DPy: Code Smells Detection Tool for Python

Aryan Boloori, Tushar Sharma
Dalhousie University, Canada
{aryan.boloori, tushar}@dal.ca

***Abstract*—Code smells violate best practices in software development that make code difficult to understand and maintain. Code smell detection tools help practitioners detect maintainability issues and enable researchers to conduct repository mining and empirical research involving code smells. Though significant efforts have been made to effectively detect smells in code, majority of the available tools target programming languages such as Java. Despite the most popular language, a code smell detection tool that can identify not only implementation-level code smells but also support detection of smells at the design granularity is lacking. This paper presents DPy, a code smell detection tool for Python. The tool currently supports eight design smells, eleven implementation smells, and various code quality metrics for Python code. Our replication package includes the tool, instructions to use it, all the validation data and scripts [1]**

***Index Terms*—Code smells, Python, Repository mining.**

I. Introduction

Code smells violate established design principles and best practices, typically affecting maintainability negatively [2], [3]. Research has shown that they not only significantly reduce developers' ability to comprehend the source code [4], but also increase the change-proneness and fault-proneness of affected source code entities [5]–[7]. Researchers in the field have explored various characteristics of smells, including their causes, impacts, and detection methods [3].

Code smells can be broadly classified into three main categories based on their scope and granularity—*implementation*, *design*, and *architecture* smells [8], [9]. Detecting these smells is not only important for software developers but also for researchers, as the produced code quality information is instrumental in empirical and mining studies within the software engineering domain. Given their importance, researchers and tool vendors have developed many smell detection tools supporting a wide variety of smells. These tools can be divided into five categories [3]—*metric-based* [10], [11], *rule/Heuristic-based* [12], [13], *history-based* [14], [15], *optimization-based* [16], [17], and *machine learning-based* [18], [19].

Python has emerged as a prominent programming language in recent years [20]–[22]. According to the well-known TIOBE index [20], Python is the *most* prominent programming language currently. Its versatility makes it a favorite among programmers for many different tasks, from creating websites and building software to working on artificial intelligence and data science research. Despite the popularity and relevance, the majority of academic attempts to offer a code smell detection tool support Java as their targeted programming language [3]. There have been some attempts to fill the gap; however, the current set of tools for Python focuses only on low-level syntactic issues, different quality attributes such as security, or artifacts such as tests. Specifically, Pylint [23], Pyflakes [24], and SonarQube [25] target linting and syntactic issues and do not focus on design issues arising at higher granularity. Similarly, tools such as Clone Digger [26] identify code clones and PyNose [27] detect test smells in Python code. Therefore, the community lacks a comprehensive code smell detection tool for Python that not only supports implementation-level maintainability issues but also helps developers find design issues in their code.

In this paper, we introduce DPy (pronounced as */di:pai/*), a code smell detection tool for Python. The tool supports detection of eight design smells, four of which apply to both class and module (*i.e.,* a file) in Python, an area that has received limited attention from existing tools so far. Also, the tool identifies eleven kinds of implementation smells. Furthermore, traditional code quality metrics at the function (or method), module, and class level are provided by the tool. The results of the analysis *i.e.,* detected code smells and code quality metrics are exported in CSV or JSON formats, allowing further consumption of the results.

We provide a comprehensive replication package [1] that includes the tool, detailed instructions to use and configure the tool, as well as data and results of manual validation.

II. Related Work

In this section, we first elaborate on the code smell detection tools for programming languages other than Python. Later, we focus on the existing tools specifically for Python.

A. Smell detection tools

There are numerous code smell detection tools offered by researchers and tool vendors [28], [29]. Among the five categories of smell detection tools, metric-based and rule/heuristic-based tools are most prominent and available [3]. For example, Mashiach *et al.* [30] focused on detecting 35 different code smells including design smells in C++ programs and introduced a tool called *CLEAN++*. Sharma *et al.* [31], [32] developed *Designite* and *DesigniteJava* to support detection of implementation, design, architecture, test, and testability smells in C# and Java codebases. Almashfi *et al.* [33] proposed $\text{TAJS}_{\text{lint}}$ to detect code smells in JavaScript programs. Peruma *et al.* [34] introduced *tsDetect* to find test smells in Java programs. Fontana *et al.* [35] proposed *Arcan* to detect architecture smells in Java applications. Virgínio *et al.* offered *JNose* to cover test smells in Java codes.

