Empirical Software Engineering (2023) 28:64
https://doi.org/10.1007/s10664-023-10303-0

On the spread and evolution of dead methods in Java desktop applications: an exploratory study

Danilo Caivano[1] · Pietro Cassieri[2] · Simone Romano[3] · Giuseppe Scanniello[3]

Accepted: 6 February 2023 / Published online: 14 April 2023

Abstract

Background. Dead code is a code smell. It can refer to code blocks, fields, methods, etc. that are unused and/or unreachable—e.g., if a method is unused and/or unreachable, it is a dead method. Past research has shown that the presence of dead code in source code harms its comprehensibility and maintainability. Nevertheless, there is still little empirical evidence on the spread of this code smell in the source code of commercial and open-source software applications.

Aims. Our goal is to gather, through an exploratory study, empirical evidence on the spread and evolution of dead methods in open-source Java desktop applications.

Method. We quantitatively analyzed the commit histories of 23 open-source Java desktop applications, whose software projects were hosted on *GitHub*. To investigate the spread and evolution of dead methods, we focused on dead methods detected at a commit level. The total number of analyzed commits in our study is 1,587. The perspective of our exploratory study is that of both practitioners and researchers.

Results. We can summarize the most important take-away results as follows: *(i)* dead methods affect open-source Java desktop applications; *(ii)* dead methods generally survive for a long time before being "buried" or "revived;" *(iii)* dead methods that are then revived tend to survive less, as compared to dead methods that are then buried; *(iv)* dead methods are rarely revived; and *(v)* most dead methods are stillborn, rather than becoming dead later. Given the exploratory nature of our study, we believe that its results will help researchers to conduct more resource- and time-demanding research on dead methods and, in general, on dead code.

Conclusions. We can conclude that developers should carefully handle dead code (and thus dead methods) since it is harmful, widespread, rarely revived, and survives for a long time in software applications.

Keywords Code smell · Dead code · Unused code · Exploratory study · Java desktop applications · Open-source · GitHub

Communicated by: Maria Teresa Baldassarre, Markos Kalinowski

This article belongs to the Topical Collection: *Special Issue on International Symposium on Empirical Software Engineering and Measurement (ESEM)*

✉ Simone Romano
siromano@unisa.it

Extended author information available on the last page of the article.

1 Introduction

In the last two decades, the Software Engineering (SE) community has displayed an increasing interest in *code smells* (also known as *bad smells in code*) because they are indicators of potential problems in source code (Wake 2003). A number of catalogs of code smells have been proposed in the literature (e.g., (Fowler 1999; Wake 2003)). These catalogs characterize and list different code smells, one of them is *dead code* (also known as *unused code* (Wake 2003), *unreachable code* (Fard and Mesbah 2013), or *lava flow* (Brown et al. 1998)). Dead code is defined as unnecessary source code since it is unused and/or unreachable (i.e., never executed) (Haas et al. 2020; Mäntylä et al. 2003; Wake 2003). This code smell can refer to code blocks, fields, methods, etc. If a method is unused and/or unreachable, we can refer to dead code as a *dead method*.

Different aspects related to code smells have been investigated in the SE research field. Just to name a few, researchers have investigated the impact of code smells on source code comprehensibility and maintainability (Hermans and Aivaloglou 2016), the fault- and change-proneness of smelly classes (Khomh et al. 2009; Palomba et al. 2018), the spread and evolution of code smells (Chatzigeorgiou and Manakos 2014; Tufano et al. 2017), and the knowledge and perception of developers about code smells (Palomba et al. 2014; Yamashita and Moonen 2013). The interest of the SE community in code smells has been also manifested through the implementation of supporting tools for code-smell detection (Moha et al. 2010).

The presence of dead code is claimed to harm source code comprehensibility and maintainability (Fard and Mesbah 2013; Mäntylä et al. 2003). These claims are well-founded since (Romano 2018; Romano et al. 2016; 2020) have gathered evidence, through a series of experiments, that the presence of dead code significantly hinders the comprehensibility of unfamiliar source code and also negatively affects its maintainability. However, little empirical evidence has been gathered on the spread of dead code in the source code of commercial and open-source software applications —this evidence does not regard Java code. For example, (Boomsma et al. 2012) reported that, in a subsystem of a commercial web-based software application written in PHP, the developers removed 30% of the subsystem's files because these files were actually dead. On the other hand, (Eder et al. 2012) observed that, in a commercial web-based software application written in .NET, 25% of methods were dead.

There is little empirical evidence on the role of dead code during the evolution of software applications. In particular, (Eder et al. 2012) reported that, during the maintenance and evolution of a commercial (.NET) web-based software application, 7.6% of the modifications affected dead methods. Moreover, 48% of these modifications were unnecessary (e.g., because dead methods were removed later).

Summing up, dead code *(i)* harms source code comprehensibility and maintainability; *(ii)* it seems to be widespread in the source code of software applications, despite little empirical evidence is available; and *(iii)* the role of dead code during software evolution has not been adequately studied.

In this paper, we present the results of an exploratory study whose main goal is to improve the body of knowledge on the spread and evolution of dead code—more specifically, we focus on dead methods—in open-source Java desktop applications[1] given that empirical evidence shows that this code smell hinders source code comprehensibility and maintainability (Romano et al. 2016; 2020). Our study is the first one to investigate how developers deal with dead code during software evolution by leveraging the information available in open-source software repositories—as opposed to Romano et al. (Romano et al. 2020) that investigated how developers deal with dead code by considering the perspective of developers (i.e., by interviewing them). To gather evidence on the spread and evolution of dead methods, we analyzed the commit histories of 23 open-source Java desktop applications. The software projects of these applications were hosted on *GitHub*. We gathered information on dead methods at a commit level, for a total of 1,587 commits. To detect dead methods at a commit level, we used *DCF* (Romano and Scanniello 2018). It is a prototype of a supporting tool (simply tool, from here onwards) that is freely available on the web and represents the state of the art for the detection of dead methods in Java desktop applications.

We can summarize the most important take-away results of our study as follows:

- dead methods affect open-source Java desktop applications;
- dead methods generally survive for a long time, in terms of commits, before being "buried" (i.e., removed) or "revived" (i.e., used);
- dead methods that are then revived tend to survive less, as compared to dead methods that are then buried;
- dead methods are rarely revived;
- the majority of dead methods are dead since their creation.

This paper extends our past one (i.e., (Caivano et al. 2021)). Compared to this paper, we provide the following extensions:

- We add ten open-source Java desktop applications to our past dataset. The data analyses now take into account the commit histories of 23 Java desktop applications with software project hosted on GitHub. This makes the study presented in this paper the largest one on dead code in terms of studied software applications.
- We extend the presentation and discussion of the results in light of the new data. We also highlight whether, or not, there are different patterns when passing from the old dataset (comprising 13 software applications) to the new dataset (comprising 23 software applications).
- We provide further analyses.
- We update the discussion on the related work.

Paper structure In Section 2, we introduce background information as well as related work. 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. Section 6 concludes the paper.

[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.

2 Background and Related Work

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

2.1 Dead Code Detection

Given a given application, we can detect dead code by using dynamic or static code analysis. In the remainder on this section, we first focus on the detection of dead code by using dynamic code analysis and than by using static code analysis. We conclude discussing the differences between the refactoring and optimization perspectives when detecting dead code.

2.1.1 Dynamic Code Analysis

As far as the use of dynamic code analysis is concerned, (Boomsma et al. 2012) proposed an approach for detecting dead files in web-based software applications written in PHP. To determine if a PHP file is dead or not, the approach monitors the execution of a target software application in a given time frame. A PHP file is deemed dead if it is not used in that time frame. Boomsma et al. then applied this approach in a case study consisting of a commercial web-based software application written in PHP. Thanks to the proposed approach, the developers were able to remove 2,740 dead files in one of the subsystems of the studied software application. Similarly, (Eder et al. 2012) exploited dynamic code analysis to detect dead methods in their case study, which consists of a commercial web-based application written in .NET, to investigate modifications to dead methods. In particular, the authors monitored the execution of methods in a given time frame—methods not executed in this time frame were considered dead. Fard and Mesbah (2013) presented *JSNOSE*, a code-smell detection tool for web-based applications with client-side written in JavaScript. JSNOSE leverages dynamic and static analyses to detect 13 smells, including dead code, in client-side code. To detect dead code (more specifically, dead statements) JSNOSE counts either the execution of statements (so using dynamic code analyses) or reachability of statements (so using static code analysis). 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 target software applications.

