UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MARCELO GABRIEL DOS SANTOS VITORINO FECHINE

HOW DEVELOPERS DISCUSS CODE SMELLS DURING CODE REVIEW: A REPLICATION

CAMPINA GRANDE - PB
2024

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

How developers discuss Code Smells during Code Review: A replication

Marcelo Gabriel dos Santos Vitorino Fechine

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

João Arthur Brunet e Tiago Lima Massoni
(Orientadores)

Campina Grande, Paraíba, Brasil

F291h Fechine, Marcelo Gabriel dos Santos Vitorino.

How developers discuss Code Smells during Code Review: a replication / Marcelo Gabriel dos Santos Vitorino Fechine – Campina Grande, 2024.

54 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Tecnologia e Recursos Naturais, 2024.

"Orientação: Prof. Dr. João Arthur Brunet Monteiro, Prof. Dr. Tiago Lima Massoni."

Referências.

1. Software Engineering. 2. Code Review. 3. Code Smells. I. Monteiro, João Arthur Brunet. II. Massoni, Tiago Lima. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA ITAPUANA SOARES DIAS GONÇALVES CRB-15/093

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secretaria-copin@computacao.ufcg.edu.br / copin@copin.ufcg.edu.br

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

MARCELO GABRIEL DOS SANTOS VITORINO FECHINE

HOW DO DEVELOPERS DISCUSS CODE SMELLS DURING CODE REVIEW: A CASE STUDY

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 05/03/2024

Prof. Dr. JOÃO ARTHUR BRUNET MONTEIRO, Orientador, UFCG

Prof. Dr. TIAGO LIMA MASSONI, Orientador, UFCG

Profa. Dra. MELINA MONGIOVI BRITO LIRA, Examinadora Interna, UFCG

Prof. Dr. DIEGO ERNESTO ROSA PESSOA, Examinador Externo, IFPB

Documento assinado eletronicamente por **JOAO ARTHUR BRUNET MONTEIRO**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 06/03/2024, às 13:23, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **TIAGO LIMA MASSONI**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 06/03/2024, às 13:59, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **MELINA MONGIOVI CUNHA LIMA SABINO**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 08/03/2024, às 09:31, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **4254248** e o código CRC **C5C77A4B**.

Referência: Processo nº 23096.013314/2024-31 SEI nº 4254248

Resumo

Code smells são sintomas de possíveis problemas no código e que indicam a necessidade de uma refatoração. Uma das formas de detecção de smells é através da Code Review, prática importante no desenvolvimento de software que visa promover boas práticas de programação. Todavia, entre os estudos que conhecemos, apenas um investigou code smells durante reviews, mas não diretamente as discussões. Isso limita o entendimento do tema e aumenta o risco de viés, pois afeta a validade e generalização dos resultados. Visando mitigar essa falta, Xiaofeng Han et al. (2021) analisaram manualmente 19.146 comentários e extraíram 1.190 revisões relacionadas a smell. Foi encontrado que i) code smells não são comumentes encontrados nas revisões de código, ii) smells são causados por violações de convencções de código, iii) revisores usualmente fornecem feedbacks contrutivos e iv) desenvolvedores geramente seguem as recomendações e praticam as mudanças sugeridas no código. Apesar do estudo fornecer resultados relevantes, existe uma limitação na evidência empírica e na aplicabilidade atual, pois o estudo foi feito utilizando 1) projetos de uma mesma organização e 2) projetos que usam, essencialmente, a mesma tecnologia (python). Visando aumentar a confiabilidade científica e a extensão da aplicabilidade do estudo fizemos uma replicação sob um novo contexto. Para isso, 26 desenvolvedores foram envolvidos em um processo de análise manual em 18.850 comentários que extraiu 2.164 smell reviews dos projetos gRPC, Neovim e Keycloak, que utilizam as linguagens C++, Java e Vim. Através desse conjunto, classificações foram realizadas de forma a replicar o estudo do trabalho original. Para esse novo contexto, o estudo confirmou os resultados i), ii) e iv), com destaque para o resultado de 831 (70%) de causas de smell não fornecidas, o que fortalece o resultado do estudo anterior pela semelhança da frequência encontrada e indica a necessidade de outras abordagens para investigar o tipo da causa. Além disso, as discussões sobre a interpretação da ação dos revisores foi extendida e em 1.781 (82%) das ocorrências, o smell é capturado mas nenhuma ação de correção é realizada. Assim, esse estudo permite para os desenvolvedores: Aprender como as decisões de código são percebidas e como lidar com smells. E para os revisores: Aprender como melhorar a comunicação e construção de revisões através dos tipos de smell mais frequentes e como as suas sugestões são recebidas pelos desenvolvedores.

Abstract

Code smells are symptoms of possible problems in the code and indicate the need for refactoring. One of the ways to detect smells is through Code Review, an important practice in software development that aims to promote good programming practices. However, among the studies we know of, only one investigated code smells during reviews, but not directly the discussions. This limits the understanding of the topic and increases the risk of bias, as it affects the validity and generalizability of the results. Aiming to mitigate this lack, Xiaofeng Han et al. (2021) manually analyzed 19,146 comments and extracted 1,190 smell-related reviews. It was found that i) code smells are not commonly found in code reviews, ii) smells are caused by violations of code conventions, iii) reviewers usually provide constructive feedback and iv) developers generally follow recommendations and implement suggested changes to the code . Although the study provides relevant results, there is a limitation in the empirical evidence and current applicability, as the study was carried out using 1) projects from the same organization and 2) projects that essentially use the same technology (python). Aiming to increase the scientific reliability and extend the applicability of the study, we carried out a replication under a new context. To this end, 26 developers were involved in a manual analysis process of 18,850 comments that extracted 2,164 smell reviews from the gRPC, Neovim and Keycloak projects, which use the C++, Java and Vim languages. Through this set, classifications were carried out in order to replicate the study of the original work. For this new context, the study confirmed results i), ii) and iv), with emphasis on the result of 831 (70%) smell causes not provided, which strengthens the result of the previous study due to the similarity of frequency found and indicates the need for other approaches to investigate the type of cause. Furthermore, discussions about the interpretation of the reviewers' actions were extended and in 1,781 (82%) of the occurrences, the smell was captured but no corrective action was taken. Thus, this study allows developers to: Learn how code decisions are perceived and how to deal with smells. And for reviewers: Learn how to improve communication and construction of reviews through the most frequent types of smells and how your suggestions are received by developers.

Agradecimentos

Este trabalho não seria possível sem a ajuda de inúmeras pessoas. Gostaria de agradecer e dedicá-lo:

- Ao meu filho Lorenzo, pois o seu nascimento foi celebrado e vivido ao longo desses dois anos de mestrado, e me deu ainda mais forças para conquistar esse objetivo. Estendo a dedicatória, também, aos seus futuros irmãos;
- A minha esposa Sabrina, que vivenciou comigo mais uma conquista acadêmica. O que acontece desde os tempos de escola, mas dessa vez mais especial: compartilhou esse momento com a gestação e nascimento do nosso filho;
- A minha mãe Cristina, que há um ano, foi diagnosticada com câncer em seu estágio mais avançado e entre as suas cirurgias, quimioterapias e consultas que passamos juntos sempre me apoiou para que eu nunca desista de nenhum objetivo. Hoje, celebro a conclusão do mestrado simultâneamente a sua cura;
- Ao meu pai, minha irmã, avós, sogros e aos demais parentes;
- Aos meus especiais orientadores, João Arthur e Tiago Massoni;
- Aos professores com quem já tive experiências acadêmicas: Lívia Sampaio, Rohit Geyi, Cláudio Campelo, Joseana Fechine, Marília Marcy, Jens Andreas Faustich e Laurent Borgmann;
- Aos 26 colegas que contribuíram com o trabalho através das análises para a replicação, em especial, João Soares.
- Aos amigos: Caio Sanches, Gabriel Almeida e Thiago Moura;
- A todos os membros da banca avaliadora;
- A CAPES pelo auxílio financeiro;
- A todos que me ajudaram nessa trajetória e que ainda não foram citados.

Obrigado!

Contents

List of Figures

List of Tables

Lista de Códigos Fonte

Chapter 1

Introduction

1.1 Problem

Code smells are clues or patterns in source code that suggest the presence of design problems or potential bugs. These smells indicate areas of the code that may need refactor to improve the software's readability, maintainability, and efficiency. Identifying and correcting code smells contributes to the development of more robust and high-quality code [13] [33] [3] [26] [19].

Detecting code smells is important because these clues in the source code often point to design, readability, or efficiency problems that, if left untreated, can lead to bugs, maintenance difficulties, and decreased software quality. Identifying and correcting code smells contributes to the creation of cleaner, more sustainable, and resilient code, improving the overall quality of the system and facilitating future modifications. Some studies highlight the different types of code smells that can be detected in programs: security problems, performance flaws, duplicate code, large methods or classes, and complex logic, among others [21] [13] [22].

One of the ways to detect smells is through Code Review, an important practice in software development that aims to promote good programming practices. A review is a way of evaluating a program written by another developer by identifying potential problems, making suggestions for improvement, and providing feedback [37] [29]. Review helps ensure code consistency and quality, as well as promotes the exchange of knowledge and best practices between team members, contributing to more efficient and reliable software

development.

There is an advancement of automatic tools that help with code review, including static analysis tools, like PMD, SonarQube, and Designite [42] [30]. Automatic tools provide fast and systematic detection of smells, increasing efficiency and reducing the time needed to identify and fix code issues compared to manual analysis.

However, manual review is still widely used and it is possible to point out that reviewers have complete contextual information about the programs they evaluate, as they usually review codes from the projects in which they are inserted [27]. Also, some authors highlight that the context and program domain are important to detect smells, which is often considered in manual review [40] [44]. Additionally, that approach may avoid false positives, common in automatic review [38] [12].

Despite the benefits of the review, among the studies we know, only one investigated code smells during code review [34], and not directly the smell discussions but about the side effects of code review suggestions on code smell reduction in files, this limits the understanding of the topic and increases the risk of bias, as it affects validity and generalizability of the results. To mitigate this lack, Han, Xiaofeng et al. (2021) manually analyzed 19.146 comments and extracted 1.190 reviews related to smell and found relevant results: i) code smells are not commonly found in code reviews, ii) smells are caused by violations of code conventions, iii) reviewers usually provide constructive feedback and iv) generally developers follow the recommendations and make suggested changes to the code.

Despite these results, it suffers from some threats to external validity: 1) Change sets are from a single organization, namely OpenStack. Perhaps the technique is not suitable for other organizations. 2) The example dataset used was extracted from projects that mostly use only Python code. It is not clear whether the results apply to other programming languages. These threats mean that the study results may not be generalizable to other contexts.

To improve the robustness of the study, it is beneficial to diversify the sample, including projects from different organizations that use different technologies. Replication in more varied contexts helps to validate conclusions and provide more comprehensive insights into the applicability of findings in more diverse environments [16].

1.2 Solution

Aiming to increase the scientific reliability and extend the applicability of the study, we replicated the work of Han, Xiaofeng et al. (2021) in a new context. The replication follows the original study as closely as possible, except that we deliberately changed the context to better understand how broadly applicable the results were. According to Gomez's classification, Omar S et al. (2014) for replications, our study is a replication of changing populations/experiments.

The objective of the study is to understand how common it was for reviewers to identify code smells during code review, why the code smells were introduced, what actions they recommended for those smells, and how developers proceeded with those recommendations. To this end, the study sought to answer the following research questions, initially defined in the original study:

- **RQ1:** Which code smells are the most frequently identified by code reviewers?
- **RQ2:** What are the common causes for code smells that are identified during code reviews?
- **RQ3:** How do reviewers and developers treat the identified code smells?
  - **RQ3.1:** What actions do reviewers suggest to deal with the identified smells?
  - **RQ3.2:** What actions do developers take to resolve the identified smells?
  - **RQ3.3:** What is the relationship between the actions suggested by reviewers and those taken by developers?

