Occurrence and frequency of code smells across programming languages & platforms

Viktoriia Abakumova
Institute of Computer Science
University of Tartu
Tartu, Estonia
viktoriia.abakumova@ut.ee

Serhii Murashko
Institute of Computer Science
University of Tartu
Tartu, Estonia
serhii.murashko@ut.ee

***Abstract*—Code smells are common issues that can degrade software quality and maintainability. To identify code smells detection tools and techniques on par with refactoring strategies and the best practices for preventing code smells from occurring in code. This paper presents a systematic mapping study exploring code smells in various programming languages and platforms. As a result, common code smell types, a list of tools to identify them, refactoring strategies and prevention practices were described.**

***Keywords*— code smells, code smell detection tools, refactoring, code smells prevention**

I. Introduction

Software development is a complex process that involves numerous challenges, including ensuring code quality and maintaining the software over time. As practice shows - the presence of code smells could impact the abovementioned factors. Code smells indicate potential problems in a codebase that could degrade its readability and extensibility. They are known (code smells catalogue - Fowler and Beck 1999 along with others) symptoms of design flaws or implementation issues that violate good programming practices and can lead to problems such as bugs, maintenance difficulties, reduced performance or higher energy consumption [1, 6, 8, 14].

Code smells can occur in software projects developed using different programming languages and platforms. However, the occurrence and frequency of code smells can vary depending on the language and platform used. It also affects the level of research and the variety of methods and tools for identifying code smell. Based on this, the research is conducted about existent code smell types, detecting tools and techniques, refactoring strategies and preventing practices.

II. Methodology

This paper is a systematic map and is composed of the following five steps: i) research questions definition, ii) data search, iii) study selection, iv) data extraction and v) gathered results.

A. Research Questions

This systematic mapping study aims to identify relevant papers on the occurrence and frequency of code smells across programming languages and platforms. This research aims to give an overview of different code smell types, tools and techniques that are used for identifying and preventing. It leads to the following research questions (RQs):

- RQ1: Which code smell types are common in projects?
- RQ2: What tools and techniques can be used to detect code smells?
- RQ3: What are the most effective strategies for refactoring code to remove code smells?
- RQ4: What are the best practices for preventing code smells from occurring in the first place?

It is known that source code could contain code smells, so the first research question is sophisticated to discover them. On this basis, the second research question was formulated to look at tools and techniques that help identify code smells. The third research question was asked to highlight existing refactoring techniques so that they could be applied in practice. The fourth research question was formulated to identify techniques to prevent code smells in projects so that they can also be used during work. These questions are related sequentially – identification, detection, refactoring and enhancement and further prevention of code smell.

B. Search Strategy and Data Sources

The Scopus database was selected for this mapping study. It is known for relevant and authoritative research and indexes the most reliable scientific articles (some are open-access). The search string for data sources:

```
(TITLE-ABS-KEY ("code smells")) AND (SUBJAREA (comp) OR SUBJAREA (engi))
```

Fig. 1. The search string.

As the database contains a large variety of material, limitations and keywords were selected for the search. This search query was executed and it returned 1080 results.

C. Study Selection and Quality Assessment

Applying this search string produced an extensive collection of papers, but not all of them answered the research questions of this study. Thus, further filtering was needed in the form of exclusion and inclusion criteria. The following exclusion criteria was created in order to filter out inappropriate publications:

- EC1: Published earlier than 2020 (419 remaining)
- EC2: Language is not English (410 remaining)

- EC3: Source type is not journal or conference proceeding (365 remaining)
- EC4: Is not in open access (107 remaining)
- EC5: Document type is not article conference paper (100 remaining)
- EC6: Publication stage is not final (97 remaining)
- EC7: Does not include only specific keywords (75 remaining).

It is expected that in the sample of articles, there will remain articles published in English after 2020 (this is because this area of knowledge is developing quite quickly, and we would like to see results different from the well-known smell codes). There are also exclusion criteria specific to the Scopus database. Thus, the final sample will contain freely accessed journal articles or conference papers that have already been published and contain keywords.

