Rocha *et al. Journal of Software Research and Development* (2015) 3:3
DOI 10.1186/s40411-015-0018-x

Journal of Software Engineering Research and Development
a SpringerOpen Journal

SOFTWARE Open Access

NextBug: a Bugzilla extension for recommending similar bugs

Henrique Rocha[1*], Guilherme de Oliveira[2], Humberto Marques-Neto[2] and Marco Tulio Valente[1]

*Correspondence:
henrique.rocha@dcc.ufmg.br
[1]Department of Computer Science, UFMG, 31.270-901 Belo Horizonte, Brazil
Full list of author information is available at the end of the article

Abstract

Background: Due to the characteristics of the maintenance process followed in open source systems, developers are usually overwhelmed with a great amount of bugs. For instance, in 2012, approximately 7,600 bugs/month were reported for Mozilla systems. Improving developers' productivity in this context is a challenging task. In this paper, we describe and evaluate the new version of NextBug, a tool for recommending similar bugs in open source systems. NextBug is implemented as a Bugzilla plug-in and it was design to help maintainers to select the next bug he/she would fix.

Results: We evaluated the new version of NextBug using a quantitative and a qualitative study. In the quantitative study, we applied our tool to 130,495 bugs reported for Mozilla products, and we consider as similar bugs that were handled by the same developer. The qualitative study reports the main results we received from a survey conducted with Mozilla developers and contributors. Most surveyed developers stated their interest in working with a tool like NextBug.

Conclusion: We achieved the following results in our evaluation: (i) NextBug was able to provide at least one recommendation to 65% of the bugs in the quantitative study, (ii) in 54% of the cases there was at least one recommendation among the top-3 that was later handled by the same developer; (iii) 85% of Mozilla developers stated that NextBug would be useful to the Mozilla community.

Keywords: Bugs; Recommendation systems; Bug mining techniques

Background

Software maintenance requests can be grouped and implemented as part of large software projects (Aziz et al. 2009; Marques-Neto et al. 2013; Junio et al. 2011; Tan and Mookerjee 2005). open source projects typically adopt continuous maintenance policies where the maintenance requests are addressed by maintainers with different skills and commitment levels, as soon as possible, after being registered in an issue tracking platform, such as Bugzilla and Jira (Tan and Mookerjee 2005; Mockus et al. 2002; Liu et al. 2012).

This process is usually uncoordinated, which results in a high number of issues from which many are invalid or duplicated (Liu et al. 2012). In 2005, a certified maintainer from the Mozilla Software Foundation made the following comment on this situation: "Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" (Anvik et al. 2006). The dataset with bugs reported for the Mozilla projects indicates that, in 2011, the number of issues reported per year

increased approximately 75% when compared to 2005. In this context, tools to assist in the issue processing would be very helpful and can contribute to increase the productivity of open source systems development.

Furthermore, software developers are often involved in situations of changes of context in a typical workday. These changes normally happen due to meetings, mails, instant messaging, etc., or when a given task is concluded and developers need to choose a new task to work on. Regardless the reasons, the negative effect of context changes on developers' productivity are well-known and studied. For example, in a recent survey with developers of the industry, more than 50% of the participants answered that a productive workday is one that flows without context-switches and having no or few interruptions (Meyer et al. 2014). Another study shows that developers spent at least two thirds of their time in activities related to task context, i.e., searching, navigating, and understanding the code relevant to the current task at hand (Ko et al. 2005). We argue that this time can be reduced if developers consistently decide to work on new tasks similar to the one previously concluded.

More specifically, context switches can be reduced by guiding developers to work on a set of bugs $B_0, B_1, \ldots, B_n$, where B_i requires changes on parts of the system related to a previous bug B_{i-1}, for $i > 0$. By following this workflow, context changes could be mitigated because the order of handled bugs naturally fosters a kind of periodic maintenance policy, i.e., a bug that is selected, comprehended, and changed at a given time helps other bug corrections in a near future.

In this paper, we claim that a simple form of periodic maintenance policy can be promoted in open source systems by recommending similar maintenance requests to maintainers whenever they manifest interest in handling a given request. Suppose that a developer has manifested interest in a bug with a textual description d_i. In this case, we rely on text mining techniques to retrieve open bugs with descriptions d_j similar to d_i and we recommend such bugs to this maintainer.

More specifically, we present *NextBug*, a tool to recommend similar bugs to maintainers based on the textual description of each bug stored in Bugzilla, an issue tracking system widely used by open source projects. The proposed tool is compatible with the software development process followed by open source systems for the following reasons: (a) it is based on recommendations and, therefore, maintainers are not required to accept extra bugs to fix; (b) it is a fully automatic and unsupervised approach which does not depend on human intervention; and (c) it relies on information readily available in Bugzilla. Assuming the recommendations effectively denote similar bugs and supposing that the maintainers would accept the recommendations pointed out by NextBug, the tool can contribute to introduce gains of scale in the maintenance of open source systems similar to the ones achieved with periodic policies (Banker and Slaughter 1997). We also report a quantitative and qualitative study focusing on the Mozilla ecosystem. In the quantitative study, we applied NextBug to a dataset of 130,495 bugs reported for Mozilla systems. In the qualitative study, we performed a survey asking Mozilla developers if they would use a tool to recommend similar bugs to work on.

