On the impact of release policies on bug handling activity: A case study of Eclipse

Zeinab Abou Khalil, Eleni Constantinou, Tom Mens, Laurence Duchien

On the impact of release policies on bug handling activity: A case study of Eclipse

Zeinab Abou Khalil[a,c,d], Eleni Constantinou[b], Tom Mens[a], Laurence Duchien[c,d]

[a] *Software Engineering Lab, University of Mons*
Avenue Maistriau 15, B-7000 Mons, Belgium
zeinab@aboukhalil@umons.ac.be – tom.mens@umons.ac.be
[b] *Eindhoven University of Technology*
Groene Loper 5, 5612 AE, Eindhoven, Netherlands – e.constantinou@tue.nl
[c] *Université de Lille*
Lille, France – laurence.duchien@univ-lille.fr
[d] *INRIA Lille – Nord Europe*

Abstract

Large software projects follow a continuous development process with regular releases during which bugs are handled. In recent years, many software projects shifted to rapid releases that reduce time-to-market and claim a faster delivery of fixed issues, but also have a shorter period to address bugs. To better understand the impact of rapid releases on bug handling activity, we empirically analyse successive releases of the Eclipse Core projects, focusing on the bug handling rates and durations as well as the feature freeze period. We study the impact of Eclipse's transition from a yearly to quarterly release cycle. We confirm our findings through feedback received from five Eclipse Core maintainers. Among others, our results reveal that Eclipse's bug handling process is becoming more stable over time, with a decreasing number of reported bugs before releases, an increasing bug fixing rate and an increasingly balanced bug handling workload before and after releases. The transition to a quarterly release cycle continued to improve bug handling. In addition, more effort is spent on bug fixing during the feature freeze period, while the bug handling rates do not differ between both periods.

Keywords: Bug handling process, Rapid release cycle, Feature freeze, Continuous software development, Software maintenance, Empirical software engineering

1. Introduction

Continuous software engineering is a common practice for large collaborative software projects [1]. Every major release provides a significant amount of new or modified functionality compared to the previous release. Developers strive to resolve as many bugs as possible before the next release deadline [2]. To do so, they follow a bug handling process and rely on dedicated collaborative bug tracking tools (such as Bugzilla and Mantis).

More and more large software projects are switching to a rapid release cycle [3] to reduce their time-to-market [4]. Rapid release policies might, however, negatively affect the number of bugs being handled, since releases are delivered more often, and the community may have less time to address unresolved bugs. A good preparation is, therefore, of paramount importance to increase the success of adopting a rapid release policy.

Another common practice of large software projects is to impose a *feature freeze* when approaching the next release deadline [5, 6]. During the feature freeze period that lasts until the release date, all work on adding new features is suspended, shifting the effort towards fixing bugs, and carrying out a series of test-and-fix iterations to improve quality and stability. Each such iteration results in a new so-called *release candidate*.

Following the Goal-Question-Metrics (GQM) approach [7], we study the evolution of Eclipse – a large and long-lived open source project – with the aim to analyze its bug handling process for the purpose of assessing the impact of rapid releases and feature freeze periods from the point of view of maintainers in the context of bug handling. This overall objective is divided into two main goals, each composed of two research questions that will guide the case study design and empirical analysis:

Goal 1: The first research goal aims to study if the transition to rapid releases resulted in less time for the community to handle bugs before the release, leading to potentially more post-release bugs in need of resolution. To do so, we analyze if the bug handling activity is different before and after each release, and whether and how this changed after the transition to rapid releases. Our investigation will be guided by two research questions:

RQ1.1: *How does the bug handling rate evolve across releases?* We empirically analyze if the bug handling rate increases for successive releases. We also investigate the bug handling rate and before and after each release, as well as if any notable difference could be observed for the rapid releases.

RQ1.2: *How does the bug handling time differ across releases?* Since maintainers strive to deliver project releases with as few bugs as possible, we study whether 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 these two periods in terms of days elapsed to triage and fix bugs.

Goal 2: The second research goal aims to study to which extent bug handling activity is affected by the presence of feature freezes, and whether the transition to rapid releases has lead to an observable difference as their shorter duration can potentially affect the bug handling activity. Our investigation will be guided by two research questions:
RQ2.1: *How does the feature freeze period impact bug handling rate?* This research question analyses the bug handling rate before and during the feature freeze period of each considered release, the effort spent in these periods, and whether all of this has changed for rapid releases.
RQ2.2: *How does the feature freeze period impact bug handling time?* This research question focuses on bug handling time before and during the feature freeze period of each considered release. We study whether bugs are indeed triaged and fixed faster during such feature freeze periods. We also study whether the triaging and fixing time of bugs targeting the current release increases during these periods, since maintainers may prefer to focus on bugs of the upcoming release rather than on those targeting the current release that has already been delivered.

This manuscript is an extension over previous work [8] that carried out a longitudinal case study of bug handling for the Eclipse Core projects over a 15-year period. To achieve this, we analyzed 17 consecutive major releases (from 3.0 till 4.10) and studied how the bug handling time and rate differ before and after an upcoming scheduled release; and how it evolves over time. We relied on four measurements to quantify bug handling: triaging and fixing time, and resolution and fixing rate. In contrast, the current manuscript focuses on a specific and important aspect in the evolution of Eclipse Core, namely how the transition to a rapid release policy during the 4.x series has influenced the bug handling process. To do so, we compare the bug handling activity of seven annual releases ($4.2 - 4.8$) against seven quarterly releases ($4.9 - 4.15$), based on a dataset of over 36K bug reports from Bugzilla. We follow a mixed-method approach, by quantitatively analysing the impact of

rapid releases on bug handling activity, and supporting this analysis with qualitative anecdotal evidence about the perceived benefit of the transition by consulting five Eclipse Core maintainers. In addition, we analyze the effect of the feature freeze period of each release on bug handling, and whether and how this has changed after the adoption of a rapid release policy. We additionally study to which account the bug severity plays a role in each research question as more severe bugs can be expected to be prioritized and thus handled more quickly [9]. This effect might be reduced in rapid releases where there may not be enough time to handle all pending bugs, potentially leading to an increased backlog for less severe bugs.

The remainder of this paper is structured as follows. Section 2 discusses the related work. Section 3 describes the experimental setup of the Eclipse Core case study. Section 4 presents a preliminary analysis on the bug handing process evolution in Eclipse. Section 5 and Section 6 present our research findings and Section 7 discusses these findings. Section 8 elaborates on the threats to validity of our analysis. Finally, Section 10 concludes and discusses avenues of future research.

2. Related Work

2.1. Short Release Cycles

Prior research studied the benefits and challenges of adopting rapid releases. Such releases are claimed to offer reduced time-to-market and faster user feedback [10]. End users may benefit from this because they get faster access to functionality improvements and security updates [11]. Moreover, Zimmermann [12] has shown that the adoption of a shorter release cycle has successfully managed to provide more stable versions, with less breaking changes, that are easier to upgrade. In our study, we will analyze the impact of rapid releases on the bug handling process.

Joshi et al. [13] introduced a publicly available dataset consisting of 994 open source projects on GitHub featuring rapid releases. This dataset, along with its documentation and scripts, is aimed to facilitate future empirical research in release engineering and agile software development. We did not use it in the current manuscript, because none of the projects fit our selection criteria of being very large, long-lived, having migrated recently to a fixed rapid release cycle, and having sufficient and reliable bug history data available on Bugzilla.

Maleknaz et al. [14] analyzed (among others) the release cycle times of 6,003 mobile apps on Google Play as a *treatment* with the aim to predict as *outcome* the customer satisfaction expressed through an app rating. To do so, they introduced a generic analytical approach called the Gandhi-Washington Method (GWM). For the specific scenario of mobile app rating, the method consists of encoding and summarising the sequence of release cycle times of each app using regular expressions over the alphabet S (short release cycle), M (medium release cycle), L (long release cycle); followed by statistical tests over those generated expressions to determine causal effects on the outcome variable. They found that apps with sequences of long releases followed by sequences of short releases have the highest median app rating. Apps with sequences of long followed by sequences of medium releases get a lower median rating. Finally, apps with sequences of long releases exclusively get the lowest median rating.

2.2. Bug handling in short release cycles

Khomh et al. [10] 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 fewer bugs are fixed during the testing period and that bugs are fixed faster under a rapid release model. In a follow-up work [15], the authors reported that, although post-release bugs are fixed faster in a shorter release cycle, a smaller proportion of bugs is fixed compared to the traditional release model. Interviews conducted with six Mozilla employees revealed that they can be "less effective at triaging bugs with the rapid release" and that more beta testers using the rapid releases can generate more bugs.

Da Costa et al. [3] 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 issues are triaged and fixed faster in rapid releases. In a follow-up work [16] they reported that triaging time is not significantly different among the traditional and rapid releases.

2.3. Bug triaging

Saha et al. [17] 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 Core 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. [18] 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 [19] analyzed 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.

2.4. Bug fixing time prediction and estimation

Panjer [20] carried out a case study on Eclipse projects, and showed that the most influential factors affecting bug fixing time are found in initial bug report fields (e.g., severity, product, component, and version), and post-submission information (e.g., comments).

Giger et al. [21] found that the assigned developer, bug reporter and month when the bug was reported to have the strongest influence on the bug fixing time in Eclipse, Mozilla and Gnome. Marks et al. [22] 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. [23] 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.