A set of inclusion criteria was created:

- IC1: Journals and conference papers mentioning code smell types
- IC2: Journals and conference papers discussing tools and techniques to detect code smells
- IC3: Journals and conference papers reviewing different strategies of refactoring code to remove code smells
- IC4: Journals and conference papers proposing practices for preventing code smells (20 remaining)

Inclusion criteria concerned research questions and content of articles to choose articles that reveal the topic and can answer the questions posed.

Exclusion criteria were applied to the original collection of 1080 papers. To exclude papers not published during the previous four years, EC1 was applied, which resulted in 419 papers. Then EC2 criteria eliminated papers which are not available in English. EC3, EC4, EC5 were applied to eliminate articles which are not in open access and with inappropriate document type or stage.

As a result of utilizing the exclusion criteria, 75 papers remained. After applying exclusion criteria, each of the papers' abstracts were read through manually. The inclusion criteria were applied by analyzing the abstracts and keywords to contain material relating to the RQs raised so 20 papers were chosen.

Additional quality criteria was used to assess the quality of the selected studies:

- QC1: Are the research questions in the paper clearly defined and relevant to the current mapping study?

This was done manually by reading the phrasing and justification of the research questions in the selected papers and assessing whether the article could answer more than one research question of this studying mapping.

Consequently, 15 papers total were included to be used in this study.

D. Data Extraction

The selected papers were read through, and needed information was gathered into a data extraction table. Data items that were looked for, in addition to article title and author names, were code smell types, tools and techniques for their detection, strategies for refactoring code and the best practices for preventing code smells from occurring.

TABLE I. DATA EXTRACTION FORM

Data Item	Value	RQ
ID	An identifier of the article	
Article Title	Name of the article	
Author(s)	Name(s) of the author(s)	
Types	Code smell types that are common in projects	RQ1
Tools and techniques	Tools and techniques that can be used to detect code smells	RQ2
Refactoring strategies	The most effective strategies for refactoring code to remove code smells	RQ3
Prevent practices	The best practices for preventing code smells from occurring in the first place	RQ4

III. MAIN FINDINGS

A. RQ1 – Which code smell types are common in projects?

TABLE II. COMMON TYPES

Common Types	Source
Archetectural smells	[1]
Test smells	[1], [13], [15]
Spreadsheet smells	[1]
Common (catalogue code smells)	[1], [3], [7], [14]
Elixir specific code smells (design-related smells and low-level concerns smells)	[4]
Microservice-specific code smells, code smells in distributed microservice applications	[5]
Code smells on energy consumption	[6]
Mobile code smells	[6]
SQL code smells	[8]

Table 2 presents the general overview of code smell types which were mentioned in 15 publications to answer RQ1.

B. RQ2 – What tools and techniques can be used to detect code smells?

TABLE III. TOOLS AND TECHNIQUES FOR DETECTION

Tools/technique	Source
Open source static analysis tools	[1], [3], [5], [7], [14]
PMD	[1], [3], [5], [7], [9]
SonarQube	[1], [3]
Designite	[1], [5]

Tools/technique	Source
Technique based on ensemble machine learning and deep learning approaches	[2], [9]
Spotbugs	[3], [5], [7]
Arcade	[3], [5]
Arcan	[3], [5]
Credo	[4]
FindBugs	[5], [7], [9]
CheckStyle	[5], [7], [9]
AI Reviewer	[5]
Hotspot Detector	[5]
Massey Architecture Explorer	[5]
MSA Nose	[5]
Sonargraph	[5]
STAN	[5]
Structure 101	[5]
Jdeodorant	[6]
SQLInspect	[8]
Decor	[8]
LiveRef	[11]
Live refactoring	[12]
PyNose	[13]
TestLint	[15]
TeCReVis	[15]

The tools and techniques mentioned in Table 3 could be broadly categorized into two groups: open-source static analysis tools and techniques based on ensemble machine learning and deep learning approaches.

The first group consists of PMD, SonarQube, and Designite, which are open-source static analysis tools used to analyze source code for bugs, vulnerabilities, and design issues. PMD identifies common programming issues, SonarQube is a platform for continuous inspection of code quality, and Designite detects design issues such as tight coupling, low cohesion, and violations of design principles.