We also present in this paper the following improvements over our first work on NextBug (Rocha et al. 2014): (a) new features introduced in NextBug version 0.9, including recommendation filters and logging files; (b) an extended section presenting the tool and its architecture in more detail; (c) more tools presented and analysed in the related

tools section; (d) a new quantitative study using the full dataset of 130,495 bugs (the previous conference paper used a subset of this dataset); (e) a new section describing the qualitative study.

The remainder of this paper is organized as follows. Section 2 discusses related tools, including tools for finding duplicated issue reports in bug tracking systems and also tools proposed to assign bugs to developers. The architecture, the central features, and an example of usage of NextBug are described in Section 2. We present the dataset and the evaluation of the tool in Section 2. Conclusions and future work are offered in Section 2. Finally, the availability and requirements for NextBug are presented in Section 2.

Related tools

Most open source systems adopt an Issue Tracking System (ITS) to support their maintenance process. Normally, in such systems both users and testers can report modification requests (Liu et al. 2012). This practice usually results in a continuous maintenance process where maintainers address the change requests as soon as possible. The ITS provides a central knowledge repository which also serves as a communication channel for geographically distributed developers and users (Anvik et al. 2006; Ihara et al. 2009).

Recent studies focused on finding duplicated issue reports in bug tracking systems. Duplicated reports can hamper the bug triaging process and may drain maintenance resources (Cavalcanti et al. 2013). Typically, tools for finding duplicated issues rely on traditional information retrieval techniques such as natural language processing, vector space model, and cosine similarity (Alipour et al. 2013; Sun et al. 2011; Wang et al. 2008). One of such approaches, called REP, analyzes both the textual information (using traditional techniques) along such with categorical information available in bug reports to improve the accuracy when finding duplicated reports (Sun et al. 2011).

When we compare techniques to find duplicated reports (Alipour et al. 2013; Sun et al. 2011; Wang et al. 2008) with NextBug, some differences emerge. First, duplicated bug techniques exclude bugs before they become available for developers to work on, i.e., duplicated techniques are applied in earlier stages of the maintenance process. On the other hand, NextBug is applied at later stages in the maintenance process, i.e., it shows bug recommendations when the developer is selecting bugs to work on. Second, NextBug only executes if the developer actively clicks on it. Therefore, NextBug causes no additional overhead unless the user wants to see its recommendations. The duplicate bug techniques do not specify whether their approach will be always executed and the overhead costs for such techniques. The third difference is the design objective; techniques to detect duplicated bugs need to be very precise to sort out duplicated bugs and not similar ones. NextBug aims to do the opposite and therefore needs to be precise to find similar bugs and not duplicated ones.

Tools to recommend the most suitable developer to handle a software issue are also reported in the literature (Anvik and Murphy 2011; Tamrawi et al. 2011; Kagdi et al. 2012). Most of them can be viewed as recommendation systems that suggest developers to handle a reported bug. For instance, Anvik and Murphy (2011) proposed an approach based on supervised machine learning that requires training to create a classifier. This classifier assigns the data (bug reports) to the closest developer.

Tools to recommend developers are usually applied at the same stages of the maintenance process as NextBug. However, these tools are more appropriate to organizations

where most of the maintenance work is assigned to developers by their manager. NextBug is design to help in open source projects, where most developers usually have the liberty to chose what to work on.

We found in the literature few studies dealing with similar issue reports. There is one approach that suggests the fixing effort (in person-hours) required to correct an issue by using similar issues already fixed as training data (Weiss et al. 2007). Another approach also tries to predict the fixing time of bugs but to classify them as fast or slow fixing (Giger et al. 2010). These approaches could also improve maintenance productivity like NextBug, particularly if developers decide to work on fast fixing bugs. However, unlike NextBug, they do not help to reduce context switches which may reduce their productivity gains. Another study that deals with similar issue reports relies on similarity to extract terms and locate source code methods that may cause defects based on polymorphism (Wang et al. 2010). This is a completely different application domain than NextBug, as this study tries to predict defects, while NextBug aims to help developers address bugs in a more productive manner.

BugMaps is a tool that extracts information about bugs from bug tracking systems, link this information to other software artifacts, and provides many interactive visualizations of bugs, which are called bug maps (Hora et al. 2012). BugMaps-Granges extends BugMaps with visualizations for causal analysis of bugs (Couto et al. 2014b; Couto et al. 2014a).

To conclude, to the best of our knowledge, no available tools are capable of recommending similar bugs to help maintainers of open source systems.

Implementation

In this section, we present NextBug version 0.9, an open source tool available under the Mozilla Public License (MPL). In Section 2, we describe the main features. We present the tool's architecture and its main components in Section 2. Finally, in Section 2, we present an example of NextBug recommendation.

Main features

Currently, there are several ITSs that are used in software maintenance such as Bugzilla, Jira, Mantis, and RedMine. NextBug is implemented as a Bugzilla plug-in because this ITS is used by the Mozilla project, which was used to validate our tool. However, NextBug could be extended and applied to other ITS.

When a developer is analysing or browsing an issue, NextBug can recommend similar bugs in the usual Bugzilla web interface. NextBug uses a textual similarity algorithm to verify the similarity among bug reports registered in Bugzilla. This algorithm is described in Section 2.