Rwemalika et al. [24] studied the characteristics and differences between pre-release bugs and post-release bugs in 37 industrial Java projects. They found that post-release bugs are more complex to fix since they require modification

of several source code files, written in different programming languages and configuration files.
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 the current manuscript and its predecessor [8], we focus on the bug triaging and fixing time, and on bug resolution and fixing rate. We are not aware of any related study comparing these metrics before and after the release is delivered, and how they evolve over successive releases considering the bug severity level. Moreover, we study the impact of feature freeze periods on the bug handling process.

3. Experimental Setup

This section presents the experimental setup, including the selected case study, the data extraction process, the metrics used to quantify the answers to the research questions, and the process followed to receive qualitative feedback from Eclipse maintainers to confirm these quantitative results. The dataset and scripts used for the current study are publicly available in a replication package on Zenodo [25].

3.1. Selected Case Study

We have selected the Eclipse Core as a case study because it is a long-lived open source ecosystem, with a large community of contributors. It has fixed durations for both annual or rapid releases, allowing to make fair comparisons across successive releases. More importantly, during its 4.x release series, it has switched in 2018 from a yearly to a quarterly (i.e., every 13 weeks) release policy.
Eclipse has been widely studied in software evolution research [26, 27] and in research about bug handling in particular [20, 28]. It is a large and active project whose bug reporting activity is hosted on the Bugzilla bug tracking platform [29]. This platform manages and keeps track of its bugs, allowing developers to manage all concerns of the bug life cycle effectively.
The development of Eclipse is subdivided into the Core projects and the plugins. The Core projects have a stable development community and a regular release process. We focus only on bugs related to the Core projects since the plugins do not follow the same regular release policy and their bug handling activity can be affected by the absence of factors such as continuous bug monitoring and continuity of the plugin development. The Core projects

of Eclipse are *Platform*, *JDT*, *Equinox* and *PDE* [30]. We do not consider the *e4* and *Incubator* projects in our analysis because: (i) *e4* is an incubator for community exploration of future technologies of Eclipse and uses a different versioning scheme than the other projects; (ii) *Incubator* contained only very few reported bugs (49), all with an unspecified version of the Eclipse release. The main research goals and associated research questions that will be explored in Section 5 and Section 6 focus on the 4.x release series, starting with release 4.2 in June 2012. Until release 4.8 in June 2018, Eclipse followed an **annual** "simultaneous release" scheme[1] where in June a new release was delivered simultaneously for every Core project. Each release 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 from an annual to a **quarterly** release policy (i.e., every 13 weeks) without intermediate update releases.

The Eclipse Core project release schedule, for both annual and quarterly releases, is composed of several successive milestone builds (M_1, M_2, etc.), followed by a *feature freeze* period after the last milestone, during which maintainers are no longer allowed to introduce new features, and have to concentrate instead on fixing bugs for the release under development. During this freeze period, a number of successive release candidates (RC_1, RC_2, etc.) are created.

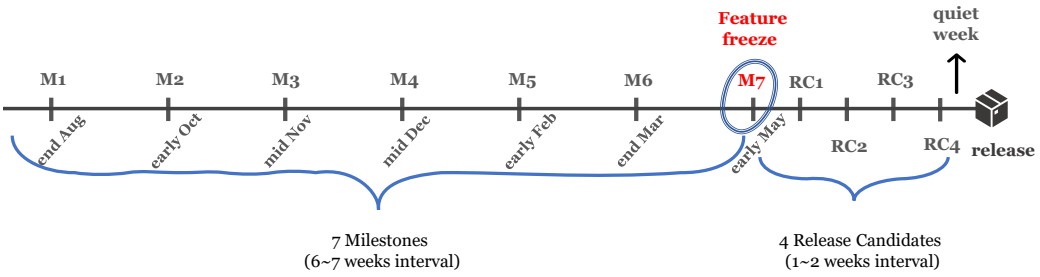


Figure 1: Annual release schedule for releases 4.2 till 4.8 of Eclipse Core projects.

Fig. 1 shows the annual release schedule, including 7 milestones occurring at

[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`.

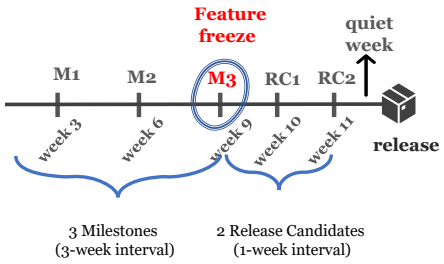


Figure 2: Quarterly release schedule for releases of Eclipse Core projects since 4.9.

roughly 6-week intervals, followed by 4 release candidates during the feature freeze period. Fig. 2 shows the quarterly release schedule, including 3 milestones occurring at 3-week intervals, followed by 2 weekly release candidates during the feature freeze period. In both annual and quarterly releases, the last week before a release is always a quiet week during which there are no further builds. This week is reserved for final in-depth testing and preparation for the release.

3.2. Extracting and Processing Bug Report Data

Our empirical analysis is based on bug report data extracted from Bugzilla for each release of each Eclipse Core project. A typical bug report contains a wide variety of fields. A description of those fields that are relevant in the context of our empirical analysis is summarised in Table 1.

Each bug report is accompanied by a *history* containing all events that occurred during the bug's life cycle (e.g., a reporter created the bug, the bug was assigned to someone, the status and resolution type of a bug were changed, etc.). Eclipse bug handling activity follows a dedicated process, confirming to the Bugzilla life cycle.[2] A bug report starts in the `UNCONFIRMED` status. Once the bug is confirmed as a real problem its status changes to `NEW`. The status of a bug becomes `ASSIGNED` when a programmer takes charge of fixing the bug. Once a bug resolution is proposed, the bug status changes into `RESOLVED`. Eventually, a bug reaches the `CLOSED` status. The lifecycle of some bugs can be more complex; for example, a bug could be `REOPENED` because the initial resolution was or has become inappropriate.

Table 2 provides a fictitious example of what such a bug modification history typically looks like. The first column shows the email of the person *Who*

[2] `https://wiki.eclipse.org/Development_Resources/HOWTO/Bugzilla_Use`

Field	Description
ID	The bug ID.
Description	Descriptive information that helps in the debugging process such as the reported error message, the steps to reproduce the error, etc.
Product	The product which is affected by the bug (i.e., JDT, PDE, Platform, Equinox).
Reporter	The person who reported the bug.
Reported	The date and time of bug creation.
Assignee	The person in charge of resolving the bug.
Version	The version of Eclipse the bug was found in (e.g. 3.2, 4.5).
Severity	The estimated bug impact as perceived by the bug reporter (i.e., enhancement, trivial, minor, normal, major, critical, blocker).
Status	The bug's latest status (i.e., `NEW`, `ASSIGNED`, `RESOLVED`, `VERIFIED`, `CLOSED`, `REOPEN`)[a].
Resolution	The latest resolution status of a `RESOLVED` bug (i.e., `FIXED`, `WONTFIX`, `DUPLICATE`, `INVALID`, `WORKSFORME`, `NOT_ECLIPSE`, `MOVED`, `REMIND`, `LATER`).

[a]See `https://wiki.eclipse.org/Development_Resources/HOWTO/Bugzilla_Use`

Table 1: Fields of Eclipse Bugzilla bug reports that are used for the empirical analysis.

performed the change, the second column *When* this change occurred, the third column *What* was changed, and the last two columns show the old value that was *Removed* and the new value that was *Added* to replace the old value. For instance, the first modification was performed by Donald at 9.23 AM EDT on the 4th of October 2016. Donald changed the value of the *Assignee* field to daisy@eclipse.org and the status value from `NEW` to `ASSIGNED`.

The *Severity* of bugs is reported by their owners based on their personal perspective. The Eclipse community considers seven levels [3]: enhancement, trivial, minor, normal, major, critical and blocker. We excluded 28,579 bugs marked as enhancement from our analysis, since feature enhancements are considered to be new functionality requests rather than bugs [31].

To extract the bug histories of all reported Eclipse bugs we used the Bugzilla API[4]. Since we focus on Eclipse Core projects only, we only considered bug reports for which the *Product* field was tagged with *Platform*, *JDT*, *Equinox*

[3]`https://wiki.eclipse.org/Eclipse/Bug_Tracking#Severity`

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

Who	When	What	Removed	Added
donald@eclipse.org	2016-10-04	cc		dagobert@eclipse.org
	09:23:28	Assignee		daisy@eclipse.org
		Status	NEW	ASSIGNED
daisy@eclipse.org	2016-10-06	Version	4.5	4.5.1
	11:02:01	Attachment #111253	0	1
daisy@eclipse.org	2016-10-25	Severity	normal	critical
	11:05:34	Status	ASSIGNED	RESOLVED
		Resolution	—	FIXED

Table 2: Bug report history for a fictitious example of an Eclipse bug report in Bugzilla.

or *PDE*. We extracted 215,591 bug histories corresponding to these projects. Our dataset was fetched on 27 July 2020, and the earliest and latest dates of reported bugs in our dataset correspond to 11 October 2001 and 27 July 2020, respectively.

In a next step, we filtered bugs based on their *Version* field. As our goal is to study the bug resolution process in relation to each Eclipse release, we only considered those bug reports whose version corresponds to an actual Eclipse release ranging between 3.0 and 4.15. To this end, we excluded 3,296 bugs with unspecified *Version* field and 33,701 bugs corresponding to versions outside of the specified version range. 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 2,569 bugs that corresponded to versions that are not listed in the official releases of Eclipse, i.e., 4.0 and 4.1. Our final dataset consists of 143,606 bug reports, of which 107,397 (i.e., 74.8%) belonging to the 3.x version range, and 36,209 (i.e., 25.2%) belonging to the 4.x version range. While Section 4 focuses on all these bugs, Section 5 and Section 6 focus only on the 4.x release range during which the transition to a faster release cycle took place. Within this range, there are 29,831 bug reports for annual releases (4.2→4.8) and 6,378 bug reports for quarterly releases (4.9→4.15).