The second group consists of Spotbugs, Arcade, Arcan, Credo, FindBugs, CheckStyle, AI Reviewer, Hotspot Detector, Massey Architecture Explorer, MSA Nose, Sonargraph, STAN, Structure 101, Jdeodorant, SQLInspect, Decor, LiveRef, Live refactoring, PyNose, TestLint, and TeCReVis, which are all tools and techniques based on ensemble machine learning and deep learning approaches. These tools analyse source code, microservice architectures, software architecture, SQL queries, and test coverage to identify potential issues and perform real-time automated code refactoring.

C. *RQ3 – What are the mose effective strategies for refactoring code to remove code smells?*

TABLE IV. REFACTORING STRATEGIES

Refactoring strategies	Source
Manual Refactoring	[1], [6]
Capture code smells without refactoring	[1]
Ignore code smells	[1]
Ongoing refactoring	[6]
Multi objective automatic refactoring	[6]
Reactive refactoring	[10]
Proactive refactoring	[10]
Live refactoring	[11], [12]

Table 4 presents the general refactoring strategies to remove code smells found across 15 papers to answer RQ3. The described refactoring technologies concerned manual code changes and ongoing processes that should be integrated into the development cycle. Multi-objective automatic refactoring involves using automated tools to refactor code with multiple objectives, such as improving performance, readability, and maintainability simultaneously. Reactive and proactive refactoring were mentioned from different perspectives of the codebase improvement approach - first in response to a specific issue and second as a preventative measure.

D. *RQ4 – What are the best practices for preventing code smells from occurring in the first place?*

TABLE V. PREVENT PRACTICES

Prevent practices	Source
Code review	[1], [14], [15]
Code convention	[1], [14], [15]
Review-based detection	[1], [11]
Familiarization with existing code	[1]
Automatic screening tests	[5], [15]
Code smell prediction	[9]
Machine learning code smell detection	[9]
Prediction models	[10]
Quality gates	[14]

Table 5 presents prevent practices of development process and code analysis found across 15 papers to answer RQ4. Approaches to ensure high-quality code include code review, code convention adherence (improve the readability and maintainability of the code), review-based detection, familiarization with existing code, automatic screening tests

(could catch common issues and errors early in development), code smell prediction, machine learning code smell detection and prediction models (could identify potential issues in the codebase), and quality gates (could ensure that only high-quality code is promoted to production). These approaches can be combined to ensure the codebase is of the highest possible quality.

IV. Results

A. RQ1 – Which code smell types are common in projects?

Table 2 has an overview of the most common code smell types. They vary on specific language or platform that leads to a different problem with speed, energy consumption on mobile devices, tests. In the case of Elixir there are bunch of different code smells caused by Elixir design. Therefore, not all of the code smells in the catalogue may be found in specific languages or platforms this is worth paying attention to when e.g. switching from language to language.

B. RQ2 – What tools and techniques can be used to detect code smells?

Table 3 has presented an overview of tools and techniques that can be used to detect code smells. The most common were open-source static analysis tools, PMD and separately were mentioned other popular tools such as Spotbugs, FindBugs, CheckStyle and SonarQube. All mentioned tools use different approaches to analyze and detect code smells focusing on aspects of the project such as code itself, commits, comments, architecture, tests, some even uses ensemble machine learning and deep learning approaches.

C. RQ3 – What are the mose effective strategies for refactoring code to remove code smells?

Table 4 has presented an overview of strategies for refactoring code to remove code smells. Main strategies are manual and live refactoring and other related to using some tools from Table 3 and automation processes. However, in some cases it mentioned that sometimes the most effective way would be to capture, but don't fix detected code smells or even just ignore them at all.

D. RQ4 – What are the best practices for preventing code smells from occurring in the first place?

Table 5 has given us an overview of the best practices for preventing code smells. The two mains of them are the most common practices in companies: following code conventions and performing code review. Moreover, using quality gates or prediction models brings results too.

V. Conclusion

In conclusion, code smells are a common problem in software development that can affect the quality and maintainability of software projects. This systematic mapping study aimed to identify the most common types of code smells, tools and techniques used to detect them, strategies for refactoring code, and best practices for preventing code smells from occurring in the first place. The study analyzed 15 papers from the Scopus database, and after applying exclusion criteria, the papers revealed that there are various code smells and detection tools, and refactoring strategies and preventive practices. The results can help software developers and project managers understand the importance of detecting and removing code smells to improve the quality and maintainability of software projects.

