Do Developers Modify Dead Methods during the Maintenance of Java Desktop Applications?

Pietro Cassieri
University of Basilicata
Potenza, Italy
pietro.cassieri@studenti.unibas.it

Simone Romano
University of Salerno
Fisciano, Italy
siromano@unisa.it

Giuseppe Scanniello
University of Salerno
Fisciano, Italy
gscanniello@unisa.it

Genoveffa Tortora
University of Salerno
Fisciano, Italy
tortora@unisa.it

Danilo Caivano
University of Bari
Bari, Italy
danilo.caivano@uniba.it

ABSTRACT

Background: Dead code is a code smell. It can refer to code blocks, variables, parameters, fields, methods, classes, *etc.* that are unused and/or unreachable.
Aim: Results from past empirical studies indicate that dead code is widespread in both desktop and web-based software applications. Also, researchers have shown that both comprehensibility and maintainability of source code are negatively affected when dead code is present. Nevertheless, we still know little about maintenance operations involving dead code.
Method: We conducted an exploratory empirical study on 13 open-source Java desktop applications, whose software projects were hosted on GitHub, to provide preliminary evidence on whether, and to what extent, developers modify dead code—more specifically, dead methods—when they deal with the maintenance of open-source Java desktop applications.
Results: The most important results of our study can be summarized as follows: *(i)* developers modify dead methods; *(ii)* dead methods are modified to a different extent as compared to alive methods; *(iii)* developers spend time modifying dead methods that are removed in subsequent commits; and *(iv)* developers modify dead methods that are later revived to a different extent as compared to dead methods that are later removed.
Conclusions: One of the conclusions of our study is: developers should remove dead methods, whose presence and purpose are not properly documented, to avoid unnecessary modifications to dead methods during the maintenance of software applications.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Code smell, dead code, unused code, unreachable code, lava flow

EASE 2022, June 13–15, 2022, Gothenburg, Sweden

ACM ISBN 978-1-4503-9613-4/22/06…$15.00
https://doi.org/10.1145/3530019.3530032

ACM Reference Format:
Pietro Cassieri, Simone Romano, Giuseppe Scanniello, Genoveffa Tortora, and Danilo Caivano. 2022. Do Developers Modify Dead Methods during the Maintenance of Java Desktop Applications?. In *The International Conference on Evaluation and Assessment in Software Engineering 2022 (EASE 2022), June 13–15, 2022, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3530019.3530032

1 INTRODUCTION

Code smells are warning signs about potential problems in source code [51]. They contribute to the technical debt of a software application and possibly worsen source code maintainability [49]. The software engineering community has been showing great interest in code smells: several code-smell catalogs have been proposed (*e.g.,* [23, 30, 51]) and several studies have been conducted to increase our body of knowledge on code smells. These studies have covered: the detrimental effects of code smells on source code comprehensibility and maintainability, including their fault- and change-proneness (*e.g.,* [26, 33]); the spread and evolution of code smells (*e.g.,* [18, 49]); and developers' knowledge and perception about code smells (*e.g.,* [34, 52]). Tools for aiding developers in detecting code smells and refactoring software applications have been proposed too (*e.g.,* [14, 31]).

Different code smells have been characterized and listed in code-smell catalogs, one of them is *dead code.* This code smell can be defined as unnecessary source code since it is unused and/or unreachable (*i.e.,* never executed) [24, 29, 51]. Dead code is also known as *unused code* [51], *unreachable code* [22], or *lava flow* [16]. It can refer to code blocks, variables, parameters, fields, methods, classes, *etc.* For example, if a method is unused and/or unreachable, it is referred to as a *dead method.*

Dead code is supposed to have detrimental effects while comprehending and maintaining source code [22, 29]. This code smell seems to be also widespread in both industrial and open-source software applications. Boomsma *et al.* [15] reported that, in a subsystem of an industrial web-based software application, written in PHP, 30% of the subsystem's files were removed because they were actually dead. Eder *et al.* [21] also brought evidence on the spread of dead code in an industrial web-based software application. In particular, the researchers observed that, in the considered software application (written in .NET), 25% of methods were dead. Recently, Caivano *et al.* [17] have shown that dead methods are widespread

in open-source Java desktop applications;[1] these dead methods also survived for a long time before being removed or revived.

Empirical studies have been conducted to ascertain the supposed detrimental effects of dead code while comprehending and maintaining source code. Romano *et al.* [37, 41, 42], in their experiments, found that dead code significantly worsens the comprehensibility of unfamiliar source code. They also observed that some participants in the experiments unnecessarily modified dead code while carrying out a maintenance task. Similarly, Eder *et al.* [21], in their case study on an industrial web-based software application, gathered evidence on modifications to dead methods and reported that 48% of these modifications were unnecessary (*e.g.*, because dead methods were removed later). Nevertheless, we still know little about the maintenance operations that involve dead code. In particular, no study has investigated the commit histories of open-source Java desktop applications to understand whether, and to what extent, developers modify dead methods.

In this paper, we present the results of an exploratory study to determine whether, and to what extent, developers modify dead methods. To that end, we analyzed the commit histories of 13 open-source Java desktop applications, whose software projects were hosted on GitHub. Our study leverages the dataset by Caivano *et al.* [17], who analyzed a total of 1,044 commits in these 13 software applications and followed the evolution of 2,008 dead methods from their introduction up to their removal or revival, if occurred during the analyzed commit histories (*i.e.,* some dead methods could be still dead in the last analyzed commits). To detect dead methods, the researchers used *DCF* [39]—a tool representing the state of the art for the detection of dead methods in Java desktop applications. Our study is the first one that considers the commit histories of open-source Java desktop applications to understand whether, and to what extent, developers modify dead methods. The main results of our study can be summarized as follows: *(i)* developers modify dead methods; *(ii)* dead methods are modified to a greater extent than alive methods; *(iii)* developers spend time modifying dead methods that are removed in subsequent commits; and *(iv)* developers modify dead methods that are later revived to a greater extent than dead methods that are later removed.

Paper structure. In Section 2, we report background information about dead code and discuss work related to ours by highlighting the novelty of our study. In Section 3, we describe the design of our exploratory study, while we show the obtained results in Section 4. We discuss these results, by providing implications from both researcher and practitioner perspectives, in Section 5. In this section, we also delineate possible limitations of our study. We conclude the paper with final remarks in Section 6.

2 RELATED WORK

In this section, we first summarize work on the detection of dead code and then present empirical studies that contribute to the body of knowledge on this code smell.

