What Makes a Satisficing Bug Report?

Tommaso Dal Sasso,[1] Andrea Mocci,[1] Michele Lanza,[1] Marco Vit,[2] Alberto Bacchelli[2]
[1]*REVEAL @ Faculty of Informatics – University of Lugano, Switzerland*
[2]*Delft University of Technology, The Netherlands*

***Abstract*—To ensure quality of software systems, developers use bug reports to track defects. It is in the interest of users and developers that bug reports provide the necessary information to ease the fixing process. Past research found that users do not provide the information that developers deem *ideally useful* to fix a bug. This raises an interesting question: What is the *satisficing* information to speed up the bug fixing process?**

We conducted an observational study on the relation between provided report information and its lifetime, considering more than 650,000 reports from open-source systems using popular bug trackers. We distilled a meta-model for a *minimal* bug report, establishing a basic layer of core features. We found that few fields influence the resolution time and that customized fields have little impact on it. We performed a survey to investigate what users deem easy to provide in a bug report.

I. Introduction

When users file a bug report for a software project, their main hope is that developers will fix it quickly, to minimize its impact. But what information should they provide to make this happen? There is a stark mismatch between what developers perceive as *optimally* useful in this respect (*i.e.,* steps to reproduce, stack traces, and test cases) and what users are able, or willing to provide [24].

Bug tracking systems and software projects should define a reasonable common ground for information to be provided in bug reports, so that it is not too demanding for users, yet provides enough information to developers. Nevertheless, considering what popular issue trackers and projects (*e.g., Bugzilla*,[1] *JIRA*,[2] *FogBugz*,[3] and the issue tracker provided with *GitHub*[4]) demand from users, we see that it is quite diverse and specialized. In particular, each bug tracking system provides a core set of common fields, which are complemented with additional fields that reflect their domains and custom data customizable by project owners. For example, the commercial issue tracker JIRA defines several fields that describe in detail aspects pertaining to the time management of issues, while the GitHub issue tracker provides a minimal (and criticized[5]) model that, together with the integration with the Git versioning system, conceives a bug report as a conversation among developers, fostering the philosophy of collaborative development proposed by GitHub [20]. Overall, there is currently no consensus among software projects and creators of bug reporting systems on essential mandatory fields to be filled by users in each report, optional fields that give useful additional information, and free space for users willing to provide more detailed descriptions.

Our vision is to define the minimum set of information needed to describe a software defect, to clarify what should be required by each bug reporting system. In this paper, we make a step in this direction: We investigate what makes a *satisficing*[6] bug report. We move from defining a good or more precisely an *optimal* bug report and adopt a more pragmatic view on what users should provide.

We conduct our investigation in three steps: (1) We investigate what users and developers perceive as *difficult* in writing a report, by means of an online questionnaire; (2) we investigate the usage and evolution of issue tracker data, by means of a large-scale quantitative analysis of the status changes in submitted bug reports and the impact that customized fields have on the resolution of a defect; (3) we study which fields developers use to describe defects, by means of a further quantitative analysis on the lifetime of reports in relation to the evolution of report's state and its *completeness* in its core and customized fields.

Our results show that providing more fields in a report relates to the fixing time: In particular, the bug reports with longer descriptions tend to be solved quicker. While this might be intuitive, issue trackers still do not emphasize this aspect during the submission of a new bug report, putting the accent on the customization capabilities of the platform. At the same time, the project-specific bug report fields have little impact to the fixing time. This paper provides insights on the current issue tracking practices and defines guidelines in building the foundations for a new model of an issue tracking system.

The contributions of this work are:

- A survey that identifies the components of a bug report considered difficult to provide by users (Section III-B).
- A dataset of 650,000 bug reports, collected from the issue trackers of BUGZILLA and JIRA (Section III-C).
- The model for a minimal bug report, that puts the emphasis on the shared components of a bug report (Section IV)
- An analysis of the usage of the fields in an issue tracker, from active open source projects (Section IV).

[1] https://www.bugzilla.org/
[2] https://jira.atlassian.com/
[3] https://www.fogcreek.com/fogbugz/
[4] https://github.com/
[5] https://github.com/dear-github/dear-github

[6] Satisficing is a neologism coined by Simon [17], [18] combining the verbs *to satisfy* and *to suffice*, and it is used to describe a solution that is roughly satisfactory and meets some criteria of sufficiency and is better than an optimal solution that would be too complex or would imply too strong constraints.

II. Issue Tracking Systems

To obtain an overview of the salient features of issue trackers, we briefly present four platforms, selected by importance and overall adoption in open-source systems: BUGZILLA, JIRA, the GITHUB issue tracker and FOGBUGZ.

1) Bugzilla: One of the oldest and most popular issue tracking systems that influenced many other issue trackers. Developed by the Mozilla Foundation, it is used both by open source projects[7] (*e.g.,* Linux) and by industrial and government customers (*e.g.,* NASA). The Mozilla Foundation itself uses BUGZILLA to manage the issues of its projects, like Firefox. BUGZILLA includes several fields, allowing for a great level of detail in specifying an issue. However, such a freedom of choice also produces a complex interface where many of the values are often left empty, or set to their default value. Figure 1 shows a typical submission form with both fields for fixed-option choices and text boxes for narrative (*e.g.,* to describe how to reproduce the bug).

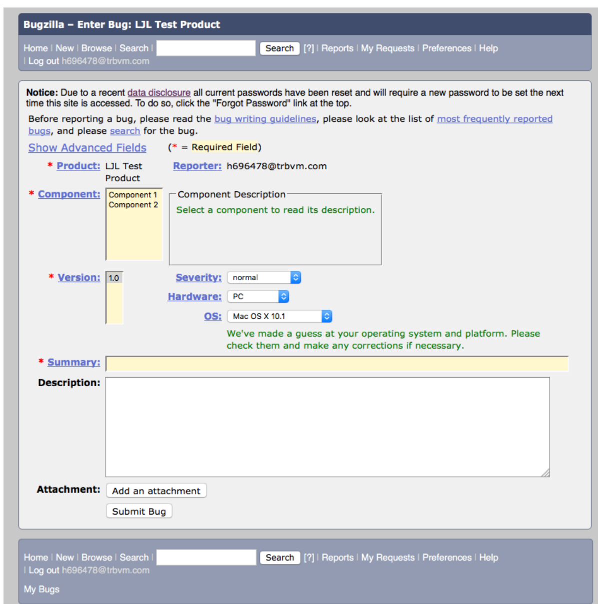

Fig. 1. Bugzilla Submission Form

2) JIRA: A successful commercial bug reporting systems used by several customers like Twitter, Linkedin, and Ebay.[8] JIRA supports the same model as BUGZILLA, augmenting its capabilities with a polished user interface and a tight integration with other development tools, especially with the version control system.