For the remainder of the analysis, we partitioned all reported bugs into groups according to their major release number. For example, group 4.7 contains all reported bugs whose *Version* field prefix is 4.7. The aforementioned processing steps mitigate several threats that could bias the results of our study.

As pointed out by Tu et al. [32], incorrect use of bug tracking data may

threaten the validity of research findings because the values of bug report fields (e.g., *Version*, *Status*, *Severity*) may change over time. They recommend researchers that rely on such data to mitigate data leakage risks by having a full understanding of their application scenarios, the origin and change of the data, and the influential bug report fields. We therefore assessed this threat for the bug report fields *Version* and *Status* in Eclipse.[5]
Examining the bugs that changed their *Version* field during the bug fixing cycle, we found 1,437 bugs that were reassigned to different releases throughout their history, out of which 1,291 bugs that were reassigned 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 21 out of 1,233 `RESOLVED` bugs are resolved in multiple major Eclipse releases; thus, the impact on our analyses is minimal. In all our research questions, we consider the impact of bug severity on prioritization of bugs. We used the categorization strategy of Gomes et al. [33] to aggregate bugs into two groups: *severe* (including blocker, critical, and major severity) and *non-severe* (including normal, minor and trivial severity). The threats related to changes in the bug severity field during the bug history was examined, and we found 2,290 bug reports that changed their severity over time, out of which 1,503 bugs being reassigned to different severity category. In those cases, we used the latest severity category assigned to each bug, as changes in the severity level indicate that prior severity levels were not accurate.
A reported bug is considered as resolved if somewhere in the bug history the *Status* field is changed to `RESOLVED`. The corresponding resolution date can be found in the *When* column of the bug history. The *Resolution* field allows to mark a `RESOLVED` bug as `FIXED`, and the fixing date can be found in the *When* column of the bug history. Since the value of the *Status* field can be modified multiple times, we register the *last* date that the bug was marked as `RESOLVED`. 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.
Similary, the *Status* field allows to mark a reported bug as `ASSIGNED` and the assignment date can be found through the *When* column of the bug history.

[5]For the *Severity* field we already discussed earlier in this subsection how we coped with this threat.

In case of multiple reassignments of a bug to different developers, we register the *first* assignment date, reflecting the moment when the bug was triaged for the first time. Note that the Eclipse community has been assigning bugs using an alternative process since 2009[6]: 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` 5,449 bugs corresponding to this case.

3.3. Proposed Bug Handling Metrics

This section introduces all formal notations and metrics that are needed to address the different research questions. We introduce the following notations to refer to the sets of bugs considered during our analysis:

- B_{report} is the set of all reported bugs
- $B_{\text{assign}} \subseteq B_{report}$ is the set of all `ASSIGNED` bugs
- $B_{\text{resolve}} \subseteq B_{report}$ is the set of all `RESOLVED` bugs
- $B_{\text{fix}} \subseteq B_{resolve}$ is the set of all `FIXED` bugs
- The superscript notation B^r constrains these sets to bugs targeting a specific release r. For example, $B^{4.7}_{\text{resolve}}$ contains all ***RESOLVED*** bugs for which the *Version* field refers to release `4.7`.

Based on these sets, we define functions returning the dates corresponding to specific activities in the history of a given bug report:

- $D_{\text{report}} : B_{\text{report}} \rightarrow Date$ 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.

[6] `https://wiki.eclipse.org/Platform_UI/Bug_Triage`

Using the above sets and functions, we define four metrics that will be used to answer the research questions. We define bug **triaging time** T_{triage} and bug **fixing time** T_{fix} as follows:

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

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

Given a date range $d = [d_1, d_2[$, we define

$$B_{\text{resolve}}(d) = \{b \in B_{\text{resolve}} \mid D_{\text{resolve}}(b) \in d\}$$

and similarly for $B_{\text{report}}(d)$ and $B_{\text{fix}}(d)$. Using this notation, we define bug **resolution rate** *ResRate* as the proportion of reported bugs that have been `RESOLVED` in the considered date range, and bug **fixing rate** *FixRate* as the ratio of `FIXED` over *reported* bugs:

$$\mathit{ResRate}(d) = \frac{|B_{\text{resolve}}(d)|}{|B_{\text{report}}(d)|} \qquad \mathit{FixRate}(d) = \frac{|B_{\text{fix}}(d)|}{|B_{\text{resolve}}(d)|}$$

The following example illustrates these two 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 $\mathit{ResRate} = \frac{20}{30} = 0.66$, and $\mathit{FixRate} = \frac{12}{20} = 0.6$.

3.4. Applying the metrics to specific Eclipse releases

Since our empirical analysis aims to relate bug handling activity to specific time periods, such as the period separating two successive releases, the feature freeze period, and the development period preceding the feature freeze, we introduce functions and sets enabling us to work with such information. Let R be the ordered set of considered Eclipse releases. We define the following functions:

- date: $R \to \mathit{Date}$ returns the date of a given release
- prev: $R \to R$ returns the previous release of a given release
- next: $R \to R$ returns the next release of a given release
- freeze: $R \to \mathit{Date}$ returns the start of the feature freeze period for a given release

Given a release $r \in R$, we can focus the analysis on **only those bugs targeting release** r, based on the value of their *Version* field in the bug report. To do so, we introduce different release-dependent data ranges:

- $d^{<\text{freeze}}(r) = [\text{date}(\text{prev}(r)), \text{date}(\text{freeze}(r))[$ is the development period of release r
- $d^{>\text{freeze}}(r) = [\text{date}(\text{freeze}(r)), \text{date}(r)[$ is the feature freeze period of release r
- $d^{\text{before}}(r) = [\text{date}(\text{prev}(r)), \text{date}(r)[\quad = \quad d^{<\text{freeze}}(r) \cup d^{>\text{freeze}}(r)$
- $d^{\text{after}}(r) = [\text{date}(r), \text{date}(\text{next}(r))[$

Based on these data ranges, we restrict the set of reported bugs B_{report} (see Section 3.3) as follows:

- $B^{\text{before}}_{\text{report}}(r) = \{b \in B^r_{\text{report}} \mid D_{\text{report}}(b) \in d^{\text{before}}(r)\}$
- $B^{\text{after}}_{\text{report}}(r) = \{b \in B^r_{\text{report}} \mid D_{\text{report}}(b) \in d^{\text{after}}(r)\}$
- $B^{<\text{freeze}}_{\text{report}}(r) = \{b \in B^r_{\text{report}} \mid D_{\text{report}}(b) \in d^{<\text{freeze}}(r)\}$
- $B^{>\text{freeze}}_{\text{report}}(r) = \{b \in B^r_{\text{report}} \mid D_{\text{report}}(b) \in d^{>\text{freeze}}(r)\}$

In a similar way, we restrict the sets B_{assign}, B_{resolve} and B_{fix}.
With these notations, we can compute the *resolution rate* before and after each release, as well as during the development period or the freeze period of any given release, **restricted to only those bugs targeting release** r, by using

$$\begin{array}{cc} \mathit{ResRate}(d^{\text{before}}(r)) & \mathit{ResRate}(d^{\text{after}}(r))) \\ \mathit{ResRate}(d^{<\text{freeze}}(r)) & \mathit{ResRate}(d^{>\text{freeze}}(r)) \end{array}$$

In a similar way, we restrict the *fixing rate* to a specific period related to a given release r.
To focus on bugs belonging to specific severity groups, we use the auxiliary function $\mathit{severity} : B_{\mathit{report}} \rightarrow \{\mathit{non\text{-}severe}, \mathit{severe}\}$ to return the severity group (as explained in Section 3.2) of a given bug. Given a set of bugs B and a severity group s, we define $B \mid_s = \{b \in B \mid \mathit{severity}(b) = s\}$ as the subset of all bugs in B belonging to this severity group. Using this definition, we can for example define $B^{\text{before}}_{\text{report}} \mid_{\mathit{severe}} (r)$ and similar for all other possible variations.

3.5. Statistical Methods

The quantitative analyses in Sections 5 and 6 rely on a range of statistical tools. Most of our analyses aim to compare two populations by testing if their distributions are different. We test all statistical hypotheses for different significance levels α. We reject a null hypothesis if $p < \alpha$, and denote this with * if $\alpha = 0.05$, ** if $\alpha = 0.01$, and *** if $\alpha = 0.001$.

Since software engineering data often do not meet the normality assumption [34], and this is the case for Eclipse bug data in particular [35], we select appropriate non-parametric tests that do not require this assumption [36]. Normality is tested for both populations that are compared using the Kolmogorov–Smirnov test. In case both populations are normally distributed, we compare their distributions and mean values with a *two-sided t-test* if the populations are related; otherwise, we use the *Welch t-test* if the samples are independent or of unequal size. For non-normal distributions, we use the *Wilcoxon rank sum test* if the samples are related; otherwise, we use the *Mann–Whitney U test*. We compute the *effect size* using Cliff's delta d [37, 38] and interpret the results using [38]. In all cases where the null hypothesis is rejected, the sign of d allows us to determine which of both distributions is higher than the other one.

