento é apresentado, de forma resumida, na Figura 5.12.

Figura 5.12 – Resumo do Experimento 2

Fonte: O Autor

A seguir, são apresentadas as seguintes etapas:

1. **Identificação dos projetos para avaliação**. O experimento considerou 5 projetos *open-source*, de diferentes tamanhos e propósitos, visando ampliar os resultados. Os repositórios considerados foram selecionados diretamente no *GitHub*, com o apoio de algumas ferramentas[8] para suporte à seleção dos projetos. Foi utilizado os seguintes critérios: i) ser um projeto ativo (alterações realizadas nos últimos 3 meses); ii) ser um projeto maduro (mais de 5 anos de desenvolvimento); iii) ter mais de 400 *stars*; iv) ser um projeto escrito predominantente em linguagem *Java*;
2. **Mapeamento para análise das versões do projeto**. Identificou-se as janelas de tempo mais relevantes para análise do projeto (etapa 2). Entende-se por relevante as janelas de tempo que contém ao menos, 6 versões por projeto, com vários tipos de alterações: i) dezenas a centenas de *commits*, ii) centenas de alterações nos

[8] *Repo Reapers* <https://reporeapers.github.io/>
Outra fonte utilizada foi o *Corpus-2021* <https://github.com/Corpus-2021>

arquivos, iii) centenas até milhares de refatorações realizadas no código;

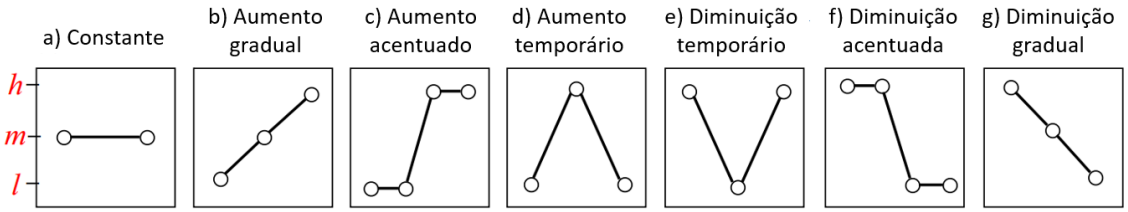
3. **Retorno na linha do tempo, considerando a versão a ser analisada**. O projeto é clonado para análise. Após esta ação, é realizada a primeira verificação e contagem de todas as versões do projeto, total de *commits* do projeto e por versão e maiores contribuidores. Para esta coleta, foi usado comandos disponíveis no controle de versão *git* (Apêndice O);
4. **Coleta das informações do projeto nas versões selecionadas**. Nesta etapa, executou-se o *DR-Tools Code Health* (opção *–analyze*) em cada versão selecionada, com o objetivo de coletar dados sobre métricas, *smells* e *ranking* dos elementos de código mais problemáticos. Também, foi utilizado o *RefactoringMiner*[9] (TSANTALIS; KETKAR; DIG, 2022), ferramenta de mineração de refatorações com alto *precision* e *recall*, que detecta mais de 90 tipos de refatorações.
5. **Análise da evolução dos *smells* com o *DR-Tools Smell Churn***. Com todos os dados gerados por versão, desenvolveu-se uma ferramenta para apoiar as análises de evolução dos *smells*, tomando como base os resultados gerados pelo *DR-Tools Code Health*. Detalhes da ferramenta são apresentados no Apêndice N.

   Com ela, foi possível implementar uma nova métrica de análise, denominada *smell churn*, que permite avaliar quais *smells* foram removidos, adicionados ou ainda estão presentes entre as versões analisadas.
6. **Análise dos resultados e discussão**. Ao analisar a evolução do projeto, é feita uma comparação de uma dada versão (*e.g.,* v_0) com a outra versão posterior (*e.g.,* v_1), avaliando se houve piora, melhora ou se manteve constante ao longo do tempo em relação aos *smells*.

   Para esta avaliação, é usado o *Dynamic Time Warping* (DTW), técnica proposta por Kruskall and Liberman (1983), que visa encontrar uma distância topológica entre dois sinais. Esta técnica foi usada recentemente em outros trabalhos sobre *smells* e evolução de software (VAUCHER et al., 2009; SAS; AVGERIOU; UYUMAZ, 2022). Assim, neste experimento, considerou-se cada série de valores de cada característica das instâncias de um *smell* como um sinal (ou série temporal). Em seguida, compara-se cada sinal a uma série de sinais predefinidos (*templates*, Figura 5.13), cada um com um rótulo correspondente. Dependendo do modelo que estiver matematicamente mais próximo do sinal, um rótulo é atribuído a ele.

[9]Mais informações da ferramenta podem ser obtidas em <https://github.com/tsantalis/RefactoringMiner>

Figura 5.13 – *Templates* dos rótulos de classificação de evolução de tendência. Dado que o *smell* seja considerado um sinal *S*, pode-se ter as seguintes variáveis: *h* = máximo *S*; *l* = mínimo *S*; e *m* = *(h + l)/2*.



Fonte: (SAS; AVGERIOU; UYUMAZ, 2022)

Também, é investigado o impacto dos *smells* em atributos de qualidade nas granularidades de *namespace, type* e *method*. O mapeamento destes atributos de qualidade com *smells* foram baseados nas próprias referências que os definem (FOWLER et al., 1999; LIPPERT; ROOCK, 2006; SURYANARAYANA; SAMARTHYAM; SHARMA, 2014; FONTANA et al., 2017), descritos em detalhes na Tabela 4.4.

5.2.2 Desenvolvimento

Após a escolha dos projetos, com base nos critérios previamente definidos, usou-se o recurso *compare*[10], do *GitHub* para ajudar nas detecção das janelas de tempo, considerando o número de *commits* (MCDONALD; GREER, 2019) e número de arquivos modificados (FARAGÓ; HEGEDŰS; FERENC, 2015a), primeiramente. Em um segundo momento, foi considerado também o número de refatorações realizadas. Definiu-se, pelo menos, 6 janelas de tempo para análise. É possível que em uma janela de tempo englobe mais de uma versão. Para este estudo, é relevante identificar um número considerável de modificações realizadas no código, passíveis de investigação.

Nestas janelas, considerou-se inicialmente períodos maiores para iniciar as análises e, quando necessário, reduzindo para períodos menores (meses) em projetos mais ativos. Para as seleções, também se observou o tamanho dos ciclos de *releases* (datas, diferença de tamanho de um para o outro, crescimento entre eles) e volume de dados (se incluir muitos dados históricos na análise, os resultados podem esconder tendências recentes). Também, avaliou-se o uso de versões *major* e versões *minor*[11]. Outra consideração feita foi selecionar análises de outras *branches* ou somente a principal (*main/master*).

[10]Mais informações em <https://docs.github.com/en/repositories/releasing-projects-on-github/comparing-releases>

[11]Mais informações em <https://semver.org/lang/pt-BR/>

Optou-se pela *branch* principal, pois contém a maioria das mudanças significativas realizadas no projeto. Todas essas avaliações foram feitas considerando as funcionalidades disponíveis diretamente na plataforma do *GitHub*.

Os projetos são clonados para que se tenha acesso a todo o histórico do repositório. São utilizados comandos do *git* para navegar entre as versões, contar o número de *commits*, obter as informações de contribuição e analisar especificamente cada alteração realizada, quando necessário. Depois, a partir de cada versão mapeada nas janelas de tempo, é executado o *DR-Tools Code Health* no modo CLI para coletar todos os dados sobre métricas, *smells* e *ranking* dos elementos de código. Os dados são gerados em diretórios específicos de cada versão, para posterior análise. Também, é executado o *RefactoringMiner* para minerar as refatorações entre duas versões. Estas informações das refatorações são importantes para se investigar o fenômeno de redução/aumento dos *smells*, em uma análise mais específica. Todos os comandos executados nesta etapa estão disponíveis no Apêndice O.

Todos estes dados são estruturados em uma planilha específica por projeto (Apêndice M). Nesta planilha, são armazenadas as informações do projeto como *link* do repositório, número de *github stars*, número de versões totais e versões analisadas. Também são organizados o número de contribuições por desenvolvedor, estatísticas como número de *namespaces*, *types* e *methods*, LOC, complexidade ciclomática, média/mediana e desvio padrão de LOC por *types* e total de *commit* realizados até a versão analisada. Ainda, são coletadas as informações de refatorações realizadas por *commit* em cada versão. Em relação aos *smells*, são coletados por versão as seguintes informações para *namespaces* (número total de *smells* por *namespace*, número de *namespaces* com mais de um *smell* e respectivos percentuais), *types* (número total de *smells* por *type*, número de *types* com mais de um *smell* e respectivos percentuais) e *methods* (número total de *smells* por *method*, número de *methods* com mais de um *smell* e respectivos percentuais), além da computação dos percentuais de densidade de *smells* por granularidade. Da mesma forma que o Experimento 1 (descrito na Seção 5.1), usou-se o mesmo formato de computação da densidade dos *smells*. Especificamente, são contabilizados individualmente os *smells* por granularidade, considerando sua tendência de aumento/redução, analisados pelo *DR-Tools Code Health* (Tabela 4.4).

Por fim, é calculado o *smell churn* de cada *smell* por versão analisada. Assim, é possível avaliar quantos foram adicionados e removidos, comparando duas versões pertencentes a janela de tempo avaliada. Com esta métrica, pode-se ter uma dimensão mais

fidedigna da evolução dos *smells* por versão, verificando quais ações contribuíram para tal. Ainda, são tabulados os impactos dos atributos de qualidade de cada *smell* detectado por granularidade e versão.

5.2.3 Resultados e Discussão

Neste experimento, foram analisados 5 projetos *open-source* de contexto e tamanhos diferentes, totalizando 30 versões. Os projetos são:

- **JetUML**[12]**:** Ferramenta de modelagem UML. Possui 34 versões, 27 contribuidores e 609 *github stars*;
- **Apache Ant**[13]**:** Ferramenta para automação de *builds* de projetos Java. Possui 154 versões, 72 contribuidores e 409 *github stars*;
- **Apache Storm**[14]**:** Sistema de processamento de fluxo distribuído. Possui 52 versões, 363 contribuidores e 6.5 mil *github stars*;
- **SpotBugs**[15]**:** Ferramenta de análise de *bugs* para projetos Java. Possui 82 versões, 174 contribuidores e 3.3 mil *github stars*;
- **Apache Cassandra**[16]**:** Sistema de banco de dados escalável e distribuído. Possui 305 versões, 444 contribuidores e 8.5 mil *github stars*.

A Tabela 5.9 apresenta os dados quantitativos coletados de cada projeto, por versão analisada. Estes dados incluem *namespaces, types, methods*, número de linhas de código (SLOC) e número total de *commits* realizados até aquela dada versão. Com **exceção do *Apache Storm* e do *SpotBugs*, todos tiveram um crescimento contínuo de todas as informações listadas**.

O ***Apache Storm*, na versão *v2.6.2* teve redução no número de *namespaces, types* e *methods***. **No entanto, teve um aumento no LOC do projeto**. Isso pode ser explicado pela adição de elementos de código (classes, métodos) concentrando mais funcionalidades e, consequentemente, com classes e métodos maiores.

O ***SpotBugs*, na versão *3.1.0*, teve uma redução nos *namespaces, types, methods* e LOC**. A justificativa para essa redução é a exclusão de recursos desnecessários, bem

[12]<https://github.com/prmr/JetUML/>
[13]<https://github.com/apache/ant/>
[14]<https://github.com/apache/storm/>
[15]<https://github.com/spotbugs/spotbugs/>
[16]<https://github.com/apache/cassandra/>

como a adequação das funcionalidades nas classes do projeto. Foram identificadas, no intervalo entre as versões *3.0.2_preview2* e *3.1.0*, um total de 5294 arquivos Java modificados.

Tabela 5.9 – Projetos analisados com as respectivas versões e informações quantitativas

Projeto	Versão	Namespaces	Types	Methods	SLOC	Commits
JetUML	0.1	4	59	408	6394	77
	1.0	5	83	688	8955	608
	2.0	14	121	1012	12118	978
	3.0	14	162	1054	12527	1045
	3.5	16	163	1227	14261	2269
	3.6	16	175	1233	13902	2442
Apache Ant	ant_190	79	1184	13393	130617	12821
	ant_194	80	1208	13249	135643	13015
	ant_195	80	1210	10322	136299	13164
	ant_1.10.4_rc1	87	1252	14017	137829	14216
	ant_1.10.11_rc1	92	1299	14753	144839	14730
	ant_1.10.14_rc1	92	1307	14891	146374	14963
Apache Storm	0.9.0.1	52	445	5288	46232	1753
	v0.10.0	123	882	9275	91733	4574
	v1.0.0	208	1418	16412	17033s4	6286
	v1.2.4	292	1843	20385	212826	7971
	v2.0.0	344	2301	25749	279437	9907
	v2.6.2	**280**	**2108**	**25513**	289459	10798
SpotBugs	2.0.0	164	2396	17910	200257	13819
	3.0.0_rc1	174	2565	18144	204461	14813
	3.0.2_preview2	176	2645	18833	217817	15322
	3.1.0	**173**	**2569**	**16529**	**186794**	15235
	4.1.0	176	2617	16638	187208	16281
	4.8.5	199	3003	18409	214808	17376
Apache Cassandra	cassandra-1.2.16	50	737	8556	96725	9207
	cassandra-2.1.0	63	927	10978	122360	13424
	cassandra-3.1	75	1226	15172	162907	18760
	cassandra-3.11.14	94	1502	19340	211040	24255
	cassandra-4.0-beta1	112	1797	23580	246428	25418
	cassandra-5.0-beta1	150	2434	33902	347465	29278

Fonte: O Autor

Durante o estudo, foram coletados as refatorações realizadas entre as versões selecionadas. **Observa-se, de forma geral**, conforme apresentado na Tabela 5.10, **que os *commits* possuem tamanhos variados, alterando de poucos a milhares de arquivos**. Alguns intervalos entre as versões analisadas apresentaram um maior número de *commits* que, de certa forma, poderia indicar também um maior número de refatorações aplicadas. **O intervalo entre as versões *v1.2.4* e *v2.0.0* do *Apache Storm* apresentou o maior**

número de *commits* (n=2589). Já **o *Apache Ant*, entre as versões *ant_194* e *ant_195*, apresentou o o menor número de *commits*** (n=145).

O *JetUML* apresentou média de 2.73 refatorações por *commit* (mediana=2.45; DP=1.26). O ***Apache Ant* obteve um valor médio de 5.68 refatorações por *commit*** (mediana=0.65; DP=0.65). Já o *Apache Storm* apresentou média de 3.62 refatorações por *commit* (mediana=3.22; DP=4.10). O ***SpotBugs* obteve uma média de 0.35 refatorações por *commit*** (mediana=0.31; DP=0.28). Finalmente, o *Apache Cassandra* apresentou média de 4.17 refatorações por *commit* (mediana=4.15; DP=4.15).

Tabela 5.10 – Projetos analisados com as respectivas versões, *commits* e número de refatorações

Projeto	Versão 1	Versão 2	Commits	Refatorações	Ref/Commits
JetUML	0.1	1.0	483	1050	2.17
	1.0	2.0	347	1684	4.85
	2.0	3.0	824	1267	1.54
	3.0	3.5	407	1081	2.66
	3.5	3.6	173	424	2.45
Apache Ant	ant_190	ant_194	195	126	0.65
	ant_194	ant_195	145	87	0.66
	ant_195	ant_1.10.4_rc1	894	632	0.71
	ant_1.10.4_rc1	ant_1.10.11_rc1	426	11165	26.21
	ant_1.10.11_rc1	ant_1.10.14_rc1	197	50	0.25
Apache Storm	0.9.0.1	v0.10.0	2180	1974	0.91
	v0.10.0	v1.0.0	1499	3322	2.22
	v1.0.0	v1.2.4	1235	6154	4.98
	v1.2.4	v2.0.0	2589	17573	6.79
	v2.0.0	v.2.6.2	657	2115	3.22
SpotBugs	2.0.0	3.0.0_rc1	984	531	0.54
	3.0.0_rc1	3.0.2_preview2	475	207	0.44
	3.0.2_preview2	3.1.0	450	141	0.31
	3.1.0	4.1.0	484	100	0.21
	4.1.0	4.8.5	1061	255	0.24
Apache Cassandra	cassandra-1.2.16	cassandra-2.1.0	2087	8652	4.15
	cassandra-2.1.0	cassandra-3.1	2331	8366	3.59
	cassandra-3.1	cassandra-3.11.14	2424	4158	1.72
	cassandra-3.11.14	cassandra-4.0-beta1	959	6422	6.70
	cassandra-4.0-beta1	cassandra-5.0-beta1	1939	9151	4.72

Fonte: O Autor

Porém, **ao analisar o número de refatorações, notou-se uma grande variação também**. As **mesmas versões do *Apache Storm* (*v1.2.4/v2.0.0*) apresentaram o maior número de refatorações** (n=17573). Onde se observou **o menor número absoluto de refatorações foi no *Apache Ant***, nas versões *ant_1.10.11_rc1/ant_1.10.14_rc1*

(n=50). Quando **é relacionado as duas medidas**, pela razão do número de refatorações pelo número de *commits* realizados entre duas versões, nota-se que **o *Apache Ant* obteve maior número de refatorações por *commit*** (n=26.21), entre as versões *ant_1.10.4_rc1* e *ant_1.10.11_rc1*. Já **o *SpotBugs* (*3.1.0/4.1.0*) obteve 0.21** refatorações por *commit*, **sendo o menor**. Em alguns casos nos projetos, os desenvolvedores implementaram refatorações para melhorar a manutenibilidade e compreensão do código. Isto está de acordo com o que foi apresentado por Palomba *et al.* (PALOMBA et al., 2017). Em outros, os refatorações foram utilizados para melhorar a coesão e acomodar novas funcionalidades, descritas nas próprias mensagens dos *commits*. Esta melhor documentação da mensagem das refatorações nos *commits* é conhecida por *self-affirmed refactoring* (ALOMAR; MKAOUER; OUNI, 2019).

A relação entre o tamanho dos *commits* e o número de refatorações realizadas é complexa e pode variar de acordo com diversos fatores. O estilo de desenvolvimento e as práticas de versionamento adotadas pelo time podem influenciar essa relação, com alguns desenvolvedores preferindo agrupar várias refatorações em um único *commit*, enquanto outros optam por *commits* menores e mais granulares. Assim, **observou-se que não existe uma relação direta entre o número de *commits* e as refatorações**.

Para ajudar na análise da evolução dos *smells* e também considerar qualquer ato de priorização, foi implementado uma ferramenta para coletar os *smells* inseridos e removidos por granularidade. Mais detalhes da construção e uso da ferramenta são apresentados no Apêndice N. Seguindo a mesma lógica do *code churn* (MUNSON; ELBAUM, 1998), é possível investigar quantos e quais *smells* foram inseridos ou removidos ou se mantiveram de uma versão para outra. A Tabela 5.11 apresenta um resumo da quantidade de *smells* inseridos e removidos. Também, é exibido o *smell churn rate*, computado pela razão dos *smells* removidos pelos adicionados.

Os dados apresentados podem ajudar a entender o fenômeno da evolução dos *smells*, visto que é possível ter uma noção mais fidedigna do que acontece entre as versões. O ***JetUML* apresentou o maior *smell churn rate* médio** (média=39.00; mediana=36.71; DP=11.22). Já o *Apache Ant* apresentou uma média de *smell churn rate* de 5.45 (mediana=1.80; DP=5.65). No *Apache Storm*, obteve-se um *smell churn rate* médio de 16.48 (mediana=10.05; DP=10.82). O *SpotBugs* apresentou uma média de 5.17 (mediana=2.17; DP=5.78) de *smell churn rate*. Finalmente, o *Apache Cassandra* apresentou um *smell churn rate* médio de 21.96 (mediana=24.09; DP=6.01).

Todos os dados apresentados de forma resumida neste experimento podem ser

Tabela 5.11 – Projetos analisados com o respectivo *smell churn* e versões

Projeto	**Versão**	**Adicionados**	**Removidos**	**Smell Rate**	**Churn**
JetUML	1.0	262	70	26.72	
	2.0	523	192	36.71	
	3.0	**461**	**261**	**56.62**	
	3.5	352	146	41.48	
	3.6	239	80	33.47	
Apache Ant	ant_194	3730	55	1.47	
	ant_195	4169	487	11.68	
	ant_1.10.4_rc1	4103	474	11.55	
	ant_1.10.11_rc1	3838	69	1.80	
	ant_1.10.14_rc1	3823	28	0.73	
Apache Storm	v0.10.0	2189	220	10.05	
	v1.0.0	5285	1732	32.77	
	v1.2.4	4932	420	8.52	
	v2.0.0	7693	1728	22.46	
	v2.6.2	6820	586	8.59	
SpotBugs	3.0.0_rc1	4860	346	7.12	
	3.0.2_preview2	4949	68	1.37	
	3.1.0	4961	718	14.47	
	4.1.0	4289	93	2.17	
	4.8.5	4473	32	0.72	
Apache Cassandra	cassandra-2.1.0	3561	858	24.09	
	cassandra-3.1	4898	1428	29.15	
	cassandra-3.11.14	5347	778	14.55	
	cassandra-4.0-beta1	6970	1740	24.96	
	cassandra-5.0-beta1	8720	1487	17.05	

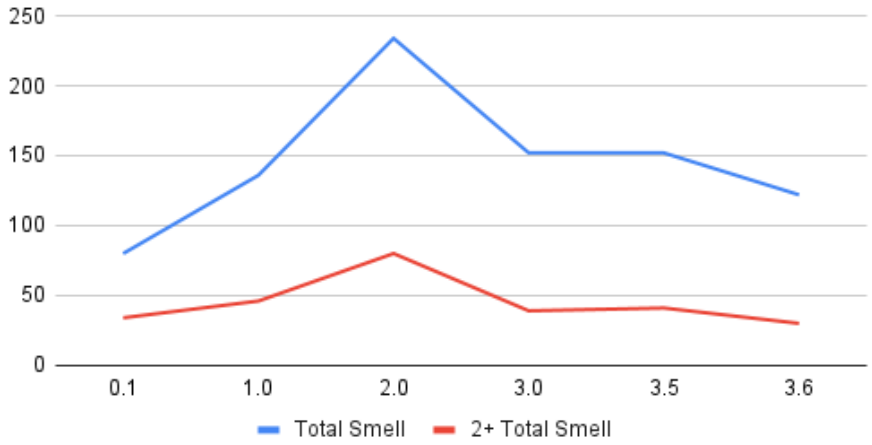
Fonte: O Autor

acessados no Apêndice M.

Smells, Priorização e Evolução de Software: Nesta etapa, são discutidos mais detalhes sobre os *smells* e a ação de priorização durante a evolução das versões. Para observar o comportamento dos *smells* em relação a sua evolução, utilizou-se a técnica DTW para análise. Assim, a evolução do número de instâncias de um dado *smell* pode ser representada por um sinal $S = (s_1, s_2, s_3, ..., s_n)$, onde s_i, $1 \leq i \leq n$ na versão i. O sinal é comparado com a série de sinais pré-definidos pela técnica, com um rótulo correspondente (conforme apresentado previamente na Figura 5.13). Da mesma forma que Sas *et al.* (SAS; AVGERIOU; UYUMAZ, 2022), entende-se que a aproximação oferecida pelo modelo ao classificar os sinais é suficiente para o propósito deste estudo, pois os modelos selecionados representam casos simples e gerais, simplificando a interpretação e análise.

O primeiro projeto analisado é o *JetUML*. Conforme apresentado na Figura 5.14, é possível observar a evolução dos elementos de código com *smells*, considerando todas as granularidades (*namespace, type, method*). Na linha azul, é exibido a quantidade de elementos com pelo menos um *smell*, enquanto a linha vermelha representa elementos de código com mais de um *smell*. Observa-se, a partir da *0.1*, um aumento gradual dos *smells* até chegar no ápice da *2.0*. Entre a versões *2.0* e *3.0*, ocorreu uma diminuição gradual, seguida de um comportamento constante (*3.5*), com uma leve diminuição em *3.6*, confirmando a tendência de queda dos *smells*.

Figura 5.14 – JetUML - Evolução dos elementos de código com *smell* (azul) e com mais de um *smell* (vermelho), considerando todas as granularidades

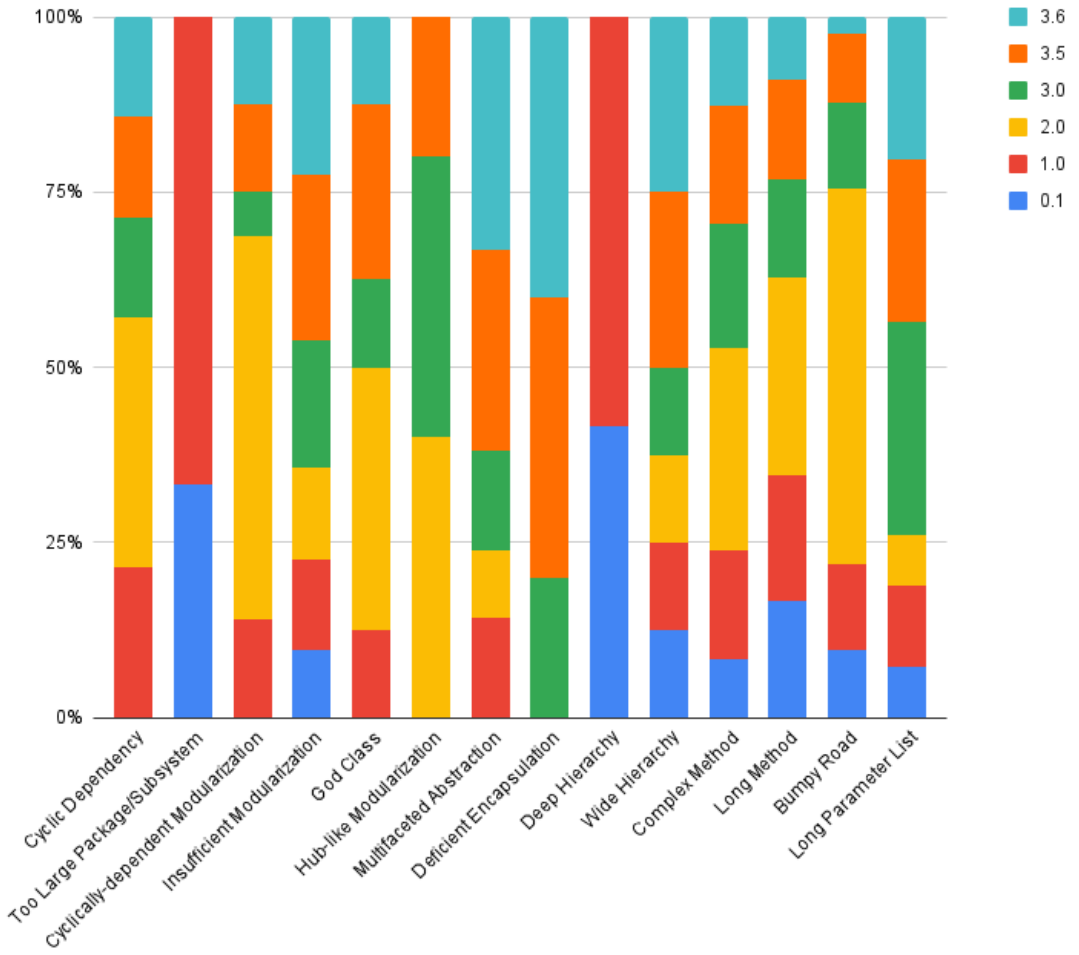


Fonte: O Autor

A Figura 5.15 mostra a distribuição de todos os *smells* detectados, considerando a versão analisada.

As versões *1.0, 3.0* e *3.5* apresentaram maior variação de tipos diferentes de *smells*

Figura 5.15 – JetUML - Distribuição dos *smells* detectados por versão analisada



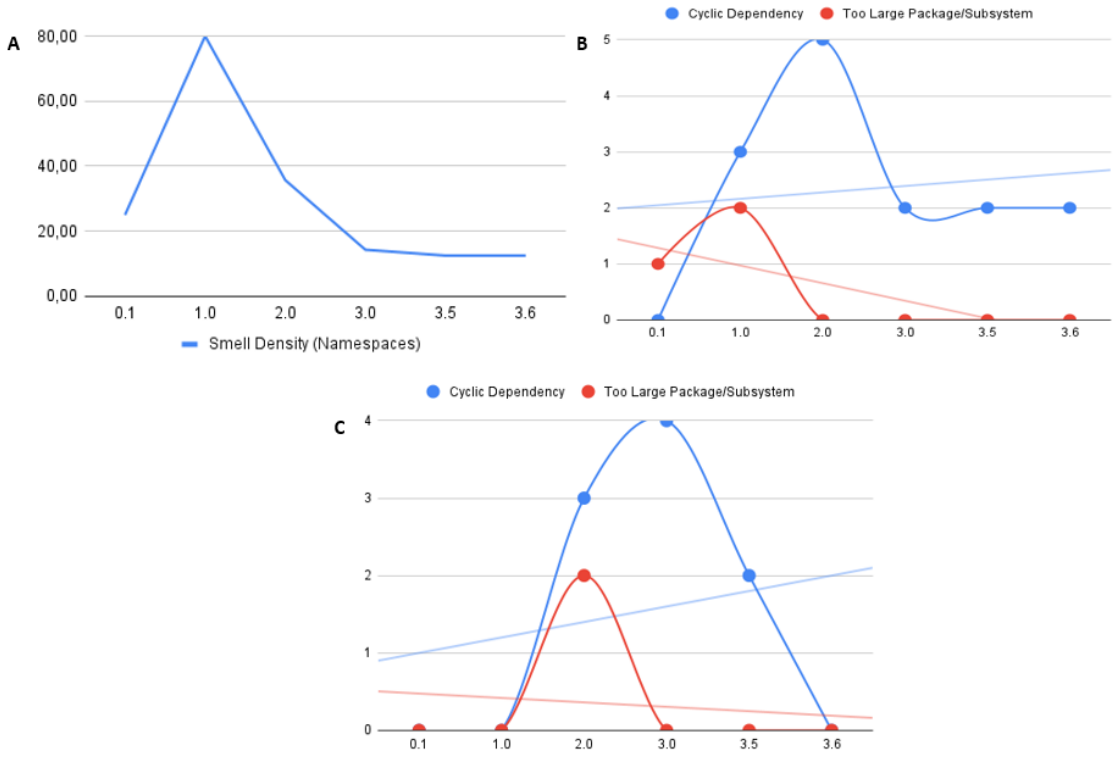
Fonte: O Autor

(13 dos 14 *detectados*). Já as versões *0.1* (9), *2.0* e *3.6* (ambos com 12) apresentaram a menor variação de *smells*. **Embora a versão *2.0* tenha menor variação, também apresentou o maior número de instâncias das versões das analisadas do *JetUML***. Os ***smells* mais difusos foram o *Insufficient Modularization, Wide Hierarchy, Complex Method, Long Method, Bumpy Road* e *Long Parameter List*, presentes em todas as versões**. Os ***smells Large Package/Subsystem* e *Deep Hierarchy* estiveram presentes apenas nas duas versões iniciais** (*0.1* e *1.0*).

A Figura 5.16 ilustra as informações relacionadas a *namespaces* das versões investigadas do *JetUML*.

A **densidade percentual dos *smells* (Figura 5.16, A) nos *namespaces* indicou que na versão *1.0* chegou a 80%, com diminuição gradual nas versões subsequentes**. Isto ocorreu não só pelo aumento do número de *namespaces* no projeto (Figura 5.16,

Figura 5.16 – Evolução dos *smells* de *namespaces* nas versões analisadas do JetUML - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

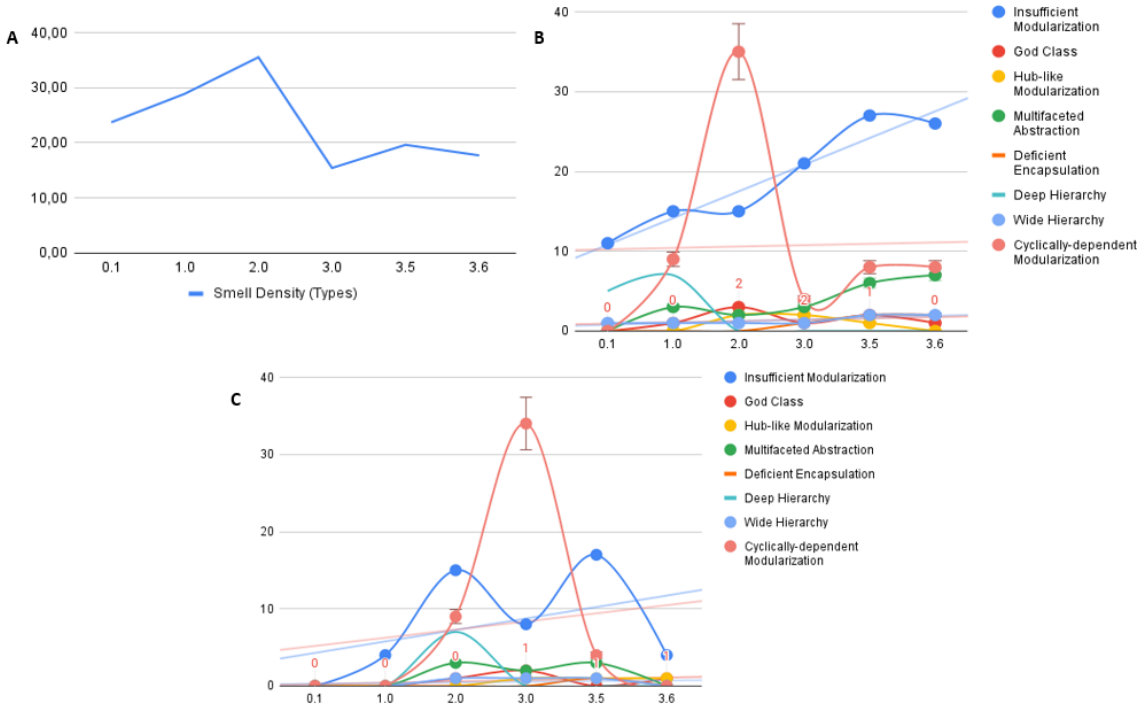
B) quanto pela redução gradual dos *smells* (Figura 5.16, C). **Entre as versões *2.0* e *3.5* ocorreram as maiores reduções dos *smells***. *Cyclic Dependency* foi a mais difusa e que também apresentou maior número de instâncias.

Na Figura 5.17 são apresentadas informações relacionadas a *types* das versões investigadas.

Na granularidade de *types*, nota-se na Figura 5.17(A) um comportamento de aumento gradual (versões *0.1* a *2.0*) das instâncias de *smells* por *types*, **chegando em torno de 35%. Houve também uma diminuição gradual considerável na versão *3.0*, com um aumento temporário entre as versões *3.5* e *3.6***. Ao observar o gráfico de evolução (Figura 5.17, B) e remoção (Figura 5.17, C), **os *smells* que tiveram maior adição de instâncias (*Cyclically-dependent Modularization, Insufficient Modularization*) também foram os que apresentaram maiores reduções**. Isso **ajuda a explicar essa esta tendência de queda dos *smells***. Os **outros *smells* da granularidade estão presentes em número baixo e controlado**, mas **não se observa um esforço de redução** dos mesmos.

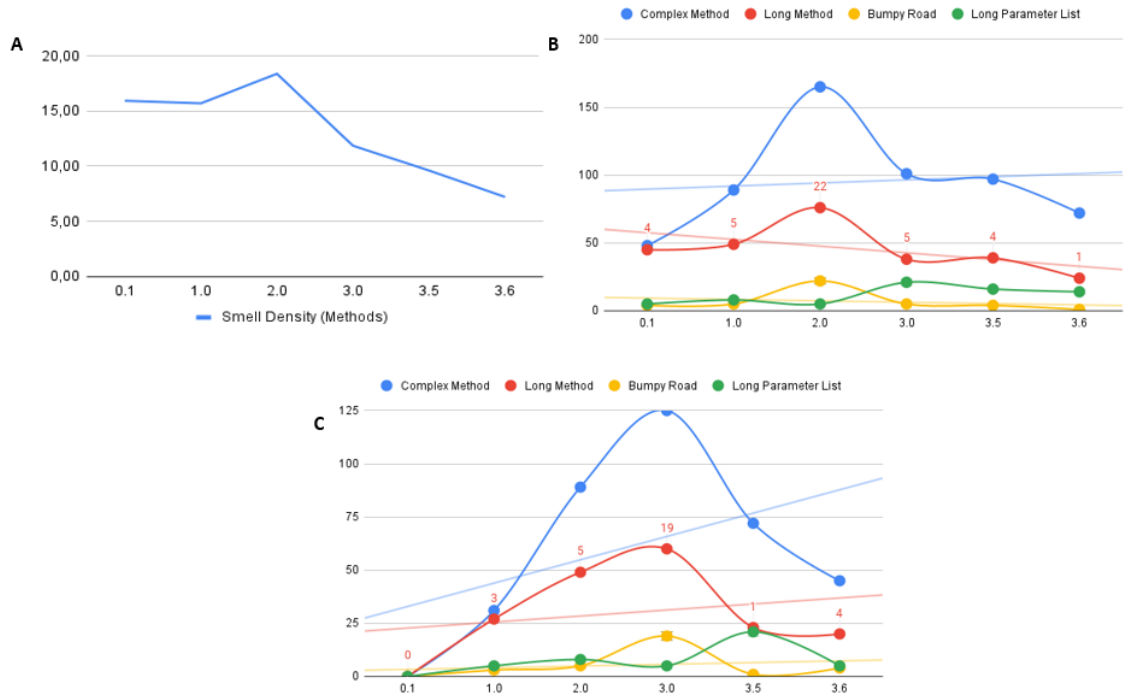
Finalmente, na Figura 5.18 são apresentados as informações de *methods* analisadas.

