A Longitudinal Analysis of Bug Handling Across Eclipse Releases

Zeinab Abou Khalil*§, Eleni Constantinou*, Tom Mens*, Laurence Duchien§, Clément Quinton§
{*zeinab.aboukhalil, eleni.constantinou,tom.mens*}*@umons.ac.be*
{*laurence.duchien, clement.quinton*}*@univ-lille.fr*
* University of Mons, Belgium
§ University of Lille & Inria Lille, France

***Abstract*—Large open source software projects, like *Eclipse*, follow a continuous software development process, with a regular release cycle. During each release, new bugs are reported, triaged and resolved. Previous studies have focused on various aspects of bug fixing, such as bug triaging, bug prediction, and bug process analysis. Most studies, however, do not distinguish between what happens *before* and *after* each scheduled release. We are also unaware of studies that compare bug fixing activities across different project releases. This paper presents an empirical analysis of the bug handling process of *Eclipse* over a 15-year period, considering 138K bug reports from *Bugzilla*, including 16 annual Eclipse releases and two quarterly releases in 2018. We compare the bug resolution rate, the fixing rate, the bug triaging time and the fixing time *before* and *after* each release date, and we study the possible impact of "release pressure". Among others, our results reveal that Eclipse bug handling activity is improving over time, with an important decrease in the number of reported bugs before releases, an increase in the bug fixing rate and an increasingly balanced bug handling workload before and after releases. The recent transition from an annual to a quarterly release cycle continued to improve the bug handling process.**

I. Introduction

In continuous software engineering, every major software release provides a significant amount of new or modified functionality compared to the previous release. Developers strive to resolve as many bugs as possible in current and upcoming releases. Unfortunately, not all bugs can be resolved and fixed before the release deadline. In order to address the most important bugs for each release, a bug handling process is followed, relying on dedicated collaborative bug tracking tools such as Bugzilla. Researchers have investigated different factors in the bug handling process that may affect bug triaging and fixing time, and provided models for predicting and estimating this time [1]–[10]. They also studied techniques to improve the process of bug fixing [11], [12], bug triaging [13]–[15], or assigning developers to bug reports [16]–[21].

The research community has investigated several aspects of pre- and post-release bugs. The number and rate of resolved and fixed bugs are important indicators of software quality [22]. Rapid release policies might affect bug handling quality, since releases are delivered more often, and the community may have less time to address unresolved bugs. Khomh et al. [23], [24] analyzed Mozilla Firefox to study the effect of switching to a rapid release cycle on the speed and proportion of bugs being fixed. Interviews with Mozilla developers revealed that the success of such a new release policy on bug handling depends on the prior preparation to adapt to this policy [24]. In the second half of 2018, Eclipse has also transitioned from an annual to a quarterly (13-week) release cycle. It therefore becomes relevant to investigate how this change has affected Eclipse's recent bug handling activity.

In this paper, we study the bug history of 18 major releases of the Eclipse core projects over a 15-year lifetime. In particular, we investigate: 1) how the bug handling time differs *before* and *after* an upcoming scheduled release; and 2) if and how the switch to a quarterly release cycle affects the bug handling process. To do so, we rely on four measurements to quantify bug handling: triaging and fixing time, and resolution and fixing rate. Our investigation is guided by four research questions:

RQ$_0$: ***How does the number of bugs differ before and after each release?*** This exploratory question provides an initial understanding of the magnitude of bug-related activity for each Eclipse release in the period *before* the release is scheduled and the period *afterwards.*

RQ$_1$: ***How do the bug resolution and fixing rates evolve over releases, and how do they differ before and after each release?*** We intuitively expect that Eclipse maintainers handle bugs more intensively in the period *before* than *after* the upcoming release date, as they strive to resolve and fix as many bugs as possible in the upcoming release. Moreover, to assess the bug handling process improvement over time, we study the bug resolution and fixing rates and check whether or not they are increasing across releases.

RQ$_2$: ***How do the bug triaging and fixing time differ before and after each release?*** We assume that intense bug handling activity *before* an upcoming release leads to faster bug triaging and fixing times than *after* the release. We thus investigate the differences between the two time periods in terms of days elapsed to triage and fix bugs.

RQ$_3$: ***How do the bug triaging and fixing time change when approaching a release deadline?*** Based on 40 interviews, Murphy-Hill et al. [25] observed that developers become more conservative in their changes to avoid the risk of introducing new bugs, and focus more on bug fixing activities as the release deadline approaches. Similarly, we assume that for Eclipse, bugs are triaged and fixed faster closer to a release deadline, due to the increased pressure to resolve as many bugs as possible for the upcoming release. We also assume that in the light of an upcoming release, the triaging and fixing time of bugs of the current release will increase, since developers may prefer to focus on bugs of the upcoming release rather than on the ones of the current, already published, release. To this end, we empirically compare the differences in bug triaging and fixing time of the current and upcoming release in function of the upcoming release deadline.

The research questions are analyzed for three ranges of Eclipse releases: 1) the 3.x release series; 2) all 4.x releases following an annual release cycle; and 3) all 4.x releases following a quarterly release cycle. We report and discuss the findings for each release range so as to identify the changes in the bug handling process throughout Eclipse's history, as well as the effect of switching from an annual to a quarterly release cycle.

II. Related Work

A. Bug fixing in short release cycles

Khomh et al. [23] empirically studied the effect of rapid releases on software quality for Mozilla Firefox. They quantified quality in terms of runtime failures, presence of bugs and outdatedness of used releases. Related to our work, they compared the number of reported, fixed and unconfirmed bugs and the fixing time during both the testing period, i.e., the time between the first alpha version and the release, and the post-release period. They showed that less bugs are fixed during the testing period and that bugs are fixed faster under a rapid release model. In a follow-up work [24], the authors reported that, although post-release bugs are fixed faster in a shorter release cycle, a smaller proportion of bugs are fixed compared to the traditional release model. Interviews conducted with six Mozilla employees revealed that they can be "less effective at triaging bugs with rapid release" and that more beta testers using the rapid releases can generate more bugs.

Da Costa et al. [26] studied the impact of Mozilla's rapid release cycles on the integration delay of addressed issues. They showed that compared to the traditional release model, the rapid release model does not integrate addressed issues more quickly into consumer-visible releases. They also found that the issues are triaged and fixed faster in rapid releases. They extended their work in [27] where they reported that issue triaging time is not significantly different among the traditional and rapid releases.

B. Bug triaging

Saha et al. [1] extracted code change metrics, such as the number of changed files, to identify the reasons for delays in bug fixes and to improve the overall bug fixing process in four Eclipse projects: JDT, CDT, Plug-in Development Environment (PDE), and Platform. Their results showed that a significant number of long-lived bugs could be reduced through careful triaging and prioritization if developers would be able to predict their severity, change effort and change impact in advance.

Zhang et al. [8] studied factors affecting delays incurred by developers in bug fixing time. They analyzed three Eclipse projects: Mylyn, Platform and PDE. They found that metrics such as severity, operating system, description of the issue and comments are likely to impact the delay in starting to address and resolve the issue.

Hooimeijer and Weimer [7] presented the correlation between bug triaging time and the reputation of a bug reporter. They designed a model that uses bug report fields, such as bug severity and submitter's reputation, to predict whether a bug report will be triaged within a given amount of time in the Mozilla Firefox project.

C. Bug fixing time prediction and estimation

Panjer [3] carried out a case study on Eclipse projects, and showed that the most influential factors affecting bug fixing time are found in initial bug properties (e.g., severity, product, component, and version), and post submission information (e.g., comments).

