TU Delft

Delft University of Technology

Beyond Technical Aspects
How Do Community Smells Influence the Intensity of Code Smells?

Palomba, Fabio; Tamburri, Damian Andrew; Arcelli Fontana, Francesca; Oliveto, Rocco; Zaidman, Andy; Serebrenik, Alexander

DOI
10.1109/TSE.2018.2883603

Publication date
2021

Document Version
Final published version

Published in
IEEE Transactions on Software Engineering

Citation (APA)
Palomba, F., Tamburri, D. A., Arcelli Fontana, F., Oliveto, R., Zaidman, A., & Serebrenik, A. (2021). Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells? *IEEE Transactions on Software Engineering*, *47*(1), 108-129. Article 8546762. https://doi.org/10.1109/TSE.2018.2883603

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Green Open Access added to TU Delft Institutional Repository

'You share, we take care!' - Taverne project

https://www.openaccess.nl/en/you-share-we-take-care

Otherwise as indicated in the copyright section: the publisher is the copyright holder of this work and the author uses the Dutch legislation to make this work public.

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE*, Damian Andrew Tamburri, *Member, IEEE*, Francesca Arcelli Fontana, *Member, IEEE*, Rocco Oliveto, *Member, IEEE*, Andy Zaidman, *Member, IEEE*, and Alexander Serebrenik, *Senior Member, IEEE*

Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

Index Terms—Code smells, organizational structure, community smells, mixed-methods study

1 Introduction

Software engineering is, by nature, a "social" activity that involves organizations, developers, and stakeholders who are responsible for leading to the definition of a software product that meets the expected requirements [1]. The social interactions among the involved actors can represent the key to success but can also be a critical issue possibly causing additional project costs from an organizational and socio-technical perspective [1], [2].

In the recent past, the research community devoted effort to understanding so-called *social debt* [3], which refers to the presence of non-cohesive development communities whose members have communication or coordination issues that make them unable to tackle a certain development problem and that can lead to unforeseen project cost. One of the recent advances in this research field is represented by the definition of *community smells*, which were defined by Tamburri et al. [2], [4] as a set of socio-technical characteristics (e.g., high formality) and patterns (e.g., repeated condescending behavior, or rage-quitting), which may lead to the emergence of social debt. From a more actionable and analytical perspective, community smells are nothing more than *motifs* over a graph [5]; motifs are recurrent and statistically significant sub-graphs or patterns over a graph detectable using either the structural properties and fashions of the graph or the graph salient features and characteristics (e.g., colors in the case of a colored graph). For example, the *organizational silo effect* [4] is a recurring network substructure featuring highly decoupled community structures.

In turn, community smells are often connected to circumstances such as technical debt [6], i.e., the implementation of a poor implementation solution that will make the maintainability of the source code harder.

In this paper we aim at empirically exploring the relation between social and technical debt, by investigating the connection between two noticeable symptoms behind such types of debt: community and code smells. The latter refer to poor implementation decisions [7] that may lead to a decrease of maintainability [8] and an increase of the overall project costs [9].

We conjecture that the presence of community smells can influence the persistence of code smells, as the circumstances reflected by community smells (e.g., lack of communication or coordination between team members) may lead the code to be less maintainable, making code smells worse and worse over time.

- F. Palomba is with the University of Zürich, Zürich 8006, Switzerland. E-mail: palomba@ifi.uzh.ch.
- D.A. Tamburri and A. Serebrenik are with Eindhoven University of Technology, Eindhoven 5612 AZ, The Netherlands. E-mail: {d.a.tamburri, a.serebrenik}@tue.nl.
- F.A. Fontana is with University of Milano Bicocca, Milano 20126, Italy. E-mail: arcelli@unimib.it.
- R. Oliveto is with University of Molise, Campobasso 86100, Italy. E-mail: rocco.oliveto@unimol.it.
- A. Zaidman is with Delft Unversity of Technology, Delft 2628 CD, The Netherlands. E-mail: a.e.zaidman@tudelft.nl.

Manuscript received 3 Feb. 2018; revised 10 Nov. 2018; accepted 19 Nov. 2018. Date of publication 27 Nov. 2018; date of current version 8 Jan. 2021. (Corresponding author: Fabio Palomba.) Recommended for acceptance by A. Garcia. Digital Object Identifier no. 10.1109/TSE.2018.2883603

0098-5589 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

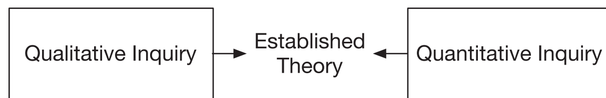

Fig. 1. Convergence Mixed-Methods, qualitative and quantitative inquiry converge towards a confirmed theory [10], [11], [12].

Our empirical investigation features a convergence mixed-methods approach [10], [11], [12] (see Fig. 1) where quantitative and qualitative research are run in parallel over the same dataset with the goal of converging towards theoretical saturation. As mentioned, the theoretical saturation in question is to be achieved via mixed-methods convergence; our assumption is therefore that, if both qualitative and quantitative data lead to the same conclusions, then our theory is saturated. Conversely, any disagreement between the two theories would lead us to an improved version of our initial theory that code and community smells are somehow connected.

Following this research method, our *qualitative* investigation features a survey of 162 original developers of 117 releases belonging to 9 Apache and Eclipse systems in order to answer the research question below:

- *RQ1: What concerns affect the developers' decision to eliminate or preserve code smells?*

In other words, we aim at eliciting possible reasons for making developers decide whether to remove code smells; the purpose of this study is understanding whether community-related issues might influence developer decisions to retain or remove a code smell. This study is motivated by the fact that developers, even if they perceive code smells as implementation problems, are not inclined to remove them by performing refactoring operations [13], [14], [15], [16], [17]. The survey findings confirmed our initial hypothesis, as over 80 percent of practitioners explicitly mention that avoiding community problems (e.g., repeated disagreements) is the reason why code smells are *not* refactored. This means that *it is more convenient to keep a technical smell than deal with a community smell*. Thus, the survey findings highlighted that the persistence of code smells not only depends on technical factors studied by past literature [8], [18], [19], but also on *the other fish of the sea*, i.e., additional aspects related to the social debt occurring in software communities that have not been studied yet.

In parallel with the qualitative inquiry, we *quantitatively* evaluate to what extent the community-related factors measured over the 9 projects in our dataset impact code smell intensity [20], [21], i.e., an estimation of the severity of a code smell:

- *RQ2: To what extent can community smells explain the increase of code smell intensity?*
- *RQ3: Does a community-aware code smell intensity prediction model improve the performance of models that do not consider this information?*

We present a novel code smell intensity prediction model that explicitly considers community-related factors when predicting the future intensity of code smells, with the aim of providing developers and project managers with a practical technique that would allow them to preventively take actions that preserve the maintainability of the source code (e.g., refactoring of the team composition). To this aim, we systematically investigate the relationship between *all* the automatically detectable community smells [3], [4], i.e., *Organizational Silo*, *Black Cloud*, *Lone Wolf*, and *Bottleneck*, and five code smells, i.e., *Long Method* [7], *Feature Envy* [7], *Blob* [22], *Spaghetti Code* [22], and *Misplaced Class*. All these code smells turned out to be relevant from the developers' perspective in our survey study. As a result, we found that a code smell intensity prediction model built using community smells is able to more accurately predict the future intensity of code smells than a model that does not explicitly consider the status of software communities. The accuracy of the devised prediction model is also confirmed by ten industrial project managers, who were surveyed to qualitatively assess the latent relation between the community and code smells considered by our model.

Contributions and Implications. In summary, the original research contributions of this article are:

- A large survey study involving 162 practitioners aimed at analysing the reasons why code smells are not refactored;
- As a side effect of the survey study, we reveal the existence of 4 previously unknown community smells;
- A large-scale quantitative study where we assess the impact of community-related information on the performance of a code smell intensity prediction model.
- A comprehensive replication package, containing anonymised qualitative and quantitative data used in our study [23].

Our study has relevant implications for researchers, practitioners, and tool vendors:

(1) Our findings represent a call for *community-aware* software evolution techniques, that explicitly consider community-related factors to recommend practitioners how to evolve their code. Thus, both researchers and tool vendors should take into account those aspects when developing new tools;
(2) Practitioners should carefully monitor the evolution of software communities, and, emergence of community smells. If needed, practitioners should take preventive actions;
(3) Our study promotes a *comprehensive research approach* for software maintenance and evolution: indeed, we show that the circumstances occurring within the development community directly affect the way developers act in the source code. Thus, our study encourages researchers to take into account community-related aspects when studying the underlying dynamics of software evolution.

Structure of the Paper. Section 2 introduces the terminology we use in the paper, while Sections 3 and 4 outline our research design and results for our research questions. Section 5 outlines a theoretical convergence between the two sides of our study, while Section 6 address the threats to validity we detected and addressed. Section 7 outlines related work. Finally, Section 8 concludes the paper.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

2 Terminology

Our study aims at understanding the role of *community smells* as factors contributing to the persistence of code smells. The aforementioned concepts are defined as follows.

A *software development community* is a specific type of social network upon which certain properties hold constantly (e.g., informal communication across electronic channels of open-source projects) [24], [25]. Across such development, social networks and their many possible properties (e.g., informality, goals, membership selection, intercommunication protocols), communities can develop conditions that potentially lead to socio-technical problems. Such conditions have been defined as *community smells* [2], [4] in analogy to code smells. The analogy signifies that, on the one hand, community smells do identify unlikable circumstances (e.g., the lack of communication across different modules of a software system), but, on the other hand, these conditions do not necessarily stop or void the organizational behavior across the community. Rather, they prove detrimental and cause additional project cost in several possible ways (e.g., recurrent delays in communication, wrongful knowledge sharing) [4]. Finally, with the term *project*, we identify the goal or shared practice that the community maintains as its central endeavor, e.g., the Apache Spark community holds the delivery of the Apache Spark product as its key *project*. Specifically to the context of a project, on the one hand social debt indicates the accumulated effect of problematic organizational conditions (e.g., community smells); on the other hand, technical debt [26], notably refers to the additional project cost connected to problematic technical conditions, represented in our study by *code smells* [7], namely poor design or implementation solutions applied by programmers during the development of a software product.

3 Surveying Software Developers

The *goal* of this study is to elicit possible reasons making developers decide whether to remove code smells, with the *purpose* of understanding if any community-related issue (e.g., a community smell) might influence their decisions. The specific research question targeted with the qualitative study is the following:

- *RQ1: What concerns affect the developers' decision to eliminate or preserve code smells?*

3.1 Context of the Study

The *context* of the study is represented by 117 major releases of 9 large open-source projects belonging to two software ecosystems, i.e., Apache and Eclipse. Table 1 reports the list of systems in the dataset along with their (i) number of commits in the observed time period, (ii) number of developers, and (iii) size as minimum-maximum number of classes and KLOCs in the considered time period. The selection of these systems was driven by our willingness to analyse projects presenting different (a) codebase size, (b) longevity, (c) activity, and (d) population. Starting from the list of projects for the two ecosystems, we randomly selected 9 of them having a number of classes higher than 500, with a change history at least 5 years long, having at least 1,000 commits, and with a number of contributors higher than 20. We used reference sampling thresholds from literature [27], [28], as they allowed us to focus on large and very active projects having a notable amount of contributors: this is essential to observe the presence of both community smells (very small communities are likely to have less organizational issues) and code smells (small systems contain less code smell instances [8]).

TABLE 1
Software Projects in Our Dataset

System	#Commits	#Dev.	#Classes	KLOCs
Apache Mahout	3,054	55	800-813	202-204
Apache Cassandra	2,026	128	546-586	102-111
Apache Lucene	3,784	62	5,187-5,506	131-142
Apache Cayenne	3,472	21	2,727-2,854	518-542
Apache Pig	2,432	24	824-826	351-372
Apache Jackrabbit	2,924	22	842-872	473-527
Apache Jena	1,489	38	646-663	187-231
Eclipse CDT	5,961	31	1,404-1,415	189-249
Eclipse CFX	2,276	21	651-655	98-106
Overall	**32,889**	**436**	**546-5,506**	**98-542**

As for code smell types, we investigated:

(1) *Long Method:* a method that implements more than one function, being therefore poorly cohesive [7];
(2) *Feature Envy:* a method which is more interested in a class other than the one it actually is in, and that should be moved toward the envied class [7];
(3) *Blob Class:* a class usually characterised by a high number of lines of code, low cohesion, and that monopolises most of the systems's processing [22];
(4) *Spaghetti Code:* a class without a well-defined structure, usually declaring many long methods [22];
(5) *Misplaced Class:* a class that has more relationships with a different package than with its own package [9];

The choice of focusing on these smells was driven by the desire to understand how "eliminate or preserve" decisions are made for different types of code smells (e.g., highly complex classes like *Blob* or source code violating OOP principles like *Feature Envy*) having different granularities (e.g., method-level smells like *Long Method* or class-level like *Blob*).

3.2 Detecting Code Smells

The first step to answer *RQ1* is concerned with the automatic detection of the code smells considered. To this aim, we relied on DECOR [29]. The tool uses a set of rules, called "rule cards",[1] describing the intrinsic characteristics that a class has when affected by a certain smell type. For instance, the approach marks a class as a *Blob* instance when it has an LCOM5 (Lack of Cohesion Of Methods) [30] higher than 20, a number of methods and attributes higher than 20, a name that contains a suffix in the set {*Process*, *Control*, *Command*, *Manage*, *Drive*, *System*}, and it has an one-to-many association with data classes.

Among the code smell detection tools available in the literature [31], [32], [33], [34], [35], [36], we selected DECOR

1. http://www.ptidej.net/research/designsmells/

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

because it has been employed in previous investigations on code smells demonstrating good performance in terms of precision, recall, and scalability [18], [37], [38], [39], [40], [41], [42]. Overall, DECOR identified 4,267 code smell instances over the 117 considered releases, i.e., a mean of ≈36 instances per release. A report of the distribution of each code smell type is available in our online appendix [23].

To verify that the tool was actually suitable for this study, we also validated the performance of DECOR on two of the systems in the dataset, i.e., CASSANDRA and LUCENE. In particular, we compared the recommendations provided by the tool with a publicly available oracle reporting the actual code smell instances affecting the systems [43]. As a result, we found that the average precision of the tool was 79 percent, with a recall of 86 percent. Based on these results, we can claim that the selected code smell detector has a performance similar the one declared in previous studies [29], being sufficiently accurate for conducting our study.

