An Empirical Study of Long Lived Bugs

Ripon K. Saha Sarfraz Khurshid Dewayne E. Perry
Department of Electrical and Computer Engineering
The University of Texas at Austin, USA
ripon@utexas.edu, khurshid@ece.utexas.edu, perry@mail.utexas.edu

***Abstract*—Bug fixing is a crucial part of software development and maintenance. A large number of bugs often indicate poor software quality since buggy behavior not only causes failures that may be costly but also has a detrimental effect on the user's overall experience with the software product. The impact of *long lived* bugs can be even more critical since experiencing the same bug version after version can be particularly frustrating for user. While there are many studies that investigate factors affecting bug fixing time for entire bug repositories, to the best of our knowledge, none of these studies investigates the extent and reasons of long lived bugs. In this paper, we analyzed long lived bugs from five different perspectives: their proportion, severity, assignment, reasons, as well as the nature of fixes. Our study on four open-source projects shows that there are a considerable number of long lived bugs in each system and over 90% of them adversely affect the user's experience. The reasons of these long lived bugs are diverse including long assignment time, not understanding their importance in advance etc. However, many bug-fixes were delayed without any specific reasons. Our analysis of bug fixing changes further shows that many long lived bugs can be fixed quickly through careful prioritization. We believe our results will help both developers and researchers to better understand factors behind delays, improve the overall bug fixing process, and investigate analytical approaches for prioritizing bugs based on bug severity as well as expected bug fixing effort.**

***Index Terms*—Bug tracking system, bug triaging, bug survival time**

I. Introduction

Software development and maintenance is a complex process. Although developers and testers try their best to make their software error free, in practice software ships with bugs. The number of bugs in software is a significant indicator of software quality since bugs can adversely affect users experience directly. Therefore, developers are generally very active in finding and removing bugs.

To ensure high software quality for each release, developers/managers triage bugs carefully and schedule the bug fixing tasks based on their severity and priority. Despite such a rigorous process, there are still many bugs that live for a long time. We believe the impact of these *long lived* bugs (for our study, bugs that are not fixed within one year after they are reported) is even more critical since the users may experience the same failures version after version. Therefore, it is important to understand the extent and reasons of these long lived bugs to improve software quality.

A number of previous studies have investigated the overall factors affecting bug fix time. Giger et al. [7] empirically investigated the relationships between bug report attributes and the time to fix. Zhang et al. [23] predicted overall bug fix time in commercial projects. Canfora et al. [6] used survival analysis to determine the relationship between the risk of not fixing a bug within a given time frame and specific code constructs changed when fixing the bug. Zhang et al. [22] examined factors affecting bug fixing time along three dimensions: bug reports, source code involved in the fix, and code changes that are required to fix the bug.

While these studies are useful in understanding the overall factors related to bug fix time, we know of no study that has specifically investigated long lived bugs to understand why they take such a long time to be fixed and how important they are. We point out that analyzing entire bug datasets using various machine learning or data mining techniques (as done in previous work) is not sufficient in understanding long lived bugs due to the imbalanced dataset, i.e., containing relatively low proportion of long lived bugs compared to others. Imbalanced dataset is a major problem in most data mining applications since machine learning algorithms can be biased towards the majority class due to over-prevalence [8]. Therefore, if we automatically analyze all the bug reports using a standard data mining technique, it is highly likely that the main factors behind long lived bugs would get lost. In this paper, we conduct an exploratory study focused solely on long lived bugs to understand their extent and reasons with respect to following research questions:

1) **What proportion of the bugs are long lived?** The answer to this question is important since if there are few long lived bugs, there may be little reason to worry.
2) **How important long lived bugs are in terms of severity?** It is important to understand how crucial these bugs were from the perspective of both developers and users. If they are minor or trivial bugs, their impact would be less on overall software quality.
3) **Where was most of the time spent in the bug fixing process?** The answer to this question is important to identify the time consuming phases so that developers as well as researchers can work on improving the process involving that phase.
4) **What are common reasons for long lived bugs?** To improve the bug fixing process, first we need to understand the underlying reasons for delay. Delineating the common reasons of long lived bugs will help researchers deal with the problem more systematically.
5) **What is the nature of long lived bug fixes?** The answer to this question will help us in better understanding the bug fixing process, estimating change efforts, and so on, which will be useful in exploring potential approaches for improving overall bug fixing process.

We study four open source projects namely JDT, CDT, PDE, and Platform from the Eclipse product family, [1] and make the following key observations:

1) Despite advances in software development and maintenance processes, there are a significant number of bugs in each project that survive for more than one year.
2) More than 90% of long lived bugs affect users' normal working experiences and thus are important to fix. Moreover, there are several duplicate bug reports for these long lived bugs, which indicates the users' demand for fixing them.
3) The average bug assignment time of these bugs was more than one year despite the availability of a number of automatic bug assignment tools that could have been used. The bug fix time after the assignment was another year on average.
4) Reasons for long lived bugs are diverse. While problem complexity, reproducibility, and not understanding the importance of some of the bugs in advance are the common reasons, we observed there are many bug-fixes that got delayed without any specific reason.
5) Unlike previous studies [22], we found that a bug surviving for a year or more does not necessarily mean that it requires a large fix. We found that 40% of long-lived bug fixes involved few changes in only one file.

We believe these findings will play an important role in developing new approaches for bug triaging as well as improving the overall bug fixing process.

II. Background

A. Bug Tracking System:

Generally project stakeholders maintain a bug database for tracking all the bugs associated with their projects. There are several online bug tracking systems such as Bugzilla, JIRA, Mantis etc. These systems enable developers/managers to manage bug database for their projects. Different repositories may have different data structures and follow different life cycles of bugs. The dataset used in our work was extracted from Bugzilla, a popular online bug tracking system. Therefore, the rest of the discussion in this paper regarding the bug tracking system is only limited to Bugzilla.

Any person having legitimate access to a project's bug database can post a change request through Bugzilla. A change request could be either a bug or an enhancement. In Bugzilla, however, both bugs and enhancements are represented similarly and referred as bugs with an exception that for enhancements severity field is set to enhancement. Generally bug reporters provide a bug summary, bug description, the suspected product, and the component name with its severity.

According to Eclipse Bugzilla documentation, the severity level can be one of the following values, which actually represents the degree of potential harm.[2]

`Blocker`: These bugs block the development and/or testing work. There exists no workaround.