To answer these questions, the list of terms that are related to code smells, used by the original work [Table 1.1], was also used to carry out our study classification, this list contains terms initially defined by Martin Fowler (1999) [13] and cited in the systematic review by M. Zhang et al. (2011) [46] and Tahir et al. (2020) [40]. The list includes general terms like "code smell", "bad smell", "bad pattern", "anti-pattern", and "technical debt". Based on those terms, the authors built a keyword set [Table 1.2], that was created from data processing that included removing stopwords, punctuation, and numbers. Also, was applied the

Table 1.1: Code smell terms used in classification. Table extracted of Han, Xiaofeng et al. (2021)

Code Smell Terms			
Accidental Complexity	Anti Singleton	Bad Naming	Blob Class
Circular Dependency	Coding by Exception	Complex Class	Complex Conditionals
Data Class	Data Clumps	Dead Code	Divergent Change
Duplicated Code	Error Hiding	Feature Envy	Functional Decomposition
God Class	God Method	Inappropriate Intimacy	Incomplete Library Class
ISP Violation	Large Class	Lazy Class	Long Method
Long Parameter List	Message Chain	Middle Man	Misplaced Class
Parallel Inheritance Hierarchies	Refused Bequest	Bad Naming	Shotgun Surgery
Similar Subclasses	Softcode	Spaghetti Code	Speculative Generality
Suboptimal Information Hiding	Swiss Army Knife	Temporary Field	Use Deprecated Components

identifier splitting rules and the stem of each term was obtained. That set was used to guide the keyword search process.

Initially, we ranked open source projects available on Github that have the highest average number of comments on pull requests 1.3. We've considered pull requests in any state (open, closed, aborted). Thus, we extract 104,321 review comments from the gRPC, Neovim and Keycloak projects. They were selected using the following criteria: 1) To use different programming languages: C++, Java, and Vim, respectively, and 2) To extract a sample that allows us to have a keyword smell filtered set quantity similar to the quantity of the original

Table 1.2: Code smell keywords used in mining.

Code Smell Keywords			
smell	smelly	anti	pattern
bad	technical	debt	accidental
complexity	complex	singleton	bad
naming	blob	circular	circularity
dependency	dependent	exception	complex
complexity	complex	complexity	conditional
condition	data class	clump	dead
death	unused	useless	divergent
divergence	duplicated	duplicate	duplication
clone	hiding	hide	envy
decompose	decomposition	god	brain
inappropriate	intimacy	incomplete	library
ISP	violate	violation	large
big	lazy	parameter list	long
chain	middle	misplace	misplaced
parallel	inheritance	obsession	refuse
refused	bequest	shotgun	surgery
similar	subclass	softcode	spaghetti
speculative	generality	suboptimal	hiding
hide	swiss	army	knife
temporary	temporal	deprecated	deprecate
component			

study. The details of the filtering processing can be found in Section 3.

Using those comments extracted in the previous step and the set of code smell keywords, we did a keyword search to find comments related to code smells. Thus, 19,149 comments that had at least one of the keywords were collected.

Next, a qualitative study involved 26 experienced developers, who in 13 pairs, independently and manually analyzed 18,850 comments filtered in the previous step. This quantity

Table 1.3: Github projects with highest review comments average

Projects	**PR's**	**Comments AVG**	**Date**
Neovim	**14729**	**3.68**	**13/09/23**
React	13860	3.55	12/09/23
gRPC	**23090**	**3.52**	**14/09/23**
Quarkus	19679	3.42	13/09/23
Distrobox	294	3.13	10/09/23
LazyGit	1310	2.66	10/09/23
TLDR	9521	2.51	12/09/23
DLP	2174	2.44	10/09/23
Backstage	14194	2.19	12/09/23
FD	582	2.15	10/09/23
Keycloak	**12600**	**1.77**	**12/09/23**

was selected for simplicity from the set of 19,149 to allow each pair to analyze a similar quantity of 1,450 comments. This study sought to understand the context of the comments to confirm previous keyword searches and remove false positives. Thus, 4,563 smell comments were extracted.

With this set, to confirm previous findings, two experienced developers from the last set, including the author of this work, carried out a new qualitative analysis, seeing the entire thread of comments and source code when needed, to simultaneously and manually analyze those results. Only two developers performed this step because of the time range available to finish the analysis and the higher availability demanded on simultaneous meetings. Thus, with 0,95 Cohen's Kappa coefficient of agreement, 3,702 smell comments were extracted in this step.

Additionally, as a result of a random sample selection, 96 smell comments were added to the set of comments eligible to be classified, with 95% C.I. and 3% E.M. This sampling followed the same steps as the rest, and it used the comments that were not extracted at the keywords search step.

In the end, the 3,798 review comments were mapped into groups of pull requests, including specific info like thread of comments, pull request url and source code change. This

process resulted in 2,164 smell reviews, and those were manually analyzed to answer the research questions. This step allows us to obtain observations and a deep understanding of the comments. The methodology of this work is explained in detail in Section 3, and the results of this study are presented in Section 4.

Understanding how code smells are discussed during code reviews is crucial, as it allows for the early identification of design and quality issues in the software, contributing to more robust code production. This enriches the state of the art by improving the understanding of effective practices in detecting and mitigating code smells.

Replication strengthens the knowledge base of the scientific community and contributes to the continued advancement of software engineering. Replicating experiments allows independent verification of conclusions, promoting the reliability and practical applicability of results in industry and academia.

1.3 Main Contributions

A replication was carried out on a sample of pull requests and obtained the following contributions:

- Dataset of 2,164 smell reviews from three different projects (gRPC, Keycloak, and Neovim).
- Replication that discovered the following hypotheses:
  - **Smells** are not commonly discussed in reviews (only 2,164 smell reviews were found from a set of 104,321 review comments).
  - Duplicated Code (32,9%) and Bad Naming (9%) are the most commonly smells discussed.
  - **Reviewers** usually capture the smell but don't recommend explicitly fixing it (82%).
  - **Developers** usually fix the code smells found (94%).
  - **Developers** usually agree with the reviewer suggestions (79%).

- A replication package[1] with the scripts to reproduce this study.

1.4 Outline

This document is structured as follows: Section 2 provides the necessary basis for understanding this work. Section 3 describes the methods used to replicate the original study. Section 4 presents the results of replication. Section 5 presents the threats to the replication. Section 6 provides an overview of related work. Section 7 concludes this work by outlining the main results and possible locations for future work.

[1] https://zenodo.org/records/10607935

Chapter 2

Backgroung

2.1 Code Smell

Martin Fowler (1999) explains that code smells are indications of problems in the code that can be solved through refactoring. Therefore, the author defines some refactoring suggestions if a developer comes across code smells, but emphasizes that his suggestions may be debatable depending on the developer's interpretation and the code being analyzed.

Some examples are:

- **Duplicated Code:** When identical or very similar pieces of code appear in multiple parts of the system, it can lead to difficult maintenance and inconsistencies.
- **Long Method:** Very long and complex methods can be difficult to understand, maintain, and test. They also tend to mix responsibilities, which undermines modularity.
- **Large Class:** Classes that have many responsibilities or many methods and attributes can become difficult to manage and understand.
- **Feature Envy:** Occurs when a method of a class accesses more methods and attributes of another class than of the class in which it is defined. This may indicate a poor distribution of responsibilities.
- **Dead Code:** Refers to portions of code that are no longer used or reachable within the application, cluttering the codebase and increasing maintenance overhead.

- **Switch Statements:** Excessive use of switch statements to make decisions can increase complexity and make code more difficult to extend and maintain.

- **Bad Naming**: Involves poorly chosen identifiers (variables, functions, etc.) that obscure the purpose and functionality of the code, hindering readability and comprehension.

- **Swiss Army Knife**: Refers to a class or module that tries to do too much, violating the Single Responsibility Principle (SRP), leading to high complexity and low cohesion in the codebase.

- **Circular Dependency**: Occurs when modules or classes circularly depend on each other, leading to tangled and brittle code.

M. Zhang et al. (2011) performed a systematic literature review to understand the types of code smells identified to date and summarize the main findings from the software engineering community. The article presents different definitions of smells and highlights the lack of a standard definition. This reflects the subjective and context-dependent nature of code smells. The authors propose a comprehensive classification of the different types of smell, grouping them into categories such as "Excessive Complexity", "Lack of Cohesion", and "Excessive Coupling", among others.

We consider, in this work, the list of code smell terms presented in Table 1.1, which includes the list made by Tahir et al. (2020), that contains smells defined by Fowler (1999) and M. Zhang et al. (2011). Also, we consider code smells as a subset of design problems, because code smells are often indicators of questionable practices at the code level, while design issues can encompass broader issues related to the overall structure, architecture, and approach of the system. Therefore, although all code smells are design problems, not every design problem is a code smell, as deeper problems may exist beyond the scope of issues identified by code smells.

In Source Code 2.1, written in Python, the functions `print_file_content` and `convert_csv_to_pdf` have a code duplication when reading the file (`with open(file_path, 'r') as file: csv_data = file.read()`). This can be considered a code smell because it violates the DRY (Don't Repeat Yourself)

principle. Reading the file is a common concern between the two methods, but they have different logic after reading. One approach to eliminate code duplication would be to extract the file reading logic into a separate method and call it in both methods.

Código Fonte 2.1: Reading a file with the code smell: Duplicated Code

```
def print_file_content(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
        # Print file content
        print(data)
    except FileNotFoundError:
        print(f"File not found: {file_path}")

def convert_csv_to_pdf(file_path):
    try:
        with open(file_path, 'r') as file:
            csv_data = file.read()
        # Convert CSV to PDF
        pdf_data = convert_csv_to_pdf_function(csv_data)
        # Save or print the PDF data (implementation details not shown)
        # ...
    except FileNotFoundError:
        print(f"File not found: {file_path}")

def convert_csv_to_pdf_function(csv_data):
    # Actual implementation to convert CSV to PDF (not shown for brevity)
    # ...
    return pdf_data
```

In Source Code 2.2, written in Python, in turn, there is no visible code smell because the design problem in the example given is more related to the overall architecture of the system and the ability to extend the code cleanly, as the problem presents the lack of flexibility to add new types of cars such as hybrids, steam engines, etc. If we want to add a new type of car, we need to modify the class `Car` existing, violating the Open/Closed Principle. A more extensible approach would be to use inheritance and create separate classes for different

types of cars.

Código Fonte 2.2: Classes with the design problem: Class hierarchy between type of cars

```
class Car:
    def __init__(self, make, model, year, is_electric):
        self.make = make
        self.model = model
        self.year = year
        self.is_electric = is_electric

    def start_engine(self):
        if self.is_electric:
            print("Starting electric engine...")
        else:
            print("Starting normal engine...")

normal_car = Car("Toyota", "Corolla", 2022, False)
electric_car = Car("Tesla", "Model S", 2022, True)
```

Code smells are not obvious syntax errors or bugs, but rather indicators of potential design issues that can affect the maintainability, extensibility, and readability of the code. These smells are perceived as clues to possible improvements in the software design. Resolving code smells often involves refactoring the code, which means restructuring existing code without changing its external behavior.

2.2 Code Review

Code review is an important practice in software development that aims to improve the quality and reliability of written code. It is a way of evaluating code written by another developer, identifying possible problems, and suggestions for improvement, and providing [37] [29] feedback.

MCR differs from traditional review (software inspection), initially defined in 1976 by Fagan, M. E. (1976) [11] as a manual code inspection structure with the aim of improving the quality of software projects. As it is based on more efficient and collaborative approaches, it is carried out through collaboration and automation tools, which allow developers to work

together more efficiently and effectively. Furthermore, modern review is also based on efficient static analysis methods, which allows you to detect problems in the code long before it is run [36].