The analysis of $RQ1.2$, reported in Section 5, uses the technique of *survival analysis* (a.k.a. event history analysis or duration modeling) to model the expected time duration until the occurrence of a specific *event of interest* with the aim to estimate the survival rate of a given population [39]. Survival analysis is frequently used in medical sciences (e.g., to study the effect of a particular treatment on patients suffering from a disease), economics and sociology. It has also been applied in software evolution research [40, 41]. Survival analysis models take into account the fact that some observed subjects may be "censored", either because they leave the study during the observation period, or because the event of interest was not observed for them during the observation period. A common non-parametric statistic used to estimate survival functions is the Kaplan-Meier estimator [42]. To compare survival curves, we use the *log-rank test* [43] to test the null hypothesis that there is no difference between the populations in the probability of an event at any time point.

The analysis of $RQ2.2$ reported in Section 6, uses boxen plots [44] to show the distributions of triaging and fixing time. These plots visualise different quartile values and convey precise estimates of head and tail behavior.

3.6. Feedback from Eclipse Maintainers

In order to verify if the empirical results we obtained correspond to the perception of the Eclipse community, we complemented the quantitative analysis with a small qualitative analysis by consulting Eclipse Core maintainers that actively experienced the transition from the yearly to the quarterly release cycle. We identified maintainers that were active in bug fixing activities, both *before* and *after* the change in release policy by manually analysing their presence and historical activity in the extracted bug tracker data.
Four out of five of the respondents reported they have at least been contributing to Eclipse Core for five years, while the remaining one has been contributing for two years. Using an online form[7], we solicited their experience on the transition to a rapid release cycle and how they perceived this had impacted the bug handling process. The first three questions (#3–#5) collected demographic information. Questions #6–10 focused on the impact of switching to a rapid release cycle, by means of open questions about the Eclipse preparation and difficulties they faced during the transition. Questions #11–17 inquired about the quantitative results we obtained for $RQ1.1$ to $RQ2.2$, in order to check whether our results confirm with their perception about the transition to a rapid release cycle. Moreover, we asked questions to (1) check if there were any other important changes in the Eclipse release process and if changes in tooling may have affected the effectiveness of bug handling and related activities; (2) ask their opinion if the community needs more contributors to be involved in bug handling; and (3) check if and how the transition to quarterly releases impacted the pressure on the developers. Finally, questions #17–20 informed about the perceived benefit of rapid releases in general, the respondent's preference of the type of release cycle (fixed or variable), and any advice for future adopters of rapid releases.

4. Preliminary Analysis

4.1. The evolution of Eclipse bug handling

Before starting to study the actual research questions presented in Section 1, we carry out a preliminary analysis of how Eclipse bug handling evolved over time since release 3.0. This was the first release coordinated and shipped

[7]The online form can be found in our replication package [25].

in June 2004 by the independent Eclipse Foundation; previous releases were coordinated and controlled by IBM.
To do so, for each considered release, we computed B^r_{report}, B^r_{resolve}, B^r_{fix} and B^r_{assign}, without any restriction on the date range. Fig. 3 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 annual releases. Starting from annual release 4.2, the number of bug reports appears to become stable. For the quarterly releases starting from 4.9, the numbers are still stable, but much lower than for the annual releases (because of their shorter release duration).

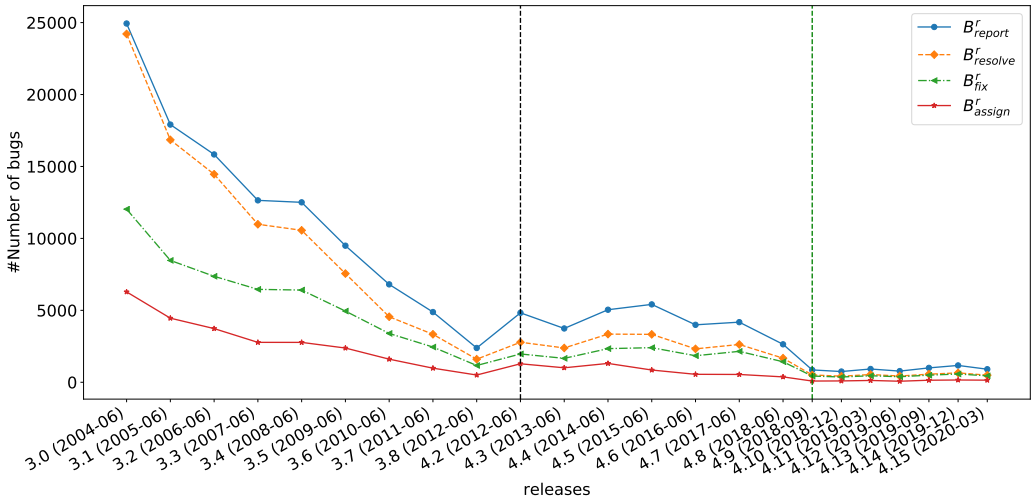


Figure 3: Number of bugs targeting a specific release. The black vertical dashed line indicates the switch from Eclipse 3.x to 4.x. The green vertical dashed line indicates the transition from an annual to a quarterly cycle since 4.9.[8]

Fig. 4 shows the evolution of *ResRate* and *FixRate* per considered release r. We observe a *decreasing* resolution rate, while the fixing rate is *increasing* across releases. For the 3.x release range, *ResRate* decreases from 0.97 to 0.67, while *FixRate* increases from 0.50 to 0.72. Starting from release 3.6, *FixRate* is consistently higher than *ResRate*, and the difference between both rates continues to become more pronounced over time.

[8]In all our figures, the black vertical line signifies the switch from Eclipse 3.x to 4.x and the green vertical dashed line signifies the transition from an annual to a quarterly cycle.

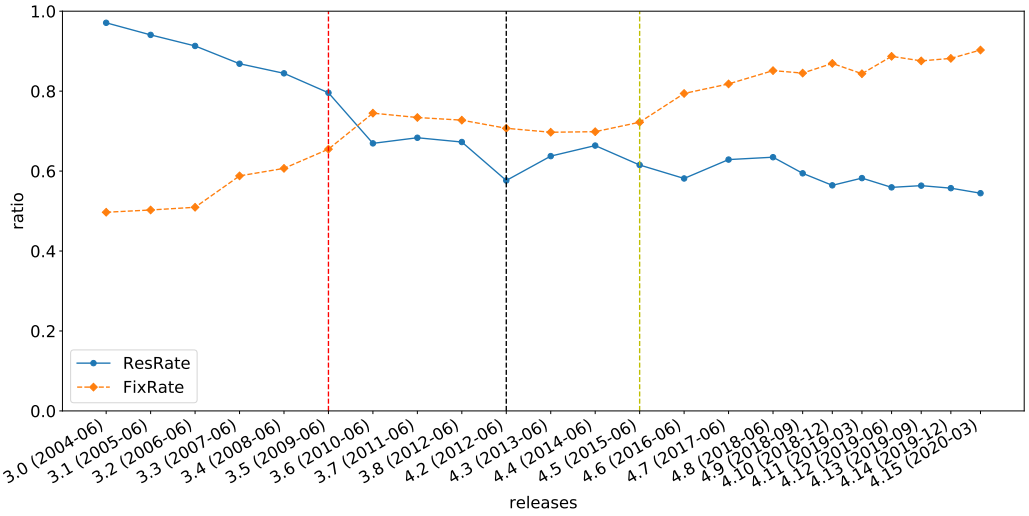


Figure 4: Resolution and fixing rates per targeted Eclipse release. The red vertical line indicates the last release where REMIND/LATER resolution was used. The yellow vertical line corresponds to the introduction of the AERI error reporting tool.

The observed change in behaviour since release 3.6 can be explained by the practice of "resolving" a bug by giving it the LATER or REMIND status in the *Resolution* field, corresponding to the desire to postpone the bug resolution.[9] 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 in Fig. 4). By analyzing the delays of a follow-up resolution of the REMIND and LATER bugs (3,633 bugs), we find that the majority of them (56%) lingered for more than 3 years before getting their final resolution. We study these events by applying survival analysis to model the expected time duration for a bug to be subsequently resolved without REMIND/LATER resolution. The event of interest is the time when a bug has been RESOLVED, and the duration is computed from the first time the bug was marked as LATER or REMIND until the first resolution that is different from LATER and REMIND. The Kaplan-Meier survival curves in Fig. 5 visualize the survival model. We observe that, for more than 50% of the bugs that were marked as LATER or REMIND, it takes more than 1,220 days to actually resolve them later on (see red dashed lines in Fig. 5). The follow-up resolution statuses are, in decreas-

[9]See https://www.eclipsezone.com/eclipse/forums/t83053.html

ing order of frequency: `WONTFIX` (1,257 bugs), `INVALID` (1,058 bugs), `FIXED` (495 bugs), `DUPLICATE` (253 bugs), `WORKSFORME` (242 bugs) and `NOT_ECLIPSE` (11 bugs). 317 of the `REMIND/LATER` bugs (i.e., 8.7%) are not resolved with another resolution until today. We checked if the severity group (*non-severe* or *severe*) of the bugs marked as `LATER/REMIND` influences the resolution time but we did not observe any such effect.

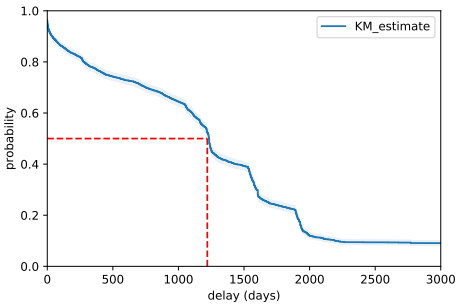


Figure 5: Kaplan-Meier survival curves modeling the time duration until a bug marked as `LATER` or `REMIND` gets resolved at some later time.