Figure 1 shows an usage example of our tool. This figure shows a real bug from the Mozilla project, which refers to a FirefoxOS application issue related to a mobile device camera (Bug 937928). As we can observe, Bugzilla shows detailed information about this bug, such as a summary description, creation date, product, component, operational system, and hardware information. NextBug extends this original interface by showing a list of bugs similar to the browsed one. This list is shown on the lower right corner. Another important feature is that NextBug is only executed if its Ajax link is clicked and, thus, it

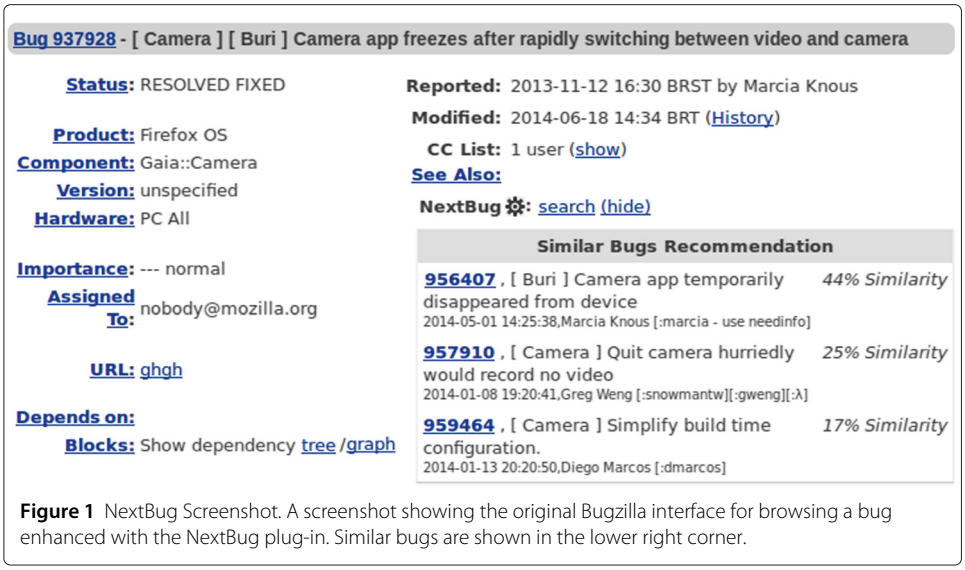

Figure 1 NextBug Screenshot. A screenshot showing the original Bugzilla interface for browsing a bug enhanced with the NextBug plug-in. Similar bugs are shown in the lower right corner.

does not cause additional overhead or hinder performance to developers who do not wish to follow similar bug recommendations.

In Figure 1, NextBug suggested three similar bugs to the one which is presented on the screenshot.

As we can note, NextBug not only detects similar bugs but it also reports an index to express this similarity.

By proposing NextBug, our final goal is to guide the developer's workflow by suggesting similar bugs to the one he/she is currently browsing. If a developer chooses to handle one of the recommended bugs, we claim he/she can minimize the context change inherent to the task of handling different bugs and, consequently, improve his/her productivity.

An important new feature of NextBug regarding its first released version (Rocha et al. 2014) is the support of filters to configure the provided recommendations. We implemented this feature after conducting a survey with Mozilla developers. Many developers said the previous version of our tool is indeed useful but we should provide a feature to customize the search results. The following comments show a sample feedback received from 66 Mozilla developers:

*"It would not be bad, as long as it could be configured to do the recommendations according to a set of parameters." – R.C.** (We show only the initials from survey subjects to preserve their anonymity).

"If it presented similar bugs that I did not file, and which were not assigned to anyone, that might be useful." – B.B.

"It would probably need to be able to at least check if a bug is already assigned to the current user (or to another user, which probably makes them ineligible, too)." – T.S.

The recommendations produced by NextBug follow the established criteria set up by the filters. By clicking on the gear icon next to the NextBug label, a dialog popup shows the configuration options for the filters (Figure 2). These filters include the following options: (a) similarity threshold, i.e, the minimum similarity index required for a recommendation to be considered valid, (b) maximum number of recommendations given by NextBug, (c) maximum and minimum bug severities considered valid for the recommendations, (d)

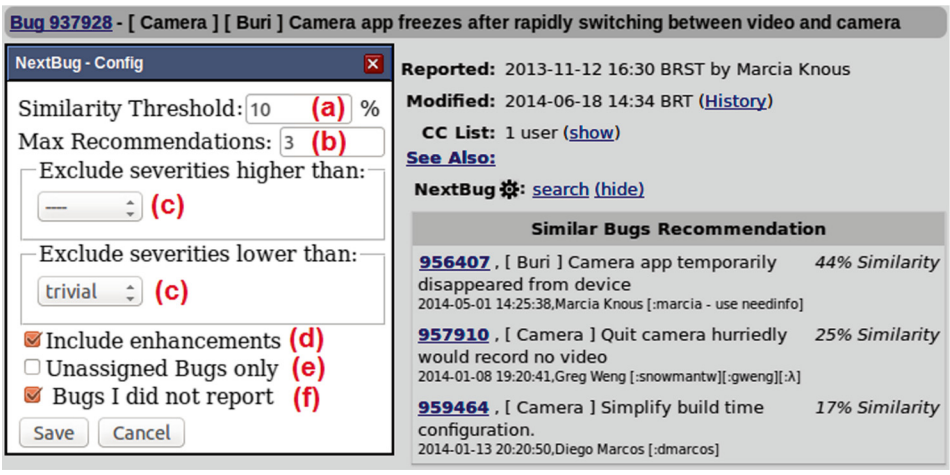

Figure 2 NextBug Config. The NextBug dialog showing the configuration options for filtering recommendations. These filter options are highlighted in **(a)** – **(f)**.

option to consider enhancements requests along with with bugs when presenting the recommendations for similar bugs, (e) searching only for unassigned bugs, i.e., recommend only bugs that no one is currently working on, and (f) shows only bugs that have not been reported by the current user; this filter might be useful because the current user is probably aware of the bugs he/she reported and he/she may not want them to be recommended back to him/her. Except for the first and second filters, all other filters are optional, e.g., the user is not required to check for unassigned bugs if he/she doesn't want to.

