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

Giammaria Giordano[1], Giusy Annunziata[1], Andrea De Lucia[1] and Fabio Palomba[1]

[1]*University of Salerno (Italy) - SeSa Lab*

Abstract

To deal with continuous change requests and the strict time-to-market, practitioners and big companies constantly update their software systems to meet users' requirements. This practice force developers to release immature products, neglecting best practices to reduce delivery times. As a possible result, *technical debt* can arise, *i.e.,* potential design issues that can negatively impact software maintenance and evolution and, in turn, increase both the time-to-market and costs. *Code smells*—sub-optimal design decisions identifiable by computing software metrics and providing a general overview of code quality —are common symptoms of technical debt. While previous research focused on code smells primarily considering them in the context of Java, the growing popularity of Python, particularly for developing artificial intelligence (AI)-Enabled systems, calls for additional investigations. This preliminary analysis addresses this gap by exploring the diffusion of Python-specific code smells, and the activities performed by developers that induce the introduction of code smells in their systems. To perform our preliminary investigation, we selected 200 AI-Enabled systems available in the NICHE dataset; We extracted 10,611 information on the releases using PYDRILLER, and PYSMELL to extract information about code smells. The results reveal several insights: 1) Code smells related to object-oriented principles are rarely detected in Python; 2) Complex List Comprehension is the most prevalent and the most long-alive smell; 3) The main activities that can induce code smells are evolutionary. This study fills a critical gap in the literature by providing empirical evidence on the evolution of code smells in Python-based AI-enabled systems.

Keywords

Software Maintenance, Software Evolution, Software Refactoring, Code Smells,

1. Introduction

To meet customers' needs and due to the continuous environmental changes, practitioners and big companies update their source code as fast as possible [1]. The continuous *change requests* and the stringent *time-to-market* force developers to release immature products, putting aside best practices to decrease the delivery time [2, 3]. As a possible output of this process could be the introduction of *technical debt* [4]—*i.e.,* potential design issues that can negatively impact the software system during maintenance and evolution. One of the *symptoms* of technical debt is

IWSM/MENSURA 23, September 14–15, 2023, Rome, Italy

giagiordano@unisa.it (G. Giordano); gannunziata@unisa.it (G. Annunziata); adelucia@unisa.it (A. D. Lucia); fpalomba@unisa.it (F. Palomba)

https://giammariagiordano.github.io/giammaria-giordano/ (G. Giordano); https://giusyann.github.io/ (G. Annunziata); https://docenti.unisa.it/003241/home (A. D. Lucia); https://fpalomba.github.io/ (F. Palomba)

0000-0003-2567-440X (G. Giordano); 0009-0002-0742-7261 (G. Annunziata); 0000-0002-4238-1425 (A. D. Lucia); 0000-0001-9337-5116 (F. Palomba)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

the presence of so-called *code smells* [5], *i.e.,* sub-optimal design decisions applied by developers during the software development. Code smells are detectable by calculating software metrics that provide insight into the quality of the project. They can indicate a negative impact on software maintenance and increase the effort required to perform evolutionary activities.

Over the last decades, several researchers targeted studies to investigate code smells from different angles. On the one hand, by proposing both static and machine learning tools useful to detect the presence of code smells [6, 7, 8] and their impact on code quality. On the other hand, researchers are focused on understanding when and especially the motivations that lead to introducing code smells [9, 10]. In addition, it is noticed that their presence negativity impacts code attributes—*e.g.,* code comprehension [11] and change-bug proneness [12, 13]. Despite the willingness spent by researchers on this topic, we noticed that several previous work consider as "lowest common denominator" the adoption of Java programming language [14, 15, 16]. Although the use of Java is consolidated over time, other programming languages—*i.e.,* Python—are increasingly widespread; a recent statistic[1] reports that Python jumped over Java in terms of diffusion in the last few years. Although the possible reasons for this overtaking are multiple, we noticed that it is common practice for practitioners and big companies to select programming languages that allow combining different paradigms—*e.g.,* object-oriented and procedural— with taking full advantage of the features of each of the paradigms, characteristics that are by default in Python. Moreover, from this perspective, we noticed that only a tiny subset of previous work focuses on detecting code smells for Python projects but considers only traditional systems [17, 18]. However, taking into account that Python is one of the most popular programming languages to build AI-Enabled systems[2] and considering the different philosophy in terms of the mindset of Python, we noticed a lack of empirical investigation on the diffusion of code smells in AI-Enabled systems, and the related activities performed by developers during the introduction of them. Seek consolidated literature, on code smells, in traditional evolutionary systems drove us to investigate them [9, 19]. That work emphasizes identifying the diffusion of code smells and the activities most likely to cause their introduction as a first step in keeping effort low during software maintenance [20].

To fill this gap, we investigated both the diffusion of the code smells and the activities that led developers to introduce them in AI-Enabled systems. To conduct our analysis, we selected over 200 AI-Enabled systems provided by NICHE dataset [21], considered over 10,600 releases identified with PyDriller, and extracted information on code smells using PySmell.