Coming back to Fig. 4, 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* continues to increase. This change coincides with the introduction of an Automated Error Reporting Client (called AERI)[10] since June 2015 (yellow vertical line). AERI facilitates reporting errors as users do not need to create Bugzilla entries and *'it automatically uploads issues to a central server, providing valuable information as to where the issue may exist in Eclipse'*.[11]

In turn, users can provide comments with their reports which are helpful when fixing bugs. According to Sewe's report on AERI [45], commented bug reports are more than twice as likely to be fixed compared to those without user comments.

The bug handling metrics reported in Figures 3 and 4 for the 3.x series are quite different from the recent 4.x series, implying that the way in which bugs were handled during the 3.x series does not reflect the current practices

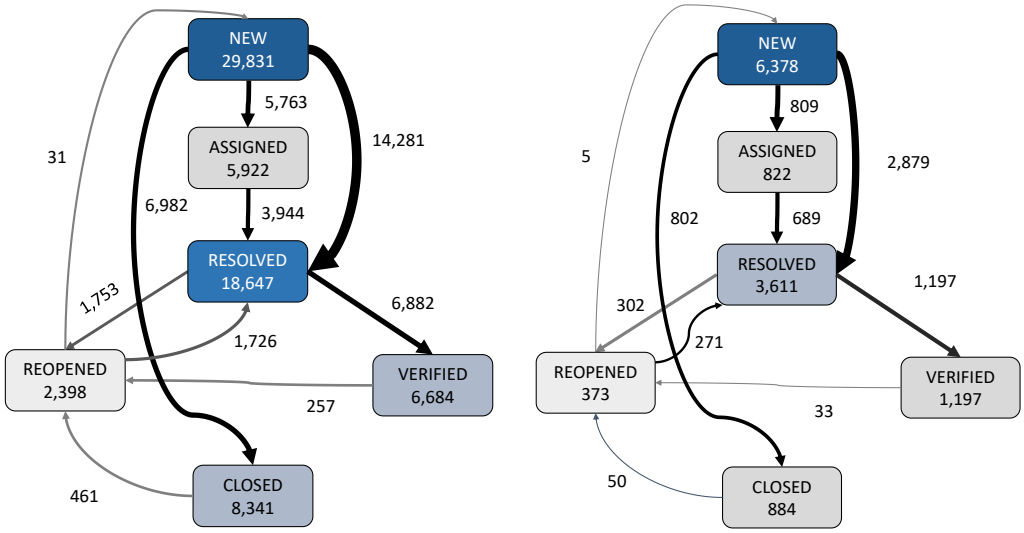
[10]See `https://www.eclipse.org/community/eclipse_newsletter/2016/july/article3.php`

[11]https://www.infoq.com/news/2015/03/eclipse-mars-reporting/

of Eclipse Core maintainers. Because of this, Sections 5 and 6 only focus on the 4.x series during which the transition to rapid releases happened.

4.2. Process mining of Eclipse bug handling

As mentioned in Section 3.2, Eclipse bug handling activity follows the Bugzilla life cycle.[12] To analyse the Eclipse bug cycle process, we apply Disco[13], a commercial process mining tool, to the event logs (containing the bug report history) retrieved from Bugzilla. We do this separately for the bug reports corresponding to the Eclipse annual releases and the Eclipe quarterly releases, respectively. Fig. 6a and Fig. 6b show the obtained process maps based on the bug reports for both sets of releases.



(a) Annual releases (b) Quarterly releases

Figure 6: Process maps computed based on the event logs of the extracted Bugzilla bug reports for Eclipse annual and quarterly releases.

We have 6 nodes each corresponding to an activity (i.e., step in the bug handling process) in the process map for both annual and quarterly releases, and a directed edge that represents the transition between two activities (node). The number associated to each node corresponds to the absolute

[12] `https://www.bugzilla.org/docs/2.18/html/lifecycle.html`
[13] https://www.fluxicon.com/

frequency of occurrence of all activities, and the number associated to each edge indicates the absolute frequency of occurrence of each transition. The shade and thickness of each edge reflect its frequency (i.e., the darker and thicker, the more frequent).

In order to determine how faithfully Eclipse bug reports are actually following the recommended Bugzilla life cycle, we used an algorithm proposed by Gupta et al. [46] to compute the *degree of conformance* (a.k.a. *fitness*). Conformance checking aims to detect inconsistencies between a design-time process model (here: the Bugzilla life cycle) and the as-is process model extracted from the run-time event logs (here, the bug life cycles extracted for Eclipse annual and quarterly releases, respectively). The higher the fitness, the better the design-time process model describes the recorded run-time process. A fitness of 1 indicates that the design-time process model reproduces every trace in the event log; a value of 0 means that the design-time process model cannot repeat any of the run-time cases. We obtained a high fitness of 0.72 for annual Eclipse releases and an even higher fitness of 0.86 for quarterly Eclipse releases. This shows that the Eclipse bug handling process conforms to the recommended Bugzilla process model.

As a second analysis, we aimed to determine if the bug handling process has changed after the transition from annual to quarterly releases. To do so, we carried out a χ^2-test. The null hypothesis H_0 states that there is no statistically significant difference between the absolute frequency of bug statuses (node values in the process maps) for the annual and quarterly releases process. We could reject H_0 with high confidence ($p = 0$), signifying that the difference between bug status frequencies of annual and quarterly releases is statistically significant. Given that such a difference has been found between annual and quarterly releases, Section 5 and Section 6 will explore how the bug handling differs (e.g., in terms of bug handling rate and time) between both types of releases.

5. Goal 1: Impact of rapid releases on the bug handling process

Our first research goal aims to study if the transition to rapid releases resulted in less time for the community to handle bugs before the release, leading to potentially more post release bugs in need of resolution. In this section, we analyse if the bug handling activity is different before and after each release, and whether and how this changed after the transition to rapid releases. Our investigation is guided by the first two research questions (RQ1.1 and RQ1.2).

The quantitative results that will be presented throughout this section will be corroborated by the feedback we received from the five consulted Eclipse maintainers (cf. Section 3.6).

*RQ*1.1 ***How does the bug handling rate evolve across releases?***

In this research question, we study the difference in bug resolution and fixing rate before and after each release. We intuitively expect that contributors handle bugs more intensively in the period *before* than *after* the upcoming release date, as they strive to not deliver a buggy release.

First, we compute, for each release r in the 4.x series, the resolution rate before and after the release, i.e., $ResRate(d^{\text{before}}(r))$ and $ResRate(d^{\text{after}}(r))$. Fig. 7 suggests a slightly decreasing rate after the different releases, while the rate is fluctuating before the release. A linear regression analysis confirms this trend for the resolution rate after each release ($R^2 = 0.74$). The regression analysis could not reveal any linear trend for the resolution rate before each release because of a too small R^2 value (0.0018).

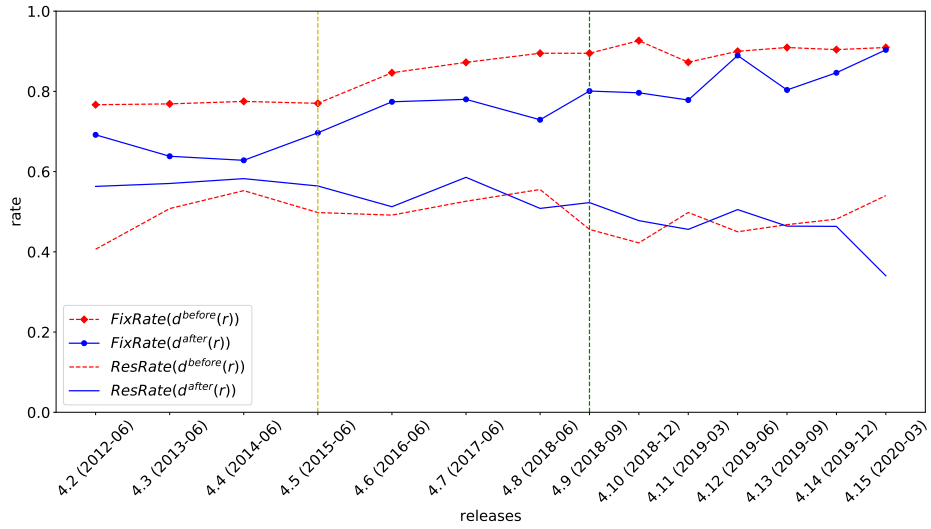


Figure 7: Evolution of *ResRate* and *FixRate* before and after each 4.x release. The yellow vertical line corresponds to the introduction of the AERI error reporting tool.

The resolution rates *before* and *after* each release fluctuate between 0.4 and 0.6. This signifies that around 1 out of 2 bugs do not get resolved. We used a Wilcoxon rank sum test to verify the null hypothesis $H0_1^r$ stating that there

is no statistically significant difference between the resolution rates before and after the release. We could not reject $H0_1^r$ so we have no evidence that the resolution rates before and after each release are different.
Next, we computed the fixing rate before and after each release r, i.e., $FixRate(d^{\text{before}}(r))$ and $FixRate(d^{\text{after}}(r))$. Fig. 7 suggests that the fixing rate *before* each release is higher than *after* the release. Using the Wilcoxon rank sum we test the null hypothesis $H0_1^f$ stating that there is no statistically significant difference between fixing rates before and after a release. $H0_1^f$ was rejected ($p < 0.05$) with large effect size ($d = 0.54$). This shows that the bug fixing rate before the release date is higher than after that date. A linear regression analysis confirms an increasing linear trend for fixing rate before each release ($R^2 = 0.89$) as well as after each release ($R^2 = 0.74$).
Four out of five of the consulted Eclipse Core maintainers stated that they do not really differentiate between bugs before and after the release. The received responses were based on a 5-point Likert scale, i.e., participants rated factors using ranks from 1 (strongly disagree) to 5 (strongly agree). This complies with our observation that there is no difference between the resolution rate before and after release. However, as we see in Fig. 7, a higher proportion of bugs is fixed before compared to after the release.

