The Journal of Systems and Software 152 (2019) 165–181

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Not all bugs are the same: Understanding, characterizing, and classifying bug types

Gemma Catolino[a,*], Fabio Palomba[b], Andy Zaidman[c], Filomena Ferrucci[a]

[a] University of Salerno, Italy
[b] University of Zurich, Switzerland
[c] Delft University of Technology, the Netherlands

ARTICLE INFO

Article history:
Received 6 July 2018
Revised 4 March 2019
Accepted 6 March 2019
Available online 7 March 2019

Keywords:
Bug classification
Taxonomy
Empirical study

ABSTRACT

Modern version control systems, e.g., *GitHub*, include bug tracking mechanisms that developers can use to highlight the presence of bugs. This is done by means of bug reports, i.e., textual descriptions reporting the problem and the steps that led to a failure. In past and recent years, the research community deeply investigated methods for easing bug triage, that is, the process of assigning the fixing of a reported bug to the most qualified developer. Nevertheless, only a few studies have reported on how to support developers in the process of understanding the type of a reported bug, which is the first and most time-consuming step to perform before assigning a bug-fix operation. In this paper, we target this problem in two ways: first, we analyze 1280 bug reports of 119 popular projects belonging to three ecosystems such as Mozilla, Apache, and Eclipse, with the aim of building a taxonomy of the types of reported bugs; then, we devise and evaluate an automated classification model able to classify reported bugs according to the defined taxonomy. As a result, we found nine main common bug types over the considered systems. Moreover, our model achieves high F-Measure and AUC-ROC (64% and 74% on overall, respectively).

1. Introduction

The year 2017 has been earmarked as *The Year That Software Bugs Ate The World*.[1] It serves as an apt reminder that software engineers are but human, and have their fallacies when it comes to producing bug-free software. With modern software systems growing in size and complexity, and developers having to work under frequent deadlines, the introduction of bugs does not really come as a surprise.

Users of such faulty software systems have the ability to report back software failures, either through dedicated issue tracking systems or through version control platforms like GitHub. In order to do so, a user files a so-called *bug report*, which contains a textual description of the steps to perform in order to reproduce a certain failure (Breu et al., 2010; Hooimeijer and Weimer, 2007).

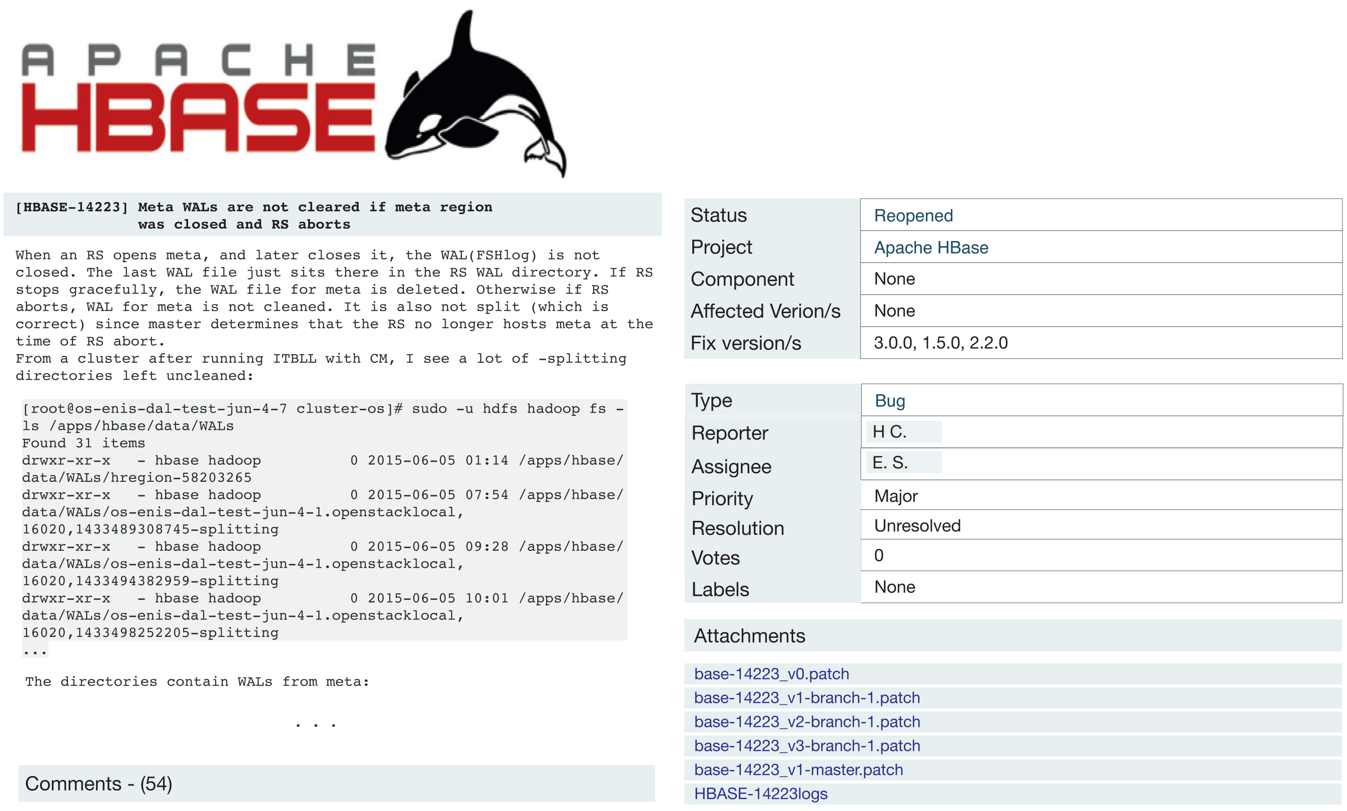
Once a failure is known and reported, the bug localization and fixing process starts (Zeller, 2009; Beller et al., 2018). Developers are requested to (i) analyze the bug report, (ii) identify the bug type, e.g., if it is a security- or a performance-related one, and (iii) assign its verification and resolution to the most qualified developer (Zhang and Lee, 2013). The research community proposed methodologies and tools to identify who should fix a certain bug (Anvik, 2006; Anvik et al., 2006; Anvik and Murphy, 2011; Jeong et al., 2009; Murphy and Cubranic, 2004; Xuan et al., 2015; 2017), thus supporting developers once they have diagnosed the bug type that they have to deal with.

However, there is still a lack of approaches able to support developers while analyzing a bug report in the first instance. As a matter of fact, understanding the bug type represents the first and most time-consuming step to perform in the process of bug triage (Akila et al., 2015), since it requires an in-depth analysis of the characteristics of a newly reported bug report. Unfortunately, such a step is usually performed manually before the assignment of a developer to the bug fix operation (Breu et al., 2010). Perhaps more importantly, most of the research approaches aimed at supporting the bug triage process treat all bugs in the same manner, without considering their type (Zaman et al., 2011).

We argue that the definition of approaches able to support developers in the process of understanding the bug types can be beneficial to properly identify the developer who should be assigned to its debugging, speeding-up the bug analysis and resolution process.

* Corresponding author.
E-mail addresses: gcatolino@unisa.it (G. Catolino), palomba@ifi.uzh.ch (F. Palomba), a.e.zaidman@tudelft.nl (A. Zaidman), fferrucci@unisa.it (F. Ferrucci).
[1] https://tinyurl.com/y8n4kxgw, last visited April 17, 2018.

https://doi.org/10.1016/j.jss.2019.03.002
0164-1212/© 2019 Elsevier Inc. All rights reserved.


[HBASE-14223] Meta WALs are not cleared if meta region was closed and RS aborts

```
When an RS opens meta, and later closes it, the WAL(FSHlog) is not
closed. The last WAL file just sits there in the RS WAL directory. If RS
stops gracefully, the WAL file for meta is deleted. Otherwise if RS
aborts, WAL for meta is not cleaned. It is also not split (which is
correct) since master determines that the RS no longer hosts meta at the
time of RS abort.
From a cluster after running ITBLL with CM, I see a lot of -splitting
directories left uncleaned:

 [root@os-enis-dal-test-jun-4-7 cluster-os]# sudo -u hdfs hadoop fs -
 ls /apps/hbase/data/WALs
 Found 31 items
 drwxr-xr-x   - hbase hadoop          0 2015-06-05 01:14 /apps/hbase/
 data/WALs/hregion-58203265
 drwxr-xr-x   - hbase hadoop          0 2015-06-05 07:54 /apps/hbase/
 data/WALs/os-enis-dal-test-jun-4-1.openstacklocal,
 16020,1433489308745-splitting
 drwxr-xr-x   - hbase hadoop          0 2015-06-05 09:28 /apps/hbase/
 data/WALs/os-enis-dal-test-jun-4-1.openstacklocal,
 16020,1433494382959-splitting
 drwxr-xr-x   - hbase hadoop          0 2015-06-05 10:01 /apps/hbase/
 data/WALs/os-enis-dal-test-jun-4-1.openstacklocal,
 16020,1433498252205-splitting
 ...

 The directories contain WALs from meta:

                    . . .
```

Comments - (54)

Status	Reopened
Project	Apache HBase
Component	None
Affected Verion/s	None
Fix version/s	3.0.0, 1.5.0, 2.2.0

Type	Bug
Reporter	H C.
Assignee	E. S.
Priority	Major
Resolution	Unresolved
Votes	0
Labels	None

Attachments

base-14223_v0.patch
base-14223_v1-branch-1.patch
base-14223_v2-branch-1.patch
base-14223_v3-branch-1.patch
base-14223_v1-master.patch
HBASE-14223logs

Fig. 1. Bug reported and reopened in Apache HBase.

1.1. Motivating example

To support our statement, let us consider a real bug report from Apache HBase,[2] one of the projects taken into account in our study. This project implements a scalable distributed big data store able to host large relational tables atop clusters of commodity hardware. On August 15th, 2015 the bug report shown in Fig. 1 was created.[3]

The developer who opened the report (i.e., H. C.) encountered an issue related to a network-related problem. Specifically, due to the wrong management of the so-called World Atlas of Language Structures (WALS), a large set of structural (phonological, grammatical, lexical) properties of languages gathered from descriptive materials (such as reference grammars). Specifically, when a Region Server (RS) aborts its operations, the directory containing the WAL data is not cleaned, causing possible data incoherence or inconsistency issues. Looking at the change history information of the system, the class `HRegionServer`—the file containing the reported bug—has been mostly modified by developer G. C., who was indeed assigned to the resolution of this bug report on May 30, 2017. Such an assignment is in line with the recommendations provided by existing bug triaging approaches (Shokripour et al., 2013; Tian et al., 2016), that would suggest G. C. as an assignee since he has a long experience with this class. However, *not all bugs are the same*: a more careful analysis of the types of changes applied by G. C. reveals that they were mainly focused on the configuration of the server rather than on the communication with the client. As a result, the bug was marked as ‘`resolved`’ on September 17, 2017: however, the bug was not actually fixed and was reopened on October 6, 2018. This indicates that the experience of a developer on a certain class—taken as relevant factor within existing bug triaging approaches (Shokripour et al., 2013; Tian et al., 2016)—might not be enough for recommending the most qualified developer to fix a bug. In other words, we conjecture that understanding the type of a newly reported bug might be beneficial for bug triage. At the same time, it might reduce the phenomenon of bug tossing (Jeong et al., 2009)—which arises when developers re-assign previously assigned bugs to others, as in the example above—by allowing a more correct bug assignment.

1.2. Our work and contributions

In this paper, we aim to perform the first step towards the (i) empirical understanding of the possible bug types and (ii) automated support for their classification. To that end, we first propose a novel taxonomy of bug types, that is built on the basis of an iterative content analysis conducted on 1280 bug reports of 119 software projects belonging to three large ecosystems such as Mozilla, Apache, and Eclipse. To the best of our knowledge, this is the first work that proposes a general taxonomy collecting the main bug types. This also enables the construction of a dataset of labeled bug reports that can be further exploited.

In the second place, we build an automated approach to classify bug reports according to their type; in other words, we built our classification model training our classifier using the textual content of the bug report to predict its type.

We empirically evaluate the classification model by running it against the dataset coming as output of the taxonomy building phase, measuring its performance adopting a 100 times 10-fold cross validation methodology in terms of F-Measure, AUC-ROC, and Matthew’s Correlation Coefficient (MCC).

The results of the study highlight nine different types reported in bug reports, that span across a broad set of issues (e.g., GUI-related vs Configuration bugs) and are widespread over the

[2] https://hbase.apache.org .
[3] The full version of the bug report is available here: https://goo.gl/rS8iQU.

considered ecosystems. In addition, the classification model shows promising results, as it is able to classify the bug types with an F-Measure score of 64%.

