An Exploratory Study on Dead Methods in Open-source Java Desktop Applications

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

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

Simone Romano
University of Bari
Italy
simone.romano@uniba.it

Giuseppe Scanniello
University of Basilicata
Italy
giuseppe.scanniello@unibas.it

ABSTRACT

Background. Dead code is a code smell. It can refer to code blocks, fields, methods, *etc.* that are unused and/or unreachable. Empirical evidence shows that dead code harms source code comprehensibility and maintainability in software applications. Researchers have gathered little empirical evidence on the spread of dead code in software applications. Moreover, we know little about the role of this code smell during software evolution.
Aims. Our goal is to gather preliminary empirical evidence on the spread and evolution of dead methods in open-source Java desktop applications. Given the exploratory nature of our investigation, we believe that its results can justify more resource- and time-demanding research on dead methods.
Method. We quantitatively analyzed the commit histories of 13 open-source Java desktop applications, whose software projects were hosted on GitHub, for a total of 1,044 commits. We focused on dead methods detected at a commit level to investigate the spread and evolution of dead methods in the studied applications. The perspective of our explorative study is that of both practitioners and researchers.
Results. The most important take-away results can be summarized as follows: *(i)* dead methods seems to affect open-source Java desktop applications; *(ii)* dead methods generally survive for a long time, in terms of commits, before being "buried" or "revived;" *(iii)* dead methods are rarely revived; and *(iv)* most dead methods are dead since the creation of the corresponding methods.
Conclusions. We conclude that developers should carefully handle dead methods in open-source Java desktop applications since this code smell is harmful, widespread, rarely revived, and survives for a long time in software applications. Our results also justify future research on dead methods.

ESEM '21, October 11–15, 2021, Bari, Italy
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8665-4/21/10…$15.00
https://doi.org/10.1145/3475716.3475773

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Code smell, dead code, unused code, exploratory study, Java desktop applications, open-source, GitHub

ACM Reference Format:
Danilo Caivano, Pietro Cassieri, Simone Romano, and Giuseppe Scanniello. 2021. An Exploratory Study on Dead Methods in Open-source Java Desktop Applications. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '21), October 11–15, 2021, Bari, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3475716.3475773

1 INTRODUCTION

Code smells (also known as *bad smells in code*) are indicators of potential problems in source code [47]. In the last two decades, the Software Engineering (SE) community has displayed an increasing interest in code smells. Catalogs of code smells have been proposed [19, 47] and various aspects related to code smells have been investigated in the SE field. A not exhaustive list of these aspects follows: the negative impact of code smells on source code comprehensibility and maintainability [22], the increased fault- and change-proneness of source code files containing code smells [25, 30], their spread and evolution [16, 45], and the knowledge and perception of developers about code smells [31, 48]. In the SE field, a number of supporting tools for detecting code smells have been proposed as well [28].

Code-smell catalogues characterize and list different code smells, one of them is *dead code* (also known as *unused code* [47], *unreachable code* [18], or *lava flow* [15]). This smell can be defined as unnecessary source code since it is unused and/or unreachable (*i.e.,* never executed) [21, 26, 47]. Dead code can refer to code blocks, fields, methods, *etc.* For example, if a method is unused and/or unreachable, it is referred to as a *dead method.*

The presence of dead code in a software application is claimed to harm its source code comprehensibility and maintainability [18, 26]. Recently, Romano *et al.* [34, 39, 40] have shown that these claims are well-founded since they observed, in their experiments, that dead code significantly hinders the comprehensibility of unfamiliar source code and it also negatively affects the maintainability of unfamiliar source code. Nevertheless, little empirical evidence has been gathered on the spread of dead code in software applications. For example, Boomsma *et al.* [14] reported that, in a subsystem of an industrial web-based software application written in PHP, the developers removed 30% of the subsystem's files because these files were actually dead. Eder *et al.* [17] observed that, in an industrial web-based software application written in .NET, 25% of methods were dead. Moreover, the role of dead code during the evolution of software applications has received little empirical attention although a few preliminary investigations have shown that developers uselessly spend some effort modifying dead code. In particular, Eder *et al.* [17] reported that, during source code maintenance and evolution tasks, 7.6% of the modifications affected dead methods. Based on our study of the state of the art, we can conclude that: *(i)* dead code (and dead methods, in particular) harms source code comprehensibility and maintainability; *(ii)* this code smell seems to be widespread, although little empirical evidence is available; and *(iii)* we know little on the role of this code smell during software evolution.

In this paper, we present the results of an exploratory study whose main goal is to have a better and deeper understanding of dead code by specifically focusing on dead methods. In particular, we aim to improve our body of knowledge on the spread and evolution of dead methods in open-source Java desktop applications[1] given that there is empirical evidence that this kind of code smell hinders the comprehensibility and maintainability of source code in Java desktop applications [39, 40]. Our study is the first one that focuses on how developers deal with dead code during software evolution by leveraging the information available in source code repositories—Romano *et al.* [40] investigated how developers deal with dead code by considering the perspective of developers (*i.e.,* by interviewing them). To gather initial empirical evidence on the spread and evolution of dead methods, we analyzed the evolution of 13 open-source Java desktop applications, whose software projects were hosted on GitHub, for a total of 1,044 commits. To detect dead methods in the commits of these applications, we used *DCF* [37]. It is a prototype of a supporting tool (simply tool, from here onwards) freely available on the web, representing the state of the art for the detection of dead methods in Java desktop applications. The most important take-away results of our study can be summarized as follows: *(i)* dead methods seems to affect open-source Java desktop applications; *(ii)* dead methods generally survive for a long time, in terms of commits, before being "buried" (*i.e.,* removed) or "revived" (*i.e.,* re/used); *(iii)* dead methods are rarely revived; and *(iv)* most dead methods are dead since the creation of the corresponding methods. Although the number of the studied software applications is not so large, we believe that this can be considered acceptable given the exploratory nature of our investigation and its results can justify more resource- and time-demanding research on dead methods.

Paper Structure. In Section 2, we introduce related work and background information. In Section 3, we present the design of our exploratory study, while the obtained results are shown in Section 4. A discussion of the results, including implications from both researcher and practitioner perspectives, is provided in Section 5. In this section, we also highlight possible limitations of our study. We conclude with final remarks in Section 6.