Similarly, several tools are offered by tool vendors. *JArchitect* [36] measures code quality metrics in Java projects and proposes suggestions to improve code quality. *CppDepend* [37] aims at finding and analyzing different code quality issues in C++ applications. *NDepend* [38] focuses on .NET programs and gives a comprehensive analysis about code quality, security issues and some other useful features such as code legacy and code reviews. *SonarQube* [25] is a widely adopted tool investigating different quality aspects of code bases in different programming languages such as C#, JavaScript, TypeScript, php.

B. Code smell detection tools for Python

Chen *et al.* [39] investigated Python code smells and proposed a code smell detection tool named *Pysmell*. The tool identifies eleven code smells, five of which are general code smells and six of them are specific to Python. Vavrová *et al.* [40] analyzed and compared nine design defects in approximately 32 million lines of code in different Python projects and scrutinized their corresponding thresholds and detection strategies. Wang *et al.* [27] focused on Python test smells and introduced a new tool called *PyNose* to detect test smells in Python code sources. Test smells are test-specific code smells [41]. Gupta *et al.* [42] analyzed the severity level of five implementation code smells in twenty Python projects with SonarQube [25]. Vatanapakorn *et al.* [43] leveraged machine learning models to predict five code smells in Python code bases. Rope [44] assists programmers in efficiently improving their code quality. While the tool's primary focus is not on code smell detection, it applies refactoring mainly associated with implementation practices. Furthermore, *Pyright* [45] is a type checker tool which focuses solely on static type checking in Python code bases. *Pylance* [46] is a Python extension in VS Code IDE that helps programmers to improve their productivity. This tool also recognizes issues such as type issues and undefined variables.

Some of the tools mentioned above are unavailable or not actively maintained. For example, *Pysmell* is no longer available publicly. The majority of the tools for Python identify syntactic issues or low-level implementation issues, completely neglecting design issues from their analysis. This effort fills the gap by supporting implementation smells and identifying design smells.

III. DPy

A. Tool architecture

Figure 1 presents the architecture of the tool. DPy parses the input source code into Abstract Syntax Tree (AST) with the help of a Python library `ast`[1]. The *source model* layer contains a hierarchical source code model for each source code entity including *project*, *package, module, class, function* and down to *statement*. Each source model entity contains the relevant information for that entity; for example, a *module* entity holds information about a Python module including the functions, classes, and statements contained in that module. This layer collects information presents in AST and resolves symbols (*i.e.,* infer type and caller-callee information). Type information for user-defined types is missing in Python since it is a dynamic language. To address the challenge, we implemented a scope-based type inference system that attempts to identify the types, modules, and functions utilizing available information (such as import statements). The top layer of the tool contains the smell detection and code quality metrics computation logic. The layer accesses the source model, identifies smells and computes metrics, and outputs the generated information in either CSV or JSON files. The behavior of the tool can be configured using a configuration file that can be specified with other arguments while invoking the tool.

[1]https://docs.python.org/3/library/ast.html

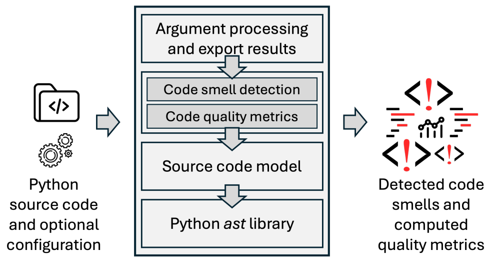
Python source code and optional configuration
Argument processing and export results
Code smell detection
Code quality metrics
Source code model
Python ast library
Detected code smells and computed quality metrics

Fig. 1. Architecture of DPy tool.

B. Code smells selection

We inherit the majority of code smells to support in our tool from DesigniteJava [31], which in turn, inherit the supported code smells from known taxonomies and catalogs, such as one proposed by Fowler *et al.* [2] and Suryanarayana *et al.* [9]. We also added two implementation smells *long message chains* and *long lambda functions* that are supported by other code smell detection tools for Python such as Pysmell [49].

