PYTHON CODE SMELL REFACTORING ROUTE GENERATION BASED ON ASSOCIATION RULE AND CORRELATION

Guanglei Wang

Department of Computer Science and Technology
Shanghai Normal University
Shanghai 200234, P. R. China
wgl208431@163.com

Junhua Chen

Department of Computer Science and Technology
Shanghai Normal University
Shanghai 200234, P. R. China
chenjh@shnu.edu.cn

Jianhua Gao[‡]

Department of Computer Science and Technology
Shanghai Normal University
Shanghai 200234, P. R. China
jhgao@shnu.edu.cn

Zijie Huang

Department of Computer Science and Engineering
East China University of Science and Technology
Shanghai 200237, P. R. China
hzj@mail.ecust.edu.cn

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Code smell is a software quality problem caused by software design flaws. Refactoring code smells can improve software maintainability. While prior work mostly focused on Java code smells, only a few prior researches detect and refactor code smells of Python. Therefore, we intend to outline a route (i.e., sequential refactoring operation) for refactoring Python code smells, including LC, LM, LMC, LPL, LSC, LBCL, LLF, MNC, CCC and LTCE. The route could instruct developers to save effort by refactoring the smell strongly correlated with other smells in advance. As a result, more smells could be resolved by a single refactoring. First, we reveal the co-occurrence and the inter causation between smells. Then, we evaluate the smells' correlation. Result highlights 7 groups of smell with high co-occurrence. Meanwhile, 10 groups of smell correlate with each other in a significant level of Spearman correlation coefficient at 0.01. Finally, we generate the refactoring route based on the association rules, we exploit an empirical verification with 10 developers involved. The results of Kendall's Tau show the proposed refactoring route has a high inter-agreement with the developer's perception. In

‡ Corresponding author

This work is supported by National Nature Science Foundation of China (61672355)

conclusion, we propose 4 refactoring routes to provide guidance for practitioners, i.e., {LPL→LLF}, {LPL→LBCL}, {LPL→LMC}, and {LPL→LM→LC→CCC→MNC}.

Keywords: Python code smell; co-occurrence; correlation; code refactoring; empirical software engineering.

1. Introduction

Code smell is a software quality problem which may not be the direct cause of software fault, but hinders software maintenance work. Fowler [1] proposed 22 kinds of code smells that violate design rules, which can be divided into three classes according to different granularity, i.e., methods, classes, and packages. At present, researchers focus mainly on code smell detection for Java projects [2]. However, little is known about Python software smell since Python language type and its inherent characteristics as a dynamic language is difficult to measure and comprehend [3]. Compared with Java, there are fewer tools for Python code smell detection, e.g., Psmell can effectively detect 10 kinds of Python code smells [3]. Moreover, it is not possible to use the Java metric threshold directly for Python smell detection because of the structural differences in the characteristics of different languages (e.g., weakly-typed and strongly-typed), which makes it impossible to use the Java metric threshold (e.g., TCC) directly for Python smell detection [3], the details are given in Section 2.4.

As a subsequent process in Software Quality Assurance (SQA) activity, the methodologies and strategies to refactor smells are also major concerns of practitioners and researchers. As for Java code refactoring, Optimize Streams is implemented as a plug-in to the popular Eclipse IDE, which assists developers in writing optimal stream software in a semantics-preserving fashion [4]. For Python refactoring, HARP enables holistic analysis that spans across computation graphs and their hosting Python code [5].

In order to integrate Python smell detection results to practical SQA activities, we intend to generate refactoring strategy by revealing potential relationship among Python code smells. Our research questions include revealing the relationship between smells and a refactoring strategy. Empirical research shows that developers tend to abandon software quality static analysis tools [6], because they generate too much uninterpretable results. In response, researchers should offer refactoring strategies concerning the priority and relationship among design problems such as code smells [7]. In line with prior research, Python code smells detection also lacks enough interpretation of detection results [3].

To make the best use of Python smell detection result, we intend to fill the gap by outlining a route for refactoring 10 common Python code smells, the route means developer’s sequential refactoring operation for the code smells. The scope of our evaluation in code smells includes the most 2 common smells (i.e., Long Parameter List (LPL), Long Method (LM)), and other 8 smells occurred in 9 active open-source Python projects on GitHub, including 55,206 Python files in 94 versions [3]. The motivation of generating a sequential route is to save effort, i.e., after refactoring the smells in the front of a route, the smells appear later may also be eliminated [8], otherwise developers may spend extra time comprehending more code related to code smells. Thus, we generate the route according to the co-occurrence and correlation among them [9]. To measure the validity of the generated route, we investigate the extent of agreement between the generated order and novice and experienced developers' manually generated order by calculating Kendall's Tau [10], which is a rank statistic used to measure the ordinal association between two random variables. Then, we verify empirically the validity of the generated routes. The relevant descriptions of the 10 Python code smells are shown in Table 1.

Table 1. Explanation of Python code smells

Python code smells	Description
Large Class (LC)	The class code is too long
Long Method (LM)	The function code is too long
Long Message Chain (LMC)	An overly coupled message chain
Long Parameter List (LPL)	Too many parameters in a function
Long Scope Chaining (LSC)	The nesting level of a function is too deep
Long Base Class List (LBCL)	Too many super classes inherited in one class
Long Lambda Function (LLF)	The code of a lambda function is too long
Multiply-Nested Container (MNC)	A container with multiple nesting
Complex Container Comprehension (CCC)	A container production contains too complex code
Long Ternary Conditional Expression (LTCE)	The code of a ternary function expression is too long

The main contributions and innovations of this paper are as follows:

- To the best our knowledge, we are the first to generate refactoring guidelines by evaluating the co-occurrence and correlation of Python smells. Simultaneously, we find the code smells with the highest co-occurrence, and the smells with the highest correlation in 9 active open-source Python projects on GitHub.
- We discover correlated and co-occurred Python smells in quantitative evaluation. Meanwhile, we make a comparative study about the smell relationship between Java software and Python software. More importantly, we explore the relationship between python code smells.
- We propose a refactoring route to provide guidance for practitioners' refactoring task. Specially, we verify the rationality of the refactoring strategy through Kendall's Tau coefficient. We provide developers with Python smell refactoring route as follows: {LPL→LLF}, {LPL→LBCL}, {LPL→LMC} and {LPL→LM→LC→CCC→MNC}.

