Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study

Giammaria Giordano, Giusy Annunziata, Andrea De Lucia, and Fabio Palomba

University of Salerno, Italy

IWSM MENSURA
September 15, 2023
Rome, Italy

giagiordano@unisa.it

SCAN ME!
I'm the paper

Background

The continuous **Change Requests** and the stringent **time-to-market force** developers t**o release immature products**, **putting aside best practices** to decrease the delivery time

Code Smells

A **symptom** of **poor design** that **can lead** to **increased effort** during both **maintenance** and **evolution activities**

State-of-the-art

JSpIRIT: A Flexible Tool for the Analysis of Code Smells

Santiago Vidal*, Hernán Vázquez*, J. Andrés Díaz-Pace*, Claudia Marcos†
ISISTAN Research Institute, UNICEN, Argentina
Email: {svidal, hvazquez, adiaz, cmarcos}@exa.unicen.edu.ar
*CONICET, Argentina
†CIC-Buenos Aires, Argentina

Alessandro Garcia, Willian Oizumi
PUC-Rio, Brazil
Email: {afgarcia, woizumi}@inf.puc-rio.br

Abstract—Code smells are a popular mechanism to identify structural design problems in software systems. Since it is generally not feasible to fix all the smells arising in the code, some of them are often postponed by developers to be resolved in the future. One reason for this decision is that the improvement of the code structure, to achieve modifiability goals, requires extra effort from developers. Therefore, they might not always spend this additional effort, particularly when they are focused on deliv... smell... the co... modif... tools ... the m... we pr... of coc... smell... smell... illustr... tool. ... be ali...

of design problems [4], [5], [6], [7]. Therefore, the occurrence of smell agglomerations indicates that the technical debt is increasing. A number of tools have been proposed for detecting single instances of code smells (but not agglomerations) [8], [9], [10]. Once detected, the smells can be fixed through a number of refactoring strategies [3].

In an ideal world, fixing all smells would "pay" much of

Java Quality Assurance by Detecting Code Smells

Eva van Emden*
CWI
The Netherlands
http://www.cwi.nl/~eva/
eva@cwi.nl

Leon Moonen‡
CWI
The Netherlands
http://www.cwi.nl/~leon/
leon@cwi.nl

Abstract

Software inspection is a known technique for improving software quality. It involves carefully examining the code, the design, and the documentation of software and checking these for aspects that are known to be potentially problematic based on past experience.

Code smells are a metaphor to describe patterns that are generally associated with bad design and bad programming practices. Originally, code smells are used to find the places in software that could benefit from refactoring. In this paper, we investigate how the quality of code can be automatically assessed by checking for the presence of code smells and how this approach can contribute to automatic code inspection.

We present an approach for the automatic detection and visualization of code smells and discuss how this approach can be used in the design of a software inspection tool. We illustrate the feasibility of our approach with the development of jCosmo*, a prototype code smell browser that detects and visualizes code smells in* Java *source code. Finally, we show how this tool was applied in a case study.*

Keywords: *software inspection, quality assurance, Java, refactoring, code smells.*

1. Introduction

Software inspection is a known technique for improving software quality. It was first introduced in 1976 by Fagan [10] and has since been reported on by numerous others, for example [18, 13]. Software inspection involves carefully examining the code, the design, and the documentation of software and checking these for aspects that are known to be potentially problematic based on past experience.

It is generally accepted that the cost of repairing a bug is much lower when that bug is found early in the development cycle. One of the advantages of software inspection is that the software is analysed *before* it is tested. Thus, potential problems are identified in the beginning of the cycle so that they can be solved early, when it's still cheap to fix them.

Traditionally, software inspection is a formal process that involves labor-intensive manual analysis techniques such as formal code reviews and structured walk-throughs. Inspection is a systematic and disciplined process that is guided by well-defined rules. These strict requirements often backfire, resulting in code inspections that are not performed well or sometimes even not performed at all.

These problems are addressed by tools that automate the software inspection process. We distinguish two approaches:

1. Tools that *automate* the inspection *process*, making it easier to follow the guidelines and record the results.
2. Tools that perform *automatic code inspection*, relieving the programmers of the manual inspection burden.

We concentrate on the second type: tools that perform automatic inspection. Such tools are interesting since automatic inspection and reporting on the code's quality and conformance to coding standards allows early (and repeated) detection of signs of project deterioration. Early feedback enables early corrections, thereby lowering the development costs and increasing the chances for success.

1.1. Code smells

The existing tools that support automatic code inspection (for example, the well-known C analyzer Lint [15]) tend to focus on improving code quality from a technical perspective. The fewer bugs (or defects) there are present in a piece of code, the higher the quality of that code. From this perspective, code inspection boils down to low-level bug-chasing and we see this reflected in the tools which typically look for problems with pointer arithmetic, memory (de)allocation, `null` references, array bounds errors, etc.

In this paper, we will focus on a different aspect of code quality: Inspired by the metaphor of "*code smells*" introduced in the refactoring book [12], we review the code for problems that are generally associated with bad program

* Visiting research assistant from the Rigi Group, Computer Science Department, University of Victoria, Victoria, B.C., Canada.
‡ Corresponding author.

Lightweight Detection of Android-Specific Code Smells: The aDoctor Project

Fabio Palomba[1,2], Dario Di Nucci[2], Annibale Panichella[3], Andy Zaidman[1], Andrea De Lucia[2]
[1]Delft University of Technology — [2]University of Salerno — [3]University of Luxembourg
f.palomba@tudelft.nl, ddinucci@unisa.it, annibale.panichella@uni.lu, a.e.zaidman@tudelft.nl, adelucia@unisa.it

***Abstract*—Code smells are symptoms of poor design solutions applied by programmers during the development of software systems. While the research community devoted a lot of effort to studying and devising approaches for detecting the traditional code smells defined by Fowler, little knowledge and support is available for an emerging category of Mobile app code smells. Recently, Reimann *et al.* proposed a new catalogue of Android-specific code smells that may be a threat for the maintainability and the efficiency of Android applications. However, current tools working in the context of Mobile apps provide limited support and, more importantly, are not available for developers interested in monitoring the quality of their apps. To overcome these limitations, we propose a fully automated tool, coined aDoctor, able to identify 15 Android-specific code smells from the catalogue by Reimann *et al.* An empirical study conducted on**

defined by Reimann *et al.* [18]. These Android-specific smells may threat several non-functional attributes of mobile apps, such as security, data integrity, and source code quality [18]. As highlighted by Hetch *et al.* [19], these type of smells can also lead to performance issues.

The aforementioned reasons highlight the need of having specialized detectors that identify code smells in Mobile apps. Hetch *et al.* [20] first faced the problem by devising Paprika, a code smell detector for Android apps. However, the tool is able to detect a limited number of the Android-specific code smells defined by Reimann *et al.* (just 4 out of the total 30), and is not publicly available.

On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2]
Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, fferrucci@unisa.it, gravino@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.,* Java, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. These latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on Java, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Bequest* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether

DECOR: A Method for the Specification and Detection of Code and Design Smells

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur

Abstract—Code and design smells are poor solutions to recurring implementation and design problems. They may hinder the evolution of a system by making it hard for software engineers to carry out changes. We propose three contributions to the research field related to code and design smells: 1) DECOR, a method that embodies and defines all the steps necessary for the specification and detection of code and design smells, 2) DETEX, a detection technique that instantiates this method, and 3) an empirical validation in terms of precision and recall of DETEX. The originality of DETEX stems from the ability for software engineers to specify smells at a high level of abstraction using a consistent vocabulary and domain-specific language for automatically generating detection algorithms. Using DETEX, we specify four well-known design smells: the antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and their 15 underlying code smells, and we automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on XERCES v2.7.0, and discuss the precision of these algorithms on 11 open-source systems.

Index Terms—Antipatterns, design smells, code smells, specification, metamodeling, detection, Java.

Investigating the evolution of code smells in object-oriented systems

Alexander Chatzigeorgiou · Anastasios Manakos

Received: 29 June 2011 / Accepted: 6 April 2013 / Published online: 21 April 2013

Abstract Software design problems are known and perceived under many different terms, such as code smells, flaws, non-compliance to design principles, violation of heuristics, excessive metric values and anti-patterns, signifying the importance of handling them in the construction and maintenance of software. Once a design problem is identified, it can be removed by applying an appropriate refactoring, improving in most cases several aspects of quality such as maintainability, comprehensibility and reusability. This paper, taking advantage of recent advances and tools in the identification of non-trivial code smells, explores the presence and evolution of such problems by analyzing past versions of code. Several interesting questions can be investigated such as whether the number of problems increases with the passage of software generations, whether problems vanish by time or only by targeted human intervention, whether code smells occur in the course of evolution of a module or exist right from the beginning and whether refactorings targeting at smell removal are frequent. In contrast to previous studies that investigate the application of refactorings in the history of a software project, we attempt to analyze the evolution from the point of view of the problems themselves. To this end, we classify smell evolution patterns distinguishing deliberate maintenance activities from the removal of design problems as a side effect of software evolution. Results are discussed for two open-source systems and four code smells.

Keywords Code smell · Refactoring · Software repositories · Software history · Evolution