Giger et al. [4] found that the assigned developer, bug reporter and month when the bug was reported have the strongest influence on the bug fixing time in Eclipse, Mozilla, and Gnome. Marks et al. [5] studied different features of a bug report in relation to bug fixing time using bug repository data from Mozilla and Eclipse. The most influential factors on bug fixing time were bug location and bug reporting time.

Zou et al. [28] investigated the characteristics of bug fixing rate and studied the impact of a reporter's different contribution behaviors to the bug fixing rate in Eclipse and Mozilla. Among others, they observed an increase in fixing rate over the years for both projects. On the other hand the observed rates were not high, especially for Mozilla.

All these studies are valuable in understanding the overall bug fixing process, factors affecting bug fixing time, bug fixing time estimation and triaging automation. In our work, we focus on the bug triaging and fixing time, and on bug resolution and fixing rate. We are not aware of any study comparing these metrics before and after the release is delivered, and how they evolve over successive releases.

III. Experimental Setup

This section presents our experimental design, the selected case study, the data extraction process and the metrics that will be used to answer the research questions. The datasets and scripts generated for the current study are publicly available in a replication package [29].

A. *Selected Case Study*

We selected Eclipse as a case study because it: 1) is a large and mature project that has had a sufficiently long release history with associated bug reporting activity 2) has been widely studied in previous software evolution research [30], [31] and in research about bug fixing in particular [3], [6]; and 3) is a popular open source project that has a stable development community and a regular release process.

Eclipse relies on the Bugzilla bug tracking platform [32] to manage and keep track of its bugs, allowing developers to effectively manage all concerns of the bug life cycle. The Eclipse community separates its development between the core Eclipse projects and its plugins. We focus only on bugs related to the core of Eclipse, as the bug handling activity for plugins can be affected by the absence of factors such as continuous bug monitoring and continuity of the plugin development. The core of Eclipse is composed of the projects *Platform*, *JDT*, *e4*, *Incubator*, *Equinox* and *PDE* [33]. *e4* is an incubator for community exploration of future technologies of Eclipse and uses a different versioning scheme than the other projects; we therefore excluded it from our analysis. Since our focus is on the relation between bug reports and the official Eclipse releases, we also excluded *Incubator* because the few (49) bugs reported for this project do not provide a version of the Eclipse release, i.e., all bugs provide an *unspecified* version.

Our analysis of Eclipse starts from version 3.0, which is the first release coordinated and shipped by the independent Eclipse Foundation; previous releases were coordinated and controlled by IBM. Since release 3.0 in June 2004 until release 4.8 in June 2018, Eclipse followed an annual "simultaneous release" scheme[1] where a new release was delivered every year in June. In June 2012, Eclipse switched from the 3.x series to the 4.x series, highlighted by a "dual" release of 3.8 and 4.2 that were maintained in parallel[2].

Each annual release since 3.3 until 4.5 was followed by two "service releases" in September and February, respectively. Releases 4.6 and 4.7 had three "update releases" in September, December and March. Since release 4.9, Eclipse has switched to a new quarterly release policy (i.e., every 13 weeks) without intermediate update releases.

B. *Bug Tracker Data*

Our empirical analysis is based on bug report data extracted from Bugzilla for each Eclipse release. A typical bug report contains a variety of fields, such as the description, product, component, status, reporter, fixer, severity, version, target milestone, and operating system. Each bug report is accompanied by a *history* containing all events that occurred during the bug's life cycle (e.g., bug reporter and creation time, change of resolution and status field of a bug, assigned developer, etc.).

All these aspects play a role in the bug handling process. Using the Disco process mining tool [34] we extracted the bug handling process based on the 138,445 bug reports in our dataset; the process to acquire this dataset is explained in detail in Section III-C. The results are shown in Fig. 1. Typically, a bug gets introduced as `NEW` and then either becomes `CLOSED` (5.5%) or `ASSIGNED` to a developer who becomes responsible for its resolution (36.9%).

It is important to note that many bugs get `RESOLVED` without ever being `ASSIGNED` (54.3%). The *resolution status* of a `RESOLVED` bug may contain different possible values. In the majority of the cases (55.3%), the status will be `FIXED` (i.e., the bug has been resolved by fixing it). However, many bugs get resolved by marking them as `DUPLICATE` (18.6% of the cases) or as `WORKSFORME` (implying that the bug cannot be reproduced by the assignee, corresponding to 10.1% of the cases). Other resolution statuses are `INVALID` (7.4%), `WONTFIX` (7.2%), `NOT_ECLIPSE` (1.4%) and `MOVED` (0.01%). Less than 1 out of 3 bug reports ever reach the `VERIFIED` state after they have been `RESOLVED` (28%). Hence, one should not rely on this state to carry out empirical analyses about bug handling.

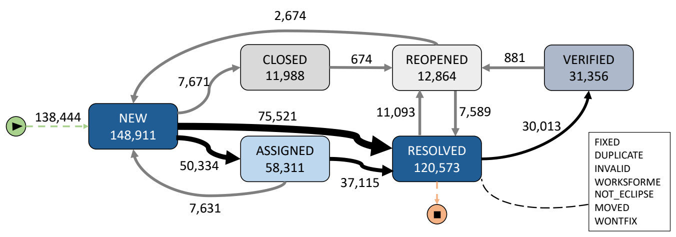


Fig. 1: The life-cycle of Eclipse bug reports.

C. *Data Processing*

To extract the Eclipse bug reports and the respective bug histories we used the Bugzilla API[3]. Since we focus on the core of Eclipse, we only considered bug reports for which the product field was tagged as *Platform*, *JDT*, *Equinox* or *PDE*. We initially extracted 205,031 bugs for these projects and excluded 27,799 bugs that were marked as *enhancement* in the *severity* field, since feature enhancements are considered to be new functionality requests rather than bugs [35].

In a next step we filtered bugs based on the *version* field. As the goal of this work is to study the bug resolution process in relation to each Eclipse release, we only considered those bug reports whose version field value corresponds to an actual Eclipse release ranging between 3.0 and 4.10. To this end, we excluded 3,237 bugs with unspecified version field and 32,985 bugs corresponding to versions smaller

[1]This terminology is used by the Eclipse foundation to reflect a coordinated release effort including the Eclipse Platform and other Eclipse projects. See https://wiki.eclipse.org/Simultaneous_Release.

[2]https://www.eclipse.org/eclipse/development/plans/eclipse_project_plan_4_2.xml

[3]https://bugzilla.readthedocs.io/en/latest/api/core/v1/bug.html#search-bugs

than 3.0 or larger than 4.10. We did not consider release 4.11 because at the time of analysis (March 2019) the full dataset was not yet available for extraction.

We restricted ourselves to values that actually correspond to valid Eclipse releases; e.g., the valid version values of the 4.7 release found in our dataset were 4.7, 4.7.1, 4.7.1a, 4.7.2, 4.7.3 and 4.7.0 Oxygen. From the remaining bugs, we excluded 2,565 bugs that corresponded to versions that are not listed in the official releases of Eclipse, i.e., 3.9, 3.10, 4.0 and 4.1. Our final dataset (including releases 3.0→3.8, 4.2→4.10) consists of 138,445 bug reports.

For the remainder of the analysis, we partitioned all reported bugs into groups according to their major release number as we focus on the major Eclipse releases. For example, group 4.7 contains all reported bugs whose version field prefix is 4.7. For the specific case of the "dual release" of Eclipse 3.8 and 4.2 (that both have the same release date), a single group, 3.8/4.2 was created.