Figura 5.17 – Evolução dos *smells* de *types* nas versões analisadas do JetUML - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

Figura 5.18 – Evolução dos *smells* de *methods* nas versões analisadas do JetUML - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

A granularidade de *methods* apresenta (Figura 5.18, A) um comportamento constante nas duas primeiras versões (versões *0.1, 1.0*) das instâncias de *smells* por *methods*. Já nas versões seguintes (*2.0* em diante), observa-se um aumento temporário com uma redução gradual a partir da versão *3.0*. Também, **nota-se que o número de métodos com *smells* é baixo**, **não chegando a 20%**. Nota-se um crescimento nas 3 primeiras versões (*0.1, 1.0, 2.0* - Figura 5.18, B) com uma **diminuição gradual considerável na versão *3.0*, seguido de um comportamento constante com uma leve queda na sequência entre as versões *3.5* e *3.6***. Ao observar o gráfico de evolução (Figura 5.18, B) e remoção (Figura 5.18, C), **os *smells* que tiveram maior adição de instâncias (*Complex Method, Long Method*) também foram os que apresentaram maiores reduções**. Isso **ajuda a explicar essa esta tendência de queda dos *smells* nos *methods***. Os **outros *smells* dos *methods* estão presentes em número baixo e controlado**, **sem se observar um esforço de redução** dos mesmos. Outro ponto a ser destacado **onde se realizou um maior número de refatorações, entre as versões *1.0* e *2.0*, não foi onde se obteve maior redução de *smells***.

O segundo projeto analisado é o *Apache Ant*. Conforme apresentado na Figura 5.19, é possível observar a evolução dos elementos de código com *smells*, considerando todas as granularidades (*namespace, type, method*). Na linha azul, é exibido a quantidade de elementos com pelo menos um *smell*, enquanto a linha vermelha representa elementos de código com mais de um *smell*. Observa-se, **a partir da *ant_190*, um comportamento constante com nenhuma variação significativa dos *smells* até a versão *ant_1.10.14***.

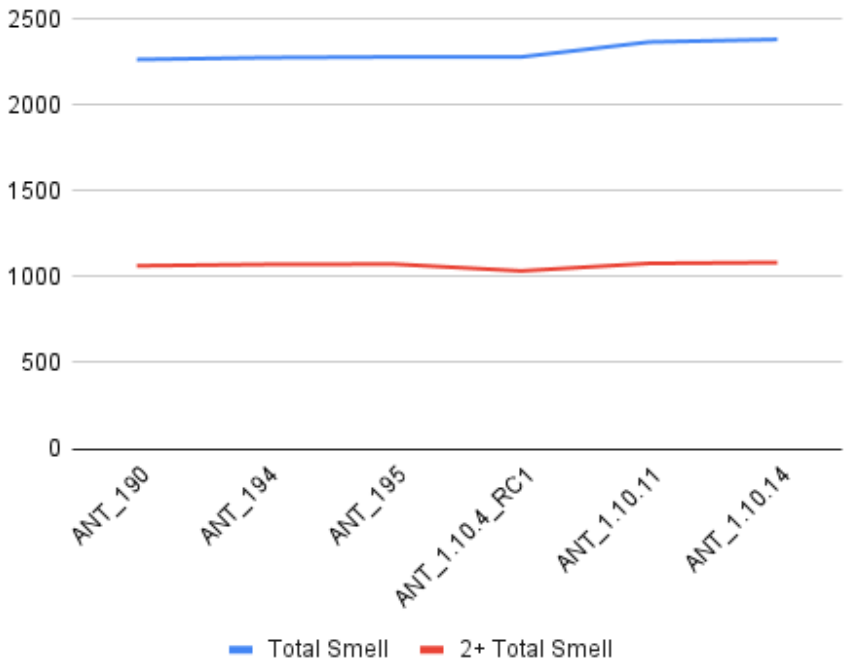
A Figura 5.20 apresenta a distribuição de todos os *smells* detectados, considerando a versão analisada.

Todas as versões do *Apache Ant* apresentaram todos os tipos diferentes de *smells*. **As grande maioria das versões analisadas não apresentaram grandes variações observáveis nas instâncias dos *smells***. **Apenas o *smell Deficient Encapsulation* teve um acréscimo considerável a partir da versão *ant_1.10.4*, se mantendo nas versões subsequentes**.

A Figura 5.21 apresenta as informações relacionadas a *namespaces* das versões investigadas do *Apache Ant*.

A **densidade percentual dos *smells* (Figura 5.21, A) nos *namespaces* indicou que, mesmo tendo uma leve redução, não é algo significativo a partir da versão *ant_194*. Apenas 16% dos *namespaces* apresentaram algum tipo de *smell***. Observando o comportamento praticamente constante do número de *namespaces* no projeto

Figura 5.19 – Apache Ant - Evolução dos elementos de código com *smell* (azul) e com mais de um *smell* (vermelho), considerando todas as granularidades



Fonte: O Autor

(Figura 5.21, B) ocorreu apenas uma remoção de uma instância do *Cyclic Dependency* (Figura 5.21, C). *Cyclic Dependency* e *Too Large Package/Subsystem* são difusos, com um número constante de instâncias.

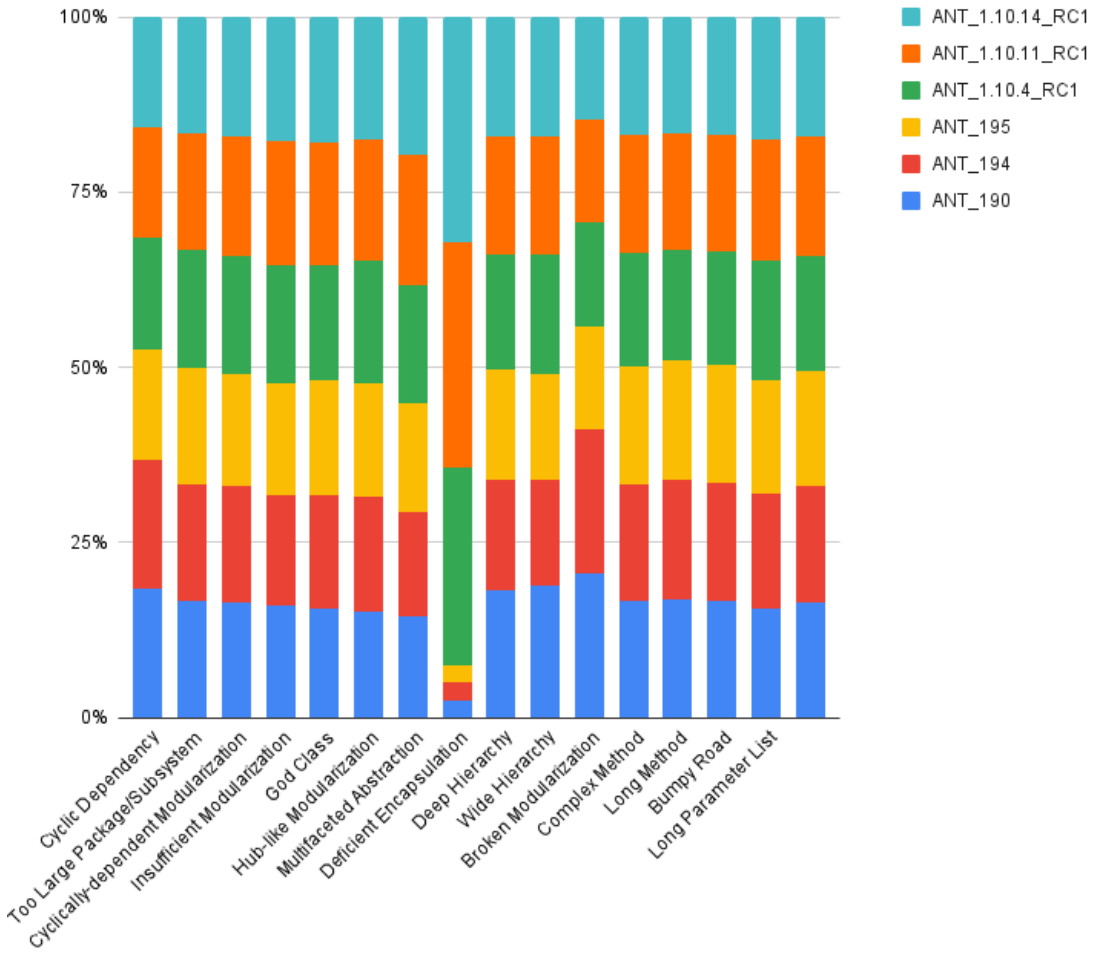
Na Figura 5.22 são mostradas as informações relacionadas a *types* das versões analisadas.

Na granularidade de *types*, observa-se na Figura 5.22(A) um comportamento de aumento leve, mas não significativo (versões *ant_195* a *ant_1.10.4*) das instâncias de *smells* por *types*, **chegando a quase 31%**. Ao analisar o gráfico de evolução (Figura 5.22, B), é possível observar que o *smell Insufficient Modularization* se destaca dos demais, mesmo mantendo um comportamento constante em número de instâncias. Já na remoção (Figura 5.22, C), **os *smells Insufficient Modularization e Multifaceted Abstraction* foram os mais removidos, embora sem representar um número significativo que indique queda**. Isso **ajuda a explicar esta tendência de comportamento constante dos *smells***. Os **outros *smells* da granularidade estão presentes em número baixo e controlado**, sem **se observar um esforço de redução** dos mesmos.

Finalmente, na Figura 5.23 são apresentados as informações de *methods* analisadas.

Diferentemente da granularidade de *types*, a granularidade de *methods* apre-

Figura 5.20 – Apache Ant - Distribuição dos *smells* detectados por versão analisada

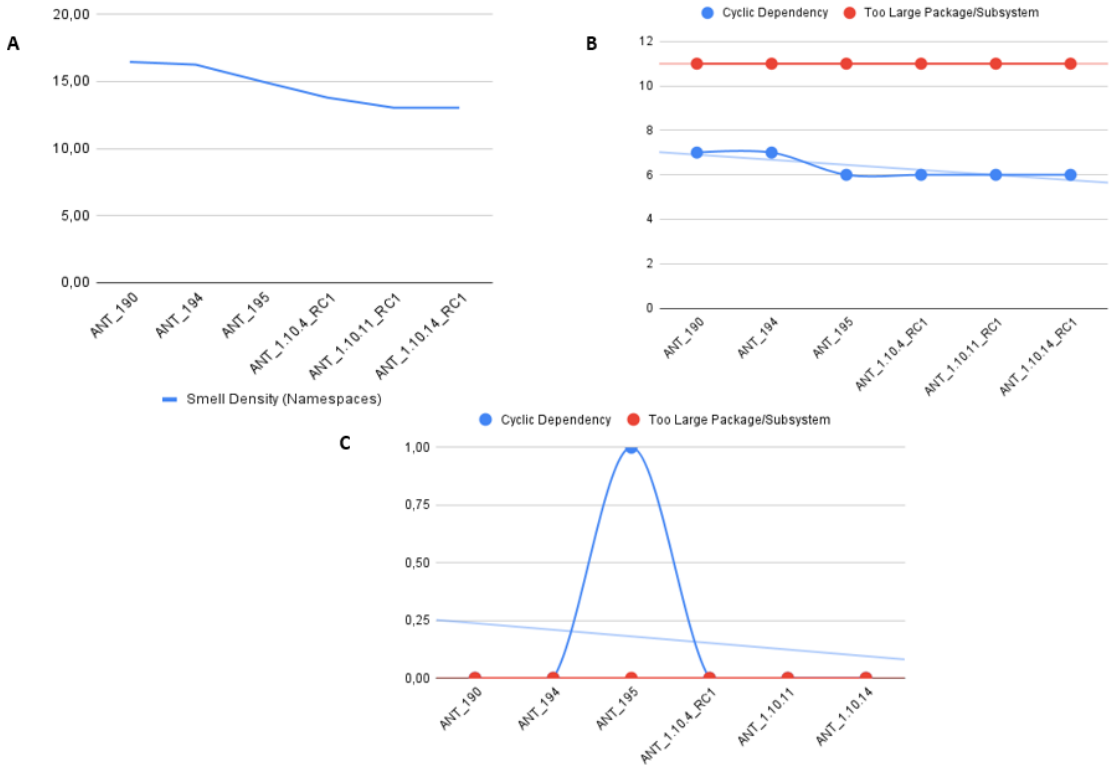


Fonte: O Autor

senta (Figura 5.23, A) um comportamento de queda leve nas versões *ant_195* e *ant_1.10.4*, porém nada significativo. **Nota-se que o número de métodos com *smells* é baixo, não chegando a 15%**. Mais especificamente, é possível observar um comportamento praticamente constante em todas as versões analisadas (Figura 5.23, B). Ao analisar o gráfico de remoção (Figura 5.23, C) **os *smells* que tiveram maior redução de instâncias foram *Complex Method* e *Long Method***. No entanto, as adições foram no mesmo nível o que explica essa tendência constante. Os **outros *smells* dos *methods* estão presentes em número baixo e controlado, sem se observar um esforço de redução** dos mesmos. Outro destaque **está relacionado com o número de refatorações. Nas versões *ant_1.10.4* e *ant_1.10.11* foi aplicado o maior número de refatorações (n=11165), porém sem redução significativa de *smells***.

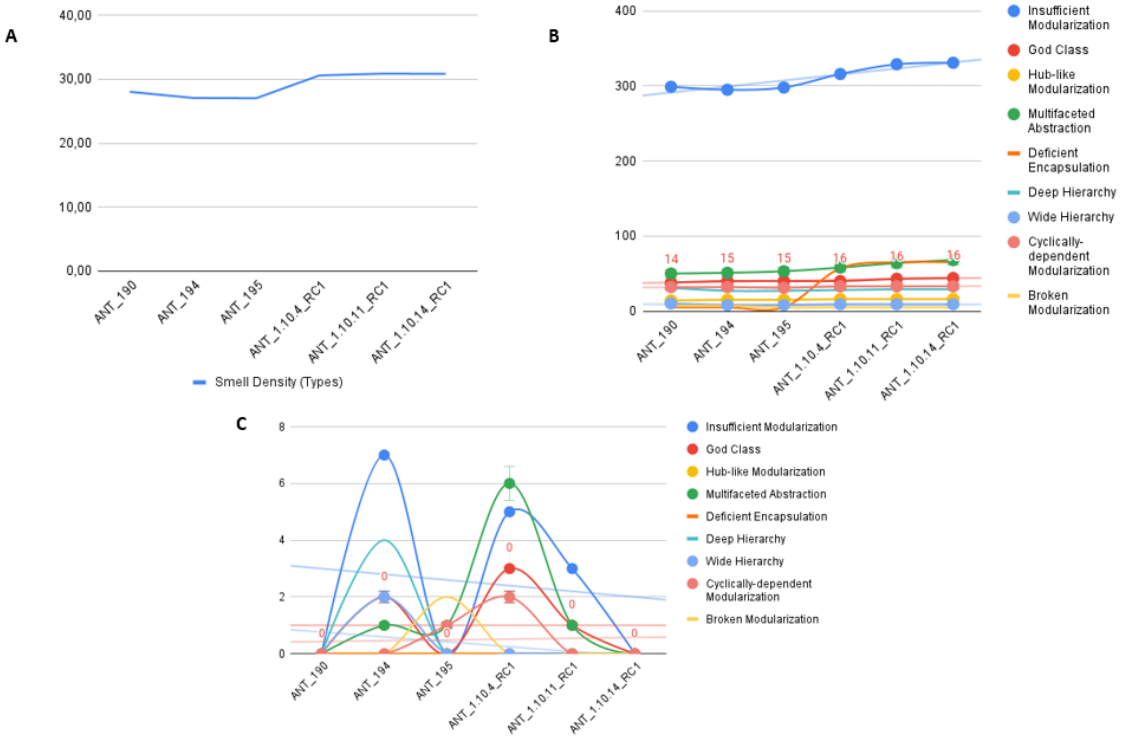
Outro projeto analisado foi o *Apache Storm*. Conforme apresentado na Figura

Figura 5.21 – Evolução dos *smells* de *namespaces* nas versões analisadas do Apache Ant - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



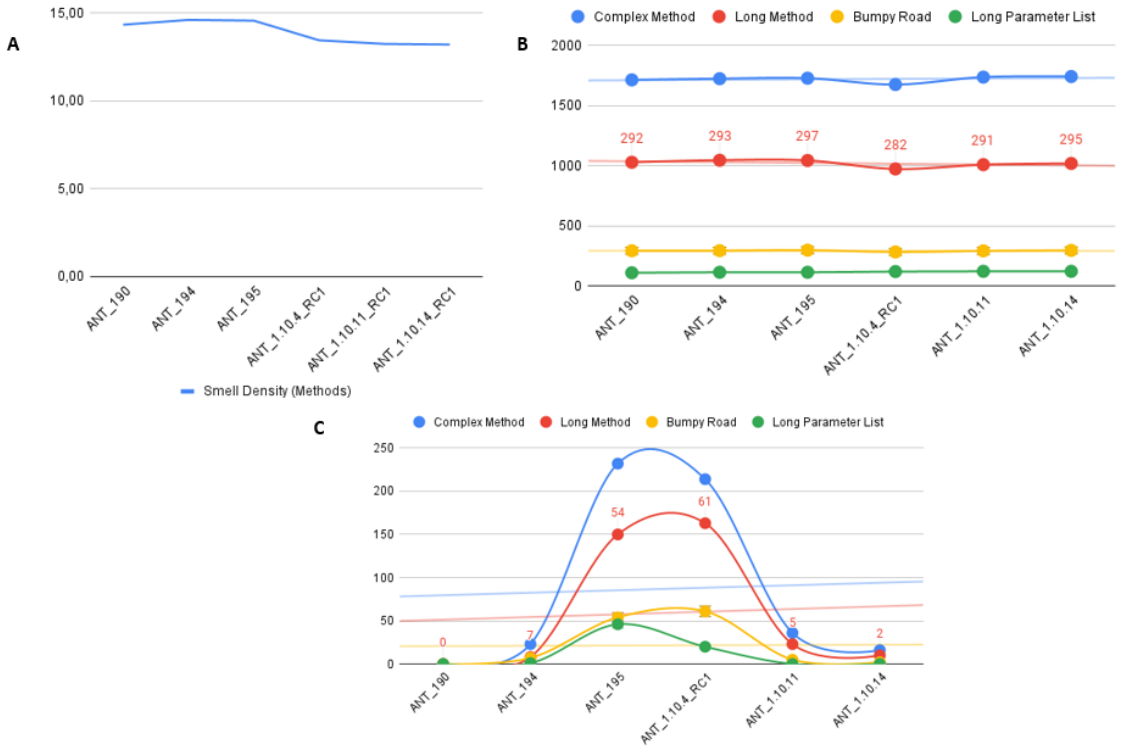
Fonte: O Autor

Figura 5.22 – Evolução dos *smells* de *types* nas versões analisadas do Apache Ant - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

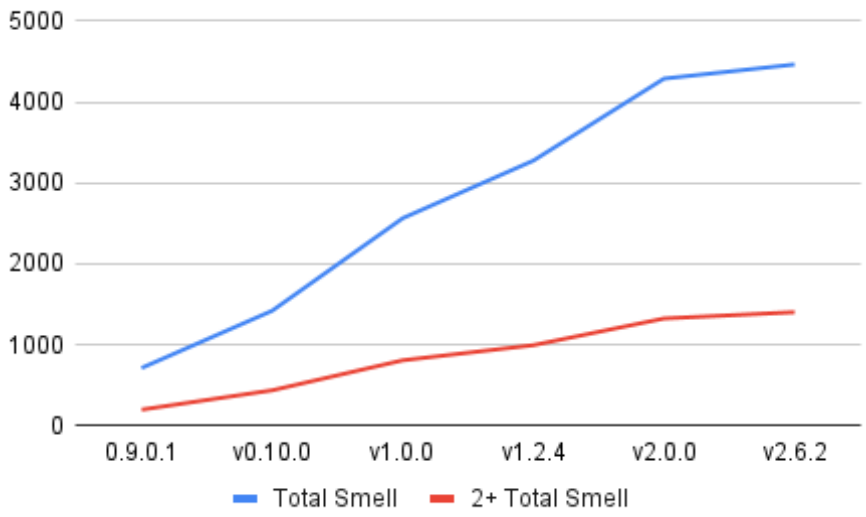
Figura 5.23 – Evolução dos *smells* de *methods* nas versões analisadas do Apache Ant - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

5.24, é possível observar a evolução dos elementos de código com *smells*, considerando todas as granularidades (*namespace, type, method*).

Figura 5.24 – Apache Storm - Evolução dos elementos de código com *smell* (azul) e com mais de um *smell* (vermelho), considerando todas as granularidades



Fonte: O Autor

Na linha azul, é exibido a quantidade de elementos com pelo menos um *smell*, enquanto a linha vermelha representa elementos de código com mais de um *smell*. Observa-se que, a partir da *0.9.0.1*, tem-se um aumento gradual dos *smells* até chegar no ápice da *2.0.0*. Entre a versões *2.0.0* e *2.6.2*, ocorreu um aumento baixo, não significativo como os anteriores. Porém, este projeto possui uma tendência de aumento dos *smells*.

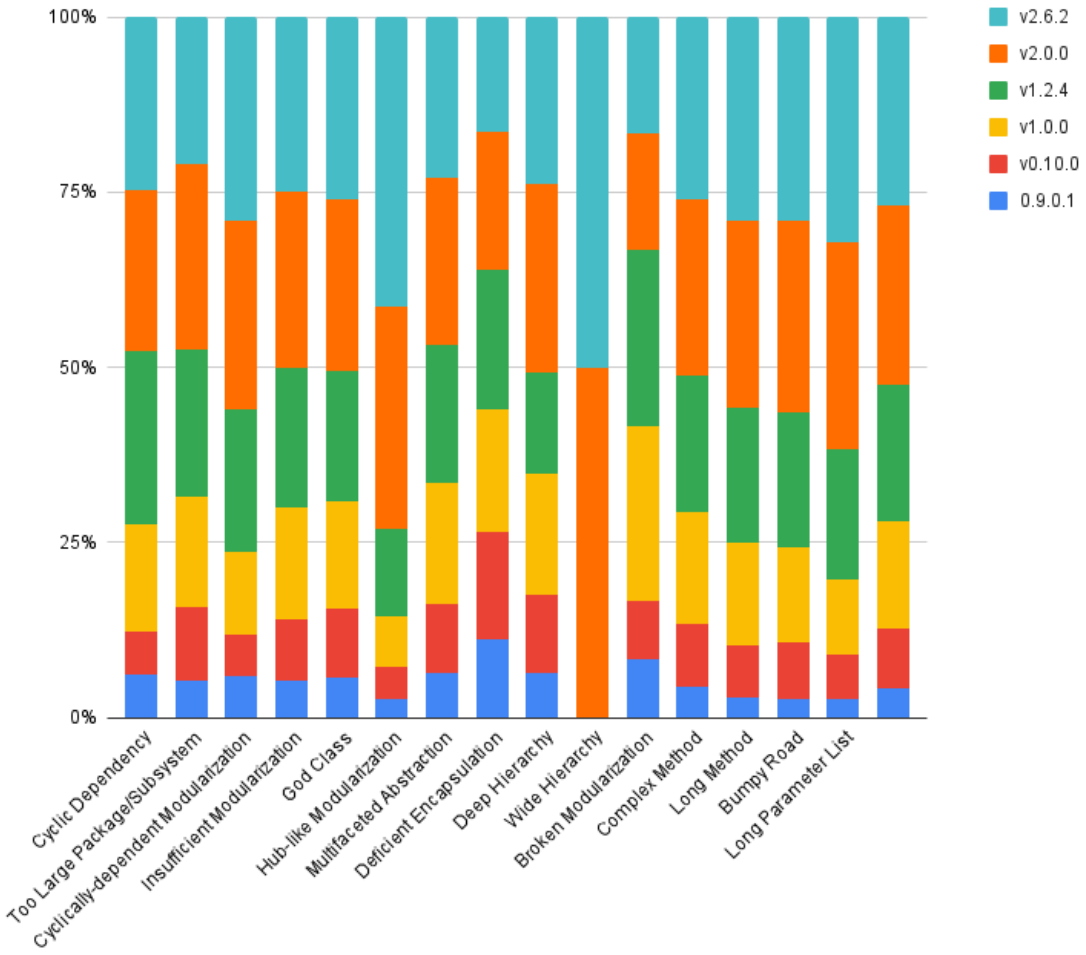
A Figura 5.25 ilustra a distribuição de todos os *smells* detectados, considerando a versão investigada.

As versões *0.9.0.1* e *0.10.0* apresentaram, inicialmente o menor número de *smells* em relação as outras versões. A partir da *1.0.0*, nota-se um crescimento das instâncias de todos os *smells*. **As versões *2.0.0* e *2.6.2* concentram o maior número de instâncias das versões analisadas do *Apache Storm*. Com exceção do *Wide Hierarchy*, todos os *smells* estão presentes em todas as versões analisadas**.

A Figura 5.26 ilustra as informações relacionadas a *namespaces* das versões analisadas do *Apache Storm*.

A **densidade percentual dos *smells* (Figura 5.26, A) nos *namespaces* indicou que na versão *0.9.0.1* chegou a 12%, com diminuição gradual nas versões subsequentes**. Isto ocorreu, principalmente, pelo aumento do número de *namespaces* no projeto. **Na versão *0.10.0* para a *1.2.4* (Figura 5.26, B) houve um aumento das instâncias dos *smells***. Ambos os *smells* detectados são difusos. Na versão *2.0.0* (Figura 5.26, C) ocorreu

Figura 5.25 – Apache Storm - Distribuição dos *smells* detectados por versão analisada



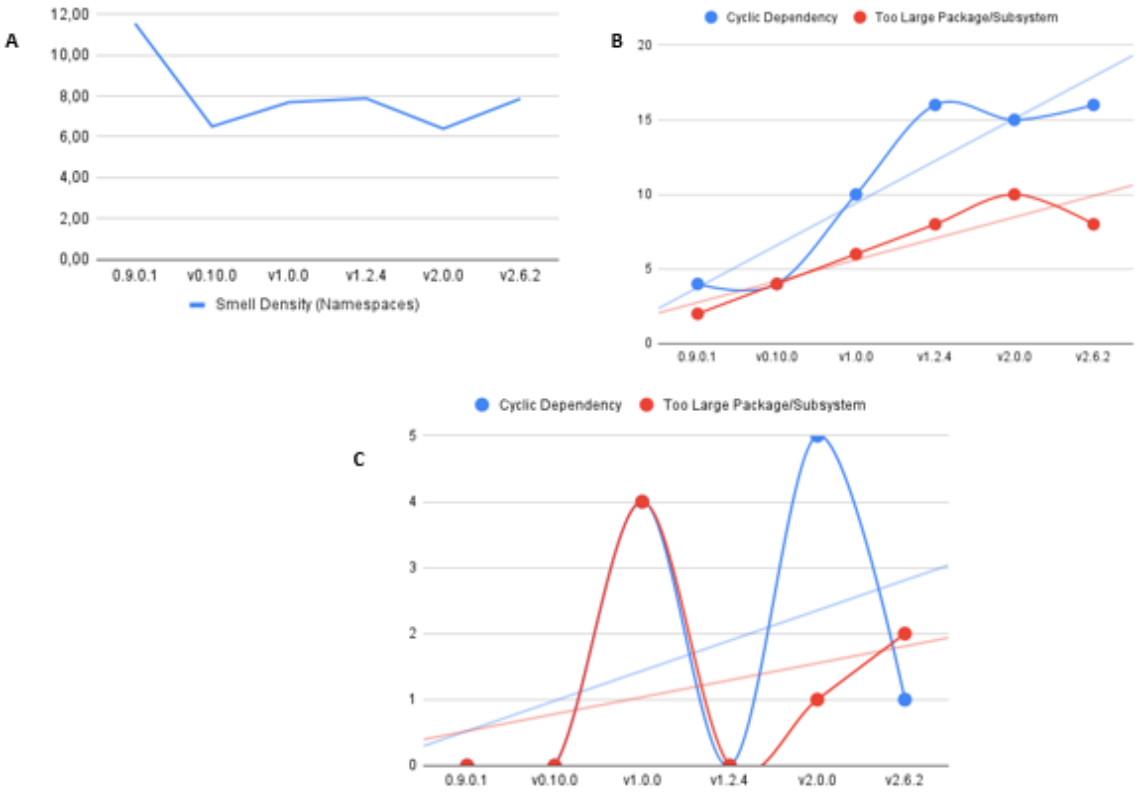
Fonte: O Autor

a maior redução de *smells* nesta granularidade, mas sem representação significativa.

Na Figura 5.27 são apresentadas informações relacionadas a *types* das versões analisadas.

Na granularidade de *types*, observa-se na Figura 5.27(A) um comportamento de diminuição gradual (versões *0.9.0.1*, com um comportamento constante nas demais versões das instâncias de *smells* por *types*, **chegando em torno de 25%**. Ao analisar o gráfico de evolução (Figura 5.27, B) e remoção (Figura 5.27, C), **os *smells* que tiveram maior adição de instâncias (*Insufficient Modularization, Multifaceted Abstraction*) também foram os que apresentaram maiores reduções**, não tendo influência na redução geral. Isso **ajuda a explicar essa esta tendência de aumento dos *smells***. Os **outros *smells* da granularidade estão presentes em número baixo e controlado**, mas **não se observa um esforço de redução** dos mesmos.

Figura 5.26 – Evolução dos *smells* de *namespaces* nas versões analisadas do Apache Storm - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



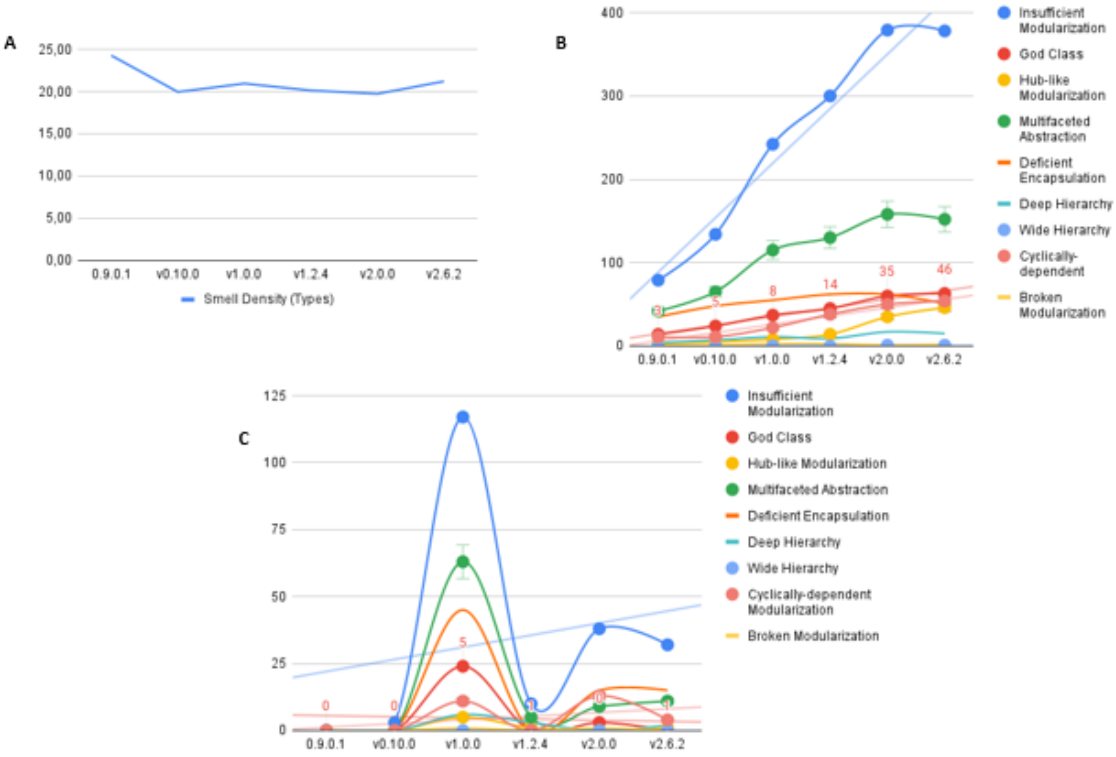
Fonte: O Autor

Finalmente, na Figura 5.28 são apresentados as informações de *methods* investigadas.

A granularidade de *methods* apresenta (Figura 5.28, A) um comportamento de leve aumento nas versões do *Apache Storm* das instâncias de *smells* por *methods*, possivelmente influenciada pela adição de novos métodos. Também, **observa-se que o número de métodos com *smells* é baixo, em torno de 16%**. Nota-se um crescimento contínuo em todas versões (Figura 5.28, B). Ao analisar o gráfico de remoção (Figura 5.28, C), **os *smells* que tiveram uma adição contínua de instâncias (*Complex Method, Long Method*) também foram os que apresentaram maiores reduções**. Especificamente, o *Complex Method* obteve uma redução temporária entre as versões *1.2.4* e *2.6.2*. Isso **reforça a explicação da tendência de aumento dos *smells* nos *methods***. Os **outros *smells* dos *methods* estão presentes em número baixo e controlado, sem se observar um esforço de redução** dos mesmos. Outro aspecto de destaque está relacionado **nas versões com maior número de refatorações, ocorrido entre as versões *1.2.4* e *2.0.0*, sem redução de *smells***.

O quarto projeto investigado é o *SpotBugs*. Conforme ilustrado na Figura 5.29,

Figura 5.27 – Evolução dos *smells* de *types* nas versões analisadas do Apache Storm - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

observa-se a evolução dos elementos de código com *smells*, considerando todas as granularidades (*namespace, type, method*). Na linha azul, é exibido a quantidade de elementos com pelo menos um *smell*, enquanto a linha vermelha representa elementos de código com mais de um *smell*. Nota-se, de forma geral, **um comportamento constante com uma leve variação de aumento e diminuição temporária dos *smells* entre as versões *3.0.0_rc1* até a *4.1.0***.

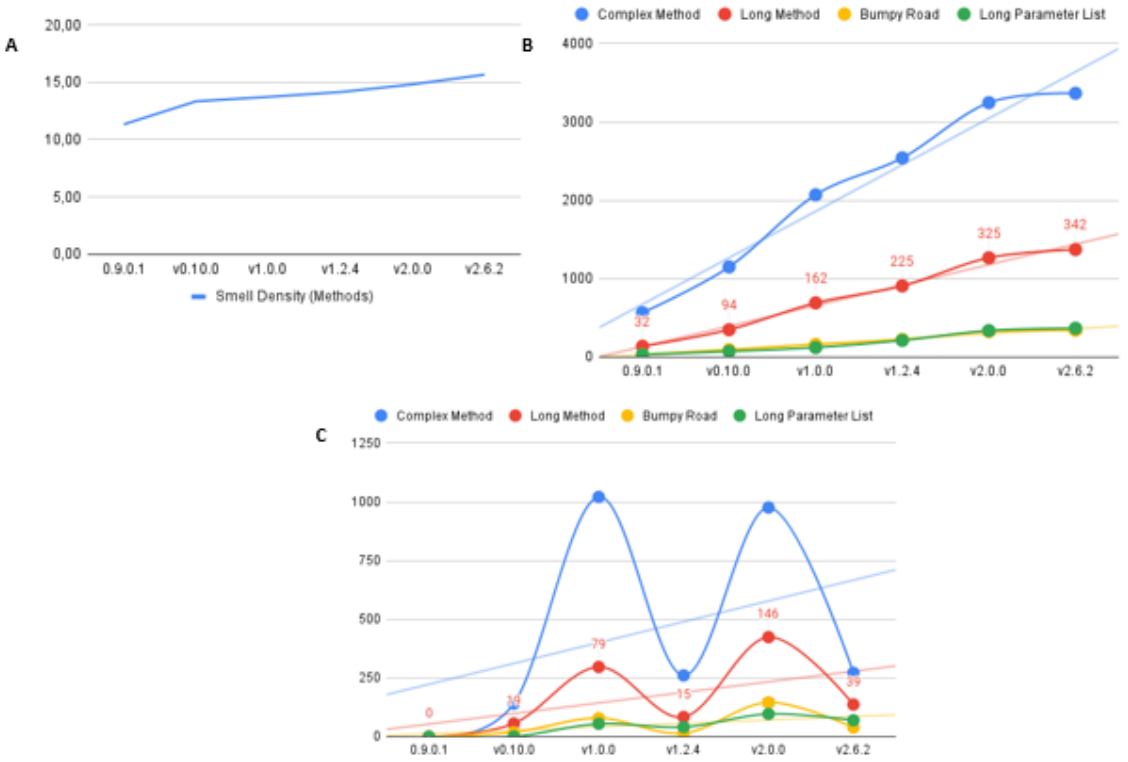
A Figura 5.30 ilustra a distribuição de todos os *smells* detectados, considerando a versão investigada.

Todas as versões do *SpotBugs* apresentaram todos os tipos diferentes de *smells*. **Da mesma forma que o *Apache Ant*, as versões analisadas não apresentaram grandes variações observáveis nas instâncias dos *smells***.

A Figura 5.31 apresenta as informações relacionadas a *namespaces* das versões analisadas do *SpotBugs*.