2 RELATED WORK AND BACKGROUND

In this section, we first summarize research about the detection of dead code and then empirical studies on dead code.

2.1 Detecting Dead Code

Dead code, in general, can be detected by using dynamic and static code analyses. As far as the use of dynamic code analysis is concerned, Boomsma *et al.* [14] proposed an approach for detecting dead files in web-based software applications written in PHP. The approach consisted of monitoring the application execution in a given time frame so as to determine the usage of PHP files. A file is deemed dead if it is not used in that time frame. The authors applied their approach on an industrial web-based software application. Thanks to the proposed approach, the developers removed 2,740 dead files. A similar approach was used by Eder *et al.* [17] to detect dead methods in their case study on the modifications affecting dead methods during source code maintenance and evolution tasks. The main drawback of approaches for dead-code detection based on dynamic code analysis is that their detection capability strongly depends on the input data used to exercise the software applications being analysed.

As for dead-code detection approaches based on static code analysis, Romano *et al.* [38] proposed *DUM*. It detects dead methods in Java desktop applications. DUM first builds a graph-based representation of the application—nodes are methods, while directed edges are *caller-callee* relationships. DUM deems nodes reachable from starting nodes (*i.e.,* nodes deemed alive at the outset like those corresponding to main methods) as alive. Any node unreachable from the starting node is deemed dead. DUM (and its prototype of a supporting tool [36]) was compared with *JTombstone* and *CodePro AnalytiX*. DUM outperformed these two baselines for comparison in terms of correctness and accuracy of the detected dead methods, while exhibiting good completeness in detecting dead methods. Romano and Scanniello [37] exploited static code analysis to detect dead methods as well. In particular, the authors proposed *DCF*, a tool based on *RTA*—an algorithm for call graph construction that is known to be fast and to well-approximate virtual method calls [43]—that detect dead methods in Java desktop applications. DCF first identifies alive methods–*i.e.,* reachable nodes from some starting nodes (*e.g.,* representing main methods) in the call graphs built through RTA. Methods that are not alive are considered dead. DCF exploits the RTA implementation available in *Soot* [46]. The authors compared DCF with JTombstone, CodePro AnalytiX, and DUM-Tool. The results of this evaluation show that DCF outperformed the other tools in terms of correctness (average precision equal to 84%) and accuracy (average f-measure equal to 85%) of the detected dead methods. As for completeness (average recall equal to 87%) in detecting dead methods, DCF was comparable to DUM-Tool. Based on our study of the literature on the detection of

[1]With desktop applications, we mean 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.*

dead methods, DCF represents the state of the art; moreover, it is freely available on the web. This is why we used DCF in our study.

The approaches presented above [14, 17, 36–38] takes a refactoring perspective, rather than an optimization perspective, when tackling dead code. In other words, dead code is detected because developers mainly aim to make their source code easier to comprehend and maintain—*i.e.,* developers are not mainly interested in making their source code faster and lighter (as it happens when taking an optimization perspective). Summing up, the refactoring perspective leads to two practical implications. First, developers who want to remove dead code focus on source code—*i.e.,* they are not interested in removing dead code from software applications' dependencies like frameworks and libraries (as done by Obbink *et al.* [29] in the context of JavaScript web-based software applications). Second, the removal of dead code is permanent and affects source code—as opposed to an optimization perspective in which dead code removal can be temporary and can affect intermediate representations of source code only (*e.g.,* bytecode). In our study, we take a refactoring perspective (detecting dead smells in source code), rather than an optimization one. In other words, we are not interested in detecting dead code for optimization reasons.

2.2 Empirical Studies on Dead Code

Dead code is claimed to have a negative effect on both source code comprehensibility and maintainability [18, 26]. On the other hand, only a few empirical studies have been conducted to verify these claims. In particular, Romano *et al.* [39] conducted an experiment with 47 participants. The authors asked a group of participants to comprehend and then maintain Java source code containing dead code, while the other participants were asked to do the same on source code deprived of dead code. The authors found that dead code hinders source code comprehensibility, while they could not demonstrate the negative effect of dead code on source code maintainability. Later, Romano *et al.* [40] replied that experiment three times. The joint results from the baseline and replicated experiments confirm that dead code harms source code comprehensibility. Moreover, they showed that dead code negatively affects source code maintainability when developers deal with unfamiliar source code and developers unconsciously and uselessly modified dead code. Romano *et al.* [40] also interviewed six developers on when and why dead code is introduced and how developers perceive and cope with it. The authors found, for example, that although developers consider dead code harmful, this code smell is consciously introduced to anticipate future changes or consciously left in source code because developers think to reuse it someday. These findings motivated our research (see Section 3.1). Eder *et al.* [17] conducted a case study to investigate the amount of maintenance affecting dead methods in an industrial web-based software application written in .NET. The authors detected dead methods by monitoring the application execution for two years so recording the usage of methods. A method was deemed dead if it was not used in the considered time frame. The authors reported that during source code maintenance and evolution tasks, 7.6% of the modifications affected dead methods so suggesting that developers uselessly spend some effort modifying dead methods. Scanniello [41] used the Kaplan-Meier estimator to analyze the death of methods across different releases of five software applications. The author reported that, on two out of five software applications, the developers avoided introducing dead code or removed dead methods as much as possible. Later, Scanniello [42] presented an preliminary study to understand which software metrics are useful predictors for dead methods. Five out of 13 software metrics were identified as predictors for dead methods. LOC (Lines Of Code) was the best predictor so suggesting that the larger a class, the higher the probability that its methods are dead.

As compared to the research summarized above [17, 39–42], in our study, we quantitatively investigated both spread and evolution of dead code (in particular, dead methods) in the commit histories of 13 open-source Java desktop applications developed in software projects hosted on GitHub. As for the evolution of dead code, we specifically investigated the lifespan of dead methods, whether developers remove dead methods and re/use dead methods, and the introduction of dead methods.

3 STUDY DESIGN

The *goal* of our study is to quantitatively analyze the commit histories of open-source Java desktop applications with the *purpose* of investigating *(i)* the spread of dead methods (*e.g.,* their relative number) and *(ii)* the evolution of such a code smell (*e.g.,* the lifespan of dead methods). The *perspective* is that of both practitioners and researchers interested in dead methods for refactoring reasons. The former might be interested in improving their knowledge on dead code, so that such a code small can be properly managed during source code maintenance and evolution tasks. The latter might be interested in our study results to delineate future research (*e.g.,* defining new approaches to ease the detection of dead methods). The *context* consists of 13 open-source Java desktop applications whose software projects were hosted on GitHub.