3) GitHub: A popular web-based GIT repository hosting service, used for the development of several popular open source projects. Together with the GIT hosting service, GITHUB offers a simple issue tracker to manage the defects during development. Figure 2 shows the interface of the GITHUB bug submission form.

[7]See https://www.bugzilla.org/installation-list/

[8]See https://www.atlassian.com/company/customers

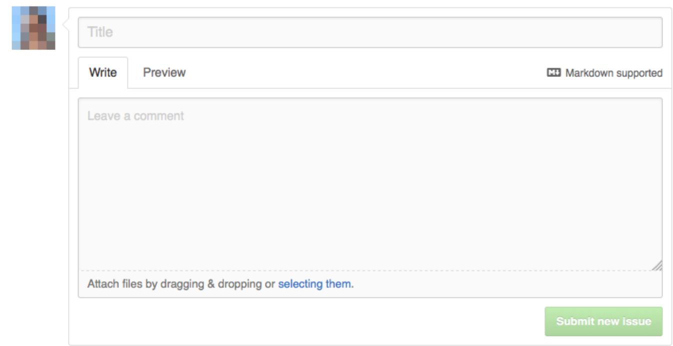

Fig. 2. GitHub Bug Submission Form

GITHUB adopts a simplified model of a bug report and a strong integration with the source code (thus giving the possibility to link issues with specific commits).

4) FogBugz: In this system, the model of a bug report is similar to BUGZILLA, but more polished and user-friendly, due to its clean user interface and filtering capabilities. Differently from GITHUB, FOGBUGZ[9] does not simplify the report model, but it lets users add custom filters and views.

Reflection. To understand the essential traits of bug reports, we analyze how the data included in bug reports influences their lifetime. We next analyze the features of a set of bug reporting systems, to distill a model of common/specific fields for their bug reports. This model serves as a basis for further empirical analysis, to determine how these commonalities and customizations influence the life of the reports.

III. Research Method

To determine what makes a satisficing bug report, we first need a way to rate the quality of a bug report, then we can conduct quantitative analysis to determine which features relate with higher quality. Measuring the quality of bug reports is hard to do in an automated and unbiased way. For this reason, researchers proposed different metrics to measure it [13], all with their limitations, but reasonable enough to be realistic. In this paper, we decide to consider the *lifetime* of a bug report (*i.e.,* the time between the opening and resolution of a defect) as a viable proxy for its quality rating, as the time spent dealing fixing software defects is crucial in reducing the time the system contains a problem. Limitations of this proxy metric include the fact that the trivial bugs, or the non-issues, are the ones that require less time to fix, and that the severity can also have a not negligible impact on how quickly developers decide to fix a problem. Nevertheless, the information shared in the bug report has to be satisficing enough to let developers understand whether it is a trivial fix, an urgent matter, or something that can wait longer. For this reason, we find lifetime of a bug report a useful approximation in aggregate statistical analyses to provide a high-level view over bug repositories.

[9]See https://www.fogcreek.com/fogbugz/

We investigate how users and developers use issue tracking systems and the impact that the provided information has on the lifetime of a bug report. According to Zimmermann *et al.* the information provided by submitter can be partial or incorrect [24]. To understand what is reasonable for a user to provide in a report, we conducted a survey asking developers what they think are the difficult elements to provide. We then focus on two of the main components that compose a bug report: (1) its state and (2) the core and optional attributes, to understand how the provided data is used.

A. Research Questions

When collecting information about software defects, it is important to know when the submitted data is reliable and accurate. Our goal is to investigate what users can easily provide and what is harder to obtain; we structure our investigation into the following question:

> **RQ1.** What are the elements that are perceived as difficult to provide when reporting a defect?

To understand the relationship between what is described in a bug report and its lifetime, we have to consider the different kind of data that reports can provide. This is not trivial, because different platforms offer different fields to provide information, with different meaning and values. As a first step for our quantitative evaluation, we investigate how to define a meta-model to comprehensively describe information stored across different issue reporting systems:

> **RQ2.** What is a comprehensive unified meta-model for describing data from different bug tracking systems?

After having defined the meta-model, we can quantitatively investigates several aspects of reports related to their lifetime and evolution. During development, a bug report changes its state, sometimes several times, ideally converging to a closed state. The changes in the state of a report are important to understand its evolution [10]. We are interested in considering the evolution of the states and see whether the aggregate of these changes can provide knowledge on the inner logic of an issue tracker. This leads to the following question:

> **RQ3.** What are the most frequent states and state transitions in bug reports?

Together with a state, a bug report comes with a set of attributes that describe the properties of a report. These attributes can also be defined by the users, to create project-specific customized fields. We investigate the completeness of core and custom fields with respect to the lifetime of a bug, considering the following research question:

> **RQ4.** Does the completeness of standard and project-specific attributes in a bug report relate to its lifetime?

To answer our questions, we both run a survey (Section III-B) and we collect, model, and analyze a large dataset of bug reports from open source projects (Section III-C).

B. Online Questionnaire

Zimmerman *et al.* asked users and developers what they think are the useful elements in a bug report and how hard it is, in their opinion, to provide those elements [24]. We proposed a similar questionnaire to the *Pharo* open source community to further understand what it is reasonable to expect from users submitting a bug report. The questionnaire is composed of two parts: (1) We collect demographic information inquiring about expertise with programming and with submitting, handling, and fixing bug reports; and (2) we collect information about respondents' perception of how difficult it is to provide different kinds of information when submitting a bug report. All the questions are formulated as statements (*e.g.,* "It is easy to provide a description of the failure") and the respondents have to declare their agreement using a 5-level Likert-type scale. We map the results into an integer scale from -2 (*i.e.,* "strongly disagree") to 2 (*i.e.,* "strongly agree").

We advertised the survey through the development mailing list of Pharo and we received a total of 22 complete responses. Table I summarizes the respondents' expertise. The respondents are experienced with object-oriented programming and with the Pharo IDE. While they have experience in submitting and handling bug reports, their experience is lower in participating in discussions about bug reports and much lower in having reports assigned to them. For this reason, we deem the respondents' sample to be in line with the aim of our survey. In fact, we are especially interested in knowing the point of view of submitters of bug reports, rather than the view of the developers that "consume" these reports.