The principal results indicated that: 1) The code smells regarding the object-oriented principles are rarely detected during our analysis, and this suggests that Python developers tend to use other reuse mechanisms to build AI-Enabled systems; 2) Complex List Comprehension has been observed 1465 times and is both the most present and the most long alive; 3) Code smells does not follow a specific and common pattern over time, but the trend seems to be project-dependent; 4) The evolutionary activities are the most common activities that can induce developers to introduce code smells.

Our work makes the following main contributions:

- A preliminary analysis of the diffusion of code smells in AI-Enabled systems;

[1] https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/
[2] https://bootcamp.berkeley.edu/blog/ai-programming-languages/

- A preliminary investigation on the activities performed by developers that led to the introduction of code smells in AI-Enabled systems;
- A publicly available replication package [22] containing raw data and scripts used to conduct our work that researchers can use to replicate or extend this work.

Paper structure. Section 2 discusses the background information and the related work connected to this work. Section 3 reports an overview of the method applied. Section 4 summarizes the results obtained. Section 5 shows the threats to validity and the mitigation strategies applied. Section 6 expands the discussion section and provides takeaway messages. Lastly, section 7 concludes the paper.

2. Background and Related Work

This section describes the background information on Python-specific code smells. In addition, we overview the current literature.

2.1. Background

With the proliferation of object-oriented (o.o.) programming languages, most effort has been spent by researchers to identify the presence of code smells in source code. Fowler released the first definition of code smells [5] in 1997, proposing an informal catalog of 22 code smells, identifying most of them by looking at the bad practices of object-oriented programming languages—*e.g., Large Class*—and discovered that their presence could increase the effort to perform maintenance activities. After the catalog was published, several studies investigated code smells in traditional systems, primarily emphasizing Java projects [16, 23, 24]. Nevertheless, Python developers underline substantial differences between Python and other programming languages in terms of syntax and mindset—*i.e.,* they do not limit to "translating" from other programming languages to Python—but instead, change the development approaches. These differences are so substantial that the Python community coined the term "Pythonic way"[3] to refer to the practice of writing code snippets that leverage the unique constructs and features provided by Python.

Due to the above considerations, a more specific re-definition of code smells is necessary to fit better the language's specific characteristics—a key example is *Complex List Comprehension.* The smell refers to a list comprehension that contains multiple and articulate expressions. Python provides expressions and operations that can be applied for each element of a list without needing a loop; However, if these expressions are too elaborated, the understandability could be affected by causing possible defect-proneness. To mitigate it, developers should consider replacing a complex list comprehension with a loop when possible.

2.2. Related Work

In the context of our work, we survey the state-of-the-art by discussing studies on Python-specific code smells both from an evolutionary point of view and not.

[3] https://medium.com/swlh/the-pythonic-way-6ad73abfbb00

Van Oort et al. [25] investigated the presence of Python-specific code smells by selecting 74 artificial intelligence projects using *PyLint*. The authors noticed two important facts: 1) The code smell most frequent is *duplicated code*, and 2) They found with a manual investigation that PyLint is not sufficiently able to detect code smells in AI-Enabled systems due to the similarity between the detection rules applied by the tool and mathematical expressions that in a non-negligible number of cases, caused unfair matches causing many false positives. Chen et al. [26] performed an empirical investigation of code smells in Python projects analyzing 106 popular repositories using PySmell. As a principal outcome, they found that *Long Parameter List* and *Long Method* were the most prevalent smells. Concerning previous work, we performed three additional steps: 1) We changed the domain of the experiment objects from traditional systems to AI-enabled systems; 2) We increased the number of projects from 106 to 200; and 3) We considered not only the diffusion but also the activities performed by developers that induced code smells.

To the best of our knowledge, the most similar study was performed by Chen et al. [17] in 2016, where they empirically evaluated Python-specific code smells from an evolutionary perspective by selecting 110 releases of 5 traditional Python projects. The authors found that the presence of code smells evolves over time but not statistically significantly. Also, in this case, the differences between our work are multiple: Firstly, we selected AI-Enabled systems as experimental objects instead of traditional ones. Secondly, we extended the original study in terms of the number of releases and projects. Finally, we also investigated the activities that led developers to introduce code smells in those systems.