With the popularization of distributed version control systems like Git and the advent of code hosting platforms like GitHub and Bitbucket, code review began to be integrated directly into the development workflow. These platforms offered a more efficient and collaborative way to perform reviews, allowing team members to review, comment, and collaborate on code asynchronously and in real-time.

Key concepts in modern code review include an emphasis on early detection of problems, the continuous integration of reviews into the development process, the automation of code reviews, and the promotion of a culture of constructive feedback. Furthermore, modern code review goes beyond bug detection, incorporating concerns about code readability, good programming practices, and knowledge sharing among team members [37][[14][15][29].

Some studies carried out a systematic review of code review where they explain that several tools can be used to allow it and that each software team has its particularities in the review process, which can be defined as follows: The developer creates the code and, then sends the modification to a versioning tool. Other team members review the code and provide feedback, and there may be iterations for adjustments. After approval and passing tests, the changes are merged into the main project code and eventually deployed to the production environment. The process aims to ensure quality, consistency, and effective collaboration. [Figure 2.1] [15][14]

2.3 Pull-based software development

Pull-based software development is a collaborative development model in which contributions are made through Pull Requests (PRs). In this context, developers create isolated copies of the source code in their branches, implement the desired changes, and later propose these changes through PRs. These requests serve as a formal mechanism for other team members to review, discuss, and eventually merge changes into the main [17] branch.

Contribution through PRs is crucial in this model as it provides a framework for collaborative reviews, detailed discussions, and adjustments before changes are incorporated. This

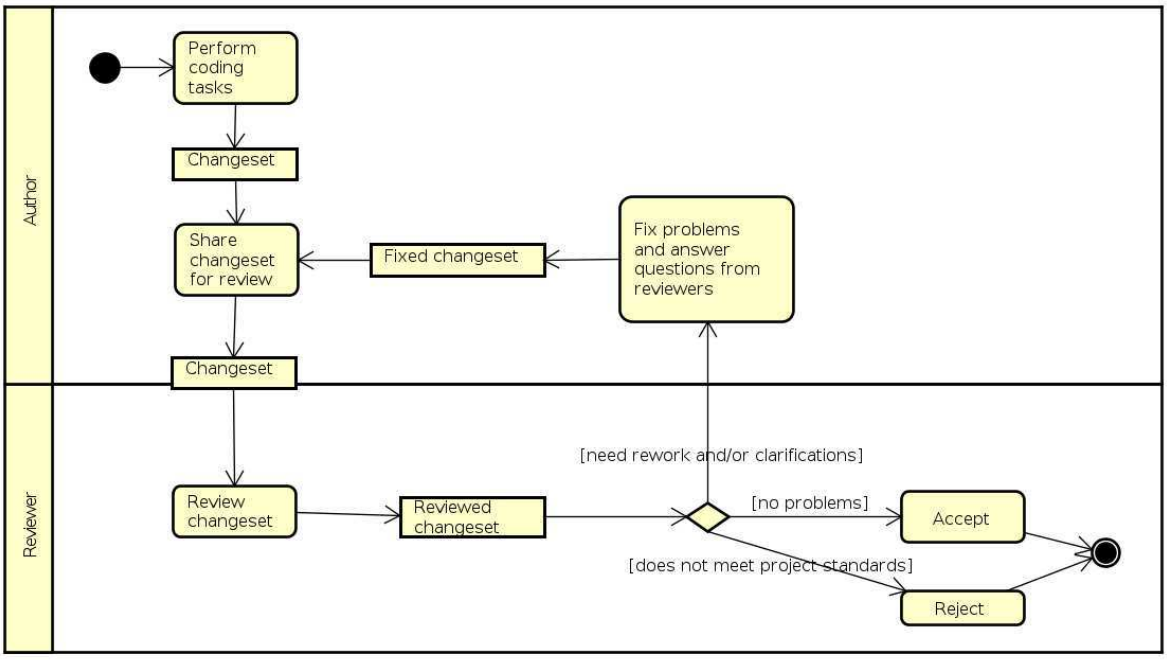

Figure 2.1: Workflow of modern code review. Figure extracted from Freire, Victor et al. (2016)

promotes transparency, quality, and effective collaboration in software development.

For open-source projects, the benefit is notable. Pull-based development makes it easier for external developers to participate, allowing them to contribute to projects without needing direct access to the main branch. This expands the collaborator base and promotes diversity of ideas and experiences [17].

GitHub, a leading code hosting platform, plays a key role in this scenario. With more than 100 million repositories and millions of contributors, GitHub is essential for collaboration on software projects [1]. It offers robust tools for code review, issue tracking, and continuous integration [2]. These numbers underscore the importance of GitHub as a global hub for developers, facilitating discovery, contribution, and collaboration on an international scale.

A Pull Request (PR) on GitHub [Figure 2.2] can contain all changes proposed by a developer on a given branch. In addition to the code itself, a PR can contain detailed descriptions of the changes made, justifications for the changes, screenshots, or relevant links. Comments and discussions between collaborators are also part of the content of a PR, providing full context about the proposed changes.

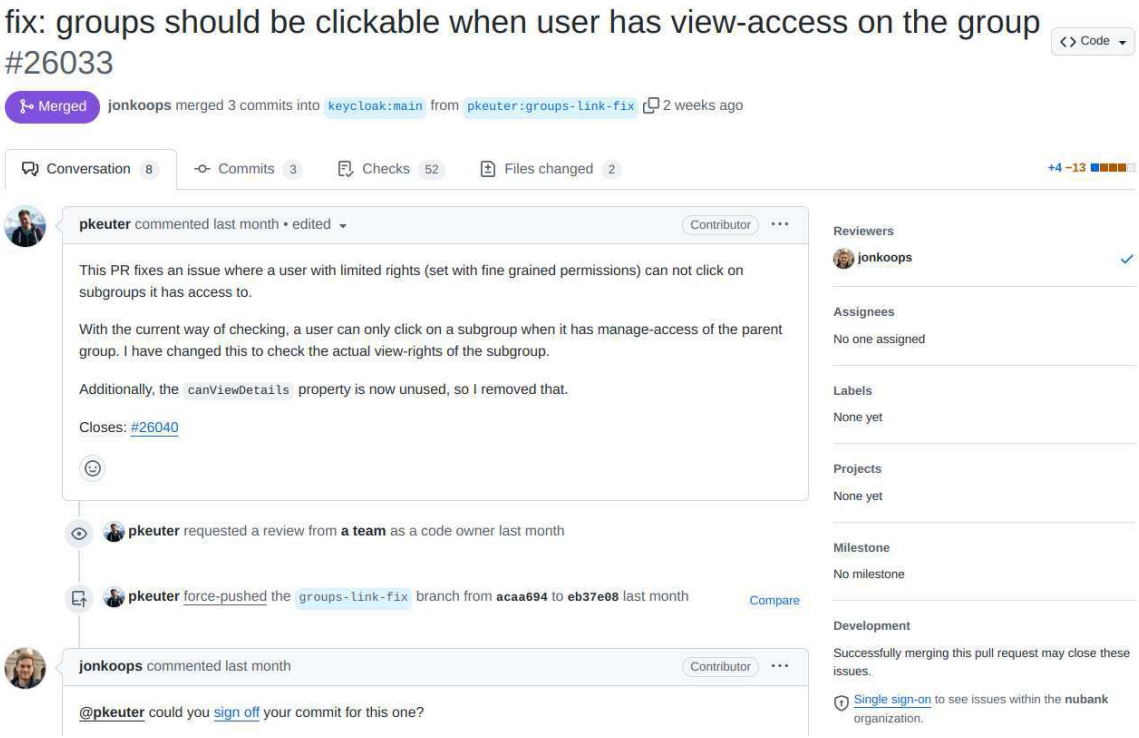

Figure 2.2: An example of a pull request from GitHub.

2.4 Replication of Studies in SE

Replication of academic studies in the area of Software Engineering involves repeating previous research to confirm, validate, or refute its results. This practice is fundamental to guarantee the reliability and generalization of findings, contributing to the construction of a solid body of knowledge in the area.

It is notable that Software Engineering historically has a low number of replications. According to systematic analyses and reviews, only a small percentage of studies in the area are replicated [23]. This finding suggests a significant gap in the validation and verification of the results obtained, raising questions about the reliability of conclusions in many studies.

The lack of replications in Software Engineering can be attributed to several factors, including the varying complexity of software projects, a lack of standardization in research practices and methods, as well as a culture that has historically emphasized the production of new knowledge at the expense of systematic validation [5][8].

2.4.1 Gómez et al. classification of SE replications

Gómez et al. [16] noticed a lack of agreement among Software Engineering (SE) researchers regarding the definition of a replication, along with identified deficiencies in previously proposed classifications. Consequently, they devised a classification system for SE replications with the purpose of rectifying these issues. Specifically, their classification seeks to achieve four objectives: "(1) clearly delineate the boundaries of a replication and permissible alterations; (2) comprehensively outline all potential modifications to the original experiment; (3) employ self-explanatory terminology; and (4) link each replication type with its verification purpose(s)" [16].

In this classification, experiments have four dimensions:

- **Operational:** consists of the operational definitions of the cause and effect constructs. For example, in this work, the cause construct is the ClusterChanges technique and its operationalization is JClusterChanges and how it is applied in the experiment.
- **Protocol:** the materials and instruments used. This dimension includes the experimental design, measuring instruments, and data analysis techniques.
- **Population:** the subjects and/or objects used in the experiment. In this work, for instance, the objects are the pull requests to which JClusterChanges is applied.
- **Experimenters:** roles performed in the experiment. If the same experimenters participate in the replication, but they perform different roles, the replication is considered to have changed this dimension. Examples of roles are the data analyst and the experimental designer.

Given that all dimensions can undergo changes in a replication, a critical question emerges regarding the minimum criteria for a study to retain its status as a replication. According to Gómez et al. [16], for a study to qualify as a replication, it must conduct an experiment and retain some of the hypotheses from the original study. Specifically, "at least two treatments and one response variable need to be shared" [16].

Furthermore, the nomenclature of replication is determined by the altered dimensions, except in cases where either no dimensions were changed or all dimensions were changed.

In such instances, they are respectively termed repetition and reproduction. For example, a replication that involves modifications to both the protocol and population is labeled as a changed-protocol/-population replication.

Chapter 3

Methodology

In this chapter, we describe the methods to replicate the quantitative substudy of the original study from Han, Xiaofeng et al. (2021) . First, we explain the study design and the rationale behind it. Then, we explain how data was collected. Finally, we list the main threats to validity.

3.1 Study Design

The goal of this study is to minimize the threats to external validity from the original study which were enumerated in Section 1.2. To accomplish this, we perform a replication where we change the population dimension as recommended by Gómez et. al [16] classification of SE replications in Section 2.4.1. Hence, this is a changed-population/changed-experimenters replication.

To address the problems presented in Section 1.1, on the importance of studying smell discussions, the following research questions were defined by Han, Xiaofeng et al. (2021) :

- **RQ1:** Which code smells are the most frequently identified by code reviewers?
- **RQ2:** What are the common causes for code smells that are identified during code reviews?
- **RQ3:** How do reviewers and developers treat the identified code smells?
  - **RQ3.1:** What actions do reviewers suggest to deal with the identified smells?

- **RQ3.2:** What actions do developers take to resolve the identified smells?
- **RQ3.3:** What is the relationship between the actions suggested by reviewers and those taken by developers?

To answer those questions, a replication was carried out to seek to understand the particularities, challenges and solutions adopted in the specific context, using qualitative and quantitative data to extract valuable insights. For each research question, the following plans were made, all following the proposal by Han, Xiaofeng et al. (2021) :

3.1.1 Design of RQ1

The comments were analyzed in order to identify and record the type of smell discussed. The list of terms defined in Table 1.1, which contains terms such as "Dead Code" or "Duplicated Code", was used as inspiration for choosing terms and analyzing this research question. When a reviewer used general terms to discuss the smell, such as "smelly", "anti-patter", "comments", "lazy", "Inappropriate", "Large", among others, we classified the smell type of those reviews as "general".

