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

code smell instances tends to arise slightly during multiple releases, which implies that Python programs may be suffering code smells further.

The remainder of this paper is structured as follows. Section Ⅱ introduces the definition and the detection of code smells in Python code. Section Ⅲ describes our empirical study and presents the results. Section Ⅳ discusses related work and section Ⅴ concludes our work.

II. METHODOLOGY

Before we can perform automatic detection of code smells in Python programs, there are two important issues that need to be discussed: the list of Python smells that we will detect and the detection strategy.

A. Definition of Python Smells

Fowler et al. [3] presented a list of code smells recognized by the public to look for potential refactoring. These code smells range from simple bad patterns such as "Large Class" to more complex ones such as "Shotgun Surgery". Subsequently, most researches into code smells in object-oriented programs target similar smells with Fowler's. However, Python supports multiple programming paradigms and introduces flexible grammatical constructs, thus these generic code smells are obviously insufficient. To be complementary, we have studied various resources including online resources [27-29] and reference manuals [30, 31] to look for other bad coding patterns in Python code. In this paper we predefine a list of Python smells which consist of several generic code smells appropriate for Python and some additional ones in Python code. All Python smells in this category violate the principle of maintainability in different aspects.

The generic Python smells include the following 5 items. The first four items were originally proposed by Fowler et al. [3] and the last one was described by Fard and Mesbah [33].

Large Class (LC): A class that has grown too large.

Long Parameter List (LPL): a method or a function that has a long parameter list.

Long Method (LM): a method or a function that is too long.

Long Message Chain (LMC): an expression that is accessing an object through a long chain of attributes or methods by the dot operator.

Long Scope Chaining (LSC): a method or a function that is multiply-nested.

It is worth noting, the definition of some generic smells here are slightly different from their originals on account of the nature of Python. For Long Message Chain, previous work only focused on a sequence of methods. However, each attribute/method of classes or instances in Python is an object and has its own attributes/methods, thus both attributes and methods can constitute a long chaining expression. Next we introduce 6 additional Python smells mentioned in [27-31].

Long Base Class List (LBCL): a class definition with too many base classes. Unlike some other dynamic languages such as Ruby and JavaScript, Python supports a limited form of multiple inheritance. If an attribute in Python is not found in the derived class during execution, it is searched recursively in the base classes declared in the base class list in sequence. Too long base class list will limit the speed of interpretive execution. Besides, as derived classes may override methods of their base classes, and methods in base classes have no special privacy and privileges in Python, a class which inherits too many base classes is complicated and dangerous [30]. For the sake of designing reliable and extensible classes, if a class definition has a long base class list, it is reasonable to consider the possibility of merging some base classes.

Useless Exception Handling (UEH): a try...except statement that does little. A try statement in Python can have more than one except clause to specify handlers for different selected exceptions. There are two kinds of writing styles which make exception handling useless. First, it has only one except clause and it catches a too general exception such as *Exception* and *StandardError* or even can catch all exceptions. The damage of this writing style is hiding the true exception in the try clause [27, 28, 30]. Second, all exception clauses in the statement are empty, which means whatever exceptions in the try clause will not be responded. It is never wise to ignore the selected exception [28, 30]. The right way of handling exceptions in Python code is always catching specific exceptions and carefully dealing them.

Long Lambda Function (LLF): a lambda function that is overly long, in term of the number of its characters. Lambda is a powerful construct that allows the creation of anonymous functions at runtime. A lambda function can only contain one single expression. If lambda is overly long (i.e. it contains too complex operations), it turns out to be unreadable and loses its benefits [28]. In order to avoid too many one-expression long functions in Python programs, lambda should only be used in packaging special or non-reusable code, otherwise explicit *def* statements can always take the place of them.

Complex List Comprehension (CLC): a list comprehension that is too complex. List comprehensions in Python provide a concise and efficient way to create new lists, especially where the value of each element is dependent on each member of another iterable or sequence, or the elements satisfy a certain condition. However, when list comprehensions contain complex expressions, they are no longer clear. Apparently, it is hard to analyze control flows of complex list comprehensions. Therefore, if list comprehensions contain multiple loops or conditions, they are recommended to be replaced with traditional control statements and loops [28].