Moreover, the similarity threshold (i.e., the first filter) needs some experimentation from the user to fit his preferences. We suggest an initial value of 10%, which usually results in a good number of relevant recommendations. As the user increases this value, NextBug gives less recommendations but more precise ones as the tool becomes more strict. The decision is up to the user, who can decide if he/she wants fewer but more precise recommendations, or more recommendations but not as precise.

Figure 3 shows NextBug's architecture, including the system main components and the interaction among them. As described in Section 2, NextBug is a plug-in for Bugzilla.

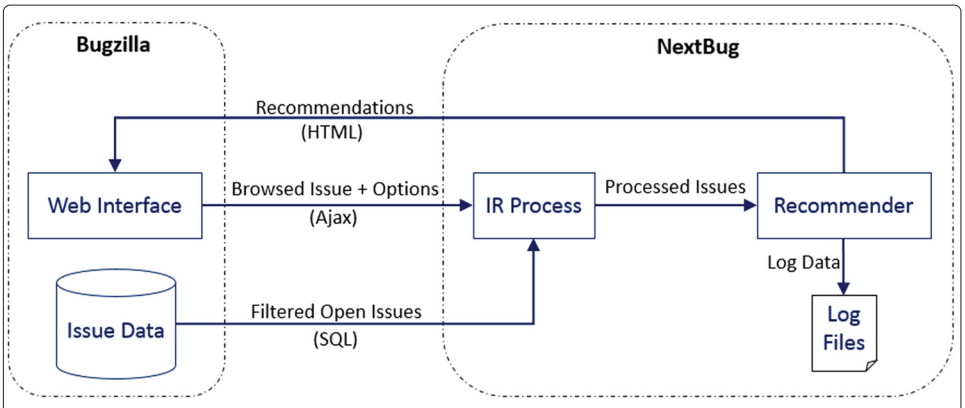

Figure 3 NextBug Architecture. Main NextBug components, their interaction with other components and with the Bugzilla system.

Therefore, it is implemented in Perl, the same language used in the implementation of Bugzilla. Basically, NextBug instruments the Bugzilla interface used for browsing and for selecting bugs reported for a system. NextBug registers an Ajax event in this interface that calls NextBug passing the browsed issue and the filter options as input parameters. We opt for an Ajax event because it is executed only if it is called by the developer and therefore it does not cause additional overhead.

Architecture and Algorithms

Algorithm 1 summarizes the processing of a NextBug Ajax event. It selects the filtered open issues that follows the criteria defined by the filter options (line 3), and then performs standard IR techniques on these open issues along with the browsed issue (lines 4-9). The processed issues are passed to the recommender system which selects the ones that are more relevant to the browsed issue (line 10). Finally, the produced recommendations are returned to the Bugzilla interface (line 14).

Algorithm 1 Recommendation Algorithm

```
1:  function NextBug-Event( BrowsedIssue, Options)
2:      StartTime = get-System-Time-Milisecs( );
3:      FilteredOpenIssues = get-Open-Issues( Options );
4:      q = IR-Processing( BrowsedIssue );
5:      D = ∅;
6:      for each issue d' ∈ FilteredOpenIssues do
7:          d_j = IR-Processing(d');
8:          D = D ∪ d_j ;
9:      end for
10:     Recommendations = recommender( q, D, Options );
11:     EndTime = get-System-Time-Milisecs( );
12:     ExecutionTime = EndTime − StartTime;
13:     log( q, Recommendations, ExecutionTime );
14:     return Recommendations ;
15: end function
```

Figure 3 shows that NextBug architecture has two central components: *Information Retrieval (IR) Process* and *Recommender*. We discuss these two components in the following subsections.

Information retrieval process component

The *IR Process* component obtains the filtered open issues currently available on the Bugzilla system along with the browsed bug. Then it relies on the following standard IR techniques for natural language processing: tokenization, stemming, and stop-words removal (Wang et al. 2008; Runeson et al. 2007). We implemented all such techniques in Perl. After this initial processing, the issues are transformed into vectors using the Vector Space Model (VSM) (Runeson et al. 2007; Baeza-Yates and Ribeiro-Neto 1999). VSM is a classical information retrieval model to process documents and to quantify their similarities. The usage of VSM is accomplished by decomposing the data (available bug reports and queries) into t-dimensional vectors, assigning weights to each indexed term. The

weights w_i are positive real numbers that represent the i-th index term in the vector. To calculate w_i we used the following equation, which is called a *tf-idf* weighted formula:

$$w_i = (1 + \log_2 f_i) \times \log_2 \frac{N}{n_i}$$

where f_i is the frequency of the i-th term in the document, N is the total number of documents, and n_i is the number of documents in which the i-th term occurs.

