2018 IEEE International Conference on Software Maintenance and Evolution

Dead Code

Simone Romano
University of Basilicata
Potenza, Italy
simone.romano@unibas.it

***Abstract*—Dead code is a bad smell. It is conjectured to be harmful and it appears to be also a common phenomenon in software systems. Surprisingly, dead code has received little empirical attention from the software engineering research community. This post-doctoral track paper shows the main results of a multi-study investigation into dead code with an overarching goal to study *when* and *why* developers introduce dead code, *how* they perceive and cope with it, and *whether* dead code is harmful. This investigation is composed of semi-structured interviews with software professionals and four experiments at the University of Basilicata and the College of William & Mary. The results suggest that it is worth studying dead code not only in maintenance and evolution phases, where the results suggest that its presence is detrimental to developers, but also in design and implementation phases, where source code is born dead because developers consider dead code as a sort of reuse means. The results also foster the development of tools for detecting dead code. In this respect, two approaches were proposed and then implemented in two prototypes of supporting tool.**

***Index Terms*—bad smell, dead code, unused code, unreachable code, lava flow**

I. Research Problem and Hypotheses

Bad smells (or simply smells) are indicators of potential problems in software systems [2]. There exist different smells in the literature, one of them is *dead code* (aka *unused code* [2], *unreachable code* [3], or *lava flow* [4]). It can be defined as unnecessary source code since it is unused and/or unreachable (*i.e.,* never executed) [2], [5]. Dead code is conjectured to be harmful (*e.g.,* [2], [3], [5]). In this respect, Mäntylä asserted that dead code hinders source code comprehension [5], while Fard and Mesbah [3] stated that dead code negatively affects source code maintainability because it makes source code more difficult to understand. Moreover, developers could waste time by maintaining source code that is actually dead [6].

Dead code seems to be a quite common problem too [4], [6], [7]. Brown *et al.* [4] reported that, during the code examination of an industrial software system, they found a large amount of source code (between 30% and 50% of the total) that was not understood or documented by any developer currently working on that system. Later, they discovered that this code was dead. Boomsma *et al.* [7] and Eder *et al.* [6] observed large amounts of dead code in industrial software systems too. Boomsma *et al.* [7] reported that, on a subsystem of a web system written in PHP, the developers removed 2,740 dead files, namely about 30% of the subsystem's files. Eder *et al.* [6] reported that, on a software system written in .NET, about 25% of the method genealogies[1] was dead. Moreover, the results of a survey by Yamashita and Moonen [8] indicate that dead code detection is one of the features software professionals would like to have in their supporting tools.

Although there is some consensus on the fact that dead code is a common phenomenon, might be harmful, and matters to software professionals, dead code has been marginally investigated (*i.e.,* [6], [9], [10]). In particular, there is a lack of empirical investigations into *when* and *why* dead code is introduced, *how* developers both perceive and cope with this smell, and *whether* dead code is really harmful. To fill this gap, a multi-study investigation into dead code was designed and conducted with the overarching goal to answer the following research questions [1], [11]:

RQ1. *When and why do developers introduce dead code?*
RQ2. *How do developers perceive and cope with dead code?*
RQ3. *Is dead code harmful?*
- **RQ3.a.** *Is dead code harmful to comprehend (unfamiliar) source code?*
- **RQ3.b.** *Is dead code harmful to perform modification/change tasks on (familiar and unfamiliar) source code?*

The multi-study investigation is graphically summarized in Figure 1. It took place between December 2015 and December 2016 and comprised two kinds of studies: semi-structured interviews and controlled experiments. In particular, six software professionals (*i.e.,* project managers and developers) were interviewed to understand *when* and *why* developers introduce dead code and *how* they perceive and cope with this smell (*i.e.,* RQ1 and RQ2). To study *whether* dead code is harmful (*i.e.,* RQ3), a first experiment (*i.e.,* the baseline one) with students was conducted at the University of Basilicata [12]. This experiment was then replicated once at the University of Basilicata and twice at the College of William & Mary. In any experiment, the participants (*i.e.,* students) were divided into two groups: participants who had to accomplish tasks (*e.g.,* comprehension and/or modification tasks) in the presence of dead code; and participants who had to accomplish the same tasks as the other group but in the absence of this smell.

The results from the multi-study investigation suggest that it is worth studying dead code not only in maintenance and evolution phases, where the results suggest that its presence is detrimental to developers, but also in design and implementation phases. The results also motivate research on techniques

[1]A method genealogy is the list of methods that represent the evolution of a single method over different versions of a software system.

