Preliminary Experimental Evaluation of Automated Language-Independent Code Smell Detector

Yanqiao Chen
Allegheny College

Gregory M. Kapfhammer
Allegheny College

***Abstract*—A code smell is a code-design-related issue that can lead to decrease in code quality, thereby hindering developers from reading or maintaining a software program. Code smell detection tools help developers by automatically checking the existence of code smells. Code smell detection tools have the potential to be language-independent, because common code smells occur in many different programming languages. A language-independent code smell detector can set unified detection rules across programming languages, which is especially useful in multi-language software development. However, few code smell detection tools support multiple programming languages. To fill the aforementioned gap, this paper presents a tool, called `TreeNose`, for detecting code smells across programming languages. `TreeNose` can detect 5 types of code smells across multiple languages (i.e., Python, Java, and JavaScript) and its extensible design enables it to support additional languages. This paper answers three research questions to explore the quality of `TreeNose` and structural patterns of code smells in different languages. In order to answer those questions, we evaluated `TreeNose` on 15 open-source projects implemented in the three chosen programming languages, comparing its results to those arising from the use of traditional language-specific code smell detectors. One of his paper's key results is that `TreeNose` has a precision above 90% for the chosen code smells and subject programs. The experimental results also reveal that code smells are language-independent, with varying prevalence in the studied programming languages.**

I. Introduction

Code smells are a series of code-design-related defects. They decrease readability [5] and maintainability [19] [25] of software projects, which block the potential for the future maintenance [8]. Due to the negative impact of code smells on software projects, developers acknowledge the importance of detecting them. The manual detection in software projects is is an error-prone and resource-consuming process [20]. Therefore, code smell detection tools have been developed to automate this process. Different code smell detection tools rely on various detection strategies, such as abstract syntax tree (AST) analysis, Machine Learning, and Static Analysis. Popular code smell detection tools, PMD [3] and CheckStyle [4] for example, become one step of continuous integration in open-source software projects like Apache Commons Lang [1] and Jenkins [2].

One of the characteristics of modern software projects is the use of multiple programming languages [12]. The combinations of programming languages allow developers to rely on functionalities and libraries that are not available in a single programming language. However, the complexity of multi-language software projects increases the difficulty of project comprehension and maintenance [11], [14], [15]. Along with the complexity, the multi-language software projects also introduce the challenge of code smell detection. The existing code smell detection tools are designed to detect code smells in a single programming language. Therefore, developers have to configure multiple code smell detection tools in multi-language software projects [2].

While few code smell detection tools are language-independent, most code smells are. For example, the Long Method, one of the most prevalent code smells [26], can exist in tons of programming languages. Due to the language-independence of code smells, code smell detection tools also have the potential to be language-independent. Emden and Moonen, the builders of the first code smell detection tool, indicated that their detection approach in Java has the potential to be applied to other programming languages in the future [23]. A language-independent code smell detection tool can provide a unified detection experience for multi-language software projects. It can also avoid the overhead of configuring multiple code smell detection tools with various detection approaches.

This paper present a language-independent code smell detection tool, named `TreeNose`, to detect 5 types of code smells, Complex Conditional, Long Class, Long Method, Long Message Chain, Long Parameter List, across multiple programming languages. `TreeNose` implements Tree-sitter [6], a general parser generator, to parse the source code of multiple programming languages into the nodes of an AST. `TreeNose` can traverse software projects and parse the source code of multiple programming languages into AST code structure nodes. After unifying the Tree-sitter language-specific nodes, `TreeNose` queries the unified nodes to detect targeted code smells with thresholds, which are configured by developers. `TreeNose` is designed to be highly extensible, allowing developers to add programming languages without rewriting source code.

To evaluate the accuracy of `TreeNose`,we evaluated `TreeNose` on 15 open-source projects implemented in the three chosen programming languages, comparing its results to those arising from the use of traditional language-specific code smell detectors. We also conducted evaluations on the characteristics of code smells in different programming languages.

