Allegheny College
Department of Computer Science

Senior Thesis

TreeNose: A Language-Independent Code Smell Detector

by

Yanqiao Chen

ALLEGHENY COLLEGE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Project Supervisor: **Gregory M. Kapfhammer**
Co-Supervisor: **Janyl Jumadinova**

22 Mar 2024

Abstract

Code smells are a series of code-design-related issues which can lead to the dropdown of code quality and hinder developers from reading or maintaining the software programs. Code smell detection tools in this case have been developed to automatically check the existence of code smells based a set of metrics and thresholds. Passing a range of code smell detections has been considered a necessary step to ensure the high quality and maintainability of code in modern software development. We believe code smells occur on the logic level regardless of syntaxes of languages, therefore code smell detection tools have the potential to be language-independent too. However, util now few of code smell detection tools in market can support multiple programming languages. TreeNose, therefore, was built to explore the unknown and address the lack of language-independent code smell detection tools. TreeNose is able to detect 5 kinds of code smells across multiple OOP languages: Python, Java, and JavaScript. Furthermore, we designed three research questions in our research: 1. How is TreeNose's performance in different languages? 2. Do code smells happen in software projects regardless of the selection of language? 3. Do some code smells occur more often than others in some languages? In order to answer those questions, we implemented TreeNose on a series of open source projects written in different programming languages and collected data from their TreeNose reports. As the results of the experiments, we draw the following conclusions: 1. TreeNose proves to have high precision on code smell detection regardless of the selection of programming language 2. Programming languages have tendency on different code smells 3. No programming language is significantly more smelly-prone than others.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Background & Motivation

Throughout the development of software programs, programmers put efforts to build modules and ensure the programs function as expected. From the first line of code programmers wrote in the life, they've already had expectations on the software programs: what the program should do, what it takes as inputs, and what it returns as outputs. However, if one only focuses on the functionality and neglects the code design, technical debt, a metaphor from Cunningham [5] indicating "design debt programmers pay back promptly in the future to ship the program swiftly", will emerge and expand to be a large problem.

Programs with bad design are in a bad shape. While they function and output as expected, yet they are't friendly when it comes to reading, changing and maintaining in the future. Some people may state code smells aren't real problems since code design doesn't matter as long as the program functions wonderfully. For people with such an opinion, please have a look on the following two Python functions.

```python
def function1(r):
    var1 =  3.14 * r * r
    return var1

def cal_circle_area(radius):
    area = 3.14 * radius * radius
    return area
```

Functions above programmatically work the same to calculate the area of a circle with its radius. However, it's obvious to observe that the second program is more well-structured to understand its functionality. We can simply identify the high-level functionality of the method and the purpose of the variable. In fact, the first program contains two kind of *code smell*, a concept used to identify the potential issues in code design and code quality. The code smells are Uncommunicative Name and Meaningless Name.

Code smells is an metaphor indicating a series of potential code design defects. They reduce the readability, maintainability and changeability of software projects, which block the potential for future maintenance. The term was first defined by Fowler and Beck back to 1999 to plot the situations where code needs to be refactored: When the code stinks, it's time to change[10]. During the lifespan of a software development, instances of code smells often remain unresolved for up to 50% of the lifespan and most code smells are introduced at creation [18] [16] [25].

If a large size of code smells remain in a system, we would safely assume this system is hard to change. Those problems mainly happen when software developers make bad decisions on the structures of code either because of recklessness or lack of experience. One of the most famous example is Facebook at its early age. With its motto "Move fast and Break things" coined by Mark Zuckerberg, Facebook deployed its feature amazingly fast making it vigorously growing as a start-up. Nevertheless, this culture also made Facebook's system imprudent with bad code quality and code design. Back to 2015 Facebook engineer Simon Whitaker gave a talk at iOS Dev UK 2015 and mentioned that iOS can't handle Facebook Scale, and in the same year, some people found out there are more than 18,000 classes in Facebook's application [19]! The bad code quality makes Facebook suffering from refactoring (the process of changing internal structure of a software without changing its observable behaviors [10]) and rewriting code. Eventually, Zuckerberg checked their well-known internal motto to "Move fast with stable infrastructure." Facebook suffered from bad code design and eventually paid the exponential technical debt. To raise the attention on the essence of good code design, Martin Fowler wrote: design activities pay off and positively and consistently impact the robust evolution of software systems. To visualize his hypothesis, he plotted the following graph

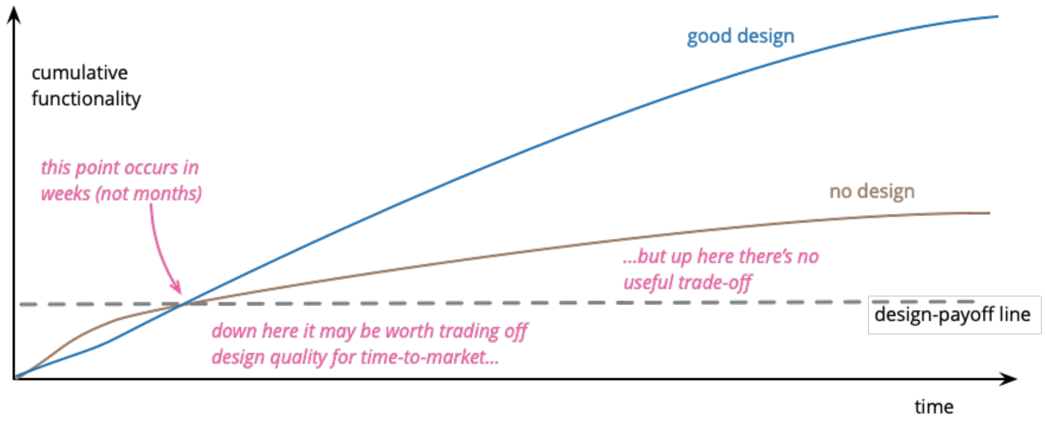


Figure 1: Martin Fowler Design Stamina Hypothesis Pseudo-Graphb

In his design stamina hypothesis [9], Martin Fowler admitted that writing code with code design in mind takes more effort and more time. However, the benefit of good code design comes fast and makes a huge difference consistently in the rest of the lifespan.

1.2 Code Smell Detection Tools

Recognizing the toxicity of code smells, both research and software development communities acknowledge the significant impact of detecting and resolving these issues throughout the lifespan of software development. Throughout the development of this field in the last two decades, people have been devoting to minimize the effects of code smells. And during this process, automated code smell detection plays a substantial role.

Code smell detection tools aim to automate the process of detecting, considering the fact that manual detection in large software system is an error-prone and resource-consuming activity [24]. Automated code smell detection has advantages on efficiency, effectiveness, and repeatability. Therefore, automated code smell detection tools have been deeply researched and developed to detect a series of code smells since the debut of the first code smell detection tool designed by van Emden back to 2002 [7]. Nowadays, passing a range of code smell detection tools has been considered a significant step to ensure the high quality and maintainability of code in modern software development.

There are different techniques to detect code smells: text-based [17], machine-learning-based [20], refactoring-based[8], and AST-based [4] [21] [22] [14]. Among them, AST-based code smell detection technique is the most common.

AST-based code smell detection tools automatically scan through the Abstract Syntax Tree(AST), a data structure to represent the low-level grammar representation of code snippets. AST filters out the information and syntaxes that are useless to computers, like comments and indentations, and preserves the abstract logic and syntactic grammars, in order to make the code more general and ready to be compiled to binary digits. The following is a simple example in Python.

Variable Assignment in Python:

```
a = 1
```

AST Parsed from the Code Above:

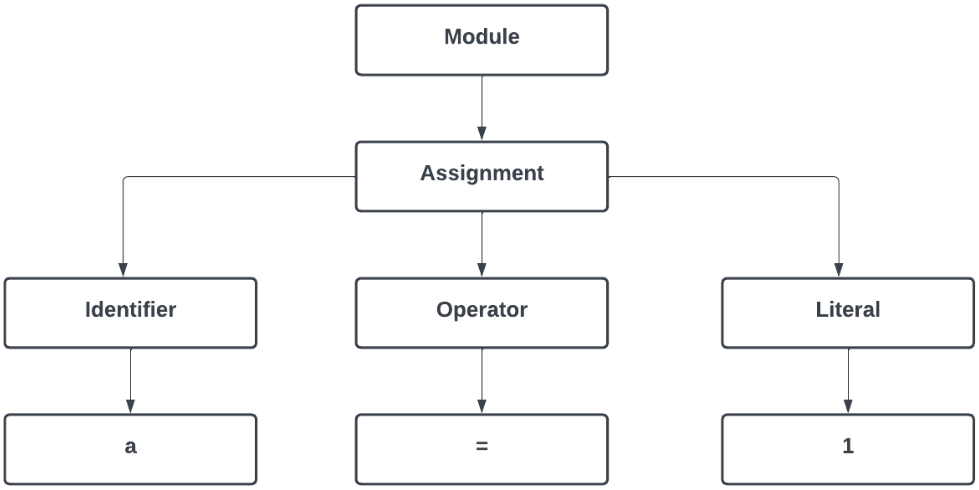


Figure 2: Python AST representation

Shown in the Figure 2, the code tokens are granted with roles: `a` is identified as an identifier and `=` is an operator. Like so, AST represents a module into token pieces, grant roles to them, and assign relationships among them.

After an AST-based detector parses source code into AST, it will search for the unreasonable snippets of syntactic structures based on a set of predefined rules. A normal AST-based code smell detection tool has the following steps:

1. **Code Extractor**: extracts target source code from a file or a directory
2. **Parser**: parses source code into low-level Abstract Syntax Tree
3. **Code Smell Detector**: detects the unreasonable syntactic grammars in the AST
4. **Reporter**: reports the code smells in form of texts, files or prompts

1.3 Goals of this Project

As mentioned in Section 1.1, code smells occur because of bad decisions of software developers on the structures of code. With the scope of code smells, most of those bad decisions aren't related to the selection of languages. For example, Complex Nested Iteration Loop is considered as one of the most critical code design issues since it's hard to understand the purpose of each layer of iteration loop. We believe this defect could happen in a large amount of programming languages, like Python, C, Rust, and Java, despite syntaxes of the iteration loops. The code snippets below show a series of Complex Iteration in three different programming languages (Python, Java, and Rust). Those languages have different syntaxes on the FOR statement, but deep down they actually function and output in the same approach.

Code Snippets in Python:

```
for i in range(5):
    for j in range(10):
        for k in range(15):
            print(i * j * k)
```

Code Snippets in Java:

```
for (int i = 0; i < 5; i++) {
   for (int j = 0; j < 10; j++) {
       for (int k = 0; k < 15; k++) {
           System.out.println(i * j * k);
       }
   }
}
```

Code Snippets in Rust:

```
for i in 0..5 {
    for j in 0..10 {
        for k in 0..15 {
            println!("{}", i * j * k);
            }
        }
    }
```

All the code snippets above will return the same results, printing 750 numbers on terminals. Same as the functionality, all the code above contain the same kind of code smell: Complex Iteration Loop (CIL), a overcomplicated iteration statements making its purpose unclear. They contain the CIL regardless their syntaxes. The syntax of FOR loop, the usage of bracket or indentation, the syntax of print statement, none of them exempts the code above from suffering from CIL. Actually, most common code smells, just like CIL, are language-independent. In most objective-oriented languages, common code smells like Duplicated Code and Long Method occur regardless of the selection of programming language. We think the main reason that code smells are language-independent is because code smells happen on the logic level other than code implementation level. Consequently, code smells occur on the control flows, while their instances cast into syntaxes of the selected programming language.

While code smells are language-independent, few code smell detection tools are. Back to 2002, when Emden and Moonen built the first code smell detection tool in the world to serve Java softwares, they said that their approach has the potential to be extended to other object-oriented languages [7]. However,to our best knowledge, no code smell detection tool in market is able to detect code smells across multiple languages til now. PMD [6], the closest tool to achieve so, documented that they support multiple languages but mainly focuses on Java. Their detections in other languages are limited into a small range of code smells. To conclude, we safely assume PMD implement detectors separately for each language because the differences between languages are bothering in its case. This characteristic of those tools, being language-exclusive, limits them to only serving single specific programming language, making them highly rely on the syntaxes of that language.

Inspired by the versatility of code smells and the lack of language-independent code smell detection tools in the market, our team decided to build TreeNose, a language-independent code smell detection tool, and conduct a research with it.

There are the three research questions this research aims to answer:

1. Do code smells happen in software projects regardless of the selection of language?
2. Do some code smells occur more often than others in some languages?
3. How is TreeNose's performance in different language?

TreeNose is like a ruler used to measure lengths of different items. It provides universal detection techniques and threshold to evaluate code quality across different languages. That's why we built a language-independent code smell detection tool on the AST layer of code structure. It currently supports three programming languages: Java, JavaScript, and Python. And It takes source code as input, and then checks and reports the presence of five kinds of code smells: **Complex Conditional**, **Long Class**, **Long Method**, **Long Message Chain**, and **Long Parameter List**.