2576-3148/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSME.2018.00092

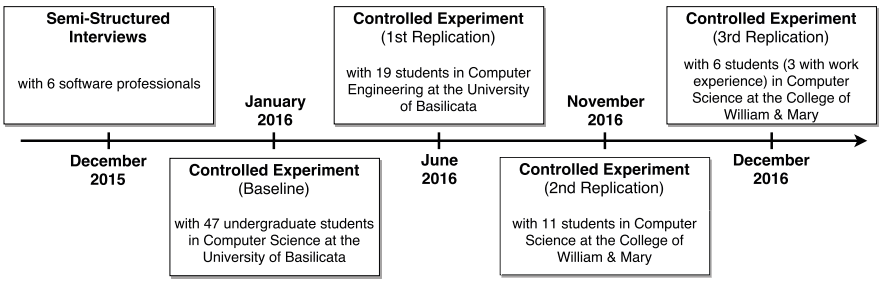

Fig. 1. Summary of the multi-study investigation into dead code.

for detecting dead code. In this respect, two approaches for the detection of dead code in Java software systems were proposed [1], [13], [14]. These approaches were implemented in two prototypes of supporting tool (simply tools from here onwards), *DUM-Tool* [15] and *DCF*, and empirically assessed. The obtained results suggest that DUM-Tool and DCF are both good solutions for developers, who need to remove dead code.

II. Semi-Structured Interviews

To investigate RQ1 and RQ2, six software professionals were interviewed.

Participants. They were software professionals in the contact network of the Software Engineering (SE) research group at the University of Basilicata. They had from four to ten years of work experience. Two out of six participants were project managers, whereas the others were developers. The workforce of the companies, where the participants were employed as either project managers or developers, ranged from about 10 to 40,000 employees.

Procedure. The used procedure was similar to the one Murphy-Hill *et al.* [16] used in their study. Each interview involved the interviewer and only one software professional (*i.e.,* the interviewee) at a time. The interviews were audio-recorded. Each of them consisted of four parts: *(i)* gathering of information about the interviewee (*e.g.,* her position in the company); *(ii)* open-ended question where the interviewee could freely talk about dead code according to her experience; *(iii)* the interviewee selected two SE topics (from a list) and then she discussed dead code in relation to the selected topics; and *(iv)* the interviewer chose (other) two SE topics and then the interviewee discussed dead code in relation to the chosen topics.

Analysis Procedure. After transcribing the audio recordings of the interviews, the transcriptions were analyzed by means of a thematic analysis template [17].

III. Controlled Experiments

A summary of the controlled experiments, conceived to study RQ3, is shown in Table I.

Participants. They were under/graduate students at either the University of Basilicata or the College of William & Mary (see Table I for further details). In total, 83 students took part in the experiments.

Procedure. In each experiment, we divided the participants into two groups: *DC*, group of participants administered with a code base containing dead code; and *NoDC*, group of participants administered with the same code base as the DC group, but such a code base was deprived of dead code. Then the participants were asked to perform some tasks on that code base. In particular, in UniBas1 (*i.e.,* the baseline experiment), the participants had to perform first a comprehension task on the code base of LaTazza—a small application (see Table I)—and then a modification task on the same code base. Therefore, the comprehension task was performed on source code unfamiliar to the participants, while the modification one was carried out on familiar source code—the participants had got familiar with that source code throughout the previous task.

The baseline experiment was replicated once at the University of Basilicata—UniBas2—and twice at the College of William & Mary—W&M1 and W&M2 (see Table I). Unlike UniBas1, the participants in the replications had to perform either a comprehension task (UniBas2 and W&M1) or a modification task (W&M2) on larger code bases (see Table I). It is worth mentioning that the participants in W&M2, unlike those in UniBas1, carried out the modification task on unfamiliar source code since they had not performed any task on that source code.

Experimental Objects. Table I provides LOC (Lines of Code), number of types (*e.g.,* classes), and number of methods for the applications (*i.e., LaTazza*, *aTunes*, and *LaTeXDraw*) used in the experiments. These applications were all implemented in Java. LaTazza is a desktop application for managing sales and supplies of a coffeemaker. As for aTunes and LaTeXDraw, the former is an open-source cross-platform media-player, while the latter is a graphical drawing editor.

To obtain a version deprived of dead code of each application, dead code (dead methods and classes in particular) was identified and then removed from the original version of the application.