To sum up, the contributions made by this paper are:

1. A *taxonomy* reporting the common bug types raised through bug reports, and that has been manually built considering a large corpus of existing bug reports;
2. An in-depth analysis of the characterization of the different bug types discovered. In particular, we took into account three different perspectives such as frequency, relevant topics discussed, and the time needed to fix each bug type.
3. A novel *bug type classification model* to automatically classify reported bugs according to the defined taxonomy.
4. A large *dataset* and a replication package (Catolino et al., 2018) that can be used by the research community to further study the characteristics of bug reports and bugs they refer to.

The remainder of the paper is as follows. Section 2 overviews the related literature on bug report analysis and classification. Section 3 describes the research methodology adopted to conduct our study, while in Section 4 we report the achieved results. In Section 5 we deeper discuss our findings and the implications of our work. Section 6 examines the threats to the validity of the study and the way we mitigated them. Finally, Section 7 concludes the paper and provides insights into our future research agenda.

2. Background and related work

Our work revolves around the problem of classifying bugs according to their type with the aim of supporting and possibly speeding-up bug triaging activities. Thus, we focus this section on the discussion of the literature related to bug classification studies and approaches. A comprehensive overview of the research conducted in the context of bug triaging is presented by Zhang et al. (2016).

2.1. Bug classification schemas

The earliest and most popular bug classification taxonomy was proposed by IBM (Chillarege et al., 1992), which introduced the so-called *Orthogonal Defect Classification* (ODC). This taxonomy includes 13 categories and classifies bugs in terms of the involved program structure: as an example, a bug is assigned to the category “Assignment” in case it has to do with a problem occurring in a statement where a variable assignment is performed; similarly, the category “Function” is the one characterizing bugs located in the code implementing a function. As such, ODC can indicate the program structure involved in the bug, but *not the type of the issue*. For instance, a bug of type “Assignment” can be both caused by a performance or a security issue. Thus, the 13 categories included in ODC can be related to any of the higher-level categories discovered in this paper (see Section 4 for further details): on the one hand, this excludes the possibility of an abstraction, i.e., the 13 categories of ODC cannot be matched/related to the categories we discovered; on the other hand, we argue that the two taxonomies can be used in a *complementary* manner, e.g., a developer can first use our taxonomy to understand the type of bug that occurred and then refine the process by using ODC to characterize the program structure causing that bug type. In this sense, we argue that the proposed taxonomy represents a novel contribution that can further and complementarily support developers when understanding the cause behind a newly identified defect.

Another popular bug characterization schema was developed by Hewlett-Packard (Freimut et al., 2005). In this case, bugs are characterized by three attributes: (i) “origin”, that is the activity in which the defect was introduced (e.g., during requirement specification or design); (ii) “mode”, which describes the scenarios leading to a bug; and (iii) “type”, that describes more in-depth the origin of a bug, by specifying if it is hardware- or software-related. It is important to note that the attribute “type” of this classification schema is not intended to be used for the specification of the bug type (e.g., a performance issue), but rather it provides more context on the location of a bug.

More recent works defined ad hoc taxonomies (i) for specific application types or (ii) aiming at characterizing particular bug types. As for the former category, Chan et al. (2007) proposed a taxonomy that captures the possible failures that arise in Web service composition systems, while Bruning et al. (2007) provided a corresponding fault taxonomy for service-oriented architectures according to the process of service invocation. Similarly, Ostrand and Weyuker (1984) conducted an industrial study involving an interactive special-purpose editor system, where a group of developers were asked to categorize 173 bugs based on the error they referred to: as a final result, a taxonomy was built. Lal and Sureka (2012) analyzed commonalities and differences of seven different types of bug reports within Google; they provided guidelines to categorize their bug reports. Moreover, recent studies (Leszak et al., 2002; Buglione and Abran, 2006) showed how developers manually classify defects into the ODC categories based on the reported descriptions using, for example, root cause defect analysis.

As for the second category (related to the characterization of particular bug types), Aslam et al. (1996) defined a classification of security faults in the Unix operating system. More recently, Zhang et al. (2018) analyzed the symptoms and bug types of 175 TensorFlow coding bugs from GitHub issues and StackOverflow questions. As a result, they proposed a number of challenges for their detection and localization. Tan et al. (2014) proposed work closest to ours: they started from the conjecture that three bug types, i.e., semantic, security, and concurrency issues, are at the basis of most relevant bugs in a software system. Thus, they investigated the distribution of these types in projects such as Apache, Mozilla, and Linux. Finally, they performed a fine-grained analysis on the impact and evolution of such bugs on the considered systems; they proposed a machine learning approach using all the information of a bug report to automatically classify semantic, security, and concurrency bugs and having an average F-Measure of ≈70%. As opposed to the work by Tan et al. (2014), we start our investigation without any initial conjecture: as such, we aim at providing a wider overview of the bug types and their diffusion on a much larger set of systems (119 vs 3); furthermore, we aim at producing a high-level bug taxonomy that is *independent* from a specific type of system, thus being generically usable. Finally, the presented bug type classification model is able to automatically classify all the identified bug types, thus providing a wider support for developers.

2.2. Bug classification techniques

Besides classification schemas, a number of previous works devised automated approaches for classifying bug reports. Antoniol et al. (2008) defined a machine learning model to discriminate between bugs and new feature requests in bug reports, reporting a precision of 77% and a recall of 82%. In our case, we only consider bug reports actually reporting issues of the considered applications, since our goal is to classify bugs. Hernández-González et al. (2018) proposed an approach for classifying the impact of bugs according to the ODC taxonomy (Chillarege et al., 1992): the empirical study conducted on two systems, i.e., *Compendium* and *Mozilla*, showed good results. At same time, Huang et al. (2015), based on the ODC classification, proposed AutoODC, an approach for automating ODC classification by casting it as a supervised text classification problem and integrating experts' ODC experience and domain knowledge; they built two models trained with two different classifiers such as Naive Bayes and Support Vector Machine on a larger defect list extracted from FileZilla. They reported promising results. With respect to this work, our paper aims at providing a more comprehensive characterization of the bug types, as well as providing an automated solution for tagging them.

Thung et al. (2012) proposed a classification-based approach that can automatically classify defects into three super-categories that are comprised of ODC categories: *control and data flow, structural*, and *non-functional*. In a follow-up work (Thung et al., 2015), they extended the defect categorization. In particular, they combined clustering, active learning and semi-supervised learning algorithms to automatically categorize defects; they first picked an initial sample, extract the examples that are more informative for training the classification model, and incrementally refining the trained model. They evaluated their approach on 500 defects collected from JIRA repositories of three software systems. Xia et al. (2014) applied a text mining technique in order to categorize defects into fault trigger categories by analyzing the natural-language description of bug reports, evaluating their solution on 4 datasets, e.g., Linux, Mysql, for a total of 809 bug reports. Nagwani et al. (2013) proposed an approach for generating the taxonomic terms for software bug classification using LDA, while Zhou et al. (2016) combined text mining on the defect descriptions with structured data (e.g., priority and severity) to identify corrective bugs. Furthermore, text-categorization based machine learning techniques have been applied for bug triaging activities (Murphy and Cubranic, 2004; Javed et al., 2012) with the aim of assigning bugs to the right developers. With respect to the works mentioned above, our paper reinforces the idea of using natural language processing to automatically identify the bug types; nevertheless, we provide a more extensive empirical analysis of the types of bugs occurring in modern software systems, as well as their categorization according to different perspectives such as frequency, relevant topics, and time required to be fixed.

On the basis of these works, in the context of our research we noticed that there is a lack of studies that try to provide automatic support for the labeling of bugs according to their type: for this reason, our work focuses on this aspect and tries to exploit a manually built taxonomy of bug types to accomplish this goal.

3. Research methodology

In this section, we report the empirical study definition and design that we follow in order to create a bug type taxonomy and provide a bug type classification model. From a high-level perspective, our methodology is *exploratory* in nature (Stebbins, 2001), and, as such, enabled us to freely explore the problem without predefined conjectures and/or hypotheses. For this reasons, we do not formulate any kind of hypothesis on type, distribution, and characteristics of different types of defects.

3.1. Research questions

The *goal* of the study is threefold: (i) *understanding* which types of bugs affect modern software systems, (ii) characterizing them to better describe their nature, and (iii) *classifying* bug reports according to their type. The *purpose* is that of easing the maintenance activity related to bug triaging, thus improving the allocation of resources, e.g., assigning a bug to the developer that is more qualified to fix a certain type of issue. The *quality focus* is on the comprehensiveness of the bug type taxonomy as well as on the accuracy of the model in classifying the bug types. The *perspective* is that of both researchers and practitioners: the former are interested in a deeper understanding of the bug types occurring in software systems, while the latter in evaluating the applicability of bug type prediction models in practice. The specific research questions formulated in this study are the following:

- **RQ$_1$**: *To what extent can bug types be categorized through the information contained in bug reports?*
- **RQ$_2$**: *What are the characteristics, in terms of frequency, topics, and bug fixing time, of different bug types?*
- **RQ$_3$**: *How effective is our classification model in classifying bugs according to their type exploiting bug report information?*

In RQ$_1$ our goal is to categorize the bug types through the analysis of bug reports that are reported in bug tracking platforms. Second, in RQ$_2$ we analyze (i) frequency, (ii) relevant topics, and (iii) bug fixing time of each category with the aim of characterizing them along these three perspectives. Finally, in RQ$_3$ we investigate how effectively the categories of bug types can be automatically classified starting from bug reports via standard machine learning techniques, so that developers and project managers can be automatically supported during bug triaging. In the following subsections, we detail the design choices that allow us to answer our research questions.

3.2. Context selection

In order to answer our research questions, we first needed to collect a large number of bug reports from existing software projects. To this aim, we took into account bug reports of three software ecosystems such as Mozilla,[4] Apache,[5] and Eclipse.[6] The selection of these systems was driven by the results achieved in previous studies (Bettenburg et al., 2007; Sun et al., 2010; Zimmermann et al., 2010), which reported the high-quality of their bug reports in terms of completeness and understandability. We randomly sampled 1280 bug reports that were 'fixed' and 'closed': as also done in previous work (Tan et al., 2014), we included them because they have all the information required for understanding the bug types (e.g., developers' comments or attachments). It is important to note that we checked and excluded from the random sampling the so-called *misclassified* bug reports, i.e., those that do not contain actual bugs (Antoniol et al., 2008; Herzig et al., 2013), by exploiting the guidelines provided by Herzig et al. (2013). In the end, our dataset is composed of bug reports from 119 different projects of the considered ecosystems.

Table 1
Characteristics of ecosystems in our dataset.

Ecosystem	Project	Bug reports
Apache	60	406
Eclipse	0.939	0.9444
Mozilla	20	430
Overall	119	1280

Table 1 contains for each ecosystem the (i) number of projects we considered, and (ii) number of bug reports taken into account. The final dataset is available in our online appendix (Catolino et al., 2018).

3.3. RQ$_1$: toward a taxonomy of bug types

To answer our first research question, we conducted three iterative *content analysis sessions* (Lidwell et al., 2010) involving

[4] https://bugzilla.mozilla.org.
[5] https://bz.apache.org/bugzilla/.
[6] https://bugs.eclipse.org/bugs/.

two software engineering researchers, both authors of this paper, (1 graduate student and 1 research associate) with at least seven years of programming experience. From now on, we refer to them as *inspectors*. Broadly speaking, this methodology consisted of reading each bug report (both title and summary, which reports its detailed description), with the aim of assigning a label describing the bug type that the reported problem refers to. It is important to note that in cases where the bug report information was not enough to properly understand the type of the bug, we also considered patches, attachments, and source code of the involved classes, so that we can better contextualize the type of the bug by inspecting the modifications applied to fix it. The final goal was to build a *taxonomy* representing the bug types that occur during both software development and maintenance. In the following, we describe the methodology followed during the three iterative sessions, as well as how we validate the resulting taxonomy.

3.3.1. Taxonomy building

Starting from the set of 1280 bug reports composing our dataset, overall, each inspector *independently* analyzed 640 bug reports.

Iteration 1: The inspectors analyzed an initial set of 100 bug reports. Then, they opened a discussion on the labels assigned to the bug types identified so far and tried to reach consensus on the names and meaning of the assigned categories. The output of this step was a draft taxonomy that contains some obvious categories (e.g., security bugs), while others remain undecided.

Iteration 2: The inspectors first re-categorized the 100 initial bug reports according to the decisions taken during the discussion, then used the draft taxonomy as basis for categorizing another set of 200. This phase was for both assessing the validity of the categories coming from the first step (by confirming some of them and redefining others) and for discovering new categories. After this step was completed, the inspectors opened a new discussion aimed at refining the draft taxonomy, merging overlapping bug types or characterizing better the existing ones. A second version of the taxonomy was produced.

