Journal of Software Engineering Research and Development, 2024, 12:4, doi: 10.5753/jserd.2024.3554
This work is licensed under a Creative Commons Attribution 4.0 International License..

Investigating the Social Representations of Harmful Code

Rodrigo Lima [**Universidade Federal de Pernambuco** | ***rsl@cin.ufpe.br***]

Jairo Souza [**Universidade Federal de Pernambuco** | ***jrmcs@cin.ufpe.br***]

Baldoino Fonseca [**Universidade Federal de Alagoas** | ***baldoino@ic.ufal.br***]

Leopoldo Teixeira [**Universidade Federal de Pernambuco** | ***lmt@cin.ufpe.br***]

Rafael Mello [**Universidade Federal do Rio de Janeiro** | ***rafaelmello@dcc.ufrj.br***]

Márcio Ribeiro [**Universidade Federal de Alagoas** | ***marcio@ic.ufal.br***]

Rohit Gheyi [**Universidade Federal de Campina Grande** | ***rohit@dsc.ufcg.edu.br***]

Alessandro Garcia [**Pontifícia Universidade Católica do Rio de Janeiro** | ***afgarcia@inf.puc-rio.br***]

Context. Harmful Code denotes code snippets that harm the software quality. Several characteristics can cause this, from characteristics of the source code to external issues. By example, one might associate Harmful Code with the introduction of bugs, architecture degradation, and code that is hard to comprehend. However, there is still a lack of knowledge on which code issues are considered harmful from the perspective of the software developers community. **Goal.** In this work, we investigate the social representations of Harmful Code among a community of software developers composed of Brazilian postgraduate students and professionals from the industry. **Method.** We conducted free association tasks with members from this community for characterizing what comes to their minds when they think about Harmful Code. Then, we compiled a set of associations that compose the social representations of Harmful Code. **Results.** We found that the investigated community strongly associates Harmful Code with a core set of undesirable characteristics of the source code, such as bugs and different types of smells. Based on these findings, we discuss each one of them to try to understand why those characteristics happen. **Conclusion.** Our study reveals the main characteristics of Harmful Code by a community of developers. Those characteristics can guide researchers on future works to better understand Harmful Code.

Keywords: *Harmful Code, Social Representations*

1 Introduction

During software development, developers change the source code to implement new requirements, improve the code, or fix bugs. While those changes are inherent to software evolution, they may introduce code snippets that are likely to harm software quality (hereafter called Harmful Code (15)). In our study (15), we consider harmful a code snippet that contains smells and has one or more functional bugs reported in its history.

One might argue about different aspects of Harmful Code, such as code that introduces bugs, degrades the architecture, and is hard to comprehend. However, few studies focus on understanding the characteristics of Harmful Code (15; 23). The understanding of Harmful Code is beneficial for various reasons. First, it would contribute to guiding developers on preventing and combating the incidence of Harmful Code. Moreover, it would be more feasible to develop effective tools for Harmful Code detection. On the other hand, these benefits become harder to achieve if there is no understanding or consensus about Harmful Code among software developers.

Some studies have investigated harmful code by analyzing code smells that are more harmful to the software. To identify those smells, Olbrich et al. (23) analyzed which code smells change more frequently. In our previous work, we analyzed the existence and detection of Harmful Code in open source projects (15). This research considers a code snippet harmful if it contains smells and has one or more functional bugs reported in its history. However, despite the existence of related work based on the assumption that certain characteristics of the source code are harmful, we could not find previous studies investigating the understanding of Harmful Code from the practitioners' perspective. This understanding may reveal a core set of quality issues in the source code that may shed light on improving the practice.

This paper investigates the understanding of developers about Harmful Code. In particular, we conducted an empirical study to observe the social representations of Harmful Code in a community of software developers. For this purpose, we ground our research in the theory of social representations from Social Psychology (26). This theory considers that a concept that is collectively seen through the set of beliefs, values, and behaviours unconsciously shared by the members from particular communities, such as software developers from a particular country. Consequently, these representations influence how these members behave and communicate (21).

The first experiences with the theory of social representations in software engineering are recent (9; 8; 4). However, these experiences resulted in practical resources to improve the software development practice (9; 8; 4). Besides, the theory of social representations has been successfully used in different research fields (27; 19; 26; 17).

The findings of our investigation reveal a set of core issues of Harmful Code that goes beyond current research directions. Some of these issues address undesirable characteristics of the source code, including bad programming practices that harm software maintainability. However, our findings also reveal that the incidence of Harmful Code is a po-

tential source of functional and non-functional bugs that may directly harm the user experience in software. In this way, we discuss alternatives for preventing the incidence of Harmful Code.

The remainder of this paper is structured as follows. Section 2 introduces the theory of social representations and correspondent techniques used in this study. Section 3 presents the experimental study aiming at characterizing the social representations of Harmful Code. Section 4 shows the study results. Section 5 discusses the main findings of the study. Section 6 presents the threats to validity. Section 7 discusses previous work on social representations and Harmful Code and, finally, Section 8 concludes this work.

2 Theory of Social Representations

The theory of Social Representations (22) comes from the domain of social psychology. Based on this theory, social representations aim to establish an order that enables the members of a certain group (community) to guide themselves in their material and social world. In this way, social representation means the collective elaboration of a social object by a particular community for behaving and communicating (22). In this definition, a *social object* corresponds to an object socialized by two or more individuals from a community.