3.1.2 Design of RQ2

Thematic analysis [7] was carried out to find the causes for the identified smells, but we also used the main causes found in the original study:

- **Violation of coding conventions:** certain violations of coding conventions (e.g. naming convention) cause the smell. (Example: “moreThanOneIp (CamelCase) is not our naming convention”).
- **Lack of familiarity with existing code:** developers introduced the smell due to unfamiliarity with the functionality or structure of the existing code. (Example: “This useless line because None will be returned by default”).
- **Unintentional mistakes of developers:** the developer forgets to fix the smell or introduces the smell by mistake. (Example: “You can see I renamed all of the other test methods and forgot about this one” 18).

- **Improper design:** The smell is identified to be related to the improper design of the code. (Example: "...If that's the case something is smelly (too coupled)....").

- **Detection by code analysis tools:** The reviewer points out that the smell was detected by code analysis tools. (Example: "pass is considered as dead code by python coverage tool").

During the classification process, we selected the discussion excerpts that point to a possible cause for the smell discussed. Next, we looked over all the code that we created to identify common patterns among them, but no additional themes than the previously defined ones were found during the classification. We then reviewed the generated themes by returning to the dataset and comparing our themes against it to define the causes of the smells. When no cause was found, the review was classified as "cause not provided/unknown".

3.1.3 Design of RQ3

Manual analyses of code reviews were carried out to identify the actions suggested by the reviewer and carried out by the developers.

- **RQ3.1:** We characterized the actions suggested by the reviewers using the categories proposed by Tahir et al. (2018) , as carried out in the original study:

    1. **Fix:** recommendations are made to refactor the code smell.
    2. **Capture:** detect that there may be a code smell, but no direct refactoring recommendations are given.
    3. **Ignore:** recommendations are to ignore the identified smells.

- **RQ3.2:** We investigated developers' reactions to reviewers who pointed out code smells in their code. This study was conducted in three phases: initially, we examined the developer's response to the reviewer during the discussion. We then investigate the source code file(s) associated with the change before the review and the modifications made to the source code after the review. Finally, if developers have not responded to reviewers or made changes to the source code, we check the status of the corresponding change (i.e., whether it has been merged or abandoned). We consider that

the identified code smells were resolved in at least one of the following scenarios: 1) the original developer recognized the need for refactoring (as part of the review discussion), 2) changes were made to the source code file(s), and 3) the smelly code was later abandoned.

- **RQ3.3:** Based on the results of the previous questions, we did a tree map with the following categories:

  1. **A developer** agreed with the reviewer's recommendations.
  2. **A developer** disagreed with the reviewer's recommendations
  3. **A developer** did not respond to the reviewer's comments.

  We mapped those categories in two actions:

  1. **Fixed** the smell (i.e., refactoring was done)
  2. **Ignored** the change (i.e., no changes were performed to the source code with regard to the smell)

The entire classification process was carried out by two reviewers. A third author (co-author of this work) was involved in cases of disagreement between the two coders. The manual analysis process took around two months of part-time work for the coders. We also provided a full replication package containing all the data, scripts, and results from the manual analysis online[1].

3.2 Data Collection

This section explains how we collected a sample of 2164 reviews from GitHub [Figure 3.1].

3.2.1 Software Project Sample Selection

1. **Crawling review comments:** To select the data, using the script [1][2] available in our replication repository, a ranking was made with the GitHub projects that have

[1]https://zenodo.org/records/10607935
[2]https://zenodo.org/records/10607935

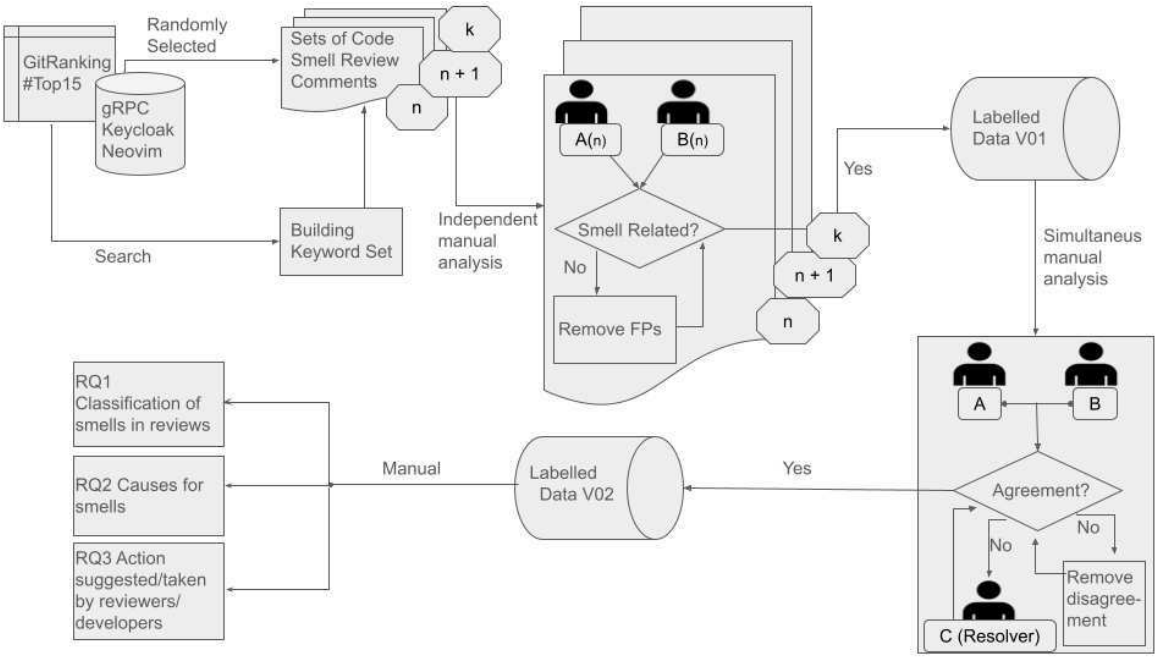

Figure 3.1: Overview of replication methodology

the highest average number of comments per pull requests between 09/11/2023 and 09/14/2023. Table 1.3 presents the top 15 projects from different contexts. Thus, we extract 104,321 review comments from the gRPC, Neovim, and Keycloak projects, which use the C++, Java, and Vim languages, respectively.

2. **Smell keywords search:** Using those comments extracted in the previous step and the list of code smell keywords 1.2, with the script [2][3] we did a keyword search to find comments related to code smells. Thus, 19,149 comments that had at least one of the keywords were collected [Table 3.1].

Table 3.1: Overview of collected data

Project	Raw quantity	Keyword filtered qty	Keyword used qty	Interval
Neovim	39372	5965	5666	01/14 - 08/23
gRPC	46483	9125	9125	01/14 - 08/23
Keycloak	18466	4059	4059	01/14 - 08/23
Total	**104321**	**19149**	**18850**	

gRPC [4] is known as Google Remote Procedure Call and is an open-source remote proce-

[3]https://zenodo.org/records/10607935

[4]https://grpc.io/

dure calling system first developed at Google in 2015 as the next generation RPC infrastructure, Stubby. Its main project on github mostly uses C++ as its standard language. Keycloak [5] is an open source software product that enables single sign-on with identity and access management for modern applications and services. It was initially developed in 2014 and uses Java as its main programming language. Neovim [6] is a Vim refactoring project, which seeks to maximize the tool's contribution and extension.

For simplicity and to reduce documentation difficulties found in the original work to reproduce the codes, all scripts used in this selection were done from scratch following the goals to allow the software project sample selection [7].

3.2.2 Pull Request Sampling

For data mining, we followed the step-by-step guide done by Han, Xiaofeng et al. (2021) . All steps are illustrated in Figure 3.1 and are explained in the details:

1. **Manual Independent Analysis:** A qualitative study involved voluntarily 26 experienced developers, who in 13 pairs, independently and manually analyzed one of the 13 sets of 1,450 comments (total of 18,850) filtered in the previous step. This study sought to understand the context of the comments to confirm previous keyword searches and remove false positives when both reviewers in a pair agreed that the discussion was not related to smells. Thus, 4,563 smell comments were extracted. Among the reviewer characteristics: They have at least three years of experience with software development and, in addition, they all declared having previous contact with code review.

2. **Manual Simultaneous Analysis:** To confirm previous findings, two experienced developers, including the author of this work, carried out a new qualitative analysis, seeing the entire thread of comments and source code when needed, to simultaneously and manually analyze those results. Thus, with 0,95 Cohen's Kappa coefficient of agreement, 3,702 smell comments were extracted in this step. A total of 23 comments required analysis by a third reviewer to ensure mining agreement and 15 additional

[5] https://www.keycloak.org/
[6] https://neovim.io/
[7] https://zenodo.org/records/10607935

comments were extracted. The discussion below shows an example of that. It may have raised doubts because the discussion was edited to no longer highlight a smell. However, a third reviewer also analyzed the comment and considered it as a smell discussion, considering the reviewer's initial suggestion.

> **Link:** https://api.github.com/repos/grpc/grpc/pulls/19971
>
> **Reviewer:** Feels like 'LocalConnectionType' reads better alongside the rest of Python API.
>
> **Developer:** This API is exposed internally. Change the naming might need another large-scale change.EDIT: Turn out there are only a few call sites that I can manually change. Updated.
>
> ---
>
> **Project:** gRPC
>
> **Smell keywords:** [naming, large]
>
> **Methodology:** Pull request sampling through keyword search

A total of 3,798 comments remained at the end of the process.

3. **Contextual information:** As carried out in the original study, the contextual information of each comment was stored in an external text[8] file for later analysis, containing: 1) the URL to the code change , 2) the type of the identified code smell, 3) the discussion between reviewers and developers, and 4) a URL to the source code. We ended up with a total of 2,068 smell-related reviews.

4. **Random selection sample** The 85,471 review comments that were not keywords filtered were used to create an additional random sample of 1,054 comments [Table 3.2], based on a 95% confidence level and 3% margin of error [20]. Steps 1 to 3 were made to this sample and additionally, 96 smell reviews were added to the final set. Resulting in 2,164 smell-related reviews.

5. **Classification process** The classification process was carried out with the same two authors from step 2 and sought to answer the research questions through the design presented in Section 3.1. It is important to highlight that, as in the original study, the

[8]https://zenodo.org/records/10607935

results from the random sample and keyword filtering were merged and classified simultaneously. Therefore, no specific results were cataloged for the random sample. The entire classification process took 2 months of partial dedication from the developers.

The results of this study are presented in Section 4.

Table 3.2: Overview of Random Sample with 95% IC and 3% ME

Project	**row quantity**	**keyword used qty**	**Diff**	**Random selection sample**
Neovim	39372 (38%)	5666		401 (38%)
gRPC	46483 (44%)	9125		465 (44%)
Keycloak	18466 (18%)	4059		191 (18%)
Total	**104321**	**18850**	**85471**	**1054**

Chapter 4

Results

Next, we present the results and discuss them.

For each of the research questions addressed in the original study by Han, Xiaofeng et al. (2021) , we carried out new analyses based on the new data context studied.

4.1 RQ1: Which code smells are the most frequently identified by code reviewers?

We identified 2,164 smell reviews, which compared to the initial number of comments studied (104,321) demonstrates that smells are not commonly discussed in code reviews. Under a new context, the most frequent smell in code reviews is Duplicate Code (39,9%). Also, Bad Naming (9%) and Dead Code (7,1%) are discussed with high frequency during reviews. However, the General type of smell had the highest frequency in the replication (48,9%) [Figure 4.1]. The terms such as "code smell", "smelly", "comments", "lazy", "Inappropriate" and "Large" were used to categorize as a General type, as mentioned in Section 3.1.1. The reviewer discussion below shows an example of a duplicated code comment:

Link: https://api.github.com/repos/grpc/grpc/pulls/22774

Reviewer: Instead of refactoring out this `php_grpc_clean_persistent_list` function in `channel.c`, perhaps we can limit our changes to only `php_grpc.c`, and refactor out these 4 lines instead, and call them in those 2 separate places, but within `php_grpc.c`

Project: gRPC

Smell keywords: [list]

Methodology: Pull request sampling through keyword search

In the example, there is duplication because the `clean_pesistent` function does not need to be refactored again in the `channelc` class. All modification and use of the method can be done reusably in the `php_grpc.c` class.

