A SYSTEMATIC LITERATURE REVIEW: EFFECT OF CODE SMELLS ON NON-FUNCTIONAL ATTRIBUTES OF SOURCE CODE

Nazira Munalbayeva
University of Salerno
Salerno, Italy
n.munalbayeva@studenti.unisa.it

ABSTRACT

Context: Code smells adversely effect on software quality and impede its maintenance and development. Therefore, in recent years, in the software community, this topic has become relevant for research.
Objective: The main goal of this work is to study current researches and determine the relationship between smells and non-functional attributes.
Method: We had an overview about code smells effect on non-functional attributes, and what tools and techniques have been used to study this relationship. We studied 21 primary studies in detail and synthesized results.
Results: Maintainability, performance, and security are more affected by code smells. The code smells that most affect different quality non-functional attributes are GodClass, LongMethod, SpaghettiCode and FeatureEnvy. GodClass.

Author Keywords

Systematic Review, Secondary Studies, Code smells, Anti-patterns, Non-functional attributes, Non-functional requirements

INTRODUCTION

Non-functional requirements (NFR) are recognized as very important factors for software projects [1], [2], [3]. If the NFRs are not properly resolved, a number of potential problems may arise. For example, software may be of poor quality, or developers may spend more time and money to fix software errors [1]. One reason for poor quality is code smells. Code smells refers to an anomaly in the source code that demonstrates a violation of basic design principles, such as encapsulation, modularity, abstraction, and hierarchy [4]. They appear in poor design and poor implementation methods. Code smells are the main reasons indicating possible flaws in the development of a software system. [5]. There are many types of code smells that can affect a class, a separate method, their group, or an entire subsystem. Also, they can impact on non-functional requirements, such as testability [28], maintainability [27] [8], comprehensibility [22], performance, reliability [29] and etc. In recent years, code smells have been studied for various reasons. The main reason is the negative impact on software quality. Software quality is an important aspect, therefore there is a large number of research works in this area. A large number of scientific papers require analysis and synthesis. Analyzing and synthesizing available information can help the software community understand existing knowledge and identify problems. There have been a few attempts to understand current practices and provide an overview of the existing knowledge about software smells. Singh et al. [11] provide a systematic literature review (SLR) on code smells and refactoring in object-oriented software systems. The review covers 238 primary studies. The study focuses on methods and tools for detecting code smells. Also in this review, methods and tools for their refactoring were analyzed. Similarly, Zhang et al. [12] reviewed the studies from 2000 to 2009. They find such gaps in the literature: modern studies choose a small amount of smells for their research, some smells are poorly studied by the community, such as message chains. In addition, the study emphasizes that the effects of code smells are poorly understood. Therefore, Cairo et al. [13] reviewed the impact of code smells on Software Bugs. The review consisted of references over the past decade to this topic. Based on the results of the research was discovered the influence of code smells on the appearance of various software bugs. Also, Sharma et al. [14] investigated smell-related resources published between 1999 and 2016. They presented the available knowledge in a generalized form. They classified detection methods of smell. The authors concluded that currently available tools can detect only a very small amount of smells. After reviewing many articles, we noticed that there was no systematic review of the effect of code smells on NFAs. Therefore, we began to conduct this SLR to reduce research gaps. In this study, we examine the effects of code smells on NFAs of source code. Our goal is to identify and analyze how code smells affect to NFAs. The rest of the article is organized as follows: Section 2 - presents the research methodology. Section 3 - provides answers to research questions. Section 4 - Threats to validity, Section 5 - outlines the limitations of this work, and section 6 - outlines the conclusions of the systematic review.

CHI'16, May 07–12, 2016, San Jose, CA, USA

ISBN 123-4567-24-567/08/06...$15.00

DOI: http://dx.doi.org/10.475/123_4

RESEARCH METODOLOGY

We planned, conducted, and reported the review by following the SLR process suggested by Kitchenham et all. [20]. The protocol for this review was developed during the planning phase of this SLR. The protocol includes six stages: research questions, search strategy design, study selection, quality assessment, data extraction, and data synthesis. After the formation of the review protocol, a number of steps were taken in the review. At the first stage, we formed research questions that must be answered in the SLR. In the second step, we described a search strategy, including the search terms and the selection of sources to identify primary research. The third step is to identify relevant studies based on research questions. This step also defines the inclusion and exclusion criteria for each primary study. At the next stage, we determine the criteria for assessing quality by creating a quality assessment questionnaire for analysis and evaluation of studies. The final steps involve the data extraction process to collect the necessary information to answer research questions, and we also in this step develop methods for data synthesis. In the following Subsection we will present the details of the process.

Figure 1. Systematic literature review process.

Research questions

With this project, we want to investigate how much code smells impact on a system, and which are their effects on non-functional requirements. In order to achieve our goal, we will analyze many scientific researches and try to answer our questions. The main goal of this work is to study current research and determine the relationship between smells and NFAs. To structure our studies and RQs we studied and evaluated other tertiary studies [16],[17], that helped to understand how to constructed RQs. We aim to answer the following research questions by conducting the SLR:

RQ1: In which way a Code Smell Impact on NFAs?
RQ2: How the connection between code smells and NFAs was determined?
RQ3: Are all code smells impactful on NFAs?