TABLE I
EXPERTISE OF THE PARTICIPANTS OF THE SURVEY (AVERAGE)

Activity	Average
Experience with Object Oriented programming languages	1.5
Knowledge of Pharo	1.3
Have often bug reports assigned	-0.4
Often handle bug reports	0.6
Often participate in discussion in bug reports	0.3
Often submit bug reports	0.7

C. Data Collection

To understand what users and developers collect and provide in bug reports, we mined the contents of the issue trackers of several software projects. To collect real development data for our study, we consider the Apache Foundation and the Mozilla Foundation: Both platforms contain a considerable number of popular and active open source projects, with years of development history. Moreover, both platforms host several projects tracked on public, dedicated bug trackers: Mozilla uses BUGZILLA, Apache uses JIRA. They offer a public REST API to access their repositories in JSON format, allowing for a clean and reliable data collection.

TABLE II
PROJECTS IN THE DATASET

Ecosystem	Project	Issues First	Last	Count	Age (days)	Frequency
Apache	Cassandra	Mar 7, 2009	Jul 8, 2015	9,723	2,314	5h 42m
	Hadoop	Jul 24, 2005	Jul 8, 2015	10,191	3,635	8h 33m
	Lucene	Oct 9, 2001	Jul 8, 2015	6,641	5,019	18h 8m
	Maven	Nov 20, 2002	Jul 23, 2015	4,663	4,628	23h 49m
	Mahout	Jan 30, 2008	Jun 25, 2015	1,752	2,702	37h 6m
	Pig	Nov 2, 2007	Jul 7, 2015	767	2,804	87h 44m
	Sorl	Jan 25, 2006	Jul 8, 2015	7,728	3,451	10h 43m
	Zookeeper	Jun 6, 2008	Jul 3, 2015	2,207	2,582	28h 4m
Mozilla	Air Mozilla	Apr 14, 2009	Jun 16, 2015	509	2,254	106h 16m
	Bugzilla	Apr 15, 1998	Jul 27, 2015	19,395	6,312	7h48m
	Core	Mar 28, 1997	Jul 17, 2015	292,358	6,684	33m
	Firefox	Jul 30, 1999	Jul 8, 2015	155,078	5,821	54m
	Firefox for Android	Sep 11, 2008	Jul 28, 2015	18,906	2,510	19m
	SeaMonkey	Nov 10, 1995	Jul 27, 2015	92,757	7,198	1h 51m
	Thunderbird	Jan 2, 2000	Jul 8, 2015	42,247	5,666	3h13m

We built a downloader and an importer to collect the data, serialize the contents of each report, and store the polished data in a POSTGRESQL database. Table II describes our dataset.

The dataset contains more than 650,000 bug reports, 15% of which were still open during the data collection phase. Table III shows an aggregated summary of the dataset we collected. Each bug tracker has a different set of bug report states.

TABLE III
CONTENTS OF THE DATASET

	Apache	Mozilla	Total
Open issues	7,545	91,336	98,881
Closed issues	36,127	529,914	566,041
Total Issues	43,672	621,250	**664,922**

Table IV details them, for each tracker, with the counts of the bug reports for each state at the moment of the download.

TABLE IV
DIFFERENT STATES OF BUG REPORTS IN BUGZILLA AND JIRA, WITH THE COUNT OF THE REPORTS CURRENTLY IN EACH STATE AND THE TOTAL SUM OF ALL THE TIMES A BUG REPORT REACHED A STATE.

Tracker	**State**	**Current**	**Total**
JIRA	Closed	21,847	22,460
	Resolved	14,280	33,386
	Open	6,736	43,203
	Patch Available	471	18,944
	Reopened	235	3,042
	In Progress	84	2,175
	Awaiting Feedback	14	15
	Testing	4	86
	Ready to Commit	1	3
Bugzilla	RESOLVED	391,919	579,488
	VERIFIED	136,783	143,082
	NEW	65,816	353,264
	UNCONFIRMED	19,821	297,319
	ASSIGNED	3,701	129,057
	REOPENED	1,998	32,745
	CLOSED	1,212	1,537

D. Data Analysis Techniques

The large volume of data we collected allows us to explore the usage of issue trackers and to investigate the common practices of bug tracking. Understanding these aspects can help us to answer our questions and verify whether the usage of the properties of a tracker influences the life of a report.

To investigate our research questions, we adopt the following approach. To answer RQ3, we build a transition diagram of all the state changes for each issue tracker, to highlight the common patterns in the growth of a report, and we weight the diagram with the values from the dataset. To answer RQ4, we build a machine-learning-based prediction model to verify how completeness of fields of a bug report relates to its lifetime.

IV. RESULTS

RQ1: What are the elements that are perceived as difficult to provide while reporting a defect?

We asked respondents how easy it is to provide 13 different elements in a report, using a 5-level Likert scale from -2 ("strongly disagree") to 2 ("strongly agree"). Figure IV shows a summary of their answers, sorted by increasing difficulty as reported by the respondents. The majority of the users does not find excessively hard to provide most of the elements. This is due to the fact that the Pharo community is composed of experienced programmers. Interestingly, finding the assignee is not considered excessively difficult: Again, this can relate to the community experience, that has a strong core of well-known developers that work as hub when dealing with defects. The elements considered to be harder to provide are the entity (*e.g.,* class, file) that likely contains the defect, the steps to reproduce the failure, and a test case showing the defect.

Conclusion. Figure IV shows that some elements are perceived as more difficult to provide when submitting a bug report. There is a set of easier elements, like screenshots, descriptions of the failure, stack traces, and the details of the operating system and hardware. Those elements are useful in identifying the defect, but are less effective than other elements we identified to support its resolution.