RQ1: The most frequently identified smells in code reviews.

Smells are not commonly discussed during code reviews. The most frequent smells in reviews are **Duplicated Code, Bad Naming, and Dead Code**.

4.2 RQ2: What are the common causes for code smells that are identified during code reviews?

Figure 4.2 presents the results found on the most common causes of smells. 831 (70%) of the reviews have a cause not provided. This frequency is our closest result to the one found in the original study (70%) and confirms the previous finding. Identifying the cause of code smells is a challenge in itself, as it would require deeper knowledge of the code and nuances of that software. Often the cause is not even known yet. This is the kind of thing that is trickier to infer just based on PR comments and motivates a future study extension to better handle these possible causes. See example below:

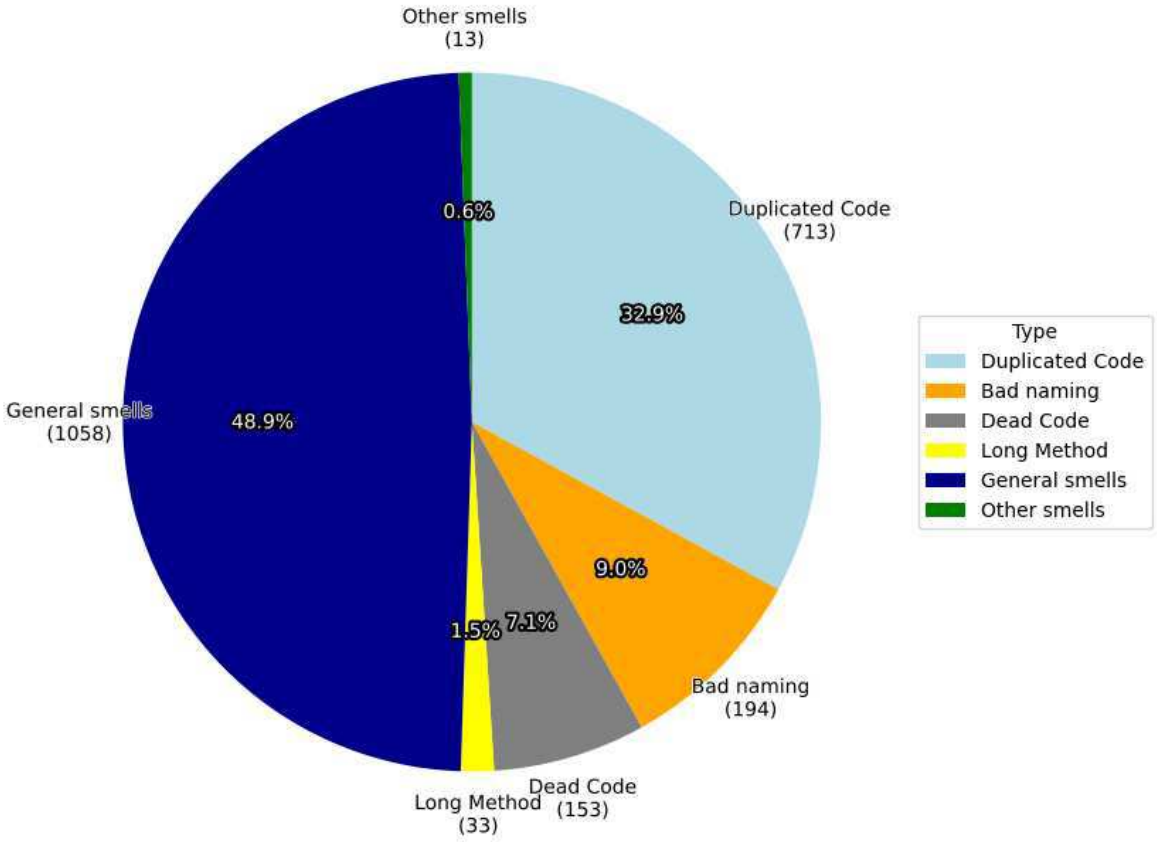

Figure 4.1: Number of reviews for the identified code smells.

Link: https://api.github.com/repos/grpc/grpc/pulls/17643

Developer: I have updated the comment to clarify why we have the `if` condition.

Project: gRPC

Smell keywords: [condition]

Methodology: Pull request sampling through keyword search

The cause is not provided here because the comment aims to improve readability but even those who have familiarity with the code might need clarification to understand why the developer chose to use the if in that case. We believe that every smell has an underlying cause, however, during code review discussions, the cause is not always provided due to some reasons, such as lack of adequate documentation or simply because the reviewer/author may not be aware of the underlying cause of the smell.

However, it was possible to notice that the lack of familiarity with the existing code, pre-

sented in 198 (9%) of the reviews, the inappropriate design, in 214 (10%), and violation of code conventions, in 147 (7%) of the occurrences, had a similar frequency. This demonstrates the influence of project knowledge, knowing established programming conventions, and familiarity, to avoid smells. Other possible causes were identified, but to a lesser extent, such as unitentional errors (85 reviews) and detection by code analysis tools (11 reviews).

RQ2: The common causes for smells as identified during code reviews.

Most of the smells discussions don't have a specific cause provided during code reviews. The **lack of familiarity, violation of code conventions, and inappropriate design** have a low and similar frequency to be pointed as the cause of the smell.

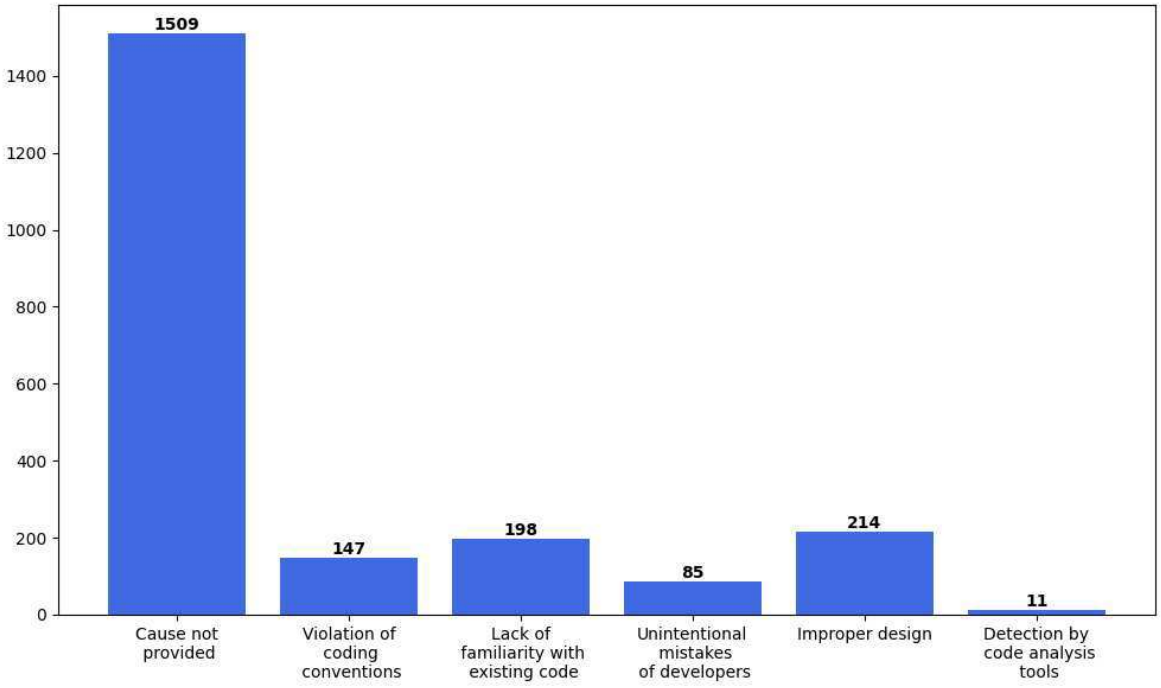

Figure 4.2: Reasons for the identified smells.

4.3 RQ3: How do reviewers and developers treat the identified code smells?

4.3.1 RQ3.1: What actions do reviewers suggest to deal with the identified smells?

Table 4.1 shows that only 290 (13%) reviews present correction recommendations, in contrast, 1781 (82%) reviews capture the smell but do not provide any additional suggestions. At a lower frequency, reviewers recommended ignoring smells in 22 (1%) of the occurrences. Comments where reviewers directly point out the smell even if they indirectly suggest modifications or criticize the code were considered as "smell capture". Therefore, only direct comments were marked as correction recommendations, which may explain the low frequency, independently whether they provided an implementation suggestion or not. See the example below, where the reviewer indirectly pointed out a readability problem, but neither provided directly a recommendation for fixing it nor a suggestion on how this should be done:

Link: https://api.github.com/repos/grpc/grpc/pulls/10210

Reviewer: Please comment the thinking behind this condition?

Project: gRPC

Smell keywords: [condition]

Methodology: Pull request sampling through keyword search

Table 4.1: Actions recommended by reviewers to resolve smells in the code

Reviewer's recommendation	**Count**
Fix (without recommending any specific implementation)	204
Fix (provided specific implementation)	86
Capture (just noted the smell)	1781
Ignore (no side effects)	22

4.3.2 RQ3.2: What actions do developers take to resolve the identified smells?

To answer this question, we analyzed the developer's actions, mentioned in Section 3.1.3, like understanding the developer's response to reviewers, the changes made to the source codes, and the status of their pull requests to infer if the code smell was solved. Table 4.2 presents the developer's correction for the identified smells. Most types of code smells identified had a high percentage of correction, especially the most frequent smells, such as Duplicated code (96% fixed), Bad Naming (98% fixed), and Dead Code (82% fixed), totaling 94% of smells corrected.

Table 4.2: Developers' actions to code smells identified during reviews

Code smell	**Reviews**	**Fixed by developers**	**% of fixes**
Duplicated Code	713	687	96%
Bad Naming	194	190	98%
Dead Code	153	150	98%
Long Method	33	27	82%
Circular Dependency	11	4	36%
Swiss Army Knife	2	1	50%
General Smell	1058	977	92%
Total	**2164**	**2044**	**94%**

4.3.3 RQ3.3: What is the relationship between the actions suggested by reviewers and those taken by developers?

To answer this research question, Figure 4.3 presents a map between reviewers' recommendations and the resulting actions taken by developers. The replication shows that, in the majority, 1713 (79%) reviews had discussions in agreement with the reviewer, of which 1671 (77%) resulted in code fixes. Less frequently, 407 (18%) of the reviews were not responded to, although in the majority of them, 326 (15%) resulted in code correction. The replication shows that the developer disagreed with the reviewer in 2% of occurrences and in 24 (1% of occurrences) the code was still corrected. An example where the developer disagreed

with the reviewer but made changes is shown below because the developer believes that the argument presented by the reviewer contradicts what has been discussed previously:

Link: https://api.github.com/repos/grpc/grpc/pulls/1163

Reviewer: I'm pretty sure you aren't supposed to have binary metadata values unless the metadata key ends with `'-bin'`.

Reviewer: The `or premetadata` smells stale here, given that calling `add_metadata` after `premetadata` is how terminal metadata gets added, right?

Developer: This contradicts what I've heard offline; ostensibly metadata values have to both have a length value and be null-terminated. Yes, it's redundant, but that was my impression from offline conversations. This sounds like it needs a clarifying issue to me.EDIT: Although, the core implementation appears to Do The Right Thing™ and use the length as far as I can tell.