Nevertheless, it's worthy to mention that not all the programming languages are built the same and used to serve the same purpose. Two tree structures parsed from HTML and Java are not likely to have a lot in common. For this reason, this research only focuses on several general purpose OO programming languages: **Python**, **Java**, and **JavaScript**. Moreover, as mentioned above, some code smells defects are language-exclusive. For instance, the semi-colon missing problem in JavaScript and Java are not applicable to Python. Those language-exclusive problems are either filtered out during the parsing, or not included in the scope this research.

To investigate those questions, we implemented TreeNose in open-source softwares in different languages and compare its reports with ones generated from other code smell detection tools.

The remainder of this thesis is as follows:

1. **Related Work**: researches and work done by others in this field
2. **Method of Approach**: The method of detecting code smells language-independently

3. **Experiment**: A set of researches on the behaviors of TreeNose and code smells
4. **Conclusions and Future Work**: Summarization of this paper and potential future work

1.4 Ethnical Consideration

Considering the experiment utilizes the online open-source software projects as subjects, the quality of those open-source projects vary, which can lead to a high standard deviation on code design quality statistic analysis. Therefore, the selection of samples directly affects the results. Furthermore, the design of the language-independent code smells detection tool may also directly impact the accuracy of the results, especially on overwhelming gigantic software projects. Both false positive and false negative may occurs because of the limitations of this research. Because of those limitations are non-avoidable, the final results and the answers to the research questions can't conclude objective facts. For instance, one of the research questions, *are some programming languages more error-prone*, may come to the conclusion that certain language is more error-prone than language another based on our results. But this conclusion may have potential social harm if a software engineering practitioner reads this paper and affirms a language is inferior to another.

2. Related Work

2.1 Code Smells

Code smells, a famous metaphor in the field of code design, refers to a series a degraded code design problems that hurdle the readability, comprehensibility, changeability, and maintainability. It was firstly defined by Beck and Fowler back to 1999 to identify the situation where the code is needed to be changed. Beck and Fowler defined 22 kinds of code smells defects including duplicated code, Long Method, and large class [10].

Though code smell defects are harmful in scope of code design, they are not technically bugs. Code smells don't block the software programming from functioning in the current stage. Instead, they indicate the potential hinders and high costs of maintenance. One way to think about it is that code smells defects make the software projects fragile and less trustable.

Throughout the last two decades code smells is gradually becoming a well-known metaphor for the aspect of code design problems that are potentially harmful for software projects when it comes to maintenance, the span of the term, code smells, is gradually increasing along with the size of researches in this community. New code design defects like base class depends on subclass [15] are introduced by researchers and added into the scope of code smells. A recent catalog built by Jerzyk contains 56 kinds of Code Smells [11]. However, while there is a robust community around the topic of code smells, there is no universal standard that is accepted by this community to define what defects are code smells and what aren't. The tertiary systematic review paper points out the necessity of standardizing code smells namings considering the phenomena of different names that refer to the same code smells [13]. For instance, Large Class, a well-known code smells indicating a class is handling too much work, is also known as: Large Object, Brain Class, Complex Class, God Class, God Object [11]. While some kinds of code smells aren't widely accepted in the community, others are. An empirical study conducted by Yamashita & Moonen [26] indicates that several defects, including duplicated code, Long Method and accidental complexity, are the most prevalent among the software developers. Because of the lack of standardization of the definition of code smells, the scope of this research only covered several well-known and widely-acceptable code defects in this research. Inspired by the research done by Yamashita & Moonen [26], we decided to cover the following code smells in the experiment: Complex Conditional, Long Class, Long Method, Long Message Chain, and Long Parameter List.

The researches on code smells provide valuable information about the definitions of code smells with their toxicity. With those researches, our team is able to have deep understandings enough to start detecting them in appropriate methods.

2.2 Automated Code Smells Detection

Since code smells are toxic in the software projects, it's essential to detect them to further eliminate them from the projects to improve the quality of the code and ease the future software maintenance. One of the most efficient methods to detect them is by implementing automatic code smells detection tools. It's resource-consuming and error-prone to detect code smells manually, especially in the large software projects [24]. Code smells detection tools are, therefore, developed to automate this procedure. Van Emden and Moonen built the first formalized code smells detection tool for Java. The tool was designed to analyze the meta-model, a data structure reflection of Java language like abstract syntax tree (AST), that is parsed from Java source code [7]. Since then, a significant amount of code smells detection tools emerged to detect different kinds of code smells in different programming languages. Some of them, like Pylint, CheckStyle, and ESlint, are so popular in their target languages that themselves become a part check of code view in software projects.

Nevertheless, code smells detection tools are not standardized on the scope of code smells. Give the vagueness of definition of code smells, tools implemented different techniques, metrics, and thresholds on the same code smell [16]. This divergence leads to the large gaps of detection quality among the code smell detection tools. A study done by Paive indicates that different code smell detections tools provide around 20% different defect reports on the same software project [16]. The result of this study is aligned with the fact. While both PMD (a Java-target code analyzer) and Pysmell (a Python-target code smell detection tool) [4] are able to detect Long Class code smell efficiently, they implement very different metrics. Pysmell checks two metrics: 1. length of lines with threshold 2. the number of sub-attributes and sub-methods. On the other side of spectrum, PMD checks: 1. weighed method count 2. access to foreign data 3. tight class cohesion. Even if two code smell detection tools could implement the same metics, the thresholds may still vary. Black and pycodestyle, two widely-used code smell detection tools in Python, have different default thresholds on the line-to-long code smell problem. Black suggests any line more than 88 characters is considered as too long but pycodestyle suggests 79. These kinds of divergences among code smell detection tools are limited into a single programming language, it could also appear across languages. Checkstyle has a threshold of 80 characters, which differs from both pycodestyle and Black. This divergence will be harmful significantly the most in software written in multi-languages where multiple code smell detection tools have to been implemented for different languages. we will elaborate this topic more in the Section 2.3.

The discovery on the code smell detection tools on the current state inspires us to build TreeNose, to address the lack of a language-independent code smell detection tool.

2.3 Multi-language Code smells

While most traditional code smell detection tools work on the code structure level, there are some that work on plain text level. Text level detection like regular expression check is generally considered as naive and vulnerable, thanks to the fact that text is more human-friendly but not computer-friendly. For example, one programmer may add comments in a code closure while another may not on the same program. While AST automatically ignores those comments, a regular expression check will have a hard time to filter out this kind of noise. However, the detection quality difference between AST-based approach anad plain-text based approach has been significantly shortened, thanks to the flourishing development of machine learning(ML). Kreimer [12] designed a decision tree(DT) in 2004 to detect two code smells, Long Method and Long Class, with a high accuracy. And since the early

2010s, there are increasingly more researches on ML-based code smell detection with different ML models [20] [23]. ML-based code smells detections have an advantage that AST-based code smell detections don't: They are language-independent. Based on the fact that ML-based code smell detection models are always trained with the raw source code, they are text-based and aren't restricted by the grammars of languages. This feature grants ML-based code smell detection approaches to serve multiple languages and adapt to new languages as long as models can be trained with the new programming languages.

There are few researches and tool developments that focus on the language-independent code smells and multi-system code smells. Van Emden and Moonen indicated that the approach they used in the Java code smells detection has the potential to be generalized in other object-oriented languages [7]. Since then, though more code smells detection tools have been developed to serve more object-oriented languages there is few language-independent code smells detection tool.

Two researches by Abidi and her research team [1] [2] had a peek on the code smells in multi-languages system. Inspired by the fact that a increasing amount of software projects leverage more than one single language to achieve the goals, they indicated the existence of code smells related to multi-language systems, and catalogued 12 kinds of code smells which are unique to multi-language systems [1]. These two researches expand the scope of code smells out of mono-language systems.

The researches conducted in this section emphasizes the significance of language-independent code smells in multi-language systems leading by their popularity and complexity. Those researches firm the igh need of a language-independent code smell detection tool to resolve the issues led by multi-language systems.

2.4 Tree-sitter: A Language-independent Parser Generator

As mentioned in the Introduction chapter, while different programming languages have different syntaxes, they tend to have the similar code structures and ASTs. To detect code smells across programming languages, we need a tool to represent code in different programming languages into similar code structures. To accomplish this task, this research implements tree-sitter [3], an intermediate code structure parser generator tool.

Tree-sitter is a parser generator tool able to parse programming languages in any supported language, where the parsing language doesn't need to be the same with the runtime language. Parsing is a process of analyzing a string of tokens in programming languages. It represents those tokens or raw texts into tree nodes, helping computer to understand the structure of the code. Parser is a crucial part of a compiler and exists in most modern programming language runtime environments.

Tree-sitter is consist of two components, a grammar and a binding. A grammar guides the parser to decompose the raw syntaxes into AST code structures. On the other hand, a binding provides a runtime environment to execute the tree-sitter with grammars and provide the features like generating tree-structures and querying in them. A grammar and a binding are like a chef and a waiter: A chef decides the menu and a waiter interacts with the customers to send requests to the chef and serve the dishes to customers. Like so, a binding returns the tree-structure from the code a user provides. While the parsing rules of the code are defined by the parsers.

The separation of parsers and grammars grants tree-sitter the ability to work on language A and parse different language B. Since grammars are purely written in JSON, and parsers are written in pure C language, tree-sitter is robust enough to support a significant amount of programming languages. You can parse Python code with a Java binding, or parse HTML in C# [3].

Tree-sitter allows users to combine any supported grammar with any supported binding, inspiring us to implement it in one binding and parse different languages and get their code structures, the representations that are more general and less language-specific. Here are the code structures parsed from the `hello world` examples in different languages.

Hello World in Python with It's Tree-sitter Structure:

```
print("Hello World!")
```

```
expression_statement(0,0) , (0,21)
       call(0,0) , (0,21)
               identifier(0,0) , (0,5)
               argument_list(0,5) , (0,21)
                       ((0,5) , (0,6)
                       string(0,6) , (0,20)
                               string_start(0,6) , (0,7)
                               string_content(0,7) , (0,19)
                               string_end(0,19) , (0,20)
                       )(0,20) , (0,21)
```

Hello World in Java with It's Tree-sitter Structure:

```java
System.out.println("Hello World!");
```

```
expression_statement(0,0) , (0,35)
        method_invocation(0,0) , (0,34)
                field_access(0,0) , (0,10)
                        identifier(0,0) , (0,6)
                        .(0,6) , (0,7)
                        identifier(0,7) , (0,10)
                .(0,10) , (0,11)
                identifier(0,11) , (0,18)
                argument_list(0,18) , (0,34)
                        ((0,18) , (0,19)
                        string_literal(0,19) , (0,33)
                                "(0,19) , (0,20)
                                string_fragment(0,20) , (0,32)
                                "(0,32) , (0,33)
                        )(0,33) , (0,34)
        ;(0,34) , (0,35)
```

Hello World in JavaScript with It's Tree-sitter Structure:

```javascript
console.log('Hello World!')
```

```
expression_statement(0,0) , (0,27)
        call_expression(0,0) , (0,27)
                member_expression(0,0) , (0,11)
                        identifier(0,0) , (0,7)
                        .(0,7) , (0,8)
                        property_identifier(0,8) , (0,11)
                arguments(0,11) , (0,27)
                        ((0,11) , (0,12)
                        string(0,12) , (0,26)
                                '(0,12) , (0,13)
                                string_fragment(0,13) , (0,25)
                                '(0,25) , (0,26)
                        )(0,26) , (0,27)
```

In this simply example, the code structures provided by different tree-sitter parsers are highly in common. Those different languages don't agree on the syntaxes, while they function the same. The statements printing `Hello world` utilize different method names, and different quote signs to represent string literals. However, their AST code structure trees, other the other hand, are much more aligned in this example! For instance, all three of code structures start with a expression statement and contain methods, identifiers and strings. To conclude, the code structures provided by tree-sitter are more language-independent than the raw syntaxes.

Tree-sitter enables us to implement code smell detection on the universal code structure level for different languages. However, it's worthy to mention that the code structures provided by tree-sitter are similar but not exactly the same. Giving the simplest example,`Hello World` example we provided above, doesn't have the same structures in different languages. Some nodes like `method_invocation` and `call`, two different node names in the code structure tree, actually refer to the same type statement, the call statement. This divergence is mainly led by two reasons:

1. Programming languages aren't working in the same approach
2. Tree-sitter itself doesn't have strictly defined rules for its grammars

Programming languages are developed under different philosophies to solve different different problems. You can't expect HTML to be anything alike C. Thus, the same program written in different languages wouldn't have the same AST trees. For example, if you want to create a class declare some attributes (also known as fields), you are able to do so in any method under a class in Python, but not possible at all in Java since Java strictly limits you to define attributes directly under the class level other than method level.

One of the problems of tree-sitter is: it's not aligned with the terminologies used to define the same kinds of tokens different languages. In the Hello World example, all three language parsers return the Arguments token to indicate the parameter, Hello World!, in the log function calls. However, this kind of token, Arguments, does't have a unified name, it's called arguments in JavaScript, but `argument_list` in both Java and Python. The divergences among programming languages causes one of the biggest problems our tool aims to solve to detect code smells across languages.