The theory of social representations has been applied to support research in diverse fields, including health (19), social development (17), education (27), and psychology (26). Recent works have applied the theory of social representations to investigate some social objects of Software Engineering, including the identification of code smells (9; 8) and confusing code (4).

A *community* can be any group of individuals sharing common values and culture, such as a software company. In this way, the social representations of Harmful Code in this company comprise the system of beliefs, values, and ideas shared by the company developers about this social object. This system establishes a code for social exchanging, naming, and classifying the different aspects of their world (22). For instance, let's suppose that developers from a certain company believe that novice developers are the authors of Harmful Code. This belief will influence how teams from this company deal with code reviews. For instance, this belief could lead to a more laborious approach to reviewing code implemented by these developers. Consequently, reviewers may feel less motivated to review the code produced by novice developers. Unfortunately, this preconception could lead to precipitately blaming these developers when Harmful Code is detected in the project. Once the company teams are aware of these representations, they can work on promoting clarifying actions to overcome this negative association made between harmful code and novice developers.

To characterize the social representations of a social object, researchers stimulate members from the investigated community to reveal what is in their unconscious, avoiding censoring their thoughts. For this purpose, individuals should perform free association tasks. Free association is a technique from psychoanalysis (3) based on asking individuals to quote what first comes to mind when they think about a particular social object. This question should be immediately answered, and quotations provided should be written down in the same uttered order, i.e., the order that came to the individual's mind. Then, the evoked terms are submitted to open coding, resulting in the set of *associations* to be further analyzed. For instance, *bug* may be an association of Harmful Code, resulting from grouping evoked terms such as *faults* and *defects*.

Figure 1 shows the procedure to characterize social representations of social objects through the following steps:

Step 1: Free Association Task. In this step, individuals freely express their thoughts (in our study, *terms*) related to a specific social object (in our study, Harmful Code). For example, in our study, we asked each participant to express up to five terms. While participant #6 expressed the words *Unnecessary Repetition, Bad Writing, Long Methods, and Non-intuitive variables* about Harmful Code, participant #25 expressed the words *Hard to Understand, badly structured, Non-modular, and Code Smells.*

Step 2: Clusterization. This step aims to cluster the terms expressed by the participants in associations. To do that, we follow the sub-steps. **Sub-step 2.1: Terms Clustering** In this sub-step, researchers individually analyze each term expressed by the participant and define associations to represent the term; and **Sub-step 2.2: Consensus Meeting** Each researcher defines the associations related to each term afterwards, they have a meeting to obtain consensus regarding the most representative association to the term. For example, in our study, the researchers defined the association *Code Smell* to represent the term *Unnecessary Repetition* of the participant #6 and the association *Lack of Comprehension* to represent the term *Hard to understand* of the participant #25.

Step 3: Frequency & AOE Computation. After coding the associations, they should be submitted to a comparative analysis. The rank-frequency method is a common approach for analyzing the corpora of associations obtained during free association tasks (3; 16). This ranking is performed based on the frequency of the association and the average order of their evocation or appearance (AoE). While frequency helps to identify how common the associations are made, AOE helps to identify how promptly the associations come to individuals' minds (3). The frequency of an association (F) is calculated by counting how many times the community evoked its corresponding terms. For instance, suppose that the association *Bugs* is composed of the term *bugs*, evoked 14 times, *Not working correct*, evoked once, *Crash*, evoked once, *Incorrect*, evoked once, *Fixes*, evoked once, and *failures*, evoked 3 times. Thus, the frequency of this association is 21. The AOE of an association is calculated by the sum of the orders in which the subjects had evoked the corresponding terms, divided by the frequency of this association. A low AOE value for an association means that the group makes the association promptly. For instance, let us consider the *Bugs* association example. Supposing that the term *bugs* was evoked 13 times in the first place, five times in the second place, and three times in the fourth place, the AOE of the association *Bugs* is 1.52, i.e., $((13 * 1) + (5 * 2) + (3 * 3)/21)$.

Step 4: Four Zones Distribution. In this step, we allocate the associations across four zones (see Figure 2) to identify the strongest and the weakest associations made by partici-

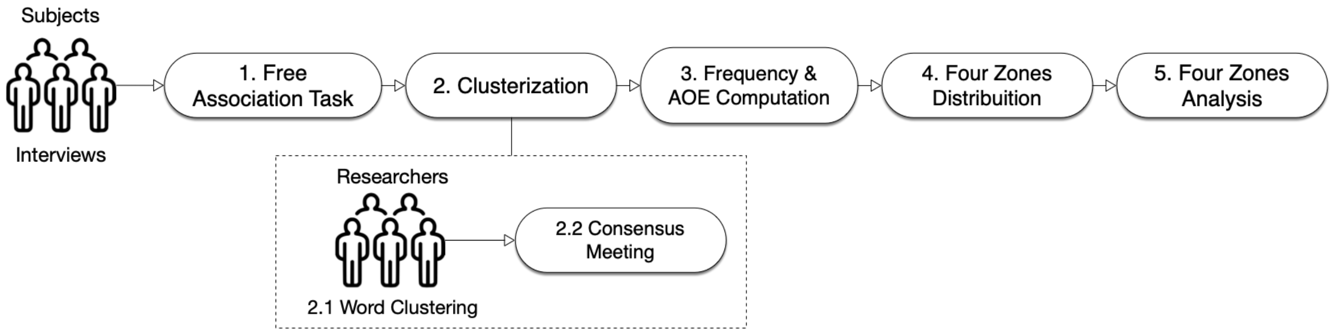

Figure 1. Social Representation characterization procedure

Central System Frequency equal or higher than mean AOE lower than average (more important and more frequent associations)	**Potential Changing Zone** Frequency equal or higher than mean AOE equal or higher than average (less important and more frequent associations)
Potential Changing Zone Frequency lower than mean AOE lower than average (more important and less frequent associations)	**Periphery** Frequency lower than mean AOE equal or higher than average (less important and less frequent associations)