The structure of the paper is as follows: Section 2 introduces the background and related work, which gives the current research status in this field. Section 3 gives co-occurrence and correlation of Python code smell. In section 4 we analyze the content and results of the experiment, in which we provide a refactoring route for developers. Section 5 introduces threats to validity. Finally, we conclude the full paper and propose future work.

2. Related Work

2.1. *Code smell detection*

Code smell is a software quality problem caused by software design flaws or bad programming habits. Fowler [1] proposed 22 kinds of code smells, which involve many aspects such as method, class and application level.

Pecorelli [11] et al. studied God Class, Spaghetti Code, Data Class, Large Class and Long Method in 13 open-source projects including 125 releases. Metric-based methods were applied to compare large-scale machine learning and heuristic methods. Although researchers are committed to the study of code smells, they still do not know the extent to which the code smell in the software system affects the maintainability of the software.

Palomba [12] et al. detected 13 code smells in 395 releases of 30 open-source projects, then manually verified 17,350 examples, and they found that smells characterized by long or complex codes are widely distributed, which possessed significant variability and error-proneness.

Tufano [13] et al. conducted a large-scale study on the change history of 200 open-source projects, and they found that most of the smell instances were introduced when the original code was created. Surprisingly, researchers with heavy workload and high release pressure are more likely to introduce code smells. Code smells have a long lifecycle and are rarely removed directly due to refactoring, on the contrary, refactoring may introduce new smells.

Vavrova [2] et al. studied 5 kinds of Python code smells such as Feature Envy, Data Class, Long Method, Long Parameter List and Large Class as well as 4 kinds of anti-patterns: God Class, Swiss Army Knife, Functional Decomposition and Spaghetti Code, 9 design flaws in Python source codes are detected. In addition, they developed a design defect detection tool for Python code.

Kessentini et al. [14] found that code smell detection methods can be summarized into seven categories: search-based [15], metric-based [16], visualizat- ion [17], symptom [18] and manual [19]. Sae-Lim [20] et al. ranked the code smell detection results by considering the developer's current environment, the results showed that coarse-grained code provides better ranking than fine-grained code, compared with the context-based smell sorting, smell ranking based on severity provides more relevant results.

2.2. *Smell correlation evaluation*

Bigonha et al. [21] evaluated the software metric threshold for code smell identification and software system fault prediction, and identified Large Class and Long Method for Java software systems. The research found that the metric threshold is an effective basis for evaluating software quality as well as helping developers focus on the category with the highest severity of the problem. Vavrova [2] et al. studied the design flaws of Python and analyzed Python modules through design flaw detection tools. They found that: the most common design flaws are Long Method and the smell of Swiss Army Knife. On the contrary, Spaghetti Code and Large Class occur least frequently.

Metric-based code smell detection methods have been widely applied. Padilha [22] studied whether it can detect three types of smells: Divergent Change, Shotgun Surgery, and God Class. The results show that in general, paying attention to the metric threshold contributes developers to detect code smells. But the selection of the metric threshold is still very controversial. In order to perform smell detection and fault prediction on object-oriented metric thresholds, Bigonha et al. [21] detected 5 smells on 12 open-source Java systems, performed metrics at each class level such as DIT, LCOM, NOF, NOM, NORM, NSC, NSF, NSM, SIX and WMC. Liu et al. [23] have proposed a method based on deep learning to detect Feature Envy, evaluated on 7 open-source Java applications. The results show that it is better than the latest method in detecting Feature Envy. Terra et al. [24] published a data set called Qualitas.class, which provided compiled Java projects for 111 systems included in the dataset, aiming to provide researchers with detecting Java code smells and refactoring work.

Most researchers focus on smell detection in Java projects, but there is little research literature on Python code smell. Therefore, this paper is based on the research of Chen et al. [3] to reveal the co-occurrence and correlation of the detected 10 Python code smells. On the basis, we provide developers with a refactoring route in order to reduce workload, which is verified manually by both novice and experienced developers. The statistical results of Kendall's Tau show that the strategy we proposed has a high inter-agreement with the developer's perception.

2.3. *Association rule mining of code smells*

Agrawal et al. [9] proposed an efficient algorithm to generate association rules between database items, combining data mining and association rules for the first time. Alfadel et al. [25] applied association rule analysis methods to evaluate whether the design pattern is consistent with code smells of different granularity levels are related, the results show that there is a positive correlation between design patterns and code smells. Palomba et al. [26] used association rule mining to discover the co-occurrence relationship between code smells. On the one hand, they emphasized some predictable co-occurrence relationships, such as Long Method and Spaghetti Code, Long Method and Long Parameter List, they also reveal co-occurrence relationships that some studies have missed, finding that Message Chains and Refused Bequest also have co-occurrence relationships.

Palomba et al. [26] detected 13 types of smells based on 30 Java software systems, found that 6 groups of code smells are frequently co-occurring such as {Message Chain, Spaghetti Code}, {Message Chain, Complex Class}, {Message Chain, God Class}, {Message Chain, Refused Bequest}, {Long Method, Spaghetti Code} and {Long Method, Feature Envy}. Jaafar et al. [29] studied the co-occurrence of anti-patterns and clones as well as the relationship between co-occurrence and class prone to failure, results showed that the percentage of co-occurrences involving anti-patterns and clones was between 63% and 32%, class prone to failure with anti-pattern and clone co-occurrence is significantly increased. Fontana et al. [30] evaluated co-occurrence smells in 74 systems in the Qualitas.class data set [31], found that code smells tend to cluster together and interact in multiple ways, and that smell clusters have a greater impact on the maintainability of software than isolated smells.

2.4. *Metric threshold comparison of Python and Java*

Table 1 shows 10 Python code smells, in which LPL, LM, LSC, LC and LMC are code smells that can be applied to multiple object-oriented programming languages such as Java, Python, JavaScript, etc., while LBCL, LLF, LTCE, CCC and MNC belong to Python code smell.

Mayvan [27] et al. compared all the metric thresholds about Java in the past 20 years, then outlined the standards code smells metrics. Fard [28] et al. proposed a metric-based method to detect JavaScript code smells automatically, in which LSC smell measurement Standards also apply to the Java language. This section compares the metrics of Java code smell and Python code smell. The Java metrics involved in LPL, LM, LSC, LC, and LMC code smell are shown in Table 2.