In conclusion, we believe the team in tree-sitter makes a significant contribution on the language-independency and generality when it comes to parsing.

3. Method of Approach

3.1 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 [10]. Since then a variety of code smells have been introduced to describe a wide range of anti-patterns, bad habits and code design problems. In this research, we chose five kinds of the most common, widely-accepted and, researched code smells that could potentially exist in multiple programming languages: Complex Conditional, Long Class, Long Method, Long Message Chain, and Long Parameter List.

3.1.1 Complex Conditional (CC)

As one of the most basic and important components in the control flow, conditional statements are widely utilized in every programming language in different forms. In some languages like Python, conditional statements are represented by IF statement while in other languages like Java and JavaScript, conditional statements are consist of both IF statements and Switch statements. This popularity of conditional statements leads CC rampant. CC indicates a conditional statement is too complex to understand for any of the following reasons:

1. Too many conditional cases
2. Condition statement is too complex
3. The size of the body inside is too big

In this research, we mainly focus on the first case: conditional statement with too many cases. In the future, we will extend TreeNose's ability to detect CC from the perspective of case 2 and case 3.

Here are several examples of code with too many conditional cases:

Complex Switch Statement:

```
int value = 5;
switch (value) {
    case 1:
        System.out.println("Value is 1");
        break;
    case 2:
        System.out.println("Value is 2");
        break;
    case 3:
        System.out.println("Value is 3");
        break;
    case 4:
        System.out.println("Value is 2");
        break;
    default:
        System.out.println("No match!");
}
```

Complex IF statement:

```
int value = 5;

if (value == 1) {
    System.out.println("Value is 1");
} else if (value == 2) {
    System.out.println("Value is 2");
} else if (value == 3) {
    System.out.println("Value is 3");
} else if (value == 4) {
    System.out.println("Value is 4");
} else {
    System.out.println("No match!");
}
```

Compound Complex Conditional statement:

```
int value = 5;

if (value <= 2){
    switch (value) {
    case 1:
        System.out.println("Value is 1");
        break;
    case 2:
        System.out.println("Value is 2");
        break;
    case 3:
        System.out.println("Value is 3");
        break;
    }
} else if (value <= 4) {
    switch (value) {
    case 3:
        System.out.println("Value is 3");
        break;
    case 4:
        System.out.println("Value is 2");
        break;
    }
} else (
    System.out.println("No match!");
)
```

All code shown above contains Complex Conditional code smell. In the case of Java, both IF statements and SWITCH statements could contain too many cases. Furthermore, when combining if statements within switch statements and vice versa, the resultant structure may contribute to a compound and complex conditional statement.

3.1.2 Long Class (LC)

When a class is consist of too many fields or methods, it's considered as too long [10]. LC, also known as Blob, LC, Brain Class and Large Class, is a bloater smell on the abstraction level. A LC often comprises too many components, making it hard to comprehend. LCes are so powerful that it's nearly impossible to run the program without using it. In other words, long classes normally handle excessive functionalities, making it against Single Responsibility Principle (SRP). Because of the popularity of class in modern programming languages, LC widely exist in the modern programming languages.

Here is an example of code snippet with LC in Python

```python
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_item(self, item):
        self.items.append(item)

    def remove_item(self, item):
        self.items.remove(item)

    def calculate_total_price(self):
        total_price = 0
        for item in self.items:
            total_price += item.price
        return total_price

    def apply_discount(self, discount):
        for item in self.items:
            item.apply_discount(discount)

    def checkout(self):
        self.apply_discount(0.1)  # Apply 10% discount
        total_price = self.calculate_total_price()
        print(f"Total Price after discount: ${total_price}")

class Item:
    def __init__(self, name, price):
        self.name = name
        self.price = price
    def apply_discount(self, discount):
        self.price *= (1 - discount)


# Usage
cart = ShoppingCart()
item1 = Item("Shirt", 20)
item2 = Item("Jeans", 30)
cart.add_item(item1)
cart.add_item(item2)
cart.checkout()
```

the ShoppingCart class comprises 6 methods, making it extremely large. Some methods like checkout and applying_discount within shouldn't be under ShoppingCart class since those functionalities aren't children of ShoppingCart system. To address the code smell in this example, it's preferable to extract the checkout and apply_discount methods into a new class named CheckOutProcessor to house them.

3.1.3 Long Method (LM)

LM was first defined by Fowler and Beck. The more lines of code a method has, the more difficult it's to understand. The longer a method is, the more mental frustration it introduces when one tries to decompose it [10]. LM is considered as one of the most popular code smells [26]. It's mainly because of the popularity of methods and functions in programming languages.

3.1.4 Long Message Chain (LMC)

LMC happens when a series of clients calling objects by objects [10]. Normally those clients contribute to a gigantic expression with a significant amount of objects attributes and method calls. The presence of LMC indicates the program comprises bad code design where redundant steps must be taken to achieve certain task.

In the case of TreeNose, we identify Long Message Chain by counting two syntactic components: 1. method call 2. object attribute. While tools in market only define message chains as pure method call chains or pure object attribute chains, TreeNose will also put hybrid chains in count.

Here is an example of LMC in Python:

```
class A:
    def __init__(self):
        self.value = 42

    def get_val(self):
        return self.value

class B:
    def __init__(self):
        self.a_instance = A()

    def get_A(self):
        return A()

class C:
    def __init__(self):
        self.b_instance = B()

    def get_B(self):
        return B()

c_instance = C()

c_instance.b_instance.a_instance.value

c_instance.get_B().get_A.value
```

Have a look on the last two lines and we will get a consensus that the calling chains are long and hard to comprehend. To understand the purposes of these two chains, a strong understanding on functionalities of each method call or attribute with its expected inputs and outputs are preferred.

3.1.5 Long Parameter List (LPL)

LPL is another code smell introduced by Fowler and Beck.[10] It indicates the extremely long data flow to methods as inputs. A long list of parameters of one method constantly suggest that this method is in charge of too many tasks, ending up to be too powerful and against SRP. While LPL could be a consequence of avoiding global data, a bad data structure is generally considered evil, LPL themselves are confusing in their own right because they are the manifestations [10]. Just like LMs, LPLs are prevalent in modern programming languages along with the popularity of method statements.

Here is an example of LPL in Python:

```python
def method_with_long_params(a, b, c, d, e, f):
    print(a + b + c + d + e + f)
```

6 parameters are extremely excessive for such a method. And it's better to bind some parameters.

3.2 System Design of TreeNose

Code structure-based and AST-based detection is proved to be precise, which make it the most popular detection strategy in the field of code smells. To apply AST-based detection, TreeNose detect code snippets on the code structure level. The diagram below shows the architecture of the tool.

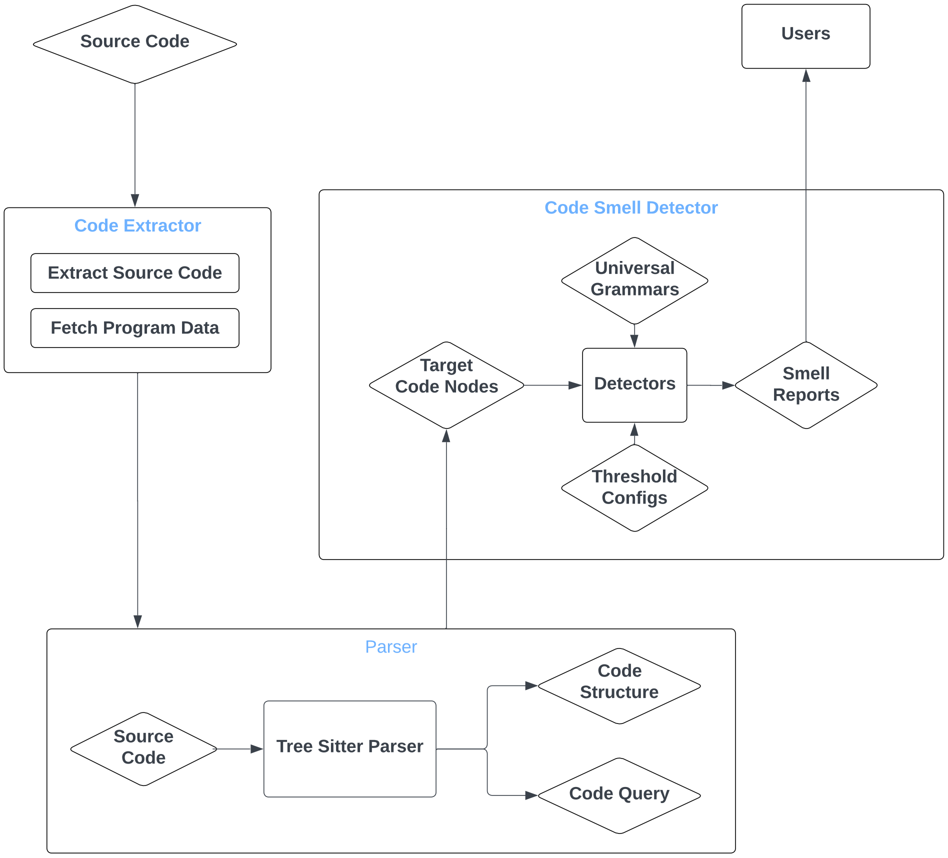


Figure 3: Architecture of the Program

There are 5 important modules shown in Figure 3: 1. Input 2. Code extractor 3. Parser 4. Code Smell Detector 5. Output. The rest of this section will explain the purpose and

3.2.1 Input Source Code

TreeNose are designed to serve software projects across Python, Java, and JavaScript, where the entire programs, including both source code and test code, are the inputs. Some people may argue that test code shouldn't be covered in the coverage, due to the fact that test cases don't contribute to the functionalities of the program. We think that may be a valid point, but only to users. As we introduced before, code smells are toxic since they block the potentials of maintenance. In other words, code smells matter to software developers other than users. While test cases don't matter to users and won't introduce any bug to them, they are valid and crucial parts of the software in development. During the processes of evaluation and development testing, TreeNose demonstrated to smoothly run in a range of large open-source projects, making it trustworthy and stable.

3.2.2 Code Extractor

To fetch the source code inside those open-source projects, we built a file system traverser to recursively traverse the folders and files of one project and fetch the target source code files. When mentioning target files, we check the extension name to match the target language to fetch all the files written in the target language.

While TreeNose fetches the most parts of the program, there are some parts in the program it considers as noise: Programs sometimes utilize others' programs as the dependencies. While the code quality of those dependencies don't matter to software developers since they the users but not developers of those dependencies. Therefore, we implemented a `path_ignore` feature in regular expression fashion to filter out those noises in the code extractor.

Here is an example of `path_ignore` file and the code of implementing it.

path_ignore example:

```
^.*node_modules.*$
(?:^|\/)\.[^\/]+
```

Parsing path_ignore:

```
function generateIgnoreRegex() {
    const text = fs.readFileSync('./configs/.path_ignore', 'utf8');
    const lines = text.trim().split("\n").
    filter(line => line.trim() !== "");
    const ignoreRegex = '(' + lines.join(")|(") + ')';
    return ignoreRegex
}
```

This example filters out paths anywhere with “node_modules” in it and paths with hidden files or folders. the `generateIgnoreGex` function in JavaScript code combines all the regex in `path_ignore` file to generate a long ignorance regular expression.

3.2.3 Tree-Sitter Parsing with Categorized-Grammars

Tree-sitter is able to parse source code into code structures in format of tree based on the grammars. It returns the root node of the tree sitters as an object. One node object contains valuable attributes like *children nodes* and *child count* and *Position*, as well as methods like *toString* and *walk*.

With the root node, tree-sitter provides the functionality to query patterns in the root node and return the matching nodes.

To better understand the process, we provide a short code example here:

```
// Set the tree-sitter parser to the target language
const targetLang = require(`tree-sitter-python`);
parser.setLanguage(targetLang);

// Get the tree
const tree = parser.parse(sourceCode);

const pattern = "method_definition"
// Set up query pattern
const query = new Parser.Query(targetLang, `(${pattern}) @${pattern}`);

// capture the match nodes
captures = query.matches(tree.rootNode).map((x) => x.captures)
```

This feature of query is perfect to narrow down the code scope. the code snippet above captured method definitions from the source code, allowing TreeNose to locate candidate methods with potential LM and LPL code smells.

However, as mentioned in the Tree-sitter Section, the code structures generated from different languages don’t always perfectly. For example, function definition statement is called **function_definition** in Python tree-sitter but called **method_definition** in JavaScript tree-sitter. The divergences among different tree-sitter grammars hinder us to fetch code snippets across different languages under the same set of rules. To generate a language-independent tool, TreeNose is configured with a language-independent syntax grouper that categories tree-sitter grammars and syntaxes from different languages into universal ones. As tree-sitter grammars are written in JSON, we implemented a group JSON configuration file to bind grammars.

Here is an example of generalized JSON configuration file:

```
{
    "grammars":{
  "method_definition": {
      "python": [
          "function_definition"
      ],
      "javascript": [
          "method_definition",
          "function_declaration",
          "arrow_function",
          "generator_function_declaration",
          "function"
      ],
      "java": [
          "method_declaration", "constructor_declaration"
      ]
  }
}
}
```

Programming languages have various implementations on the same abstract statement. In the case of method definition, JavaScript and Java have several types of them. To TreeNose, the differences between sub-types of method definitions don't bother as all kinds of them include our desired components: parameters and line of codes. By putting them under the same key, `method_definition`, TreeNose is able to treat them language-independently.