The aforementioned processing steps mitigate several threats that could bias the results of our study. Additional factors that can bias our results are explained by Tu et al. [36]. The misuse of issue tracking data may threaten the validity of findings because the values of issue fields (severity, priority, component, version, etc.) often change over time. They recommend researchers that rely on such data to have a full understanding of their application scenarios, the origin and change of the data, and the influential issue fields to manage data leakage risks.

In the context of our work, we examined the threats related to the bug report fields *version*, *resolved*, *fixed,* and *assignee*. The *version* field is used to indicate the version(s) affected by the bug report and usually contains the numbers or names of the releases. Examining the bugs that changed the *version* field during the bug fixing cycle, we found 4,063 bugs that were reaffected to different releases throughout their history, out of which 3,621 bugs that were reaffected to different major releases. We handle such bugs by considering them only for the last major release they affected as including the same bugs in multiple releases would bias the results for our pre-release analyses. From these bugs, only 98 (out of 109,367 resolved bugs) are resolved in multiple major Eclipse releases, thus the impact on our analyses is minimal.

A reported bug is marked as *resolved* and the resolution date is tracked by locating in the bug history a modification of the status field to the value RESOLVED. Similarly, a *resolved* bug is marked as *fixed* and the fixing date is tracked by locating the presence of the *last* FIXED status in the bug history. If the *resolved* status is modified multiple times, the *last* date that the bug was marked as RESOLVED will be registered. We opt for this strategy as the presence of multiple resolutions of the same bug implies that resolutions prior to the last one were not satisfactory.

A reported bug is marked as *assigned* and the assignment date is tracked by checking for the presence of the ASSIGNED value in the status field of the bug history. In case of multiple reassignments, we rely on the *first* recorded assignment date, reflecting the moment when the bug was triaged for the first time. The Eclipse community has also been assigning bugs using an alternative process since 2009[4]: it is possible to use an "assigned to" task without using the status field. This assignment method always assigns bugs to an email address in the form [component_name]-triaged@eclipse.org. We identified and marked as *assigned* 4,811 bugs corresponding to this case.

D. Proposed Bug Handling Metrics

We introduce the notation B_{report} to refer to the set of all reported bugs that we considered for our analysis. Similarly, B_{assign} refers to the subset of all assigned bugs, B_{resolve} to all resolved bugs, and B_{fix} to all fixed bugs. A superscript notation allows to restrain these sets to bugs targeting a specific release. For example, $B^{4.7}_{\text{resolve}}$ contains all *resolved* bugs for which the version field refers to 4.7.

Since our empirical analysis aims to relate bug handling activity to release dates, we need to compare the dates corresponding to specific activities against these release dates. To do so, we define the following functions:

- $D_{\text{report}} : B_{\text{report}} \rightarrow Date$ returns the creation date of a bug report.
- $D_{\text{assign}} : B_{\text{assign}} \rightarrow Date$ returns the *first* date the bug report status field has been set to ASSIGNED.
- $D_{\text{resolve}} : B_{\text{resolve}} \rightarrow Date$ returns the *last* date the bug report status field has been set to RESOLVED.
- $D_{\text{fix}} : B_{\text{fix}} \rightarrow Date$ returns the *last* date the bug report status field has been set to RESOLVED with value FIXED for the resolution field.

Using these functions, we define bug **triaging time** and bug **fixing time** as:

$$\forall b \in B_{\text{assign}} : T_{\text{triage}}(b) = D_{\text{assign}}(b) - D_{\text{report}}(b)$$

$$\forall b \in B_{\text{fix}} : T_{\text{fix}}(b) = D_{\text{fix}}(b) - D_{\text{report}}(b)$$

Let $d =]d_1, d_2]$ be a date range. We define bug **resolution rate** *ResRate* as the proportion of reported bugs that have been resolved in the considered date range. We also define two different **fixing rate** metrics. $FixRate_{\text{report}}$, taken from [24], computes the ratio of fixed over *reported* bugs. It relies on an implicit assumption that all reported bugs will eventually reach the RESOLVED status at some point in their lifecycle. This is not always the case based on our process mining results (Fig. 1) and the observations of [37]: many bugs are CLOSED without ever reaching RESOLVED status. We therefore define a variant metric $FixRate_{\text{resolve}}$ as the ratio of fixed over *resolved* bugs:

$$ResRate(d) = \frac{|B_{\text{resolve}}(d)|}{|B_{\text{report}}(d)|}$$

$$FixRate_{\text{report}}(d) = \frac{|B_{\text{fix}}(d)|}{|B_{\text{report}}(d)|} \quad FixRate_{\text{resolve}}(d) = \frac{|B_{\text{fix}}(d)|}{|B_{\text{resolve}}(d)|}$$

[4]https://wiki.eclipse.org/Platform_UI/Bug_Triage

Example. The following example illustrates these three metrics. Suppose 30 bugs are *reported* during date range d, of which 20 bugs are *resolved* and 12 of those resolved bugs are actually *fixed*. Then resolution rate is $ResRate = \frac{20}{30} = 0.66$, and fixing rates are $FixRate_{\text{report}} = \frac{12}{30} = 0.4$ and $FixRate_{\text{resolve}} = \frac{12}{20} = 0.6$.

E. Separation before and after Eclipse releases

Let $R = \{3.0,\ 3.1,\ 3.2,\ 3.3,\ 3.4,\ 3.5,\ 3.6,\ 3.7,\ 3.8/4.2,\ 4.3,\ 4.4,\ 4.5,\ 4.6,\ 4.7,\ 4.8,\ 4.9.\ 4.10\}$ be the ordered set of considered Eclipse releases. We define the following functions:

- date: $R \rightarrow Date$ returns the date of a given release
- prev: $R/\{3.0\} \rightarrow R$ returns its previous release
- next: $R/\{4.10\} \rightarrow R$ returns its next release

Given a release $r \in R$, we focus our analysis on **only those bugs that target this release**, based on the value of their *version* field in the bug report. We study the period *before* and *after* the release by restricting the date range to the dates between two successive releases. For example, by bugs reported *after* release $r = 4.5$ we mean all bugs targeting that release, and being created in the date range $[\text{date}(4.5),\text{date}(4.6)[$. Similarly, the period *before* release 4.5 considers the date range $[\text{date}(4.4),\text{date}(4.5)[$.

We introduce a shorthand notation for the fixing and resolution rates before and after each release. For example, the resolution rates before and after a release r are defined as follows, **restricting to only those bugs targeting** r:

$$ResRate^{\text{before}}(r) = ResRate([\text{date(prev(r))},\text{date(r)}[)$$
$$ResRate^{\text{after}}(r) = ResRate([\text{date(r)},\text{date(next(r))}[)$$

We similarly restrict B^{report}, B^{fix}, B^{assign} and B^{resolve}. For example,

$$B^{\text{before}}_{\text{report}}(r) = \{b \in B^{r}_{\text{report}} \mid D_{\text{report}}(b) \in [\text{date(prev(r))},\text{date(r)}[\ \}$$
$$B^{\text{after}}_{\text{report}}(r) = \{b \in B^{r}_{\text{report}} \mid D_{\text{report}}(b) \in [\text{date(r)},\text{date(next(r))}[\ \}$$

F. Preliminary Analysis

For each considered release, we computed the number of reported, resolved, fixed and assigned bugs targeting this release, without any restriction on the date range. Fig. 2 shows these statistics and reveals that the number of reported bugs targeting a given release (blue line) is monotonically decreasing all along the 3.x range of releases. Starting from release 3.8/4.2, the number of bug reports appears to become stable. For the quarterly releases starting from 4.9, the numbers decrease again.