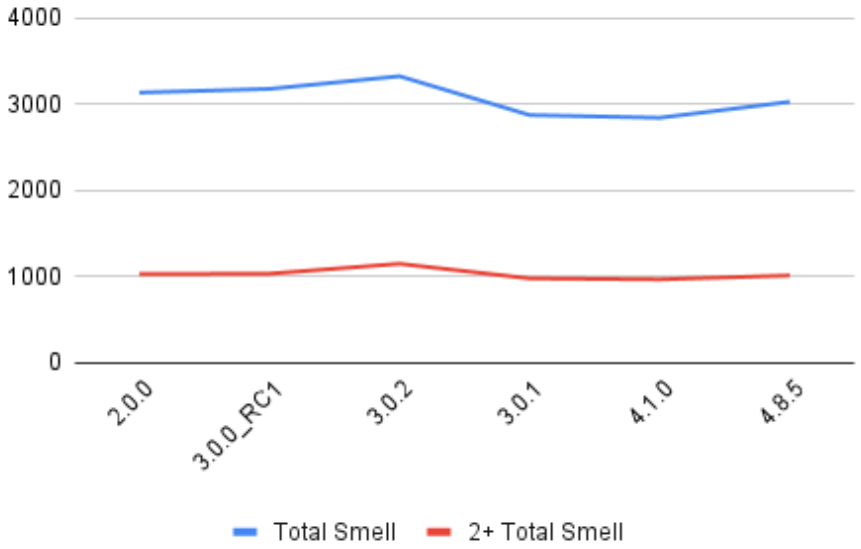
A **densidade percentual dos *smells* (Figura 5.31, A) nos *namespaces* indicou um comportamento constante nas versões analisadas. Cerca de 10% dos *namespaces* apresentaram algum tipo de *smell***. Observando o comportamento praticamente

Figura 5.28 – Evolução dos *smells* de *methods* nas versões analisadas do Apache Storm - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



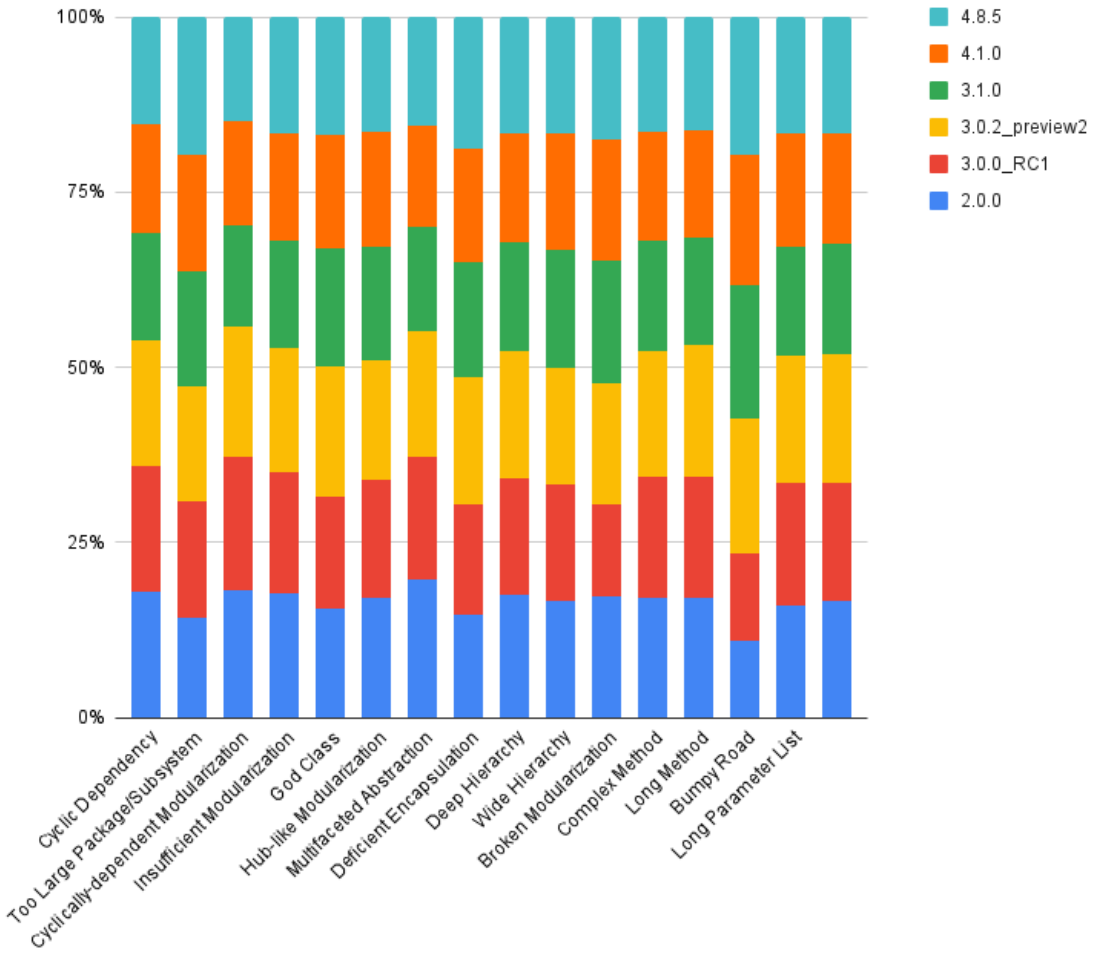
Fonte: O Autor

Figura 5.29 – SpotBugs - Evolução dos elementos de código com *smell* (azul) e com mais de um *smell* (vermelho), considerando todas as granularidades



Fonte: O Autor

Figura 5.30 – SpotBugs - Distribuição dos *smells* detectados por versão analisada



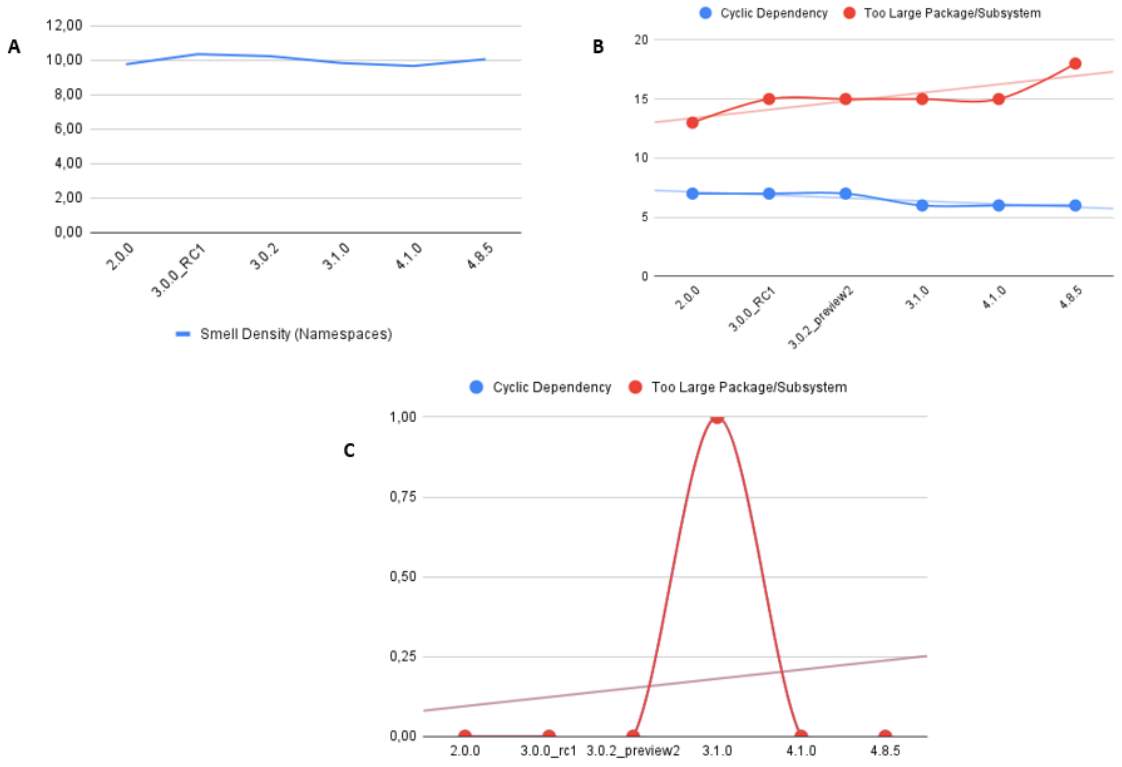
Fonte: O Autor

constante do número de *namespaces* no projeto (Figura 5.31, B) com adições nada significativas do *Too Large Package/Subsystem*. **Ocorreu apenas uma remoção de cada instância do *Cyclic Dependency* e *Too Large Package/Subsystem*, na versão *3.1.0*** (Figura 5.31, C). *Cyclic Dependency* e *Too Large Package/Subsystem* são difusos, com um número constante de instâncias.

Na Figura 5.32 são mostradas as informações relacionadas a *types* das versões investigadas.

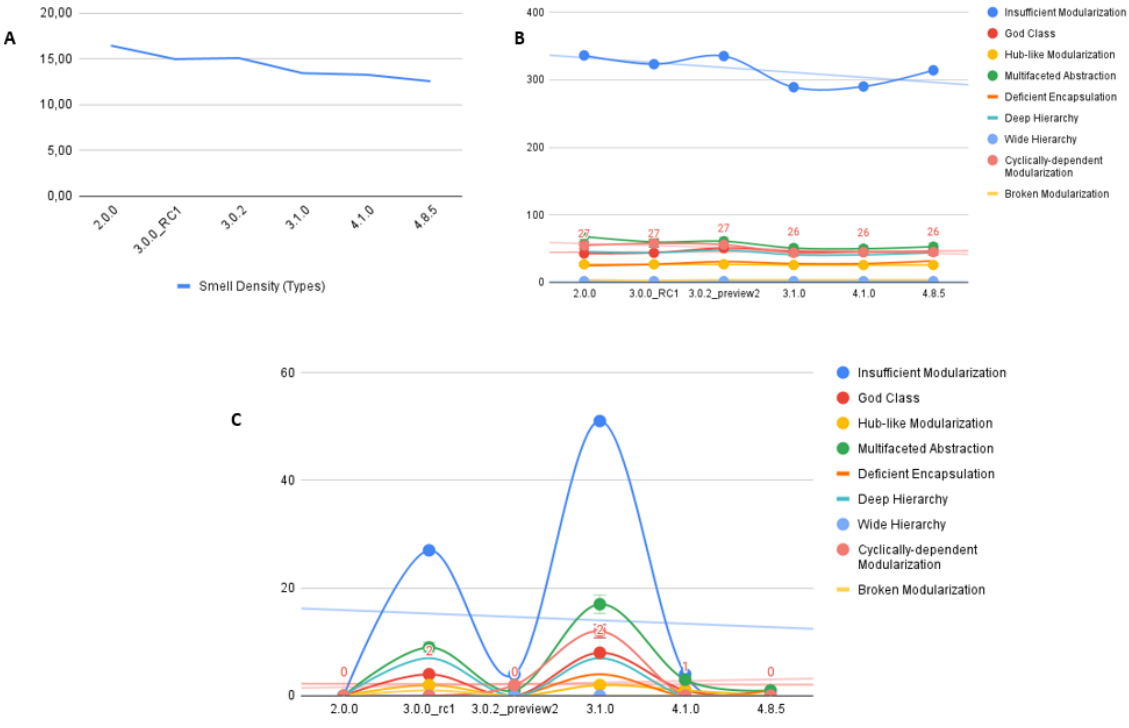
Na granularidade de *types*, observa-se na Figura 5.32(A) um comportamento de redução leve, mas não significativo em todas as versões das instâncias de *smells* por *types*. A **proporção de *types* infectados não ultrapassou 17%**. Ao analisar o gráfico de evolução (Figura 5.32, B), é possível observar que o *smell Insufficient Modularization* se destaca dos demais, mesmo mantendo um comportamento constante com aumento e

Figura 5.31 – Evolução dos *smells* de *namespaces* nas versões analisadas do SpotBugs - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

Figura 5.32 – Evolução dos *smells* de *types* nas versões analisadas do SpotBugs - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos

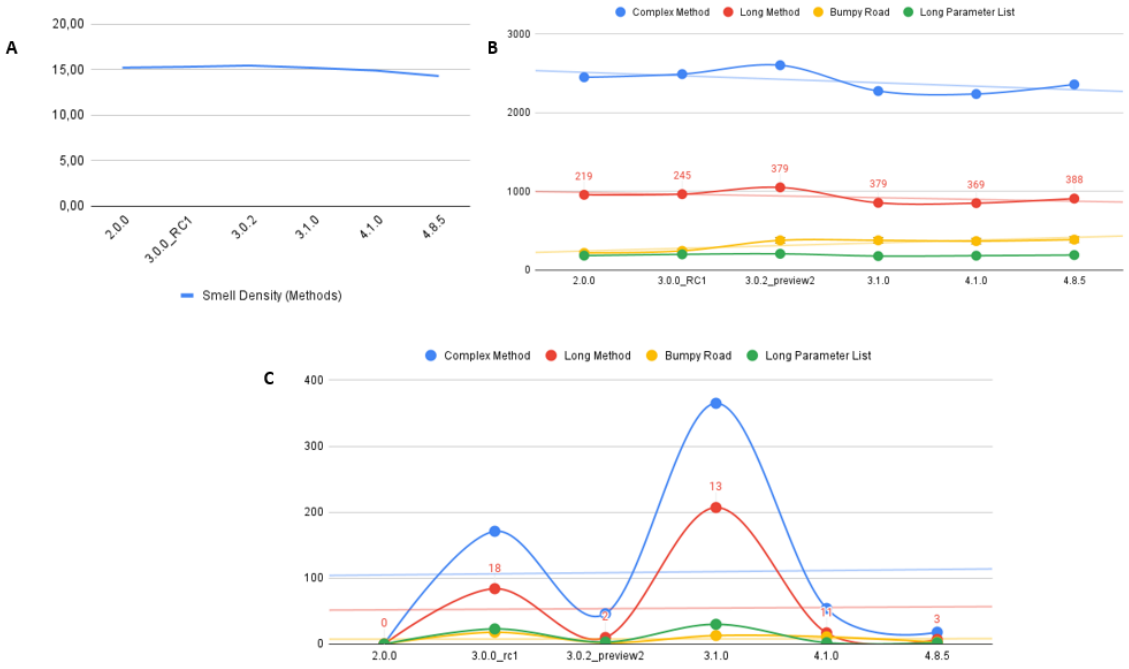


Fonte: O Autor

redução temporária no número de instâncias. Já na remoção (Figura 5.32, C), **o mesmo *smell* foi o mais removido, embora sem representar um número significativo que indique queda, justamente pelas adições na mesma frequência**. Isso **ajuda a explicar essa esta tendência de comportamento constante dos *smells***. Os **outros *smells* da granularidade estão presentes em número baixo e controlado**, sem **se observar um esforço de redução** dos mesmos.

Finalmente, na Figura 5.33 são apresentados as informações de *methods* investigadas.

Figura 5.33 – Evolução dos *smells* de *methods* nas versões analisadas do SpotBugs - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



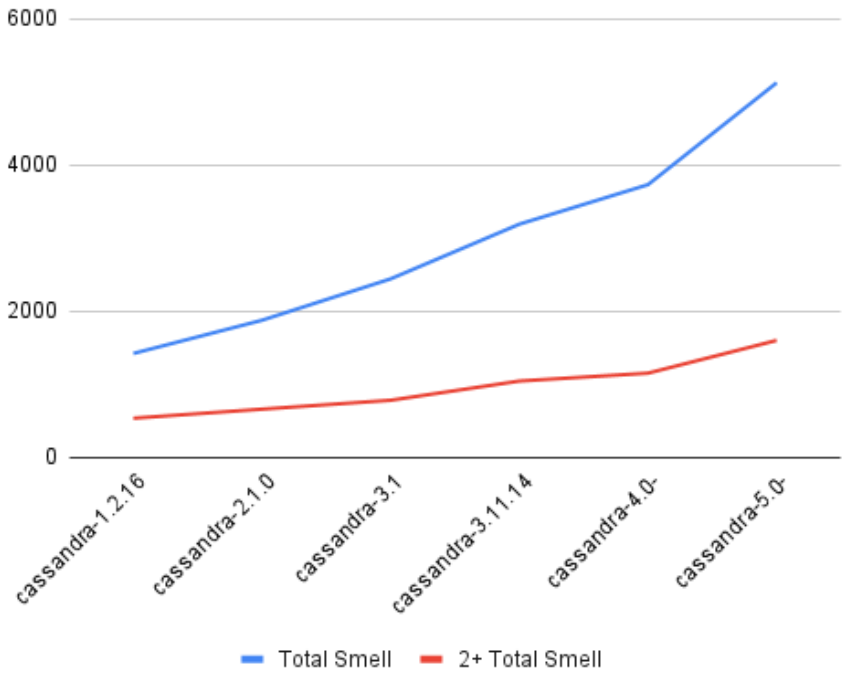
Fonte: O Autor

A granularidade de *methods* apresenta (Figura 5.33, A) um comportamento constante em todas as versões (similar a análise dos *namespaces*), porém nada significativo. **Nota-se que o número de métodos com *smells* é baixo, não ultrapassando 15%**. Mais especificamente, nota-se um comportamento praticamente constante em todas as versões analisadas (Figura 5.33, B), com uma leve oscilação nos *smells*. Ao analisar o gráfico de remoção (Figura 5.33, C) **os *smells* que tiveram maior redução de instâncias foram *Complex Method* e *Long Method***. No entanto, as adições foram com a mesma frequência o que explica essa tendência constante. Os **outros *smells* dos *methods* estão presentes em número baixo e controlado, sem se observar um esforço de redução** dos mesmos. Em **relação ao número de refatorações, uso significativo de refatorações**

não afetou a redução dos *smells*.

Finalmente, o quinto projeto investigado foi o *Apache Cassandra*. Conforme apresentado na Figura 5.34, é possível notar a evolução dos elementos de código com *smells*, considerando todas as granularidades (*namespace, type, method*).

Figura 5.34 – Apache Cassandra - Evolução dos elementos de código com *smell* (azul) e com mais de um *smell* (vermelho), considerando todas as granularidades



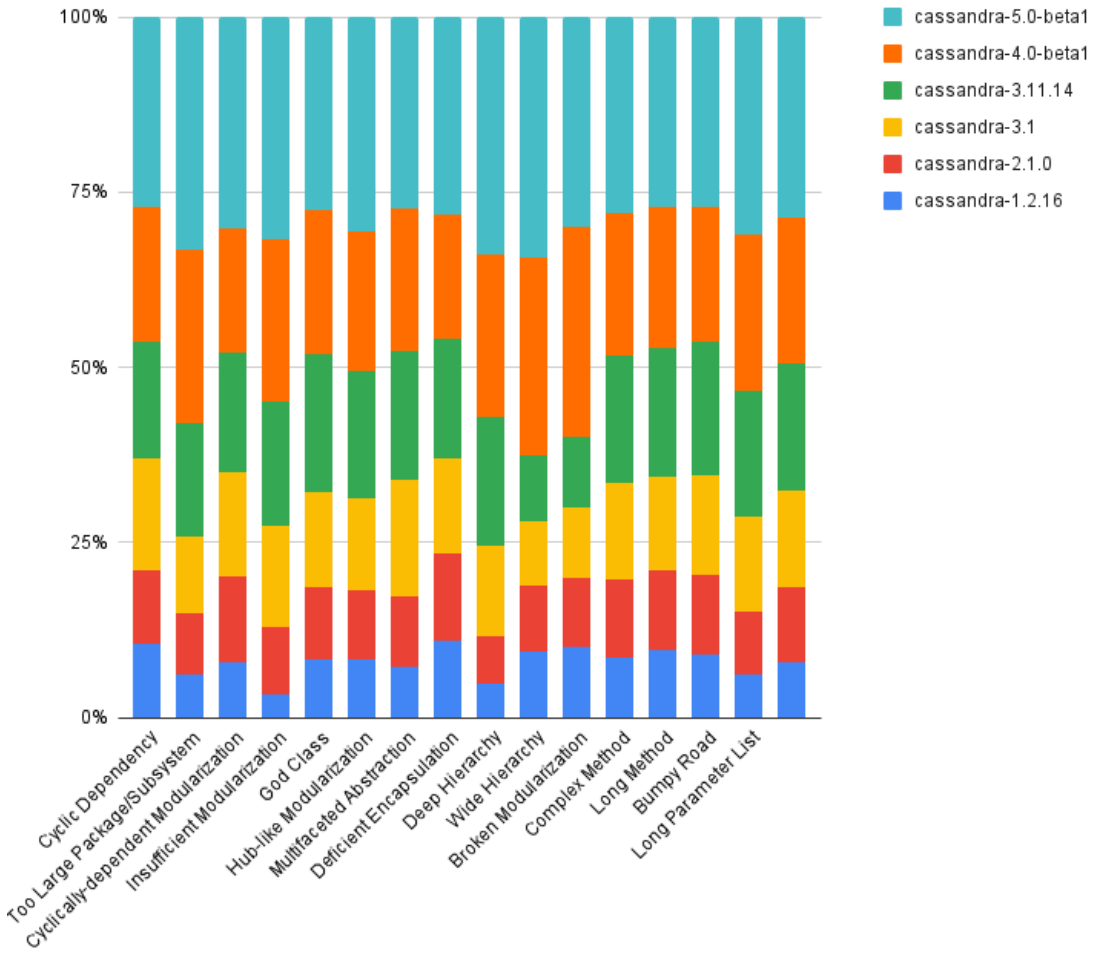
Fonte: O Autor

Na linha azul, é exibido a quantidade de elementos com pelo menos um *smell*, enquanto a linha vermelha representa elementos de código com mais de um *smell*. Observa-se que, a partir da *cassandra-1.2.16*, tem-se um aumento gradual dos *smells* até chegar no ápice da versão *cassandra-5.0-beta1*. Entre a versões *cassandra-1.2.16* e *cassandra-4.0-beta1*, ocorreu um aumento contínuo, com um aumento ainda mais considerável para a *cassandra-5.0-beta1*. Portanto, **este projeto possui uma tendência de aumento crescente dos *smells*.**

A Figura 5.35 ilustra a distribuição de todos os *smells* detectados, considerando a versão analisada.

Da mesma forma que o *Apache Storm*, as versões *cassandra-1.2.16* até *cassandra-3.1* apresentaram, inicialmente o menor número de *smells* em relação as outras versões. A partir da versão *cassandra-3.11.14*, observa-se um crescimento das instâncias de todos os *smells*. **As versões *cassandra-4.0-beta1* e *cassandra-5.0-beta1* concentram o maior**

Figura 5.35 – Cassandra - Distribuição dos *smells* detectados por versão analisada



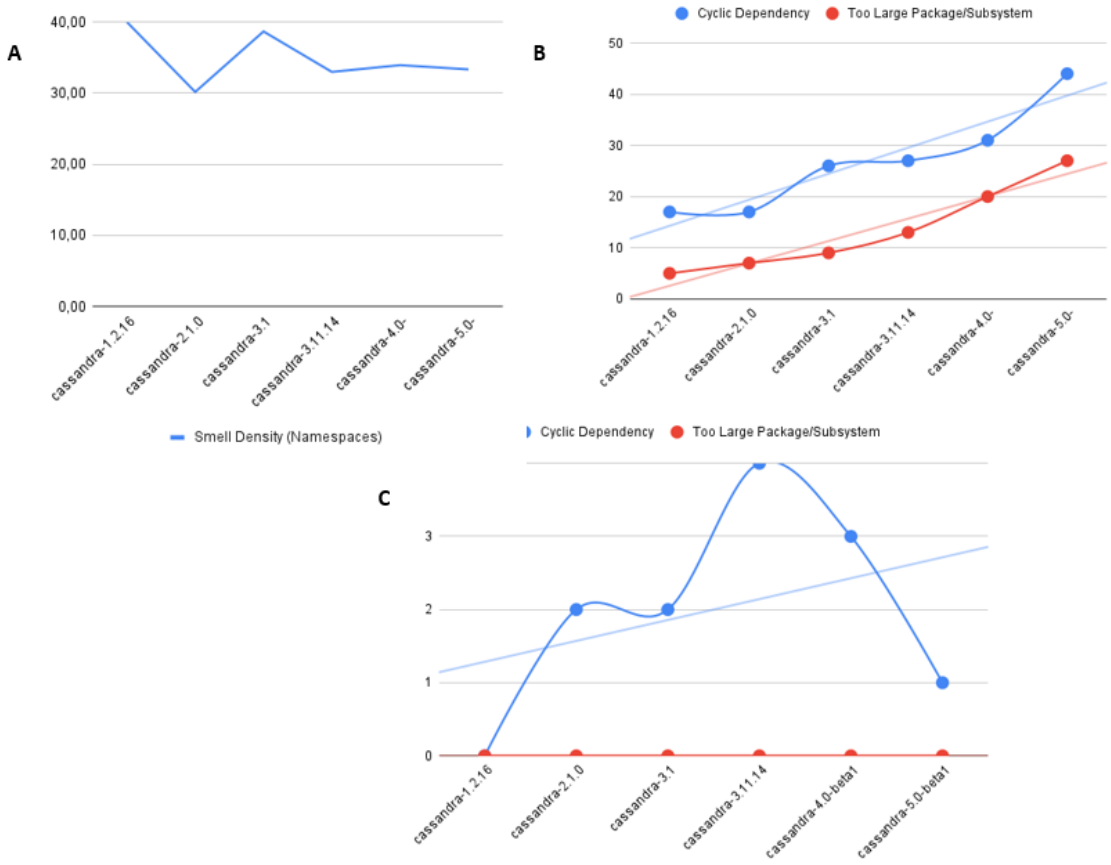
Fonte: O Autor

número de instâncias das versões investigadas do *Apache Cassandra*. Todos os *smells* detectados estão presentes em todas as versões analisadas.

A Figura 5.36 mostra as informações relacionadas a *namespaces* das versões investigadas do *Apache Cassandra*.

A **densidade percentual dos *smells* (Figura 5.36, A) nos *namespaces* indicou que na versão *cassandra-1.2.16* chegou a 40%, com variação de diminuição/aumento temporário nas versões seguintes**, estabilizando na versão *cassandra-4.0-beta1*. Isto ocorreu, principalmente, pelo aumento do número de *namespaces* no projeto. **A partir da versão *cassandra-2.1.0* (Figura 5.36, B) houve um aumento das instâncias dos *smells***. Ambos os *smells* detectados são difusos. O *smell Too Large Package/Subsystem* não apresentou redução. Apenas o *Cyclic Dependency* teve redução de *smells* nesta granularidade, em um número muito baixo de instâncias (Figura 5.36, C).

Figura 5.36 – Evolução dos *smells* de *namespaces* nas versões analisadas do Apache Cassandra - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

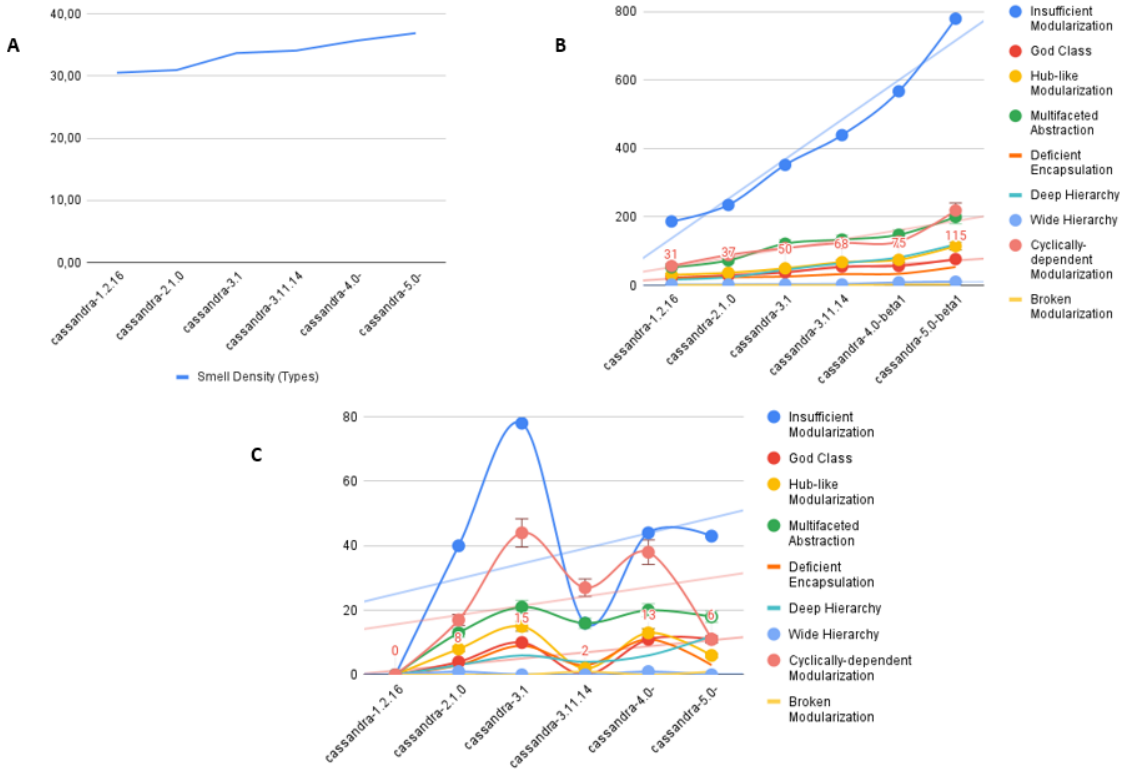
Na Figura 5.37 são apresentadas informações de *types* das versões investigadas.

Na granularidade de *types*, observa-se na Figura 5.37(A) um comportamento de aumento gradual a partir da versão *cassandra-2.1.0* das instâncias de *smells* por *types*, **oscilando entre 30% e 40% dos *types***. Ao analisar o gráfico de evolução (Figura 5.37, B) e remoção (Figura 5.37, C), **o *smell* que teve maior adição de instâncias (*Insufficient Modularization*) também foi o que apresentou maior redução**, não tendo influência na redução geral. Isso **ajuda a explicar essa esta tendência de aumento dos *smells***. Os **outros *smells* da granularidade estão presentes em número baixo e controlado**, mas **não se observa um esforço de redução** dos mesmos.

Finalmente, na Figura 5.38 são apresentados as informações de *methods* analisadas.

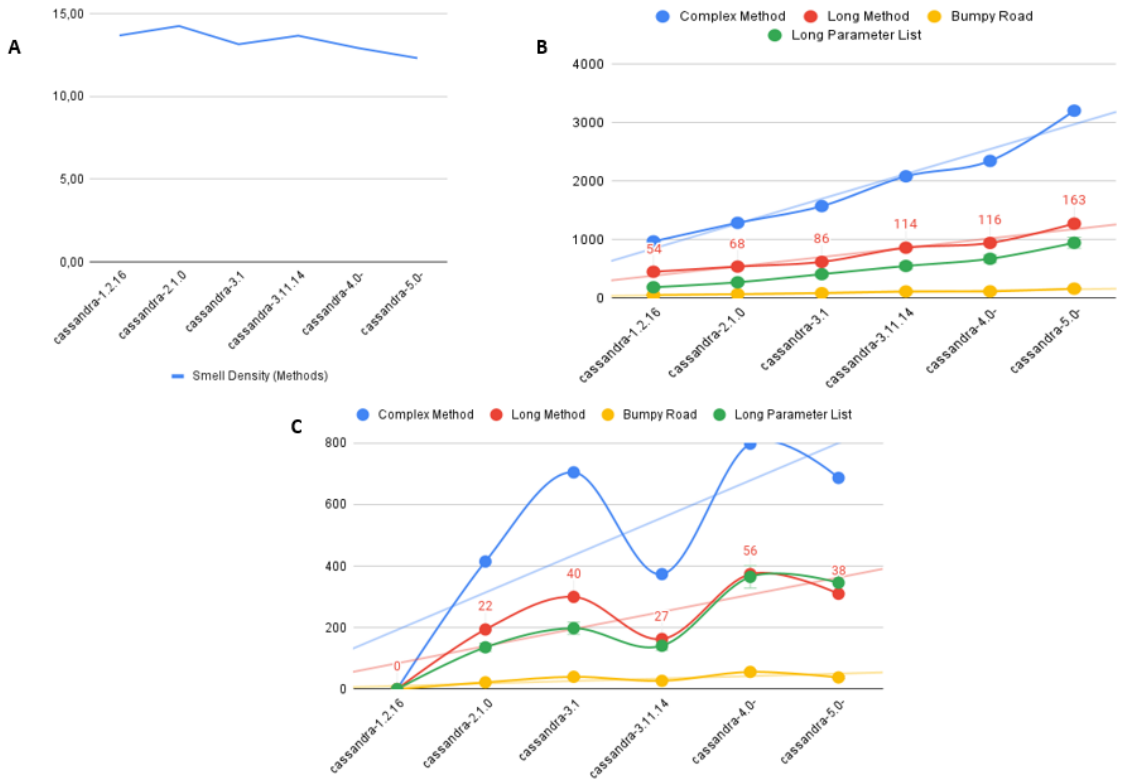
A granularidade de *methods* apresenta (Figura 5.38, A) um comportamento de leve diminuição nas versões do *Apache Cassandra* das instâncias de *smells* por *methods*, possivelmente influenciada pela adição de novos métodos. Também, **observa-se**

Figura 5.37 – Evolução dos *smells* de *types* nas versões analisadas do Apache Cassandra - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

Figura 5.38 – Evolução dos *smells* de *methods* nas versões analisadas do Apache Cassandra - (A) Densidade percentual dos *smells*. (B) Evolução dos *smells* específicos da granularidade. (C) Momento em que os *smells* foram removidos



Fonte: O Autor

que o número de métodos com *smells* é baixo, **entre 12% e 15%**. Nota-se uma crescimento contínuo em todas versões (Figura 5.38, B). Ao analisar o gráfico de remoção (Figura 5.38, C), **os *smells* que tiveram uma adição contínua de instâncias também foram os que apresentaram maiores reduções, com destaque para o *Complex Method***. Isso **reforça a explicação da tendência de aumento dos *smells* nos *methods***. Os **outros *smells* dos *methods* estão presentes em número baixo e controlado**, **sem se observar um esforço de redução** dos mesmos. Como observado anteriormente nos projetos, **as refatorações realizadas não tiveram relação com a redução de *smells***.

Observa-se que de forma geral, como já relatado em trabalhos anteriores (CHATZIGEORGIOU; MANAKOS, 2010; PETERS; ZAIDMAN, 2012), o número de *smells* tem crescido ao longo tempo e, de certa forma, os desenvolvedores ficam relutantes ao aplicar refatorações para removê-los. De fato, os *smells* tem sido utilizados como indicador de manutenibilidade já há algum tempo (YAMASHITA; COUNSELL, 2013). Inclusive, não notou-se relação do uso das refatorações com as reduções dos *smells*. Alguns trabalhos já haviam alertado sobre este aspecto (SILVA; TSANTALIS; VALENTE, 2016; CEDRIM et al., 2017; LACERDA et al., 2020). Segundo Tufano et al. (2017), cerca de 80% dos *smells* nunca são removidos dos sistemas e sua principal causa de remoção é a remoção do código em si, ao invés de refatorações.

Embora todos os *smells* sejam resultados de violações de princípios de *design* em sistemas de software OO, os motivos por trás de um determinado *smell* podem explicar as causas e a negatividade desse *smell*, o que ajudaria na priorização (VERMA; KUMAR; VERMA, 2023).

Assim, **dos projetos analisados, apenas um (*JetUML*) apresentou redução de efetiva dos *smells* entre as versões**, ao contrário do que apontado por Chatzigeorgiou and Manakos (2010). Esta redução é o fenômeno pesquisado para identificar se teve algum tipo de priorização, mesmo que *ad hoc*. Neste caso, o desenvolvedor líder do projeto foi contatado. O objetivo foi entender se teve alguma estratégia de priorização utilizada no projeto. **Conforme relatado pelo desenvolvedor, as principais melhorias são motivadas pensando em mudanças futuras**, ou seja, ajustando e adaptando o código. Ainda, **quando se pode implementar pequenas melhorias, elas são incentivadas**, ou seja, **são mais oportunísticas do que planejadas**. Muitas destas modificações foram compiladas em um material disponibilizado pelo próprio líder do projeto[17].

De fato, o ato de priorizar é fundamental devido ao grande número de *smells*,

[17]<https://www.cs.mcgill.ca/~martin/papers/software2019.pdf>

ao menos para apoiar os desenvolvedores em suas decisões. Assim, neste estudo exploratório, buscou-se analisar visualmente, combinando diferentes gráficos para apoiar as investigações. Para observar o comportamento dos *smells* em relação a sua evolução, utilizou-se a técnica DTW para análise. Ao combinar os gráficos de densidade percentual dos *smells* com sua evolução e, consequentemente remoção, propicia uma visão geral dos acontecimentos relacionados a granularidade analisada. O *smell churn* pode ser uma medida interessante de estudo na evolução e difusão dos *smells*. Esta medida pode servir de início para ajudar a entender quando e como os *smells* são inseridos e removidos.