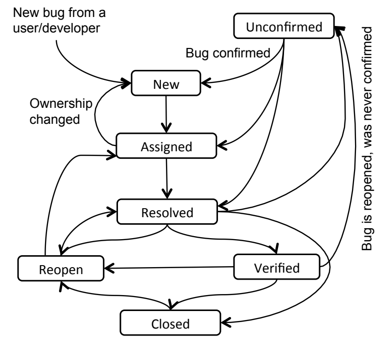

Fig. 1. Life Cycle of a Bug in Bugzilla

`Critical`: These bugs cause program crashes, loss of data, or severe memory leaks.

`Major`: These bugs result major loss of function.

`Normal`: These are regular issues. There are some loss of functionality under specific circumstances.

`Minor`: These bugs cause minor loss of functionality, or other problems where an easy workaround was present.

`Trivial`: These are generally cosmetic problems such as misspelled words or misaligned text.

Reporters also specify the software version, the platform and operating system where they encountered the bug so that developers can easily reproduce it. Bug reporters also can attach files to the bug report such as screen shots, failing test cases etc. Once a bug is posted, all other related developers can make comments regarding the bug to discuss different issues. Therefore, a bug repository has rich set of information that can be analyzed to gain insight about bugs.

B. Bug Life Cycle

The overall bug fixing process in a system is directly related to the bug life cycle maintained by the bug tracking system. Although Eclipse projects have different schemes for using Bugzilla, a common life cycle for a bug is as follows: [3]

Validation: At the start of each day, each project/component team leader triages `NEW` bugs to verify if the bug is really a bug and if the provided information is correct. In case of any inconsistencies, the bug triager can correct them. The bug triager also can request further information to validate a bug if it is necessary. If there is no response within a week, the team leader closes the bug marking `RESOLVED`, `INVALID`, or `WONTFIX`. However, the reporter can reopen the bug anytime if she has more information.

Prioritization In this stage, the triager first determines whether a bug is a feature request. If so, the severity of the bug is changed to `enhancement`. Otherwise, she checks the severity level of the bug to make sure that it is consistent with the bug description. Then the priority of the bug is set based on following guidelines: [4]

`P1`: These bugs are a must fix for the indicated target milestone.

`P2`: These bugs are very important for the indicated target milestone. Generally developers try to resolve all the `P2` bugs.

[1]http://www.eclipse.org

[2]http://wiki.eclipse.org/Eclipse/Bug_Tracking

[3]http://wiki.eclipse.org/Development_Resources/HOWTO/Bugzilla_Use

[4]http://wiki.eclipse.org/WTP/Conventions_of_bug_priority_and_severity

TABLE I
DATA SET

System	Change Requests	# Bugs	# Enhancements	# Bug Fixed
JDT	46,308	38,520	7,788	18,873
CDT	14,871	12,854	20,17	7,260
PDE	13677	11,958	1,719	6,854
Platform	90,691	78,120	12,571	33,738
Total	165,547	141,452	24,095	66,725

`P3`: It is the default priority. If the bug triager is uncertain about the priority of a bug or it is actually a normal bug, she can set `P3` priority. Then the assigned developer can adjust it if appropriate.

`P4`: These bugs should be fixed if time permits.

`P5`: These are valid bugs, but there are no plans to fix. Also `P5` priority indicates that help is wanted.

Fixing: At this point, a bug remains in the component's "inbox" account until a developer takes the bug, or the team leader assigns it to them. After fixing the bug, the developer mark it as `RESOLVED-FIXED`.

Verification: Once a bug is fixed, it is assigned to another committer on the team to verify. Ideally, all bugs should be verified before the next integration build. Once the verifier tests that the bug is completely resolved, she changes the bug status to `VERIFIED`. Figure 1 represents all possible state transitions of a bug in Bugzilla.

III. STUDY SETUP

A. Subject Systems

We choose four open source projects: JDT, CDT, PDE, and Platform from the Eclipse product family for our study. There are mainly two reasons for choosing these projects. First, Eclipse projects are highly successful and have been widely used in software engineering research. Second, although these projects belong to the same product family, they are from different domains. The Eclipse Platform defines the set of frameworks and common services that collectively make up infrastructure required to support the use of Eclipse. The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products. On the other hand, JDT and CDT provide a fully functional Integrated Development Environment based on the Eclipse platform for developing Java, and C and C++ applications. We have used Lamkanfi et al's [11] bug dataset to extract the bug information associated with these projects. This dataset includes all the bug reports and their histories from their inception to March 2011 for these four projects (extracted from Eclipse Bugzilla database). [5] A more detailed description of the dataset is presented in Table I.

B. Terms and Metrics

We make use of bug tracking and version control system's information to calculate metrics that we were interested in. This section defines different terms and metrics that we use in the rest of the paper.

Bug Introduction Time (T_I): This is the timestamp when the buggy code is committed for the first time for a given bug.

[5] https://bugs.eclipse.org/bugs/

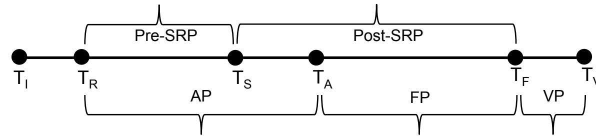

Fig. 2. An Example Timeline of a Bug

Bug Reporting Time (T_R): This is the timestamp when a bug is reported to the Bugzilla system by a user/developer.

Bug Assignment Time (T_A): This is the timestamp when a bug was officially assigned to the right developers through Bugzilla. If a bug is assigned to multiple developers, we use the assignment time of the developer who fixed the bug. If a bug is fixed by multiple developers, we use the assignment time of the developer who committed the last changes.

Bug Severity Realization Time (T_S): This is the timestamp when the actual severity of a given bug was understood by the developers and thus the severity field of that bug was changed for the last time.

Bug Fix Time (T_F): This is the timestamp when a developer officially marked a bug as `FIXED` in Bugzilla through the `resolution` field.

Bug Assignment Period (AP): This is the lapse time between when the bug was opened and when it was assigned to the right developer. Mathematically, $AP = T_A - T_R$

Bug Fixing Period (FP): This is the period of time that developers took to fix a bug. It should be noted that it is not the actual coding time of the bug-fix. Instead, it is the time period between the bug assignment time and the bug fix time. Mathematically, $FP = T_F - T_A$

Pre-Severity Realization Period (Pre-SRP): This is the period of time developers took to understand the actual severity of the bug. Therefore, pre-severity realization time is the time between bug reporting time and the time when the severity was changed for the last time. Mathematically, $Pre\text{-}SRP = T_S - T_R$.