3.3 RQ1. Survey Design & Data Analysis

The goal of the survey was twofold: (i) to help practitioners by highlighting the code smells in their code, and (ii) to elicit the data we needed by questioning them on each detected code smell, asking for comments and explanations over that smell, as well as an elaboration over the reasons why the smell was not addressed yet.

We opt for a limited, cognitively simple set of questions, to promote responses by reducing the amount and cognitive complexity of questions posed while increasing the developers' immediate benefit (in our case, by raising awareness over a code problem) [44], [45]. The following list of mandatory questions was selected for our inquiry:

1) *Were you aware of this code smell?*
2) *Can you think of any technical root causes for the smell?*
3) *What are the reasons or risks that lead you to decide whether or not to refactor the smell?*

To survey developers that actually have knowledge on social and technical circumstances around a smelly file, we decided to focus on the developers that worked on a smelly class the most (in terms of commits). Thus, we contacted 472 developers that worked with one distinct smelly class instance in any of the releases we considered. It is worth noting that we excluded developers that worked with more than one smelly class. The rationale here is that developers who worked on several smelly classes might potentially not really be focused on the history of one specific smelly class, e.g., they might have confused situations appearing in the context of another smelly class with those of the class we were targeting. To avoid any possible bias, we preferred to be conservative and exclude them from the target population of our survey: all in all, we discarded 168 developers.

Being aware of ethical issues commonly associated with empirical software engineering studies, such as confidentiality and beneficence, we adhered to the inquiry guidelines of Singer and Vinson [46]. As such, we prepared an introductory text and clarified the anonymity of their responses. To bootstrap the survey, we used bulk-emailing and email auto-compose tools, posing care in not spamming any participant—every single developer was never contacted more than once. As a result, 162 developers responded out of the 472 contacted ones, for a response rate of 34, 32 percent—that is almost twice than what has been achieved by previous papers (e.g., [41], [47], [48]). In our opinion, there are three aspects that have contributed to this relatively high response rate: (1) we contacted developers that committed the highest number of changes to a smelly class: this means that we only targeted developers who were expert of the considered classes and that might have been more interested in gathering further information on the class they were mainly in charge of; (2) looking at the overall number of commits, we noticed that the involved developers are among the most active in the project; and (3) the time and amount of information required from developers were limited, in order to encourage them to reply to our e-mails.

It is worth highlighting that, given the methodology followed to recruit survey participants, we did not collect detailed information on the profiles of our interviewees. For instance, we did not collect data on their programming experience. However, this does not represent a threat in our case. Indeed, we were interested in surveying developers that actually worked on code smell instances, so that we could ask the reasons why they did not refactor them: in this sense, as far as they concretely worked on code smells, it is fine for the type of questions we posed. At the same time, while developers experience may have played a role in the answers they provided, looking at the overall number of commits, the involved developers are among the most active in the considered projects: thus, we could assume that they are among the most expert ones for the considered projects.

Concerning data analysis, given the exploratory nature of *RQ1*, we applied Straussian Grounded Theory [49] as follows: (i) *microanalysis*—we labelled survey responses, applying a single label per every piece of text divided by standard text separators (comma, semicolon, full-stop, etc.); (ii) *categorisation*—we clustered labels which were semantically similar or identical, i.e., applying the semantic similarity principle [50], a direct consequence of this step is the renaming of labels to reflect categories of labels; (iii) *categories saturation*, i.e., elaboration of *core-categories*—this step entails continued addition of labels to other or new core-categories until no uncategorised label remained; (iv) *taxonomy building*—we represented the tree of categories and labels to visualise the grounded-theory extracted from our survey responses. Indeed, our choice for Straussian Grounded-Theory is more appropriate for explorative contexts since it does not assume the presence of any previous theory to be tested over the data but rather it adopts a constructivist theory-proving approach wherefore a theory is directly and purely generated from the data. This not withstanding, to increase inter-rater reliability, two authors independently coded the dataset, subsequently evaluating coding agreement, via the Krippendorff's alpha Kr_α [51]. Agreement measures to 0.87, considerably higher than the 0.80 standard reference score [52] for Kr_α.

3.4 RQ1. Analysis of the Results

Over 80 percent of the practitioners admitted being aware of the problems we discovered, sometimes also highlighting that the problems were indeed *well-known* across the

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

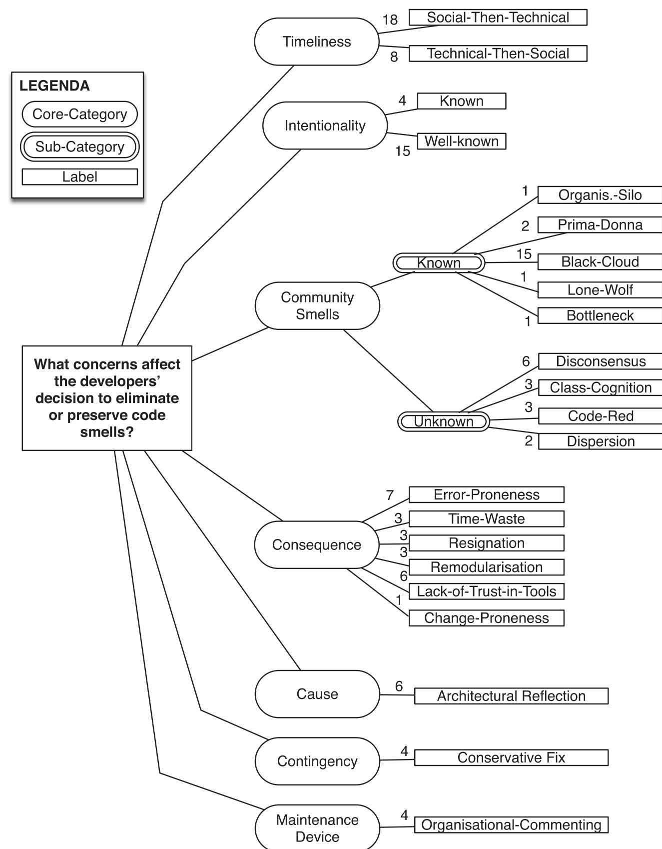

Fig. 2. A grounded-theory of concerns affecting the developers' decision to eliminate or preserve code smells, edges on the leaves represent concept frequency counts.

community. It is important to note that, although a complementary "unknown" category was present, it was *never* applied, since developers were always aware or well aware of the problems we highlighted. This latter, however, could be due to a confirmation bias—respondents might have felt uncomfortable admitting that they were not well aware of problems in their code. The output of our data analysis process, summarizing the evidence from the developer survey, is shown in Fig. 2. Generally speaking, developers highlighted the presence of some well-known factors that lead them to avoid refactoring. For instance, the fear of introducing defects while modifying the structure of a system as well as the lack of automated solutions to perform refactoring have been frequently reported by our interviewees. This indicates the presence of important technical "barriers" that do not allow developers to promptly improve the quality of source code. At the same time, our participants pointed out some interesting observations that confirm our hypotheses on the role of community smells and, more in general, community-related aspects on the persistence of code smells. In particular, our findings reveal not only that previously known community smells represent an important factor in the refactoring decisional process, but also that there are further community smells that were unknown up to now but that influence the way developers act on code smells. Table 2 provides an overview of the smells we observed.

In the following sections we discuss our findings, focusing on: (a) the community smells that were re-confirmed, meaning that they were previously reported in industry and are re-appearing in open-source as well; (b) newly-emerging community smells, meaning the smells that were never previously observed in industry; (c) other aspects and theoretical underpinnings around community- and technical-related factors.

3.4.1 Re-Confirmed Community Smells

The first finding that emerged from the survey analysis is that community smells [4], i.e., symptoms of the presence of social issues within a software development community, represent one important factor leading developers to not spend time in eliminating code smells: 80 percent of practitioners explicitly mentioned that avoiding community

TABLE 2
Community Smells from Our Survey, an Overview

Community Smell	Definition	#
Prima Donna [2], [4]	Repeated condescending behavior, superiority, constant disagreement, uncooperativeness by one or few members.	7
Black Cloud [2], [4]	Swarming of email or other communication around a new design or refactoring exercise—overly complex and disagreeing repeated communication obfuscates actual truth.	15
organizational Silo [2], [4]	Siloed areas of the development community that do not communicate, except through one or two of their respective members.	1
Lone Wolf [2], [4]	Defiant contributor who apply changes in the source code without considering the opinions of her peers.	1
Bottleneck [2], [4]	One member interposes herself into every interaction across sub-communities	1
Dissensus *new*	Developers cannot reach consensus w.r.t. the patch to be applied—same condition recurs for other patches in other very complex areas of the code	6
Class Cognition *new*	The affected class, if refactored, would be made significantly more complex to discourage further intervention and introducing a massive overhead to newcomers and other less-experienced contributors	3
Dispersion *new*	A fix in the code causes a previously existing group or modularised collaboration structure in the community to split up or rework their collaboration because functionality becomes re-arranged elsewhere	2
Code Red *new*	This smell identifies an area of code (a class + immediately related ones) which is so complex, dense, and dependent on 1-2 maintainers who are the only ones that can refactor it	2

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

problems (e.g., repeated disagreements) and other social "smells" [4], is the reason why code smells are *not* addressed, meaning that it is more convenient to keep a technical smell than dealing with a community smell. More specifically, we could confirm the recurrence of five previously known community smells such as *Black Cloud* (mentioned 15 times), *Prima Donna* (2), *Organizational Silo* (1), *Lone Wolf* (1), and *Bottleneck* (1). The participants reported that increasingly confusing information sharing and communication is one of the most prominent reasons why they avoid refactoring (i.e., the *Black Cloud* smell). Furthermore, repeated uncooperative, condescending, or even defiant behavior with respect to technical or organizational arrangements in a community by a single member (the *Prima Donna* effect) can motivate them to prefer avoiding any type of restructuring for the fear of introducing additional chaos in the community.

A smaller number of developers also reported how the presence of sub-teams that do not communicate with each other (the *Organizational Silo*) or the absence of communication with one of the members who prefer working independently from the others (the *Lone Wolf*) can explain their refactoring decisions. Finally, the absence of flexibility in the community—indicated by the presence of a member that tries to interpose herself into every formal communications (the *Bottleneck*)—make that developers are not always aware of the design decisions made by other people in the community and, for this reason, they sometimes avoid restructuring to not introduce defects and/or worsening the overall program comprehensibility.

An interesting example of the discussion made so far is presented in the quotation below, where a developer commented on an instance of *Long Method* that was not refactored:

> *"We are aware that this code is problematic, but we have neither time and tools to correctly perform a splitting. Furthermore, we have two subteams working on it, and the communication with the other subteam is not good."*

Besides explaining that the lack of tools and time are important factors in the decisional process, the developer clearly pointed out the presence of an *Organizational Silo* that involves two sub-teams that cannot properly communicate with each other. As a consequence, the developer preferred to avoid any type of pervasive modification which may have led to introduce additional problems. All in all, the results presented and discussed so far can already confirm our conjecture: community smells can influence the persistence of code smells.

3.4.2 Newly-Emerging Community Smells

Besides the five known community smells, our data indicates the existence of 4 *previously unknown* community smells recurring at least twice in two different projects, and reported by two different developers. For example, we discovered in 3 different projects that developers repeatedly manifested a previously unknown *Dissensus* community smell, namely, inability to achieve consensus on how to proceed despite repeated attempts at it—as a consequence, the code smell was kept as-is. For instance, a developer reported that:

> *"Yes, we know this problem. But every time we talk about it, we are not able to find a common solution."*

Note that this smell is not completely unknown in organizational literature: indeed Bergman et al. [53] indicate that social conflict is associated with reduced productivity and inability to reach consensus.

Our results also indicate that in all projects targeted by our survey, practitioners often did not refactor code smells since refactoring would cause a previously unknown *Class Cognition* community smell, namely, that refactoring would cause the modular structure and refactored classes to be more difficult to understand and contribute to [54], e.g., for newcomers. This is the case of a developer who analyzed a *Feature Envy* instance reporting:

> *"Generally I try not to perform re-organization of the code that implies the modification of the location of code components. This because (i) other developers could waste time and effort in understanding the new environment of the method, and (ii) I cannot simply identify a suitable new location for the code."*

Thus, she indicated that the re-location of the method could have caused comprehensibility issues to other developers. The two smells discussed above were *intuitively, but precisely* described by 6 and 3 distinct developers. In addition, we revealed the existence of the *Code-red* community smell, that denotes the existence of extremely complex classes that can be managed by 1-2 people at most. As an example, one of the participants who analyzed a *Blob* instance explicitly reported that:

> *"Totally un-understandable code is difficult to touch. I modified this class only for fixing a potential bug, but generally only 1 or 2 devs can substantially modify it."*

Finally, we found the *Dispersion* community smell, which concerns a fix or refactoring that caused a previously existing group of modularised collaboration to fragment and work haphazardly because of functionality rearrangements. In contrast to the *Class Cognition* community smell, this smell has nothing to do with code understandability [55] to newcomers, but rather it refers to making normal maintenance activities in the community more difficult to carry out and coordinate. To better explain the nature of this smell, let us consider the following quote from one of our surveyed developers:

> *"If the algorithm implemented in the method would be split, then the developers working on that code would become crazy since they are able to work pretty well on the existing code."*

In this case, the developer was analyzing a *Long Method* instance but they did not proceed with an *Extract Method* refactoring in order to avoid the risk of other team members losing their knowledge on the algorithm implemented in the method, threatening its future reliability.

In conclusion, we can observe how all the newly emerging smells are socio-technical, i.e., blend together social and technical aspects, which confirms the need for further quantitative analysis and exploration of the mutual relation between code and community smells. It is worth mentioning that developers were not made aware of the notion of

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

community smells and spontaneously, intuitively expressed the repeated community characteristics causing or relating to code smells—this intuition, feeling of unease, is by definition the indication of a community smell [4].

3.4.3 Additional Aspects Influencing Refactoring Decisions

While most of the developers directly pointed out community smells as one of the main reasons leading them to avoid refactoring of code smells, some participants also indicated the existence of additional aspects that impact on their refactoring decisions. Specifically, our data indicates that one of the most common reasons to avoid refactoring is the fear of (i) wasting time or (ii) the technical consequences of this action. Specifically, 7 developers pointed out the risks to introduce new defects while performing refactoring, thus confirming the findings by Kim et al. [56], who reported that developers do not think of refactoring as a behavior-preserving activity and, as a consequence, it may introduce new defects in the codebase. At the same time, 6 developers identified the lack of trust in the refactoring tools as the main cause to not remove code smells—this is, again, in line with previous findings in the field [56], [57]. Interestingly, one developer reported that a co-occurring aspect to consider when removing a code smell is whether the class also contains a clone: in this case, the refactoring would be much more costly as other code clones should be checked for the presence of the smell and eventually refactored.