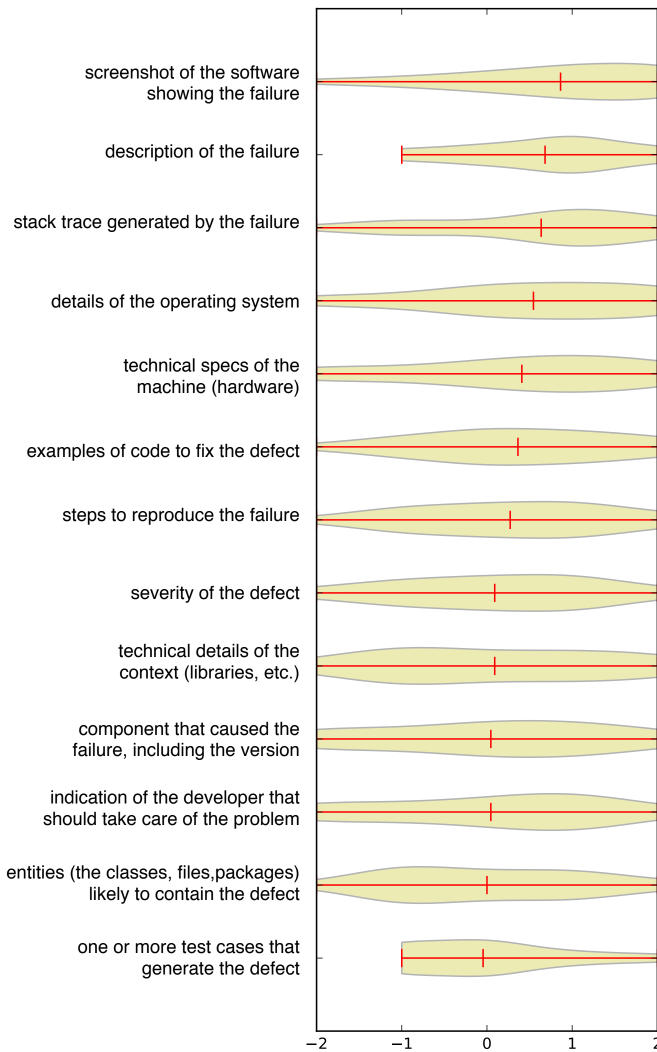

Fig. 3. Survey results: The higher the values, the easier it is to provide the corresponding information, according to the respondents' perception.

RQ2: What is a comprehensive unified meta-model for describing data from different bug tracking systems?

To devise a unified meta-model for the data we collected from the different issue trackers, we extract the model for each separate platform by reverse engineering the data and by using the documentation for the various trackers. We identify the entities that compose a bug report, the fields composing it, and the relation between the various entities. We intersect the list of each bug report and select the most common ones, to summarize the salient traits of a bug report.

Anatomy of a Report. Issue trackers are platform independent: They share a flexible common core structure to meet all the possible requirements of a software system's development process. A bug report is then built around a text description of an issue, where the user can specify the steps to reproduce the issue or include snippets of code that exemplify the context where the issue may happen.

The text description is complemented by additional metadata, used to improve the report and to track the evolution of the bug, and it can also contain attachments, like stack traces and patches. While the description of the issue and the possibility to attach files is common to all issue trackers, the metadata used to integrate the description differ in each platform. We can classify these attributes in three layers:

- *Common:* metadata in every report in each platform, *i.e.,* the core set of attributes that describes a bug report.
- *Platform specific:* metadata that are used throughout a single platform.
- *Project specific:* custom metadata set by the users, used in a single project.

The Model. From the list of entities in a tracker and their list of metadata, we built a model to access the data. Given our focus, we present a view of the model from the submitter's point of view. Figure 4 shows the conceptual diagram of the unified model for a typical bug tracking system with the frequencies of use for the common fields and trimmed of the post-report information.

- *Issue:* The main entity representing a bug report, with the text description and the metadata provided by the user.
- *Comment:* User-provided additional information on a report.
- *Edit:* A change in the existing report. It can group several changes.
- *AttributeEdit:* A change to a single element: It contains the modified attribute, the added, and removed text.
- *Link:* The relation (if any) to another report. A link maps the connection and defines the type of relation (*e.g.,* parent or duplicate).
- *Project:* The project the issue tracker refers to (*e.g.,* Firefox).
- *Product:* A single instance of an issue tracking platform (*e.g.,* Bugzilla or JIRA).
- *Component:* The area of the code affected by the defect.
- *Versions:* The software version(s) where the bug was observed.
- *Milestone:* The software version(s) targeted for a fix, for planning purposes.

There are additional attributes that are not present in every platform. To map these specific elements, there are entities that derive from ISSUE (*e.g.,* BUGZILLA_ISSUE).

These entities contain the fields `other_fields` and `custom_fields`. These are two *dictionary* fields that collect all the fields that are not represented in each model, in an unstructured fashion. The field `other_fields` contains the information from a specific bug tracker, shared in all the projects in that database (like the field `alias` in BUGZILLA). The field `custom_fields` contains non standard attributes that are customized by the maintainer of each project. For example, the attribute `cf_status_firefox41`, of the project FIREFOX in BUGZILLA. Some fields may seem redundant: For example, the field `updated_at` of ISSUE could be derived by the information contained it the EDITS; we tolerate a small degree of duplication of the data, in exchange for flexibility and completeness with different bug reporting systems.