Recommender component

The *Recommender* component receives the processed issues and verifies the ones similar to the browsed one. The similarity is computed using the cosine similarity measure (Baeza-Yates and Ribeiro-Neto 1999; Runeson et al. 2007). More specifically, the similarity between the vectors of a document d_j and a query q is described by the following equation, which is called the cosine similarity because it measures the cosine of the angle between the two vectors:

$$Sim(d_j, q) = \cos(\Theta) = \frac{\overrightarrow{d_j} \bullet \overrightarrow{q}}{||\overrightarrow{d_j}|| \times ||\overrightarrow{q}||} = \frac{\sum_{i=1}^{t} w_{i,d} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t}(w_{i,d})^2} \times \sqrt{\sum_{i=1}^{t}(w_{i,q})^2}}$$

Since all the weights are greater or equal to zero, we have $0 \leq Sim(d_j, q) \leq 1$, where zero indicates that there is no relation between the two vectors, and one indicates the highest possible similarity, i.e., both vectors are actually the same.

The issues are then ordered according to their similarity before being returned to Bugzilla. Since NextBug executes as an Ajax event, the recommendations are showed in the same Bugzilla interface used by developers when browsing and selecting bugs to fix.

Logging files are also updated with anonymous information about the process before the recommendations return to Bugzilla interface. The anonymity is important to preserve the user's privacy and to prevent NextBug being perceived as a spyware program. The logging files register execution time data, the browsed issue, and the recommendations given. Therefore, it is possible to monitor how long it takes for NextBug event to execute. The anonymous data collected can also be used in future studies to support further analysis and evaluation on NextBug.

Example of recommendation

Table 1 presents an example of a bug opened for the component `DOM:Device Interfaces` of the `Core` Mozilla product and the first three recommendations (top-3) suggested by NextBug for this bug. As we can observe in the summary description, both the query and the recommendations require maintenance in the `Device Storage` API,

Table 1 Example of Recommendation

	Sim	Bug ID	Summary	Creation	Fixed
Bug	–	788588	Device Storage - Default location for device storage on windows should be NS_WIN_PERSONAL_DIR	2012-09-05	2012-09-06
Top-1	56%	754350	Device Storage - Clean up error strings	2012-05-11	2012-10-17
Top-2	47%	788268	Device Storage - Convert tests to use public types	2012-09-04	2012-09-06
Top-3	42%	786922	Device Storage - use a properties file instead of the mime service	2012-08-29	2012-09-06

used by Web apps to access local file systems. Moreover, all four issues were handled by the same developer (Dev ID 302291).

We can also observe that the three recommended issues were created before the original query. In fact, the developer fixed the bugs associated to the second and the third recommendations in the same date on which he/she fixed the original query, i.e., on 2012-09-06. However, he/she only resolved the first recommended bug (ID 754350) 41 days latter, i.e., on 2012-10-17. Therefore, NextBug could have helped this maintainer to discover quickly the related issues, which probably demanded more effort without a recommendation tool.

Results and discussion

In this section, we initially describe the dataset used to evaluate NextBug in quantitative terms (Section 2). Section 2 presents the main results from this quantitative study. Finally, Section 2 shows the main findings of a qualitative study with Mozilla developers.

Dataset

We used a dataset with bugs from the Mozilla project to evaluate NextBug. Mozilla is composed of 69 products from different domains which are implemented in different programming languages. The Mozilla project includes some popular systems such as Firefox, Thunderbird, SeaMonkey, and Bugzilla. We considered only issues that were actually fixed from January 2009 to October 2012, in a total of 130,495 issues. More specifically, we ignored issue types such as "duplicated", "incomplete", and "invalid". Figure 4 shows the monthly number of issues fixed in this time frame.

Mozilla issues are also classified according to their severity in the following scale: *blocker*, *critical*, *major*, *normal*, *minor*, and *trivial*. Table 2 shows the number and the percentage of each of these categories in our dataset. This scale also includes enhancements as a particular severity category.

Table 2 also shows the number of days required to fix the issues in each category. We can observe that *blocker* bugs are quickly corrected by developers, showing the lowest values for maximum, average, standard deviation, and median measures among the considered categories. The presented lifetimes also indicate that issues with *critical* and *major* severity are closer to each other. Finally, *enhancements* are very different from the others, showing the highest values for average, standard deviation, and median.

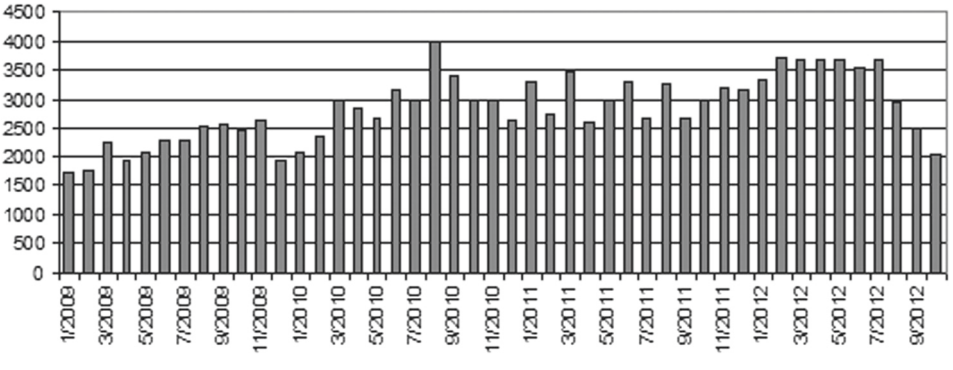

Figure 4 Fixed issues per month. Bar chart showing fixed issues per month for Mozilla systems between jan 2009 to oct 2012.