This syntax grouper is highly configurable and extendable, which enables the tool to fit new programming languages in the future. With the generalized language-independent syntaxes, the tool will be able to detect code smells regardless of the syntaxes of programming languages.

3.2.4 Code Smell Detector

The code smell detector is in charge of analyzing the generalized code structure and detecting code smells within. AST-based Code smell detection tools usually leverage a series of metrics and thresholds, allowing them to distinguish code with bad smells from the one with bad smells. However, Different code smell detection tools may analyze the same code smell from different perspectives. In other words, those tools may leverage different thresholds and metrics on the same kind of code smell. As metrics and thresholds vary, so do the accuracy of detections. Therefore the selection of the right metrics and thresholds is the first step of success. After rounds of exploration, we finally decided to implement the thresholds and metrics that are highly aligned with Pysmell [4]. The researchers in Pysmell defined those metrics and thresholds based on programming experience and literature review [4]. And as a significance, Pysmell achieved the average precision of 97.7% in the experiment designed by the Pysmell team. Table 1 presents metric-based detect strategies for each smell in TreeNose.

Table 1: Detection Metrics and Thresholds

Code Smell	Thresholds	Metrics
Complex Conditional	LC	LC: length 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	LMC: length of message chains

In TreeNose, all thresholds are stored in the same configuration file, allowing users to change them to fit their own needs.

After getting the thresholds, detector analyzes the associated syntax nodes captured from tree-sitter to verify presence of code smells. The detector fetches the generalized syntaxes and grammars in this step as inputs, which enable the entire process of detector language-independent. As an achievement, new languages can be easily introduced by implementing grammars with the need of modifying the detector.

3.2.5 Code Smell Report

The captured code smells needs to be display to users in an organized and readable method. TreeNose, therefore, implements a sets of CSV (Comma Separated Values) files to record those code smells. Each type of code smell is resided under separate CSV (Comma Separated Values) file with the key attributes like smell type, location, threshold, and actual captured amount. Here is a short example.

```
Line,Smell,Number of Lines,Threshold,File
127,long method,105,100, ./examples/file1.py
```

The `Line` and `File` attributes help users to locate the code smells in the project. And `Numbers of lines` and `Threshold` explain why the code snippet is smelly.

3.3 Results

TreeNose currently resides in a public repository. Users and researchers can easily merge TreeNose under their projects by cloning it with the command

```
git clone git@github.com:ReadyResearchers-2023-24/YanqiaoChenArtifact.git
```

Then install the runtime environment with following commands

```
cd src
npm install
```

Once the setup is done, TreeNose provides the CLI (Command Line Interface) to run it. It requires three flags to function:

- input: Path to a directory or to a single file
- lang: The target programming language that the target file is written in
- out: Path to a directory the reports will reside

- smell (optional): The smell types to detect, you are able to pass several code smells separated by space

Where each flag requires certain attribute:

1. `lang` flag accepts: 1. java 2. python 3. javascript
2. `smell` flag accepts: 1. long-class 2. long-param 3. long-method 4. long-message-chain 5. complex-conditional
3. if a directory is provided to flag `input`, TreeNose will fetch all the target files under both the current directory and all of its subdirectories in any level.
4. By default TreeNose checks all kinds of code smells when no `smell` flag needs to be set up

Here is an example:

```
node main.js --input ../example_codes/python_codes --lang python --out ../reports
```

This command will take `input` as code source input, select all files under matching `lang`, detect the desired code smells set by `smell`, and finally generate several csv detection reports under the `output` path.

4. Experiments

4.1 Experimental Design

This research conducts three research questions:

1. How is TreeNose's performance in different languages?
2. Do code smells happen in software projects regardless of the selection of language?
3. Do some code smells occur more often than others in some languages?

To answer the research questions and evaluate the precision of TreeNose, our team conducted a series of experiments.

4.1.1 Research Question 1

Question 1 asks about the performance of TreeNose, where the term "performance" refers to the accuracy of TreeNose. High accuracy score is essential to verify the quality of TreeNose and keeps the high confidence in TreeNose when researchers and programmers utilize it.

To answer question 1, we implemented both the already-exist code smell detection tools and TreeNose on the same open-source software projects, and then compare their reports on the same kinds of code smells. In this research, we selected three code smell detection tools as counterparts: Pysmell @[pysmell], Jscent @[Jscent], and DesigniteJava @[DesigniteJava]. TABLE 2 plots the information of the selected code smell detection tools.

Table 2: Selected Open-Source Code Smell Detection Tools

Tool	Supported Language	Supported Code Smell
Pysmell	Python	Complex Conditional, Long Class, Long Method, Long Message Chain, Long Parameter List
Jscent	JavaScript	Long Class, Long Method, Long Message Chain, Long Parameter List, Long switch statement*
DesigniteJava	Java	Complex Conditional, Long Method, Long Message Chain, Long Parameter List

Note: 1. long switch statement: long switch statement is one kind of Complex Conditional, another kind is long if statement.

If there was any difference between the results of theirs with ours, we picked some samples of those differences and verify them by checking the source code manually. The tool performance is evaluated by the precision of the results of TreeNose, where the precision score follows this formula:

$$Precision = \frac{TruePositive}{TruePositive + FalseNegative}$$

Note:

- **True Positive**: Captured code smells that are truly smelly
- **False Positive**: Captured code smells that aren't actually smelly
- **True Positive + False Negative**: Total of captured code smells

By comparing the output datasets of those tools with ones of TreeNose, we are able to get the answer of if TreeNose is of high quality as much as those tools. If there is any difference between the reports of TreeNose and other tools, we conducted a manual code check to verify the several samples of difference parts.

4.1.2 Research Question 2

Question 2 is critical to this research, considering the biggest significance of this research is to be able to detect code smells in different programming languages. TreeNose comes with the ability to detect code smells across languages under the same set of rules, metrics and thresholds. This feature of TreeNose makes it different from other code smell detection tools like the ones mentioned above. If we applied different tools for projects in different languages, the divergences of detection methods in tools will significantly affect the quality of reports. On the other hand, TreeNose is a perfect candidate to detect in open-source projects in different languages under the same rules.

In this research, TreeNose was utilized on multi-language projects too. More and more open-source projects nowadays select multiple languages to maximize the strengths of each language and avoid the disadvantages.

While some people may say that to filter out the effect of quality divergences of open-source projects, it's ideal to compare the proportion of each smell in softwares that are written in different languages but function the same. However, it's difficult to find such an open-source project, and we failed to find one. Instead, to minimize the effect of the softwares' a wide variety of qualities, we select at least 3 software projects for each programming language, and at least one of those projects is large with 10,000 lines of code or plus.

To ensure versatility and accuracy of the result, all the open-source projects are relatively well-known and widely-used.

If the result of this experiment presents that code smells are language-independent, then we will do further analysis on the distribution of the smells by applying the following formula.

$$AveragePortion = \frac{\sum PortioninInEachProject}{NumberofProjects}$$

We get the portion of each code smell in single project and then calculate the average, which prevents a few giant projects from dominating the result.

After TreeNose generates the code smell reports, our team manually verify the existence of code smells in those projects.

4.1.3 Research Question 3

Question 3 is a extension of question 2 and yet from a different point of view. On the basis of the experiment mentioned above, we fetch the results and calculate the occurrence of each code smell in the entire program. Here is the formulas we will implement in the experiment:

$$Occurrences of Smell = \frac{\sum \frac{Amount of Smell in One Project}{Total Line of Code In One Project}}{Amount of Projects}$$

4.2 Evaluation

4.2.1 RQ 1: How is TreeNose's Performance in Different Languages?

By comparing the reports of TreeNose with the results of others, TreeNose presents high precision. Under the same detection techniques and thresholds, TreeNose is able to fully cover the results of other code smell detection tools and sometimes capture more code smells than other tools.

TreeNose and Pysmell

In the research of Pysmell, Chen and his team achieved average 97.7% precision [4]. With that saying, we'd compare the reports of TreeNose and Pysmell to see if TreeNose is able to achieve such a high score.

As mentioned above in Section 3.3.4, the detection metrics are highly aligned with Pysmell. However, they aren't fully the same. For the metrics that are different, we tweaked the threshold of Pysmell based on some experimental examples to make the report more alike to TreeNoes. Here are a table of differences on the metics:

Table 3: Detection Metrics and Thresholds in TreeNose and Pysmell

Code Smell	TreeNose	Pysmell
Long Class	LOC > 200, NOM > 20	CLOC > 10
Long Method	LOM > 100	MLOC > 70
Long Parameter List	PAR > 5	PAR > 5

Code Smell	TreeNose	Pysmell
Complex Conditional	NOC > 3	not supported
Long Message Chain	LMC > 4	LMC > 4

- LOC: lines of class
- CLOC: cyclomatic complexity
- LOM: length of method
- MLOC: lines of code in methods
- PAR: number of parameters
- NOC: number of cases
- LMC: length of message chains

Note: LOM and MLOC aren't exactly the same. In the scope of Pysmell, MLOC only contains the lines with code, where doesn't include pure comments and blank lines. While in TreeNose, the LOM put everything line of method in count. Here is an example:

```python
def i_am_a_method():
    """comment 1"""
    # comment 2

    print("hello world")
```

This method is consist of LOM score 5 and LOM score 2.

By applying both Pysmell and TreeNose to the following Python open-source projects, we are able to check the accuracy of TreeNose.

Table 4: Information about Selected Python Open-Source Projects

Open-Source Project	Version	Description
django	1.8.2	a high-level Python web framework
numpy	v1.9.2	a fundamental package for scientific computing with Python.

Open-Source Project	Version	Description
nltk	3.0.2	a suite of open-source Python modules, data sets, and tutorials supporting research and development in Natural Language Processing

Here is the table of reports:

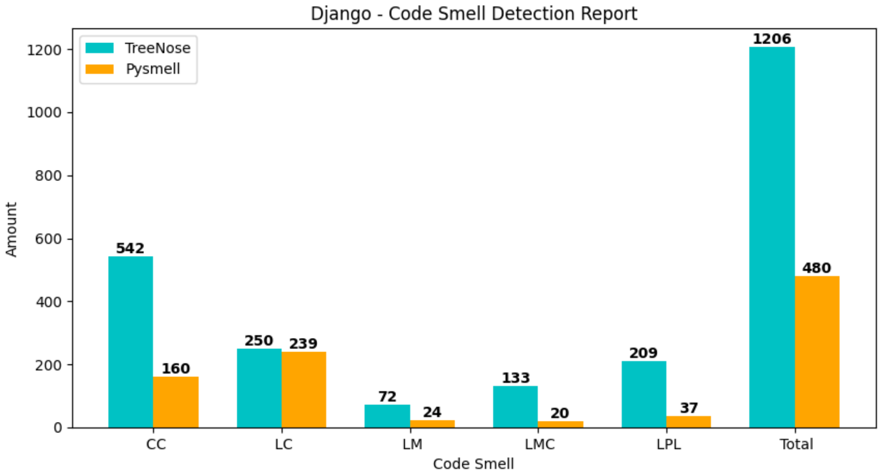


Figure 4: Django Code Smell Report

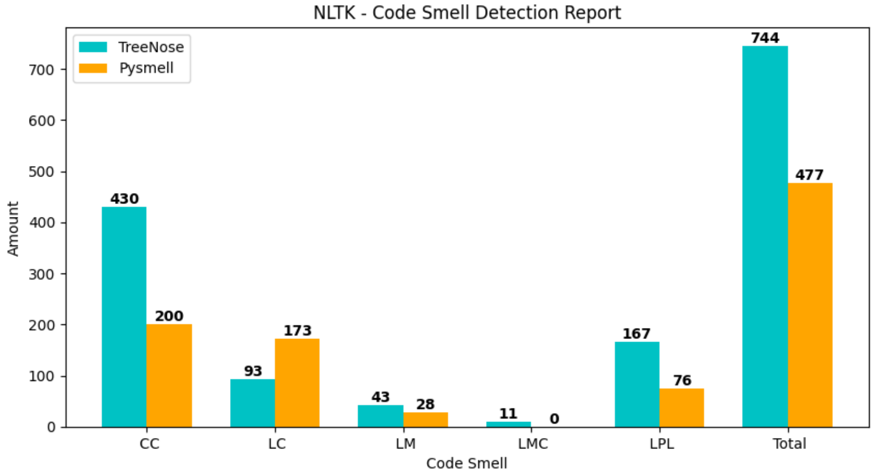


Figure 5: NLTK Code Smell Report

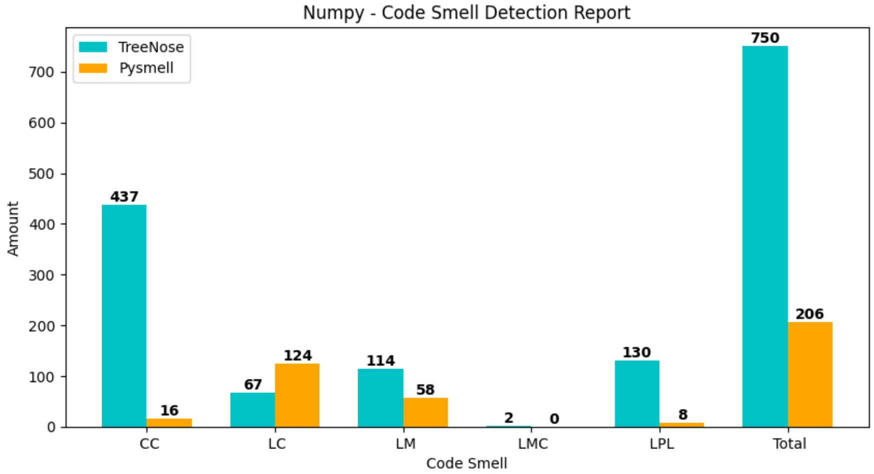


Figure 6: Numpy Code Smell Report

Shown in reports above, TreeNose is able to report 3 times more code smells than Pysmell. While the difference between metrics on certain smells may be a significant factor, our team tweaked the thresholds of TreeNose in advance to make their reports more alike.

