Merging Smell Detectors: Evidence on the Agreement of Multiple Tools

Anonymous Author[†]
Affiliation
Email

Anonymous Author
Affiliation
email

Anonymous Author
Affiliation
email

Anonymous Author
Affiliation
email

ABSTRACT

Technical Debt estimation relies heavily on the use of static analysis tools looking for violations of pre-defined rules. Largely, Technical Debt principal is attributed to the presence of low-level code smells, unavoidably tying the effort for fixing the problems with mere coding inefficiencies. At the same time, despite their simple definition, the detection of most code smells is non-trivial and subjective, rendering the assessment of Technical Debt principal dubious. To this end, we have revisited the literature on code smell detection approaches backed by tools and developed an Eclipse plugin that incorporates six code smell detection approaches. The combined application of various smell detectors can increase the certainty of identifying actual code smells that matter to the development team. We also conduct a case study to investigate the agreement among the employed code smell detectors. To our surprise the level of agreement is quite low even for relatively simple code smells threating the validity of existing TD analysis tools and calling for increased attention to the precise specification of code and design level issues.

CCS CONCEPTS

• Maintaining software • Software maintenance tools • Quality assurance

KEYWORDS

Code Smells, Technical Debt, Principal, Refactoring

ACM Reference format:

FirstName Surname, FirstName Surname and FirstName Surname. 2022. Code Smell Detection: Evidence from multiple Tools. In *Proceedings of 5th International Conference on Technical Debt (TechDebt'2022). ACM, New York, NY, USA, 5 pages.* https://doi.org/10.1145/1234567890

1. Introduction

Poor software quality can be expressed in various ways: as non-conformance to design principles, lack of design patterns, excessive metric values, violations of heuristics, existence of Technical Debt and through the presence of code smells. Code smells, introduced by Fowler in 1999 [1], are defined as bad practices when writing code and have become very popular among developers since they capture in a systematic, yet simple manner, habits leading to less maintainable code. While an experienced developer can usually identify code smells simply by looking through the code given that he / she is familiar with their definition, smell identification should be automated considering their frequency in even small-scale systems. Code smells form one of the main pillars for Technical Debt (TD) assessment in most contemporary TD management tools.

To facilitate the automatic identification of code smell instances representing opportunities for refactoring, various tools have been developed, either commercial or as research prototypes, that are capable of detecting code smells and displaying them in a user-friendly way. Such automation of the detection process can have a very positive impact on both quality and productivity. An empirical study [2] showed that automated smell detection can save time and when compared to a manual review of the results it increases the confidence about the detected smells.

Although automated detection of smells can be of great help, extra care needs to be taken when selecting a tool, since the adopted detection techniques determine the obtained results. Rasool and Arshad [3] presented a review of state-of-the-art tools for code smell detection, which vividly showed that precision and recall differ largely between tools even for those that rely on the same technique. Such reviews are important because they can highlight advantages and disadvantages of using these tools and distinguish the most appropriate one in each occasion.

Despite the availability of various code smell detection tools, it can be time consuming to either find already analyzed cases or get access to executable files, plug-ins or source code of the tools of interest, to extract conclusions from the results or to conduct case studies. In this paper, we introduce an Eclipse plug-in [4], namely: *Smell Detection Merger*, which incorporates six (6) code smell detection tools (some of them were introduced as research products) and aggregates the results. Smell Detection Merger allows the user to easily access and use the underlying tools to conduct experiments in a direct way and get aggregate results.

Accurate code smell detection is of paramount importance in the context of Technical Debt (TD) estimation for two reasons: First, TD tools rely on static source code analyzers to identify code smells as TD issues, whose refactoring effort is summed up to measure the principal of the software. Unreliable smell detection approaches render the resulting TD measurement inaccurate. As an indicative example of the importance of code smells for TD quantification, project *commons-imaging* of the Apache ecosystem, entails according to the widely used SonarQube TD assessment tool 1,307 TD issues of which 1,236 refer to code smells

(94.5%). In terms of effort to repay TD, the principal is measures as 26 days, out of which 25 days correspond to the effort for mitigating code smells. Second, most TD tools focus on code-level smells leaving out smells pertaining to the design of methods, classes and packages. We performed a case study on the level of agreement among the examined smell detectors to shed light into the maturity of the code smell concept in terms of common definition and common detection approaches.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. In section 3, the developed plug-in is presented while Section 4 presents a case study on an open-source project using our plug-in and sheds light into the level of agreement among tools. Finally, we conclude in section 5.