Iteration 3: The inspectors re-categorized the 300 bug reports previously analyzed. Afterward, they completed the final draft of the taxonomy verifying that each kind of bug type encountered in the final 339 bug reports was covered by the taxonomy.

Following this iterative process, we defined a taxonomy composed of 9 categories. It is important to note that at each step we computed the inter-rater agreement using the Krippendorff's alpha Kr_{α} (Bauer, 2007). During the sessions, the agreement measure ranged from 0.65, over 0.76, to 0.96 for the three iterative sessions, respectively. Thus, we can claim that the agreement increased over time and reached a considerably higher value than the 0.80 standard reference score usually considered for Kr_{α} (Antoine et al., 2014).

Taxonomy validation. While the iterative content analysis makes us confident about the comprehensiveness of the proposed taxonomy, we also evaluated it in an alternative way: specifically, we involved 5 industrial developers having more than 10 years of programming experience. They were all contacted via e-mail by one of the authors of this paper, who selected them from her personal contacts.

We provided them with an EXCEL file that contained a list of 100 bug reports randomly selected from the total 1139 in the dataset (we excluded 141 of them as explained in Section 4). Each developer analyzed a different set of bug reports and was asked to categorize bugs according to the taxonomy of bug types we previously built. During this step, the developers were allowed to either consult the taxonomy (provided in PDF format and containing a description of the bug types in our taxonomy similar to the one we discuss in Section 4.1) or assign new categories if needed.

Once the task was completed, the developers sent back the file annotated with their categorization. Moreover, we gathered comments on the taxonomy and the classification task. As a result, all the participants found the taxonomy *clear* and *complete*: as a proof of that, the tags they assigned were exactly the same as the ones assigned during the phase of taxonomy building.

3.4. RQ$_2$: characterizing different bug types

In the context of this research question, we aimed at providing a characterization of the different bug types discovered in RQ$_1$. More specifically, we took into account three different perspectives such as frequency, relevant topics discussed, and time needed to fix each bug type. In the following subsections, we report the methodology applied to address those perspectives.

Frequency analysis. To study this perspective, we analyzed how frequently each category of bug type in our taxonomy appears. We computed the frequency each bug report was assigned to a certain type during the iterative content analysis. It is worth noting that in our study a bug could not be assigned to multiple categories because of the granularity of the taxonomy proposed: we preferred, indeed, working at a level that allows its generalization over software systems having different scope and characteristics. In Section 4 we present and discuss bar plots showing the frequency of each category of bug type in the taxonomy.

Topics analysis. With this second investigation, we aimed at understanding what are the popular topics discussed within bug reports of different nature. To perform such an analysis, we exploited a well-known topic modeling approach called Latent Dirichlet Allocation (LDA) (Blei et al., 2003; Hecking and Leydesdorff, 2018). This is a topic-based clustering technique, which can be effectively used to cluster documents in the topics space using the similarities between their topics distributions (Wei and Croft, 2006). Specifically, for each category of our taxonomy, we adopted the following steps:

1. First, we extracted all the terms composing the bug reports of a certain category.
2. An Information Retrieval (IR) normalization process (Baeza-Yates and Ribeiro-Neto, 1999) was applied. In particular, as bug reports are written in natural language, we first applied (i) spelling correction, (ii) contractions expansion, (iii) nouns and verbs filtering, and (iv) singularization. Then, terms contained in the bug reports are transformed by applying the following steps: (i) separating composed identifiers using the camel case splitting, which splits words based on underscores, capital letters, and numerical digits; (ii) reducing to lower case letters of extracted words; (iii) removing special characters, programming keywords and common English stop words; (iv) stemming words to their original roots via Porter's stemmer (Porter, 1980).
3. Finally, the preprocessed terms are given as input to the LDA-GA algorithm devised by Panichella et al. (2013). This is an enhanced version of the standard LDA approach that solves an important problem, namely the setting of the parameter k, that is the predefined number of topics to extract. In particular, LDA-GA relies on a genetic algorithm that balances the internal cohesion of topics with the separation among clusters. In this way, it can estimate the ideal number of clusters to generate starting from the input terms (Panichella et al., 2013).

In Section 4 we report and discuss the topics given as output by the LDA-GA algorithm.

Time-to-fix analysis. To investigate such a perspective, we followed the procedure previously adopted by Zhang et al. (2012). Specifically, we mined a typical bug fixing process where (i) a user defines a bug report, (ii) the bug is assigned to a developer, (iii) the developer works and fixes the bug, (iv) the code is reviewed and tested, and (v) the bug is marked as resolved. Correspondingly, we computed five time intervals:

- *Time before response* (TBR). This is the interval between the moment a bug is reported and the moment it gets the first response from development teams.
- *Time before assigned* (TBA). This measures the interval between a bug getting the first response and its assignment.
- *Time before change* (TBC). This is the interval between a bug getting assigned and the developer starting to fix the bug, namely she performs the first commit after the bug has been assigned.
- *Duration of bug fixing* (DBF): This measures the interval between the developer starting and finishing the bug fixing, namely the time between the first and last commit before the bug has been marked as solved.
- *Time after change* (TAC): This is the interval between the developer finishing the bug fixing and the status of the bug being changed to resolved.

To compute these metrics, we relied on the evolution of the history of each bug report using the features available from the issue tracker. In particular, we mined (1) the timestamp in which a bug has been opened and that of the first comment for computing TBR; (2) the timestamp of the first comment and the one reporting when a bug report changed its status in "assigned" for DBA; (3) the timestamp of the "assigned" status and that of the first commit of the author involving the buggy artifact for TBC; (4) the timestamp of the first and last commit before the bug is marked as solved for DBF; and (5) the timestamp of the last commit and the one reporting the bug as "resolved" for TAC. For all the metrics, in Section 4 we report descriptive statistics of the time in terms of hours. It is important to note that, as done in previous work (Zhang et al., 2012), we filtered out all bugs whose final resolution was not fixed to ensure that only actually fixed bugs were investigated. It is important to note that the detailed results and script of these analyses are available in the online appendix (Catolino et al., 2018).

3.5. RQ$_3$: automated classification of bug types

Our final research question we focused on assessing the feasibility of a classification model able to classify bug types starting from bug reports. We relied on machine learning since this type of approach can automatically learn the features discriminating a certain category, thus simulating the behavior of a human expert (Pantic et al., 2007). As a side effect of this research question, we also pose a baseline against which future approaches aimed at more accurately classifying bug types can be compared. The following subsections detail the steps followed when building and validating our bug type classification model.

Independent and dependent variables. Our goal was to classify the bug type based on bug report information. We exploited summary messages contained in such reports as *independent variables* of our bug type classification model: our choice was driven by recent findings that showed how in most cases bug report summaries properly describe a bug, thus being a potentially powerful source of information to characterize its type (Zimmermann et al., 2010). Moreover, we did not include the title of a bug report as an independent variable because it might contain noise that potentially limits the classification performance (Zhou et al., 2016).

It is important to note that not all the words contained in a summary might actually be representative and useful to characterize the type of a bug. For this reason, we needed to properly preprocess them (Chowdhury, 2003).

In our context, we adopted the widespread *Term Frequency-Inverse Document Frequency* (TF-IDF) model (Salton and Buckley, 1988), which is a weighting mechanism that determines the relative frequency of words in a specific document (i.e., a summary of bug report) compared to the inverse proportion of that word over the entire document corpus (i.e., the whole set of bug report summaries in our dataset). This technique measures *how characterizing* a given word is in a bug report summary: for instance, articles and prepositions tend to have a lower TF-IDF since they generally appear in more documents than words used to describe specific actions (Salton and Buckley, 1988). Formally, let C be the collection of all the bug report summaries in our dataset, let w be a word, and let $c \in C$ be a single bug report summary, the TF-IDF algorithm computes the relevance of w in c as:

$$\mathrm{relevance}(w, c) = f_{w,c} \cdot \log(|C|/f_{w,C}) \tag{1}$$

where $f_{w,\, c}$ equals the number of times w appears in c, $|C|$ is the size of the corpus, and $f_{w,\, C}$ is equal to the number of documents in which w appears. The weighted words given as output from TF-IDF represent the independent variables for the classification model. It is important to note that the choice of TF-IDF was driven by experimental results: specifically, we also analyzed the accuracy of models built using more sophisticated techniques such as Word2Vec (Goldberg and Levy, 2014) and Doc2Vec (Le and Mikolov, 2014). As a result, we observed that the use of TF-IDF led to an improvement of F-Measure up to 13%. Therefore, we focus on TF-IDF in the remainder of the paper.

As for *dependent variable*, it was represented by the bug types present in our taxonomy.

Classifiers. With the aim of providing a wider overview of the performance achieved by different classifiers, we experimented with classifiers previously used for prediction purposes by the research community, i.e., (i) Naive Bayes, (ii) Support Vector Machines (SVM), (iii) Logistic Regression, and (iv) Random Forest. These classifiers have different characteristics and different advantages/drawbacks in terms of execution speed and over-fitting (Nasrabadi, 2007). It is important to note that before running the models, we also identified their best configuration using the Grid Search algorithm (Bergstra and Bengio, 2012). Such an algorithm represents a brute force method to estimate hyperparameters of a machine learning approach. Suppose that a certain classifier C has k parameters, and each of them has N possible values. A grid search basically considers a Cartesian product $f_{|_{k \times N}}$ of these possible values and tests all of them. We selected this algorithm because recent work in the area of machine learning has shown that Grid Search is among the most effective methods to configure machine learning algorithms (Bergstra and Bengio, 2012).

After the experimental analysis, we found that Logistic Regression provided the best performance. For this reason, in Section 4 we only report the findings achieved using this classifier. A complete overview of the performance of the other models built with different classifiers is available in our online appendix (Catolino et al., 2018).

Validation strategy. To validate the model, we adopted 10-fold cross validation (Stone, 1974). It splits the data into ten folds of equal size applying a stratified sampling (i.e., each fold has the same proportion of each bug type). One fold is used as a test set, while the remaining ones as a training set. The process is repeated 100 times, using each time a different fold as a test set. Given that the distribution of the dependent variable is not uniform (see more in Section 4.2), we took into account the problem of training data imbalance (Chawla et al., 2002). This may appear when the number of data available in the training set for a certain class (e.g., the number of bugs belonging to a certain type) is far less than the amount of data available for another class (e.g., the number of bugs belonging to another type). More specifically, we applied the Synthetic Minority Over-sampling Technique (SMOTE) proposed by Chawla et al. (2002) to make the training set uniform with respect to the bug type available in the defined taxonomy. Since this approach can be run once per time to over-sample a certain minority class, we repeated the over-sampling until all the classes considered have a similar number of instances.

Finally, to cope with the randomness arising from using different data splits (Refaeilzadeh et al., 2009), we repeated the 10-fold cross validation 100 times, as suggested in previous work (Hall et al., 2011). We then evaluated the mean accuracy achieved over the runs (Stone, 1974).

To measure the performance of our classification, we first computed two well-known metrics such as *precision* and *recall* (Baeza-Yates and Ribeiro-Neto, 1999), which are defined as follow:

$$\text{precision} = \frac{\text{TP}}{\text{TP}+\text{FP}} \qquad \text{recall} = \frac{\text{TP}}{\text{TP}+\text{TN}} \tag{2}$$

where TP is the number of true positives, *TN* the number of true negatives, and FP the number of false positives. In the second place, to have a unique value representing the goodness of the model, we compute the F-Measure, i.e., the harmonic mean of precision and recall:

$$\text{F-Measure} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{3}$$

Moreover, we considered two further indicators. The first one is the area under the ROC curve (AUC-ROC) metric. This measure quantifies the overall ability of the classification model to discriminate between the different categories. The closer the AUC-ROC to 1, the higher the ability of the model. In contrast, the closer the AUC-ROC to 0.5, the lower the accuracy of the model. Second, we computed the Matthews Correlation Coefficient (MCC) (Baldi et al., 2000), a regression coefficient that combines all four quadrants of a confusion matrix, thus also considering true negatives:

$$\text{MCC} = \frac{(\text{TP} * \text{TN}) - (\text{FP} * \text{FN})}{\sqrt{(\text{TP}+\text{FP})(\text{TP}+\text{FN})(\text{TN}+\text{FP})(\text{TN}+\text{FN})}} \tag{4}$$

where TP, TN, and FP represent the number of (i) true positives, (ii) true negatives, and (iii) false positives, respectively, while *FN* is the number of false negatives. Its value ranges between -1 and $+1$. A coefficient equal to $+1$ indicates a perfect prediction; 0 suggests that the model is no better than a random one; and -1 indicates total disagreement between prediction and observation.

