CLEAN++: Code Smells Extraction for C++

Tom Mashiach, Bruno Sotto-Mayor
Ben Gurion University of the Negev
Be'er Sheva, Israel
Emails: {tommas,machadob}@post.bgu.ac.il

Gal Kaminka
Bar Ilan University
Ramat Gan, Israel
Email: galk@cs.biu.ac.il

Meir Kalech
Ben Gurion University of the Negev
Be'er Sheva, Israel
Email: kalech@bgu.ac.il

***Abstract*—The extraction of features is an essential step in the process of mining software repositories. An important feature that has been actively studied in the field of mining software repositories is bad code smells. Bad code smells are patterns in the source code that indicate an underlying issue in the design and implementation of the software. Several tools have been proposed to extract code smells. However, currently, there are no tools that extract a significant number of code smells from software written in C++. Therefore, we propose CLEAN++ (Code smeLls ExtrActioN for C++) [1]. It is an extension of a robust static code analysis tool that implements 35 code smells. To evaluate CLEAN++, we ran it over 44 open-source projects and wrote test cases to validate each code smell. Also, we converted the test cases to Java and used two Java tools to validate the effectiveness of our tool. In the end, we confirmed that the CLEAN++ is successful at detecting code smells.**
The tool is available at https://github.com/Tomma94/CLEAN-Plus-Plus.

I. Introduction

The process of extracting and analyzing data from software repositories is an important task to uncover relevant and actionable information about software systems. An important feature that has been actively studied in the field of mining software repositories is bad code smells [2]–[7]. **Bad code smells**, also known as design defects [8], are patterns in the code that indicate an underlying issue in the design and implementation of a software system. Besides being great indicators of code quality and guidelines for refactoring practices, they are also important features for the task of defect prediction [9], [10], which predicts how likely a specific software component (e.g., class or method) is to be defective.

Bad code smells are detected using rule-based tools where specific software metrics are evaluated against thresholds to verify whether code smells exist in those software components. Currently, there are several tools to detect bad code smells. However, most of the tools are either written to detect code smells in Java, such as DesigniteJava [11] and Organic [12], or, if written for C++, they are either old tools that are no longer available, or they are tools that detect a very small number of code smells.

The contribution of this paper is by proposing the Code smeLls ExtActioaN tool for C++ (**CLEAN++** [1]), a novel code smell extraction tool written for software systems in C++ that extracts 35 code smells. The extracted code smells are based on the set of conceptual smells initially defined by Fowler, M. et. al [13] and Brown, W. [14], and later derived into rules [15], [16]. CLEAN++ implements the same rules as defined in both DesigniteJava [11] and Organic [12], two well-known code smells extractors for Java that have been used in the literature [9], [10], [17], [18]. CLEAN++ extends OCLint, a rule-based static code analysis tool that uses an abstract syntax tree (AST) visitor provided by the Clang API to reliably visit each node in the AST and collect the metrics necessary to detect the code smells.

To examine the validity of CLEAN++, we created test cases for each bad code smell. We ran our tool on the test cases and verified that the relevant code smells were retrieved. Moreover, we converted those test cases to Java using a C++-to-Java converter [19]. We executed CLEAN++ and both DesigniteJava and Organic on the test cases and compared the similarities between the detected code smells. Beyond retrieving the correct code smells using the test cases, we ran CLEAN++ on a set of 44 C++ Github projects and analyzed the detected code smells to validate the tool on real-world open source projects.

II. Code Smells

Bad code smells are patterns in the source code that entail underlying problems in the design and implementation of the source code. These problems are not directly linked to a defect in the software but are bad practices that increase the code's technical debt and complexity, and most likely lead to the introduction of defects. For instance, a piece of code that is duplicated across related classes does not indicate that those classes have defects, but increases the software's complexity and hinders the code's maintainability and development, thus increasing the probability of defects.

The most familiar smells in the literature were proposed by Fowler, M. et al. [13] and Brown, W. [14]. They published conceptual definitions of code smells as patterns of bad-quality software development that can be fixed through the application of refactoring techniques. However, they only described the conceptual idea of bad code smells and did not propose a means to detect them. Therefore, posterior studies proposed formal methods of detection based on the smells' description [16], [20]–[22]. These studies use code metrics extracted from the source code (e.g., number of lines of code) and build conditional rules using sets of thresholds that assert if a specific code smell is present in a particular software component (such as a class or a method). The commonly used thresholds are either predefined constants derived from experience [21], or are values based on the statistical distribution of the compared metrics in the project [16], [20].