Post-Severity Realization Period ($Post$-SRP): This is the time developers took to fix the bug after realizing the actual severity time. Mathematically, $Post\text{-}SRP = T_F - T_S$.

Bug Verification Period (VP): This is the period of time that a developer took to verify a bug after it is marked as `FIXED` in Bugzilla. Mathematically, $VP = T_V - T_F$.

Bug Survival Period (SP): This is the period that a bug was exist in the system. Although it should be ideally the time period between the bug introduction time (T_I) and bug fixing time (T_F), in our study it is the time period between T_R and T_F. It should be noted that since T_R is always greater than T_I, our calculated SP never overestimates actual SP. However, we do not subtract the time period from SP when a bug was temporarily closed. Figure 2 visually presents all the terms and metrics in a timeline.

C. Identification of Faulty Source Code

Previous studies [9] showed that when developers fix bugs they often put the bug id in their commit message. Therefore, to get the version histories and commit messages of these four projects, first we accessed their git repositories. Then using JGit APIs, we extracted all the commit messages from the histories and searched all numbers. [6] Then we matched each number with the bug IDs. To further ensure that those are indeed bug IDs, we only accepted those commits that contain the term `bug(s)` (case insensitive). In this way, we reduced the chance of getting false positives, although we might missed some true mappings. Then we used git *diff* to compute following metrics for bug fixes:

Number of Changed Files: It is the number of files that went for changes in the bug fixing commit. If a bug was fixed in multiple commits, it is the total number of distinct files in all commits.

Number of Hunks: A hunk is a chunk of adjacent lines that was changed. For a bug fix spanning over multiple commits, it is total number of hunks in all commits. This is useful to understand how many times developers had to move here and there to fix a bug.

Code Churn: This is the total number of changed lines. Since we use git *diff* itself, the changes in comments were counted as well. For multiple commits, it is the total number of changed lines in all commits. It should be noted that if a line is changed, it is considered as a line deletion first and then addition of another line. Thus the value of code churn for a line change is two.

IV. Study Results

RQ1: What proportion of the bugs are long lived? Defining long lived bugs is subjective since the time threshold for deciding whether a bug is long lived or short lived could vary across projects, persons, or studies. In this research question, we analyze the survival time of all the fixed bugs in each subject system and define the long lived bugs more concretely for our study.

To this end, we first group bugs based on their survival period (SP) and count the number of bugs in each group as shown in Table II. Results show that around 50%(+/-4%) of the total (fixed) bugs were fixed within a week. This indicates that even in open source project, developers are active in fixing bugs. 83%-90% of bugs were fixed within six months. However, as the results show, 10% to 17% of bugs took more than six months to be fixed.

Although many of us believe that a bug could be considered as long lived if it survives more than six months, in this study we have considered only those bugs as long lived that survive more than one year. There are two main reasons behind this decision. First, we wanted to be more conservative so that we can investigate really long lived bugs. Second, since 2006, the Eclipse Foundation has coordinated an *annual simultaneous release* for all projects. Therefore, if a bug was not fixed in one year, it is expected that the bug propagated through at least two major releases. And it would not be a pleasant experience for a user if s/he experiences the same bug in subsequent major versions of a software.

Surprisingly, even for such a conservative definition, we found more than 4,000 long lived bugs, in total, in these four subject systems. We believe this is a huge number and thus it is important to investigate them quantitatively and qualitatively.

[6] http://www.eclipse.org/jgit/

TABLE II
Bug Fix Time

Time	JDT		CDT		PDE		Platform	
	# Bugs	[%]	# Bugs	[%]	# Bugs	[%]	# Bugs	[%]
<1 day	5,009	26.54	1,911	26.32	2,154	31.43	8,244	24.44
1-7 days	4,788	25.37	1,485	20.45	1,570	22.91	7,420	21.99
8-30 days	3,704	19.63	1,230	16.94	1293	18.86	6,442	19.09
1-6 months	3,604	19.10	1,426	19.64	1,212	17.68	7,101	21.05
6-12 months	855	4.53	567	7.81	353	5.15	2,173	6.44
>1 year	913	4.84	641	8.83	272	3.97	2,358	6.99
Total	18,873	100.00	7,260	100.00	6,854	100.00	33,738	100.00

TABLE III
Importance of Long Lived Bugs

System	Blocker	Critical	Major	Normal	Minor	Trivial
JDT	1	5	49	712	119	27
CDT	2	4	61	523	37	14
PDE	0	6	20	224	13	9
Platform	6	42	221	1856	170	63
Total	9	57	351	3315	339	113

There are 5%-9% of total bugs in systems which takes more than one year to be fixed.

RQ2: How important long lived bugs are in terms of severity? There are two fields in Bugzilla that indicate the importance of a bug: i) severity and ii) priority. However, based on their usage, severity is more important than priority since severity actually indicates how harmful a bug is. On the other hand, priority often describes the relative work schedule of fixing a bug set by the developers for a given milestone. For example, if there are 10 critical bugs in a system but developers have time to fix only five bugs, they can set higher priority to any five bugs based on some consideration and set a relatively lower priority to others. Sometimes, developers can set high priority to even a less severe bug, if it is expected to fix easily than a critical bug. Therefore, for this research question, we emphasize severity over priority.

Table III presents the importance of long lived bugs based on their severity. Our results show that almost 90% of the long lived bugs have severity level of `normal` or above. Project-wise the proportion varied from 84% to 90%. According to the Eclipse Bugzilla documentation, only `minor` and `trivial` bugs do not interfere with normal work or use, which means that any bugs having severity level `normal` and above adversely affect user experiences. Taking that information into account, we can anticipate that the long bug fix process of these bugs was not due to the fact that they were trivial.

Now let us a take closer look into more severe bugs: `critical` and `major` (`blocker` bugs generally do not interfere directly users). Our results show that only 1% to 2% of long lived bugs were `critical`, whereas 5% to 10% of long lived bugs were `major` in each system. The absolute number ranged from 4 to 42 for `critical` and 20 to 221 for `major` bugs. Considering that a `critical` bug causes program crashes and/or data loss and a `major` bug causes major loss of function, these numbers are high, especially since all of them took more than one year to be fixed.

