Communication Papers of the of the 17th Conference on Computer Science and Intelligence Systems pp. 313–318
DOI: 10.15439/2022F171
ISSN 2300-5963 ACSIS, Vol. 32

A Data Analysis Study of Code Smells within Java Repositories

Noah Lambaria
Baylor University
One Bear Place #97141, Waco, USA
Email: Noah_Lambaria1@baylor.edu

Tomas Cerny
Baylor University
One Bear Place #97141, Waco, USA
Email: tomas_cerny@baylor.edu

***Abstract*—Although code smells are not categorized as a bug, the results can be long-lasting and decrease both maintainability and scalability of software projects. This paper presents findings from both former and current industry individuals, aiming to gauge their familiarity with such violations. Based on the feedback from these individuals, a collection of smells were extracted from a sample size of 100 Java repositories in order to validate some of the smells that are typically encountered. After analyzing these repositories, the smells typically encountered are *Long Statement*, *Magic Number*, and *Unutilized Abstraction*. The results of this study are applicable for developers and researchers who require insight on the frequencies of code smells within a typical repository.**

I. Introduction

THE term "code smell" dates back to the 1990s, where Kent Beck first defined the term. Martin Fowler was another individual responsible for popularizing the term within his book *Refactoring*, which addressed code smells with the application of Java examples [1]. Although many authors and researchers have defined an abundant amount of code smells, what is deemed to be a code smell or not is subjective and abstract. This is due to the vast variations of smells that exist, and what is considered to be a harmful and non-harmful smell. An example of a code smell that this study and future work aims to address is the *God Class*, as well as several other smells. Alves et al. mention that *God Classes* can be up to 13 times more likely to contain faults embedded within the smell itself [2]. For this reason, it is imperative to dive deeper within large classes and methods and examine their occurrences since these smells are detrimental. Our goal is to compare the frequencies of such smells compared to others with our selected tool, which will be elaborated on in further sections.

Evident in the works of Fontana et al., anti-patterns and code smells have the potential to impact Technical Debt (TD) and Architectural Degradation (AD) [3]. Their research suggested that some code anomalies were more inclined to be better indicators of TD than others. Because of instances such as these, code smells can be detrimental to software systems over the course of time. To resolve these conflicts, there are abundance of different static analyzers within the software community that assist code smell identification.

This research was funded by National Science Foundation grant number 1854049 and a grant from Red Hat Research https://research.redhat.com

For these reasons, it is important to further examine the repercussions that code smells have, and what types of smells are common in code bases. Within these next couple of sections, we will be covering how we collected our data along with the results found based on the repositories examined. Before discussing our results, however, we want to highlight other works that have impacted our approach to our case study and rationale.

Through undertaking this study, there were many challenges we encountered. Firstly, we were unsure what static analyzers were available for Java that best fit our needs. Although there are many add-ons and plugins that assist developers in refactoring their code in common IDEs (Integrated development environment)s, we wanted to utilize a tool that was unrestricted to a specific extension. As further discussed in later sections, the accuracy of each tool can differ, which was why our goal was to find a tool that could provide extensive feedback on different smells. Thus, ensuring that false positives of particular smells proved to be concerning and a challenge. Our research challenge was to observe the frequency of the Long Method and God classes, comparing this data to related works that have highlighted the severity of smells such as these. Furthermore, we sought to determine how often they appear in repositories compared to other smells.

This paper is organized by discussing some similar studies and works that have been conducted in more recent years. We will then cover the prevalent smells we recorded out of the repositories examined, highlighting the frequency of each smells. These findings are beneficial to developers who are interested in our selected tool and the common implementation, design, and architectural smells that exist in popular Java libraries on GitHub. We also discuss potential validity concerns, and how we minimized and reduced potential risks while collecting and analyzing the data. Further studies need to be conducted to gain knowledge on various tools, which we will examine in our future work.

II. Related work

Many papers have aimed to better detect code smells within software. Within the work of Paiva et al. [4], researchers conducted comparison studies on various code smell tools.

Their research suggests that each tool had a different amount of accuracy when identifying code smells such as *God Classes* or *God Methods*. The challenges of creating such tools with minimal inaccuracy is difficult, as some code smells could be erroneous or not be flagged properly.