[1]Caivano *et al.* [17] referred to desktop applications as both applications based on Graphical User Interface (GUI) frameworks like *Swing* or *SWT* (*i.e.,* GUI-based applications) and applications based on Command Line Interface (*i.e.,* CLI-based application). Therefore, desktop applications are not libraries, framework, web-based applications, mobile applications, *etc.* The above-mentioned definition of desktop applications is the same as we used in this paper.

2.1 Dead Code Detection

The detection of dead code can leverage dynamic and static code analyses. As for the former, Boomsma *et al.* [15] presented an approach for dead-file detection in web-based software applications written in PHP. In this approach, a target software application was monitored for a given period of time in order to determine the usage of PHP files. Files not used, in that period of time, were tagged as dead. The approach was then applied in a case study consisting of an industrial web-based software application written in PHP. The authors reported that, thanks to their approach, the developers of this application could remove 2,740 dead files. Similar to Boomsma *et al.* [15], Eder *et al.* [21] exploited dynamic code analysis to detect dead methods in their case study, consisting of a web-based software application written in .NET, in order to investigate modifications to dead methods. The main drawback of the approaches based on dynamic code analysis is that their dead-code detection capability strongly depends on the input data used to exercise the target software application.

As for the detection of dead code based on static code analysis, Romano *et al.* [40] proposed an approach, *DUM*, for the detection of dead methods in Java desktop applications. DUM first builds a graph-based representation of the target application where nodes are methods and directed edges are *caller-callee* relationships. Nodes reachable from starting nodes (*i.e.,* nodes deemed alive from the beginning like those representing main methods) are deemed alive, while nodes unreachable from the starting nodes are deemed dead. DUM was implemented in a tool, *DUM-Tool* [38], and its performance compared to that of *JTombstone* and *CodePro AnalytiX*. The performance of DUM-Tool in terms of correctness and accuracy of the detected dead methods was better than the two competing tools, while exhibiting good completeness in detecting dead methods. Later, Romano and Scanniello [39] proposed *DCF*, a tool based on *RTA*—an algorithm for constructing call graphs, which was known to be fast and to well-approximate virtual method calls [47]—for detecting dead methods in Java desktop applications. DCF exploits the RTA implementation available in *Soot* [50]. To detect dead methods in a Java desktop application, DCF identifies alive methods (*i.e.,* reachable nodes from some starting nodes like those representing main methods) in the call graphs built by using the RTA algorithm. Methods that are not alive are deemed dead. The performance of DCF was compared to that of JTombstone, CodePro AnalytiX, and DUM-Tool. DCF achieved better correctness (average precision equal to 84%) and accuracy (average f-measure equal to 85%) in detecting dead methods, as compared to the other tools. DCF also showed good completeness in detecting dead methods (average recall equal to 87%) comparable to that of DUM-Tool. Summing up, DCF can be considered the state of the art for the detection of methods that are dead in Java desktop applications.

The above-mentioned approaches (*i.e.,* [15, 21, 38–40]) take a refactoring perspective, rather than an optimization one, when detecting dead code. In other words, developers detect dead code because they mainly want to make their source code easier to comprehend and maintain, rather than making their source code faster and lighter. Taking a refactoring perspective when dealing with dead code has two practical implications. First, developers remove dead code from source code—*i.e.,* they are not interested

in removing dead code from software applications' dependencies (*e.g.,* frameworks and libraries) as done by Obbink *et al.* [32] in the context of JavaScript web-based software applications. Second, dead code removal is a permanent operation carried out on source code—in contrast to an optimization perspective where dead code removal can be temporary and can be carried out on intermediate representations of source code only (*e.g.,* bytecode in the case of Java). In the empirical study presented in this paper, we take a refactoring perspective.

2.2 Empirical Studies on Dead Code

Romano *et al.* [41] conducted an experiment with 47 participants to ascertain the supposed detrimental effects of dead code on source code comprehensibility and maintainability. In that experiment, a group of participants had to comprehend and then maintain a Java codebase containing dead code, while another group of participants had to do the same on a codebase without dead code. While the authors could not demonstrate the negative effect of dead code on source code maintainability, they found that dead code negatively affects source code comprehensibility. Later, Romano *et al.* [42] replicated that experiment three times. The results of these replications confirmed that dead code has a detrimental effect on source code comprehensibility. The authors also observed that during a maintenance task, some participants in the replications wasted time in modifying dead code that did not contribute in any way to the resolution of the maintenance task. One of the most remarkable differences with respect to our study is that: we analyze commit histories of open-source Java desktop applications to gather evidence on modifications to dead code, while Romano *et al.* [42] on that respect conducted human-oriented experiments. Romano *et al.* [42] also interviewed six developers to understand when and why dead code is introduced and how developers perceive and tackle this code smell. One of the main findings is that, although developers deem dead code harmful, they consciously introduce this code smell to anticipate future changes or consciously do not remove dead code because they think to use it someday.

Eder *et al.* [21] conducted a case study, consisting of an industrial web-based software application written in .NET, to investigate the modifications to dead methods. The authors monitored the application execution for two years in order to detect the usage of methods—methods not used in that period of time were deemed dead. The authors found that 8% of the dead methods underwent modifications during software maintenance. The authors also observed that 48% of the modifications to dead methods were unnecessary (*e.g.,* because dead methods were removed later). This is the closest study to ours. Nevertheless, there are several differences between our study and the one by Eder *et al.* [21]. The most important ones can be summarized as follows: *(i)* we considered a larger set of software applications (while Eder *et al.* [21] studied a single software application); *(ii)* we focused on Java desktop applications whose software projects were hosted on GitHub (while Eder *et al.* [21] studied an industrial web application written in .NET); and *(iii)* we studied other aspects related to dead code (*e.g.,* unlike Eder *et al.* [21], we compared dead methods that are later removed and those that are later revived in terms of their modifications). Therefore, our findings could provide support to the ones by Eder *et al.* [21] and, at the same time, complement them. In both cases, our findings should allow us to expand the body of knowledge on dead code.

Scanniello [43] exploited the Kaplan-Meier estimator to analyze the death of methods across different releases of five Java software applications. On two applications, the author observed that developers avoided the introduction of dead methods or removed dead methods as much as possible. Scanniello [44] also conducted a preliminary study to determine which software metrics are good predictors for the presence of dead methods. LOC (Lines Of Code) was the best predictor: the greater the LOC of a class, the higher the likelihood that its methods are dead.

Recently, Caivano *et al.* [17] investigated the spread and evolution of dead methods in the commit histories of 13 open-source Java desktop applications whose software projects were hosted on GitHub. They found that dead methods are widespread in the studied applications, survived for a long time before being removed or revived, are rarely revived, and are mostly dead since the creation of the corresponding methods (rather than becoming dead later).