Still in the context of technical factors, ≈ 10 percent of the respondents elaborated on the perceived technical mediators for unresolved code smells, pointing to well-known *architectural reflection* phenomena, such as architecture erosion or architecture drift [58]. They also pointed out that a high number of dependencies toward other classes can be an important reason to avoid a refactoring action.

Furthermore, developers are often scared of one key *contingency*, that is, modifying classes which are subject of both code and community smells—refactoring these classes is avoided or limited to *conservative-fix* only. Finally, our data also indicates that developers devised a new *maintenance device* to address classes which carry a strong indication of code and community smells, besides re-organisation and re-modularisation. On the one hand, community smells exist at the boundary of people and code, i.e., they are patterns which include both a people and a code component. On the other hand, developers reportedly used *organizational commenting* within code, that is, including maintenance and evolution instructions in source code comments such that, for example, newcomers can contribute knowing what to touch and what not to modify at all.

In conclusion, our main results from the analysis of the additional factors influencing the persistence of code smells show that (i) fault-proneness, (ii) lack of tools, (iii) co-occurrence of code clones, and (iv) coupling of a class are the main technical factors explaining the willingness of developers to perform refactoring.

3.5 Summary of Findings

In summary, the main output of our qualitative analysis revealed that the decision on whether to refactor a code smells is dependent on a number of different factors. It is indeed not only dependent on community or technical factors, but rather their *combination* better fits the developers' willingness or ability to maintain code smells. This seems to indicate that *community-aware* code smell prioritisation approaches could better pinpoint to developers which code smells can be more easily removed, thus providing a more practical solution to deal with them.

The results also provide a clear indication that community and code smells are influenced by each other. We adopted the *Timeliness::Social-Then-Technical* code to responses of developers saying that they did not address a code smell because it would cause a community smell—the community smell is then the effect of the code smell. Conversely, the opposite is true for the *Timeliness::Technical-Then-Social* code. Through content analysis we observed that, for over 70 percent of the reported code smells, the decision not to refactor was due to a potential community smell, i.e., *Timeliness::Social-Then-Technical*. This evidence seems to indicate a dimension of intentionality for code smells themselves—oftentimes it is more convenient to keep code smells rather than addressing community smells. This result is particularly important, as it suggests the need for practical solutions aiming at *anticipating* situations that might become critical for persistence in the next future.

> *Summary for RQ1.* Our evidence shows that community smells, together with other technical factors, influence the maintenance decisions for code smells. At the same time, we observed that in several cases it is more convenient to keep code smells rather than addressing community smells. These findings suggest the need for (1) *community-aware* approaches for assessing the refactorability of code smells and (2) automated ways to anticipate critical situations that may lead developers to not refactor a code smell at all.

4 Community versus Code Smells

In parallel to addressing *RQ1*, our *goal* was to study the relationship between *community* and *code* smells *quantitatively*, with the *purpose* of understanding to what extent community smells can be exploited to diagnose the persistence of code smells. Therefore, this side of the study addresses the following research questions:

- *RQ2: To what extent can community smells explain the increasing of code smell intensity?*
- *RQ3: Does a community-aware code smell intensity prediction model outperform models that do not consider this information?*

In *RQ2*, we perform a fine-grained measurement of the extent to which code smell intensity can be explained by the presence of community smells, while in *RQ3* we study the feasibility of an automated solution that supports developers when diagnosing future code smell intensity by explicitly taking into account the status of the software community. With fine-grained, we indicate the feature of our study of mapping code and community smells at the level of software artifacts and the actual people working on them; more specifically, every community smell was

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

reported as relevant for our study (and the preparation of the statistical model) if and only if it reflected on code-smelly software code artifacts. This is the finest-grained approach possible since it looks at the community structure and technical structure around software at its most fine-grained level. A more coarse-grained approach could have considered the organizational structure as a whole, e.g., by correlating its organizational characteristics/smells and its technical outputs.

The choice of devising a community-aware code smell intensity prediction model aimed at predicting the future intensity of a code smell instance comes from some observations:

- As shown in the context of *RQ1*, developers tend to prefer keeping a code smell in the source code rather than dealing with a community smell. Thus, one might think that a code smell that co-occurs with a community smell may not be taken into account by a developer. While this may be true, the role of prediction models is that of *anticipating* situations where the co-occurrence might lead to more serious maintainability issues. For example, suppose in a certain release R_i, a code smell has a low intensity and that a prediction model predicts the intensity of this smell to increase in the subsequent release R_{i+1} because the features of the model related to community smells. In this case, a project manager may immediately take action, trying to fix the community-related issues with the aim of preventing the smell to increase in its intensity.
- As shown by recent papers [59], [60], the perception of code smells heavily depends on their intensity. We argue that approaches able to precisely indicate the future severity of code smell instances might allow developers to (i) understand the possible criticism of the software being developed that may arise in the short-term future and (ii) deal with or simply monitor evolution of the code smells [19].
- In the past, we have shown that the intensity of code smells has a strong impact on fault-proneness [8] and can be actually used to identify parts of source code that are likely to be defective [61]: thus, intensity prediction models can help developers assess when a certain refactoring or other program transformations must be applied to not incur possible additional maintainability and/or reliability problems.
- Intensity can be used as a means for selecting the code smell instances that need to be more urgently fixed. As not all the smells are or can be removed, an approach able to rank them based on their severity might be worthwhile to allow developers selecting the instances on which to focus more, or even those that are more relevant to manage because of the co-occurrence of a community smell.

For all the reasons reported above, we believe that the definition of a community-aware code smell intensity prediction model can be one of the most practical approaches that developers and project managers can use to diagnose the future persistence of code smells and eventually take decisions on which instances should be refactored.

4.1 Context of the Study

The software systems and the code smells involved in the *context* of this second study are the ones used for answering *RQ1*. In addition, in this study we considered 4 of the community smells defined by Tamburri et al. [4], namely:

(1) *Organizational Silo Effect:* siloed areas of the developer community that essentially do not communicate, except through one or two of their respective members;
(2) *Black-cloud Effect:* information overload due to lack of structured communication or cooperation governance;
(3) *Lone-wolf Effect:* unsanctioned or defiant contributors who carry out their work irrespective or regardless of their peers , their decisions and communication;
(4) *Bottleneck or "Radio-silence" Effect:* an instance of the "unique boundary spanner" [62] problem from social-networks analysis: one member interposes herself into every formal interaction across two or more sub-communities with little or no flexibility to introduce other parallel channels;

The choice of these community smells come from the results of previous literature which theorises the co-occurrence with or the causality between code smells/problems and all four effects we seek for [2], [4], [63].

Conversely, the community smells we identified in the parallel qualitative study (see rows 3–6 of Table 2), were not known yet during the setup of the quantitative study and are not considered; what is more, they are an original contribution of this study and are currently under operationalisation. Similarly, we could not consider the prima-donna smell because of the lack of approaches and/or tools actually able to identify its presence.

4.2 Detecting Community Smells

In order to detect community smells, we exploit the CodeFace4smells tool, a fork of CodeFace [64] designed to identify developers' communities. Starting from the developer networks built by CodeFace, we detect instances of the considered smells according to the following formalizations.

Organizational Silo. Let $G_m = (V_m, E_m)$ be the communication graph of a project and $G_c = (V_c, E_c)$ its collaboration graph. The set of *Organizational Silo* pairs S is defined as the set of developers that do not directly or indirectly communicate with each other, more formally:

$$\{(v_1, v_2) | v_1, v_2 \in V_c, (v_1, v_2) \notin E_m^*\},$$

where E_m^* is the transitive closure of E_m. With *transitive closure* we indicate the transitive closure of a graph. More specifically, given a directed graph, the operation finds out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. With *reachable* we mean that there is a path from vertex i to j. The reach-ability matrix is called *transitive closure* of a graph.

Similarly, the set of *lone wolf* pairs L is defined as the set of collaborators that do not directly or indirectly communicate with each other, more formally:

$$\{(v_1, v_2) | v_1, v_2 \in V_c, (v_1, v_2) \in E_c, (v_1, v_2) \notin E_m^*\}.$$

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

It follows that, by definition, $L \subseteq S$, meaning that lone-wolves are a subset, or a specific instance of organisational silo effect.

Black-Cloud and Lone Wolf. Detection of the *Black cloud and Lone wolf* smells starts with the detection of vertex clusters as already implemented in CodeFace. More specifically, let $P = \{p_1, \ldots, p_k\}$ be a mutually exclusive and completely exhaustive partition of V_m induced by the clustering algorithm. From the partition, *black cloud* is the set of pairs of developers C that connect otherwise isolated sub-communities, more formally:

$$\{(v_1, v_2) | v_1, v_2 \in V_m, (v_1, v_2) \in E_m, \forall i, j(((v_1 \in p_i \wedge v_2 \in p_j) \Rightarrow i \neq j) \wedge \forall v_x, v_y((v_x \in p_i \wedge v_y \in p_j \wedge (v_x, v_y) \in E_m) \Rightarrow v_x = v_1 \wedge v_y = v_2))\}.$$

Bottleneck. Finally, the *bottleneck* set B is the set of developers interposing themselves into every interaction across two or more sub communities. More formally:

$$\{v | v \in V_m, \exists i(v \in p_i \wedge \forall v_x(v_x \in p_i \Rightarrow v = v_x))\} \cup \{v | v \in V_m, \exists v_x, i, j(v \in p_i \wedge v_x \in p_j \wedge (v, v_x) \in E_m \wedge \forall v_y, v_z((v_y \in p_i \wedge v_z \in p_j \wedge (v_y, v_z) \in E_m) \Rightarrow v_y = v)\}.$$

Meaning that developers can interpose themselves into interactions if either they are the only member of their cluster (left-hand side of the expression above) or they communicate with a member of the different cluster, and they are the only member of their cluster communicating with this different cluster (right-hand side of the expression above); both these instances are united to form the set of Bottleneck effects existing in a developer social network.

It is important to point out that the detection techniques described were also evaluated in order to assess their actual ability to identify community smells. Specifically, we ran CodeFace4Smells on 60 open-source projects and, through a survey study, we asked the original developers of such systems whether the results given by the tool actually reflect the presence of issues within the community. As a result, we discovered that the recommendations of the tool highlight real community-related problems. Furthermore, it should be noted that the effectiveness of the operationalisations above rely on the proven effectiveness of the approach by Joblin et al. [64], building upon the "Order Statistics Local Optimization Method" (OSLOM) [65] featured inside CodeFace, which was never previously applied before on developer networks. Further details of the operationalisation and evaluation are discussed in the accompanying technical report [66].

It should be also noted that the projects considered for the scope and context of this study were a selected subset of the 60 projects with which we evaluated the CodeFace4Smells tool; therefore, the smells we detected constitute actual and validated occurrences. For the sake of completeness, we provide the full technical report of how the tool was evaluated.[2]

4.3 RQ2. Factors That Intensify Code Smells

To answer *RQ2* and properly assess the role of community smells in the variation of intensity of code smells, we defined a model relating a set of independent variables (formed by both community smells and other control factors) to a dependent variable (that is, the intensity of code smells). The following sections describe them further.

4.3.1 Dependent Variable

The variable of interest is code smell intensity. In the first place, to compute the code intensity value, we consider how much the value of a chosen metric exceeds a given threshold [67]. The conjecture is that the higher the distance between the actual code metric value and the corresponding fixed threshold value, the higher the intensity of the code smell. In our case, the code smell detector classifies a code entity (i.e., a method, a class, or a package) as smelly analysing whether code metrics used by the detector exceed the predefined threshold defined in the corresponding rule card [29]. In the second place, the actual measurement was done as suggested by previous work [20], [68]: (i) we computed the differences between actual metric values and reported thresholds; (ii) we normalised the obtained difference scores in $[0; 1]$, and (iii) we measured the final intensity as the mean of those normalised scores. Note that we are aware that the mean operator might be biased by the presence of outliers [69]: however, experimental tests—further described in Section 6—showed that our results would not change if the aggregation would have been done using the median. Subsequently, we converted the floating-point double value in a nominal value in the set $\{NULL, LOW, MEDIUM, HIGH\}$: if a class is non-smelly (i.e., the detector does not detect any code smell instance), its intensity is $NULL$, while if the class is smelly (intensity > 0), then the code smell intensity is categorised as LOW, $MEDIUM$, or $HIGH$. To assign the intensity to one of these classes, we analysed the distribution of the intensity values for a given project. Thus, if a code smell intensity is lower than the first quartile of the distribution it has been assigned to LOW; if it is between the first and third quartile, it has been assigned to $MEDIUM$; if it is higher than the third quartile, its corresponding class is $HIGH$. Our choice of using quartiles to discriminate the levels of smelliness of code components is given by the fact that quartiles represent classical methods for measuring the skewness of data as is in our case; we simply chose to map each quartile to an individual class (low, etc.) thus making our research design void of any misinterpretation from a statistical perspective. Note that since our study focuses on five different code smells (see Section 3.1), we computed and analysed the intensity for each smell independently.

4.3.2 Independent Variables

We aim at understanding the impact of community smells on the intensity of code smells. Thus, based on the output of CodeFace4Smells, we analysed whether a certain class C_i has been modified by developers involved in a community smell in a time between the releases R_{j-1} and R_j. Thus, we computed four boolean values representing the involvement of such class in any of the four community smells considered. These metrics represent the principal factors that we wanted to analyse.

2. https://tinyurl.com/CodeFace4Smells

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

4.3.3 Non-Community-Related Control Variables

While the results of our parallel study highlighted that community smells might affect the way developers treat code smells, it is important to remark that other factors related to the structure of source code (e.g., number of lines of code) as well as the development process (e.g., number of commits performed on a class) might be the primary source of information to understand code smell intensity. For this reason, we defined a list of technical factors having the role to *control* confounding effects when evaluating the role of community smells. Specifically, for each class we computed the following metrics:

Lines of Code. LOC of a class is widely recognised as a potential confounding factor of phenomena occurring on a certain code entity [70], [71]. Thus, we compute the LOC of each class C_i in a release R_j.

Coupling between Object Classes. The number of external dependencies of a class might represent an important factor that influences the persistence of code smells [29]: it is worth noting that the practitioners' answers to the survey (see Section 3.4) confirm the relevance of coupling. In our context, we compute the CBO metric [72] of a class C_i in a release R_j.