Long Element Chain (LEC): an expression that is accessing an object through a long chain of elements by the bracket operator. Long Element Chain is directly caused by nested arrays. It is unreadable especially when a deep level of array traversing is taking place [31]. One factor of this bad programming custom is applying nested list comprehensions. The occurrence results from the fact that the initial expression in a list comprehension can be any arbitrary expression, including another list comprehension. Similar with Long Message Chain, in order to avoid nested arrays, one alternative is to reduce array dimensionality.

Long Ternary Conditional Expression (LTCE): a ternary conditional expression that is overly long. The ternary operator defines a new conditional expression in Python, with the value of the expression being *X* or *Y* based on the truth value of *C* in the form of "*X* if *C* else *Y*". The motivation of its addition to Python 2.5 was the prevalence of error-prone attempts to using *and* and *or*. However, it is rather long when it involves other constructs such as lambda functions. Saving a few characters in it would be difficult when it contains several long variable names. As a result, though the ternary conditional expression is a concise way of the conditional expression to help avoid ugly, a long one is unreadable [28, 30]. The solution to refactoring is to transform it into a traditional conditional expression.

Code smells are always subjective as they are often based on experiences and opinions. The sources of Python smells include traditional patterns and those we thought to be useful indicators of potential harmful aspects of Python code. Unfortunately, there is little concrete evidence that can support the damages caused by code smells. Take LLF for example, some developers suggest minimizing the use of lambda functions is advisable and regard a long one as a code smell, while some others consider lambda functions cause no harm in maintenance and contribute to the convenience of Python programming. What is worse, this list of code smells can never be complete. There will always be domains and applications where a different collection of code smells should be applied. For instance, we find that UEH often occurs in test code.

B. Detection of Python Smells

1) Detection Strategy

A common heuristic based approach to detecting code smells is the use of code metrics and defined thresholds [16-18, 33]. Similarly, we adopt a metric-based smell detection strategy. Some metrics and thresholds in detecting generic code smells have been proposed in previous work, but are inappropriate for Python code. We rely on programming experiences and literature description to determine the metrics and criteria for Python smells, which are shown in TABLE I.

2) Implementation

We implemented Pysmell to detect code smells in Python programs. The tool was designed following the architecture shown in Figure 1. There are three main components in Pysmell.

- **Code Extractor**. During this procedure, the tool checks the architecture of the Python system, and then identifies useful Python files and program entities.
- **AST Analyzer**. The extracted code is fed into this component. It parses the code into an Abstract Syntax Tree and then analyzes it by traversing the tree. Furthermore, it collects different metrics together with the information of corresponding patterns including their locations in files.
- **Smell Detector**. Pysmell is configurable by users to support different criteria and thresholds in detecting Python smells. Based on the metrics AST Analyzer has collected, this component applies them to user-defined criteria to identify code smell instances.

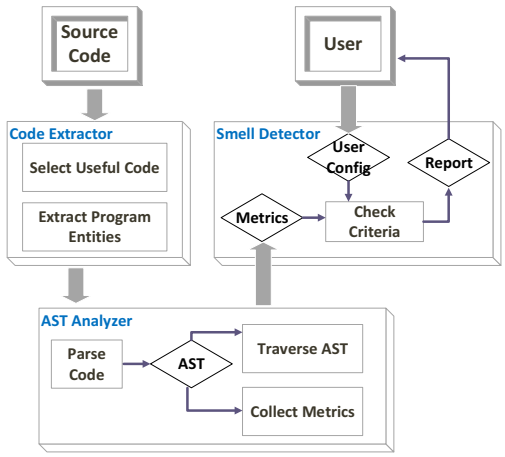

Figure 1. Architecture of Pysmell.