Search strategy and study selection

The search involved in four digital libraries: IEEEXplore, ACM, Elsevier, and Scopus. The syntax of the search string was the same for all libraries, including only the title, keywords, and abstract. Only for searching in Scopus we a little bit changed and adapted the search string. We formed the search string by incorporating alternative terms and synonyms using boolean expression 'OR' and combining main search terms using 'AND'. The following general search string were used for the identification of primary studies: ("Non-functional attributes" OR "NFA" OR "Non-functional requirements") AND ("code-smell" OR "bad smell" OR "code anomalies") AND ("effect" OR "impact" OR "influence") ("Anti patterns" OR "APs" OR "Anti-pattern"). We also added anti-patterns in search terms, because software engineering researchers and practitioners often use the terms "anti-pattern" and "smell" interchangeably [18]. We started the review with an automatic search, supplemented by a snowballing process, to identify potentially important studies. The snowball entailed following links from one article to another to find other related articles [19]. This was done both in the opposite and in the forward direction. The snowballing in the opposite direction means following the list of references, and the forward direction means referring to papers concerning an article that was deemed relevant. Potentially relevant studies were identified based on an analysis of their names and annotations. Immediately discarded studies that were not related to the search. Duplicate reports were removed during the selection process. The identified articles were assessed to determine if they met the following screening criteria. The inclusion and exclusion criteria are shown in Table-1.

Inclusion Criteria	Exclusion Criteria
1. The article should be published in a scientific journal or conference;	1. Publications published before 2010;
2. Publications published from 2010 January;	2. Research-based solely on expert advice, without convincing evidence;
3. Publications that involve an empirical study;	3. Studies that are focused only on code smells or on NFA of source code without any discussion regarding the relationship and/or influence that the first expert on the second;
4. If several journal articles report the same study, the most recent article should be included.	If several journal articles report the same study, the last study should be excluded.

Table 1. Inclusion and Exclusion criteria

Quality assessment criteria

We formed a questionnaire to assess the relevance and quality of research. The quality questionnaire was developed taking into account proposals submitted by Kitchenham et al. [20]. In the Table-2 presents the quality assessment questions that were used to weight the studies. All artifacts obtained as a result of the study selection process are available in the GitHub repository [21].

Only those research papers that can answer all quality assessment questions positively were included in the SLR. Finally, after careful reviews, discussions, and brainstorming sessions, the final decision on inclusion / exclusion for each study was made. All stages and results of study selection process are shown in the Figure 2 and in repository [21].

Q #	Quality questions	Yes	No
Q1	Is there a clear statement of the research objectives?		
Q2	Is the document based on empirical evidence?		
Q3	Are the estimation methods well defined and deliberate?		
Q4	Are methods and tools for determining the effects of code smells described?		
Q5	Have certain types of code smells been analyzed?		
Q6	Have types of NFAs been analyzed?		
Q7	Are the effects of code smells on the NFA determined?		
Q8	Are the limitations of study analyzed explicitly?		

Table 2. Quality assessment questions.

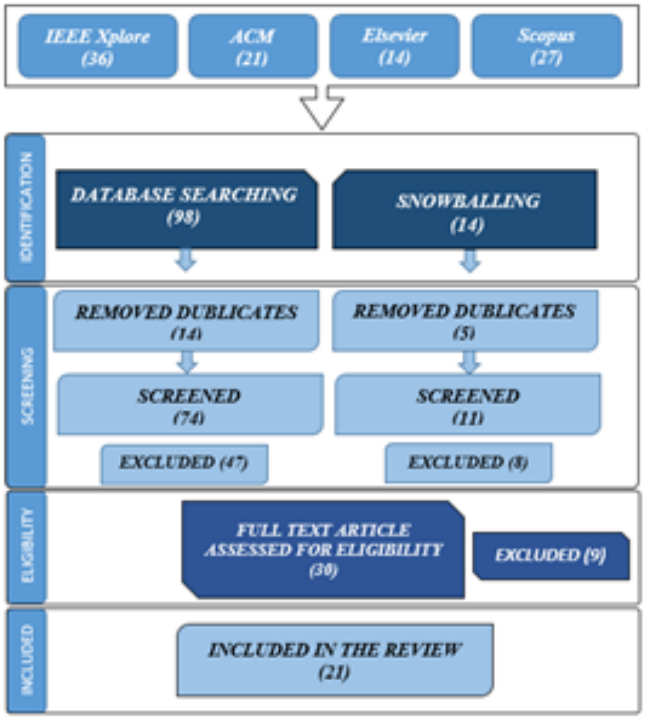

Figure 2. Study selection process.

Data extraction and data synthesis

Based on the results of the selection process described previously, we listed and classified the selected primary studies to enable a clear understanding of aims, methodologies, and findings. We structured the spreadsheet for data extraction. In this form, we find the main information that we consider relevant regarding the papers [21]. In general, we consider:

1. identification number;
2. year;
3. title;
4. objectives or aims;
5. code smells;
6. analysed projects;
7. research questions and respective answers;
8. In which way a Code Smell Impact on NFAs?
9. How the connection between code smells and NFAs was determined?
10. Are all code smells impactful on NFAs?

The basic objective while synthesizing data is to accumulate and combine facts and figures from the selected primary studies in order to formulate a response and resolve the research questions. Collection of a number of studies which state similar and comparable viewpoints and results helps in providing research evidence for obtaining conclusive answers to the research questions. We utilize a number of techniques to synthesize data collected from our primary studies. In order to answer the research questions, we used visualization techniques such as line graph, box plots, pie charts and bar charts. We also used tables for summarizing and presenting the results.