Total Commits. A potential confounding factor might be the number of commits performed on a class C_i: here, the conjecture is that the higher the number of times the class changes the higher its proneness to deteriorate over time [48]. Hence, we compute the number of commits modifying C_i up to the release R_j.

Class Change Process. The way a class C_i changes between releases R_{j-1} and R_j might impact its size and complexity [73], [74], thus possibly increasing the intensity of code smells. For this reason, we measured (i) number of lines of code added or modified in the class between R_{j-1} and R_j (a. k.a., code churn) and (ii) number of commits performed on the class between R_{j-1} and R_j.

Developer-Related Factors. Besides structural and process metrics, also *who* touches a class C_i might influence the intensity of code smells [75], [76]. For this reason, we computed the number of developers who committed changes to C_i between R_{j-1} and R_j. Next, we computed two metrics measuring the experience of these developers. The first metric is *commit tenure* [48]: it computes the general experience within the same ecosystem as the number of months since the developer's first event on any APACHE (for APACHE PROJECTS) or ECLIPSE (for ECLIPSE PROJECTS) repositories; the second one is *project tenure* [48] and measures the experience of a developer on the project of interest as the number of months since her first event on the project repository. Finally, metrics for developers that committed changes to C_i between R_{j-1} and R_j are aggregated by computing medians. As this decision influences our research design, it constitutes a threat to validity which encourages further replication of this study, e.g., considering more structured metrics that address the activity of the developer—for example, it might be better to re-define the reputation/experience of the developer as the "success rate" prior to committing the report under consideration (see Hooimeijer et al. [77]).

Maintainability Measures. Previous work showed that classes affected by problems in the past are more likely to be problematic in the future [78], [79]. Hence, we measured (i) code smell persistence, i.e., number of previous releases (up to the release R_j) in which the class C_i has been affected by a certain smell type and (ii) the value of the code smell intensity in the previous release R_{j-1}.

Moreover, we also considered the presence of code clones and fault-proneness. As for the former, we employed the DECKARD tool [80], a technique able to identify Type-3 clones: based on the output of the tool, we marked the class as affected or not by a clone. It is important to note that we selected this tool since it is publicly available and has high detection accuracy [80]. As for the fault-proneness, we have been interested in measuring what will be the fault proneness of C_i in a subsequent release: to tackle this problem, we adopted the hybrid bug prediction model devised by Di Nucci et al. [75]. It is able to provide an indication about the fault-proneness of classes based on a mixture of product, process, and developer-based metrics. Also in this case, the choice of using this model is instigated by its high accuracy [75]. We also empirically assessed the performance of these two approaches on a subset of projects in our dataset, showing that the levels of accuracy reported by the original authors still hold in our context: more details on this assessment are reported in Section 6.

4.3.4 Community-Related Control Variables

Finally, we also considered as possible confounding factors aspects related to the community structure, as represented by the intersection of a communication network (CommNet; stemming from mailinglist data) and a collaboration network (CollNet; stemming from co-commit relationships among developers). Specifically, we controlled for:

Truck-Factor. Originally formulated as "The number of people on your team who have to be hit with a truck before the project is in serious trouble"[3] and established in software engineering literature as well [81], [82], [83]. We operationalise truck-factor based on core and peripheral community structures identified by CODEFACE, as the degree of ability of the community to remain connected without its core part. Further details on how core and periphery members are determined can be found in the work of Joblin et al. [64].

Socio-Technical Congruence. Paraphrased from previous work [84] as "the state in which a software development organisation harbors sufficient coordination capabilities to meet the coordination demands of the technical products under development" and operationalised in this study as the number of development collaborations that do communicate over the total number of collaboration links present in the collaboration network.

Core-Periphery Ratio. This ratio has been confirmed to regulate communities [64]. We operationalise it as the ratio between the median centrality of periphery members and the median centrality of the core members. In other words, we considered the importance of core developers with respect to periphery ones.

Turnover. This quantity reflects the amount of people who migrate from the community across subsequent 3-month time-windows of our analysis [85], [86], [87]:

3. http://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor/

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

$$TO(\text{CommNet}, \text{CollNet}) = \frac{\text{Leaving}}{(\text{Populus} + \text{Size})\,/\,2} * 100\%,$$

where, *CommNet* and *CollNet* are conjuncted using a 1-Elementary Symmetric Sum between adjacency matrices [88], i.e., $(V_m \cup V_c, E_m \cup E_c)$ in the notation above. Variables in the formula above are as follows: (1) *Leaving* is the number of members who left the project in the analysed window; (2) *Populus* is the total number of members who populated the community in the previous analysis window; (3) *Size* is the total number of members populating the community in the currently analysed window. Similar formulations of turnover exist [48], [89] but we chose the formulation above since it matches the definition of turnover and, by the way in which CodeFace computes the formula variables, our formulation accounts for both core and periphery member turnover; this differentiation is previously absent in literature and the easiest to operationalise with our available tooling, e.g., CodeFace determines *Populus* for both core and periphery communities, combining both into one after a normalisation based on amount of contribution.

Smelly-Quitters. This ratio reflects the amount of people P who were part of a community smell C_X for two subsequent time windows T_1 and T_2 but then left the community for the remaining time windows (T_{2+y} where $y > 0$) in the available range of data for the total set of community smells found, i.e., *C*. More formally:

$$P = \frac{\sum P(C_X)}{C}.$$

The quantity in question is tailored from the social-networks analysis metrics also used for Social Network Disorder measurement [90], [91].

4.3.5 Data Analysis

To answer our research question, we build a classification model and evaluate the extent to which community smells are relevant by quantifying *information gain* [92] provided by each independent variable in explaining the dependent variable. We opted for this technique because it is able to *quantify* the actual gain provided by a certain feature to the performance of the model. The same would not be possible with other techniques like, for instance, the Wrapper technique [93]—which is among the most popular ways to assess feature selection [75], [94], [95].

We exploited the *Gain Ratio Feature Evaluation* algorithm [92] integrated in the Weka framework [96], that ranks the independent variables in descending order based on the information gain provided. To statistically verify the ranks provided by the algorithm, we adopted the Scott-Knott test [97]. This test is widely used as a multiple comparison method in the context of analysis of variance [98] because it is able to overcome a common limitation of alternative multiple-comparisons statistical tests (e.g., the Friedman test [99]), namely the fact that such tests enable the possibility for one or more treatments to be classified in more than one group, thus making it hard for the experimenter to really distinguish the real groups to which the means should belong [100]. In particular, the test makes use of a clustering algorithm where, starting from the whole group of observed mean effects, it divides, and keeps dividing the sub-groups in such a way that the intersection of any two groups formed in that manner is empty. In other words, it is able to cluster the different ranks obtained into statistically distinct groups, making more sound and easier the interpretation of results. For these reasons, this test is highly recommended and particularly suitable in our context. It is worth noting that the selection of this test was driven by our specific need to perform statistical comparisons over multiple datasets. In this regard, the use of more popular statistical techniques like, for instance, Wilcoxon [101] or Cliff's Delta [102] is not recommended because they might lead to inappropriate interpretation of the results or even wrong application of statistical tests [103].

4.4 RQ3. Evaluating a Community-Aware Code Smell Intensity Prediction Model

As a final step of our study, we evaluated to what extent software developers can benefit from the usage of community smell-related information when evaluating the future intensity of code smells, in order to improve the scheduling of refactoring operations. To this aim, we took advantage of the results of *RQ2* to build a code smell intensity prediction model for each code smell considered in the study. Specifically, starting from the output of the *Gain Ratio Feature Evaluation* algorithm [92], to avoid model over-fitting [104] we first considered as relevant only the metrics providing information gain higher than 0.10 as suggested by previous work [92]. Then, we built three prediction models: (i) based on technical metrics, (ii) based on technical metrics and community smells, and (iii) based on technical metrics, community smells and the other community-related metrics presented in Section 4.3.4. We selected these models since we could (i) quantify how much the addition of only community smell information into a model considering technical metrics improves its performance and (ii) test whether the addition of further metrics characterising the software community is actually needed or the information provided by community smells is already enough to capture relevant community-related aspects.

As for the classifier, the related literature [105], [106] recommended the use of the *Multinomial Regression* technique [107], as it is among the most reliable ones. However, other machine learning algorithms might still perform better in the context of code smell intensity prediction. Thus, to select the most appropriate classifier we experimented with seven different classifiers that have different characteristics and make different assumptions on the underlying data, i.e., *ADTree*, *Decision Table Majority*, *Logistic Regression*, *Multilayer Perceptron*, *Multinomial Regression*, *Support Vector Machine*, and *Naive Bayes*. We selected these approaches because they were previously used in the context of code smell prediction [108]. Specifically, we compared their performance using the same validation strategy and evaluation metrics reported later in this section. As a result, we could actually confirm the superiority of *Multinomial Regression*, which achieved an AUC-ROC 8 percent higher with respect to *Support Vector Machine*, namely the classifier which performed the best after *Multinomial Regression*. A complete report of this comparison is available in our online appendix [23].

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

We measured the performance of the prediction models by applying an *inter-release* validation procedure, i.e., we train the prediction models using the data of release R_{j-1} and test it on the data of release R_j. In this way, we simulate a real-case scenario where a prediction model is updated as soon as new information is available. To quantify the performance, we use the F-Measure [109], and the Area Under the Receiver Operation Characteristic curve (AUC-ROC), i.e., the overall ability of the model to discriminate between true and false positive instances. We also report the Matthews Correlation Coefficient (MCC) [110] and the Brier score [111]. MCC represents the degree of correlation between the independent variables adopted by the model and the dependent variable: the closer MCC is to 1 the higher the accuracy of the model. The Brier score measures the distance between the probabilities predicted by a model and the actual outcome. Higher performance is obtained when the score is close to 0. All scores are produced as percentages in Table 4. To quantify whether and how much the performance of the community-aware models improves with respect to prediction models that do not consider community-related factors as predictors, we built baseline code smell intensity prediction models that use as predictors only the relevant variables that do not measure community aspects. We also statistically verified the performance of the models built by applying the Scott-Knott [97] test on the AUC-ROC achieved by the experimental models.

Meaningfulness of the Results. While the analysis of the performance of the machine learning models provided us with insights on the extent to which they are properly able to classify the future intensity of code smells, we also verified whether the results provided by the models are actually meaningful in practice. To this aim, we conducted a further qualitative analysis in which we involved ten industrial project managers with an experience ranging between 5 and 18 years. With the aim of increasing the generalisability of our findings we checked that none of the participants of the first survey have participated in the second survey. We involved project managers because they are expected to have a stronger knowledge of the entire development teams they manage and the artifacts they produce, as opposed to developers that might have a deep knowledge only of their development team. We invited them by e-mail and we asked them to fill-in a brief survey composed of 4 questions, one for each community smell considered. In particular, we adopted a vignette-based approach [112], where participants were asked to reason about possible scenarios occurring in a real context rather than answering direct questions about the relationships between community and code smells: the rationale behind the use of such an approach is that participants might not be aware of the formal definitions of community and code smells, thus a reasoning based on scenarios might facilitate the survey comprehension. Note that this approach has already been successfully applied in the context of software engineering [113]. In our case, each question started with a scenario presenting the situations in which a community is affected by one of the community smells analysed. All the scenarios are reported in our online appendix [23]. For sake of understandability of the methodology, we report herein the case of Organizational Silo. In particular, we presented the following scenario:

> *"Suppose your development team is working on the definition of a web-based application for the scheduling of resources. During the development, you recognize the existence of independent sub-teams that do not communicate with each other except through one or two of their respective members".*

The participants were first invited to answer a preliminary question:

> *"Do you think this situation can lead to the introduction of code smells, i.e., poor implementation choices, in the source code?"*

If the answer was "yes", we proposed them a set of scenarios describing the typical symptoms of the code smells considered in this paper. For instance, we presented the following scenario in case of the Blob code smell:

> *"A poorly cohesive large class that centralises the behavior of the system, and that is usually characterised by a large number of unrelated attributes and methods and by dependencies with classes acting as data holders".*

Then, we asked them to indicate the likelihood that each scenario might arise in combination with the community issue. In other words, they were required to answer to the following question:

> *"Do you think this situation can appear in combination with the organizational scenario previously proposed?"*

The participants were allowed to answer using a five-point Likert scale [114] ranging between *"Very unlikely"* to *"Very likely"*, and rated a total of ten scenarios: in particular, besides the scenarios related to the five code smells considered in this study, we also provided a set of scenarios describing five code smells from the catalog by Fowler [7] that we did not take into account, i.e., *Long Parameter List*, *Data Clumps*, *Primitive Obsession*, *Refused Bequest*, and *Parallel Inheritance Hierarchies*. This was done with the aim of limiting confirmation biases. It is important to note that the additional smells proposed do not share any characteristics with those studied in this paper, thus being suitable as a baseline. We expected the participants to indicate that the code smells we studied (e.g., *Long Method*) are more likely to emerge in the presence of the related community smell, than the baseline code smell (e.g., *Data Clumps*) in the presence of the same community smell.