A. Chatzigeorgiou (✉) · A. Manakos
Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
e-mail: achat@uom.gr

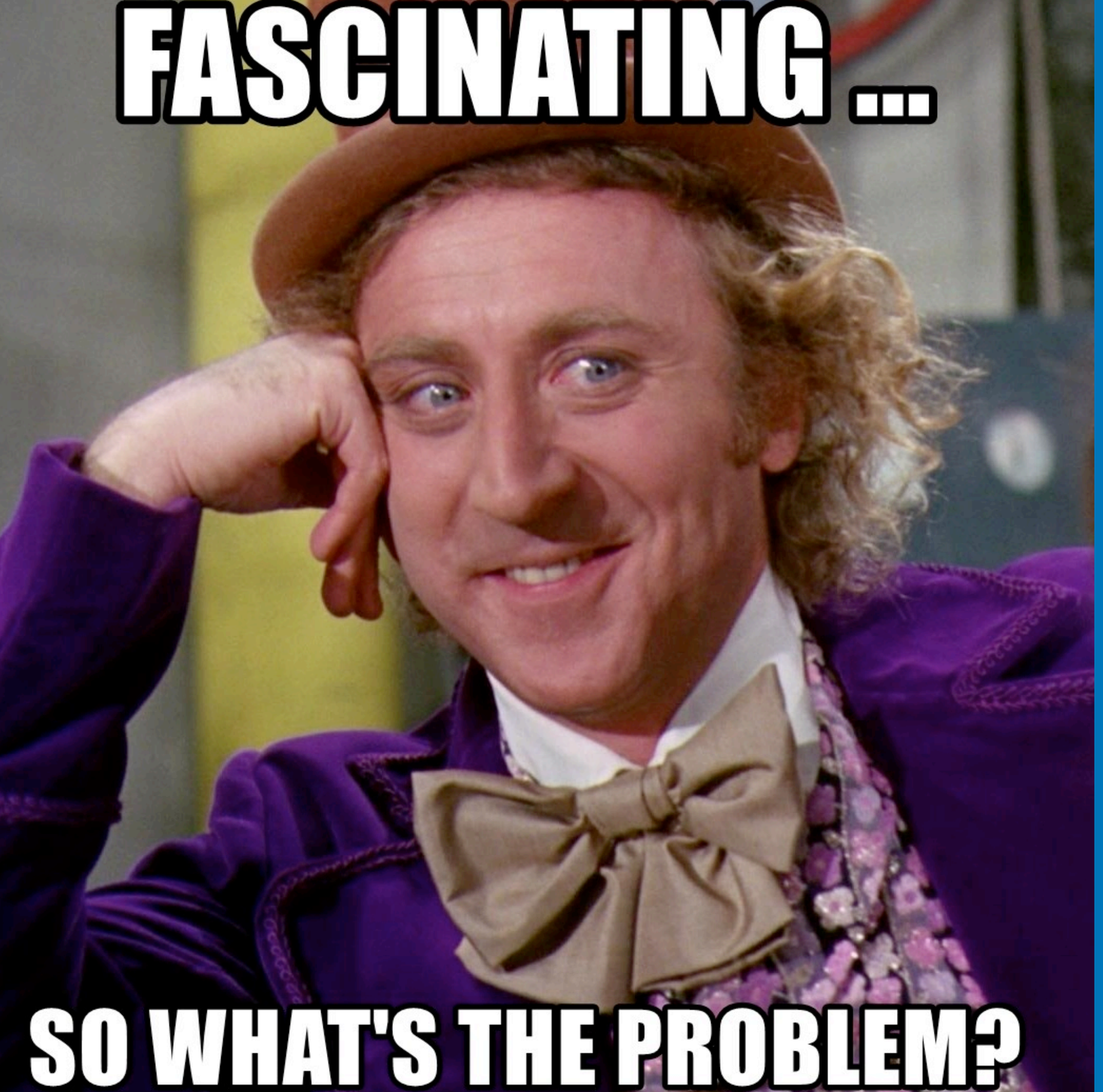
A. Manakos

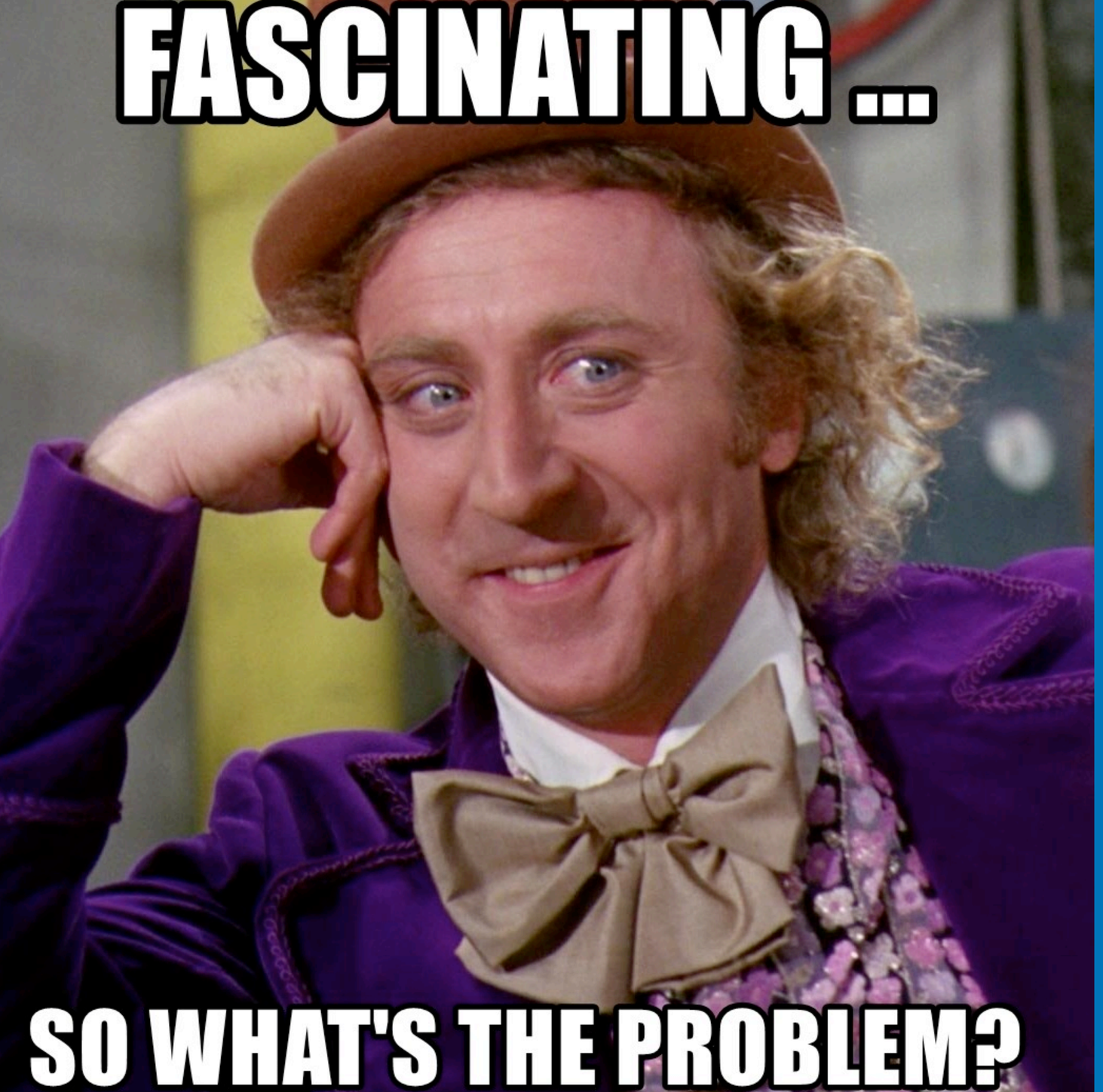
1 Introduction

The design of software systems can exhibit several problems which can be either due to inefficient analysis and design during the initial construction of the software or more often, due to software ageing, where software quality degenerates over time [27]. Declining quality of evolving systems is also something that is expected according to Lehman's 7th law of software evolution [18]. The importance that the software engineering community places on the detection and resolution of design problems is evident from the multitude of terms under which they are known. Some researchers view problems as non-compliance with design principles [20], violations of design heuristics [29], excessive metric values, lack of design patterns [12] or even application of anti-patterns [3].

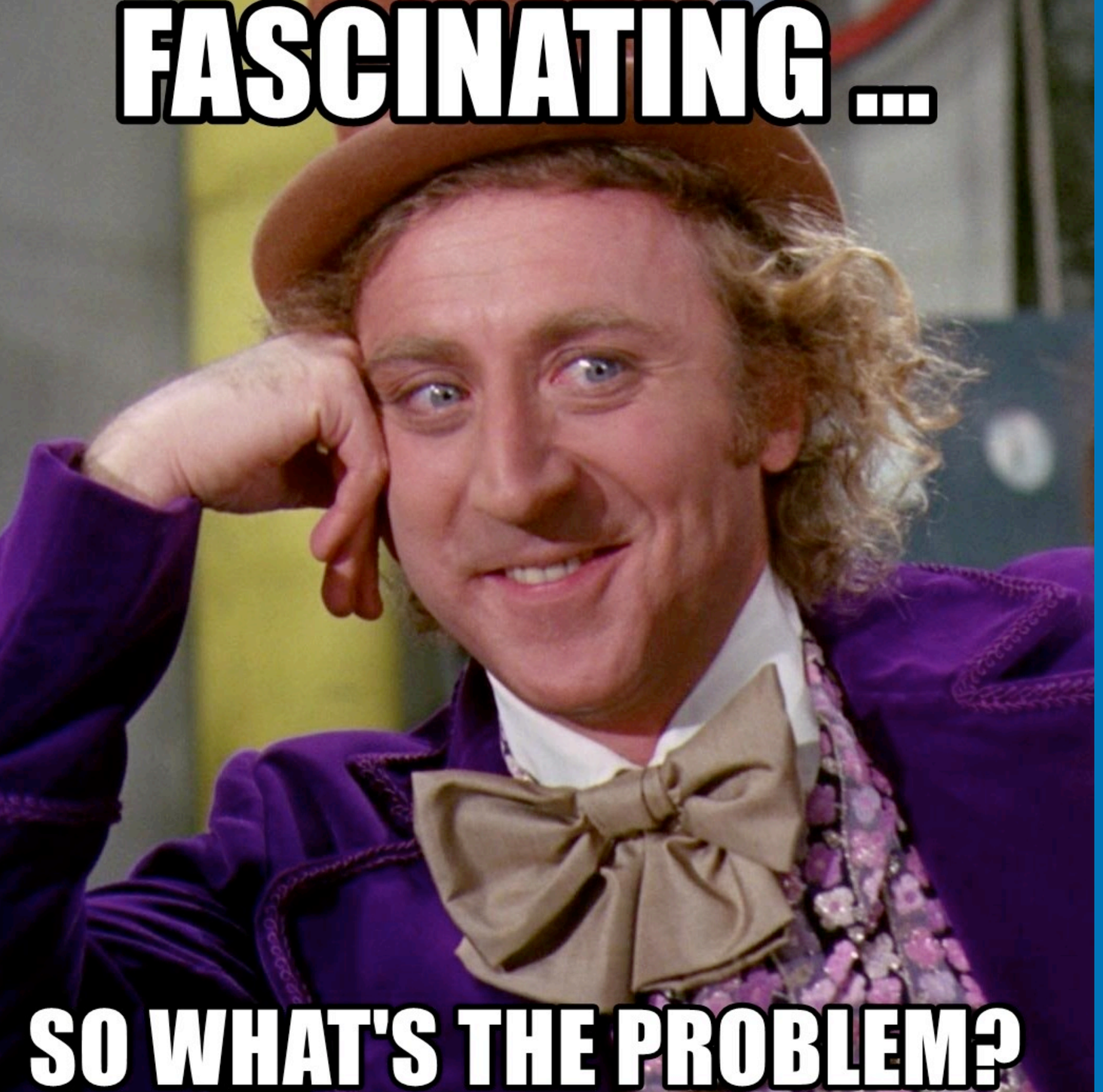
According to Fowler [11], design problems appear as "bad smells" at code or design level and the process of removing them consists in the application of an appropriate refactoring, i.e. an improvement in software structure without any modification of its behavior. Refactorings have been widely acknowledged mainly because of their simplicity which allows the automation of their application. Moreover, despite their simplicity, the cumulative effect of successive refactorings on design quality can be significant. Their popularity is also evident from the availability of numerous tools that provide support for the application of refactorings relieving the designers from the burden of their mechanics [24].

According to the recommendations proposed by Lehman and Ramil for software evolution planning [18], quality should be continuously monitored as systems evolve. This implies that past versions of a software system should be analyzed to track changes in evolutionary trends. To this end,


FASCINATING ...
SO WHAT'S THE PROBLEM?