We face a rather peculiar issue in selecting design smells due to the inherently different programming paradigm followed by Python from traditional object-oriented languages such as Java. Specifically, a general belief is that Python programs rarely use object-oriented features, particularly inheritance hierarchy. If it is indeed true, then we do not need to include hierarchy smells (such as deep hierarchy, or rebellious hierarchy) to DPy. To confirm the assumption, we analyzed ten most used and recognized, based on the number of stars, Python repositories. These repositories include `Numpy`, `Pandas`, `nltk`, `PyTorch`, and `Django`. The complete list of the repositories and computed metrics, including number of modules and classes, and DIT can be found in our replication package. The selected repositories have an average of $1,411$ modules (*i.e.,* Python files) and $3,835$ classes. Among the classes, we found 42% classes on average with DIT $>= 1$ (*i.e.,* at least one super class) and 17% with DIT $>= 2$. These values suggest that hierarchy is an important aspect of Python programs that we cannot discard. Therefore, we include the support for hierarchy smell detection in DPy.

TABLE I
DESIGN SMELLS SUPPORTED BY DPY

Design smell	Scope	Description	Detection rule
Multifaceted abstraction [8], [9]	Class, module	An abstraction has more than one responsibility assigned to it.	We detect this smell when a non-trivial abstraction (NOM$>$ 3) shows low degree of cohesion (LCOM$>=$ 0.8) [8], [31], [38]
Insufficient modularization [8], [9]	Class, module	An abstraction is not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both.	When an abstraction is large in terms of implemented methods (NOPM $>$ 20 or NOM$>$ 30), or overly complex (WMC$>$ 100) [8], [31], [47]
Hub-like modularization [8], [9]	Class, module	An abstraction has both incoming and outgoing dependencies with a large number of other abstractions	When an abstraction has excessive incoming (FAN-IN$>$ 7) and outgoing (FAN-OUT$>$ 7) [48] dependencies
Broken modularization [8], [9]	Class, module	Data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions	When an abstraction defines data members (NOF$>$ 4) but do not implement any methods that use the data members (NOM$=$ 0) [31].
Deep hierarchy [8], [9]	Class	An inheritance hierarchy is excessively deep	When depth of inheritance is large (DIT $>$ 6) [31]
Broken hierarchy [8], [9]	Class	A supertype and its subtype conceptually do not share an "IS-A" relationship	We detect this smell when both the types do not share any method names and the supertype do not declare any abstract method overridden by its subclasses
Rebellious hierarchy [8], [9]	Class	A subtype rejects the methods provided by its super-type(s)	This smell gets detected when an overridden method in a subtype rejects the method implementation *i.e.,* contains only `pass`, `return`, or `raise` statement [31]
Wide hierarchy [8], [9]	Class	An inheritance hierarchy is too wide	When a class has too many sub-classes (NC $>$ 5) [31]

C. Smell detection strategies

DPy supports detection of eleven implementation smells and eight design smells. We summarize the supported smells and corresponding detection mechanism in Table I and Table II.

Verbosity conversion: We inherit commonly used metric thresholds as used in literature for most of the cases. However, it is unsound to use the same size-related thresholds for Python code from literature that are prescribed for Java or C++. We cannot adopt those thresholds in our tool because Python's inherently concise syntax leads to significantly different code verbosity compared to other programming languages. For instance, while a method length threshold of 100 lines works for Java, this same metric threshold cannot be directly applied to Python due to its more concise programming style. We need a suitable approach that accounts for Python's inherently less verbose nature.

To find the verbosity factor, we analyzed the RosettaCode repository [50]. This repository contains many programming problems and corresponding solutions in different programming languages. We identified all programming problems where both Python and Java solutions were present in the repository. We counted the lines of code of all the selected problems in both Java and Python. We excluded comments and blank lines of the code files. We analyzed $1,226$ problems and obtained the rounded LOC averages of 47 and 31 for Java and Python code respectively. Based on the analysis, we derive 0.67 as the verbosity factor. Therefore, a metric threshold in Java set to 100 can be set to 67 for the Python code.