The above-mentioned studies [17, 21, 41–44] contribute to the body of knowledge on dead code in several respect. However, only our study and those by Eder *et al.* [21] and Romano *et al.* [42] bring empirical evidence on modifications to dead code although, as highlighted before, they present remarkable differences.

3 EMPIRICAL STUDY

The *goal* of our study is to analyze the commit histories of open-source Java desktop applications with the *purpose* of understanding whether, and to what extent, dead methods are modified during the execution of maintenance operations. The *perspective* is that of both practitioners and researchers interested in dealing with dead methods for refactoring reasons. The *context* consists of open-source Java desktop applications hosted on GitHub.

3.1 Research Questions

Based on our study goal, we formulated the following Research Questions (RQs).

RQ1. *Do developers modify dead methods in Java desktop applications and, if so, to what extent?*

With this RQ, we want to understand whether dead methods, although unused in a given period of time, are modified during software maintenance of Java desktop applications and, if so, to what extent developers modify them (*e.g.,* how many dead methods are modified by at least one commit).

RQ2. *Are there differences between alive and dead methods in Java desktop applications in terms of modifications?*

This RQ aims to deepen the study of RQ1. We expect that dead methods are less intensively modified than alive methods because dead code is supposed not to be completely updated with respect to the rest of the software application [16, 30]. A positive answer to this RQ would allow confirming the belief that dead code is not completely updated as a software application evolves. In other words, the gathered empirical evidence would allow moving from common beliefs/opinions to actual facts.

Table 1: Some information about the studied applications.

Application	Description	# Stars	Last Analyzed Commit	# Analyzed Commits	# Alive Methods	# Dead Methods
4HWC Autonomous Car[*] [1]	A simulator that allows verifying the movements of an autonomous car	1	5a5c472	102	165	27
8_TheWeather [2]	An application that shows the current weather condition, as well as short- and long-term forecasts for an user-specified location	1	f6abd54	38	241	87
BankApplication [3]	An application that provides support for some banking operations	72	6856256	102	853	227
bitbox [4]	A utility tool for bit operations	1	af2af8b	15	367	210
Density Converter [5]	A tool that helps converting single or batches of images to specific formats and density versions	225	e70dcad	162	794	270
Deobfuscator-GUI [6]	It provides a GUI for a popular Java deobfuscator based on CLI.	112	deb003e	29	122	128
graphics-tablet [7]	A drawing application	0	8a3df4c	35	1232	407
JavaANPR [8]	An application to automatically recognize number plates from vehicle images	125	eff9acf	256	1374	237
javaman [9]	A Java implementation of the popular *Bomberman* game	0	7a03c36	58	380	142
JDM [10]	An application to manager the download of files	0	a435b4d	25	58	17
JPass [11]	An application to manage passwords	81	c6b13af	134	323	71
MBot [12]	An application to record and automate mouse and keyboard events	0	ff07dac	21	41	5
SMV APP [13]	An application to that provides support to a car repair shop	1	4c17370	67	340	180

[*] From here onwards, we refer to 4HWC Autonomous Car simply as 4HWC.

RQ3. *Do developers modify removed and revived dead methods in Java desktop applications and, if so, to what extent?*

We defined and studied RQ3 to understand whether and to what extent: *(i)* developers modify dead methods that will be never used (*i.e.,* we focus on *removed dead methods*) and *(ii)* developers update dead methods before reviving them (*i.e.,* we focus on *revived dead methods*). Removed dead methods are those dead methods that are removed from software applications during their maintenance. Any modification to removed dead methods can be intended as a waste of time because this kind of dead method is added to the software application, underwent modifications, and then is removed. On the other hand, revived dead methods are dead methods that are later used in software applications. Modifications to revived dead methods should not be considered a waste of time because these modified dead methods become operative in subsequent commits. It is worth mentioning that, in empirical studies like ours (*e.g.,* [18, 48]) where commit histories are analyzed to gather information about an event of interest, researchers are forced to analyze finite commit histories although the studied software applications continue to evolve with time. An event of interest—*i.e.,* the removal or revival of the dead method in our case—can thus occur outside the observation period [25]. Therefore, in our study, we dealt with *censored dead methods*, namely dead methods that, in the last analyzed commit, are still dead. In other words, for these dead methods, we could not record the event of interest (*i.e.,* the revival or removal of the dead method). We did not consider censored dead methods in RQ3 (and RQ4) because we are not aware whether these methods will be removed or revived later.

RQ4. *Are there differences between removed and revived dead methods in Java desktop applications in terms of modifications?*

With this RQ, we want to understand whether removed and revived dead methods are modified to a different extent. For example, if revived dead methods are modified more than removed dead methods, we could postulate that developers are aware of the presence of revived dead methods and then consciously used them as a means of anticipating changes and re/using source code.

3.2 Study Context and Planning

To conduct our study, we leveraged the dataset by Caivano *et al.* [17]. The authors detected dead methods and followed their evolution across the commit histories of 13 Java desktop applications (for a total of 1,044 commits) whose software projects were hosted on GitHub. Since dead methods are hard to detect without tool support [51], the authors exploited DCF [39]. As mentioned in Section 2.1, this tool represents the state of the art for the detection of dead methods in Java desktop applications. The authors detected dead methods in the master (*i.e.,* main) branch of the studied software applications. When detecting dead methods, the test directory of each software application was discarded—this was to avoid that methods belonging to test classes (*e.g.,* test methods) were detected as dead. Also, the detection of dead methods focused on "internal" methods only. That is, the authors did not detect dead methods in the dependencies (*e.g.,* libraries or frameworks) of the studied applications because these methods do not matter when taking a refactoring perspective.

In Table 1, we summarize some information on the software applications in Caivano *et al.*'s dataset [17]. The authors chose these applications to have a heterogeneous set of software applications in terms of: *(i)* size (*e.g.,* number of methods); *(ii)* number of stars, giving an indication of applications' popularity; *(iii)* lifespan, in terms of the number of commits; and *(iv)* application domain. Also, the choice of these software applications was driven by DCF (*e.g.,* it was conceived to detect dead methods in Java desktop applications).

Thanks to Caivano *et al.*'s dataset [17], we could know, for each detected dead method, the following information:

- *dead-method-introducing commit*, namely the commit in which the dead method is introduced into the source code (*i.e.,* when the method is created already dead or when it becomes dead after being alive in the previous commit);

Table 2: Values of %ModifiedMethods for dead and alive methods.