2.1.2 Static Code Analysis

As for dead-code detection approaches based on static code analysis, (Romano et al. 2016) proposed *DUM*. It was conceived to detect dead methods in Java desktop applications. DUM first builds a graph-based representation of the target software application, where nodes are methods and directed edges are *caller-callee* relationships. The approach by Romano et al. deems nodes reachable from starting nodes (e.g., those corresponding to main methods) as alive. Any unreachable node from the starting nodes is dead. DUM was implemented in a tool named *DUM-Tool* (Romano and Scanniello 2015). To assess the validity of their approach, (Romano et al. 2016) compared DUM-Tool with two baselines: *JTombstone* and *CodePro AnalytiX*. DUM-Tool outperformed these two baselines in terms of correctness and accuracy of the detected dead methods, while exhibiting good completeness in detecting dead methods.

Similarly, (Romano and Scanniello 2018) exploited static code analysis to detect dead code in Java desktop applications. In particular, the authors proposed *DCF*, a tool based on the *Rapid Type Analysis* (RTA) algorithm to detect dead methods. RTA is an algorithm for call graph construction that is known to be fast and to well-approximate virtual method calls (Tip and Palsberg 2000). DCF first identifies alive methods. These methods are reachable nodes from some starting nodes in the call graphs built through the RTA algorithm. More specifically, DCF identifies the following starting nodes: *(i)* main methods "internal" to the target application and *(ii)* methods used to customize the serialization/deserialization process of objects (these methods are thought to be invoked through reflection). When building the call graphs, DCF is capable of inferring some kinds of implicit calls (e.g., calls to `run()` methods of threads, initializes, etc.). After identifying alive methods, DCF marks those methods that are not alive as dead. To assess the validity of DCF, as well as the underlying approach, Romano and Scanniello compared DCF with JTombstone, CodePro AnalytiX, and DUM-Tool. The results of this comparison indicated that DCF outperformed the other tools in terms of: correctness of detected dead methods (average precision equal to 84%) and accuracy of detected dead methods (average f-measure equal to 85%). 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 approaches/tools to detect dead methods, DCF represents the state of the art. This is why we used DCF to detect dead methods in our study. Moreover, it is freely available on the web so allowing other researchers to replicate our study.

2.1.3 Refactoring vs. Optimization Perspective

The approaches discussed above (Boomsma et al. 2012; Eder et al. 2012; Fard and Mesbah 2013; Romano and Scanniello 2015; 2018; Romano et al. 2016) take a refactoring perspective, rather than an optimization one, when tackling dead code. That is, developers taking a refactoring perspective detect and then remove dead code because source code deprived of dead code is easier to comprehend and maintain (e.g., (Eder et al. 2012; Romano et al. 2020)). On the contrary, developers taking an optimization perspective want to make their software applications faster and lighter (e.g., (Obbink et al. 2018)). Taking a refactoring perspective leads to the following practical implications:

1. developers remove dead code from source code—i.e., they are not interested in removing dead code from software applications' dependencies like frameworks and libraries (e.g., as done by Obbink et al. (2018) in the context of JavaScript web-based software applications);
2. the removal of dead code is a permanent operation carried out on the source code of a target software application—as opposed to an optimization perspective in which the removal of dead code can be temporary and can be carried out on intermediate representations of source code (i.e., bytecode in the case of Java) without affecting source code.

To conclude, our study takes a refactoring perspective. That is, we are not interested in detecting dead code from an optimization perspective where dead code removal can be temporary and only affect intermediate representations of source code (e.g., bytecode) and, therefore, developers do not notice the removal of dead code.

2.2 Empirical Studies on Dead Code

Researchers have claimed that the presence of dead code in software applications has negative effects on both source code comprehensibility and maintainability (Fard and Mesbah 2013; Mäntylä et al. 2003). To verify whether these claims were well-founded, (Romano et al. 2016) designed and conducted a controlled experiment with 47 participants. The authors split the participants into two groups. The participants in the first group were asked to comprehend and then maintain Java source code containing dead code, while the participants in the other group were asked to do the same on source code deprived of dead code. The results of that experiment suggested that dead code hinders source code comprehensibility, while the authors did not observe a negative effect of dead code on source code maintainability. Later, (Romano et al. 2020) replicated that experiment three times. The combined analysis of the data from the baseline experiment and replications confirmed with stronger evidence that dead code harms source code comprehensibility. Moreover, Romano et al. found that the presence of dead code negatively affects source code maintainability when developers deal with unfamiliar source code. The authors also noted that, during the maintenance task, some participants wasted time in modifying dead code that did not contribute in any way to the resolution of that task. Romano et al. also presented the findings from an interview study with six developers. The goal of this interview study was to understand 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 use it someday. The findings from this interview study help us to motivate our empirical study (see Section 3.1). As a consequence, our study can be seen as a form of triangulation with respect to the interview study by Romano et al.

Eder et al. (2012) conducted a case study to investigate the amount of maintenance affecting dead methods in a commercial web-based software application written in .NET. As mentioned in Section 2.1, the authors leveraged dynamic code analysis to detect dead methods. The authors observed that, during the evolution of the studied software application, 7.6% of the modifications affected dead methods. Moreover, they reported that 48% of the modifications to dead methods were unnecessary (e.g., because dead methods were removed later).

Scanniello (2011) used the Kaplan-Meier estimator to analyze the death of methods across different releases of five software applications. He reported that, on two out of these five software applications, the developers avoided introducing dead code or removed dead methods as much as possible. Later, (Scanniello 2014) presented a preliminary study whose goal was to understand which software metrics are predictors for dead methods. Five out of 13 software metrics were identified as good predictors for dead methods. LOC (Lines Of Code) was the best predictor. Although it is not surprising, this implies that the larger a class, the higher the probability that its methods are dead.

As compared to the research discussed above (Eder et al. 2012; Romano et al. 2016; 2020; Scanniello 2011; 2014), we quantitatively studied both the spread and evolution of dead methods in the commit histories of 23 open-source Java desktop applications. These applications are developed in the context of open-source software projects hosted on GitHub. As for the evolution of dead code, we quantitatively studied the lifespan of dead methods, whether developers remove dead methods and use dead methods, and when dead methods are introduced.

3 Study Design

The *goal* of our study is to analyze, from a quantitative point of view, the commit histories of open-source Java desktop applications with the *purpose* of investigating *(i)* the spread of dead methods (i.e., their relative number) and *(ii)* their evolution (e.g., the lifespan of dead methods). The *perspective* of our study is that of practitioners and researchers interested in dead methods for refactoring reasons. Practitioners 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. On the other hand, researchers might be interested in conducting future research on dead code in light of our results. The *context* consists of 23 open-source Java desktop applications whose software projects were hosted on GitHub.

3.1 Research Questions

The Research Questions (RQs) of the study presented in this paper are the same as our past study (Caivano et al. 2021). As far as the spread of dead methods is concerned, we defined and then investigated the following RQ.

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

This RQ aims 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 a Java desktop application, the higher the likelihood for a developer to bump into dead methods during source code maintenance and evolution tasks. Since dead code harms both source code comprehensibility and maintainability (Romano et al. 2016; 2020), 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 four RQs.

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

Romano et al. (2020) found, as highlighted in Section 2, that developers can consciously introduce or leave dead code in a software application because they think to use it later. If a dead method has a long lifespan, its future use should be less likely. This is because that dead code is not updated during the evolution of a software application—i.e., dead code was written at a time when the software application was different (Martin 2008).

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

The study of this RQ allows us to understand if developers remove dead methods from source code. We can postulate that if developers do not remove dead methods from source code, they are unaware of their presence or they believe that dead methods are not harmful (Romano et al. 2020). Our study is purely quantitative and, therefore, we are not able to discern between these two scenarios. Nevertheless, the study of RQ3 can provide useful indications on whether developers take care of dead code. In other words, the results from RQ3 can support the qualitative findings of the interview study by Romano et al. (2020), so further contributing to the body of knowledge on dead code.