Experimental Tasks. Both comprehension and modification tasks were conceived so that participants, who worked on code bases having dead code, were likely to deal with this smell when performing the tasks. As for the comprehension tasks, the participants were provided with a code base and a comprehension questionnaire, comprising open-ended questions,

TABLE I
SUMMARY OF THE CONTROLLED EXPERIMENTS.

Characteristic	UniBas1 (baseline experiment)	UniBas2 (1st replication)	W&M1 (2nd replication)	W&M2 (3rd replication)
Participants	47 undergraduate students in Computer Science	19 graduate students in Computer Engineering	11 under/graduate students in Computer Science	6 under/graduate students (3 with work experience) in Computer Science
Experimental Object	LaTazza (1,291 LOC, 18 types, 116 methods)[a]	aTunes (42,357 LOC, 778 types, 4,067 methods)[a]	aTunes (42,357 LOC, 778 types, 4,067 methods)[a]	LaTeXDraw (65,320 LOC, 252 types, 3,130 methods)[a]
Investigated RQ/s	RQ3.a, RQ3.b	RQ3.a	RQ3.a	RQ3.b
Experimental Tasks	1) Source code comprehension 2) Source code modification	1) Source code comprehension	1) Source code comprehension	1) Source code modification
Experimenters	Researchers from the University of Basilicata	Researchers from the University of Basilicata	Researchers from the College of William & Mary	Researchers from the College of William & Mary

[a] The code metrics (*e.g.,* LOC) refer to the original version of the application (*i.e.,* the version containing dead code).

TABLE II
SUMMARY OF THE DEPENDENT VARIABLES.

Variable	Construct
CTime	Comprehension Effort
CF CAvg	Comprehension Effectiveness
CηF CηCnt	Comprehension Efficiency
MTime	Modification Effort
MAvg	Modification Effectiveness
MηCnt	Modification Efficiency

on the provided code base. As for the modification task, the participants were asked to implement change requests on the code base we provided them.

Variables. Participants provided with the original code base of LaTazza, aTunes, or LaTeXDraw comprised the *treatment group*, while those provided with the code base deprived of dead code comprised the *control group*. Therefore, *Method* is the main independent variable.

The dependent variables considered in the experiments are reported in Table II. The construct each dependent variable estimates is also reported. For example, both variables CF and CAvg estimate the comprehension effectiveness construct. CTime and MTime are, respectively, the time a participant spent to complete the comprehension task and modification task. Therefore, the lower the values for these variables, the better it is. As for the other dependent variables, the higher the value, the better it is.

Hypotheses Formulation According to the RQ3.a and RQ3.b, the following null hypotheses were formulated:

HN_{CX}. There is not a significant difference in the CX (*i.e.,* CTime, CF, CAvg, CηF, or CηCnt) values when the code base contains or not dead code.

HN_{MX}. There is not a significant difference in the MX (*i.e.,* MTime, MAvg, or MηCnt) values when the code base contains or not dead code.

Design of the Experiments. The design of each experiment was *one factor with two treatments* [18]. That is, the same experimental object (*e.g.,* LaTazza) was used for both treatments (*i.e.,* NoDC and DC) in that experiment. The participants were randomly assigned to the treatments.

Analysis Procedure. To analyze the data concerning the investigation of RQ3.a (*i.e.,* those from UniBas1, UniBas2, and W&M1), we first used descriptive statistics and then inferential statistics, *i.e.,* Linear Mixed Model (LMM) analysis methods.

Given the different objectives between UniBas1 and W&M2 with respect to the investigation of RQ3.b—UniBas1 investigated the effect of dead code on the modifiability of familiar source code while W&M2 focused on unfamiliar source code—, the data from these experiments were analyzed separately. As for UniBas1, we analyzed the data by means of both descriptive and inferential (*i.e.,* unpaired two-sided t-test or Mann-Whitney U test) statistics. As for W&M2, only descriptive statistics are reported—no statistical test was run because of the low number of participants.

For any test of significance, the probability of committing Type-I-error was fixed to 5% (as customary), *i.e.,* $\alpha = 0.05$.

IV. RESULTS AND DISCUSSION

RQ1. *When and why do developers introduce dead code?*

The results indicate that source code fragments can be: *(i)* born dead; *(ii)* made dead or *(iii)* inherited dead. The case *(i)* can occur during the software design and implementation phases: programmers are aware that they have just designed/written dead code, but they plan to use it someday. The case *(ii)* can happen during any software maintenance activity (*e.g.,* adaptive maintenance): developers can be either aware or not that their changes have made some code fragments dead. Finally, the case *(iii)* occurs when developers inherit software systems with dead code from another company.