3. Method

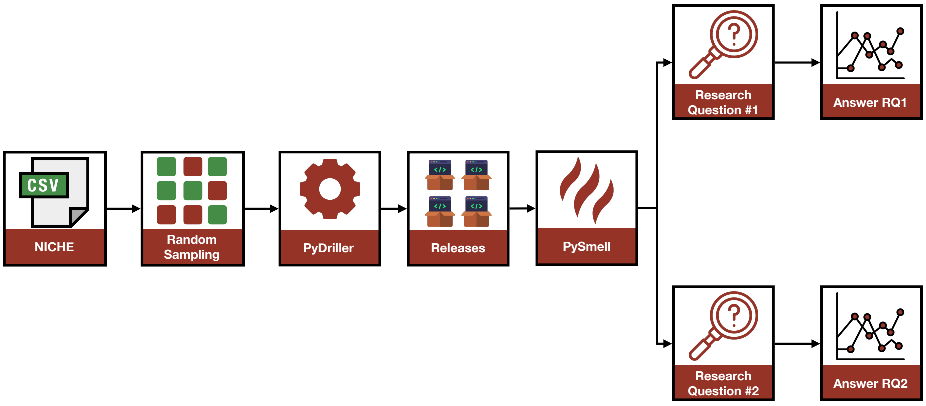

Figure 1: Overview of the method applied in this study.

The ultimate *goal* of this preliminary study is to analyze the diffusion of code smells in AI-Enabled systems and understand the activities performed by developers that, in turn, induced the introduction of code smells, with the *aim* to identify how code smells are distributed in AI-Enabled systems, and what stages of development are most likely to introduce code smells. The *perspective* is for both developers and researchers. The former are interested in avoiding an incidental introduction of code smells that can increase the effort during maintenance and evolutionary activities. The latter are interested in enhancing the knowledge of code smells during the software evolution in systems different from Java. For this reason, we formulated the following research questions:

> 🔍 **RQ$_1$.** *What is the diffusion of code smells in AI-Enabled systems?*

> 🔍 **RQ$_2$.** *What are the activities that most frequently lead to the introduction of code smells in AI-Enabled systems?*

The *objective* of the **RQ$_1$** is to assess the diffusion of code smells in terms of *frequency*, *density*, and *variation* with the *purpose* to give a general overview of code smells in AI-Enabled systems. For these reasons, we identified three sub-research questions:

- **RQ1$_1$** What is the frequency of code smells in AI-Enabled systems?
- **RQ1$_2$** What is the density of code smells in AI-Enabled systems?
- **RQ1$_3$** What is the variation of code smells in AI-Enabled systems?

While the *objective* of the **RQ$_2$** is to identify commits that introduced new code smells with the *purpose* of identifying which kinds of activities most frequently induce the introduction of new code smells. To conduct our experiments, we followed the empirical software engineering principles and guidelines of Wohlin et al. [27]. In addition, we follow the *ACM/SIGSOFT Empirical Standards* [4] to report our results. We include all the material, including scripts, raw data, and figures, in our online appendix publicity available [22]. Figure 1 provides an overview of the method applied to perform our study.

3.1. Dataset Selection

The *context* of this experiment is composed of 200 AI-specific projects and over 10,600 releases.

More specifically, to perform our analysis, we used NICHE dataset [21]—*i.e.,* a dataset published in 2023 that contains 572 AI-specific projects. The reasons why we chose this dataset are multiple. On the one hand, the authors filter out unpopular projects—*i.e.,* projects with less than 100 stars and no longer active projects. On the other hand, they manually verified information about the quality of the projects using a heuristic approach by labeling 400 projects

[4] Available at: https://github.com/acmsigsoft/EmpiricalStandards. We followed the "General Standard" and "Repository Mining"guidelines.

as “well-engineered” according to 8 distinct dimensions: Architecture, Community, Continuous Integration, Documentation, History, Issues, License, and Unit Testing.

Architecture. The projects have a clear definition of the components and how they communicate with other parts of the software system.

Community. All projects have many collaborators that maintain the repository.

Continuous Integration (CI). The projects use a CI mechanism that ensures stable source code for development or release.

Documentation. All projects provide documentation and additional material useful during maintenance activities.

History. The projects have a long history, which indicates that developers frequently perform maintenance tasks to guarantee a good level of viability.

Issues. The management activities have been done only using the GitHub issue, thus improving the traceability between requirements and source code.

License. All the projects explicitly expose a license of use useful to understand the terms of conditions about the partial or total reuse of system components.

Unit Test. To ensure a good quality level, all the projects show unit tests used to verify the correctness of the component.

Starting from the initial dataset, we focused on the 400 projects labeled as “well-engineered” and randomly selected a statistically significant sampling of 200 projects, considering a confidence level of 95%, and a margin error of 5%.

3.2. Data Collection

Once we identified the sample of projects, due to the time-consuming activity, we set up PyDriller [28] to extract only commits marked as “release” according to GitHub, and pull out the corresponding commit message. We decided to focus only on these commits because they are typically released after a more meticulous inspection by developers[5]. At the end of this step, we collected information on over 10,600 releases. To extract Python-specific code smell, we used PySmell [17]—*i.e.,* a code smell static analyzer tool. The principal motivations that drove to use it are that: 1) PySmell can detect 11 types of Python-specific code smell, and the authors manually validated the instances of code smells; 2) The tool is one of the most used in previous work on this topic [29, 26].