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

Based on the findings by Romano et al. (2020), developers can consciously introduce or leave dead code in the source code, because they think to use it later. We formulated this RQ to understand whether developers actually use dead methods in the future. In other words, if the introduction of dead code can be intended as a means for future re/use of source code.

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 focus our attention on the introduction of dead methods in the source code. Understanding when this code smell is introduced in a given software application could help us to delineate/define a better counteraction to deal with this smell. 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 possibly warn them if they create a method that is dead.

3.2 Study Context and Planning

We focused our study on open-source Java desktop applications with software projects hosted on GitHub. We considered GitHub because it is a very popular hosting platform for software projects and gave us the possibility to access the data of open-source software applications.

As for the detection of dead methods, we used DCF (Romano and Scanniello 2018). Dead methods are hard to detect without tool support (Wake 2003); therefore, the use of a dead-method detection tool was needed. We opted for DCF because: *(i)* it is available on the web; *(ii)* it was empirically validated; and *(iii)* it represents the state of the art for the detection of dead methods when taking a refactoring perspective (see Section 2.1).

DCF was conceived to detect dead methods in Java desktop applications, both GUI-based and CLI-based. To detect dead methods, DCF requires the bytecode of the target software application, including the bytecode of its dependencies. This implies that we needed to build the target software application before running DCF. To automate the building process of a target software application (without running its tests) and then apply DCF, we focused on software applications that used *Maven*—a popular build-automation tool for Java applications.

In Table 1, we provide some information on the 23 software applications we considered in our study. We chose these applications (i.e., both those used in our past study (Caivano et al. 2021) and the new applications) to have a heterogeneous set of software applications in terms of: *(i)* size, intended as the number of methods and classes; *(ii)* number of stars, giving an indication of software applications' popularity; *(iii)* lifespan, in terms of the number of commits; and *(iv)* application domain (see Appendix A). Moreover, the choice of these software applications was driven by DCF—it was conceived to detect dead methods in Java desktop applications and the target software applications had to be compatible

Table 1 A summary of the studied software applications

Application	# Stars	Last Analyzed Commit	Lifespan	# Methods*	# Classes*
4HWC Autonomous Car	1	5a5c472	102	137	37
8_TheWeather	1	f6abd54	38	346	37
BankApplication	72	6856256	102	537	121
bitbox	1	af2af8b	15	548	64
Density Converter	225	e70dcad	162	384	71
Deobfuscator-GUI	112	deb003e	29	221	47
graphics-tablet	0	8a3df4c	35	697	81
JavaANPR	125	eff9acf	256	786	92
javaman	0	7a03c36	58	230	38
JDM	0	a435b4d	25	82	16
JPass	81	c6b13af	134	366	82
MBot	0	ff07dac	21	56	16
SMV APP	1	4c17370	67	408	96
Calculator	9	cfa2504	112	259	60
Desktop Weather Widget	0	ea1acf7	69	86	18
IOU	1	c4ca28c	60	211	43
JavaAppBuilder	0	3c942ca	42	445	56
Metis Dictionary	1	c4ca28c	25	363	35
mvn-gui	0	e3db771	40	274	61
PocketMine-GUI	10	e1538b4	45	290	48
SaveMyPass	1	82bc309	31	507	98
SoccerQuizApp	0	0a6ca87	46	92	17
Swing Chat	2	02b64b2	73	113	21

Above the central horizontal line, we report the software applications used in our past study (Caivano et al. 2021). Below that line, we summarize the new software applications

*# Methods and # Classes mean, respectively, the maximum number of methods and classes in a commit.

†We are aware that this table exceeds the current page margins. We decided to leave this table (and other tables and figures that follow) as it is to preserve the readability of the table and because the final template of *EMSE* has larger margins.

with DCF—and by Maven—we needed Maven to automatically build the target software applications.

We are aware that some of the studied software applications are not so huge. However, we believe that this can be considered acceptable given the exploratory nature of our study and, especially, the long time it takes to detect dead methods. In that respect, we would like to recall that we needed to first build the target software application on each commit and then run DCF on that commit—on the software applications used to assess DCF, the average execution time of DCF was equal to six minutes (Romano and Scanniello 2018). In other words, while this study can contribute to enlarging the 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).

Given a software application, we locally cloned the corresponding (Git) software repository. For each commit of the master (i.e., main) branch of the considered software application, we applied DCF to gather both dead and alive (i.e., not dead) methods. To that end, we first built the software application at a commit level and then we ran DCF—as mentioned above, DCF requires the bytecode of the target software application, including the bytecode of its dependencies. It is worth mentioning that DCF reports both dead and alive methods that are "internal" to the target application. That is, the methods of the dependencies of the target application are not returned in the DCF report since such a kind of methods is not of interest when taking a refactoring perspective. If we cannot build the target application for a given commit, we skipped that commit. 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 Analyses

To perform our data analyses, we exploited the *R* statistical environment.[2] In the following of this section, we present the data analyses by RQ.

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

RQ2. Unlike RQ1, RQ2 aims to study the evolution of each dead method along the commit history of each software application. In other words, to study RQ2, we followed the evolution of each dead method. In particular, given a dead method, we computed its survival time in terms of commits (*SurvTime*)—i.e., the interval of consecutive commits, in the master branch, from the first to the last commits in which DCF detected that method as dead. A variable like *SurvTime* under/overestimates survival time when the event of interest (in our case, dead-method removing[3] and dead-method reviving commit[4]) occurs after the observation period. In studies like ours (e.g., (Chatzigeorgiou and Manakos 2014; Tufano et al. 2016)), researchers are forced to analyze a finite commit history despite the studied software application continues to evolve with time. In other words, the event of interest can occur outside the analyzed commit history. Therefore, we can distinguish two kinds of data points: *complete* and *censored* (Jr. 2011). Complete data points are the ones for which the event of interest has occurred during the observation period, while censored data points are the ones for which the event of interest has not occurred yet. By translating these definitions to our study, complete data points are dead

[2]https://www.r-project.org

[3]The dead-method removing commit (if any) of a dead method is the one in which that dead method is removed from the source code.

[4]The dead-method reviving commit (if any) of a dead method is the one in which that dead method is made alive.

methods for which we know both their dead-method introducing[5] and removing/reviving commits. In other words, removed/revived dead methods are complete data points. On the other hand, censored data points are dead methods—also referred to as censored dead methods, from here onwards—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 and thus compute an unbiased estimate of how long dead methods survive, we leveraged the *Kaplan-Meier* (KM) survival analysis (Kaplan and Meier 1958). In particular, for each software application, we built the KM survival curve, which graphically depicts the survival probability of dead methods at any point of time. From the KM survival curve, we then computed the KM median survival time (*KMMedSurvTime*), which is defined as the time for which the survival probability is equal to 0.5 (Goel et al. 2010). The KM median survival time is used, in studies like ours, to have an unbiased estimate of survival time by taking into account both complete and censored data points (Rich et al. 2010). Summing up, we used *KMMedSurvTime* to estimate how long dead methods survive in terms of commits—the *SurvTime* values and the censoring information (i.e., whether a given method was censored or not) were used to build the KM survival curves (from which the *KMMedSurvTime* are computed).

RQ3. Similarly to RQ2, we followed the evolution of dead methods along the commit history of the studied software applications. In particular, we computed the relative number of removed dead methods for each application. We named this variable *%RemovedDeadMethods*. To summarize the distribution of the values for *%RemovedDeadMethods*, we used descriptive statistics. To compute *%RemovedDeadMethods*, we used the information about the dead-method removing commit of each dead method, not the information 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 RQ4.

RQ4. To answer this RQ, we followed the evolution of dead methods along the commit history of each software application. In particular, we computed the relative number of revived dead methods for each software application. We named this variable *%RevivedDeadMethods*. 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 number of dead methods born dead (*%DeadBornMethods*) and the relative number of dead methods became dead (*%DeadBecameMethods*) for each Java desktop application. Given an application, the sum of the values for *%DeadBornMethods* and *%DeadBecameMethods* is 100%. We also used descriptive statistics to summarize the distribution of the values for both these variables. To determine if there was a (statistically) significant difference in the relative numbers of dead-born methods and dead-become ones (in each software application), we ran a (one-proportion) *Z-test* (Wilson 1927).

[5]The dead-method introducing commit of dead method is the one in which that dead method is introduced into the source code. The introduction of a dead method happens either when a method is created already dead (i.e., it was born dead) or when a method becomes dead after being alive in the previous commit.