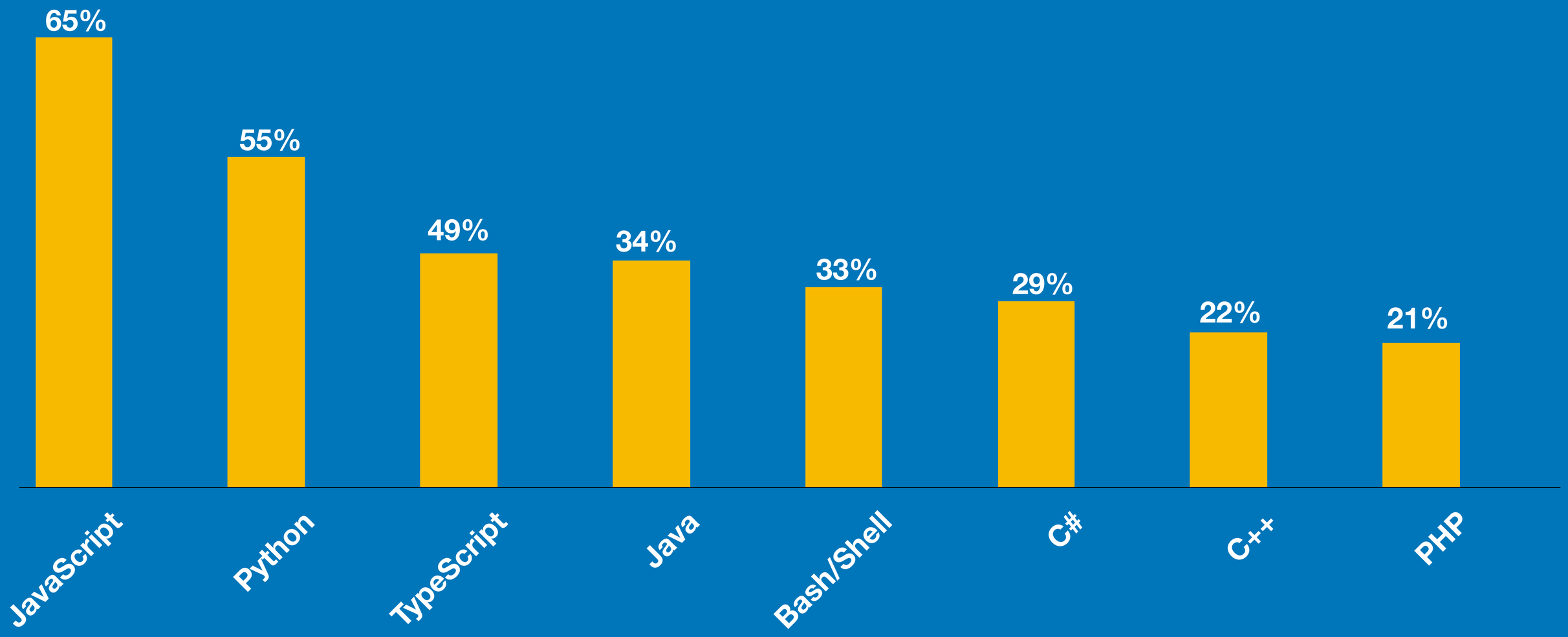
Most **previous work** only considered **Java systems**!



Most **previous work** only considered **Java systems**!

Java is **not** the **most used** programming language **nowadays!**

Most Used Programming Languages in 2022



https://survey.stackoverflow.co/2022#most-used-technologies-language

Why this change of direction?

Why this change of direction?

Developers tend to use **flexible programming languages** to **combine multiple paradigms**

Why this change of direction?

Developers tend to use **flexible programming languages** to **combine multiple paradigms**

Programming languages like **Python** can be easily used to build **Artificial Intelligence-Enabled Systems**

Why this change of direction?

Developers tend to use **flexible programming languages** to **combine multiple paradigms**

The core of most **Artificial Intelligence-Enabled Systems** (e.g., **ChatGPT**) is written in **Python**!

used to build **Artificial Intelligence-Enabled Systems**

State-of-the-Art

Does Python Smell Like Java?

Tool Support for Design Defect Discovery in Python

Nicole Vavrová[a] and Vadim Zaytsev[a,b]

a Universiteit van Amsterdam, The Netherlands

b Raincode Labs, Belgium

Abstract

The *context* of this work is specification, detection and ultimately removal of harmful code that are associated with defects in design and implementation of software. In partic five code smells and four antipatterns previously defined in literature. Our *inquiry* is ab nine design defects in source code written in Python programming language, which is sub from all prior research, most of which concerns Java and other C-like languages. Our *ap* software engineers: we have processed existing research literature on the topic, extracte definitions of defects and their concrete implementation specifications, programmed the let it loose on a huge test set obtained from open source code from thousands of GitH it comes to *knowledge*, we have found that more than twice as many methods in Python too long (statistically extremely longer than their neighbours within the same project) long parameter lists are seven times less likely to be found in Python code than in Java found that Functional Decomposition, the way it was defined for Java, is not found in Pyth Spaghetti Code and God Classes are extremely rare there as well. The *grounding* and the results comes from the fact that we have performed our experiments on 32,058,823 lines of is by far the largest test set for a freely available Python parser. We have also designed the a way that it aligned with prior research on design defect detection in Java in order to eas we treat our own actions as a partial replication. Thus, the *importance* of the work is both Python grammar of highest quality, applied to millions of lines of code, and in the desig tool which works on something else than Java.

ACM CCS 2012

- **Software and its engineering → Parsers; Software defect analysis; Patterns;**

Keywords Python, code smells, antipatterns, design defects, parsing, software analysis

Understanding metric-based detectable smells in Python software: A comparative study

Chen Zhifei[a], Chen Lin[*,a], Ma Wanwangying[a], Zhou Xiaoyu[b], Zhou Yuming[a], Xu Baowen[*,a]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China

[b] School of Computer Science and Engineering, Southeast University, Nanjing 210096, China

ARTICLE INFO

Keywords:
Python
Code smell
Detection strategy
Software maintainability

ABSTRACT

Context: Code smells are supposed to cause potential comprehension and maint development. Although code smells are studied in many languages, e.g. Java and C or tool support addressing code smells in Python.

Objective: Due to the great differences between Python and static languages, the g and detect code smells in Python programs and to explore the effects of Python s ability.

Method: In this paper, we introduced ten code smells and established a metric-b three different filtering strategies to specify metric thresholds (Experience-Bas Strategy, and Tuning Machine Strategy). Then, we performed a comparative stu detection strategies perform in detecting Python smells and how these smells aff with different detection strategies. This study utilized a corpus of 106 Python projec

Results: The results showed that: (1) the metric-based detection approach perfor smells and Tuning Machine Strategy achieves the best accuracy; (2) the three detec different smell occurrences, and Long Parameter List and Long Method are more pr several kinds of code smells are more significantly related to changes or faults in

Conclusion: These findings reveal the key features of Python smells and also provide detection strategy in detecting and analyzing Python smells.

1. Introduction

Code smells [2,14] are particular bad patterns in source code which violate important principles of software design and implementation issues. Particularly, code smells indicate when and what refactoring can be applied [38,43–44]. It does not mean that no code smells are allowed to appear, but rather that code smells are essential hints about beneficial refactoring. Various studies have confirmed the effects of code smells on different maintainability related aspects [54,61,62], especially changes [4–6,57–58], effort [7–9], modularity [55], comprehensibility [10,11], and defects [12,46,56–58].

Existing approaches of detecting code smells include metric-based [1,16–18,26], machine learning [19–21], history-based [22–23], textual-based [60], and search-based [41] approaches. A large group of code smells can be measured by software metrics to quantify their characteristics, hence metric-based detection technique becomes the most common way of detecting code smells. Measuring code smells requires proper quantification means of desig which raises a set of challenge. Above all, m gineer mostly clueless concerning the ultim that it indicts. Credible thresholds are establis metrics as an effective measurement instrum challenge for metric-based detection techniq

Python [25] is a typical dynamic scripti remarkably simple and expressive syntax. To both a small and large scale, Python replaces with short constructs, which allow Python concepts in fewer lines of code than would be In spite of multiple advantages of these cons tures also bring in serious difficulty in progra code in Python reduces readability and abus fected by bad patterns [30–31]. As a result, c occur in Python programs.

Due to the great differences between Pyt

Python Code Smell Detection Using Machine Learning

Natthida Vatanapakorn, Chitsutha Soomlek and Pusadee Seresangtakul

Department of Computer Science, College of Computing, Khon Kaen University

Khon Kaen, Thailand

Email: natthida.w@kkumail.com, chitsutha@kku.ac.th, pusadee@kku.ac.th

***Abstract*—Python is an increasingly popular programming language used in various software projects and domains. Code smells in Python significantly influences the maintainability, understandability, testability issues. This paper proposes a machine learning-based code smell detection for Python programs. We trained eight machine learning models with a dataset based on 115 open-source Python projects, 39 class-level software metrics, and 22 function-level software metrics. We intended to identify five code smell types in both class and function levels, i.e., *long method*, *long parameter list*, *large class long scope chaining*, and *long based class list*. Correlation-based feature selection (CFS) and logistic regression-forward stepwise (conditional) selection were employed to improve the performance of the model. This research concluded with an empirical evaluation of the performance of the machine learning approaches against the tuning machine method. The results show that the machine learning method achieved 99.72% accuracy when identifying *long method* and *long base class list*. The machine learning-based code smell detection also outperformed the tuning machine method. Moreover, we also found a set of high-impact features that contributed most when identifying each type of code smell.**

***Index Terms*—code smells, machine learning, Python**

I. Introduction

Software maintenance is critical in software development as software products are constantly changed to support user requirements and new technology [1]. Therefore, software maintenance will end after the software reaches its 'end of life'. A common problem in software maintenance that developers face is poorly-design code. As a result, the source code is difficult to understand and hard to maintain. Martin Fowler [2] defined code smells as a problem caused by design flaws.

Researchers have proposed various techniques to detect code smells in a program [3]–[8]. Machine learning is a widely used technique for detecting code smells with promising results [4], [9]–[14]. For examples, [4] and [9] demonstrated that the random forest technique outperformed other classifiers in detecting code smells. Furthermore, most of the studies have used open-source software in the Java programming language [10], whereas more recent and modern software projects tend to incline towards Python [15]. Software projects in Python also suffer from maintenance issues. As such, code smell detection for the Python programming language will play an important role in mitigating problems.

Compared to Java, there are a very limited number of studies on code smell detection in Python projects. In recent works relative to the research problem, we refer the readers to [16]–[19]. Code smell detection is a complex task, due to the lack of common definitions and subjectivity issues. Manual detection by human experts requires a lot of effort and is time-consuming and expensive.

In this research, we employ machine learning techniques to capture the human perspectives on code smells and automatically identify code smells in Python projects. We evaluated the performance of eight supervised machine learning techniques; decision tree [20], gradient boosted trees [21], random forest [22], support vector machine [23], k-nearest neighbors [24], logistic regression [25], multilayer perceptron [26], and naive bayes [27], when identifying *long method*, *long parameter list*, *large class*, *long base class list*, and *long scope chaining*. In addition, we compared the results produced by our best models to that of the tuning machine method [17]. We also investigated the high impact features that contributed most in this research context.

The main contributions of our research are as follows:

- We proposed eight machine learning models for detecting five types of code smells in Python and their performance evaluation results. We also compared the results with those from the tuning machine method [17].
- We created a dataset by collecting 61 software metrics in class and function levels from 115 Python open-source projects using Python static code analysis tools; Pysmell [17], Understand (SciTools) [28], Cohesion [29], and our handcraft program.
- We applied feature selection techniques, like correlation-based feature selection (CFS) and logistic regression-forward stepwise (LRFS) to find the most relevant features for each type of code smell.

We made the dataset and source code publicly available at https://github.com/NatthidaW/pythoncodesmell to support the research community with reproducible content.

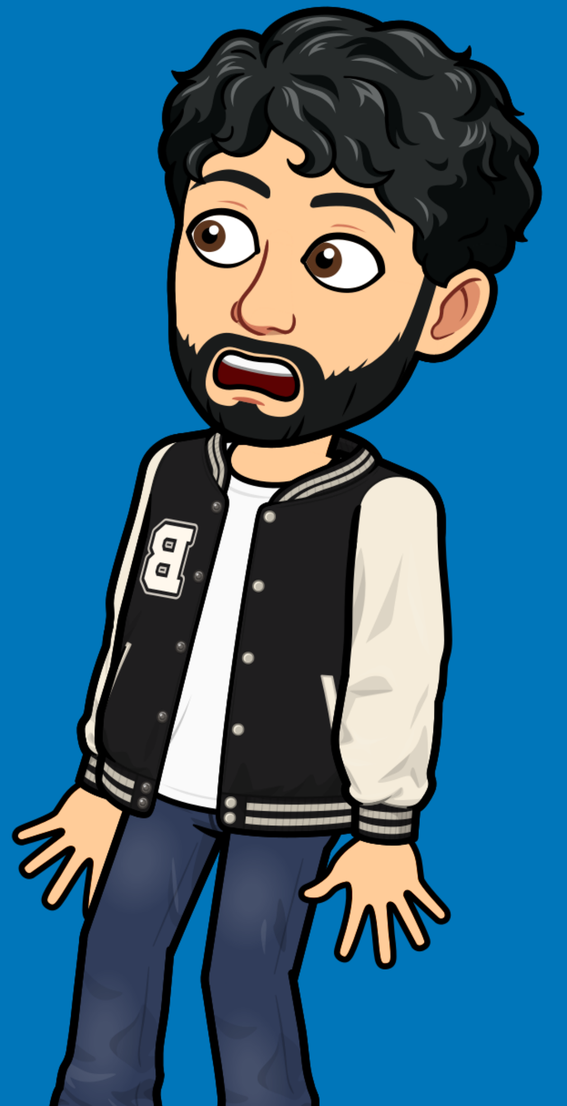
II. Related Work

Many researchers have applied machine learning techniques to code smell detection with promising results [30]–[34]. Examples of machine learning techniques used to identify code smells are the decision tree [31]–[34], random forest [30], [33], [34], logistic regression [34], and ensemble methods [35].