Finally, to better perform our analysis, we discarded projects not useful for our study *i.e.,* projects with zero instances of code smells and projects with fewer than two releases. At the end of this phase, we obtained 34 projects useful for our analysis.

To the sake of comprehension, we report the list of code smells detectable by PySmell with the relative detection rule in Table 1.

[5] https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases

Code Smell	Acronyı	Description	Detection rule
Large Class	LG	A class with a large number of operations.	Lines of code (LOC) $\geq$ 200 or Number of Attributes (NOA) + Number of Methods (NOM) > 40.
Long Parameter List	LPL	A method or a function that contains a long list of parameters.	Number of Parameters (PAR) $\geq$ 5.
Long Method	LM	A method or a function containing many Lines of Code (LOC).	Method Lines of Code (MLOC) $\geq$ 100.
Long Message Chain	LMC	An expression that accesses an object using a long line of dot operations.	Length of Message Chain (LMC) $\geq$ 4.
Long Scope Chain	LSC	A method or a function that shows a multiple-nested.	Depth of Closure (DOC) $\geq$ 3.
Long Base Class List	LBCL	A class that has been defined with too many base classes	Number of Base Classes (NBC) $\geq$ 3.
Useless Exception Handling	UEH	An exception too many generic or that contains an empty statement.	Number of Except Clauses ((NEC)= 1 and Number of General Exception Clauses (NGEC) = 1) or NEC = Number of Empty Except Clauses (NEEC)
Long Lambda Function	LLF	A lambda function that contains multiple and complex expressions.	Number of Characters in One Expression (NOC) $\geq$ 80.
Complex List Comprehension	CLC	A list comprehension that contains multiple and complex expressions.	Number of Loops (NOL) + Number of Control Conditions (NOCC) $\geq$ 4.
Long Element Chain	LEC	An expression accessing an object using a long list of bracket operators.	Length of Element Chain (LEC) $\geq$ 3.
Long Ternary Conditional Expression	LTCE	A ternary conditional expression too many long.	Number of Characters in One Expression (NOC) $\geq$ 40.

Table 1
List of Code Smells detectable with PySmell with the related detection rule.

3.3. Data Analysis

Once terminated the data collection, we analyzed the information from both quantitative and qualitative standpoints.

To address $\mathbf{RQ_1}$, we analyzed code smells diffusion in terms of *frequency*, *density*, and *variation* over time. More in detail, to analyze the *frequency*, we built a Python script to count for each release the instances of code smells, and then, we aggregated results independently from the project to provide a generic overall. To identify the *density*, we calculated the ratio between the number of smells and lines of code (LOC) for each release for all projects. Lastly, we clustered results in a time interval to calculate the *variation*.

To address $\mathbf{RQ_2}$, we analyzed the activities that led developers to introduce code smells. To address this, we performed the following steps: 1) We merged in a single *CSV* file all the output files; 2) For each pair release R_i, $\mathrm{R}_{i+1} \in$ project P_j, we labeled the release R_{i+1} as "*increase*" if the difference in terms of the number of code smells between the version R_{i+1} and R_i is more than 0; "*stable*", in the difference equal to 0; and lastly, "*decrease*" if the difference is lower than 0; 3) To identify what activities have been done by developers who have introduced code smells we labeled the commit marked as "*increase*" with "Bug fixing ", "Evolutionary Activity", "Refactoring", or "Other" according to the corresponding commit message, as also done in previous work [30] by using a manual "pattern-matching" strategy—*i.e.*, we manually verify the presence of specific keywords, *e.g.*, "fix bug" to indicate a bug fixing activity—in the commit message. To perform this step, the first two authors of this work independently labeled each commit marked as "*increase*" based on what they felt was the category that corresponded to the most appropriate activity, and in case of discordance, were discussed by involving the other authors of the study until convergence was reached.

At the end of this step, all authors agreed on the assigned categories. Table 2 shows the labels with the relative descriptions.

Label	Description
Bug Fixing	A commit removes a bug in the source code
Evolutionary Activity	A commit that introduces a new feature in the system
Refactoring	A commit that performs a refactoring activity
Other	A commit that does not provide sufficient information to be labeled

Table 2
Description of the labels used.

Two aspects are worth discussing. First, due to ambiguous commit messages, we decided to discard from our analysis commits labeled as "Other" to avoid possible noise—*e.g.*, message commits not written in English. Second, we give, in some cases, a combination of two or more labels—*e.g.*, Bug-Fixing and Refactoring—because, in some cases, the commit messages referred to more than one activity.

4. Analysis and Discussion of the Results

In this section we report the main results of our analysis and discuss findings and implications.

4.1. RQ1_1. On the frequency of Python-specific code smell

To address the *RQ1_1*, we analyze the frequency of Python-specific code smell according to the section 3. Figure 2 indicated the frequency of code smells.