D. Code quality metrics

DPy computes common code quality metrics and exports them in CSV and JSON formats. These metrics not only used by the tool to detect various smells but also aid developers understand their program better and enhance their code quality. At class and module level the supported metrics are: Lines of Code (LOC), Weighted Methods per Class (WMC), Number of (Public) Methods (NOM, and NOPM), Number of (Public) Fields (NOF and NOPF), Lack of Cohesion among Methods (LCOM), Number of incoming and outgoing dependencies (FAN-IN and FAN-OUT), and Depth of Inheritance (DIT, applicable to only classes). The tool computes the following method-level metrics: Lines of Code (LOC), Cyclomatic Complexity (CC), and Parameter Count (PC).

E. Usage

DPy is a console application that can be run on Windows, Linux, and Mac operating systems. A project located at `/path/to/project` can be analyzed using the following command.

```
./DPy analyze –i /path/to/project –o /path/to/out/folder
```

The command analyze the Python project and generates the smells and metrics output in the default (JSON) format in the specified output folder.

F. Tool validation

This section elaborates the human evaluation process and results for DPy. We defined the following selection criteria to identify the subject systems: number of commits more than $5,000$, different domains (security, utility and AI), number of contributors more than 20, and lines of code more than $5,000$ but less than ten thousand to keep manual analysis manageable. We used SEART GITHUB search [53] to find the repositories. After applying the selection criteria, we identified Codespell [54], Mava [55], and Maltrail [56]. We found that Python programmers tend to implement their logic using functions that typically follow a non-object-oriented programming style, leading to fewer class-level design smells. To cover more design smells in our evaluation, we added another project Elevant [57] by manually checking the presence of object-oriented code in the projects one-by-one selected by applying our initial selection criteria.

Two evaluators manually examined the source code of the selected subject systems and documented the supported code smells that they found. Both evaluators are graduate students and familiar with code smells and design principles. They were allowed to use IDE features (such as "find" and "find

TABLE II
IMPLEMENTATION SMELLS SUPPORTED BY DPY

Implementation smell	Description	Detection rule
Long statement	An overly long statement	When a statement has more than 80 characters
Long parameter list [39]	A function with too many input arguments	When a function takes more than four parameters
Long method [39]	A long method	When the size of a function is more than 67 lines
Long identifier	A long identifier (*i.e.,* function, class, and field, or local variable name)	When the length of an identifier is more than 20 characters
Empty catch block	A try-catch block with empty except block	When an `except` block contains only `pass` or a `return` statement
Complex method	An overly complex function	When the McCabe's Cyclomatic Complexity [51] of a function is more than 7 [52]
Complex conditional [2]	A conditional statement with too many logical operators	When the number of logical operators (`and` and `or`) is more than 2 in a single conditional statement
Missing default	A `match-case` statement with no default case	When there is not a default case (*i.e.,* `case _` block) in Python `match-case` statement
Long lambda function [39]	An excessively long lambda function	When the length of a `lambda` function is more than 80 characters [39]
Long message chain [39]	A long series of method calls are chained together	When more than two methods are chained together [39]
Magic number	Any undefined number	When there is a numerical literal, except commonly used 0, -1, and 1, without any definition for that

usage" (of a variable)) and external tools to collect code quality metrics to help them narrow their search space. Both evaluators carried out their analyses independently. After completing their manual analysis, they matched their findings to spot any differences. We used *Cohen's Kappa* to measure the inter-rater agreement between the evaluators. The obtained result, $\kappa = 0.87$ on average among all the smells, shows a strong agreement between the evaluators. The evaluators discussed the rest of their findings and resolved the conflicts.

We used our tool on the subject systems and identified code smells. We manually matched the ground truth prepared by the evaluators and tool's results. We classified each smell instance as true positive (TP), false positive (FP), and false negative (FN). We obtained precision=0.96 and recall=0.93. Table III presents summary of the results of the manual evaluation for the supported smells. The detailed smell-specific results can be found in our replication package [1].