DISCUSSION AND RESULT

This section presents and discusses the findings of this review. Firstly, we present an overview of the selected studies. It also outlines the characteristics of the selected studies which are listed in Table A in the Appendix. Studies are indicated by the letter "S" and the corresponding number. Table A in the Appendix provides a unique identifier for each selected primary study along with a link. These identifiers will be used in all subsequent sections to refer to their primary research. Secondly, we report and discuss the review findings according to the research questions, one by one in the separate subsections.

Overview of selected studies

We identified 21 studies (see Table 6 in Appendix) in the field of effects of code smells on NFA. These papers were published during the time period 2010-2020. Among them, 14 (66%) papers appeared in conference proceedings, 6 (34%) papers were published in journals. Figure 3 depicts the time distribution of the selected studies.

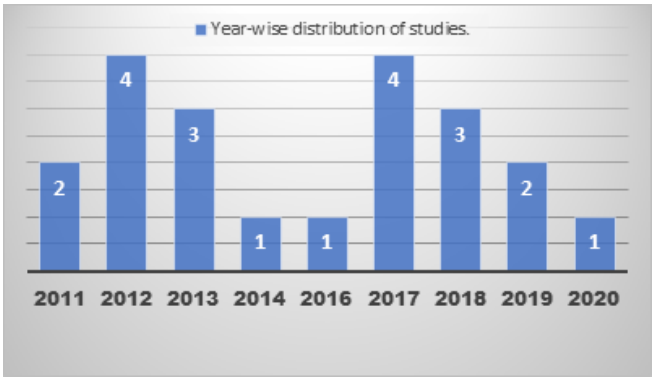

Figure 3. Timeline distribution of papers.

In this SLR we researched the NFAs as maintainability, reliability, performance, and security. The pie chart (Figure 4) describes the percentage of attributes that are included in the SLR. We can see that maintainability has received dominant attention in recent years and 50% of the literature included in the review about the effect of code smells on maintainability.

Further in Table-3 we can see the list of active authors, that researching the influence of code smells on different non-functional attributes. The second column of the table indicates their articles that were included in this SLR.

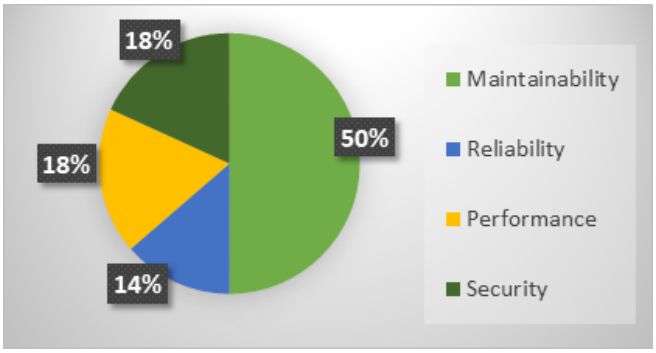

Figure 4. Percentage of included attributes in the review %.

N	Authors	Numbers of Study
1	Aiko Yamashita	S6, S9
2	Fabio Palomba	S17, S20
3	Foutse Khomh	S1, S4, S5, S8, S21
4	Gabriele Bavota	S3, S17
5	Giuliano Antoniol	S1, S4, S7
6	Massimiliano Di Penta	S4, S7, S17
7	Pascal Gadient	S13, S16
8	Yann-Gaël Guéhéneuca	S1, S4, S7, S8, S21

Table 3. Active authors that researching relationships between NFAs and code smells.

Figure-5 describes how many research papers have studied code smells that impact on non-functional attributes. We found that GodClass was investigated in 13 articles and is the most relevant smell of codes that impacts too many types of NFAs. Also, SpaghettiCode, MessageChain, RefusedBequest, and FeatureEnvy was explored in 6-8 articles in this SLR.

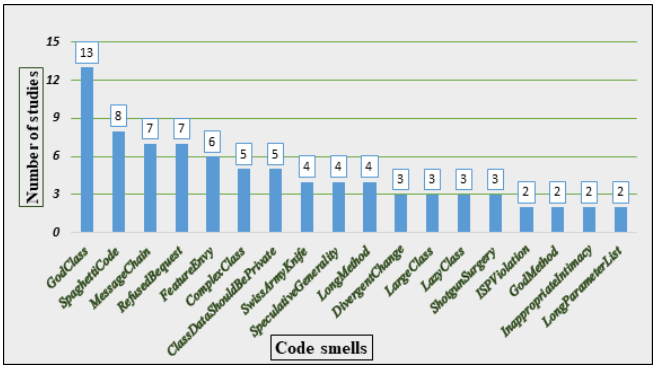

Figure 5. Number of articles that studied code smells impact.

RQ1: In which way a Code Smell Impact on NFA?

In this subsection, we use evidence collected from the selected studies to answer Research Question (RQ1): In which way a Code Smell Impact on NFA? In the Table-3 presents how each of the selected studies contributed to answer RQ1. In this case, the following 21 studies provide evidence of the influence code smells on NFA. We divided them into groups, in connection with the attributes that were investigated in them.

NFAs	Study
Maintainability	S1, S2, S3, S4, S5, S6, S7, S9, S15, S17, S21
Reliability	S4, S8, S10
Performance	S11, S14, S18, S20
Security	S12, S13, S16, S19

Table 4. Attributes and studies.