The key contributions of this paper are as follows:

1) A language-independent code smell detecter that detects

5 types of code smells across programming languages.
2) An evaluation of `TreeNose` to evaluate its accuracy in detecting code smells in multiple programming languages.
3) An experiment to reveal the prevalence and distribution of code smells in different programming languages.

II. Background and Related Work

Code smell origins from the book *Refactoring* by Martin Fowler and Kent Beck [8]. Since then, this concept has been widely adopted by the software engineering community. The community has defined more specific code smells [7] [13] [10] [24] to describe the issues in software development. Jerzyk, recently, built an online code smell catalog with 56 common code smells [9]. At the same time, community was also exploring the characteristics of code smells. Yamashita conducted several empirical studies to investigate the effects of code smells on maintainability [25] [19]. She also conducted another study to discover developers' opinion of code smells [26], where she found the top 15 most common code smells in developers' perception. Researchers, like Tufano [22] and Peters [17], conducted the studies to discover the characteristics of code smells in lifespan of software projects. Santana, Cruz, and, Figueiredo found the strong agglomerations of certain code smells highly occur in software projects [18]. Pascarella found active code reviews significantly reduce the severity of code smells [16].

Because of the negative impact of code smells on software projects, developers acknowledge the importance of detecting them. The manual code smell detection was proven to be error-prone and resource-consuming [21]. Therefore, code smell detection tools have been developed to automate this process. Different code smell detection tools rely on various detection strategies, such as abstract syntax tree (AST) analysis, Machine Learning, and Static Analysis. Emden and Moonen built the first code smell detection tool in Java [23]. Since then, many language-specific code smell detection tools have been developed, such as PMD [3] and CheckStyle [4]. Chen [7] built a code smell detection tool in Python targeting on Python specific code smells. All the tools mentioned above rely on AST to detect code smells. There are also developers who use Machine Learning. For example, Pontillo utilized Machine Learning to detect test smells, similar to code smell, to achieve better performance than heuristic-based techniques, but had a challenge of overcoming F1 score of 0.51.

III. Approach

This chapter resolves 2 questions: 1. what's the definitions of list of code smells that `TreeNose` will detects, and 2. what's the approach that `TreeNose` uses to detect code smells across multiple programming languages.

A. Definitions of the Selected Code Smells

Code smells was first defined by Folwer and Beck back to 1999 in their book *Refactoring*, where they defined 22 kinds of code smells [8]. Sine then, many researchers dedicated to

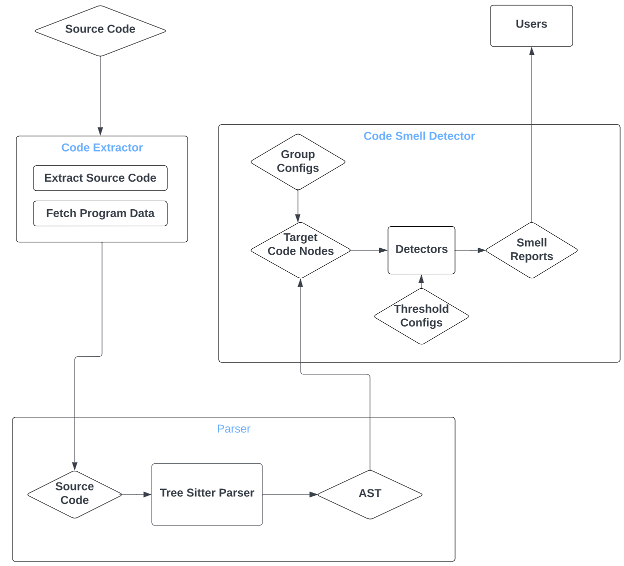

Fig. 1. TreeNose: the Code Smell Detection Architecture

define more code smells and to improve the existing ones. This paper will focus on the code smells that are top 15 most common in an empirical study conducted by Yamashita & Moonen [26]. Among them, we selected 5 code smells that occur across multiple programming languages: 1. Complex Conditional, 2. Long Class, 3. Long Method, 4. Long Parameter List, and 5. Long Message Chain. The definitions of these code smells are as follows:

Complex Conditional (CC): this smell occur when a conditional clause contains too many conditions, such as nested if-else statements, and enormous switch-case statements. This smell makes the code hard to read and maintain on logic level.