3.4 Data Availability

The interested reader can find our replication package, which contains the scripts for the data analyses and the raw data of our study, on a public repository issuing research outputs with DOIs (i.e., Figshare) (Romano 2022).

4 Results

In this section, we present the results of our study. The results are arranged according to the defined RQs.

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

In Fig. 1, we show the box plots summarizing the distributions of the *%DeadMethods* values. These box plots suggests that dead methods are quite widespread in all the software applications. The software application with the highest average value for *%DeadMethods* is bitbox (mean = 36.749%). On the other hand, the software application with the lowest average value is Calculator (mean = 0.454%). Besides JDM and Calculator (having a minimum values for *%DeadMethods* equal to 0%), the minimum values for *%DeadMethods* range in between 1.198% (SaveMyPass) and 35.593% (bitbox). This suggests the presence of dead methods in every commit of these software applications, so confirming that this code smell is widespread. We can also note that, in the case of SMV APP and Desktop Weather Widget, the maximum values for *%DeadMethods* (96.296% and 94%, respectively) are very close to 100% (we deepen this point in Section 4.4).

The line plots (see Fig. 2) allow us to better understand how the values of $\%DeadMethods$ change across the commit history of each software application. For most of the software applications, the $\%DeadMethods$ values remain quite constant across the commit histories. Clear exceptions are Deobfuscator-GUI, javaman, MBot, SMV APP, Desktop Weather Widget, JavaAppBuilder, Metis Dictionary, and SoccerQuizApp. For

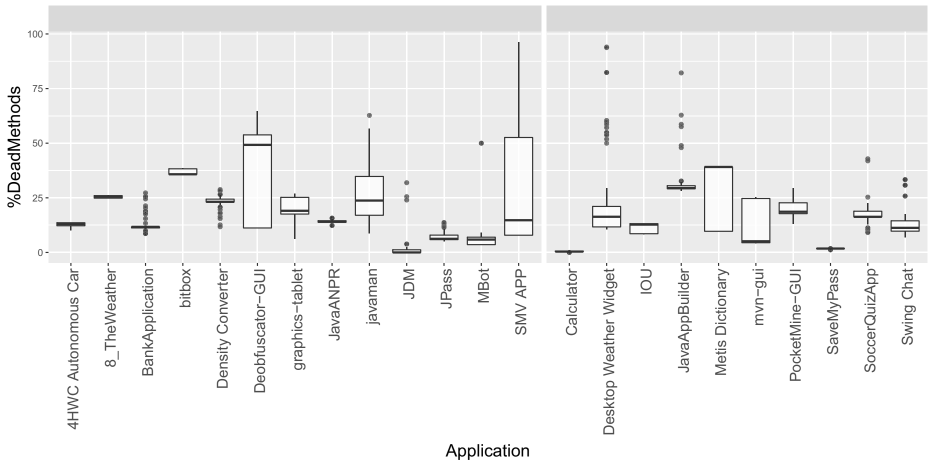

Fig. 1 Box plots for *%DeadMethods*. On the left-hand side, the software applications used in our past study (Caivano et al. 2021). On the right-and side, the new software applications

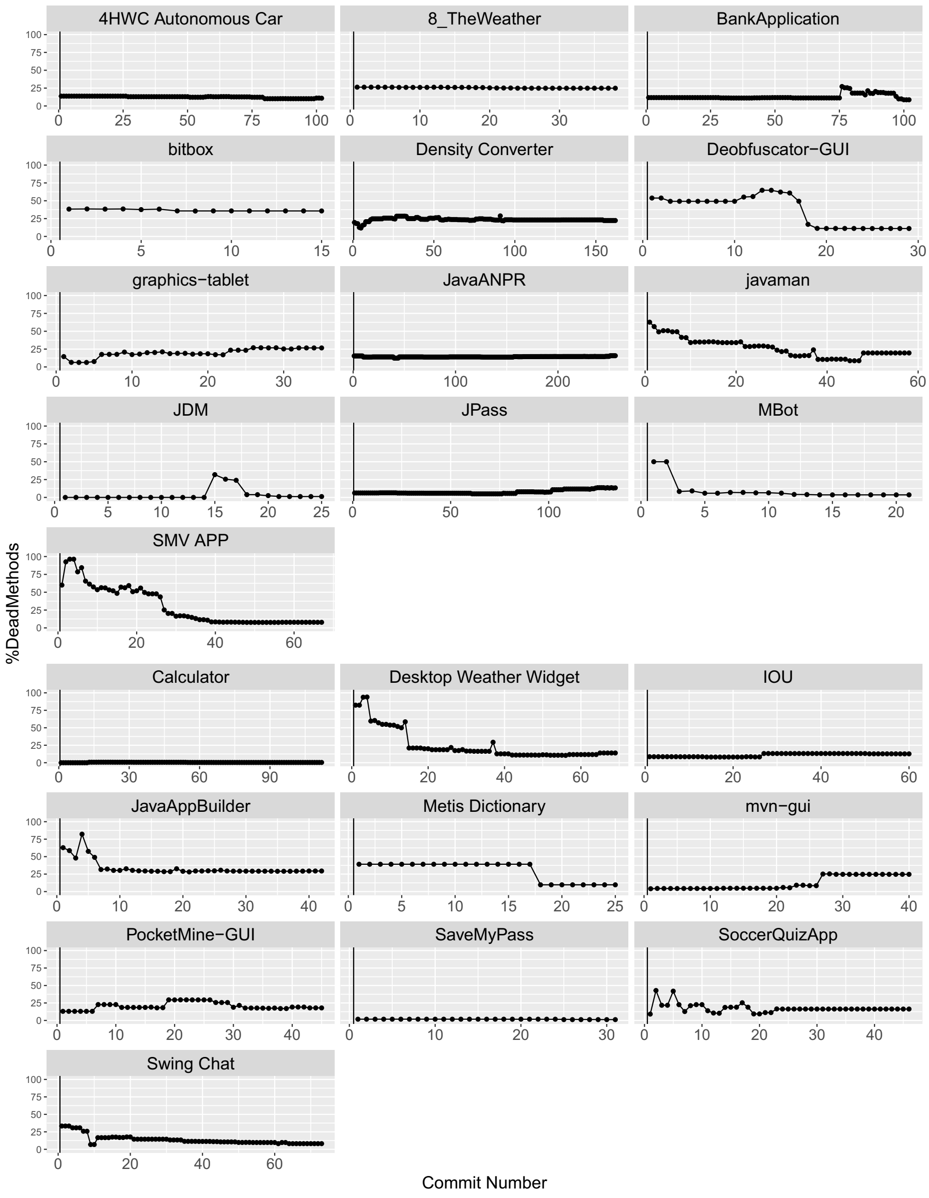

Fig. 2 Box plots for *%DeadMethods*. On the top, the software applications used in our past study (Caivano et al. 2021). On the bottom, the new software applications

these applications, we can also notice that the values for $\%DeadMethods$ decrease across the commits. The study of the next RQs can help us to better understand such a trend.

As for the comparison of the results from the extended dataset and the past one (Caivano et al. 2021), we do not obverse huge differences. Therefore, we can confirm (with stronger evidence) the presence of dead methods in Java desktop applications, as well as the presence of different trends for the relative number of dead methods in their commit histories.

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

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

In Table 2, we show the results of the survival analysis of dead methods. It is worth recalling that, in RQ2 (as well as in the RQs that follow), we considered the evolution of each (distinct) dead method in the commit history of the studied software application. That is to say that, if a given dead method, *m*, is present in more than one commit, we consider it only once. In Table 2, we report the number of (distinct) dead methods across the commit history of each application; the number of events (i.e., the number of dead-method removing/reviving commits, in our case); and the *KMMedSurvTime* value, along with the 95% confidence interval, estimated from the KM survival curve.