Mantainability

One important non-functional attribute is maintainability. Recently, many scientists have begun to figure out the effect of code smells on maintainability. Code smells indicate problems with code quality, such as comprehensibility and variability, which can lead to faults. Also, the presence of code smells decreases the understandability of the code and leads to an increase in the cost of maintenance. To help reduce the cost of maintenance, the researchers proposed several approaches that make it easier to understand the program and identify parts of the source code of software systems that are prone to changes and faults. Further, take a closer look at how code smells effect on maintainability.

Abbes et al. [22] determined that systems with combinations of anti-patterns as Blob and Spaghetti code take longer to understand. They require more effort and force subjects to respond less correctly. Understanding the code was easy when the system did not contain any anti-patterns. The results Bavota et al. [24] show that smells of test code are quite common in software systems and that they adversely affect the maintainability of software systems. More precisely, they affect the accuracy and time of test suites. Sjoberg et al. [30], found that files with Feature Envy and God Class code smells are used for several purposes, and the implementation is used instead of the interface. Shotgun Surgery was associated with more effort than files without these smells. The smell of Refused Bequest was associated with a slight decrease in effort. Also, Politowski et al, [42], defined that the anti-pattern Blob or Spaghetti Code present in the source code reduces the productivity of developers, increasing the time spent on tasks, reducing the accuracy of their answers and increasing their necessary efforts. Romano et al. [26], analyzed that Classes affected by anti-patterns change more often as the system develops. Classes affected by antipatterns Complex Class, Spaghetti Code, and SwissArmyKnife are more likely to change than classes affected by other antipatterns. Certain antipatterns lead to certain types of source code changes, for example, API changes appear more often in classes affected by Complex Class, Spaghetti Code and SwissArmyKnife antipatterns. Zazworka et al. [23] show that the god classes are generally more prone to change, and in some cases more prone to defects. This is a strong indicator that the smell of the god class is important for monitoring and management in software development projects, and that they are actually associated with technical debt. Also, Palomba et al. [38], determined, that classes that are affected by code smells have a statistically significant higher change and fault susceptibility than classes that are not affected by code smells. In addition, the authors observed a very clear trend, indicating that the higher the number of smells affecting the class, the higher

its change- and fault-proneness. Yamashitaa et al. [27], concluded that systems that contain more code smells than others are too complex and comprehensive. These systems will be very difficult to maintain. They affect to maintainability factors: Design suited to problem domain, Inheritance, Simplicity, Use of components, Design consistency, Logic Spread. Ban et al. [36], demonstrated that anti-patterns adversely affect the maintainability of software. The most important ones to avoid are Long Functions, Large Class Codes and Shotgun Surgeries. As for the Shotgun Surgery, the main goal is to reduce the connection by moving or retrieving methods or fields, or even identifying a superclass. Considering all the above results, we can conclude that almost all the smells of the code affect the maintainability of the system. But as shown in studies, different smells affect differently, the authors advise paying attention to smells that complicate understanding of the code. Accordingly, which require more time and effort. Effect of code smells to maintainability shown in the Table-4. We can see that God Class and Spaghetti Code more impacts to maintainability quality.

Quality aspects	Code smells
Simplicity and comprehension	GodClass, GodMethod, LazyClass, MessageChains, LongParameterList, SpaghettiCode
Design consistency	AlternativeClasseswithDifferentInterfaces, ISPViolation, DivergentChange, TemporaryField
Time – consuming	SpaghettiCode, GodClass
Effort – consuming	ShotgunSurgery, SpaghettiCode, GodClass
Change – prone	InappropriateIntimacy, LongMethod, SpaghettiCode, SpeculativeGenerality, ComplexClass, RefusedBequest, SwissArmyKnife, GodClass
Require a higher number of test cases	GodClass, ComplexClass, AntiSingleton, SwissArmyKnife
Increase future maintenance costs	GodClass

Table 5. Code smells impact to maintainability

Reliability

Software reliability is the likelihood that a program runs smoothly for a period of time. Malfunctions in the system during operation occur due to faults. Sometimes code smells are the cause of such failures. They negatively affect systems, making classes more fault-prone. Khomh et al. [25] determined that classes with anti-patterns are more likely to be subject to high fault-fixing changes. Specifically, they found the Message Chain is consistently associated with a high fault rate. Jaafar et al. [29] observed three systems (ArgoUML, JFreeChart, and XercesJ) with 12 types of anti-patterns and their dependencies on faults. They concluded that if classes changed with anti-patterns, they will be significantly more fault-prone than other similar classes. However, they do not rule out the possibility that there will be no impact on fault tolerance for classes that have been modified with SpaghettiCode, ClassDataShouldBePrivate, ComplexClass, and LongParameterList. Also, Hall et al [31], investigated the relationship between faults and code smells in three systems, such as Eclipse, ArgoUML, and Apache Commons. In different systems, code smells affected the appearance of faults in the system in different ways. For example, there is a significant relationship between failures and MessageChains in two systems. In cases where DataClumps is combined with large files, there are more crashes in two systems, and less in one. On two systems containing MessageChains, they are associated with a large number of errors. These results show that code smells manifest themselves differently in different systems, probably depending on the application area and development context.