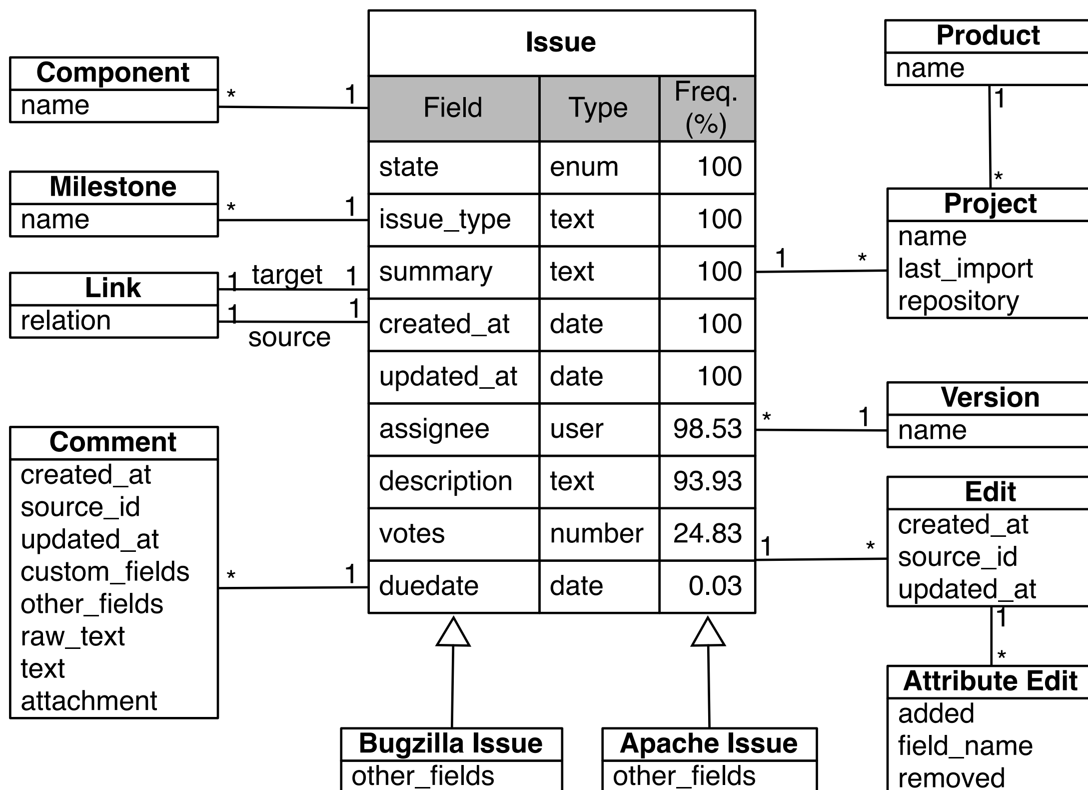

Fig. 4. Conceptual diagram of the model of a new bug report

RQ3: What are the recurrent states and transitions in reports?

We tracked the evolution of bug reports using the `state` attribute, which is an enumeration from a set of predefined states. Table IV shows the states used in the two bug trackers we consider. Each platform proposes different conventions to map the state of a report. Often, different projects use the same states in a different context and a different distribution, *e.g.,* bug reports in JIRA converge toward the CLOSED state, while in BUGZILLA they converge toward a state called RESOLVED. We analyze the state changes by building a transition graph, with an approach similar to the one used by D'Ambros *et al.* [10].

Figure 5 and Figure 6 show the transition diagrams for JIRA and BUGZILLA obtained by the collected data.

In the diagrams each node is a state, where the area grows with the number of reports that traverse that state, as presented in Table IV. Each arc between two states indicates a transition from one state to another and its width represents the total number of transitions. The diagram excludes all edges that make up less than 1% of all the transitions. Given Figure 5 and Figure 6 we can classify the states in three groups:

- *Active states:* The first group contains the most active states (*i.e.,* touched by the majority of bug reports), that are often involved in loops between them.
- *Intermediate states:* These states (*e.g.,* TESTING, IN PROGRESS, REOPENED) indicate states where an action is taking place or expected (*e.g.,* a patch is waiting for review or the continuous integration server is running the tests).
- *Unused states:* Some states are rarely used: AWAITING FEEDBACK and READY TO COMMIT. They represent some corner cases that detail extremely specific aspects of the fixing activity. Their very low usage may hint at a little interest in tracking these aspects in this way.

The analysis highlights that some projects do adopt customized states to track the intermediate aspects of their projects' workflow, but they tend to be not used in practice.

Conclusion. The analysis on the usage of the states in Section IV seems to suggest that:

- A simple model with a few states, as the one described by D'Ambros *et al.* [10], satisfies the need of tracking the state of an issue;
- Adding customized values to describe additional specific and intermediate steps in the fixing process is not working to track a better evolution of the state of a report.

The latter aspect is strengthened by the fact that JIRA offers less states than BUGZILLA, but these additional states are rarely used in practice.

RQ4: Does the completeness of standard and project-specific attributes in a bug report relate to its lifetime?

To investigate the impact that the fields have on solving a defect, we considered the *lifetime* (defined as the time to the final fix) of the closed reports.