As shown in Table 2, we could estimate the *KMMedSurvTime* values for all the considered software applications, with the exceptions of: 8_TheWeather, bitbox, IOU, and SaveMyPass. This is because, in these four software applications, a high number of (distinct) dead methods was censored (i.e., in the last analyzed commit, these methods were still dead) and their *SurvTime* values were overall higher than the *SurvTime* values of removed/revived dead methods. In particular, only one dead method (out of 87) was removed/revived during the observation period of 8_TheWeather—its *SurvTime* value was equal to two, while the median[6] of the *SurvTime* values of the censored dead methods was equal to 38. As for bitbox, 15 out of 210 dead methods were removed/revived during the observation period, while four out of 30 dead methods were removed/revived during the observation period of IOU. The medians of removed/revived dead methods were lower than those of censored dead methods for both bitbox (six vs. 15) and IOU (18.5 vs. 60). As for SaveMyPass, the events of interest were three (while the number of dead methods was equal to nine)—the medians of the *SurvTime* values were equal to 24 and 31 for the removed/revived and censored dead methods, respectively.

For the greater part of the other software applications, we can observe that dead methods tend to survive for many commits. In particular, for twelve applications, the *KMMedSurvTime* values were equal to or greater than ten (commits), by reaching a maximum of 83 (commits) for JPass. For these applications, it would be the case of understanding from a qualitative perspective why dead methods survive for many commits. This point could represent a future direction for the research presented in this paper. It is worth noting that only on seven applications, the *KMMedSurvTime* values are lower than 10 (commits). Moreover, although we could not compute the *KMMedSurvTime* value for SaveMyPass, we know the lower bound of the confidence interval (24). This suggests that the *KMMedSurvTime* value is at least equal to 24 (commits).

The results presented above are not so different from those reported in our past paper (Caivano et al. 2021). In other words, we can confirm that, in open-source Java desktop

[6]We mean the traditional median (i.e., the middle value of a dataset (Wohlin et al. 2012)), not the estimate computed from the KM survival curve (i.e., *KMMedSurvTime*).

Table 2 Survival analysis results

Application	# (Distinct) Dead Methods	# Events	*KMMedSurvTime*	
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]
Calculator	2	1	8	[8, −]
Desktop Weather Widget	86	73	4	[2, 7]
IOU	30	4	−	[−, −]
JavaAppBuilder	369	228	6	[4, 8]
Metis Dictionary	143	137	17	[17, 17]
mvn-gui	76	16	26	[24, −]
PocketMine-GUI	133	75	11	[11, 13]
SaveMyPass	9	3	−	[24, −]
SoccerQuizApp	60	44	3.5	[2, 5]
Swing Chat	37	28	10	[5, 28]

Above the central horizontal line, the software applications used in our past study (Caivano et al. 2021). Below that line, the new software applications

applications, dead methods tend to survive for a long time, in terms of commits, before developers bury or revive them.

Further analysis To deepen the study of RQ2, we analyzed whether removed dead methods survived as much as revived ones. To do so, we could not use the KM survival analysis to estimate how long removed dead methods and revived ones survive, respectively. This is because we did not know which censored dead methods will be removed in the future, or which ones will be revived. Therefore, we directly used the *SurvTime* values and showed their distributions for removed and revived dead methods, respectively, by means of box plots (see Fig. 3). These box plots also depict the distributions for censored dead method to provide a complete picture of the results. As shown in Fig. 3, for eight software applications, we could not compare removed and revived dead methods with respect to *SurvTime* because of the absence of data points (i.e., no removed dead method or no revived dead method). For the remaining software applications, we can observe that the boxes for removed and revived dead methods either mostly overlap or are higher for removed dead methods. We also verified if there were (statistically) significant differences (at a 5% significance level) in the

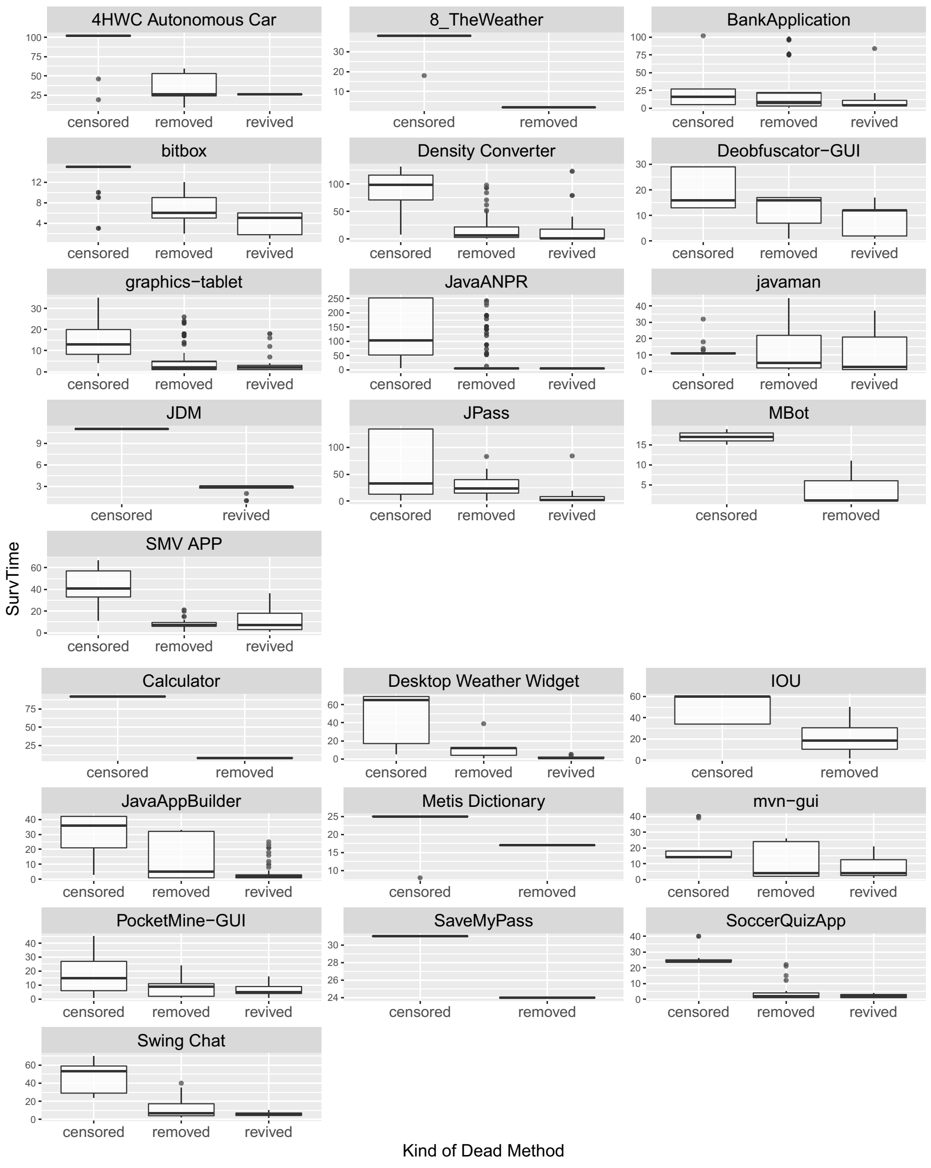

Fig. 3 Box plots for *SurvTime*. On the top, the software applications used in our past study (Caivano et al. 2021). On the bottom, the new software applications

SurvTime values between removed and revived dead methods. To do so, we planned to use the (unpaired two-tailed) *t-test* (Welch 1947) but the underlying assumption of normality[7]

[7]To check the normality assumption, we used the *Shapiro-Wilk test* (Shapiro and Wilk 1965).

was never met. Consequently, we used the non-parametric alternative to the t-test, namely the (unpaired two-tailed) *Mann-Whitney U test* (Mann and Whitney 1947). In case of a statistically significant difference, we used the *Cliff's δ effect size* to estimates the magnitude of such a difference, which is considered: *negligible* if $\delta < 0.147$; *small* if $0.147 \leq \delta < 0.33$; *medium* if $0.33 \leq \delta < 0.474$; or *large* otherwise (Romano et al. 2006).

In Table 3, we show the results (i.e., p-values) of the Mann-Whitney U test, along with the Cliff's δ values (when there were significant differences). We can observe that, on five software applications, the difference between removed and revived dead methods, in terms of *SurvTime*, was significant. These software applications are: Density Converter, Deobfuscator-GUI, JPass, Desktop Weather Widget, and JavaAppBuilder. As shown in Fig. 3, the significant differences are due to higher *SurvTime* values for removed dead methods. The magnitude of the significant differences ranges from small to large. Summing up, both box plots and inferential analysis (i.e., Mann-Whitney U test) suggest that removed dead methods tend to survive more than revived ones.