According to our results, it is possible to make several considerations. First, we noticed that 3 of the 11 code smells categories were not detectable during our analysis—*i.e., Large Class*, *Long Base Class List*, and *Useless Exception Handling*. Considering that previous work on object-oriented languages underlines the predominance of these smells [31] and that all of them referred to improper use of the object-oriented principle, the main assumption of the absence of this family of smells is that developers do not adopt or only partially adopt object-oriented approaches to build AI-enabled systems, but prefer others reuse strategies. Second, we noticed a clear gap between the first two smells—*i.e., Complex List Comprehension* and *Long Ternary Conditional Expression*—which appear respectively 1400 and 226 times and the other 6 smells. In both cases, the smells refer to syntactic contractions to reduce the lines of code required to perform an operation. This result suggests a possible correlation between the Python philosophy that encourages developers to write compact code snippets and the massive presence of these smells.

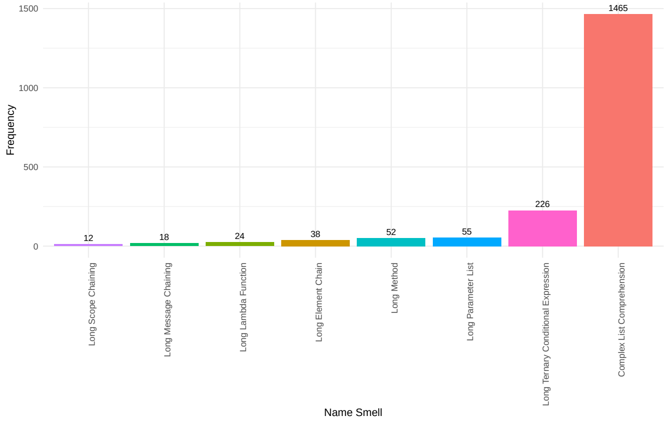

Figure 2: Results of the frequency of Python-specific code smell over time.

4.2. RQ1$_2$. On the density of Python-specific code smell

To address the *RQ1$_2$* we analyze the density of code smells from an evolutionary perspective. We observed that they often do not exhibit a consistent pattern of increase/decrease over time. Instead, they appear to be influenced by external factors, as exemplified by the anomaly observed in row 4, column 6, where an unstable pattern can be observed. These anomalies lead us to believe that the code smells introduction and their removal could vary causally due to software evolution activities. Figure 3 provides the density overview for all the projects under analysis.

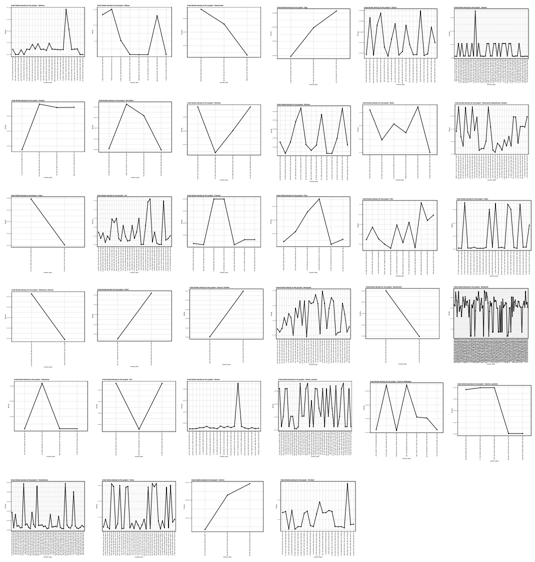

Figure 3: Overview of the density of code smells for each project analyzed.

4.3. RQ1$_3$. On the variation of Python-specific code smell

Finally, to address the *RQ1$_3$* we analyzed the code smells variation. We decided to show the results in a 3-month interval for readability reasons. Figure 4 shows the code smells trend over time. Looking at the figure, several considerations can be made. In the first place, no common pattern has been identified, suggesting that the code smells variation also seems project-dependent. Perhaps more interesting, we noticed that 80% of the projects had been affected at least once by a *Complex List Comprehension*. This outcome reinforces the results of **RQ1$_1$**, showing that introducing this kind of smell is frequent in these systems. Lastly, we noticed that this smell is also one of the longest-lived, as in some cases, its presence covers a period from 2017 to 2023.