3.1 Research Questions

As far as the spread of dead methods is concerned (and in line with the above-mentioned goal), we defined and then investigated the following Research Question (RQ).

RQ1. Are dead methods spread in open-source Java desktop applications?

We defined this RQ to understand whether open-source Java desktop applications are affected by dead methods. We can postulate that the higher the spread of dead methods in desktop applications, the higher the likelihood for developers to bump into dead methods during source code maintenance and evolution tasks. Since dead code harms both source code comprehensibility and maintainability [39, 40], the negative impact of dead methods on both source code comprehensibility and maintainability would be amplified.

As far as the evolution of dead methods is concerned, we formulated and investigated the following RQs.

RQ2. How long do dead methods survive in open-source Java desktop applications?

Based on the qualitative findings by Romano *et al.* [40], developers can consciously introduce dead code or consciously leave dead code in a software application because they think to re/use it later. If a dead method has a long lifespan, its future re/use should be less likely. This is because dead code is not updated when a software application evolves—*i.e.,* dead code was written at a time when the software application was different [27].

RQ3. Do developers bury dead methods in open-source Java desktop applications?

We defined this RQ to study if developers remove dead methods from source code. If developers do not remove dead methods, we can postulate that they are unaware of its presence or they believe that dead methods are not harmful [40]. Since our investigation is purely quantitative, we are not able to discern between these two scenarios. However, the study of this RQ could provide relevant indications on whether developers take care of such a code smell. That is, the results from this RQ could support the qualitative findings by Romano *et al.* [40], so increasing our body of knowledge on this code smell.

RQ4. Do developers revive dead methods in open-source Java desktop applications?

Since developers can consciously introduce dead code or consciously leave dead code in a software application because they think to re/use it later [40], we formulated this RQ to understand whether developers really use dead methods in the future.

RQ5. In open-source Java desktop applications, were dead methods mostly "born" dead or do they mostly become dead later?

With this RQ, we want to complete our study on the evolution of dead methods by specifically focusing on their introduction. Understanding when dead methods are introduced can help us to define a better counteraction. For example, if dead methods are mostly introduced when the corresponding methods are created, then *just-in-time* dead-method detection tools are more advisable. Such a kind of tools should continuously monitor developers while coding and warn them if they create a method that is dead.

3.2 Study Context and Planning

The study focused on open-source Java desktop applications whose software projects were hosted on GitHub. We took into account projects hosted on GitHub because of the overwhelming popularity of this platform and the possibility to access open-source applications. Since dead methods are hard to detect without tool support [47], we used DCF [37]. We opted for this tool for the following reasons: *(i)* it is available on the web, *(ii)* it was empirically validated; and *(iii)* it represents the state of the art for detecting dead methods for refactoring purposes. DCF detects dead methods in Java desktop applications, both GUI-based and CLI-based. This tool requires the bytecode of the application being analyzed, including the bytecode of its dependencies, to detect dead methods. To automate the building process of the studied applications and then apply DCF, we considered applications using *Maven*—a build automation tool for Java applications. In other words, DCF pushed us to investigate on dead methods in Java desktop applications, whose building process was automated by using Maven.

In Table 1, we summarize some information about the 13 studied applications. The choice of these applications was primarily driven by the used dead-method detection tool (as above-mentioned) and then by trying to include heterogeneous applications in terms of: *(i)* size, intended as the number of methods and classes; *(ii)* number of stars, giving an indication of applications' popularity; *(iii)* lifespan, in terms of the number of commits; and *(iv)* application domain. We are aware that these applications are not so large; however, we believe that this can be considered acceptable given the exploratory nature of our study (and the long time that DCF requires to detect dead methods). That is, while this study can contribute increasing our body of knowledge of dead code, its results can justify more resource- and time-demanding research on dead code—*e.g.,* large-scale studies on larger applications randomly sampled from GitHub.

We locally cloned the (Git) software repositories of the studied applications. For each application, we gathered both dead and alive (*i.e.,* not dead) methods from the first to the last commits of the master (*i.e.,* main) branch. In particular, we first compiled each commit by using Maven, and then we ran DCF to gather both dead and alive methods of that commit. If a commit did not compile, we skipped that commit. It is worth mentioning that DCF reports both dead and alive methods that are "internal" to the analyzed applications. That is, the methods of the dependencies (*e.g.,* libraries or frameworks) of the analyzed applications are not reported by DCF since they are not of interest when taking a refactoring perspective. Moreover, when detecting dead methods, we discarded the test directory to avoid DCF from returning methods belonging to test classes (*e.g.,* test methods) as dead.

3.3 Data analysis

In this section, we describe our data analysis arranged by RQ.

RQ1. To answer this RQ, we computed, for each commit of the studied applications, the relative number of dead methods (*%DeadMethods*). We then exploited descriptive statistics (*e.g.,* median, mean, *etc.*) and boxplots to summarize the distribution of the values of the *%DeadMethods* variable. We also exploited line plots to show the values of the *%DeadMethods* variable across the commit history of each application.

RQ2. Differently from RQ1, we are interested in studying the evolution of each dead method (*i.e.,* we followed the evolution of each dead method) along the commit history of each application. For each dead method, we computed its survival time in terms of commits ($SurvivalTime$), namely the interval of consecutive commits, in the master branch, from the first to the last commits in which that method was detected as dead. Such a kind of variable can wrongly estimate survival time when the event of interest (in our case, *dead-method removing/reviving* commit[2]) occurs after the observation time. In studies like ours [16, 44], researchers are forced to analyze a finite commit history although the studied applications continue to evolve with time. In other words, the event of interest can occur outside the analyzed commit history. Therefore, we distinguish between two kinds of data points: *complete* and *censored* [23]. Complete data points are, in our case, dead methods for which we know both their dead-method introducing and removing/reviving commits.[3] On the other hand, censored data

[2]Given a dead method, its dead-method removing commit is the one in which the dead method is removed from the source code. On the other hand, the dead-method reviving commit of a dead method is the one in which the dead method is made alive. In both case, the dead method is no longer present in the source code.