Answer to RQ2: In the studied open-source Java desktop applications, dead methods generally survive for a long time, in terms of commits, before being buried or revived. Also, removed dead methods tend to survive more than revived ones.

Table 3 P-values from the Mann-Whitney U test concerning the comparison between removed dead methods and revived ones with respect to *SurvTime*, along with Cliff's δ values when there were significant differences (in these case, p-values are reported in bold)

Application	p-value	Cliff's δ
4HWC Autonomous Car	1	–
BankApplication	0.217	–
bitbox	0.159	–
Density Converter	**0.002**	0.323 (small)
Deobfuscator-GUI	**0**	0.481 (large)
graphics-tablet	0.853	–
JavaANPR	0.62	–
javaman	0.088	–
JPass	**0.014**	0.561 (large)
SMV APP	0.678	–
Desktop Weather Widget	**0**	0.883 (large)
JavaAppBuilder	**0**	0.467 (medium)
mvn-gui	0.677	–
PocketMine-GUI	0.483	–
SoccerQuizApp	0.305	–
Swing Chat	0.442	–

Above the central horizontal line, the software applications used in our past study (Caivano et al. 2021). Below that line, the new software applications

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

In Table 4, we show the percentage of (distinct) dead methods that are removed (i.e., *%RemovedDeadMethods*) in each application, as well as the percentage of dead methods that are revived (i.e., *%RevivedDeadMethods*) or censored (i.e., *%CensoredDeadMethods*). While the *%RevivedDeadMethods* values are of interest to the study of the next RQ, the *%CensoredDeadMethods* values are shown to have a full picture of the results. Given an application, the sum of the *%RemovedDeadMethods*, *%RevivedDeadMethods*, and *%CensoredDeadMethods* values is 100%.

For eight software applications (out of 23), dead methods appear not to be removed at all: the *%RemovedDeadMethods* values range in between 0% (JDM) and 22.078% (JPass). For the remaining software applications, it seems that developers pay attention to the removal

Table 4 Results regarding the relative number of removed and revived dead methods

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
Calculator	50	0	50
Desktop Weather Widget	36.047	51.163	12.791
IOU	13.333	0	86.667
JavaAppBuilder	12.195	52.304	35.501
Metis Dictionary	95.804	0	4.196
mvn-gui	17.105	3.947	78.947
PocketMine-GUI	40.602	20.301	39.098
SaveMyPass	33.333	0	66.667
SoccerQuizApp	45	30	25
Swing Chat	54.054	21.622	24.324
Mean	38.812	19.864	41.324
SD	24.755	24.899	28.117
Median	41.549	12.847	35.501

The relative number of censored dead methods is also shown for completeness. Above the central horizontal line, the software applications used in our past study (Caivano et al. 2021). Below that line, the new software applications (and then some descriptive statistics) of dead methods. Indeed, the *%RemovedDeadMethods* values range in between 33.333% (SaveMyPass) and 95.804% (Metis Dictionary). By looking at both Table 4 and Fig. 2, we can grasp that there are different removal patterns. For example, in javaman, the developers progressively removed dead method, while in Deobfuscator-GUI, the developers removed dead methods in a shorter number of commits.

In Table 4, we also show the values of mean, SD, and median of *%RemovedDeadMethods* when aggregating the data from all the software applications. The average value for *%RemovedDeadMethods* of the studied applications is 38.812%, while the median value is 41.549%. These values confirm that, in general, the developers paid attention to the removal of dead methods. The SD value is high (24.755) so remarking that the developers took care of dead method removal in a different fashion. That is, developers behave differently on different software applications.

The results presented above are consistent with those reported in our past paper (Caivano et al. 2021), so allowing, also in this case, strengthening our overall conclusion.

Answer to RQ3: In most of the studied open-source Java desktop applications, the developers paid attention to dead methods by giving them a decent burial. Only for few software applications, this does not hold.

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

The *%RevivedDeadMethods* values suggest that developers rarely revive dead methods (see Table 4). In particular, on 13 software applications (out of 23), the *%RevivedDeadMethods* values are quite small, ranging from 0% (seven cases) to 14.286%. Only in four cases (i.e., JDM, SMV APP, Desktop Weather Widget, and JavaAppBuilder), the *%RevivedDeadMethods* values are higher than 50%. As for JDM, we can note a peak in the middle of the commit history shown in Fig. 2. This allows us to postulate that the developers consciously introduced some dead methods to use them in the subsequent commits. A similar postulation can be done for SMV APP, Desktop Weather Widget, and JavaAppBuilder as well. However, in these three software applications, the peak of dead methods arises at the beginning of the commit history (see Fig. 2).

Again, the results presented above are consistent with those reported in our past paper (Caivano et al. 2021), so allowing us to strengthen our conclusion.

Answer to RQ4: In the studied open-source Java desktop applications, the developers rarely revived dead methods. Only in few occasions, the developers seem to introduce dead methods 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 became dead later (*%DeadBecameMethods*), as well as the p-values returned by the Z-test. In this table, we also report some descriptive statistics.

Table 5 Results regarding the relative number of dead methods born dead and became dead, as well as p-values from the Z-test (in bold those suggesting significant differences)

Application	*%DeadBornMethods*	*%DeadBecameMethods*	p-value
4HWC Autonomous Car	92.593	7.407	**0**
8_TheWeather	100	0	–
BankApplication	69.259	30.741	**0**
bitbox	98.095	1.905	**0**
Density Converter	95.139	4.861	**0**
Deobfuscator-GUI	98.718	1.282	**0**
graphics-tablet	88.698	11.302	**0**
JavaANPR	99.578	0.422	**0**
javaman	84.507	15.493	**0**
JDM	100	0	–
JPass	76.623	23.377	**0**
MBot	100	0	–
SMV APP	77.049	22.951	**0**
Calculator	100	0	–
Desktop Weather Widget	96.512	3.488	**0**
IOU	66.667	33.333	0.1
JavaAppBuilder	72.087	27.913	**0**
Metis Dictionary	100	0	–
mvn-gui	69.737	30.263	**0.001**
PocketMine-GUI	97.744	2.256	**0**
SaveMyPass	100	0	–
SoccerQuizApp	80	20	**0**
Swing Chat	96.296	3.704	**0**
Mean	89.535	10.465	
SD	12.038	12.038	
Median	96.296	3.704	

Above the central horizontal line, the software applications used in our past study (Caivano et al. 2021). Below that line, the new software applications (and then some descriptive statistics)

The comparison between the values of *%DeadBornMethods* and *%DeadBecameMethods* clearly indicates that dead methods were mostly born dead rather than becoming dead later. The *%DeadBornMethods* values range in between 66.667% and 100% (while the *%DeadBecameMethods* values range in between 0% and 33.333%). The observed difference in the percentages of dead-born and dead-became methods is almost always significant (at a 5% significance level) as the p-values from the Z-test suggest. In particular, in six cases (out of 23), we could not run the test because the values of %DeadBecameMethods were equal to 0%; in the other cases, the test always suggested a significant difference with the only exception of IOU. We can also observe high median (96.296%) and mean (89.535%) *%DeadBornMethods* values and a low SD value (12.038). These descriptive statistics confirm that most dead methods were born dead and this outcome holds for any software application.

These results confirm those we obtained on the hold dataset (Caivano et al. 2021).

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

4.6 Final Remarks about the New Dataset

As mentioned before, we considered ten further open-source Java desktop applications in order to extend our past dataset (Caivano et al. 2021). Regardless of the RQ, the results observed on the new dataset (comprising 23 Java desktop applications) are consistent with those reported by Caivano et al. (2021) (observed on a subset of 13 Java desktop applications) so allowing us to strengthen our conclusions.