Fig. 3 shows the corresponding evolution of *ResRate*, $FixRate_{\text{report}}$ and $FixRate_{\text{resolve}}$. We observe a *decreasing* resolution rate all along the Eclipse releases, while $FixRate_{\text{resolve}}$ is increasing and $FixRate_{\text{report}}$ tends to be stable across releases. For the 3.x release range, *ResRate* decreases from 0.96 to 0.64, while $FixRate_{\text{resolve}}$ increases from 0.50 to 0.76. Starting from release 3.6, $FixRate_{\text{resolve}}$ is consistently higher than *ResRate*. We assume that this change in behaviour has to do with the practice of "resolving" a bug by giving it the `LATER` or `REMIND` status in the

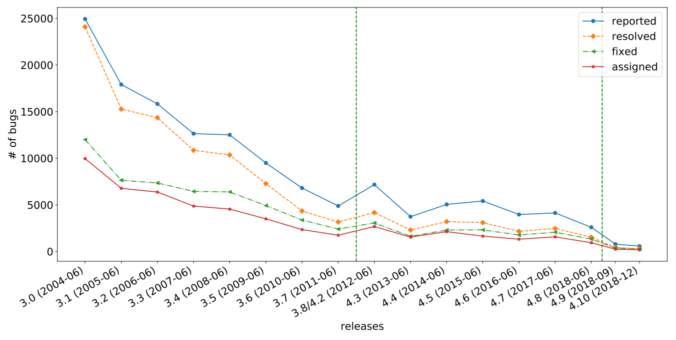


Fig. 2: Number of bugs targeting a specific release. The vertical dashed lines indicate the switch from Eclipse 3.x to 4.x, and from an annual to a quarterly cycle after 4.8.

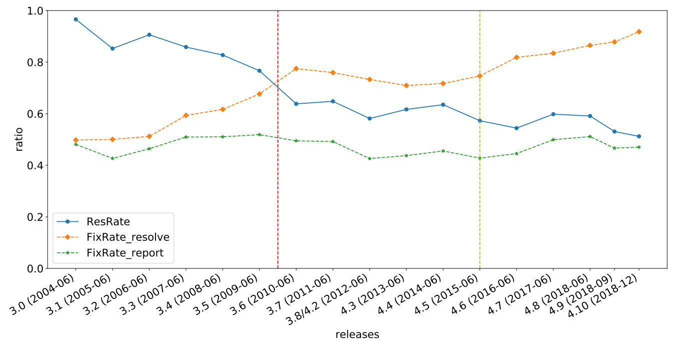


Fig. 3: Resolution and fixing rates per targeted Eclipse release.

resolution field, corresponding to the desire to postpone the bug resolution.[5] This practice was fairly common early on in the 3.x release range, but gradually declined and is no longer used since release 3.6 (red vertical line). By analyzing the delays of a follow-up resolution of the REMIND/LATER bugs (3479 bugs), we found that most of them lingered for more than 3 years before getting their final resolution.

For the 4.x annual release range we observe a stability in the rates up until release 4.5, after which *ResRate* continues to decrease and $FixRate_{\text{resolve}}$ continues to increase. We assume that this change is caused by the introduction of an Automated Error Reporting Client (called AERI)[6] since June 2015 (yellow vertical line).

Fig. 4 compares the number of bugs reported before and after each release: $\mid B^{\text{before}}_{\text{report}}(r) \mid$ and $\mid B^{\text{after}}_{\text{report}}(r) \mid$. As for Fig. 2 we observe a monotonic decrease in number of reported bugs for the 3.x release range, with a ratio $\frac{|B^{\text{after}}_{\text{report}}(r)|}{|B^{\text{before}}_{\text{report}}(r)|}$ between 0.38 and 0.66. This ratio increases to the range from 0.75 to 1.07 for the 4.x release with the exception of release 4.8 and 4.10. The exception for 4.8 is possibly caused by the quarterly release policy starting at 4.9 and for 4.10 because it is newly released.

[5]See https://www.eclipsezone.com/eclipse/forums/t83053.html
[6]See https://www.eclipse.org/community/eclipse_newsletter/2016/july/article3.php

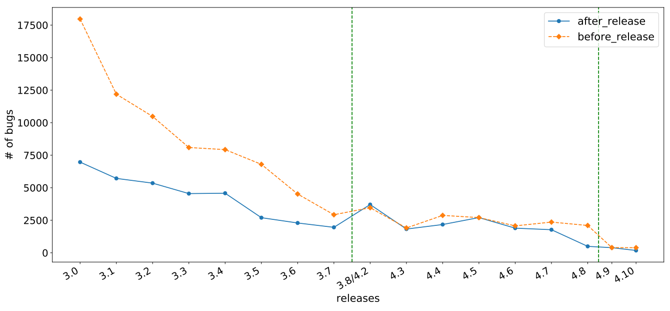


Fig. 4: Reported bugs before and after each release.

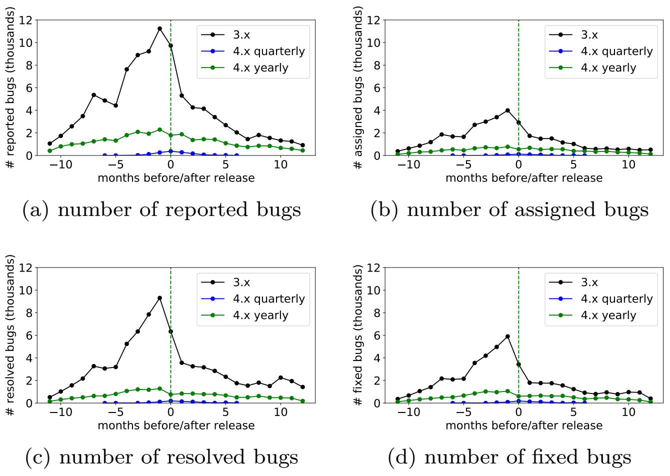
IV. Results

This section reports quantitative evidence for the research questions. All statistical hypothesis tests use a significance level $\alpha = 0.05$, rejecting the null hypothesis for $p < 0.05$. When computing the effect size we use Cliff's delta [38], [39], and interpret the results using [40].

RQ_0 ***How does the number of bugs differ before and after each release?***

Fig. 5 shows, before and after each release, the monthly evolution (i.e., consecutive 30-day time periods) of the number of reported, assigned, resolved and fixed bugs, respectively, aggregated per release series as the sum of bugs. For the 3.x series, we generally observe an important increasing trend during the first eleven months before the release date, followed by a decreasing trend starting from one month before the upcoming release deadline (probably due to imminent deadline pressure) until the date of the next release. For the 4.x series, the results are similar, but much less pronounced.

To test the difference between the number of bugs before and after the release we use a Wilcoxon rank sum test that replaces the Mann–Whitney U test when the observations are paired. The test makes no assumptions about the distributions, making it appropriate for our analyses since the values in our dataset are not normally distributed. The null hypothesis H_0 states that the distribution of the monthly number of bugs *before* the release is not different from the respective distribution *after* the release. H_0 could be rejected with large effect size for the populations of reported ($p = 0.006$, $d = 0.61$), assigned ($p = 0.006$, $d = 0.61$) and fixed ($p = 0.011$, $d = 0.56$) bugs, respectively, but only for the 3.x release series. For the population of *resolved* bugs for the 3.x release series we could not reject H_0. For the 4.x series of annual and quarterly releases, we could not reject H_0 except for the number of reported bugs for the annual releases, but the effect size is medium.



(a) number of reported bugs (b) number of assigned bugs (c) number of resolved bugs (d) number of fixed bugs

Fig. 5: Monthly evolution of the number of bugs before/after a release, aggregated by release series (y-axis is measured in thousands of bugs).