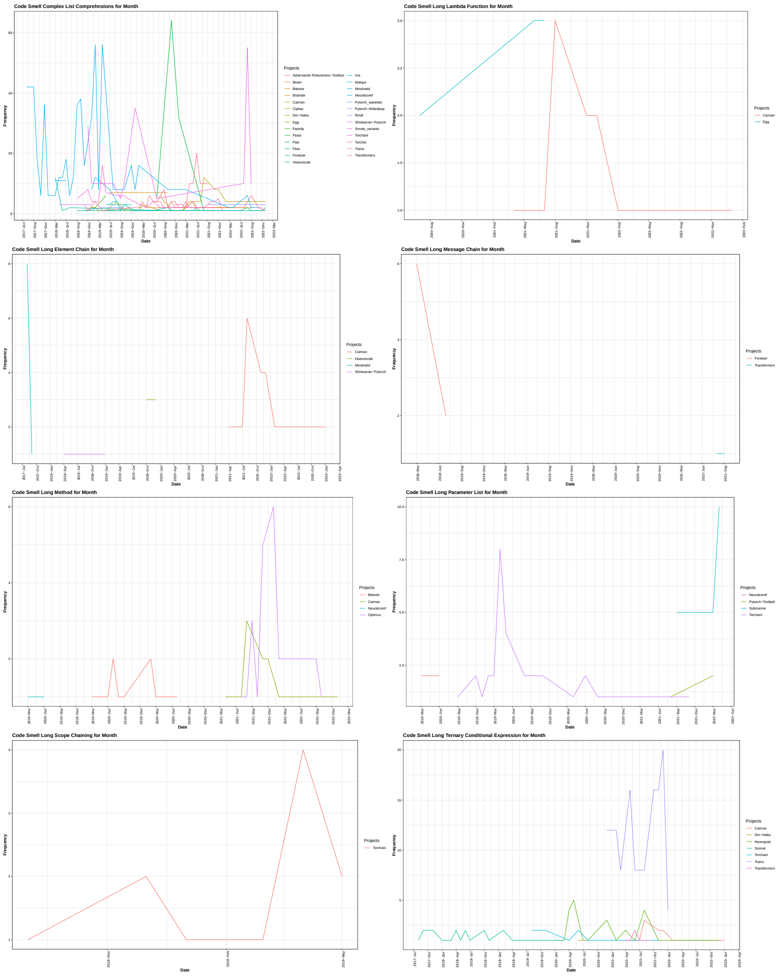

Figure 4: Results of the variation of Python-specific code smell over time for the month.

Key findings of RQ_1.

The density of code smells does not follow a specific pattern but varies depending on the project being considered. Code smells related to object-oriented practices are never detected during our analysis. The most frequent smell is *Complex List Comprehension*, with 1465 observations that are also the longest-lived.

4.4. RQ_2. On the activities that led developers to introduce code smells in AI-Enabled systems

To address the RQ_2, we analyzed activities that led developers to introduce code smells in AI-Enabled systems as specified in the section 3. Figure 5 shows the results obtained.

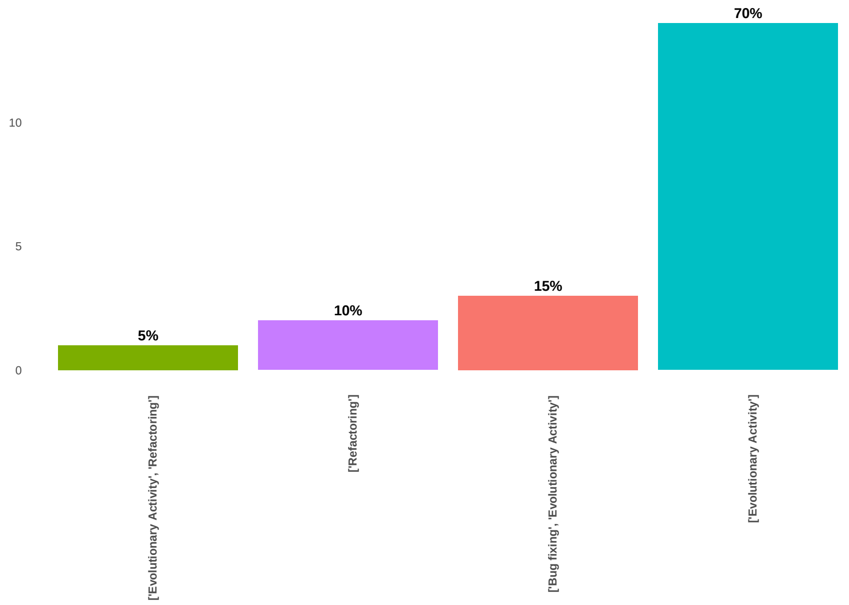

Figure 5: Activities performed by developers during the introduction of code smells.

The main outcome of this RQ is that developers introduce code smells in 70% of the cases during evolutionary activities. More in detail, by analyzing the commit messages it was observed that in most cases the commits, were related to merge activities or had generic messages indicating a system update. Although no further clarification can be provided for those generic commit messages related to system upgrades, it is assumed that the complexity and criticality of merge activities increase the likelihood of introducing code smells. Furthermore, due to the unstable trend exhibited over time, we can assume the developers tend to neglect the adoption of *quality assurance* tools throughout the software life cycle for monitoring code quality attributes. This highlights an unawareness regarding the potential impact of software quality degradation, which can lead to increased system complexity and the introduction of software bugs.

Key findings of RQ$_2$.

Code smell introduction is most common in evolutionary activities. In particular, we noticed that the merge operations could drastically increase the possibility of introducing them in AI-Enabled systems. Finally, our findings suggest a lack of awareness by practitioners of the importance of monitoring quality attributes of source code as their systems evolve.