Severity Realization Period (SRP): As a part of this research question, we are also interested in investigating how

TABLE IV
ANALYSIS OF SEVERITY FOR CRITICAL AND MAJOR BUGS

System	# Bugs	Sev. Changed	Proportion	Max. Changed
JDT	54	23	42.59%	3
CDT	65	21	32.31%	4
PDE	26	11	42.31%	3
Platform	263	115	43.73%	5

TABLE V
TIME NEEDED FOR UNDERSTANDING BUG SEVERITY

System	Pre-SRT (Days)			Post-SRT (Days)		
	Avg.	Med.	Max.	Avg.	Med.	Max
JDT	374	163	1890	338	320	1712
CDT	80	7	590	760	700	1442
PDE	348	388	1274	300	34	1201
Platform	351	164	2208	498	410	2730

long it takes to understand the severity of the bugs. Our initial hypothesis was that:

Perhaps it took long time to realize the severity of these important (severity level of `critical` or `major`) bugs. But once the severity was realized it should not take long time to fix them since they are important problems to solve.

For this analysis, we have considered only those bugs that have severity level of `major` or higher because they are the most important ones. From Table IV, we see that the severity level of 32%-43% of such bugs was corrected later. This indicates that the bug reporters could understand the actual severity level for more than 50% of the bugs at the time of bug posting. Therefore, it is evident that developers took more than one year to fix a large number of bugs even after they realized that the bugs are very important.

Now we analyze the bugs, which severity has corrected later in Table V. Our results show that it took almost a year on average to realize the correct severity level of the bug in three of the four projects. Only exception is the CDT, where the average $Pre\text{-}SRP$ was 80 days. The maximum $Pre\text{-}SRP$ of each system shows that for some bug it took several years to realize the severity. On the other hand, for these bugs, it took another year on average to be fixed. For CDT, which was the best in terms of average $Pre\text{-}SRP$, $Post\text{-}SRP$ was more than two years. From the maximum $Post\text{-}SRP$, we see that some bugs took even three to eight years to be fixed after developers realized the actual severity level. Therefore, we can say that for most of long lived bugs $Post\text{-}SRP$ was high regardless of their $Pre\text{-}SRP$.

Duplicate Bugs: Severity is certainly the most reliable information to understand the importance of a bug since it is determined by the bug reporters and supported by developers. However, a large number of duplicate bugs also may express their importance since they often indicate that the scope of the master bug is large and/or the affected users/other developers are getting frustrated [4]. Therefore, in addition to the severity level, we also investigated the number of duplicated bugs.

Table VI presents an overview of duplicated bugs of long lived bugs. Results show that for 9% to 23% of long lived bugs, users/developers submitted multiple bug reports. From the maximum number of duplicate bugs, we see that some bugs have more than 20 duplicated bug reports. The middle columns present more fine grained results of duplicated bugs.

TABLE VI
DUPLICATE BUGS

System	# Bugs	# Dup Bugs	# Duplicated Bugs (NOD)						Max NOD
			1	2	3	4	5	>5	
JDT	913	210	101	42	27	10	13	17	26
CDT	641	52	36	8	4	3	0	1	6
PDE	272	50	32	8	2	0	2	6	15
Platform	2358	495	271	102	51	23	15	33	20

More than 90% of long lived bugs affect users' normal working experiences and thus are important to fix. However, it took a long time to fix these bugs even after realizing their severity. Moreover, there are several duplicate bug reports for these long lived bugs, which indicates the users' demand for fixing them.

RQ3: Where was most of the time spent in the bug fixing process? A bug fixing process majorly can be divided into three phases in terms of activity: i) assignment phase ii) fixing phase, and iii) verification phase. In this research question, we analyze the time taken by team leads/developers in each phase. Our initial hypothesis about the long lived bugs was:

Perhaps it took a long time to find the appropriate developers for these bugs. But once the right developers were found, it should not take too long to fix them.

Table VII presents the average, median, and maximum time of both assignment period (AP) and fixing period (FP) in terms of days for all long lived bugs. Our results show that it took more than 1.5 years on average to assign the bugs to the appropriate developers. The median AP also shows that the data is fairly normally distributed. The maximum AP shows that it can take more than six years to assign sum bugs to the correct developers. We have also observed that more than 10% of long lived bugs were reassigned 5 times or more. While we understand that there may be some other reasons for bug reassignment such as prioritizing work items or workload balancing, these findings indicate that the assignment of these long lived bugs was complex and time consuming, supporting our initial hypothesis.

However, unlike our expectations, the average FP of all the systems was quite high: around a year. By seeing the median FP for CDT, we understand the data is skewed. But for the other three subject systems, it is not the case. Also the maximum FP shows that, like the bug assignment, it took more than five years for some bugs to be fixed after they assigned to the right developers.

On the other hand, for the verification period, we found that most of the bugs were never verified, at least according to Bugzilla data. However, if they do get verified, the verification time is pretty small: less than a month for each subject system.

Bug assignment and bug fixing are still time intensive processes, despite the availability of automatic bug assignment tools that could have been used.

RQ:4 What are common reasons for long lived bugs? To answer this question, we first manually analyzed all the `critical` and `major` bug reports from JDT. We have

TABLE VII
BUG ASSIGNMENT TIME VS. BUG FIXING TIME

System	Assignment Period (AP)			Fixing Period (FP)		
	Avg.	Med.	Max.	Avg.	Med.	Max
JDT	463	374	2745	407	376	2854
CDT	603	552	2035	330	97	1815
PDE	484	482	2728	437	393	1622
Platform	459	373	3326	407	409	2854

intentionally chosen the highly severe bugs, since they should be taken seriously by the developers and thus, we will be able to identify the actual reasons of delay. We also analyzed 50 recent (`critical` or `major`) long lived bugs from PDE and Platform. Since JDT and CDT are from similar domain, we did not take any bugs from CDT. In this way, we identified a set of 105 (= 55+25+25) bug reports for manual analysis.

Tagging Methodology: As we discussed in Section II, each bug report contains a summary, description and a list of developers comments, which often provide rich information about the problems associated the bug. In order to identify the underlying reasons, first, we read the bug summary and description to understand the nature of bugs. Second, we carefully analyze developers' comments to understand the reasons for any delays since developers often discuss different problems associated with a given bug through comments. For most of the cases, the actual reasons were easily identifiable. We have quoted several key comment(s) for most of the categories to better understand the tagging procedure. In the few cases where the reasons were ambiguous, we relied on contextual information. The following summarizes a taxonomy of common reasons for long lived bugs that we found in the subject systems.

1) Hard to understand: Understanding/locating buggy statements/files in a software project is hard. Sometimes, identifying even the buggy component can be hard. For example, there is a bug (#128563) in JDT, where developers had hard time in understanding if it is a VM or JDT bug. The following comments explains the situation:

"I found something quite interesting. If you move the classes from the two output folders into the same directory and you run from there, it works fine. We generate exactly the same bytecodes in both cases. The VM should behave the same. Might be a VM bug."