Figure 2. The four zones of the social representations based on the rank-frequency method.

pants based on their frequency (F) and AOE values to discern the most and least significant associations, as per the recommendations of studies on social representations (13). This step is critical in identifying which associations are at the core of the social representations and which are on the periphery. To support this distribution, we calculate the following thresholds: the mean frequency of the associations, and the average AOE, i.e., the ratio between the sum of the AOEs and the sum of the frequencies of the associations (3). For instance, in our study, we have a mean frequency of 21 and an average AOE of 2.58. Thus, we allocated the association *Bugs* to the Central system zone since this association presents a frequency equal to 21 and an AOE equal to 1.52;

Step 5: Four Zones Analysis. In this step, we examine these zones. The central system zone, which represents the core elements of social representations, contains associations with higher frequencies and lower AOEs, signifying their prominence and promptness in the minds of individuals. The potential changing zones indicate associations that are either frequently evoked but not as promptly, or vice versa, suggesting areas of potential change. Meanwhile, the periphery embodies the least frequent and less promptly evoked associations, thus considered irrelevant in the social representations framework (13).

3 Study Design

During software development, developers are continuously affected by issues related to the quality of the source code. Some of these issues may harm software development and maintenance, while others may harm its use. Therefore, we should understand Harmful Code from the perspective of its authors, i.e., the developers, able to recognize its incidence and assess its impact. The collective knowledge built and shared among developers is a relevant source for establishing a comprehensive understanding of Harmful Code. In this way, the goal of our study is *to characterize the social representations of Harmful Code from the perspective of software developers*. To do that, we aim to answer the following research question (RQ):

RQ. What are the social representations of Harmful Code by software developers?

By answering RQ, we want to identify the core set of values, ideas, and beliefs collectively built by developers about Harmful Code, which results in their social representations. For this purpose, we should first establish a community of developers to investigate. In this way, investigating social representations in a country-wide community is a common practice. Among others, members of a country-wide community are influenced by unique cultural aspects that influence how individuals see and classify social objects (5). Hence, we opted to investigate Brazilian software developers, working with software development in Brazil or abroad.

3.1 Instrumentation

For each participant, we applied a free association task, composed of the following stimulating question:

> *What immediately comes to mind when you think about Harmful Code? Please provide up to five words in the order they come to your mind.*

Before applying the task, we oriented each participant on avoiding overthinking their answers. We arbitrarily established a limit of 10 seconds as a threshold for considering an answer as immediately given. Besides, we assure that the participants were unaware of the specific research topic before their participation. The procedure of asking for up to five words is a common practice in free association tasks (9), allowing to gather the associations made at different relevance levels (3). After the stimulating question, the researcher applied the following open question:

> *Which word do you understand to be the most relevant? Please justify your answer.*

We applied this question to stimulate the respondents to give conscious and well-formed arguments about the evoked terms. These arguments help us to better understand the context in which the developers evoked these terms. However, since we are using the rank-frequency method, we do not use

this content to modify the order of relevance reported during the free association task.

After the free association task, the study participants were invited by email to answer a characterization questionnaire[1]. Table 1 summarizes the questions asked. Some of these questions are intentionally redundant to capture the developers' diversity of profiles from different perspectives. We observed that C++ or Java developers are the ones most expressing the terms related to the associations in the central system. Particularly, Java developers tend to express the term code smell when they think about harmful code. We used a four-level *Likert* scale (Very Low, Low, High, Very High) to characterize the self-perception of the developers, we choose this scale to prevent participants from choosing a neutral option.

3.2 Data Analysis

First, for each participant, a researcher (1st author) listened to the recording and transcribed the answers to a spreadsheet. Then, three researchers performed a separate coding of the evoked terms (1st author, 2nd author, and 5th author). The main goal of these analyses is to identify opportunities for clustering terms with similar meanings into a single association. After concluding the individual coding activities, one researcher composed the final set of associations based on the agreements found. Finally, the researchers discussed the disagreements in a single meeting until reaching a consensus.

Then we normalized the corpus responses by excluding repeated terms of the same association for the same participant. For instance, let us suppose that one developer quoted *Bugs*, followed by *Defects*. Although these are different terms, they were clustered into a single association (*Bugs*), counted a single time. Therefore, as recommended in the social representation analysis, the second evocation should be discarded. After reaching the set of associations, we discarded those made by a single participant. Finally, we applied the rank-frequency method (see Section 2) for distributing the associations among the four zones. More details about the process can be found on the spreadsheet.[2]

3.3 Execution

A single researcher performed the free association task from August to October 2020. Due to the COVID-19 pandemic, the task was performed through individual online meetings with video sharing. All of the study participants agreed with the recording of their participation. Among others, we used these recordings to double-check whether the participants immediately answered the stimulating question. In total, 53 software developers participated in the study, evoking 155 terms. We used video calls so the researcher could check that participants did not have external distractions during the free association task. We also checked that all participants promptly answered the free association task. Therefore, we did not discard any evoked terms in the analysis.