Limitations

Previous work focused only on **Traditional Systems!**

Previous work did not consider **AI-Enabled Systems** from an **evolutionary standpoint!**



PySmell

Detecting Code Smells in Python Programs

Zhifei Chen[1], Lin Chen[2+], Wanwangying Ma[3], Baowen Xu[4+]
State key Laboratory of Novel Software Technology
Nanjing University
Nanjing, China
{[1]chenzhifei, [3]wwyma}@smail.nju.edu.cn, {[2]lchen, [4]bwxu}@nju.edu.cn

***Abstract*—As a traditional dynamic language, Python is increasingly used in various software engineering tasks. However, due to its flexibility and dynamism, Python is a particularly challenging language to write code in and maintain. Consequently, Python programs contain code smells which indicate potential comprehension and maintenance problems. With the aim of supporting refactoring strategies to enhance maintainability, this paper describes how to detect code smells in Python programs. We introduce 11 Python smells and describe the detection strategy. We also implement a smell detection tool named Pysmell and use it to identify code smells in five real world Python systems. The results show that Pysmell can detect 285 code smell instances in total with the average precision of 97.7%. It reveals that Large Class and Large Method are most prevalent. Our experiment also implies Python programs may be suffering code smells further.**

Keywords*—Python; code smells; program maintenance; refactoring

I. Introduction

The notion of code smells was introduced initially by Fowler [3], who presented 22 code smells and associated them with refactoring strategies. Code smells are particular bad patterns in source code, which violate important principles in implementation and design issues. The reason for widespread research on code smells is that it suggests an approach of evaluating maintainability. Particularly, code smells can indicate *when* and *what* refactoring can be applied. In recent decades, related work mainly focuses on empirical studies which investigated the effects of code smells on different maintainability related aspects, such as changes [4-6], effort [7-9], comprehensibility [10,11] and defects [12]. Various empirical evidences on these effects highlight the essential role of code smell detection in supporting program maintenance and guaranteeing program quality.

Early smell detection approaches tend to be manual [13], which have efficiency limitations [14, 15]. Automated detection allows code smells to be analyzed and detected consistently in large scale code bases. To this end, a wide variety of detection approaches have been proposed. Existing approaches of automated smell detection consist of metrics [16-18], machine learning [19-21] and some others [22, 23].

Although there are a range of approaches and tools of detecting various code smells, no work focuses on Python smells. Python is a typical dynamic scripting language [25], many of whose features make it so appealing for rapid development and prototyping. Python has a remarkably simple and expressive syntax, making length of code much shorter than many others. Due to active communities and copious documentation, quite a lot of programmers begin learning programming with Python in fact. However, because of its flexibility and dynamism, Python is a particularly challenging language to write code in and maintain. In particular, opposite to design patterns [1], code smells in Python programs reduce maintainability and cause difficulties in evolving and enhancing the Python software [2].

By detecting potential code smells in Python programs, the final goal of this study is to support refactoring strategies to enhance maintainability and to enable the study of their impacts on software quality and software maintenance. To address this problem, this paper introduces 11 code smells in Python programs and determines the metrics and criteria used for identifying them. Meanwhile, we try to give alternatives to those smells to improve the code. The scope of these smells is a part collection of the most unfortunate but occasionally subtle issues recognized in Python code. There are always reasons to use some of these Python smells, but in general using these code smells makes for less readable, more buggy, and less "Pythonic" (or idiomatic) code. In order to evaluate our work, we check five real world Python systems by our detection tool named Pysmell to investigate the prevalence, distribution and evolvement of Python smells. Results of our experiment can help researchers focus their effort on developing effective detection and elimination strategies and tools for those Python smells which are most prevalent or most easily ignored.

Our work makes the following main contributions:

- We present 11 code smells in Python programs and respective metrics and criteria for detection, including 5 generic ones and 6 additional ones.
- We propose the metric-based strategy in smell detection and implement a detection tool named Pysmell which allows user configuration for Python programs. We evaluate the performance of Pysmell in detecting five Python systems. The results show that Pysmell can find 285 code smell instances with the average precision of 97.7% and the average recall of 100%.
- We explore the prevalence of code smells and find out that Long Method and Long Class are the most prevalent ones in Python programs. We also observe the changes of smell instances in different releases of each Python system. The results show that the number of

978-1-5090-4517-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SATE.2016.10

PySmell

Detecting Code Smells in Python Programs

Zhifei Chen[1], Lin Chen[2+], Wanwangying Ma[3], Baowen Xu[4+]
State key Laboratory of Novel Software Technology
Nanjing University
Nanjing, China
{[1]chenzhifei, [3]wwyma}@smail.nju.edu.cn, {[2]lchen, [4]bwxu}@nju.edu.cn

***Abstract*—As a traditional dynamic language, Python is increasingly used in various software engineering tasks. However, due to its flexibility and dynamism, Python is a particularly challenging language to write code in and maintain. Consequently, Python programs contain code smells which indicate potential comprehension and maintenance problems. With the aim of supporting refactoring strategies to enhance maintainability, this paper describes how to detect code smells in Python programs. We introduce 11 Python smells and describe the detection strategy. We also implement a smell detection tool named Pysmell and use it to identify code smells in five real world Python systems. The results show that Pysmell can detect 285 code smell instances in total with the average precision of 97.7%. It reveals that Large Class and Large Method are most prevalent. Our experiment also implies Python programs may be suffering code smells further.**

***Keywords*—Python; code smells; program maintenance; refactoring**

I. INTRODUCTION

The notion of code smells was introduced initially by Fowler [3], who presented 22 code smells and associated them with refactoring strategies. Code smells are particular bad patterns in source code, which violate important principles in implementation and design issues. The reason for widespread research on code smells is that it suggests an approach of evaluating maintainability. Particularly, code smells can indicate *when* and *what* refactoring can be applied. In recent decades, related work mainly focuses on empirical studies which investigated the effects of code smells on different maintainability related aspects, such as changes [4-6], effort [7-9], comprehensibility [10,11] and defects [12]. Various empirical evidences on these effects highlight the essential role of code smell detection in supporting program maintenance and guaranteeing program quality.

Early smell detection approaches tend to be manual [13], which have efficiency limitations [14, 15]. Automated detection allows code smells to be analyzed and detected consistently in large scale code bases. To this end, a wide variety of detection approaches have been proposed. Existing approaches of automated smell detection consist of metrics [16-18], machine learning [19-21] and some others [22, 23].

Although there are a range of approaches and tools of detecting various code smells, no work focuses on Python smells. Python is a typical dynamic scripting language [25], many of whose features make it so appealing for rapid development and prototyping. Python has a remarkably simple and expressive syntax, making length of code much shorter than many others. Due to active communities and copious documentation, quite a lot of programmers begin learning programming with Python in fact. However, because of its flexibility and dynamism, Python is a particularly challenging language to write code in and maintain. In particular, opposite to design patterns [1], code smells in Python programs reduce maintainability and cause difficulties in evolving and enhancing the Python software [2].

By detecting potential code smells in Python programs, the final goal of this study is to support refactoring strategies to enhance maintainability and to enable the study of their impacts on software quality and software maintenance. To address this problem, this paper introduces 11 code smells in Python programs and determines the metrics and criteria used for identifying them. Meanwhile, we try to give alternatives to those smells to improve the code. The scope of these smells is a part collection of the most unfortunate but occasionally subtle issues recognized in Python code. There are always reasons to use some of these Python smells, but in general using these code smells makes for less readable, more buggy, and less "Pythonic" (or idiomatic) code. In order to evaluate our work, we check five real world Python systems by our detection tool named Pysmell to investigate the prevalence, distribution and evolvement of Python smells. Results of our experiment can help researchers focus their effort on developing effective detection and elimination strategies and tools for those Python smells which are most prevalent or most easily ignored.

Our work makes the following main contributions:

- We present 11 code smells in Python programs and respective metrics and criteria for detection, including 5 generic ones and 6 additional ones.
- We propose the metric-based strategy in smell detection and implement a detection tool named Pysmell which allows user configuration for Python programs. We evaluate the performance of Pysmell in detecting five Python systems. The results show that Pysmell can find 285 code smell instances with the average precision of 97.7% and the average recall of 100%.
- We explore the prevalence of code smells and find out that Long Method and Long Class are the most prevalent ones in Python programs. We also observe the changes of smell instances in different releases of each Python system. The results show that the number of

978-1-5090-4517-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SATE.2016.10

- Complex Class
- Long Parameter List
- Long Method
- Long Message Chain
- Long Scope Chain
- Long Base Class List
- Unless Exception Handling
- Long Lambda Function
- Complex List Comprehension
- Long Element Chain
- Long Ternary Conditional Expression

Complex List Comprehension

```
numbers_str = ["24", "15", "21", "27", "35", "40", "45", "50", "121", "363"]
filtered_numbers = [int(num) ** 2 for num in numbers_str if (int(num) % 3 == 0  and len(num) >= 2 and "5" not in num
and str(int(num) ** 2) == str(int(num) ** 2)[::-1])]
```

Complex List Comprehension

```
numbers_str = ["24", "15", "21", "27", "35", "40", "45", "50", "121", "363"]
filtered_numbers = [int(num) ** 2 for num in numbers_str if (int(num) % 3 == 0  and len(num) >= 2 and "5" not in num
and str(int(num) ** 2) == str(int(num) ** 2)[::-1])]
```

Refactoring Strategy

```
numbers_str = ["24", "15", "21", "27", "35", "40", "45", "50", "121", "363"]
filtered_numbers = []
for num in numbers_str:
    num_int = int(num)
    if num_int % 3 == 0:
        if len(num) >= 2:
            if "5" not in num:
                square = num_int ** 2
                if str(square) == str(square)[::-1]:
                    filtered_numbers.append(square)
```

Goal

We aim to investigate the **diffusion** of **Code Smells** in **AI-enabled Systems over time** and the **activities performed** by developers that can **induce Code Smell proliferation**

Research Questions

Research Questions

What is the **diffusion** of
Code Smells in **AI-Enabled Systems**?

Research Questions

What is the **diffusion** of **Code Smells** in **AI-Enabled Systems**?

What are the **activities** that **most frequently** lead to the **introduction** of **Code Smells** in **AI-Enabled Systems**?

Research Questions

What is the **diffusion** of **Code Smells** in **AI-Enabled Systems**?

What are the **activities** that **most frequently** lead to the **introduction** of **Code Smells** in **AI-Enabled Systems**?

Sub-Research Questions

What is the **frequency** of
Code Smells in **AI-Enabled Systems**?

Sub-Research Questions

What is the **frequency** of
Code Smells in **AI-Enabled Systems**?