An example of a code smell is the *God Class*. This smell occurs when a class is too large and concentrates many responsibilities on itself [14]; in a class diagram, it can be visualized as a single complex controller class surrounded by simple data classes. One simple rule to detect if a class is the *God Class* [12] is to evaluate whether the number of lines of code in the class (COLC) is higher than 500 and if its tight class cohesion (TCC), which measures the cohesion between the public methods of a class, exceeds the average TCC of all classes in the project.

In addition to these code smells, Suryanarayana et al. generalized existing code smells from the literature and proposed a catalog of design smells that detect violations on the four object-oriented programming design principles [23]:

- **Abstraction** focuses on the simplification of entities.
- **Encapsulation** focuses on the separation of concerns and information hiding.
- **Modularization** focuses on the creation of cohesive and loosely coupled.
- **Hierarchy** focuses on the creation of a hierarchical organization of abstractions.

In our tool, we implemented 35 code smells from these different sources, targeting software in C++. We include a list of the implemented code smells, including each smell's description and its respective rule together with the source code.

III. Related Tools

A number of tools have been proposed to detect code smells in C++ programs [24]. However, most of them are either old tools that are not available anymore and some of them only implement a small number of code smells (< 5). For instance, Arcan is a tool that detects code smells for C++ projects that implements five code smells [25]. To our knowledge, there is no tool specifically designed to detect many types of code smells in C++ programs.

On the other hand, several other tools were designed to detect code smells in Java programs [24]. From the same authors that proposed the catalog of design smells based on the principles of object-oriented programming, Suryanarayana et al. published a code quality assessment tool called DesigniteJava. It extracts 18 design code smells and 10 implementation smells [11]. Moreover, another tool is called Organic; it is an Eclipsed plugin that collects 11 code smells from Java projects based on [12]. They implement the rules published by Bavota, G. et al. [15].

In our tool, we implemented 35 code smells derived from the rules used both in DesigniteJava and the Organic tools for software written in C++.

IV. Tool Description

To implement CLEAN++, we extended a software analysis tool called OCLint [26], which applies static analysis to collect metrics from C++ code, to be able to extract a combination of 35 design smells. In this section, we start by describing what OCLint is and how does it work, and then we describe how we extended OCLint to implement CLEAN++.

A. OCLint background

OCLint is a tool that helps to improve the quality of C++ code by looking for potential problems. This static code analysis tool uses the source code's AST by interacting with the Clang API to detect defects that are not visible to compilers. OCLint looks for possible bugs, unused code, complicated code, code smells, and other static analysis metrics; although, it only provides a small number of code smells, which CLEAN++ solves. An advantage of this tool is that it is open-source, allowing the community to expand its capabilities by adding custom rules.

Clang is part of the LLVM project that is used to parse C-based languages source code and generate AST representation of the code [27]. The Clang's AST nodes are represented by large hierarchy classes for different types of nodes, For example, a node representing a C++ method in the source code is an object of *clang::CXXMethodDecl* class. This class is a derived class of the base class *clang::Decl* that represents a declaration in general. Another example is a node that represents an if statement. This node is an object of *clang::IfStmt* derived from the base class *clang::Stmt*.

OCLint is an extendable rule-based tool. Each rule is implemented as a subclass of *RuleBase*, and from here, there are two main categories of rule creation. There are rules created from reading the source code line by line, and there are rules that recognize patterns in the AST. CLEAN++ implements smells that are based on the rules category of the latter. Therefore, to recognize patterns in the AST, OCLint uses the *AbstractASTVisitorRule* class, which implements a visitor design pattern mechanism [28] to interact with the Clang's AST nodes node. In this pattern, the entire AST is traversed recursively from the tree's root, and each node is visited in depth-first pre-order traversal.

The rules analyze each AST representation of the source code and find nodes that match the predefined patterns. All the rule violations are collected and eventually sent to the reporter, which displays them along with the name of the affected code file.

B. CLEAN++

CLEAN++ is designed to detect bad code smells in C++ projects. Our choice of OCLint for this purpose is due to its stability as a static analysis tool and due to its extendability feature to define custom rules. We added a set of new rules designed to detect 35 code smells: (1) nine **design** smells and (2) six **implementation** smells, based on the rules implemented in DesigniteJava [11], [23]; and (3) 12 **class-level** smells, (4) and eight **method-level** smells based on the rules implemented in Organic [12], [15]. Since these code smells are detected based on rules set on extracted metrics, for each class and method we parsed the AST to measure the metrics required for each particular smell and then assessed its relevancy against specific thresholds.