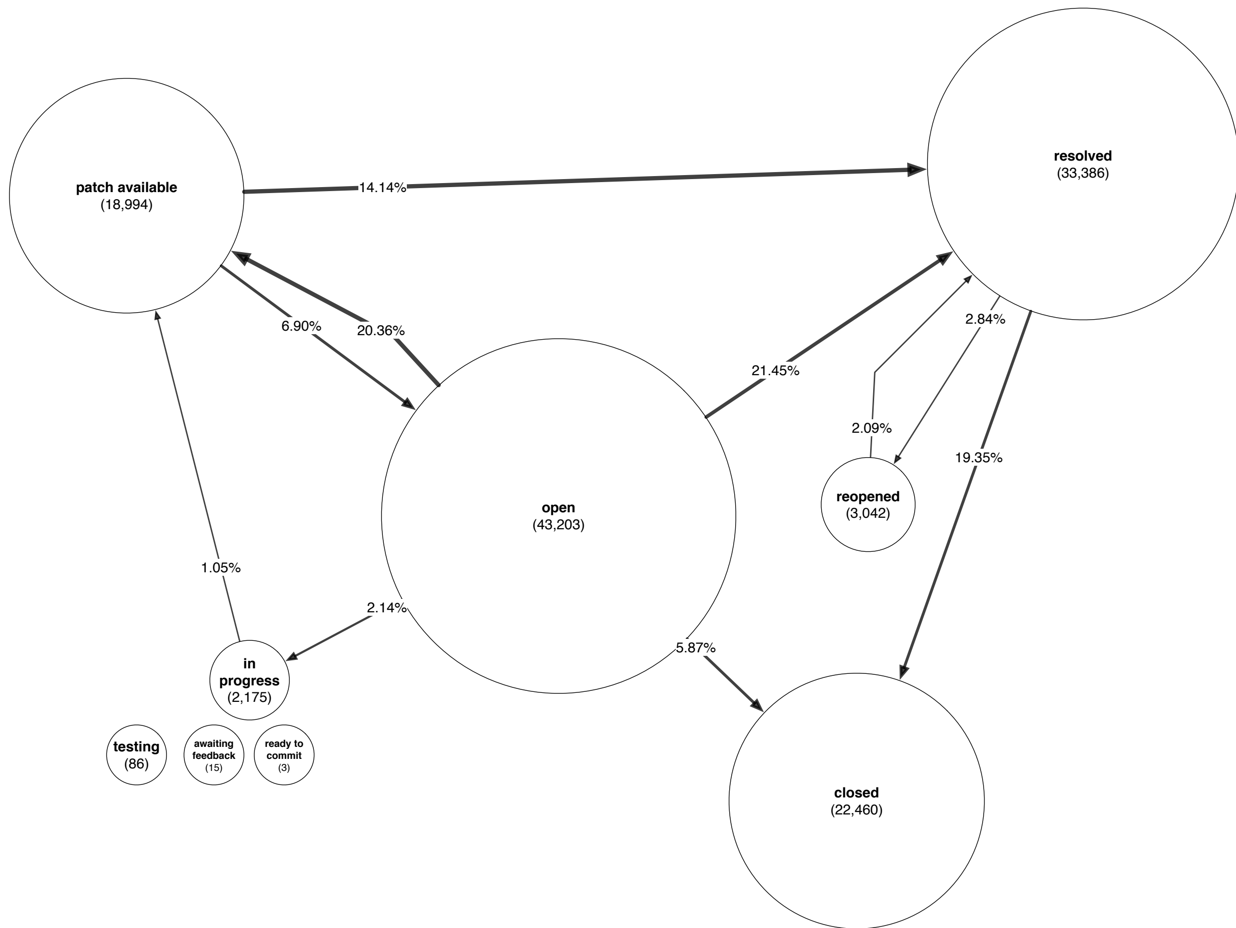

Fig. 5. Transition graph of all the states in JIRA

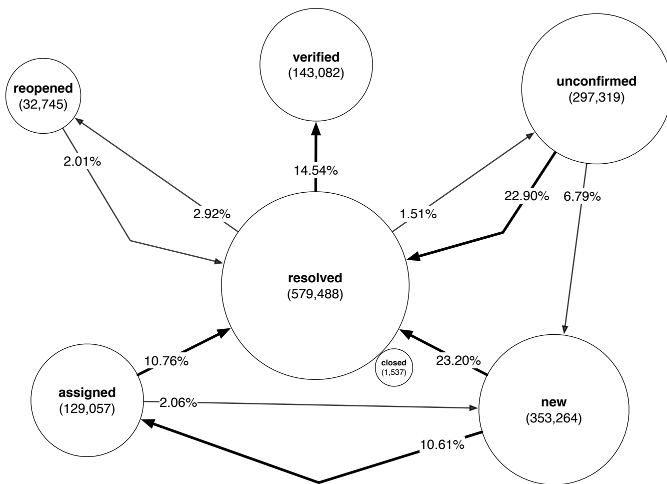

Fig. 6. Transition diagram of all the states in BUGZILLA

In addition to its standard set of attributes, each issue tracker we consider allows projects to define additional fields to customize the structure of a bug report. In our study, we group all the attributes in three *layers*:

- *Core Fields:* The fields that are common to all projects and all the issue trackers. They map the essential information to describe a software defect;
- *Tracker-Specific Fields:* The fields that are shared among all the projects in an issue tracker, but are not present in all the platforms;
- *Project-Specific Fields:* The fields that are customized by the user and appear only in a single project.

Each project in our dataset specifies its own set of custom fields. We also investigate whether these fields have a measurable impact on the lifetime of a bug report. Table V shows a count of project-specific attributes in our dataset.

We now explore the relation between the various attributes adopted by the different platforms/projects have with the effectiveness of a bug report, measured as its lifetime.

Preparing the data. To interact with the dataset, we created a *vector space model* to allow us to test statistical and machine learning approaches. Predicting the exact lifetime of a report would be unpractical and unnecessary: a timeframe for the resolution would provide a useful, human-understandable measure, while allowing more accurate predictions. To introduce such a degree of tolerance, we divide the reports into *buckets* according to their lifetime.

TABLE V
NUMBER OF CUSTOM FIELDS PER PROJECT

Project	# fields	Avg. lifetime (d)	Max lifetime (d)
Air Mozilla	5	154	1,004
Bugzilla	5	343	5,650
Core	142	227	5,936
Firefox	112	235	5,314
Firefox for Android	89	76	2,176
SeaMonkey	90	278	5,437
Thunderbird	74	259	5,451
Cassandra	11	61	1,728
Hadoop	13	172	3,012
Lucene	13	182	3,787
Mahout	11	94	1,235
Maven	9	402	3,443
Pig	11	72	2,149
Sorl	7	148	2,858
Zookeeper	12	158	2,108

Using bucketing we can deal with discrete values and adopt a classification approach, as opposed to a regression to predict a continuos variable. We split the lifetime space into four buckets: less than one day, less than one week, less than one month, and more than one month. We chose these intervals because they reflect humane time periods and they describe increasing timespans, reflecting that the longer a bug report stays open, the less relevant its exact resolution time becomes. After bucketing the issues, we model each report as a vector of booleans (each field maps an attribute of the report and its value is 1 iff the user filled it) and associate it with its classification into a bucket of lifetime, which we can feed to different prediction algorithms.

Principal Component Analysis. To understand the relation between the completeness of a bug report and the fixing time of a defect, we want to inspect how much each field contributes to the lifetime of a bug report. For this purpose, we use *Principal Component Analysis* (PCA) [1] to extract the variance between the different fields. PCA is a statistical procedure that aims to extract only the salient features from a data table. PCA transforms the existing data into variables called *principal components*, which are described as a linear combination of the existing features. The other features are then projected on the principal components.

The components extracted by PCA represent the eigenvectors of the covariance matrix. Internally, PCA implements a *single value decomposition* to extract the scores of the factors.

We use PCA to determine which combination of fields carries the most information with respect to the lifetime of a defect, by observing which elements are selected to compose the principal components. To interpret the results and obtain a general set of fields that influence the lifetime of a bug report, we consider the core fields of the projects. This operation gives us the important fields that impact the lifetime of a bug report.

After running PCA, we obtain a set of new components that can be used to map the dataset. We are not interested in the new features per-se, but — since the features of the dataset are the fields of the bug reports — we investigate which original features were selected to describe the components.

We then inspect how the components are calculated, obtaining the following selected fields:

- *assignee_id*: the person the bug report is assigned to;
- *creator_id*: the person that submitted the bug report;
- *description*: the number of words in the description of a bug report;
- *duedate*: if the bug report has a due date;
- *reporter_id*: the person that initially reported the defect (can be different than the creator)
- *summary*: the number of words in the summary of the bug report.

These fields were extracted by the algorithm as the most relevant in impacting the lifetime of a bug report. Although they do not represent the whole amount of information that is needed to describe a software defect, the fact that they were selected by PCA indicates that their contribution in determining the lifetime of a report is significant. It follows that users and developers should take these elements into account when submitting a bug report and the issue tracker should ensure that these fields are exploited accordingly.

Predicting the Lifetime of a Defect. We studied the core fields that are the most relevant in impacting the lifetime. Now we investigate how the lifetime gets influenced by the different fields defined by each project. For such an analysis PCA is not suited, as the data is too sparse and the features would be discarded in the process. We therefore adopt a *machine learning* approach to estimate an approximate lifetime of a bug report given its "completeness," *i.e.,* the number of completed fields when submitted.