Project: gRPC

Smell keywords: [data, smell]

Methodology: Pull request sampling through keyword search

RQ3: The reviewers' recommendations and developers' actions.

Reviewers usually capture the smell but don't directly recommend fixes. **Developers often fix** the discussed smells. In most of the smell discussions, the **developer agreed with the reviewer and fixed the code**.

4.4 Discussion

Although we pointed out the motivations to manually detect and discuss smells, because of the project domain and contextual information that it provides, the results of RQ1 present that the code smells are not widely discussed during code reviews. This might represent that most of the smells have been detected and fixed in previous steps, using automatic techniques, for example. Also, Duplicated Code was the type more discussed in our analyses, it tends to be discussed more in code reviews due to its immediate visibility and its direct impact on code maintainability and readability, often generating more discussion than other types of smells.

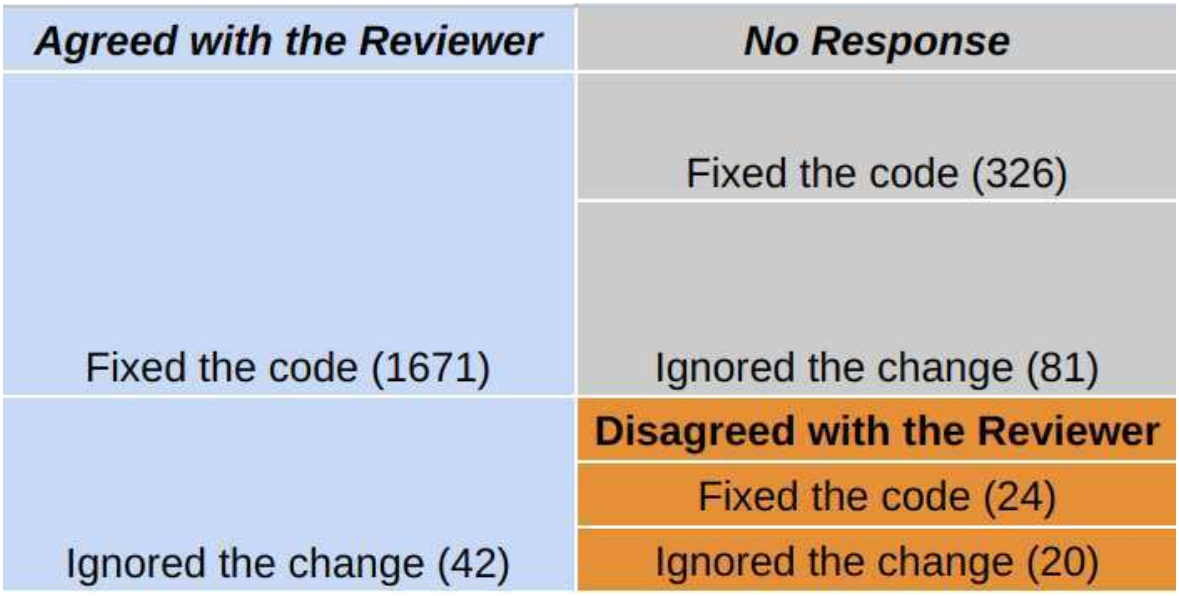

Figure 4.3: A treemap of the relationship between developers' actions in response to reviewers' recommendations regarding code smells identified in the code.

Its detection is more direct and its ramifications are widely recognized as critical to software quality.

Most of the findings at RQ2 show that reviews usually don't provide specific causes for the identified smells. However, the causes pointed out, like lack of familiarity, violation of code conventions, and inappropriate design, show us that most of the smells might be avoided whether the developer has the domain of the project.

The results of RQ3 present that, in the majority, developers have review discussions in agreement with the reviewer's suggestions, and most of its reviews result in code fixes. This analysis indicates that suggestions or concerns raised by reviewers have been acknowledged and addressed by developers through source code adjustments. These numbers suggest remarkable effectiveness in the review process, where constructive communication between reviewers and developers often leads to tangible code improvements.

Chapter 5

Threats to Validity

During the course of this work, a new study [18] was published by the same authors of the original work [19]. In the new approach, an extension of the original work was made by including a set of data from Qt community projects that use C++ as the majority programming language, keeping the original research questions and including new ones. Given this, some threats to validity addressed as a motivation for replicating this study may have been corrected in the new extension. Still, we promote an approach that has a different vision, focusing on the gRPC, Neovim, and Keycloak projects, which use the C++, Java, and Vim languages.

As 26 reviewers contributed to this work, even with similar experiences their analysis can be put at risk because the qualitative decisions on what is code smells or not may vary. Also, the familiarity with the programming languages might be different. To reduce this threat, before starting the manual analysis, the reviewers received a guide about what should be considered a smell discussion, the document contains the smell background explained in Section 2 and aims to guide everyone with the same concepts.

At RQ1, to classify "general" terms we have used the definition made by Han, Xiaofeng et al. (2021) and their examples like "code smell", "smelly", but we also considered more terms as general types, like "lazy" and "Inappropriate". The original study allows room for interpretation of the examples and a specific study can be carried out on those topics to improve the accuracy of those general terms definition.

The analysis of RQ2 for the specific provided causes for code smells had each one at most 10% of occurrences and this small sample might not reflect the reality. To mitigate this,

a comparison with the results for the same RQ in the original work by Han, Xiaofeng et al. (2021) was done and showed similar results for a different data context.

Also, for RQ3 a low number (2%) of results shows that the developer disagreed with the reviewer. This low frequency might not be realistic enough to imply results, but it gives space to deeper manual analysis of those kinds of threads, especially about human behaviors during code reviews.

Chapter 6

Related Work

We found studies that present techniques and highlight the importance of smell detection. Furthermore, we identified other works that relate to code reviews and software development. In this chapter, we summarize these techniques.

6.1 Studies on Code Smells Detection

Some studies analyze code smell statistics in projects of different sizes and complexities and discuss the implications of the results for software development. The analyses are performed by extracting code smells from repositories for different programming languages and versioning tools [38] [43] [10].

Pereira dos Reis, José, et al. (2022) analyzed 80 studies to identify the most common approaches to scent detection. They present that the most commonly used approaches for detecting code smells are search-based (30.1%), metrics-based (24.1%), and symptom-based approaches (19.3%). Furthermore, machine learning (ML) techniques are used in 35% of studies. However, the study emphasizes the need to reduce the subjectivity associated with the definition and detection of smells as a challenge.

Azeema, M, et al. (2019) also highlight the challenge of reducing subjectivity and promote a systematic review of machine learning techniques for smell detection. The study shows that Decision Trees and Support Vector Machines are the Most commonly used machine learning algorithms for code smell detection.

Lin, T et al. (2021) conducted research on code smell detection based on deep learning.

Their solution uses a convolutional neural network to train a model to detect several common code smell issues in software engineering. The solution achieves a satisfied F2 score with an average above 0.75.

S, Shcherban et. al (2020) [39] present a proposal on how to automatically detect code smell-related textual discussions in stack overflow (SO) using an automatic classifier. They conducted an experiment using a manually labeled dataset that contains 3000 code-smell and 3000 non-code-smell posts to evaluate the performance of different classifiers when automatically identifying code-smell discussions. The results show that the machine learning approach can effectively locate code-smell posts even if the posts' titles and/or tags cannot be of help. However, this approach was not used in this replication because it is a preliminary investigation and neither code nor instructions are provided to guide reproduction.

Although there are several smell detection techniques, our study, however, focuses on manually detected smells in code reviews, motivated by the benefit of contextual information characteristic of the reviewer. Still, the results from Section 4 show that there are few discussions about code smells in code review. This leaves room for new studies that seek to understand the impact of automatic smell detections on reducing discussions during code review.

6.2 Studies on what is discussed on Code Review

Several studies [29] [28], [24] have explored the impact of modern code reviews on software quality. Other research has also addressed the impact of code reviews on different aspects of software quality, such as vulnerabilities [6], design decisions [45], anti-patterns [31] and code smells [32] [34].

Pascarella, L., et al. (2019) present a study where that checks the detection and reduction of several code smells in files during reviews. For those reviews with reduction, authors manually analyzed reviews and found that reviewer suggestions provided "side effects" that helped to reduce smells. Our study differs from it because we directly analyze the smell review discussions with the goal of understanding the developer's and reviewer's perspectives.

The study by Ferreira, I, (2021) seeks to understand the characteristics of incivility in the discussion of code review, hitherto little explored in software engineering, which can occur

due to disagreements between developers and reviewers. This involves heated discussions and sometimes involves personal attacks and unnecessary disrespectful comments. They performed a qualitative analysis on 1,545 posts and found that more than half (66.66%) of non-technical discussions included uncivil characteristics. Furthermore, frustration, swearing and impatience are the most common characteristics of rude arguments. Our study does not seek to understand behavioral existence characteristics in code review discussions, but mainly as actions suggested/taken by reviewers/developers and how it can impact review standards.

6.3 Study by Han, Xiaofeng et al. (2021)

6.3.1 Overview

The objective of this study was to investigate the concept behind code smells identified during code reviews and what actions are suggested by these reviewers and carried out by developers in response to the identified smells. This work was the basis for our replication. What motivated us to explore it was the existence limitations in the empirical evidence and current applicability, highlighted in Section 1.2.

To this end, they created the research questions presented in Section 1.2. Two active projects on OpenStack were chosen for the analysis (Nova and Neutron) and a total of 309,311 comments were collected [Table 6.1] to compose the data processing steps described in Figure 6.1.

Table 6.1: An overview of the subject projects (Nova and Neutron). Table extracted from Han, Xiaofeng et al. (2021)

Project	Review Period	Code Changes	Comments
Nova	Jan 14 - Dec 18	22,762	156,882
Neutron	Jan 14 - Dec 18	15,256	152,429
Total		38,018	309,311

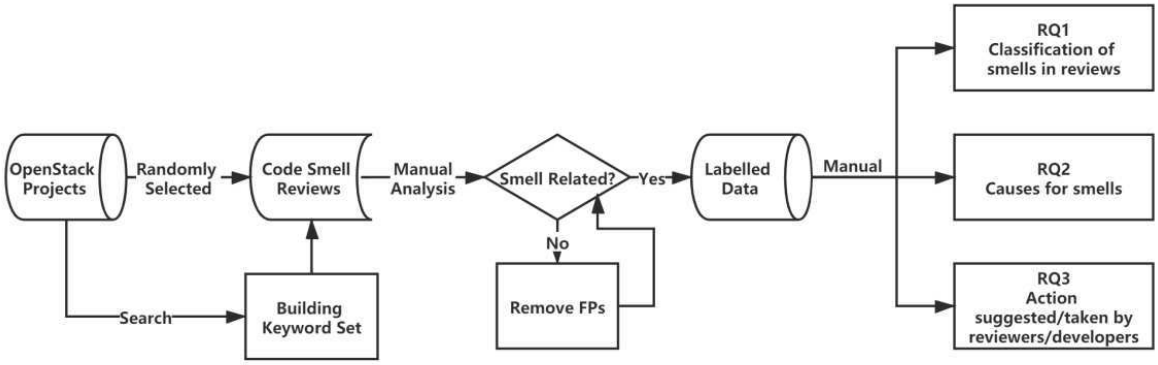

Figure 6.1: Overview of data mining. Figure extracted from Han, Xiaofeng et al. (2021)

6.3.2 Mining Process

1. **Creation of the set of keywords:** A set of keywords from the list of terms [Table 1.1] was constructed using variations of terms that refer to code smell, such as "code smell", " bad pattern", etc. This set considered the smells initially defined by Martin Fowler (1999) and also those extracted by Tahir et al. (2020).

2. **Data extraction:** Python code was developed to collect review comments that have at least one of the keywords described in the set. At this stage, 18,082 comments were collected.