Finally, the project managers had the opportunity to leave a comment on each scenario evaluated. Once we had collected the participants' feedback, we verified that the results of the prediction models were in line with the opinions of project managers, thus assessing the extent to which the devised prediction model outputs meaningful recommendations. Specifically, we computed the number of times the managers highlighted relations similar to those produced by the code smell intensity prediction model, i.e., whether both the model and managers related the presence of a certain community smell to the emergence of a certain code smell. The counting process was manually conducted by the first two authors of this paper in a joint meeting where they inspected each survey and verified the concordance between a manager's answer and the relations discovered by the machine learning model.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
RQ2—Results after Information Gain Analysis

Long Method			Feature Envy			Blob			Spaghetti Code			Misplaced Class		
Metric	Gain	Scott-Knott	Metric	Gain	Scott-Knott	Metric	Gain	Scott-Knott	Metric	Gain	Scott-Knott	Metric	Gain	Scott-Knott
LOC	**0.83**	**92**	**CBO**	**0.87**	**85**	**Period Commits**	**0.74**	**91**	**LOC**	**0.77**	**85**	***Organizational Silo***	**0.85**	**91**
Churn	**0.68**	**81**	**Previous Intensity**	**0.79**	**82**	**Previous Intensity**	**0.72**	**81**	**Churn**	**0.73**	**80**	**ST-Congruence**	**0.76**	**90**
Previous Intensity	**0.68**	**84**	**Churn**	**0.76**	**78**	**ST-Congruence**	**0.72**	**79**	**Previous Intensity**	**0.71**	**77**	**Previous Intensity**	**0.61**	**77**
Period-Commits	**0.67**	**82**	***Lone Wolf***	**0.59**	**76**	**CBO**	**0.69**	**72**	**Period Commits**	**0.65**	**71**	***Black-Cloud***	**0.55**	**75**
CS-Persistence	**0.55**	**78**	**ST-Congruence**	**0.56**	**71**	**Churn**	**0.65**	**75**	***Lone Wolf***	**0.56**	**66**	**CBO**	**0.45**	**61**
Black-Cloud	**0.55**	**70**	**LOC**	**0.55**	**70**	**Project Tenure**	**0.65**	**65**	**CS-Persistence**	**0.45**	**63**	**Committers**	**0.35**	**59**
Clones	**0.46**	**67**	**Clones**	**0.53**	**64**	**LOC**	**0.57**	**59**	**Ratio Core-Periphery**	**0.38**	**60**	**Period Commits**	**0.33**	**55**
Organizational Silo	**0.27**	**63**	**Project Tenure**	**0.39**	**53**	***Organizational Silo***	**0.45**	**57**	***Organizational Silo***	**0.24**	**56**	**Ratio Core-Periphery**	**0.32**	**51**
ST-Congruence	**0.18**	**59**	**Truck Factor**	**0.23**	**41**	**Truck Factor**	**0.38**	**51**	**Project Tenure**	**0.23**	**45**	**Truck Factor**	**0.24**	**45**
Turnover	**0.17**	**43**	**Period Commits**	**0.15**	**16**	***Bottleneck***	**0.34**	**50**	**Smelly Quitters**	**0.22**	**44**	**Smelly Quitters**	**0.17**	**42**
Commit Tenure	**0.15**	**35**	**Smelly Quitters**	**0.12**	**13**	**Clones**	**0.27**	**44**	**CBO**	**0.13**	**41**	**CS-Persistence**	**0.14**	**33**
Ratio Core-Periphery	**0.12**	**27**	**Committers**	**0.12**	**11**	**Committers**	**0.24**	**41**	**Total Commits**	**0.12**	**34**	**Churn**	**0.12**	**31**
CBO	**0.12**	**23**	*Bottleneck*	0.09	8	**Turnover**	**0.23**	**33**	***Bottleneck***	**0.12**	**31**	**Project Tenure**	**0.12**	**24**
Fault-proneness	**0.12**	**14**	*Organizational Silo*	0.08	7	**Smelly Quitters**	**0.16**	**27**	Committers	0.09	5	*Bottleneck*	0.09	11
Lone Wolf	0.09	8	Fault-proneness	0.05	4	**Fault-proneness**	**0.14**	**19**	Truck Factor	0.08	4	Total Commits	0.08	8
Total Commits	0.08	8	Turnover	0.05	4	Ratio Core-Periphery	0.09	9	Commit Tenure	0.08	4	Clones	0.08	8
Bottleneck	0.07	6	Ratio Core-Periphery	0.04	2	*Black Cloud*	0.04	5	Fault-proneness	0.07	3	*Lone Wolf*	0.07	5
Committers	0.07	4	Total Commits	0.03	2	*Lone Wolf*	0.03	3	Clones	0.06	2	Turnover	0.07	3
Truck Factor	0.05	2	CS-Persistence	0.03	2	CS-Persistence	0.03	2	Turnover	0.05	1	LOC	0.02	1
Smelly Quitters	0.04	2	*Black Cloud*	0.02	1	Total Commits	0.01	1	*Black Cloud*	0.04	1	Commit Tenure	0.02	0
Project Tenure	0.02	1	Commit Tenure	0.01	0	Commit Tenure	0.01	1	ST-Congruence	0.03	1	Fault-proneness	0.01	0

*Metrics contributing more than the threshold of 0.10 are highlighted in **bold**, while community smells are reported in* italic.

4.5 Analysis of the Results

For sake of clarity and to avoid redundancies we discuss the results for *RQ2* (see Table 3) and *RQ3* (see Tables 4 and 5) together. Table 3 reports for each code smell the ordering of features according to the results of the information gain algorithm, along with the likelihood that a certain feature was ranked at the top by the Scott-Knott test (a likelihood of 80 percent means that in 80 percent of the datasets the gain provided by the feature was statistically higher than others). Looking at the results, we can observe that for all the code smells but Misplaced Class technical control factors such as LOC, CBO, Code Churn, and number of commits between two releases are the ones that more closely influence the code smell intensity. These results were quite expected, since previous work has shown there exist technical aspects of source code (e.g., LOC) that "naturally" influence the evolution of code smells [115], [116]. At the same time, also the previous intensity of a code smell plays an important role when explaining the future intensity: likely, this is a reflection of the fact that developers generally tend to not refactor code smells and, when they do, the refactoring is most of the times not effective [13], thus increasing the intensity of smells over time.

Other technical factors have a relevant role only when considering some code smell types: for example, the fault-proneness seems to have a relationship only with Long Method and Blob intensity, while it has marginal values in all the other cases.

Turning the attention to the independent variables that we wanted to assess, in the first place we observed that community smells represent relevant features *for all* the considered code smells, confirming the results achieved when surveying developers. It is worth noting that the community-related control factors (e.g., socio-technical congruence) are often not able to explain the dependent variable better than community smells, meaning that our independent variables are generally more powerful "predictors" than such control factors. For instance, the organizational Silo provides a higher entropy reduction (0.27) than socio-technical congruence (0.18) when considering the Long Method smell. In the second place, it is important to note that different community smells are related to different code smells, meaning that circumstances occurring within a software community somehow influence the persistence of code smells. In the case of Long Method, such circumstances are related to non-structured or even missing communications between developers, as highlighted by the fact that community smells like Black Cloud and

TABLE 4
RQ3—Performance of Code Smell Intensity Prediction Models Built with and without Community-Related Factors—All Numbers Are Percentages

Model	Long Method					Feature Envy					Blob					Spaghetti Code					Misplaced Class				
	F-M	AUC	MCC	Brier	Scott-Knott	F-M	AUC	MCC	Brier	Scott-Knott	F-M	AUC	MCC	Brier	Scott-Knott	F-M	AUC	MCC	Brier	Scott-Knott	F-M	AUC	MCC	Brier	Scott-Knott
Cassandra																									
Basic + Comm.-Smells + Comm.-Factors	71	68	73	28	66	69	68	78	36	64	72	74	81	31	77	71	76	75	38	63	74	73	83	28	61
Basic + Comm.-Smells	70	68	72	29	63	66	66	77	37	59	70	71	78	33	68	70	70	73	38	60	72	71	79	30	58
Basic	68	66	71	29	45	62	62	75	40	51	64	64	68	39	50	67	62	67	39	61	63	63	59	40	52
Jackrabbit																									
Basic + Comm.-Smells + Comm.-Factors	69	66	74	32	69	71	67	79	32	82	69	78	75	31	76	71	75	74	36	65	75	70	75	28	77
Basic + Comm.-Smells	70	72	72	31	68	67	68	75	39	77	70	72	74	39	76	68	73	68	35	75	75	69	72	28	71
Basic	68	64	71	36	66	63	64	73	28	56	68	63	70	39	59	65	68	66	42	61	61	65	63	35	62
Lucene																									
Basic + Comm.-Smells + Comm.-Factors	71	73	74	27	78	68	66	73	32	71	71	73	76	38	73	70	70	77	39	81	78	74	79	26	62
Basic + Comm.-Smells	71	71	73	30	76	66	66	74	35	67	70	68	75	34	71	69	71	76	39	80	75	71	73	28	61
Basic	70	63	71	37	56	59	66	75	36	66	63	58	73	31	58	64	71	71	41	60	62	62	59	30	55
Pig																									
Basic + Comm.-Smells + Comm.-Factors	71	72	73	30	67	65	68	76	30	79	69	69	82	30	79	69	65	68	34	77	72	71	78	27	81
Basic + Comm.-Smells	69	71	72	31	65	64	66	76	31	74	68	68	79	32	77	67	63	71	35	63	70	71	76	27	78
Basic	68	68	71	31	62	62	63	71	33	64	66	59	70	36	61	62	61	72	43	59	61	60	64	34	62
CFX																									
Basic + Comm.-Smells + Comm.-Factors	73	73	73	31	79	67	76	73	29	77	73	68	80	30	83	71	62	71	41	87	76	73	77	28	81
Basic + Comm.-Smells	71	71	72	32	75	66	75	73	29	75	73	67	80	30	81	70	72	69	41	86	75	72	77	29	78
Basic	67	70	71	36	71	60	71	70	31	66	64	62	68	31	61	66	72	62	41	59	62	65	63	36	61

Model	Long Method					Feature Envy					Blob					Spaghetti Code					Misplaced Class				
	F-M	AUC	MCC	Brier	Scott-Knott	F-M	AUC	MCC	Brier	Scott-Knott	F-M	AUC	MCC	Brier	Scott-Knott	F-M	AUC	MCC	Brier	Scott-Knott	F-M	AUC	MCC	Brier	Scott-Knott
Cayenne																									
Basic + Comm.-Smells + Comm.-Factors	70	69	73	28	74	69	66	73	26	71	73	73	82	30	75	69	69	67	40	74	74	70	82	23	67
Basic + Comm.-Smells	70	68	73	31	72	67	70	71	27	69	73	73	82	30	75	66	67	64	40	71	71	68	79	25	65
Basic	69	67	71	37	62	62	69	68	39	61	64	65	73	35	63	63	63	62	40	66	61	60	58	34	58
Jena																									
Basic + Comm.-Smells + Comm.-Factors	71	72	72	30	71	69	68	78	37	81	71	75	79	36	77	71	76	68	34	78	78	77	76	21	75
Basic + Comm.-Smells	69	65	73	33	67	69	66	77	29	79	68	71	74	34	74	69	72	71	38	63	73	67	69	29	75
Basic	67	69	72	33	61	63	68	71	43	59	68	63	66	39	61	67	67	66	39	55	64	64	59	37	61
Mahout																									
Basic + Comm.-Smells + Comm.-Factors	72	73	73	30	76	66	73	74	29	79	68	74	82	34	91	70	71	67	38	78	75	73	77	27	87
Basic + Comm.-Smells	71	69	71	32	73	63	71	74	33	77	65	72	79	36	86	69	70	67	38	74	74	72	77	28	81
Basic	67	64	70	37	62	60	63	68	42	58	61	61	72	44	61	65	64	69	37	55	60	58	60	33	59
CDT																									
Basic + Comm.-Smells + Comm.-Factors	72	70	71	33	75	69	70	72	31	79	72	70	77	29	76	71	63	67	41	84	77	72	78	25	70
Basic + Comm.-Smells	71	68	71	33	72	68	70	74	31	75	70	67	74	31	73	69	69	66	39	83	74	68	72	26	70
Basic	68	65	70	34	65	61	68	75	35	55	65	56	71	42	59	68	70	63	38	62	60	56	60	33	56
Overall																									
Basic + Comm.-Smells + Comm.-Factors	71	70	73	30	73	65	69	75	31	75	71	73	79	33	81	71	70	70	38	76	76	73	78	26	72
Basic + Comm.-Smells	70	69	73	31	71	64	66	74	32	73	69	68	74	33	78	68	67	67	39	71	75	73	76	27	68
Basic	68	67	71	34	61	61	66	72	36	58	65	61	70	36	63	65	66	66	40	61	62	61	61	35	59

Grey rows indicate the most performing models based on AUC-ROC values.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

TABLE 5
RQ3—Each Cell Contains How Many Project Managers Indicated the Relation between the Code and Community Smell as Very Unlikely, Unlikely, Neither Unlikely nor Likely, Likely, or Very Likely

	Org. Silo	Black Cloud	Lone Wolf	Bottleneck
Long Method	**-,-,-,2,8**	**-,-,-,3,7**	9,1,-,-,-	10,-,-,-,-
Feature Envy	9,1,-,-,-	10,-,-,-,-	**-,1,-,4,5**	10,-,-,-,-
Blob	**-,-,-,-,10**	8,2,-,-,-	10,-,-,-,-	**-,-,-,-,10**
Spaghetti Code	**9,1,-,-,-**	10,-,-,-,-	**-,-,1,1,8**	**9,1,-,-,-**
Misplaced Class	**-,-,1,2,7**	**10,-,-,-,-**	9,1,-,-,-	9,1,-,-,-
Long Parameter List	10,-,-,-,-	6,3,-,1,-	10,-,-,-,-	10,-,-,-,-
Data Clumps	10,-,-,-,-	10,-,-,-,-	10,-,-,-,-	10,-,-,-,-
Primitive Obsession	10,-,-,-,-	10,-,-,-,-	10,-,-,-,-	10,-,-,-,-
Refused Bequest	10,-,-,-,-	10,-,-,-,-	9,1,-,-,-	6,2,1,1,-
Parallel Inheritance Hierarchies	7,2,1,-,-	10,-,-,-,-	10,-,-,-,-	10,-,-,-,-

Relationships between community and code smells expected according to the prediction model are reported in bold; those with the grey background have been confirmed by the survey.

organizational Silo are highly influential. For Feature Envy and Spaghetti Code instances, we noticed that the Lone Wolf community smell provides an important contribution in explaining their intensity: this means that the presence of developers working regardless of their co-committing developers has an impact of the evolution of both methods and classes of a software system. The same happens with Blob, where the organizational Silo and Bottleneck are highly relevant, suggesting that missing communications or lack of flexibility do not allow an effective management of Blob instances. Finally, in the case of Misplaced Class we found that the existence of siloed areas of a community is the most important factor characterizing its intensity, thus indicating once again that community issues might have a notable influence on technical aspects of the source code.

These observations are confirmed by the survey with project managers, whose results are reported in Table 5. In the first place, we notice that project managers considered as meaningful most of the relationships between community and code smells that were ranked at the top by the information gain analysis (see Table 3), while they did not confirm three community-code smell pairs, i.e., Spaghetti Code-Organizational Silo, Spaghetti Code-Bottleneck and Misplaced Class-Black Cloud. This result might be due to a lack of awareness of community or code smells [59], however further analyses aimed at investigating the presence/absence of such relationships are needed. In terms of control, we see that the managers are much more inclined to select the links between the code smells we have focused upon rather than other code smells. This strengthens our confidence that the relations suggested by the prediction model correspond to perceptions of the practitioners. Indeed, they generally agreed with the associations between community and code smells extracted from the information gain analysis. It is worth mentioning one of the comments left by a project manager when evaluating the relationship between Blob and Organizational Silo:

> *"The creation of extremely complex and poorly cohesive classes in the presence of non-communicating sub-teams is quite common because in such a scenario developers do not share information with each other about the ideal structure to implement, therefore creating classes that perform a lot of different things".*

> *Summary for RQ2.* We conclude that community smells significantly influence the intensity of all the code smells considered in this study, being even more important than other community-related metrics (e.g., socio-technical congruence). Nevertheless, we observed that technical factors still give the major explanation of the variation of code smell intensity.