Since OCLint extends the Clang visitor pattern over its API, using the visitor pattern, we were able to recursively visit only the AST nodes belonging to specific Clang classes that we wish to measure. For example, the rule that searches for the *Large Class* smell visits all the class nodes in the project. For each of them, it counts the number of lines of code in the class (CLOC), and if it exceeds some threshold, this class is reported as having a *Large Class* code smell.

To run CLEAN++ on a project, the project has to contain a compilation database defined in a *compile-command.json* file. OCLint requires the compilation database to obtain information about the source files, compilation options, and preprocessor defines, which are required to build a code base. Thus, it allows OCLint to provide accurate and consistent results, even when analyzing complex and large codebases.

Given the compilation database for the target project, to run CLEAN++, the user needs to execute the *oclint-json-compilation-database* script. It is created after installing CLEAN++ and it is located inside the build folder in *oclint-repo*. When executing it, the flags *-p <project-path>* and – *-rule=<rule-name>* should be included. The *<project-path>* specifies the project directory path where the *compile-command.json* can be found. The *rule-name* specifies the name of the smell to be extracted. When executing CLEAN++ it can be run either by extracting a single code smell or chaining the *-rule* flag to extract a combination of more than one smell. To extract all the code smells available in CLEAN++ we provide a utility script in our tool's repository.

After its execution, the output of CLEAN++ follows OCLint's report format, and it is redirected to the standard output. The report describes which smells were detected in each file of the project and their location; in addition, it summarizes the total number of files that were analyzed, the number of those files where smells were detected, and the total number of detected code smells.

V. Evaluation

In this section, we evaluate CLEAN++. We start by describing how we setup the experiments, and then we present and describe the results.

A. Experimental Setup

To evaluate CLEAN++, our goal is to measure how reliable the tool is in detecting the implemented code smells. Therefore, we wrote test cases to cover each implemented code smell. Each test case is a project that includes both files with the code smell under test and files without it. We ran CLEAN++ over each test case and verified that the tested code smell was only detected in the relevant files. For example, considering that we are testing the detection of the *Missing Default* code smell. This code smell occurs when there is a method that uses a switch statement without a default code block. Therefore, if the test case for this code smell contains two files – the first with a switch case statement without default and the second with a statement that includes it – the test measures whether the *Missing Default* code smell is detected in the first file and not in the second.

To validate the results from both the reports of CLEAN++ execution's on the test cases, we compared CLEAN++'s reports with the results of the combination of both DesigniteJava and Organic tools on the same tests. Since DesigniteJava and Organic are code smell extraction tools for projects written in Java, we applied a C++ to Java converter tool [19] to create equivalent Java versions of the test cases. After creating equivalent tests in Java, we executed both DesigniteJava [11] and Organic [12] and collected the reports of the detected smells for each file.

Moreover, we extended the evaluation of CLEAN++ to include code smell detection on real open-source projects written in C++. We randomly selected 44 projects available in Github (Table I); we compiled them, generated the compilation database, and ran CLEAN++ on the projects to generate the smell detection reports.

Project Name	Total	Abstraction	Encapsulation	Modularization	Hierarchy
openal-soft	1,245	770	444	31	0
PEGTL	1,197	1,150	46	1	0
gperftools	1,081	646	354	73	8
flatbuffers	552	467	46	39	0
duckdb	538	350	168	20	0
C-Plus-Plus	466	309	129	28	0
re2	336	245	76	15	0
FTXUI	293	225	44	23	1
coost	287	179	82	25	1
json	277	214	62	1	0
sentencepiece	256	205	45	6	0
pugixml	256	178	64	14	0
rpi-rgb-led-matrix	234	211	14	5	4
mergerfs	199	152	44	3	0
UDPspeeder	192	109	77	6	0
enkiTS	181	107	74	0	0
libco	180	109	68	3	0
libfm	179	121	46	11	1
xlearn	173	168	2	0	3
WebServer	160	146	8	6	0
design-patterns-cpp	150	128	1	0	21
zopfli	114	62	48	4	0
git-crypt	101	82	18	1	0
yoga	97	65	29	3	0
ChaiScript	92	68	24	0	0
BlingFire	56	46	9	1	0
backward-cpp	51	51	0	0	0
spdlog	50	37	7	4	2
Dobby	47	31	14	2	0
muduo	39	35	4	0	0
kfr	38	20	18	0	0
rpclib	35	31	4	0	0
handy	34	21	12	1	0
OpenCC	29	20	8	1	0
LeetCode	29	29	0	0	0
matrix	27	25	1	1	0
magic	26	26	0	0	0
PacVim	22	18	2	2	0
marl	20	12	8	0	0
hardware-effects	14	10	4	0	0
hnswlib	7	7	0	0	0
SEAL	7	7	0	0	0
2048	5	5	0	0	0
cxxopts	3	3	0	0	0

TABLE I: Number of code smells detected in each project and the number of smells divided per category.