Application	%ModifiedMethods	
	Dead Methods	Alive Methods
All	32.9% (661/2,008)	38.1% (2,394/6,290)
4HWC	18.5% (5/27)	37.6% (62/165)
8_TheWeather	1.1% (1/87)	26.1% (63/241)
BankApplication	28.2% (64/227)	43.4% (370/853)
bitbox	52.9% (111/210)	45.8% (168/367)
Density Converter	35.2% (95/270)	35.5% (282/794)
Deobfuscator-GUI	11.7% (15/128)	21.3% (26/122)
graphics-tablet	31% (126/407)	36.1% (445/1,232)
JavaANPR	32.5% (77/237)	26.5% (364/1,374)
javaman	33.8% (48/142)	51.1% (194/380)
JDM	94.1% (16/17)	50% (29/58)
JPass	9.9% (7/71)	39.3% (127/323)
MBot	20% (1/5)	100% (41/41)
SMV APP	52.8% (95/180)	65.6% (223/340)

- *dead-method-removing commit* (if any), namely the commit in which the dead method is removed from the source code;
- *dead-method-reviving commit* (if any), namely the commit in which the dead method is made alive.
- *lifespan (of removed or revived dead method)* in terms of commits, namely the interval of consecutive commits from the first commit to the last one in which the method was detected as dead. Note that, if the method is removed or revived, the last commit in which the method was detected as dead is the commit that preceded the dead-method-removing or dead-method-reviving commit; otherwise, the last commit in which the method was detected as dead corresponds to the last analyzed commit.

The above-mentioned information was necessary, but not sufficient, to study our RQs. Therefore, we had to complement this information with others: *(i)* the lifespan of alive method (*i.e.,* the interval of consecutive commits from the first commit to the last one in which the method was alive) and *(ii)* which commits modified each method during the lifespan of that method. To that end, we used *PyDriller* [46]. To gather information on alive methods, we followed the same procedure as Caivano *et al.* [17] applied for dead methods. That is, for each software application in the used dataset, we considered only commits belonging to the master branch, we discarded the test directory, and we focused on "internal" methods only.

Table 3: Some descriptive statistics for #ModifyingCommits for dead and alive methods.

Application	#ModifyingCommits					
	Dead Methods			Alive Methods		
	Mean	Med	Max	Mean	Med	Max
All	0.528	0	16	0.736	0	15
4HWC	0.259	0	3	0.648	0	7
8_TheWeather	0.011	0	1	0.577	0	11
BankApplication	0.414	0	2	0.632	0	6
bitbox	0.967	1	2	0.768	0	3
Density Converter	0.837	0	16	0.724	0	13
Deobfuscator-GUI	0.195	0	4	0.32	0	10
graphics-tablet	0.314	0	2	0.582	0	14
JavaANPR	0.793	0	6	0.811	0	10
javaman	0.38	0	3	0.753	1	6
JDM	0.941	1	1	0.828	0.5	10
JPass	0.099	0	1	0.808	0	15
MBot	0.2	0	1	2.341	2	10
SMV APP	0.611	1	4	1.247	1	14

3.3 Dependent and Independent Variables

In our empirical study, we defined two independent variables. We named the former as KindOfMethod. It is a nominal variable and has the following levels: Dead (indicating that a method is dead) and Alive (indicating that a method is alive). We named the latter as KindOfDeadMethod. We used it to distinguish between the considered kinds of dead method (*i.e.,* revived and removed). This independent variable is nominal and has the following levels: Revived and Removed.

To answer our RQs, we took into account the following dependent variables: #ModifyingCommits and %ModifiedMethods. The former is computed by counting how many commits modified each method during its lifespan. As for the latter, we computed the proportion of methods that were modified at least once during their lifespan (*i.e.,* the relative number of methods whose value for #ModifyingCommits was greater than zero).

To study RQ1 and RQ2, we manipulated the KindOfMethod independent variable. In other words, we computed the values of #ModifyingCommits and %ModifiedMethods for the levels of KindOfMethod. As far as RQ3 and RQ4 are concerned, we manipulated the KindOfDeadMethod independent variable—*i.e.,* we computed the values of #ModifyingCommits and %ModifiedMethods for the levels of that independent variable.

4 RESULTS

We present the results of our study according to the defined RQs.

4.1 RQ1: Do Developers Modify Dead Methods?

In Table 2, we show the values for %ModifiedMethods for dead and alive methods while, in Table 3, we report mean, median (med), and maximum (max) values of #ModifyingCommits. We can observe in Table 2 that 32.9% of dead methods are modified at least once (*i.e.,* 661 out of 2,008 dead methods) by considering all software applications together. When considering the software applications individually, we can note that the percentage of modified dead methods is greater than zero in all cases. This means that there is at least one modified dead method in each software application (see Table 2). Furthermore, in nine out of 13 software applications, the

percentage of modified dead methods is equal to or greater than 20%. However, there is variability in the values of `%ModifiedMethods` (ranging from 1.1% for 8_TheWeather to 94.1% for JDM). Based on these results, we can assert that developers modify dead methods although the percentage of modified dead methods depends on the software application.

By looking at the values of `#ModifyingCommits` aggregated from all software applications (Table 3), we can note that each dead method underwent on average less than one modification (0.528) in its lifespan. The median value is zero so suggesting that the greater part of dead methods is never modified. We can also note that there is a dead method (in Density Converter) that underwent 16 modifications in its lifespan. Furthermore, nine out of 13 software applications have dead methods that underwent more than one modification. The results for `#ModifyingCommits` show, with stronger evidence, that developers modify dead methods.

Answer to RQ1: It seems that developers modify dead methods in Java desktop applications although the percentage of modified dead methods and the extent to which these methods are modified depends on the software application.

4.2 RQ2: Are There Differences between Alive and Dead Methods in Terms of Modifications?

We can observe in Table 2 that 2,394 alive methods (out of 6,290) are modified at least once when studying all software applications together. In other words, 38.1% of alive methods are modified at least once. The results in Table 2 also show that alive methods are modified more than dead methods (38.1% vs. 32.9%). We further investigated this difference (in terms of `%ModifiedMethods`) by performing statistical inference. In particular, we used the *Chi-squared test* [35] to study if there is a statistically significant difference between the proportion of modified dead methods and that of modified alive methods. This test returned a p-value less than 0.001, which is lower than the fixed α value equal to 0.05.[2] This means that the difference between these proportions is statistically significant (and in favor of alive methods as suggested by the values in Table 2). To estimate the size of such a difference, we applied *Cramer V* [20], an effect size measure for the Chi-squared test. We obtained 0.031 as the Cramer V value. This indicates that the effect size is negligible.[3] We also analyzed the differences between alive and dead methods, in terms of `%ModifiedMethods`, across the software applications. We observed that in ten out of 13 software applications, the percentage of modified alive methods is greater than the one of modified dead methods (see Figure 1).