TABLE II. SUBJECT PROJECTS

Name	LOC	#Files	Description
django	214,997	2,106	a high-level Python Web framework
ipython	105,522	788	a command shell for interactive computing for Python
matplotlib	135,459	815	a python 2D plotting library
scipy	173,714	522	a python-based tool for mathematics, science, and engineering
numpy	131,854	361	a fundamental package for scientific computing with Python

TABLE I. METRIC-BASED CRITERIA FOR PYTHON SMELLS

Code Smell	Level	Criteria	Metric
Large Class (LC)	Class	LOC >= 200 or NOA+NOM > 40	LOC: lines of code; NOA: number of attributes; NOM: number of methods
Long Parameter List (LPL)	Function	PAR >= 5	PAR: number of parameters
Long Method (LM)	Function	MLOC >= 100	MLOC: method lines of code
Long Message Chain (LMC)	Expression	LMC >= 4	LMC: length of message chain
Long Scope Chaining (LSC)	Function	DOC >= 3	DOC: depth of closure
Long Base Class List (LBCL)	Class	NBC >= 3	NBC: number of base classes
Useless Exception Handling (UEH)	Code Block	(NEC = 1 and NGEC = 1) or NEC = NEEC	NEC: number of except clauses; NGEC: number of general exception clauses; NEEC: number of empty except clauses
Long Lambda Function (LLF)	Expression	NOC >= 80	NOC: number of characters in one expression
Complex List Comprehension (CLC)	Expression	NOL + NOCC >= 4	NOL: number of loops; NOCC: number of control conditions
Long Element Chain (LEC)	Expression	LEC >= 3	LEC: length of element chain
Long Ternary Conditional Expression (LTCE)	Expression	NOC >= 40	NOC: number of characters in one expression

III. Empirical Study and Evaluation

The aim of this study is to find out the characteristics of code smells appearing in Python programs. The empirical study is performed by Pysmell provided with the detection strategy in TABLE I. We outline three research questions in this study as follows.

RQ1: How does Pysmell perform in detecting Python smells?

RQ2: How do code smells appear in Python programs?

RQ3: How do code smells evolve in different releases of Python systems?

A. Experimental Processing

We perform the experiment using five open-source Python systems: *django*, *ipython*, *matplotlib*, *scipy* and *numpy* which belong to different domains. Some information on these systems is presented in TABLE II. We can see from the table that all target systems are fairly large. We use Pysmell to detect code smells in these systems in our experiment, with the metrics and thresholds configured as TABLE I.

To answer RQ1, tool performance is assessed by the precision and the recall of Pysmell behaving in this experiment. Precision evaluates the number of true smells identified among all detected smells, while recall evaluates the number of true smells identified among all existing smells. Two masters and two Ph.D. students were asked to check smell occurrences in subject projects manually. Then we compared the result of manual inspection and that of Pysmell to calculate precision and recall. The whole task cost about 80 person-hours.

To answer RQ2, we explore the prevalence of each code smell in subject projects. On the one hand, we discover which code smells tend to be more prevalent in Python programs than others. On the other hand, we figure out how terribly Python projects are suffering code smells.

To answer RQ3, we measure code smell instances in different releases of each Python system. As bad programming customs damage the maintainability of the software, to facilitate the evolution of the Python system, it is useful to know if code smells in Python code increase or decrease along with multiple releases of the system. Therefore, we select 22 versions of each subject system respectively according to release time intervals. The sizes of selected versions of one system are near the same, thus we ignore the effects of program sizes. Then we count the total number of code smell instances contained in each selected version. Accordingly, we can conclude the trend of code smells in Python systems through the changes of code smell instances in different releases.

B. Empirical Results

This subsection describes the results of the empirical study we did to answer above research questions.

RQ1: the performance of Pysmell

In TABLE III, the last two rows report the precision and the recall of each kind of Python smell detected in subject projects. Time overheads of our experiment are really low, thus they are not reported. Our results show that Pysmell has an average precision of 97.7% and an average recall of as high as 100% in detecting Python smells, which validates its effectiveness. We manually find out three false positives. Two of them lie in Large Class detection. This is due to calculation issues in our tool. In both cases, the metrics calculated by Pysmell are a little higher than true values, which makes our tool wrongly identify them as code smells. The other false positive is produced in detecting Long Element Chain. It results from the fact that the ways of accessing elements are mixed in special cases.