5. Threats to Validity

In this subsection, we discuss possible threats to the validity that could have affected the results and the strategies we applied to mitigate them.

Construction Validity. This threat regards the relationship between theory and observation. The crucial aspect in our case regards the dataset exploited. We are conscious that the project selection can influence the results obtained. However, to mitigate this aspect, we selected NICHE dataset—*i.e.,* a dataset manually labeled and validated by other researchers as "well engineered" projects according to 8 different dimensions. Another threat regarding the data collection phase: to mitigate this aspect, we used well-established tools—*i.e.,* PyDriller and PySmell. The first has been used to extract information on releases, while the second has to extract information on Python-specific code smells. In any case, we made all script, additional material, and row data publicly available for the sake of verifiability. While we recognize possible limitations of these two tools, they represent the state of the art.

Conclusion Validity. The main threat that can affect the conclusion validity refers to the use of PySmell to detect Python-specific code smells for AI-Enabled systems. While previous research underlines that other tools cannot work in AI-Enabled systems, no studies have been performed on PySmell. As part of our agenda, we will investigate the precision and recall of this tool on AI-Enabled systems.

External Validity. This threat is mainly connected with the generalizability of results. To mitigate this aspect, we analyzed 200 projects and 10,600 releases of open-source projects with different domains and different characteristics in terms of size, number of classes, and so on. Furthermore, we planned to conduct further analysis by increasing the number of projects and commits to assess our preliminary results.

6. Further Discussion and Take-Away Messages

The analysis of the results opens the door to several implications and take-away messages useful to increase the awareness of the presence of Python-specific code smells in AI-enabled systems. We argue our discussion points that led to deriving such practical implications.

Awareness is the key point. Based on our results, developers need to be aware of the potential impact of code smells on their systems during the software evolution process. Despite the existing body of research that emphasizes the importance of monitoring quality attributes to prevent a subsequent increase in effort [32, 23], there is still a need for further empirical investigations to determine the extent to which code smells can pose a danger.

☞ *More empirical research needs to be done to begin to make developers aware of the potential issues associated with the presence of code smells.*

Different mindsets imply different smells. The different mindsets Python developers adopt to build systems that encourage using specific code constructors to avoid loops or minimize the lines of code required to perform tasks can induce the proliferation of other code smells that are not commonly found in other programming languages. Although these smells are, in some cases, different from those identified in the literature, this does not imply that their presence cannot still lead to a decrease in software quality over time. A critical analysis of the peculiarities of the language should be performed to investigate what best practices the Python community should adopt to mitigate the presence of code smells in their systems.

☞ *A thorough analysis of the Python community is needed to understand whether what they consider best practices may be antipatterns that can cause code smells.*

7. Conclusion and future work

In this paper, we conducted a preliminary investigation on the diffusion of code smells in AI-Enabled systems in terms of frequency, density, and variation and the activities performed by developers that induced their introduction. We selected 200 AI-Enabled and over 10,600 releases. We used PySmell and PyDriller. The latter has been used to extract information on Python-specific code smells, while the former has to obtain information about releases and commit messages. The results indicated that the code smell most frequent and longest alive is Complex List Comprehension. Their variations do not follow a specific pattern over time but seem project-dependent. The activities often cause code smells are evolutionary and principally related to merge activities. Furthermore, code smells related to the incorrect use of object-oriented principles are rarely detected, and this suggests developers prefer using other reuse mechanisms to build AI-Enabled systems. Our findings underline developers' need for more awareness of monitoring quality attributes during the software evolution to avoid the incidental introduction of code smells in their projects, and a deeper investigation of the approaches adopted by Python developers to write source code is necessary to assess if these practices can induce the proliferation of code smells. As part of our agenda, we plan to manually investigate the precision and recall of PySmell on AI-Enabled systems and perform a large-scale analysis by increasing the number of projects and commits.

Acknowledgments

Fabio is partially supported by the Swiss National Science Foundation - SNF Project No. PZ00P2_-186090 (TED). This work has been partially supported by the Qual-AI and EMELIOT national research projects, which have been funded by the MUR under the PRIN 2022 and 2020 programs, respectively (Contracts 2022B3BP5S and 2020W3A5FY)

References

[1] M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68 (1980) 1060–1076. doi:10.1109/PROC.1980.11805.

[2] P. Kruchten, R. L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice, IEEE Software 29 (2012) 18–21. doi:10.1109/MS.2012.167.

[3] J. Münch, K. Schmid, Perspectives on the Future of Software Engineering, 2013.

[4] W. Cunningham, The wycash portfolio management system, in: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum), OOPSLA '92, Association for Computing Machinery, New York, NY, USA, 1992, p. 29–30. URL: https://doi.org/10.1145/157709.157715. doi:10.1145/157709.157715.

[5] M. Fowler, Refactoring: Improving the design of existing code, in: 11th European Conference. Jyväskylä, Finland, 1997.