[1] https://bit.ly/harmful-code-characterization-form
[2] http://bit.ly/harmful-code-associations

3.4 The community

After analyzing the answers given to the characterization questions, we found diversity in the sample, counting with the participation of 53 software developers. These individuals are Brazilian developers who work in companies inside and outside Brazil. The sample is composed of 4 developers working for a software house based in San Francisco (USA). This company has a medium size and develops software for clients worldwide. Another 5 developers belong to a New York-based startup that develops a training platform for English exams. The other 44 developers work in companies based in Brazil or are currently in Academia (in Brazil).

At the time of the study execution, most of the study participants were currently playing the role of software developers. However, there are also project managers, tech leaders, lecturers, and researchers, among others. Most of the study participants hold Bachelor's degrees (36,9%), while others are undergraduate students (18%) or hold Master's degrees (42,1%).

The sample of Brazilian developers investigated is composed of professionals having diverse backgrounds. Several developers have experience in other fields of computer science, such as data science, artificial intelligence, and human-computer interaction. From the 53, 80% has at least three years of experience in software development, indicating the predominance of experienced developers in the sample. Most of them declared having a high experience level (65%), represented by an average experience of 4.66 years/nine projects and a median experience of four years/six projects. The four programming languages they more frequently mentioned are JavaScript (87.50%), Python (87.50%), Java (80%), and PHP (42.50%). All the other programming languages comprise less than a quarter of the sample.

Background in code smell identification. 71% of the developers declared familiarity with the concept of code smells. Regarding their experience in code smells, most of these developers predominantly declared having a low (33%) or high (43%) experience level, represented by an average experience of 2.03 years/three projects and a median experience of two years/two projects. Moreover, 13% of the participants declared a very high experience in code smell identification tasks. Only three developers declared no previous experience in code smell identification.

Background in bug fixing. We also asked developers about their experience in bug fixing, where 100% of the developers declared some experience in bug fixing, with an average of 6 years and 12 projects fixing bugs. Despite that, 79% declared high experience in bug fixing, and 10% declared a very high experience.

4 Results

The 53 developers evoked 155 terms. We found a total or partial agreement among the researchers on coding 78.21% of these terms. After a 2-hour meeting involving the three researchers responsible for the coding activities, the remaining disagreements were solved. Besides, the final name of each association was established. The coding activities resulted in a distribution of the 155 terms evoked among 53 associations.

Table 1. Items of the Characterization Questionnaire

Questionnaire Item	Type of Answer
Highest academic degree	Nominal
Are you currently working in the industry? If so, what is your current role?	Open
In the following lines, briefly summarize your experience with software development.	Open
What are the programming languages are you familiar with?	Open
How do you perceive your background in Software Development?	Likert scale
How do you perceive your background in the identification of code smells?	Likert scale
How do you perceive your background in bug fixing?	Likert scale
Number of software development projects	Numeric
Experience in software development in years	Numeric
Experience in code smells identification in years	Numeric
Number of software development projects in which you have identified code smells	Numeric
Experience with bug fixing in years	Numeric
Number of software projects in which you participated fixing bugs	Numeric

After allocating each original term with its corresponding associations, we identified 25 cases in which the same association was made two or more times by the same developer. After discarding these repetitions, we found that 5 associations appeared only once in the whole data set. Thus, they were also discarded. Therefore, 23 associations remained in the data set.

We applied the ranking-frequency method over the 23 associations, resulting in the four zones of Harmful Code social representations presented in Table 2. Associations with the frequency (F) equal or higher than 7 and with AOE (Average order of their evocation or appearance) lower than 2.58 compose the central system of the social representations. There are also the associations located in the periphery, which should be discarded.

The **central system** of the social representations reveals that developers strongly associate Harmful Code with the incidence of two concrete issues while developing software, i.e., functional *Bugs* and different types of *Code smells*. The developers also strongly associate Harmful Code with two non-functional quality issues: *Lack of Performance* and *Lack of Security*. Finally, we also found that developers strongly believe that Harmful Code is a source of *Waste*.

Besides the issues mentioned on the central system, we can see that other non-functional issues appear in the **potential changing zones**: *Lack of Comprehension*, *Lack of Modularization*, and *Lack of Tests*. The potential changing zones also reveal that developers associate Harmful Code to the implementation of *Workarounds* that may lead to the *Rework* of resources. Besides, other associations include *Lack of Maintainability*, *Lack of Standards*, and *Lack of Exception Handling*. In this way, the associations made with *Code Smells*, *Lack of Comprehension*, *Lack of Modularization*, *Lack of Maintainability*, *Waste*, *Rework*, and *Bugs* strengthen the perception that code smells may affect the comprehensibility, modularization, and maintainability of the source code and, consequently, leading to bugs, rework, and waste.

The community of developers investigated shares a strong belief that Harmful Code is associated with code smells and bugs. This suggests that the mitigation of code smells and bugs may be intricately linked to the prevention and combat of harmful code. In particular, code smells and bugs associated with performance, security, and exception handling. Fortunately, the literature offers a variety of tools and techniques for detecting code smells and bugs. Our research sheds light on the potential effectiveness of using a combination of these tools to tackle harmful code.