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, affect the studied open-source Java desktop applications. This finding complements those of past research reporting a large amount of dead code in commercial web-based applications written in PHP and .NET (Boomsma et al. 2012; Eder et al. 2012). Therefore, our finding (on Java desktop applications) joined with past ones (on commercial web-based applications) seem to indicate that dead code affects software applications regardless of: the programming language (Java vs. .NET vs. PHP); kind of software application (desktop vs. web-based); kind of license (open-source vs. commercial); and application domain (e.g., from conversion of images to weather forecast in our research, and from insurance to customers relationship management in the research by Eder et al. (2012) and Boomsma et al. (Boomsma et al. 2012)). This outcome is of interest to **Practitioners**. In particular, practitioners should be aware that, whatever the software application is (i.e., regardless of programming language, kind of software application and license, and application domain), they could bump into dead code and then be exposed to its negative effects (Romano et al. 2016; 2020). Therefore, practitioners should take care of dead code, so removing it from the source code or avoiding the introduction of this code smell as much as possible. Nevertheless, we believe that dead code deserves further attention from the SE research community since, for example, there is a lack of empirical evidence on the spread of dead code in software applications for mobile devices. **Researchers** could be interested in conducting empirical studies in order to fill this gap.

We observed a variability in the amount of dead methods across the studied software applications (on average, dead methods accounted for 0.454% to 36.749% of all methods). This finding supports the one by Scanniello (2011), who observed that developers avoided introducing dead methods or removed dead methods as much as possible on two out of the five open-source Java desktop applications studied. The variability in the amount of dead methods could be due to factors internal to software projects—e.g., LOC, as suggested by the preliminary investigation by Scanniello (2014). Therefore, we foster **researchers** to deepen the study of which factors can predict the introduction of dead methods in open-source software applications. **Researchers** could be also interested in studying if these results hold for commercial software applications. Our findings justify future work on this matter.

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 effects of this code smell, in terms of source code comprehensibility and maintainability (Romano et al. 2016; 2020), is greater. **Practitioners** should thus keep under control the spread of dead methods and, therefore, we recommend them to periodically plan code reviews (with tool support) to detect and remove dead methods from open-source Java desktop applications.

We found that most dead methods were born dead, rather than becoming dead later. In other words, most dead methods are dead since the creation of the corresponding methods. This result is consistent with previous findings on code and test smells (Tufano et al. 2016; Tufano et al. 2017), and it contradicts the common wisdom for which code smells are due to side effects of software evolution (Parnas 1994). **Practitioners** should pay attention to the design of open-source Java desktop applications by trying to avoid as much as possible 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 thus provide practitioners with just-in-time dead-method detection tools—i.e., tools highlighting the presence of dead methods in real time, while developers code. **Practitioners** could clearly take advantage of such a kind of dead-method 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 (Romano et al. 2020). However, we found that developers rarely revive dead methods. We can thus make Martin's recommendation for **practitioners** (in particular, those working on open-source Java desktop applications) 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 to developers when they have to comprehend and maintain source code. This is because source code deprived of dead code is easier to comprehend and maintain (Romano et al. 2016; 2020), and developers uselessly spend some effort modifying dead code (Eder et al. 2012; Romano et al. 2020).

Dead methods generally survive for a long time, in terms of commits. Accordingly, the future re/use of dead methods, in a given software application, should be unlikely because dead methods are not updated with the rest of that software application (Martin 2008). Furthermore, the results of our further analysis suggest that revived dead methods tend to survive less (with respect to removed dead methods); this is because the revival of dead methods usually happens in a short commit frame. Based on these findings and the known detrimental effects of dead code (Eder et al. 2012; Romano et al. 2016; 2020), we recommend again **practitioners** avoid introducing dead methods and remove them from open-source Java desktop applications whenever it is possible.

The results of the survey by Yamashita and Moonen (2013) 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 to developers. In some of the studied software applications, we observed that developers paid 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 (2013). This outcome is clearly relevant for **researchers** since dead code seems to be relevant from a practical point of view. In some other applications, we found that the developers did not remove dead methods. We can only postulate possible reasons, namely: *(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, **researchers** should investigate on the above-mentioned postulations and inform developers about the negative effects of dead methods and their uselessness.

Summing up, our study has the merit of expanding the body of knowledge on dead code. In particular, we bring empirical evidence on the spread and evolution of dead methods in open-source Java desktop applications and complement/support the findings of past research on this code smell, including our past study (Caivano et al. 2021) we here extends by considering ten more software applications. Nevertheless, we do not consider our findings conclusive; rather, we believe that our findings 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. Finally, our findings could foster **researchers** to replicate our study in a different context (e.g., on web-based or mobile applications).

5.2 Threats to Validity

In the section, we discuss the threats that might affect the validity of our results. Based on the validity schema by Wohlin et al. (2012), these threats to validity fall in the following categories: external, conclusion, and construct.[8]

5.2.1 External Validity

Threats to external validity deal with the generalizability of results (Wohlin et al. 2012). Both the software applications considered in our study—almost all software applications used SWING as a GUI framework—and their number might affect external validity. For example, the studied software applications might be scarcely representative of open-source Java desktop applications whose projects are hosted on GitHub or might not represent the universe of Java desktop applications. While we gather empirical evidence on the spread and evolution of dead methods (so increasing the body of knowledge on dead code), we believe that the gathered evidence can justify more resource- and time-demanding research on dead methods. For example, we advise further work on a larger number of open-source Java desktop applications (e.g., randomly sampled from GitHub). To deal with external validity threats, in this paper, we extend our past study (Caivano et al. 2021) by considering ten further Java desktop applications with software projects hosted on GitHub. The results presented in this paper confirm those of our past study.

[8]We do not have any threat to internal validity since such threats are defined as influences that can affect the independent variable with respect to causality, and we did not investigate causality in our study.

5.2.2 Construct Validity

Threats to construct validity concern the relation between theory and observation (Wohlin et al. 2012). The data collection approach might affect the results. In particular, when a commit did not compile, we skipped that commit (see Section 3.2). Although we relied on a popular build-automation tool (i.e., Maven) to automatically build the software applications, the lack of ability to build open-source applications at a commit level is an inherent limitation to any study similar to ours (Tufano et al. 2017).

To detect dead methods, we used DCF. We opted for this tool for several reasons. It is freely available on the web (and some of the developers of that tool took part in this research). And more importantly, the validity of DCF was empirically assessed on a gold standard by comparing it with other baseline tools to detect dead code (Romano and Scanniello 2018). The results of this comparison suggested 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 our results. In particular, methods invoked indirectly by means of reflection are not supported by DCF, except for those used to customize the serialization/deserialization process of objects. That is, the use of reflection might represent an issue when detecting dead methods with DCF.

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.

The metrics used to answer our RQs might pose a threat to construct validity. However, there is no accepted metric to quantitatively assess the spread and evolution of dead methods. The proposal of these metrics might represent another contribution of our paper.

5.2.3 Conclusion Validity

Threats to conclusion validity concern issues that affect the ability to draw the correct conclusion (Wohlin et al. 2012). We addressed such a kind of threats by checking the assumptions of the statistical tests before using them (in case these assumptions were not met, we used the non-parametric alternatives). Also, we used robust statistical tests.

As for the survival analysis, the use of the Kaplan-Mayer method might have affected the validity of the results. However, this method represents one of the best options for survival analysis (Tufano et al. 2017).

6 Conclusion

In this paper, we present the results of an exploratory study on dead methods in open-source Java desktop applications. We quantitatively analyzed the commit histories of 23 open-source Java desktop applications, whose software projects were hosted on GitHub, for a total of 1,587 commits. We studied the spread of dead methods and the evolution of this code smell. The most important take-away results of our study can be summarized as follows: *(i)* dead methods affect open-source Java desktop applications; *(ii)* dead methods generally survive for a long time before being "buried" or "revived;" *(iii)* dead methods that are then revived tend to survive less (with respect to dead methods that are then buried); *(iv)* dead methods are rarely revived; and *(v)* 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, due to the exploratory nature of our research, our findings allow reaching a better understanding of the presence and evolution of dead methods in open-source Java desktop applications so enlarging the body of knowledge on this code smell. Our findings also justify future work. For example, researchers could be interested in conducting large-scale studies on software applications larger than ours or focus their attention on another kind of software applications (e.g., software applications for mobile devices or software applications developed by using model-driven principles or advanced programming techniques intended to preserve compliance to user requirements).

Appendix

In this appendix, we provide a brief description of the studied Java Desktop applications, including the links to their repositories on GitHub.