Table 2. Catalog of Java metrics

Java metric	Description
NOP	Number of Parameters
MLOC	Line of Code in a Method
VG	McCabe's Complexity
NOLV	Number of Local Variables
MNOB	Maximum Number of Branches
LSC	Length of Scope Chain
WMC	Weighted Method Count
ATFD	Access to Foreign Data
TCC	Tight Class Cohesion
CLOC	Line of Code in a Class
NOM	Number of Methods in a Class
NOF	Number of Fields
MCC	Method Calling Chain
MLOC	Line of Code in a Method

Table 3. Catalog of Python metrics

Python metric	Description
PAR	Number of Parameters
MLOC	Method Lines of Code
DOC	Depth of Closure
CLOC	Class Lines of Code
LMC	Length of Message Chain
NBC	Number of Base Classes
NOC	Number of Characters
NOO	Number of Operators and Operands
NOL	Number of Lines
NOFF	Number of Clauses and Filter Expressions
LEC	Length of Element Chain
DNC	Depth of Nested Container
NCT	Number of Container Types

Different from traditional object-oriented metrics, Chen et al. [3] defined 8 new metrics to measure Python code smell: NBC, NOC, NOO, NOL, NOFF, LEC, DNC, and NCT. The metrics involved in 10 Python code smells are shown in Table 3.

Mayvan et al. [27] proposed a multi-step process using quality metrics and refactoring opportunities to detect Java code smells, and conducted a systematic literature review of all code smells formally defined using quality metrics in the field. The selection of the threshold by looking for a threshold with recognition higher than 50%, if the threshold is not unified, then choose the loosest threshold in the literature.

Chen [3] et al. selected the Python smell threshold by comparing the threshold based on experience, the threshold based on statistic and the threshold based on tuning machine. Among them, the threshold selection based on the tuning machine achieves the best accuracy, which is used as the standard threshold.

2.5. *Detection strategy of Python and Java*

Mayvan et al. [27] outlined the metrics and threshold standards of Java code smell. Accordingly, this section detects five types of Java code smell detection strategies including LPL, LM, LSC, LC, and LMC. In addition, 10 Python code smell detection strategies are compared with the above. The results show that the detection strategies for smells that exist in both Java and Python code are almost the same. However, the detection strategy of code smell only belongs to Python contains more threshold metrics for logical judgment.

Table 4. Java code smell detection strategy

Java smell	Detection strategy
LPL	NOP > 5
LM	MLOC > 50 | VG >5 | ((NOP > 4 | NOLV > 4) & (MNOB > 4))
LSC	LSC > 3
LC	(WMC > 47 & ATFD > 5 & TCC < 0.33) | CLOC > 100 | NOM > 14 | NOF > 8
LMC	MCC > 3

Table 5. Python code smell detection strategy

Python smell	Detection strategy
LPL	PAR ≥ 5
LM	MLOC ≥ 52
LSC	DOC ≥ 4
LC	CLOC ≥ 37
LMC	LMC ≥ 4
LBCL	NBC ≥ 3
LLF	(NOC ≥ 73) & ((PAR ≥ 4) | (NOO ≥ 15))
LTCE	(NOC ≥ 101) | (NOL ≥ 3)
CCC	((NOC ≥ 92) & (NOFF ≥ 3)) | (NOO ≥ 22)
MNC	(LEC ≥ 3) | ((DNC ≥ 3) & (NCT ≥ 2))

Chen [3] et al. detected 10 Python code smells based on metric threshold strategy of the tuning machine, evaluated the abused code smells of dynamic type, found that dynamic type behaviors implied potential threats. In addition, they implemented a positioning method to identify performance-related code smells in the software, which are sorted in order of priority. In order to compare the difference between detection strategy of Python and Java, we summarized the results as follows. Table 4 and Table 5 respectively show the detection strategy of Java code smell and Python code smell.

3. Co-occurrence and Correlation of Python Code Smell

3.1. *Dataset construction*

The original code smell dataset can be found on GitHub from Chen [3] et al, which is collected through Python smell detection tool named Psmell. In order to reveal the relationship between smells, we grouped the related smells in each project and got the smelly files, then we label files with smell relations. The division of smelly groups is explored by combining two or more of the 10 Python code smells. The dataset collection process is based on popularity, which contains 9 active open-source Python projects on GitHub. Projects include 55,206 Python files, the size of the project can meet the needs of research at present.

The data collection process includes data preprocessing, association rule mining, and Spearman correlation coefficient analysis. First, in order to obtain valid data, we extract the smelly files (i.e., python files with code smell) from 9 Python projects involving various versions. Second, we label the smelly files by "1" to indicate the presence of smell, and "0" to represent there is no smell in files. Since positive samples are sparsely distributed, it is necessary to merge the various versions with the project as the category. Third, we combine the 10 kinds of code smells in pairs. If there exists no smell group in all projects, the smell group will be removed. Fourth, we explore the relationship between smells through association rules. The support rate indicates the degree of co-occurrence between smells,

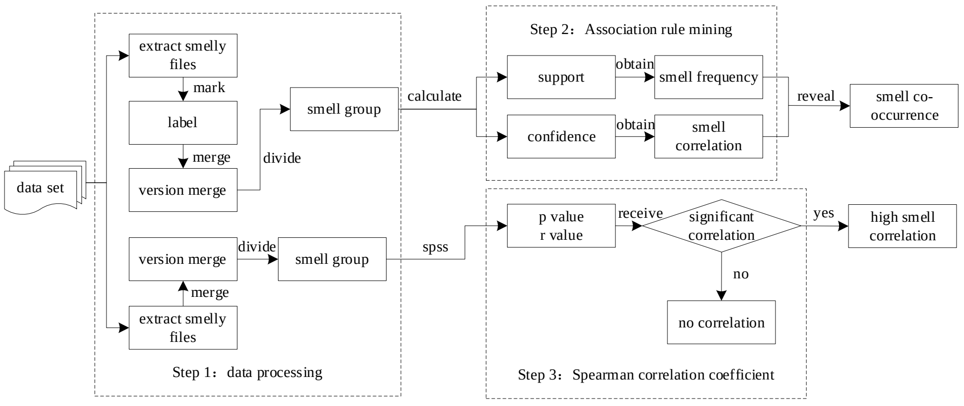

Fig. 1. Flow chart of co-occurrence and correlation evaluation of smells

and confidence rate reveals the causal relationship between smells. Finally, the correlation between smells is obtained by Spearman's correlation coefficient. If the correlation between smells is significant, and the group of smells appeared in multiple projects, it indicates a higher degree of correlation. The co-occurrence and correlation evaluation process of Python code smell is shown in Fig. 1.