TABLE III
RESULTS OF MANUAL VALIDATION; MVI STANDS FOR MANUALLY VERIFIED INSTANCES

Code Smells	MVI	TP	FP	FN
Implementation smells	899	838	33	61
Design smells	30	30	0	0
Total	**929**	**868**	**33**	**61**

As Table III shows, though the tool performed well in general, we observed some false positives and false negatives in identifying implementation smells. This is primarily due to two factors. First, DPy does not support nested entities, leading to the omission of certain smells embedded within them. Second, implementation smells often involve numerous edge cases that require additional handling. For example, when a statement is split into multiple lines, we need to consider all the fragments of the statement to determine whether the entire length of the statement is crossing the threshold. We plan to address these issues in future releases. In contrast, we did not observe any inaccuracies with design smells.

IV. THREATS TO VALIDITY

Construct validity measures the degree to which tools and metrics actually measure the properties that they are supposed to measure. In dynamic languages such as Python, type information is missing and that may influence the accuracy of the tool. To mitigate this concern, first, we implemented a scope-based type inference system that attempts to identify the types, modules, and functions utilizing available information (such as import statements). We also employed a comprehensive set of tests for DPy to rule out obvious deficiencies. Additionally, we found the results of tool's manual validation very satisfactory.

It is common for smell detection tools, including DPy to use various metric thresholds to detect smells. It is a known and accepted fact that there is no one globally accepted threshold set for various metrics [58]. We chose the thresholds that are commonly used by the software engineering community. To account the conciseness of the language, we conducted an experiment to compute the verbosity factor. Moreover, we made the thresholds customizable within the tool to let users choose a set of appropriate thresholds.

V. CONCLUSIONS

We propose a new tool DPy to detect code smells at implementation and design granularity. The tool, in addition to eight design and eleven implementation smells, provides commonly used code quality metrics for a comprehensive code quality analysis.

Limitations: Currently, the tool does not support nested class definitions. Also, the type and symbol resolution mechanism that identifies the types and functions associated with symbols statically is limited by our own implementation based on incrementally increasing scope rules. Though it performs well, we would like to extend its evaluation and improve its ability to associate the correct types and functions with symbols.

Future work: In the future, apart from addressing the known limitations, we would like to extend the scope of the tool. It includes widening the scope of supported smells to other design and implementation smells as well as introducing support to detect test and architecture smells. We also aim to integrate the tool with build systems and continuous integration pipelines.

References

[1] *DPy: Code Smells Detection Tool for Python*. Zenodo, Dec. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.14279535

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*, ser. Addison-Wesley signature series. Addison-Wesley, 2019.

[3] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[4] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," in *Advances in Computers*. Elsevier, 2011, vol. 82, pp. 25–46.

[5] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, Jun 2012.

[6] F. Palomba and A. Zaidman, "Notice of retraction: Does refactoring of test smells induce fixing flaky tests?" 2017, Retracted, p. 1 – 12.

[7] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[8] T. Sharma, P. Singh, and D. Spinellis, "An empirical investigation on the relationship between design and architecture smells," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4020–4068, Sep. 2020.

[9] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed. Morgan Kaufmann, 2014.

[10] R. Marinescu, "Measurement and quality in object-oriented design," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, Dec. 2005, pp. 701–704.

[11] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, 2014.

[12] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[13] T. Sharma, P. Mishra, and R. Tiwari, "Designite — A Software Design Quality Assessment Tool," in *Proceedings of the First International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities*, ser. BRIDGE '16. ACM, 2016.

[14] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[15] S. Fu and B. Shen, "Code Bad Smell Detection through Evolutionary Data Mining," in *International Symposium on Empirical Software Engineering and Measurement*. IEEE, Nov. 2015, pp. 41–49.

[16] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-Smell Detection as a Bilevel Problem," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 1, pp. 6–44, Oct. 2014.

[17] A. Ouni, R. G. Kula, M. Kessentini, and K. Inoue, "Web Service Antipatterns Detection Using Genetic Programming," in *GECCO '15: Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. ACM, Jul. 2015, pp. 1351–1358.