5 Discussion

The findings of our study reveal five main characteristics of Harmful Code. The first five of these characteristics address issues that may also be considered harmful for the system users. **The first** one is the incidence of *code smells*. For example, a common code smell is the God Class, a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. God Classes can rapidly grow out of control, making it increasingly harder for developers to understand them in the process of bug fixing and adding new features. The incidence of *code smells* is a great concern, as the second most frequently evoked term (45.28%). Code smells typically require extra reading and comprehension effort from developers (1), leading them to search for support in the code documentation. In this way, the incidence of coarse-grained code smells such as God Class and Spaghetti Code is frequently associated with modularity problems in the system implementation. The strong association of code smells with Harmful Code is justified in different ways, such as *Subject #07:* "...because they are usually structural problems and make understanding difficulty", and *Subject #18:* "...because we usually identify the Harmful Code with them.".

The second main characteristic of Harmful Code is the *incidence of functional bugs*. One may see that strongly associating Harmful Code to functional bugs is somehow expected, since these bugs typically lead to the system malfunctioning, which is harmful to the system users. Consequently, bug fixing is a natural priority in software maintenance activities. In our study, a third of the participants concluded that bug is the more relevant term they evoked. Their arguments for this choice include *Subject #03: "Bugs interfere in the system functionality"* and *Subject #08:* "Bugs create an anomaly". Besides, one developer argued that *Subject #20:* "Bugs affect the most important user of the system, that is the client". Despite one may expect that the incidence of bugs would be a strong association, it is important to note that only (41%) of the developers evoke from their unconscious that bugs as the greatest source of harmfulness. Thus, most of the developers (59%) did not associate Harmful Code with functional bugs. In this way, the findings of our study suggest that the Brazilian software development community believes that other issues in the source code may be as Harmful as bugs.

The aforementioned two first main characteristics of Harmful Code address the definition of harmful code proposed in (15), in which we define Harmful Code as a source

Table 2. Four zones of Harmful Code social representations by developers.

Central system (F≥7, AOE <2.58)		**Potential Changing Zone (F≥7, AOE ≥2.58)**	
Name(F)	**AOE**	**Name(F)**	**AOE**
Code Smells (24)	2.00	*Lack of Comprehension* (8)	2.75
Bugs (21)	1.52	*Lack of Tests* (7)	2.71
Waste (10)	2.50	*Lack of Modularization* (7)	2.71
Lack of Performance (8)	1.88		
Lack of Security (8)	2.38		
Potential Changing Zone (F<7, AOE<2.58)		**Periphery (F<7, AOE≥2.58)**	
Name(F)	**AOE**	**Name(F)**	**AOE**
Lack of Maintenability (6)	1.17	*Badly Written* (6)	2.67
Rework (6)	2.33	*Lack of Readability* (6)	2.67
Difficulty (4)	2.50	*Bad Documentation* (6)	3.29
Lack of Standards (4)	2.00	*Refactoring* (3)	2.67
Work around (3)	2.33	*Debugging* (3)	3.00
Lack of Exception Handling (2)	2.50	*Bad Names of Code Elements* (3)	4.00
		Lack of Communication (2)	3.00
		Lack of Scalability (2)	3.00
		Legacy Code (2)	3.00

code snippet containing smells and one or more (functional) bugs. However, the results of our investigation on the developers' social representations indicate the need for expanding this definition. **The third** main characteristic of Harmful Code is the *Waste*. Based on the developers' arguments, the word "Waste" in this study clustered many evoked terms with converging meanings, such as "Loss", "Cost", "Project Damage", "Headache" and "Finance Problem". One may think that these terms are not related to technical activities, since they are mainly related to negative feelings and financial issues. We interpret that the developers evoked these terms as they believe that harmful code is a source of waste for its maintainers and supporters, directly harming software development productivity.

The fourth main characteristic of Harmful Code is the *lack of performance*. This aspect highlights how inefficient or poorly optimized code can significantly impact the overall functionality and user experience of a system. Performance issues often manifest as slow response times, increased resource consumption, and diminished throughput, which are detrimental to both the user's experience and the system's reliability (Story).

The fifth main characteristic of Harmful Code is the *lack of security*. Security can be related to the information and services being protected, the skills and resources of adversaries, and the costs of the potential assurance remedies; security is an exercise in risk management (25). In this context, developers associated Harmful Code with the lack of security, mainly due to the perceived harmfulness of security breaches to systems' users. For instance, the developers argued that: *Subject #16:* "Security Failures, because if someone is able to explore the failures, it will directly impact the users' information and the application work", *Subject #16:* "Vulnerabilities, because we work with users' data, if this gets leaked the whole company is damaged."

Considering the mentioned main characteristics of Harmful Code, one can see that the incidence of functional bugs and code smells are recurrent concerns that are strongly connected. While developers fixing bugs should avoid harming the source code design, developers removing code smells should attempt to preserve the original system behavior. In both cases, developers should have in mind the need to preserve or even enhance the systems' security and performance. For instance, validating the incidence of a certain code smell includes checking whether potential improvements in the source code structure would not harm the system performance (6). Next, we discuss other potentially relevant characteristics of Harmful Code identified in our study.