3.2. *Spearman correlation*

Palomba et al. [32] designed and evaluated a fault prediction model for smell perception, using Spearman rank correlation and variance expansion factor function to study the multicollinearity of the model. O'brien et al. [33] elaborated that the variance expansion factor and tolerance are widely used measures in the study of multicollinearity between independent variables. Researchers are able to reduce collinearity by eliminating one or more variables. This paper focuses on the correlation between smells but does not involve the establishment of models, so the Spearman correlation coefficient is mainly applied to indicate the degree of correlation between smells.

By calculating the Spearman correlation coefficient between smells, the significant smells indicate a high degree of correlation between smells. Because of the significant differences between projects, Therefore, we combined 9 projects to indicate the most frequent and significantly correlated smell groups to point the high degree of correlation of smells. As shown in Eq. (1):

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)} \tag{1}$$

In which *n* is the qualities of data, and d_i indicates the difference between two data orders.

3.3. *Association rule mining*

Association rules used to discover the degree of co-occurrence between smells. The specific expression of association rules is as follows: $I = \{i1, i2, ..., in\}$ is a set of *n* attributes called items, which indicate the existence of attributes in the element (*i.e.*, the items of *I*). $T = \{t1, t2, ..., tm\}$ is a set of *m* transactions (*i.e.*, the set of all the elements), association rules are defined as the implication formula of $X{=>}Y$, where $X, Y \subseteq I$, and $X \cap Y = \varnothing$. Set *T* refers to the composition of all the methods of the specific system under study, and set *I* refers to the specific smell indicated in each project. If two code smells affect the same Python file at the same time, it is considered that there may be a co-occurrence relationship between them. Specifically, for two disjoint code smells CS_{left} and CS_{right}, if they have a co-occurrence relationship, it can be expressed by the implicit expression of association rule $CS_{left} => CS_{right}$, that is, if the Python file is affected by CS_{left}, so the same Python file should also be affected by CS_{right}. In this paper, the Support obtained by association rule indicates the occurrence frequency of the smell groups, as shown in Eq. (2). Confidence is used to explain the causal relationship between smell groups, in other words, the appearance of a certain smell often indicates the appearance of another smell, which as shown in Eq. (3):

$$Support = \frac{|CS_{left} \cup CS_{right}|}{T} \tag{2}$$

$$Confidence = \frac{|CS_{left} \cup CS_{right}|}{|CS_{left}|} \tag{3}$$

Among them, T is the total number of Python files in the system under study, called transactions, by applying the Apriori algorithm [9] mining association rules, the minimum effective Support and Confidence can be set in the algorithm.

Studying one kind of code smell in the Python files may imply the existence of another kind of smell, and the frequency of its appearance can be evaluated, we can make a certain code smell as CS_i, then calculate the percentage of the number of times that the smell co-occurs with another code smell CS_j in the Python file. Eq. (4) used to express the co-occurrence frequency of two smells:

$$co - occurrences_{csi,j} = \frac{|cs_i \wedge cs_j|}{|cs_i|},\ i \neq j \tag{4}$$

In which $|CS_i|$ is the number of times the smell CS_i appears, and $|CS_i \wedge CS_j|$ means the frequency of the smell CS_i and CS_j appear at the same time.

3.4. *Kendall's Tau*

Kendall’s Tau is a rank statistic used to measure the ordinal association between two random variables [10], which can be expressed as follows: Let (x_1, y_1), …, (x_n, y_n) be a set of observations of the joint random variables X and Y, such that all the values of (x_i) and (y_i) are unique (ties are neglected for simplicity). Any pair of observations (x_i, y_i) and (x_j, y_j), where $i < j$, are said to be concordant if the sort order of (x_i, x_j) and (y_i, y_j) agrees: that is, if either both $x_i > x_j$ and $y_i > y_j$ holds or both $x_i < x_j$ and $y_i < y_j$; otherwise they are said to be discordant. The Kendall τ coefficient is defined as Eq. (5):

$$\tau = \frac{(number\ of\ concordant\ pairs) - (number\ of\ discordant\ pairs)}{\binom{n}{2}} \tag{5}$$

Where $\binom{n}{2} = \frac{(n)(n-1)}{2}$ is the binomial coefficient for the number of ways to choose two items from n items. The Kendall’s Tau coefficient are defined as follows in Eq. (6):

$$Kendall's\ \tau\ Agreement = \begin{cases} weak, & if\ |\tau| \leq 0.3 \\ moderate, & if\ 0.3 < |\tau| \leq 0.6 \\ strong, & if\ |\tau| > 0.6 \end{cases} \tag{6}$$

4. Experiment and analysis

The goal of our research is to generate refactoring strategy by revealing potential relationship among Python code smells, with the purpose of integrating Python smell detection results to practical SQA activities. To these ends, we propose 4 research questions as follows:

- RQ1: Which Python smells co-occur most frequently with each other?

  The measurement can reflect the co-occurrence relationship of smells by calculating the number of code smells in each Python file, then we compare the percentages in Python files that are affected by a single smell and two or more smells.
- RQ2: Which Python code smells have strongest correlation with each other?

  Since the sample set involves 94 versions, and the obtained positive samples are relatively rare, each Python file is regarded as a sample. then the positive samples with the code smell with the co-occurrence relationship are extracted, after that the Spearman correlation coefficient is used to evaluate the degree of correlation between smells in each project.
- RQ3: Which co-occurred python code smells have strong associations relationship? What is the inter causation between them?

  The association rules are used to discover the co-occurrence relationship between smells. Moreover, the strong association between smells can be further extracted through the lift of association rules. Thus, the priority order of the appearance of smells could be revealed.
- RQ4: Can we generate a route for refactoring Python smells that is close to developers' perception?

  To ensure the accuracy of the experimental result, we invited 10 developers to generate smell's refactoring route based on the association rules. Kendall's Tau shows that the strategy we proposed has a high inter-agreement with the developer's perception.