> For Eclipse 3.x releases, more bugs are reported, assigned and fixed *before* the upcoming release than *after* the release, indicating that more effort is spent for handling bugs before the upcoming release. For Eclipse 4.x no difference is observed, suggesting an increasingly balanced bug handling workload before and after the release date.

RQ_1 ***How do the bug resolution and fixing rates evolve over releases, and how do they differ before and after each release?***

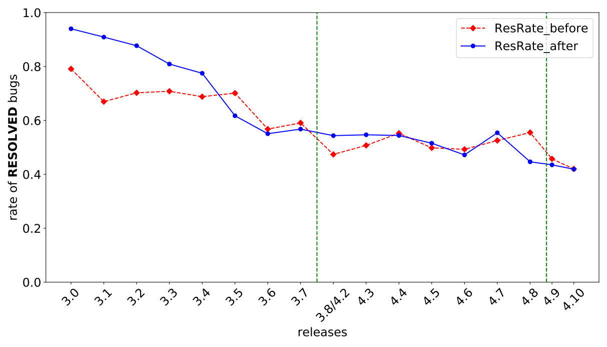
First, we compute the resolution rate before and after each release date. Fig. 6a shows a decreasing trend of *ResRate* (y-axis) over the different releases (x-axis). A linear regression analysis confirms this trend for $ResRate_{\text{before}}$ ($R^2 = 0.785$) and $ResRate_{\text{after}}$ ($R^2 = 0.863$).

Using a Wilcoxon rank sum test we verified the null hypothesis $H0_{1r}$ that there is no statistical difference between the resolution rates before and after the release. We could not reject $H0_{1r}$ ($p = 0.85$), i.e., the resolution rates before and after each release are similar. This is visually confirmed starting from release 3.5. The reason of decreasing resolution rates specifically after release 3.5 is that the Eclipse community has decided in 2007 to stop using the resolution statuses `LATER` and `REMIND` as they gave rise to bugs that remained unresolved for too long[7]. For the 4.x series, the resolution rates *before* and *after* fluctuate somewhere between 0.4 and 0.6. This signifies that around 1 out of 2 bugs do not get resolved.

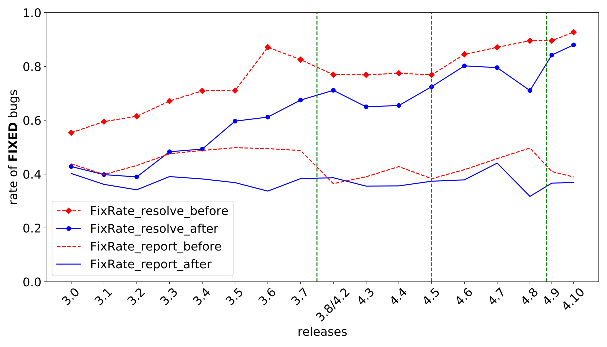
> The resolution rate tends to decrease over releases, but there is no significant difference in resolution rate before and after each release. One out of two bugs does not get resolved in the 4.x release range.

Next, we compute the fixing rates ($FixRate_{\text{report}}$ and $FixRate_{\text{resolve}}$) before and after each release date. Fig. 6b shows that the fixing rates *before* each release (horizontal

[7]https://bugs.eclipse.org/bugs/show_bug.cgi?id=157642



(a) Resolution rate evolution



(b) Fixing rate evolution

Fig. 6: Evolution of the resolution and fixing rate before and after each release. (The red vertical line corresponds to the introduction of the AERI error reporting tool.)

red lines) are higher than the fixing rates *after* the release (blue lines). Using the Wilcoxon rank sum we test the null hypothesis $H0_{1f}$ that there is no statistically significant difference between fixing rates before and after a release. $H0_1^f$ was rejected for $FixRate_{\text{report}}$ ($p = 6 \times 10^{-5}$) with large effect size ($d = 0.8$). $H0_1^f$ was rejected for $FixRate_{\text{resolve}}$ ($p = 0.02$) with medium effect size ($d = 0.47$). This shows that the bug fixing rate before the release date is higher than after that date.

It is interesting to note the difference in behaviour between $FixRate_{\text{report}}$ and $FixRate_{\text{resolve}}$. Consistent with what we already saw in Fig. 3, $FixRate_{\text{report}}$ remains stable over time and $FixRate_{\text{resolve}}$ increases over time. A linear regression analysis confirms this positive trend for $FixRate_{\text{resolve}}^{\text{before}}$ ($R^2 = 0.816$) and $FixRate_{\text{resolve}}^{\text{after}}$ ($R^2 = 0.899$). As in Fig. 3 we observe that $FixRate_{\text{resolve}}$ is improving faster since release 4.5, probably because of the introduction and continued use of the AERI error reporting tool.

> The fixing rate before a release is higher than after a release. The rate of fixed bugs over resolved bugs is increasing over time. The introduction of an automatic error reporting tool coincides with a further improvement of bug fixing rate.

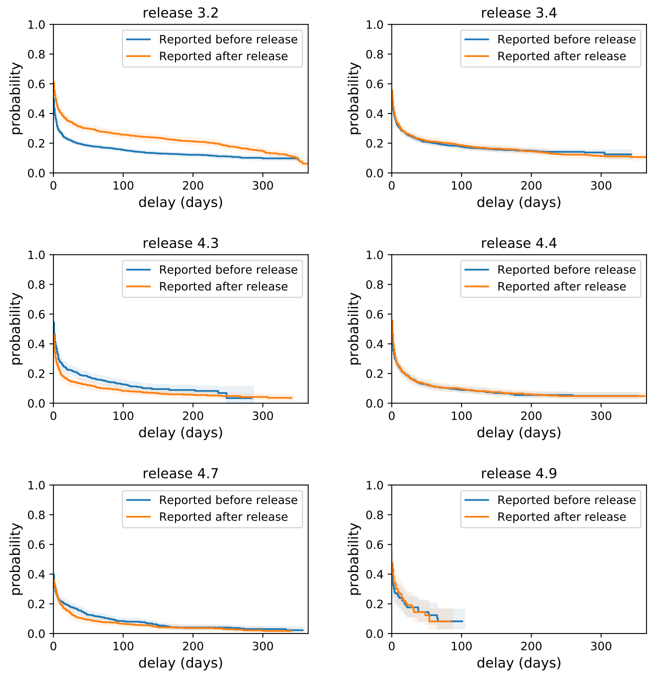


Fig. 7: Kaplan-Meier survival curve with 95% confidence interval (indicated by the shaded areas) for triaging time before and after each release.

RQ_2 *How do the bug triaging and fixing time differ before and after each release?*

For each release, we compute the bug triaging time $T_{\text{triage}}(b)$ for each assigned bug b before and after the release date. Triaging time *before* a release is computed for bugs targeting that release that are reported and assigned in the date range [date(prev(r)),date(r)[, while *after* the release we use the range [date(r),date(next(r))[. Similarly, we calculated the bug fixing time before and after each release for bugs targeting that release that are reported and fixed in the respective date ranges.

For both time measurements, we use the technique of survival analysis [41] to model the expected time duration until the occurrence of an event of interest. In the triaging case, the event of interest is bug assignment and the duration to the event is computed from the creation date of the bug until the first assignment date. In the fixing case, the event of interest is a bug being fixed and the duration to this event is calculated from the creation date of the bug until the last fix date.