Table 2 Issues per severity

Severity	Issues		Days to resolve				
	Number	%	Min	Max	Avg	Dev	Med
blocker	2,720	2.08	0	814	15.44	52.25	1
critical	7,513	5.76	0	1258	37.87	99.52	6
major	7,508	5.75	0	1275	41.59	109.83	5
normal	103,385	79.23	0	1373	46.27	108.84	8
minor	3,660	2.80	0	1355	77.05	161.72	11
trivial	2,109	1.62	0	1288	80.84	164.74	11
enhancement	3,600	2.76	0	1285	126.14	195.25	40
Total	130,495	100	–	–	–	–	–

Quantitative study

For the quantitative part of our study, we simulated the use of NextBug for each bug in the dataset, i.e., all bugs were processed as if they were the *browsed issue* selected by a developer and processed by Algorithm 1 (described in Section 2). We considered bugs that were open when the *browsed issue* was created for the *filtered open issues* used by the Algorithm 1. For this study, we configured NextBug with a similarity threshold of 10%.

In this evaluation, we defined a relevant recommendation as one that shares the same developer with the browsed issue. More specifically, we consider that a recently created issue is related to a second opened issue when they are handled by the same developer. The assumption in this case is that NextBug fosters gains of productivity whenever it recommends issues that are later fixed anyway by the same developer.

We used three metrics in our evaluation: Feedback, Precision, and Likelihood. These metrics are inspired by the evaluation followed by the ROSE recommendation system (Zimmermann et al. 2004). Although, ROSE targets a different context, their metrics are appropriate to evaluate any kind of recommendation system.

Before we present the equations to calculate the metrics, we must first define the following sets:

- A_q**:** the set of recommendations provided by NextBug.
- $A_q(k)$**:** top-k issues in A_q ordered by textual similarity (only defined for $|A_q| \geq k$).
- R_q**:** the set of open issues when the query (or browsed issue) is processed, ordered by textual similarity. These open issues must also share the same developer as the query. This is the formal definition of relevant recommendations used in our evaluation.
- Z**:** the set of all processed queries. For our evaluation, the Z set is composed by 130,495 queries (i.e., the same size as our dataset).
- Z_k**:** a subset of Z with the queries that returned at least k recommendations.

Feedback: measures the number of recommendations provided to a given query. Formally, the feedback $Fb(k)$ is the percentage of queries with at least k recommendations, as follows:

$$Fb(k) = \frac{|Z_k|}{|Z|}$$

For example, suppose a recommendation system that executed 100 queries ($|Z| = 100$). If all those queries returned at least 1 recommendation each, then Feedback for $k = 1$,

i.e., Fb(1) would be 100%. On the other hand, if only 40 of those queries returned at lest 3 recommendations, then $Fb(3) = 40\%$.

Precision: measures the ratio of relevant recommendations. More specifically, we define the precision of the first k recommendations provided by NextBug as follows:

$$P_q(k) = \frac{|A_q(k) \cap R_q|}{|A_q(k)|}$$

Moreover, the overall precision in our dataset is defined as the average of the precisions achieved for each query, as follows:

$$P(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} P_q(k)$$

Suppose that a query returned 4 recommendations. If the first recommendation is a relevant one, then $P_q(1) = 100\%$. Otherwise, if the first recommendation is not relevant, then $P_q(1) = 0\%$. Moreover, if among all 4 recommendations only the second one is relevant, then the precision values would be $P_q(2) = 50\%$, $P_q(3) = 33\%$, and $P_q(4) = 25\%$.

Likelihood: is a common measure to assess the usefulness of recommendations. In our particular approach, it checks whether there is a relevant recommendation among the top-k suggested issues. Formally, we define the likelihood of the top-k recommendations provided by our approach as follows:

$$L_q(k) = \begin{cases} 1 & \text{if } A_q(k) \cap R_q \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Therefore, $L_q(k)$ is a binary measure. If there is at least a relevant recommendation among the top-k recommendations, it returns one; otherwise, it returns zero.

The overall likelihood in our dataset is defined as the average of the likelihood measured for each query, as follows:

$$L(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} L_q(k)$$

Suppose that a query returned 4 recommendations but only the third one is relevant. Then the likelihood values would be $L_q(1) = 0\%$, $L_q(2) = 0\%$, $L_q(3) = 100\%$, and $L_q(4) = 100\%$.

Summary: Feedback is the ratio of queries where NextBug makes at least k recommendations. Precision is the ratio of relevant recommendations among the top-k recommendations provided by NextBug. Finally, Likelihood indicates whether at least one relevant recommendation is included among NextBug's top-k suggestions.

Results: Figure 5 shows the average feedback (left chart), precision (central chart) and likelihood (right chart) up to $k = 5$. The results presented in this figure are summarized as follows:

- NextBug achieves a feedback of 0.65 for $k = 1$. Therefore, on average, NextBug makes at least one suggestion for 65% of the bugs, i.e., for every five bugs NextBug provides at least one similar recommendation for three of them. Moreover, NextBug shows on average 3.2 recommendations per query.
- NextBug achieves a precision of at least 0.31 for all values of k. In other words, 31% of NextBug recommendations are on average relevant (i.e., further handled by the same developer), no matter how many suggestions are given.