2. Related Work

In the literature there are various tools on smell detection; therefore, comparisons between them are essential to determine which of them produce the best results, their underlying detection techniques and their accuracy. Fontana et al. [5], in their experimental assessment, reviewed several code smell detectors and presented a list of all the smells supported by the tools in order to find the common ground between them. Their case study considered four tools, namely JDeodorant, inFusion, PMD and CheckStyle while findings on six code smells have been compared. Interestingly, the results showed very high agreement ratio between the tools for each smell type in all of the selected versions. A comparative study was also conducted by Hamid et al. [6] that consisted of a comparison of two tools, namely. JDeodorant and inCode, followed by an extensive analysis of their detection results on two common code smells, Feature Envy and God Class. The authors concluded that the results between the tools differed significantly, mainly due to the different approaches they follow on detecting these smells.

Rasool and Arshad [3] performed an extensive review on the state-of-the-art techniques and tools for mining code smells from source code. Code smell detection techniques and tools are classified based on their detection methods into: manual detection, metrics-based, symptoms-based, probabilistic-based, visualization-based, search-based and cooperative-based techniques. Based on results in the literature the authors observed a wide disparity in the results among different tools. The same conclusion was reached through experimentation, in which the authors installed three code smell detection tools, namely CheckStyle, JDeodorant, and PMD for comparing their results on four common code smells.

Another extended review was conducted by Fernandes et al. [7] on the state-of-the-art code smell detectors in order to provide an overview for each tool. For each of the tools they listed essential details like the programming language it was developed as well as the target language that can be analyzed, the employed detection technique, the code smells it can detect, if it is free to use, if a GUI is provided, etc. In terms of this review, the authors performed an experiment with four of the tools, i.e., inFusion, JDeodorant, PMD, JSpIRIT and compared the results of detecting two smells, namely Large Class and Long Method. Kaur and Dhiman [8] performed a review of tools and techniques for code smell detection. The authors highlighted that the comparison between tools and approaches can be a challenging task due to the fact that many of them are not publicly available to researchers and also the experiments that are conducted with these tools, are often carried out on different projects for different smell types. For their review, they considered JDeodorant and CodeNose tools for the detection of smells such as Feature Envy, Large Class, Long Method, Long Parameter List, Refused Bequest.

Table 1 provides a summary of previous studies along with number of tools and smells considered in this paper. As it can be seen, we perform our study with a larger set of tools and on a significantly larger set of code smells.

Paper	Tools	Code Smells
Fontana et al. [5]	JDeodorant inFusion PMD CheckStyle	Duplicate Code, Feature Envy, God Class, Large Class, Long Method, Long Parameter List
Hamid et al. [6]	JDeodorant inCode	Feature Envy, God Class
Rasool and Arshad [3]	CheckStyle JDeodorant PMD	Large Class, God Class, Long Parameter List, Long Method
Fernandes et al. [7]	inFusion JDeodorant PMD JSpIRIT	Large Class, Long Method
Kaur and Dhiman [8]	JDeodorant CodeNose	Feature Envy, Large Class, Long Method, Long Parameter List, Refused Bequest
This work	CheckStyle, DuDe PMD JDeodorant JSpIRIT Organic	Brain Class, Brain Method, Class Data Should be Private, Complex Class, Data Class, Dispersed Coupling, Duplicate Code, Feature Envy, God Class, Intensive Coupling, Lazy Class, Long Method, Long Parameter List, Message Chain, Refused Bequest, Shotgun Surgery, Spaghetti Code, Speculative Generality, Tradition Breaker, Type Checking