Performance

Performance determines how well a system performs certain functions under certain conditions. Examples are response speed, bandwidth, power consumption, and storage capacity. Service levels, including performance requirements, are often based on end-user support. Therefore, the studies included in the SLR are mainly related to the Android system. Hecht et al. [32], provides that the code smells Getter / Setter (IGS), MemberIgnoringMethod (MIM), and HashMapUsage (HMU) have a significant impact on the user interface and memory performance. Correcting these smells in the Android code effectively improves the user interface and memory performance. Carette et al. [35] found that fix at least one smell of Android code (HMU, IGS or MIM) reduces the power consumption of the mobile phone. Verdecchia et al. [39] proved that refactoring code smells can lead to a significant reduction in power consumption of software applications. Refactoring code smells positively affects the environmental aspect by reducing the energy impact of software and the technical aspect by increasing the convenience of maintaining software over time. Also, Palomba et al. [41] investigate the Impact of Code Smells on the Energy Consumption of Mobile Applications. Code smells such as InternalSetter, LeakingThread, MemberIgnoringMethod and SlowLoop consume 87 times more than methods that are affected by other code smells. Moreover, the authors found that refactoring these code smells reduces energy consumption in all situations.

Security

Software security is one of the most pressing issues in Software Engineering. Therefore, vulnerabilities that are manifested due to code smells have been studied. For instance, Islam et al. [33] found that security vulnerabilities in cloned code are significantly more dangerous than in non-cloned code. Clones are syntactically identical code fragments with variations in identifier names, literals, types, layout and comments, and also allow further differences, such as adding, deleting, or modifying operators that are most unsafe. Ghafari et al. [34] investigated 160 applications and found that despite the variety of applications in terms of popularity, size, and release date, most suffer from at least three different smells of security. And the smells of security are actually an indicator of security vulnerabilities. Gadient ae al. [37] found that the default value for task affinity configurations does not protect the application from theft of user interface components. A system that contains code smells like DenialofService, IntentSpoofing, IntentHijacking is more vulnerable than a system that does not contain this code smells. Elkhail et al. [40] discovered that code smells can slow down the development process and can increase the risk of faults and crashes.

Summary for RQ1

Maintainability, performance, and security are more affected by code smells. The code smells that most affect different quality non-functional attributes are GodClass, LongMethod, SpaghettiCode and FeatureEnvy. GodClass most affects attribute quality. This affects maintainability, complexity, changeability, and performance. Feature Envy and InappropriateIntimacy make the code more change-prone. ComplexClass, SpaghettiCode, and SwissArmyKnife make the code more change-prone. If the source code contains a combination of code smells, it most reduces the quality of non-functional attributes. Therefore, a combination of all kinds of code smells should be avoided. To prevent the impact of code smells on performance, we basically need to refactor the code. Code smells of the Android system, e. g. Getter / Setter (IGS), MemberIgnoringMethod (MIM), and HashMapUsage (HMU) mostly affect performance aspects like user interface, Energy Consumption, and memory performance. Code smells on security can slow down the development process and can increase the risk of faults and crashes. Thus, it will increase the risks of software security vulnerabilities. The most impactful code smells on security is DenialofService, IntentSpoofing, IntentHijacking.

RQ2: How the connection between code smells and NFA was determined?

To answer this question, we scanned all articles that were included in the systematic analysis. Defined that different tools and techniques were used to determine the relationship between code smells and NFAs. Depending on the NFAs, this process was carried out differently. But articles in this SLR extensively used the DECOR method (Detection of dEfects for CORrection) to detect code smells [[22], [25], [26], [26], [29], [42]]. Abbes et al. [22] conducted an Empirical Study of the Impact of anti-patterns On Program Comprehension. They used the Detex anti-pattern detection technique, which is based on the DECOR method, to ensure that every system has at least one occurrence of the Blob anti-pattern or SpaghettiCode. Then authors confirm the classes manually. For the experiment, they used a combination of one Blob and one SpaghettiCode to analyze its effects on understandability of source code. Then they measured the effort and the subjective workload of the task using the NASA Task Load Index (TLX). Later, Politowskia et al. [42] continued this experiment. They used different tasks, each participant was assigned 2 systems: one containing two occurrences of the Blob anti-pattern or spaghetti code, and one without any anti-pattern. They measured average correct answers, time and effort spend for understanding the impact of anti-patterns. Also, they measured the participant's effort using the NASA Task Load Index. To analyze statistical computing was used the R9 software environment, which consisted of a preliminary analysis and a statistical hypothesis test. For preliminary analysis, the authors applied descriptive statistics to the dependent variables and used boxplots10 to analyze the data. They tested null hypotheses using the linear mixed model analysis method. In most cases, researches used methods for detecting code smells, such as Metrics-based smell detection, History-based smell detection, and Machine learning-based smell detection. In some studies, refactoring code smells has been used effectively to detect the effects of code smells [[23], [28], [30], [31], [33], [35], [39], [41]]. They compared to code that contained code smells and code after refactoring code smells. Palomba et al. [41] conducted a research to identify the Impact of Code Smells on Energy Consumption. The authors performed an analysis using the aDoctor and PETrA tools. These tools were developed and evaluated by the authors earlier. aDoctor is an Android-specific code smell detector that extracts structural properties from source code to detect code smells. PETrA is a software tool that evaluates the energy profile of mobile applications. Using these tools, they found that some code smells are causes 87 times more energy consumption than other code smells. In addition, they used the refactoring method as a way to increase energy efficiency by removing the smell of code. As a result, it was known that it was possible to increase the energy efficiency of source code methods by refactoring code smells.

Summary for RQ2.