VI. References

[1] Han, X., Tahir, A., Liang, P., Counsell, S., Blincoe, K., Li, B., & Luo, Y. (2022). Code smells detection via modern code review: A study of the OpenStack and qt communities. Empirical Software Engineering, 27(6) doi:10.1007/s10664-022-10178-7

[2] Dewangan, S., Rao, R. S., Mishra, A., & Gupta, M. (2022). Code smell detection using ensemble machine learning algorithms. Applied Sciences (Switzerland), 12(20) doi:10.3390/app122010321

[3] Islam, M. R., Al Maruf, A., & Cerny, T. (2022). Code smell prioritization with business process mining and static code analysis: A case study. Electronics (Switzerland), 11(12) doi:10.3390/electronics11121880

[4] Da Matta Vegi, L. F., & Valente, M. T. (2022). Code smells in elixir: Early results from a grey literature review. Paper presented at the IEEE International Conference on Program Comprehension, , 2022-March 580-584. doi:10.1145/3524610.3527881 Retrieved from www.scopus.com

[5] Walker, A., Das, D., & Cerny, T. (2020). Automated code-smell detection in microservices through static analysis: A case study. Applied Sciences (Switzerland), 10(21), 1-20. doi:10.3390/app10217800

[6] Sehgal, R., Mehrotra, D., Nagpal, R., & Sharma, R. (2022). Green software: Refactoring approach. Journal of King Saud University - Computer and Information Sciences, 34(7), 4635-4643. doi:10.1016/j.jksuci.2020.10.022

[7] Klima, M., Bures, M., Frajtak, K., Rechtberger, V., Trnka, M., Bellekens, X., . . . Ahmed, B. S. (2022). Selected code-quality characteristics and metrics for internet of things systems. IEEE Access, 10, 46144-46161. doi:10.1109/ACCESS.2022.3170475

[8] Muse, B. A., Rahman, M. M., Nagy, C., Cleve, A., Khomh, F., & Antoniol, G. (2020). On the prevalence, impact, and evolution of SQL code smells in data-intensive systems. Paper presented at the Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020, 327-338. doi:10.1145/3379597.3387467 Retrieved from www.scopus.com

[9] Pecorelli, F., Lujan, S., Lenarduzzi, V., Palomba, F., & De Lucia, A. (2022). On the adequacy of static analysis warnings with respect to code smell prediction. Empirical Software Engineering, 27(3) doi:10.1007/s10664-022-10126-5

[10] SAE-LIM, N., HAYASHI, S., & SAEKI, M. (2021). Supporting proactive refactoring: An exploratory study on decaying modules and their prediction. IEICE Transactions on Information and Systems, E104D(10), 1601-1615. doi:10.1587/transinf.2020EDP7255

[11] Fernandes, S., Aguiar, A., & Restivo, A. (2022). LiveRef: A tool for live refactoring java code. Paper presented at the ACM International Conference Proceeding Series, doi:10.1145/3551349.3559532 Retrieved from www.scopus.com

[12] Fernandes, S. (2021). A live environment for inspection and refactoring of software systems. Paper presented at the ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 1655-1659. doi:10.1145/3468264.3473100 Retrieved from www.scopus.com

[13] Wang, T., Golubev, Y., Smirnov, O., Li, J., Bryksin, T., & Ahmed, I. (2021). PyNose: A test smell detector for python. Paper presented at the Proceedings - 2021 36th IEEE/ACM International Conference on

Automated Software Engineering, ASE 2021, 593-605. doi:10.1109/ASE51524.2021.9678615 Retrieved from www.scopus.com

[14] Falessi, D., & Kazman, R. (2021). Worst smells and their worst reasons. Paper presented at the Proceedings - 2021 IEEE/ACM International Conference on Technical Debt, TechDebt 2021, 45-54. doi:10.1109/TechDebt52882.2021.00014 Retrieved from www.scopus.com

[15] Garousi, V., & Küçük, B. (2018). Smells in software test code: A survey of knowledge in industry and academia. Journal of Systems and Software, 138, 52-81. doi:10.1016/j.jss.2017.12.013