[3]Given a dead method, its dead-method introducing commit is the one in which the dead method is introduced into the source code (*i.e.,* when a method is created already dead or when a method becomes dead after being alive in the previous commit).

Table 1: Some information about the studied applications.

Application	Description	# Stars	Last Analyzed Commit	# Analyzed Commits	# Methods	# Class
4HWC Autonomous Car [1]	A simulator that allows verifying the movements of an autonomous car	1	5a5c472	102	118-137	34-37
graphics-tablet [7]	A drawing application	0	8a3df4c	35	226-697	41-81
BankApplication [3]	An application that provides support for some banking operations	72	6856256	102	376-537	79-121
bitbox [4]	A utility tool for bit operations	1	af2af8b	15	470-548	55-64
Density Converter [5]	A tool that helps converting single or batches of images to specific formats and density versions	225	e70dcad	162	51-384	14-71
Deobfuscator-GUI [6]	It provides a GUI for a popular Java deobfuscator based on CLI.	112	deb003e	29	124-221	27-47
JavaANPR [8]	An application to automatically recognize number plates from vehicle images	125	eff9acf	256	425-786	61-92
javaman [9]	A Java implementation of the popular *Bomberman* game	0	7a03c36	58	59-230	12-38
JDM [10]	An application to manager the download of files	0	a435b4d	25	5-82	2-16
JPass [11]	An application to manage passwords	81	c6b13af	134	305-366	71-82
MBot [12]	An application to record and automate mouse and keyboard events	0	ff07dac	21	2-56	1-16
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	323-346	34-37
SMV APP [13]	An application to that provides support to a car repair shop	1	4c17370	67	5-408	2-96

points are dead methods for which we know their dead-method introducing commits but not their dead-method removing/reviving commits. To take into account both complete and censored data points, when estimating how long dead methods survive, we leveraged *survival analysis* [23]. In particular, for each application, we built the *Kaplan-Meier survival curve* [24], which graphically depicts the survival probability of dead methods at any point of time. From the Kaplan-Meier survival curve, we then computed the *median survival time*, namely the time for which the survival probability is equal to 0.5 [20]. The median survival time is used, in studies like, ours to estimate the survival time by taking into account both complete and censored data points [33]. In our case, the median survival time estimates how long dead methods survive in terms of commits. The $SurvivalTime$ values, along with the censoring information (*i.e.,* whether a given method was censored or not), were used to build the Kaplan–Meier survival curves.

RQ3. Similarly to the previous RQ, we followed the evolution of dead methods along the commit history of each application. To answer RQ3, we computed the relative number of removed dead methods ($\%RemovedDeadMethods$) for each application and we also summarized the distribution of the $\%RemovedDeadMethods$ values by using descriptive statistics. To compute $\%RemovedDeadMethods$, we used the information about the dead-method removing commit of each dead method, and not the one about the dead-method reviving commit. In other words, if a dead method becomes alive during the commit history of an application, we did not consider that dead method as removed—such a dead method is revived and it is the subject of the next RQ.

RQ4. We again followed the evolution of dead methods along the commit history of each application to answer this RQ. In particular, we computed the relative number of revived dead methods ($\%RevivedDeadMethods$) for each application and we also summarized the distribution of the $\%RevivedDeadMethods$ values through descriptive statistics. To compute $\%RevivedDeadMethods$, we used the information about the dead-method reviving commit.

RQ5. To answer this RQ, we followed the evolution of dead methods one last time. In particular, we counted the relative numbers of dead methods born dead ($\%DeadBornMethods$) and became dead ($\%DeadBecameMethods$) for each application. Dead-born methods are the ones that were dead since their creation, while dead-became methods are the ones that were alive before becoming dead. Given an application, the sum of the values of $\%DeadBornMethods$ and $\%DeadBecameMethods$ is 100%. We used both variables to provide an indication of those methods that were born dead with respect to those methods that become dead later in the commit histories. We also exploited descriptive statistics to summarize the distribution of the values for *%DeadBornMethods* and *%DeadBecameMethods*.

To perform the data analysis, we exploited *R*. Both analysis script and raw data are available in our replication package [35].

4 RESULTS

In this section, we present the results of our study arranged by RQ.

4.1 RQ1. Are Dead Methods Spread in Open-source Java Desktop Applications?

In Table 2, we report some descriptive statistics—*i.e.,* mean, Standard Deviation (SD), minimum (min), median, and maximum (max)—to summarize the distributions of the relative number of dead methods (*%DeadMethods*) for the studied applications. We can observe that dead methods are quite widespread in these applications. The application with the highest average number of dead methods is bitbox (mean = 36.749%). On the other hand, the application with the lowest average number of dead methods is JDM (mean = 3.912%). Besides JDM, the minimum values range in between 3.571% and 35.593%. This suggests that, in any commit, dead methods are present so confirming that this code smell is widespread. We can also note that, in the case of SMV APP, the maximum value of *%DeadMethods* (96.296%) is very close to 100%.

In Figure 1, we show the distributions of the *%DeadMethods* values by means of boxplots. Some boxplots present outliers. However, when this happens, the boxes are flattened and look like lines so suggesting a low variability in the *%DeadMethods* values. Only three applications out of 13 had a high variability in the values of the *%DeadMethods* variable. These applications are: Deobfuscator-GUI, javaman, and SMV APP. The boxes of Deobfuscator-GUI and SMV APP are also skewed.

The line plots shown in Figure 2 allow better comprehending how the values of *%DeadMethods* change across the commit history of each application. For most applications, the *%DeadMethods* values remain quite constant across the commit histories. The exceptions are Deobfuscator-GUI, javaman, MBot, and SMV APP. For these applications, we can note that the values for *%DeadMethods* decreases across the commits. The study of the next RQs can help us to better understand such a trend.

Answer to RQ1: Although we observed different trends of the relative number of dead methods in the commit histories of the studied applications, dead methods seem to be widespread. That is, open-source Java desktop applications seem to be affected by dead methods.

4.2 RQ2. How Long Do Dead Methods Survive in Open-source Java Desktop Applications?

In Table 3, we show the results of the survival analysis applied to (distinct) dead methods. It is worth recalling that in this RQ (as well as in the RQs that follow) we are interested in studying each dead method by following its evolution in the commit history. That is to say that, if a given dead method m is present in more than one commit, we consider it only once. For each application in Table 3, we report the number of (distinct) dead methods across the commit history (#); the number of events (# Events)—*i.e.,* dead-method removing/reviving commits, in our case—; and the values of the median survival time, estimated from the Kaplan-Mayer curves, along with the 95% confidence interval (95%CI).