Long Class (LC): a class handles too much work. Normally it occurs with too many fields. Or a class has too many lines because of the extremely long fields.

Long Method (LM): a method grows too long. Normally it occurs with too many lines of code. The longer a method is, the harder it is to understand the procedure.

Long Message Chain (LMC): a long chain of object calls. This smell occurs when a method or an attribute calls another method or an attribute, and so on.

Long Parameter List (LPL): a method or a function has too many parameters. When a method has too many parameters, it often indicates that the method is doing too much work.

B. Detection Procedure

`TreeNose` adopts the AST-based approach to detect code smells. The Fig. 1 discloses the architecture of `TreeNose`. `TreeNose` executes 3 steps to detect target code smells. 1. Extract source code from the project, 2. Parse the source code to AST, 3. Analyze the AST to detect code smells.

TABLE I
METRIC-BASED CRITERIA FOR IDENTIFYING CODE SMELLS IN TREENOSE

Code Smell	**Criteria**	**Metrics**
Complex Conditional	NC $>$ 5	LC: number of cases
Long Class	LOC $>$ 200 or NOM $>$ 20	LOC: lines of code NOM: number of methods
Long Method	LOC $>$ 100	LOC: lines of code
Long Parameter List	NOP $>$ 5	NOP: number of parameters
Long Message Chain	LMC $>$ 3	LC: length of chains

Step 1: Extract Source Code: `TreeNose` extracts recursively extract source code matching with the target programming language from the project. During this procedure, `TreeNose` fetches all target files unless the file or the path is in the ignore list.

Step 2: Parse Source Code to AST: `TreeNose` parses the source code to AST with TreeSitter [6]. TreeSitter is a parser generator tool that generates AST in multiple programming languages. It decouples the parser from the language grammar, making it possible to parse the source code in multiple programming languages with the same parser. `TreeNose` uses the TreeSitter parser to parse the source code to AST.

Step 3: Analyze AST to Detect Code Smells: `TreeNose` analyzes the AST to detect code smells. Before the analysis, our developers categorize the similar AST nodes like method and function across programming languages into language-independent groups. This categorization enables `TreeNose` to execute the same detection process for the same language-independent group across multiple programming languages. During this analysis, `TreeNose` queries the AST and searches for associated AST components within the associated group. When locating the target component, `TreeNose` calculates the detection metrics against the detection thresholds to make a decision. Finally, `TreeNose` generates reports with the list of code smells detected in the project.

With customized detection thresholds, `TreeNose` verifies the AST parsed from source code against the detection metrics. Table I shows the detection metrics with default thresholds for each code smell.

IV. Evaluation

To evaluate the performance of `TreeNose` in different programming language systems, we implemented `TreeNose` in 9 large open-source projects with at least 10,000 lines of code written in Java, JavaScript, or Python. After building the confidence on the performance of `TreeNose`, we utilized `TreeNose` to analyze the structural patterns of the selected code smells, i.e., distribution and prevalence, in the different programming languages. To achieve so, we implemented `TreeNose` in up to 15 open-source projects, including ones written in the combination of the selected languages. We designed the following research questions:

RQ1: How does `TreeNose` perform in different languages? This RQ aims to evaluate the accuracy of `TreeNose` in detecting code smells in different programming languages. We selected 3 language-specific code smell detection tools to compare `TreeNose`'s detection accuracy with theirs.

RQ2: How do code smells distribute in various languages? By investigating the distribution of code smells in different programming languages, this RQ is designed to understand languages' tendencies to contain specific code smells.

RQ3: How often do code smells occur in various languages? This RQ intends to evaluate the prevalence of the selected code smells in different programming languages.