Lack of Maintainability. Maintainability is an internal software quality attribute, so it does not directly harm users, but may significantly harm the developers' work. Lack of maintainability generates *waste* of development efforts, leading to *rework*, which is another main characteristic of Harmful Code. This lack of maintainability may be caused by several issues, which include code smells (another main characteristic). For large systems, the maintenance phase tends to have a comparatively much longer duration than all other previous life-cycle phases taken together, resulting in more effort (2). Developers that associated maintenance issues with Harmful Code include *Subject #32:* "The code will be harmful because, it will take much more time to maintain, turning it harmful.", *Subject #37:* "It can not only break the part related to the code itself, but it can also break other parts of the software, things that were working perfectly before.", and *Subject #38:* "Because let's say you are looking for the part of the code that is making it slow, then you will spend a lot of time searching for it.".

Lack of Modularization and Lack of Standards. These characteristics address undesirable technical aspects of the source code some developers promptly associate with harmful code. For instance, developers argued that: *Subject #05:* "If a code is not properly modularized, it is very difficult to refactor it to a better code", and that *Subject #29:* "The lack of standards is the worst problem because you don't know what

the code action will do". Software systems are prone to repeated debugging and feature enhancements throughout their evolution. In this way, several studies reveal that large-scale software systems tend to gradually deviate from the original architecture, and might deteriorate into unmanageable monoliths (29). In this context, the lack of modularization and the lack of standards may be considered a headache for software developers, once they require extra effort from developers to reach a proper comprehension. In recent work, it was found that developers from different Brazilian companies strongly associated the lack of modularization with confusing code (4).

Lack of Comprehension and Difficulty. Some developers promptly associated harmful code with badly written code and difficult to understand. To justify this association, they raised arguments such as *Subject #06:* "badly written code... it is very hard to understand" and *Subject #17:* "...if the code was badly written, it probably will put at risk the code quality and security". Badly written code directly harms software maintenance activities once developers should commonly understand code elements they did not implement. Consequently, developers typically face several issues and challenges to reaching proper program comprehension and performing their maintenance activities. Not rarely, badly written code results in confusing code, which may be composed of different characteristics, such as long lines, abusive nesting, and bad indentation (4; 12) identified that poor quality lexicon impairs program comprehension, increasing developers' efforts to perform maintenance activities. Lavaliée et al. (14) investigated the impact of organizational values on software architecture and code quality. They reported the case of a company in which frequent changes to product priorities affect the code, resulting in the incomplete implementation of software changes, leaving dead code and code fragments.

Rework. It addresses a main side-effect for developers. This characteristic is directly associated with development costs once companies estimate the projects considering the substantial effort associated with rework in code (20). Fairley and Wilshire (11) described different types of rework, including evolutionary rework, avoidable retrospective rework, and avoidable corrective rework. The different types are associated with different effects on quality and productivity. Developers who associated Harmful Code with rework argued that: *Subject #50:* "Rework, it already happened a lot with me, delivers a product fast with bugs because we had deadlines and after rework on it.", and *Subject #21:* "Rework and understand the whole code again to rework on it".

Workaround. It is usually related to temporary fixes and temporary solutions for critical problems. Once software systems are formalized collections of knowledge rather than physical artifacts, the software development process allows its authors to follow shortcuts (32). Even undesirable, these shortcuts often remain in the source code without being refactored or evolved. Consequently, workarounds are prone to increase the technical debt of software systems. Software developers are commonly experienced in creating and dealing with workarounds, recognizing their potential to deliver Harmful Code. Workarounds to solve technical issues often result from conscious decisions to address time-to-market pressure (32). However, stakeholders may not be familiar with the side effects of workarounds, including additional costs and poor quality (32).

6 Limitations and Threats to Validity

In this section, we present the threats to validity. From the perspective of social representations and social psychology, each community of individuals preserves unique cultural aspects that reflect how their members behave despite its size and coverage (21). Thus, limitations for generalizing social representations are expected. To mitigate this issue, we opted to narrow our focus to characterize the social representation of developers from a specific country but including individuals working in different countries, which enriches the sample with the influence of cultural diversity.

An internal threat to validity addresses possible issues in applying the free association task, once the participants should immediately evoke what comes to their minds. In this sense, we count on the support of a researcher with experience in social representations for identifying the best practices for performing the free association task. This support includes (i) setting an affordable environment for data gathering through video conferences mainly due to the COVID-19 pandemic; (ii) how introducing the study to the participant since the invitation to avoid "spoilers" that would harm the free association task; and (iii) how to guide the interviewee to immediately answer after listening to the stimulating question.

As in any qualitative study, researchers' background may strongly influence data analysis. We adopted different strategies to mitigate this bias. First, we applied data analysis techniques frequently used and recommended for social representations analysis (Section 2). Second, we involved specialists in social representations for supporting data analysis. Third, researchers from different software engineering research groups were involved in the data analysis. These researchers have experience in topics addressing program comprehension, code smells, and software maintenance. As described in Section 3, three researchers coded the terms evoked separately. These researchers discussed all divergences found in their codes through a single meeting.