4.1. *Experiment environment*

The experiment is based on 94 versions of 9 Python projects with high attention, including 55,206 Python files, the project names are: Ansible, Boto, Django, Ipython, Matplotlib, Nltk, Numpy, Scipy, and Tornado. Meanwhile, the experimental environment is: 8.00GB RAM, Inter(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59GHz, 8,705 Python files with smell were extracted through the Jupyter Notebook development environment, then label it and complete the experiment with SPSS. The project description is shown in Table 6.

Table 6. Project description table

Project	Line of code	Number of files	Description
Ansible	44,086	394	A new automated operation and maintenance tool
Boto	119,905	703	Python interface for Amazon Web Services
Django	214,997	2,106	Advanced Web Application Framework
Ipython	105,522	788	Interactive computing system of Python
Matplotlib	135,459	815	2D drawing library of Python
Nltk	73,053	263	Python-based natural language processing toolset
Numpy	131,854	361	Python scientific computing basic package
Scipy	173,714	522	Set of tools for scientific computing
Tornado	28,455	108	Web server architecture and asynchronous network library

4.2. *Experimental data*

Since the experimental data is based on 9 Python projects, and the number of smelly files contained in each project varies greatly, the projects with more Python files are divided into multiple sets of data as much as possible, the distribution of 10 kinds of code smells in the data set are shown in Fig. 2.

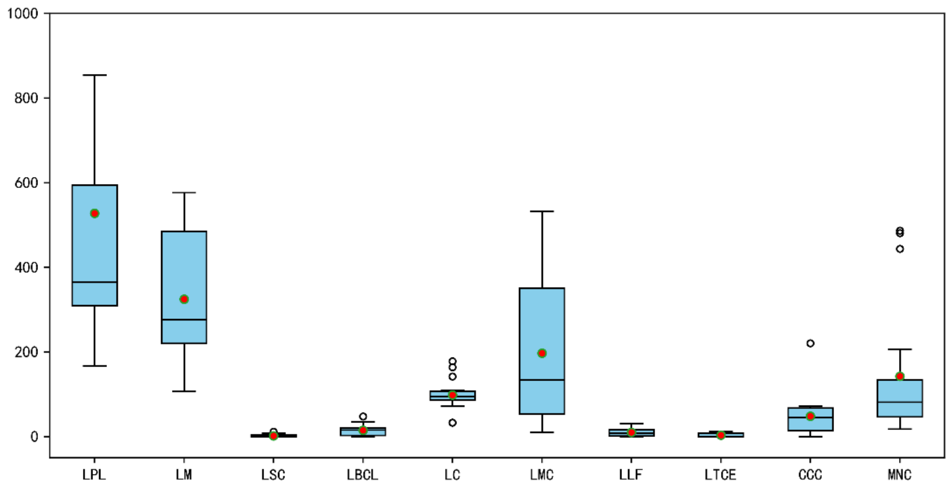

Fig. 2. Distribution of the number of 10 code smells in the data set

It can be found from Fig. 2 that in terms of a single smell, LPL, LM and LMC are the three most frequent smells while the smells of LSC, LBCL, LLF and LTCE are less frequent, which reveals the point of smell refactoring.

Table 7. The distribution of several code smells with the highest co-occurrence frequency in the data set

Smell / group	LPL	LM	LC	LMC	LLF	CCC	MNC	Quantity	Percentage (%)
1	**1**	**1**	0	0	0	0	0	**733**	**29.27**
2	**1**	0	**1**	0	0	0	0	**222**	**8.87**
3	0	**1**	**1**	0	0	0	0	**196**	**7.83**
4	**1**	0	0	**1**	0	0	0	**184**	**7.35**
5	0	**1**	0	0	0	0	**1**	**102**	**4.07**
6	**1**	**1**	**1**	0	0	0	0	**267**	**10.66**
7	**1**	0	**1**	**1**	0	0	0	**52**	**2.08**
8	**1**	**1**	0	**1**	0	0	0	**51**	**2.04**
9	**1**	**1**	0	0	0	0	**1**	**47**	**1.88**
10	**1**	**1**	0	0	0	**1**	0	**36**	**1.44**
11	0	**1**	**1**	**1**	0	0	0	**36**	**1.44**
12	**1**	**1**	**1**	**1**	0	0	0	**24**	**0.96**
13	**1**	**1**	**1**	0	0	**1**	0	**18**	**0.72**
14	**1**	**1**	**1**	0	**1**	**1**	**1**	**11**	**0.44**

The Python files in the project are often accompanied by the existence of two or more smells. Table 7 shows the distribution of several code smells (“1” means smell exists and “0” means absent) with the highest co-occurrence frequency in the experimental data Through Eq. (4). Among them, LPL and LM are the smell groups with the highest degree of co-occurrence, at the same time, the co-occurrence frequencies of {LPL, LC}, {LM, LC}, and {LPL, LMC} are also followed closely.

4.3. ***Experimental results and analysis***

This section answers RQ1-RQ4 based on the experimental results.

RQ1: Which Python smells co-occur most frequently with each other?

Table 8 combined with Eq. (2) gives the distribution of the co-occurrence frequency of Python code smell through pairwise combination, the smell groups with a low degree of co-occurrence were deleted. The degree of Support of association rules can well reflect the degree of co-occurrence between smells, the experimental results show that there are 7 pairs of smells with the highest co-occurrence frequency. The smell group of {LPL, LM} topped the list, followed by {LPL, LC}, {LPL, LMC}, {LM, LC}, {LM, CCC}, {LM, MNC} and {LC, MNC}.

Code smells have co-occurrence relationship among each other by nature [26], i.e., the appearance of a certain smell is usually accompanied by the occurrence of another code smell. This relationship can be identified by the Confidence of the association rule. Table 9 is derived from Eq. (3) to obtain the Confidence between the smells in 9 projects (bold fonts with Confidence above 0.5). The experimental results show that LPL and LM usually have a significant co-occurrence relationship, which means that LPL and LM are the focus of refactoring smells. Meanwhile, the appearance of LMC easily induces the appearance of LPL, and CCC also usually causes LPL to occur. Similarly, MNC largely makes LM happen, and CCC can easily cause the appearance of LC. All in all, the co-occurrence relationship between smells is complicated, but the smells that have a significant co-occurrence relationship can provide guidance for refactoring route.