What is the **density** of
Code Smells in **AI-Enabled Systems**?

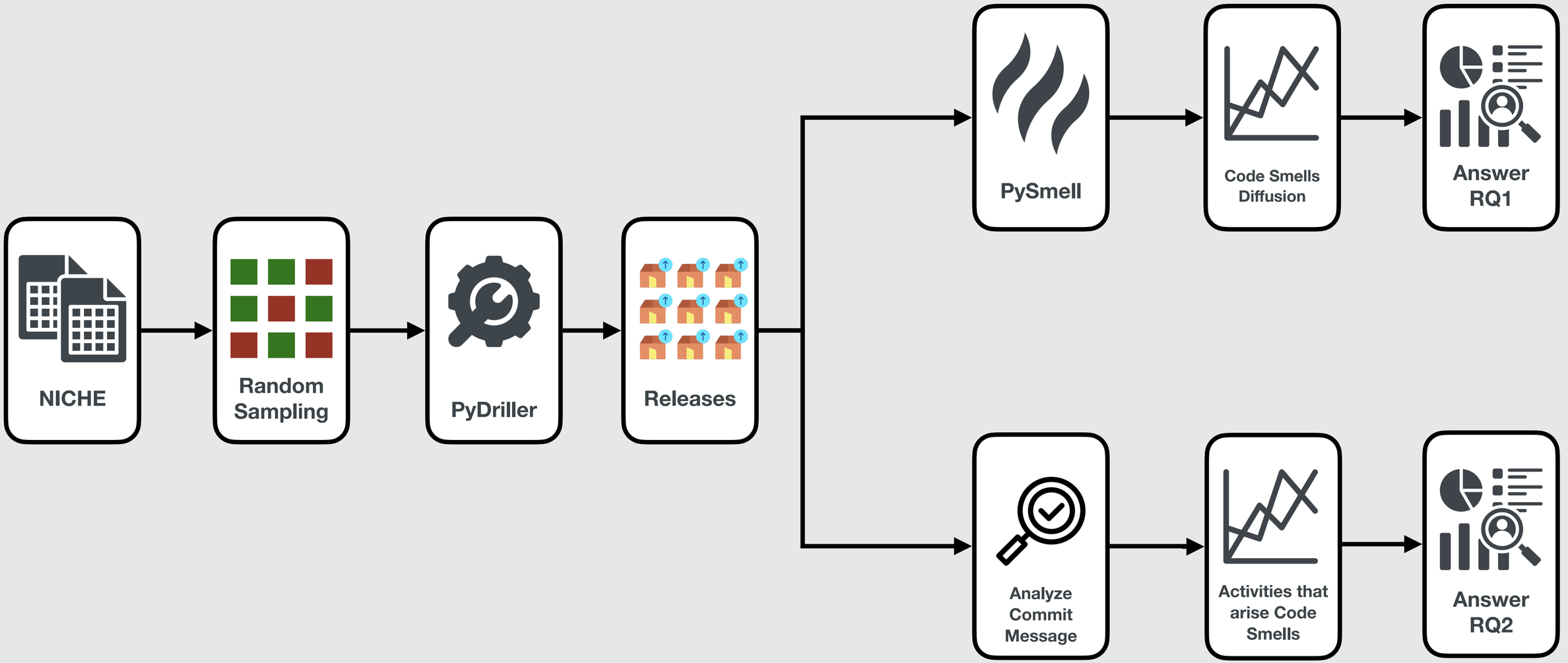
Sub-Research Questions

What is the **frequency** of
Code Smells in **AI-Enabled Systems**?

What is the **density** of
Code Smells in **AI-Enabled Systems**?

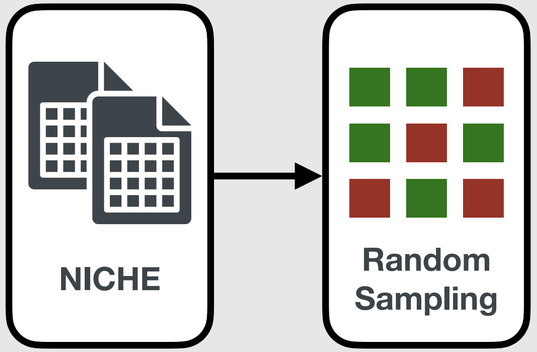
What is the **survival** of
Code Smells in **AI-Enabled Systems**?

Research Process



Research Process

Research Process



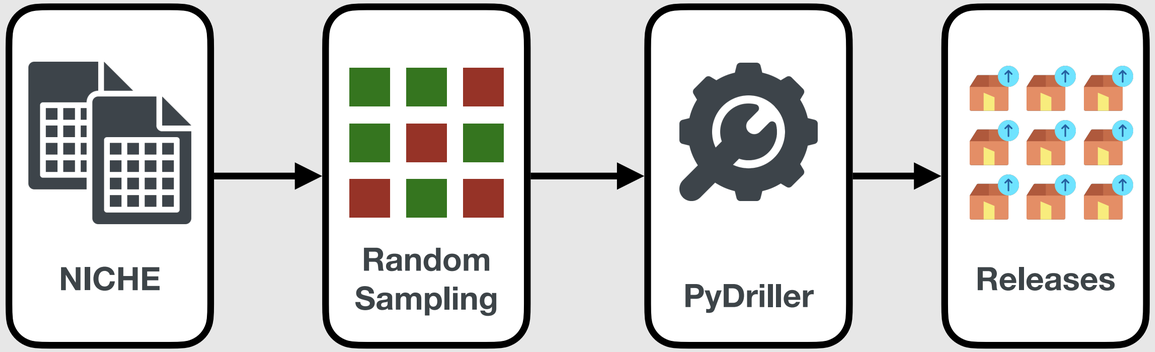
Research Process

We selected 200 AI-Enabled Systems

NICHE

Sampling

Research Process



Research Process

10,600 releases analyzed

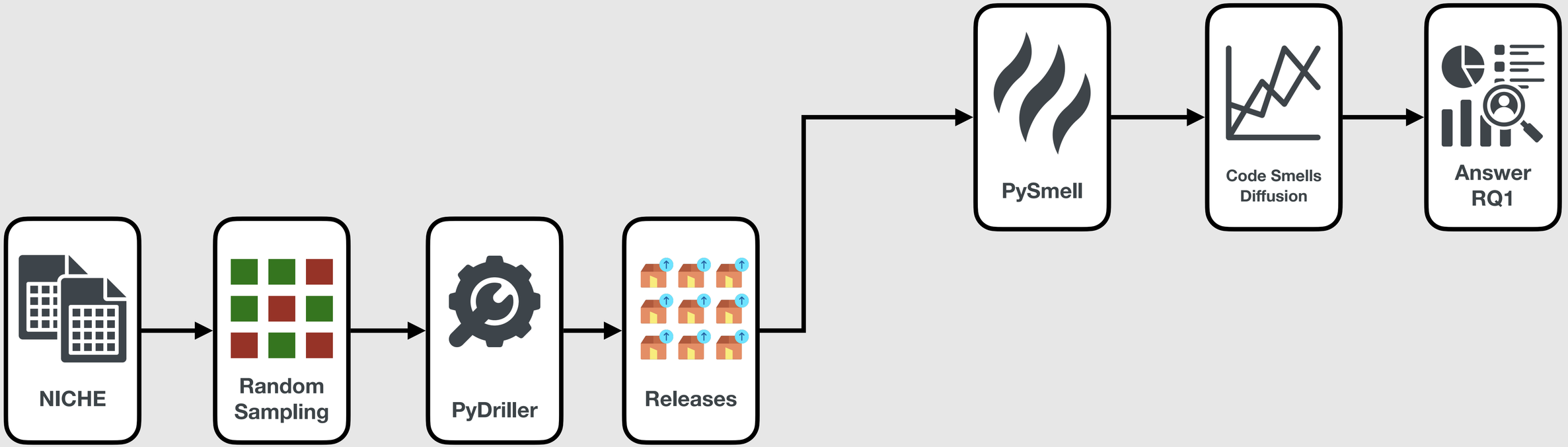
NICHE

Sampling

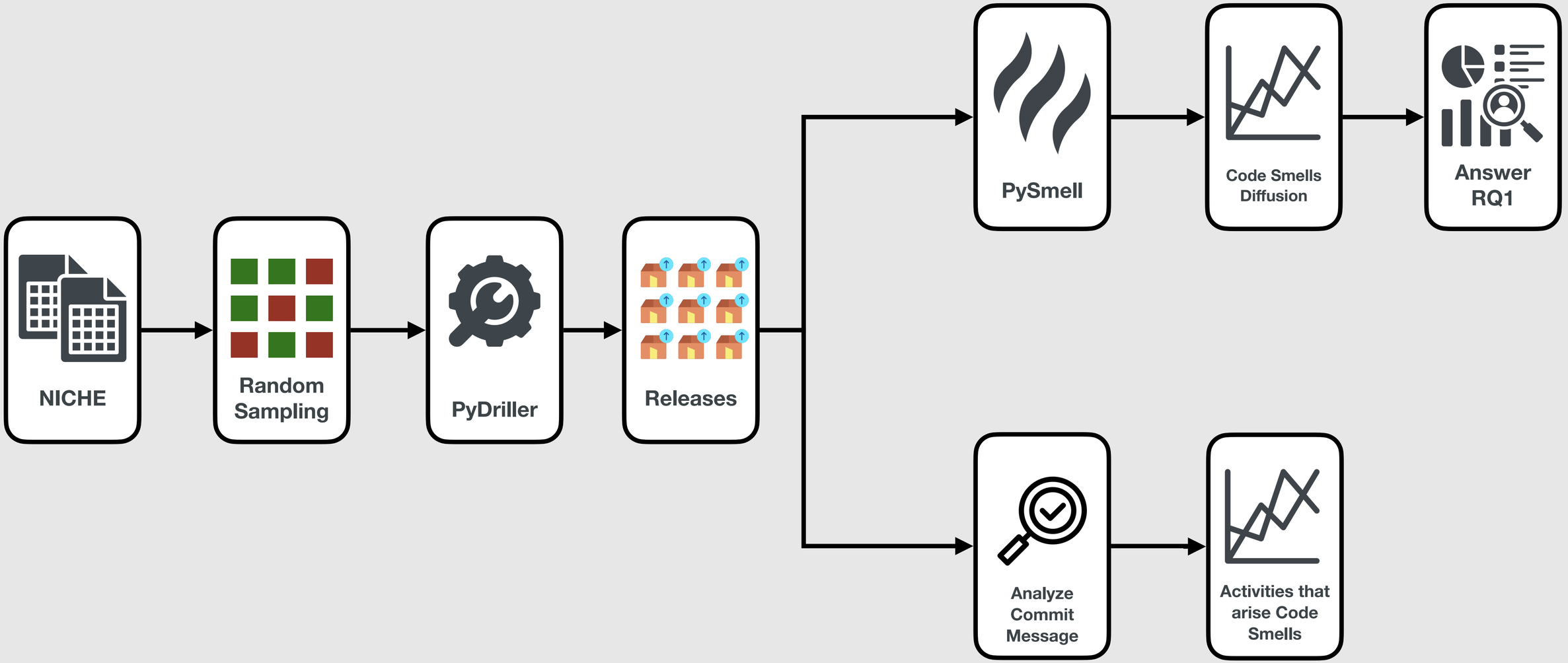
PyDriller

Releases

Research Process



Research Process



How we identified the smell introduction

How we identified the smell introduction

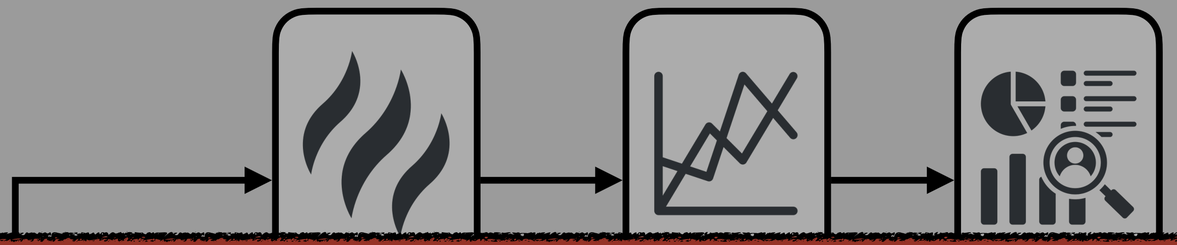
We calculated the **number of smells for each release** for **all the projects**