To answer the RQ2 we focused on the frequent methods that were used in research papers to determine the effect of code smells on NFAs. There is still no universal method to detect all types of code smells and also, in different systems, code smells differently exert their influence. These factors influenced the creation of different techniques and tools. In several studies, they used the refactoring method to find the connection between code smells and NFAs.

RQ3: Are all code smells impactful on NFA?

In previous research questions, we determined that code smells affect NFAs in different ways. In response to this question, we wanted to find out whether some code smells might not affect the NFAs, so that the developers would not spend effort on refactoring in some cases. And analyzing the articles included in the SLR, we found that some types of code smells affect the factors of NFAs less. Abbes et al. [22] found that the appearance of one anti-Blob pattern or one spaghetti code in the source code of a software system does not complicate its comprehensibility. Researchers have compared the source code without an anti-pattern to verify this. Bavota et al. [24] investigated the effect of test smells on maintainability. They concluded that the Lazy Test smell does not have a strong effect on program maintenance and comprehension. Sabane et al. [28] determined that classes with anti-patterns like method chains or Lazy Classes do not greatly affect the cost of testing. Sjoberg et al. [30] explored the quantitative impact of code smells on maintenance efforts. They found that code smell RefusedBequest was associated with less effort than other code smells. Hall at al. [31] conducted their experiment on different systems like Eclipse, ArgoUML, and Apache Commons and investigated the effect of five smells on the number of faults. The results show that smell SwitchStatements did not affect failures in all three systems. It was also discovered that DataClumps increased the number of Faults in one system, but reduced the number of faults in two other systems. The results show that smells manifest differently in systems, probably depending on the subject area and development context. Researchers also found that certain smells reduce the number

of faults rather than increase them. For example, MiddleMan reduced the number of faults in ArgoUML, and SpeculativeGenerality reduced in Eclipse. This is an important conclusion, as it suggests that arbitrary code smells refactoring can actually be unproductive.

Summary for RQ3.
Analyzing the results, we can conclude that code smells are not always the main cause of errors in the code. Sometimes some types of smells reduce faults. Also refactoring is not an absolute solution to improve the quality of the code.

THREATS TO VALIDITY

The following validity issues were considered in interpreting the results of this review. In substantiating the conclusion, there is a risk of biased data extraction. This was resolved by defining a form of data mining to answer research questions. Conclusions and implications are based on the extracted data. Threats to internal validity were considered at the review selection stage. The studies included in this review were identified through a rigorous multi-stage selection process. The studies identified in the SLR have been compiled from numerous literature databases covering relevant journals and conferences. One possible threat is bias in the choice of publications. In the present work, this was solved by specifying a research protocol that defines research questions and research objectives, inclusion and exclusion criteria, search strings that will be used in the information search strategy and data extraction strategy. Another potential bias is due to code smells. This makes it possible not to paint the whole picture, because not all code smells and, their effect on NFAs was detected. The goal was not to get an exhaustive list of code smells that effect on NFAs. At least those that are more relevant and representative and, impact on software product qualities.

LIMITATIONS

This systematic review considered several primary studies to identify code smells effect on NFA. A limitation in this review is that not all types of NFAs and code smells were covered. Thus, the effect is not definitely conclusive. We got results for certain types of NFA. Moreover, while identifying the impact of smells on NFA, each primary study may use different techniques and methods. Though we have exhaustively searched all the stated digital search libraries, there still may be a possibility that a suitable study may be left out. Also, this review does not include any unpublished research studies. We have assumed that all the studies are impartial, however if this is not the case, then it poses a threat to this study.

CONCLUSION

This paper presents an overview that informs about the impact of code smells on NFAs. To systematize empirical studies, a systematic review was carried out in the databases IEEEXplore, ACM, Elsevier, and Scopus. 21 articles with experimental study designs met the criteria, and were analyzed in-depth. **RQ1.** As a result, we received answers to research questions that determined the impact on attributes as Maintainability, Reliability, Performance and Security. We found that Maintainability, Performance, and Security are more affected by code smells. Source code with code smells becoming more fault-prone and change-prone. If the source code contains a combination of code smells, it most reduces the quality of non-functional attributes. Code smells on security can slow down the development process and can increase the risk of faults and crashes. **RQ2.** We determine that there is still no universal method to detect all types of code smells and, in different systems, code smells differently exert their influence. These factors influenced the creation of different techniques and tools. In several studies, they used the refactoring method to find the connection between code smells and NFAs. **RQ3**. We concluded that code smells are not always the main cause of errors in the code. Sometimes some types of smells reduce faults. Also refactoring is not the absolute solution to improve the quality of the source code.

ACKNOWLEDGMENTS

The author is grateful to the professor Fabio Palomba for scientifically-based, and interesting lectures on the class "Software dependability". The author would like to thank professor Dario Di Nucci for his guidance, constant input and suggestions throughout the work.

APPENDIX

Appendix A. A list of selected papers used in this Systematic Literature Review is provided in the following Table.