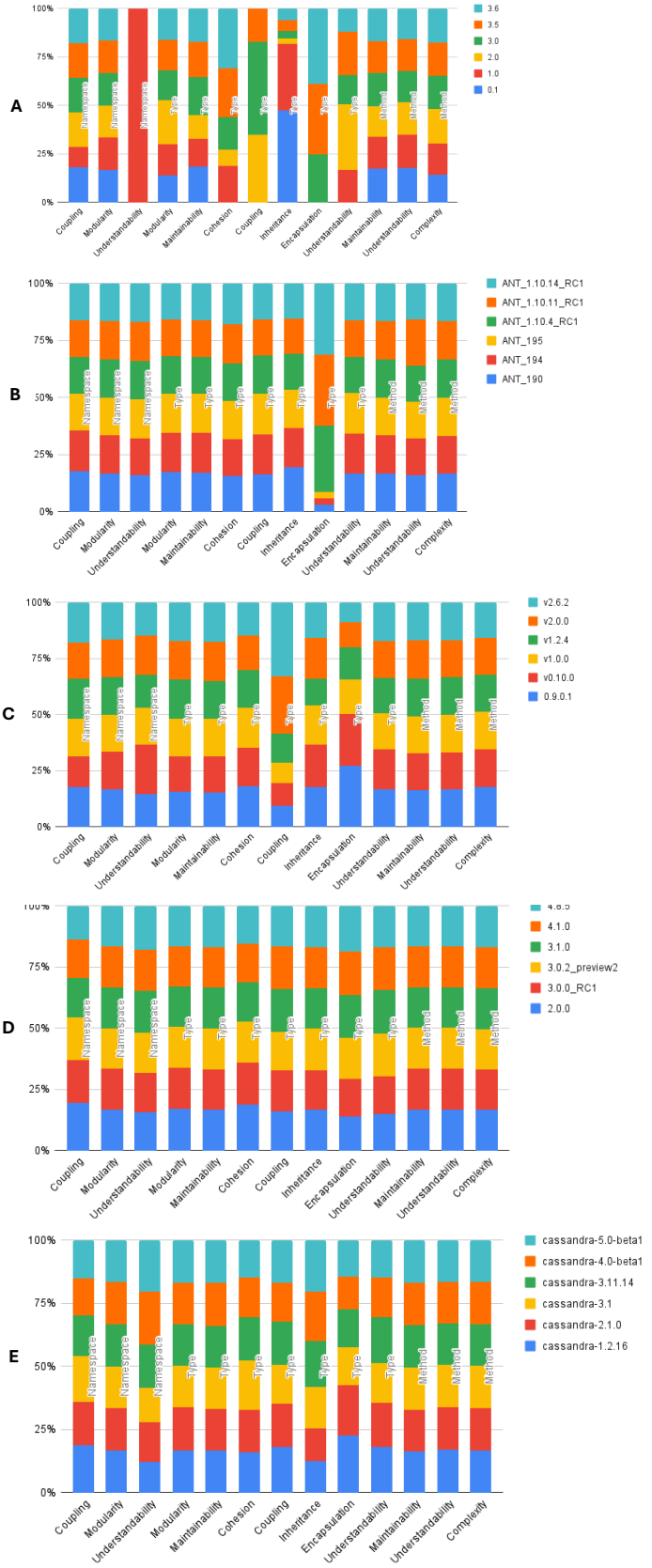
Resposta resumida a RQ3

Na maioria dos projetos analisados, se observou um aumento gradual dos *smells* ou um comportamento constante, que reforça os resultados de Tufano et al. (2017).
O *smell churn rate* pode ser uma medida que ajuda a entender as ações de priorização. Quanto maior for essa medida, mais se obteve redução de *smells* entre versões.
Apenas o projeto *JetUML* apresentou queda de *smells* entre as versões analisadas, com *smell churn rate* médio de 39.00.
Não foi utilizado um método de priorização, apenas uma estratégia de lançamento de versão e melhorias oportunistas, ou seja, que podem ser feitas em elementos de código que estão sendo modificados.

Smells e o Impacto na Qualidade: Vários modelos e padrões surgiram para definir e avaliar a qualidade de software (NISTALA; NORI; REDDY, 2019). Os mais recentes, como ISO/IEC 25010 (NORMALISATION, 2011) e a ISO 5055:2021 (CURTIS; MARTIN; DOUZIECH, 2022), definem características e atributos de qualidade, como a manutenibilidade. Embora se tenha essas características definidas, ainda há dificuldade de medi-las (BAKOTA et al., 2011). Conforme Vassallo et al. (2019), além dos *smells*, a qualidade tem sido um grande motivador na maioria dos casos e que necessitam de atenção. Sabendo que os atributos de qualidade são afetados por vários *smells* e que os *smells* impactam mais de um atributo de qualidade, procurou-se investigar como os *smells* têm impactado os diferentes atributos de qualidade durante a evolução do software. O *DR-Tools Code Health* associa os *smells*, de diferentes granularidades, com atributos de qualidade impactados por estes, seguindo o que recomenda Yamashita (2015). Na Tabela 4.4 é apresentada a lista completa dos *smells* e sua conexão com os atributos.

Assim, conforme apresentado pela Figura 5.39, são exibidos os percentuais de distribuição dos atributos de qualidade impactados por granularidade e versões dos projetos analisados.

Figura 5.39 – Impacto nos atributos de qualidade, segundo os *smells* detectados em cada versão dos projetos - (A) JetUML (B) Apache Ant (C) Apache Storm (D) SpotBugs (E) Apache Cassandra



Fonte: O Autor

No *JetUML* (Figura 5.39, A), **na granularidade de *namespace***, *coupling* e *modularity* se mantiveram basicamente da mesma forma em todas as versões. O *understandability* foi impactada somente na versão *1.0*. Assim, ***modularity* obteve um impacto médio de 50.00 (mediana=50.00; DP=0.0), sendo o mais impactado**. Já o *understandability* foi o menos impactado (média=3.33; mediana=0.0; DP=8.16). Na **granularidade de *type***, *modularity* e *maintainability* foram impactados, praticamente, da mesma forma em todas as versões. *Cohesion* e *understandability* foram impactados nas mesmas versões (*1.0, 2.0, 3.0, 3.5, 3.6*), mas de forma diferente. A *inheritance* foi mais impactada nas primeiras versões *0.1* e *1.0*, enquanto *encapsulation* foi mais impactada nas últimas versões (*3.0, 3.5, 3.6*). Os atributos **mais impactados foram *modularity* (média=47.48; mediana=45.51; DP=8.55) e *maintainability*** (média=35.54; mediana=37.73; DP=6.14). Já o atributos menos impactados foram *encapsulation* (média=1.18; mediana=0.88; DP=1.33) e *coupling* (média=1.23; mediana=0.64; DP=1.52). Na **granularidade de *method*** o impacto nos atributos *maintainability, understandability* e *complexity* foram muito similares, em todas as versões. O **atributo mais impactado foi *maintainability*** (média=39.24; mediana=39.40; DP=1.10) e o menos impactado foi *complexity* (média=24.66; mediana=25.49; DP=2.04).

O *Apache Ant* (Figura 5.39, B) manteve o mesmo padrão de impacto em quase todos os atributos de qualidade em todas as versões nas granularidades de *namespace, type* e *method*. A exceção foi o *encapsulation* que, nas primeiras 3 versões (*ant_190* a *ant_195*) foi bem baixa, se acentuando nas versões seguintes. Assim, **na granularidade de *namespace*, o *modularity* obteve um impacto médio de 50.00 (mediana=50.00; DP=0.0), sendo o mais impactado**. Já o *coupling* foi o menos impactado (média=18.25; mediana=17.65; DP=0.92). Na **granularidade de *type***, os **mais impactados foram *maintainability*** (média=41.24; mediana=41.29; DP=1.45) **e *modularity*** (média=38.61; mediana=38.73; DP=1.43). Os menos impactados foram *coupling* (média=1.70; mediana=1.68; DP=0.08) e *encapsulation* (média=3.50; mediana=3.34; DP=3.18). **Na granularidade de *method*, o mais impactado foi *understandability*** (média=38.68; mediana=37.08; DP=4.04) e o menos impactado foi *complexity* (média=24.56; mediana=24.57; DP=0.10).

O *Apache Storm* (Figura 5.39, C) manteve o mesmo padrão de impacto em quase todos os atributos de qualidade em todas as versões nas granularidades de *namespace, type* e *method*. As exceções foram o *encapsulation* (maior impacto na versão *0.9.0.1*) e o *coupling* (maior impacto na versão *2.6.2*). Na **granularidade de *namespace*, o**

modularity* obteve um impacto médio de 50.00 (mediana=50.00; DP=0.0), sendo o mais impactado**. Já o *understandability* foi o menos impactado (média=18.96; mediana=17.71; DP=3.27). Na **granularidade de *type*, os mais impactados foram *maintainability (média=36.96; mediana=37.32; DP=1.88) **e *modularity*** (média=33.77; mediana=34.40; DP=1.57). Os menos impactados foram *inheritance* (média=1.33; mediana=1.40; DP=0.21) e *coupling* (média=1.85; mediana=1.27; DP=1.13). **Na granularidade de *method*, o mais impactado foi *maintainability*** (média=37.44; mediana=37.39; DP=0.68) e o menos impactado foi *complexity* (média=27.07; mediana=27.00; DP=0.89).

O *SpotBugs* (Figura 5.39, D), similar ao *Apache Ant*, também manteve o mesmo padrão de impacto em todos os atributos de qualidade, considerando as granularidades *namespace, type* e *method* em todas as versões. **Na granularidade de *namespace*, o *modularity*** obteve um impacto médio de 50.00 (mediana=50.00; DP=0.0), sendo **o mais impactado**. Já o *coupling* foi o menos impactado (média=15.07; mediana=15.10; DP=1.74). **Na granularidade de *type*, os mais impactados foram *maintainability*** (média=40.56; mediana=40.58; DP=0.22) **e *modularity*** (média=38.26; mediana=38.13; DP=0.56). Os menos impactados foram *coupling* (média=2.76; mediana=2.73; DP=0.12) e *encapsulation* (média=2.97; mediana=3.07; DP=0.32). Na **granularidade de *method*, o mais impactado foi *maintainability*** (média=37.61; mediana=37.61; DP=0.12) e o menos impactado foi *complexity* (média=26.66; mediana=26.65; DP=0.23).

Finalmente, o *Apache Cassandra* (Figura 5.39, E) manteve um padrão similar de impacto nos atributos de qualidade em todas as versões dos demais projetos. As exceções foram o *encapsulation* (maior impacto na versão *cassandra-1.2.16*) e o *inheritance* (maior impacto na versão *cassandra-5.0-beta1*). **Na granularidade de *namespace*, o *modularity*** obteve um impacto médio de 50.00 (mediana=50.00; DP=0.0), sendo o **mais impactado**. O *understandability* foi o menos impactado (média=15.61 mediana=15.42; DP=3.31). Na **granularidade de *type*, os mais impactados foram *modularity*** (média=38.85; mediana=38.78; DP=0.61) **e *maintainability*** (média=37.86; mediana=38.01; DP=0.72). Os menos impactados foram *encapsulation* (média=2.44; mediana=2.21; DP=0.55) e *understandability* (média=3.38; mediana=3.38; DP=0.28). Na **granularidade de *method*, o mais impactado foi *maintainability*** (média=40.56; mediana=40.65; DP=0.49) e o menos impactado foi *complexity* (média=24.83; mediana=24.78; DP=0.17).

Nota-se que, com a associação do atributo de qualidade ao *smell* detectado, tem-se um mesmo padrão nos projetos que se mantiveram constante em relação a evolução dos *smells*. Novamente, o *JetUML* foi o que apresentou maiores variações de impacto

nos atributos de qualidade. Ao fazer a adição ou remoção de um dado *smell* impacta de forma geral na versão analisada. Assim, é necessário ampliar o número de projetos para observar o impacto nestes atributos, até avaliando possível padrões que possam emergir destas análises.

Resposta resumida a RQ4

De modo geral, os impactos nos atributos de qualidade nas diferentes granularidades e versões mantiveram um padrão similar. Possivelmente este resultado é consequência da associação de atributos de qualidade com *smell* detectados.
A exceção foi o *JetUML* por apresentar maior variação no impacto dos atributos de qualidade.
O atributo de qualidade mais impactado pelos *smells* na granularidade de *namespace* foi *modularity*.
Na granularidade de *type*, os atributos mais impactados foram *modularity* e *maintainability*.
Já granularidade de *method*, o atributo de qualidade mais impactado foi *maintainability*.

5.2.4 Ameaças à Validade

Nesta subseção, discute-se as ameaças à validade, incluindo a validade interna, externa, de construto e de conclusão, e as diferentes táticas adotadas para mitigá-las.

A validade interna refere-se aos fatores que podem ter influenciado este estudo. Foram escolhidos 5 projetos *open-source* conhecidos da comunidade, com certa reputação e atualização. Também, utilizou-se estratégias de detecção configuradas com parâmetros testados e ajustados previamente em mais de 50 projetos. No entanto, estes parâmetros usados podem influenciar os resultados da pesquisa.

A validação externa trata da generalização dos resultados da pesquisa. Certamente, é necessário ampliar a pesquisa para mais projetos e, talvez, ampliar a janela de tempo considerada. Embora o *DR-Tools Code Health* tenha a função de customização implementada, justamente para adequar os parâmetros e pesos ao contexto o projeto, este recurso não foi explorado. Para minimizar este impacto, os parâmetros utilizados na configuração *default* foram definidos a partir de uma variedade de projetos de contextos e tamanhos diferentes. Também, os *smells* especificados no *DR-Tools Code Health* abrangem um conjunto pequeno, porém relevante de *smells* conhecidos. Seria interessante ampliar esse conjunto de *smells*.

Tanto a validade de construto quanto a validade de conclusão são discutidas em conjunto. Conforme comentado anteriormente, para fins de delimitação de escopo, foram escolhidos somente 5 projetos, o que pode influenciar os resultados. Foi feito uso da

estatística descritiva, muito utilizada em trabalhos desta natureza. Pelo fato de ser um estudo exploratório, baseado na identificação de padrões gráficos, poderia-se utilizar em conjunto alguns métodos estatísticos específicos que poderiam ajudar a analisar os dados e entender melhor certos comportamentos. Uma vez que este estudo está focado no ato de priorização e no impacto dos *smells* na qualidade na evolução de software, não se levou em conta outros possíveis fatores influentes, como a percepção dos desenvolvedores que trabalharam nos sistemas ou os processos de desenvolvimento subjacentes. Em estudos futuros, é necessário encontrar uma maneira de lidar com esses fatores.

5.3 Depoimentos de Uso

Além dos experimentos descritos anteriormente, também foi realizada uma avaliação empírica da ferramenta *DR-Tools Code Health* com três desenvolvedores, de três empresas diferentes, localizadas na Alemanha (Dev#1), EUA (Dev#2) e Brasil (Dev#3). Os participantes possuem mais de 10 anos de experiência e, atualmente, trabalham em posições de liderança de times.

O processo de avaliação foi conduzido por meio de entrevistas semiestruturadas (MCGRATH; LILJEDAHL, 2019; BÖRSTLER et al., 2023), no período de outubro e novembro de 2023. Resumidamente, foram seguidas as seguintes diretrizes: i) elaboração e testagem do roteiro da entrevista, com um profissional experiente, com mais de 20 anos de atuação em desenvolvimento de software; ii) contato, seleção dos potenciais participantes e agendamento das entrevistas; iii) realização das entrevistas por vídeo[18], com duração média de 90 minutos, incluindo uma apresentação geral do *DR-Tools Code Health* para os participantes; iv) transcrição das entrevistas e análise dos dados.

A seguir, são apresentados os principais comentários dos participantes.

O feedback fornecido pelo **Dev#1** foi:

> *"Acredito que o* DR-Tools Code Health *tem grande potencial como ferramenta para apoiar a equipe de desenvolvimento em todas as etapas de um projeto. No geral, fiquei surpreso com a facilidade de execução em diferentes projetos, já que muitas ferramentas semelhantes com as quais trabalhei exigiam várias configurações para se adaptar a cada aplicação. Para incentivar seu uso, é necessário permitir alguma personalização da análise, por*

[18]Foi utilizado a ferramenta *Google Meet*

exemplo: executar apenas em determinados arquivos (útil para executar em Pull Requests), ignorar certos 'smells' dependendo do projeto, ou até mesmo permitir que o desenvolvedor marque partes do código que podem não ser relevantes para a análise (por exemplo, queremos implementar a ferramenta em um projeto, mas há uma parte legada que não queremos alterar e não queremos que ela gere ruído)."

O feedback do **Dev#2** é empolgante:

"Após vários anos estudando e testando ferramentas que incentivam uma melhor qualidade de código por parte dos engenheiros de software, me deparei com algo realmente diferente por meio da proposta do DR-Tools Code Health*. A metáfora de um código ficando doente e, acima de tudo, dos sintomas permitindo a identificação de padrões é excelente, assim como, por sua vez, a compreensão da melhoria ou resolução desses problemas! O modo interativo do* DR-Tools Code Health *me permitiu compreender os principais 'sintomas' e até mesmo algumas 'doenças' bem conhecidas por aqueles que estudam/praticam 'code smells' em apenas alguns minutos e com profundidade. Ele traz para mim algo que ferramentas estáticas de análise de código e visualização frequentemente carecem: feedback rápido e a possibilidade de correlações entre sintomas de um código potencialmente vulnerável. Além disso, é adaptável às necessidades de nossas equipes/produtos. Indo além, acredito que integrar o DR-Tools Code Health com ferramentas como o ELK stack (Elasticsearch, Logstash e Kibana) poderia fornecer ainda mais clareza e melhor compreensão dos sintomas e suas relações."*

Por fim, o terceiro feedback, fornecido pelo **Dev#3**:

"Conduzimos um teste do DR-Tools Code Health *em nosso produto, que atualmente adota uma arquitetura baseada em microsserviços. Temos 23 microsserviços em produção. Inicialmente, começamos com um monólito e validamos a ideia antes de migrar o produto para microsserviços, mas ainda temos alguns serviços legados (o primeiro criado) em Java. Durante a análise de métricas e 'code smells' fornecidos pelo* DR-Tools Code Health*, alcançamos uma compreensão clara da evolução de nosso código. Especificamente*

> *em um código legado migrando para os microsserviços, identificamos problemas significativos como classes muito grandes, alta complexidade e dependências circulares. Nos serviços criados inicialmente, observamos uma melhoria considerável se comparada ao código legado. Ao adotar a arquitetura hexagonal, conseguimos eliminar dependências circulares. No entanto, identificamos alguns desafios, como classes grandes e um domínio muito extenso, ajudando-nos a refletir se tudo presente nesses serviços realmente pertence ao domínio ou se seria necessário dividi-los em serviços distintos. E com nosso projeto mais recente, tivemos um grande progresso. Identificamos muito poucos 'code smells', o que nos dá confiança de que os serviços podem ser facilmente evoluídos e mantidos. Acreditamos que o* DR-Tools Code Health *é uma excelente ferramenta para nos ajudar a escolher qual parte do código deve ser evoluída, mantida ou até mesmo extraída para outros serviços. Em resumo, as informações detalhadas e significativas fornecidas pela ferramenta nos ajudam a fazer análises mais precisas."*

Os *feedbacks* relatados anteriormente destacam aspectos positivos e recomendações de melhoria. A ferramenta foi elogiada por sua simplicidade, facilidade de uso e *feedback* rápido (fornecido pelo modo interativo), com um grande potencial para ajudar os desenvolvedores em seu trabalho diário, auxiliando na tomada de decisões para a evolução e manutenção do código. Além disso, o *feedback* inclui recomendações de opções de personalização mais orientadas para usabilidade, como analisar alguns arquivos específicos, ignorar certos *code smells* ou marcar como irrelevante um código sendo analisado.

A integração com ferramentas complementares, como o ELK[19] também foi sugerida para análises gráficas e históricas. No geral, o *feedback* positivo é um estímulo para melhorias, tornando o *DR-Tools Code Health* uma ferramenta alternativa no suporte às decisões dos desenvolvedores, garantindo a qualidade do código e aumentando a eficiência.

[19]Mais informações em <https://www.elastic.co/pt/elastic-stack>

6 CONSIDERAÇÕES FINAIS

Aplicações reais estão em constante mudança para satisfazer as exigências do mercado e as expectativas de seus usuários. De outra maneira, não seria possível adaptar o sistema às tendências, legislações e inovações tecnológicas emergentes. Para tanto, os times de desenvolvimento de software precisam estar constantemente preocupados com a melhoria contínua e a qualidade do produto e serviços oferecidos (CANFORA; PENTA; CERULO, 2011). Contudo, existem ainda muitos desafios envolvidos na manutenção e evolução de software, entre eles está a necessidade de lidar com códigos complexos com diversos *smells*, que diminuem a produtividade, aumentam o risco e elevam os custos do projeto.

Diversas técnicas de detecção foram propostas para a identificação destes *smells* a partir do código. Essas técnicas retornam os elementos de código identificados que, por sua vez, requerem muito esforço para sua remoção, usando refatorações (KAUR et al., 2021) ou outras técnicas apropriadas (MARTIN, 2008; SEEMANN, 2021; MARTIN, 2021; BECK, 2023). Além disso, todos estes *smells* não são igualmente importantes ou relevantes para os objetivos do software ou mesmo pensando em sua qualidade (LENARDUZZI et al., 2021). Conforme Bavota et al. (2015), os desenvolvedores não resolvem os *smells* com frequência, e uma das possíveis causas estejam ligadas a falta de um método de priorização e filtragem que leve em consideração aspectos de qualidade e a própria percepção dos desenvolvedores naquele momento do projeto.

Assim, esta tese teve como objetivo explorar esta lacuna. Priorizar os *smells* é uma prática fundamental no desenvolvimento de software, pois ajuda os desenvolvedores a focarem seus esforços nas áreas mais críticas e impactantes do projeto. Portanto, o time pode concentrar seus recursos e tempo em resolver os problemas que têm o maior impacto na qualidade, manutenibilidade e desempenho do software. Também, esta prática ajuda a melhorar a qualidade geral do código, reduzir o acúmulo de dívida técnica e aumentar a robustez e confiabilidade do sistema.

De forma geral, o *DR-Tools Code Health* ajuda os desenvolvedores na priorização dos elementos de código mais problemáticos. A abordagem proposta demonstra a importância da priorização, onde os desenvolvedores podem tomar decisões ao fazer melhorias ou refatorações no código. Mais importante do que fazer refatorações, é fazer refatorações de forma consciente, nas partes identificadas e avaliadas como prioritárias. Ao identificar e resolver os *smells* mais críticos e prejudiciais primeiro, os desenvolvedores podem

fazer progressos significativos na limpeza e refatoração do código, tornando-o mais fácil de entender, manter e evoluir ao longo do tempo. Portanto, esta pesquisa investigou a importância da priorização de *smells* e propôs uma abordagem prática para melhorar a detecção, classificação e *ranking* dos *smells* presentes nos elementos problemáticos de um projeto de software.

Visando responder os questionamentos levantados na Motivação (Seção 1.1), o *DR-Tools Code Health* detecta *smells* em diferentes elementos de código, permitindo essa diferenciação de severidade e diversidade entre elementos de código de mesma granularidade. O *DR-Tools Code Health* também associa os *smells* a atributos de qualidade, possibilitando a filtragem e análise dos atributos mais afetados. O *DR-Tools Code Health* considera, na sua estrutura, análises específicas nas granularidades de métodos, classes e pacotes e suas respectivas composições. Com isso, é possível avaliar o acúmulo dos *smells* e considerá-los na priorização. Ainda, através de customizações permitidas pela abordagem, pode-se ajustar pesos e parâmetros utilizados na detecção e priorização, melhorando a eficácia dos resultados.

Ao longo deste trabalho, foram apresentadas pesquisas que fundamentam o tema (Capítulo 2), trabalhos relacionados a abordagens de detecção de problemas no código, ferramentas, priorização e filtragem (Capítulo 3). Também, foi apresentado em detalhes a estruturação e características do modelo de priorização (Capítulo 4), método de uso e aplicabilidade. Também, foi desenvolvida uma ferramenta *open-source* homônima para dar suporte ao método. Finalmente, avaliou-se o *DR-Tools Code Health* com experimentos envolvendo a indústria e projetos *open-source* (Capítulo 5).

No **experimento voltado a percepção dos desenvolvedores** (Seção 5.1), obteve-se os seguintes resultados para as RQs:

***RQ1* - A abordagem de priorização proposta ajuda os desenvolvedores a identificar os elementos de código (classes, métodos) mais problemáticos?** O *DR-Tools Code Health* auxilia os desenvolvedores na identificação dos elementos de código mais problemáticos. Nos métodos, obteve-se uma *precision* médio de 0.90 e mediana de 1.00 com coeficiente *kappa* médio de 0.82 (mediana=1.00), indicando uma concordância quase perfeita. Quando analisada as classes, a *precision* médio ficou de 0.73, mediana de 0.80, além do *kappa* médio de 0.65 (mediana=0.74), indicando uma concordância substancial. Na composição de classes + métodos, obteve-se uma *precision* médio de 0.80, mediana de 0.93, com coeficiente *kappa* médio de 0.73 (mediana=0.81), indicando uma concordância substancial. Essa composição se mostrou mais relevante para os participantes do

que simplesmente analisar os *smells* na granularidade de classe, o que pode indicar que os elementos de código merecem mais atenção ao observar os *smells* de métodos e classes juntos.

***RQ2* - Ao confrontar os resultados analisados pela abordagem de priorização proposta, muda a percepção dos desenvolvedores sobre os elementos de código mais problemáticos?** O *DR-Tools Code Health* teve um impacto significativo na percepção dos participantes em relação aos elementos de código analisados nos projetos de software. A maioria dos participantes (87%) relatou uma mudança em sua percepção, especialmente na granularidade de métodos. No entanto, uma parcela dos participantes (7-33%) não percebeu mudanças ou não observou diferenças, destacando a necessidade de uma investigação mais detalhada sobre a eficácia da ferramenta. Ainda, houve casos em que os participantes não registraram comentários extras e indicaram não ter mudado sua percepção, mesmo confirmando os resultados apresentados pelo *DR-Tools Code Health*.

No **experimento relacionado a ação de priorização e impacto dos *smells* em atributos de qualidade durante a evolução de software** (Seção 5.2), os resultados obtidos foram:

***RQ3* - Os desenvolvedores fazem algum tipo de priorização ao remover os *smells* durante a manutenção e evolução de software?** Na maioria dos projetos analisados, observou-se um aumento gradual dos *smells* ou um comportamento constante, que reforça os resultados de Tufano et al. (2017). O *smell churn rate*, desenvolvido especialmente para este experimento, pode ser uma medida que ajuda a entender as ações de priorização. Quanto maior for essa medida, mais se obteve redução de *smells* entre versões. Apenas o projeto *JetUML* apresentou queda de *smells* entre as versões analisadas, com *smell churn rate* médio de 39.00. Não foi utilizado um método de priorização, apenas uma estratégia de lançamento de versão e melhorias oportunistas, ou seja, que podem ser feitas em elementos de código que estão sendo modificados.

***RQ4* - Como os *smells* têm impactado a qualidade durante a evolução de software?** Os impactos nos atributos de qualidade nas diferentes granularidades e versões mantiveram um padrão similar. Possivelmente este resultado é consequência da associação de atributos de qualidade com *smell* detectados. A exceção foi o *JetUML* por apresentar maior variação no impacto dos atributos de qualidade. O atributo de qualidade mais impactado pelos *smells* na granularidade de *namespace* foi *modularity*. Na granularidade de *type*, os atributos mais impactados foram *modularity* e *maintainability*. Já

granularidade de *method*, o atributo de qualidade mais impactado foi *maintainability*.

O *DR-Tools Code Health*, através de um indicador agregado de diferentes critérios, visa identificar códigos que são difíceis de manter e correm mais riscos de defeitos e mudanças, além de detectar códigos com acúmulo de complexidade. A seleção dos critérios utilizados foram baseados em estudos e pesquisas realizadas. Porém, é possível adaptar e incluir outros critérios e formas de computação da priorização para futuras avaliações. O objetivo da abordagem apresentada não é prever os problemas que sofrerão refatorações, mas apoiar os desenvolvedores na priorização e filtragem dos problemas para tomada de decisão em conjunto, com base no código. A abordagem foi projetada para ser usada no dia a dia do desenvolvimento, durante atividades de programação, revisões de código e até mesmo apoiando o processo de *build* do sistema.

6.1 Trabalhos Futuros

Embora esta pesquisa tenha avançado significativamente no desenvolvimento de um modelo de priorização de *smells* para apoiar a manutenção e evolução de software, existem diversas direções promissoras para trabalhos futuros que podem aprimorar e expandir ainda mais essa área de estudo. São listadas algumas delas:

- **Expansão dos Estudos sobre *Smells*:** A exploração e identificação de novos tipos de *smells* que possam impactar a qualidade e a manutenibilidade do software pode ser um caminho promissor. Além dos *smells* tradicionais já estudados, investigar e categorizar novas classes de *smells* emergentes, como *security smells* (PONCE et al., 2022; OPDEBEECK; ZEROUALI; ROOVER, 2023) e *test smells* (PERUMA et al., 2020), pode enriquecer significativamente o conjunto de critérios considerados na análise e priorização do código. Também, a avaliação das co-ocorrências dos *smells* (MARTINS et al., 2021), tanto dentro de módulos individuais quanto em nível de sistema, pode oferecer *insights* valiosos sobre padrões de degradação de qualidade e áreas críticas que necessitam de atenção prioritária durante o processo de manutenção e evolução do software. Essa análise mais abrangente e contextualizada dos *smells* pode aprimorar a eficácia das estratégias de priorização e refatorações, contribuindo para a construção de sistemas mais robustos e sustentáveis.
- **Customizações no Modelo de Priorização:** Uma área de pesquisa promissora envolve a investigação de customizações adicionais no modelo de priorização pro-

posto (VERMA; KUMAR; VERMA, 2023). Por exemplo, explorar a inclusão de novos critérios de avaliação ou ajustar os pesos dos critérios existentes para refletir melhor as necessidades específicas de diferentes contextos dos projetos de desenvolvimento de software.

- **Geração de *Datasets* para Implementação usando *Machine Learning*:** A geração de *datasets*, incluindo *multi-label code smell* (GUGGULOTHU; MOIZ, 2020; YADAV; RAO; MISHRA, 2024) pode ser facilitada pelo *DR-Tools Code Health*. Esses *datasets* podem ser utilizados para treinar modelos de ML para identificar e priorizar *smells* automaticamente, aumentando a eficiência do processo de manutenção e evolução de software.
- **Recomendações de Refatorações usando IA Generativa:** Investigar a aplicação de técnicas de Inteligência Artificial generativa para a geração automática de recomendações de refatorações e heurísticas de limpeza (ALOMAR et al., 2024) para os *smells* identificados. Isso poderia envolver o desenvolvimento de modelos de IA que não apenas identificam os problemas (LIU et al., 2024), mas também sugerem soluções específicas para abordá-los, reduzindo ainda mais a carga cognitiva dos desenvolvedores.
- **Incorporação de outras Ferramentas ao *DR-Tools Suite:*** Expandir o conjunto de ferramentas incorporadas ao *DR-Tools Suite* para oferecer suporte a uma gama mais ampla de atividades de desenvolvimento e manutenção de software. Isto pode incluir analisadores para outras linguagens de programação, *plugins* para integração contínua, integrações com IDEs, e até mesmo integração com ferramentas de *chat* (por exemplo, *Slack*), para facilitar a colaboração e comunicação entre os membros do time de desenvolvimento de software.

Essas são apenas algumas das muitas direções possíveis para futuras pesquisas nesta área. Espera-se que essas sugestões inspirem e orientem pesquisadores e profissionais interessados em continuar a avançar no campo da manutenção e evolução de software através da priorização de *smells*.

6.2 Trabalhos de Conclusão Derivados

A partir do projeto *DR-Tools Suite*, alguns Trabalhos de Conclusão de Curso (TCCs) foram desenvolvidos. A seguir, é apresentado, até o momento, os TCCs deri-

vados deste projeto.

- **Lucas da Silva Heim:** *Integração Contínua para Verificação da Qualidade utilizando a Ferramenta DR-Tools*. TCC apresentado no curso de Ciência da Computação da Universidade do Vale do Rio dos Sinos (UNISINOS), em 2023/1. Este trabalho foi orientado pelo autor desta tese.
- **Glauber de Souza Rosa:** *Integration of ChatGPT to Software Engineering Smell Detection Tool*. TCC apresentado no curso de Engenharia de Computação da Universidade Federal do Rio Grande do Sul (UFRGS), em 2023/2. Este trabalho foi orientado pelo Prof. Marcelo Pimenta e co-orientado pelo autor desta tese.
- **Fábio Petkowicz:** *Análise Evolutiva de* Code Smells *em Sistemas de Software: Um Estudo de Caso*. TCC do curso de Engenharia de Computação da Universidade Federal do Rio Grande do Sul (UFRGS), previsto para conclusão em 2024/1. Este trabalho foi orientado pelo Prof. Marcelo Pimenta e co-orientado pelo autor desta tese.
- **Klaus Kretzer Steinmeier:** *Protótipo de Plataforma para Apoiar a Manutenção e Compreensão de Sistemas de Software*. TCC do curso de Ciência da Computação da Universidade do Vale do Rio dos Sinos (UNISINOS), previsto para conclusão em 2024/2. Este trabalho foi orientado pelo autor desta tese.

6.3 Publicações e Apresentações Realizadas

Ao longo do doutorado, alguns trabalhos relacionados a tese foram submetidos e aprovados, tanto em periódico especializado da área como em conferência internacional. Os referidos trabalhos são:

- **Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations**, publicado no *Journal of Systems and Software (JSS) - 2020*, escrito com colaboração de Fábio Petrillo (UQAC - Canadá), Marcelo Pimenta (UFRGS) e Yann Gaël Guéhéneuc (Concordia University - Canadá);
- **Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations**, aceito para apresentação no *ICSME 2020 - International Conference on Software Maintenance and Evolution, Journal First Track*, escrito com colaboração de Fábio Petrillo (UQAC - Canadá), Marcelo Pimenta (UFRGS) e Yann Gaël Guéhéneuc (Concordia University - Canadá);

- **DR-Tools: a suite of lightweight open-source tools to measure and visualize Java source code**, aceito para publicação no *ICSME 2020 - International Conference on Software Maintenance and Evolution, Tool Demo Track*, escrito com colaboração de Fábio Petrillo (UQAC - Canadá) e Marcelo Pimenta (UFRGS).

Além destes trabalhos acadêmicos, o autor também realizou algumas apresentações de assuntos relacionados ao tema da tese em conferências nacionais da indústria. São elas:

- **eXtreme Programming e a Academia**, palestra realizada na *XP Conf BR*, em 2018;
- **Voltando para as raízes do desenvolvimento ágil**, palestra realizada no *DevOps-Day - edição Porto Alegre*, em 2019;
- **Melhorando o design de código através de Metáforas**, palestra realizada nas trilhas de *Design de Código e eXtreme Programming no The Developers Conference - TDC*, (POA, 2019) e (SP, 2020);
- **Análise de Código: precisamos falar disso!**, palestra realizada na trilha de *Design de Código e eXtreme Programming no The Developers Conference – TDC*, (POA, 2020);
- **Analisando a qualidade do código com DR-Tools Suite**, palestra realizada no *ConFLOSS - Conferência de Free/Libre e Open Source Software - 2a Edição*, (online, 2021);
- **Painel sobre Design de Código e Dívida Técnica**, painelista convidado para a trilha de *Design de Código e eXtreme Programming no The Developers Conference – TDC*, (online, 2021);
- **Análise de Código: precisamos falar disso!**, palestra realizada no *Agile Brazil*, (online, 2021);
- **A Entrega de Valor encontra Equipas de Tecnologia**, palestra realizada no *Agile Days - Portugal*, (online, 2022).

REFERÊNCIAS

ABBES, M. et al. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: **2011 15th European Conference on Software Maintenance and Reengineering**. [S.l.: s.n.], 2011. p. 181–190. ISSN 1534-5351.

ABUHASSAN, A.; ALSHAYEB, M.; GHOUTI, L. Software smell detection techniques: A systematic literature review. **Journal of Software: Evolution and Process**, Wiley Online Library, p. e2320, 2020.

AGHAJANI, E. et al. Software documentation: the practitioners' perspective. In: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering**. [S.l.: s.n.], 2020. p. 590–601.

AL-SHAABY, A.; ALJAMAAN, H.; ALSHAYEB, M. Bad smell detection using machine learning techniques: A systematic literature review. **Arabian Journal for Science and Engineering**, Springer, p. 1–29, 2020.

ALAVINIA, S. M. et al. Prioritization of rehabilitation domains for establishing spinal cord injury high performance indicators using a modification of the hanlon method: Sci-high project. **The journal of spinal cord medicine**, Taylor & Francis, v. 42, n. sup1, p. 43–50, 2019.

ALFAYEZ, R. et al. A systematic literature review of technical debt prioritization. In: **Proceedings of the 3rd International Conference on Technical Debt**. [S.l.: s.n.], 2020. p. 1–10.

ALKHARABSHEH, K. et al. Software design smell detection: a systematic mapping study. **Software Quality Journal**, Springer, v. 27, n. 3, p. 1069–1148, 2019.

ALLMAN, E. Managing technical debt. **Communications of the ACM**, ACM New York, NY, USA, v. 55, n. 5, p. 50–55, 2012.

ALOMAR, E.; MKAOUER, M. W.; OUNI, A. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In: IEEE. **2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)**. [S.l.], 2019. p. 51–58.

ALOMAR, E. A. et al. How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. **Expert Systems with Applications**, v. 167, p. 114176, 2021. ISSN 0957-4174. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S095741742030912X>.

ALOMAR, E. A. et al. How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations. **arXiv preprint arXiv:2402.06013**, 2024.

ALVAREZ, M. et al. Priority setting in health research in cuba, 2010. **MEDICC review**, v. 12, n. 4, p. 15–19, 2010.

ALVES, N. S. et al. Identification and management of technical debt: A systematic mapping study. **Information and Software Technology**, v. 70, p. 100–121, feb 2016. ISSN 09505849.