3. **Identification of smell-related keywords-searched comments by two authors independently:** In this step, two authors manually analyzed the same set of comments, and excluded those that were not related to code smell. At the end of the process, the comments where both authors marked as not related to smell were deleted and Cohen's Kappa agreement coefficient was calculated, resulting in a value of 0.85, which indicates high agreement. A total of 3,666 comments remained after this process.

4. **Identification of smell-related keywords-searched comments by two authors together:** At this stage, simultaneously, the same two authors analyzed the comments and consulted the source code when necessary. Furthermore, the authors tried to agree on the relationship between the comment and code smell. At the end of the process, only the comments where the two authors agreed about being related to code smell were kept. A third author also participated in the process when doubts arose and the

initial authors did not agree. Thus, this resulted in a reduction to 1,235 comments.

5. **Contextual information about the review comment:** At this stage, the contextual information of each review comment was collected and includes the following information: 1) Url of the code change; 2) The type of code smell identified; 3) The discussion between reviewers and developers and 4) The Url to the source code. At the end of this process, 1,164 reviews related to code smells were collected.

6. **Identification of smell-related comments in randomly-selected set:** With 95% confidence and 3% margin of error [20], a sample with an additional 1,064 comments was collected from a set of 291,229 comments, which represents the rest of comments that did not pass the keyword filtering described in step 2). After this extraction, steps 3) to 5) were followed and another 16 smell-related reviews were collected. Therefore, a sample of 1,190 smell-related reviews was used for the remaining research analyses. As an artifact of this phase of the study, a replication package was made available online [1].

To improve the robustness of the study, we diversify the sample, including projects from different organizations that use different technologies. Our data processing is described in Section 1.2, it tries to follow the original study as closely as possible, except that in step 3, instead of using only two reviewers we involved 26 experienced developers, who in 13 pairs, independently and manually analyzed the data.

6.3.3 Results

In response to the research questions, the authors made the following assessments:

RQ1: Which code smells are the most frequently identified by code reviewers?

From the set of 1,190 smell-related reviews, in comparison with the total population studied, it was identified that the proportion of smells found is low. Even so, among the identified smells *duplicated code* has the highest frequency, with 620 occurrences [Figure 6.2], followed by *bad naming* and *dead code*, with 304 and 221 occurrences, respectively. To a lesser extent, long method, *circular dependency* and *swiss army* were also identified. In 11 reviews, smells were described with general terms, such as *code smell*

[1] https://doi.org/10.5281/zenodo.4468035

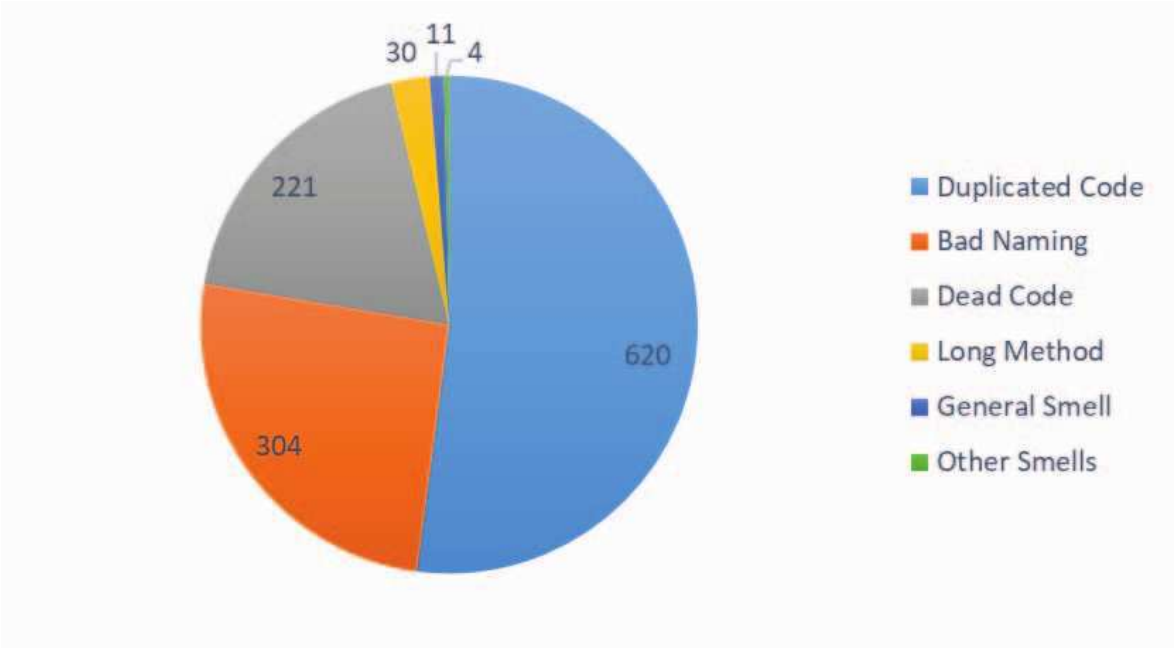

Figure 6.2: Number of reviews for the identified code smells. Figure extracted from Han, Xiaofeng et al. (2021)

RQ2: What are the common causes for code smells that are identified during code reviews?

Using Thematic Analysis, we defined the causes presented in Section 4, and the following causes were found:

Figure 6.3 shows that most reviews do not provide any information about the possible cause of the smell, they just point out the problem. Still, it was possible to understand that in 276 (26%) reviews the cause for the smell was *code convention violation*. In smaller quantities, the causes of *lack of familiarity with existing code* (40 reviews) (3%), *intentional errors* (19 reviews), and *improper design* (18 reviews) were also found.

RQ3: How do reviewers and developers treat the identified code smells?

The results of RQ3 were divided into three sub-questions:

RQ3.1: What actions do reviewers suggest to deal with the identified smells?

Table 6.2 presents the results of this question, which indicate that after the classification made by the authors, it was extracted that in the majority of reviews (870, 73% of the total reviews), reviewers recommend corrections for the identified smells. In 303 of these reviews (35%), the reviewers provided specific implementations for correction, while in 272 of the occurrences (23%) the smells were highlighted but were suggested for correction. In 48

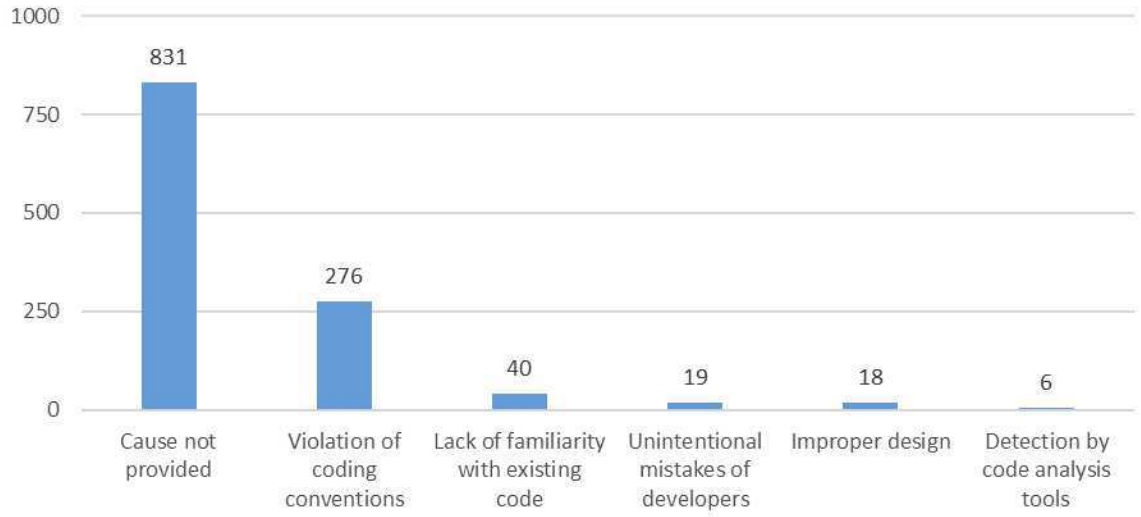

Figure 6.3: Reasons for the identified smells. Figure extracted from Han, Xiaofeng et al. (2021)

reviews (4%), reviewers suggested ignoring the code smell found.

Table 6.2: Actions recommended by reviewers to resolve smells in the code. Table extracted from Han, Xiaofeng et al. (2021)

Reviewer's recommendation	**Count**
Fix (without recommending any specific implementation)	567
Fix (provided specific implementation)	303
Capture (just noted the smell)	272
Ignore (no side effects)	48

RQ3.2: What actions do developers take to resolve the identified smells?

Table 6.3 shows that 1029 (86%) were corrected by the developers. The rest of the reviews did not cause changes to the source code after the reviewer's suggestion, which indicates, according to the authors, that the developers understood in that situation that the smell may not require priority attention.

RQ3.3: What is the relationship between the actions suggested by reviewers and those taken by developers?

Figure 6.4 demonstrates a map between reviewers' recommendations and resulting developer actions. In 775 (65%) of the reviews analyzed, developers agreed with reviewers' suggestions and followed exactly the same recommended actions, whether fixing or ignoring.

Table 6.3: Actions made by developers to resolve smells in the code. Table extracted from Han, Xiaofeng et al. (2021)

Code smell	**Reviews**	**Fixed by developers**	**% of fixes**
Duplicated Code	620	508	82%
Bad Naming	304	276	91%
Dead Code	221	210	95%
Long Method	30	25	83%
Circular Dependency	3	2	67%
Swiss Army Knife	1	1	100%
General Smell	11	7	64%
Total	**1190**	**1029**	**86%**

Among these cases, there were 20 situations in which developers agreed to ignore the smell (i.e., a problem was identified, but the reviewer considered its impact to be insignificant).

In 23% of the reviews (274 cases), developers implemented the necessary changes to source code files without even responding directly to reviewers in the review system. We observed an additional 66 reviews (5%) in which developers disagreed with the reviewers' opinions and chose to ignore recommendations for code refactoring and smell removal.

Similarly, in 75 reviews (6%), developers did not respond to reviewers or change the source code. In these cases, it was assumed that the developers did not find the recommendations on how to deal with specific smells in the code useful and therefore decided not to make changes.

6.3.4 Comparison to the replication

It was possible to compare the main results of the original study with the replication, presented in Section 4:

RQ1: Which code smells are the most frequently identified by code reviewers?