Triaging time results: Fig. 7 shows a selection of Kaplan-Meier survival curves for specific releases, together with their confidence intervals, before and after the release date.[8] We generally observe that triaging time before a release tends to be lower than triaging time after that release. We also observe that the difference between the survival curves tends to decrease over successive releases;

[8]The full set of survival curves can be found in [29]

Eclipse 3.x			Eclipse 4.x (annual)			Eclipse 4.x (quarterly)		
rel.	$H0_2^t$	$H0_2^f$	rel.	$H0_2^t$	$H0_2^f$	rel.	$H0_2^t$	$H0_2^f$
3.0	R	R	3.8/4.2	R	R	4.9	–	–
3.1	R	R	4.3	R	–	4.10	–	R
3.2	R	R	4.4	–	R			
3.3	R	R	4.5	R	R			
3.4	–	R	4.6	R	–			
3.5	R	R	4.7	–	R			
3.6	R	R	4.8	–	–			
3.7	R	R						

TABLE I: Logrank test for difference between the survival distributions of **triaging time** (resp. **fixing time**) *before* and *after* each release. R indicates that $H0$ is rejected.

for some releases, like 4.7, the difference is very low or absent. To statistically confirm the difference in triaging time before and after each release, we use a logrank test [42]. The null hypothesis $H0_2^t$ states that there is no difference between the triaging time survival distributions before and after a release. Table I reports for which releases $H0_2^t$ can be rejected. We can reject $H0_2^t$ for all releases except 3.4, 4.4, and from 4.7 till 4.10. Especially for the latest releases, including all quarterly releases, we cannot observe any statistically significant difference.

Fixing time results: The survival curves for bug fixing time (not shown due to lack of space) reveal that for the 3.x release range and most of the annual 4.x releases, it takes less time to fix a bug before compared to after a release. With a statistical logrank test we verify hypothesis $H0_2^f$ that there is no difference between the fixing time survival distributions before and after a release. We cannot reject $H0_2^f$ for releases 4.3, 4.6, 4.8 and 4.9. The difference between 4.9 and 4.10 is explained by the shorter timespan of bugs in release 4.10 as it is the newest release in our dataset; this explains why bugs appear to be fixed faster for 4.10 after the release.

> Bugs tend to get triaged and fixed faster before a release than after it. The transition from an annual to a quarterly release policy appears to have been beneficial, since no statistical difference in triaging time can be observed before and after such releases.

RQ_3 *How do the bug triaging and fixing time change when approaching a release deadline?*

For each release, we compute triaging time $T_{\text{triage}}(b)$ for each bug b, by analyzing the population of reported bugs that were actually `ASSIGNED` in a particular date range. We compute bug fixing time $T_{\text{fix}}(b)$ in a similar way for `FIXED` bugs. We group our results in two date ranges based on when the assignment or fixing took place: 1) during the first 9 months after the *current* release (henceforth called *early period*) and 2) during the last 3 months before the *next* release (henceforth called *pressure period*). We consider the period of 9 months after the current release for quarterly releases because the development teams start early working on the releases following the next release. Release 4.10 is excluded from this analysis as its activity duration is not sufficient to perform the experiment.

Firstly, we investigate the bug activity of the current and next release for the pressure period (blue boxes in Fig. 8). We visually observe that during the pressure period, bugs for the next release are triaged (left column) and fixed (right column) *faster* than for the current release. This can be explained by an increasing deadline pressure, requiring developers to prioritise the bugs of the next release and triage and fix them fast to deliver the release with fewer bugs. Mann-Whitney U tests (see Table II) confirm that the null hypothesis $H0_{3a}^t$, stating that the time to *triage* bugs of the current and next new release is not different from in the pressure period, can be rejected for all releases except 4.8. We found a *negligible* effect size for release 4.4, a *small* effect for 8 releases, *medium* for release 4.5 and *large* for 4 releases. Similarly, we confirm that the null hypothesis $H0_{3a}^f$, i.e., the time to *fix* bugs of the current and next new release is the same in the pressure period, can be rejected for all releases (see Table II). We found a *small* effect size for release 4.8, *medium* for releases 3.8/4.2, 4.6 and 4.9, and *large* for 11 releases.

Secondly, for both the *current* and the *next* release, we investigate the differences in triaging and fixing time between the early period (red boxes in Fig. 8) and the pressure period (blue boxes).

For the *current* release, we observe that the triaging and fixing are longer in the pressure period than in the early period. These results indicate that bugs triaged and fixed in the pressure period have been open for a long time and that developers tend to triage many bugs that had lived for

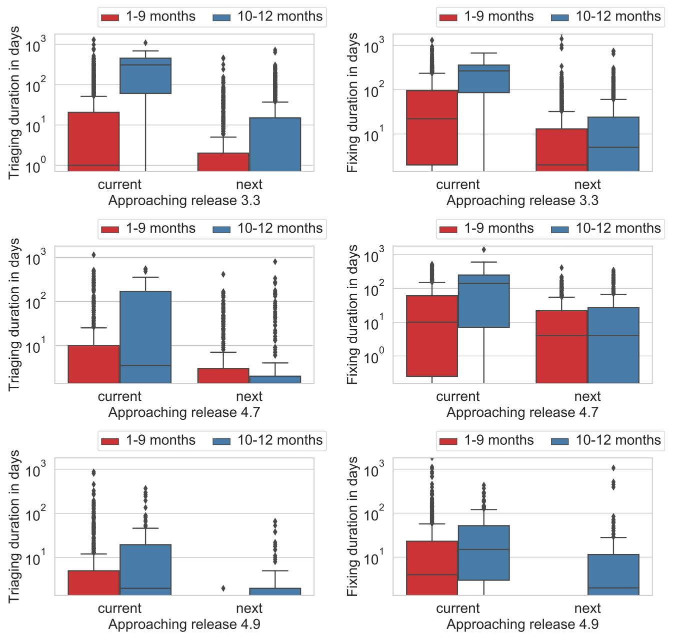


Fig. 8: Triaging time distributions (left figures) and fixing time distributions (right) when approaching next release.

	triaging			fixing		
rel.	$H0^t_{3a}$	$H0^t_{3b}$	$H0^t_{3c}$	$H0^f_{3a}$	$H0^f_{3b}$	$H0^f_{3c}$
Eclipse 3.x						
3.1	**S**	**S**	N	**L**	**L**	N
3.2	**L**	**M**	N	**L**	**M**	**S**
3.3	**L**	**L**	**S**	**L**	**L**	N
3.4	**S**	**S**	–	**L**	**M**	N
3.5	**S**	–	–	**L**	**S**	N
3.6	**S**	**S**	N	**L**	**L**	N
3.7	**L**	**M**	N	**L**	**M**	N
Eclipse 4.x (annual)						
3.8/4.2	**L**	**M**	–	**M**	**S**	N
4.3	**S**	N	–	**L**	**M**	N
4.4	N	-	N	**L**	**M**	**S**
4.5	**M**	**S**	N	**L**	**M**	N
4.6	**S**	N	–	**M**	**S**	N
4.7	**S**	**S**	–	**L**	**M**	–
4.8	–	–	N	**S**	**S**	–
Eclipse 4.x (quarterly)						
4.9	**S**	**S**	–	**M**	**S**	**L**

TABLE II: Mann-Whitney U test for **triaging** and **fixing** time when approaching a release deadline. Whenever a significant difference is found, cell values summarize the effect size: N(egligible), **S**(mall), **M**(edium), **L**(arge); – indicates that $H0$ could not be rejected.

a long time in the current release before releasing the next one. Mann-Whitney U tests (see Table II) confirm that the null hypothesis $H0^t_{3b}$, stating that the *triaging time* of the current release in the early period is not different from the pressure period, can be rejected for all releases except 3.5, 4.4 and 4.8. We found a *negligible* effect size for 4.3 and 4.6, a *small* effect for 6 releases, *medium* for 3 releases and *large* for release 3.3. The respective results for the *fixing time* show that the null hypothesis $H0^f_{3b}$ can be rejected for all releases. The effect sizes range between *small* and *large*. Therefore, for most releases, bugs of the current release took longer to triage or fix during the pressure period compared to the early period.