AMANATIDIS, T. et al. The developer's dilemma: Factors affecting the decision to repay code debt. In: **Proceedings of the 2018 International Conference on Technical Debt**. New York, NY, USA: Association for Computing Machinery, 2018. (TechDebt '18), p. 62–66. ISBN 9781450357135. Available from Internet: <https://doi.org/10.1145/3194164.3194174>.

AMORIM, L. et al. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In: IEEE. **2015 IEEE 26th international symposium on software reliability engineering (ISSRE)**. [S.l.], 2015. p. 261–269.

AMPATZOGLOU, A. et al. Exploring the relation between technical debt principal and interest: An empirical approach. **Information and Software Technology**, Elsevier, v. 128, p. 106391, 2020.

ANICHE, M. et al. Code smells for model-view-controller architectures. **Empirical Software Engineering**, Springer, v. 23, n. 4, p. 2121–2157, 2018.

ARCOVERDE, R. et al. Prioritization of code anomalies based on architecture sensitiveness. In: **2013 27th Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2013. p. 69–78.

AVERSANO, L.; CARPENITO, U.; IAMMARINO, M. An empirical study on the evolution of design smells. **Information**, Multidisciplinary Digital Publishing Institute, v. 11, n. 7, p. 348, 2020.

AZIZI, H. et al. Effective programs and strategies on suicide prevention: combination of review of systematic reviews with expert opinions. Research Square, 2020.

BAABAD, A. et al. Software architecture degradation in open source software: A systematic literature review. **IEEE Access**, IEEE, v. 8, p. 173681–173709, 2020.

BAEZA-YATES et al. **Modern information retrieval**. [S.l.]: ACM press New York, 1999.

Bakar, N. S. A. A.; Boughton, C. V. Validation of measurement tools to extract metrics from open source projects. In: **2012 IEEE Conference on Open Systems**. [S.l.: s.n.], 2012. p. 1–6. ISSN null.

BAKOTA, T. et al. A probabilistic software quality model. In: IEEE. **2011 27th IEEE International Conference on Software Maintenance (ICSM)**. [S.l.], 2011. p. 243–252.

BALTES, S.; RALPH, P. Sampling in software engineering research: A critical review and guidelines. **Empirical Software Engineering**, Springer, v. 27, n. 4, p. 94, 2022.

BALTUSSEN, R.; NIESSEN, L. Priority setting of health interventions: the need for multi-criteria decision analysis. **Cost effectiveness and resource allocation**, BioMed Central, v. 4, n. 1, p. 1–9, 2006.

BANDI, A.; WILLIAMS, B. J.; ALLEN, E. B. Empirical evidence of code decay: A systematic mapping study. In: IEEE. **2013 20th Working Conference on Reverse Engineering (WCRE)**. [S.l.], 2013. p. 341–350.

BANSIYA, J.; DAVIS, C. G. A hierarchical model for object-oriented design quality assessment. **IEEE Transactions on Software Engineering**, v. 28, n. 1, p. 4–17, Jan 2002. ISSN 0098-5589.

BAVOTA, G. et al. An experimental investigation on the innate relationship between quality and refactoring. **Journal of Systems and Software**, Elsevier, v. 107, p. 1–14, 2015.

BAVOTA, G. et al. Methodbook: Recommending move method refactorings via relational topic models. **IEEE Transactions on Software Engineering**, IEEE, v. 40, n. 7, p. 671–694, 2013.

BECK, K. **Tidy First?: A Personal Exercise in Empirical Software Design**. [S.l.]: O'Reilly, 2023.

BECK, K.; ANDRES, C. **Extreme Programming Explained: Embrace Change (2nd Edition)**. [S.l.]: Addison-Wesley, 2004.

BEHUTIYE, W. N. et al. Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. **Information and Software Technology**, v. 82, p. 139–158, feb 2017. ISSN 09505849.

BESKER, T.; MARTINI, A.; BOSCH, J. Managing architectural technical debt: A unified model and systematic literature review. **Journal of Systems and Software**, Elsevier Inc., v. 135, p. 1–16, jan 2018. ISSN 01641212.

BESSGHAIER, N.; OUNI, A.; MKAOUER, M. W. A longitudinal exploratory study on code smells in server side web applications. **Software Quality Journal**, Springer, v. 29, p. 901–941, 2021.

BIGONHA, M. A. et al. The usefulness of software metric thresholds for detection of bad smells and fault prediction. **Information and Software Technology**, v. 115, p. 79 – 92, 2019. ISSN 0950-5849.

BÖRSTLER, J. et al. Developers talking about code quality. **Empirical Software Engineering**, Springer, v. 28, n. 6, p. 128, 2023.

BOUWERS, E.; DEURSEN, A. van; VISSER, J. Evaluating usefulness of software metrics: An industrial experience report. In: **2013 35th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2013. p. 921–930. ISSN 1558-1225.

BRAUN, V.; CLARKE, V. **Thematic analysis**. [S.l.]: American Psychological Association, 2012.

BRIAND, L. C.; DALY, J. W.; WÜST, J. **A Unified Framework for Cohesion Measurement in Object-Oriented Systems**. 1997.

BROOKS, F. P. No silver bullet: Essence and accidents of software engeneering. **IEEE Computer**, v. 20, n. 4, p. 10–19, April 1987.

BROOKS, F. P. **Mythical Man-Month, Anniversary Edition, The: Essays On Software Engineering**. [S.l.]: Addison-Wesley, 1995.

BROWN, W. H. et al. **AntiPatterns: Refactoring software, architectures, and projects in crisis**. Canada: John Wiley and Sons, Inc, 1998. ISBN 0?471?19713?0.

CAIRO, A. S.; CARNEIRO, G. D. F.; MONTEIRO, M. P. The Impact of Code Smells on Software Bugs: a Systematic Literature Review. **Journal of Information Science, Technology and Engineering**, v. 9, n. October, p. 1–21, 2018.

CANFORA, G.; PENTA, M. D.; CERULO, L. Achievements and challenges in software reverse engineering. **Communications of the ACM**, ACM, v. 54, n. 4, p. 142–151, 2011.

CARDOSO, B.; FIGUEIREDO, E. Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In: **Proceedings of the Annual Conference on Brazilian Symposium on Information Systems: Information Systems: A Computer Socio-Technical Perspective - Volume 1**. Porto Alegre, Brazil, Brazil: Brazilian Computer Society, 2015. (SBSI 2015), p. 46:347–46:354.

CASSELL, K. et al. Visualizing class refactoring via clustering. **Technical Report ECSTR10-17**, 2010.

CATOLINO, G. A blessing in disguise? assessing the relationship between code smells and sustainability. In: IEEE. **2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.], 2020. p. 779–780.

CEDRIM, D. et al. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. [S.l.: s.n.], 2017. p. 465–475.

CHAMBERLIN, D. D.; GILBERT, A. M.; YOST, R. A. A history of system r and sql/data system. In: **Proceedings of the seventh international conference on Very Large Data Bases-Volume 7**. [S.l.: s.n.], 1981. p. 456–464.

CHARALAMPIDOU, S. **Managing technical debt through software metrics, refactoring and traceability**. Thesis (PhD) — University of Groningen, 2019.

CHATZIGEORGIOU, A.; MANAKOS, A. Investigating the evolution of bad smells in object-oriented code. In: IEEE. **2010 Seventh International Conference on the Quality of Information and Communications Technology**. [S.l.], 2010. p. 106–115.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object-oriented design. **IEEE Computer Society Trans. Software Engineering**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 20, n. 6, p. 476–493, june 1994. ISSN 0098-5589.

CHOI, B. C. et al. The pan american health organization-adapted hanlon method for prioritization of health programs. **Revista Panamericana de Salud Pública**, Pan American Health Organization, v. 43, 2019.

CHOUDHARY, A.; SINGH, P. Minimizing refactoring effort through prioritization of classes based on historical, architectural and code smell information. In: **QuASoQ/TDA@ APSEC**. [S.l.: s.n.], 2016. p. 76–79.

CODABUX, Z.; WILLIAMS, B. J. Technical debt prioritization using predictive analytics. In: IEEE. **2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)**. [S.l.], 2016. p. 704–706.

CODABUX, Z. et al. An empirical assessment of technical debt practices in industry. **Journal of software: Evolution and Process**, Wiley Online Library, v. 29, n. 10, p. e1894, 2017.

Code Ahoy. **Tech Debt Developer Survey Results 2020 - Impact on Retention**. 2020. Available from Internet: <https://codeahoy.com/2020/02/17/technical-debt-survey/>.

COLEMAN, C.; GRISWOLD, W. G.; MITCHELL, N. Do cloud developers prefer clis or web consoles? clis mostly, though it varies by task. **arXiv preprint arXiv:2209.07365**, 2022.

County Health Rankings and Roadmaps. **Guide to Prioritization Techniques**. 2013. Available from Internet: <https://www.countyhealthrankings.org/resources/guide-to-prioritization-techniques>.

CUNNINGHAM, W. The wycash portfolio management system. **ACM SIGPLAN OOPS Messenger**, ACM New York, NY, USA, v. 4, n. 2, p. 29–30, 1992.

CUNNINGHAM, W. **Technical Debt**. 2014. Available from Internet: <http://c2.com/cgi/wiki?TechnicalDebt>.

CURTIS, B.; MARTIN, R. A.; DOUZIECH, P.-E. Measuring the structural quality of software systems. **Computer**, IEEE, v. 55, n. 3, p. 87–90, 2022.

CURTIS, B.; SAPPIDI, J.; SZYNKARSKI, A. Estimating the size, cost, and types of technical debt. In: IEEE. **2012 Third International Workshop on Managing Technical Debt (MTD)**. [S.l.], 2012. p. 49–53.

DAVILA, N.; NUNES, I. A systematic literature review and taxonomy of modern code review. **Journal of Systems and Software**, Elsevier, v. 177, p. 110951, 2021.

DEANE, H. C. et al. Predict prioritisation study: establishing the research priorities of paediatric emergency medicine physicians in australia and new zealand. **Emergency medicine journal**, BMJ Publishing Group Ltd and the British Association for Accident . . . , v. 35, n. 1, p. 39–45, 2018.

DETOFENO, T.; MALUCELLI, A.; REINEHR, S. Technical debt guild: managing technical debt from code up to build. **Journal of Software Engineering Research and Development**, p. 1–1, 2023.

DHAMBRI, K.; SAHRAOUI, H.; POULIN, P. Visual detection of design anomalies. In: IEEE. **2008 12th European Conference on Software Maintenance and Reengineering**. [S.l.], 2008. p. 279–283.

DIEHL, S. **Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software**. [S.l.]: Springer, 2007.

DIGKAS, G. et al. Can clean new code reduce technical debt density. **IEEE Transactions on Software Engineering**, IEEE, 2020.

DIGKAS, G. et al. How do developers fix issues and pay back technical debt in the apache ecosystem? In: IEEE. **2018 IEEE 25th International Conference on software analysis, evolution and reengineering (SANER)**. [S.l.], 2018. p. 153–163.

EISENBERG, R. J. A threshold based approach to technical debt. **ACM SIGSOFT Software Engineering Notes**, ACM New York, NY, USA, v. 37, n. 2, p. 1–6, 2012.

ELSSAMADISY, A. **Patterns of Agile Practice Adoption: The Technical Cluster**. [S.l.]: C4Media, 2007.

EMDEN, E. V.; MOONEN, L. Java quality assurance by detecting code smells. In: **Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)**. Washington, DC, USA: IEEE Computer Society, 2002. p. 97–. Available from Internet: <http://portal.acm.org/citation.cfm?id=882506.885134>.

ERNST, N. A. et al. Measure it? manage it? ignore it? software practitioners and technical debt. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering**. [S.l.: s.n.], 2015. p. 50–60.

FARAGÓ, C.; HEGEDŰS, P.; FERENC, R. Cumulative code churn: Impact on maintainability. In: IEEE. **2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.], 2015. p. 141–150.

FARAGÓ, C.; HEGEDŰS, P.; FERENC, R. Cumulative code churn: Impact on maintainability. In: IEEE. **2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.], 2015. p. 141–150.

FEATHERS, M. **Working Effectively with Legacy Code**. [S.l.]: Prentice Hall, 2004.

FENSKE, W. et al. When code smells twice as much: Metric-based detection of variability-aware code smells. In: **Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on**. [S.l.: s.n.], 2015. p. 171–180.

FERNANDES, E. et al. A review-based comparative study of bad smell detection tools. In: **Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2016. (EASE '16), p. 1. ISBN 9781450336918. Available from Internet: <https://doi.org/10.1145/2915970.2915984>.

FILÓ, T. G. S.; BIGONHA, M. A. da S. A catalogue of thresholds for object-oriented software metrics. In: **2015 (SOFTENG)**. [S.l.: s.n.], 2015. p. 48–55.

FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: An experimental assessment. **Journal of Object Technology**, 2012.

FONTANA, F. A. et al. Antipattern and code smell false positives: Preliminary conceptualization and classification. In: IEEE. **2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)**. [S.l.], 2016. v. 1, p. 609–613.

FONTANA, F. A.; FERME, V.; ZANONI, M. Poster: Filtering code smells detection results. In: **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**. [S.l.: s.n.], 2015. v. 2, p. 803–804.

FONTANA, F. A.; FERME, V.; ZANONI, M. Towards assessing software architecture quality by exploiting code smell relations. In: IEEE. **2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics**. [S.l.], 2015. p. 1–7.

FONTANA, F. A. et al. Automatic metric thresholds derivation for code smell detection. In: IEEE. **2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics**. [S.l.], 2015. p. 44–53.

FONTANA, F. A. et al. Towards a prioritization of code debt: A code smell intensity index. In: IEEE. **2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)**. [S.l.], 2015. p. 16–24.

FONTANA, F. A. et al. Comparing and experimenting machine learning techniques for code smell detection. **Empirical Software Engineering**, Springer, v. 21, n. 3, p. 1143–1191, 2016.

FONTANA, F. A. et al. Arcan: A tool for architectural smells detection. In: IEEE. **2017 IEEE International Conference on Software Architecture Workshops (ICSAW)**. [S.l.], 2017. p. 282–285.

FONTANA, F. A.; ZANONI, M. Code smell severity classification using machine learning techniques. **Knowledge-Based Systems**, v. 128, p. 43 – 58, 2017. ISSN 0950-7051. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0950705117301880>.

FOWLER, M. **Domain-specific languages**. [S.l.]: Pearson Education, 2010.

FOWLER, M.; BECK, K. **Refactoring: Improving the Design of Existing Code - Second Edition**. [S.l.]: Pearson, 2019.

FOWLER, M. et al. **Refactoring: Improving the Design of Existing Code**. [S.l.]: Addison-Wesley, 1999.

FU, S.; SHEN, B. Code bad smell detection through evolutionary data mining. In: IEEE. **2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.], 2015. p. 1–9.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.]: Addison-Wesley, 1994.

GAROUSI, V. et al. Closing the gap between software engineering education and industrial needs. **IEEE Software**, p. 1–1, 2019. ISSN 0740-7459.

GAROUSI, V.; MANTYLA, M. V. A systematic literature review of literature reviews in software testing. **Information and Software Technology**, v. 80, p. 195 – 216, 2016. ISSN 0950-5849.

GAROUSI, V. et al. Characterizing industry-academia collaborations in software engineering: evidence from 101 projects. **Empirical Software Engineering**, Apr 2019. ISSN 1573-7616. Available from Internet: <https://doi.org/10.1007/s10664-019-09711-y>.

GAROUSI, V.; SHEPHERD, D. C.; HERKILOGLU, K. Successful engagement of practitioners and software engineering researchers: Evidence from 26 international industry-academia collaborative projects. **IEEE Software**, p. 1–1, 2019. ISSN 0740-7459.

GEARY, J. **I Is an Other: The Secret Life of Metaphor and How it Shapes the Way We See the World**. [S.l.]: Harper Perennial, 2012.

GHEZZI, G.; GALL, H. C. A framework for semi-automated software evolution analysis composition. **Automated Software Engineering**, Springer, p. 1–34, 2013.

GOUES, C. L. et al. Bridging the gap: From research to practical advice. **IEEE Software**, v. 35, n. 5, p. 50–57, Sep. 2018. ISSN 0740-7459.

GRADISNIK, M.; HERICKO, M. Impact of Code Smells on the Rate of Defects in Software: A Literature Review. In: **SQAMIA 2018: 7th Workshop of Software Quality**. [S.l.: s.n.], 2018. (SQAMIA 2018, October), p. 1–21.

GRAY, D.; BROWN, S.; MACANUFO, J. **Game Storming: A playbook for innovators, rulebreakers, and changemakers**. [S.l.]: O' Reilly Media, 2010.

GRIFFITH, I. et al. A simulation study of practical methods for technical debt management in agile software development. In: IEEE. **Proceedings of the Winter Simulation Conference 2014**. [S.l.], 2014. p. 1014–1025.

GUGGULOTHU, T.; MOIZ, S. A. Code smell detection using multi-label classification approach. **Software Quality Journal**, Springer, v. 28, n. 3, p. 1063–1086, 2020.

GUO, Y.; SPÍNOLA, R. O.; SEAMAN, C. Exploring the costs of technical debt management–a case study. **Empirical Software Engineering**, Springer, v. 21, n. 1, p. 159–182, 2016.

GUPTA, A.; SURI, B.; MISRA, S. A Systematic Literature Review: Code Bad Smells in Java Source Code. In: **ICCSA 2017**. [S.l.: s.n.], 2017. p. 665–682.

GUYATT, G. et al. Evidence-based medicine: a new approach to teaching the practice of medicine. **Jama**, American Medical Association, v. 268, n. 17, p. 2420–2425, 1992.

HAENDLER, T.; FRYSAK, J. Deconstructing the refactoring process from a problem-solving and decision-making perspective. In: INSTICC. **Proceedings of the 13th International Conference on Software Technologies - Volume 1: ICSOFT,**. [S.l.]: SciTePress, 2018. p. 363–372. ISBN 978-989-758-320-9.

HALEPMOLLASI, R.; TOSUN, A. Exploring the relationship between refactoring and code debt indicators. **Journal of Software: Evolution and Process**, Wiley Online Library, v. 36, n. 1, p. e2447, 2024.

HALL, T. et al. Some code smells have a significant but small effect on faults. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 23, n. 4, p. 33:1–33:39, sep. 2014. ISSN 1049-331X.

HANLON, J. J.; PICKETT, G. E. **Public health administration and practice**. [S.l.]: Time Mirror/Mosby, 1984.

HOZANO, M. et al. Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers. In: **ICEIS (2)**. [S.l.: s.n.], 2017. p. 474–482.

HOZANO, M. et al. Smells are sensitive to developers! on the efficiency of (un) guided customized detection. In: IEEE. **2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)**. [S.l.], 2017. p. 110–120.

HOZANO, M. et al. Are you smelling it? investigating how similar developers detect code smells. **Information and Software Technology**, v. 93, p. 130 – 146, 2018. ISSN 0950-5849. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0950584916303901>.

IBRAHIM, A. A. E.; KAMEL, A.; HASSAN, H. Object oriented metrics and quality attributes: A survey. In: **Proceedings of the 10th International Conference on Informatics and Systems**. [S.l.: s.n.], 2016. p. 312–319.

IEEE, C. S. **Guide to the Software Engineering Body of Knowledge (SWEBOK): Version 3.0**. [S.l.]: IEEE Computer Society Press, 2014.

International Standards Organisation - ISO. **International Standard ISO/IEC 9126. Information technology: Software product evaluation: Quality characteristics and guidelines for their use**. 1991.

International Standards Organisation - ISO. **ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models**. 2011. Available from Internet: <https://www.iso.org/standard/35733.html>.

JAAFAR, F. et al. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. **Empirical Softw. Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 21, n. 3, p. 896–931, jun. 2016. ISSN 1382-3256. Available from Internet: <http://dx.doi.org/10.1007/s10664-015-9361-0>.

JEFFERSON, A.; WAINER, M. Looking for smells: Visualizing java code with dotplots. In: **Proceedings of the International Conference on Software Engineering Research and Practice (SERP'08)**. Las Vegas, Nevada, USA: [s.n.], 2008.

JÜRJENS, J. et al. **Maintaining security in software evolution**. [S.l.]: Springer International Publishing, 2019.

KAMIYA, T.; KUSUMOTO, S.; INOUE, K. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. **IEEE Transactions on Software Engineering**, IEEE, v. 28, n. 7, p. 654–670, 2002.

KAUR, A. A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. **Archives of Computational Methods in Engineering**, Springer, v. 27, n. 4, p. 1267–1296, 2020.

KAUR, A. et al. Prioritization of code smells in object-oriented software: A review. **Materials Today: Proceedings**, 2021. ISSN 2214-7853. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S221478532038826X>.

KAYARVIZHY, N. Systematic review of object oriented metric tools. **International Journal of Computer Applications**, v. 135, n. 2, p. 8–13, February 2016. Published by Foundation of Computer Science (FCS), NY, USA.

KEMPPAINEN, T. The benefits of test automation in software development. 2022.

KERIEVSKY, J. **Refactoring to Patterns**. [S.l.]: Addison-Wesley, 2004.

KHOMH, F.; PENTA, M. D.; GUEHENEUC, Y. An exploratory study of the impact of code smells on software change-proneness. In: **2009 16th Working Conference on Reverse Engineering**. [S.l.: s.n.], 2009. p. 75–84. ISSN 1095-1350.

KHOMH, F. et al. An exploratory study of the impact of antipatterns on class change- and fault-proneness. **Empirical Software Engineering**, v. 17, n. 3, p. 243–275, Jun 2012. ISSN 1573-7616. Available from Internet: <https://doi.org/10.1007/s10664-011-9171-y>.

KHOMH, F. et al. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. **Journal of Systems and Software**, Elsevier, v. 84, n. 4, p. 559–572, 2011.

KITCHENHAM, B. et al. Systematic literature reviews in software engineering – a tertiary study. **Information and Software Technology**, v. 52, n. 8, p. 792 – 805, 2010. ISSN 0950-5849. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0950584910000467>.

KITCHENHAM, B. A.; BUDGEN, D.; BRERETON, P. **Evidence-based software engineering and systematic reviews**. [S.l.]: CRC press, 2015.

KOZIOLEK, H. Sustainability evaluation of software architectures: A systematic review. In: **Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS**. New York, NY, USA: Association for Computing Machinery, 2011. (QoSA-ISARCS '11), p. 3–12. ISBN 9781450307246.

KRUCHTEN, P.; NORD, R. L.; OZKAYA, I. Technical debt: From metaphor to theory and practice. **IEEE Software**, v. 29, n. 6, p. 18–21, Nov 2012. ISSN 0740-7459.

KRUSKALL, J.; LIBERMAN, M. Thesymmetric time warping problem: From continuous to discrete. **Time Warps, String Edits and Macromolecules: The Theory and Practice of SequenceComparison. Addison-Wesley**, 1983.

LACERDA, G. **DR-Tools Suite**. 2024. Available in https://drtools.site/.

LACERDA, G. et al. Code smells and refactoring: A tertiary systematic review of challenges and observations. **Journal of Systems and Software**, v. 167, p. 110610, 2020.

LACERDA, G.; PETRILLO, F.; PIMENTA, M. S. Dr-tools: a suite of lightweight open-source tools to measure and visualize java source code. In: IEEE. **2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.], 2020. p. 802–805.

LAHTINEN, S.; LEPPANEN, M. Refactoring patterns, practices for daily work. In: **Proceedings of the 10th Travelling Conference on Pattern Languages of Programs**. New York, NY, USA: ACM, 2016. (VikingPLoP '16), p. 6:1–6:8. ISBN 978-1-4503-4200-1. Available from Internet: <http://doi.acm.org/10.1145/3022636.3022642>.

LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. **biometrics**, JSTOR, p. 159–174, 1977.

LANZA, M. **Object-Oriented Reverse Engineering Coarse-grained, Fine-grained, and Evolutionary Software Visualization**. Thesis (PhD) — University of Bern, 2003.

LANZA, M.; MARINESCU, R. **Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems**. [S.l.]: Springer Verlag, 2010.

LAPLANTE, P. A.; KASSAB, M. **What every engineer should know about software engineering**. [S.l.]: CRC Press, 2022.

LENARDUZZI, V. et al. A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. **Journal of Systems and Software**, Elsevier, v. 171, p. 110827, 2020.

LENARDUZZI, V. et al. A critical comparison on six static analysis tools: Detection, agreement, and precision. **arXiv preprint arXiv:2101.08832**, 2021.

LEPPANEN, M. et al. Decision-making framework for refactoring. In: **2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)**. [S.l.: s.n.], 2015. p. 61–68.

LEPPANEN, M. et al. Refactoring-a shot in the dark? **IEEE Software**, IEEE, v. 32, n. 6, p. 62–70, 2015.

LI, Z.; AVGERIOU, P.; LIANG, P. A systematic mapping study on technical debt and its management. **Journal of Systems and Software**, Elsevier Ltd., v. 101, p. 193–220, mar 2015. ISSN 01641212.

LIM, E.; TAKSANDE, N.; SEAMAN, C. A balancing act: What software practitioners have to say about technical debt. **IEEE software**, IEEE, v. 29, n. 6, p. 22–27, 2012.

LINSTONE, H. A.; TUROFF, M. et al. **The delphi method**. [S.l.]: Addison-Wesley Reading, MA, 1975.

LIPPERT, M.; ROOCK, S. **Refactoring in large software projects: performing complex restructurings successfully**. [S.l.]: John Wiley & Sons, 2006.

LIU, H. et al. Scheduling of conflicting refactorings to promote quality improvement. In: **Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE'07)**. New York, NY, USA: ACM, 2007. p. 489–492. Available from Internet: <http://dl.acm.org/citation.cfm?id=1321716>.

LIU, H. et al. Prompt learning for multi-label code smell detection: A promising approach. **arXiv preprint arXiv:2402.10398**, 2024.

LORENZ, M.; KIDD, J. **Object-oriented Software Metrics: a Practical Guide**. Upper Saddle River, NJ, USA: Prentice Hall, Inc., 1994. ISBN 0-13-179292-X.

LOWE, W. W.; PANAS, T. Rapid construction of software comprehension tools. **International Journal of Software Engineering and Knowledge Engineering**, World Scientific, v. 15, n. 06, p. 995–1025, 2005.

LUJAN, S. et al. A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction. In: **Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation**. [S.l.: s.n.], 2020. p. 1–6.

MACIA, I. et al. On the relevance of code anomalies for identifying architecture degradation symptoms. In: IEEE. **2012 16th European Conference on Software Maintenance and Reengineering**. [S.l.], 2012. p. 277–286.

MAIGA, A. et al. Smurf: A svm-based incremental anti-pattern detection approach. In: IEEE. **2012 19th Working Conference on Reverse Engineering**. [S.l.], 2012. p. 466–475.

MALHOTRA, R.; SINGH, P. Exploiting bad-smells and object-oriented characteristics to prioritize classes for refactoring. **International Journal of System Assurance Engineering and Management**, Springer, v. 11, n. 2, p. 133–144, 2020.

MANTYLA, M.; LASSENIUS, C. Subjective evaluation of software evolvability using code smells: An empirical study. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 11, p. 395–431, September 2006. ISSN 1382-3256. Available from Internet: <http://portal.acm.org/citation.cf?id=1146474.1146489>.

MANTYLA, M.; VANHANEN, J.; LASSENIUS, C. A taxonomy and an initial empirical study of bad smells in code. In: **Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM'03)**. Washington, DC, USA: IEEE Computer Society, 2003. p. 381–. ISBN 0-7695-1905-9. Available from Internet: <http://dl.acm.org/citation.cfm?id=943571>.

MANTYLA, M.; VANHANEN, J.; LASSENIUS, C. Bad smells - humans as code critics. In: **Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)**. Washington, DC, USA: IEEE Computer Society, 2004. p. 399–408. ISBN 0-7695-2213-0. Available from Internet: <http://portal.acm.org/citation.cfm?id=1018431.1021447>.

MARCILIO, D. et al. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In: IEEE. **2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)**. [S.l.], 2019. p. 209–219.

MARCILIO, D. et al. Spongebugs: Automatically generating fix suggestions in response to static code analysis warnings. **Journal of Systems and Software**, Elsevier, p. 110671, 2020.

MARINESCU, C. et al. iplasma: An integrated platform for quality assessment of object-oriented design. In: IEEE. **2005 IEEE International Conference on Software Maintenance (ICSM Industrial and Tool Volume)**. [S.l.], 2005.

MARINESCU, R. Detection strategies: Metrics-based rules for detecting design flaws. In: IEEE. **20th IEEE International Conference on Software Maintenance, 2004. Proceedings.** [S.l.], 2004. p. 350–359.

MARINESCU, R. Assessing technical debt by identifying design flaws in software systems. **IBM Journal of Research and Development**, IBM, v. 56, n. 5, p. 9–1, 2012.

Martin Fowler. **Technical Debt**. 2023. Available from Internet: <https://martinfowler.com/bliki/TechnicalDebt.html>.

Martin Fowler. **Technical Debt Quadrant**. 2023. Available from Internet: <https://martinfowler.com/bliki/TechnicalDebtQuadrant.html>.

MARTIN, R. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. [S.l.]: Pearson, 2017.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. [S.l.]: Prentice Hall, 2008.

MARTIN, R. C. **Clean Craftsmanship: Disciplines, Standards, and Ethics**. [S.l.]: Pearson, 2021.

MARTIN, R. C.; MARTIN, M. **Agile Principles, Patterns, and Practices in C#**. [S.l.]: Prentice Hall, 2007.

MARTINS, J. et al. Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2020. p. 52–61.

MARTINS, J. et al. How do code smell co-occurrences removal impact internal quality attributes? a developers' perspective. In: **Proceedings of the XXXV Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2021. p. 54–63.

MAYVAN, B. B.; RASOOLZADEGAN, A.; JAFARI, A. J. Bad smell detection using quality metrics and refactoring opportunities. **Journal of Software: Evolution and Process**, Wiley Online Library, p. e2255, 2020.

MCCABE, T. A complexity measure. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Los Alamitos, CA, USA, v. 2, p. 308–320, 1976. ISSN 0098-5589.

MCCALL, J. **Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager**. General Electric, 1977. Available from Internet: <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA049055>.

MCCONNELL, S. Managing technical debt. **Construx Software Builders, Inc**, p. 1–14, 2008.

MCDONALD, J.; GREER, D. Investigating evolution in open source software. In: SPRINGER. **Computational Science and Its Applications–ICCSA 2019: 19th International Conference, Saint Petersburg, Russia, July 1–4, 2019, Proceedings, Part V 19**. [S.l.], 2019. p. 242–256.

MCGRATH, P. J. P. C.; LILJEDAHL, M. Twelve tips for conducting qualitative research interviews. **Medical Teacher**, Taylor & Francis, v. 41, n. 9, p. 1002–1006, 2019. PMID: 30261797. Available from Internet: <https://doi.org/10.1080/0142159X.2018.1497149>.

MEDARD, N.; MANASSE, N.; JEAN BAPTISTE, K. Universal health insurance in rwanda: major challenges and solutions for financial sustainability case study of rwanda community-based health insurance part i. **Pan African Medical Journal**, Pan African Medical Journal, v. 37, n. 55, 09 2020. ISSN 1937-8688. Available from Internet: <https://www.panafrican-med-journal.com/content/article/37/55/full>.

MEILIA, P. D. I. et al. A review of causal inference in forensic medicine. **Forensic Science, Medicine and Pathology**, Springer, p. 1–8, 2020.

MEREMIKWU, M. et al. Priority setting for systematic review of health care interventions in nigeria. **Health Policy**, Elsevier, v. 99, n. 3, p. 244–249, 2011.

MISBHAUDDIN, M.; ALSHAYEB, M. UML model refactoring: a systematic literature review. **Empirical Software Engineering**, v. 20, n. 1, p. 206–251, feb 2015. ISSN 1382-3256.

MOHA, N. et al. Decor: A method for the specification and detection of code and design smells. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Washington, DC, USA, 2010.

MOLNAR, A.-J.; MOTOGNA, S. A study of maintainability in evolving open-source software. **arXiv preprint arXiv:2009.00959**, 2020.

MSHELIA, Y. U.; APEH, S. T.; EDOGHOGHO, O. A comparative assessment of software metrics tools. In: IEEE. **2017 International Conference on Computing Networking and Informatics (ICCNI)**. [S.l.], 2017. p. 1–9.

MUMTAZ, H.; SINGH, P.; BLINCOE, K. A systematic mapping study on architectural smells detection. **Journal of Systems and Software**, Elsevier, p. 110885, 2020.

MUNSON, J. C.; ELBAUM, S. G. Code churn: A measure for estimating the impact of code change. In: IEEE. **Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)**. [S.l.], 1998. p. 24–31.

MURPHY-HILL, E.; BARIK, T.; BLACK, A. P. Interactive ambient visualizations for soft advice. **Information Visualization**, Sage Publications Sage UK: London, England, v. 12, n. 2, p. 107–132, 2013.

MURPHY-HILL, E.; BLACK, A. Seven habits of highly effective smell detector. In: **Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE'08)**. New York, NY, USA: ACM, 2008. p. 36–40. Available from Internet: <http://dl.acm.org/citation.cfm?id=1454261>.

MURPHY-HILL, E.; BLACK, A. An interactive ambient visualization for code smells. In: **Proceedings of 6th ACM Symposium on Software Visualization (SOFTVIS'10)**. Salt Lake City, Utah, USA: [s.n.], 2010.

MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. How we refactor, and how we know it. **IEEE Transactions on Software Engineering**, v. 38, n. 1, p. 5–18, Jan 2012. ISSN 0098-5589.

NEIGER, B. L.; THACKERAY, R.; FAGEN, M. C. Basic priority rating model 2.0: current applications for priority setting in health promotion practice. **Health promotion practice**, SAGE Publications Sage CA: Los Angeles, CA, v. 12, n. 2, p. 166–171, 2011.

NISTALA, P.; NORI, K. V.; REDDY, R. Software quality models: A systematic mapping study. In: IEEE. **2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)**. [S.l.], 2019. p. 125–134.

NORMALISATION, O. internationale de. **Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models**. [S.l.]: ISO, 2011.

NYAMAWE, A. S. et al. Recommending refactoring solutions based on traceability and code metrics. **IEEE Access**, IEEE, v. 6, p. 49460–49475, 2018.

OIZUMI, W. et al. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: IEEE. **2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)**. [S.l.], 2016. p. 440–451.

OIZUMI, W. et al. Revealing design problems in stinky code: a mixed-method study. In: **Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse**. [S.l.: s.n.], 2017. p. 1–10.

OIZUMI, W. et al. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In: **2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.: s.n.], 2019. p. 346–357.

OPDEBEECK, R.; ZEROUALI, A.; ROOVER, C. D. Control and data flow in security smell detection for infrastructure as code: Is it worth the effort? In: IEEE. **2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)**. [S.l.], 2023. p. 534–545.

OutSystems. **Study Reveals Majority of IT Leaders Consider Technical Debt One of the Biggest Threats to Innovation as They Build Back**. 2021. Available from Internet: <https://www.outsystems.com/news/study-reveals-technical-debt-is-threat-to-innovation/>.

PALOMBA, F. et al. Mining version histories for detecting code smells. **IEEE Transactions on Software Engineering**, IEEE, v. 41, n. 5, p. 462–489, 2015.

PALOMBA, F. et al. A large-scale empirical study on the lifecycle of code smell co-occurrences. **Information and Software Technology**, v. 99, p. 1 – 10, 2018. ISSN 0950-5849. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0950584918300211>.

PALOMBA, F. et al. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. **Empirical Software Engineering**, Springer, v. 23, n. 3, p. 1188–1221, 2018.

PALOMBA, F. et al. A textual-based technique for smell detection. In: IEEE. **2016 IEEE 24th international conference on program comprehension (ICPC)**. [S.l.], 2016. p. 1–10.

PALOMBA, F. et al. Beyond technical aspects: How do community smells influence the intensity of code smells? **IEEE transactions on software engineering**, IEEE, 2018.

PALOMBA, F. et al. An exploratory study on the relationship between changes and refactoring. In: IEEE. **2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)**. [S.l.], 2017. p. 176–185.

PANTIUCHINA, J.; LANZA, M.; BAVOTA, G. Improving code: The (mis) perception of quality metrics. In: IEEE. **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.], 2018. p. 80–91.

PANTIUCHINA, J. et al. Why developers refactor source code: A mining-based study. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 29, n. 4, p. 1–30, 2020.

PARNAS, D. L. Software aging. In: IEEE. **Proceedings of 16th International Conference on Software Engineering**. [S.l.], 1994. p. 279–287.

PARNIN, C.; GORG, C.; NNADI, O. A catalogue of lightweight visualizations to support code smell inspection. In: **Proceedings of 4th ACM Symposium on Software Visualization (SOFTVIS'08)**. New York, NY, USA: [s.n.], 2008.

PECORELLI, F. et al. Developer-driven code smell prioritization. In: **International Conference on Mining Software Repositories**. [S.l.: s.n.], 2020.

PERUMA, A. et al. Tsdetect: An open source test smells detection tool. In: **Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering**. [S.l.: s.n.], 2020. p. 1650–1654.

PETERS, R.; ZAIDMAN, A. Evaluating the lifespan of code smells using software repository mining. In: IEEE. **2012 16th European conference on software maintenance and reengineering**. [S.l.], 2012. p. 411–416.

PETRILLO, F. et al. Visualizing interactive and shared debugging sessions. In: IEEE. **2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)**. [S.l.], 2015. p. 140–144.