How we identified the smell introduction

We calculated the **number of smells for each release** for **all the projects**

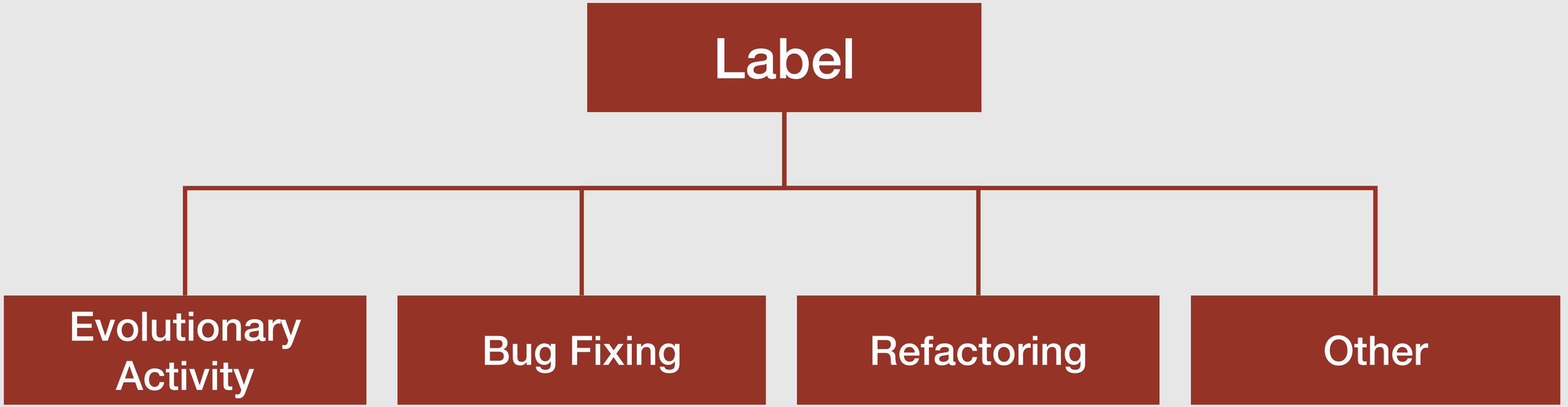
We marked the release Ri+1 as **“Increase”** if the difference in terms of the **number of smells** between the **release Ri+1** and **Ri** is **greater** than 0

Research Process



We **automatically analyzed** the **commit messages** and **labeled** them into 4 **categories**

Labels commit messages



Labels commit messages

We **removed** the commits labeled as “**Other**”

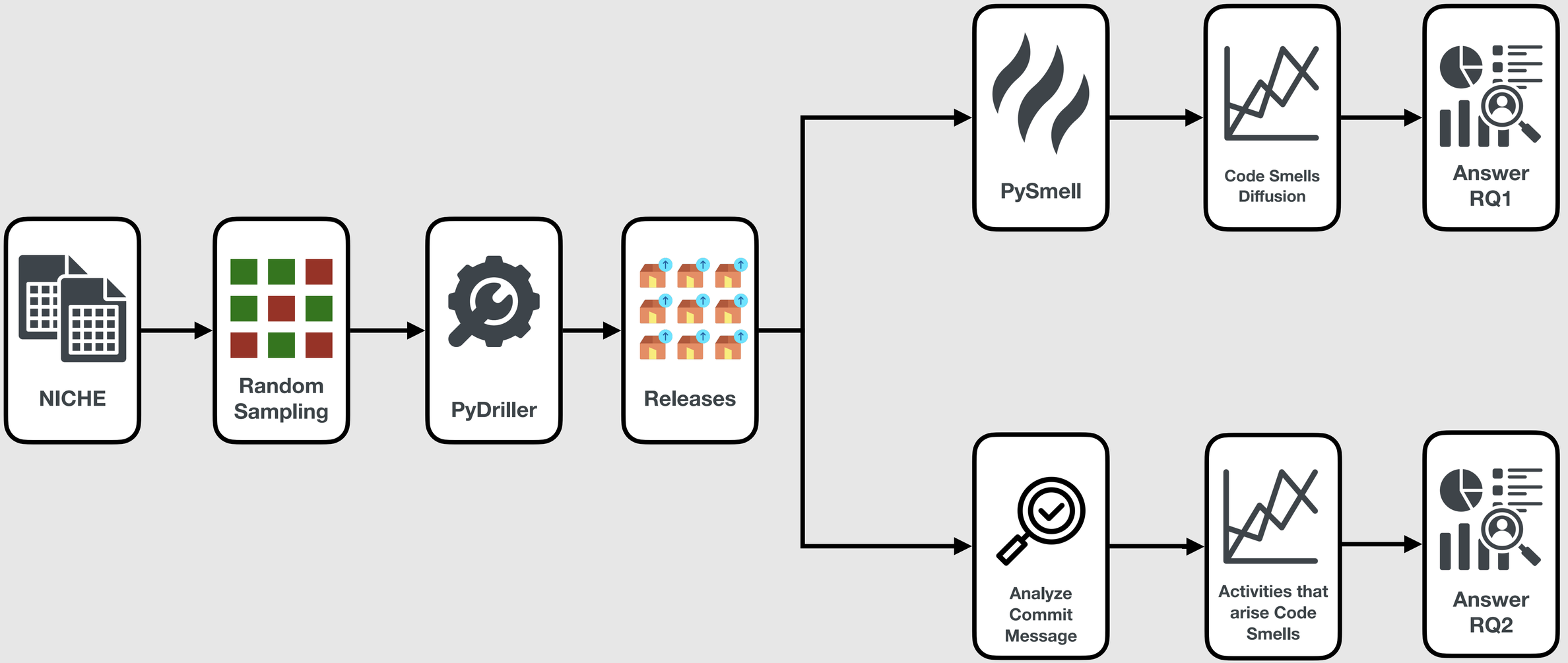
- Evolutionary Activity
- Bug Fixing
- Refactoring
- Other

Labels commit messages

We **removed** the commits labeled as “**Other**”

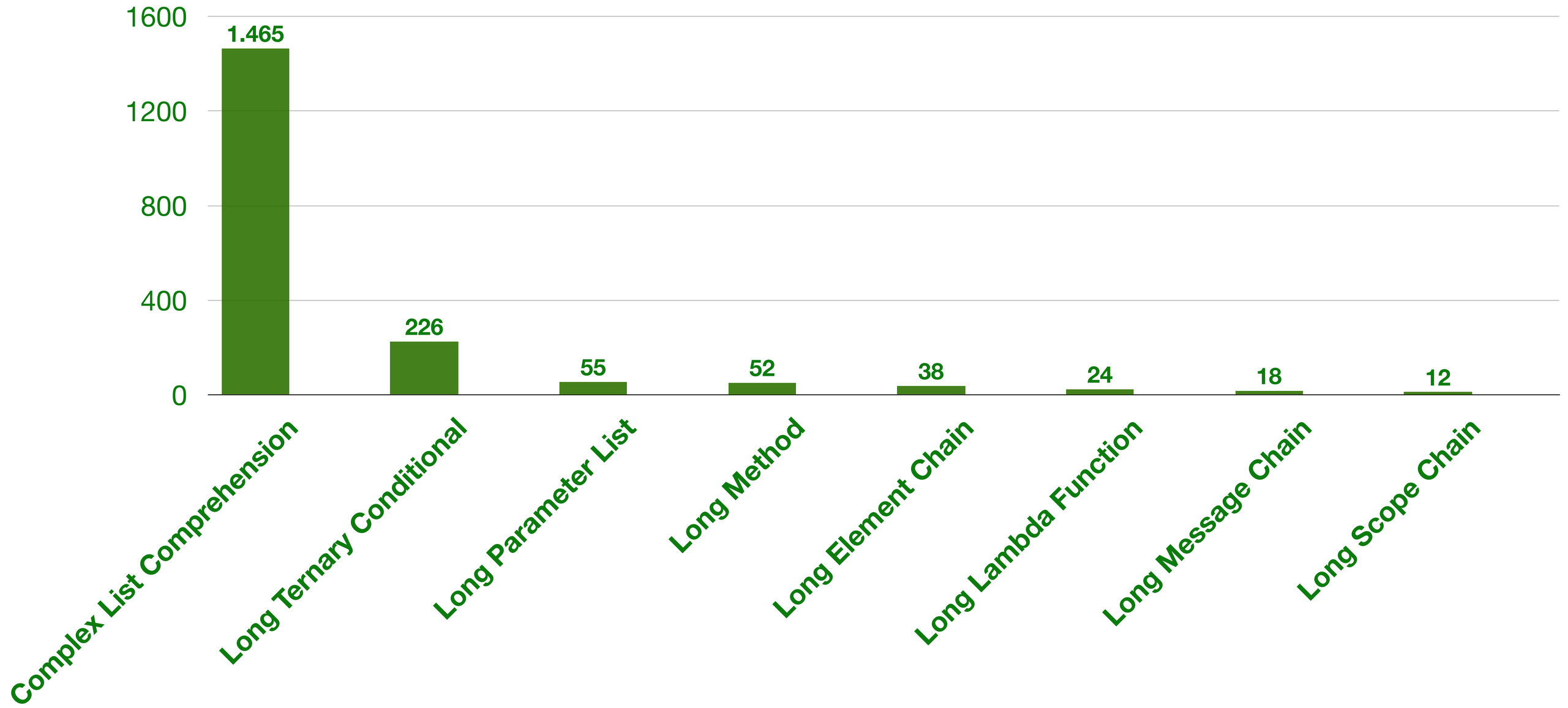
If a commit **message** referred to **more than one operation**, we **labeled it** with **multiple labels** (e.g., Bug Fixing and Evolutionary Activity)

Research Process



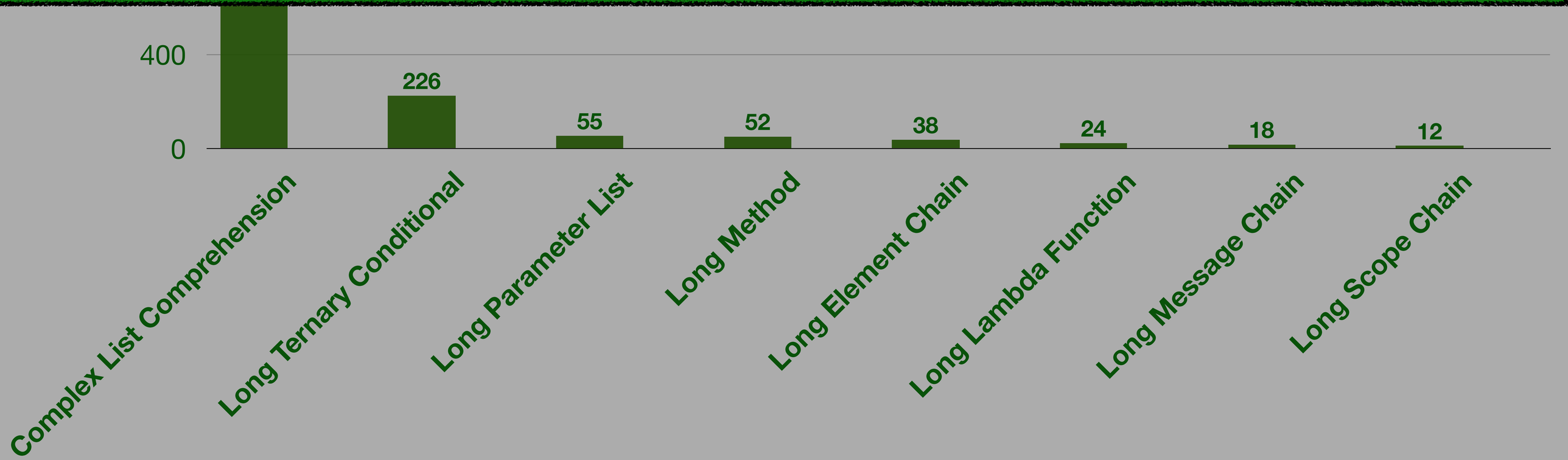
On the **frequency** of **Code Smells** in **AI-Enabled Systems**