Actually the differences between the detection metrics of Pysmell and TreeNose disclose valuable information. For example, they implement similar metrics on the detection of LMC and LPL, but end up with large gap on the reports. After checking some samples, we found that Pysmell isn't able to cover all the corner cases, while TreeNose is able. In the case of LPL, one of the main differences is that TreeNose is able to correctly recognize default parameter. For example, TreeNose will report 6 parameters, but Pysmell will report 3. In the comparison of the reports of TreeNose and Pysmell, TreeNose is shown to be able to fully cover the LPL smells reported by Pysmell.

```
def problematic_method(a, b, c, d = 1, e = 2, f = 3):
    pass
```

In the case of LMC, Pysmell and TreeNose has different definition on the LMC. TreeNose counts only attribute but TreeNose counts both attribute and method call. In the following example, TreeNose reports 2 LMC smells, while Pysmell reports only the first one.

```
class A:
    def __init__(self):
        self.value = 42

    def get_val(self):
        return self.value

class B:
    def __init__(self):
        self.a_instance = A()

    def get_A(self):
        return A()

class C:
    def __init__(self):
        self.b_instance = B()
    def get_B(self):
        return B()

c_instance = C()

c_instance.b_instance.a_instance.value

c_instance.get_B().get_A.value
```

We carefully checked the reports of TreeNose and compared them with Pysmell and finally verify that TreeNose is able to achieve more than 90% precision.

TreeNose and Jscent

Jscent is a JavaScript-focus code smell detection tool. Here is a table of the metrics and thresholds that Jscent and TreeNose implement on the target code smells:

Table 5: Detection Metrics and Thresholds in TreeNose and Jscent

Code Smell	TreeNose	Jscent
Long Class	LOC > 200, NOM > 20	LOC > 200, NOCC > 20
Long Method	LOM > 100	LOM > 100
Long Parameter List	PAR > 5	PAR > 5
Complex Conditional	NOC > 3	not supported
Complex Switch Statement*	NOC > 3	NOC > 3
Long Message Chain	LMC > 4	LMC > 4

- LOC: lines of class
- NOCC: number of children in class
- LOM: length of method
- PAR: number of parameters
- NOC: number of cases
- LMC: length of message chains

Note: 1. Complex Switch Statement is one kind of Complex Conditional. Another kind is Complex If Statement.

As we can see, Jscent and TreeNose apply almost the same metrics for detection, which ideally will make the reports of theirs more alike than Pysmell and TreeNose.

In this research, we implemented both TreeNose and Jscent to following open-source projects:

Table 6: Information about Selected JavaScript Open-Source Projects

Open-Source Project	Version	Description
Moment.js	2.30.0	A project that parses, validates, manipulates, and displays dates in javascript
Lodash	4.0.0	A modern JavaScript utility library delivering modularity, performance, & extras
Chance.js	1.1.0	Random generator helper for JavaScript

After implementing two tools, here is the table of the experimental result.

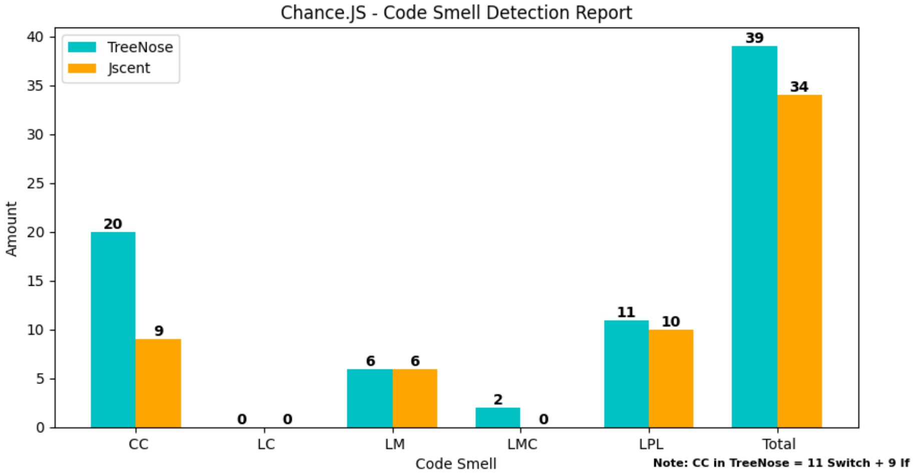


Figure 7: Chance Code Smell Report

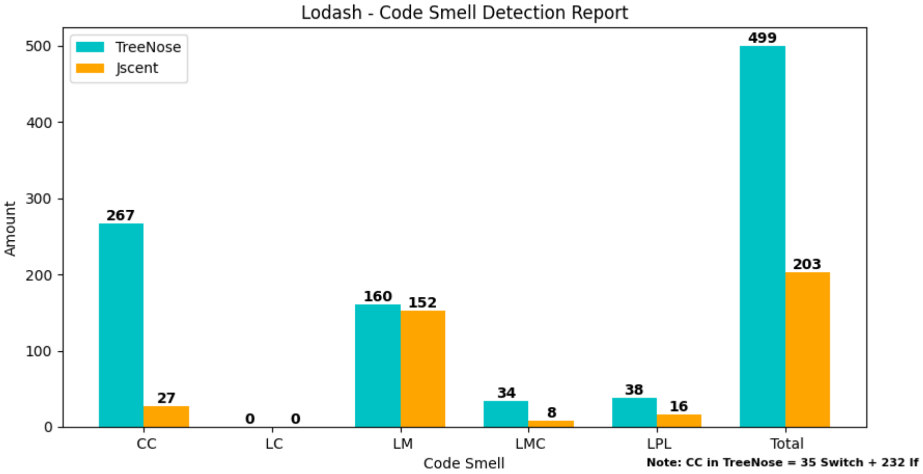


Figure 8: Lodash Code Smell Report

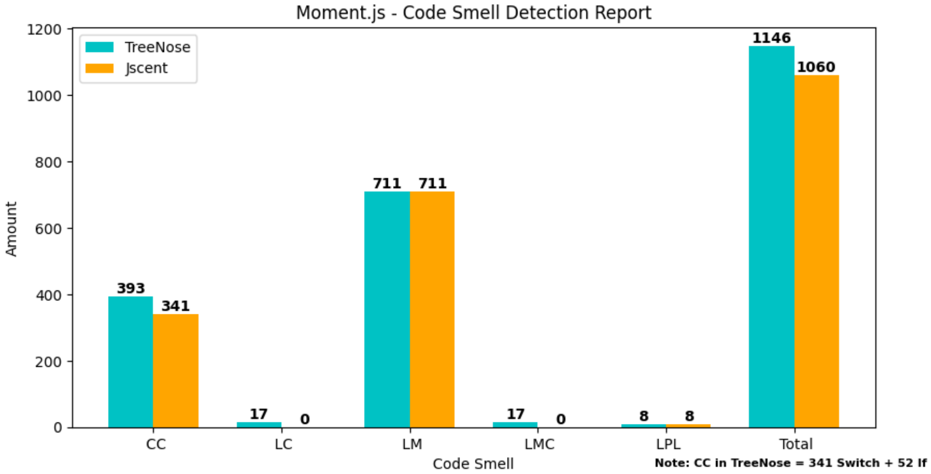


Figure 9: Moment Code Smell Report

As shown above, TreeNose returns the reports that are high in common with Jscent, especially when it comes to Long Method code smell. For the difference parts of their reports, we verified the source code manually and draw the following conclusion.

1. TreeNose counts switch cases plus the if statements in single switch case while Jscent counts only switch cases
2. TreeNose counts both method calls and attributes message chains, while JScent checks only pure attribute chains
3. TreeNose is able to achieve more than 90% accuracy in JavaScript

TreeNose and DesigniteJava

DesigniteJava is a code quality assessment tool for Java. It detects amazingly 17 kinds of design smells and 10 implementation smells. Unfortunately, it does't cover all the code smells detected in TreeNose. Here is the table of the code smells with their metrics that both DesigniteJava and TreeNose support.

Table 7: Detection Metrics and Thresholds in TreeNose and Jscent

Code Smell	TreeNose	Jscent
Long Method	LOM > 100	MLOC > 100
Long Parameter List	PAR > 5	PAR > 5
Complex Conditional	NOC > 3	not supported
Complex Switch Statement*	NOC > 3	NOC > 3

Note: - CLOC: cyclomatic complexity - LOM: length of method - MLOC: lines of code in methods - PAR: number of parameters - NOC: number of cases

Note:

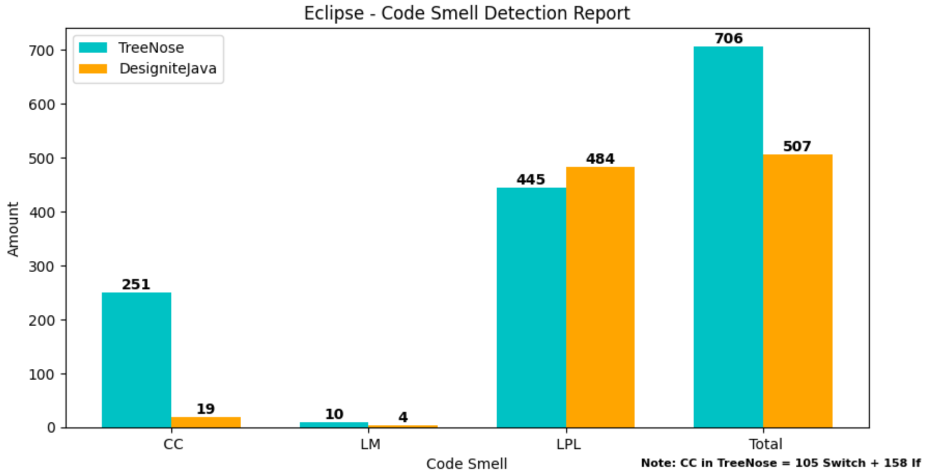
1. Complex Switch Statement is one kind of Complex Conditional. Another kind is Complex If Statement.

we run both TreeNose and DesigniteJava on the following open-source projects to compare their results.

Table 8: Information about Selected Java Open-Source Projects

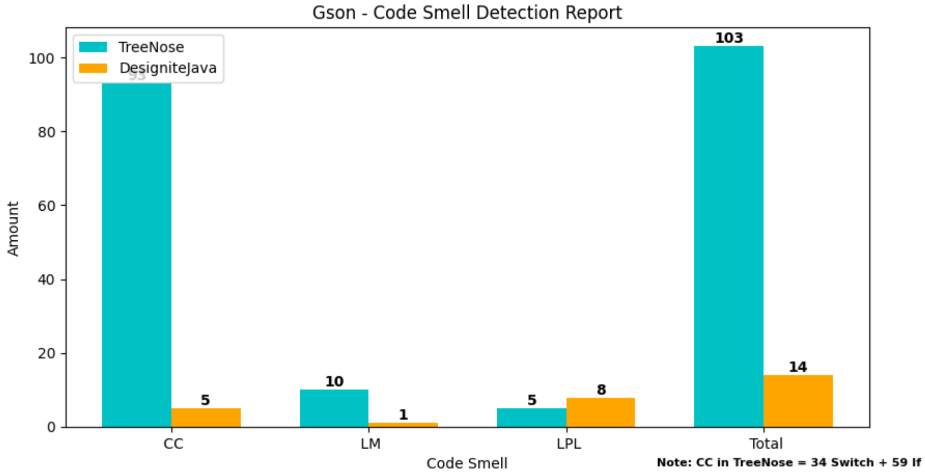
Open-Source Project	Version	Description
Maven	3.9.3	Apache Maven core
Gson	2.10.0	A Java serialization/deserialization library to convert Java Objects into JSON and back
Eclipse	11.1.0	Collections framework for Java with optimized data structures and a rich, functional and fluent API

After running both tools, here are the tables of results.



