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

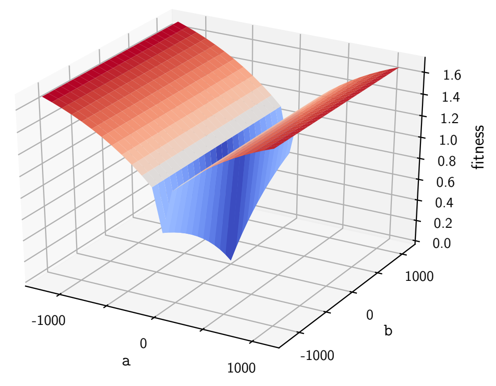

Fig. 1. Add your caption here. Captions for figures go *below* the figure.

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 section should detail your approach as clearly as possible, at a high enough level that the reader can understand. This section may include a motivating example, explaining why your approach is needed.

You may want to include some algorithms and figures to help illustrate the points you are making, to introduce the motivating example, or give an overview of the architecture of your tool.

The information in `figures/plain-figure.tex` describes how to format a figure, included here as Figure 1.

TABLE I
Add your caption here. Captions for tables go *above* the table.

Algorithm	**Best Fitness**
DOG	78
CAT	60
Mean	69

IV. Evaluation

Evaluation sections should include a list of research questions near the beginning.

A. Subjects

You should describe details of the subjects that you used in your study, summarised in a table, such as the example. The information in the file "`figures/plain-figure.tex`" describes how to format a figure, included here as Table I.

B. Methodology

This section should then include a description of the methodology used to answer them, in a dedicated subsection, like this.

You should also include a description of the tooling to implement your technique, and run the experiments.

C. Threats to Validity

The section should end with a discussion of the threats to validity of the experimental design.

Some authors choose to place this after the results. However if written *before* the results, it comes across more clearly that you have carefully thought everything through.

An example of a threat might be the type of statistical tests you used. Here, you can explain why they were appropriate. For example, you may have used non-parametric tests, such as the Mann-Whitney U-Test, because you could not assume the distribution of your results was normal.

Another threat might be the types of subjects you used, whether the code you wrote to perform was reliable (you should write tests to mitigate this threat), and so on.

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.