7 Related Work

Although we could not find previous work investigating social representations of Harmful Code, there are reported studies on the social representations of other software development topics: Mello et al. (4) investigated the social representations of confusing code among two distinct communities of software developers from industry. They conducted free association tasks with the developers to characterize their minds about confusing code. Then they compiled and classified associations composing the social representations of confusing code by each community. The results of them showed that developers of both communities strongly associate confusing code with a common set of undesirable characteristics of the source code, such as different types of code smells

and bad naming of code elements. Another study from Mello et al. (10) conducted an empirical study on the social representations of smell identification by two communities. One community is composed of postgraduate students from different Brazilian universities. The other community is composed of practitioners located in Brazilian companies. One of the key findings is that the community of students and practitioners has relevant differences in their social representations. Students shared a strong belief that smell identification is a matter of measurement, while practitioners focus on the structure of the source code and its semantics. The investigation with the community of software developers was then expanded (5), revealing a clear gap between code smell research and practice (7). The investigation of the developers' social representations contributed to the authors composing evidence-based recommendations for developers identifying code smells (6). The latest work from Mello et al. (18) made a compilation of the previous works and provided practical advice for development teams planning and optimizing their efforts in identifying code smells.

Palomba et al. (24) conducted a survey to investigate developers' perception on bad smells, they showed to developers code entities affected and not by bad smells, and asked them to indicate whether the code contains a potential design problem, nature, and severity. The authors learned the following lessons from the results: I. There are some smells that are generally not perceived by developers as design problems. II. The instance of a bad smell may or may not represent a problem based on the "intensity" of the problem. III. Smells related to complex/long source code are generally perceived as an important threat by developers. IV. Developer's experience and system's knowledge pay an important role in the identification of some smells. Sae-Lim et al. (28) investigated professional developers to determine the factors that they use for selecting and prioritizing code smells. They found that *Task Relevance* and *Smell Severity* were most commonly considered during code smell selection, while *Module Importance* is employed most often for code smell selection. Souza et al. (30) investigated the developers' viewpoints on the relevance of certain assumptions to avoid bug-introducing changes. In particular, they analyzed which assumptions developers can make during software development. As a result, they identified five viewpoints among developers regarding their assumptions around bug-introducing changes.

In our work (15), we presented a study to understand and classify code harmfulness. First, we analyzed the occurrence of Clean, Smelly, Buggy, and Harmful code in open-source projects as well as which smell types are more related to Harmful Code. Further, we investigated to which extent developers prioritize refactoring Harmful Code. We also evaluated the effectiveness of machine learning techniques to detect Harmful and Smelly code. Finally, we investigated which metrics are most important in Harmful Code detection. As a result, we found that even though we have a high number of code smells, only 0.07% of those smells are harmful. Also, we performed a survey with 60 developers investigating to which extent developers prioritize refactoring Harmful Code. The majority (53.8%) of the developers prioritize code associated with bugs, and most of them (30%) prioritize Harmful Code when refactoring.

8 Conclusions

This paper presented an investigation of social representations of harmful code. We conducted an empirical study with 53, using the theory of social representations (21). The results shed light on the challenge of identifying Harmful Code, revealing its main characteristics from the perspective of a particular community. We understand that reaching a comprehensive characterization of Harmful Code is essential to prevent and combat the incidence of Harmful Code in software systems. In this work, we characterized the social representations of Harmful Code from a sample of Brazilian software developers. The findings of our study reveal an initial set of core issues of Harmful Code, discussed in the paper. This set includes Bugs, Code Smells, Lack of Performance, Lack of Security, and potentially other technical issues also discussed. We understand that our findings may be used for driving future investigations on the identification of Harmful Code in software systems. In future work, we intend to refine the initial set of core issues found by replicating our study in distinct communities of software developers.

9 Data Availability

The datasets generated during and/or analyzed during the current study are available.[3]

Acknowledgements

This work is partially supported by FAPEAL (60030.0000002725/2022, 60030.0000000161/2022), FACEPE (APQ-0570-1.03/14, APQ-0399-1.03/17), CNPq (423125/2021-4, 315532/2021-1, 310313/2022-8, 404825/2023, 442884/2023, 315711/2020-5, 141276/2020-7, 141054/2019-0, 465614/2014-0), CAPES (88887.136410/2017-00, 88887.373933/2019-00), PRONEX (APQ/0388-1.03/14), FAPERJ (200.510/2023, 010002285/2019, 211.033/2019, 202621/2019), IEEA-RJ (001/2021), the Alexander Von Humboldt Foundation.[4], and INES (www.ines.org.br). The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions.

References

[1] Abbes, M., Khomh, F., Guéhéneuc, Y. G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension. *CSMR*, pages 181–190.

[2] Aggarwal, K. K., Singh, Y., and Chhabra, J. K. (2002). An integrated measure of software maintainability. *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 235–241.

[3] https://bit.ly/harmful-code-dataset

[4] https://www.humboldt-foundation.de/

[3] Dany, L., Urdapilleta, I., and Lo Monaco, G. (2015). Free associations and social representations: some reflections on rank-frequency and importance-frequency methods. *Quality and Quantity*, 49(2):489–507.

[4] de Mello, R., da Costa, J., de Oliveira, B., Ribeiro, M., Fonseca, B., Gheyi, R., Garcia, A., and Tiengo, W. (2021). Decoding confusing code: Social representations among developers. In *2021 2021 IEEE/ACM 13th CHASE*, pages 11–20, Los Alamitos, CA, USA. IEEE Computer Society.

[5] De Mello, R., Oizumi, W., Uchôa, A., Souza, J., Oliveira, R., Fonseca, B., Oliveira, D., and Garcia, A. (2019). Investigating the social representations of the identification of code smells by practitioners and students from Brazil. *ACM International Conference Proceeding Series*, pages 457–466.