ID	Author	Title	Year
S1 [22]	Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, Giuliano Antoniol	An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension	2011
S2 [23]	Nico Zazworka, Michele A. Shaw, Forrest Shull, Carolyn Seaman	Investigating the Impact of Design Debt on Software Quality	2011
S3 [24]	Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, David Binkley	An empirical analysis of the distribution of unit test smells and their impact on software maintenance	2012
S4 [25]	Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuca, and Giuliano Antoniol	An exploratory study of the impact of antipatterns on class change- and fault-proneness	2012
S5 [26]	Daniele Romano, Paulius Raila, Martin Pinzger, Foutse Khomh	Investigating Multi-Touch and Pen Gestures for Diagram Editing on Interactive Surfaces	2012
S6 [27]	Aiko Yamashita, Leon Moonen	Do code smells reflect important maintainability aspects?	2012
S7 [28]	Aminata Sabane, Massimiliano Di Penta, Giuliano Antoniol, Yann-Gael Gueheneuc	A Study on the Relation Between Ant patterns and the Cost of Class Unit Testing	2013
S8 [29]	Fehmi Jaafar , Yann-Gael Gueheneuc , Sylvie Hamel, Foutse Khomh	Mining the Relationship between Anti-patterns Dependencies and Fault-Proneness	2013
S9 [30]	Dag I.K. Sjoberg ; Aiko Yamashita ; Bente C.D. Anda ; Audris Mockus ; Tore Dyba	Quantifying the Effect of Code Smells on Maintenance Effort	2013

ID	Author	Title	Year
S10 [31]	Tracy Hall, Min Zhang, David Bowes, Yi Sun	Some Code Smells have a Significant but Small Effect on Faults	2014
S11 [32]	Geoffrey Hecht, Naouel Moha, Romain Rouvoy	An Empirical Study of the Performance Impacts of Android Code Smells	2016
S12 [33]	Md Rakibul Islam, Minhaz F. Zibran, Aayush Nagpal	Security Vulnerabilities in Categories of Clones and Non-Cloned Code: An Empirical Study	2017
S13 [34]	Mohammad Ghafari, Pascal Gadient, Oscar Nierstrasz	Security Smells in Android	2017
S14 [35]	Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, Romain Rouvoy	SInvestigating the Energy Impact of Android Smells	2017
S15 [36]	Denes Ban	The Connection between Antipatterns and Maintainability in Firefox	2017
S16 [37]	Pascal Gadient, Mohammad Ghafari, Patrick Frischknecht, Oscar Nierstrasz	Security Code Smells in Android ICC	2018
S17 [38]	Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, Andrea De Lucia	On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation	2018
S18 [39]	Roberto Verdecchia, Rene Aparicio Saez, Giuseppe Procaccianti, Patricia Lago	Empirical Evaluation of the Energy Impact of Refactoring Code Smells	2018
S19 [40]	Abdulrahman Abu Elkhail ; Tomas Cerny	On Relating Code Smells to Security Vulnerabilities	2019
S20 [34]	Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, Andrea De Lucia	On the Impact of Code Smells on the Energy Consumption of Mobile Applications, Information and Software Technology	2019
S21 [42]	Cristiano Politowskia, Foutse Khomhb, Simone Romanoc, Giuseppe Scanniellod, Fabio Petrilloe, Yann-Gaël Guéhéneuca and Abdou Maigaf	A Large Scale Empirical Study of the Impact of Spaghetti Code and Blob Anti-patterns on Program Comprehension	2020

Table 6. List of selected studies from the search string

REFERENCES

1. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, Nonfunctional requirements in software engineering, Massachusetts: Kluwer Academic Publishers, 2000.
2. D. Firesmith, "Using quality models to engineer quality requirements", Journal of Object Technology, vol. 2, pp. 67-75, 2003.
3. C. Ebert, "Putting requirement management into praxis: dealing with nonfunctional requirements", Information and Software Technology, vol. 40, pp. 175-185, 1998.
4. G. Booch, Object-oriented analysis and design, Addison-Wesley, 1980
5. M. Fowler, K. Beck, Refactoring: improving the design of existing code, Addison-Wesley, 1999.
6. A. Sabané, M. Di Penta, G. Antoniol, Y. Guéhéneuc, 2013, A study on the relation between antipatterns and the cost of class unit testing. In: CSMR '13: Proceedings of the 17th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, 2, pp. 167–176, 2013.
7. A. Yamashita, L. Moonen, To what extent can maintenance problems be predicted by code smell detection? - an empirical study, Information and Software Technology, 55 (12), 2013.
8. F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R.Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Empirical Software Engineering, v. 23, pp.1188–1221, 2018.
9. M Abbes, F Khomh, YG Gueheneuc, G Antoniol, A. Zhang, M., Hall, T., Baddoo, N., Apr. 2011. Code Bad Smells: A review of current knowledge. Journal of Software Maintenance and Evolution 23 (3), 179–202. rical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, 15th European Conference on Software Maintenance and Reengineering, pp. 181-190, 2011.
10. F. Jaafar, Y.G. Guéhéneuc, S. Hamel, F. Khomh, Mining the relationship between anti-patterns dependencies and fault-proneness, 20th Working Conference on Reverse Engineering (WCRE), pp. 351-360, 2013.
11. S. Singh, S. Kaur, A systematic literature review: Refactoring for disclosing code smells in object oriented software, Ain Shams Engineering Journal, 9(4), pp. 2129-2151, 2018.
12. M. Zhang, T. Hall, N. Baddoo, Code Bad Smells: A review of current knowledge, Journal of Software Maintenance and Evolution, 23 (3), 179–202, 2011.
13. A. Cairo, G. Carneiro, M. P. Monteiro, The Impact of Code Smells on Software Bugs: A Systematic Literature Review, Information, 9, 273, 2018.
14. T. Sharma and D. Spinellis, A Survey on Software Smells, Journal of Systems and Software 138, pp. 158-173, 2017.
15. B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey and S. Linkman, Systematic literature reviews in software engineering – A systematic literature review, Information and Software Technology, 51(1), p.7–15, 2009.
16. S. Imtiaz, M. Bano, N. Ikram, M. Niazi, A tertiary study: Experiences of conducting systematic literature reviews in software engineering, in: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, EASE '13, Association for Computing Machinery, New York, NY, USA, p. 177–182, 2013.
17. M. U. Khan, S. Sherin, M. Z. Iqbal, R. Zahid, Landscaping systematic mapping studies in software engineering: A tertiary study, Journal of Systems and Software 149, pp. 396 – 436, 2019.