A. Subjects

To answer the RQs, the subjects must stand for the real-world software projects in their respective programming languages. They should be actively used and contain a large amount of code. To achieve this goal, the subjects in the experiments are 1. have commits on main branch in the last year, 2. have at least 10 thousands lines of code (KLOC), and 3. have more than 1,000 stars on GitHub. . Within those constraints, we selected 3 subjects for each programming language, Java, JavaScript, and Python, respectively. Table II shows the characteristics of the subjects. All the subjects have been proven to have commits on main branch in the last year. Though TreeNose supports both Python 2 and 3, Pysmell only supports Python 2 and therefore all the selected subjects in Python are written in Python 2. And as for the contents of those projects, we decided to include both source code and test code in the analysis, it's because the test code is also important for developer to comprehend and maintain in software development.

TABLE II
CHARACTERISTICS OF SUBJECT SYSTEMS

Project	**Organization**	**Language**	**Version**	**Stars**	**KLOC**
Eclipse Collections	Eclipse	Java	11.1.0	2.4k	596
Gson	Google	Java	2.10	23.1k	59
Maven	Apache	Java	3.9.3	4.2k	129
Django	Django	Python	1.8.2	77.7k	282
Numpy	Numpy	Python	1.9.2	27k	167
NLTK	NLTK	Python	3.0.2	13.2k	97
Moment.js	Moment.js	JavaScript	2.30.0	47.9k	331
Lodash	Lodash Utilities	JavaScript	4.0.0	59.2k	109
Chance.js	Chance	JavaScript	1.1.0	6.4k	20

B. Methodology

To answer the RQ1, we compared `TreeNose` with 3 language-specific code smell detection tools, namely Pysmell (Python), Jscent (JavaScript), and DesigniteJava (Java). All 3 code smell detectors are language-specific AST-based code smell detection tools that are open-source, where Pysmell was proven to achieve 97.7% precision in a study. In the experiment, we compared TreeNose with the combination of the other tools. The combination of the other tools mimic the scenario where developers utilize multiple code smell detectors to detect language-specific code smells in multi-language systems. Those detectors support most code smells detected by `TreeNose`, allowing us to implement them in the same sets of projects with `TreeNose` to compare their performances. To quantify the performance of `TreeNose` and the other tools, we adopted precision and recall, which are defined as follows:

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{1}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{2}$$

To represent both precision and recall in a single metric, we calculated the F 1 score, the harmonic mean of precision and recall, which is defined as follows:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3}$$

To our best knowledge, there is no annotated dataset for code smells in our selected programming languages, we hence had to design an annotation study by ourselves. First, we implemented both `TreeNose` and the other tools with their default thresholds to analyze the same sets of projects. We, furthermore, conducted such a study with 2 internal developers to manually verify code smells. Specifically, we built two spreadsheets for the study. One for the code smell reports generated by `TreeNose`, and another for the code smell reports generated by the other tools. Within the code smell reports generated by the other tools, we randomly selected 5 samples out of each code smell in each programming language system, and then put those samples in the spreadsheet with their related source code and a column for the detection decision of `TreeNose`. After that, we asked the developers to independently inspect the samples and determine whether the samples are code smells or not. Then, we built a consensus from the inspection results, after the discussion between developers. By this approach, we can compare the decisions between the `TreeNose` and the developers, where both can potentially agree or disagree with the other detectors. We defined the True Positive of `TreeNose` as the number of code smells that are detected by both `TreeNose` and the developers among the samples selected from the other tools, the False Positive as the number of code smells that are detected by `TreeNose` but not by the developers, and the False Negative as the number of code smells that are detected by the developers but not by `TreeNose`. With those metrics, we calculated the precision, recall, and F1 score of `TreeNose`. Then, we conducted the same study with the samples from `TreeNose` to calculate the F1 score of the other tools. Finally, we utilized the F1 scores of `TreeNose` and the other tools to compare their performances. It's noteworthy that not all code smells detected by `TreeNose` are detected by the other tools. For those code smells, we skipped them in the calculation of the F1 score of `TreeNose` because there is no detection results from the other tools, and tagged them as not detected in the calculation of the F1 score of the other tools. We made this decision because the other tools silently failed to detect those code smells.