After two years, another developer commented–*"I believe this is our bug, we should not reference a non accessible type in our bytecode. The fact it works at times feel like unspecified behavior from the VM."*

2) Uncertain how to fix: Sometimes developers may know how to solve a bug, but need to wait for making the solution consistent/robust with other parts of the software. The following comments in bug # 3849 represents such a scenario:

"*I would like to defer this until we know how we will implement the new Code Manipulation Infrastructure. This is only possible if we get a better undo story. Currently we can only push undo commands on the refactorings undo stack if a file is save. Otherwise the next save would flush the current undo stack which would remove the undo object for extract method.*"

3) Hard to fix: This kind of bug is hard to fix. There were lots of group discussions for a long time regarding different alternative solutions and finally the group agreed on some specific solution.

4) Risky to fix: Sometimes, bugs are caught just before the release. Then if developers think that it would be risky to change the relevant code, they generally defer it for the next release although the bug is important. Then it takes a long time to fix the bug. The following developers' comments on bug #80,000 in JDT represents such a scenario:

"Will investigate during RC2 whether there's a low risk fix for this."

Two weeks later, the same developer commented: *"Sorry, too risky to touch at this point."*

5) Incomplete fix: This is considered as one of the common problems for taking a long time to fix bugs. Developers often miss corner cases while bug fixing and need to re-fix again until the problem is not fully solved. Here is a developer's comment regarding a bug fix for #38746 in JDT.

"The fix for this problem is not sufficiently robust. Please see Bug 75454 for more information as to how things can go wrong. Not only does the situation described there happen once, but it happens 60 times on start-up (10 minutes of start-up time)."

There are also lots of other reasons for incomplete bug fixes. For a comprehensive set of reasons for incomplete bug fixes, please refer to [15].

6) Importance was not realized until duplicate bugs were reported: We found many bugs where there were some activities around the bug for some time, which we observed by reading developers' comments. After that there was no activity for a long time. Then somebody pointed out some duplicate bugs and everybody started talking again; the bug was fixed quickly. The following comment on bug # 16114 in PDE represents such an example:

"*this one is experienced by several users (see the duplicates for more info). Looks like something causes certain fragment files on the disk to be in use and when we try to delete the project (even with 'force' option), we fail. This leave us with a partially deleted project that causes more trouble after that."*

7) Reproducibility: There are some bugs that take a long time to reproduce, but once it is reproduced, it is fixed quickly. For example, it took 1 year and 4 months to reproduce the bug #268833 in Platform but took only one day to fix. This problem often happens from low quality bug report, execution difference due to platforms, and so on.

There are some interesting bugs, where users know how to reproduce the bug but it happens for some special cases and thus needs some time to reproduce. For this kind of bug, if users submit the bug without concrete data, it takes a long time to reproduce the required data that developers need to analyze the bug. Therefore, the bug fix gets delayed although the responsible developer is ready to fix it. For example, to debug an "out of memory" problem in JDT (# 54831), developers needed a heap dump, which was not submitted when the bug was posted. When the assigned developer asked for it, the bug reporter (who is actually another developer of JDT) was busy with his own work and could not submit the heap dump on time. As a result, it took a long time to fix the bug.

8) Schedule issue: Sometimes developers also feel that a bug is important to fix. However, they have more important bugs at hand that should be fixed earlier. Therefore, although the other bugs are important, they are generally deferred. For example, there is a `blocker` bug (#10800) in JDT that prevented users to put space in VM arguments. `Blocker` bugs are considered as the most severe bugs. However, such a severe bug was deferred due to scheduling issues. Certainly, other developers were not very happy about that. The following developers' comments illustrate the scenario more clearly.

"Can more explanation be given as to why this issue has been marked as LATER? Does this mean it will not be fixed any time soon? If so, I find it very unfortunate as this is a very serious bug and requires nasty work arounds. If not, then my apologies..."

In reply, the responsible team leader said: *"In this case, 'LATER' means probably not for the final 2.0 release (tentatively scheduled for sometime in May). Quite simply, this problem was not deemed as critical as a lot of other problems that need to be solved for 2.0. The debug committers have A LOT to do before 2.0. But the beauty of an open source project is that if someone feels strongly about a particular feature or bug, they can make a contribution. If you would like to contribute a fix, I would be happy to review it."*

Finally, it took more than two years to fix the bug.

9) Reopened due to misunderstanding These bug are not really long lived. They got fixed within months after they were reported. But the reporter misunderstood something and reopened the bug again. Then the other developers clarified the mistakes the reporter was making and finally again marked it as `FIXED` and `RESOLVED`.

10) Infrequent use case: This kind of bug is important to fix considering their destruction ability. However, they are not too frequent use cases. Therefore, developers just defer it for next milestone. For example, due to the bug # 130874 in JDT, a user can lose his/her Java code template references. However, developer deferred it by making following comment.

"We should definitely fix this during 3.5. Too late for 3.4 and really not a very common case."

11) Others: There are also other reasons for delay in bug fixing such as expert developers are on vacation, dependency on other bugs to be fixed, and various document fixing.

12) As-usual delay: We have not found any specific reasons for these bugs by analyzing developers' comments and thus we considered them as as-usual delay. Since these bugs are already marked as important, if there are some specific reasons (mentioned above) to make delay, it is highly likely that developers will discuss it like the other bugs. However, it is also possible that the fixes were deferred due to scheduling issues. The following comment for bug #149316 in JDT can be served as an example of as-usual delay.

"*Thanks for the good examples, sorry for the wait. fixed > 20080422.*"

We encountered an interesting finding while analyzing the bug reports manually. We started our manual investigation with JDT and listed all the common reasons from there. We have not found any new common reason when analyzing the bug reports for PDE or Platform. Therefore, we believe that this is a comprehensive list of reasons for long lived bugs. It should be noted that these reasons are not mutually exclusive.