The descriptive statistics for `#ModifyingCommits` (summarized in Table 3) are consistent with the values for `%ModifiedMethods`: alive methods are, on average, modified more than dead methods (0.736 vs. 0.528), when considering all software applications

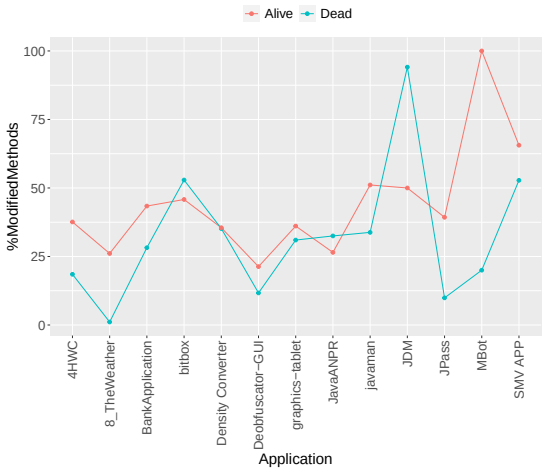

Figure 1: Line plot for `%ModifiedMethods` for dead and alive methods.

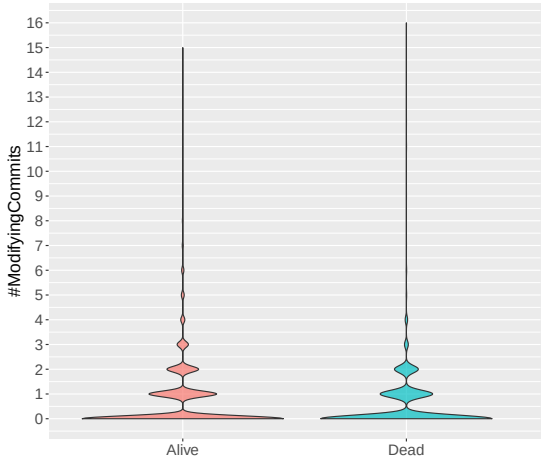

Figure 2: Violin plot for `#ModifyingCommits` for alive and dead methods.

together. In Figure 2, we show, by exploiting a violin plot, the distributions of the `#ModifyingCommits` values for alive and dead methods. The violins for alive and dead methods are skewed—suggesting that the data are not normally distributed and follow log-normal distributions—and most of the values are zero, one, and two. However, the density of methods with a number of modifying commits greater than zero is higher for alive methods. To verify if there is a statistically significant difference between alive and dead methods with respect to `#ModifyingCommits`, we applied the (two-sided) *Mann-Whitney U test* [28]. We opted for this non-parametric test because the considered distributions were not normal (the p-values returned by *Shapiro test* [45] were less than $\alpha = 0.05$). The Mann-Whitney U test returned a p-value less than 0.001 (thus lower than $\alpha = 0.05$). This indicates that such a difference is statistically significant (and in favor of alive methods as suggested by the values in Table 3). To estimate the magnitude of this difference, we computed

[2]For any performed statistical test, we fixed (as customary) an α value equal to 0.05. It indicates 5% chance of a Type-I error occurring (*i.e.*, rejecting the null hypothesis when it is true). When a statistical test returns a p-value less than the fixed α value, we can reject the null hypothesis and then accept the alternative one.

[3]In the case of two categories, the magnitude of a difference, estimated by means of Cramer V, is deemed: *negligible*, if $V < 0.1$; *small*, if $0.1 \leq V < 0.3$; *medium*, if $0.3 \leq V < 0.5$; or *large*, otherwise [27].

Table 4: Values of %ModifiedMethods for removed and revived dead methods.

%ModifiedMethods	
Removed Dead Methods	**Revived Dead Methods**
29.2% (249/852)	37% (129/349)

the *Cliff's* δ effect size [19]. The effect size is negligible, being the attained value equal to 0.065.[4] We can also observe that alive methods are, on average, modified more than dead methods on ten software applications out of 13 so confirming the overall outcome (Table 3).

Further Analysis. To increase our confidence in the results shown above, we considered an additional dependent variable that takes into account the lifespan of dead and alive methods. This variable, we named Normalized#ModifyingCommits, is computed as $\frac{\#ModifyingCommits}{lifespan}$. Normalized#ModifyingCommits assumes values in the interval [0, 1]. Zero means that a method is never modified (*i.e.,* there is no commit modifying that method), while one indicates that all commits in the lifespan of a method modify it. Given a method, a high value for Normalized#ModifyingCommits implies that developers were very active, in terms of modifications, on the source code of that method. To determine if there was a statistically significant difference (in terms of Normalized#ModifyingCommits) between dead and alive methods, we applied the (two-sided) Mann-Whitney U test (since the distributions were not normal as suggested by the results of the Shapiro test). The Mann-Whitney U test returned a p-value equal to 0.018 ($\alpha = 0.05$), so indicating that there is a statistically significant difference. The difference was in favor of alive methods. To estimate the size of this difference, we computed the Cliff's δ effect size that was negligible (0.03). Summarizing, the results of this further analysis are consistent with those obtained for both %ModifiedMethods and #ModifyingCommits.

Answer to RQ2: Dead methods are modified to a different extent with respect to alive methods.

4.3 RQ3: Do Developers Modify Removed and Revived Dead Methods?

In Table 4, we show the values for %ModifiedMethods for removed and revived dead methods while, in Table 5, we report mean, median, and max values of #ModifyingCommits. As for removed dead methods, the value of %ModifiedMethods (see Table 4) is 29.2% (*i.e.,* 249 removed dead methods underwent modifications before their removal, out of a total of 852 removed dead methods). This indicates that developers modify dead methods that are later removed so spending time in useless modifications to source code. The mean value of #ModifyingCommits is 0.434 (see Table 5). This means that removed dead methods are modified, on average, by less than one commit. The median value of #ModifyingCommits is zero while the max is 11—it was observed only once for Density Converter. The distribution of the values for #ModifyingCommits is depicted in Figure 3 (left side). The violin is skewed and most of the values are zero and one with a noticeable peak for the former value. The data follow a log-normal five-modal distribution.