18. M. Linares-V´asquez, S. Klock, C. McMillan, A. Saban´e, D. Poshyvanyk, Y. Gueheneuc, Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps, In: ICPC 2014: Proceedings of the 22nd International Conference on Program Comprehension. ACM, pp. 232–243, 2014.

19. M. Skoglund, and P. Runeson, Reference-based search strategies in systematic reviews, In Proceedings of the 13th international conference on Evaluation and Assessment in Software Engineering, Swindon, UK, 2009, p. 20–21.

20. B. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi, S. Linkman, Systematic literature reviews in software engineering – a tertiary study, Information and Software Technology 52 (8), pp. 792 – 805, 2010.

21. N.Munalbayeva, Online repository "SLR on the Effect of Code Smells on NFA" https://github.com/nazirakz/Effect-of-Code-Smells-on-Non-Functional-Attributes-of-Source-Code

22. M. Abbes, F. Khomh, Y. Guéhéneuc, and G. Antoniol, An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension, 15th European Conference on Software Maintenance and Reengineering, 2011, DOI: 10.1109/CSMR.2011.24

23. N. Zazworka, M.A. Shaw, F. Shull, and C. Seaman, Investigating the impact of design debt on software quality. In: MTD '11: Proceedings of the 2nd Workshop on Managing Technical Debt. ACM, Fraunhofer USA, Inc, 2011, pp. 17–23.

24. G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: IEEE International Conference on Software Maintenance, ICSM. IEEE, Universita di Salerno, Salerno, Italy, 2012, pp. 56–65.

25. F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering. 17 (3), 2012, 243–275.

26. D. Romano, P. Raila, M. Pinzger, and F. Khomh, Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes, 19th Working Conference on Reverse Engineering, 2012, DOI:10.1109/WCRE.2012.53

27. A. Yamashita; and L. Moonen, Do code smells reflect important maintainability aspects?, 28th IEEE International Conference on Software Maintenance (ICSM), 2012, DOI:10.1109/ICSM.2012.6405287

28. A. Sabané, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, A study on the relation between antipatterns and the cost of class unit testing. In: CSMR '13: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, 2013, pp. 167–176 .

29. F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, Mining the relationship between anti-patterns dependencies and fault-proneness, In: Proceedings - Working Conference on Reverse Engineering, WCRE. IEEE, Ecole Polytechnique de Montreal, Montreal, Canada, 2013, pp. 351–360 .

30. D.I.K. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, 2013. Quantifying the effect of code smells on maintenance effort, IEEE Trans. Softw. Eng., 2013, 39 (8), pp.1144–1156.

31. T. Hall, M. Zhang, D. Bowes, and Y. Sun, Some code smells have a significant but small effect on faults. ACM Trans. Softw. Eng. Methodol. (TOSEM), 2014, 23 (4), pp. 33–39

32. G. Hecht, N. Moha, and R. Rouvoy, An empirical study of the performance impacts of Android code smells. In: MOBILESoft '16: Proceedings of the International Workshop on Mobile Software Engineering and Systems. ACM, Universite Lille 2 Droit et Sante , 2016.

33. M. R. Islam, M. F. Zibran, and A. Nagpal, Security Vulnerabilities in Categories of Clones and Non-Cloned Code: An Empirical Study, ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017, DOI: 10.1109/ESEM.2017.9

34. M. Ghafari, P. Gadient, and O. Nierstrasz, Security Smells in Android, IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2017, DOI: 10.1109/SCAM.2017.24

35. A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, Investigating the Energy Impact of Android Smells, 24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER), Feb 2017, Klagenfurt, Austria. pp.10.

36. D. Ban, The Connection between Antipatterns and Maintainability in Firefox, Acta Cybernetica, 2017, 23, 471–490.

37. P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz, Security Code Smells in Android ICC, Empirical Software Engineering Journal (EMSE), 2018.

38. F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Empirical Software Engineering, 2018, 23, pp.1188–1221

39. R. Verdecchia, R. A. Saez, G. Procaccianti, and P. Lago, Empirical Evaluation of the Energy Impact of Refactoring Code Smells, 5th International Conference on ICT4S, Toronto, Canada, 2018

40. A. A. Elkhail; and T. Cerny, On Relating Code Smells to Security Vulnerabilities, IEEE 5th Intl Conference on Big Data Security on Cloud, IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS), 2019.

41. F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, On the Impact of Code Smells on the Energy Consumption of Mobile Applications, Information and Software Technology, 2019, 105, pp. 43-55.

42. C. Politowskia, F. Khomhb, S. Romanoc, G. Scanniellod, F. Petrilloe, Y-G. Guéhéneuca, and A. Maigaf, A Large Scale Empirical Study of the Impact of Spaghetti Code and Blob Anti-patterns on Program Comprehension, Information and Software Technology, June 2020, v.122.