B. Results

We executed CLEAN++ over two variants of a set of 35 test cases in C++ and executed the bad code smells detection tools for Java on the same equivalent test cases in Java. Then, we collected all the detected smells for each inspected file and compared the occurrence of code smells between the Java and C++ versions. Figure 1 shows the difference between the total number of detected code smells over each test case for the C++ version compared to the Java version. We observe that the number of code smells is mostly equivalent among the C++ and Java versions of the same test cases, except for a few specific scenarios where the Java tools detect a larger number

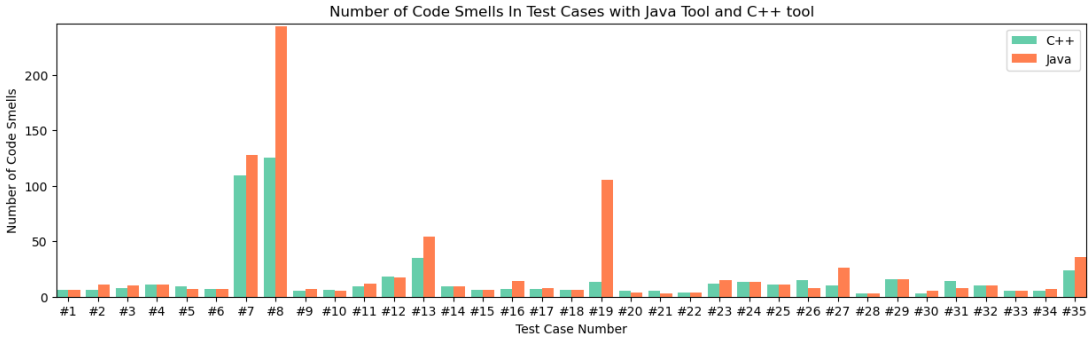

Fig. 1: Total number of code smells detected for each test case between the C++ and the Java converted variant of the test.

of smells. Moreover, Table II lists the cosine similarities from the detected smells between the tools (C++ or Java) from each specific test case. We observe that most test cases have a very high similarity, with an average of 0.895. It shows that both tools report closely the same smells in most of the test cases.

#Test	Cosine Similarity	#Test	Cosine Similarity	#Test	Cosine Similarity	#Test	Cosine Similarity
1	0.786	10	0.972	19	0.476	28	1.000
2	0.801	11	0.935	20	0.965	29	1.000
3	0.822	12	0.683	21	0.832	30	0.976
4	1.000	13	0.673	22	1.000	31	0.811
5	0.798	14	1.000	23	0.967	32	1.000
6	1.000	15	1.000	24	0.949	33	1.000
7	0.872	16	0.883	25	0.985	34	0.946
8	1.000	17	0.978	26	0.796	35	0.854
9	0.954	18	1.000	27	0.6	**Avg**	**0.895**

TABLE II: Cosine similarity between the C++ and Java versions of each test case.

In general, we observed an equivalent detection of code smells between CLEAN++ and the tools for Java code. However, we found some mismatches in some of the test cases, which should not be expected as the rules and implementations between the tools are the same. Therefore, we investigated possible issues in the experimental setup and identified the following:

- The converter used to create an equivalent Java version from the C++ tests and open-source projects may cause physical changes in the code that, despite matching the equivalence between languages, it may affect the measured metrics in a project. For example, when converting a class from a C++ file to a class in Java, depending on how the converter is implemented and also from specific rules in the language, the number of lines of code in the class (CLOC) may be different; to which we verified in some instances of the converted files. This, in turn, may influence some of the bad code smells, such as the *God Class* smell that is detected when the number of lines of code in the class exceeds 500.
- Due to differences between specific rules in C++ and Java, such as multiple inheritance, and differences between virtual methods in C++ and abstract methods in Java, there are assumptions defined in the smell detection task that may lead to inconsistencies in the detected rules, which may justify differences in the detected smells, but are not inherently wrong since they follow the principles of the smell under the definitions of the programming language.

Moreover, we ran CLEAN++ over 44 open-source projects. The results show that CLEAN++ succeeds in detecting code smells from different files of real projects. Table I shows the total number of code smells detected by CLEAN++ for every tested project and the number of smells over each category of the object-oriented design principles.

VI. Threats to Validity

In this section, we identify limitations that may threaten the validity of CLEAN++.