Our selected code smell detection tool was *DesigniteJava* [1] [5]. Developed by Sharma et al., this particular tool is newer and provides a detailed assessment report about architectural, design, and implementation smells. Some of the naming conventions for each smell slightly differ, such as the terms *God Class* and *Insufficient modularization*, which are interchangeable for describing a class that can be further broken up to reduce complexity. These slightly differ from a *God Component*, which is an architectural smell that denotes an excessively large component (e.g., a package) that can be reduced or further broken up.

Further details associated with the definitions of each term (e.g. Unutilized abstraction, Deficient Encapsulation) can be found the works of Sharma et al. [6]. Although *Designite* supports the examination of C# code, we were most interested in Java projects only. The tool also offers more features and flexibility than other code smell detection tools such as *JDeodorant*.

JDeodorant, an Eclipse plug-in developed by Tsantalis et al. at Concordia University and the University of Macedonia [7], currently supports detection of 5 types of code smells at the time of this writing. Because of its dedication specifically to Java, it was a tool that was considered to be used for our selected repositories. However, the amount of time required to operate the tool on hundreds of repositories would be dramatically different due to DesigniteJava's flexibility and lack of limitations. This enables researchers the capability of creating scripts or programs to enhance the overall automation of collecting code smell results without the restriction and usage of Eclipse.

Another tool that has been used for code analsyis is *iPlasma*, which Marinescu et al. described [8]. This tool provides analysis of various metrics and can detect violations such as duplicate code. It is available in object-oriented languages such as C++ and Java.

Prior to this study, we asked ourselves what the pitfalls related to the current static analyzers and tools out there to detect various metrics were. Samarthyam, Suryanarayana, and Sharma mentioned some of the downsides to tools such as Sonargraph [9]. A non-exhaustive list of some of these worries were the following: lack of extensive support of architectural smells, lack of contextual information related to each smell, and limited availability of popular IDE (Integrated Development Environment) support for refactoring architectural smells [10]. While inspecting code smells, it is also important to filter out harmful and less severe smells. Highlighted in the upcoming sections, some of the uncertainties of these static analyzers are the filtering methods used along with the potential for false positive smells. In the works of Fontana et al. [11], they devised strong and weak filters which can be used to alleviate possible false positive instances. Furthermore, the strong filters proved to increase overall precision on detecting such smells.

[1]https://github.com/tushartushar/DesigniteJava

TABLE I
BRIEF OVERVIEW OF SOME OF THE SURVEY QUESTIONS ASKED

Question 2	On a scale from 1-5, how familiar are you with the term "code smells"?
Question 3	On a scale from 1 to 10, how often do you encounter violations within your software?
Question 6	Are you familiar with static code analyzers?
Question 7	If you answered "Yes" to the previous question, what types (or name) of tool(s) did/do you use to monitor violations (code smells) within your codebase/project?
Question 8	At your company, did you measure / use any kind of calculation to assess technical debt? (e.g., monthly reports on developer/cloud-usage cost/performance stats, etc.)
Question 9	At your company, did you use any visualization tool to assess technical debt? If so, how did you visualize it?
Question 10	Besides static code analyzers, how much time would you say you spent manually observing and inspecting code for violations per week?
Question 13	How much time did you typically spend refactoring code at your company, per week?

Similar case studies have been done in the past, such as in the writings of Sharma, Fragkoulis, and Spinellis, where they examined C# repositories in order to inform the reader about characteristics of code smells in C# code bases [6]. Their results showed that both the *Unutilized Abstraction* and *Magic Number* were the most frequently occurring smells in C# code [6]. Rather than selecting their repositories manually, they utilized *RepoReaper* [12] to gather their repositories. The overall findings are similar to our results, which will be further discussed in later sections.

III. PRACTITIONER SURVEY

Before data collection, we sought to identify the typical amount of time developers spend refactoring code and inspecting it for violations. A survey was created in order to gain insight from individual's experiences with code violations. This allowed us to attain an understanding on the currently adopted tools and how familiar the respondents were of code smells. A total of 14 questions were asked with initial questions pertaining to a generalization of technical debt and evaluating economical cost associated with it. Due to relevancy of the paper, those questions were omitted from the first table. The feedback was collected through google forms and served primarily as a basis for determining what tools were used to monitor code violations. A total of 14 individuals were surveyed and their feedback will be discussed in further sections.

Many surveys have been conducted in the past to accumulate feedback from software developers. An example of such is in the work of Yamashita and Moonen, where 85 software developers were surveyed to gain insight on their thoughts of code smells [13]. The results from this study showed that

32% of respondents had no prior knowledge of code smells. Furthermore, the most mentioned smells that were familiar to the developers were smells such as the *Large Class* and *Long Method*.