To answer the RQ2 and RQ3, we implemented `TreeNose` in up to 20 open-source projects, including the subjects in RQ1. We analyzed software projects written in 3 selected programming languages, and ones in the combination of the selected programming languages. We implemented `TreeNose` in the projects and analyzed the occurrence of each code smell in the projects. Analyzing the code smell reports of `TreeNose` helps to understand the distribution and prevalence of the selected code smells in different programming languages, and furthermore the target languages' tendencies to contain specific code smells.

To minimize the bias introduced by the different sizes of the projects in the RQ2, we calculate the number of each code smell in individual projects (s_k) and divide it by the total number of code smells in that project (S) to calculate the project-level percentage of the code smell. Then we the sum of the project-level percentages in the projects in the language, and divide it by the number of the projects in the language (NOP) to calculate the percentage of the code smell in the language (P_k).

In the experiment of the RQ2, we calculate the percentage of each code smell out of all the selected code smells as follows:

$$P_s = \frac{\sum_1^k \frac{s_k}{S_k}}{NOP}\% \tag{4}$$

As for the RQ3, we would like to quantify the prevalence of the code smells as the occurrence of the code smells in each 1000 lines of code on average (C_s). to calculate it, we divide the number of each code smell in individual projects (s_k) by the total number of lines in the project (NOL_k) to calculate the project-level prevalence of the code smell, and then take the average of the occurrences in each project. Finally by taking the average of the occurrences in the projects in the language, we calculate the prevalence of the code smell in the language as follows:

$$C_s = \frac{\sum_1^k \frac{s_k}{NOL_k}}{NOP} \tag{5}$$

With the prevalence score C_s, we want to quantify how much one language outperforms the others. Therefore, we utilized the discrepancy formula to calculate the occurrence discrepancy score. we compared the occurrence of one language (C_s) against the same code smell in the combination of the other programming languages (C_O). we calculate the occurrence discrepancy score (D) of each code smell in the projects as follows:

$$D = \frac{C_O - C_s}{C_O} \tag{6}$$

In this equation, a positive score indicates that the code smell occurs less frequently in the language than the other languages, while a negative score indicates that the code smell occurs more frequently in the language than the other languages. The discrepancy score has a range of negative infinity to 1, where 1 means the code smell doesn't occur in the language at all, and 0 means the code smell occurs in the language as frequently as the other languages.

C. Threats to Validity

The definition of code smell, code that is hard to comprehend or maintain, subjective. Therefore, one of concerns in this research is on the definitions of the selected code smells. We noted that different code smell detection tools utilize different metrics and thresholds to detect the same code smell. Therefore, the metrics of `TreeNose` may not be consistent with the other tools. As a result, the selection of metrics and thresholds in `TreeNose` may not be optimal for detecting selected code smells. To mitigate the threats, we conducted an annotation study with 2 developers to evaluate the performance of `TreeNose` in detecting code smells.

Another treat is related to the annotation study. To evaluate the performance of `TreeNose`, we conducted an annotation study with 2 developers. Due to the limited number of developers in the study, the results may not be generalizable to all developers. We also noted that the amount of subjects and samples are limited in the study. Though we ensured the subjects are actively used and well-maintained, the results may still not be generalizable to all software projects. To mitigate the threats, we plan to conduct more studies with more developers and subjects in the future.

The third threat is related to the limitation of the selected code smell detectors. They didn't detect all the selected code smells, and the results may be biased by the missing code smells. To mitigate the threats, we plan to implement more code smell detectors in the future to compare the performance of `TreeNose` with more detectors.