RQ2: the appearance of Python smells

To investigate the prevalence of Python smells, TABLE III also gives the number of smell instances detected in the system, and the figure in parentheses denotes its proportion in all kinds of code smell instances in the system. Pysmell finds 285 smell instances in five subject systems, and among them *django* and *matplotlib* contain the most ones. Apparently, programmers tend to use unfamiliar syntax constructs new in Python cautiously to avoid bad writing patterns. From the seventh row of TABLE III we can see from the table that LM and LC are the most prevalent code smells in Python programs, followed by LSC and LPL. In contrary, Pysmell can only identify less than 5 smell instances for CLC, LEC and LTCE. Especially, CLC does not appear in these systems. However, these code smells may become prevalent in other Python programs. In order to provide users with more choices, Pysmell implements the detection of all 11 Python smells.

RQ3: the evolution of Python smells

Figure 2 gives the chart of code smell instances in 22 releases of each system. In Figure 2, *django*, *matplotlib* and *scipy* contain more code smells and the latter halves of the three all meet an arising. Additionally, the number of code smells detected in *ipython* fluctuates along with 22 releases, but it still arises from 42 to 47. Finally, the number of Python smells in *numpy* stays relatively stable. In summary, the changes of code smells in history are not significant in Python systems, but the overall trends show increasing. According to these results, there is a possibility that Python smells are associated with system development. In particular, developers seem to have not paid sufficient attention to code smells in Python code even after numerous releases of the system. Instead, Python code may be suffering code smells further and further during the evolution. It stimulates and values the research of this paper in return.

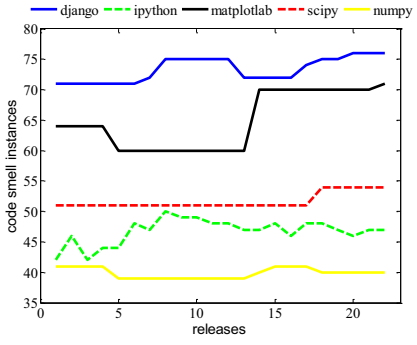

Figure 2. Code smell instances in different releases of five Python systems

TABLE III. THE NUMBER OF SMELL INSTANCES DETECTED IN SUBJECT PROJECTS

	LC	LPL	LM	LMC	LSC	LBCL	UEH	LLF	CLC	LEC	LTCE	Total	#smell types
django	18 (23.7%)	2 (2.6%)	14 (18.4%)	1 (1.3%)	24 (31.6%)	6 (7.9%)	0	11 (14.5%)	0	0	0	76	7
ipython	16 (33.3%)	1 (2.1%)	8 (16.7%)	0	10 (20.1%)	1 (2.0%)	11 (22.9%)	0	0	0	1 (2.1%)	48	7
matplotlib	17 (24.3%)	11 (15.8%)	23 (32.9%)	4 (5.7%)	6 (8.6%)	2 (2.9%)	7 (10.0%)	0	0	0	0	70	7
scipy	4 (7.5%)	24 (45.3%)	14 (26.4%)	0	1 (1.9%)	0	6 (11.3%)	0	0	4 (7.5%)	0	53	6
numpy	7 (18.4%)	3 (7.9%)	23 (60.5%)	1 (2.6%)	2 (5.3%)	0	2 (5.3%)	0	0	0	0	38	6
Total	62 (21.8%)	41 (14.4%)	82 (28.8%)	6 (2.1%)	43 (15.1%)	9 (3.2%)	26 (9.1%)	11 (3.9%)	0	4 (1.4%)	1 (0.4%)	285	10
#smelly systems	5	5	5	3	5	3	4	1	0	1	1	5	-
Precision	97.6%	100%	100%	100%	100%	100%	100%	100%	-	80%	100%	97.7%	-
Recall	100%	100%	100%	100%	100%	100%	100%	100%	-	100%	100%	100%	-

The figure in parentheses indicates its proportion in all kinds of smell instances detected in the system.

C. *Threats to Validity*

In order to evaluate the performance of Pysmell, we perform manual inspection to identify all true smell instances in subject projects. This work is time-consuming especially when the size of the suspected program is large and the specifications of code smells are not restrictive enough. As future work, we will ask some skilled developers to confirm the findings and validate on other Python systems.

Moreover, it is never precise of detecting Python smells. The reason is that the approach to detecting Python smells in this paper is metric-based. The criteria for demanding a refactoring is subjective. One needs to decide the actual criteria before performing Python smells detection, for example, what is the maximum size of methods that is allowed. The thresholds selected in this paper generally depend upon experiences from skilled programmers and resources from the website [27-29] and literature [30, 31]. Based on this consideration, Pysmell allows user-defined detection thresholds and criteria.