Table 1: Summary of related work experiments

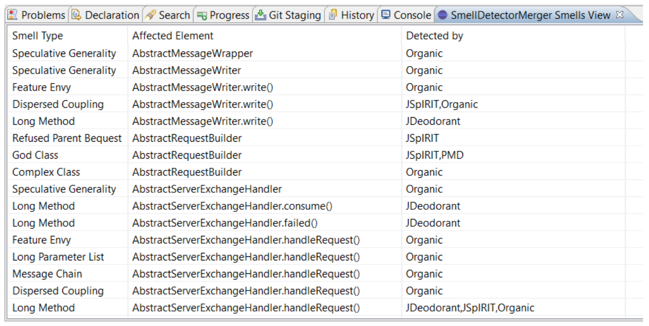
3. Smell Detector Merger

For the purpose of this study, a non-systematic survey was conducted to identify existing tools for code smell detection written in Java, which support the examination of Java projects. During this research, several tools were found, but only 6 were eventually included under the umbrella of the unified detector we developed, due to various difficulties. The search for tools was done in Google Scholar to find articles related to code smell detectors, as well as reviews or comparisons of such tools. Primarily, we used "*code smell detection tools*" and "*code smell detection techniques*" as our search strings to get relevant results. These searches mainly returned literature reviews such as [7] and [8]. Such studies present various detectors, their implemented detection techniques and their accuracy. Many of the tools that we have identified in these reviews or individually, were not available or maintained by the original authors any more, while others were not a good fit for our purpose, since their main feature was visual representation of the detected smells directly in the IDE. In addition, there were a few more for which we could not get access to their code and we had difficulties contacting their authors. Nevertheless, some of the tools we used are among the most commonly seen tools in literature reviews related to code smell detection.

So far, we have integrated the following six tools: CheckStyle [9], DuDe [10], PMD [11], JDeodorant [12], JSpIRIT [13], and Organic [14]. The plug-in was developed in Java 8 and tested using Eclipse IDE version 2021-06 (4.20.0). The first half of the tools are standalone and are executed by invoking either a `jar` or `bat` file via our plug-in. The second half consists of plug-ins, which were exported as `jar` files and then added to our plug-in as external libraries to have access to their internal classes and methods. For some of them, a few minor changes were made to their source code that had no effect to their detection logic, to expose publicly specific functionality for our convenience. In addition, we need to highlight that some of the tools provide a level of customization, though this is not currently supported via our plug-in. We aim to include this as part of future work.

The code smells that each of the previously mentioned tools can detect add up to a total count of twenty. Some of the smells are more "*popular*" and can be detected by more than one tool (e.g., *Long Method* and *God Class*), while others are less popular and can only be detected by a single tool (e.g., *Tradition Breaker* and *Type Checking*). For more details about the smells that can be detected by each tool, either the corresponding references or the GitHub[1] page of our plug-in can be used.

The developed plug-in offers various options to the user to customize the detection process, by defining the set of tools to be used for the detection, along with the type of smell (or all of them) to be detected. After the completion of the detection process, the results are displayed in a single view in the IDE, as shown in Figure 1**.** Each row displays a different smell that was detected and consists of the smell type, the affected element (could be either a class or a method) and a list of tools that detected the smell separated with a comma. If needed, each column can be sorted alphabetically in ascending or descending order. Moreover, if the user double-clicks on a smell, the corresponding resource will open in the IDE at the line in which the smell was detected. Additionally, the detected smells can be filtered to keep only those that were detected by more than 2 (in absolute number) or 50% of the tools. The previous two filtered sets can also be used as (tentative) gold standard sets. This is very important because as such they can be utilized to calculate precision and recall for each of the tools for all its supported smell types. Finally, it is worth mentioning that the results of the detected smells as well as their filtered sets can be exported to a `CSV` file for later use.



Smell Type	Affected Element	Detected by
Speculative Generality	AbstractMessageWrapper	Organic
Speculative Generality	AbstractMessageWriter	Organic
Feature Envy	AbstractMessageWriter.write()	Organic
Dispersed Coupling	AbstractMessageWriter.write()	JSpIRIT,Organic
Long Method	AbstractMessageWriter.write()	JDeodorant
Refused Parent Bequest	AbstractRequestBuilder	JSpIRIT
God Class	AbstractRequestBuilder	JSpIRIT,PMD
Complex Class	AbstractRequestBuilder	Organic
Speculative Generality	AbstractServerExchangeHandler	Organic
Long Method	AbstractServerExchangeHandler.consume()	JDeodorant
Long Method	AbstractServerExchangeHandler.failed()	JDeodorant
Feature Envy	AbstractServerExchangeHandler.handleRequest()	Organic
Long Parameter List	AbstractServerExchangeHandler.handleRequest()	Organic
Message Chain	AbstractServerExchangeHandler.handleRequest()	Organic
Dispersed Coupling	AbstractServerExchangeHandler.handleRequest()	Organic
Long Method	AbstractServerExchangeHandler.handleRequest()	JDeodorant,JSpIRIT,Organic