Note: CC in TreeNose = 105 Switch + 158 If

Figure 10: Eclipse Code Smell Report



Note: CC in TreeNose = 34 Switch + 59 If

Figure 11: Gson Code Smell Report

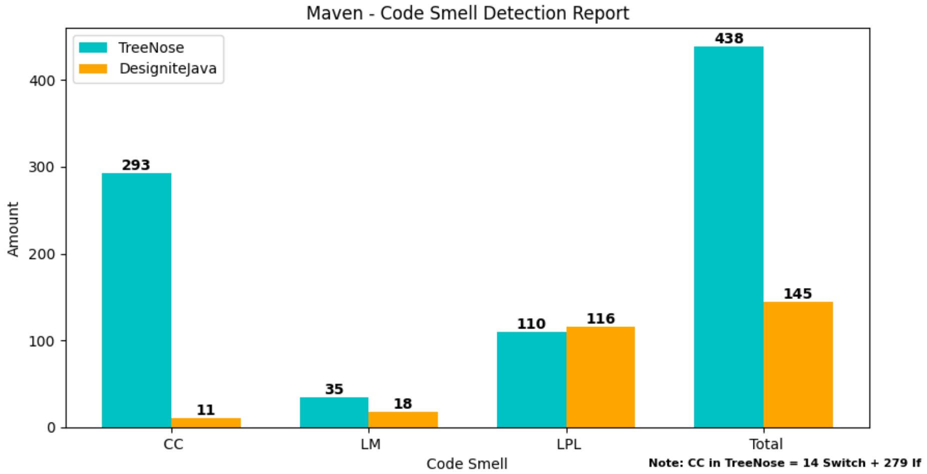


Figure 12: Maven Code Smell Report

After carefully reviewing the results of both tools, we found that TreeNose was almost able to cover all the code smells reported by DesigniteJava. However, DesigniteJava reports more Long Parameter List code smells than TreeNose. After verification, the code smells that were't reported by TreeNose are prone to be true positive. In other words, DesigniteJava did better than TreeNose on detection of Long Parameter List. However, since TreeNose covered 90% code smells that DesigniteJava detects, we will confidentially believe that TreeNose achieves a high precision on code smell detection in Java. After comparing TreeNose with three code smell detection tools in different different language, we conclude that TreeNose is able to achieve over 90% precision.

4.2.2 RQ2: Do code smells happen in software projects regardless of the selection of language?

This question is significant on discussion of if code smells are language-independent or not. The high precision score calculated in RQ1 shows TreeNose is reliable on code smell detection. Therefore, we can confidentially apply TreeNose to more open-source projects to discover our RQ2 and RQ3. To achieve more precise language-level code smell detection and filter out the effect of quality divergence of open-source projects, we implemented TreeNose to more open-source projects in the experiment of RQ 2 and RQ 3 on basis of the projects selected in the last experiment.

Table 9: Open-Source Project Detail for Experiment 2 and 3

Open-Source Project	Version	Language
Eclipse	11.1.0	Java
GSON	2.10	Java
Maven	3.9.3	Java
Spring Framework	5.3.32	Java
JUnit	5.10.0	Java
Netty	4.1.68	Java
Django	1.8.2	Python
NumPy	v1.9.2	Python
NLTK	3.0.2	Python
Pandas	2.2.1	Python
Pytest	8.1.1	Python
Flask	3.0.2	Python
Chart.js	v3.3.0	JavaScript
Moment.js	2.30.0	JavaScript
Lodash	4.0.0	JavaScript
Chance.js	1.1.0	JavaScript
Riot.js	v9.1.0	JavaScript
Express.js	4.18.3	JavaScript
Node CLI	v10.5.0	JavaScript
Zulip	8.0	JavaScript + Python
Node.js	v21.7.0	JavaScript + Python
Wagtail	6.0.0	JavaScript + Python
JPython	v2.7.3	Java + Python

After collecting the code smells in each project, we calculate the proportion of the smells in each project and calculate the averages of code smells in all the projects in the same language to compute the proportion of the smells on the language level. The benefit of this formula is that it can avoid the mathematics bias because of some extremely large projects.

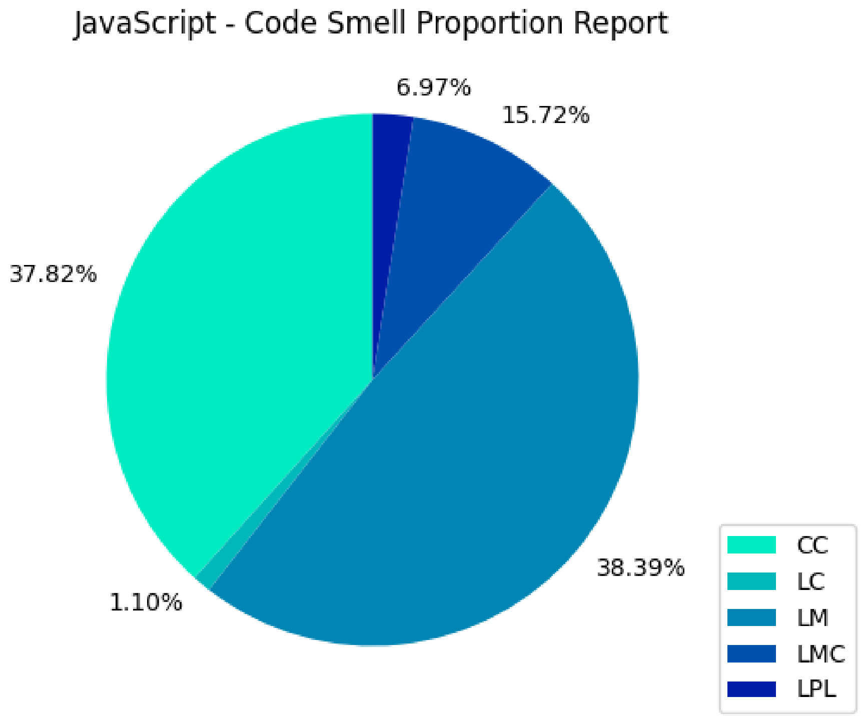


Figure 13: Code Smell Proportion Pie Chart in JavaScript

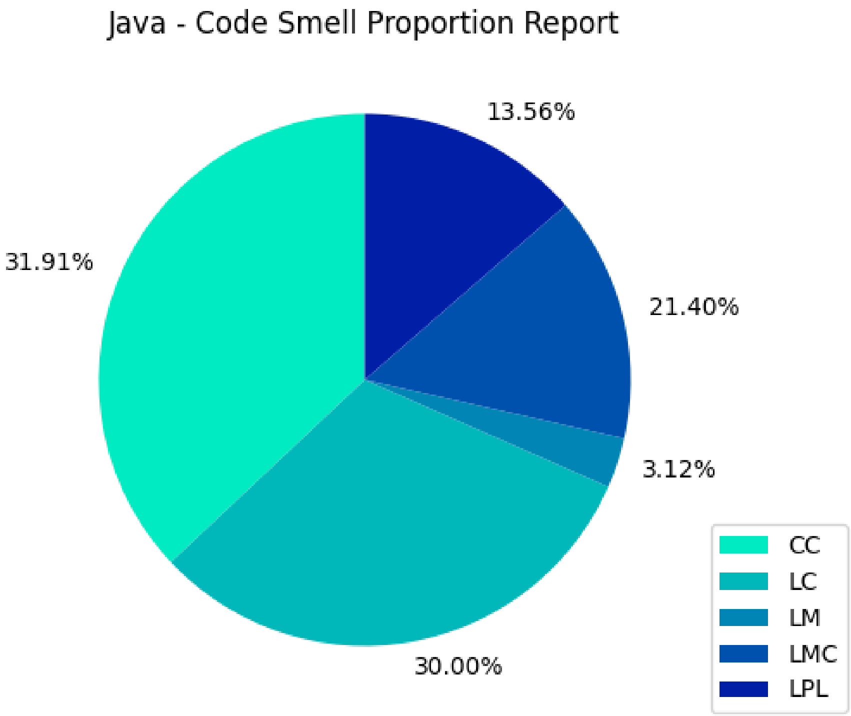


Figure 14: Code Smell Proportion Pie Chart in Java

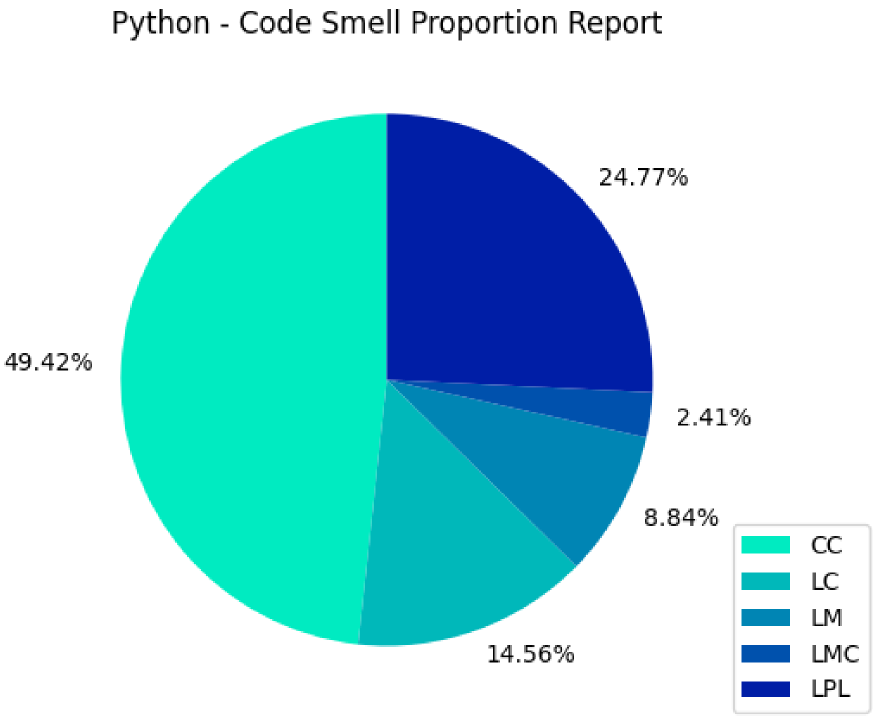


Figure 15: Code Smell Proportion Pie Chart in Python

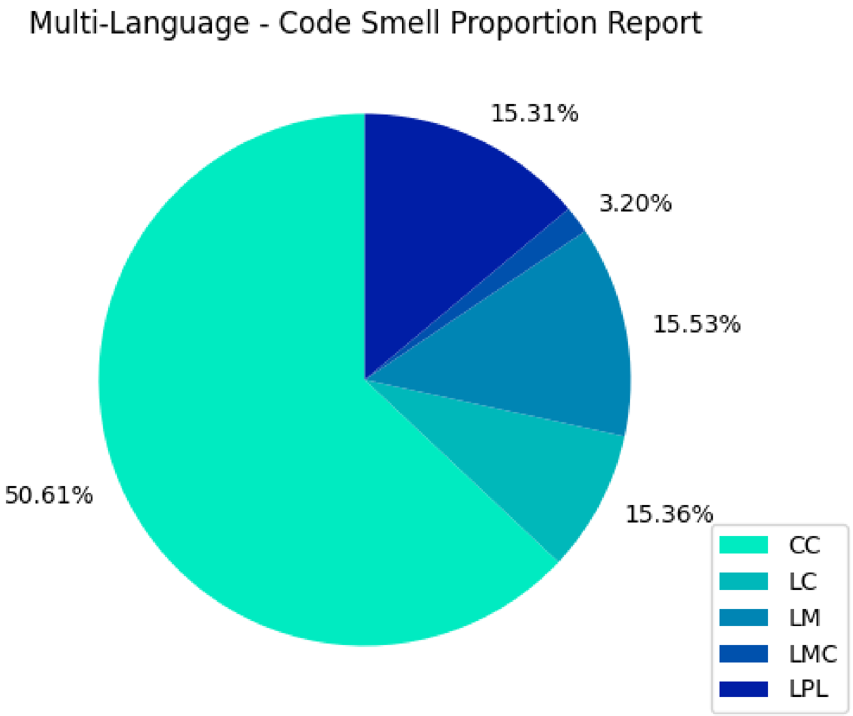


Figure 16: Code Smell Proportion Pie Chart in Multi-Language System

As shown above in the pie charts, there are some interesting phenomena that can answer RQ2. As we can see in the four pie charts, all 5 kinds of code smells exist at certain point in the open-source projects written in the selected languages. This result perfectly concludes that code smells happens regardless of the selection of programming language, which matches with our intuitions mentioned in the Chapter 1 where we claimed that code smells happen because of the bad decision on logic levels other than the syntax of programming languages.

Not only the results of this experiment confirms the existence of every kind of code smell in all 3 languages, they also present critical information on the distribution of code smells. One of the most significant characteristics is that there is a substantial portion of Complex Conditional code smell in all 4 charts. We guess there are 2 key reasons: 1. Based on our own daily coding experience, we think conditional statements, both If statements and Switch statements, play an important role on the decision making flow, and thus are widely used in software programs. 2. Complex Conditional isn't broadly recognized as a kind of code smell by programming community. We were caught attention by the exploratory survey done by Yamashita and Moonen [26]. They did a survey on the popularity of code smells, where Complex Conditional ranks #14, a extremely low rank compared with Long Method (#2) and Large Class (#4). As a result, both the popularity of conditional statements and the lack of attention on Complex Conditional lead it to be one of the most common code smells in the development of software projects.