RQ2. *How do developers perceive and cope with dead code?*

Developers perceive dead code as harmful when comprehending and modifying source code. They also believe that dead code can negatively impact software performances. Nevertheless, it seems that developers consider dead code as a means of anticipating changes and reusing code.

In the management of software projects, no specific work activity is planned to cope with dead code. There is also a

TABLE III
RESULTS CONCERNING THE INVESTIGATION OF RQ3.A.

Variable	Mean		SD		p-value
	NoDC	DC	NoDC	DC	
CTime	51.824	47.047	25.569	22.765	0.252
CF	0.755	0.598	0.181	0.239	**<0.001**
CAvg	0.606	0.409	0.174	0.247	**<0.001**
CηF	0.018	0.017	0.01	0.012	**0.028**
CηCnt	0.075	0.062	0.042	0.057	**0.001**

lack of tool support to detect this smell. Namely, developers are not provided with ad-hoc tools for detecting dead code.

When a developer identifies dead code, the removal of this smell is: *(i)* performed if this operation is low-risk and low-cost; *(ii)* ignored if it is high-risk; and *(iii)* postponed if it is high-cost. When dead code is removed, developers either exploit version control systems to keep track of the performed changes or they comment out dead code. Commenting out dead code causes the introduction of new smells (*i.e.,* commented out code [19]) in source code. When the removal of dead code is ignored or postponed, developers use comments to warn other developers about the presence of this smell.

RQ3.a *Is dead code harmful to comprehend (unfamiliar) source code?*

Table III reports the values of mean and Standard Deviation (SD) for each dependent variable (among those concerning the investigation of RQ3.a) and each treatment (*i.e.,* NoDC or DC). This table also reports the p-values (in bold those less than α) returned by applying the LMM analysis methods.

Comprehension Effort. The descriptive statistic values (see Table III) suggest that there is not a big difference between NoDC and DC with respect to CTime. Indeed, the participants in NoDC spent slightly more time to complete the comprehension tasks. The p-value returned by applying the LMM analysis method seems to suggest that this slight difference is not significant—HN_{CTime} cannot be rejected.

Comprehension Effectiveness. The descriptive statistic values for CF and CAvg suggest that there is a difference between NoDC and DC in favor of NoDC. Confirmations came from the LMM analysis method—both p-values are less than α. Therefore, we can reject both HN_{CF} and HN_{CAvg} and accept the alternative hypotheses. Namely, the presence of dead code significantly penalizes comprehension effectiveness.

Comprehension Efficiency. As for CηF and CηCnt, we can observe that the NoDC values are slightly better than the DC ones. These differences in the values of CηF and CηCnt are also significant since both $HN_{C\eta F}$ and $HN_{C\eta Cnt}$ can be rejected. Therefore, the presence of dead code significantly penalizes comprehension efficiency.

Summary. The presence of dead code significantly worsens comprehension of unfamiliar code.

RQ3.b. *Is dead code harmful to perform modification/change tasks on (familiar and unfamiliar) source code?*

Table IV provides the values of mean and SD, for each dependent variable and each treatment, grouped by familiarity with the code base (*i.e.,* familiar code base or unfamiliar code base). This table also provides the p-values (in bold those less than α) returned by the tests of significance (if applied).

TABLE IV
RESULTS CONCERNING THE INVESTIGATION OF RQ3.B. WHEN THE CODE BASE IS FAMILIAR OR UNFAMILIAR.

Code base	Variable	Mean		SD		p-value
		NoDC	DC	NoDC	DC	
Familiar	MTime	52.95	51.667	8.153	12.719	0.575
	MAvg	0.54	0.533	0.216	0.235	0.893
	MηCnt	0.054	0.054	0.028	0.031	0.73
Unfamiliar	MTime	153	181.667	92.715	59.911	-
	MAvg	0.417	0.333	0.289	0.144	-
	MηCnt	0.016	0.009	0.016	0.007	-

Modification Effort. The descriptive statistic values (see Table IV) for MTime suggest that there is not a notable difference between NoDC and DC when the participants are familiar with the source code. Moreover, the test of significant does not allow rejecting HN_{MTime}.

Conversely, we can observe that, when working on unfamiliar source code, the participants in DC spent, on average, much more time than the others.

Modification Effectiveness. As for MAvg, the values of the descriptive statistics for NoDC and DC are very similar one another when the source code is familiar. Moreover, the tested null hypothesis (*i.e.,* HN_{MAvg}) cannot be rejected.