[6] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, W. Oizumi, Jspirit: a flexible tool for the analysis of code smells, in: 2015 34th International Conference of the Chilean Computer Science Society (SCCC), IEEE, 2015, pp. 1–6.

[7] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur, Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (2009) 20–36.

[8] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, Lightweight detection of android-specific code smells: The adoctor project, in: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), IEEE, 2017, pp. 487–491.

[9] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Transactions on Software Engineering 43 (2017) 1063–1088.

[10] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, An empirical investigation into the nature of test smells, in: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, 2016, pp. 4–15.

[11] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 181–190. doi:10.1109/CSMR.2011.24.

[12] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness 17 (2012) 243–275. URL: https://doi.org/10.1007/s10664-011-9171-y. doi:10.1007/s10664-011-9171-y.

[13] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 482. URL: https://doi.org/10.1145/3180155.3182532. doi:10.1145/3180155.3182532.

[14] F. A. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: An experimental assessment., J. Object Technol. 11 (2012) 5–1.

[15] E. Van Emden, L. Moonen, Java quality assurance by detecting code smells, in: Ninth

Working Conference on Reverse Engineering, 2002. Proceedings., IEEE, 2002, pp. 97–106.

[16] G. Giordano, A. Fasulo, G. Catolino, F. Palomba, F. Ferrucci, C. Gravino, On the evolution of inheritance and delegation mechanisms and their impact on code quality, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 947–958. doi:10.1109/SANER53432.2022.00113.

[17] Z. Chen, L. Chen, W. Ma, B. Xu, Detecting code smells in python programs, in: 2016 international conference on Software Analysis, Testing and Evolution (SATE), IEEE, 2016, pp. 18–23.

[18] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, B. Xu, Understanding metric-based detectable smells in python software: A comparative study, Information and Software Technology 94 (2018) 14–29.

[19] W. Fenske, S. Schulze, D. Meyer, G. Saake, When code smells twice as much: Metric-based detection of variability-aware code smells, in: 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2015, pp. 171–180. doi:10.1109/SCAM.2015.7335413.

[20] A. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects?, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 306–315. doi:10.1109/ICSM.2012.6405287.

[21] R. Widyasari, Z. Yang, F. Thung, S. Q. Sim, F. Wee, C. Lok, J. Phan, H. Qi, C. Tan, Q. Tay, et al., Niche: A curated dataset of engineered machine learning projects in python, arXiv preprint arXiv:2303.06286 (2023).

[22] G. Giordano, G. Annunziata, A. De Lucia, F. Palomba, Understanding developer practices and code smells diffusion in ai-enabled software: A preliminary study — online appendix, https://figshare.com/s/d7b26dc76bc5c7aa06c8, 2023.

[23] Z. Soh, A. Yamashita, F. Khomh, Y.-G. Guéhéneuc, Do code smells impact the effort of different maintenance programming activities?, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, 2016, pp. 393–402. doi:10.1109/SANER.2016.103.

[24] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, On the impact of code smells on the energy consumption of mobile applications, Information and Software Technology 105 (2019) 43–55. URL: https://www.sciencedirect.com/science/article/pii/S0950584918301678. doi:https://doi.org/10.1016/j.infsof.2018.08.004.

[25] B. Van Oort, L. Cruz, M. Aniche, A. van Deursen, The prevalence of code smells in machine learning projects, in: 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN), 2021, pp. 1–8. doi:10.1109/WAIN52551.2021.00011.

[26] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, B. Xu, Understanding metric-based detectable smells in python software: A comparative study, Information and Software Technology 94 (2018) 14–29. URL: https://www.sciencedirect.com/science/article/pii/S0950584916301690. doi:https://doi.org/10.1016/j.infsof.2017.09.011.

[27] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[28] D. Spadini, M. Aniche, A. Bacchelli, Pydriller: Python framework for mining software repositories, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engi-

neering, ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 908–911. URL: https://doi.org/10.1145/3236024.3264598. doi:10.1145/3236024.3264598.

[29] N. Vatanapakorn, C. Soomlek, P. Seresangtakul, Python code smell detection using machine learning, in: 2022 26th International Computer Science and Engineering Conference (ICSEC), 2022, pp. 128–133. doi:10.1109/ICSEC56337.2022.10049330.

[30] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 1, 2015, pp. 403–414. doi:10.1109/ICSE.2015.59.

[31] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, T. Dybå, Quantifying the effect of code smells on maintenance effort, IEEE Transactions on Software Engineering 39 (2013) 1144–1156. doi:10.1109/TSE.2012.89.

[32] O. Ancán, C. Cares, Are relevant the code smells on maintainability effort? a laboratory experiment, in: 2018 IEEE International Conference on Automation/XXIII Congress of the Chilean Association of Automatic Control (ICA-ACCA), 2018, pp. 1–6. doi:10.1109/ICA-ACCA.2018.8609845.