While four charts have some characteristics in common, their differences are also interesting and prominent. Put your eyes on the proportion of Long Method and Long Class in charts and you will find the huge difference of the ratio of Long Class to Long Method to Long Class.

Table 10: Long Method to Long Class ratios

System	LM to LC
Java	0.1
JavaScript	34.9
Python	0.6
Multi-Language	1.01

We can see open-source projects in Java contains a large portion of Long Class code smell (30%) and yet a tiny portion of Long Method (3.12%), while Javascript presents the opposite phenomenon with 1.1% of Long Class and 38.39% of Long Method.

4.2.3 RQ3: Do some code smells occur more often than others in some languages?

To get the answers to this research question, the population of each code smell in the open-source project is needed to calculate the occurrence of each code smell. To achieve so, we implemented TreeNose int the open-source project listed in Table 9 in the last section. After collecting data and running the experiment, the results in the following bar charts represent the occurrence of each code smell per 1000 lines of code in each language system.

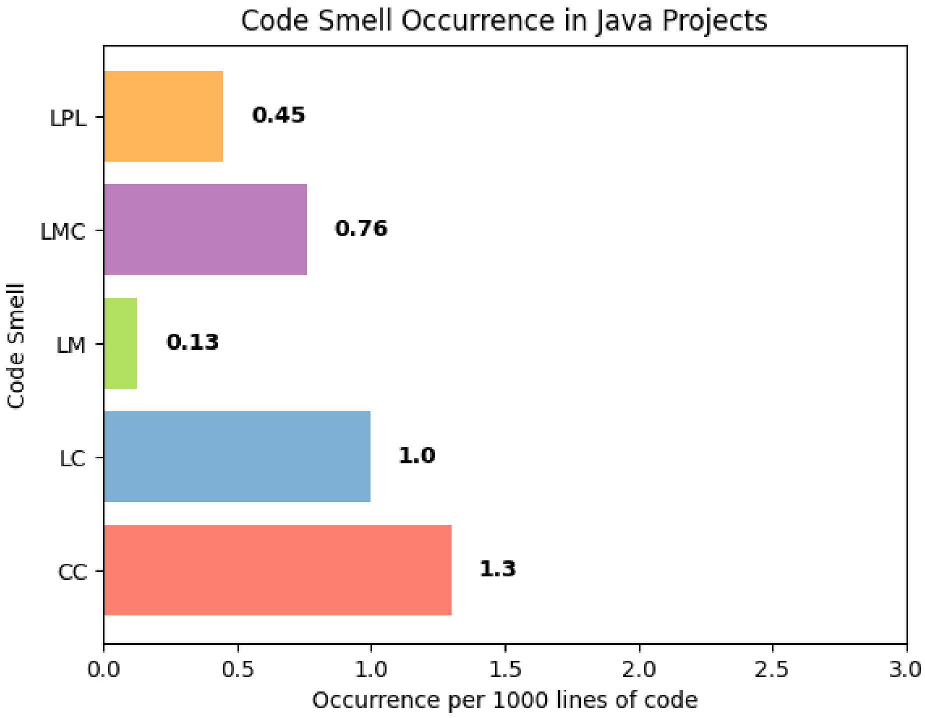


Figure 17: Code Smell Occurrence in Java projects

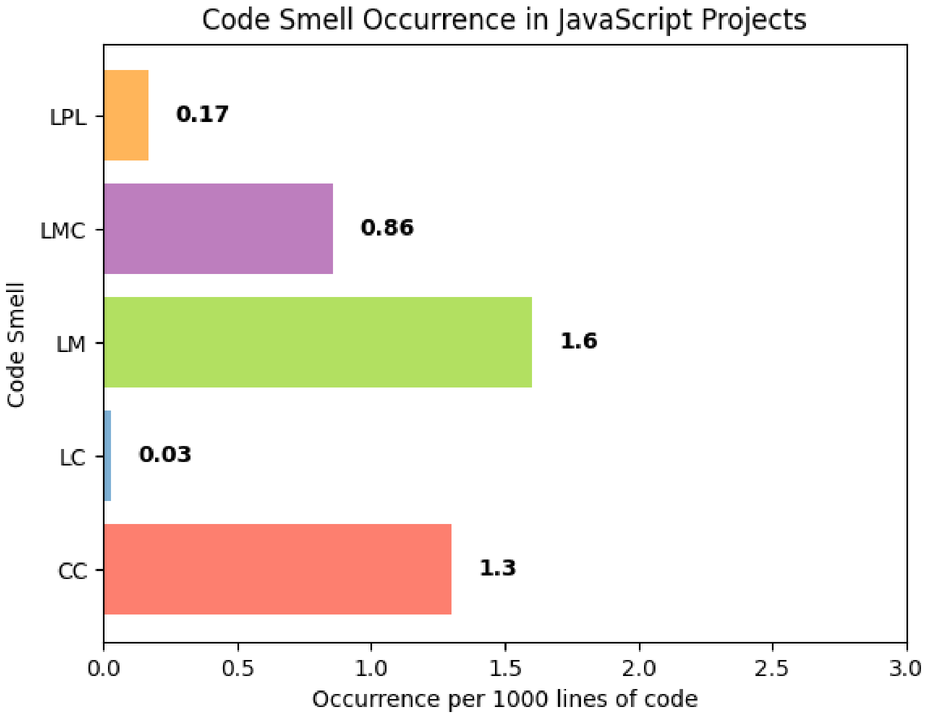


Figure 18: Code Smell Occurrence in JavaScript system

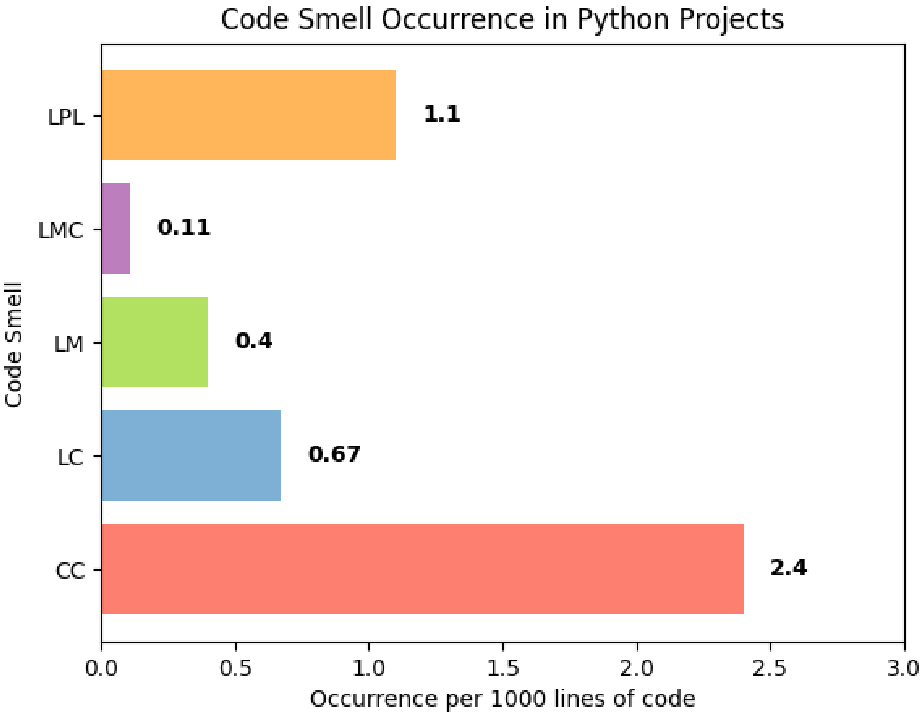


Figure 19: Code Smell Occurrence in Python system

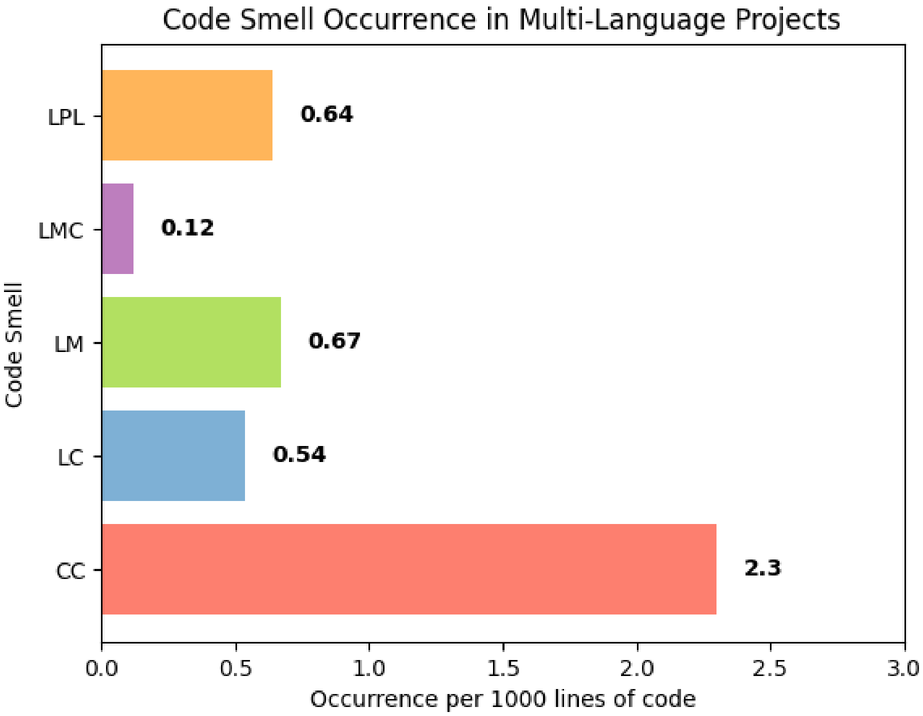


Figure 20: Code Smell Occurrence in Multi-Language system

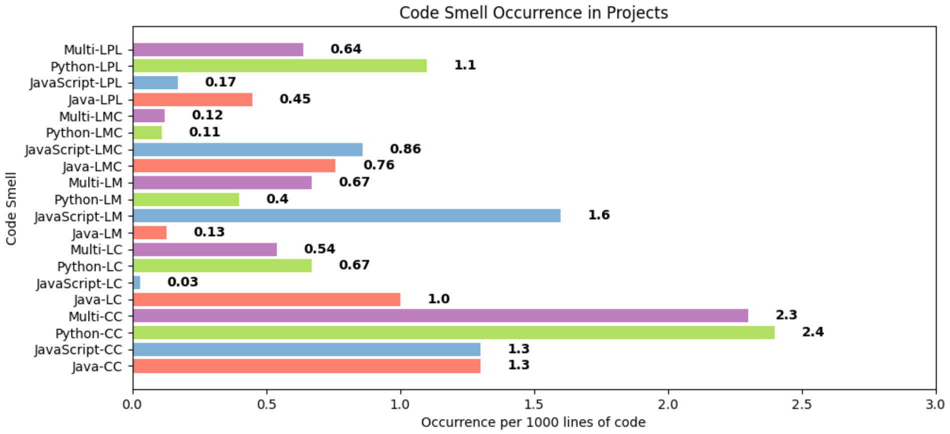


Figure 21: Code Smell Occurrence in every languages

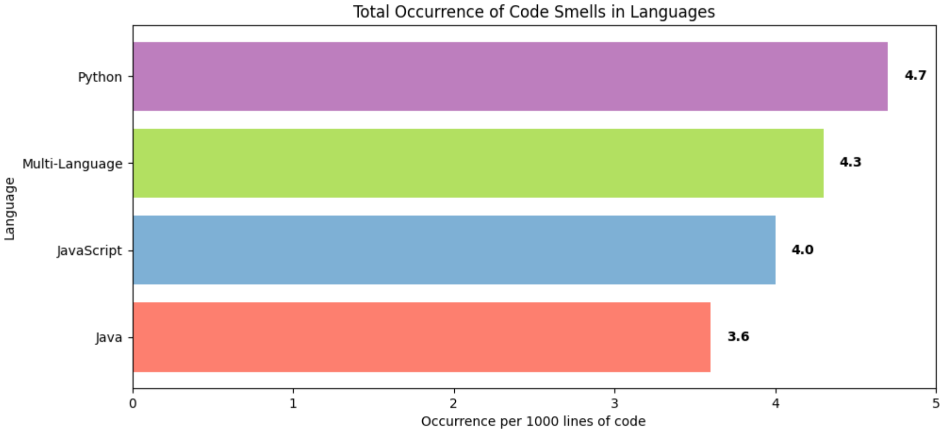


Figure 22: Total of Code Smell Occurrence inf every languages

Shown in the last chart, the language with highest occurrence, Python, has 30% higher occurrence than the language with the lowest, Java. When some people see this result, they may conclude that some programming languages are more smelly than others, and therefore the languages with high occurrences in this experiment, like Python, tend to be toxic and smelly.