On the frequency of Code Smells



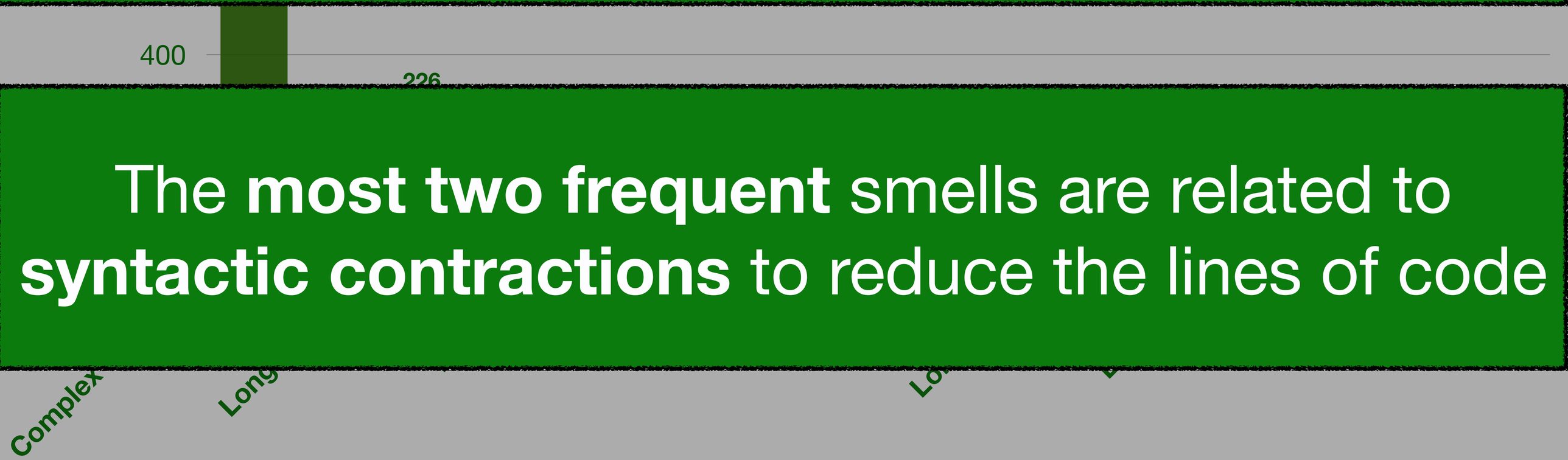
On the frequency of Code Smells

Code Smells related to **Object-Oriented programming languages** (e.g., complex class) are **never detected**



On the frequency of Code Smells

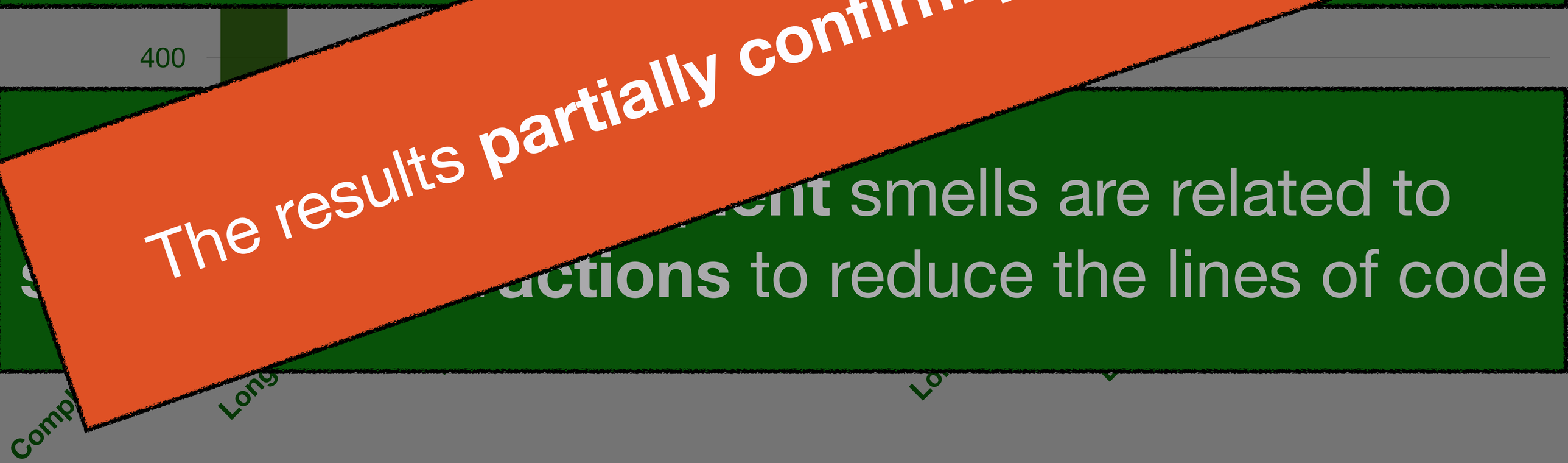
Code Smells related to **Object-Oriented programming languages** (e.g., complex class) are **never detected**



The **most two frequent** smells are related to **syntactic contractions** to reduce the lines of code