4. Analysis of the results

In this section, we report the results of our study, discussing each research question independently.

4.1. *RQ$_1$: taxonomy of bug types*

The manual analysis of the 1280 bug reports led to the creation of the taxonomy of 9 bug types, described in the next subsections. At the same time, we had to discard 141 bug reports for two reasons. In particular, 18 of them—all found in Mozilla—were related to bug reports listing multiple bugs to solve before the release of a new version of the system: from a practical point of view, they represent a to-do list rather than accurate bug reports. For this reason, we decided to exclude them as we could not identify a specific category to which to assign them. On the other hand, 123 bug reports could be considered as false positives due to proposals for improvement or suggestions on how to fix existing bugs: also in this case, we did not consider these suitable for the scope of our study. To some extent, the latter category of false positives highlights how the use of a fully automated filtering technique like the one proposed by Herzig et al. (2013) (used in the context selection phase to gather bug reports actually reporting observed malfunctions) is not enough to discard misclassified bugs, i.e., the results of such tools must always be double-checked to avoid imprecisions. At the end of this process, the final number of bug reports classified was 1139. In the following, we explain each category of bug type in our taxonomy, reporting an example for each of them. Given the excessive length of the bug reports analyzed, we do not report the entire summary in the examples but we highlight the main parts that allow the reader to understand the problem and why we marked it as belonging to a certain category.

A. Configuration issue. The first category regards bugs concerned with building configuration files. Most of them are related to problems caused by (i) external libraries that should be updated or fixed and (ii) wrong directory or file paths in `xml` or `manifest` artifacts. As an example, the bug report shown below falls under this category because it is mainly related to a wrong usage of external dependencies that cause issues in the web model of the application.

Example summary.
"JEE5 Web model does not update on changes in web.xml"
[Eclipse-WTP Java EE Tools] - Bug report: `190198`

B. Network issue. This category is related to bugs having connection or server issues, due to network problems, unexpected server shutdowns, or communication protocols that are not properly used within the source code. For instance, in the following, we show an example where a developer reports a newly introduced bug due to a missing recording of the network traffic of the end-users of the project.

Example summary.
"During a recent reorganization of code a couple of weeks ago, SSL recording no longer works"
[Eclipse-z_Archived] - Bug report: `62674`

C. Database-related issue. This category collects bugs that report problems with the connection between the main application and a database. For example, this type of bug report describes issues related to failed queries or connection, such as the case shown below where the developer reports a `connection stop` during the loading of a Java Servlet.

Example summary.
"Database connection stops action servlet from loading"
[Apache Struts] - Bug report: `STR-26`

D. GUI-related issue. This category refers to the possible bugs occurring within the Graphical User Interface (GUI) of a software project. It includes issues referring to (i) stylistic errors, i.e., screen layouts, elements colors and padding, text box appearance, and buttons, as well as (ii) unexpected failures appearing to the users in form of unusual error messages. In the example below, a developer reports a problem that arises because she does not see the actual text when she types in an input field.

Example summary.
"Text when typing in input box is not viewable."
[Mozilla-Tech Evangelism Graveyard] - Bug report: `152059`

E. Performance issue. This category collects bugs that report performance issues, including memory overuse, energy leaks, and methods causing endless loops. An example is shown below, and reports a problem raised in the Mozilla project where developers face a performance bug due to the difficulties in loading an external file.

Example summary.

"Loading a large script in the Rhino debugger results in an endless loop (100% CPU utilization)"

[Mozilla-Core] - Bug report: 206561

F. Permission/deprecation issue. Bugs in this category are related to two main causes: on the one hand, they are due to the presence, modification, or removal of deprecated method calls or APIs; on the other hand, problems related to unused API permissions are included. To better illustrate this category, in the following we provide an example for each of the causes that can fall into this category. The first involves a bug appearing in the case of an unexpected behavior when the method of an external API is called. The second mentions a bug that appears through malformed communication with an API.

Example summary.

"setTrackModification(boolean) not deprecated; but does not work"

[Eclipse-EMF] - Bug report: 80110

Example summary.

"Access violation in DOMServices::getNamespaceForPrefix (DOMServices.cpp:759)"

[Apache-XalanC] - Bug report: XALANC-55

G. Security issue. Vulnerability and other security-related problems are included in this category. These types of bugs usually refer to reload certain parameters and removal of unused permissions that might decrease the overall reliability of the system. An example is the one appearing in the Apache Lenya project, where the Cocoon framework was temporarily stopped because of a potential vulnerability discovered by a developer.

Example summary.

"Disable cocoon reload parameter for security reasons"

[Apache-Lenya] - Bug report: 37631

H. Program anomaly issue. Bugs introduced by developers when enhancing existing source code, and that are concerned with specific circumstances such as exceptions, problems with return values, and unexpected crashes due to issues in the logic (rather than, e.g., the GUI) of the program. It is important to note that bugs due to wrong SQL statements do not belong to this category but are classified as database-related issues because they conceptually relate to issues in the communications between the application and an external database, rather than characterizing issues arising within the application. It is also worth noting that in these bug reports developers tend to include entire portions of source code, so that the discussion around a possible fix can be accelerated. An example is shown below and reports a problem that a developer has when loading a resource.

Example summary.

"Program terminates prematurely before all execution events are loaded in the model"

[Eclipse-z_Archived] - Bug report: 92067

I. Test code-related issue. The last category is concerned with bugs appearing in test code. Looking at bug reports in this category, we observed that they usually report problems due to (i) running, fixing, or updating test cases, (ii) intermittent tests, and (iii) the inability of a test to find de-localized bugs. As an example, the bug report below reports on a problem occurred because of a wrong usage of mocking.

Example summary.

"[the test] makes mochitest-plain time out when the HTML5 parser is enabled"

[Mozilla-Core] - Bug report: 92067

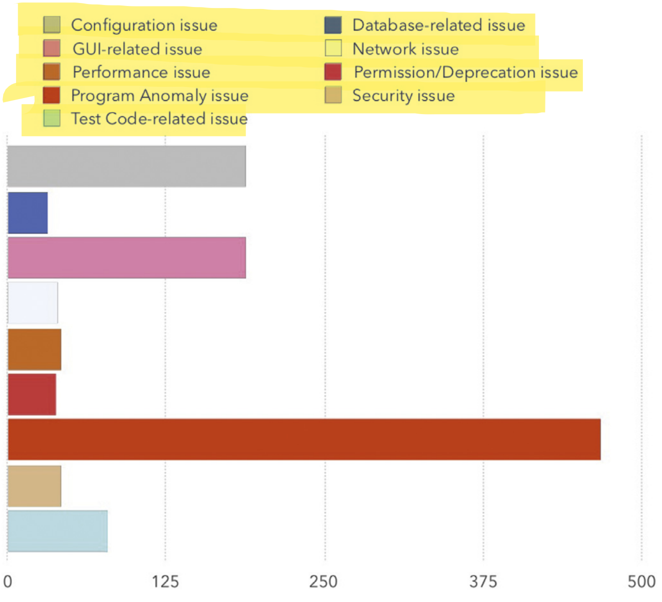


Fig. 2. RQ$_2$—frequency of each category of bug type.

4.2. RQ$_2$: the characteristics of different bug types

After we had categorized and described the taxonomy, we focused on determining the characteristics of the different bug types discovered. For the sake of comprehensibility, in this section, we individually discuss the results achieved for each considered aspect, i.e., frequency, relevant topics, and time-to-fix process.

Frequency analysis. With this first perspective, we aimed at studying how prevalent each bug type is in our dataset.

Fig. 2 shows the diffusion of bug types extracted from the 1139 analyzed bug reports. As depicted, the most frequent one is the *Functional Issue*, which covers almost half of the entire dataset (i.e., 41.3%). This was somehow expected as a result: indeed, it is reasonable to believe that most of the problems raised are related to developers actively implementing new features or enhancing existing ones. Our findings confirm previous work (Tan et al., 2014; Aranda and Venolia, 2009) on the wide diffusion of bugs introduced while developers are busy with the implementation of new code or when dealing with exception handling.

GUI-related problems are widely present in the bug reports analyzed (17% of the total number of issues in the dataset). Nowadays, GUIs are becoming a major component of many software systems because they shape the interaction with the end-user. As such, they can evolve and become more complex, thus attracting as many bugs as the codebase (Memon, 2002). This result somehow confirms the findings reported by Tan et al. (2014), who also discovered that GUI-related issues are highly popular in modern software systems.

The third most popular type is the *configuration issue* one, as 16% of the bug reports referred to this type. Since the use of external libraries and APIs is growing fast (Robbes et al., 2012; Mileva et al., 2009; Salza et al., 2018), bugs related to how an application communicates or interacts with external components are becoming more frequent. Moreover, McDonnell et al. (2013) recently showed that the API adaptation code tends to be more bug-prone, possibly increasing the chances of such category of bugs. At the same time, it is also worth noting that some recent findings (Bezemer et al., 2017) also reported that issues with configuration files (e.g., the presence of unspecified dependencies (Bezemer et al., 2017)) represent a serious issue for developers, which might lead them to introduce more bugs.

Table 2
RQ_2—relevant topics of each category of bug type.

Categories	Topic 1	Topic 2	Topic 3	Topic 4	Topic 5
Configuration issue	link	file	build	plugin	jdk
Network issue	server	connection	slow	exchange	–
Database-related issue	database	sql	connection	connection	–
GUI-related issue	page	render	select	view	font
Perfomance issue	thread	infinite	loop	memory	–
Permission/deprecation issue	deprecated	plugin	goal	–	–
Security issue	security	xml	packageaccess	vulnerable	–
Program anomaly issue	error	file	crash	exception	–
Test code-related issue	fail	test	retry	–	–

After these first three bug types, we discovered that 7% of the bug reports in our dataset referred to *test code bugs*. Using another experimental setting, and observing the relative diffusion of this type, we confirm the results of Vahabzadeh et al. (2015), who showed that the distribution of bugs in test code does not variate too much with respect to that of production code. Our findings are also in line with what is reported in recent studies on the increasing number of test-related issues (Luo et al., 2014; Palomba and Zaidman, 2017; 2019; Bell et al., 2018; Zaidman et al., 2008; 2011; Beller et al., 2017).

Performance issues comprise 4% of the total number of issues. This result confirms the observation from Tan et al. (2014). Indeed, they discovered that bugs related to performance are much less frequent than functional bugs and that their number usually decreases over the evolution of a project. A likely explanation for the relatively low diffusion of this bug type is that the developers involved in the software ecosystems considered in the study often use performance leak detection tools during the development. For instance, the Mozilla guidelines[7] highly recommend the use of such tools to limit the introduction of performance leaks in the project as much as possible.

Other specific bug types such as *network*, *security*, and *permission/deprecation* appear to be less diffused over the considered dataset, i.e., they are the cause of ≈4% reported bugs. Interestingly, our findings related to security-related bugs are not in line with those reported in the study by Tan et al. (2014). Indeed, while they found that this type is widespread in practice, we could only find a limited number of bug reports actually referring to security problems. Finally, the least spread bug type is *database-related*, that arises in 3% of the cases, confirming that such bugs represent a niche of the actual issues occurring in real software systems (Schröter et al., 2006): in this regard, it is worth remarking that replications of our study targeting database-intensive applications would be beneficial to further verify our finding.

To broaden the scope of the discussion, we noticed that the diffusion of the bug types discussed so far is independent from the type of system considered. Indeed, we observed similar distributions over all three ecosystems analyzed, meaning that the same bug types might basically be found in any software project. This supports our next step: the creation of an automated solution to classify the bug types, something which could be immediately adopted for improving the diagnosis of bugs.

Topics analysis. Table 2 reports the results achieved when applying the LDA-GA algorithm over the bug reports of each category of bug type present in our taxonomy. It is important to note that LDA-GA found up to five different clusters that describe the topics characterizing each bug type; a '–' symbol is put in the table in case LDA-GA did not identify more topics for a certain bug type. From a general point of view, we can observe that there is almost zero overlap among the terms describing each bug type: on the one hand, we notice that all the topics extracted for each category are strictly related to the description of the categories discussed above (e.g., the word "test" describes test-related issues); on the other hand, the lack of overlap is a symptom of a good systematic process of categorization of the bug reports.

Going more in depth, the topics extracted for the configuration issue category are very much linked to problems appearing in configuration files and concerned with build issues ("build", "file", "jdk"), caused by wrong file paths (i.e., "link") or external components that should be updated (i.e., "plugin"). A similar discussion can be delineated in the case of network issues. In particular, based on the bug reports belonging to this category, we found words such as "server" and "connection" that represent topics strictly related to network information, together with words reporting the likely causes of these issues, i.e., "slow" connection or problems due to the "exchange" of data over the network.