- **4HWC Autonomous Car:** a simulator that allows verifying the movements of an autonomous car (http://github.com/4hwc/4HWCAutonomousCar).
- **8_TheWeather:** an application that shows the current weather condition, as well as short and long-term forecasts for an user-specified location (http://github.com/workofart/WeatherDesktop).
- **BankApplication:** an application that provides support for some banking operations (http://github.com/derickfelix/BankApplication).
- **bitbox:** a utility tool for bit operations (http://github.com/fusiled/bitbox).
- **Calculator:** a calculator (https://github.com/javadev/calculator).
- **Density Converter:** a tool that helps converting single or batches of images to specific formats and density versions (http://github.com/patrickfav/density-converter).
- **Deobfuscator-GUI:** it provides a GUI for a popular Java deobfuscator based on CLIDeobfuscator-GUI (http://github.com/java-deobfuscator/deobfuscator-gui).
- **Desktop Weather Widget:** an application that shows weather information (https://github.com/kivimango/weather-widget).
- **graphics-tablet:** a drawing application (http://github.com/alexdoublesmile/5-app-graphics-tablet).
- **IOU:** an application that helps people keep track of money owed by one to each other (https://github.com/donalmurtagh/iou).
- **JavaANPR:** an application to automatically recognize number plates from vehicle images (http://github.com/oskopek/javaanpr).
- **JavaAppBuilder:** A Java desktop application builder (https://github.com/pedrohenriquebr/appbuilder).
- **javaman:** a Java implementation of the popular Bomberman game (http://github.com/malluce/javaman).
- **JDM:** an application to manage the download of files (http://github.com/iamabs2001/JDM).
- **JPass:** an application to manage passwords (http://github.com/gaborbata/jpass).
- **MBot:** an application to record and automate mouse and keyboard events (http://github.com/znyi/MBot).
- **Metis Dictionary:** an English-Hungarian dictionary (https://github.com/gaborbata/metis-dictionary).

- **mvn-gui:** an application to manage Apache Maven projects (https://github.com/oguzhancevik/mvn-gui).
- **PocketMine-GUI:** it provides a GUI for PocketMine-MP (https://github.com/PEMapModder/PocketMine-GUI).
- **SaveMyPass:** an application to manage passwords (https://github.com/tiagoppinho/SaveMyPass).
- **SMV APP:** an application to that provides support to a car repair shop (http://github.com/bfriscic/ZavrsniRad).
- **SoccerQuizApp:** an application to evaluate the knowledge of Soccer and Futsal rules. (https://github.com/cicciog/SoccerQuizApp)
- **Swing Chat:** it simulates a chat application for multiple users (https://github.com/ingokuba/swing-chat).

Funding Open access funding provided by Università degli Studi di Salerno within the CRUI-CARE Agreement.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

Boomsma H, Hostnet BV, Gross HG (2012) Dead code elimination for web systems written in php: lessons learned from an industry case. In: Proceedings of international conference on software maintenance, pp 511–515. IEEE

Brown WH, Malveau RC, McCormick HWS, Mowbray TJ (1998) AntiPatterns: Refactoring software, architectures, and projects in crisis, 1st edn. Wiley

Caivano D, Cassieri P, Romano S, Scanniello G (2021) An exploratory study on dead methods in open-source java desktop applications. In: ACM/IEEE international symposium on empirical software engineering and measurement, pp 10:1–10:11

Chatzigeorgiou A, Manakos A (2014) Investigating the evolution of code smells in object-oriented systems. Innov Syst Softw Eng 10(1):3–18

Eder S, Junker M, Jürgens E, Hauptmann B, Vaas R, Prommer KH (2012) How much does unused code matter for maintenance? In: Proceedings of international conference on software engineering, pp 1102–1111. IEEE

Fard AM, Mesbah A (2013) Jsnose: detecting javascript code smells. In: Proceedings of international working conference on source code analysis and manipulation, pp 116–125. IEEE

Fowler M (1999) Refactoring: improving the design of existing code, 1st edn. Addison-Wesley

Goel MK, Khanna P, Kishore J (2010) Understanding survival analysis: Kaplan-meier estimate. International Journal of Ayurveda Research 1:274–278

Haas R, Niedermayr R, Roehm T, Apel S (2020) Is static analysis able to identify unnecessary source code? ACM Trans Softw Eng Methodol 29(1):6:1–6:23

Hermans F, Aivaloglou E (2016) Do code smells hamper novice programming? a controlled experiment on scratch programs. In: Proceedings of international conference on program comprehension, pp 1–10

Jr. RGM (2011) Survival Analysis, 2nd edn. Wiley

Kaplan EL, Meier P (1958) Nonparametric estimation from incomplete observations. J Am Stat Assoc 53(282):457–481

Khomh F, Di Penta M, Gueheneuc YG (2009) An exploratory study of the impact of code smells on software change-proneness. In: Proceedings of working conference on reverse engineering, pp 75–84. IEEE

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. Ann Math Stat 18(1):50–60

Mäntylä M, Vanhanen J, Lassenius C (2003) A taxonomy and an initial empirical study of bad smells in code. In: Proceedings of international conference on software maintenance, pp 381–384. IEEE

Martin RC (2008) Clean code: a handbook of agile software craftsmanship, 1st edn. Prentice Hall

Moha N, Gueheneuc YG, Duchien L, Le Meur AF (2010) Decor: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36

Obbink NG, Malavolta I, Scoccia GL, Lago P (2018) An extensible approach for taming the challenges of javascript dead code elimination. In: Proceedings of international conference on software analysis, evolution and Reengineering, pp 291–401

Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2018) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir Softw Eng 23(3):1188–1221

Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD (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, pp 101–110. IEEE

Parnas DL (1994) Software aging. In: Proceedings of international conference on software engineering, pp 279–287. IEEE

Rich JT, Neely JG, Paniello RC, Voelker CCJ, Nussenbaum B, Wang EW (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 143:331–336

Romano J, Kromrey JD, Coraggio J, Skowronek J (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, pp 1–3

Romano S (2018) Dead code. In: Proceedings of international conference on software maintenance and evolution, pp 737–742

Romano S (2022) On the spread and evolution of dead methods in java desktop applications: a replication package. https://figshare.com/s/bc90817003996de8855c

Romano S, Scanniello G (2015) Dum-tool. In: Proceedings of international conference on software maintenance and evolution, pp 339–341. IEEE

Romano S, Scanniello G (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, pp 167–174. IEEE

Romano S, Scanniello G, Sartiani C, Risi M (2016) A graph-based approach to detect unreachable methods in java software. In: Proceedings of symposium on applied computing, pp 1538–1541. ACM

Romano S, Vendome C, Scanniello G, Poshyvanyk D (2016) Are unreachable methods harmful? results from a controlled experiment. In: Proceedings of international conference on program comprehension, pp 1–10. IEEE

Romano S, Vendome C, Scanniello G, Poshyvanyk D (2020) A multi-study investigation into dead code. IEEE Trans Softw Eng 46(1):71–99

Scanniello G (2011) Source code survival with the kaplan meier. In: Proceedings of international conference on software maintenance, pp 524–527

Scanniello G (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, pp 392–397

Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3/4):591–611

Tip F, Palsberg J (2000) Scalable propagation-based call graph construction algorithms. In: Proceedings of conference on object-oriented programming, systems, languages, and applications, pp 281–293. ACM

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2016) An empirical investigation into the nature of test smells. In: Proceedings of international conference on automated software engineering, pp 4–15. ACM

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2017) There and back again: can you compile that snapshot? Journal of Software: Evolution and Process 29(4):e1838

Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Trans Softw Eng 43(11):1063–1088

Wake WC (2003) Refactoring workbook, 1st edn. Addison-Wesley

Welch BL (1947) The generalisation of student's problems when several different population variances are involved. Biometrika 34(1-2):28–35

Wilson EB (1927) Probable inference, the law of succession, and statistical inference. J Am Stat Assoc 22(158):209–212

Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B, Wessln A (2012) Experimentation in software engineering springer

Yamashita A, Moonen L (2013) Do developers care about code smells? an exploratory survey. In: Proceedings of working conference on reverse engineering, pp 242–251. IEEE

Affiliations

Danilo Caivano[1] · Pietro Cassieri[2] · Simone Romano[3] · Giuseppe Scanniello[3]

Danilo Caivano
danilo.caivano@uniba.it

Pietro Cassieri
pietro.cassieri@studenti.unibas.it

Giuseppe Scanniello
gscanniello@unisa.it

1 University of Bari, Bari, Italy

2 University of Basilicata, Potenza, Italy

3 University of Salerno, Fisciano, Italy