For the *next* release, we do not observe a difference between the *triaging* time of the bugs in the pressure period and early period. Bugs for the next release tend to be triaged in the same way regardless of the considered period. Developers try to triage these bugs as soon as possible. Mann-Whitney U tests (Table II) confirmed that the null hypothesis $H0^t_{3c}$, stating that *triaging time* of the next release in the early period is not different from in the pressure period, can be rejected for the 3.x series except for 3.4 and 3.5. We cannot reject $H0^t_{3c}$ for any of the 4.x releases except for 4.4, 4.5 and 4.8. Even for those releases where $H0^t_{3c}$ can be rejected, the effect size is *negligible* or *small* (for 3.3). This implies that there is little or no difference between the triaging time of the next release in the early period and in the pressure period.

For the *fixing* time, we visually observed a longer time during the pressure period. The null hypothesis $H0^f_{3c}$, stating that bug fixing time of the next release in the early period is not different from in the pressure period, can be rejected for all releases except 4.7 and 4.8. For all the releases where $H0^f_{3c}$ can be rejected, the effect size is *negligible* except for 3.2, 4.4 and 4.9. This implies that, with the exception of those releases, there is very little difference in the time to fix bugs of the next release during the early period than during the pressure period.

Bug triaging time increases as the next release is approaching. It takes less time to triage and fix bugs during the pressure period of the next release compared to the current release. Bugs of the current release that are triaged and fixed in the pressure period have been open for longer compared to the respective ones in the early period. The fact that the next release is approaching does not affect the triaging and fixing time of bugs for the next release.

V. Discussion

Changes in the bug fixing process can have a direct impact on the efficiency of bug handling activity. As we have shown in Section III-F and RQ1 in Section IV, the Eclipse community stopped using resolution statuses indicating that the bug still needed futher processing (`LATER` or `REMIND` status). This decision naturally caused a decrease in the resolution rate, but the fixing rate increased. The introduction of an Automated Error Reporting client (AERI) was also considered as beneficial by the Eclipse community. AERI facilitates reporting errors as users do not need to create Bugzilla entries; such entries are handled by the Eclipse community based on the reports they receive. In turn, users can provide comments with their reports which are helpful when fixing bugs; according to an AERI report [43] commented reports are more than twice as likely to be fixed compared to the ones without a user comment. Similarly, our empirical analysis has confirmed the positive effects of AERI on the bug fixing rate.

The bug fixing rate has been studied by Zou et al. [28] who analysed the characteristics of bug fixing rates in Eclipse and Mozilla. They measured bug fixing rate, defined as the number of fixed over closed bugs, at a yearly basis from 2001 to 2014, and found that the rate increased from 0.49 to 0.77 for Eclipse and 0.32 to 0.72 for Mozilla. They took into account all issues in all Eclipse projects (not only the Eclipse core) without distinguishing between real bugs and enhancements. For the fixing rate they considered only closed bugs, while we found many Eclipse bugs to be `FIXED` without ever being `CLOSED` (see Fig. 1). While we measured the fixing rate of real bugs in the Eclipse core at a release basis, our results align with their reported fixing rates (see Fig. 3). They found that the overall fixing rate of Mozilla (0.44) is much lower than Eclipse (0.66) and report that more bugs tend to be invalid for Mozilla than for Eclipse. Khomh et al. [23], [24] studied bug handling activity in Mozilla Firefox. They analysed two years of bug activity, whereas we performed a longer study considering 15 years of bug activity for Eclipse. Khomh et al. reported that the transition to a rapid release cycle for Mozilla Firefox has led to shorter bug fixing time, but less bugs are triaged and fixed in a timely fashion. Our results for Eclipse partially align as we also found less bugs to be triaged and fixed. In contrast to Firefox, however, we observed that the fixing rate for Eclipse has improved after transitioning to shorter release cycles (see Section III-F). A possible reason for this difference is that developers from the Firefox community stated that they were not given enough time to prepare for the transition to smaller release cycles [24], whereas the Eclipse community has been working towards this transition for over a year [44] and has started to introduce intermediate (quarterly) "update" releases since Eclipse 4.6 in 2016. Our results highlight the importance of well preparing the transition from a traditional to a rapid release policy so as the community to become more effective in bug handling activities. Other projects can benefit from the lessons learned about how both Firefox and Eclipse carried out such a transition. While preparing a release, various nightly and integration builds are made available to developers only for testing purposes. However, milestone and release candidates are intended for adoption and testing by users before the scheduled annual release [45]. This might explain the high number of reports before an official release.

Our results show that developers focus more on triaging and fixing bugs of the next release than ones impacting users in the current release. Moreover, bugs of the current release that are triaged in the pressure period have been open for a long time. This can happen because long-lived and possibly complex bugs are planned to be fixed for the next release. The severity of a reported bug is a critical factor in deciding how soon it needs to be fixed [46].

VI. Threats to Validity

Threats to *construct validity* in our study concern the bug assignment identification and multiple instances of an activity. We minimized the first threat by identifying bug assignments based on both the status and existing practices (see Section III-C). We mitigated the second threat by considering the date of the first assignment, and the date of the last resolution and fixing activities.

A threat to *internal validity* is that our empirical analysis relies on Bugzilla bug reports. We mitigate this threat by verifying that all Eclipse bugs are managed in Bugzilla; even the bugs submitted using AERI are stored in Bugzilla. Another threat stems from bugs not being tagged with a valid release; we excluded such bugs as it is not feasible to properly address such cases. Finally, bugs can be tagged with different major versions throughout their history; we quantified the possible bias from such cases and found that only 98 bugs can affect our analysis; the bias they introduce is insufficient to alter our findings.

Regarding *external validity*, we cannot generalize our results as we only analyzed a single case study of Eclipse. While the followed methodology is applicable to other systems, the obtained findings are not generalizable. Smaller and less mature projects are likely to reveal other evolutionary characteristics in their bug fixing behavior. Even for Eclipse itself, the findings are only based on the core projects that have a large number of bugs and an active developer community. The findings may differ for smaller and/or peripheral projects within Eclipse that have smaller communities.

We mitigate threats to *reliability validity* by providing a publicly available replication package [29] and a detailed description of the followed methodology in Section III.

VII. Conclusion

We conducted an empirical study of the bug handling activity in the Eclipse core projects over a 15-year lifetime. We analysed 16 annual and two quarterly releases, allowing us to study the transition to a rapid release cycle in autumn 2018. We evaluated the evolution of bug triaging time, bug fixing time, bug resolution rate and bug fixing rate. We compared these metrics *before* and *after* each release date and between release ranges.

Over time, the number of reported bugs per release is decreasing, suggesting that Eclipse and its bug handling process are mature and of high quality. More bugs are reported, assigned and fixed before each release deadline than after it. This difference is becoming smaller with newer releases, mostly because there are less reported bugs before each rapid release.

While the bug resolution rate is decreasing over time to rather low values, the fixing rate is becoming very high (close to or above 90%). This improved efficiency seems to be due to a combination of a well-managed bug handling policy and the recent introduction (in June 2015) of an automated error reporting tool.

When approaching a release deadline, effort is shifting from bug triaging to fixing, and priority is given to fixing bugs of the next release as opposed to handling bugs of the current release.

Overall, we did not observe any negative effect of the transition from an annual to a quarterly release cycle. On the contrary, some observed differences in triaging and fixing times, and rates before and after releases are no longer present.

