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 [12] [15] of software projects, which block the potential for the future maintenance [7]. 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 [13]. 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 [9]. 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 [8], [10], [11]. 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 [16], 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 [14]. 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

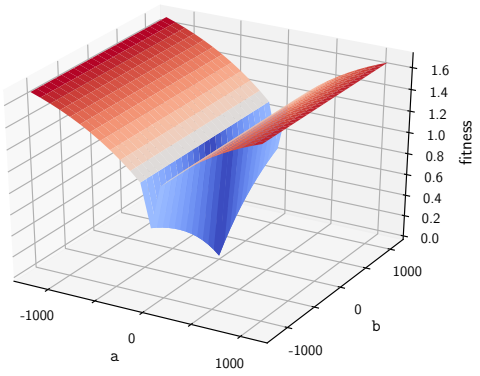

Fig. 1. Add your caption here. Captions for figures go *below* the figure.

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

The background section introduces past work (by you and others) that the reader needs to know to understand the rest of your paper. It is different from the "Related Work" section, Section VI, in which you should cite work related to your paper, but which is not integral to the basis of your approach.

Think of it as content behind the "`import`" statements of a program. What is the minimal information that the reader needs to know to understand your technique? (What would correspond to the libraries that you would "`import`" if it were a computer program?

Everything else can go in "Related Work". Don't put everything in this section, "Background" because it will make your work sound more derivative and will delay the reader before getting to the part you want to grandstand, which is your method, in the next secion, called "Approach".

III. Approach

This chapter resolves 2 questions: 1. The list of code smells that TreeNose will detects, and 2. The approach that TreeNose will use to detect code smells across multiple programming languages.

A. Definitions of 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 [7]. Sine then, many researchers dedicated to 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 [15]. 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 adapts the AST-based approach to detect code smells. 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. 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.

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 ASTs in multiple programming languages. It decouple 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 and then generalize the language-specific AST into a common AST.

Step 3: Analyze AST to Detect Code Smells: TreeNose analyzes the AST to detect code smells. During this procedure, TreeNose traverses the AST and checks the detection metrics against the detection thresholds. Finally, TreeNose generates reports with the list of code smells detected in the project.

IV. Evaluation

To evaluate the performance of TreeNose in different programming language environments, we implemented TreeNose in 9 large open-source projects with at least 10,000 lines of code written in Java, JavaScript, or Python. After ensuring the

TABLE I
METRIC-BASED CRITERIA FOR 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

performance of TreeNose, we utilized TreeNose to analyze the structural patterns of the selected code smells like distribution in the different programming languages. Therefore, 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 well-maintained, allowing the experiments to filter out the noise in softwares and focus on the code smells on programming language level. To achieve this goal, the subjects in the experiments are 1. have commits on main branch in the last year, 2. have at least 10,000 lines of code, and 3. have more than 1,000 stars on GitHub.

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 detecting code smells in Python. 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 compare it with the other tools, we adapted precision and recall, which are defined as follows:

$$Precision = \frac{TP}{TP + FP} Recall = \frac{TP}{TP + FN}$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \quad (1)$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \quad (2)$$

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:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (3)$$

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 calculate the F1 score of TreeNose. 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. With 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. With the independent inspection results, we built a consensus after the developers discussed the samples they disagreed on. Here, 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 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 compared the F1 scores of TreeNose and the other tools to evaluate 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 reports in each code smell in the projects. Analyzing the code smell reports of TreeNose in the projects helps to

understand the distribution and prevalence of the selected code smells in different programming languages, and furthermore the tendencies of code smells on each programming language. To minimize the bias introduced by the different sizes of the projects in the RQ2 and the RQ3, we calculate the metrics of each code smell in individual projects and take the average of the metrics in those projects to calculate the language-level metrics.

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

$$Percentagethecodesmellinthelanguage = \frac{\sum \frac{Numberoftheoccurrenceofthecodesmell}{Totalnumberofthecodesmells}}{Numberoftheprojectsinthelanguage} \quad (4)$$

As for the RQ3, we count the occurrence of each code smell per 1,000 lines of code in the projects, 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:

$$Prevalenceofthecodesmellinthelanguage = \frac{\sum \frac{Numberoftheoccurrenceofthecodesmell}{Totalnumberofthelinesofcode}}{Numberoftheprojectsinthelanguage} \quad (5)$$

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 may 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. There may be developers not agree with the results of TreeNose, which leads to our second threats to validity. 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.

V. Results

In this section, you should relate your results to each of your research questions, one by one.

Any more anecdotal observations, or anything you observed in the course of answering your research questions that you did not deliberately set out to investigate should be included in a subsection at the end of this section, called “Discussion”.

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

This section should begin by reminded the reader about your approach and its motivation, and that no other previous research addresses the same problem.

You may want to include some more summary statistics from the results, demonstrating its worth.

You should then conclude the paper with some ideas for future work.

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] Martin Fowler and Kent Beck. *Bad Smells in Code*. 1999.

[8] 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.

[9] 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.

[10] 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.

[11] Zaigham Mushtaq and Ghulam Rasool. Multilingual source code analysis: State of the art and challenges. In *2015 ICOSST*, 2015.

[12] 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.

[13] 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.

[14] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *9th Working Conference on Reverse Engineering, 2002. Proceedings.*, 2002.

[15] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE ICSM*, 2012.

[16] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th WCRE*, 2013.