In the context of database-related issues, our findings provide two main observations. The words contained in these bug reports contain clear references to problems occurring with databases, like "database", "SQL", or "connection". At the same time, it is worth noting that the word "connection" occurs twice and, more importantly, is in overlap with a word appearing in network-related bug reports. On the one hand, it is important to note that LDA analysis can generate multiple topics having the same discriminant word (Blei et al., 2003): indeed, each document (i.e., bug reports, in our case) is viewed as a mixture of various topics. That is, for each document LDA-GA assigns the probability of the bug report belonging to each topic. The probability sums to 1: hence, for every word-topic combination there is a probability assigned and it is possible that a single word has the highest probability in multiple topics (Blei et al., 2003). From a practical perspective, this may indicate that problems with the database connection can be the principal cause of this type of bugs. As for the overlap between network and database issues, this is somehow expected: both the types of bugs have to deal with connection problems of different nature. This might possibly create noise for automated solutions aiming at discriminating different bug types. Regarding the GUI-related issues, we find that the topics are represented by words clearly related to a GUI interface of a software project i.e., "page", "render", "select", "view", and "font". For instance, they could concern problems with the rendering of a certain "font" or an entire page; in any case, these are problems with the visualization of the interface of a system. As for performance-related issues, the topics extracted faithfully represent problems connected with excessive memory consumption; indeed, words such as "thread", "infinite", "loop", and "memory" are the four topics that most frequently appear and that better describe those bug reports. On the other side, in the category related to security issues we found topics linked to problems of "package access", but also to "vulnerable" components that may lead to "security" problems. Regarding the topic "XML", it is important to note that there are a number of security issues involving the configuration of XML parsers and how they

[7] https://developer.mozilla.org/en-US/docs/Mozilla/Performance.

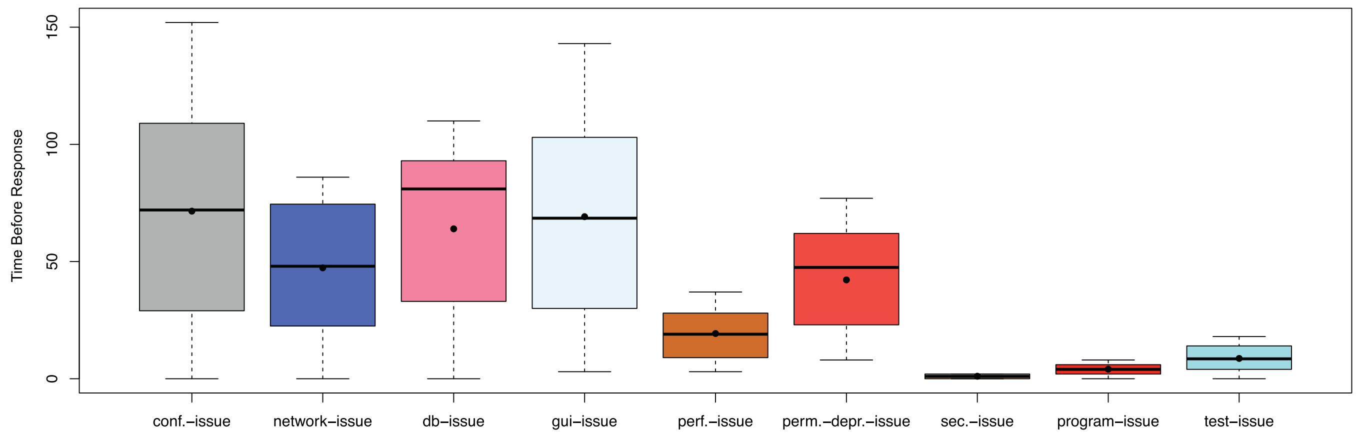


Fig. 3. RQ_2—box plots reporting the time before response (TBR) for each identified bug type.

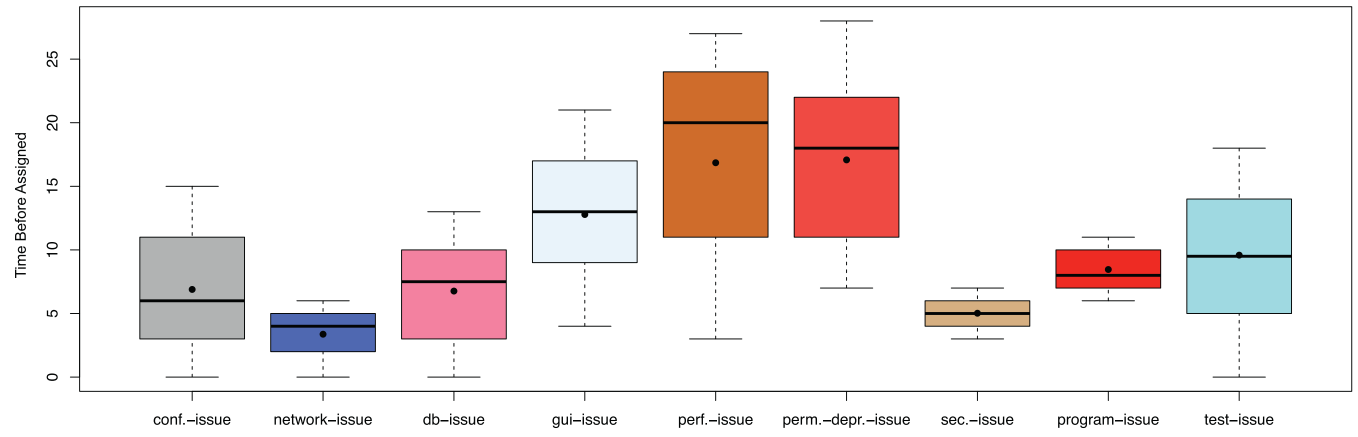


Fig. 4. RQ_2—box plots reporting the time before assigned (TBA) for each identified bug type.

interact with the document structure (Moradian and Håkansson, 2006; Lowis and Accorsi, 2011). For example, let us consider the validation against untrusted external DTDs (Document Type Declaration) files. The DTD of an XML document is one way to define the valid structure of the document, i.e., the rules that specify which elements and values are allowed in the declaration. A security problem may arise in case the server's XML parser accepts an arbitrary external DTD URL and attempts to download the DTD and validate the XML document against it. In this case, an attacker could input any URL and execute a Server Side Request Forgery (SSRF) attack where the attacker forces the server to make a request to the target URL.

When considering program anomalies, we noticed that the topics extracted are strictly connected with the description of the category given in the context of RQ_1. Indeed, topics such as “error”, “file”, “patch”, “crash”, and *exception* are concerned with problems caused by issues in the logic of the program (e.g., a wrong return value or an exception). Finally, permission/deprecation and test code-related issues follow the same discussion: all the words extracted by LDA-GA have clearly something to do with their nature: as an example, the word “retry” appearing in tests is connected with a JUnit annotation (`@Retry`) that highlights the presence of some form of test flakiness, i.e., unreliable tests that exhibit a pass and fail behavior with the same code (Palomba and Zaidman, 2017).

All in all, our results indicate that the words characterizing the identified bug types are pretty disjoint from each other. As a consequence, it is reasonable to use the words occurring within bug reports to classify them according to the defined taxonomy. This is a clear motivation for adopting a natural language-based machine learning approach like the one proposed in RQ_3.

Time-to-fix analysis. Figs. 3–7 depict box plots of the five metrics considered to measure the time required from the entire bug fixing process of each bug type belonging to the taxonomy, i.e., *time before response, time before assigned, time before change, duration of bug fixing*, and *time after change*, respectively. The black dots presented in the figures represent the mean value of each distribution.

From a high-level view, we could first confirm that not all bugs are the same, as each bug type has its own peculiarities in terms of the time required for the entire bug fixing process. Looking more in-depth on the single indicators, the first observation is related to TBR: in this case, we see that security-related issues are those having the smallest time from reporting to the first response from the development team. This is somehow expected, since security issues have a high harmfulness for the overall reliability of a software system (Di Penta et al., 2008). More specifically, both mean and median values are equal to 2, indicating that in a pretty short time window the developers tend to take action after a potential vulnerability is detected. Moreover, the distribution is all around the median, meaning that there is no variability in the time to response among the analyzed security issues: thus, we can claim that independently from the system or other factors such as their frequency of appearance, these issues are seriously taken into account by developers.

Also bugs due to program anomalies present a limited time interval between their discovery and the first reaction from developers. Also in this case, the result is quite expected: indeed, this category relates to issues in the inner-working of a program that

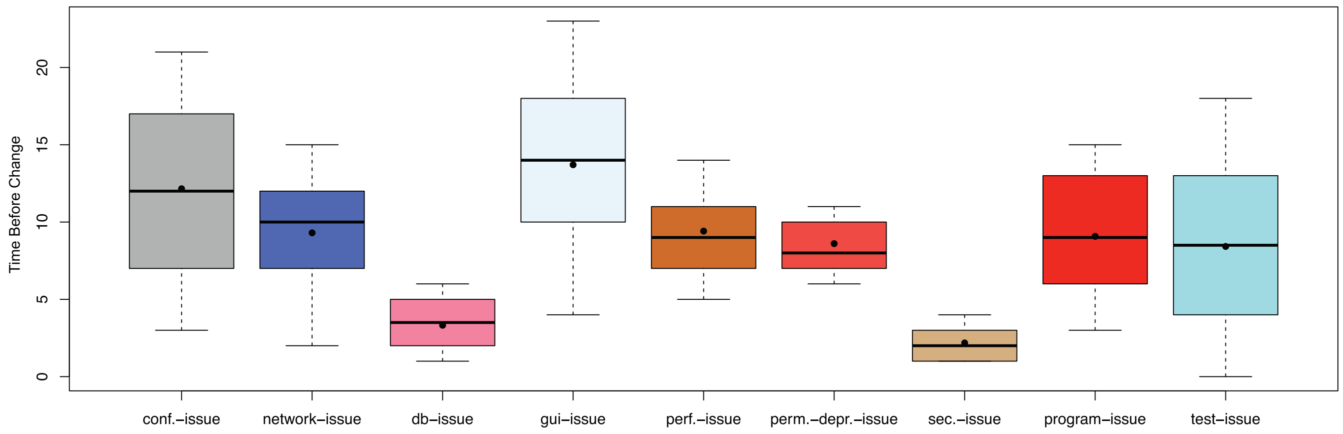


Fig. 5. RQ$_2$—box plots reporting the time before change (TBC) for each identified bug type.

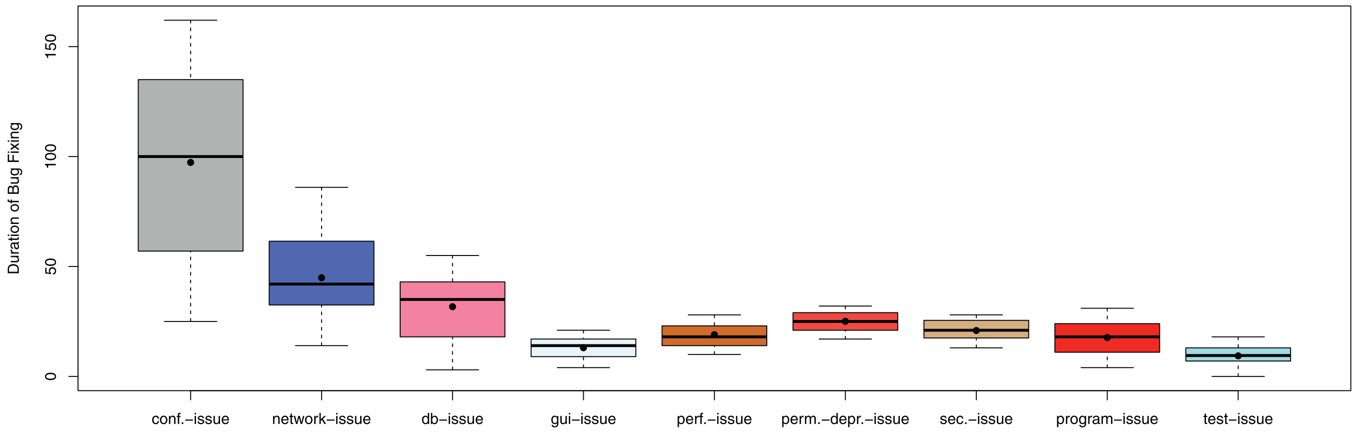


Fig. 6. RQ$_2$—box plots reporting the duration of bug fixing (DBF) for each identified bug type.