The last threat is due to our decision of including the test code in the analysis. Test code is important for developers to comprehend and maintain, but it may have different conventions and styles from the source code. Therefore, the same thresholds and metrics of TreeNose may not be optimal for the test code. To mitigate the threats, we plan to conduct more studies with separate analysis of the source code and test code in the future.

V. RESULTS

In this section, we present the results of our evaluation.

Table III shows the confusion matrixes of `TreeNose` and the other tools in the annotation study. TN represents `TreeNose`, and OD represents the other tools. On the average, `TreeNose` significantly outperformed the other tools in terms of precision, recall, and F1-score. With the 0.94 F1 score, we conclude that `TreeNose` achieved a high level of accuracy in detecting the code smells. When there is no value in the table, it means it's not possible to calculate the recall. For example, `TreeNose` flagged all the positive when we selected samples from the `TreeNose` detection. Therefore, it's not possible to calculate the recall or the F1 score of `TreeNose` in this environment. The same applies to the OD-selected samples. Another interesting observation is that the OD group achieves 1.0 precision in the `TreeNose`-selected samples, but the recall is down to 0.32, which results in a low F1 score of 0.48. This indicates that the OD group can hardly detect all the code smells in the `TreeNose`-selected samples.

Table IV discloses the percentage of the code smells in each programming language system. Looking at the table, we can see Complex Conditional (CC) counts for 42% of code smells on average, indicating that it is the most common code smell across programming languages. the percentage of Long Class (LC) and Long Method (LM) vary significantly in the Java and JavaScript systems. In the Java system, LC occurs 10 times more than LM, while in the JavaScript system it's the opposite, LM occurs 30 times more than LC. This indicates that the programming languages have huge different tendencies in terms of code smells.

Table V shows the occurrence discrepancy scores of the code smells in the systems. Since no programming language contains positive values in every code smell, we conclude no programming language outperforms the others in every code smell. By looking at each code smell, we also see programming languages have strong tendencies in some code smells with at least 1 time worse performance than others. For example, the Java system has a strong tendency in the Long Class (LC) with 3 times worse performance than others, while the JavaScript system has a strong tendency in the Long Method (LM) code smell. Python has a strong tendency in the Long Parameter List (LPL). Multi-Language system, on the other hand, has weaker tendencies in code smells compared to the other systems with zero negative values less than - 1.0.

VI. RELATED WORK

This section should cite any work related to your paper, but which is not integral to the basis of your approach.

It is a useful catch-all for anything that did not appear in background, and helps satisfy the referees that you know your field by demonstrating a knowledge of other work that is going on in the area.

While discussing related work, it is important to keep making it clear to the reader the ways in which each paper is different from your own, and/or addresses a different problem. In general, it is a mistake to be overly critical of others' work, unless discussing the limitations of that work helps to draw out key differences between their approach and yours.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented `TreeNose`, a language-independent code smell detection tool, to detect code smells. `TreeNose` uses TreeSitter AST to detect code smells without the need for language-specific rules. `TreeNose` can detect 5 code smells: Complex Conditional (CC), Long Class (LC), Long Method (LM), Long Method Chain (LMC), and Long Parameter List (LPL). We evaluated `TreeNose` with an annotation study in three programming language system, i.e., Java, JavaScript, and Python, and compared it with other language-specific tools in a set of high-quality open-source projects in target programming languages. As a result, `TreeNose` achieved a high level of accuracy with 0.94 F1 score in

TABLE III
CONFUSION MATRIX FOR TREENOSE AND OD FROM ANNOTATION STUDY

	TN Precision	OD Precision	TN Recall	OD Recall	TN F1	OD F1
TreeNose-selected Samples	0.94	1	-	0.32	-	0.48
OD-selected Samples	0.89	0.65	0.92	-	0.94	-
Average	0.92	0.83	0.92	0.32	0.94	0.48