IV. RELATED WORK

A. *Python Analysis*

There have been some empirical studies for dynamic behavior of Python programs in literature. Holkner and Harland [34] investigated the use of Python dynamic features in open programs and found that most dynamic activities occur at program startup instead of the whole lifetime. What is in slight contrast to [34], the results of Åkerblom et al. [35] show that dynamic behavior is neither buried in Python library, nor prominently occurs during program startup phases.

Typical analysis techniques to support development of Python applications also have been proposed. Chen et al. [36] proposed a constraint based approach named PyConcept in order to check type related bugs in Python code. Gorbovitski et al. [37] proposed a flow- and trace- sensitive may-alias analysis mechanism for full Python. A hybrid dynamic slicing approach was proposed in [32], which relies on a modified Python bytecode interpreter to capture data dependences. Similarly, Chen et al. [38] also presented an information flow control approach for Python. Xu et al. [39] described a static slicing approach for Python first-class objects. The work analyzes the definitions of first-class objects and then constructs the dependence model for system slicing. However, there have been no researches in detecting code smells in Python code.

B. *Automated Detection of Code Smells*

Fowler et al. [3] described 22 code smells referring to low-level design problems textually and suggested that developers should adopt refactoring. Apparently, manual detection of code smells is always time-consuming and error-prone, and it is nearly impossible to be performed in large code bases, thus multiple automated techniques have been proposed in the past a few decades. Munro et al. [16] described a metric-based approach to smell detection by specifying each smell then identifying heuristics based on metrics. Marinescu et al. [17] and Rao and Reddy [18] also developed a metric-based approach to smell detection in object oriented programs. Emden et al. [24] presented a prototype code smell browser called jCosmo which can detect and visualize code smells in Java source code. Their approach can be used in software inspection. Moha et al. [26] proposed DECOR, a method that allows the specification and detection of code smells to support automatically generation of detection algorithms. However, none of these studies involve language-specific code smells. JSNose [33] is a tool similar to our work involving the dynamic language. Their metric-based approach combines static and dynamic analysis to detect smells in JavaScript code. However, although they also described additional code smells in JavaScript, their work focuses on web applications but neglects other features of JavaScript. Our work can identify various Python code smells in the consideration of Python nature ranging from first-class objects to confusing constructs.

V. CONCLUSIONS

Detecting code smells plays an essential role in improving the quality of software systems, facilitating their evolution, and thus reducing the overall cost of development and maintenance. However, most of previous work ignored the detection of code smells in dynamic languages. In this paper, we introduced 11 code smells in Python code and proposed a metric-based technique in detecting them. In the context of the detection

technique, we implemented a tool called Pysmell which is supported by user configuration. After that, we applied Pysmell to five large systems and discussed its performance. We also explored the appearance of each code smell by investigating the prevalence. In addition, our empirical study showed that the number of code smells tends to increase along with multiple releases of the Python system, which implies Python programs may be suffering code smells further during the evolution.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns—Elements of Reusable Object-Oriented Software, first ed. Addison-Wesley, 1994.

[2] R. S. Pressman. Software Engineering: A Practitioner's Approach. McGraw Hill, June 2000.

[3] M. Fowler, J. Brant, W. Opdyke, D. Roberts. Refactoring: Improving the Design of Existing Code. XP/Agile Universe 2002: 256.

[4] F. Khomh, M. Di Penta and Y.-G. Guéhéneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*. 2009:75–84.

[5] S. M. Olbrich, D. Cruzes and Dag I. K. Sjøberg. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM)*. 2010:1-10.

[6] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*. 2008:227–236.

[7] I. S. Deligiannis, I. Stamelos, L. Angelis et al. A controlled experiment investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software (JSS). 2004, 72(2): 129-143.

[8] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of Systems and Software (JSS). 2007, 80(7): 1120–1128.

[9] Dag I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, T. Dybå. Quantifying the Effect of Code Smells on Maintenance Effort. IEEE Transactions on Software Engineering (TSE). 2013, 39(8): 1144-1156.