Another survey that has been conducted is evident in the works of Golubev et al. [14], where over 1100 individuals were surveyed in order to determine how often they spent refactoring code. Furthermore, they assessed how developers refactored their code. Their findings suggested that two-thirds of developers spent longer than an hour refactoring code for every instance spent working. They also found that 40.6% of developers refactored code almost every day [14].

Based on the tiny sample we collected from our survey, almost all respondents that are currently in the industry said that they spent less than 5 hours a week refactoring code. On the contrary, 70% of the respondents for the Graduate survey stated that they spent 5 hours of more a week refactoring code while they were in industry. Although further studies would need to be conducted, a possibility for this differentiation is that the Graduate students who opted to participate in the survey are newer developers or have not spent a lot of time in industry. Furthermore, these individuals could be a younger demographic that has not had an immense exposure as a software developer compared to their peers. As a result, it could take more time for those individuals to refactor code due to the lack of experience. Based on this minuscule sample-size however, much more research would need to be conducted for further evaluation and verification of the possibilities mentioned.

From the individuals surveyed, some tools utilized for code quality were SonarQube, JavaParser, and Jacoco. Others mentioned linters such as ESLint and golint to help improve the quality of production-based code while working with continuous integration pipelines. When asked whether or not there was any visualization tool used for assessing technical debt, all respondents from both surveys stated that they were unfamiliar such tools or did not employ any. For this reason, we sought tools that can be beneficial to developers, which are discussed in prior sections.

Code smells can take a lot of time to evaluate and identify. Because of the immense amount of allocated resources spent, researchers have attempted to alleviate the time spent refactoring by developing tools that can minimize the amount of time in identifying the smells to fix. This can impact the cost of the company as funds would be used for refactoring and fixing rather than innovation and enhancements, impacting profits over a long period of time.

IV. Proposed Method

A. Methodology

For this study, we gathered 100 repositories that were as close as possible to being fully written in Java due to the uncertainty of how DesigniteJava would perform. Another requirement we decided upon was that each repository contains at least 3000 lines of code for it to be considered. As mentioned in further sections, this was due to alleviating potential inconsistencies and ensuring that one particular smell would not be disproportionate to other smells. This particular minimum value was selected due to testing the tool with smaller repositories and discovering this value to be suitable and adequate for our sample size.

Most repositories examined were libraries and frameworks that were highly active and recommended when searching for repositories on GitHub. These repositories were chosen at random to prevent any biases. Each repository would also be thoroughly checked to ensure that no other languages would be scanned in by our chosen tool.

B. Data Extraction

Once validated and identified, the repositories are downloaded and unzipped. DesigniteJava is then utilized to generate an XML file, which can be read in by *QScored* [15]. Provided by Sharma et al., QScored agent is readily available for analyzing information from DesigniteJava and uploading it to QScored for a visual representation along with computing a raw score for the particular repository [15]. Our primary focus was extracting the data returned by the DesigniteJava output and parsing it to collect the summation of total smells within that particular repository as well as examining the number of instances of *God Classes or God Components* and *Long Methods* detected. For this reason, QScored was only used for initial testing to visualize some of the earlier repositories. It was also used for pinpointing files within each repository that had high levels of lines of code, which is beneficial for our future work. We tested out the feature and modified the sample python source code provided with a granted API key to see how simplistic it is to convert the XML file to a visual depiction on QScored.

To organize the data, a shell script was created for running each repository selected against DesigniteJava. This script also generated new sub-directories to store each file obtained by the tool for every repository examined. For each repository, the results were redirected from standard output and appended to one large result file. The name of the repository would also be appended subsequently after redirection occurred so that the data would have proper association for future parsing. After running the detection tool on all of the repositories, the result file would be parsed and sorted into manageable data using other programs. A majority of the original output data would be discarded and only the total lines of code, top two code smells recorded, and number of instances of particular smells for each repository would remain for the final data set. The smells that remained in our final data set were the following: *Long Method*, *Magic Number*, *Long Statement*, *God Components*, and *Unutilized Abstraction*. As further discussed in our results section, these were ultimately chosen as the remaining smells due to their popularity and sole focus of our research.