TABLE IV
PERCENTAGE OF CODE SMELLS IN PROGRAMMING LANGUAGE SYSTEMS

	CC	LC	LM	LMC	LPL
Java	31.91%	30%	3.12%	21.4%	13.56%
JavaScript	37.82%	1.1%	38.39%	15.72%	6.97%
Python	49.42%	14.56%	8.84%	2.41%	24.77%
Multi-Language	50.61%	15.36%	15.53%	3.2%	15.31%
Average	42.94%	15.26%	16.47%	10.69%	15.15%

TABLE V
DISCREPANCY SCORES OF CODE SMELLS IN PROGRAMMING LANGUAGE SYSTEMS

	CC	LC	LM	LMC	LPL
Java	↑ 0.35	↓ 1.42	↑ 0.85	↓ 1.09	↑ 0.29
JavaScript	↑ 0.35	↑ 0.96	↓ 3.00	↓ 1.61	↑ 0.77
Python	↓ 0.47	↓ 0.28	↑ 0.50	↑ 0.81	↓ 1.62
Multi-Language	↓ 0.38	↑ 0.05	↑ 0.06	↑ 0.79	↓ 0.12

detecting selected code smells. As a comparison, the combination of other tools achieved 0.48 F1 score. We concluded `TreeNose` is an effective tool for detecting code smells across programming language systems. We also analyzed the distribution and prevalence of code smells in the programming language systems with `TreeNose`. As results of the analysis, we found that 1. Complex Conditional (CC) is the most common code smell across programming languages with 42% average proportion. 2. Programming languages have strong tendencies in some code smells with at least 1 time worse performance than others. 3. systems written in multi-language have weaker tendencies in code smells compared to the single language systems with zero negative values less than - 1.0.

In the future, we plan to extend `TreeNose` to detect more code smells and evaluate it with more programming language systems. As for annotation, we plan to conduct studies with more participants and subjects to increase the reliability of the results. We also wish to implement `TreeNose` in a real-world software development environment to see how it can help developers maintain the quality of their codebase.

REFERENCES

[1] Apache commons lang, 2009.
[2] Jenkins, 2010.
[3] Pmd, 2012.
[4] Checkstyle, 2013.
[5] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European CSMR*, 2011.
[6] Max Brunsfeld. tree-sitter, 2013.
[7] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. Detecting code smells in python programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 18–23, 2016.
[8] Martin Fowler and Kent Beck. *Bad Smells in Code*. 1999.
[9] Marcel Jerzyk and Lech Madeyski. *Code Smells: A Comprehensive Online Catalog and Taxonomy*, pages 543–576. Springer Nature Switzerland, Cham, 2023.
[10] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf, 1st edition, 2010.
[11] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd SANER*, 2016.
[12] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *Proc. Fifth Working Conf. Reverse Eng. (Cat. No. 98TB100261)*, 1998.
[13] Robert Cecil Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.
[14] Philip Mayer and Andreas Schroeder. Cross-language code analysis and refactoring. In *Proc. 2012 IEEE 12th Int. Working Conf. Source Code Anal. Manip.*, SCAM '12, 2012.
[15] Zaigham Mushtaq and Ghulam Rasool. Multilingual source code analysis: State of the art and challenges. In *2015 ICOSST*, 2015.
[16] Luca Pascarella, Davide Spadini, Fabio Palomba, and Alberto Bacchelli. On the effect of code review on code smells. *ArXiv*, abs/1912.10098, 2019.
[17] Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 411–416, 2012.
[18] Amanda Santana, Daniel Cruz, and Eduardo Figueiredo. An exploratory study on the identification and evaluation of bad smell agglomerations. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1289–1297, New York, NY, USA, 2021. Association for Computing Machinery.
[19] Dag I.K. Sjøberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. (8), 2013.
[20] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999.
[21] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, page 47–56, New York, NY, USA, 1999. Association for Computing Machinery.
[22] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414, 2015.
[23] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *9th Working Conference on Reverse Engineering, 2002. Proceedings.*, 2002.
[24] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., USA, 1 edition, 2003.
[25] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE ICSM*, 2012.
[26] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th WCRE*, 2013.