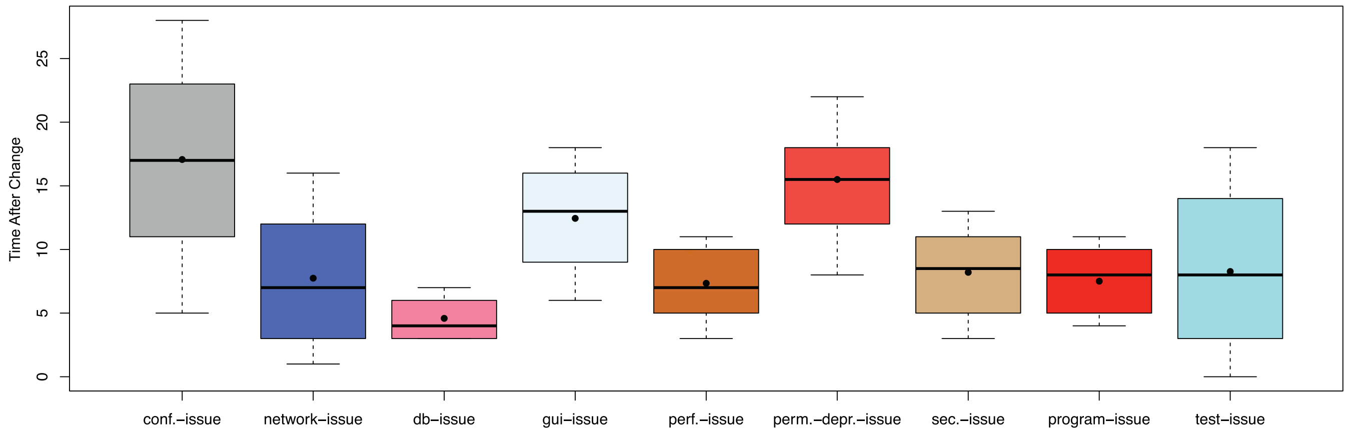


Fig. 7. RQ$_2$—box plots reporting the time after change (TAC) for each identified bug type.

might potentially have negative consequences on reliability and make the system less appealing for end-users (Bavota et al., 2015; Palomba et al., 2018; 2017). The distribution is close to the median, thus highlighting that developers pay immediate attention to these bugs.

A more surprising result is the one obtained for test-related issues. Even though they are generally perceived as less important than bugs appearing in production (Meyer et al., 2014; Tufano et al., 2016), our data shows that developers react pretty fast to their reporting: both mean and median are equal to 7, meaning that the reaction of developers is observed within one week. Also in this case, the distribution is not scattered and, therefore, we can claim that the short-time reaction to test-related issues represents a rule rather than the exception. Likely, this result reflects the ever increasing importance that test cases have in modern software systems, e.g., for deciding on whether to integrate pull requests or build the system (Beller et al., 2017; Gousios et al., 2015; Beller et al., 2015; 2019).

Performance issues have a median time before response of 12. When comparing this value with those achieved by other types of bugs, we can say that it is pretty low and that, as a consequence, the developers' reaction to this kind of bugs is fast. Our finding confirms previous analyses conducted in the field of performance bug analysis (Jovic et al., 2011). As for the rest, all the other bug types have much higher distributions, and this likely indicates that developers tend to focus first on issues that directly impact functionalities and reliability of the system.

Turning the attention to TBA (Fig. 4), we observe that network-related issues are those assigned faster for fixing. If we compare the time required to performance-related issues or permission/deprecation issues to be assigned, we can hypothesize that the observed findings are strictly connected to the difficulty to find good assignees. For instance, a possible interpretation of our results is that, based on the developers' expertise and workload, a certain type of bug is assigned faster than others. While further investigations around this hypothesis would be needed and beneficial to study the phenomenon deeper, we manually investigated the bugs of our dataset to find initial compelling evidence that suggests a relation between time-to-assignment and developer-related factors. As a result, looking at both bug reports and comments, we found 21 cases (out of the total 42) in which the assignment of performance issues has been delayed because of the lack of qualified personnel. For example, let consider the following comment made by a Mozilla developer:

"I'm reluctant to do such a task, not really and expert... maybe something for E.?"

Conversely, in the cases of network-related and security issues, we observed that there exist specific developers that have peculiar expertise on these aspects: this potentially make the assignment faster. Nonetheless, our future research agenda includes a more specific investigation on the factors impacting bug fixing activities; in the context of this paper, we only limit ourselves in reporting that it is possible to observe differences in the way different bug types are treated by developers.

As for the TBC (Fig. 5), we still observe differences among the discovered bug types. Security issues are those that developers start working on faster: as explained above, this is likely due to the importance of these issues for reliability. At the same time, bugs due to database-related problems have a small time interval between assignment and beginning of the actual fixing activities (median = 4 h). Also in this case, it is reasonable to believe that these bugs can cause issues leading end-users not to interact with the system in a proper manner and, therefore, they represent issues that are worth to start fixing quickly. More surprisingly, the fixing process of program anomalies requires a higher number of hours to be started. While more investigations would be needed, we can conjecture that factors like severity and priorities assigned for their resolution have an important impact on how fast developers deal with them.

Looking at DBF, namely the duration of bug fixing, we can observe that the differences are less evident than the other metrics. Indeed, the fixing duration of most of the bugs ranges between 2 and 30 h. This is especially true for program anomalies, GUI and test code-related issues, and security problems. A different discussion is the one for database- and network-related issues: despite them being quickest in terms of TBA and TBC, respectively, their duration is much longer than other bugs. Factors like the complexity of the solution or priority assigned to them might explain such a difference. Overall, however, it seems that developers tend to focus more and more quickly on certain types of bugs, confirming the fact that not all bugs are treated in the same manner.

Finally, when considering the TAC reported in Fig. 7, we observe that the majority of bug types have a similar time after that the corresponding patches have been submitted. Most likely, this heavily depends on the processes adopted within the projects to control for the soundness of a patch: for instance, most of the modern projects perform code review activities of all the newly committed code changes, and have standard procedures to assess the validity of the change before integration in the code base (Pascarella et al., 2018). The only exception to this general discussion is related to the configuration-issue, which takes up to 33 hours to be integrated: however, given previous findings in literature (Anvik and Murphy, 2011; Mockus et al., 2002; Twidale and Nichols, 2005), we see this as an expected result because configuration-related discussions generally trigger more comments by developers since a change in configuration files might impact the entire software project. As a consequence, they take more time to be actually integrated.

Table 3
RQ_3—performance (in percentage) achieved by the bug type prediction model. P = Precision; R = Recall; F-M = F-Measure; AR = AUC-ROC; MCC = Matthews correlation coefficient.

Categories	Logistic regression				
	P	R	F-M	AR	MCC
Configuration issue	46	52	49	68	66
Database-related issue	71	63	67	72	76
GUI-related issue	61	68	65	77	65
Network issue	36	40	38	56	59
Performance issue	67	57	62	65	67
Permission/deprecation issue	86	55	67	69	74
Program anomaly issue	68	65	67	74	68
Security issue	76	74	75	88	85
Test code-related issue	90	70	79	93	88
Overall	67	60	64	74	72

4.3. RQ_3: automated classification of bug types

Table 3 reports, for each bug type, the mean *precision, recall, F-measure, AUC-ROC*, and *Matthews correlation coefficient* achieved by our bug type prediction model over the 100 runs of the 10-fold cross validation. We observed that the F-Measure ranges between 35% and 77%, the AUC-ROC between 56% and 93%, while the MCC between 59% and 88%. Thus, overall, we can claim that the devised prediction model is reasonably accurate in identifying the type of a bug by exploiting bug report information. It is important to remark that the model considers the words composing the bug report summary as an independent variable: the model is already able to achieve high performance for most of the categories only taking into account such words, meaning that our initial step toward the automatic classification of bug types based on bug report information can be considered successful. Nevertheless, further research on the features that influence the type of bugs (e.g., structural information of the involved classes) might still improve the performance. We plan to perform a wider analysis of additional features in our future research.

Looking more in-depth into the results, the first interesting observation can be made when analyzing the performance of the model on the *Test Code-related issue* category. In this case, it reaches the highest F-Measure, AUC-ROC, and MCC values (i.e., 77%, 93%, and 88%, respectively). Since the model relies on bug report words, the result can be explained by the fact that the terms used by developers in bug reports involving test-related issues are pretty powerful to discriminate this type. As a matter of fact, 87% of the test-related bug reports in our dataset contain terms like "test" or "test suite", as opposed to bug reports related to different types. This means that a textual-based learning model can more easily classify bug type. For instance, let us consider the bug report number 358221 available on the Lyo project of the *Eclipse* ecosystem, which reports the following summary:

"Investigate possible test suite bug when ServiceProviderCatalog contains ref to serviceProvider resource"

Similar observations can be made to explain the results for the *Security issue* category (AUC-ROC = 88%). Also in this case, developers frequently adopt terms like “security” or “vulnerability” to describe a bug having this bug type.

Still, categories related to *Functional Issue*, *GUI-related issue*, and *Network issue* can be accurately classified by the model. Specifically, F-Measure values of respectively 67%, 64%, and 62% are reached. On the one hand, these results confirm that a textual-based approach can be effective in classifying the type of bugs. At the same time, our findings eventually reveal that developers follow specific patterns when describing issues related to different categories.

Turning the attention toward the categories for which the model does not perform very well, there are two main cases to discuss. The first one is related to the *Configuration issue* type, which has an F-Measure = 48%. To better understand the possible causes behind this result, we manually analyzed the bug reports belonging to this bug type. Let us consider two cases coming from the Apache XalanC project (bug reports number `XALANC-44` and `58288`):

“Could not compile”

“VE hangs; times out; then throws NPE doing pause/reload”

Looking at these bug reports, we could not immediately understand the type they refer to. Indeed, during the taxonomy building phase we could analyze other information like developers’ discussions and attachments; however, since our classification model is only based on words composing the summary, sometimes it cannot associate such words to the correct bug type. To some extent, our finding contextualizes the findings by Zimmermann et al. (2010) on the quality of bug reports, showing it varies depending on the bug type that developers have to report.

A similar situation arises when considering *database-related* issues. While in RQ$_2$ we discovered that the corresponding bug reports have textual characteristics that might be exploited to identify their type, we also highlighted the presence of overlapping words with other categories that may preclude the correct functioning of the model. As such, this finding indicates once again that the performance of our bug type classification model may be improved by considering further bug report components such as developers’ discussions and attachments.

To conclude the discussion, it is worth relating the performance of the classification model to the results reported in RQ$_2$ on the diffusion of each bug type. Put into this context, the devised model is able to properly predict all the most diffused categories, with the notable exception of *configuration* issues. As such, we argue that the model can be useful in practice and that more research is needed in order to improve its capabilities in detecting configuration-related problems.

5. Discussion and implications

Our results highlighted a number of points to be further discussed as well as several practical implications for both practitioners and researchers.

Discussion. At the beginning of our investigation, we conjectured that the knowledge of the underlying bug types could be useful information to exploit to improve bug triaging approaches. Our findings clearly highlighted that bugs are different in nature, have different characteristics with respect to the way developers deal with them, and can be classified with a pretty high accuracy using machine learning models. We argue that this information can be useful for the bug triaging process for the following three main reasons:

- **Raising awareness on the decision-making process.** In the first place, through RQ$_2$ we discovered that different bugs are subject to a different bug fixing process with respect to both the time they required to be assigned and to be actually fixed and integrated into production. Our automatic classification technique can first support developers in the decision-making process, as it can immediately pinpoint the presence of bugs having a nature making them potentially more dangerous than others: as an example, our technique can tag a newly reported bug report as a security issue, raising the *awareness* of developers on the need to take prompt actions, thus possibly speeding up their reaction time, considering its assignment and resolution as well as the in-between activities, e.g., pushing the assigned developer(s) to perform the required bug fixing action in a timely manner.
- **Comprehending the bug types.** As a complementary support to awareness, the findings reported in our study have the potential to make developers more focused on the signaled type of a reported bug, thus reducing the time required in the understanding of the problem. We hypothesise that this could help reduce the time required to (i) assign a bug to the most-skilled developer and (ii) involve the most-qualified developers in the discussion on how to fix the bug. For instance, the output of the proposed approach would support developers in timely spotting configuration-related issues, that are those having the most time fixing process according to our analyses. As a result, community shepherds and developers could use this information to take actions and involve the appropriate set of experienced developers in an effort of finding a solution to fix the newly submitted bug.
- **Improving bug triage.** Finally, the bug type prediction model proposed in this study can be exploited within existing but triaging approaches to improve their performance. As reported by Shokripour et al. (2013), current approaches can be broadly divided into two sets: activity- and location-based. The former identifies the most-skilled developer to be assigned to a new bug on the basis of her previous activities, namely on the analysis of which bugs she fixed, while the latter takes into account the location of the bug within the source code. More recently, Tian et al. (2016) proposed a model that combined these two approaches: they considered both developers’ previous activities (e.g., developer bug fixing frequency) and code locations associated with a bug report as similarity features in order to capture the similarity between a bug report and developers’ profile. Nevertheless, all these approaches only consider the developers’ perspective, without taking into account the nature of the bug that needs to be fixed. Our model can complement all the existing techniques by complementing the information on the location of a newly reported bug with developers’ activities aimed at fixing *specific bug types rather than their merely attitude to resolve bugs*: we envision the definition of novel ensemble approaches and/or search-based algorithms that can exploit the bug type together with developers’ experience and location of the bug to improve the identification of the most-skilled developer that can fix the bug.