As far as revived dead methods are concerned, 37% is the value for %ModifiedMethods (see Table 4). That is, 129 revived dead methods underwent modifications before their revival, out of a total of 349 revived dead methods. This indicates that developers modify dead methods that are later revived so allowing us to assume that developers do not waste time implementing these modifications because modified dead methods became operative in subsequent commits. As shown in Table 5, the median value of #ModifyingCommits is zero, while the max is 4—it was observed only once for Density Converter. In Figure 3 (right side), we can observe the violin depicting the distribution of the #ModifyingCommits values for revived dead methods. This violin is skewed and most of the values are zero and one. The data are not normally distributed and follow a log-normal three-modal distribution.

Answer to RQ3: Developers modify removed dead methods in Java desktop applications. This allows us to postulate that developers spend time implementing useless modifications to source code. Developers also modify revived dead methods. We can thus suppose that developers are aware that these methods are dead and use them as a means of anticipating changes and re/using source code.

Table 5: Some descriptive statistics for #ModifyingCommits for removed and revived dead methods.

%ModifyingCommits					
Removed Dead Methods			**Revived Dead Methods**		
Mean	**Med**	**Max**	**Mean**	**Med**	**Max**
0.434	0	11	0.418	0	4

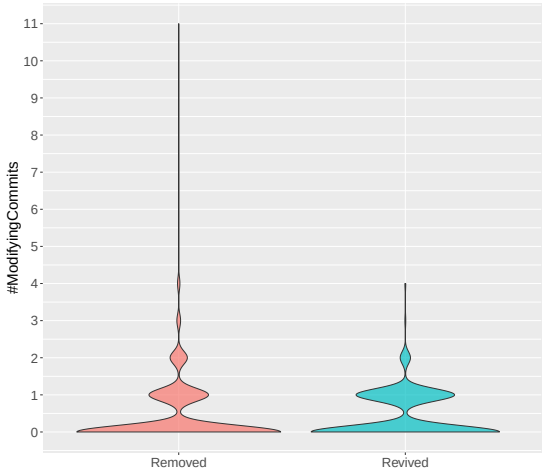

Figure 3: Violin plot for #ModifyingCommits for removed and revived dead methods.

[4]The magnitude of a difference, estimated by means of Cliff's δ, is deemed: *negligible*, if $\delta < 0.147$; *small*, if $0.147 \leq |\delta| < 0.33$; *medium*, if $0.33 \leq |\delta| < 0.474$; or *large*, otherwise [36].

4.4 RQ4: Are There Differences between Removed and Revived Dead Methods in Terms of Modifications?

The values for `%ModifiedMethods` (see Table 4) show a difference between removed and revived dead methods. In particular, the percentage of modified dead methods is greater for revived ones (37% vs. 29.2%). To better investigate such a difference, we performed the Chi-squared test [35]. This test indicates a statistically significant difference since the p-value (*i.e.,* 0.011) was less than the fixed α value (*i.e.,* 0.05). This difference is in favor of revived dead methods and the effect size is negligible—the value of Cramer V is 0.076.

As for `#ModifyingCommits`, there is not a huge difference in the mean values (see Table 5). The violin plot in Figure 3 confirms this outcome even if we can observe slight differences in the distributions of the values for `#ModifyingCommits` between removed and revived dead methods. For example, we can notice that the density of ones is higher for revived dead methods and the violin is shorter. We verified if there is a significant difference between removed and revived dead methods with respect to `#ModifyingCommits`. To that end, we ran the Mann-Whitney U test because the `#ModifyingCommits` values for removed and revived dead methods were not normally distributed (as the results of the Shapiro test suggested). The Mann-Whitney U test returned a p-value equal to 0.057 (> 0.05, *i.e.,* the the fixed α value). Therefore, we do not find a statistically significant difference between removed and revived dead methods with respect to `#ModifyingCommits`.

Further Analysis. To increase our confidence in the results shown above, we performed an additional statistical analysis on `Normalized#ModifyingCommits`. We applied the (two-sided) Mann-Whitney U test because the distributions were not normal (as the Shapiro-test p-values suggested). The Mann-Whitney U test returned a p-value equal to 0.002 (thus less than $\alpha = 0.05$). That is, there is a statistically significant difference between revived and removed dead methods. This difference is in favor of revived dead methods—*i.e.,* developers were more active on such a kind of dead method. To estimate the size of this difference, we computed the Cliff's δ effect size, which was negligible (0.093). Summarizing, the results of this further analysis are consistent with those obtained for `%ModifiedMethods`—in Section 5, we discuss why the results for `Normalized#ModifyingCommits` differ from the ones for `#ModifyingCommits`.

Answer to RQ4: Removed and revived methods are modified to a different extent. That is, the proportion of modified revived dead methods is greater than the proportion of removed dead methods. In addition, the number of commits that modify revived and removed dead methods seems to be nearly similar. We also noted that, when taking into account the variability in the lifespans of dead methods, revived ones underwent a higher number of modifications as compared to remove ones.

5 DISCUSSION

In this section, we discuss the results of our empirical study, by also delineating implications for both practitioners and researchers, and then threats to validity.

5.1 Overall Discussion and Implications

We found that developers modify dead methods in (open-source) Java desktop applications. Similarly, Romano *et al.* [42] experimentally observed that developers modified dead code while performing maintenance tasks on Java desktop applications. These outcomes are coherent with the one by Eder *et al.* [21], who observed that dead methods were modified in a (proprietary) industrial web-based application written in .NET. Based on our outcomes and those of past empirical studies (*i.e.,* [21, 42]), we can postulate that developers modify dead code regardless of the programming language used to implement the software application. It also seems that dead code is modified whatever the kind of application (*i.e.,* desktop vs. web-based), the kind of license (*i.e.,* open-source vs. proprietary), and the application domain (*e.g.,* from conversion of images to weather forecast in our study, and insurance in the study by Eder *et al.* [21]) is. This is of interest for **practitioners** since they could bump into dead methods during software maintenance regardless of the software application. Therefore, practitioners should be informed about the presence of dead methods so that they can adequately deal with them (*e.g.,* evolving dead methods, or removing them, depending on whether developers have planned, or not, to make them operative later) when maintaining source code. **Researchers** could be interested in further studying how, and to what extent, dead code is modified. For example, we observed that the extent to which dead methods are modified depends on the software application; therefore, researchers could be interested in studying which factors can affect modifications to dead code.

We observed that developers modified alive methods to a greater extent than dead methods. This outcome seems to confirm the belief that dead code does not evolve as alive code—*i.e.,* while a given application evolves, dead code is frozen or almost frozen [16, 30]. **Practitioners** should avoid as much as possible the introduction of dead methods. In case dead methods are consciously introduced (*e.g.,* dead methods will be operative in a few commits), all developers should be informed about their presence and purpose (*e.g.,* by commenting these aspects in the source code).