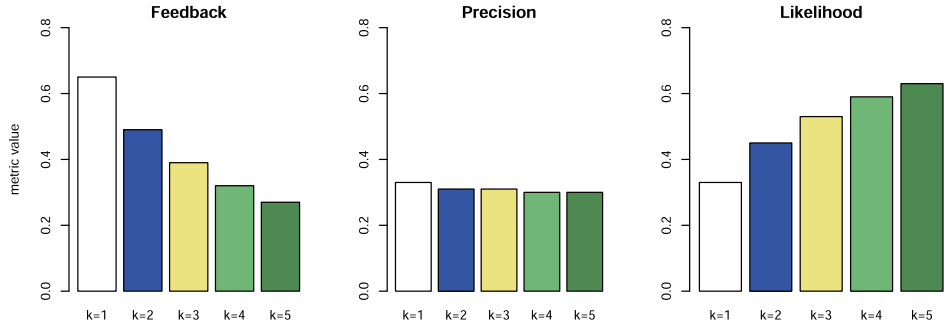

Figure 5 Evaluation Results. Average results for Feedback, Precision, and Likelihood considering $k = 1$ to $k = 5$.

- NextBug achieves a likelihood of 0.54 for $k = 3$. Therefore, in about 54% of the cases, there is a top-3 recommendation that is later handled by the same developer responsible for the original bug.

Since we are not aware of other next bug recommendation tool, we compared our results with a recommender system targeting other software engineering tasks. For example, ROSE (Zimmermann et al. 2004) achieved the following results for recommending co-change relations in the Eclipse system: feedback 64% ($k = 1$), precision 30% ($k = 1$), and likelihood 57% ($k = 3$). Therefore, we believe our results are good enough to encourage developers to adopt NextBug in their maintenance tasks.

Qualitative study

For the qualitative part of our study, we conducted a survey with Mozilla developers. During one week, we analysed bugs that were fixed for Mozilla systems on their official Bugzilla website (<https://bugzilla.mozilla.org/>, verified 2014-11-01). For each fixed bug, we used NextBug to give recommendations based on the bugs that were still open at each day.

We sent by e-mail the top-3 recommendations suggested by NextBug to Mozilla maintainers with two questions. The first question asked whether the recommendations given were relevant, i.e., if the recommendations were similar bugs that could be fixed next. The second question asked if they thought a tool like NextBug to recommend similar bugs would be useful for developers from their community. A total of 66 maintainers (from 176) answered our survey. The following comments show a sample of the feedback we received from those developers:

"(...) this would help contributors to discover and be active on the code." – N.P.

"(...) anything that would make parsing and organising bugs easier would be amazing – they're currently very hard for some of the staff to keep track of!" – K.B.

Our results in this survey are summarized as follows: (i) 77% of the developers found our recommendations relevant, i.e., they answered yes for our first question; (ii) 85% of the developers confirmed that a tool to recommend similar bugs would be useful to the Mozilla community, i.e., they answered yes for the second question.

Conclusion

This paper presented NextBug version 0.9, a tool for recommending similar bugs. NextBug is implemented in Perl as a plug-in for Bugzilla, a widely used Issue Tracking

Systems (ITS), specially popular in open source systems. NextBug relies on information retrieval techniques to extract semantic information from issue reports in order to identify the similarity of open bugs with the one that is being handled by a developer.

We evaluate NextBug with a dataset of 130,495 Mozilla bugs, achieving feedback results of 65%, precision results around 31%, and likelihood results greater than 54% (considering top-3 recommendations for likelihood). We also conducted a survey with Mozilla developers using recommendations provided by NextBug. From such developers, 77% of them ranked our recommendations as relevant and 85% confirmed that NextBug would be useful to the Mozilla community.

Future work

First, we plan to make NextBug an official Bugzilla extension. We believe that once NextBug becomes an official plug-in, real open source projects will install the tool in their Issue Tracking System (ITS). Second, we want to acquire log data from real working projects using our tool. We plan to use this data to better analyse the tool's usage by developers, and thus, propose future improvements for NextBug. Third, we want to apply our tool algorithm in a slightly different context, creating another tool called PreviousBug. Instead of looking for similar open issues, PreviousBug would look for similar resolved bugs. Thus, PreviousBug would help developers implement a given task by looking at similar tasks already resolved in the past. PreviousBug would require an integration with Version Control System to retrieve the parts of code related to resolved tasks.

Availability and requirements

Availability information and requirements are as follows:

- **Project name:** NextBug
- **Project home page:** <http://aserg.labsoft.dcc.ufmg.br/nextbug/>
- **Operating system(s):** NextBug is a plugin for Bugzilla, and as such it executes in the same operating systems as Bugzilla.
- **Programming language:** Perl
- **License:** Mozilla Public License.
- **Other requirements:** Bugzilla 3.6 or higher.
- **Any restrictions to use by non-academics:** no additional restrictions as described in the license.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

HR and GO worked on the implementation of the tool. HR and MTV worked on the conception and design of the tool. HR, HMN, and MTV wrote and revised this manuscript. All authors read and approved the final manuscript.

Acknowledgements

This work is supported by CNPq, CAPES, and FAPEMIG.

Author details

[1] Department of Computer Science, UFMG, 31.270-901 Belo Horizonte, Brazil. [2] Department of Computer Science, PUC Minas, 30.535-901 Belo Horizonte, Brazil.