We verify the impact on the prediction of the different levels of attributes using various machine learning algorithms on our model, by employing the SCIKIT-LEARN analysis tools [15]. In particular, we used *Naïve Bayes* [14], *Decision Trees* [14], *AdaBoost* [6], and *Random Forest* [8] and validated our approach using k-fold cross-validation. We balance the training dataset to get homogeneous buckets containing 50,000 bug reports each, to prevent the different distribution of the sets to give a bias towards the biggest buckets [3]. In this context, a random classifier would correctly classify 0.25 of the instances, so a classifier is better than random if it achieves a higher proportion. Table VI shows the prediction results: Each column represent a classifier, while each row represents each layer of attributes we add to the model.

TABLE VI
PREDICTION RESULTS: PROPORTION OF BUG REPORTS CLASSIFIED IN THE CORRECT TIME BUCKET, WITH INCREMENT OVER RANDOM CLASSIFICATION (25% CORRECTLY CLASSIFIED BUG REPORTS).

	NB	DT	AdaBoost	RF
Common	0.27 (+0.02)	0.27 (+0.02)	0.27 (+0.02)	0.27 (+0.02)
+ words	0.28 (+0.03)	0.28 (+0.03)	0.29 (+0.04)	0.28 (+0.03)
+ tracker	0.36 (+0.11)	0.36 (+0.11)	0.42 (+0.17)	0.36 (+0.11)
+ project	0.37 (+0.12)	0.36 (+0.11)	0.42 (+0.17)	0.37 (+0.12)

In the first round we use the *common* attributes displayed in Figure 4; in the second, we add the number of words that compose the summary and the description of the report; in

the third, we add the *tracker* features, *i.e.,* the attributes that appear in some issue trackers; in the last, we add the *project* features, *i.e.,* the non standard attributes that are customized by the users of the platform. We follow this order to increasingly add the more and more specific fields and evaluate the impact that the different customizations have on the overall model. We can see from Table VI that the best results are achieved by AdaBoost [6] using the tracker-specific fields, with an overall accuracy of 0.42. Differently from the shared and tracker-specific fields, the project-fields may vary over time. They are, in fact, constantly added: Firefox, for example, adds a new custom field specific for each release, which happens once every 6 weeks. This mutability can raise the question whether the contribution of these fields is diluted in such a long timespan. To mitigate this effect, we recompute our experiments on the subset of bug reports collected in the timeframe that starts exactly one year before the dataset collection. The new dataset is composed of 31,472 bug reports. Table VII shows the results of our second batch of experiments.

TABLE VII
PREDICTION RESULTS FOR BUG REPORTS OF LAST YEAR.

	NB	DT	AdaBoost	RF
Common	0.27 (+0.02)	0.28 (+0.03)	0.28 (+0.03)	0.28 (+0.03)
+ words	0.30 (+0.05)	0.30 (+0.05)	0.33 (+0.08)	0.30 (+0.05)
+ tracker	0.34 (+0.09)	0.37 (+0.12)	0.46 (+0.21)	0.39 (+0.14)
+ project	0.35 (+0.10)	0.39 (+0.14)	0.46 (+0.21)	0.42 (+0.17)

Indeed, the results on the most recent dataset do not differ significantly from the results based on much longer timespans.

Conclusion. From our study using PCA, we observe that there exists a set of core elements of a bug report that impact and influence its lifetime.

Comparing these result with the perceived difficulty presented in RQ1, we see that some of these elements, like a description of the problem, the screenshot or the stack trace, compose the description field that we saw impacting the resolution time. Another relevant element is the assignee of the report, but users find it hard to provide it.

Interestingly, the elements that are the most useful in the resolution of a software defect are also harder to provide. From the experience with the various issue trackers and their interfaces, we believe that a user submitting a bug report should be offered a clean interface, that minimizes the amount of required information, highlights the most effective elements, and progressively requires the harder or less relevant ones.

The AdaBoost machine learning model achieves the best results, yet it can only predict the lifetime of a limited number of bug reports. The increment over a random classifier prediction is particularly small for the *common* attributes. This can be explained by the terse nature of the core model. Moreover, the *tracker* fields improve prediction, showing a relation between more detailed bug reports and bug lifetime.

After calculating the lifetime of each bug report in the tracker, we compare data from the two considered platforms. By analyzing the average lifetime of the bug reports in each platform we note that they have a longer lifespan on BUGZILLA than on JIRA, with an average fixing time of 239 and 166 days, respectively.

Even if the longer life of BUGZILLA projects may explain this phenomenon, we measure a gap between the lifetime of the reports in the two platforms (109 days for BUGZILLA and 93 days for JIRA), even when we restrict ourselves to consider reports submitted after 2009 (*i.e.,* when all the projects were active). There is an interesting, unexplained substantial difference in the way bug reports are processed in the two platforms. Studies can be designed and carried out to determine whether and how the bug reporting system itself leads to this behavior or there is a possibly unconscious self-selection of projects in using one or the other system.

Concerning project-specific attributes, from the results of the test, depicted in Table VI and Table VII, it emerges that they have the least weight in predicting the lifetime of a report. This suggests that they are not related to the fixing time. This may be a hint that these fields probably track collateral aspects of the evolution of a report that are not related to how quick a bug will be solved.

Last, we examine which fields impact the prediction the most: They are the number of words of the description and the summary, suggesting that an accurate description of the problem is important to engage the developers.

The fields that connect the issue with other reports are also relevant, for example the dependent issues, as well as the fact that a bug report is already assigned at the time of submission.

V. DISCUSSION

Threats to Validity. Dealing with large amounts of data can pose some problems in creating an abstraction sufficiently broad to comprise all the aspects of the data, but still specific enough to capture its details. We spent a considerable amount of time dealing with the representation of the data, extracting its features and cleaning the unneeded parts.

In particular, we carefully excluded from our prediction model all the fields that could yield a-posteriori information on the lifetime of a report. In such a large dataset it is still hard, however, to guarantee the complete soundness of the whole corpus, that could contain hidden relation between some attributes.

There is the concern that the lifetime of a bug report, that we used as a measure of quality of a bug report, is not relevant for our task. However, this metric proved to be an interesting open problem in the field and it represents an interesting heuristic in determining the effectiveness of a report.

Future Work. Developers tend to prefer simpler models to depict software defects. Even when provided with customization means, the additional information did not show a correlation with the lifetime of a report. Modern issue trackers like JIRA and BUGZILLA are complex interfaces over a set of tables in a relational database and the need for additional features over time makes those platforms grow over time, progressively turning them into inflexible *colossi*.