Figure 1: Smell detection results displayed in a single view

A notable advantage offered by this plug-in, is that it can save a significant amount of time to researchers who want to perform reviews for existing code smell detection tools. By using this plug-in, one no longer has to get access to each separate tool, download and install it independently, get the results for each one of them and eventually compare the results with the rest. Instead, the user can select the tools of interest as well as the smell(s) to be detected and get the combined results with one click.

4. Empirical Evidence on Tools Agreement

To perform a preliminary evaluation of the tools, we conducted a small-scale study on an open-source project. The project we used is `Apache HTTP Components` [15]; which provides a set of low-level tools for implementing the `HTTP` and other relevant protocols. The project consists of four Maven sub-projects, which were all imported in Eclipse and then each one was analyzed using Smell Detector Merger. The results were exported to a `CSV` file in order to post-process them.

Figure 2 provides an overview of the smells that were detected by Smell Detector Merger. The top left cell indicates the aggregate results, while all other cells correspond to a single smell. The bar charts display the tools that detected each smell, along with the total occurrences they found. It becomes obvious that in most cases there is a notable difference in the total number of identified smells between two or more tools. For some smell types the difference is substantial, as for example in the case of ***Shotgun Surgery***, for which *JSpIRIT* detected 91 instances, while *Organic* only 13. The same applies for ***Long Method***, for which *JDeodorant* detected the most instances (727 smells), while *Organic* detected 192 instances and *CheckStyle*, only reported 5 occurrences. Through the figure the reader can grasp the popularity of each smell, and can focus on the smells which can be detected by at least two of the integrated tools. From the entire set of 20 supported smells, 8 are detected by only one of the tools, while the other 12 are supported by more than one tools, enabling a comparison of the results.

[1] https://github.com/detectormerger/SmellDetectorMerger

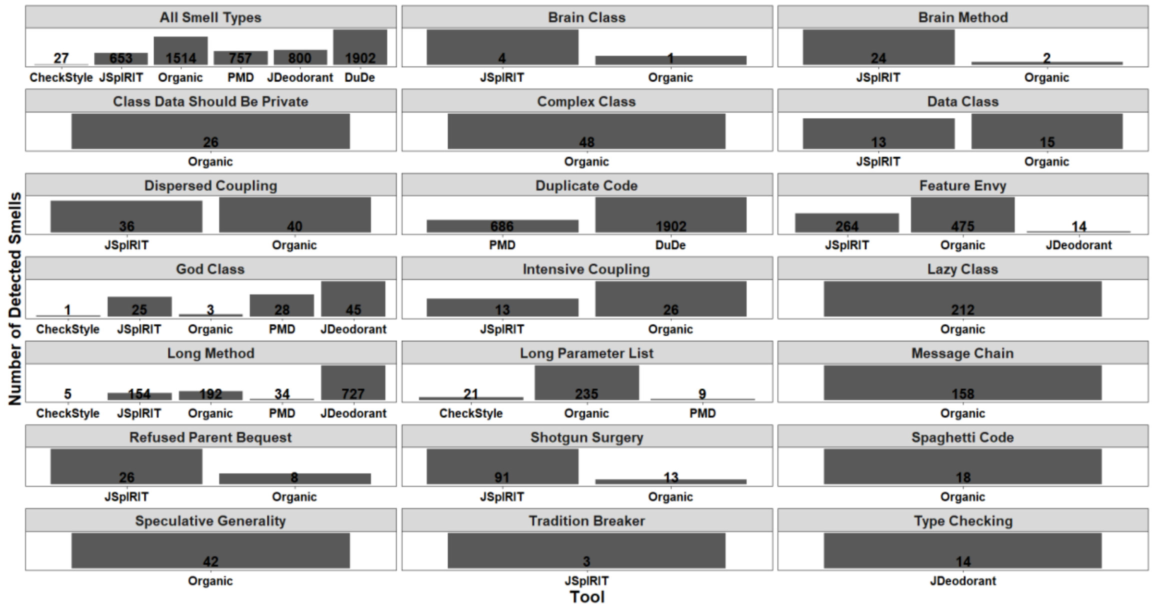


Figure 2: Overview of the detected smells and their total occurrences found per tool