As further evidence of the relationship between community and code smells, when considering the performance achieved by the investigated prediction models (see Table 4), we observe that the "Basic + Comm.-Smells" model, i.e., the one containing both technical factors and community smells, achieves higher performance than the model built without community-related factors when considering all the code smells. For instance, the models including community smell information have an overall AUC-ROC 2, 4, 3, 3, and 11 percent higher that the baseline models when considering Long Method, Feature Envy, Blob, Spaghetti Code, and Misplaced Class, respectively. Thus, our results suggest that the presence of community smells is a factor to take into account when predicting the future persistence of code smells. It is worth noting that the addition of other community-related factors (rows labeled with "Basic + Comm.-Smells + Comm.-Factors" in Table 4) provides a limited boosting of the performance (on average, 2.4 percent in terms of AUC-ROC): this result seems to confirm that community smells are more important than other community-related factors when diagnosing the causes for the persistence of code smells.

The results are (i) consistent over all the individual systems in our dataset and (ii) statistically significant, since the Scott-Knott rank systematically reports "Basic + Comm.-Smells + Comm.-Factors" before "Basic + Comm.-Smells" and "Basic" (see column "SK-ESD" in Table 4). It is important to note that even if sometimes the difference between the "Basic + Comm.-Smells + Comm.-Factors" and "Basic + Comm.-Smells" models is relatively small, we can still observe that, despite the presence of community-related factors, community smells still help improving the performance of the model. This means that there is always a gain in considering such smells. From a practical perspective, the results tell us that explicitly considering community smells enables a developer to better evaluate how the persistence status of code smells will be in the future release, possibly applying preventive actions aimed at improving the quality of both code and community alike.

> *Summary for RQ3.* We observe that the addition of community smells as independent variable of a code smell intensity prediction model enables an improvement of the prediction capabilities. Furthermore, a model including information related to both community smells and other community-related factors improves the accuracy of the prediction of the future intensity of code smells even more. Thus, we conclude that organizational information should be taken into account when analysing technical problems possibly occurring in the source code.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

5 Theoretical Convergence

Recall that the aim of our study was understanding the role of *community smells* as factors contributing to the persistence of code smells. In the following we outline the convergence of the theories generated by both qualitative and quantitative inquiry over the scope of the research design previously outlined. The convergence in question is outlined using the theoretical lemmas that our data leads to and finally, the theoretical convergence that both sides lead to.

From a qualitative perspective, our data shows that open-source developers have a tendency to indeed develop community smells which were previously only reported in closed-source. Furthermore, those reported community smells are repeated causes for keeping code smells such as they are, out of fear of tampering with a delicate community structure.

Lemma 1. *Together with technical factors, community smells influence the developers' decisions not to eliminate code smells.*

Furthermore, from a quantitative perspective, statistical modeling indicates that community smells are among factors intensifying the severity of code smells and contributing to prediction of code smell intensity. Community smells are less important than technical factors but more important than other community-related metrics.

Lemma 2. *Together with technical factors, community smells influence the intensity of code smells.*

The theory that both sides of our study converge towards, is strikingly simple but of considerable importance:

> *Theoretical Convergence.* Community Smells Influence Code Smells' Severity.

The impact that this convergence leads to is indeed considerable since the above conclusion indicates at least that: (a) community smells are a force to be reckoned with in terms of software maintenance and evolution; (b) community smells are a variable to consider when calculating and managing technical debt. Further research is needed to understand and scope the impact and implications of the above conclusion.

6 Threats to Validity

A number of threats might have affected our study. This section discusses how we addressed them.

6.1 Threats to Construct Validity

Threats to *construct* validity concern the relationship between theory and observation. In our study, they are mainly due to the measurements we have performed. In the first place, we classified code smell intensity in three levels of severity plus one indicating the absence of smells following the guidelines provided in previous work [20], [68]. Specifically, we computed the final severity as the mean of the normalized scores representing the difference between the actual value of a certain metric and its predefined threshold. We are aware that our observations might be biased by the fact that the mean operator can be affected by outliers. To verify this aspect and understand the extent whether the mean represented a meaningful indicator, we completely re-ran our study computing the final intensity using the *median* operator rather than the mean. As a result, we did not observe differences with respect to the results reported in Section 4. While a complete report of this additional analysis is available in our online appendix [23], we can conclude that in our case the use of the mean operator did not influence our findings. At the same time, we are aware that the intensity computation process is quite complex and for this reason we are planning a sensitivity study to further evaluate the way code smell intensity is set.

Furthermore, in *RQ2* we relied on Deckard [80] and the hybrid bug prediction model built by Di Nucci et al. [75] to compute code clones and fault-proneness, respectively. In this regard, while those approaches have been extensively empirically evaluated by the corresponding authors, it is also important to note that they might still have output a number of false positive information that might have biased the measurements we have done in the context of our work. To account for this aspect, we assessed the performance of the two approaches targeting the systems in our dataset. We could not perform a comprehensive evaluation because of the lack of publicly available datasets reporting defects and clones present in all the considered systems (the definition of such datasets would have required specialized methodologies that go out of the scope of this paper). In particular, we performed the following steps:

- In the case of the DCBM defect prediction model, we could test its performance on two of the systems included in the study, i.e., Lucene and Pig. Indeed, the study conducted by Di Nucci et al. [75] comprises these two systems and therefore a ground truth reporting the actual defects contained in their releases is available. Starting from the publicly available raw data, we ran the model and evaluated its F-Measure and AUC-ROC. The results indicated that on Lucene the F-Measure was 83 percent, while the AUC-ROC reached 79 percent; in the case of Pig, instead, the F-Measure was 86 percent and the AUC-ROC was 78 percent. Note that the percentages were computed considering the median values obtained by running the model over all the considered releases. Of course, we cannot ensure that such performance holds for all the systems considered, however our additional analysis makes us confident of the fact that this approach can be effectively adopted in our context. It is important to remark that our results are in line with those reported by Di Nucci et al. [75], but they are *not* the same. This is due to the slightly different validation strategy: while they trained and tested the model using a 3-months sliding window, we needed to perform a release-by-release strategy. This implied training and testing the model on different time windows. We see some additional value in this way to proceed: we could not only confirm previous results, but as a side contribution we provide evidence that the findings by Di Nucci et al. [75] hold with a different validation strategy.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

- With respect to DECKARD, we assessed its precision[4] on the same set of systems considered for the defect prediction model, i.e., LUCENE and PIG. Specifically, we ran the tool over each release and manually evaluated whether its output could be considered valid or not. To avoid any sort of bias, we asked to an external industrial developer having more than 10 years of experience in Java programming to evaluate the precision of DECKARD for us: the task required approximately 16 work hours, as the tool output was composed of 587 candidate clones. As a result, 439 of the candidates were correctly identified by DECKARD, leading to a precision of 75 percent. Also in this case, we believe that the tool has a performance reasonably high to be used in our context.

Finally, to identify code smells we relied on DECOR. Our choice was mainly driven by the fact that this tool can identify all the smells considered in our study. Furthermore, it has a good performance, thus allowing us to effectively detect code smell instances. As a further proof of that, we empirically re-assessed its accuracy on two systems of our dataset, finding that its precision and recall are around 79 and 86 percent, respectively.

6.2 Threats to Internal Validity

Threats to *internal* validity concern factors that might have influenced our results. The first survey (*RQ1*) was designed to investigate which are factors that influence the persistence of code smells without explicitly asking developers' opinions on social aspects of software communities: this was a conscious design choice aimed at avoiding possible biases (e.g., developer might not have spontaneously highlighted community-related factors) and allowing developers to express their opinions freely. Moreover, to avoid errors or misclassification of the developers' answers, the Straussian Grounded Theory [49] process was first conducted by two of the authors of this paper independently; then, disagreements were solved in order to find a common solution. Furthermore, we are making all the data freely and openly available [23] to further encourage replication of our work.

At the same time, the survey did not include a specific question on whether a developer considered the proposed smell as an actual implementation problem; thus, it is possible that the decision to not refactor some of the instances was just driven by their lack of perception. However, we tried to keep the survey as short and quick as possible in order to stimulate developers to answer. For this reason, we limited it to the essential questions needed to address our research question. At the same time, it is important to note that the developers were free to say that the piece of code was not affected by any smell: while this did not avoid potential confirmation biases, the fact that some developers explicitly reported the absence of an actual implementation problem in the analysed code (this happened 3 times) partially indicate that the involved developers were not biased when answering the survey.

4. We could not assess the recall because of the lack of an oracle.

Still in the context of *RQ1*, we only invited developers who worked on single smelly classes rather than involving those who worked on several smell instances because of our willingness to survey participants having a deep knowledge on both technical and social dynamics behind that classes. As a consequence of this choice, we excluded 168 developers out of the total 640. However, it is worth recognizing that (i) developers who worked with multiple smelly classes might have had a better knowledge of the overall software system design and (ii) code smells appearing in different classes might have been linked to higher-level design or community problems. Thus, replications of our study targeting this potential threat are desirable.

Furthermore, to evaluate the effectiveness of the code smell intensity prediction model, we used an inter-release validation procedure while an n-fold (e.g., 10-fold) validation could be applied on each release independently and then average/median operations could be used to interpret the overall performance of the model on a certain project. While this strategy could be theoretically implemented, the results would not be comparable with those reported in our paper. Indeed, on the one hand we would have a set of independent classifications that are time-independent; on the other hand, an evaluation done explicitly exploiting temporal relations, i.e., a time-dependent analysis. Thus, even if we would have done this analysis, it would have not increased at all the confidence of the results, but rather created confusion. Given the nature of our analysis and the need to perform a time-sensitive analysis, we believe that our validation strategy is the best option in our case.

6.3 Threats to Conclusion Validity

Threats to *conclusion* validity concern the relation between treatment and outcome. In *RQ2*, to avoid a wrong interpretation of the results due to the missing analysis of well-known factors influencing code smell intensity, we exploited a number of control factors that take into account both social and technical aspects of software development. Moreover, we adopted an appropriate statistical tests such as the Scott-Knott one [97] to confirm our observations. Furthermore, we have adopted the well known *F*-measure which has been known to spark debates concerning its appropriateness in specific contexts (e.g., see Powers [117])—in this respect, it is worth noting that we computed other threshold-independent evaluation metrics with the aim of providing a wider overview of the considered prediction models.

6.4 Threats to External Validity

As for the generalisability of the results, we have performed a large-scale empirical analysis involving 117 releases of 9 projects. However, we are aware that we limited our analysis to projects written in Java and only belonging to open-source communities. Further replications of our study in different settings are part of our future research agenda. In *RQ3* we performed a survey with ten project managers: we are aware of the threats given by the limited number of subjects, however we invited participants having a strong level of experience and with a deep knowledge of the development teams they manage and the artifacts they produced.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

Nonetheless, we plan to extend the study by surveying more managers—although this does not warrant confirmatory or explanatory results, it would grant us more generalisability, e.g., by sampling the population randomly and/or embedding a Delphi study connotation (i.e., progressing with surveying to achieve saturation in agreement).

7 Related Work

In the past, the software evolution research community mainly focused on technical aspects of software, by (i) understanding the factors making technical products easier to maintain [8], [17], [19], [118], [119], [120], [121], [122], [123], [124] or (ii) devising techniques to support developers during different evolutionary tasks [29], [41], [42], [125]. In particular, Cunningham [126] introduced the technical debt metaphor, which refers to programming practices that lead to the introduction of bad implementation solutions that decrease source code quality and will turn into additional costs during software evolution. One noticeable symptom of technical debt is the presence of code smells [7], i.e., bad programming practices that lead to less maintainable source code. Such circumstances have been studied extensively over the years, even most recently [20], [127], [128], [129]—these studies also include statistical modelling for code smells prediction [130], [131]. What is still missing, is a statistical model which comprises both technical and organisational factors as well; in that respect, the contributions reported in this paper are completely novel.

On the community perspective beyond the technical aspects highlighted above, software communities have been mainly studied from an evolutionary perspective [132], [133], [134], while few investigations targeted the relationships between community-related information and evolution of technical products. Indeed, most of previous literature focused on social debt, i.e., unforeseen project cost connected to a suboptimal development community [3]. For instance, Tamburri et al. [2] defined a set of community smells, a set of socio-technical characteristics (e.g., high formality) and patterns (e.g., recurrent condescending behavior, or rage-quitting), which may lead to the emergence of social debt. One of the typical community smells they found is the Organizational Silo Effect, which arises when a software community presents siloed areas that essentially do not communicate, except through one or two of their respective members: as a consequence, the development activities might be delayed due to lack of communication between developers. Furthermore, Tamburri et al. [135] defined Yoshi, an automated approach to monitor the health status of open-source communities and that might potentially be used to control for the emergence of community smells.

Besides the studies on social debt, a number of empirical analyses have been carried out on the so-called socio-technical congruence [136], i.e., the alignment between coordination requirements extracted from technical dependencies among tasks and the actual coordination activities performed by the developers. While studies in this category had the intention to investigate the relationship between social and technical sides of software communities (e.g., studying how the collaboration among developers influence their productivity [137] or the number of build failures [138]), they did not address the co-existence and compounding effects between community and code smells, combining community-related quality factors both from a qualitative and quantitative perspective such as we do in this paper. In this respect, our work is unique. As a final note, it is worth highlighting that we preliminarily assessed how community smells influence code smells [139] by surveying developers on such a relation. Our results indicated that community-related factors were intuitively perceived by most developers as causes of the persistence of code smells. In this paper, we build upon this line by providing a large-scale analysis of the relation between community and code smells, and devising and evaluating a community-aware code smell intensity prediction model.

8 Conclusion

The organisational and technical structures of software are deeply interconnected [138]. We conjecture that the debts existing in both structures, namely, social and technical debt may be connected just as deeply. In this paper, we start exploring this relation from the manifestations of both social and technical debt, namely, code and community smells. While previous work offered evidence that these two phenomena occurring in software engineering *may* be correlated [4], in this paper we reported a mixed-method empirical convergence evaluation aimed at providing evidence of this relation.