Table 2: Descriptive statistics for *%DeadMethods*.

Application	Mean	SD	Min	Median	Max
4HWC Autonomous Car	12.46	1.279	10.156	12.977	13.74
8_TheWeather	25.508	0.61	24.855	25.561	26.316
BankApplication	13.055	3.722	8.632	11.702	27.249
bitbox	36.749	1.283	35.593	35.782	38.511
Density Converter	23.532	2.242	11.765	23.177	28.779
Deobfuscator-GUI	36.345	21.61	11.111	49.231	64.706
graphics-tablet	19.749	6.087	6.25	19.058	26.917
JavaANPR	14.202	0.513	12.341	14.052	15.699
javaman	26.782	13.292	8.661	23.748	62.712
JDM	3.912	8.915	0	0	31.915
JPass	7.633	2.748	4.985	6.23	13.661
MBot	9.459	13.588	3.571	5.882	50
SMV APP	29.879	26.764	7.692	14.748	96.296

As shown in Table 3, we could estimate the median survival time for any application except for 8_TheWeather and bitbox. For these two applications, we could not compute the median survival time because the number of events was too small as compared to the number of (distinct) dead methods—*i.e.,* too many (distinct) dead methods were censored. In particular, only 1 out of 87 dead methods were removed/revived in 8_TheWeather during the observation time. As far as bitbox is concerned, only 15 out of 210 dead methods were removed/revived during the observation time. We can notice that, for the greater part of the other applications, dead methods tend to survive for many commits. In particular, for eight applications, the median survival time is greater than 10 commit, reaching a maximum of 83 commits for JPass. Only on three applications, the median survival time is lower than 10 commits.

Answer to RQ2: Dead methods in open-source Java desktop applications generally survive for a long time, in terms of commits, before being buried or revived.

4.3 RQ3. Do Developers Bury Dead Methods in Open-source Java Desktop Applications?

In Table 4, we show, for each application, the percentage of (distinct) dead methods that are removed (*%RemovedDeadMethods*), revived (*%RevivedDeadMethods*), and censored (*%CensoredDeadMethods*). While *%RevivedDeadMethods* is exploited for the study of the next RQ, *%CensoredDeadMethods* is reported to let the reader have a full picture of the study of both RQ3 and RQ4. It is worth mentioning that the sum of the values for *%RemovedDeadMethods*, *%RevivedDeadMethods*, and *%CensoredDeadMethods* gives 100% for each application.

For five out of 13 applications, it seems that dead methods are not removed at all: the values for *%RemovedDeadMethods* range in between 0% (JDM) and 22.078% (JPass). For the other eight applications, we can observe that the developers seem to pay attention to the removal of dead methods. The values for