Developers modify removed dead methods in Java desktop applications, so allowing us to postulate that they unnecessarily spend time modifying dead methods that were later removed. This outcome is consistent with those by Eder *et al.* [21] and Romano *et al.* [42], who observe that developers unnecessarily modify dead code when performing maintenance tasks. A plausible justification behind modifications to dead methods that are later removed is that developers are not aware that the modified methods are actually dead [42]. To avoid waste of time, we recommend **practitioners** to remove dead methods, whose presence and purpose are not properly documented. After all, if a developer needs a removed dead method in the future, he/she can exploit the version control system to retrieve it. In other words, we make Martin's recommendation [30] ours: "Methods that are never called should be discarded. Keeping dead code around is wasteful. Don't be afraid to delete the function. Remember, your source code control system still remembers it." It is worth mentioning that dead methods could be introduced inadvertently and then unconsciously modified. For such a reason, we devise the use of just-in-time tools aimed to detect dead methods in real time (*i.e.,* while a developer codes) and provide

instant warnings about the introduction and modification of dead methods. **Researchers** could define approaches and prototypes of just-in-time tools to detect dead methods.

Unlike removed dead methods, modifications to revived dead methods should not be considered a waste of time. In fact, modifications to revived dead methods should be intended to update dead methods before making them operative in subsequent commits. We deem useful the documentation (*e.g.,* through code comments) of dead methods to be revived because they can be intended as a means to anticipate changes and re/use source code. This point is clearly of interest to **practitioners**. On the other hand, **researchers** might be interested in defining approaches to document dead methods that developers have planned to revive in subsequent commits. Developers could be warned when the lifespan of a dead method, not yet revived, becomes too large (*e.g.,* when the lifespan is larger than the average lifespan of revived dead methods).

Empirical evidence from our study shows that removed and revived dead methods are modified to a different extent. We also observed that the number of commits modifying revived and removed dead methods is nearly the same. We deem that this is due to the fact that the lifespan of removed methods is larger than the lifespan of revived ones—*i.e.,* the likelihood to modify a removed dead method is greater. The results of our further analysis (reported in Section 4.4) support our claim since we observed that, when taking into account the variability in the lifespans of dead methods, revived ones underwent a higher number of modifications as compared with removed ones. Summing up, removed and revived methods are treated in a different way in Java desktop applications. This suggests that: developers are aware of the presence of revived dead methods and then consciously use them as a means of anticipating changes and re/using code; on the other hand, developers are unaware of the presence of removed dead methods and unconsciously modify them. Our preliminary finding poses the basis for future research and this is clearly of interest for **researchers**. For example, we suggest researchers that plan surveys and interviews with the original developers of the studied software applications to complement our findings from a qualitative perspective.

Finally, our study has the merit of improving our body of knowledge on dead methods. In particular, we gathered evidence on the modifications to dead methods in open-source Java desktop applications. Also, we complemented/supported the findings of past research on this code smell (*i.e.,* [21, 42]). Nevertheless, we cannot claim that our outcomes are conclusive. We believe that these outcomes can justify **researchers** to conduct more resource- and time-demanding research on dead code and the modifications it undergoes. For example, researchers could conduct studies similar to ours but on a larger scale.

5.2 Threats to Validity

We discuss threats that might affect our results with respect to external, conclusion, construct, internal, and reliability validity.

External validity threats concern the possibility of generalizing experimental results. When datasets are small and empirical research relies on them, generalizing outcomes poses then a threat to external validity. Although we gather preliminary empirical evidence on whether, and to what extent, developers modify dead methods, caution is needed when generalizing our outcomes since the exploratory nature of our empirical study. For example, the software applications studied in our empirical study and their number might affect the generalizability of our results. To mitigate external validity threats, we advise replications of our study on a larger dataset. In this regard, we believe that our preliminary empirical evidence can justify researchers to conduct replications.

Conclusion validity threats (sometimes referred to as statistical conclusion validity) are concerned with issues that affect the ability to draw the correct conclusion about relations in the observations. In our data analysis, we applied proper statistical tests to mitigate conclusion validity threats. For example, we verified normality assumptions to apply parametric tests.

Construct validity threats concern the relationship between theory and observation. The metrics we used to answer our RQs might pose a threat to construct validity. However, there is no accepted metric to quantitatively assess the constructs studied in our study. Construct validity could be also affected by the dataset. Our study leverages the dataset by Caivano *et al.* [17], who used the DFC tool [39] to detect dead methods. The validity of this tool was assessed by comparing it with other dead-code detection tools on a ground truth. The results of this comparison show that DCF outperformed baseline tools in terms of correctness and accuracy of the detected dead methods, while exhibiting high completeness in detecting dead methods. Finally, we identified dead methods, in the commit histories, by using their signature and the fully qualified name of the belonging classes. This might affect the results.

Internal validity threats concern factors internal to an empirical study that might have influenced results. Since we leverage an existing dataset, we inherit threats concerning how this dataset was built. For example, when a commit did not compile, Caivano *et al.* [17] skipped that commit. This was unavoidable because, to detect dead methods, DCF needs bytecode. The incapability of compiling open-source applications at a commit level is an inherent limitation to any study similar to ours.

Reliability validity threats concern the possibility of replicating an empirical study. To deal with such a kind of threat, we make our raw data available on the web.[5] Also, information on the used dataset can be found in the paper by Caivano *et al.* [17].

6 CONCLUSIONS

We present an exploratory empirical study to provide preliminary evidence on whether, and to what extent, developers modify dead methods when they perform maintenance operations on Java desktop applications. To that end, we studied the commit histories of 13 open-source Java desktop applications, whose software projects were hosted on GitHub. The most important results from our study can be summarized as follows: *(i)* developers modify dead methods; *(ii)* dead methods are modified to a different extent as compared to alive methods; *(iii)* developers spend time modifying dead methods that are removed in subsequent commits; and *(iv)* developers modify dead methods that are later revived to a different extent than dead methods that are later removed. Based on our results, we delineated a set of implications for researchers and practitioners. Among them, the most important implication is: practitioners

[5] https://doi.org/10.6084/m9.figshare.19636080.v1

should remove dead methods, whose presence and purpose are not properly documented, to avoid waste of time in the future (*i.e.,* unnecessary modifications to dead methods).