[10] M. Abbes, F. Khomh, Y.-G. Guéhéneuc et al. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*. 2011:181-190.

[11] Bart Du Bois, Serge Demeyer, Jan Verelst, TomMens, and Marijn Temmerman. Does God class decomposition affect comprehensibility? In *Proceedings of the IASTED International Conference on Software Engineering*. 2006:346–355.

[12] M. D'Ambros, A. Bacchelli, and M. Lanza. On the Impact of Design Flaws on Software Defects. In *Proceedings of the 10th International Conference on Quality Software (QSIC)*. 2010:23–31.

[13] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. 1999. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1999: 47–56.

[14] M. V. Mäntylä, J. Vanhanen, and C. Lassenius. 2004. Bad smells - Humans as code critics. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*. 2004: 399–408.

[15] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw. Building empirical support for automated code smell detection. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 2010: 8.

[16] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In *Proceedings of the 11th IEEE International Symposium on Software Metrics*. 2005: 15–15.

[17] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*. 2004: 350–359.

[18] A. A. Rao and K. N. Reddy. Detecting bad smells in object oriented design using design change propagation probability matrix. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*. 2008: 1001–1007.

[19] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla. Code smell detection: Towards a machine learning-based approach. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*. 2013: 396–399.

[20] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. B. Chikha. Competitive coevolutionary code-smells detection. In *Proceedings of the 5th International Symposium on Search Based Software Engineering*. 2013: 50–65.

[21] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *Proceedings of the 9th International Conference on Quality Software (QSIC)*. 2009: 305–314.

[22] H. Liu, X. Guo, and W. Shao. Monitor-based instant software refactoring. IEEE Transactions on Software Engineering (TSE). 2013, 39(8): 1112–1126.

[23] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*. 2013: 268–278.

[24] Eva van Emden and L. Moonen. Java Quality Assurance by Detecting Code Smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*. 2002: 97-106.

[25] M. F. Sanner. Python: a programming language for software integration and development. J Mol Graph Model. 1999, 17(1): 57-61.

[26] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*. 2010, 36(1): 20–36.

[27] G. van Rossum, B. Warsaw, and N. Coghlan. Style Guide for Python Code. http://legacy.python.org/dev/peps/pep-0008/.

[28] A. Patel, A. Picard, E. Jhong, J. Hylton, M. Smart, and M. Shields. Google Python Style Guide. http://google-styleguide.googlecode.com/svn/trunk/pyguide.html.

[29] C. Lignos. Anti-Patterns in Python Programming. http://lignos.org/py_antipatterns/.

[30] D. M. Beazley. Python Essential Reference. Addison-Wesley Professional, 2009.

[31] M. Lutz. Programming Python. "O'Reilly Media, 2010.

[32] Z. Chen, L. Chen, Y. Zhou, Z. Xu, W. C. Chu, and B. Xu. Dynamic Slicing of Python Programs. In *Proceedings of the IEEE 38th Annual Computer Software and Applications Conference (COMPSAC)*. 2014: 219–228.

[33] A. M. Fard and A. Mesbah. JSNose: Detecting JavaScript Code Smells. In *Proceedings of the IEEE 13th International Working Conference on Source Code Analysis and Manipulation* (*SCAM*). 2013: 116-125.

[34] A. Holkner and J. Harland. Evaluating the dynamic behavior of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science*. 2009, 91: 19-28.

[35] B. Åkerblom, J. Stendahl, M. Tumlin, and T. Wrigstad. Tracing dynamic features in python programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. 2014: 292–295.

[36] L. Chen, B. Xu, T. Zhou, and X. Zhou. A Constraint Based Bug Checking Approach for Python. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference(COMPSAC)*. 2009, 2: 306–311.

[37] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and K. T. Tekle. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Dynamic Languages Symposium(DLS)*. 2010: 27–42.

[38] Z. Chen, L. Chen, and B. Xu. Hybrid Information Flow Analysis for Python Bytecode. In *Proceedings of the 11th Web Information System and Application Conference (WISA)*. 2014: 95-100.

[39] Z. Xu, J. Qian, L. Chen, et al. Static Slicing for Python First-Class Objects. In *Proceedings of the 13th International Conference on Quality Software (QSIC)*. 2013: 117-124.