The data collected would then be easily transferable and converted to a readable format, such as an excel sheet or CSV file. The results of the study are an open data source and

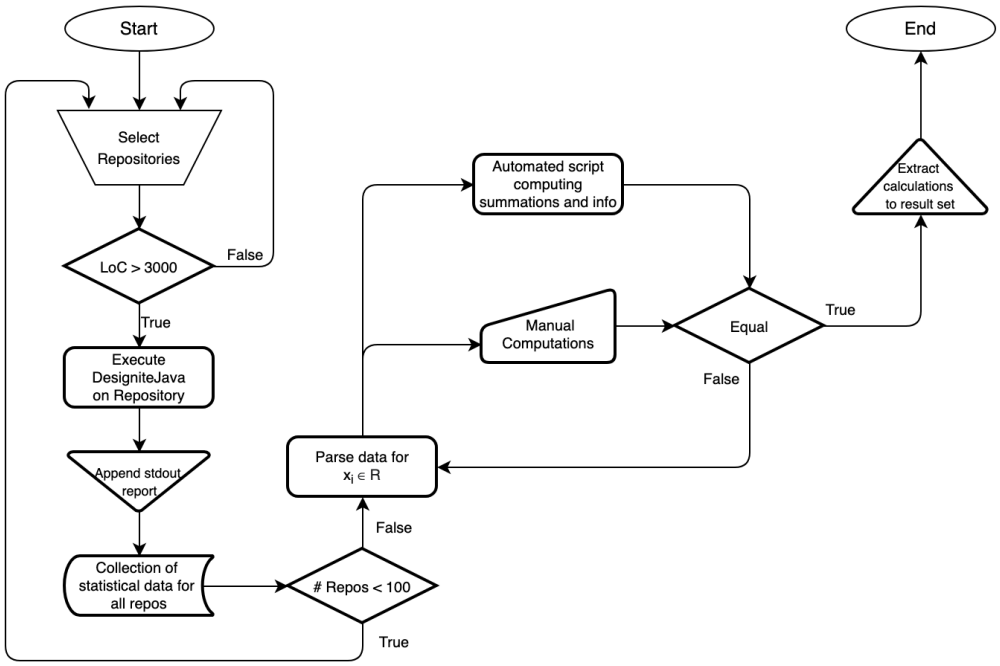

Fig. 1. Flowchart of our process

accessible through GitHub[2]. Our process for collecting the data is depicted in Figure 1, which entails each step conducted to obtain the final data set.

V. Case Study

A. Results

Prior studies have mentioned the prevalence of specific smells within software projects, however, our findings suggest that some smells such as the *Long Method* and *God Components* are fairly uncommon. Based on our results, the *Long Method* was documented for 0.144% of all methods scanned. The *Long Method* also only accounted for 0.255% of all smells collected, which is reasonably minuscule. This number was heavily impacted due to other code smells that seemed to be highly widespread. On the other hand, both the *Long Method* and *God Components* contain significantly more lines of code compared to others such as the *Magic Number*. Although they are not as prevalent, both smells still make up a significant portion of total lines of all smells. The *Magic Number* smell was one of the more common smells detected within the repositories, however, in most instances it can be omitted in real-world practices as a potential smell. It made up 48.357% of all code smells detected, making it substantial compared to other smells. The second most common smell was a *Long Statement*, which comprised of 21.425% of all smells. Both the *Magic Number* and *Long Statement* together appeared as the most common and second common code smell for 70.0% of all repositories.

[2]https://bit.ly/39L0iIH

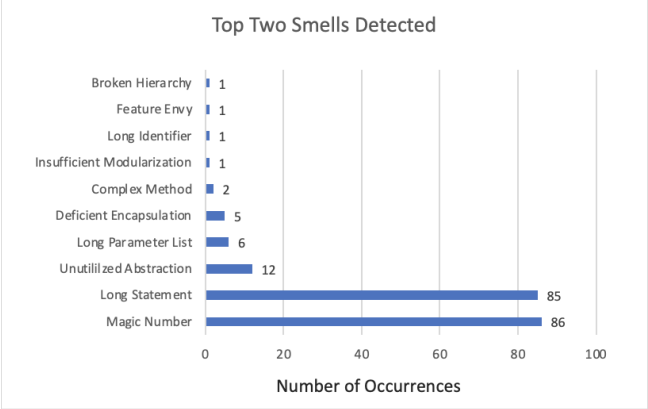

Fig. 2. Frequency of top two smells detected

TABLE II
Overall Averages of each Repository selected

Attribute	Average (AVG)
Number of Smells Detected	3924.04
Lines of Code (LoC)	59509.73
Number of Methods	6926