To this kind of opinion, we don't disagree on it. Because it's true that systems in Python contains 3 more code smells whenever systems in Java reaches 10 code smells. However, it can only means Python is more smelly in the scope of certain selected code smells. While it's true that some code smells are more frequent in certain languages, those phenomena can't lead to the result that one language is more problem-prone than others when it comes to code smells.

To prove it, have a look in the second last chart where all code smells in every system are presented, one of the first thing catching our eyes is a significant amount of Complex Conditionals in Multi-Language system and Python. This phenomenon is aligned with the pie charts in the last section, where CC counts for 49.42% proportion of entire code smells in Python projects and 50.61% in Multi-Language projects, which are much more than its proportion in Java projects (31.91%) and JavaScript projects (37.82%). However, on the other side the spectrum, systems in Java are more problematic when it comes to Long Message Chain and Long Class.

After studying data, we draw the following conclusions: 1. Certain code smells are more prevalent in some languages than others 2. No language is significantly more problem-prone in the scope of code smell 3. Every language has its own advantages and disadvantages in the scope of code design

4.3 Threats to Validity

4.3.1 Experiment in RQ1

During the evaluation of TreeNose, our team manually checked the existence of code smells in the source code. The process of manual check is error-prone and the samples we selected to verify are small compared with the size of our input open-source projects and the size of code smells TreeNose. Due to those facts, the precision of TreeNose may vary in the real-world implementations.

Also, while we proved the high precision of TreeNose, we barely did any evaluation on the recall.

```
Recall = True Positive / (True Positive + False Negative)
```

In other words, it proved that high percentage of code smells collected by TreeNose are actually code smell, but didn't prove if TreeNose is able to capture all the code smells that need to be reported.

4.3.2 Experiment in RQ 2 and RQ 3

During the experiments on RQ 2 and PQ 3, we selected more than 3 widely-used open-source projects for each programming languages to minimize the bias and to get language-level code smell analysis. The bias is introduced by the code design style of single software project. However, the selection of software projects indeed affected the results. Therefore, we suggest a larger amount of open-source projects should be implemented as inputs in the future to further minimize the effects of sample bias.

5. Conclusions and Future Work

5.1 Summary of Results

After analyzing the experiment outcomes, we conducts the following 3 results

5.1.1 TreeNose Holds High Performance

After concluding the high precision of TreeNose in the experiment associated with RQ1, TreeNose proved to perform admirably in various open-source projects, making it an effective and trustworthy language-independent code smell detection tool. TreeNose proves the feasibility of extending code smell detection to multiple programming languages, just as assumed by Emden and Moonen when they built the first code smell detection tool ever [7]. During the evaluation of TreeNose, we compared TreeNose with other code smell tools and found TreeNose is able to detect more ranges of code smells and corner cases. It proves that TreeNose addresses the need for a high quality code smell detection tools in Java, JavaScript, and Python.

As to multi-language systems, TreeNose has the prominent advantages on them. Shown in the experiment of RQ1, different code smell detection tools implement various detection metrics. Implementing them concurrently in multi-language projects will embrace inconsistency of code quality in the program. TreeNose, on the other side of spectrum, is able to serve multi-language systems with a set of unified detection metrics.

5.1.2 Code Smells Exist in Every Language but in Different portions

The answers to RQ 2 ad RQ 3 plots some interesting characteristics of code smells. All types of code smells present in all three programming languages, concluding that code sells are able to be language-independent. This discovery again emphasizes the weight of a language-independent code smell detection tool.

Moreover, the experiment of RQ2 discloses the proportion of code smells. Some code smells like CC are critical problems in every system. The popularity of code smells in different systems raise the attentions. Anther prominent discovery in this experiment is the proportion differences across different languages. LC holds a huge proportion in Java but not in JavaScript. It suggests different programming languages have different tendency on code smells.

5.1.3 No Language is More Smell-prone than Others

Inspired by differences among programming languages on code smells, we conducted RQ3 ane its experiment to analyze the occurrence of code smells. Shown in the experiment result, no programming language is prominently more smell-prone than others. While some languages tend to be more problematic on certain code smells, in overall no language stands out. In summary, the selection of programming language only affects the occurrence of certain smells but not the overall occurrence. We think it may mean the selection of programming language doesn't affect the code quality of a software program.

5.2 Future Work

5.2.1 Extend the Detection Range of TreeNose

Currently TreeNose supports to detect 5 kinds of code smells in 3 programming languages. It's not even close to the limit of it. As shown in the Architecture section, TreeNose is designed to be configurable and extendable since its detector is language-independent. As a result, TreeNose has the ability to accept more programming languages as targets. During the process supporting more programming languages, users can verify if TreeNose is truly language-independent or not.

Moreover, TreeNose will support the detection of more types of code smells like Missing Default, Feature Envy, and Unused Variable. A wider range of code smell detection will further address the needs of high quality code smell detection tools in some programming languages. We wish TreeNose eventually can help the programming community to build a code smell detection standard.

5.2.2 User Study on Precision and Recall

Recall the Section 4.1, the entire experiment was designed to compare the precision of TreeNose with other tools. The result of the experiment discloses TreeNose holds high precision, however it captures no information about the recall, another important metric utilized to calculate the accuracy.

To calculate recall, we brainstormed two methods:

1. Run TreeNose in a set of pre-tagged code snippets
2. Manually analyze the source code in open-source projects and compare the results with TreeNose

The method 1 expects an at least medium size dataset with various code snippets with or without code smells. Comparing the results of TreeNose and the tags will verify the accuracy, including both precision and recall. The process of tagging will take labor force and require the knowledge on code smells. Furthermore, a team of professionals on programming are preferred to accomplish this process.

The method 2 requires a large range of manual code quality check on open-source projects. It’s safe to assume it’s a heavy labor and time-consuming task. However, we think it’s more effective than method 1. It’s easier to encounter different implementations of the same code smell in a large environment, assisting to explore the corner cases and unexpected scenarios. As a result, this method benefits TreeNose on handling corner cases. On the other hand, comparing the results with TreeNose reports in method 2 will also conduct the calculation of accuracy, similar to method 1.

5.3 Future Ethical Implications

5.3.1 Misuse of TreeNose in Software Development

As one of the future plans, we plan to deploy TreeNose on package manager so that users in the future can easily implant TreeNose into their software projects. As TreeNose implemented in software projects, there stands a chance that programmers may misuse it, leading their softwares or the development experiment to end unpleasantly.

TreeNose detects the code smells with metrics, whose results may be against the subjective judgement of programmers. TreeNose at most two techniques to detect one kind of code smell. The chosen technique may be just the shadows of code smells other than code smells themselves. Code snippets reported by TreeNose, because their metrics are over the thresholds, may be actually easy to read and maintain in real development experiments. Due to the fact that code smells are toxic because they hinder the readability and future maintenance, programmers should retain the code that suits programmers' experiences if there is a divergence between TreeNose and subjective experiences. In conclusion, default to the reports of TreeNose may potentially introduce the unnecessary refactoring ,or further lower the code quality of softwares. A judgement based on real experience is needed before the code snippets are refactored.

5.4 Conclusions

In this research, we claimed that code smells are language-independent ,and therefore a language-independent code smell detection is needed. TreeNose was built to address the need, an AST-based language-independent code smell detection tool. TreeNose was designed to detect 5 types of code smells across 3 programming languages: Java, JavaScript, and Python. We plotted the architecture of TreeNose and walked through its key components and key features. Following that, we compared TreeNose with 3 open-source language-exclusive code smell detection tools by implementing them into over 3 medium to large widely-used open-source projects in each supported language. To conclude, TreeNose is able to achieve 95% average precision on detection. In research on the second research question, we explored the presence and distribution of code smells in systems written in different languages.

Based on the output data, we drew the conclusions: 1. code smells happen regardless of the selection of languages 2. certain code smells like CC are prevalent in every system 3. languages tend to exhibit different code smells. Shown by the answers to RQ2, programming languages behave variously on code smells. So we conducted the RQ3 with its experiment to explore the popularity of code smells in each language system. We calculated the occurrence of each code smell out of 1000 lines and compare those occurrences among different languages. As a result, there are two significant patterns: 1. no programming language is significantly more smelly-prone than others 2. Some code smells are more prevalent in certain programming languages than in others. In summary, the results in those experiments proved code smells are language-independent and TreeNose provides an effective approach to detect them across multiple languages. TreeNose was proved to have high performance in code smell detection. After adding more language interfaces and smell detection supports, our team wish TreeNose can push the consistent code smell detection experience to a level programmers never have before, especially in the projects written in multiple programming languages.

References

[1] Mouna Abidi, Manel Grichi, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Code smells for multi-language systems. In *Proceedings of the 24th european conference on pattern languages of programs* (*EuroPLop '19*), 2019. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3361149.3361161

[2] Mouna Abidi, Md Saidur Rahman, Moses Openja, and Foutse Khomh. 2021. Are multi-language design smells fault-prone? An empirical study. *ACM Trans. Softw. Eng. Methodol.* 30, 3 (February 2021). https://doi.org/10.1145/3432690

[3] Max Brunsfeld. 2013. Tree-sitter.

[4] Zhifei Chen. 2016. Pysmell.

[5] Ward Cunningham. 1992. The WyCash portfolio management system. *SIGPLAN OOPS Mess.* 4, 2 (December 1992), 29–30. https://doi.org/10.1145/157710.157715

[6] Andreas Dangel. 2012. PMD. *GitHub.* Retrieved from https://github.com/pmd

[7] E. van Emden and L. Moonen. 2002. Java quality assurance by detecting code smells. In *Ninth working conference on reverse engineering, 2002. proceedings.*, 2002. 97–106. https://doi.org/10.1109/WCRE.2002.1173068

[8] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: Dentification and application of extract class refactorings. In *Proceedings of the 33rd international conference on software engineering* (*ICSE '11*), 2011. Association for Computing Machinery, New York, NY, USA, 1037–1039. https://doi.org/10.1145/1985793.1985989

[9] Martin Fowler. 2007. Bliki: Design stamina hypothesis. *martinfowler.com.* Retrieved from https://martinfowler.com/bliki/DesignStaminaHypothesis.html

[10] Martin Fowler and Kent Beck. 1999. Bad smells in code. In *Refactoring: Improving the design of existing code.* Addison-Wesley, Addison Wesley longman, Inc.

[11] Marcel Jerzyk and Lech Madeyski. 2023. Code smells: A comprehensive online catalog and taxonomy. In *Developments in information and knowledge management systems for business applications: Volume 7*, Natalia Kryvinska, Michal Greguš and Solomiia Fedushko (eds.). Springer Nature Switzerland, Cham, 543–576. https://doi.org/10.1007/978-3-031-25695-0_24

[12] Jochen Kreimer. 2005. Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science* 141, 4 (2005), 117–136. https://doi.org/https://doi.org/10.1016/j.entcs.2005.02.059

[13] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167, (2020), 110610. https://doi.org/https://doi.org/10.1016/j.jss.2020.110610

[14] Xinghua Liu and Cheng Zhang. 2018. DT: An upgraded detection tool to automatically detect two kinds of code smell: Duplicated code and feature envy. In *Proceedings of the international conference on geoinformatics and data analysis* (*ICGDA '18*), 2018. Association for Computing Machinery, New York, NY, USA, 6–12. https://doi.org/10.1145/3220228.3220245

[15] Robert Cecil Martin. 2009. *Clean code: A handbook of agile software craftsmanship.* Prentice Hall.

[16] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (2017), 7. https://doi.org/10.1186/s40411-017-0041-1

[17] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A textual-based technique for smell detection. In *2016 IEEE 24th international conference on program comprehension (ICPC)*, 2016. 1–10. https://doi.org/10.1109/ICPC.2016.7503704

[18] Ralph Peters and Andy Zaidman. 2012. Evaluating the lifespan of code smells using software repository mining. In *2012 16th european conference on software maintenance and reengineering*, 2012. 411–416. https://doi.org/10.1109/CSMR.2012.79

[19] Quellish. 2015. How on earth the facebook IOS application is so large. *Tumblr*. Retrieved from https://quellish.tumblr.com/post/126712999812/how-on-earth-the-facebook-ios-application-is-so

[20] Rana Sandouka and Hamoud Aljamaan. 2023. Python code smells detection using conventional machine learning models. *PeerJ Computer Science*. Retrieved from https://peerj.com/articles/cs-1370/

[21] Tushar Sharma. 2021. DesigniteJava.

[22] Sarchen Starke. 2021. Jscent.

[23] Yuan Tian, David Lo, and Chengnian Sun. 2012. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *2012 19th working conference on reverse engineering*, 2012. 215–224. https://doi.org/10.1109/WCRE.2012.31

[24] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. 1999. 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*), 1999. Association for Computing Machinery, New York, NY, USA, 47–56. https://doi.org/10.1145/320384.320389

[25] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE international conference on software engineering*, 2015. 403–414. https://doi.org/10.1109/ICSE.2015.59

[26] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *2013 20th working conference on reverse engineering (WCRE)*, 2013. 242–251. https://doi.org/10.1109/WCRE.2013.6671299