Received: 8 December 2014 Accepted: 2 April 2015
Published online: 17 April 2015

References

Alipour A, Hindle A, Stroulia E (2013) A contextual approach towards more accurate duplicate bug report detection. In: 10th Working Conference on Mining Software Repositories (MSR). IEEE Press, Piscataway, NJ, USA. p 183

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug?. In: 28th International Conference on Software Engineering (ICSE). ACM, New York, NY, USA. pp 361–370

Anvik J, Murphy GC (2011) Reducing the effort of bug report triage: recommenders for development-oriented decisions. ACM Trans Softw Eng Methodol (TOSEM) 20(3):10–11035

Aziz J, Ahmed F, Laghari MS (2009) Empirical analysis of team and application size on software maintenance and support activities. In: 1st International Conference on Information Management and Engineering (ICIME). IEEE Computer Society, Washington, DC, USA. pp 47–51

Banker RD, Slaughter SA (1997) A field study of scale economies in software maintenance. Manage Sci 43:1709–1725

Baeza-Yates RA, Ribeiro-Neto B (1999) Modern Information Retrieval. 2nd edn. Addison-Wesley, Boston, MA, USA

Cavalcanti YC, Mota Silveira Neto PA, Lucredio D, Vale T, Almeida ES, Lemos Meira SR (2013) The bug report duplication problem: an exploratory study. Softw Qual J 21(1):39–66

Couto C, Pires P, Valente MT, Bigonha R, Anquetil N (2014a) Predicting software defects with causality tests. J Syst Softw 93:24–41

Couto C, Valente MT, Pires P, Hora A, Anquetil N, Bigonha R (2014b) Bugmaps-granger: A tool for visualizing and predicting bugs using granger causality tests. J Softw Eng Res Dev 1(2):1–12

Giger E, Pinzger M, Gall H (2010) Predicting the fix time of bugs. In: 2nd International Workshop on Recommendation Systems for Software Engineering (RSSE). ACM, New York, NY, USA. pp 52–56

Hora A, Couto C, Anquetil N, Ducasse S, Bhatti M, Valente MT, Martins J (2012) BugMaps: A tool for the visual exploration and analysis of bugs. In: 16th European Conference on Software Maintenance and Reengineering (CSMR), Tool Demonstration Track. IEEE Computer Society, Washington, DC, USA. pp 523–526

Ihara A, Ohira M, Matsumoto K (2009) An analysis method for improving a bug modification process in open source software development. In: 7th International Workshop Principles of Software Evolution and Software Evolution (IWPSE-Evol). ACM, New York, NY, USA. pp 135–144

Junio GA, Malta MN, de Almeida Mossri H, Marques-Neto HT, Valente MT (2011) On the benefits of planning and grouping software maintenance requests. In: 15th European Conference on Software Maintenance and Reengineering (CSMR). IEEE Computer Society, Washington, DC, USA. pp 55–64

Kagdi H, Gethers M, Poshyvanyk D, Hammad M (2012) Assigning change requests to software developers. J Softw: Evol Process 24(1):3–33

Ko AJ, Aung H, Myers BA (2005) Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In: 27th International Conference on Software Engineering (ICSE). ACM, New York, NY, USA. pp 126–135

Liu K, Tan HBK, Chandramohan M (2012) Has this bug been reported?. In: 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE). pp 28–1284

Marques-Neto H, Aparecido GJ, Valente MT (2013) A quantitative approach for evaluating software maintenance services. In: 28th ACM Symposium on Applied Computing (SAC). pp 1068–1073

Meyer AN, Fritz T, Murphy GC, Zimmermann T (2014) Software developers' perceptions of productivity. In: 22th International Symposium on Foundations of Software Engineering (FSE). pp 26–36

Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: Apache and Mozilla. ACM Trans Softw Eng Methodol 11(3):309–346

Rocha H, Oliveira G, Maques-Neto H, Valente MT (2014) NextBug: A Tool for Recommending Similar Bugs in Open-Source Systems. In: V Brazilian Conference on Software: Theory and Practice – Tools Track (CBSoft Tools). SBC, Maceio, AL, Brazil, Vol. 2. pp 53–60

Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: 29th International Conference on Software Engineering (ICSE). IEEE Computer Society, Washington, DC, USA. pp 499–510

Sun C, Lo D, Khoo S-C, Jiang J (2011) Towards more accurate retrieval of duplicate bug reports. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE Computer Society. pp 253–262

Tamrawi A, Nguyen TT, Al-Kofahi JM, Nguyen TN (2011) Fuzzy set and cache-based approach for bug triaging. In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE). ACM, New York, NY, USA. pp 365–375

Tan Y, Mookerjee VS (2005) Comparing uniform and flexible policies for software maintenance and replacement. IEEE Trans Softw Eng 31(3):238–255

Wang X, Zhang L, Xie T, Anvik J, Sun J (2008) An approach to detecting duplicate bug reports using natural language and execution information. In: 30th International Conference on Software Engineering (ICSE). ACM, New York, NY, USA. pp 461–470

Wang D, Lin M, Zhang H, Hu H (2010) Detect related bugs from source code using bug information. In: 34th IEEE Computer Software and Applications Conference (COMPSAC). IEEE Computer Society, Washington, DC, USA. pp 228–237

Weiss C, Premraj R, Zimmermann T, Zeller A (2007) How long will it take to fix this bug? In: 4th International Workshop on Mining Software Repositories (MSR). IEEE Computer Society, Washington, DC, USA

Zimmermann T, Weisgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: 26th International Conference on Software Engineering (ICSE). IEEE Computer Society, Washington, DC, USA. pp 563–572