PETRILLO, F.; PIMENTA, M. S.; FREITAS, C. M. D. S. O estado-da-arte das ferramentas de visualização de software. In: **Proceedings of 15th IberoAmerican Conference on Software Engineering (CIbSE2012)**. [S.l.: s.n.], 2012.

PHF; DHHS. **Healthy People 2010 Toolkit: A field guide for Health Planning**. [S.l.]: Public Health Foundation, 2002.

PIETRZAK, B.; WALTER, B. Leveraging code smell detection with inter-smell relations. In: SPRINGER. **International Conference on Extreme Programming and Agile Processes in Software Engineering**. [S.l.], 2006. p. 75–84.

PINA, D.; GOLDMAN, A.; TONIN, G. Technical debt prioritization: Taxonomy, methods results, and practical characteristics. In: IEEE. **2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [S.l.], 2021. p. 206–213.

PONCE, F. et al. Smells and refactorings for microservices security: A multivocal literature review. **Journal of Systems and Software**, Elsevier, v. 192, p. 111393, 2022.

POPPENDIECK, M.; POPPENDIECK, T. **Lean Software Development: An Agile Toolkit**. [S.l.]: Addison-Wesley, 2003.

POPPENDIECK, M.; POPPENDIECK, T. **Implementing Lean Software Development: from concept to cash**. [S.l.]: Addison-Wesley, 2006.

PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach - 8th edition**. [S.l.]: McGraw-Hill, 2014.

Project Management Institute. **Is Technical Debt A Management Problem? Survey Says...** 2022. Available from Internet: <https://www.projectmanagement.com/blog/blogPostingView.cfm?blogPostingID=71979&thisPageURL=%2Fblog-post%2F71979%2Fis-technical-debt-a-management-problem---survey-says---&s=03#_=_>.

RADJENOVIC, D. et al. Software fault prediction metrics: A systematic literature review. **Information and Software Technology**, v. 55, n. 8, p. 1397 – 1418, 2013. ISSN 0950-5849.

RAPU, D. et al. Using history information to improve design flaws detection. In: **Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.** [S.l.: s.n.], 2004. p. 223–232.

RASOOL, G.; ARSHAD, Z. A review of code smell mining techniques. **Journal of Software: Evolution and Process**, v. 27, n. 11, p. 867–895, nov 2015. ISSN 20477473.

RATTAN, D.; BHATIA, R.; SINGH, M. Software clone detection: A systematic review. **Information and Software Technology**, Elsevier, v. 55, n. 7, p. 1165–1199, 2013.

RATTAN, D.; KAUR, J. Systematic Mapping Study of Metrics based Clone Detection Techniques. **Proceedings of the International Conference on Advances in Information Communication Technology & Computing - AICTC '16**, p. 1–7, 2016.

REIS, J. P. d.; CARNEIRO, G. d. F.; ANSLOW, C. Code smells detection and visualization: A systematic literature review. **arXiv preprint arXiv:2012.08842**, 2020.

REVEIZ, L. et al. Comparison of national health research priority-setting methods and characteristics in latin america and the caribbean, 2002-2012. **Revista Panamericana de Salud Pública**, SciELO Public Health, v. 34, p. 1–13, 2013.

RIEL, A. **Object-Oriented Design Heuristics**. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 020163385X.

RIOS, N.; NETO, M. G. de M.; SPÍNOLA, R. O. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. **Information and Software Technology**, Elsevier, v. 102, p. 117–145, 2018.

ROSSER, L. A.; NORTON, J. H. A systems perspective on technical debt. In: IEEE. **2021 IEEE Aerospace Conference (50100)**. [S.l.], 2021. p. 1–10.

SABIR, F. et al. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. **Software: Practice and Experience**, v. 49, n. August, p. 1–37, oct 2018. ISSN 00380644. Available from Internet: <http://doi.wiley.com/10.1002/spe.2639>.

SADOWSKI, C. et al. Lessons from building static analysis tools at google. **Communications of the ACM**, ACM New York, NY, USA, v. 61, n. 4, p. 58–66, 2018.

SADOWSKI, C.; STOLEE, K. T.; ELBAUM, S. How developers search for code: a case study. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering**. [S.l.: s.n.], 2015. p. 191–201.

SAE-LIM, N.; HAYASHI, S.; SAEKI, M. Context-based code smells prioritization for prefactoring. In: **2016 IEEE 24th International Conference on Program Comprehension (ICPC)**. [S.l.: s.n.], 2016. p. 1–10.

SAE-LIM, N.; HAYASHI, S.; SAEKI, M. An Investigative Study on How Developers Filter and Prioritize Code Smells. **IEICE Transactions on Information and Systems**, v. 101, n. 7, p. 1733–1742, jul. 2018.

SAE-LIM, N.; HAYASHI, S.; SAEKI, M. Context-based approach to prioritize code smells for prefactoring. **Journal of Software: Evolution and Process**, Wiley Online Library, v. 30, n. 6, p. e1886, 2018.

SALTELLI, A. et al. **Global sensitivity analysis: the primer**. [S.l.]: John Wiley & Sons, 2008.

SANTOS, J. A. M. et al. A systematic review on the code smell effect. **Journal of Systems and Software**, v. 144, n. July, p. 450–477, oct 2018. ISSN 01641212.

SARAIVA, J. de A. et al. Classifying metrics for assessing object-oriented software maintainability: A family of metrics' catalogs. **Journal of Systems and Software**, v. 103, p. 85 – 101, 2015. ISSN 0164-1212. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0164121215000126>.

SAS, D.; AVGERIOU, P.; UYUMAZ, U. On the evolution and impact of architectural smells—an industrial case study. **Empirical Software Engineering**, Springer, v. 27, n. 4, p. 86, 2022.

SCHILDT, H. **Java: The Complete Reference, Eleventh Edition**. [S.l.]: McGraw-Hill Education, 2018.

SCHMID, K. A formal approach to technical debt decision making. In: **Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures**. [S.l.: s.n.], 2013. p. 153–162.

SCHOLTES, I.; MAVRODIEV, P.; SCHWEITZER, F. From aristotle to ringelmann: a large-scale analysis of team productivity and coordination in open source software projects. **Empirical Software Engineering**, Springer, v. 21, n. 2, p. 642–683, 2016.

SCOTT, E.; CHARKIE, K. N.; PFAHL, D. Productivity, turnover, and team stability of agile teams in open-source software projects. In: **2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [S.l.: s.n.], 2020. p. 124–131.

SECURITY MAGAZINE. **Poor software costs the US 2.4 trillion**. 2022. Available from Internet: <https://www.securitymagazine.com/articles/98685-poor-software-costs-the-us-24-trillion>.

SEDGEWICK, R.; WAYNE, K. **Algorithms**. [S.l.]: Addison-wesley professional, 2011.

SEEMANN, M. **Code that fits in you head: Heuristics for Software Engineering**. [S.l.]: Pearson, 2021.

SEHGAL, R.; MEHROTRA, D.; BALA, M. Prioritizing the refactoring need for critical component using combined approach. **Decision Science Letters**, v. 7, n. 3, p. 257–272, 2018.

SHARMA, T. **Extending maintainability analysis beyond code smells**. Thesis (PhD) — Athens University of Economics and Business, 2019.

SHARMA, T. et al. On the feasibility of transfer-learning code smells using deep learning. **arXiv preprint arXiv:1904.03031**, 2019.

SHARMA, T.; FRAGKOULIS, M.; SPINELLIS, D. House of cards: Code smells in open-source C# repositories. In: **2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.: s.n.], 2017. p. 424–429.

SHARMA, T.; SINGH, P.; SPINELLIS, D. An empirical investigation on the relationship between design and architecture smells. **Empirical Software Engineering**, Springer, v. 25, n. 5, p. 4020–4068, 2020.

SHARMA, T.; SPINELLIS, D. A survey on software smells. **Journal of Systems and Software**, Elsevier Inc., v. 138, p. 158–173, apr 2018. ISSN 01641212.

SHENEAMER, A.; KALITA, J. A Survey of Software Clone Detection Techniques. **International Journal of Computer Applications**, v. 137, n. 10, p. 975–8887, 2016.

SIKET, I.; FERENC, R. Calculating metrics from large c++ programs. In: CITESEER. **6th International Conference on Applied Informatics Eger, Hungary**. [S.l.], 2004. p. 27–31.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of GitHub contributors. In: **24th International Symposium on the Foundations of Software Engineering (FSE)**. [S.l.: s.n.], 2016. p. 1–12.

SIMMONS, S. D. **Conceptions of Refactoring: An Investigation of Stack Overflow Posts**. Dissertation (Master) — Rochester Institute of Technology, 2020.

SIMONS, C.; SINGER, J.; WHITE, D. R. Search-based refactoring: Metrics are not enough. In: BARROS, M.; LABICHE, Y. (Ed.). **Search-Based Software Engineering**. Cham: Springer International Publishing, 2015. p. 47–61. ISBN 978-3-319-22183-0.

SINGH, S.; KAUR, S. A systematic literature review: Refactoring for disclosing code smells in object oriented software. **Ain Shams Engineering Journal**, Ain Shams University, mar 2017. ISSN 20904479.

SJOBERG, D. I. K. et al. Quantifying the effect of code smells on maintenance effort. **IEEE Transactions on Software Engineering**, v. 39, n. 8, p. 1144–1156, Aug 2013. ISSN 0098-5589.

SKELTON, M.; PAIS, M. **Team Topologies: Organizing Business and Technology Teams for Fast Flow**. [S.l.]: IT Revolution Press, 2019. ISBN 9781942788812.

SKELTON, M.; PAIS, M. **Remote Team Interactions Workbook**. [S.l.]: IT Revolution Press, 2022. ISBN 9781950508617.

SOBRINHO, E. V. de P.; LUCIA, A. D.; MAIA, M. de A. A systematic literature review on bad smells—5 w's: which, when, what, who, where. **IEEE Transactions on Software Engineering**, IEEE, 2018.

SOMMERVILLE, I. **Software Engineering - 10th edition**. [S.l.]: Addison-Wesley, 2016.

SOUSA, B. L.; BIGONHA, M. A. S.; FERREIRA, K. A. M. A systematic literature mapping on the relationship between design patterns and bad smells. In: **Proceedings of the 33rd Annual ACM Symposium on Applied Computing - SAC '18**. [S.l.]: ACM Press, 2018. p. 1528–1535. ISBN 9781450351911.

SOUSA, F. A. M. d. R. et al. Estabelecimento de prioridades em saúde numa comunidade: análise de um percurso. **Revista de Saúde Pública**, SciELO Public Health, v. 51, p. 11, 2017.

SOUSA, L. et al. When are smells indicators of architectural refactoring opportunities: A study of 50 software projects. In: **Proceedings of the 28th International Conference on Program Comprehension**. [S.l.: s.n.], 2020. p. 354–365.

STEIDL, D.; EDER, S. Prioritizing maintainability defects based on refactoring recommendations. In: **Proceedings of the 22nd International Conference on Program Comprehension**. New York, NY, USA: Association for Computing Machinery, 2014. (ICPC 2014), p. 168–176. ISBN 9781450328791. Available from Internet: <https://doi.org/10.1145/2597008.2597805>.

STERLING, C. **Managing Software Debt: building for inevitable change**. [S.l.]: Addison-Wesley, 2011.

STOREY, M.-A. et al. Remixing visualization to support collaboration in software maintenance. In: IEEE. **Frontiers of Software Maintenance, 2008. FoSM 2008.** [S.l.], 2008. p. 139–148.

STRIPE. **O Coeficiente dos Desenvolvedores**. 2018. Available from Internet: <https://stripe.com/br/reports/developer-coefficient-2018>.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. **Refactoring for software design smells: managing technical debt**. [S.l.]: Morgan Kaufmann, 2014.

SZOKE, G. Automating the refactoring process. **Acta Cybernetica**, v. 23, n. 2, p. 715–735, 2017.

SZOSKE, G. et al. Designing and developing automated refactoring transformations: An experience report. In: **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.: s.n.], 2016. v. 1, p. 693–697.

TAHIR, A. et al. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. **Information and Software Technology**, Elsevier, p. 106333, 2020.

TAIBI, D.; JANES, A.; LENARDUZZI, V. How developers perceive smells in source code: A replicated study. **Information and Software Technology**, v. 92, p. 223 – 235, 2017. ISSN 0950-5849. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0950584916304128>.

TELEA, A.; BYELAS, H.; VOINEA, L. A framework for reverse engineering large c++ code bases. **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 233, p. 143–159, 2009.

TELEA, A.; VOINEA, L. Visual software analytics for the build optimization of large-scale software systems. **Computational Statistics**, Springer, v. 26, n. 4, p. 635–654, 2011.

TORNHILL, A.; BORG, M. Code red: The business impact of code quality–a quantitative study of 39 proprietary production codebases. **arXiv preprint arXiv:2203.04374**, 2022.

TORNHILL, Adam. **The Bumpy Road Code Smell: Measuring Code Complexity by its Shape and Distribution**. 2023. Available from Internet: <https://codescene.com/blog/bumpy-road-code-complexity-in-context/>.

TRAVASSOS, G. et al. Detecting defects in object-oriented design: using reading techniques to increase software quality. In: **Proceedings of the 14th OOPSLA Conference**. [S.l.: s.n.], 1999.

TRINDADE, R. P. F.; BIGONHA, M. A. da S.; FERREIRA, K. A. M. Oracles of bad smells: a systematic literature review. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2020. p. 62–71.

TSANTALIS, N.; CHAIKALIS, T.; CHATZIGEORGIOU, A. Ten years of jdeodorant: Lessons learned from the hunt for smells. In: **2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2018. p. 4–14.

TSANTALIS, N.; KETKAR, A.; DIG, D. Refactoringminer 2.0. **IEEE Transactions on Software Engineering**, v. 48, n. 3, p. 930–950, 2022.

TUFANO, M. et al. When and why your code starts to smell bad. In: **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**. [S.l.: s.n.], 2015. v. 1, p. 403–414.

TUFANO, M. et al. When and why your code starts to smell bad (and whether the smells go away). **IEEE Transactions on Software Engineering**, v. 43, n. 11, p. 1063–1088, 2017.

VALE, G. et al. Challenges of resolving merge conflicts: A mining and survey study. **IEEE Transactions on Software Engineering**, IEEE, v. 48, n. 12, p. 4964–4985, 2021.

VASSALLO, C. et al. A large-scale empirical exploration on refactoring activities in open source software projects. **Science of Computer Programming**, v. 180, p. 1 – 15, 2019. ISSN 0167-6423. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167642318302557>.

VASSALLO, C. et al. How developers engage with static analysis tools in different contexts. **Empirical Software Engineering**, Springer, v. 25, n. 2, p. 1419–1457, 2020.

VAUCHER, S. et al. Tracking design smells: Lessons from a study of god classes. In: IEEE. **2009 16th working conference on reverse engineering**. [S.l.], 2009. p. 145–154.

VERMA, R.; KUMAR, K.; VERMA, H. K. A study of relevant parameters influencing code smell prioritization in object-oriented software systems. In: IEEE. **2021 6th International Conference on Signal Processing, Computing and Control (ISPCC)**. [S.l.], 2021. p. 150–154.

VERMA, R.; KUMAR, K.; VERMA, H. K. Code smell prioritization in object-oriented software systems: A systematic literature review. **Journal of Software: Evolution and Process**, Wiley Online Library, v. 35, n. 12, p. e2536, 2023.

VIDAL, S. et al. Ranking architecturally critical agglomerations of code smells. **Science of Computer Programming**, Elsevier, v. 182, p. 64–85, 2019.

VIDAL, S. et al. Jspirit: a flexible tool for the analysis of code smells. In: IEEE. **2015 34th International Conference of the Chilean Computer Science Society (SCCC)**. [S.l.], 2015. p. 1–6.

VIDAL, S. A.; MARCOS, C.; DÍAZ-PACE, J. A. An approach to prioritize code smells for refactoring. **Automated Software Engineering**, Springer, v. 23, n. 3, p. 501–532, 2016.

WAGEY, B. C.; HENDRADJAYA, B.; MARDIYANTO, M. S. A proposal of software maintainability model using code smell measurement. In: IEEE. **2015 International Conference on Data and Software Engineering (ICoDSE)**. [S.l.], 2015. p. 25–30.

WAGNER, S. et al. Operationalised product quality models and assessment: The quamoco approach. **Information and Software Technology**, Elsevier, v. 62, p. 101–123, 2015.

WAKE, W. C. **Refactoring Workbook**. [S.l.]: Addison-Wesley, 2003.

WANG, J.; CHO, S. S.; SONG, M. Repchabug: Automatically repairing incorrect change bugs in software evolution. In: IEEE. **2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)**. [S.l.], 2022. p. 1525–1530.

WEBSTER, B. F. **Pitfalls of object oriented development**. New York, NY, USA: M & T Books, 1995.

WETTEL, R.; LANZA, M. Visually localizing design problems with disharmony maps. In: **Proceedings of the 4th ACM symposium on Software visualization**. [S.l.: s.n.], 2008. p. 155–164.

WIMALASOORIYA, C. **Understanding software maintenance in the context of software architecture evolution.** Dissertation (Master) — University of Canterbury, 2019.

WOHLIN, C. et al. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012.

YADAV, P. S.; RAO, R. S.; MISHRA, A. An evaluation of multi-label classification approaches for method-level code smells detection. **IEEE Access**, IEEE, 2024.

YAMASHITA, A. Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment. In: IEEE. **2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.], 2015. p. 421–428.

YAMASHITA, A.; COUNSELL, S. Code smells as system-level indicators of maintainability: An empirical study. **Journal of Systems and Software**, v. 86, n. 10, p. 2639–2653, 2013. ISSN 0164-1212. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0164121213001258>.

YAMASHITA, A.; MOONEN, L. Do code smells reflect important maintainability aspects? In: IEEE. **2012 28th IEEE international conference on software maintenance (ICSM)**. [S.l.], 2012. p. 306–315.

YAMASHITA, A.; MOONEN, L. Do developers care about code smells? an exploratory survey. In: IEEE. **2013 20th Working Conference on Reverse Engineering (WCRE)**. [S.l.], 2013. p. 242–251.

YAMASHITA, A.; MOONEN, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 682–691. ISBN 978-1-4673-3076-3. Available from Internet: <http://dl.acm.org/citation.cfm?id=2486788.2486878>.

ZHANG, M. et al. Improving the precision of fowler's definitions of bad smells. In: **Proceedings of the 32nd Annual IEEE Software Engineering Workshop (SEW'09)**. [S.l.]: IEEE, 2009.

ZITZLER, E. et al. Performance assessment of multiobjective optimizers: An analysis and review. **IEEE Transactions on evolutionary computation**, IEEE, v. 7, n. 2, p. 117–132, 2003.

APÊNDICE A — INSPIRAÇÃO DO MÉTODO: MÉTODO DE HANLON

Há uma grande variedade de métodos para priorização (CHOI et al., 2019), que vão desde abordagens mais simples e subjetivas - como *dot voting* e *forced ranking* (GRAY; BROWN; MACANUFO, 2010), métodos mais objetivos porém demorados - *Método Delphi* (LINSTONE; TUROFF et al., 1975), até abordagens envolvendo estatística e complexidade, exigentes em termos de capacidade computacional, como - *multi-criteria decision analysis* (BALTUSSEN; NIESSEN, 2006). Estes diferentes métodos podem ser utilizados de forma isolada ou combinada. Cabe às equipes de intervenção na área da saúde a definição de um processo próprio, com etapas, e que incorpore critérios e procedimentos que possibilitem a identificação e intervenção em necessidades prioritárias da comunidade.

O *National Association of County and City Health Officials* - NACCHO (County Health Rankings and Roadmaps, 2013) tem como objetivo melhorar a saúde pública ao aderir a um conjunto de valores como equidade, excelência, participação, respeito, integridade, liderança, ciência e inovação. O NACCHO desenvolveu um guia com 5 métodos de priorização, utilizados amplamente na área da saúde, incluindo qual abordagem se adapta melhor às necessidades das agências, bem como instruções para implementação e exemplos práticos. Segundo o NACCHO, o objetivo do guia é apoiar as agências na tomada de decisões difíceis, pois o emprego de uma técnica de priorização definida pode fornecer um mecanismo estruturado para classificar questões objetivamente, enquanto, ao mesmo tempo, coleta informações da equipe de toda a agência e leva em consideração todas as facetas das questões de saúde concorrentes. O guia contempla os métodos *multi-voting technique, strategy grids, nominal group technique, prioritization matrix* e o *Método de Hanlon*.

O *Método de Hanlon* - MH (HANLON; PICKETT, 1984) também conhecido por *Hanlon Prioritization Process* e por *Basic Priority Rating* (BPR) é uma abordagem estabelecida para considerar objetiva e explicitamente os critérios de priorização definidos e fatores de viabilidade, reduzindo assim o grau de subjetividade. A abordagem foi criada por John Joseph Hanlon em 1954, como uma tentativa de priorizar os problemas de saúde em países em desenvolvimento. A abordagem foi apresentada mais formalmente por Hanlon em várias edições de uma publicação sobre administração pública em saúde (HANLON; PICKETT, 1984). A abordagem foi aperfeiçoada por vários autores ao longo de décadas, visando maximizar o seu uso. O MH é frequentemente citado como um

método viável, conforme evidenciado por sua inclusão no *Healthy People 2010 Toolkit*, guia desenvolvido sobre a chancela do Departamento de Saúde e Serviços Humanos dos EUA, como estratégia recomendada para definição de prioridades na área da saúde (PHF; DHHS, 2002). O método engloba componentes da equação que visam acomodar as considerações técnicas e não-técnicas, consideradas em qualquer definição de prioridade, relevantes para o processo.

$$MH = (A + B) \cdot C \cdot D \tag{A.1}$$

Na fórmula original (Equação A.1), *A* representa a magnitude do problema (escala de 0 a 10), *B* é referente a severidade (escala de 0 a 20), *C* representa a eficácia da solução (escala de 0 a 10) e *D* são os aspectos de execução. Estes aspectos de execução (Equação A.2) é um agregado de atributos, representado pelo acrônimo *PEARL*. Os atributos são pertinência (*Propriety, P*), viabilidade econômica (*Economics, E*), aceitação pela comunidade (*Accceptability, A*), disponibilidade de recursos (*Resources, R*) e legalidade (*Legality, L*). Os valores para cada um dos atributos *PEARL* podem ser 0 ou 1 (se algum dos atributos obtiver 0 inviabiliza a abordagem do problema, pois o valor final é obtido pelo produto dos atributos).

$$D = P \cdot E \cdot A \cdot R \cdot L \tag{A.2}$$

Em uma das últimas versões atualizadas por Hanlon and Pickett (1984), o autor considerou a pontuação máxima de 300 pra qualquer problema de saúde, assim houve a inclusão de um denominador de 3, para manter as pontuações entre 0 e 100 (Equação A.3). Como *PEARL* (variável *D*) é um multiplicador da equação, uma pontuação 0 remove automaticamente o problema de saúde em consideração.

$$MH = \frac{(A + B) \cdot C}{3} \cdot D \tag{A.3}$$

Neiger, Thackeray and Fagen (2011) propuseram algumas alterações relevantes no método, referente a composição dos critérios, removendo o *PEARL* (Equação A.4). Segundo os autores, embora o método tenha fornecido orientações básicas na definição de prioridades, não representa a ampla gama de dados atualmente disponíveis para os tomadores de decisão. Os componentes do modelo original dão mais peso ao impacto das doenças transmissíveis em comparação a doenças crônicas. Os autores também sugeriram um novo nome, *Basic Priority Rating 2.0*, para representar o modelo revisado. Nesta

proposta, os autores subdividem o atributo relacionado a severidade (*B*), englobando critérios como urgência, gravidade, perda econômica e impacto sobre os outros, todos com pontuação entre 0 e 5, totalizando os 20 pontos.

$$MH = \frac{(A + B) \cdot C}{3} \tag{A.4}$$

O MH tem sido utilizado como abordagem para fazer priorizações em diferentes contextos dentro na área da saúde, muitas vezes com adaptações. Alvarez et al. (2010) descreveram o uso do MH, também em seu formato original (Equação A.1), para o estabelecimento de prioridades de pesquisas em saúde, visando destinar recursos para a produção de evidências e propostas voltadas à solução dos problemas de saúde da população em Cuba. Este estudo foi liderado pela Divisão de Ciência e Técnica do Ministério da Saúde Pública, envolvendo 215 profissionais de 14 territórios, com 22 especialistas do ministério, acadêmicos e gestores do sistema de saúde. A pesquisa foi dividida em fases de classificação, *ranking* e ratificação adicional das informações, resultando em 5 prioridades de pesquisa em saúde, a partir da qual os projetos são selecionados para posterior financiamento. Este processo revelou importantes diferenças de perspectivas entre os *stakeholders* de diferentes territórios e níveis do sistema de saúde.

Meremikwu et al. (2011) desenvolveram um estudo para identificar os tópicos prioritários em estudos secundários que abordam problemas comuns de saúde na Nigéria. Uma lista primária de problemas de saúde foi compilada a partir do *National Health Management Information Systems* com informações de profissionais de saúde e ONGs, de seis zonas geopolíticas da Nigéria. Os pesquisadores usaram o MH em conjunto com *Método Delphi* no estudo, resultando em 50% dos problemas relacionados a malária. Com isso, os pesquisadores observaram que a identificação e priorização de revisões sistemáticas relevantes para os cuidados de saúde na Nigéria podem resultar em uma prestação de cuidados de saúde equitativos e baseado em evidências para as pessoas, principalmente em situações no qual a decisão de cuidados de saúde e outras configurações tenham poucos recursos.

Reveiz et al. (2013) desenvolveram uma pesquisa sistemática sobre métodos e características nacionais de definição de prioridades de pesquisa em saúde de 18 países da América Latina e Caribe. O estudo contemplou trabalhos publicados entre 2002 e 2012, que refletem políticas nacionais de pesquisa em saúde e métodos de priorização utilizados. Dentre os métodos, destacam-se o *3D Combined Approach Matrix*, abordagem baseada em reuniões/consenso do grupo, *Método Delphi* e o MH. A pesquisa também revelou que

embora os países tenham definidos os critérios para classificar as prioridades, nenhum trabalho indicou como os critérios foram definidos. Os critérios usados mais comuns incluem severidade da doença, relação custo-benefício e eficácia. Segundo os autores, os sistemas nacionais de pesquisa em saúde precisam desenvolver estratégias para fortalecer o processo de priorização, coordenando os esforços regionais, nacionais e globais.

Sousa et al. (2017) descreveram a metodologia usada no processo de estabelecimento de prioridades em saúde para intervenção comunitária, em uma comunidade idosa de Portugal. O processo integrou 4 etapas sucessivas de análise e classificação dos problemas: 1) agrupamento por nível e similitude, 2) classificação de acordo com critérios epidemiológicos, 3) ordenação por especialistas e 4) aplicação do MH (adotado o formato padrão, conforme Equação A.1, com alguns ajustes). As adaptações realizadas foram referentes a mudança de escala para severidade da doença (escala de 0 a 10) e *C* que representa a eficácia da solução (escala de 0,5 a 1,5 - em que 0,5 corresponde a um problema difícil de solucionar e 1,5 representa um problema de fácil solução). Dos 19 problemas analisados, a baixa interação social na participação comunitária resultou como problema prioritário.

Deane et al. (2018) realizaram um estudo combinando o MH com o *Método Delphi* modificado para identificar as prioridades de pesquisa da medicina de emergência pediátrica (PEM) na Austrália e Nova Zelândia, realizada pelo PREDICT[1]. Segundo os autores, as prioridades de pesquisa são difíceis de determinar, muitas vezes confiando em interesses individuais ou a algum trabalho prévio. Assim, foi determinada as prioridades dos pesquisadores ativos em PEM, incluindo a ênfase na viabilidade de uma pesquisa. O MH (Equação A.5) foi usado em cada pesquisa, ponderando formalmente com uma escala de valores de 1 a 10, os domínios de prevalência da doença (*A*), a gravidade da condição (*B*) e a viabilidade de realizar a pesquisa (*C*).

$$MH = (A + 2B) \cdot C \tag{A.5}$$

Este processo de priorização estabeleceu uma lista de temas de pesquisa que incluíram oxigenação de alto fluxo na intubação de emergência, ressuscitação com volume de fluído na sepse, gestão de lesões da coluna cervical, terapia intravenosa para asma, entre outros.

Alavinia et al. (2019) propuseram um estudo para priorizar os domínios de atendimento de reabilitação de lesão ou doença de medula espinhal (*Spinal cord injury or*

[1] *Paediatric Research in Emergency Departments International Collaborative*

disease - SCI-D) com base na importância clínica e viabilidade para informar o desenvolvimento de indicadores de qualidade de atendimento de adultos com SCI-D no Canadá. O método englobava um comitê consultivo externo de 17 membros representando as principais partes interessadas (cientistas, médicos, terapeutas, pacientes e organizações) que classificaram 37 domínios de reabilitação previamente sinalizados pela equipe do projeto em relação a lacunas entre a geração de conhecimento e implementação clínica. O MH foi adaptado com escala entre 0 a 10 (Equação A.6) incluindo tamanho (*A*), gravidade segundo o ranking de domínio (*B*, multiplicado por 2, por causa de sua importância) e a efetividade da intervenção (*C*) representando o *score* de prioridade (*Priority*). Adicionalmente, um critério *EAARS* foi modificado da versão original do MH para classificar a viabilidade (*Feasibility*, Equação A.7) em uma escala de 0 a 4 (considerando 4 como alto). O *EAARS* representa: a) senso econômico do problema (*Economics*), b) aceitação da comunidade (*Acceptability*), c) disponibilidade dos serviços para pacientes internados/ambulatoriais (*Accessibility*), d) recursos disponíveis (*Resources*) e e) facilidade de implementação (*Simplicity*). O *score* final é dado pelo produto de ambos os fatores (Equação A.8).

$$Priority = [A + (2 \cdot B)] \cdot C \tag{A.6}$$

$$Feasibility = E + A + A + R + S \tag{A.7}$$

$$MH = Priority \cdot Feasibility \tag{A.8}$$

Assim, o MH modificado foi utilizado para facilitar a priorização de 11 de 37 domínios para melhorar a qualidade de atendimento de SCI-D, garantindo que os adultos com SCI-D recebam serviços de reabilitação em tempo hábil, seguro e eficaz.

Medard, Manasse and Jean Baptiste (2020) desenvolveram um estudo para avaliar os desafios enfrentados sustentabilidade financeira do *Rwanda Community-Based Health Insurance* (RCBHI). O RCBHI é considerado o principal veículo para o *Universal Health Coverage*[2] (UHC). Um conjunto de dados foram analisados usando *analyzing qualitative*

[2]É definido como a garantia de que todas as pessoas a serviços de saúde necessários (incluindo prevenção, promoção, tratamento, reabilitação, paliação) de qualidade suficiente para ser eficaz, sem expor os usuários a dificuldades financeiras. É considerado uma das 3 prioridades estratégicas do programa da Organização Mundial da Saúde (WHO). Mais informações em https://www.who.int/healthsystems/universal_health_coverage/en/

data G3658-6 approach e o MH, visto que o RCBHI enfrentava um déficit financeiro crônico. O RCBHI foi introduzido em 1999 e até 2012 estava longe de uma cobertura de saúde efetiva. Segundo os resultados apresentados, a sustentabilidade financeira do RCBHI é alcançável, porém depende da persistência de esforços e compromissos políticos para alcançar o UHC.

Azizi et al. (2020) desenvolveram uma pesquisa para apoiar gestores de saúde nas tomadas de decisão sobre estratégias de base comunitária que tenham impacto efetivo na prevenção de suicídios no noroeste do Irã. O suicídio é uma crise de saúde pública complexa e contínua que é um curso de morte mal interpretado, afetando fortemente a saúde mental e qualidade de vida dos indivíduos e da comunidade. Apesar da disponibilidade de programas de prevenção de suicídios em todo o mundo, segundo os autores, nenhuma investigação anterior foi realizada usando combinações de revisões sistemáticas com opiniões de especialistas. Estas duas estratégias foram combinadas usando o MH adaptado (Equação A.9), substituindo o *PEARL*. Assim, as estratégias foram avaliadas com base nos critérios viabilidade (*F*), importância (*I*), custo-benefício (*CE*), oportunidade (*T*) e aceitabilidade (*A*), usando uma escala de 5 pontos.

$$Score = F + I + CE + T + A \tag{A.9}$$

Segundo os resultados da priorização estão o gerenciamento de tentativas de suicídio, desenvolvimento de registro de suicídios e tratamento da depressão.

Em 2014, a PAHO/WHO adaptou o MH, que prioriza programas de controle de doenças para sua ampla gama de áreas de programas de saúde, e o usou para implementar o plano estratégico da instituição de 2014-2019 (CHOI et al., 2019). Para esta adaptação, a PAHO/WHO estabeleceu um grupo consultivo com diversos representantes de diferentes continentes (América do Norte, América Central, América do Sul e Caribe) para trabalhar no programa de controle de doenças (visão patogênica) e programas não orientado a doenças (visão salutogênica). Durante este período, o grupo liderado por pesquisadores do Canadá revisou mais de 15 métodos de priorização existentes, avaliou o MH original e as revisões subsequentes para desenvolver sua versão adaptada. A PAHO/WHO optou pelo MH (Equação A.10) por ser o mais adequado para adaptação, podendo fornecer a versatilidade necessária para acomodar questões técnicas e políticas, consideradas relevantes para uma organização tão complexa. Da equação original, foram mantidos os componentes tamanho do problema (*A*), severidade (*B*, considerando a subdivisão definida em (NEIGER; THACKERAY; FAGEN, 2011)) e eficácia da interven-

ção (*C*). Foi eliminado o componente *D*, que representa o acrônimo *PEARL*. Também, foram adicionados os componentes desigualdade (*E*, com escala entre 0 e 5) e posicionamento institucional (*F*, um fator entre 0.67 e 1.5 - uma pontuação maior que 1 indica que a PAHO/WHO deve aumentar sua cooperação técnica). O componente *PEARL* foi descartado porque serve a um propósito de pré-triagem, mas não no processo de definição de prioridades da PAHO/WHO. A constante 5.25 usada no denominador dá ao *score* um intervalo entre 0 e 100.

$$MH = \frac{(A + B + E) \cdot C}{5.25} \cdot F \tag{A.10}$$

Segundo os autores do trabalho (CHOI et al., 2019), o MH adaptado para a realidade da PAHO/WHO fornece uma abordagem refinada para priorizar programas de saúde pública que incluem áreas de controle de doenças e programas não orientados a doenças. Além de ser uma abordagem transparente, objetiva e aplicada de forma sistemática, o método pode ser útil para a WHO e governo de países com necessidades semelhantes.

O MH serviu de inspiração para o contexto de desenvolvimento de software, dada inúmeras semelhanças entre os critérios adotados no método, permitir adaptações relacionadas a questões técnicas e do time de desenvolvimento. Até onde se sabe, é o primeiro trabalho a usar o MH na área da ES.

APÊNDICE B — LISTA DE PROJETOS TESTE

Durante o desenvolvimento do método e da ferramenta *DR-Tools Code Health*, para fazer ajustes nos pesos e definições de uso padrão, foram utilizados os seguintes projetos:

1. aDoctor-master
2. apache-oodt-1.2.3-src
3. argouml-master
4. atunes-refactorings-master
5. aws-sdk-java-master
6. cassandra
7. checkstyle-master
8. cucumber-jvm
9. didactic-bank-application-main
10. dropwizard-master
11. eureka
12. findbugs-3.0.1
13. gson
14. guava
15. HealthWatcher
16. heritrix3
17. hibernate-orm-master
18. hive
19. hsqldb
20. huntbugs-master
21. jabref
22. javapoet-master
23. javassist
24. jdepend-master
25. jenkins
26. jfreechart-master

27. JHotDraw7.0.6
28. jmonkeyengine
29. jpacman-master
30. junit5-master
31. kafka
32. lucene
33. microservices-demo-master
34. MobileMedia-Cosmos-VP-v7
35. mockito
36. netbeans
37. netflix-commons
38. ozone
39. person-master
40. pmd-src-6.15.0
41. sneer-master
42. sokrates
43. sonarqube-master
44. spotbugs-master
45. spring-framework-master
46. ssl-pinning-java-main
47. stackgres-0.7.1
48. storm
49. sweethome3d-code-r8450-trunk-SweetHome3D
50. zuul

APÊNDICE C — EXEMPLO DE USO DO *DR-TOOLS CODE HEALTH*

Para exemplificar o uso do *DR-Tools Code Health*, é apresentado a seguir sua aplicação para apoiar a priorização de *smells* no código.

C.1 Definições Iniciais

Na Seção 4.2.2, foram apresentados as referências e pesos atribuídos para cada característica que compõe um *smell*.

São considerados alguns *thresholds* (RADJENOVIC et al., 2013; SARAIVA et al., 2015; BIGONHA et al., 2019) para as métricas que fazem parte da estratégia de detecção do *smell*. Os *thresholds* usados neste exemplo são apresentados na Tabela C.1.

Tabela C.1 – *Thresholds* considerados nas métricas

Métrica	**Sigla**	***Threshold***
Method Lines of Code	MLOC	30
Cyclomatic Complexity	CYCLO	10
Nested Block Depth	NBD	3
Number of Parameters	PARAM	5

Fonte: (RADJENOVIC et al., 2013; SARAIVA et al., 2015; BIGONHA et al., 2019)

As configurações iniciais para a importância de um *smell* seguem a Tabela 4.5, com os valores *Critical*=4, *High*=3 e *Normal*=1. As configurações para intervenção possuem os valores *Critical*=4, *High*=3 e *Low*=1 (Tabela 4.7).