[6] de Mello, R., Oliveira, R., Uchoa, A., Oizumi, W., Garcia, A., Fonseca, B., and de Mello, F. (2022). Recommendations for developers identifying code smells. *IEEE Software*, (01):2–10.

[7] de Mello, R., Uchôa, A., Oliveira, R., Oizumi, W., Souza, J., Mendes, K., Oliveira, D., Fonseca, B., and Garcia, A. (2019a). Do research and practice of code smell identification walk together? a social representations analysis. In *2019 ACM/IEEE ESEM*, pages 1–6. IEEE.

[8] de Mello, R., Uchôa, A., Oliveira, R., Oizumi, W., Souza, J., Mendes, K., Oliveira, D., Fonseca, B., and Garcia, A. (2019b). Do research and practice of code smell identification walk together? a social representations analysis.

[9] de Mello, R., Uchôa, A., Oliveira, R., Oliveira, D., Fonseca, B., Garcia, A., and Mello, F. (2019c). Investigating the social representations of code smell identification: A preliminary study.

[10] de Mello, R. M., Uchoa, A. G., Oliveira, R. F., de Oliveira, D. T. M., Fonseca, B., Garcia, A. F., and de Mello, F. d. B. (2019d). Investigating the social representations of code smell identification: a preliminary study. In *2019 IEEE/ACM 12th CHASE*, pages 53–60. IEEE.

[11] Fairley, R. and Willshire, M. (2005). Iterative rework: the good, the bad, and the ugly. *Computer*, 38(9):34–41.

[12] Fakhoury, S., Ma, Y., Arnaoudova, V., and Adesope, O. (2018). The effect of poor source code lexicon and readability on developers' cognitive load. *Proceedings - ICSE*, pages 286–296.

[13] Ferrara, M. and Friant, N. (2016). The application of a multi-methodology approach to a corpus of social representations. *Quality and Quantity*, 50(3):1253–1271.

[14] Lavallée, M. and Robillard, P. N. (2015). Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. *Proceedings - International Conference on Software Engineering*, 1(May):677–687.

[15] Lima, R., Souza, J., Fonseca, B., Teixeira, L., Gheyi, R., Ribeiro, M., Garcia, A., and de Mello, R. (2020). Understanding and detecting harmful code. In *SBES*, SBES '20, page 223–232, New York, NY, USA. Association for Computing Machinery.

[16] Lo Monaco, G., Piermattéo, A., Rateau, P., and Tavani, J. L. (2017). Methods for studying the structure of social representations: A critical review and agenda for future research. *Journal for the Theory of Social Behaviour*.

[17] Maloletko, A. (2018). HOW DOES THE RUSSIAN YOUTH PERCEIVE CORRUPTION. (March):2018.

[18] Mello, R. d., Oliveira, R., Ucha, A., Oizumi, W., Garcia, A., Fonseca, B., and Mello, F. d. (2023). Recommendations for Developers Identifying Code Smells. *IEEE Software*, 40(2):90–98.

[19] Morant, N. (2006). Social representations and professional knowledge: The representation of mental illness among mental health practitioners. *The British journal of social psychology / the British Psychological Society*, 45:817–38.

[20] Morozoff, E. (2010). Using a line of code metric to understand software rework. *IEEE Software*, 27(1):72–77.

[21] Moscovici, S. (1988a). Notes towards a description of social representations. *European Journal of Social Psychology*, 18:211 – 250.

[22] Moscovici, S. (1988b). Notes towards a description of social representations. *European Journal of Social Psychology*, 18:211 – 250.

[23] Olbrich, S. M., Cruzes, D. S., and Sjoøberg, D. I. (2010). Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. *ICSM*.

[24] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., and De Lucia, A. (2014). Do they really smell bad? A study on developers' perception of bad code smells. *ICSME 2014*, pages 101–110.

[25] Potter, B. and McGraw, G. (2004). Software security testing. *IEEE Security and Privacy*, 2(5):81–85.

[26] Pozzi, M., Fattori, F., Bocchiaro, P., and Alfieri, S. (2014). Do the right thing! a study on social representation of obedience and disobedience. *New Ideas in Psychology*, 35:18 – 27.

[27] Räty, H., Komulainen, K., and Hirva, L. (2012). Social representations of educability in finland: 20 years of continuity and change. *Social Psychology of Education*, 15.

[28] Sae-Lim, N., Hayashi, S., and Saeki, M. (2017). How do developers select and prioritize code smells? A preliminary study. *ICSME 2017*.

[29] Sarkar, S., Ramachandran, S., Kumar, G. S., Iyengar, M. K., Rangarajan, K., and Sivagnanam, S. (2009). Modularization of a large-scale business application: A case study. *IEEE Software*, 26(2):28–35.

[30] Souza, J., Lima, R., Fonseca, B., Cartaxo, B., Ribeiro, M., Pinto, G., Gheyi, R., and Garcia, A. (2022). Developers' viewpoints to avoid bug-introducing changes. *IST*, 143:106766.

[Story] Story, J. The impact of low-quality code — ten10.com. `https://ten10.com/blog/the-impact-of-low-quality-code/`. [Accessed 28-01-2024].

[32] Yli-Huumo, J., Maglyas, A., and Smolander, K. (2015). The benefits and consequences of workarounds in software development projects. *Lecture Notes in Business Information Processing*, 210:1–16.