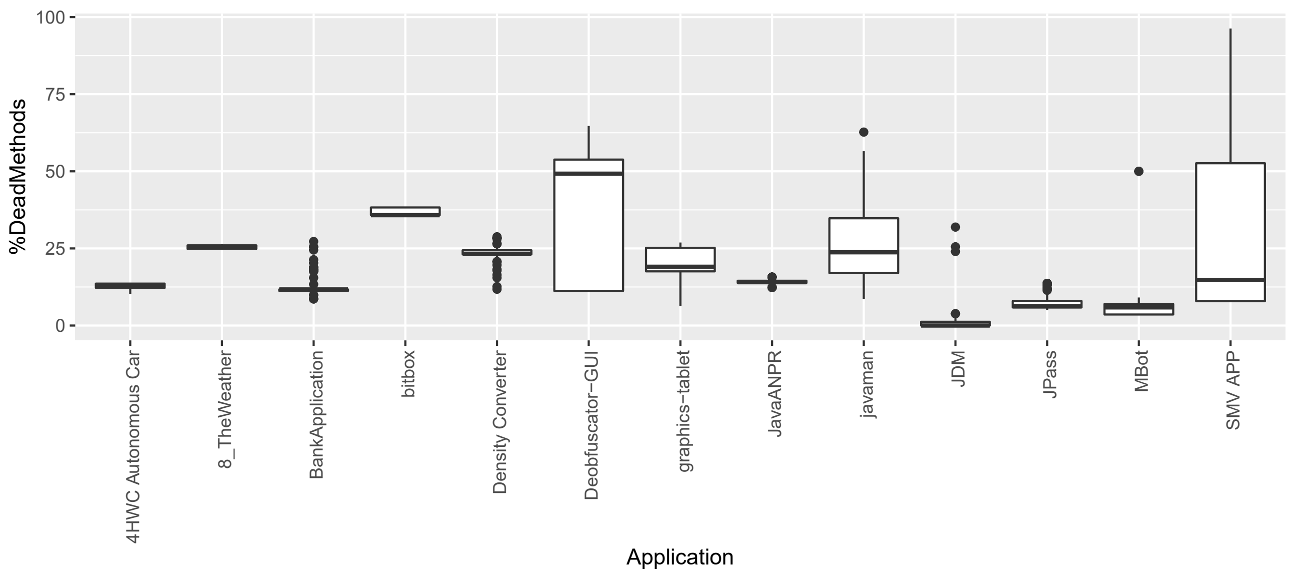

Figure 1: Boxplots for *%DeadMethods*

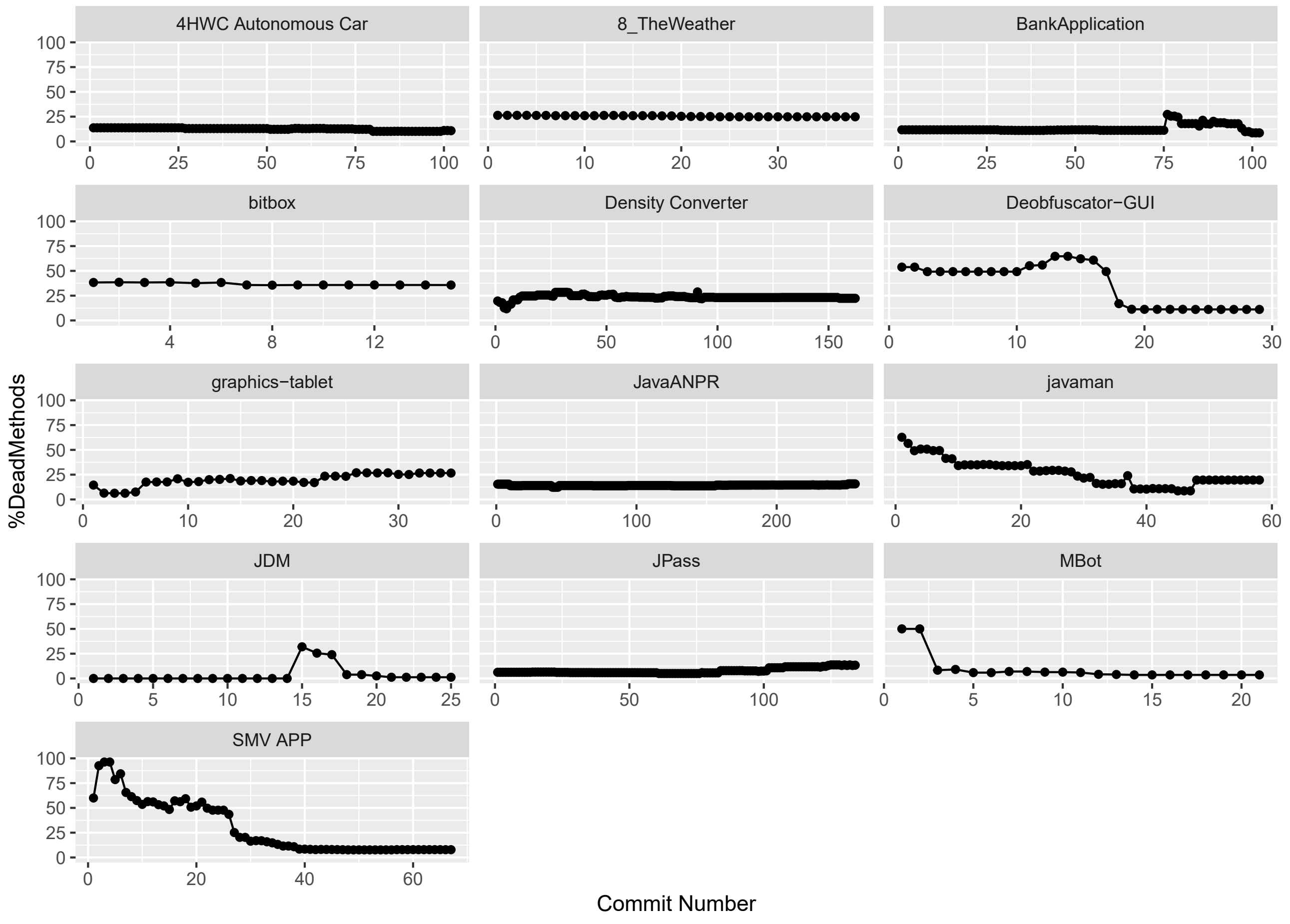

Figure 2: Line plots for *%DeadMethods*

Table 3: Results of the survival analysis.

Application	#	# Events	Median Survival Time	
			Value	95%CI
4HWC Autonomous Car	27	14	59	[26, −]
8_TheWeather	87	1	−	[−, −]
BankApplication	270	229	9	[8, 13]
bitbox	210	15	−	[−, −]
Density Converter	288	203	21	[15, 24]
Deobfuscator-GUI	156	142	14	[12, 17]
graphics-tablet	407	227	17	[7, 18]
JavaANPR	237	164	5	[5, 88]
javaman	142	97	18	[9, 22]
JDM	17	16	3	[3, 3]
JPass	77	28	83	[60, −]
MBot	5	3	11	[1, −]
SMV APP	183	151	11	[7, 14]

%RemovedDeadMethods range in between 41.549% (javaman) and 68.354% (JavaANPR). For example, we can observe that, in javaman, the developers progressively removed dead methods, as Figure 2 and the %RemovedDeadMethods value together suggest. As for Deobfuscator-GUI and MBot, the developers removed dead methods in a shorter number of commits (see Table 4 and Figure 2).

In Table 4, we also show the values of mean, SD, and median of *%RemovedDeadMethods* when considering the applications together. The average value for *%RemovedDeadMethods* of the studied applications is 38.093%, while the median value is 48.148%. These values confirm that, in general, the developers paid attention to the removal of dead methods. The SD value is high (25.759) so remarking again that, across the applications, the developers took care of dead method removal in a different fashion.

Answer to RQ3: In some of the studied open-source Java desktop applications, the developers paid attention to dead methods by giving them a decent burial, while it is not the case for few other applications.

4.4 RQ4. Do Developers Revive Dead Methods in Open-source Java Desktop Applications?

The values for *%RevivedDeadMethods* suggest that developers rarely revive dead methods (see Table 4). In particular, in eight out of 13 applications the values for *%RevivedDeadMethods* are quite small, ranging from 0% to 14.286%. Only for JDM and SMV APP, the values of this variable are higher than 50%. As for JDM, we can note a peak in the middle of the commit history shown in Figure 2. This allows postulating that the developers consciously introduced some dead methods to use them in the subsequent commits. A similar postulation can be done for SMV APP, although the peak of dead methods arises at the beginning of the commit history (see Figure 2).

Answer to RQ4: Developers rarely revive dead methods in open-source Java desktop applications. Only in a few occasions, developers seem to introduce this code smell as a means of anticipating changes and re/using code.

4.5 RQ5. In Open-source Java Desktop Applications, Were Dead Methods Mostly Born Dead or Do They Mostly Become Dead Later?

In Table 5, we show, for any application, the percentage of dead methods that were born dead (*%DeadBornMethods*) or that become dead later (*%DeadBecameMethods*). In this table, we also report some descriptive statistics. The comparison of the values of *%DeadBornMethods* and *%DeadBecameMethods* clearly indicates that dead methods mostly born dead rather than become dead later—the *%DeadBornMethods* values range in between 69.259% and 100% (while the *%DeadBecameMethods* values range in between 0% and 30.741%). We can also observe high *%DeadBornMethods* values for both median (95.139%) and mean (90.789%) and a low value for SD (10.673). These descriptive statistics confirm that most dead methods were born dead and this outcome holds for any application.