When the source code is unfamiliar, we can observe that the participants in NoDC were, on average, more effective in modifying source code than those in DC.

Modification Effectiveness. Similar to MTime and MAvg, when the source code is familiar, the descriptive statistic values for MηCnt do not indicate a huge difference between NoDC and DC. Moreover, the p-value returned by the test of significant does not allow rejecting $HN_{M\eta Cnt}$.

When the participants were unfamiliar with the source code, it seems that the participants in NoDC were more efficient in performing the modification task with respect to those in DC.

Summary. The results seem to indicate that dead code is not harmful in cases where developers have to accomplish modification tasks on familiar code bases, while dead code appears to be harmful in cases where developers dealt with modification tasks on source code which is new for them.

Moreover, we performed a further analysis to verify if the participants modified existing dead code when accomplishing the modification tasks. The results indicate that some participants (independently if the source code was familiar or not) wasted time modifying existing dead code while implementing the change requests and it seems that they were not aware of modifying dead code. When this happened, the participants were usually not capable of correctly implementing the assigned change requests.

V. SUPPORTING DEAD CODE DETECTION

Approaches. To help developers to refactor code bases, two approaches for detecting dead code were proposed and then implemented in two tools. Both approaches were conceived

TABLE V
RESULTS ON THE DETECTION OF DEAD METHODS.

Statistic	Variable	DCF	DUM-Tool	JTombstone	CodePro
Mean	precision	0.84	0.76	0.36	0.16
	recall	0.87	0.88	0.95	0.64
	f-measure	0.85	0.79	0.43	0.23
SD	precision	0.17	0.23	0.38	0.15
	recall	0.07	0.09	0.07	0.2
	f-measure	0.11	0.17	0.34	0.2

to detect dead methods in applications written in Java. The former approach was named *DUM*. To detect dead methods, DUM first builds a graph-based representation of the application being analyzed, by providing an approximation of virtual method calls, and then traverses this graph-based representation. DUM was implemented in an Ecplise plug-in named DUM-Tool. The latter approach was proposed to overcome some limitations of DUM. To this end, it relies on *RTA*—an algorithm for call graph construction, which is known to well approximate virtual method calls and to be fast [20]—to detect dead methods. This approach was implemented in a command-line tool named DCF.

Assessment. Both DUM-Tool and DCF (and thus their underlying approaches) were empirically assessed on four Java applications and compared with other tools for dead method detection: *JTombstone*[2] and *CodePro AnalytiX*.[3] To carry out the empirical assessment, DUM-Tool and DCF were run on each of the four applications and the detected dead methods were gathered. The baseline tools (*i.e., JTombstone* and *CodePro AnalytiX*) were run in similar fashion. Then the dead methods detected by each tool, on a given application, were compared with an oracle[4] in order to assess the detection of dead methods. Three constructs were estimated:

- **Correctness.** It reflects the fact that methods detected as dead are really dead.
- **Completeness.** It reflects how much the set of methods detected as dead is complete with respect to the total number of dead methods.
- **Accuracy.** It reflects both correctness and completeness.

Correctness and completeness were estimated through the *precision* and *recall* measures [21], respectively. To estimate accuracy, the balanced *f-measure* [21] of precision and recall was used.

Results. Table V reports, for each tool, mean and SD for precision, recall, and f-measure. The results in this table suggest that the detection of DCF is correct, complete, and accurate since DCF shows, on average, high values of precision, recall, and f-measure. The detection of DUM-Tool seems to be quite correct, complete, and accurate too, even though it is slightly worse than the detection of DCF (see precision and f-measure). By comparing DCF and DUM-Tool with the baseline tools, we can observe that they both outperform the baselines in terms of correctness and accuracy. JTombstone outperforms both DCF and DUM-Tool on completeness but at the cost of little correctness. Summing up, DUM-Tool and DCF are both good solutions for developers, who need to remove dead methods. However, DCF seems to be more accurate (and precise) in the detection of dead methods.

[2]jtombstone.sourceforge.net

[3]marketplace.eclipse.org/content/codepro-analytix

[4]An oracle contains which methods of a given application are dead or not.