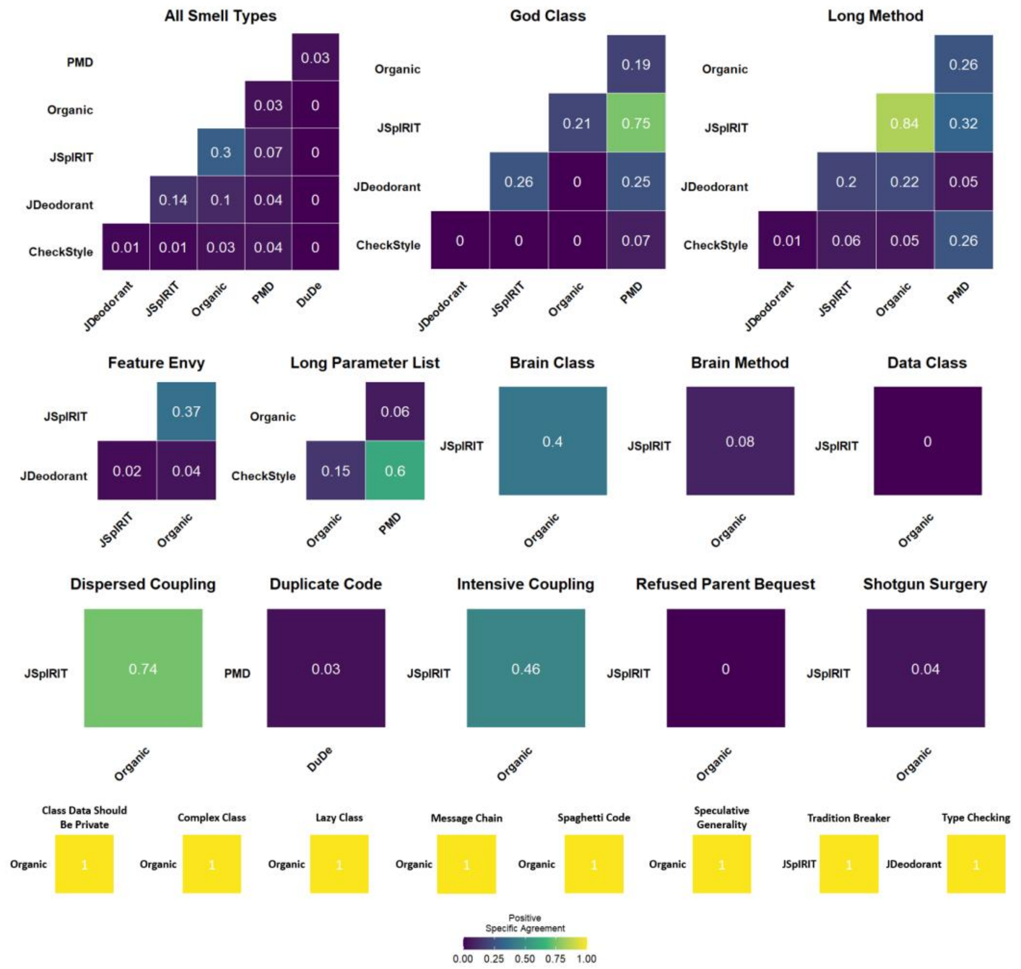


Figure 3: Positive Specific Agreement

Apart from the number of identified smell instances, it is important to investigate the agreement on the detected code smells among the tools. To achieve that, we relied on the positive specific agreement [16] measure which "*is the conditional probability that one rater will agree that a case is positive given that the other one rated it positive, where the role of the two raters is selected randomly. It approximates the proportion of positive cases that were agreed on*". We decided to use this measure because it only takes into consideration the cases in which two or more tools agree on a detected smell. In other words, artifacts which are not being reported as smelly by the tools are of less interest in the general case.