On the frequency of Code Smells

Code Smells related to **Object**
programming languages (
n



S **actions** to reduce the lines of code

The results **partially confirm previous work**

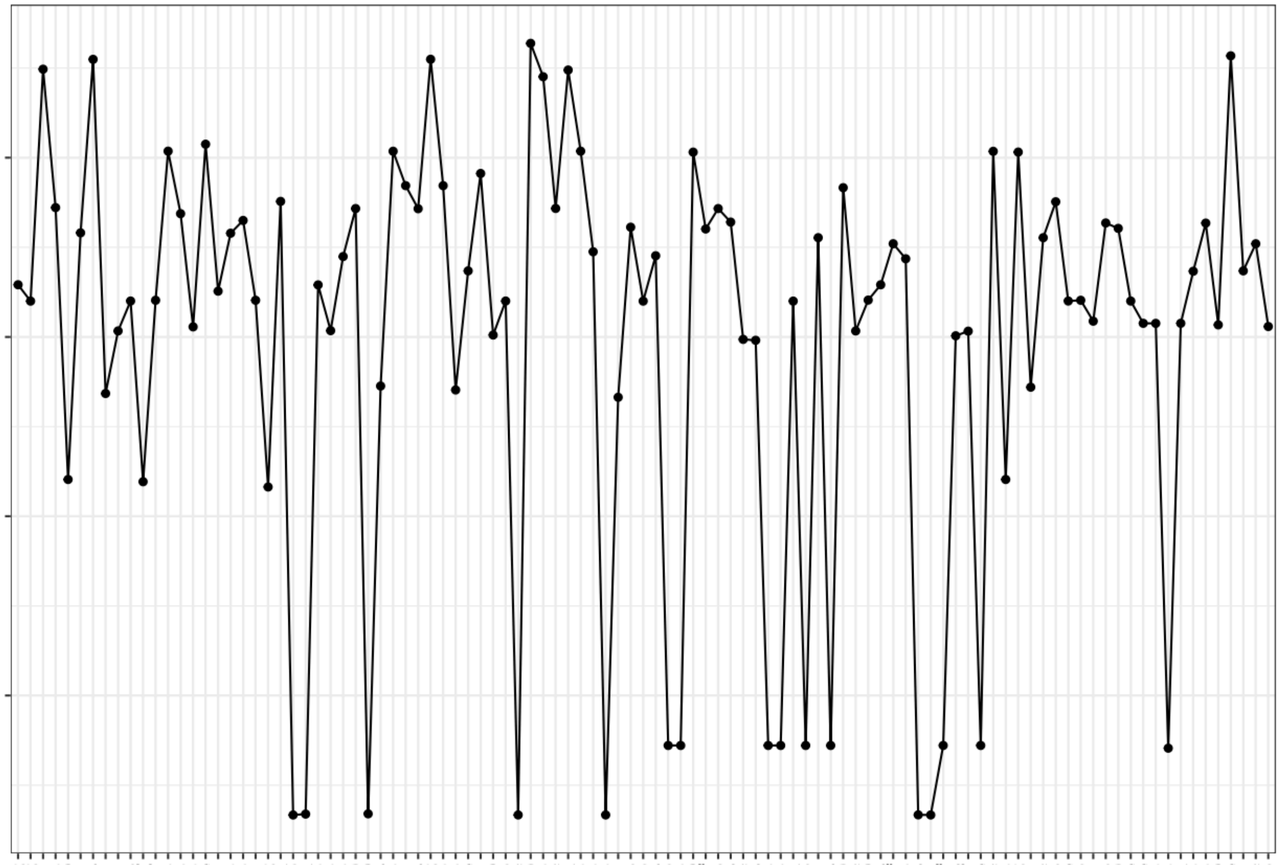
80%

Of the projects were affected at least once by a *Complex List Comprehension*

On the **density** of **Code Smells** in **AI-Enabled Systems**

On the density of Code Smells

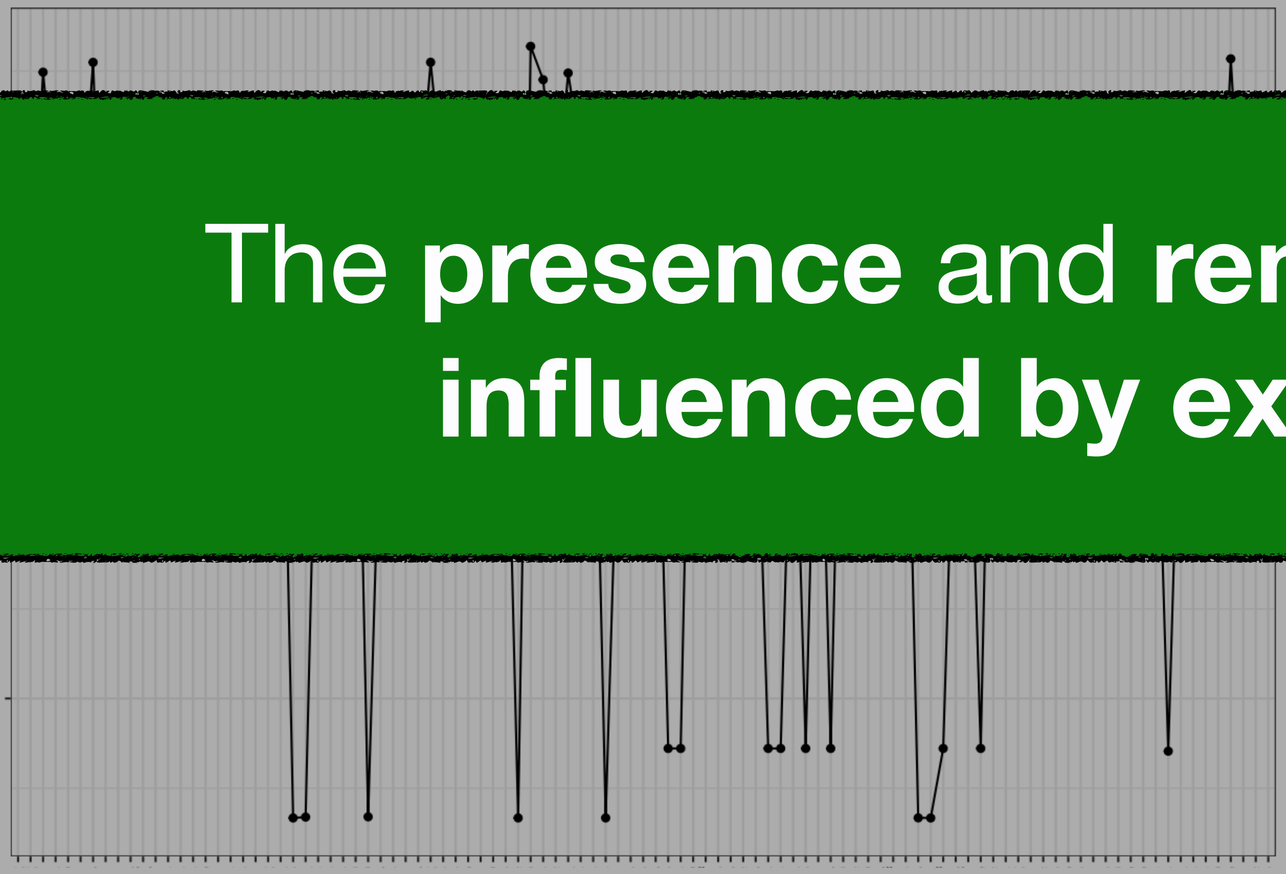
Code Smell density for the project MindMeld



We observed that the **density** of **Code Smells does not follow** a **specific trend** of **increase/ decrease over time**

On the density of Code Smells

Code Smell density for the project MindMeld



We observed that the

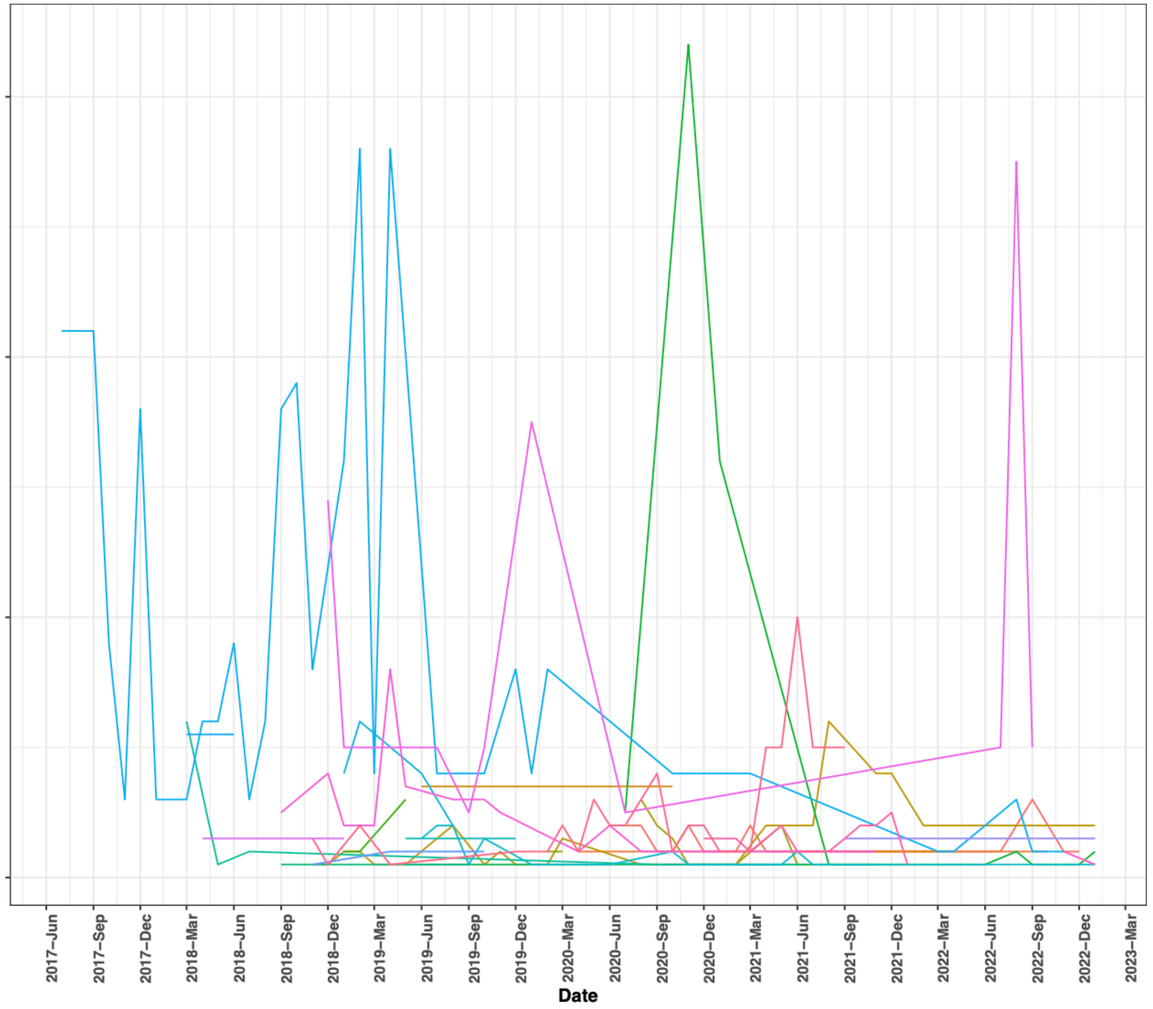
The **presence** and **removal** appear to be **influenced by external factors**

trend of **increase/ decrease over time**

On the **survival** of **Code Smells** in **AI-Enabled Systems**

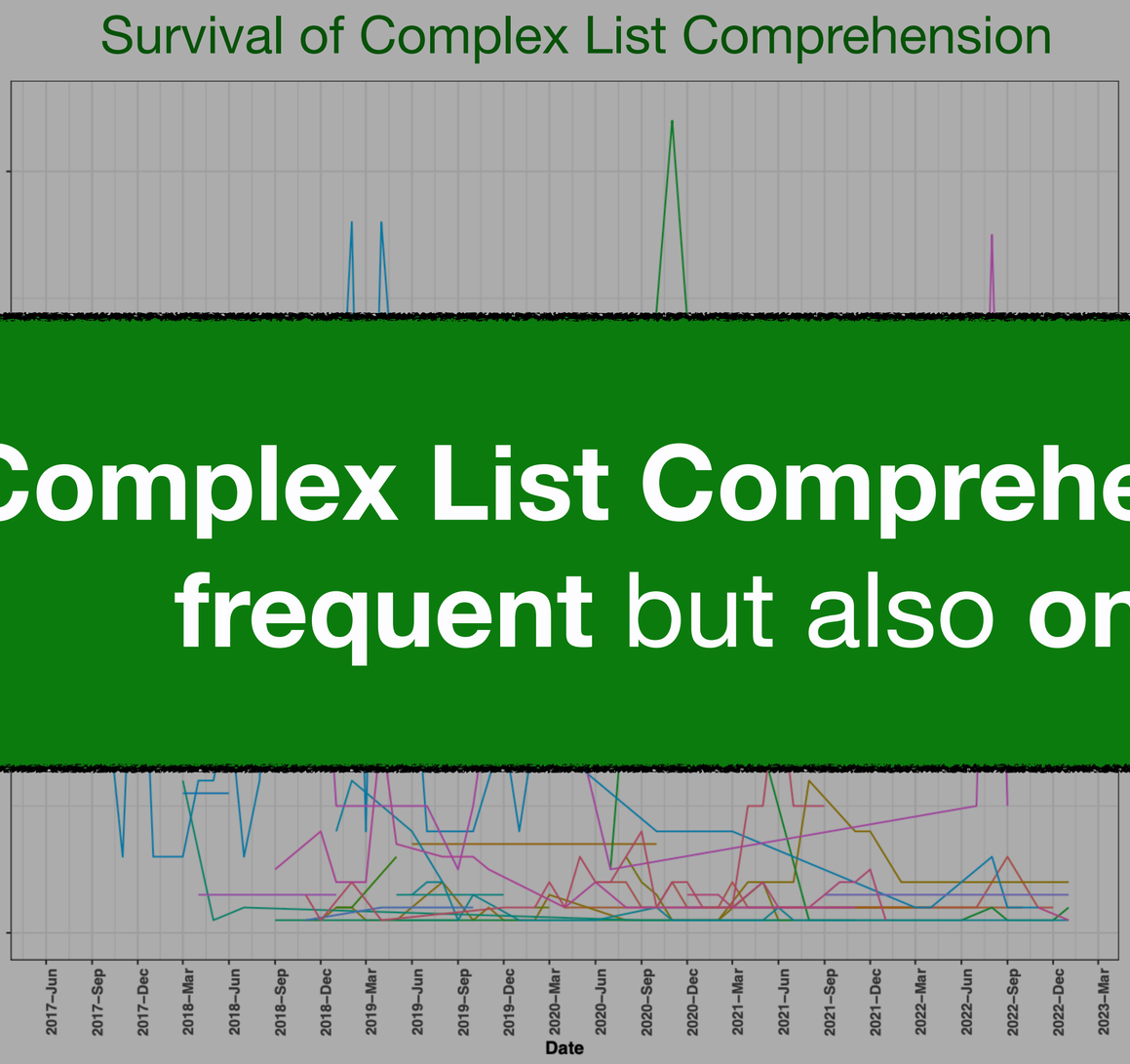
On the survival of Code Smells

Survival of Complex List Comprehension



In some cases, we found that Code Smells survived for a time period of 6 years!

On the survival of Code Smells



Complex List Comprehension is **not only** the **most frequent** but also **one of the longest-lived**

On the **activities** that led developers to introduce Code Smells in AI-Enabled Systems

70%

of Code Smells has been introduced due to Evolutionary Activities

70%

In **most cases** the **introduction** of **code smells** is due to **merge operations**

of Code Smells has been introduced due to Evolutionary Activities

The **practices** used by **Python developers** to build **AI-Enabled Systems** combined with the **best practices** used to write **Python code** can **arise** the proliferation of **specific Code Smells**

The **practices** used by **Python developers** to build **AI-Enabled Systems** combined with the **best practices** used to write **Python code** can **arise** the proliferation of **specific Code Smells**

Developers should adopt *quality assurance* tools for monitoring code quality attributes

Summing up

Summing up

We analyzed 10,600 releases
of 200 AI-Enabled Systems

Summing up

We analyzed 10,600 releases
of 200 AI-Enabled Systems

Complex List Comprehension is the most
frequent and longest-lived smell

Summing up

We analyzed 10,600 releases
of 200 AI-Enabled Systems

Complex List Comprehension is the most frequent and longest-lived smell

The Code Smells trend seems to be project-dependent

st

ect-

Future Work

st

ect-

Future Work

Assess our results
increasing the number of projects

st

ect-

Future Work

Assess our results
increasing the number of projects

Analyze the developers’ perception on Python specific Code Smells

Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study

Giammaria Giordano, Giusy Annunziata, Andrea De Lucia, and Fabio Palomba

University of Salerno, Italy

IWSM MENSURA
September 15, 2023
Rome, Italy

giagiordano@unisa.it

SCAN ME!
I'm the paper