Based on our analysis, we believe that Eclipse is handling its bug activity very well and has benefited from the transition to a rapid release cycle. It remains to be seen if even faster release cycles will continue to yield benefits or on the contrary will have negative consequences.

Acknowledgment

This research was partially supported by the FNRS and FWO under the Excellence of Science project 30446992 SECO-ASSIST, the FRQ-FNRS collaborative research project R.60.04.18.F SECO-Health and Région Hauts-de-France.

References

[1] R. K. Saha, S. Khurshid, and D. E. Perry, "Understanding the triaging and fixing processes of long lived bugs," *Information and software technology*, vol. 65, pp. 114–128, 2015.

[2] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?," in *International Workshop on Mining Software Repositories*, IEEE, 2007.

[3] L. D. Panjer, "Predicting Eclipse bug lifetimes," in *International Workshop on Mining Software Repositories*, IEEE Computer Society, 2007.

[4] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *International Workshop on Recommendation Systems for Software Engineering*, pp. 52–56, ACM, 2010.

[5] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *International Conference on Predictive Models in Software Engineering*, ACM, 2011.

[6] A. Kumar and A. Gupta, "Evolution of developer social network and its impact on bug fixing process," in *India Software Engineering Conference*, pp. 63–72, ACM, 2013.

[7] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *International Conference on Automated Software Engineering (ASE)*, pp. 34–43, ACM, 2007.

[8] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study on factors impacting bug fixing time," in *Working Conference on Reverse Engineering*, pp. 225–234, IEEE, 2012.

[9] H. Hu, H. Zhang, J. Xuan, and W. Sun, "Effective bug triage based on historical bug-fix information," in *International Symposium on Software Reliability Engineering (ISSRE)*, pp. 122–132, IEEE, 2014.

[10] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: an empirical study of commercial software projects," in *International Conference on Software Engineering (ICSE)*, pp. 1042–1051, IEEE Press, 2013.

[11] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, "An empirical analysis of bug reports and bug fixing in open source Android apps," in *European Conference on Software Maintenance and Reengineering*, pp. 133–143, 2013.

[12] M. Gupta, "Improving software maintenance using process mining and predictive analytics," in *International Conference on Software Maintenance and Evolution*, pp. 681–686, IEEE, 2017.

[13] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Joint European Software Engineering Conference and ACM SIGSOFT Symposium Foundations of Software Engineering (ESEC/FSE)*, pp. 111–120, ACM, 2009.

[14] J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, and X. Wu, "Towards effective bug triage with software data reduction techniques," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 1, pp. 264–280, 2015.

[15] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, "Automatic bug triage using semi-supervised text classification," in *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 209–214, 2010.

[16] J. Anvik, "Automating bug report assignment," in *International Conference on Software Engineering*, pp. 937–940, ACM, 2006.

[17] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *International Conference on Software Engineering*, pp. 361–370, ACM, 2006.

[18] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *ACM Symposium on Applied Computing (SAC)*, pp. 1767–1772, ACM, 2006.

[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 *Working Conference on Mining Software Repositories*, pp. 2–11, IEEE, 2013.

[20] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *Working Conference on Reverse Engineering (WCRE)*, pp. 72–81, IEEE, 2013.

[21] X. Xia, D. Lo, X. Wang, and B. Zhou, "Dual analysis for recommending developers to resolve bugs," *Journal of Software: Evolution and Process*, vol. 27, no. 3, pp. 195–220, 2015.

[22] D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P. J. Adams, I. Samoladas, and I. Stamelos, "Evaluating the quality of open source software," *Electronic Notes in Theoretical Computer Science*, vol. 233, pp. 5–28, 2009.

[23] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? An empirical case study of Mozilla Firefox," in *orking Conf. Mining Software Repositories*, pp. 179–188, IEEE, 2012.

[24] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, "Understanding the impact of rapid releases on software quality," *Empirical Software Engineering*, vol. 20, no. 2, pp. 336–373, 2015.

[25] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes," in *International Conference on Software Engineering*, pp. 332–341, IEEE, 2013.

[26] D. A. da Costa, S. McIntosh, U. Kulesza, and A. E. Hassan, "The impact of switching to a rapid release cycle on the integration delay of addressed issues - An empirical study of the Mozilla Firefox project," in *Working Conference on Mining Software Repositories*, pp. 374–385, IEEE, 2016.

[27] D. A. da Costa, S. McIntosh, C. Treude, U. Kulesza, and A. E. Hassan, "The impact of rapid release cycles on the integration delay of fixed issues," *Empirical Software Engineering*, pp. 1–70, 2018.

[28] W. Zou, X. Xia, W. Zhang, Z. Chen, and D. Lo, "An empirical study of bug fixing rate," in *Computer Software and Applications Conference (COMPSAC)*, pp. 254–263, IEEE, 2015.

[29] Z. A. Khalil, E. Constantinou, and T. Mens, "A longitudinal analysis of bug handling across Eclipse releases [replication package]." https://zenodo.org/record/3246948, June 2019.

[30] T. Mens, J. Fernandez-Ramil, and S. Degrandsart, "The evolution of Eclipse," in *International Conference on Software Maintenance*, pp. 386–395, Sep. 2008.

[31] J. Businge, A. Serebrenik, and M. van den Brand, "Survival of Eclipse third-party plug-ins," in *International Conference on Software Maintenance*, pp. 368–377, IEEE, 2012.

[32] Eclipse Foundation, "Eclipse bugzilla repository." https://bugs.eclipse.org, 2019.

[33] Eclipse Foundation, "Simultaneous release." https://wiki.eclipse.org/Simultaneous_Release, 2019.

[34] Fluxicon, "Disco process mining tool." https://fluxicon.com/disco/, 2019.

[35] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, "Are these bugs really "normal"?," in *Working Conference on Mining Software Repositories*, pp. 258–268, 2015.

[36] F. Tu, J. Zhu, Q. Zheng, and M. Zhou, "Be careful of when: an empirical study on time-related misuse of issue tracking data," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 307–318, ACM, 2018.

[37] B. Luijten, J. Visser, and A. Zaidman, "Assessment of issue handling efficiency," in *Working Conference on Mining Software Repositories*, pp. 94–97, IEEE, 2010.

[38] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychological Bulletin*, vol. 114, pp. 4994–509, November 1993.

[39] M. Hess and J. Kromrey, "Robust confidence intervals for effect sizes: A comparative study of Cohen's d and Cliff's delta under non-normality and heterogeneous variances," *American Educational Research Association*, 2004.

[40] M. R. Hess and J. D. Kromrey, "Robust confidence intervals for effect sizes: A comparative study of Cohen's d and Cliff's delta under non-normality and heterogeneous variances," in *annual meeting of the American Educational Research Association*, 2004.

[41] O. Aalen, O. Borgan, and H. Gjessing, *Survival and event history analysis: a process point of view.* Springer Science & Business Media, 2008.

[42] J. M. Bland and D. G. Altman, "The logrank test," *BMJ*, vol. 328, no. 7447, p. 1073, 2004.

[43] A. Sewe, "One year of automated error reporting." https://www.eclipse.org/community/eclipse_newsletter/2016/july/article3.php, July 2016.

[44] M. Barbero, “Simultaneous release brainstorming.” https://www.eclipse.org/lists/eclipse.org-planning-council/msg02927.html.

[45] Eclipse Foundation, “Eclipse development process.” https://www.eclipse.org/projects/dev_process/, 2019.

[46] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, “Predicting the severity of a reported bug,” in *Working Conference on Mining Software Repositories*, pp. 1–10, IEEE, 2010.