We believe that all the aspects reported above deserve more attention, especially on the basis of the results reported in our study. They are, therefore, part of our future research agenda, which is devoted to the improvement of current bug triaging approaches.

Implications. Besides the discussion points reported above, we see some important implications of our work. More specifically:

1. **A better understanding of bugs is needed.** In our work, we discovered a number of issues being reported in bug reports: bugs are different from each other, and it would be particularly useful to better study the characteristics of each of them, e.g.,

investigating whether they are introduced differently, with the aim of improving or specializing bug localization approaches and bug prediction models. Moreover, we believe that particular attention should be devoted to the understanding of functional bugs, which are those that appear more frequently in practice. For instance, further studies aimed at decomposing the category in multiple more specific sub-categories or investigating their perceived harmfulness would be beneficial to provide an improved support to developers.

2. **More research on test code bugs is needed.** Our work revealed that a large number of bugs impact test code. The research community has heavily studied production bugs (Ray et al., 2016), however, only a few studies are available with respect to bugs in test code (Luo et al., 2014; Palomba and Zaidman, 2017; Bell et al., 2018). We argue that more research on these bugs can be worthwhile to improve both quality and reliability of test cases.
3. **Configuration checkers are important.** According to our findings, configuration-related bugs are among the most popular ones. Unfortunately, little is known on this category of bugs (Bezemer et al., 2017) and there are no tools able to support developers in managing the quality of configuration files. We argue that more research aimed at devising such configuration quality checkers is needed to assist practitioners and avoid the introduction of bugs.
4. **Establishing whether the role of other bug report features would improve bug types analysis.** While a key results of our work is the good performance of a classification model relying on bug summaries as independent variable, we noticed that in some cases it cannot perform well because words contained in bug reports are not enough to identify the bug type. On the one hand, studies investigating the linguistic patterns used by developers would be worthwhile to learn how to better classify bug reports; on the other hand, the analysis of the value of other bug report features (e.g., developers' discussions) would represent the next step toward an improved support for bug types analysis.

6. Threats to validity

In this section, we discuss possible threats affecting our results and how we mitigated them.

6.1. Taxonomy validity

To ensure the correctness and completeness of the bug types identified in the taxonomy building phase, we performed an iterative content analysis that allowed us to continuously improve the quality of the taxonomy by merging and splitting categories if needed. Moreover, as an additional validation, we involved 5 expert industrial developers and asked them to classify a set of 100 bug reports according to the proposed taxonomy. They related the sampled bug reports to the same bug types as those assigned by us during the phase of taxonomy building, thus confirming the completeness and clarity of the identified bug types. Nevertheless, we cannot exclude that our analysis missed specific bug reports that hide other bug types.

6.2. Conclusion validity

Threats to conclusion validity refer to the relation between treatment and outcome. In the context of RQ_2, we extracted relevant topics within bug reports referring to different bug types using Latent Dirichlet Allocation (LDA) (Blei et al., 2003). To overcome the problem of configuring the parameter k—whose wrong configuration has been shown to bias the interpretation of the results (Peng et al., 2001)—we employed the LDA-GA version of the technique proposed by Panichella et al. (2013): this is based on a genetic algorithm that is able to exercise the parameter k until an optimal number of clusters is found. Still in RQ_2, we investigated the time required for the bug fixing process of different bug types by replicating the study of Zhang et al. (2012), thus taking into account all the metrics they employed to measure the bug fixing process. Nonetheless, it is important to point out that further empirical analyses aimed at understanding the specific reasons behind the observed findings, namely what are the factors that developers take into account when treating different bug types, would be needed: indeed, in our study, we limit ourselves to observing that not all bugs are equal and are indeed treated differently. In order to evaluate the bug type prediction model, we measured the performance using a number of different indicators such as precision, recall, F-Measure, AUC-ROC, and MCC, which can provide a wide overview of the model performance. As for the validation methodology, we relied on 10-fold cross validation. While such a strategy has recently been criticized (Tantithamthavorn et al., 2017), we tackled its main issue, i.e., the randomness of the splits, by running it 100 times. Finally, it is worth noting that before running the model, we configured its parameters using the Grid Search algorithm (Bergstra and Bengio, 2012). Given the nature of the validation strategy adopted, we discussed the overall prediction capabilities of the classification model, while we did not provide the detailed confusion matrix: however, this was not possible in our case because we built 1,000 different confusion matrices due to the 100-times 10-fold cross validation. This would have made the interpretation of the results hard.

6.3. External validity

Threats in this category mainly concern the generalizability of the results. We conducted this study on a large sample of 1280 bug reports publicly available on the bug tracking platforms of the considered ecosystems. Such a sample allowed us to get bug reports belonging to 119 different projects. However, we are aware that the proposed taxonomy may differ when considering other systems or closed-source projects. Similarly, the performance of our bug type classification model might be lower/higher on different projects than the ones reported herein.

7. Conclusion and future directions

Not all bugs are the same. Understanding their type can be useful for developers during the first and most expensive activity of bug triaging (Akila et al., 2015), i.e., the diagnosis of the issue the bug report refers to. While several previous works mainly focused on supporting the bug triage activity with respect to the identification of the most qualified developer that should take care of it (Murphy and Cubranic, 2004; Javed et al., 2012), they basically treat all bugs in the same manner without considering their type (Zaman et al., 2011).

In this paper, we started facing this limitation, by proposing (i) an empirical assessment of the possible bug types, (ii) a characterization study of the different bug types identified, and (iii) a classification model able to classify bugs according to their type.

To this aim, we first proposed a novel taxonomy of bug types, conducting an iterative content analysis on 1280 bug reports of 119 software projects belonging to three large ecosystems such as Mozilla, Apache, and Eclipse. Then, we studied the discovered bug types under three different perspectives such as (i) frequency of appearance, (ii) principal topics present in the corresponding bug reports, and (iii) time required to fix them. Finally, we devised a bug type prediction model that classifies bug reports according to the related type. We empirically evaluated ourbug type classification model by running it against the dataset that came out of the taxonomy building phase, measuring its performance adopting a 100 times 10-fold cross validation methodology in terms of F-Measure, AUC-ROC, and Matthew's Correlation Coefficient (MCC).

The results of the study highlight nine different bug types behind the bugs reported in bug reports, that span across a broad set of issues (e.g., GUI-related vs configuration bugs) and are widespread over the considered ecosystems. We observed that the bug types we discovered are treated differently with respect to the process developers follow to fix them. The proposed bug type classification model reached an overall F-Measure, AUC-ROC, and MCC of 64%, 74%, and 72%, respectively, showing good performance when adopted for the classification of the most diffused bug types.

Our future research agenda focuses on improving the devised model and better characterizing bugs referring to different types. Furthermore, we plan to exploit the proposed classification in other contexts: for instance, we envision the proposed taxonomy to be successfully employed for post-mortem analysis of bugs, as argued by Thung et al. (2012); at the same time, we will investigate whether bug prioritization approaches can benefit from information on the nature of bugs, e.g., security issues might be considered more important than GUI-related ones.

References

Akila, V., Zayaraz, G., Govindasamy, V., 2015. Effective bug triage–a framework. Procedia Comput. Sci. 48, 114–120.

Antoine, J.-Y., Villaneau, J., Lefeuvre, A., 2014. Weighted Krippendorff's alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation. In: EACL 2014, p. 10p.

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., Guéhéneuc, Y.-G., 2008. Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds. ACM, p. 23.

Anvik, J., 2006. Automating bug report assignment. In: Proc. Int'l Conference on Software Engineering (ICSE). ACM, pp. 937–940.

Anvik, J., Hiew, L., Murphy, G.C., 2006. Who should fix this bug? In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 361–370.

Anvik, J., Murphy, G.C., 2011. Reducing the effort of bug report triage: recommenders for development-oriented decisions. ACM Trans. Softw. Eng. Methodol. 20 (3), 10.

Aranda, J., Venolia, G., 2009. The secret life of bugs: going past the errors and omissions in software repositories. In: Proceedings of the International Conference on Software Engineering (ICSE). IEEE Computer Society, pp. 298–308.

Aslam, T., Krsul, I., Spafford, E., 1996. Use of A Taxonomy of Security Faults. Purdue University.

Baeza-Yates, R.A., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc.

Baldi, P., Brunak, S., Chauvin, Y., Andersen, C.A., Nielsen, H., 2000. Assessing the accuracy of prediction algorithms for classification: an overview. Bioinformatics 16 (5), 412–424.

Bauer, M.W., 2007. Content analysis. an introduction to its methodology—by Klaus Krippendorff from words to numbers. narrative, data and social science–by roberto franzosi. Br. J. Sociol. 58 (2), 329–331.

Bavota, G., Linares-Vasquez, M., Bernal-Cardenas, C.E., Di Penta, M., Oliveto, R., Poshyvanyk, D., 2015. The impact of API change-and fault-proneness on the user ratings of android apps. IEEE Trans. Softw. Eng 41 (4), 384–407.

Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., Marinov, D., 2018. Deflaker: automatically detecting flaky tests. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM.

Beller, M., Georgios, G., Panichella, A., Proksch, S., Amann, S., Zaidman, A., 2019. Developer testing in the IDE: patterns, beliefs, and behavior. IEEE Trans. Softw. Eng (1) 1-1.

Beller, M., Gousios, G., Panichella, A., Zaidman, A., 2015. When, how, and why developers (do not) test in their ides. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, pp. 179–190.

Beller, M., Gousios, G., Zaidman, A., 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, pp. 356–367.

Beller, M., Spruit, N., Spinellis, D., Zaidman, A., 2018. On the dichotomy of debugging behavior among programmers. In: Proceedings of the 40th International Conference on Software Engineering (ICSE). ACM, pp. 572–583.

Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. J. Mach. Learn. Res. 13 (February), 281–305.

Bettenburg, N., Just, S., Schröter, A., Weiß, C., Premraj, R., Zimmermann, T., 2007. Quality of bug reports in eclipse. In: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange. ACM, pp. 21–25.

Bezemer, C.-P., McIntosh, S., Adams, B., German, D.M., Hassan, A.E., 2017. An empirical study of unspecified dependencies in make-based build systems. Empirical Softw. Eng 22 (6), 3117–3148.

Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent Dirichlet allocation. J. Mach. Learn. Res. 3 (January), 993–1022.

Breu, S., Premraj, R., Sillito, J., Zimmermann, T., 2010. Information needs in bug reports: improving cooperation between developers and users. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW). ACM, pp. 301–310.

Bruning, S., Weissleder, S., Malek, M., 2007. A fault taxonomy for service-oriented architecture. In: 10th IEEE High Assurance Systems Engineering Symposium, 2007. HASE'07. IEEE, pp. 367–368.

Buglione, L., Abran, A., 2006. Introducing root-cause analysis and orthogonal defect classification at lower CMMI maturity levels. Proc. MENSURA 910, 29–40.

Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F., 2018. Not all bugs are the same:understanding, characterizing, and classifying bug types—online appendix https://figshare.com/s/dcb95c70c4472b2ac935.

Chan, K.M., Bishop, J., Steyn, J., Baresi, L., Guinea, S., 2007. A fault taxonomy for web service composition. In: International Conference on Service-Oriented Computing. Springer, pp. 363–375.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. Smote: synthetic minority over-sampling technique. J. Artif. Intell. Res. 16, 321–357.

Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.-Y., 1992. Orthogonal defect classification—a concept for in-process measurements. IEEE Trans. Softw. Eng. 18 (11), 943–956.

Chowdhury, G.G., 2003. Natural language processing. Annu. Rev. Inf. Sci. Technol. 37 (1), 51–89.

Di Penta, M., Cerulo, L., Aversano, L., 2008. The evolution and decay of statically detected source code vulnerabilities. In: Eighth IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 101–110.

Freimut, B., Denger, C., Ketterer, M., 2005. An industrial case study of implementing and validating defect classification for process improvement and quality management. In: Software Metrics, 2005. 11th IEEE International Symposium. IEEE, pp. 10–pp.