Although our study allows increasing our body of knowledge on dead methods, we cannot claim that the outcomes of our study are conclusive (*i.e.,* in line with the exploratory nature of our study, we report preliminary empirical evidence). Having said that, our study has the merit of justifying future research on the modifications to dead methods like: interviews/surveys with the original developers of the studied applications (in order to complement our findings from a qualitative perspective) and replications of our study on a larger scale (in order to extend the external validity of our results).

REFERENCES

[1] 2021. 4HWC Autonomous Car. https://github.com/4hwc/4HWCAutonomousCar.
[2] 2021. 8_TheWeather. https://github.com/workofart/WeatherDesktop.
[3] 2021. BankApplication. https://github.com/derickfelix/BankApplication.
[4] 2021. bitbox. https://github.com/fusiled/bitbox.
[5] 2021. Density Converter. https://github.com/patrickfav/density-converter.
[6] 2021. Deobfuscator-GUI. https://github.com/java-deobfuscator/deobfuscator-gui.
[7] 2021. graphics-tablet. https://github.com/alexdoublesmile/5-app-graphics-tablet.
[8] 2021. JavaANPR. https://github.com/oskopek/javaanpr.
[9] 2021. javaman. https://github.com/malluce/javaman.
[10] 2021. JDM. https://github.com/iamabs2001/JDM.
[11] 2021. JPass. https://github.com/gaborbata/jpass.
[12] 2021. MBot. https://github.com/znyi/MBot.
[13] 2021. SMV APP. https://github.com/bfriscic/ZavrsniRad.
[14] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2010. A Two-Step Technique for Extract Class Refactoring. In *Proceedings of International Conference on Automated Software Engineering*. ACM, 151–154.
[15] Hidde Boomsma, B. V. Hostnet, and Hans-Gerhard Gross. 2012. Dead code elimination for web systems written in PHP: Lessons learned from an industry case. In *Proceedings of International Conference on Software Maintenance*. IEEE, 511–515.
[16] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (1st ed.). John Wiley & Sons, Inc.
[17] Danilo Caivano, Pietro Cassieri, Simone Romano, and Giuseppe Scanniello. 2021. An Exploratory Study on Dead Methods in Open-source Java Desktop Applications. In *Proceedings of International Symposium on Empirical Software Engineering and Measurement*. ACM.
[18] Alexander Chatzigeorgiou and Anastasios Manakos. 2014. Investigating the Evolution of Code Smells in Object-oriented Systems. *Innovations in Systems and Software Engineering* 10, 1 (2014), 3–18.
[19] Norman Cliff. 1996. *Ordinal methods for behavioral data analysis*. Psychology Press.
[20] Harald Cramer. 1946. *Mathematical Methods of Statistics*. Princeton University Press. 282 pages.
[21] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. 2012. How much does unused code matter for maintenance?. In *Proceedings of International Conference on Software Engineering*. IEEE, 1102–1111.
[22] Amin Milani Fard and Ali Mesbah. 2013. JSNOSE: Detecting JavaScript Code Smells. In *Proceedings of International Working Conference on Source Code Analysis and Manipulation*. IEEE, 116–125.
[23] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code* (1st ed.). Addison-Wesley.
[24] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. 2020. Is Static Analysis Able to Identify Unnecessary Source Code? *ACM Trans. Softw. Eng. Methodol.* 29, 1 (2020), 6:1–6:23.
[25] Rupert G. Miller Jr. 2011. *Survival Analysis* (2nd ed.). John Wiley and Sons.
[26] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 2009. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *Proceedings of Working Conference on Reverse Engineering*. IEEE, 75–84.
[27] S.S. Mangiafico. 2016. *Summary and Analysis of Extension Program Evaluation in R*. 282 pages.
[28] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60.
[29] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of International Conference on Software Maintenance*. IEEE, 381–384.
[30] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1st ed.). Prentice Hall.
[31] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 36, 1 (2010), 20–36.
[32] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia Lago. 2018. An extensible approach for taming the challenges of JavaScript dead code elimination. In *Proceedings of International Conference on Software Analysis, Evolution and Reengineering*. 291–401.
[33] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.
[34] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *Proceedings of International Conference on Software Maintenance and Evolution*. IEEE, 101–110.
[35] Karl Pearson. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50, 302 (1900), 157–175.
[36] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?. In *Annual Meeting of the Florida Association of Institutional Research, February*. 1–3.
[37] Simone Romano. 2018. Dead Code. In *Proceedings of International Conference on Software Maintenance and Evolution*. IEEE, 737–742.
[38] Simone Romano and Giuseppe Scanniello. 2015. DUM-Tool. In *Proceedings of International Conference on Software Maintenance and Evolution*. IEEE, 339–341.
[39] Simone Romano and Giuseppe Scanniello. 2018. Exploring the Use of Rapid Type Analysis for Detecting the Dead Method Smell in Java Code. In *Proceedings of EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 167–174.
[40] Simone Romano, Giuseppe Scanniello, Carlo Sartiani, and Michele Risi. 2016. A Graph-based Approach to Detect Unreachable Methods in Java Software. In *Proceedings of Symposium on Applied Computing*. ACM, 1538–1541.
[41] Simone Romano, Christopher Vendome, Giuseppe Scanniello, and Denys Poshyvanyk. 2016. Are unreachable methods harmful? Results from a controlled experiment. In *Proceedings of International Conference on Program Comprehension*. IEEE, 1–10.
[42] Simone Romano, Christopher Vendome, Giuseppe Scanniello, and Denys Poshyvanyk. 2020. A Multi-Study Investigation into Dead Code. *IEEE Transactions on Software Engineering* 46, 1 (2020), 71–99.
[43] Giuseppe Scanniello. 2011. Source code survival with the Kaplan Meier. In *Proceedings of International Conference on Software Maintenance*. 524–527.
[44] Giuseppe Scanniello. 2014. An Investigation of Object-Oriented and Code-Size Metrics as Dead Code Predictors. In *Proceedings of EUROMICRO Conference on Software Engineering and Advanced Applications*. 392–397.
[45] S. Shapiro and M. Wilk. 1965. An analysis of variance test for normality. *Biometrika* 52, 3-4 (1965), 591–611.
[46] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 908–911.
[47] Frank Tip and Jens Palsberg. 2000. Scalable Propagation-based Call Graph Construction Algorithms. In *Proceedings of Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 281–293.
[48] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An Empirical Investigation into the Nature of Test Smells. In *Proceedings of International Conference on Automated Software Engineering*. ACM, 4–15.
[49] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.
[50] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers*. IBM, 214–224.
[51] William C. Wake. 2003. *Refactoring Workbook* (1st ed.). Addison-Wesley.
[52] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *Proceedings of Working Conference on Reverse Engineering*. IEEE, 242–251.