On the one hand, in a practitioner survey we observed that community-related issues (as indicated by the presence of *community smells*) are actually perceived as indicators of the persistence of code smells, thus indicating the existence of other aspects that impact the maintainability of technical implementation flaws.

On the other hand, in parallel, we experimented quantitatively over the same dataset to observe a series of implicit variables in the survey that lead to better predictions of code smells, as part of a complex causal relationship linking the social and technical debt phenomena. In response, we designed a prediction model to predict code smell intensity using community smells and several other known community-related factors from the state of the art. We found that the higher the granularity of the code smell, the larger the gain provided by this new prediction model. Such a model offers a valuable tool for predicting and managing both social and technical debt jointly, during software maintenance and refactoring.

In terms of industrial impact, the observations keyed in this work offer valuable insights into understanding and managing the joint interactions between social and technical debt at an industrial scale, for example, in terms of the need to better understand the social counterpart to every technical debt item or designing more precise strategies on how to address both at the same time. In this respect, one interesting future direction could be to replicate this study in a proprietary software setting instead of open-source environment, and highlight similarities and differences with the theory empirically evaluated in the scope of this manuscript.

Our future research agenda also includes a deeper analysis of how different feature relevance mechanisms (e.g., the

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

Gini index available with the *Random Forest* classifier [107]) impact the interpretation of the most important features of the devised model as well as how the model works in a cross-project setting [140]. Moreover, we aim at replicating our study while targeting the new community smells that emerged from our survey study, i.e., *Dissensus*, *Class Cognition*, *Dispersion*, and *Code Red*. To this aim, we will first need to define novel detection techniques: while we cannot speculate too much on this point without having empirical data, we hypothesize that some smells can be identified using natural language processing and/or structural analysis. For instance, the *Dissensus* smell arises when developers are not able to reach a consensus with respect to the patch to be applied: likely, in this case the conversations among developers will report contrasting opinions that can be identified using opinion mining techniques or contrasting sentiment detectable using sentiment analysis [141]. In this regard, it is worth remarking that recent findings on sentiment analysis [142], [143] revealed that existing tools are not always suitable for software engineering purposes, thus suggesting that the ability of detecting community smells may depend on the advances of other research fields. At the same time, *Code Red*—which is the smell arising when only 1-2 maintainers can refactor the source code—may be structurally identifiable looking at the developers' collaboration network and applying heuristics to discriminate how many developers over the history of a class applied refactoring operations on it. A similar reasoning may be done for the detection of the Dispersion smell, where historical information on the collaborations among developers might be exploited in a prediction model aimed at predicting the effect of a refactoring on the community structure of a project. Other smells like *Class Cognition* or *Prima Donna* require instead more analyses aimed at understanding the characteristics behind them that would allow their automatic detection.

ACKNOWLEDGMENTS

The authors would like to thank the Associate Editor and anonymous Reviewers for the insightful comments that allowed to significantly strengthen this paper. Palomba gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] M. E. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968. [Online]. Available: http://www.melconway.com/Home/pdf/committees.pdf

[2] D. A. Tamburri, R. Kazman, and H. Fahimi, "The architect's role in community shepherding," *IEEE Softw.*, vol. 33, no. 6, pp. 70–79, Nov./Dec. 2016.

[3] D. A. Tamburri, P. Kruchten, P. Lago, and H. van Vliet, "What is social debt in software engineering?" in *ICSE - CHASE Workshop Series*, 2013, pp. 40–49.

[4] D. A. Tamburri, P. Kruchten, P. Lago, and H. van Vliet, "Social debt in software engineering: Insights from industry," *J. Internet Serv. Appl.*, vol. 6, no. 1, pp. 10:1–10:17, 2015.

[5] U. Alon, "Network motifs: Theory and experimental approaches," *Nat. Rev. Genet.*, vol. 8, no. 6, pp. 450–461, 2007.

[6] D. A. Tamburri and E. D. Nitto, "When software architecture leads to social debt," in *Proc. 12th Working IEEE/IFIP Conf. Softw. Archit.*, 2015, pp. 61–64.

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[8] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, 2018.

[9] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[10] J. W. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Thousand Oaks, CA, USA: Sage Publications, 2002.

[11] M. Di Penta and D. A. Tamburri, "Combining quantitative and qualitative studies in empirical software engineering research," in *Proc. 39th Int. Conf. Softw. Eng. Companion*, 2017, pp. 499–500.

[12] R. B. Johnson and A. J. Onwuegbuzie, "Mixed methods research: A research paradigm whose time has come," *Educational Res.*, vol. 33, no. 7, pp. 14–26, 2004. [Online]. Available: http://edr.sagepub.com/content/33/7/14.abstract

[13] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw.*, vol. 107, pp. 1–14, Sep. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2015.05.024

[14] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems," *Innov. Syst. Softw. Eng.*, vol. 10, no. 1, pp. 3–18, Mar. 2014. [Online]. Available: http://dx.doi.org/10.1007/s11334-013-0205-z

[15] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proc. 16th Eur. Conf. Softw. Maintenance Reengineering*, 2012, pp. 411–416. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2012.79

[16] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 858–870. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950305

[17] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1063–1088, Nov. 2017.

[18] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Eng.*, vol. 17, no. 3, pp. 243–275, 2012.

[19] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Trans. Softw. Eng.*, vol. 44, no. 10, pp. 977–1000, 2018.

[20] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowl.-Based Syst.*, vol. 128, pp. 43–58, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705117301880

[21] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *Proc. 7th IEEE Int Workshop Manag. Tech. Debt*, 2015, pp. 16–24. [Online]. Available: https://doi.org/10.1109/MTD.2015.7332620

[22] 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: Wiley, 1998.

[23] F. Palomba, D. A. Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "There are other fish in the sea! how do community smells influence the intensity of code smells?" 2017. [Online]. Available: https://goo.gl/GsPb1B

[24] D. A. Tamburri, P. Lago, and H. v. Vliet, "Organizational social structures for software engineering," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 3:1–3:35, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2522968.2522971

[25] S. Magnoni, D. A. Tamburri, E. Di Nitto, and R. Kazman, "Analyzing quality models for software communities," *Commun. ACM*, 2017.

[26] P. Kruchten, R. L. Nord, I. Ozkaya, and J. Visser, "Technical debt in software development: From metaphor to theory report on the third international workshop on managing technical debt," *ACM SIGSOFT Softw. Eng. Notes*, vol. 37, no. 5, pp. 36–38, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft37.html#KruchtenNOV12

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

[27] S. Androutsellis-Theotokis, D. Spinellis, M. Kechagia, and G. Gousios, "Open source software: A survey from 10,000 feet," *Found. Trends Technol. Inf. Operations Manage.*, vol. 4, no. 3–4, pp. 187–347, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/fttiom/fttiom4.html#Androutsellis-TheotokisSKG11

[28] D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P. J. Adams, I. Samoladas, and I. Stamelos, "Evaluating the quality of open source software," *Electr. Notes Theor. Comput. Sci.*, vol. 233, pp. 5–28, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/entcs/entcs233.html#SpinellisGKLASS09

[29] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan./Feb. 2010.

[30] B. H. Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)," *Object Oriented Syst.*, vol. 3, pp. 143–158, 1996.

[31] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proc. Int. Conf. Eval. Assessment Softw. Eng.*, 2016, pp. 18:1–18:12. [Online]. Available: http://doi.acm.org/10.1145/2915970.2915984

[32] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *Proc. IEEE 25th Int. Conf. Softw. Anal. Evol. Reengineering*, 2018, pp. 612–621.

[33] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Softw. Quality J.*, vol. 25, no. 2, pp. 529–552, 2017.

[34] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *Proc. IEEE 24th Int. Conf. Softw. Anal. Evol. Reengineering*, 2017, pp. 487–491.

[35] A. Kaur, S. Jain, and S. Goel, "A support vector machine based approach for code smell detection," in *Proc. Int. Conf. Mach. Learn. Data Sci.*, 2017, pp. 9–14.

[36] A. Tahmid, M. N. A. Tawhid, S. Ahmed, and K. Sakib, "Code sniffer: A risk based smell detection framework to enhance code quality using static code analysis," *Int. J. Softw. Eng. Technol. Appl.*, vol. 2, no. 1, pp. 41–63, 2017.

[37] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *J. Object Technol.*, vol. 11, no. 2, pp. 5: 1–38, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/jot/jot11.html#FontanaBZ12

[38] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An experience report on using code smells detection tools," in *Proc. Int. Conf. Softw. Testing Verification Validation*, Mar. 2011, pp. 450–457.

[39] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodology*, vol. 23, no. 4, pp. 33:1–33:39, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629648

[40] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on B-splines," in *Proc. Eur. Conf. Softw. Maintenance Reengineering*, 2010, pp. 248–251.

[41] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Mining version histories for detecting code smells," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 462–489, May 2015.

[42] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *Proc. IEEE 24th Int. Conf. Program Comprehension*, 2016, pp. 1–10.

[43] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Landfill: An open dataset of code smells with public evaluation," in *Proc. IEEE/ACM 12th Working Conf. Mining Softw. Repositories*, 2015, pp. 482–485.

[44] J. Singer and N. G. Vinson, "Ethical issues in empirical studies of software engineering," *IEEE Trans. Softw. Eng.*, vol. 28, no. 12, pp. 1171–1180, Dec. 2002.

[45] D. A. Dillman, *Mail and Internet Surveys: The Tailored Design Method*. Hoboken, NJ, USA: Wiley, 2007.

[46] J. Singer and N. G. Vinson, "Ethical issues in empirical studies of software engineering," *IEEE Trans. Softw. Eng.*, vol. 28, no. 12, pp. 1171–1180, Dec. 2002. [Online]. Available: http://dblp.uni-trier.de/db/journals/tse/tse28.html#SingerV02

[47] G. Bavota, M. L. Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The impact of API change- and fault-proneness on the user ratings of android apps," *IEEE Trans. Softw. Eng.*, vol. 41, no. 4, pp. 384–407, Apr. 2015.

[48] B. Vasilescu, D. Posnett, B. Ray, M. G. J. van den Brand, A. Serebrenik, P. T. Devanbu, and V. Filkov, "Gender and tenure diversity in github teams," in *Proc. 33rd Annu. ACM Conf. Human Factors Comput. Syst.*, 2015, pp. 3789–3798. [Online]. Available: http://doi.acm.org/10.1145/2702123.2702549

[49] J. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[50] S. Harispe, S. Ranwez, S. Janaqi, and J. Montmain, *Semantic Similarity from Natural Language and Ontology Analysis*, vol. 8. Morgan & Claypool Publisers, 2015.

[51] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*, 2nd ed. Thousand Oaks, CA, USA: Sage Publications, 2004.

[52] J.-Y. Antoine, J. Villaneau, and A. Lefeuvre, "Weighted krippendorff's alpha is a more reliable metrics for multi-coders ordinal annotations: Experimental studies on emotion, opinion and coreference annotation," in *Proc. 14th Conf. Eur. Chapter Assoc. Comput. Linguistics*, 2014, pp. 550–559. [Online]. Available: http://dblp.uni-trier.de/db/conf/eacl/eacl2014.html#AntoineVL14

[53] J. Z. Bergman, J. R. Rentsch, E. E. Small, S. W. Davenport, and S. M. Bergman, "The shared leadership process in decision-making teams," *J. Social Psychology*, vol. 152, no. 1, pp. 17–42, 2012.

[54] E. Ammerlaan, W. Veninga, and A. Zaidman, "Old habits die hard: Why refactoring for understandability does not give immediate benefits," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering*, 2015, pp. 504–507.

[55] J. K. Chhabra, K. K. Aggarwal, and Y. Singh, "Code and data spatial complexity: Two important software understandability measures," *Inf. Softw. Technol.*, vol. 45, no. 8, pp. 539–546, 2003. [Online]. Available: http://dblp.uni-trier.de/db/journals/infsof/infsof45.html#ChhabraAS03

[56] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 633–649, Jul. 2014.

[57] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, May 2012.

[58] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Reading, MA, USA: Addison-Wesley, 2012. [Online]. Available: https://books.google.com/books?id=-II73rBDXCYC

[59] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Proc. Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 101–110.

[60] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Inf. Softw. Technol.*, vol. 92, pp. 223–235, 2017.

[61] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Trans. Softw. Eng.*, 2017, doi: 10.1109/TSE.2017.2770122.

[62] S. Wasserman and K. Faust, *Social Network Analysis. Methods and Applications.* Cambridge, U.K.: Cambridge Univ. Press, 1994.

[63] F. Koerver and B. Ruler, "The relationship between corporate identity structures and communication structures," *J. Commun. Manage.*, vol. 7, no. 3, pp. 197–208, 2003.

[64] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle, "From developer networks to verified communities: A fine-grained approach," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 563–573.

[65] A. Lancichinetti, F. Radicchi, J. J. Ramasco, and S. Fortunato, "Finding statistically significant communities in networks," *PLoS One*, vol. 6, no. 4, Apr. 2011, Art. no. e18961.

[66] S. Magnoni, "An approach to measure community smells in software development communities," *Copyright - Politecnico di Milano Master Thesis Series*, 2016. [Online]. Available: https://tinyurl.com/CodeFace4SmellsTR

[67] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM J. Res. Develop.*, vol. 56, no. 5, pp. 9:1–9:13, Sep. 2012.

[68] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. Syst. Softw.*, vol. 80, no. 7, pp. 1120–1128, Jul. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2006.10.018

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

[69] B. Vasilescu, A. Serebrenik, and M. van den Brand, "By no means: A study on aggregating software metrics," in *Proc. 2nd Int. Workshop Emerging Trends Softw. Metrics*, 2011, pp. 23–26. [Online]. Available: http://doi.acm.org/10.1145/1985374.1985381

[70] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Softw.*, vol. 12, no. 4, pp. 52–62, Jul. 1995.

[71] Y. Zhou, H. Leung, and B. Xu, "Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness," *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 607–623, Sep. 2009.

[72] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994. [Online]. Available: http://dx.doi.org/10.1109/32.295895

[73] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proc. Int. Conf. Softw. Maintenance*, 1998, pp. 24–31.

[74] V. Kovalenko, F. Palomba, and A. Bacchelli, "Mining file histories: Should we consider branches?" in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 202–213.

[75] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 5–24, Jan. 2018.

[76] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88.