TABLE VIII
REASONS OF SAMPLED LONG LIVED BUGS

Reason	# Bugs	Bug IDs
1)	7	113870 (PDE), 128563 (JDT), 241241 (Platform), 245008 (Platform), 247766 (PDE), 268833 (Platform), 278598 (PDE)
2)	3	3849 (JDT), 36204 (JDT), 133072 (PDE)
3)	14	3849 (JDT), 24951 (PDE), 36204 (JDT), 38746 (JDT), 40243 (JDT), 46407 (JDT), 67425 (JDT), 82850 (JDT), 99137 (JDT), 233643 (PDE), 233773 (Platform), 266651 (JDT), 273450 (Platform), 295200 (JDT)
4)	2	80000 (JDT), 102780 (JDT)
5)	4	1766 (JDT), 33035 (JDT), 36204 (JDT), 136135 (PDE)
6)	14	36204 (JDT), 46216 (JDT), 50735 (JDT), 109636 (JDT), 117698 (JDT), 156168 (JDT), 175226 (JDT), 224880 (Platform), 243894 (Platform), 257202 (Platform), 266651 (JDT), 267649 (Platform), 273450 (Platform), 278598 (PDE)
7)	9	1766 (JDT), 39222 (JDT), 54831 (JDT), 82850 (JDT), 83473 (JDT), 195183 (JDT), 262032 (Platform), 294650 (Platform), 298795 (Platform)
8)	7	3920 (JDT), 19251 (PDE), 46216 (JDT), 67425 (JDT), 224880 (Platform), 235572 (Platform), 277638 (Platform)
9)	8	6437 (JDT), 19248 (PDE), 28637 (JDT), 44035 (JDT), 61744 (PDE), 132333 (PDE), 158589 (PDE), 271373 (Platform)
10)	2	34033 (PDE), 130874 (JDT)
11)	8	12955 (JDT), 16686 (JDT), 20919 (PDE), 24951 (PDE), 29799 (PDE), 34399 (PDE), 231936 (PDE), 290324 (PDE)
12)	29	21100 (PDE), 26556 (JDT), 38288 (PDE), 39803 (JDT), 51862 (PDE), 89347 (JDT), 95288 (JDT), 97541 (JDT), 111419 (JDT), 128303 (PDE), 129689 (PDE), 149316 (JDT), 154823 (JDT), 175133 (JDT), 181954 (JDT), 209537 (JDT), 226595 (Platform), 234623 (Platform), 235554 (Platform), 236104 (Platform), 237025 (PDE), 238943 (JDT), 258952 (Platform), 262032 (Platform), 267173 (Platform), 275910 (Platform), 277638 (Platform), 279781 (Platform), 285101 (Platform)

> Reasons for long lived bugs are diverse. While problem complexity, problems in reproducing errors, and not understanding the importance of some of the bugs in advance are the common reasons, we observed there are many bug-fixes that were delayed without any specific reason.

RQ5: What is the nature of bug fixes? In this research question, we investigate the nature of bug fixes in terms of source code changes. More specifically, we focused on the number of changed files, number of hunks, and code churns as described in Section III-B. These metrics are often used to get a rough idea about change effort, although understanding the actual change effort is difficult and depends additionally on the implemented algorithm and code complexity itself. Analogous to previous study results [22], our initial hypothesis was:

The required source code changes to fix most of the long lived bugs would be large.

To this end, we first analyze the source code changes in

TABLE IX
ANALYSIS OF BUG FIXES

System	# Bugs	#Number of Files (NOF)						Med	Max
		1	2	3	4	5	>5	NOF	NOF
JDT	223	80	41	37	10	15	40	2	47
CDT	185	50	32	19	18	8	58	3	237
PDE	105	34	13	13	12	4	29	3	211
Platform	740	349	114	72	47	36	122	2	91
Total	1253	513	200	141	87	63	249	-	-
(%)	-	40.94	15.96	11.25	6.94	5.03	19.87	-	-

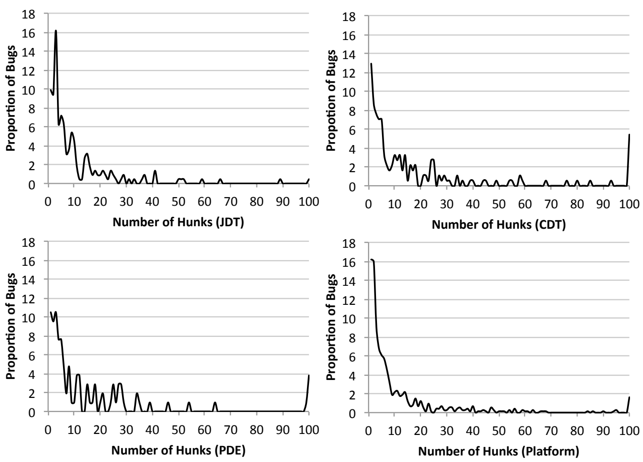

Fig. 3. Number of Hunks Vs. Proportion of Bugs

terms of number of changed files. We were able to identify fixed files for 280, 264, 154, and 1,109 bugs in JDT, CDT, PDE, and Platform respectively, using the *methodology* described in Section III-B (Table IX). As we stated in our initial hypothesis, we expected a large number of changed files for most long lived bugs. Surprisingly, we found more than 40% of fixes involved only one source code file. This proportion varied from 27% to 47% project-wise. For only 30% of long lived bugs, the required changes spanned over more than three files. From our results it is also noticeable that the maximum number of changed files in each system is quite high. However, when we manually investigated such changes, we found they are often moving files from one directory to another, or adding a test suite to test a specific or multiple bugs.

Second, we analyzed the changes in terms of hunk size to get more fine grained results. A large number of hunks indicates that developers needed to modify a lot of different places to fix the bugs. Figure 3 presents the number of bugs for each hunk size. Our results show that 43% to 53% of bugs was fixed by changing five hunks or less. More than 70% of long lived bugs for JDT and Platform was fixed within 10 hunks, whereas the numbers are 16 and 17 for PDE and CDT respectively. The median number of hunks for all long lived bugs is only 5. Considering that, this is an overestimated measurement of source code changes since we have analyzed textual *diff* results (so changes in comments have been also counted), we believe the number of hunk is low. It should be noted that we presented all the bugs that involved more than 100 hunks in the graph at the end. Therefore, there is a spike at the end of each graph.

Finally, by investigating further low level changes (at line level), we found that a considerable proportion of bugs required major changes. More specifically, for 23% of bugs, the value of code churn was more than 100. However, there are even a larger proportion of bugs that required changes less that 20 lines. For example, for 8-14% of bug fixes, the value of code churn was from 1 to 5, for 7-12% the value of code churn was from 6 to 10, and for 9-13% the value of code churn was from 11 to 20. Recalling Section III-B, it should be noted that the code churn value of one line change is 2, whereas an addition or deletion of line is 1. Therefore, a value of code churn value of 10 may be changes in only five lines. We understand that some smaller bug fixes can be complex. But at the same time, we also stress that many long lived bugs could be fixed quickly through careful prioritization.