Goldberg, Y., Levy, O., 2014. word2vec explained: deriving Mikolov et al.'s negative-sampling word-embedding method. arXiv:1402.3722.

Gousios, G., Zaidman, A., Storey, M.-A., Van Deursen, A., 2015. Work practices and challenges in pull-based development: the integrator's perspective. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 358–368.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. Developing fault-prediction models: what the research can show industry. IEEE Softw. 28 (6), 96–99.

Hecking, T., Leydesdorff, L., 2018. Topic modelling of empirical text corpora: Validity, reliability, and reproducibility in comparison to semantic maps. arXiv:1806.01045.

Hernández-González, J., Rodriguez, D., Inza, I., Harrison, R., Lozano, J.A., 2018. Learning to classify software defects from crowds: a novel approach. Appl. Soft Comput. 62, 579–591.

Herzig, K., Just, S., Zeller, A., 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 392–401.

Hooimeijer, P., Weimer, W., 2007. Modeling bug report quality. In: Proceedings of the International Conference on Automated Software Engineering (ASE). ACM, pp. 34–43.

Huang, L., Ng, V., Persing, I., Chen, M., Li, Z., Geng, R., Tian, J., 2015. AutoODC: automated generation of orthogonal defect classifications. Autom. Softw. Eng. 22 (1), 3–46.

Javed, M.Y., Mohsin, H., et al., 2012. An automated approach for software bug classification. In: 2012 Sixth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS). IEEE, pp. 414–419.

Jeong, G., Kim, S., Zimmermann, T., 2009. Improving bug triage with bug tossing graphs. In: Proceedings of the Joint Meeting of the European Software Engineering Conference & the Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM, pp. 111–120.

Jovic, M., Adamoli, A., Hauswirth, M., 2011. Catch me if you can: performance bug detection in the wild. In: ACM SIGPLAN Notices, 46. ACM, pp. 155–170.

Lal, S., Sureka, A., 2012. Comparison of seven bug report types: a case-study of Google chrome browser project. In: Software Engineering Conference (APSEC), 2012 19th Asia-Pacific, 1. IEEE, pp. 517–526.

Le, Q., Mikolov, T., 2014. Distributed representations of sentences and documents. In: International Conference on Machine Learning, pp. 1188–1196.

Leszak, M., Perry, D.E., Stoll, D., 2002. Classification and evaluation of defects in a project retrospective. J. Syst. Softw. 61 (3), 173–187.

Lidwell, W., Holden, K., Butler, J., 2010. Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach Through Design, 2nd ed. Rockport Publishers.

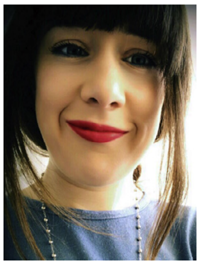
Lowis, L., Accorsi, R., 2011. Vulnerability analysis in SOA-based business processes. IEEE Trans. Serv. Comput. 4 (3), 230–242.

Luo, Q., Hariri, F., Eloussi, L., Marinov, D., 2014. An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 643–653.

McDonnell, T., Ray, B., Kim, M., 2013. An empirical study of API stability and adoption in the android ecosystem. In: Proc. Int'l Conf. on Software Maintenance (ICSM). IEEE, pp. 70–79.

Memon, A.M., 2002. GUI testing: pitfalls and process. Computer 35 (8), 87–88.

Meyer, A.N., Fritz, T., Murphy, G.C., Zimmermann, T., 2014. Software developers' perceptions of productivity. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 19–29.
Mileva, Y.M., Dallmeier, V., Burger, M., Zeller, A., 2009. Mining trends of library usage. In: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops. ACM, pp. 57–62.
Mockus, A., Fielding, R.T., Herbsleb, J.D., 2002. Two case studies of open source software development: Apache and Mozilla. ACM Trans. Softw. Eng. Methodol. 11 (3), 309–346.
Moradian, E., Håkansson, A., 2006. Possible attacks on xml web services. IJCSNS Int. J. Comput. Sci. Netw. Secur. 6 (1B), 154–170.
Murphy, G., Cubranic, D., 2004. Automatic bug triage using text categorization. In: Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE), pp. 92–97.
Nagwani, N., Verma, S., Mehta, K.K., 2013. Generating taxonomic terms for software bug classification by utilizing topic models based on latent Dirichlet allocation. In: 2013 11th International Conference on ICT and Knowledge Engineering (ICT&KE). IEEE, pp. 1–5.
Nasrabadi, N.M., 2007. Pattern recognition and machine learning. J. Electron. Imaging 16 (4), 049901.
Ostrand, T.J., Weyuker, E.J., 1984. Collecting and categorizing software error data in an industrial environment. J. Syst. Softw. 4 (4), 289–300.
Palomba, F., Linares-Vásquez, M., Bavota, G., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A., 2018. Crowdsourcing user reviews to support the evolution of mobile apps. J. Syst. Softw. 137, 143–162.
Palomba, F., Salza, P., Ciurumelea, A., Panichella, S., Gall, H., Ferrucci, F., De Lucia, A., 2017. Recommending and localizing change requests for mobile apps based on user reviews. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press, pp. 106–117.
Palomba, F., Zaidman, A., 2019. The smell of fear: on the relation between test smells and flaky tests. Empirical Softw. Eng. Springer. In press.
Palomba, F., Zaidman, A., 2017. Does refactoring of test smells induce fixing flaky tests? In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 1–12.
Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A., 2013. How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 522–531.
Pantic, M., Pentland, A., Nijholt, A., Huang, T.S., 2007. Human computing and machine understanding of human behavior: a survey. In: Artifical Intelligence for Human Computing. Springer, pp. 47–71.
Pascarella, L., Spadini, D., Palomba, F., Bruntink, M., Bacchelli, A., 2018. Information needs in contemporary code review. Proc. ACM Hum. Comput. Interaction 2 (CSCW), 135.
Peng, J., Heisterkamp, D.R., Dai, H., 2001. LDA/SVM driven nearest neighbor classification. In: Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on, 1. IEEE, p. I.
Porter, M.F., 1980. An algorithm for suffix stripping. Program 14 (3), 130–137.
Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., Devanbu, P., 2016. On the naturalness of buggy code. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 428–439.
Refaeilzadeh, P., Tang, L., Liu, H., 2009. Cross-validation. In: Encyclopedia of database systems. Springer, pp. 532–538.
Robbes, R., Lungu, M., Röthlisberger, D., 2012. How do developers react to API deprecation?: the case of a smalltalk ecosystem. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, p. 56.
Salton, G., Buckley, C., 1988. Term-weighting approaches in automatic text retrieval. Inf. Process. Manage. 24 (5), 513–523.
Salza, P., Palomba, F., Di Nucci, D., D'Uva, C., De Lucia, A., Ferrucci, F., 2018. Do developers update third-party libraries in mobile apps? In: Proceedings of the 26th Conference on Program Comprehension. ACM, pp. 255–265.
Schröter, A., Zimmermann, T., Premraj, R., Zeller, A., 2006. If your bug database could talk. In: Proceedings of the 5th International Symposium on Empirical Software Engineering, 2, pp. 18–20.
Shokripour, R., Anvik, J., Kasirun, Z.M., Zamani, S., 2013. Why so complicated? Simple term filtering and weighting for location-based bug report assignment recommendation. In: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on. IEEE, pp. 2–11.
Stebbins, R.A., 2001. Exploratory Research in the Social Sciences, 48. SAGE.
Stone, M., 1974. Cross-validatory choice and assessment of statistical predictions. J. R. Stat. Soc. Ser. B 111–147.
Sun, C., Lo, D., Wang, X., Jiang, J., Khoo, S.-C., 2010. A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. ACM, pp. 45–54.
Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C., 2014. Bug characteristics in open source software. Empirical Softw. Eng. 19 (6), 1665–1705.
Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2017. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. 43 (1), 1–18.
Thung, F., Le, X.-B.D., Lo, D., 2015. Active semi-supervised defect categorization. In: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension. IEEE Press, pp. 60–70.
Thung, F., Lo, D., Jiang, L., 2012. Automatic defect categorization. In: 2012 19th Working Conference on Reverse Engineering (WCRE). IEEE, pp. 205–214.
Tian, Y., Wijedasa, D., Lo, D., Le Goues, C., 2016. Learning to rank for bug report assignee recommendation. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). IEEE, pp. 1–10.
Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2016. An empirical investigation into the nature of test smells. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 4–15.
Twidale, M.B., Nichols, D.M., 2005. Exploring usability discussions in open source development. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 2005. HICSS'05. IEEE, p. 198c.
Vahabzadeh, A., Fard, A.M., Mesbah, A., 2015. An empirical study of bugs in test code. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 101–110.
Wei, X., Croft, W.B., 2006. LDA-based document models for ad-hoc retrieval. In: Proceedings of the 29th annual international ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp. 178–185.
Xia, X., Lo, D., Wang, X., Zhou, B., 2014. Automatic defect categorization based on fault triggering conditions. In: Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on. IEEE, pp. 39–48.
Xuan, J., Jiang, H., Hu, Y., Ren, Z., Zou, W., Luo, Z., Wu, X., 2015. Towards effective bug triage with software data reduction techniques. IEEE Trans. Knowl. Data Eng. 27 (1), 264–280.
Xuan, J., Jiang, H., Ren, Z., Yan, J., Luo, Z., 2017. Automatic bug triage using semi-supervised text classification. arXiv preprint, arXiv:1704.04769.
Zaidman, A., Van Rompaey, B., Demeyer, S., van Deursen, A., 2008. Mining software repositories to study co-evolution of production & test code. In: First International Conference on Software Testing, Verification, and Validation (ICST). IEEE Computer Society, pp. 220–229.
Zaidman, A., Van Rompaey, B., van Deursen, A., Demeyer, S., 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Softw. Eng. 16 (3), 325–364. doi:10.1007/s10664-010-9143-7.
Zaman, S., Adams, B., Hassan, A.E., 2011. Security versus performance bugs: a case study on firefox. In: Proceedings of the Working Conference on Mining Software Repositories (MSR). ACM, pp. 93–102.
Zeller, A., 2009. Why Programs Fail—A Guide to Systematic Debugging, second ed. Academic Press.
Zhang, F., Khomh, F., Zou, Y., Hassan, A.E., 2012. An empirical study on factors impacting bug fixing time. In: 2012 19th Working Conference on Reverse Engineering. IEEE, pp. 225–234.
Zhang, T., Jiang, H., Luo, X., Chan, A.T., 2016. A literature review of research in bug resolution: tasks, challenges and future directions. Comput. J. 59 (5), 741–773.
Zhang, T., Lee, B., 2013. A hybrid bug triage algorithm for developer recommendation. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, pp. 1088–1094.
Zhang, Y., Chen, Y., Cheung, S.-C., Xiong, Y., Zhang, L., 2018. An empirical study on TensorFlow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018). ACM, New York, NY, USA, pp. 129–140.
Zhou, Y., Tong, Y., Gu, R., Gall, H., 2016. Combining text mining and data mining for bug report classification. J. Softw. Evol. Process 28 (3), 150–176.
Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., Weiss, C., 2010. What makes a good bug report? IEEE Trans. Softw. Eng. 36 (5), 618–643.



Gemma Catolino received the master's degree in computer science from the University of Salerno, Italy, in 2016. She is currently working toward the Ph.D. degree at the University of Salerno, Italy, under the supervision of Prof. Filomena Ferrucci. Her research interests include effort estimation, software maintenance and evolution, mining software repositories, and empirical software engineering. She is a student member of ACM and IEEE.

Fabio Palomba is a senior research associate at the University of Zurich (Switzerland), where he works under the Zurich Empirical Software Engineering Team (ZEST). Previously, he worked as Research Associate at the Delft University of Technology (The Netherlands). He received the European Ph.D. Degree from the University of Salerno (Italy). His research activities include technical debt management, source code quality, predictive analytics for fault-, change-, and effort-prediction, social aspects of software engineering, and mobile app evolution. He has published more than 30 papers on these topics in international journals and conference proceedings. He serves and has served as referee for international journals and program committee member of several international conferences. He is a member of IEEE and ACM.

Andy Zaidman is an associate professor at the Delft University of Technology, the Netherlands. He obtained his M.Sc. (2002) and Ph.D. (2006) in Computer Science from the University of Antwerp, Belgium. His main research interests are 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 Andy Zaidman was the laureate of a prestigious Vidi career grant from the Dutch Science Foundation NWO.

Filomena Ferrucci is professor of software engineering and software project management at University of Salerno, Italy. Her main research interests include software metrics, effort estimation, search-based software engineering, empirical software engineering, and human–computer interaction. She has been program co-chair of the International Summer School on Software Engineering. Web page: http://docenti.unisa.it/ 001775/home.