Para o cálculo da priorização, definiu-se 3 *smells* na granularidade de método, conforme as características apresentadas na Tabela C.2. Cada atributo de qualidade pode ser associado a um peso (Tabela 4.6). Assim, *Maintainability (Critical), Complexity (High)* e *Understandability (High)* estão associados a valores 3, 2 e 2, respectivamente.

Finalmente, é apresentado na Tabela C.3 os elementos de código utilizados para computação da priorização, com base na presença dos *smells*. No exemplo, são considerados 3 métodos (*m1, m2, m3*), pertencentes a duas classes (*C1* e *C2*), do pacote *p1*. Para fins de simplificação, considerou-se apenas os elementos da granularidade de método para a computação da priorização.

Tabela C.2 – Definições dos *smells* considerados no exemplo

Nome	**Estratégia de Detecção**	**Impacto na Qualidade**	**Importância**	**Intervenção**
Long Method	$Method(MLOC) > Threshold(MLOC)$	*Understandability, Maintainability*	*High*	*Planned*
Complex Method	$Method(MLOC) > Threshold(MLOC) \wedge Method(CYCLO) > Threshold(CYCLO) \wedge Method(NBD) > Threshold(NBD)$	*Understandability, Maintainability, Complexity*	*Critical*	*Immediate*
Long Parameter List	$Method(PARAM) > Threshold(PARAM)$	*Maintainability*	*Normal*	*Low*

Fonte: O Autor

Tabela C.3 – Elementos de código utilizados no exemplo para cálculo da priorização

Elemento	***Smells* Detectados**	**Valores Métricos**
p1.C1.m1	*Long Method, Long Parameter List*	MLOC=200; PARAM=7
p1.C1.m2	*Complex Method*	MLOC=150; CYCLO=85; NBD=7
p1.C2.m3	*Complex Method, Long Parameter List*	MLOC=100; CYCLO=120; NBD=5; PARAM=10

Fonte: O Autor

C.2 Computando os Critérios

Após definidos os parâmetros iniciais, são calculados os critérios de severidade, representatividade, impacto na qualidade e intervenção, componentes do modelo proposto.

A **severidade de um elemento de código** considera o somatório da média dos fatores métricos componentes dos *smells* que excedem os *thresholds* e o peso dado para cada *smell* encontrado no elemento de código.

A Tabela C.4 apresenta todos os valores considerados para o cálculo da severidade. O *MetricFactor* (Equação 4.1) é calculado para cada métrica que compõe a estratégia de

detecção do DI. Depois, o *MeanFactor* (Equação 4.2) e o *SeveritySmell* (Equação 4.3) são calculados para cada *smell* presente nos métodos *m1, m2* e *m3*. A partir destes valores, é possível calcular o *SeverityCE*, ou seja, a severidade de cada método (Equação 4.4). Assim, é obtido o $Severity_{m1} = 5,350$, $Severity_{m2} = 5,278$ e $Severity_{m3} = 4,933$. Considerando os valores normalizados (Equação 4.9), tem-se $Severity_{m1} = 10,000$, $Severity_{m2} = 8,440$ e $Severity_{m3} = 1,000$.

Tabela C.4 – Métodos utilizados no exemplo para cálculo da severidade

Método	**DIs**	***MetricFactor***	***MeanFactor***	***SeveritySmell***
p1.C1.m1	*Long Parameter List*	1,400 (PARAM)	1,400	1,400
	Long Method	6,667 (MLOC)	6,667	20,000
p1.C1.m2	*Complex Method*	8,500 (CYCLO)		
		2,333 (NBD)	5,278	21,111
		5,000 (MLOC)		
p1.C2.m3	*Complex Method*	12,000 (CYCLO)		
		1,667 (NBD)	5,667	22,667
		3,333 (MLOC)		
	Long Parameter List	2,000 (PARAM)	2,000	2,000

Fonte: O Autor

Por exemplo, tomando por base o método *m1* que possui 2 *smells* (*Long Method* e *Long Parameter List*). Primeiro, o modelo calcula o *MetricFactor* de cada métrica presente em cada *smell*. Para o *Long Parameter List*, o *MetricFactor* é a razão do valor métrico de PARAM pelo seu *threshold* (ou seja, $7/5 = 1,400$). Já para o *Long Method*, o *MetricFactor* é $200/30 = 6,667$. O *MeanFactor* faz uma média do *MetricFactor* pelo número de métricas definidas no *smell* (neste caso, são os mesmos valores $1,400$ e $6,667$, pois cada *smell* só possui uma métrica em sua definição). O próximo passo é o cálculo da severidade do *smell* (*SeveritySmell*), considerando o produto do *MeanFactor* pelo peso definido pelo desenvolvedor para a importância do *smell* (*Long Parameter List* foi definido como *Normal*, peso=1) resultando em $Severity_{LongParameterList} = 1,400$. Já para o *Long Method*, o $Severity_{LongMethod}$ é calculado por $6,667 \cdot 3 = 20,000$, com importância *High*, ou peso=3). Finalmente, para o cálculo da severidade do elemento de código *m1*

($Severity_{m1}$), considera-se a soma das severidades dos *smells* ($1,400 + 20,000$) dividido pela soma dos pesos das importâncias dos *smells* ($1+3 = 4$), obtendo, $21,400/4 = 5,350$.

A **representatividade de um elemento de código** está relacionado com os *smells* em uma dada granularidade (pacote, classe, método), permitindo avaliar quais elementos de código são mais afetados pelos *smells*.

Dado que os *smells* detectados no *m1* (*Long Parameter List, Long Method*), *m2* (*Complex Method*) e *m3* (*Long Parameter List, Complex Method*) e considerando o total de instâncias dos *smells* na granularidade de método é 5 (soma dos *smells* encontrados em *m1, m2* e *m3*). Assim, com base na Equação 4.5, os valores referentes a representatividade são

$Representativity_{m1} = 0,4$, $Representativity_{m2} = 0,2$ e $Representativity_{m3} = 0,4$.

Os valores normalizados para representatividade, segundo a Equação 4.9, são $Representativity_{m1} = 10,00$, $Representativity_{m2} = 1,00$ e $Representativity_{m3} = 10,00$.

Para computar o **impacto na qualidade** é necessário usar os atributos de qualidade associados aos *smells* detectados em um elemento de código. Os atributos de qualidade são associados a pesos, conforme apresentado na Tabela 4.6. Neste exemplo, assume-se que *Maintainability* é *Critical* (peso=3) e que *Complexity* e *Understandability* são *High* (peso=2).

Conforme apresentado na Tabela C.2, *Complex Method* é o *smell* que mais afeta diferentes atributos de qualidade (*Understandability, Maintainability, Complexity*).

Como exemplo, o método *m1* possui 2 *smells* (*Long Method* impactando *Understandability* e *Maintainability*; *Long Parameter List* impactando *Maintainability*). Observando os pesos atribuídos aos atributos de qualidade vinculados, para o *Long Parameter List* o *QualityDI* é 3 e para o *Long Method* o *QualitySmell* é $3 + 2 = 5$ (Equação 4.6). Após o cálculo de cada *QualitySmell* ($Quality_{LongParameterList}$ e $Quality_{LongMethod}$), o próximo passo é calcular o impacto na qualidade de um elemento de código (*QualityCE*, Equação 4.7) através da soma dos pesos dos atributos de qualidade, multiplicado pela quantidade de atributos vinculados, ou seja, $(3 + 5) \cdot 3 = 24,000$. Assim, com base na Equação 4.7, os valores referentes ao impacto na qualidade do elemento de código são $Quality_{m1} = 24,000$, $Quality_{m2} = 21,000$ e $Quality_{m3} = 40,000$. Os valores normalizados, segundo a Equação 4.9, são $Quality_{m1} = 2,42$, $Quality_{m2} = 1,00$ e

$Quality_{m3} = 10,00$.

Finalmente, a **intervenção** é relacionada ao grau de atenção a ser dada a um elemento de código, levando em consideração os *smells* detectados neste elemento. Os pesos (Tabela 4.7) relacionados a intervenção são associados a um *smell* na sua definição. O cálculo é feito através da soma dos pesos atribuídos aos *smells* multiplicado pela diversidade, ou seja a quantidade diferente de *smells* naquele elemento de código. Como exemplo, o método *m1* possui 2 *smells* (*Long Method* com intervenção *Planned*, ou peso=3, e *Long Parameter List* com intervenção *Low*, ou peso=0,5). Portanto, o cálculo do *InterventionCE* (Equação 4.8) para o *m1* é $(0,5 + 3) \cdot 2 = 7,00$. Os valores referentes ao intervenção são $Intervention_{m1} = 7,00$, $Intervention_{m2} = 4,00$ e $Intervention_{m3} = 9,00$. Também, os valores normalizados, segundo a Equação 4.9, são $Intervention_{m1} = 7,00$, $Intervention_{m2} = 1,00$ e $Intervention_{m3} = 10,00$.

C.3 Calculando o CDI

Após todos os critérios computados, é possível calcular o CDI (Equação 4.10) dos métodos *m1, m2* e *m3*. A Tabela C.5 apresenta os resultados dos cálculos dos critérios, considerando valores absolutos e seu respectivo valor normalizado. É importante enfatizar que os critérios tem limite intervalar entre 1 e 10.

A partir destes números, é possível fazer algumas avaliações. Se for considerado apenas um critério, como severidade por exemplo, o método *m1* é o mais problemático. Se o critério escolhido for representatividade, é possível observar que dois métodos (*m1* e *3*) obtiveram o mesmo valor, indicando que sob este critério, são os que devem ser considerados para alguma ação dos desenvolvedores. No entanto, ao calcular o CDI de cada método e, portanto considerando todos os critérios, o elemento mais problemático passa a ser o método *m1*. O CDI normalizado apresenta um valor intervalar entre 1 e 100.

Tabela C.5 – Resultado da computação de cada critério (valor absoluto e normalizado) e o CDI dos métodos

Critério	***m1***		***m2***		***m3***	
	Abs	Norm	Abs	Norm	Abs	Norm
Severidade	$5,350$	$10,000$	$5,278$	$8,440$	$4,933$	$1,000$
Representatividade	$0,400$	$10,000$	$0,200$	$1,000$	$0,400$	$10,000$
Qualidade	$24,000$	$2,421$	$21,000$	$1,000$	$40,000$	$10,000$
Intervenção	$7,000$	$6,400$	$4,000$	$1,000$	$9,000$	$10,000$
CDI	**59,36**		**8,44**		**10,00**	

Fonte: O Autor

APÊNDICE D — LISTA DE MÉTRICAS POR CONTEXTO

Lista de métricas utilizadas e definidas pelo *DR-Tools Code Health*, tanto para apoiar as análises como para compor as heurísticas de detecção dos *smells*.

- *Summary (15): Total of namespaces, total of types, mean number of types/namespaces, total of lines of code (SLOC), average number of SLOC/types* (com *median and standard deviation*), *total of methods, average number of methods/types (com median* e *standard deviation), total of complexity (CYCLO),* e *average number of complexity/types (*com *median and standard deviation)*
- *Namespaces (2): Number of classes/types (NOC)* e *number of abstract classes (NAC)*
- *Types (14): Lines of code (SLOC), number of methods (NOM), number of public methods (NPM), class complexity (WMC), number of dependencies (DEP), number of internal dependencies (I-DEP), Number of other types that depend on a given type (FAN-IN), number of other types referenced by a type (FAN-OUT), number of fields/attributes (NOA), lack of cohesion in methods (LCOM3), deep in inheritance tree (DIT), number of children (CHILD), number of public attributes/fields (NPA)* e *number of cyclic dependencies (types) (CDEP)*
- *Methods (5): Lines of code (MLOC), cyclomatic complexity (CYCLO), number of invocations (CALLS), nested block depth (NBD)* e *number of parameters (PARAM)*
- *Namespace Coupling (5): Afferent coupling (CA), efferent coupling (CE), instability (I), abstractness degree (A)* e *normalized distance (D)*
- *Type Coupling (4): number of dependencies (DEP), number of internal dependencies (I-DEP), Number of other types that depend on a given type (FAN-IN)* e *number of other types referenced by a type (FAN-OUT)*
- *Dependencies (3): General dependencies (DEP), internal dependencies (I-DEP) e cyclic dependencies (CDEP)*

Para estas métricas, foram considerados os *thresholds* definidos por trabalhos como (Bakar; Boughton, 2012; RADJENOVIC et al., 2013; FILÓ; BIGONHA, 2015; SARAIVA et al., 2015; BIGONHA et al., 2019).

Tabela D.1 – Lista de métricas e referências para *thresholds*

Categoria	Acrônimo	Descrição	Thresholds Considerados
Project	SMALL	Small Project	small project with < 50 KLOC or 200 < classes
Project	MEDIUM	Medium Project	medium project with (50 KLOC <= project <= 250 KLOC) or (200 <= classes <= 1000)
Project	LARGE	Large Project	large project with > 250 KLOC or > 1000 classes
Namespace	NOC	Number of Types/Classes	Good: <= 11; Regular: between 11 and 28; Bad: > 28
Namespace	NAC	Number of Abstract Types/Classes	without references
Type	SLOC	Type/Class Line of Code	Bad: > 500
Type	NOM	Number of Functions/Methods	Good: <= 6; Regular: between 6 and 14; Bad: > 14
Type	NPM	Number of Public Methods	Good: <= 10; Regular: between 11 and 40; Bad: > 40
Type	WMC	Weighted Methods per Class	Good: <= 20; Regular: between 20 and 100; Bad: > 100
Type	DEP	Number of external (external APIs, frameworks, libs) types/classes dependencies	Bad: > 20
Type	I-DEP	Number of other internal types/classes dependencies	Bad: > 15
Type	FAN-IN	Number of other types that depend on a given Type	Bad: > 10
Type	FAN-OUT	Number of other types referenced by a type	Bad: > 15
Type	NOA	Number of Attributes/Fields	Good: <= 3; Regular: between 3 and 8; Bad: > 8
Type	LCOM3	Lack of Cohesion in Methods	Good: = 0; Regular: between 0 and 0,8; Bad: > 0,8
Type	DIT	Deep in Inheritance Tree	Bad: > 8,0
Type	CHILD	Number of Children	Bad: >= 1,0
Type	NPA	Number of Public Attributes/Fields	Bad: > 1,0
Method	MLOC	Method Lines of Code	Good: lt;= 10; Regular: between 10 and 30; Bad: > 30
Method	CYCLO	Cyclomatic Complexity	Good: <= 2; Regular: between 2 and 10; Bad: > 10
Method	CALLS	Number of Invocations	Bad: > 5
Method	NBD	Nested Block Depth	Good: <= 1; Regular: between 1 and 3; Bad: > 3
Method	PARAM	Number of Parameters	Good: <= 2; Regular: between 2 and 4; Bad: > 4
Namespace Coupling	CA	Afferent Coupling	Good: <= 7; Regular: between 7 and 39; Bad: > 39
Namespace Coupling	CE	Efferent Coupling	Good: <= 6; Regular: between 6 and 16; Bad: > 16
Namespace Coupling	I	Package Instability	range between 0=Maximally stability and 1=Maximally instability
Namespace Coupling	A	Abstractness Degree	range between 0=Minimally abstractness and 1=Maximally abstractness
Namespace Coupling	D	Normalized Distance	range between 0=exactly located in the main sequence and 1=far from the main sequence
Namespace Coupling	CDEP	Number of Cyclic Dependencies	Bad: > 1,0
Type Coupling	DEP	Number of external (external APIs, frameworks, libs) types/classes dependencies	Bad: > 20
Type Coupling	I-DEP	Number of other internal types/classes dependencies	Bad: > 15
Type Coupling	FAN-IN	Number of other types that depend on a given type	Bad: > 10
Type Coupling	FAN-OUT	Number of other types referenced by a type	Bad: > 15

Fonte: O Autor

APÊNDICE E — LISTA E CONFIGURAÇÕES DOS SMELLS NO *DR-TOOLS CODE HEALTH*

E.1 Granularidade: Namespace

Nome: Too Large Package
Descrição: Packages/Subsystems with a high number of classes/packages indicate that they serve more than one specific responsibility
Importância: NORMAL
Impacto na Qualidade: Understandability, Modularity
Ação de intervenção: PLANNED
Referência: M. Lippert, S. Roock, "Refactoring in Large Software Projects: Performing Complex Restructurings Successfully". John Wiley and Sons, 2006.

Nome: Cyclic Dependency
Descrição: This smell occurs when two or more packages/subsystems depend on each other
Importância: HIGH
Impacto na Qualidade: Coupling, Modularity
Ação de intervenção: NORMAL
Referência: M. Lippert, S. Roock, "Refactoring in Large Software Projects: Performing Complex Restructurings Successfully". John Wiley and Sons, 2006.

E.2 Granularidade: Type

Nome: God Class
Descrição: This smell is huge, complex, and has an extremely high number of fields and methods
Importância: HIGH
Impacto na Qualidade: Understandability, Maintainability
Ação de intervenção: PLANNED
Referência: Bafandeh Mayvan B, Rasoolzadegan A, Javan Jafari A. Bad smell detection

using quality metrics and refactoring opportunities. J Softw Evol Proc. 2020.

Nome: Broken Modularization
Descrição: This smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions
Importância: LOW
Impacto na Qualidade: Modularity, Maintainability
Ação de intervenção: PLANNED
Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.

Nome: Cyclically Dependent Modularization
Descrição: This smell arises when two or more abstractions depend on each other directly or indirectly
Importância: HIGH
Impacto na Qualidade: Modularity
Ação de intervenção: NORMAL
Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.

Nome: Insufficient Modularization
Descrição: This smell arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both
Importância: NORMAL
Impacto na Qualidade: Modularity, Maintainability
Ação de intervenção: PLANNED
Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.

Nome: Deep Hierarchy
Descrição: This smell arises when an inheritance hierarchy is 'excessively' deep
Importância: NORMAL
Impacto na Qualidade: Inheritance

Ação de intervenção: PLANNED
Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, “Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.

Nome: Deficient Encapsulation
Descrição: This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required
Importância: NORMAL
Impacto na Qualidade: Encapsulation
Ação de intervenção: PLANNED
Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, “Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.

Nome: Hub-like Modularization
Descrição: This smell arises when an abstraction has dependencies (both incoming and outgoing) with a large number of other abstractions
Importância: HIGH
Impacto na Qualidade: Maintainability, Coupling
Ação de intervenção: PLANNED
Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, “Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.

Nome: Multifaceted Abstraction
Descrição: This smell arises when an abstraction has more than one responsibility assigned to it
Importância: NORMAL
Impacto na Qualidade: Cohesion
Ação de intervenção: PLANNED
Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, “Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.

Nome: Wide Hierarchy
Descrição: This smell arises when an inheritance hierarchy is ’too’ wide indicating that intermediate types may be missing

Importância: NORMAL
Impacto na Qualidade: Inheritance
Ação de intervenção: PLANNED
Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, “Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.

E.3 Granularidade: Method

Nome: Long Method
Descrição: This smell occurs when a method is too long to understand
Importância: HIGH
Impacto na Qualidade: Understandability, Maintainability
Ação de intervenção: PLANNED
Referência: M. Fowler, “Refactoring: Improving the Design of Existing Code”. Addison-Wesley, 1999.

Nome: Long Parameter List
Descrição: This smell occurs when a method accepts a long list of parameters
Importância: LOW
Impacto na Qualidade: Maintainability
Ação de intervenção: LOW
Referência: M. Fowler, “Refactoring: Improving the Design of Existing Code”. Addison-Wesley, 1999.

Nome: Complex Method
Descrição: This smell occurs when a method has high cyclomatic complexity
Importância: HIGH
Impacto na Qualidade: Understandability, Maintainability, Complexity
Ação de intervenção: NORMAL
Referência: Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. “House of Cards: Code Smells in Open-Source C# Repositories". in ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017.

Nome: Bumpy Road
Descrição: This smell occurs when a method has high nested blocks
Importância: CRITICAL
Impacto na Qualidade: Understandability, Maintainability, Complexity
Ação de intervenção: IMMEDIATE
Referência: Tornhill, Adam. The Bumpy Road Code Smell: Measuring Code Complexity by its Shape and Distribution, 2023.

APÊNDICE F — LISTA DE CO-OCORRÊNCIAS DETECTADAS

O *DR-Tools Code Health* também detecta 20 tipos de co-ocorrências de *smells*, categorizados em *inter-components*, *type* e *method*. As co-ocorrências podem indicar certos antipadrões de *design* que se repetem no projeto e que, de certa forma, precisam ser eliminados ou monitorados.

F.1 Categoria: Inter-Components (4)

Descrição: Big and complex structures inter-components
Smells: God Class, Insufficient Modularization, Long Method, Complex Method
Impacto na Qualidade: Size, Complexity

Descrição: Complex structures inter-components
Smells: Insufficient Modularization, Complex Method
Impacto na Qualidade: Complexity

Descrição: Big structures inter-components
Smells: God Class, Long Method
Impacto na Qualidade: Size

Descrição: Critical!! Big and complex structures inter-components
Smells: God Class, Insufficient Modularization, Hub-like Modularization, Cyclically Dependent Modularization, Deep Hierarchy, Long Method, Complex Method, Bumpy Road
Impacto na Qualidade: Size, Complexity, Cohesion, Coupling

F.2 Categoria: Type (9)

Descrição: Big structures in types with modularization problems
Smells: God Class, Insufficient Modularization, Hub-like Modularization, Cyclically Dependent Modularization, Multifaceted Abstraction
Impacto na Qualidade: Size, Cohesion, Complexity, Coupling

Descrição: Critical!! Structures in types with too many smells
Smells: God Class, Insufficient Modularization, Hub-like Modularization, Cyclically Dependent Modularization, Multifaceted Abstraction, Deep Hierarchy
Impacto na Qualidade: Size, Cohesion, Complexity, Coupling

Descrição: Complex structures in types
Smells: God Class, Insufficient Modularization, Multifaceted Abstraction
Impacto na Qualidade: Size, Complexity

Descrição: Deep hierarchy, big and low cohesion structures in types
Smells: God Class, Insufficient Modularization, Deep Hierarchy, Multifaceted Abstraction
Impacto na Qualidade: Size, Complexity

Descrição: Complex and low cohesion structures in types
Smells: Insufficient Modularization, Multifaceted Abstraction
Impacto na Qualidade: Cohesion, Complexity

Descrição: Low cohesion and cyclic dependency structures in types
Smells: Insufficient Modularization, Cyclically Dependent Modularization
Impacto na Qualidade: Cohesion, Coupling

Descrição: Low cohesion and big structures in types
Smells: Insufficient Modularization, God Class
Impacto na Qualidade: Cohesion, Size

Descrição: Low cohesion and high coupling structures in types
Smells: Insufficient Modularization, Hub-like Modularization
Impacto na Qualidade: Cohesion, Coupling

Descrição: Low cohesion, high coupling, and big structures in types
Smells: God Class, Insufficient Modularization, Hub-like Modularization
Impacto na Qualidade: Cohesion, Coupling, Size

F.3 Categoria: Method (7)

Descrição: Big structures in methods
Smells: Long Method, Long Parameter List
Impacto na Qualidade: Size

Descrição: Complex structures in methods
Smells: Long Method, Bumpy Road, Complex Method
Impacto na Qualidade: Complexity

Descrição: Hard structures to understand in methods
Smells: Long Method, Bumpy Road
Impacto na Qualidade: Complexity, Size

Descrição: Structures difficult to maintain in methods
Smells: Complex Method, Long Method
Impacto na Qualidade: Complexity, Size

Descrição: High complexity structures in methods
Smells: Complex Method, Bumpy Road
Impacto na Qualidade: Complexity

Descrição: Critical!! Structures in methods with too many smells
Smells: Complex Method, Bumpy Road, Long Method, Long Parameter List
Impacto na Qualidade: Complexity, Size

*Descrição:*Several parameters and complex structures in methods
Smells: Complex Method, Long Parameter List
Impacto na Qualidade: Complexity, Size

APÊNDICE G — ARQUIVO DE CONFIGURAÇÃO *DEFAULT* USADO PELO *DR-TOOLS CODE HEALTH*

Conforme discutido neste trabalho, o *DR-Tools Code Health* implementa o conceito de *Profile*, permitindo a customização das informações relativas à métricas, pesos e *smells*.

Alguns *templates* dos arquivos de configuração são gerados dentro da pasta *./dr-tools/templates/* (opção *–init* via CLI).

O arquivo listado a seguir descreve a configuração padrão, usada pela ferramenta.

```
#
#           DRTools-Code-Health Config File
#
#  This file contains all information (metric thresholds, criteria) used
#
profile.name=Default
profile.description=Contains the default information, using metric
threshold values as reference

#
# -- Importance, used to smells ranking
#
importance.critical=4.0
importance.high=2.0
importance.normal=1.0
importance.low=0.5

#
# -- Intervention, used to smells ranking
#
intervention.immediate=4.0
intervention.planned=2.0
intervention.normal=1.0
intervention.low=0.5

#
# -- Quality Attributes, linked to smells and describing its impact
#    Parameters: CRITICAL, HIGH, NORMAL, and LOW
#
quality.attribute.cohesion=HIGH
quality.attribute.complexity=CRITICAL
quality.attribute.coupling=HIGH
quality.attribute.encapsulation=NORMAL
quality.attribute.inheritance=NORMAL
quality.attribute.maintainability=HIGH
quality.attribute.modularity=NORMAL
quality.attribute.testability=NORMAL
quality.attribute.understandability=HIGH

#
# -- Quality Impact, used to set weight on quality attributes
#
quality.impact.critical=3.0
quality.impact.high=2.0
quality.impact.normal=1.0
quality.impact.low=0.5

#
# -- Metric values, used in smell heuristics detection
#
#  NOC - Number of Types/Classes
#  Good: <= 11,0; Regular: between 11,0 and 28,0; Bad: > 28,0
metric.noc=28.0
#  NAC - Number of Abstract Types/Classes
#  without references
metric.nac=0.0
#  SLOC - Type Lines of Code
#  Bad: > 500,0
```

```
metric.sloc=500.0
#  NOM - Number of Methods
#  Good: <= 6,0; Regular: between 6,0 and 14,0; Bad: > 14,0
metric.nom=14.0
#  NPM - Number of Public Methods
#  Good: <= 10,0; Regular: between 10,0 and 20,0; Bad: > 20,0
metric.npm=20.0
#  WMC - Weighted Methods per Class
#  Good: <= 20,0; Regular: between 20,0 and 100,0; Bad: > 100,0
metric.wmc=100.0
#  DEP - Number of external types dependencies
#  Bad: > 20,0
metric.dep=20.0
#  I-DEP - Number of internal types dependencies
#  Bad: > 15,0
metric.i_dep=15.0
#  FAN-IN - Number of other types that depend on a given type
#  Bad: > 10,0
metric.fan_in=10.0
#  FAN-OUT - Number of other types referenced by a type
#  Bad: > 15,0
metric.fan_out=15.0
#  NOA - Number of Attributes/Fields
#  Good: <= 3,0; Regular: between 3,0 and 8,0; Bad: > 8,0
metric.noa=8.0
#  LCOM3 - Lack of Cohesion in Methods
#  Good: = 0,0; Regular: between 0,0 and 0,8; Bad: > 0,8
metric.lcom3=0.8
#  DIT - Deep in Inheritance Tree
#  Good: <= 2,0; Regular: between 2,0 and 4,0; Bad: > 4,0
metric.dit=4.0
#  CHILD - Number of Children
#  Bad: > 8,0
metric.child=8.0
#  NPA - Number of Public Attributes/Fields
#  Bad: > 1,0
metric.npa=1.0
#  CDEP - Number of cyclic dependencies (namespaces/types)
#  Bad: >= 1,0
metric.cdep=1.0
#  MLOC - Method Lines of Code
#  Good: <= 10,0; Regular: between 10,0 and 30,0; Bad: > 30,0
metric.mloc=30.0
#  CYCLO - Cyclomatic Complexity
#  Good: <= 2,0; Regular: between 2,0 and 4,0; Bad: > 4,0
metric.cyclo=4.0
#  CALLS - Number of Invocations
#  Bad: > 5,0
metric.calls=5.0
#  NBD - Nested Block Depth
#  Good: <= 1,0; Regular: between 1,0 and 3,0; Bad: > 3,0
metric.nbd=3.0
#  PARAM - Number of Parameters
#  Good: <= 2,0; Regular: between 2,0 and 4,0; Bad: > 4,0
metric.param=4.0

#
# -- List of Smells, including importance and intervention parameters
```

```
#     Importance parameters: CRITICAL, HIGH, NORMAL, LOW
#     Intervention parameters: IMMEDIATE, NORMAL, PLANNED, LOW
#  Too Large Package/Subsystem : Packages/Subsystems with a high number
of classes/packages indicate that they serve more than one specific
responsibility
#   Granularity: NAMESPACE
smell.namespace.too-large-package.importance=NORMAL
smell.namespace.too-large-package.intervention=PLANNED
#  Cyclic Dependency : This smell occurs when two or more
packages/subsystems depend on each other
#   Granularity: NAMESPACE
smell.namespace.cyclic-dependency.importance=HIGH
smell.namespace.cyclic-dependency.intervention=NORMAL
#  God Class : This smell is huge, complex, and has an extremely high
number of fields and methods
#   Granularity: TYPE
smell.type.god-class.importance=HIGH
smell.type.god-class.intervention=PLANNED
#  Broken Modularization : This smell arises when data and/or methods
that ideally should have been localized into a single abstraction are
separated and spread across multiple abstractions
#   Granularity: TYPE
smell.type.broken-modularization.importance=LOW
smell.type.broken-modularization.intervention=PLANNED
#  Insufficient Modularization : This smell arises when an abstraction
exists that has not been completely decomposed, and a further
decomposition could reduce its size, implementation complexity, or both
#   Granularity: TYPE
smell.type.insufficient-modularization.importance=NORMAL
smell.type.insufficient-modularization.intervention=PLANNED
#  Hub-like Modularization : This smell arises when an abstraction has
dependencies (both incoming and outgoing) with a large number of other
abstractions
#   Granularity: TYPE
smell.type.hub-like-modularization.importance=HIGH
smell.type.hub-like-modularization.intervention=PLANNED
#  Cyclically-dependent Modularization : This smell arises when two or
more abstractions depend on each other directly or indirectly
#   Granularity: TYPE
smell.type.cyclically-dependent-modularization.importance=HIGH
smell.type.cyclically-dependent-modularization.intervention=NORMAL
#  Multifaceted Abstraction : This smell arises when an abstraction has
more than one responsibility assigned to it
#   Granularity: TYPE
smell.type.multifaceted-abstraction.importance=NORMAL
smell.type.multifaceted-abstraction.intervention=PLANNED
#  Deep Hierarchy : This smell arises when an inheritance hierarchy is
'excessively' deep
#   Granularity: TYPE
smell.type.deep-hierarchy.importance=NORMAL
smell.type.deep-hierarchy.intervention=PLANNED
#  Wide Hierarchy : This smell arises when an inheritance hierarchy is
'too' wide indicating that intermediate types may be missing
#   Granularity: TYPE
smell.type.wide-hierarchy.importance=NORMAL
smell.type.wide-hierarchy.intervention=PLANNED
```

```
#  Deficient Encapsulation : This smell occurs when the declared
accessibility of one or more members of an abstraction is more permissive
than actually required
#   Granularity: TYPE
smell.type.deficient-encapsulation.importance=NORMAL
smell.type.deficient-encapsulation.intervention=PLANNED
#  Long Method : This smell occurs when a method is too long to
understand
#   Granularity: METHOD
smell.method.long-method.importance=HIGH
smell.method.long-method.intervention=PLANNED
#  Long Parameter List : This smell occurs when a method accepts a long
list of parameters
#   Granularity: METHOD
smell.method.long-parameter-list.importance=LOW
smell.method.long-parameter-list.intervention=LOW
#  Complex Method : This smell occurs when a method has high cyclomatic
complexity
#   Granularity: METHOD
smell.method.complex-method.importance=HIGH
smell.method.complex-method.intervention=NORMAL
#  Bumpy Road : This smell occurs when a method has high nested blocks
#   Granularity: METHOD
smell.method.bumpy-road.importance=CRITICAL
smell.method.bumpy-road.intervention=IMMEDIATE
```

APÊNDICE H — DADOS DO EXPERIMENTO 1: PESQUISA COM PROFISSIONAIS DA INDÚSTRIA

Neste Experimento (Seção 5.1), foram desenvolvidos algumas ferramentas complementares como planilhas e relatórios para apoiar as análises e investigações.

São eles:

- Planilha utilizada para compilação e análise dos dados
- Planilha com cálculos estatísticos dos dados submetidos
- Relatórios das devolutivas da análises dos projetos
- *Datasets* gerados pelo *DR-Tools Code Health* e submetidos pelos participantes

APÊNDICE I — QUESTIONÁRIO DESENVOLVIDO PARA O EXPERIMENTO 1

Para a coleta das informações referentes a percepção dos desenvolvedores, foi construído um questionário que contempla informações do participante, empresa, qualidade/*smells* e dados gerados pelo *DR-Tools Code Health* para posterior análise.

Neste Apêndice, é apresentado o questionário aplicado na pesquisa.

Ajudando os Desenvolvedores na Priorização

Este formulário é um instrumento de coleta de dados para a realização de um experimento da Tese de Doutorado de Guilherme Lacerda, sob supervisão do Prof. Marcelo Pimenta (UFRGS, Brasil) e Fabio Petrillo (ÉTS - Canadá).

Antes de continuar, solicitamos que leia atentamente o *Termo de Consentimento* da pesquisa.
Para aceitar participação, basta continuar o preenchimento do questionário.

* Indica uma pergunta obrigatória

1. 1. Nome *

2. 2. email *

Detalhando as suas informações

Nesta seção, você poderá detalhar as informações em relação a sua atuação, experiência e formação

3. 3. Qual seu cargo/posição atual? *

Marcar apenas uma oval.

- ◯ CTO/CIO
- ◯ Pessoa Arquiteta
- ◯ Pessoa Desenvolvedora Líder
- ◯ Pessoa Líder Técnica
- ◯ Pessoa Desenvolvedora Sênior
- ◯ Outro: _____

4. 4. Quanto tempo você tem de experiência na carreira em desenvolvimento de software? *

Marcar apenas uma oval.

- ◯ até 5 anos
- ◯ entre 5 e 10 anos
- ◯ entre 10 e 15 anos
- ◯ entre 15 e 20 anos
- ◯ mais de 20 anos

5. 5. Qual a sua titulação acadêmica? *
Considere a última concluída

Marcar apenas uma oval.

- ◯ Graduação
- ◯ Especialização/MBA
- ◯ Mestrado
- ◯ Doutorado

6. 6. Quanto tempo de atuação você tem na empresa atual? *
Considere a resposta somente em número e na unidade de anos

Detalhando as Informações da Empresa e do Projeto

Nesta seção, você poderá descrever maiores informações sobre a empresa e o projeto.

7. 7. Como você enquadra a sua empresa? *

Marcar apenas uma oval.

- ◯ Empresa tradicional (mais de 10 anos no mercado)
- ◯ Startup
- ◯ Empresa pública
- ◯ Outro: _____

8. 8. Qual o foco principal da empresa? *

Marcar apenas uma oval.

- ◯ Desenvolvimento de Produtos
- ◯ Fábrica de Software
- ◯ Consultoria
- ◯ Outsourcing
- ◯ Outro: _____

9. 9. Como você define o projeto que será analisado? *

Marcar apenas uma oval.

- ◯ Produto
- ◯ Biblioteca/Componente
- ◯ Solução interna da empresa
- ◯ Conjunto de serviços
- ◯ Outro: _____

10. 10. Atualmente este projeto é? *

Marque todas que se aplicam.

- [ ] Aplicação WEB
- [ ] Aplicação Desktop
- [ ] Serviços (SOA, SOAP, REST, entre outros)
- [ ] Aplicação Mobile
- [ ] Outro: _____

11. 11. Qual o tamanho do time que atua neste projeto? *

Marcar apenas uma oval.

- ◯ Pequena (até 3 pessoas)
- ◯ Média (4 a 6 pessoas)
- ◯ Grande (mais que 6 pessoas)

12. 12. Descreva resumidamente o propósito do projeto *
Você pode incluir neste resumo, além do objetivo do projeto outras informações que julgar relevante (tempo de vida, número de usuários, estilo arquitetural, entre outras)

_____

_____

_____

_____

_____

Qualidade e Smells

Nesta seção, vamos discutir mais a fundo aspectos de qualidade e sobre code smells. Caso você tenha alguma dúvida sobre code smells, você poderá consultar aqui.

13. 13. Quais técnicas/ferramentas vocês utilizam, pensando na qualidade do código? *

Marque todas que se aplicam.

- [ ] Não usamos qualquer técnica ou ferramenta
- [ ] Procuramos manter o design simples e fácil de evoluir
- [ ] Fazemos refatorações e usamos heurísticas de limpeza
- [ ] Usamos frequentemente a prática de revisão de código
- [ ] Utilizamos testes automatizados
- [ ] Utilizamos ferramentas de análise de código (Sonarqube, CodeScene, Sokrates, entre outras)

14. 14. Classifique os atributos de qualidade, considerando o que você mais valoriza para este projeto. *
A classificação vai de 1 (menos significativo) a 5 (mais significativo)

Marcar apenas uma oval por linha.

	1	2	3	4	5
Coesão	◯	◯	◯	◯	◯
Acoplamento	◯	◯	◯	◯	◯
Complexidade	◯	◯	◯	◯	◯
Encapsulamento	◯	◯	◯	◯	◯
Herança	◯	◯	◯	◯	◯
Manutenibilidade	◯	◯	◯	◯	◯
Modularidade	◯	◯	◯	◯	◯
Testabilidade	◯	◯	◯	◯	◯
Entendimento/Legibilidade	◯	◯	◯	◯	◯

15. 15. Você tem conhecimento em code smells? *

Marcar apenas uma oval.