Table 8. Support of association rules in 9 projects

Project / Smell group	Ansible	Boto	Django	Ipython	Matplotlib	Nltk	Numpy	Scipy	Tornado
LPL-LM	**0.139**	0.102	**0.072**	0.094	**0.241**	**0.161**	**0.184**	**0.172**	**0.227**
LPL-LC	0.009	**0.133**	0.05	0.048	0.197	0.045	0.055	0.034	0.054
LPL-LMC	0.023	0.02	0.061	0.031	0.108	0.002	0.005	-	0.016
LPL-CCC	-	-	0.014	0.004	0.045	0.081	0.02	0.001	0.011
LPL-MNC	-	0.033	0.012	0.004	0.032	0.026	0.006	0.007	-
LM-LC	0.027	0.044	0.051	**0.123**	0.12	0.06	0.077	0.042	0.146
LM-LMC	-	-	0.061	0.029	0.02	0.004	0.003	0.008	0.032
LM-CCC	0.009	-	0.037	0.004	0.016	0.045	0.02	0.002	0.022
LM-MNC	0.03	0.029	0.014	0.006	0.022	0.023	0.07	0.014	-
LC-LMC	-	0.013	0.023	0.011	-	0.002	0.003	-	0.065
LC-CCC	0.011	-	0.014	0.004	0.032	0.049	-	0.001	0.022
LC-MNC	0.009	0.012	0.003	0.004	0.008	0.019	0.018	0.007	-
CCC-MNC	-	-	0.007	0.004	0.007	0.023	0.001	0.001	-

Table 9. Confidence of association rules in 9 projects

Project / Smell group	Ansible	Boto	Django	Ipython	Matplotlib	Nltk	Numpy	Scipy	Tornado
LPL-LM	0.207	0.192	0.172	0.253	0.331	0.255	**0.610**	0.346	**0.519**
LM-LPL	0.439	0.372	0.194	0.199	**0.641**	**0.528**	0.261	0.320	**0.512**
LPL-LBCL	-	-	0.008	0.006	-	0.018	-	0.005	0.074
LBCL-LPL	-	-	0.258	0.143	-	**1.000**	-	0.188	0.429
LPL-LC	0.014	0.251	0.120	0.130	0.271	0.071	0.183	0.069	0.123
LC-LPL	0.121	**0.569**	0.277	0.196	**0.818**	0.279	0.323	0.318	0.303
LPL-LMC	0.034	0.037	0.147	0.084	0.148	0.003	0.017	-	0.037
LMC-LPL	**0.625**	0.202	0.360	0.456	**0.590**	**0.500**	0.238	-	0.083
LPL-LLF	-	-	0.013	0.029	0.009	0.018	-	0.005	-
LLF-LPL	-	-	0.176	**1.000**	**1.000**	0.25	-	0.143	-
LPL-CCC	-	-	0.034	0.010	0.063	0.128	0.068	0.002	0.025
CCC-LPL	-	-	0.127	0.073	**0.628**	0.413	**0.526**	0.032	**0.500**
LPL-MNC	-	0.061	0.030	0.010	0.044	0.042	0.020	0.015	-
MNC-LPL	-	0.173	0.213	0.079	**0.745**	0.304	0.048	0.114	-
LM-LC	0.086	0.162	0.139	0.260	0.318	0.196	0.109	0.078	0.329
LC-LM	0.364	0.190	0.286	0.500	0.497	0.372	0.449	0.388	**0.818**
LM-LMC	-	-	0.166	0.061	0.054	0.012	0.004	0.016	0.073
LMC-LM	-	-	0.360	0.421	0.111	**1.000**	0.143	0.455	0.167
LM-CCC	0.029	-	0.101	0.008	0.043	0.147	0.029	0.003	0.049
CCC-LM	0.800	-	0.341	0.073	0.221	0.231	0.526	0.065	**1.000**
LM-MNC	0.094	0.107	0.037	0.013	0.058	0.074	0.099	0.026	0.159
MNC-LM	0.245	0.156	0.234	0.132	**0.510**	0.261	**0.544**	0.215	**0.722**
LC-LMC	-	0.056	0.130	0.044	-	0.012	0.018	-	0.364
LMC-LC	-	0.135	0.138	0.158	-	**0.500**	0.143	-	0.333
LC-CCC	0.152	-	0.076	0.015	0.133	0.302	-	0.008	0.121
CCC-LC	**1.000**	-	0.124	0.073	0.442	0.250	-	0.032	**1.000**
LC-MNC	0.121	0.051	0.016	0.015	0.035	0.116	0.108	0.070	-
MNC-LC	0.075	0.064	0.050	0.079	0.196	0.217	0.144	0.114	-
CCC-MNC	-	-	0.060	0.073	0.093	0.115	0.026	0.032	-
MNC-CCC	-	-	0.113	0.079	0.157	0.261	0.008	0.013	-

RQ2: Which Python code smells have strongest correlation with each other?

Table 10 shows the p value of the Spearman correlation coefficient between smells at 0.01 level (the p value of 0.05 level is marked with '-' in the table), which is used to eval-

Table 10. The p-value of Spearman's correlation coefficient in 9 projects at 0.01 level

Project / Smell group	Ansible	Boto	Django	Ipython	Matplotlib	Nltk	Numpy	Scipy	Tornado
LPL-LM	-0.231	-0.105	-0.290	-0.257	0.154	-	-0.113	-0.204	-
LPL-LC	-0.315	-	-0.129	-0.187	0.234	-0.279	-	-0.107	-
LPL-LMC	-	-0.158	-	-0.115	-0.153	-	-	-0.127	-0.319
LPL-CCC	-0.142	-	-0.180	-0.143	-0.057	-0.189	0.076(-)	-0.144	-
LPL-MNC	-0.489	-0.328	-0.110	-0.134	-	-0.138	-0.200	-0.189	-0.276
LM-LC	-	-0.081	-0.051	-0.102	-0.206	0.154	-0.206	-0.083	0.423
LM-LMC	-0.131	-0.199	-	-	-0.257	0.102(-)	-0.164	-	-0.282
LM-LLF	-	-	-0.067	-	0.136	-0.142	-0.111	-0.135	-
LM-MNC	-	-0.109	-0.053	-0.131	-	-	-	-0.144	-0.285
LC-CCC	0.376	-	-0.055	-0.093	0.121	0.120	-0.091	-	0.319

uate the correlation degree of the code smell. The table shows the 10 most relevant groups of smells. The experimental results show that there is usually a negative correlation between these smells. Furthermore, {LPL-LM}, {LPL-CCC}, {LPL-MNC} and {LPL-LC} have a significant correlation generally.