GitHub adopts the opposite approach, by providing a minimal structure of a bug report that is mostly a note attached to a

commit or a piece of code. This interesting approach, however, lacks the descriptive power of the other two platforms. The need for a simpler model is hinted by the choices of the development team of BUGZILLA that on version 5.0, released in July 2015, proposes a simplified interface that asks the user for a summary and a description of the problem, polished of all the additional information.

We believe that the future of issue tracking systems lies in flexible structures that can dynamically adapt to different aspects of the development activity.

VI. Related Work

Dealing with bug reports is a non-trivial task: Not only do users have to report meaningful information and developers have to understand and reproduce a problem, but they also have to deal with the large, noisy, and sometimes duplicated information stored in issue tracking systems. To minimize the impact that dealing with reports has on the bug fixing activity, researchers proposed different approaches to support developers and to automate important steps.

Reliability of a Bug Report. The first important aspect involving bug reports is the reliability and completeness of the information contained in a report. Through questionnaires, researchers collected information on how developers perceive the quality of a bug report and consider the most influential elements that help understanding a problem [24], [4], [16]. Researchers also proposed techniques to detect and avoid bug reports that do not contain useful information [19], thus alleviating the developers from information overload. Bissyandé *et al.* investigated the impact of the issue tracker on the development of a project [7], finding that most bug reporters are not developers of the project.

Automating Management of Bug Reports. Researchers proposed different approaches to automate bug report processing [21]. For example, a crucial aspect of managing bug reports is finding the ideal person to take care of an issue, known as *triaging*; Anvik *et al.* proposed a machine learning based approach to automate this step [2]. Guo *et al.* conducted a study to predict the aspects that impact the resolution time of *MS Windows* bug reports [12]. They found that a high number of reassignment of a report decreases the likelihood of the report of being closed quickly. They also found that the reputation of the submitter is an important factor to shorten the fixing time. Given the expensive nature of the bug fixing activity, Weiss *et al.* devised an approach to estimate the cost of a bug fix in person-hours [22]. Giger *et al.* studied the issue tracker of different open source project to predict bug fixing time, finding that the assignee and the reporting month are strong predictors [11]. Also, post-release information like the assignee is useful in increasing the accuracy. Bhattacharya and Neamtiu showed the low correlation of current prediction techniques and underlined the need to find additional attributes to increase the confidence of the time estimates [5].

Bug Reports and Social Interactions. An issue tracker represents also a social aspect of the community: users can interact with developers and provide feedback in fixing a defect. Breu *et al.* performed an analysis on a sample of 600 bug reports, finding that interacting with developers provides help in fixing the defect [9]. Zhou and Mockus showed that users involved in the development activity, like bug reporting and participating in the community, are more likely to become stable, long time contributors [23].

Bug Report Databases Visualization. Researchers proposed a number of visualizations to analyze feature of the bug reporting systems. For example, D'Ambros *et al.* performed an analysis of the BUGZILLA bug repository: They summarized the diagram of the state transitions of a report and proposed a set of visualizations to support the analysis of a bug database at different levels of granularity. Their approach allows the user to navigate the history of a single issue tracker and inspect selected part of the system with customized filters [10].

VII. Conclusion

We conducted an investigation to identify the features that are relevant to obtain a *satisficing* bug report. In doing so, we provided the following contributions:

1) An overview of the perceived difficulty of submitting elements of a bug report for users;
2) A meta-model for bug reports that represents both the common and specific elements available in reports of different issue trackers;
3) A publicly available dataset of more than 650,000 bug reports, modeled according to our meta-model;
4) An analysis of the contents of the issue trackers, to identify features that are related to reports' lifecycle;
5) Evidence that increasing the number of fields provided when submitting a bug report has little relation on shortening the lifetime of a bug.

Acknowledgements. We acknowledge the Swiss National Science foundation's support for project 146734 "HI-SEA" and Google's support (Google Faculty Research Award 2014).

References

[1] H. Abdi and L. J. Williams. Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):433–459, 2010.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[3] G. E. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM Sigkdd Explorations Newsletter*, 6(1):20–29, 2004.

[4] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann. Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 21–25. ACM, 2007.

[5] P. Bhattacharya and I. Neamtiu. Bug-fix time prediction models: can we do better? In *Proceedings of MSR 2011 (8th Working Conference on Mining Software Repositories)*, pages 207–210. ACM, 2011.

[6] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[7] T. F. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, and Y. Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *Proceedings of ISSRE 2013 (24th International Symposium on Software Reliability Engineering*, pages 188–197. IEEE, 2013.

[8] L. Breiman and E. Schapire. Random forests. In *Machine Learning*, pages 5–32, 2001.

[9] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 301–310. ACM, 2010.

[10] M. D'Ambros, M. Lanza, and M. Pinzger. "a bug's life" — visualizing a bug database. In *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 113–120. IEEE CS Press, 2007.

[11] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of RSSE 2010 (2nd International Workshop on Recommendation Systems for Software Engineering)*, RSSE '10, pages 52–56, New York, NY, USA, 2010. ACM.

[12] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 495–504. IEEE, 2010.

[13] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of ASE 2017 (22nd IEEE/ACM International Conference on Automated Software Engineering)*, ASE '07, pages 34–43. ACM, 2007.

[14] T. M. Mitchell. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45, 1997.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[16] A. Schroter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 118–121. IEEE, 2010.

[17] H. A. Simon. *Models of Man: Social and Rational*. John Wiley & Sons, 1957.

[18] H. A. Simon. *The Sciences of the Artificial*. MIT Press, 3rd edition, 2001.

[19] J. Sun. Why are bug reports invalid? In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 407–410. IEEE, 2011.

[20] F. Thung, T. Bissyande, D. Lo, and L. Jiang. Network structure of social coding in github. In *Proceedings of CSMR 2013 (17th IEEE European Conference on Software Maintenance and Reengineering*, pages 323–326, March 2013.

[21] W. Weimer. Patches as better bug reports. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 181–190. ACM, 2006.

[22] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1. IEEE Computer Society, 2007.

[23] M. Zhou and A. Mockus. Who will stay in the floss community? modeling participant's initial behavior. *Software Engineering, IEEE Transactions on*, 41(1):82–99, 2015.

[24] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *Software Engineering, IEEE Transactions on*, 36(5):618–643, Sept 2010.