VI. CONCLUSION AND FUTURE WORK

This post-doctoral track paper shows the main findings of a multi-study investigation into dead code, which was conducted to study *when* and *why* developers introduce dead code, *how* they perceive and cope with it, and *whether* dead code is harmful. The most important findings emerged from this investigation are:

- **Lesson 1.** *Having source code deprived of dead code would help improving its comprehensibility and modifiability.* This might also reduce a number of issues related to well-known software engineering processes: maintenance, testing, quality assurance, reuse, and integration [22]. The researchers should provide developers with supporting tools for the detection (and removal) of dead code so as to have source code deprived of this smell. A first step towards this direction has been done thanks to DUM-Tool and DCF.
- **Lesson 2.** *Although developers perceive dead code as harmful, it seems to be considered a sort of reuse means.* The question now arises of whether such dead code is then re/used or it is simply a waste of time and a danger for source code comprehensibility and modifiability.
- **Lesson 3.** *Dead code is not always adequately removed: commenting out dead code appears to be a common bad practice.* Practitioners should be warned when they comment out dead code by remembering them to give dead code a decent burial (*i.e.,* dead code removal) [19].

Based on the findings from the multi-study, the next step should be to evangelize *dead-code-free* source code. This evangelization should aim to: *(i)* shape a new generation of developers, who take care with dead code; and *(ii)* convert professional developers, who currently do not worry about this smell, to dead-code-free source code. In the case *(i)*, the evangelization would start from the university through lessons for students on dead code. In the case *(ii)*, the means for the evangelization would be webinars and seminars for professional developers on dead code. Moreover, it would interesting to conduct studies on the evolution of dead code with the goal to understand if dead code is later revived or not.

In closing, I would like to share some advice for *novice* PhD students in SE based on my experience:

- **Advice 1.** *Share your ideas.* Other researches can give you suggestions on how to improve your ideas and realize them.
- **Advice 2.** *Have a look at research fields different from SE.* Medicine and psychology papers can be useful if you are interested in conducting quantitative or qualitative studies.

ACKNOWLEDGMENT

I would like to thank Giuseppe Scanniello, my PhD advisor, for guiding me throughout my PhD program.

REFERENCES

[1] S. Romano, "Dead code: Study and detection," Ph.D. dissertation, University of Salento and University of Basilicata, 2018. [Online]. Available: http://www2.unibas.it/sromano/downloads/thesis.pdf

[2] W. C. Wake, *Refactoring Workbook*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2013, pp. 116–125.

[4] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[5] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proceedings of the 19th International Conference on Software Maintenance*. IEEE, 2003, pp. 381–384.

[6] S. Eder, M. Junker, E. Jrgens, B. Hauptmann, R. Vaas, and K. H. Prommer, "How much does unused code matter for maintenance?" in *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 2012, pp. 1102–1111.

[7] H. Boomsma, B. V. Hostnet, and H. G. Gross, "Dead code elimination for web systems written in php: Lessons learned from an industry case," in *Proceedings of the 28th International Conference on Software Maintenance*. IEEE, 2012, pp. 511–515.

[8] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Proceedings of the 20th Working Conference on Reverse Engineering*. IEEE, 2013, pp. 242–251.

[9] G. Scanniello, "Source code survival with the kaplan meier," in *Proceedings of the 27th International Conference on Software Maintenance*. IEEE, 2011, pp. 524–527.

[10] ——, "An investigation of object-oriented and code-size metrics as dead code predictors," in *Proceedings of the 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014, pp. 392–397.

[11] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, "A multi-study investigation into dead code," *IEEE Trans. Softw. Eng.*, 2018, in press.

[12] ——, "Are unreachable methods harmful? results from a controlled experiment," in *Proceedings of the 24th International Conference on Program Comprehension*. IEEE, 2016, pp. 1–10.

[13] S. Romano, G. Scanniello, C. Sartiani, and M. Risi, "A graph-based approach to detect unreachable methods in java software," in *Proceedings of the 31st Symposium on Applied Computing*. ACM, 2016, pp. 1538–1541.

[14] S. Romano and G. Scanniello, "Exploring the use of rapid type analysis for detecting the dead method smell in java code," in *Proceedings of the 44th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2018, in press.

[15] ——, "Dum-tool," in *Proceedings of the 31st International Conference on Software Maintenance and Evolution*. IEEE, 2015, pp. 339–341.

[16] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1–11.

[17] N. King, "Using templates in the thematic analysis of text," in *Essential Guide to Qualitative Methods in Organizational Research*, C. Cassell and G. Symon, Eds. Sage, 2004, pp. 256–270.

[18] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[19] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[20] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 2000, pp. 281–293.

[21] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[22] G. Canfora and M. Di Penta, "New frontiers of reverse engineering," in *Workshop on the Future of Software Engineering*. IEEE, 2007, pp. 326–341.