RQ3: Which co-occurred python code smells have strong associations relationship?

Fig. 3 shows the relationship of strong correlation in the smell of Python code with co-occurrence relationship. We are concerned about the strong correlation of several Python code smells. Based on the Lift in association rules, if Lift is greater than 1, which indicates that the smell is strongly associated. The result is visualized in Fig. 3, it can be found that LPL is easy to cause the appearance of other code smells, the strong correlation between smells includes: {LPL→LM}, {LPL→LMC}, {LPL→LBCL}, {LPL→CCC}, {LPL→LLF} and {LPL→MNC}.

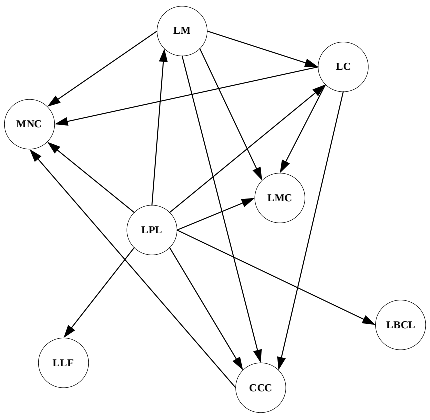

Fig. 3. The causal relationship of strong correlation with smells

RQ4: Can we generate a route for refactoring Python smells that is close to developers' perception?

The experimental data set is based on the open-source projects available on GitHub [3]. To assess the developers' perception towards refactoring the smells concerned, we invite 10 novice and experienced developers to perform smell refactoring individually. The developers have access to the offline versions of projects, and they acknowledge smells in related code components. However, online resource usage is prohibited. The developers' experience in developing Python projects is shown in Table 11. Afterwards, we collect their mostly preferred refactoring route, which is presented in Table 12. Next, we used the Kendall's Tau coefficient to calculate the inter-agreement among refactoring routes proposed by both novice and experienced developers. Result shows our refactoring route achieves a Kendall's Tau coefficient of 0.814, which indicates the inter-agreement between developers is strong. The high inter-agreement reflects the high validity of the proposed refactoring route.

Table 11. Developer Information

Developer	Population	Python Development Experience
Novice	7	1-3 Years
Experienced	3	3-10 Years

Table 12. Refactoring routes suggested by developers

Smells / Refactor route	LPL	LM	LC	CCC	MNC
Novice	1	2	4	3	5
	1	3	2	5	4
	1	2	3	4	5
	1	3	4	2	5
	2	1	3	5	4
	1	2	3	4	5
	1	2	3	5	4
Experienced	1	2	3	4	5
	1	3	2	4	5
	1	2	3	4	5

5. Threats to validity

Some threats may have influenced our study. Construct validity refers to the relationship between theory and observation. Internal validity is mainly considered possible errors in the experimental code. External validity is about the generalizability of results. Conclusion validity is related to treatment and outcome.

Construct validity. This paper contains an experiment based on 9 active Python projects studied by prior research. Thus, the data set may not comply with other scenarios. Furthermore, metric thresholds of the experiment are carried out based on prior researches [21], which may differ from practical scenarios.

Internal validity. The labels of our dataset are generated by a tool developed by us. The performance of the tool might be a threat to validity. However, the F-Measure of the tool reaches more than 97%, and we validated the results empirically. Thus, we believe the internal validity of the dataset is acceptable.

External validity. The experimental results are generated based on the data set in this paper. Whether the results are applicable to other data sets needs to be studied further. The data set can be expanded in the future to ensure the results more generalizable.

Conclusion validity. In addition, due to the varied background of different project versions, the distribution of smells in different projects has its own characteristics, so it is difficult to ensure the generalizability of experimental results. Moreover, our work includes manual validation, which may impose subjective views on refactoring based on different programming style and level of skill. To minimize this threat, we involved 9 developers with different extent of experience, all of them have experience of developing commercial Python project.

6. Conclusion and future work

In this paper, we propose a Python code smell refactoring strategy generation based on association rule and correlation. We studied 9 active open-source Python projects on GitHub, involving 55,206 files. Experimental results show that among the files affected by any smell, the frequency of smell co-occurrence reached 28.77%. In addition, we find that there exist 7 groups of code smell with the highest degree of co-occurrence, among which LPL and LM are the most co-occurrence smell group. Besides, there are 10 groups of smell with the highest degree of correlation, LPL is most likely to cause the existence of other smells. Moreover, we verify the rationality of the generated refactoring routes by comparing them with manually assigned refactoring routes by developers having different experience through Kendall’s Tau coefficient. Result shows our result derives high inter-agreement with developers. Therefore, we recommend developers to refactor the smell in the order of {LPL→LLF}, {LPL→LBCL}, {LPL→LMC} or {LPL→LM→LC→CCC→MNC}.

For future work, we plan to (1) explore the relationship between other Python code smells, (2) investigate effective metric threshold for detecting correlated pairs of smells, (3) implementing a smell detector to achieve interactive correlated smell refactoring.

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, *Addison-Wesley Professional*, 1999.

[2] 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*, **1**(2) (2017) 1-29.

[3] Z.-F. Chen, L. Chen, W.W.Y. Ma, X.-Y. Zhou, Y.-M. Zhou, and B. Xu, Understanding metric-based detectable smells in Python software, *Inf. Softw. Technol.* **94** (2018) 14-29.

[4] R. Khatchadourian, Y.-M. Tang, M. Bagherzadeh, and S. Ahmed. A Tool for Optimizing Java 8 Stream Software via Automated Refactoring, in *Proc. 18th Int. Conf. Source Code Analysis and Manipulation, SCAM*, Madrid, Spain, 2018, pp. 34-39.

[5] W.-J. Zhou, Y. Zhao, G.-Q. Zhang, and X.-P. Shen. HARP: holistic analysis for refactoring Python-based analytics programs, in *Proc. 42nd Int. Conf. Software Engineering, ICSE*, Seoul, South Korea, 2020, pp. 506-517.

[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, Why Don’t Software Developers Use Static Analysis Tools to Find Bugs? in *Proc. 35th Int. Conf. Software Engineering, ICSE*, San Francisco, CA, USA, 2013, pp. 672-681.