Some code smells are insignificant compared to others. To clarify, a few smells such as a *Magic Number* could be seen as appropriate or easily modifiable in specific instances compared to other smells that can have long-lasting effects and degrade the overall quality of the software. Because of instances where there could be valid solutions that require code segments to be

designed a particular way as well as many other factors, smells like these can be discarded under most circumstances. Other code smell that were flagged many times were *Unutilized Abstraction* and *Broken Hierarchy*. For *Unutilized Abstraction*, 6.654% of total code smells were devised of this violation. This particular smell was third most prevalent and relative to the top two smells, the overall percentage is substantially small. Out of the total lines of code analyzed, only 6.594% of all repositories contained violations. This ratio was calculated by computing the total number of smells in all 100 repositories, and divided by the total lines of code in each program. This is only an estimate, as some of the code smells could take several lines of code. The outcome was an expected result, as these libraries are likely maintained by experienced developers who practice good coding habits. Likewise, the developers are likely to follow these standards to ensure high scalability and to maintain their libraries efficiently.

From this study, we can confirm that similarly to C# code analyzed in Sharma et. al [6], that *Magic Number* and *Unutilized Abstraction* are both prevalent smells that are also common in Java repositories. Furthermore, we can conclude that the commonality between both *Long Statement* and *Magic Number* frequencies are significantly higher than other smells, which is depicted in Figure 2.

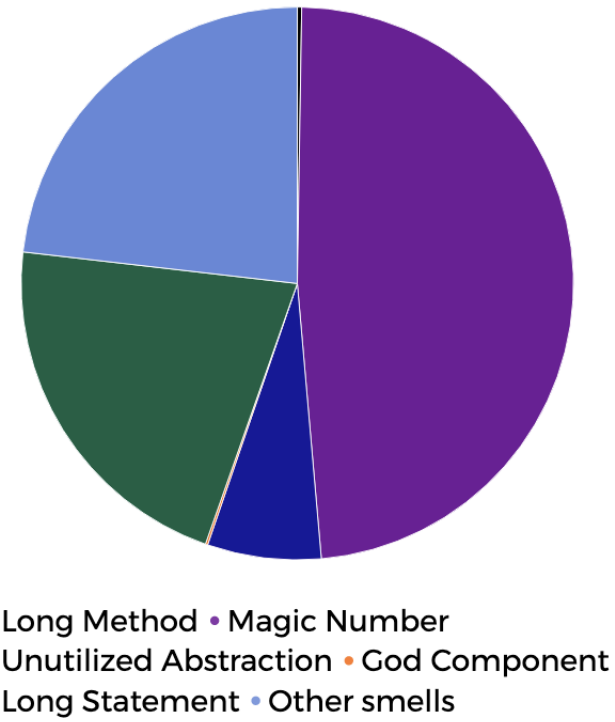

Fig. 3. Pie Chart displaying distribution percentages of particular smells out of all 100 repositories examined

B. Threats to Validity

1) Internal Validity: Bias in regards to data extraction could be a potential issue. Only Java projects were selected due to the limitations of DesigniteJava. Because of the restriction, different types of smells could potentially be more or less common in other languages. Although the data was validated through several automated checks, some of the smells were manually checked and computed. Specifically, all smells in our final data set were manually calculated.

To prevent any errors, these manual operations were compared with programs to validate that the summations equalled the summations calculated initially by hand and vice versa. In order to alleviate any potential conflicts, popular repositories from experienced developers that contained several thousands of lines of code were primarily selected to be analyzed. Because of the restrictions on the number of lines required, this removes the possibility of special cases where there could be an outlier of a particular code smell.

2) External Validity: A minor subset of individuals who were surveyed haven't been within the industry in the past couple years. To reduce these biases, the survey data was separated by those who are currently employed as a software developer, and those who recently departed from the industry, such as for Graduate School. Within the software field, new tools and technologies are rapidly evolving, which allow developers to spend significantly less time refactoring their code. As a result, these former developers could have potentially utilized outdated tools even if they have only been out of the industry for a few years, which could heavily impact the amount of time it takes to refactor code.