To support our claim, now we discuss two bug fixes to show how a simple fix can take a long time to be fixed. Most importantly, both of these bugs are considered as important by the reporters/developers.

Bug # 38260: This is a bug in the SWT component of Eclipse Platform. The bug was first reported as having a `normal` severity level. However, within 15 days, it was reconsidered to be `critical`. The following is the bug description provided by the reporter:

"*When I use the CCombo with the dialog, the dropdown list shown in the back of the Dialog.*
I did as following.
1. Create one sample application with a Group.
2. The parent of group is one shell.
3. I create a dialog.
4. I put my application to this dialog. and change the Group's parent as the dialog's parent.
Now the above descripbed pbm occured."

From the bug comments, we found that the bug was reproduced within three days and the actual problem was identified within a month. However, it took more than ten months to fix the bug. Figure 4 presents the changed code for fixing this bug and it was only a one line change.

Bug # 195183: This is a `major` bug in the Debug component in JDT. The bug summary and description (condensed) are as follows: *"JavaClassPath.performApply() uses original instead of working copy causes NPE"*
"Steps To Reproduce: I use JavaClassPath in my custom launch config and has some code like:" ....[some code snippet] *"and as soon as performApply() is called isDefaultClasspath() fails since it is passed in a null as a launchconfig even though I passed in a newly created one. This worked fine in eclipse 3.2 and it seem the culprit is that wc.getOriginal() is used instead of just wc. Resulting in a NPE."*

From the bug description, we can see that the bug report was very specific. The reporter clearly pointed out that there are some problems with the $getOriginal()$ API. Interestingly, from the bug-fix (Figure 5), we found that only one line was changed and the change was the removal of the $getOriginal()$ API. But by that time, more than two years had passed.

We believe that one year or more is too long time to fix the bugs like these two examples, especially considering that they were considered as very important.

```
--- a/bundles/org.eclipse.swt/Eclipse SWT Custom Widgets/common/org/eclipse/swt/custom/CCombo.java
+++ b/bundles/org.eclipse.swt/Eclipse SWT Custom Widgets/common/org/eclipse/swt/custom/CCombo.java
@@ -76,7 +76,7 @@ public CCombo (Composite parent, int style) {
          if ((style & SWT.READ_ONLY) != 0) textStyle |= SWT.READ_ONLY;
          if ((style & SWT.FLAT) != 0) textStyle |= SWT.FLAT;
          text = new Text (this, textStyle);
-         popup = new Shell (getShell (), SWT.NO_TRIM);
+         popup = new Shell (getDisplay(), SWT.NO_TRIM | SWT.ON_TOP);
          int listStyle = SWT.SINGLE | SWT.V_SCROLL;
          if ((style & SWT.FLAT) != 0) listStyle |= SWT.FLAT;
          if ((style & SWT.RIGHT_TO_LEFT) != 0) listStyle |= SWT.RIGHT_TO_LEFT;
```

Fig. 4. Bug Fixing Changes for # 38260 in SWT Component of Platform

```
--- a/org.eclipse.jdt.debug.ui/ui/org/eclipse/jdt/debug/ui/launchConfigurations/JavaClasspathTab.java
+++ b/org.eclipse.jdt.debug.ui/ui/org/eclipse/jdt/debug/ui/launchConfigurations/JavaClasspathTab.java
@@ -272,7 +272,7 @@ public class JavaClasspathTab extends AbstractJavaClasspathTab {
  public void performApply(ILaunchConfigurationWorkingCopy configuration) {
          if (isDirty()) {
                  IRuntimeClasspathEntry[] classpath = getCurrentClasspath();
-                 boolean def = isDefaultClasspath(classpath, configuration.getOriginal());
+                 boolean def = isDefaultClasspath(classpath, configuration);
                  if (def) {
                         configuration.setAttribute(IJavaLaunchConfigurationConstants.ATTR_DEFAULT_CLASSPATH, (String)null);
                         configuration.setAttribute(IJavaLaunchConfigurationConstants.ATTR_CLASSPATH, (String)null);
```

Fig. 5. Bug Fixing Changes for # 195183 in Debug Component of JDT

Unlike previous studies, we found that a bug surviving for a year or more does not necessarily mean that it requires a large fix. We found that 40% of long-lived bug fixes involved few changes in only one file.

V. Threats to Validity

Construct Validity: We used two artifacts: bug reports from the bug tracking system and source code changes from the version history, which are generally well understood. We have used also well known metrics in our data analysis such as various time periods, the number of changed files, the number of hunks, code churns, which are straightforward to compute. Both the used dataset and version histories are also publicly available, which enable the replication of this study. Therefore, we argue for a strong construct validity.

Internal Validity: In our study, we relied on the information from the bug tracking system and version histories. However, the information in these systems may not be completely accurate. For example, a developer may commit a bug fixing change a long time after she actually fixed the bug. Similarly, a tester may change the bug status from `FIXED` to `VERIFIED` a long time after she actually verified the bug.

To delineate the common reasons of long lived bugs, we manually analyzed bug reports. There might have been some unintentional misinterpretations during the manual verification due to the lack of domain knowledge or the lack of useful contextual knowledge. However, we held extensive discussions to minimize this threat.

The phenomena studied had yearly major releases. Systems with more frequent release cycles may well exhibit different phenomena, although there will still be long lived bugs. The number of release cycles and the lapse times for long lived bugs in this context are likely to be different.

External Validity: We have used only four subject systems in our experiment and all of them are open source projects. Although, they are very popular projects and the used dataset has more than 165,000 bug reports, our findings may not be generalizable to other open source projects or industrial projects. This risk of insufficient generalization could be mitigated by adding more subject systems (both open source and industrial). This will be explored in our future work.

VI. Related Work

The study of software bugs/faults has been an active research area for nearly two decades. Perry and Stieg [16] were among the first to analyze software faults in a large evolving software system. Since then, researchers analyzed various software artifacts relevant to bugs (e.g. bug report, bug fixing changes) to understand and to improve different steps (e.g. bug reporting, triaging, localizing, fixing) of the bug fixing processes. For example, Thung et al. [20] investigated when a bug should be reported. Bettenburg et al. [3] studied the qualities of a good bug report. In another study, Bettenburg et al. [4] investigated the extents and reasons of duplicated bug reports. Ahmed et al. [10] predicted the severity of bugs. Anvik et al. [2] and Shokripour et al. [19] proposed approaches for automatic bug assignment. Saha et al. [17] and Zhou et al. [24] proposed different approaches for automatic bug localization. To complement these studies, in this paper, we focused on long lived bugs to understand their characteristics and reasons.