We identified 2,164 smell reviews, which compared to the initial number of comments studied (104,321) demonstrates that smells are not commonly discussed in code reviews, this behavior was similar to the result found at the original work 6.3.3. In comparison with Figure 6.2, we identified that under a new context, the most frequent smell in code reviews

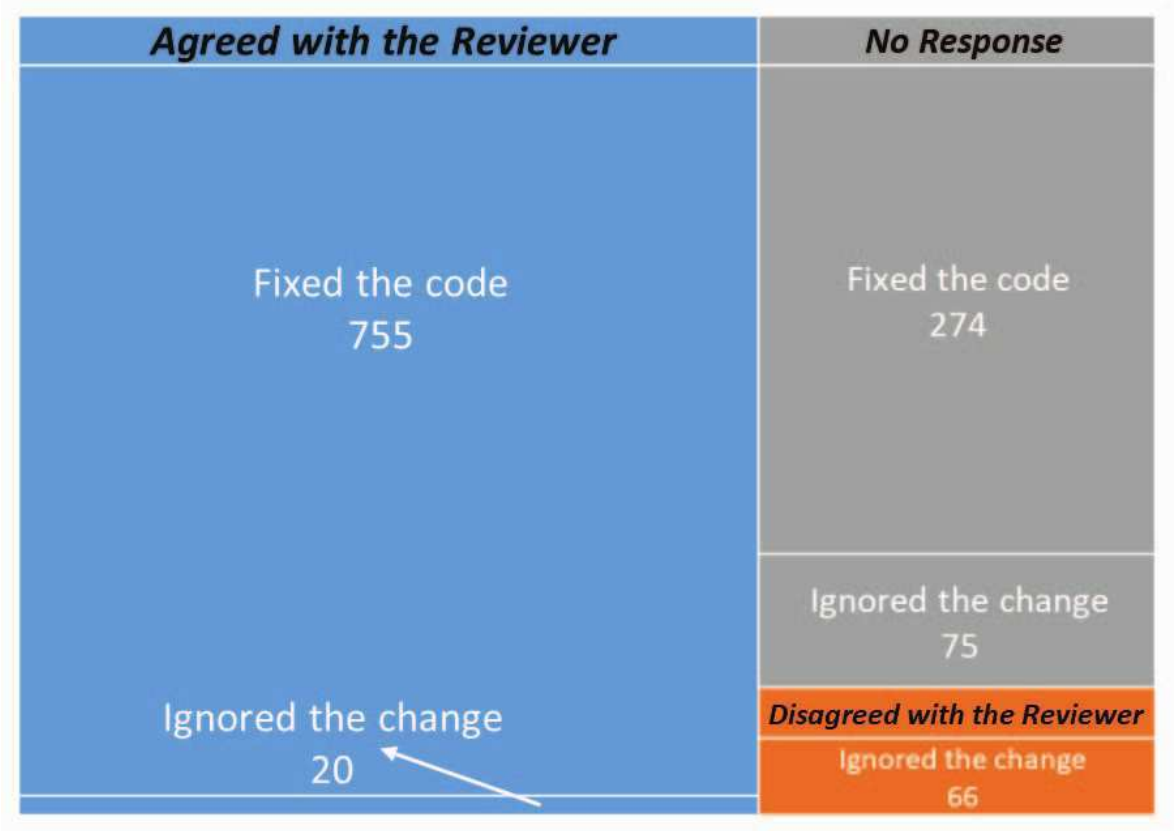

Figure 6.4: A treemap of the relationship between developers' actions in response to reviewers' recommendations regarding code smells identified in the code. Figure extracted from Han, Xiaofeng et al. (2021)

is also code duplication [Figure 4.1]. However, the general type of smell, which was one of the least frequent in the original study, had a higher frequency in our classification. This might happen because the original study doesn't indicate a clear definition for it and allows room for interpretation of the examples of general terms that may explain this variation in frequency. To classify as the "general type", in addition to terms such as "code smell" and "anti-pattern", we consider other terms like "smelly", "comments", "Inappropriate" and "Large" to categorize the general type in the replication.

RQ2: What are the common causes for code smells that are identified during code reviews?

Similar to the original study [6.3.3], 831 (70%) of the reviews have a cause not provided [4.2]. This frequency is our closest result to the one found in the original study (70%) and confirms the previous finding. But differently, while in the original study 26% reviews had the "violation of code convention" cause as the most common, in the replication besides that

cause with 7%, "lack of familiarity with existing code" (9%) and "improper design" (10%) had all similar frequencies. It's important to consider that the frequency of causes presented is not that high and the comparison shows that might not be the most frequent cause.

RQ3: How do reviewers and developers treat the identified code smells?

At a lower frequency, similar to the original study 6.2, reviewers recommended ignoring smells in 22 (1%) of the occurrences 4.1. The replication showed high frequency (82%) reviewers usually noted the smell but didn't recommend suggestions, this result is different from the original study (23%). During the classification process, we realized that some data might have uncertainty on which action should be classified, especially between the Capture and the Fix actions. It's not clear the decision points used in the original study for this, but to mitigate it we considered only comments with direct and clear indications of recommendation.

In agreement with the original study [Table 6.3], most types of code smells identified had a high percentage of correction, especially the most frequent smells, such as Duplicated code (96% fixed), Bad Naming (98% fixed) and Dead Code (82% fixed), totaling 94% of smells corrected 4.2.

In confirmation of the original results [Figure 6.4], the replication shows that, in the majority, 1713 (79%) reviews had discussions in agreement with the reviewer, of which 1671 (77%) resulted in code fixes. Less frequently, 407 (18%) of the reviews were not responded to, although in the majority of them, 326 (15%) resulted in code correction. Despite presenting a similar result in relation to the number of discussions in which the developer disagreed with the reviewer (2% of occurrences), unlike the original study, the replication shows that in 24 (1% of occurrences) the code was still corrected.

6.4 Study by Han, Xiaofeng et al. (2022)

The authors provided an extension of the original work presented in 2021 [19]. It was published during the course of this work, therefore this replication is still based on the 2021 study. The following goals were addressed in the extension:

1. Extend the dataset by including the code review data from two large projects of the Qt community;

2. Explore specific refactoring actions suggested by reviewers;

3. Investigate two additional research questions discussing the resolution time of smells and also the reasons why developers chose to ignore the identified smells.

The same methodology used in the original study was performed and the following results were found:

1. Code smells were not commonly identified in code reviews;

2. Smells were usually caused by violation of coding conventions;

3. Reviewers usually provided constructive feedback, including fixing (refactoring) recommendations to help developers remove smells;

4. Developers generally followed those recommendations and actioned the changes;

5. Once identified by reviewers, it usually takes developers less than one week to fix the smells;

6. The main reason why developers chose to ignore the identified smells is that it is not worth fixing the smell.

As mentioned in the threat of validity for this replication, some threats to validity addressed as a motivation for replicating this study may have been corrected in the extended work. However, in the extension the authors still highlight new threats that endorse the importance of this replication:

- Usage of different domains, such as the ones chosen for the replication: gRPC, Neovim, and Keycloak.

- Usage of different programming languages, such as C++, Java, and Vim, used for replication.

Different domains and programming languages can help improve the external validity and make the study results and findings more generalizable to other systems. Also, including code review discussions from other communities will supplement their findings and this may lead to more general conclusions.

Chapter 7

Conclusion

The present study replicates data from the original study by Han, Xiaofeng et al. (2021) on understanding code smell discussions in code reviews. The study highlights the importance of detecting code smells through code review, especially due to the context and domain of the program that is presented by those who use this review process. Furthermore, it allows you to understand the perspectives of developers and reviewers regarding code smells.

The study confirms the results presented by the original work, on the low frequency detection of smells using this manual review process and also on the most frequent types and suggestions of reviewers. In these analyses, replication was able to extend the interpretation on general types of smells and on suggestions that point to the smell and indirectly suggest a modification by the reviewer. This allows the evaluator to consider that the interpretative analysis of discussions is essential in the classification process.

Furthermore, the study also confirms that the majority of smells are corrected (94% of occurrences) and that in 79% of cases, the developers agree with the suggestions provided by the reviewer.

This study can benefit developers as they can gain valuable insights into how their code decisions are perceived by reviewers and learn best practices for dealing with code smells. This can improve the quality of the code they produce. It's also beneficial for reviewers, who can hone their skills by understanding how their suggestions are received by developers. This can lead to more effective communication and more constructive code reviews.

7.1 Future Work

This study provides scope for some new work, namely:

1. Extend the current study with the skill profile of participants involved in the replication process;
2. Deeper analysis to understand how the cause of smells can be better classified;
3. Analyse the effect of programming languages on code smell discussions during reviews;
4. Study of reviews that contain disagreement between the reviewer and the developer;
5. Investigate how a classifier can be used to create a document, for a specific repository, with standards of code review practices (refactoring suggestions, most discussed smells, etc.).

Bibliography

[1] GitHub. About github., 2024. Accessed on 01.14.2024.

[2] GitHub. Github's features, 2024. Accessed on 01.14.2024.

[3] Roberta Arcoverde, Everton Guimarães, Isela Macía, Alessandro Garcia, and Yuanfang Cai. Prioritization of code anomalies based on architecture sensitiveness. In *2013 27th Brazilian Symposium on Software Engineering*, pages 69–78. IEEE, 2013.

[4] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

[5] Victor R Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[6] Amiangshu Bosu, Jeffrey C Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 257–268, 2014.

[7] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[8] Andy Brooks, Marc Roper, Murray Wood, John Daly, and James Miller. Replication's role in software engineering. *Guide to advanced empirical software engineering*, pages 365–379, 2008.

[9] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[10] Ananta Kumar Das, Shikhar Yadav, and Subhasish Dhal. Detecting code smells using deep learning. In *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*, pages 2081–2086. IEEE, 2019.

[11] Michael Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers: contributions to software engineering*, pages 575–607. Springer, 2011.

[12] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 609–613. IEEE, 2016.

[13] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 2018.

[14] Victor da Cunha Luna Freire et al. Automatic decomposition of code review changesets in open source software projects. 2016.

[15] Victor da Cunha Luna Freire et al. Characterization of design discussions in modern code review. 2021.

[16] Omar S Gómez, Natalia Juristo, and Sira Vegas. Understanding replication of experiments in software engineering: A classification. *Information and Software Technology*, 56(8):1033–1048, 2014.

[17] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th international conference on software engineering*, pages 345–355, 2014.

[18] Xiaofeng Han, Amjed Tahir, Peng Liang, Steve Counsell, Kelly Blincoe, Bing Li, and Yajing Luo. Code smells detection via modern code review: a study of the openstack and qt communities. *Empirical Software Engineering*, 27(6):127, 2022.

[19] Xiaofeng Han, Amjed Tahir, Peng Liang, Steve Counsell, and Yajing Luo. Understanding code smell detection via code review: A study of the openstack community. In *2021*

IEEE/ACM 29th International Conference on Program Comprehension (ICPC), pages 323–334. IEEE, 2021.

[20] Glenn D Israel. Determining sample size. 1992.

[21] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.

[22] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17:243–275, 2012.

[23] Barbara A Kitchenham and Shari L Pfleeger. Personal opinion surveys. In *Guide to advanced empirical software engineering*, pages 63–92. Springer, 2008.

[24] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 111–120. IEEE, 2015.

[25] Tao Lin, Xue Fu, Fu Chen, and Luqun Li. A novel approach for code smells detection based on deep leaning. In *Applied Cryptography in Computer and Communications: First EAI International Conference, AC3 2021, Virtual Event, May 15-16, 2021, Proceedings 1*, pages 171–174. Springer, 2021.

[26] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt Von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *2012 16Th european conference on software maintenance and reengineering*, pages 277–286. IEEE, 2012.

[27] Steve McConnell. *Code complete*. Pearson Education, 2004.

[28] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th working conference on mining software repositories*, pages 192–201, 2014.

[29] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21:2146–2189, 2016.

[30] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.

[31] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pages 171–180. IEEE, 2015.

[32] Aziz Nanthaamornphong and Apatta Chaisutanon. Empirical evaluation of code smells in open source projects: preliminary results. In *Proceedings of the 1st International Workshop on Software Refactoring*, pages 5–8, 2016.

[33] Willian N Oizumi, Alessandro F Garcia, Thelma E Colanzi, Manuele Ferreira, and Arndt V Staa. On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development*, 3:1–22, 2015.

[34] Luca Pascarella, Davide Spadini, Fabio Palomba, and Alberto Bacchelli. On the effect of code review on code smells. *arXiv preprint arXiv:1912.10098*, 2019.

[35] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. Code smells detection and visualization: a systematic literature review. *Archives of Computational Methods in Engineering*, 29(1):47–94, 2022.

[36] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 202–212, 2013.

[37] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pages 181–190, 2018.

[38] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018.

[39] Sergei Shcherban, Peng Liang, Amjed Tahir, and Xueying Li. Automatic identification of code smell discussions on stack overflow: A preliminary investigation. In *Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, pages 1–6, 2020.

[40] Amjed Tahir, Jens Dietrich, Steve Counsell, Sherlock Licorish, and Aiko Yamashita. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. *Information and Software Technology*, 125:106333, 2020.

[41] Amjed Tahir, Aiko Yamashita, Sherlock Licorish, Jens Dietrich, and Steve Counsell. Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 68–78, 2018.

[42] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[43] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 327–337, 2020.

[44] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th working conference on reverse engineering (WCRE)*, pages 242–251. IEEE, 2013.

[45] Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. An empirical study of design discussions in code review. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2018.

[46] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice*, 23(3):179–202, 2011.