[18] U. Azadi, F. A. Fontana, and M. Zanoni, "Poster: Machine learning based code smell detection through wekanose," in *IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 288–289.

[19] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918302623

[20] T. Software, "TIOBE Index," 2024. [Online]. Available: https://www.tiobe.com/tiobe-index/

[21] S. Overflow, "Stack overflow developer survey," 2024. [Online]. Available: https://survey.stackoverflow.co/2024/

[22] G. van Rossum, "Python's Popularity and its Use in Data Science and AI," 2020. [Online]. Available: https://www.python.org/doc/essays/

[23] "Pylint," https://pypi.org/project/pylint/.

[24] "Pyflakes," https://pypi.org/project/pyflakes/.

[25] "SonarQube," https://www.sonarsource.com/.

[26] "Clone Digger," https://pypi.org/project/clonedigger/.

[27] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed, "PyNose: A Test Smell Detector For Python," in *36th IEEE/ACM ASE*, 2021, pp. 593–605.

[28] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, p. 7, Oct 2017.

[29] U. Azadi, F. Arcelli Fontana, and D. Taibi, "Architectural smells detected by tools: a catalogue proposal," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2019, pp. 88–97.

[30] T. Mashiach, B. Sotto-Mayor, G. Kaminka, and M. Kalech, "Clean++: Code smells extraction for c++," in *IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 441–445.

[31] T. Sharma, P. Mishra, and R. Tiwari, "Designite - a software design quality assessment tool," in *IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE)*, 2016, pp. 1–4.

[32] T. Sharma, "Multi-faceted Code Smell Detection at Scale using DesigniteJava 2.0," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, p. 284–288.

[33] N. Almashfi and L. Lu, "Code smell detection tool for java script programs," in *5th International Conference on Computer and Communication Systems (ICCCS)*, 2020, pp. 172–176.

[34] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "tsDetect: an open source test smells detection tool," in *Proceedings of the 28th ACM ESEC/FSE*, 2020, p. 1650–1654.

[35] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 282–285.

[36] "JArchitect," https://www.jarchitect.com/.

[37] "CppDepend," https://www.cppdepend.com/.

[38] "NDepend," https://www.ndepend.com/, 2024.

[39] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in python programs," in *International Conference on Software Analysis, Testing and Evolution (SATE)*, 2016, pp. 18–23.

[40] N. Vavrová and V. Zaytsev, "Does python smell like java? tool support for design defect discovery in python," *The Art, Science, and Engineering of Programming*, vol. 1, no. 2, Apr. 2017.

[41] A. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," 08 2001.

[42] A. Gupta, R. Gandhi, N. Jatana, D. Jatain, S. K. Panda, and J. V. N. Ramesh, "A severity assessment of python code smells," *IEEE Access*, vol. 11, pp. 119 146–119 160, 2023.

[43] N. Vatanapakorn, C. Soomlek, and P. Seresangtakul, "Python code smell detection using machine learning," in *26th International Computer Science and Engineering Conference (ICSEC)*, 2022, pp. 128–133.

[44] "Rope," https://github.com/python-rope/rope.

[45] "Pyright," https://microsoft.github.io/pyright/#/.

[46] "Pylance," https://pypi.org/project/pylance/.

[47] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.

[48] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012.

[49] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in python programs," in *International Conference on Software Analysis, Testing and Evolution (SATE)*, 2016, pp. 18–23.

[50] "Rosetta code data," https://github.com/acmeism/RosettaCodeData.

[51] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[52] N. Almashfi and L. Lu, "Code smell detection tool for java script programs," in *5th International Conference on Computer and Communication Systems*, 2020, pp. 172–176.

[53] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR*, 2021, pp. 560–564.

[54] "Codespell," https://github.com/codespell-project/codespell.

[55] "Mava," https://github.com/instadeepai/Mava.

[56] "Maltrail," https://github.com/stamparm/maltrail.

[57] "Elevant," https://github.com/ad-freiburg/elevant.

[58] R. Fourati, N. Bouassida, and H. B. Abdallah, "A metric-based approach for anti-pattern detection in uml designs," in *Computer and Information Science*, R. Lee, Ed., 2011.