Answer to RQ5: Most dead methods are introduced when the corresponding methods are added to the source code of open-source Java desktop applications.

5 DISCUSSION

In this section, we discuss the results of our empirical study and also delineate possible implications from the perspectives of both practitioners and researchers. We conclude this section by discussing threats that might affect the validity of our results.

5.1 Overall Discussion and Implications

Dead methods, and thus dead code, seem to be widespread in the studied open-source Java desktop applications. This finding complements the ones of past work that reported large amount of dead code in PHP and C++ industrial web-based applications [14, 17]. Therefore, our finding and past ones seem to indicate that dead code affects software applications regardless of the programming language (Java vs. C++ vs. PHP), kind of application (desktop vs. web-based), kind of license (open-source vs. proprietary), and application domain (*e.g.,* from conversion of images to weather forecast in our study, and from insurance to customers relationship management in the studies by Eder *et al.* [17] and Boomsma *et al.* [14]). **Practitioners** should be interested in this outcome. In particular, they should expect to bump into dead code whatever the software application is, and then they should take care of dead code (by removing it from the source code or by avoiding introducing this code smell) since it is harmful [39, 40]. Nevertheless, we believe that dead code deserves further attention from the SE community since, for example, there is a lack of empirical evidence on the spread of dead code in mobile applications. **Researchers** could be interested in bridging this gap.

We observed a variability in the amount of dead methods across the studied applications (on average, from 3.912% to 36.749%). This finding supports the one by Scanniello [41], who observed that developers avoided introducing dead methods or removed dead methods as much as possible on two out of the five open-source applications studied. The variability in the amount of dead methods could be due to factors internal to software projects—*e.g.,* LOC,

Table 4: Results regarding the relative number of removed and revived dead methods. The relative number of censored dead methods is also shown for completeness.

Application	$\%RemovedDeadMethods$	$\%RevivedDeadMethods$	$\%CensoredDeadMethods$
4HWC Autonomous Car	48.148	3.704	48.148
8_TheWeather	1.149	0	98.851
BankApplication	63.333	21.481	15.185
bitbox	3.81	3.333	92.857
Density Converter	57.639	12.847	29.514
Deobfuscator-GUI	62.821	28.205	8.974
graphics-tablet	49.386	6.388	44.226
JavaANPR	68.354	0.844	30.802
javaman	41.549	26.761	31.69
JDM	0	94.118	5.882
JPass	22.078	14.286	63.636
MBot	60	0	40
SMV APP	16.94	65.574	17.486
Mean	38.093	21.349	40.558
SD	25.759	28.345	29.452
Median	48.148	12.847	31.69

Table 5: Results regarding the relative number of dead methods born dead and became dead.

Application	$\%DeadBornMethods$	$\%DeadBecameMethods$
4HWC Autonomous Car	92.593	7.407
8_TheWeather	100	0
BankApplication	69.259	30.741
bitbox	98.095	1.905
Density Converter	95.139	4.861
Deobfuscator-GUI	98.718	1.282
graphics-tablet	88.698	11.302
JavaANPR	99.578	0.422
javaman	84.507	15.493
JDM	100	0
JPass	76.623	23.377
MBot	100	0
SMV APP	77.049	22.951
Mean	90.789	9.211
SD	10.673	10.673
Median	95.139	4.861

as suggested by the preliminary investigation by Scanniello [42]. Therefore, we foster **researchers** to study which factors are useful predictors for the introduction of dead methods in open-source applications. **Researchers** could be also interested in studying if these results also hold for proprietary applications. Our findings justify future fork on this matter. **Practitioners** could be interested in exploiting these predictors to understand whether and when applying refactorings on source code to remove dead methods. That is to say, these predictors could be a useful support in the decision process concerning the removal of dead methods.

For the software applications in which we observed that dead methods are more spread, we can postulate that the likelihood for developers to bump into dead methods, during source code maintenance and evolution tasks, is greater. Therefore, the likelihood for developers to experience the detrimental effect of this code smell, in terms of source code comprehensibility and maintainability [39, 40], is greater as well. **Practitioners** should thus keep under control the spread of dead methods. Therefore, we recommend them to periodically plan code reviews (possibly with a tool support) to detect and remove dead methods, as well as prevent the introduction of dead methods. **Researchers** could be interested in verifying our postulation by conducting, for example, experiments where the participants are asked to comprehend and/or maintain the source code of the same software application but with different spread levels of dead methods.

We found that most dead methods were born dead, rather than become dead later. That is, most dead methods are dead since the creation of the corresponding methods. This result is consistent with previous findings on code and test smells [44, 45], and it contradicts the common wisdom for which code smells are due to side effects of software evolution [32]. **Practitioners** should pay attention to the design of their application by trying to avoid as much as possible the introduction of dead methods. Accordingly, future dead-method detection tools should take into account that dead methods, in most cases, start affecting software applications since the creation of methods. **Researchers** should devote their work to just-in-time tools capable of detecting dead methods in real time—*i.e.,* while developers are coding. **Practitioners** could clearly take advantage of such a kind of detection tools.

Past qualitative research (*i.e.,* interviews with practitioners) suggests that developers can consciously introduce dead code or consciously leave dead code in a software application because they think to use it later [40]. However, we found that developers rarely revive dead methods in open-source Java desktop applications. We can thus make Martin's recommendation (for **practitioners**) ours: "When you find dead code, do the right thing. Give it a decent burial. Delete it from the system." If developers think they can reuse dead code someday, version control systems should help developers to find removed dead code. Following this recommendation should bring advantages when developers have to comprehend and maintain source code. This is because existing evidence on dead code shows that this smell hinders the comprehensibility and maintainability of unfamiliar source code [39, 40], and developers uselessly spend some effort to modify dead code [17, 40].

Dead methods generally survive for a long time, in terms of commits. Accordingly, the future re/use of dead methods should be unlikely because dead methods are not updated with the rest of software applications [27]. Based on our and past outcomes, we can recommend again **practitioners** to avoid introducing dead methods or to remove them whenever they can.