- For projects with large dimensions that contain multiple code smells, CLEAN++ may have performance issues and require significant time to produce a report.
- The tool was created for object-oriented code and will not work with imperative code. Consequently, it will not work if the C++ project does not contain at least one class.
- CLEAN++ does not include the external dependencies within the analysis, only the project files.
- As a requirement for a consistent analysis, the target project needs to include a compilation database.
- CLEAN++ is not robust against compilation errors. If the target project fails its execution, the tool will not conclude the smell detection.

VII. Summary

We proposed a new tool called CLEAN++ to extract a significant number of code smells for programs written in C++. CLEAN++ detects 35 code smells based on the same rules implemented in two code smell detection tools from programs in Java. This tool is an extension of OCLint, a static code analyzer that uses the Clang API to traverse the program abstract syntax tree and reason whether a code smell is detected. We examined our tool by creating test cases for each implemented smell and converting them to Java to compare with the existing Java tools. We observed mostly similar detection between the tools. In addition, we ran CLEAN++ on 44 real open-source projects, to verify that our tool could extract code smells from real projects.

ACKNOWLEDGMENTS

This research was funded by the ministry of science grant No. 3-17386.

REFERENCES

[1] T. Mashiach, B. Sotto-Mayor, M. Kalech, and G. Kaminka, "Tomma94/clean-plus-plus: Review ready version," Jan. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7574528

[2] B. A. Muse, M. M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the prevalence, impact, and evolution of sql code smells in data-intensive systems," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 327–338. [Online]. Available: https://doi-org.ezproxy.bgu.ac.il/10.1145/3379597.3387467

[3] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 189–200. [Online]. Available: https://doi.org/10.1145/2901739.2901761

[4] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-driven code smell prioritization," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 220–231. [Online]. Available: https://doi.org/10.1145/3379597.3387457

[5] T. Sharma and M. Kessentini, "Qscored: A large dataset of code smells and quality metrics," *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 590–594, 2021.

[6] D. Spadini, M. Schvarcbacher, A. Oprescu, M. Bruntink, and A. Bacchelli, "Investigating severity thresholds for test smells," in *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, S. Kim, G. Gousios, S. Nadi, and J. Hejderup, Eds. ACM, 2020, pp. 311–321. [Online]. Available: https://doi.org/10.1145/3379597.3387453

[7] A. Borrelli, V. Nardone, G. A. D. Lucca, G. Canfora, and M. D. Penta, "Detecting video game-specific bad smells in unity projects," in *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, S. Kim, G. Gousios, S. Nadi, and J. Hejderup, Eds. ACM, 2020, pp. 198–208. [Online]. Available: https://doi.org/10.1145/3379597.3387454

[8] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," *2011 IEEE 19th International Conference on Program Comprehension*, p. 81–90, 2011.

[9] B. Sotto-Mayor and M. Kalech, "Cross-project smell-based defect prediction," *Soft Computing*, vol. 25, no. 22, pp. 14 171–14 181, 2021.

[10] B. Sotto-Mayor, A. Elmishali, M. Kalech, and R. Abreu, "Exploring design smells for smell-based defect prediction," *Engineering Applications of Artificial Intelligence*, vol. 115, p. 105240, 2022.

[11] T. Sharma, " Designite - A Software Design Quality Assessment Tool," May 2016. [Online]. Available: https://doi.org/10.5281/zenodo.2566832

[12] S. L. Cedrim D, "opus-research/organic." 2018. [Online]. Available: https://github.com/opus-research/organic

[13] P. Becker, M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[14] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[15] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215001053

[16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[17] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira, "On the density and diversity of degradation symptoms in refactored classes: A multi-case study," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 346–357.

[18] T. Sharma, M. Fragkoulis, and D. Spinellis, "House of cards: Code smells in open-source c# repositories," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 424–429.

[19] "Tangible software solutions." [Online]. Available: https://www.tangiblesoftwaresolutions.com/index.html

[20] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, and R. Wettel, "iplasma: An integrated platform for quality assessment of object-oriented design.[c]," in *IEEE International Conference on Software Maintenance-industrial & Tool Volume*. DBLP, 2005.

[21] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *2008 12th European Conference on Software Maintenance and Reengineering*, 2008, pp. 329–331.

[22] P. Danphitsanuphan and T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in *2012 Spring Congress on Engineering and Technology*, 2012, pp. 1–5.

[23] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

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

[25] A. Biaggi, F. A. Fontana, and R. Roveda, "An architectural smells detection tool for c and c++ projects," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018, pp. 417–420.

[26] "Oclint." [Online]. Available: https://oclint.org/

[27] "Clang." [Online]. Available: https://clang.llvm.org/

[28] K. IGLBERGER, *C++ Software Design: Design Principles and patterns for high-quality software*. O'REILLY MEDIA, 2023.