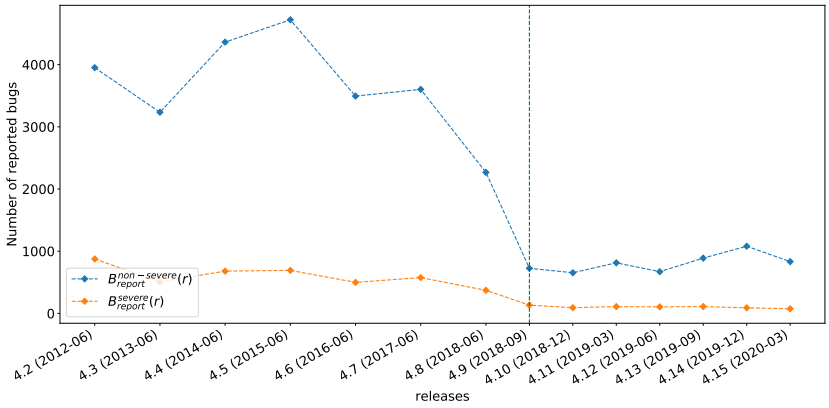


Figure 8: Evolution of the number of reported bugs per severity group.

We also investigated if maintainers differentiate between bugs according to their severity. Fig. 8 shows that the number of reported *non-severe* bugs decreases over time to become stable after the transition to a quarterly release cycle. We also computed $ResRate(d^{\text{before}}(r))$ and $ResRate(d^{\text{after}}(r))$ per

severity group (*severe* and *non-severe*) and found a decreasing evolutionary trend for both, similar to what was observed in Fig. 7.[14] Using the Wilcoxon rank sum we test the null hypotheses $H0_1^r \mid_{non-severe}$ and $H0_1^r \mid_{svvere}$ stating that there is no statistically significant difference between resolution rates before and after a release for *non-severe* and *severe* bugs, respectively. We could not reject the null hypothesis for either group, however.

When computing the bug *fixing* rates $FixRate(d^{\text{before}}(r))$ and $FixRate(d^{\text{after}}(r))$ per severity group, we found an increasing evolutionary trend, similar to what was observed in Fig. 7. Using the Wilcoxon rank sum, we test the null hypotheses $H0_1^f \mid_{non-severe}$ and $H0_1^f \mid_{severe}$ stating that there is no statistically significant difference between the fixing rates before and after a release for *non-severe* and *severe* bugs respectively. Both $H0_1^f \mid_{non-severe}$ and $H0_1^f \mid_{severe}$ were rejected ($p < 0.05$ and $p < 0.01$ respectively) with a large effect size ($d = 0.53$ and $d = 0.7$ respectively). This confirms that, for both severity groups, the bug fixing rate before the release date is higher than after that date. This signifies that maintainers strive to fix as many bugs as possible of either severity group for the next release.

Finally, we study if any difference could be observed between the two bug severity groups in terms of resolution and fixing rate. We first assess whether a difference between *non-severe* and *severe* bugs can be observed *before* a release. We carry out a first Wilcoxon rank sum test with null hypothesis $H0_1^r \mid_{before}$ stating that, before a release, there is no difference in *resolution rates* between *non-severe* and *severe* bugs. Similarly, for *fixing rate*, we carry out a test with null hypothesis $H0_1^f \mid_{before}$. We could not reject $H0_1^r \mid_{before}$ and $H0_1^f \mid_{before}$, hence there is no statistical evidence of a difference between bug severity groups.

We performed the same hypothesis test to check for a difference in resolution rate and in fixing rate between bug severity groups *after* a release. The respective null hypothesis $H0_1^r \mid_{after}$ for *resolution rate* can be rejected for $p < 0.05$ with large effect size ($d = 0.48$), and for *fixing rate* the null hypothesis $H0_1^f \mid_{after}$ can be rejected for $p < 0.01$ with large effect size ($d = 0.6$). This implies that after the release, the bug resolving and bug fixing rates for *non-severe* bugs are higher than for *severe* bugs.

Overall, our results show that there is a tendency to resolve more *non-severe* than *severe* bugs after the release. Regarding our findings, we asked the

[14] The results per severity group can be found in our replication package [25].

five Eclipse maintainers if they differentiate between bugs according to their severity when handling bugs, but this was not the case. One consulted maintainer stated that long release cycles lead to a higher amount of bugs that are not important or relevant to the bug reporter anymore because users found a workaround, or because the tooling has changed since. With shorter cycles, this is no longer the case, so it becomes less relevant for maintainers to distinguish between important and less important bugs.

Summary: The resolution rate tends to decrease over releases, but there is no significant difference before and after each release. The fixing rate is higher before than after a release. There is no significant change in bug handling rate behavior after the switch in quarterly releases. There is a tendency to resolve more non-severe than severe bugs after the release. Eclipse maintainers do not tend to differentiate between bugs based on their severity.

*RQ*1.2 ***How does the bug handling time differ before and after each release?***

We expect that bug handling activity is more intense *before* a release than *after* it, as maintainers strive to deliver project releases with as few bugs as possible. Hence, we expect to see faster bug triaging and fixing times *before* the release. In this question, we investigate the differences between these two periods in terms of days elapsed to triage and fix bugs.

Triaging time analysis. For each release r in the 4.x series of Eclipse Core, we compute bug *triaging time* $T_{\text{triage}}(b)$ before a release for the population of bugs $B^{\text{before}}_{\text{report}}(r) \cap B^{\text{before}}_{\text{assign}}(r)$ that were reported *and* assigned during that period; and similarly after a release for the population $B^{\text{after}}_{\text{report}}(r) \cap B^{\text{after}}_{\text{assign}}(r)$. We use survival analysis on these populations to model the expected time duration $T_{\text{triage}}(b)$ until *bug b gets assigned for the first time*. Fig. 9 shows a representative selection of Kaplan-Meier survival curves for specific releases, together with their confidence intervals, before and after the release date.[15] To choose the most representative curves, we refer to the log-rank test results so that we show both cases where there is, and there is no statistical difference between the triaging curves. We observe that for some releases, triaging time

[15]The full set of survival curves can be found in [25]

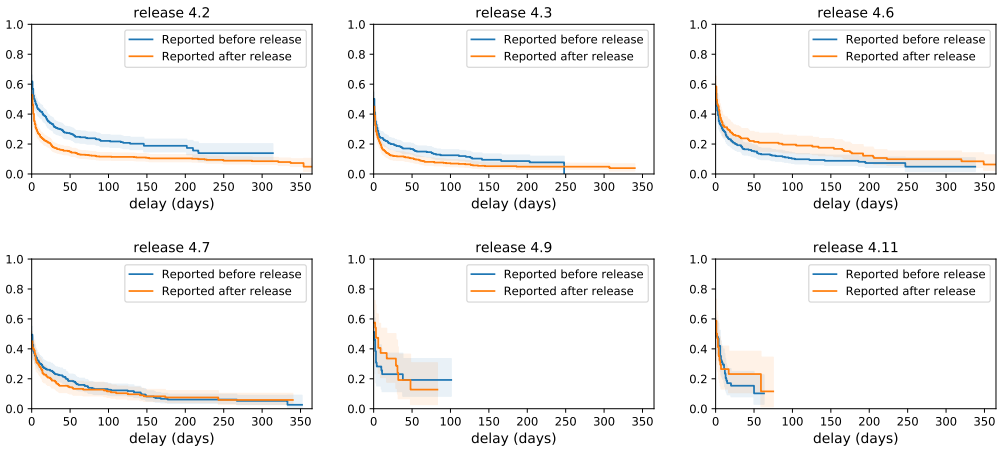


Figure 9: Kaplan-Meier survival curves with 95% confidence interval (indicated by the shaded areas) for *triaging time* before and after each release.

release	triaging time ($H0_2^t$)	fixing time ($H0_2^f$)
	Eclipse 4.x annual releases	
4.2	R^{***}	–
4.3	R^{*}	–
4.4	R^{*}	R^{***}
4.5	R^{***}	R^{***}
4.6	–	–
4.7	–	–
4.8	R^{*}	–
	Eclipse 4.x quarterly releases	
4.9	–	–
4.10	–	–
4.11	–	–
4.12	–	–
4.13	–	–
4.14	–	–
4.15	–	–

Table 3: Log-rank test for difference between the survival distributions of **triaging time** (resp. **fixing time**) *before* and *after* each release. – indicates that the null hypothesis could not be rejected. Whenever it could be rejected (R), the number of stars denotes the significance level α (*=0.05; ** =0.01; *** =0.001).

is different before than after that release. For instance, triaging time before the release is higher than after for releases 4.2 and 4.3, while the opposite is true for release 4.6. However, for most of the releases, the survival curves are partially overlapping. We also observe that the difference between the survival curves tends to decrease over successive releases. We use a log-rank test to verify the null hypothesis $H0_2^t$ stating that there is no difference between the survival distributions of the triaging time before and after a release. Table 3 reports for which releases $H0_2^t$ can be rejected. We observe that $H0_2^t$ is rejected for all annual releases except 4.6 and 4.7, while it is not rejected for any of the quarterly releases. Hence, for annual releases triaging times of the bugs before a release were different than after the release, while this is no longer the case for the quarterly releases. The transition from an annual to a quarterly release policy appears to have been beneficial since such a difference in triaging time is no longer observed. We also studied the impact of bug severity on triaging time, but the results were the same as what has been reported in Table 3. Hence, bug severity does not seem to have a measurable effect on bug triaging time.