The survey results by Yamashita and Moonen [48] indicate that dead-code detection was the 10th most desired feature (out of 29) for smell analysis tools so suggesting that dead-code removal matters for developers. In some of the studied applications, we observed that developers pay attention to dead methods by removing them. This seems to imply that the removal of dead methods matters to the developers of these applications so confirming the results by Yamashita and Moonen [48]. This outcome is clearly relevant for **researchers** since it seems that dead code is relevant from a practical point of view. In some other applications, we found that developers did not remove dead methods. Possible reasons are: *(i)* some developers are unaware of the presence of dead methods and/or *(ii)* some developers are conscious of the presence of this smell but they think that dead methods are harmless and/or believe to re/use dead methods in the future. Besides providing tools for dead-method detection (as mentioned above), **researchers** should inform developers about the negative effects of dead methods and their uselessness.

Summing up, our study has the merit of improving our body of knowledge on dead methods (and thus dead code). In particular, we gathered evidence on the spread and evolution of dead methods in open-source Java desktop applications and also complemented/supported the findings of past research on this code smell. However, we cannot claim that our outcomes are conclusive. We believe that our outcomes can justify **researchers** to conduct more resource- and time-demanding studies on dead methods like, for example, large-scale studies on applications larger than ours and randomly sampled from GitHub.

5.2 Threats to Validity

In the following of this section, we discuss threats to external, conclusion, and construct validity that might affect our results.

5.2.1 External Validity. When datasets are small and empirical research relies on them, generalizing the outcomes poses a threat to external validity. This is why caution is needed when generalizing our outcomes. Both applications considered in our study and their number might affect this kind of validity—*e.g.,* the studied applications could not be representative of the universe of open-source Java desktop applications. Although we gather initial empirical evidence on the prevalence and evolution of dead methods, we advise further studies on a large number of open-source Java desktop applications, possibly randomly sampled from GitHub. The outcomes of these studies could allow increasing the generalizability of our outcomes.

5.2.2 Conclusion Validity. Although the Kaplan-Mayer method is one of the best options for survival analysis [45], its use might affect the results concerning RQ2.

5.2.3 Construct Validity. When a commit did not compile, we skipped that commit. Although we relied on a popular build automation tool (*i.e.,* Maven) to automatically compile the applications, the lack of ability to compile open-source applications at a commit level is an inherent limitation to any study similar to ours.

When studying the evolution of dead methods, we identified each dead method by using its signature and the fully qualified name of the belonging class. This might affect the results concerning the evolution of dead methods.

To detect dead methods, we used DCF. We opted for this tool because it is freely available on the web and its validity was empirically assessed by comparing it with other dead code detection tools on a gold standard [37]. The results of this comparison show that DCF outperformed the other tools in terms of correctness and accuracy of the detected dead methods, while exhibiting high completeness in detecting dead methods (see Section 2.1). Although DCF represents the state of the art for dead method detection in Java desktop applications, its use might affect the study results.

Finally, the metrics we used to answer our RQs might pose a threat to validity. However, there is no accepted metric to quantitatively assess the spread and evolution of dead methods.

6 CONCLUSION

In this paper, we present the results of an exploratory study on dead methods in open-source Java desktop applications. We quantitatively analyze the commit histories of 13 open-source Java desktop applications, whose software projects were hosted on GitHub, for a total of 1,044 commits. We studied how spread dead methods are and the evolution of this code smell. The most important take-away results of our study can be summarized as follows: *(i)* dead methods seems to affect open-source Java desktop applications; *(ii)* dead methods generally survive for a long time, in terms of commits, before being buried or revived; *(iii)* dead methods are rarely revived; and *(iv)* most dead methods are dead since the creation of the corresponding methods. We can conclude that developers should carefully handle dead methods in open-source Java desktop applications since this code smell is harmful, widespread, rarely revived, and survives for a long time in the source code. Although caution is needed on the obtained results, due to the exploratory nature of our research, they allowed us to have a better understanding of the presence and evolution of dead methods in open-source Java desktop applications so increasing our knowledge of this code smell. Our results also justify more resource- and time-demanding research on dead methods. For example, future research could focus on large-scale studies on applications larger than ours and randomly sampled from GitHub. We also suggest future research on dead methods on different kinds of applications (*e.g.,* web-based and mobile) and written in languages different from Java.

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] 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.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code* (1st ed.). Addison-Wesley.
[20] Manish Kumar Goel, Pardeep Khanna, and Jugal Kishore. 2010. Understanding survival analysis: Kaplan-Meier estimate. *International journal of Ayurveda research* 1 (2010), 274–278.
[21] 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.
[22] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *Proceedings of International Conference on Program Comprehension*. 1–10.
[23] Rupert G. Miller Jr. 2011. *Survival Analysis* (2nd ed.). John Wiley and Sons.
[24] E. L. Kaplan and Paul Meier. 1958. Nonparametric Estimation from Incomplete Observations. *Journal of the American Statistical Association* 53, 282 (May 1958), 457–481.
[25] 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.
[26] 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.
[27] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1st ed.). Prentice Hall.
[28] 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.
[29] 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.
[30] 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.
[31] 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.
[32] David Lorge Parnas. 1994. Software Aging. In *Proceedings of International Conference on Software Engineering*. IEEE, 279–287.
[33] Jason T. Rich, J. Gail Neely, Randal C. Paniello, Courtney C. J. Voelker, Brian Nussenbaum, and Eric W. Wang. 2010. A practical guide to understanding Kaplan-Meier curves. *Otolaryngology–head and neck surgery : official journal of American Academy of Otolaryngology-Head and Neck Surgery* 143 (2010), 331–336.
[34] Simone Romano. 2018. Dead Code. In *Proceedings of International Conference on Software Maintenance and Evolution*. 737–742.
[35] Simone Romano. 2021. An Exploratory Study on Dead Methods in Open-source Java Desktop Applications: Replication Package. https://figshare.com/articles/online_resource/An_Exploratory_Study_on_Dead_Methods_in_Open-source_Java_Desktop_Applications_Replication_Package/14979939/1.
[36] Simone Romano and Giuseppe Scanniello. 2015. DUM-Tool. In *Proceedings of International Conference on Software Maintenance and Evolution*. IEEE, 339–341.
[37] 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.
[38] 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.
[39] 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.
[40] 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.
[41] Giuseppe Scanniello. 2011. Source code survival with the Kaplan Meier. In *Proceedings of International Conference on Software Maintenance*. 524–527.
[42] 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.
[43] 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.
[44] 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.
[45] 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.
[46] 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.
[47] William C. Wake. 2003. *Refactoring Workbook* (1st ed.). Addison-Wesley.
[48] 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.