The work closest to ours is the study of bug-fix time prediction, since these studies also identify the factors that are correlated to bug fixing time. Weiss et al. [21] considered the text (summary and description) in the bug report as the prime factor and used that to predict bug fix time. Panjer [14] observed that commenting activity, bug severity, product, component, and version are the most influential factors in predicting bug fix time. Giger et al. [7] found that the assigned developer, the bug reporter, and the month when the bug was reported have the strongest influence on the bug fixing time. Zhang et al. [23] also found the same results for commercial projects. Anbalagan et al. [1] found a strong relationship between bug fixing time and the number of people participating in the bug report. Marks et al. [12] observed different results for different projects. They found that bug filing time is important for

Mozilla project, whereas, bug severity is the key for Eclipse.

While these studies vary in terms of analysis and techniques used, some of the common approaches used in these studies are that researchers used various machine learning or data mining techniques to analyze the whole bug dataset in identifying the overall factors affecting bug fixing time. Bhattacharya and Neamtiu [5] pointed out that most attributes used by prior work do not correlate with bug-fix time when analyzed in isolation, and thus they emphasized on finding new attributes that correlate with bug-fix time in isolation. We stress that it is also important to analyze various kinds of bug-fixes in isolation to gain better insight about specific group of bugs. For example, Shihab et al. [18] studied and predicted reopened bugs, Park et al. [15] investigated supplementary bug-fixes, and Ngyuen et al. [13] analyzed recurring bug-fixes. In this study, we analyzed long lived bugs to advance empirical knowledge further regarding long-term delays in the bug fixing process.

There is another group of studies that investigated the actual source code changes for bug fixes to study bug fixing time. Canfora et al. [6] found relationships between different program constructs and bug survival time. For example, exception handling leads to low bug survival time. Zhang et al. [22] found that bug fixing time increases with the increase of code churns. In our study, we have also analyzed the source code changes for long lived bug-fix and showed that there are many long lived bugs that required only few changes in one file.

VII. Conclusion

Bug fixing is a fundamental and critical activity in the software development and maintenance phases since buggy behavior may cause not only costly failures but also can affect user's overall experiences with the software product. In this paper, we showed that although the software development and maintenance processes have advanced a lot, there are still a significant number of bugs in each project that survive for more than a year. More than 90% of these long lived bugs affected users' normal working experience. The average bug assignment time was more than one year and the bug fix time after the assignment was another year on average. When we analyzed the bug descriptions and the developers' comments around these bugs, we found that the reasons for long lived bugs are diverse. While problem complexity, problems in reproducing, and not understanding the importance of some of the long lived bugs in advance are the common reasons, we observed there are many bugs that were delayed without any specific reasons. Finally, by investigating the actual source code changes for these long lived bugs, we noted that a bug surviving for a year or more does not necessarily mean that it requires a large fix. In fact, we found 40% of long-lived bug fixes that involved few changes in only one file.

In summary, our results indicate that the overall bug fixing time of many, if not all, long lived bugs can be reduced through careful prioritization. Our findings also indicate that although there are a number of tools for supporting bug triaging and fixing (e.g. automatic bug assignment, bug fix time prediction), we appear to realize very few benefits from them. There may be two possible reasons: i) developers are not aware that these tools exist, or ii) the tools do not meet developers needs or expectations. In the future, we plan to conduct a developers survey to understand the reasons for this phenomenon. We believe all of these findings together will play an important role in developing new approaches for bug triaging as well as improving the overall bug fixing process.

Acknowledgement: We thank Ahmed Lamkanfi at the University of Antwerp for extending their bug dataset significantly for our study. This research was supported in part by NSF Grants CCF-0820251 and CCF-0845628.

References

[1] P. Anbalagan and M. Vouk. On predicting the time taken to correct bug reports in open source projects. In *Proc. of ICSM*, pages 523–526, 2009.
[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. of ICSE*, pages 361–370. ACM, 2006.
[3] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proc. of FSE*, pages 308–318. ACM, 2008.
[4] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful...really? In *Proc. of ICSM*, pages 337–345, 2008.
[5] P. Bhattacharya and I. Neamtiu. Bug-fix time prediction models: can we do better? In *Proc. of MSR*, pages 207–210. ACM, 2011.
[6] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta. How long does a bug survive? an empirical study. In *Proc. of WCRE*, pages 191–200. IEEE Computer Society, 2011.
[7] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proc. of RSSE*, pages 52–56. ACM, 2010.
[8] N. Japkowicz. Learning from imbalanced data sets: A comparison of various strategies. In *Proc. of the AAAI Workshop on Learning from Imbalanced Data Sets*, pages 10–15. AAAi, 2000.
[9] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, Mar. 2008.
[10] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *Proc. of MSR*, pages 1–10, 2010.
[11] A. Lamkanfi, J. Pérez, and S. Demeyer. The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information. In *Proc. of MSR*, pages 203–206. IEEE Press, 2013.
[12] L. Marks, Y. Zou, and A. E. Hassan. Studying the fix-time for bugs in large open source projects. In *Proc. of Promise*, pages 11:1–11:8. ACM, 2011.
[13] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proc. of ICSE*, pages 315–324, 2010.
[14] L. D. Panjer. Predicting eclipse bug lifetimes. In *Proc. of MSR*, pages 29–32. IEEE Computer Society, 2007.
[15] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *Proc. of MSR*, pages 40–49, 2012.
[16] D. E. Perry and C. S. Stieg. Software faults in evolving a large, real-time system: a case study. In *Proc. of ESEC*, pages 48–67, 1993.
[17] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *Proc. of ASE*, pages 345–355, 2013.
[18] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. Hassan, and K.-i. Matsumoto. Studying re-opened bugs in open source software. *Empirical Softw. Engg.*, 18(5):1005–1042, 2013.
[19] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proc. of MSR*, pages 2–11, 2013.
[20] F. Thung, D. Lo, L. Jiang, Lucia, F. Rahman, and P. Devanbu. When would this bug get reported? In *Proc. of ICSM*, pages 420–429, 2012.
[21] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proc. of MSR*, pages 1–8, 2007.
[22] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan. An empirical study on factors impacting bug fixing time. In *Proc. of WCRE*, pages 225–234. IEEE Computer Society, 2012.
[23] H. Zhang, L. Gong, and S. Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *Proc. of ICSE*, pages 1042–1051. IEEE Press, 2013.
[24] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proc. of ICSE*, pages 14–24, 2012.