To study the impact of quarterly releases on the bug handling process, we analyzed the bug triaging time before and after switching to the quarterly releases. We used survival analysis to model the expected time duration $T_{\text{assign}}(b)$ until bug b is triaged. Fig. 10 (left side) presents the survival curves for bug triaging time for all the annual releases (blue line) and for the quarterly releases (orange line). The figure shows that the bugs are triaged slightly faster after the switch to quarterly releases. For example, 90% of bugs of the quarterly releases are triaged within 50 days while it took more than 100 days to triage 90% of the annual release bugs (see red dashed lines in Fig. 10). With a statistical log-rank test, we verify that there is a difference between the bug triaging time before and after the quarterly releases ($p < 0.01$).

Summary: For annual releases, bugs tend to get triaged faster before than after the release. The transition from an annual to a quarterly release policy appears to have been beneficial, since such a difference in triaging time is no longer observed. The bug severity does not seem to have a measurable effect on bug triaging time before and after the release. Moreover, bugs are triaged faster after the switch to quarterly releases.

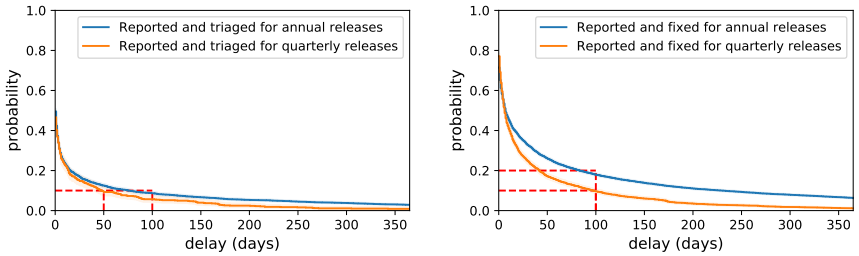


Figure 10: Kaplan-Meier survival curves for bug *triaging time* (left) and *fixing time* (right) for annual and quarterly releases.

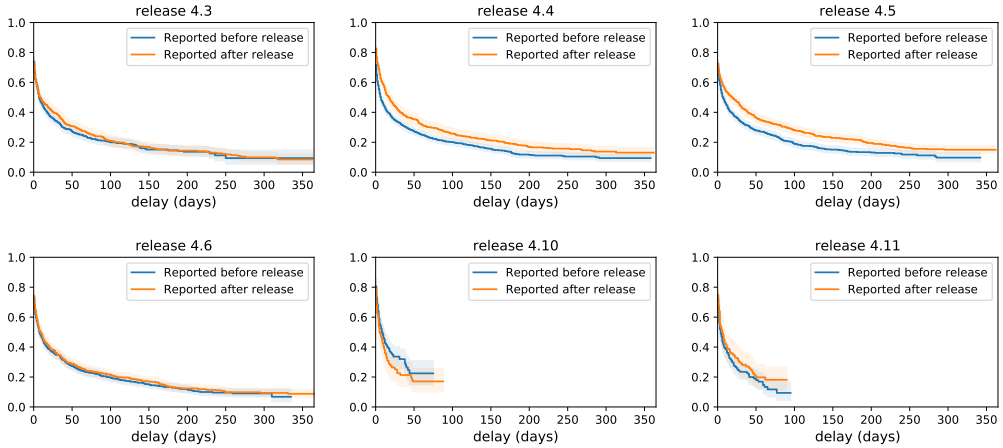


Figure 11: Kaplan-Meier survival curves with 95% confidence interval (indicated by the shaded areas) for *fixing time* before and after each release.

Fixing time analysis. Similar to the triaging time analysis, we compute bug *fixing time* $T_{\text{fix}}(b)$ before a release for the population of bugs $B^{\text{before}}_{\text{report}}(r) \cap B^{\text{before}}_{\text{fix}}(r)$ that were reported *and* fixed during that period; and similarly after a release for the population $B^{\text{after}}_{\text{report}}(r) \cap B^{\text{after}}_{\text{fix}}(r)$. We use survival analysis on these populations to model the expected time duration $T_{\text{fix}}(b)$ corresponding to *the last recorded time that the bug gets fixed*.

Fig. 11 shows a representative selection of Kaplan-Meier survival curves for the bug fixing time. The graphs reveal at most a small difference in the time to fix a bug before and after a release. It seems to take slightly less time to fix a bug before compared to after a release. With a log-rank test, we verify

hypothesis $H0_2^f$ that there is no difference between the fixing time survival distributions before and after a release. We can reject $H0_2^f$ for releases 4.4 and 4.5 only. There is no difference between fixing time before and after the release except for releases 4.4 and 4.5.
We also studied the impact of bug severity on fixing time but the results were essentially the same as what has already been reported in Table 3. Hence, bug severity does not seem to have a measurable effect on bug fixing time.
To study the impact of the release cycle on the bug handling process, we analyzed the bug fixing time before and after switching to quarterly releases. We used survival analysis to model the expected time duration $T_{\text{fix}}(b)$ until a bug b is fixed. Fig. 10 (right side) compares the survival curves for bug fixing time for all the annual releases (blue line) and for the quarterly releases (orange line). The figure shows that bugs are fixed faster for quarterly releases. With a log-rank test, we verify that there is a difference between the bug fixing time before and after the quarterly releases ($p < 0.001$). When consulting Eclipse maintainers about this phenomenon, three out of five believed that bugs are fixed faster in the quarterly releases, which is in conformance with our quantitative analysis.

Summary: There is no difference between the bug fixing time before and after the release. The bug severity does not seem to have a measurable effect on bug fixing time before and after the release. Bugs are fixed faster after the transition to the quarterly releases. The consulted Eclipse maintainers confirmed our results that bugs are fixed faster in the quarterly releases.

6. Goal 2: Impact of feature freezes on bug handling

Our second research goal aims to study to which extent bug handling activity is affected by the presence of feature freezes, and whether the transition to rapid releases has lead to an observable difference as their shorter duration can potentially affect the bug handling activity. Our investigation is guided by two research questions (RQ2.1 and RQ2.2). The quantitative results that will be presented throughout this section will be corroborated by the feedback we received from five consulted Eclipse maintainers (cf. Section 3.6).

*RQ*2.1 ***How does the feature freeze period impact bug handling rate?***

This question studies the bug handling rate before and during the feature freeze period of each considered release. As explained in Section 3.1, during the *feature freeze period*, Eclipse Core project maintainers stop introducing new features and concentrate only on fixing bugs to stabilize the upcoming release. As visualized in Fig. 1, for the annual releases of the 4.x series this period varied between 1 to 3 months.[16] For the quarterly releases, the feature freeze period starts 3 weeks before the release date.[17]

As maintainers focus on bug fixing in the feature freeze period, we study if there is a difference in the bug resolution and fixing rate in the development period and during the feature freeze. We group our results in two date ranges based on when the resolution or fixing took place: (1) during the development period $d^{<\text{freeze}}(r)$ of the *next* release r; and (2) during the feature freeze period $d^{>\text{freeze}}(r)$ of the *next* release r. For each release in the 4.x series and each period, we compute the resolution rate as well as the fixing rate.

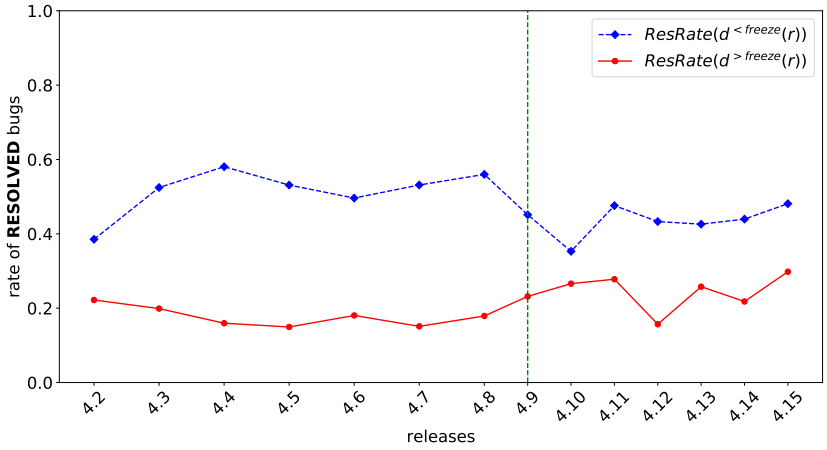


Figure 12: Evolution of *ResRate* during the development and feature freeze period for each 4.x release.