The results regarding the level of agreement are shown in Figure 3. Each sub-diagram corresponds to a smell (the top-left applies to all smells), while the tools which are capable of detecting that smell are shown vertically and horizontally. On the intersection of two different tool the calculated positive specific agreement is shown in color and numerically. It is striking that for most of the cases, the calculated positive specific agreement has a value of 0 or one very close to that. This shows that the ***different techniques used by the smell detectors, lead to very different reported smell instances***. This raises questions about which techniques are more efficient and / or implemented in a better way, as well as which of the reported cases correspond to an actual smell of the detected type in a given software product. Without a commonly-agreed ground truth it becomes challenging to rate the accuracy of each tool. Nevertheless, instances on which multiple tools agree provide some level of confidence in the existence of a problem. It should be noted that there are only a few exceptions to the previous observation, such as ***Dispersed Coupling*** which is detected by *JSpIRIT* and *Organic*, ***God Class*** detected by *JSpIRIT* and *PMD*, ***Long Parameter List*** found by *CheckStyle* and *PMD*, as well as ***Long Method*** reported by *JSpIRIT* and *Organic*, where the positive specific agreement is equal or greater than 0.6. However, a high level of agreement can be claimed only for values above 0.8 which is observed only in one case.

5. Conclusions and Threats to Validity

For the reliable assessment of Technical Debt in software projects it is of paramount importance to accurately detect code smells, which form one of the main pillars of TD. Code smell detection tools can play an important role to this goal by automatically identifying refactoring opportunities. In this paper, we integrated six widely used code smell detectors and conducted a case study with an open-source project to find the total smell instances detected by each tool, and to study the agreement among them. The results showed that for most smells the agreement level is quite low, raising questions about their subjective criteria and rendering it unsafe to consider a reported case as an actual smell.

We acknowledge that the presented case study suffers from threats to external validity as only one project was analyzed. We plan to conduct more systematic analysis on a larger number of projects. To mitigate threats to construct validity we plan to experiment with the configuration settings of each tool, wherever this is possible, so as to obtain more meaningful results.

ACKNOWLEDGMENTS

Work reported in this paper has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871177 (project: SmartCLIDE).

REFERENCES

[1] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 1999.

[2] J. Schumacher, N. Zazworka, F. Shull, C. Seaman and M. Shaw, "Building Empirical Support for Automated Code Smell Detection," Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement, pp. 1-10, 2010.

[3] G. Rasool and Z. Arshad, "A review of code smell mining techniques," Journal of Software: Evolution and Process, pp. 867-895, 9 September 2015.

[4] E. Clayberg and D. Rubel, Eclipse Plug-ins, Addison Wesley Professional, 2009.

[5] F. A. Fontana, P. Braione and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," Journal of Object Technology, January 2012.

[6] A. Hamid, M. Ilyas, M. Hummayun and A. Nawaz, "A Comparative Study on Code Smell Detection Tools," International Journal of Advanced Science and Technology, vol. 60, pp. 25-32, 2013.

[7] E. Fernandes, J. Oliveira, G. Vale, T. Paiva and E. Figueredo, "A review-based comparative study of bad smell detection tools," Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, pp. 1-12, June 2016.

[8] A. Kaur and G. Dhiman, "A Review on Search-Based Tools and Techniques to Identify Bad Code Smells in Object-Oriented Systems," Harmony search and nature inspired optimization algorithms, pp. 909-921, January 2019.

[9] CheckStyle, Available: https://checkstyle.sourceforge.io/ .

[10] R. Wettel and R. Marinescu, "Archeology of code duplication: recovering duplication chains from small duplication fragments," Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05), p. 8, 2005.

[11] PMD, Available: https://pmd.github.io/

[12] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 347-367, May/June 2009.

[13] S. Vidal, H. Vazquez, J. A. Dıaz-Pace, C. Marcos, A. Garcia and W. Oizumi, "JSpIRIT: A Flexible Tool for the Analysis of Code," 2015 34th International Conference of the Chilean Computer Science Society (SCCC), pp. 1-6, 2015.

[14] Organic, Available: https://github.com/opus-research/organic

[15] Apache HTTP Components, Available: https://hc.apache.org/

[16] G. Hripcsak and A. S. Rothschild, "Agreement, the f-measure, and reliability in information retrieval," Journal of the American medical informatics association, vol. 12, no. 3, pp. 296-298, 5 May 2005.