[77] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2007, pp. 34–43. [Online]. Available: http://dblp.uni-trier.de/db/conf/kbse/ase2007.html#HooimeijerW07

[78] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1109/TSE.2005.49

[79] L. Pascarella, F.-X. Geiger, F. Palomba, D. Di Nucci, I. Malavolta, and A. Bacchelli, "Self-reported activities of android developers," in *Proc. 5th IEEE/ACM Int. Conf. Mobile Softw. Eng. Syst.*, 2018, pp. 144–155.

[80] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng.*, May 2007, pp. 96–105.

[81] L. Williams and R. R. Kessler, *Pair Programming Illuminated*. Reading, MA, USA: Addison Wesley, 2003.

[82] G. Avelino, L. T. Passos, A. C. Hora, and M. T. Valente, "A novel approach for estimating truck factors," in *Proc. 24th IEEE Int. Conf. Program Comprehension*, 2016, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICPC.2016.7503718

[83] M. M. Ferreira, M. T. Valente, and K. A. M. Ferreira, "A comparison of three algorithms for computing truck factors," in *Proc. 25th Int. Conf. Program Comprehension*, 2017, pp. 207–217. [Online]. Available: https://doi.org/10.1109/ICPC.2017.35

[84] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams, "Using software repositories to investigate socio-technical congruence in development projects," in *Proc. Int. Workshop Mining Softw. Repositories*, 2007, Art. no. 25. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MSR.2007.33

[85] B. Lin, G. Robles, and A. Serebrenik, "Developer turnover in global, industrial open source projects: Insights from applying survival analysis," in *Proc. 12th Int. Conf. Global Softw. Eng.*, 2017, pp. 66–75.

[86] M. Nassif and M. P. Robillard, "Revisiting turnover-induced knowledge loss in software projects," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 261–272.

[87] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus, "Quantifying and mitigating turnover-induced knowledge loss: Case studies of Chrome and a project at Avaya," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 1006–1016.

[88] I. G. Macdonald, *Symmetric Functions and Hall Polynomials*. London, U.K.: Oxford Univ. Press, 1998.

[89] E. Constantinou and T. Mens, "Socio-technical evolution of the ruby ecosystem in github," in *Proc. IEEE 24th Int. Conf. Softw. Anal. Evol. Reengineering*, 2017, pp. 34–44. [Online]. Available: https://doi.org/10.1109/SANER.2017.7884607

[90] R. J. J. M. van den Eijnden, J. S. Lemmens, and P. M. Valkenburg, "The social media disorder scale," *Comput. Human Behavior*, vol. 61, pp. 478–487, 2016. [Online]. Available: http://dblp.uni-trier.de/db/journals/chb/chb61.html#EijndenLV16

[91] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. 7th ACM SIGCOMM Conf. Internet Meas.*, 2007, pp. 29–42.

[92] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986. [Online]. Available: http://dx.doi.org/10.1023/A:1022643204877

[93] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artif. Intell.*, vol. 97, no. 1/2, pp. 273–324, 1997.

[94] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Enhancing change prediction models using developer-related factors," *J. Syst. Softw.*, vol. 143, pp. 14–28, 2018.

[95] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al., "Top 10 algorithms in data mining," *Knowl. Inf. Syst.*, vol. 14, no. 1, pp. 1–37, 2008.

[96] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explorations Newslett.*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

[97] A. J. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, vol. 30, pp. 507–512, 1974.

[98] C. Gates and J. Bilbro, "Illustration of a cluster analysis method for mean separation 1," *Agronomy J.*, vol. 70, no. 3, pp. 462–465, 1978.

[99] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *J. Amer. Statistical Assoc.*, vol. 32, no. 200, pp. 675–701, 1937.

[100] S. G. Carmer and W. M. Walker, "Pairwise multiple comparisons of treatment means in agronomic research," *J. Agron. Educ.*, vol. 14, pp. 19–26, 1985.

[101] W. Conover, *Practical Nonparametric Statistics*. Hoboken, NJ: Wiley, 1998. [Online]. Available: http://www.amazon.com/exec/obidos/ASIN/0471160687/qid%3D1105043505/sr%3D11-1/ref%3Dsr%5F11%5F1/102-3876594-0537723

[102] R. J. Grissom and J. J. Kim, *Effect Sizes for Research: A Broad Practical Approach*, 2nd ed. Mahwah, NJ, USA: Lawrence Earlbaum Associates, 2005.

[103] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, no. Jan, pp. 1–30, 2006.

[104] R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Quality Quantity*, vol. 41, no. 5, pp. 673–690, Oct. 2007. [Online]. Available: https://doi.org/10.1007/s11135–006-9018-6

[105] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Eng.*, vol. 17, no. 4/5, pp. 531–577, 2012.

[106] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, vol. 1, pp. 789–800.

[107] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.

[108] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, 2016.

[109] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[110] P. Baldi, S. Brunak, Y. Chauvin, C. A. Andersen, and H. Nielsen, "Assessing the accuracy of prediction algorithms for classification: an overview," *Bioinf.*, vol. 16, no. 5, pp. 412–424, 2000.

[111] G. W. Brier, "Verification of Forecasts expressed in terms of probability," *Monthly Weather Review*, vol. 78, no. 1, pp. 1–3, Jan. 1950.

[112] J. Finch, "The vignette technique in survey research," *Sociology*, vol. 21, no. 1, pp. 105–114, 1987.

[113] D. A. Almeida, G. C. Murphy, G. Wilson, and M. Hoye, "Do software developers understand open source licenses?" in *Proc. 25th Int. Conf. Program Comprehension*, 2017, pp. 1–11. [Online]. Available: https://doi.org/10.1109/ICPC.2017.7

[114] R. Likert, "A technique for the measurement of attitudes," *Archives Psychology*, vol. 22, no. 140, p. 55, 1932.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

[115] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas.*, 2009, pp. 390–400. [Online]. Available: http://dx.doi.org/10.1109/ESEM.2009.5314231

[116] A. Tahmid, N. Nahar, and K. Sakib, "Understanding the evolution of code smells by observing code smell clusters," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, vol. 4, pp. 8–11, 2016.

[117] D. M. W. Powers, "What the f-measure doesn't measure: Features, flaws, fallacies and fixes," *CoRR*, vol. abs/1503.06410, 2015. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1503.html#Powers15

[118] G. Catolino, "Does source code quality reflect the ratings of apps?" in *Proc. 5th Int. Conf. Mobile Softw. Eng. Syst.*, 2018, pp. 43–44.

[119] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Inf. Softw. Technol.*, vol. 99, pp. 1–10, 2018.

[120] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Inf. Softw. Technol.*, no. 105, pp. 43–55, 2019.

[121] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 440–451.

[122] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 1–12.

[123] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *J. Syst. Softw.*, vol. 86, no. 6, pp. 1498–1516, 2013.

[124] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proc. Int. Conf. Softw. Maintenance Evol.*, 2018.

[125] F. Palomba, A. Zaidman, and A. Lucia, "Automatic test smell detection using information retrieval techniques," in *Proc. Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 311–322.

[126] W. Cunningham, "The wycash portfolio management system," *SIGPLAN OOPS Mess.*, vol. 4, no. 2, pp. 29–30, Dec. 1992. [Online]. Available: http://doi.acm.org/10.1145/157710.157715

[127] A. Gupta, B. Suri, V. Kumar, S. Misra, T. Blazauskas, and R. Damasevicius, "Software code smell prediction model using shannon, rnyi and tsallis entropies," *Entropy*, vol. 20, no. 5, 2018, Art. no. 372. [Online]. Available: http://dblp.uni-trier.de/db/journals/entropy/entropy20.html#GuptaSKMBD18

[128] B. Pietrzak and B. Walter, "Leveraging code smell detection with inter-smell relations," in *Proc. 7th Int. Conf. Extreme Program. Agile Processes Softw. Eng.*, 2006, pp. 75–84. [Online]. Available: http://dblp.uni-trier.de/db/conf/xpu/xp2006.html#PietrzakW06

[129] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 6:1–6:44, 2014. [Online]. Available: http://dblp.uni-trier.de/db/journals/tosem/tosem24.html#SahinKBD14

[130] F. Khomh, S. Vaucher, Y. Guéhéneuc, and H. A. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proc. 9th Int. Conf. Quality Softw.*, 2009, pp. 305–314. [Online]. Available: http://dblp.uni-trier.de/db/conf/qsic/qsic2009.html#KhomhVGS09

[131] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *J. Softw.: Evol. Process*, vol. 27, no. 11, pp. 867–895, 2015. [Online]. Available: http://dblp.uni-trier.de/db/journals/smr/smr27.html#RasoolA15

[132] M. Cataldo and S. Nambiar, "The impact of geographic distribution and the nature of technical coupling on the quality of global software development projects," *J. Softw.: Evol. Process*, vol. 24, no. 2, pp. 153–168, 2012.

[133] S. Datta, R. Sindhgatta, and B. Sengupta, "Evolution of developer collaboration on the jazz platform: A study of a large scale agile project," in *Proc. 4th India Softw. Eng. Conf.*, 2011, pp. 21–30.

[134] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, "Putting It All Together: Using Socio-technical Networks to Predict Failures," in *Proc. 2009 20th Int. Symposium Software Reliability Engineering*, Mysuru, Karnataka, 2009, pp. 109–119.

[135] D. A. Tamburri, F. Palomba, A. Serebrenik, and A. Zaidman, "Discovering community patterns in open-source: A systematic approach and its evaluation," *Empirical Softw. Eng.*, pp. 1–49, 2018.

[136] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2008, pp. 2–11. [Online]. Available: http://portal.acm.org/citation.cfm?id=1414008

[137] S. B. Fonseca, C. R. De Souza, and D. F. Redmiles, "Exploring the relationship between dependencies and coordination to support global software development projects," in *Proc. Int. Conf. Global Softw. Eng.*, 2006, pp. 243–243.

[138] I. Kwan, A. Schroter, and D. Damian, "Does socio-technical congruence have an effect on software build success? a study of coordination in a software project," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 307–324, May 2011.

[139] F. Palomba, D. A. Tamburri, A. Serebrenik, A. Zaidman, F. A. Fontana, and R. Oliveto, "How do community smells influence code smells?" in *Proc. 40th Int. Conf. Softw. Eng.: Companion Proc.*, 2018, pp. 240–241.

[140] S. Herbold, "Training data selection for cross-project defect prediction," in *Proc. 9th Int. Conf. Predictive Models Softw. Eng.*, 2013, Art. no. 6.

[141] B. Pang, L. Lee, et al., "Opinion mining and sentiment analysis," *Found. Trends® Inf. Retrieval*, vol. 2, no. 1–2, pp. 1–135, 2008.

[142] R. Jongeling, P. Sarkar, S. Datta, and A. Serebrenik, "On negative results when using sentiment analysis tools for software engineering research," *Empirical Softw. Eng.*, vol. 22, no. 5, pp. 2543–2584, 2017.

[143] N. Novielli, D. Girardi, and F. Lanubile, "A benchmark study on sentiment analysis for software engineering research," *arXiv preprint arXiv: 1803.06525*, 2018.

[144] G. Scanniello, D. Lo, and A. Serebrenik, Eds., *Proc. 25th Int. Conf. Program Comprehension*, 2017. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7959731

Fabio Palomba is a Senior Research Associate at the University of Zurich, Switzerland. He received the PhD degree in Management & Information Technology from the University of Salerno, Italy, in 2017. His PhD Thesis has been awarded with the IEEE Computer Society Best 2017 PhD Award. His research interests include software maintenance and evolution, empirical software engineering, source code quality, and mining software repositories. He was also the recipient of Distinguished and Best Paper Awards at ASE'13, ICSE'15, ICSME'17, SANER'18, and CSCW'18. He serves and has served as a program committee member of various international conferences and as referee for flagship journals in the fields of software engineering. Since 2016 he is Review Board Member of EMSE. He was the recipient of several Distinguished Reviewer Awards.

Damian Andrew Tamburri is an assistant professor with the Jheronimus Academy of Data Science (JADS) and the Technical University of Eindhoven (TU/e). His research interests include social software engineering, advanced software architectures, design, and analysis tools as well as advanced software-engineering methods and analysis tools. He is on the IEEE Software editorial board and is secretary of the International Federation for Information Processing Working Group on Service-Oriented Computing. He is a member of the IEEE.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.

Francesca Arcelli Fontana received the PhD degree in computer science from the University of Milano, Italy. She is currently in the position of associate professor with the University of Milano Bicocca. The actual research activity principally concerns the software engineering field, in particular in software evolution, software quality assessment, model-drive reverse engineering, managing technical debt, design patterns and architectural smells detection. She is the head of the Software Evolution and Reverse Engineering Lab (http://essere.disco.unimib.it/), University of Milano Bicocca. She is a member of the IEEE and IEEE Computer Society.

Rocco Oliveto is an associate professor with the University of Molise (Italy), where he is also the chair of the Computer Science Bachelor and Master programs and the director of the Software and Knowledge Engineering Lab (STAKE Lab). He coauthored more than 100 papers on topics related to software traceability, software maintenance and evolution, search-based software engineering, and empirical software engineering. His activities span various international software engineering research communities. He has served as organizing and program committee member of several international conferences in the field of software engineering. He was program co-chair of ICPC 2015, TEFSE 2015 and 2009, SCAM 2014, WCRE 2013 and 2012. He was also keynote speaker at MUD 2012. He is also one of the cofounders and CEO of datasounds, a spin-off of the University of Molise aiming at efficiently exploiting the priceless heritage that can be extracted from big data analysis. He is a member of the IEEE.

Andy Zaidman received the MSc and PhD degrees in computer science from the University of Antwerp, Belgium, in 2002 and 2006, respectively. He is an associate professor with the Delft University of Technology, The Netherlands. His main research interests include software evolution, program comprehension, mining software repositories and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE'08, WCRE'09, VISSOFT'14 and MSR'18). In 2013 he was the laureate of a prestigious Vidi career grant from the Dutch science foundation NWO. He is a member of the IEEE.

Alexander Serebrenik received the MSc degree in computer science from the Hebrew University, Jerusalem, Israel, and the PhD degree in computer science from Katholieke Universiteit Leuven, Belgium, in 2003. He is an associate professor of Model-Driven Software Engineering, Eindhoven University of Technology. His areas of expertise include software evolution, maintainability and reverse engineering, program analysis and transformation, process modeling and verification. He is involved in organisation of such conferences as ICSME, MSR, ICPC and SANER. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: TU Delft Library. Downloaded on January 25,2021 at 08:44:36 UTC from IEEE Xplore. Restrictions apply.