In regards to the repositories collected, the data extracted was reliant on the accuracy of the tool. Because of the tool being recently developed and not as widely used overall, DesigniteJava could pick up false positives of a particular smell. Conducting the study again with different static analyzers or other tools that detect code smells and comparing the results to the current data set would lessen the probability of inaccurate data. Furthermore, due to these popular repositories constantly changing with updates, the number of code smells for each repository could differ at the time of this writing.

VI. Future Work

As previously discussed, it would be beneficial to utilize these exact same repositories but with many tools to compare the rate of success and accuracy of each tool. A larger data set expanding the use of other languages would be advantageous to ensure that the most frequent smells identified within this study is widespread in other languages besides Java. Although Java and C# projects are compatible with our selected tool, further identification of analyzers that support other object-oriented languages such as Python or C++ need to be tested. This will allow for identifying potential relationships between different design, architectural, and implementation smells in contrary projects.

Because of the programs created and utilized as a result of this experiment, a data set with thousands of repositories can be applied under the same circumstances as the small selection of this case study at significant rate. The only limitation is the manual selection of each repository, since criteria must be met for every selected repository to ensure valid data collection.

For our future work, we plan to devise or identify software that will assist in fetching repositories, while adhering to our criteria. Some static analyzers only support specific languages and are potentially vulnerable to issues if conflicting files are also embedded in the same repository.

Our future work entails examining the code that was associated with the *Long Method* and *God Components* for the repositories selected, and running code smell detectors and analyzers on these sections of code. The results could potentially detect other embedded code smells within these particular smells, as well as a correlation between the creation of these oversized classes or methods. Furthermore, many tools would be utilized, ensuring that a wide range of smells are fetched from each repository.

VII. Conclusions

Based on the repositories examined, the *Long Method*, *God Components* and other architectural and design smells are typically not as detected in well-developed Java repositories compared to other code smells. The most common smells recorded were the *Long Statement* and *Magic Number*, accounting for 69.782% of all smells recorded within the chosen repositories. The *Long Method* was recorded for 0.144% of all methods in the repositories selected. It also only accounted for 0.255% of all total smells analyzed. Although insignificant smells such as the *Magic Number* disproportionately impacted other smells' percentage weight, if this smell was omitted, the *Long Method* would only slightly increase to 0.494% of all detected smells. Future studies will be conducted to gather a larger sample size and analyze the *God Components* and *Long Method* segments of code within each repository fetched. DesigniteJava is an extremely beneficial tool, which both researchers and those in industry can utilize to further evaluate production code and long-term maintainability.

Acknowledgments

This research was funded by National Science Foundation grant number 1854049 and a grant from Red Hat Research https://research.redhat.com

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915001743

[3] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards assessing software architecture quality by exploiting code smell relations," in *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*, 2015, pp. 1–7.

[4] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, p. 7, 12 2017.

[5] T. Sharma, "Designitejava," Dec. 2018, https://github.com/tushartushar/DesigniteJava. [Online]. Available: https://doi.org/10.5281/zenodo.2566861

[6] T. Sharma, M. Fragkoulis, and D. Spinellis, "House of cards: Code smells in open-source c# repositories," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '17. IEEE Press, 2017, p. 424–429. [Online]. Available: https://doi-org.ezproxy.baylor.edu/10.1109/ESEM.2017.57

[7] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1037–1039.

[8] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, and R. Wettel, "iplasma: An integrated platform for quality assessment of object-oriented design." 01 2005, pp. 77–80.

[9] "Sonargraph-quality: A tool for assessing and monitoring technical quality." [Online]. Available: https://www.hello2morrow.com/products/sonargraph/quality

[10] G. Samarthyam, G. Suryanarayana, and T. Sharma, "Refactoring for software architecture smells," in *Proceedings of the 1st International Workshop on Software Refactoring*, ser. IWoR 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–4. [Online]. Available: https://doi-org.ezproxy.baylor.edu/10.1145/2975945.2975946

[11] F. A. Fontana, V. Ferme, and M. Zanoni, "Filtering code smells detection results," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. IEEE Press, 2015, p. 803–804.

[12] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *PeerJ Preprints 4:e2617v1*, 2016.

[13] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 242–251.

[14] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer, "One thousand and one stories: A large-scale survey of software refactoring," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1303–1313. [Online]. Available: https://doi-org.ezproxy.baylor.edu/10.1145/3468264.3473924

[15] V. Thakur, M. Kessentini, and T. Sharma, "Qscored: An open platform for code quality ranking and visualization," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 818–821.