- ◯ Não, nunca ouvi falar
- ◯ Sim, tenho conhecimento básico (apenas conceitual)
- ◯ Sim, conheço intermediário do tema (conheço e consigo identificar)
- ◯ Sim, conheço profundamente o tema (estudo, procuro ajudar o time nas detecções)

16. 16. Quais code smells você considera mais problemáticos? *
Pode listar mais de um code smell

17. 17. Liste as 15 classes consideradas mais problemáticas, pela sua percepção. *
*Você pode **listar o nome completo das classes** (pacote.Classe), escrevendo **uma classe por linha**.*
*Será considerada a **ordem de inclusão para a classe mais problemática**, ou seja, a primeira será mais problemática que a segunda e assim por diante.*

18. 18. Liste os 15 métodos considerados mais problemáticas, pela sua percepção. *
*Você pode **listar a assinatura completa do método -** pacote.Classe.metodo(Tipo parametro), escrevendo **um método por linha**.*
*Será considerada a **ordem de inclusão para o método mais problemático**, ou seja, a primeira será mais problemática que a segunda e assim por diante.*

19. 19. Em uma escala de 1 a 5, qual o seu nível de satisfação com a qualidade do código deste projeto? *

Marcar apenas uma oval.

	1	2	3	4	5	
pés	◯	◯	◯	◯	◯	ótima qualidade

20. 20. Quais estratégias você utilizou para priorizar estes elementos de código (classes, métodos) do projeto? *
Liste o que você considerou, desde alguma técnica específica e/ou ferramenta para priorização

Finalizando a Pesquisa...

Antes de finalizar a pesquisa, não esqueça de submeter o arquivo compactado da análise gerada pelo DR-Tools Code Health (*opção --analyze*).
Para mais informações sobre uso e configuração da ferramenta, consulte aqui (tools > drtools-code health).

21. 21. Você tem sugestões ou comentários adicionais sobre a qualidade do código que precisam ser considerados? *

22. 22. Submeta aqui o arquivo compactado da análise gerada pelo DR-Tools Code Health, usando a *opção --analyze*. *

Arquivos enviados:

APÊNDICE J — ORIENTAÇÃO PARA PARTICIPANTES

Como padrão de resposta para os participantes, tanto no envio das informações para preenchimento questionário quanto para a devolução dos dados analisados e compilados no relatório de análise, estruturou-se mensagens contemplando as informações pertinentes em cada situação.

Neste Apêndice, é apresentada ambas as respostas para os participantes.

Orientação para o Experimento 1: “Ajudando os Desenvolvedores na Priorização”

Resposta 1

Olá,

Primeiramente, muito obrigado por ter aceito participar deste experimento.
Com certeza, sua contribuição ajudará muito na evolução deste trabalho.

Etapas do experimento:

1. Baixe o DR-Tools Code Health na sua máquina e o descompacte em uma pasta.
    a. Para executá-lo, é necessário ter o JRE versão 11 ou superior.
    b. Você pode incluir o DR-Tools Code Health no path, se quiser usá-lo mais de uma vez
    c. Neste experimento, usaremos o modo CLI (command line interface), com a opção `--analyze`
    d. Não se preocupe! O DR-Tools Code Health não enviará dados sensíveis do projeto, como código fonte. As informações coletadas são referentes a métricas, smells e ranking dos elementos de código mais problemáticos e complexos para análise
2. Escolha o projeto que você usará como base para preenchimento do questionário.
    a. Ao escolher o projeto, tenha em mente quais são as partes que você considera mais problemática/crítica para manutenção
    b. Clone o projeto na sua máquina para poder usar o DR-Tools Code Health
3. Com o projeto já baixado na sua máquina, rode o seguinte comando:

    ```
    prompt> drtools-code-health <diretorio onde esta o projeto>
    --analyze /experimento
    Exemplo: prompt> drtools-code-health
    /dir/projeto-a-ser-analisado/src --analyze /experimento
    ```

    a. Ao executar este comando, o DR-Tools Code Health vai gerar arquivos no formato CSV e JSON na pasta especificada
    b. Compacte esta pasta para submeter no final do questionário
4. Preencha o questionário, considerando suas informações de experiência profissional, na empresa, sobre o projeto e qualidade
5. Na última etapa do questionário, será solicitado o envio deste arquivo compactado junto com o questionário preenchido

Após a análise dos resultados, eu entrarei em contato para apresentar os resultados da pesquisa.

Desde já agradeço sua disponibilidade!

Resposta 2

Olá

Agradeço muito a sua participação na pesquisa.
Para fecharmos, estou enviando um relatório com os dados analisados para que você confirme os resultados da análise.

Você verá que precisa confirmar se os elementos de código (classes/métodos) encontrados e priorizados são, de fato, relevantes para o seu trabalho no dia a dia.
Além disso, peço que responda as 3 perguntas que estão em cada seção de análise (métodos, classes, classes + métodos).
Por fim, fique a vontade para fazer comentários/críticas/sugestões a respeito dos dados apresentados.

Também, fique a vontade para comentar sobre o uso da ferramenta (modo interativo), caso você a tenha utilizado em análises nos seus projetos. Estes comentários podem ser incluídos no texto da tese?

Obrigado!

APÊNDICE K — TERMO DE CONSENTIMENTO

Como é de praxe em muitas pesquisas onde há a participação de pessoas, é essencial descrever qual o objetivo, procedimentos, questões relacionadas a confidencialidade e anonimato, incluindo potenciais riscos e benefícios.

Neste Apêndice, é apresentado o termo de consentimento utilizado na pesquisa.

Título da Pesquisa

DR-Tools Code Health: Uma Abordagem para Priorização de Smells para apoiar a Manutenção e Evolução de Software

Pesquisador Responsável

Guilherme Silva de Lacerda (guilhermeslacerda@gmail.com)
UFRGS - Universidade Federal do Rio Grande do Sul
Programa de Pós-Graduação em Computação
Doutorado em Ciência da Computação

Introdução

Este termo de consentimento tem o objetivo de informar sobre a pesquisa e obter o seu consentimento voluntário para participar.

Finalidade da Pesquisa:

O objetivo desta pesquisa é avaliar se o método de priorização proposto auxilia as pessoas desenvolvedoras na identificação dos elementos de código (classes, métodos) mais problemáticos. O termo problemático refere-se aos elementos de código identificados com diferentes smells, afetando vários atributos de qualidade e impactando na manutenibilidade.

Procedimentos:

- Você será convidado a responder a um questionário online.
- As perguntas abordarão tópicos relacionados ao desenvolvimento de software, mais especificamente, sobre um projeto em que você atuou diretamente e que tem conhecimento sobre as partes mais problemáticas e complexas de manutenção.
- Antes de finalizar o questionário, é necessário executar o DR-Tools Code Health, conforme instruções do pesquisador e compactar os arquivos gerados para posterior análise.
- O tempo estimado para conclusão do questionário é de aproximadamente 15 minutos.

Confidencialidade e Anonimato:

- Suas respostas serão mantidas estritamente confidenciais.
- Dados sensíveis como nome da empresa e produto não serão divulgados.
- Os dados coletados do repositório são estritamente relacionados a métricas de código e identificação de smells.
- Os resultados da pesquisa serão usados apenas para fins acadêmicos e podem ser apresentados em conferências ou publicações científicas.

Riscos e Benefícios:

- Não há riscos significativos associados à participação.
- O benefício potencial é contribuir para o avanço do conhecimento sobre o desenvolvimento de software, mais especificamente, na área de manutenção e evolução de software.

A participação nesta pesquisa é completamente voluntária.
Ao concordar com este termo, você confirma que:

- Entendeu as informações fornecidas.
- Teve a oportunidade de fazer perguntas e recebeu respostas satisfatórias.
- Concorda voluntariamente em participar da pesquisa.

Contato:

Guilherme Silva de Lacerda (guilhermeslacerda@gmail.com)
Janeiro/2024

APÊNDICE L — RELATÓRIO DE DEVOLUÇÃO DAS ANÁLISES

Após o recebimento do questionário e dos dados do projeto, estes dados são analisados e compilados para posterior verificação dos participantes.

Neste Apêndice, é apresentado o *template* usado para este relatório.

Relatório de Devolução das Análises do Experimento

Título da Pesquisa

DR-Tools Code Health: Uma Abordagem para Priorização de Smells para apoiar a Manutenção e Evolução de Software

Objetivo

Este relatório apresenta os resultados da análise do experimento realizado para avaliar se o método de priorização proposto auxilia as pessoas desenvolvedoras na identificação dos elementos de código (classes, métodos) mais problemáticos. O termo problemático refere-se aos elementos de código identificados com diferentes smells, afetando vários atributos de qualidade e impactando na manutenibilidade.

Pesquisador Responsável

Guilherme Silva de Lacerda (guilhermeslacerda@gmail.com)
UFRGS - Universidade Federal do Rio Grande do Sul
Programa de Pós-Graduação em Computação
Doutorado em Ciência da Computação

Dados do Participante e Projeto

Participante

Nome: <nome do participante>
email: <email do participante>
Cargo: <cargo>

Empresa e Projeto

Foco da empresa: <foco>
Definição do projeto analisado: <definição do projeto>
Categoria do projeto: <categoria do projeto>
Tamanho do time: <foco>
Resumo do propósito do projeto: <resumo>

Resultado da Análise

Os resultados da análise foram divididos em 3 grupos: métricas, code smells e priorização.
Em cada grupo são exibidas algumas informações referentes a coleta e investigação dos dados bem como comentários por parte do pesquisador.
Importante: No grupo de priorização, a pessoa respondente deve preencher algumas informações de sua percepção em relação ao resultado apresentado pela pesquisa.

Métricas

Resumo
Total de pacotes: <valor>
Total de classes: <valor> (classes/pacotes: <valor> com mediana de <valor> e desvio padrão de <valor>)
Total de SLOC: <valor> (SLOC/classes: <valor> com mediana de <valor> e desvio padrão de <valor>)
Total de métodos: <valor> (métodos/classes: <valor> com mediana de <valor> e desvio padrão de <valor>)
Total de CYCLO: <valor> (CYCLO/classes: <valor>)

Estatísticas

métrica	1stQ	3rdQ	avg	median	min	max	max-min	stddev	u-fence	threshold
SLOC										
NOM										
NPM										
WMC										
DEP										
I-DEP										
FAN-IN										
FAN-OUT										
NOA										
LCOM3										
DIT										
CHILD										
NPA										
CDEP										
MLOC										
CYCLO										
CALLS										
NBD										
PARAM										

Comentários: <comentários feito pelo pesquisador>

Smells

Resumo

Total de pacotes com smells: <valor> de <total> (0.0 %)
Total de pacotes com mais de smell: <valor> de <total detectados> (0.0 % dos detectados ou 0.0% do total de pacotes)
Total de classes com smells: <valor de <total> (0.0%)
Total de classes com mais de smell: <valor> de <total detectados> (0.0 % dos detectados ou 0.0% do total de classes)
Total de métodos com smells: <valor de <total> (0.0%)
Total de métodos com mais de smell: <valor> de <total detectados> (0.0 % dos detectados ou 0.0% do total de métodos)

Quantidade de Smells detectados por granularidade

Granularidade	Smell	Instâncias Detectadas
Namespace	Too Large Package/Subsystem	valor (0.0 %)
	Cyclic Dependency	valor (0.0 %)
Type	Insufficient Modularization	valor (0.0 %)
	Cyclically-dependent Modularization	valor (0.0 %)
	Multifaceted Abstraction	valor (0.0 %)
	God Class	valor (0.0 %)
	Deep Hierarchy	valor (0.0 %)
	Hub-like Modularization	valor (0.0 %)
	Deficient Encapsulation	valor (0.0 %)
	Broken Modularization	valor (0.0 %)
	Wide Hierarchy	valor (0.0 %)
Method	Complex Method	valor (0.0 %)
	Long Method	valor (0.0 %)
	Bumpy Road	valor (0.0 %)
	Long Parameter List	valor (0.0 %)

Comentários: <comentários feito pelo pesquisador>

Ranking dos Elementos de Código

Métodos
A seguir, você deve escolher SIM ou NÃO, conforme os métodos obtidos pela priorização do DR-Tools Code Health.
SIM=Faz sentido estar nesta lista dos elementos priorizados
NÃO= Não faz sentido estar nesta lista dos elementos priorizados

#	Pessoa Desenvolvedora	DR-Tools Code Health	Confirma?
1			Se...
2			Se...
3			Se...
4			Se...
5			Se...
6			Se...
7			Se...
8			Se...
9			Se...
10			Se...
11			Se...
12			Se...
13			Se...
14			Se...
15			Se...

Comentários: <comentários feito pelo pesquisador>

Pergunta 1: O resultado apresentado pelo DR-Tools Code Health mudou a sua percepção sobre os métodos mais problemáticos e difíceis de manter?
Sem Resposta

Classes

A seguir, você deve escolher SIM ou NÃO, conforme as classes obtidas pela priorização do DR-Tools Code Health.

SIM=Faz sentido estar nesta lista dos elementos priorizados

NÃO= Não faz sentido estar nesta lista dos elementos priorizados

#	Pessoa Desenvolvedora	DR-Tools Code Health	Confirma?
1			Se...
2			Se...
3			Se...
4			Se...
5			Se...
6			Se...
7			Se...
8			Se...
9			Se...
10			Se...
11			Se...
12			Se...
13			Se...
14			Se...
15			Se...

Comentários: <comentários feito pelo pesquisador>

Pergunta 2: O resultado apresentado pelo DR-Tools Code Health mudou a sua percepção sobre as classes mais problemáticas e difíceis de manter?

Sem Resposta

Classes + Métodos

Uma outra forma de visualizar os elementos de código mais críticos é observar os problemas do nível de classe juntamente com os problemas do nível de métodos. O DR-Tools faz esta composição, conforme lista apresentada a seguir.

SIM=Faz sentido estar nesta lista dos elementos priorizados

NÃO= Não faz sentido estar nesta lista dos elementos priorizados

#	Pessoa Desenvolvedora	DR-Tools Code Health	Confirma?
1			Se...
2			Se...
3			Se...
4			Se...
5			Se...
6			Se...
7			Se...
8			Se...
9			Se...
10			Se...
11			Se...
12			Se...
13			Se...
14			Se...
15			Se...

Comentários: <comentários feito pelo pesquisador>

Pergunta 3: Na sua percepção, o resultado apresentado pelo DR-Tools Code Health compondo classes e métodos seria um melhor preditor para priorização dos problemas do que o apresentado na página anterior (somente classes)?

Sem Resposta

Parâmetros e Configurações utilizadas no DR-Tools Code Health para o Experimento

Lista de Smells

Granularidade*: *Namespace

- *Nome*: ***Too Large Package***
  - *Descrição*: *Packages/Subsystems with a high number of classes/packages indicate that they serve more than one specific responsibility*
  - *Importância*: *NORMAL*
  - *Impacto na Qualidade*: *Understandability, Modularity*
  - *Ação de intervenção*: *PLANNED*
  - *Referência*: *M. Lippert, S. Roock, "Refactoring in Large Software Projects: Performing Complex Restructurings Successfully". John Wiley and Sons, 2006.*
- *Nome*: ***Cyclic Dependency***
  - *Descrição*: *This smell occurs when two or more packages/subsystems depend on each other*
  - *Importância*: *HIGH*
  - *Impacto na Qualidade*: *Coupling, Modularity*
  - *Ação de intervenção*: *NORMAL*
  - *Referência*: *M. Lippert, S. Roock, "Refactoring in Large Software Projects: Performing Complex Restructurings Successfully". John Wiley and Sons, 2006.*

Granularidade*: *Type

- *Nome*: ***God Class***
  - *Descrição*: *This smell is huge, complex, and has an extremely high number of fields and methods*
  - *Importância*: *HIGH*
  - *Impacto na Qualidade*: *Understandability, Maintainability*
  - *Ação de intervenção*: *PLANNED*
  - *Referência*: *Bafandeh Mayvan B, Rasoolzadegan A, Javan Jafari A. Bad smell detection using quality metrics and refactoring opportunities. J Softw Evol Proc. 2020.*
- *Nome*: ***Broken Modularization***
  - *Descrição*: *This smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions*
  - *Importância*: *LOW*
  - *Impacto na Qualidade*: *Modularity, Maintainability*
  - *Ação de intervenção*: *PLANNED*
  - *Referência*: *G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.*
- *Nome*: ***Cyclically Dependent Modularization***

    - *Descrição: This smell arises when two or more abstractions depend on each other directly or indirectly*
    - *Importância: HIGH*
    - *Impacto na Qualidade: Modularity*
    - *Ação de intervenção: NORMAL*
    - *Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.*
- *Nome: **Insufficient Modularization***
    - *Descrição: This smell arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both*
    - *Importância: NORMAL*
    - *Impacto na Qualidade: Modularity, Maintainability*
    - *Ação de intervenção: PLANNED*
    - *Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.*
- *Nome: **Deep Hierarchy***
    - *Descrição: This smell arises when an inheritance hierarchy is 'excessively' deep*
    - *Importância: NORMAL*
    - *Impacto na Qualidade: Inheritance*
    - *Ação de intervenção: PLANNED*
    - *Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.*
- *Nome: **Deficient Encapsulation***
    - *Descrição: This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required*
    - *Importância: NORMAL*
    - *Impacto na Qualidade: Encapsulation*
    - *Ação de intervenção: PLANNED*
    - *Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.*
- *Nome: **Hub-like Modularization***
    - *Descrição: This smell arises when an abstraction has dependencies (both incoming and outgoing) with a large number of other abstractions*
    - *Importância: HIGH*
    - *Impacto na Qualidade: Maintainability, Coupling*
    - *Ação de intervenção: PLANNED*
    - *Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.*
- *Nome: **Multifaceted Abstraction***
    - *Descrição: This smell arises when an abstraction has more than one responsibility assigned to it*
    - *Importância: NORMAL*
    - *Impacto na Qualidade: Cohesion*

    - *Ação de intervenção: PLANNED*
    - *Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.*
- *Nome: **Wide Hierarchy***
    - *Descrição: This smell arises when an inheritance hierarchy is 'too' wide indicating that intermediate types may be missing*
    - *Importância: NORMAL*
    - *Impacto na Qualidade: Inheritance*
    - *Ação de intervenção: PLANNED*
    - *Referência: G. Suryanarayana, G. Samarthyam, T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt Morgan Kaufmann, 2014.*

Granularidade: Method

- *Nome: **Long Method***
    - *Descrição: This smell occurs when a method is too long to understand*
    - *Importância: HIGH*
    - *Impacto na Qualidade: Understandability, Maintainability*
    - *Ação de intervenção: PLANNED*
    - *Referência: M. Fowler, "Refactoring: Improving the Design of Existing Code". Addison-Wesley, 1999.*
- *Nome: **Long Parameter List***
    - *Descrição: This smell occurs when a method accepts a long list of parameters*
    - *Importância: LOW*
    - *Impacto na Qualidade: Maintainability*
    - *Ação de intervenção: LOW*
    - *Referência: M. Fowler, "Refactoring: Improving the Design of Existing Code". Addison-Wesley, 1999.*
- *Nome: **Complex Method***
    - *Descrição: This smell occurs when a method has high cyclomatic complexity*
    - *Importância: HIGH*
    - *Impacto na Qualidade: Understandability, Maintainability, Complexity*
    - *Ação de intervenção: NORMAL*
    - *Referência: Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. * "House of Cards: Code Smells in Open-Source C# Repositories". * in ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017.*
- *Nome: **Bumpy Road***
    - *Descrição: This smell occurs when a method has high nested blocks*
    - *Importância: CRITICAL*
    - *Impacto na Qualidade: Understandability, Maintainability, Complexity*
    - *Ação de intervenção: IMMEDIATE*
    - *Referência: Tornhill, Adam. The Bumpy Road Code Smell: Measuring Code Complexity by its Shape and Distribution, 2023.*

Critérios usados pelo modelo de priorização

Importance: Peso dado pelo desenvolvedor a um smell em relação a sua importância (*default=1*)

Classificação	Peso
Critical	4.0
High	2.0
Normal	1.0
Low	0.5

Quality Impact: Peso dado pelo desenvolvedor para o impacto em dado atributo de qualidade (*default=1*). Os atributos de qualidade estão parametrizados da seguinte forma:

Atributo	Classificação
Cohesion	HIGH (2.0)
Complexity	CRITICAL (3.0)
Coupling	HIGH (2.0)
Encapsulation	NORMAL (1.0)
Inheritance	NORMAL (1.0)
Maintainability	HIGH (2.0)
Modularity	NORMAL (1.0)
Testability	NORMAL (1.0)
Understandability	HIGH (2.0)

Intervention: Peso dado pelo desenvolvedor em relação a intervenção ao tratar um dado smell (*default=1*)

Classificação	Peso
immediate	4.0
planned	2.0
Normal	1.0
Low	0.5

Thresholds para as Métricas

Os thresholds usados nas métricas foram definidos com base na literatura de métricas. Os valores estão relacionados a seguir:

Métrica	Sigla	Threshold
NOC	Number of Types/Classes	28
NAC	Number of Abstract Types/Classes	0
SLOC	Type Lines of Code	500
NOM	Number of Methods	14
NPM	Number of Public Methods	20
WMC	Weighted Methods per Class	100
DEP	Number of external type dependencies	20
I-DEP	Number of internal type dependencies	15
FAN-IN	Number of other types that depend on a given type	10
FAN-OUT	Number of other types referenced by a type	15
NOA	Number of Attributes/Fields	8
LCOM3	Lack of Cohesion in Methods	0.8
DIT	Deep in Inheritance Tree	4
CHILD	Number of Children	8
NPA	Number of Public Attributes/Fields	1
CDEP	Number of Cyclic Dependencies (namespaces/types)	1
MLOC	Method Lines of Code	30
CYCLO	Cyclomatic Complexity	4
CALLS	Number of Invocations	5
NBD	Nested Block Depth	3
PARAM	Number of Parameters	4

Referências para Métricas

Danijel Radjenović, Marjan Heričko, Richard Torkar, Aleš Živkovič, Software fault prediction metrics: A systematic literature review, Information and Software Technology, Volume 55, Issue 8, 2013

Mariza A.S. Bigonha, Kecia Ferreira, Priscila Souza, Bruno Sousa, Marcela Januário, Daniele Lima, The usefulness of software metric thresholds for detection of bad smells and fault prediction, Information and Software Technology, Volume 115, 2019

T. Filo, M. Bigonha, and K. Ferreira, A catalogue of thresholds for object-oriented software metrics, in Advances and Trends in Software Engineering, 2015 The First International Conference on. IARIA, 2015

Kecia A.M. Ferreira, Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes, Heitor C. Almeida, Identifying thresholds for object-oriented software metrics, Journal of Systems and Software, Volume 85, Issue 2, 2012

Rangasamy, R.Selvarani & Nair, T.R. & Ramachandran, Muthu & Prasad, Kamakshi. Software Metrics Evaluation Based on Entropy. Handbook of Research on Software Engineering and Productivity Technologies: Implications of Globalization, 2010

Robert C. Martin and Micah Martin. 2006. Agile Principles, Patterns, and Practices in C# (Robert C. Martin). Prentice Hall PTR, Upper Saddle River, NJ, USA

P. Oliveira, M. T. Valente and F. P. Lima, Extracting relative thresholds for source code metrics, 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), Antwerp, 2014

G. Vale, A benchmark-based method to derive metric thresholds, Master's Dissertation, UFMG, 2015

E. Lima, A. Resende, T. Lethbridge, The Uncomfortable Discrepancies of Software Metric Thresholds and Reference Values in Literature, ICSEA 2016 : The Eleventh International Conference on Software Engineering Advances, 2016

Checkstyle - Size Violations. https://checkstyle.sourceforge.io/config_sizes.html, 2019

APÊNDICE M — DADOS DO EXPERIMENTO 2: PRIORIZAÇÃO E IMPACTO NA QUALIDADE DOS *SMELLS* EM PROJETOS *OPEN-SOURCE* DURANTE A EVOLUÇÃO DE SOFTWARE

Para o experimento descrito na Seção 5.2, foram criadas algumas planilhas de apoio para coleta, organização e análise dos dados para os projetos *JetUML, Apache Ant, Apache Storm, SpotBugs* e *Apache Cassandra*.

Também está disponível o *dataset* dos dados coletados com o *DR-Tools Code Health* e com o *RefactoringMiner*.

Todos estes artefatos podem ser acessados nesta pasta do *Google Drive*.

APÊNDICE N — DR-TOOLS SMELL CHURN - FERRAMENTA PARA DETECÇÃO DE SMELL CHURN

N.1 Introdução

Durante o desenvolvimento do Experimento 2 (Seção 5.2), surgiu a necessidade de investigar como, quando e quais *smells* são inseridos e/ou removidos ao se avaliar duas versões.

Portanto, foi criado o *DR-Tools Smell Churn*, uma ferramenta que implementa um método de avaliação e comparação de *smells* baseado nos resultados gerados pelo *DR-Tools Code Health*. A motivação relacionada à criação desta ferramenta se dá pelo fato de não encontrar, pelo menos até a edição deste trabalho, alguma ferramenta disponível que implemente tal métrica. Assim, foi aproveitado todo o conhecimento e dados abertos fornecidos pelo *DR-Tools Code Health* para realizar sua implementação.

N.2 Inspiração

A base de inspiração para criação deste método para avaliar as inserções/remoções dos *smells* é um conceito já bastante difundido e conhecido na Engenharia de Software: o *code churn* (MUNSON; ELBAUM, 1998).

O *code churn* é um conceito utilizado para descrever a taxa de modificação ou *churn rate* de um código ao longo do tempo. Assim, o *code churn* mede a quantidade de alterações realizadas em um determinado conjunto de arquivos ou linhas de código durante um período específico, como uma iteração de desenvolvimento, um ciclo de compilação ou o desenvolvimento de uma funcionalidade específica.

Esse conceito é fundamental para entender a dinâmica de desenvolvimento e manutenção de software (FARAGÓ; HEGEDŰS; FERENC, 2015b). Ele pode ser calculado de diferentes formas, incluindo a contagem de adições, remoções e alterações de linhas de código, o número de *commits* ou revisões realizadas em um determinado período, ou até mesmo a quantidade de vezes que um arquivo foi alterado. O *code churn* fornece *insights* valiosos sobre a atividade de desenvolvimento, a estabilidade do código e pode ser usado para identificar áreas críticas que requerem maior atenção ou *refactorings*.

Existem algumas ferramentas que implementam o *code churn*, com destaque para

o *Code Scene*[1], *Code Maat*[2], *Sokrates*[3] e o *PyDriller*[4].

N.3 Método

O método implementado baseia-se na teoria dos conjuntos[5]. Dado 2 versões de um mesmo projeto, uma versão anterior e a versão atual (v_1 e v_2, respectivamente) que pretende-se analisar dentro de uma mesma granularidade.
A primeira ação é computar o *smell churn* de v_1 (Algoritmo 6), incrementando 1 para cada *smell* detectado nos elementos de código de v_1.

Após, tem-se 4 situações distintas: Considerando todos os elementos de código da v_2 (Algoritmo 7):
Regra 1: se os elementos de código de v_1 e v_2 forem iguais e tiverem os mesmos *smells*, passa-se para o próximo elemento de v_2
Regra 2: se os elementos de código de v_2 não estiverem em v_1 , incrementa-se 1 nos adicionados para os elementos de v_2
Regra 3: se os elementos de código de v_1 e v_2 forem iguais mas com *smells* diferentes

O que for complemento relativo (v_1 -> v_2), adiciona 1 nos deletados dos elementos de código

O que for complemento relativo (v_2 -> v_1), incrementa 1 nos adicionados dos elementos de código
Se o elemento de código de v_2 não estiver em v_1, adiciona 1 nos deletados dos elementos de código

N.4 Ferramenta

Seguindo os mesmos princípios de desenvolvimento adotados no *DR-Tools Suite*, o *DR-Tools Smell Churn* é uma ferramenta de uso simples, que recebe como entrada dois arquivos gerados a partir de uma análise temporal realizada com o *DR-Tools Code Health* combinado com o *git* para retornar a versões anteriores.

A Figura N.1 apresenta as opções de uso da ferramenta.

[1]<https://codescene.io/>
[2]<https://github.com/adamtornhill/code-maat/>
[3]<https://www.sokrates.dev/>
[4]<https://pydriller.readthedocs.io/en/latest/>
[5]<https://en.wikipedia.org/wiki/Set_theory>

Algoritmo 6: Realiza a computação do *Smell Churn* de v_1

input : *Set* dos elementos de código de v_1, com os respectivos *smells*, de tamanho n; *Set* de *smells* de um dado elemento de código, de tamanho m; *Map* que conterá os *smell churns* em $SmellChurn$

output: Computação do *smell churn* em $SmellChurn$

$Elements_{v1} \leftarrow$ `GetElements` (v_1) ;
for $i \leftarrow 1$ **to** n **do**
  $Smell_{Element_{v1}} \leftarrow$ `GetSmellsOf` $(Elements_{v1}\,[i])$;
  for $j \leftarrow 1$ **to** m **do**
    $SmellChurn \leftarrow SmellChurn\ .added(Smell_{Element_{v1}}\,[j])$;
  end
end

Figura N.1 – Opções de uso do *DR-Tools Smell Churn*

```
D:\ProgramFiles\cmder
λ smell-churn

:: DRTools-Smell-Churn :: Calculates the SMELL CHURN between two versions generated by DR-Tools Code Health
Usage: smell-churn <output> <file-previous-version> <file-actual-version>


Output:
        -console  List ALL information in console format (including code elements and smells)
        -csv      List smell churn in CSV format


D:\ProgramFiles\cmder
λ
```

Fonte: O Autor

É possível, a partir dos arquivos gerados em formato CSV pelo *DR-Tools Code Health*, analisar o *smell churn*. considerando os *smells* e granularidades atualmente passíveis de detecção pelo *DR-Tools Code Health*, é possível gerar a saída em formato console ou CSV, consolidando os resultados de duas versões.

Na Figura N.2, é exibida as duas saídas de processamento do *DR-Tools Code Churn* (console e CSV).

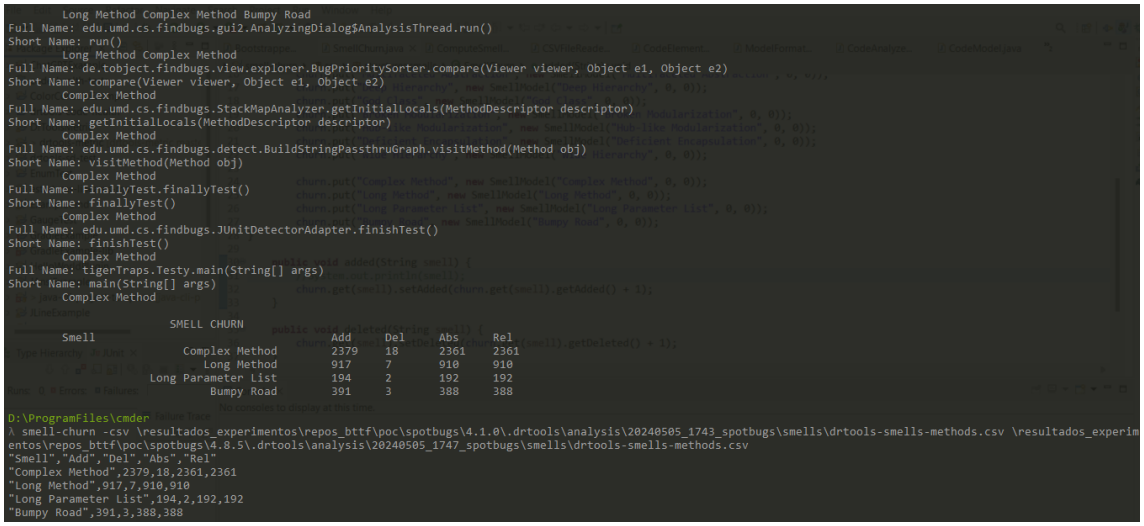
Algoritmo 7: Realiza a computação do *Smell Churn* de v_2

input : *Set* dos elementos de código de v_1, com os respectivos *smells*, de tamanho n; *Set* de *smells* de um dado elemento de código (v_1), de tamanho m; *Set* dos elementos de código de v_2, com os respectivos *smells*, de tamanho o; *Set* de *smells* de um dado elemento de código (v_2), de tamanho p; *Map* que conterá os *smell churns* em $SmellChurn$

output: Computação do *smell churn* em $SmellChurn$

```
Elements_v1 ← GetElements (v_1) ;
Elements_v2 ← GetElements (v_2) ;
for i ← 1 to o do
    if Elements_v1 [i].key() == Elements_v2 [i].key() and Elements_v2
     [i].value() == Elements_v1 [i].key() then
        continue;
    end
    if Elements_v1 [i].key() != Elements_v2 [i].key() then
        Smell_Element_v2 ← GetSmellsOf (Elements_v2 [i]) ;
        for j ← 1 to p do
            SmellChurn ← SmellChurn .added(Smell_Element_v2 [j]);
        end
        continue;
    end
    if Elements_v1 [i].key() == Elements_v2 [i].key() and Elements_v2
     [i].value() != Elements_v1 [i].key() then
        Smell_Element_v1 ← GetComplementarSetOfSmells
         (Elements_v2 [i]) ;
        for j ← 1 to m do
            SmellChurn ← SmellChurn .deleted(Smell_Element_v2 [j]);
        end
        Smell_Element_v2 ← GetComplementarSetOfSmells
         (Elements_v1 [i]) ;
        for j ← 1 to p do
            SmellChurn ← SmellChurn .added(Smell_Element_v2 [j]);
        end
    end
end
for i ← 1 to n do
    if Elements_v2 [i].key() != Elements_v1 [i].key() then
        Smell_Element_v1 ← GetSmellsOf (Elements_v1 [i]) ;
        for j ← 1 to m do
            SmellChurn ← SmellChurn .deleted(Smell_Element_v1 [j]);
        end
        continue;
    end
end
```

Figura N.2 – Exemplos de saídas geradas pelo *DR-Tools Smell Churn*



Fonte: O Autor

APÊNDICE O — FERRAMENTAS E COMANDOS UTILIZADOS NO EXPERIMENTO 2

Para desenvolvimento do Experimento 2 (Seção 5.2), foram utilizadas algumas ferramentas:

- *DR-Tools Code Health*
- Controle de Versão *Git*
- *RefactoringMiner*
- *DR-Tools Smell Churn*

Neste Apêndice, é listado os comandos executados para coleta dos dados com estas ferramentas.

Comandos referentes ao controle de versão (Git)

- **Clonar um repositório de um projeto, use o seguinte comando:**

git clone <url do projeto no git>
Exemplo:

```
prompt> git clone https://github.com/prmr/JetUML.git
```

- **Listar todas as versões (tags) de um projeto**

git tag
Exemplo:

```
prompt> git tag
```

- **Contar todos os commits de uma dada versão**

git rev-list –count <versão>
Exemplo:

```
prompt> git rev-list --count 1.8.1
```

Observação: caso se queira ver somente os commits dentro daquela versão, é necessário subtrair o número de commits da versão atual e diminuir da versão anterior.

- **Avançar ou retroceder no tempo, indo até uma dada versão**

git checkout <versão>
Exemplo:

```
prompt> git checkout 1.8.1
```

- **Avançar até a última versão**

git checkout <master|main>
Exemplo:

```
prompt> git checkout master
```

- **Listar as alterações (adição/remoção) realizadas em cada commit, listando autor, comentários e modificações realizadas**

Exemplo:

```
prompt> git log --pretty=format:"[%h] %an %ad %s" --date=short
   --numstat --after=2022-01-01
```

- **Listar os maiores contribuidores do projeto. Isso pode ser realizado em cada versão**

Exemplo:

```
prompt> git shortlog -s -n -e
```

- **Listar o commit inicial e sua respectiva tag**

Exemplo:

```
prompt> git show-ref --tags
```

- **Listar os arquivos modificados (code churn) entre duas versões**

Exemplo:

```
prompt> git diff --stat 3.1.0 3.0.2_preview2
```

- **Contar o número de ocorrências de uma palavra em um arquivo texto**

Exemplo:

```
prompt> grep -o -i ".java" \temp\arquivos.txt | wc -l
```

Usando o DR-Tools Code Health, em conjunto com o Git

- **Voltar em uma versão e rodando o drtools-code-health no modo CLI para gerar os dados referentes a métricas, smells e ranking**

Exemplo:

```
prompt> git checkout v2.0
prompt> drtools-code-health
   \resultados_experimentos\repos_bttf\JetUML\ --analyze
   \resultados_experimentos\repos_bttf\poc\JetUML\v2.0\
```

Usando o RefactoringMiner

- **Minerar os refactorings entre duas versões (tags)**

Exemplo:

```
prompt> RefactoringMiner -bt JetUML\ v0.1 v1.0
```

Usando o DR-Tools Smell Churn

- **Calcular o smell churn (adição/remoção de smells) entre duas versões. Para este cálculo, o drtools-smell-churn usa o dataset gerado pelo drtools-code-health (saída em CSV e console)**

Exemplo:

```
prompt> smell-churn -console
   \resultados_experimentos\repos_bttf\poc\jetuml\v3.5\.drtool
   s\analysis\20231012_1339_jetuml\smells\drtools-smells-metho
   ds.csv
   \resultados_experimentos\repos_bttf\poc\jetuml\v3.6\.drtool
   s\analysis\20231012_1339_jetuml\smells\drtools-smells-metho
   ds.csv
```