[7] N. Sae-Lim, S. Hayashi, and M. Saeki, Context-Based Code Smells Prioritization for Prefactoring, in *Proc. 24th Int. Conf. Program Comprehension, ICPC*, Austin, TX, USA, 2016, pp. 1-10.

[8] L.-S. Bruno, A.-S.-B. Mariza, A.-M.-F. Kecia, An exploratory study on cooccurrence of design patterns and bad smells using software metrics, *SPE*, **49**(7) (2019) 1079-1113

[9] R. Agrawal, T. Imielinski, and A. Swami, Mining association rules between sets of items in large databases, in *Pro. 12th Int. Conf. ACM SIGMOD*, Washington, USA, 1993, pp. 207–216.

[10] H. Akoglu, User's guide to correlation coefficients, *J Emerg Med*, **18**(3) (2018) 91-93.

[11] F. Pecorelli, F. Palomba, D.-D. Nucci, and A.-D. Lucia, Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection, in *Proc. 27th IEEE Int. Conf. Program Comprehension*, *ICPC*, Madrid, Spain, 2019, pp. 93-104.

[12] F. Palomba, G. Bavota, M.-D. Penta, F. Fasano, R. Oliveto, and A.-D. Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, in *40th ACM/IEEE Int. Conf. Software Engineering*, *ICSE*, **23**(3) (2018) 1188-1221.

[13] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M.-D. Penta, and A.-D. Lucia, When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away), *IEEE Trans. Softw. Eng.* **43**(11) (2017) 1063-1088.

[14] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, A cooperative parallel search-based software engineering approach for code-smells detection, *IEEE Trans. Softw. Eng.* **40**(9) (2014) 841-861.

[15] F. Palomba, G. Bavota, M.-D. Penta, R. Oliveto, D. Poshyvanyk, and A.-D. Lucia, Mining version histories for detecting code smells, *IEEE Trans. Softw. Eng.* **41**(5) (2015) 462-489.

[16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le-Meur, Decor: A method for the specification and detection of code and design smells, *IEEE Trans. Softw. Eng.* **36**(1) (2010) 20-36.

[17] E. Murphy-Hill, and A.-P. Black, An interactive ambient visualization for code smells, in *Proc. 5th SOFTVIS*, *ACM*, New York, USA, 2010, pp. 5–14.

[18] N. Moha, Y.-G. Gueheneuc, A.-F. Le-Meur, L. Duchien, and A. Tiberghien, From a domain analysis to the specification and detection of code and design smells, *Formal Aspects of Computing*, **22**(3-4) (2010) 345–361.

[19] G. H. Travassos, F. Shull, M. Fredericks, and V.-R. Basili, Detecting defects in object-oriented designs: using reading techniques to increase software quality, in *Proc. 14th ACM SIGPLAN Conf. Object-oriented programming, systems, languages, and applications*, *ACM*, 1999, pp. 47–56.

[20] N. Sae-Lim, S. Hayashi, and M. Saeki, Context-based approach to prioritize code smells for prefactoring, *J Softw-Evol Proc*, **30**(6) (2018) 1-24.

[21] M.-A.-S. Bigonha, K. Ferreira, P. Souza, B. Sousa, M. Januario, and D. Lima, The usefulness of software metric thresholds for detection of bad smells and fault prediction, *Inf. Softw. Technol.* **115** (2019) 79-92.

[22] J. Padilha, J. Pereira, E. Figueiredo, J. Almeida, A. Garcia, and C. Sant'Anna, On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study, in *Proc. 26th CAiSE*, Thessaloniki, Greece, 2014, pp. 656-671.

[23] H. Liu, Z.-F. Xu, and Y.-Z. Zou, Deep Learning Based Feature Envy Detection, in *Proc. 33rd ACM/IEEE Int. Conf. Automated Software Engineering*, *ASE*, Montpellier, France, 2018, pp. 385-396.

[24] R. Terra, L.-F. Miranda, M.-T. Valente, R.-S. Bigonha, Qualitas.class Corpus: A Compiled Version of the Qualitas Corpus, *ACM SIGSOFT Software Engineering Notes*, **38**(5) (2013) 1-4.

[25] M. Alfadel, K. Aljasser, and M. Alshayeb, Empirical study of the relationship between design patterns and code smells, *PLOS ONE*, **15**(4) (2020) 1-35.

[26] F. Palomba, G. Bavota, M.-D. Penta, F. Fasano, R. Oliveto, and A.-D. Lucia, A large-scale empirical study on the lifecycle of code smell co-occurrences, *Inf. Softw. Technol.*, **99**(2018) 1-10.

[27] B.-B. Mayvan, A. Rasoolzadegan, and A.-J. Jafari, Bad smell detection using quality metrics and refactoring opportunities, *J. Softw. Evol. Process*. **32**(3) (2020) e2255.

[28] A.-M. Fard, and A. Mesbah, Jsnose: Detecting JavaScript code smells, in *Proc. 13th IEEE Int. Conf. Source Code Analysis and Manipulation, SCAM,* Eindhoven, Netherlands, 2013, pp. 116-125.

[29] F. Jaafar, A. Lozano, Y.-G. Gueheneuc, and K. Mens, On the Analysis of Co-occurrence of Anti-patterns and Clones, in *Proc. 17th IEEE Int. Conf. Software Quality, Reliability and Security, QRS-C,* Prague, Czech, 2017, pp. 274-284.

[30] F.-A. Fontana, V. Ferme, and M. Zanoni, Towards assessing software architecture quality by exploiting code smell relations, in *Pro. 2nd IEEE/ACM International Workshop on Software Architecture and Metrics*, *SAM*, Florence, Italy: ACM, 2015, pp. 1-7.

[31] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, The qualitas corpus: A curated collection of java code for empirical studies, in *Proc. 17th Asia Pacific Software Engineering Conference*, *APSEC*, Sydney, Australia, 2010, pp. 336-345.

[32] F. Palomba, M. Zanoni, F.-A. Fontana, A.-D. Lucia, and R. Oliveto, Toward a Smell-aware Bug Prediction Model, *IEEE Trans. Softw. Eng.* **45**(2) (2017) 194-218.

[33] R.-M. O'brien, A Caution Regarding Rules of Thumb for Variance Inflation Factors, *Quality & Quantity*, **41**(5) (2017) 673–690.