Fig. 12 shows that the *resolution rate* is fluctuating during the development period and the feature freeze period. A regression analysis could

[16] https://www.eclipse.org/eclipse/development

[17] https://wiki.eclipse.org/SimRel/Simultaneous_Release_Cycle_FAQ

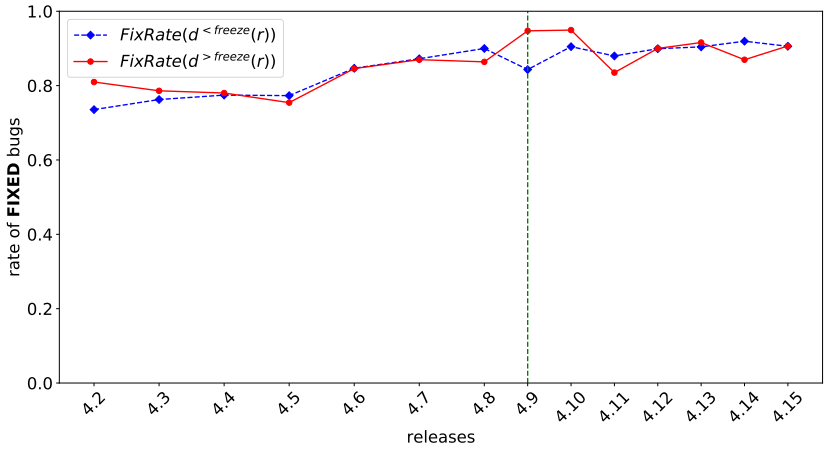


Figure 13: Evolution of *FixRate* during the development and feature freeze period for each 4.x release.

not reveal any linear trend for $ResRate(d^{<\text{freeze}}(r))$ ($R^2 = 0.07$) nor for $ResRate(d^{>\text{freeze}}(r))$ ($R^2 = 0.19$) because of a very small R^2 value. We used a Wilcoxon rank sum test to verify the null hypothesis $H0_3^r\ |_{\text{freeze}}$ stating that there is no statistical difference between the resolution rates before and during the freeze period. We could reject $H0_3^r\ |_{\text{freeze}}$ ($p < 0.001$) with the largest possible effect size ($d = 1$), since for any given release the resolution rate is higher during the development period than during the freeze period. Fig. 13 shows that the *fixing rate* is increasing over releases during the development period and feature freeze period. A linear regression analysis confirms an increasing linear trend for $FixRate(d^{<\text{freeze}}(r))$ ($R^2 = 0.9$) as well as for $FixRate(d^{>\text{freeze}}(r))$ ($R^2 = 0.58$). Both curves are overlapping over all releases; we do not observe any difference between $FixRate(d^{<\text{freeze}}(r))$ and $FixRate(d^{>\text{freeze}}(r))$. Using the Wilcoxon rank sum we test the null hypothesis $H0_3^f\ |_{\text{freeze}}$ stating that there is no statistically significant difference between fixing rates before and during the feature freeze period. $H0_3^f\ |_{\text{freeze}}$ could not be rejected ($p = 0.78$), indicating that no difference can be reported between the bug fixing rate in the development and feature freeze period.

The feature freeze period aims to focus on fixing bugs and improving the overall stability of the upcoming release. We, therefore, hypothesise that more effort is spent on fixing bugs during this period than during the development period. We quantify the *bug fixing effort* as the weekly average

number of fixed bugs in the considered period. Taking a weekly average allows us to account for the shorter total duration of the feature freeze and development periods for quarterly than for annual releases. Fig. 14 clearly suggests that bug fixing effort is higher during the feature freeze period than during the development period except for releases 4.4 and 4.12. In release 4.4, Java 8 was supported, and we find that a large number of bugs was reported and fixed for *JDT*,[18] that worked on Java 8 tooling in Eclipse.[19] This can explain the high effort during the development period for release 4.4. We attribute the low observed effort for the feature freeze period of release 4.12 to a low number of bugs that were reported for this release. A Wilcoxon rank sum test confirms our observation. We could reject the null hypothesis $H0_3^{\text{effort}}$, stating that there is no statistically significant difference between the bug fixing effort in the development and the feature freeze period, with high significance ($p < 0.001$) and large effect size ($d =$ -0.8).
In addition to the above, the difference in bug fixing effort between development and feature freeze periods appears to become more pronounced after the transition from an annual to a quarterly release cycle. This suggests that the rapid release cycle has shifted more of the bug handling effort to the feature freeze period. Unfortunately, we do not have sufficient data points yet to confirm this hypothesis.

Another aspect of the feature freeze period is that more intense testing is being carried out. These tests lead to more bugs being reported during the feature freeze period. To verify this, we investigate if maintainers during the feature freeze period focus more on fixing bugs that were reported during this period as opposed to during the development period. Considering the bugs targeting the next release r, and considering only those bugs fixed during the feature freeze period $d^{>\text{freeze}}(r)$, we group them into two categories based on when they have been reported: during the development period $d^{<\text{freeze}}(r)$ or during the feature freeze period $d^{>\text{freeze}}(r)$. Fig. 15 shows that more bugs reported in the feature freeze period (red line) are being fixed compared to those having been reported during the development period (blue line). We verified this using a Wilcoxon rank sum test with null hypothesis $H0_3^f \mid_{freeze}$ stating that, for the number of bugs fixed during the feature freeze period,

[18] `https://projects.eclipse.org/projects/eclipse/reviews/4.4.0-release-review`

[19] `https://eclipsesource.com/blogs/2014/03/25/eclipse-support-for-java-8/`

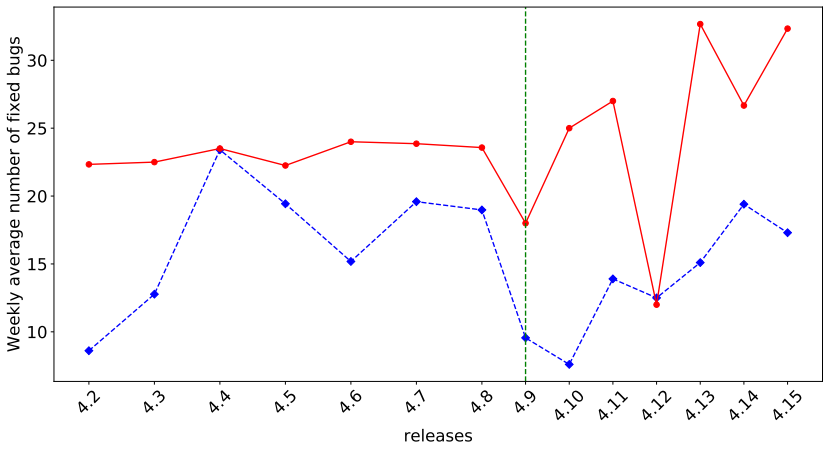


Figure 14: Comparison of *bug fixing effort* (weekly average number of fixed bugs) between development period (dashed blue lines) and feature freeze period (straight red lines) for each release.

there is no statistically significant difference between the ones created in the feature freeze period and those created in the development period. We could reject $H0_3^f \mid_{freeze}$ with high significance ($p < 0.001$) and the largest possible effect size ($d = 1$).

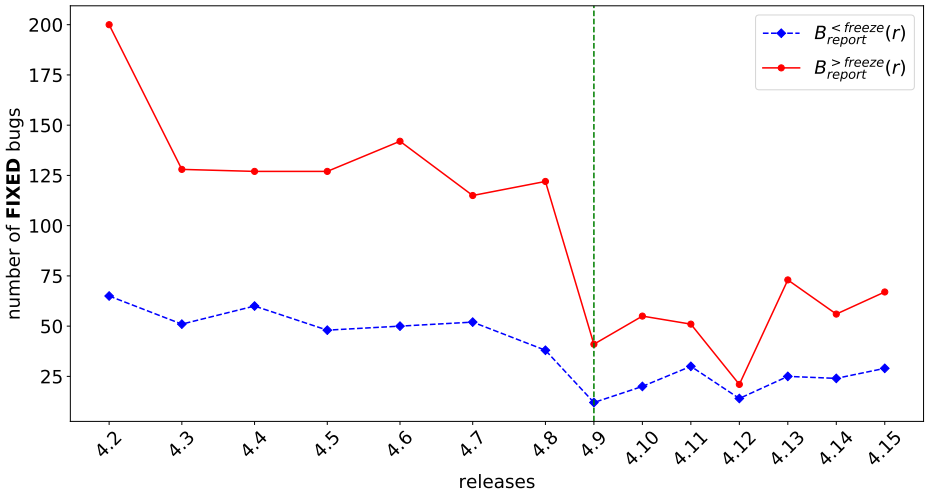


Figure 15: Evolution of number of *fixed* bugs during the feature freeze period for each 4.x release. The red line corresponds to bugs reported in the feature freeze period and the blue line to bugs reported in the development period.

Summary: There is no observable difference in fixing rate between the development period and feature freeze period of each release. However, the feature freeze period focuses more on bugs being reported in that same period than on bugs reported earlier. Moreover, more effort is spent on fixing bugs during the feature freeze period than during the development period. This difference in effort appears to have increased for quarterly releases.

*RQ*2.2 ***How does the feature freeze period impact bug handling time?***

During the feature freeze period, maintainers focus on testing and bug fixing before delivering the next release. This research question studies if, during the development period $d^{<\text{freeze}}(r)$ and the feature freeze period $d^{>\text{freeze}}(r)$ of the next release r, there is a difference in resolution time and fixing time for bugs corresponding to the current release $prev(r)$ and bugs corresponding to the next release r. On the one hand, maintainers are still involved in handling bugs of the *current* release $prev(r)$ since it may still have unresolved bugs after its release date, and new additional bugs may have been reported by its

users after the release. On the other hand, maintainers will also be involved in handling bugs of the *next* release r for which they aim to resolve as many bugs as possible before its release date.

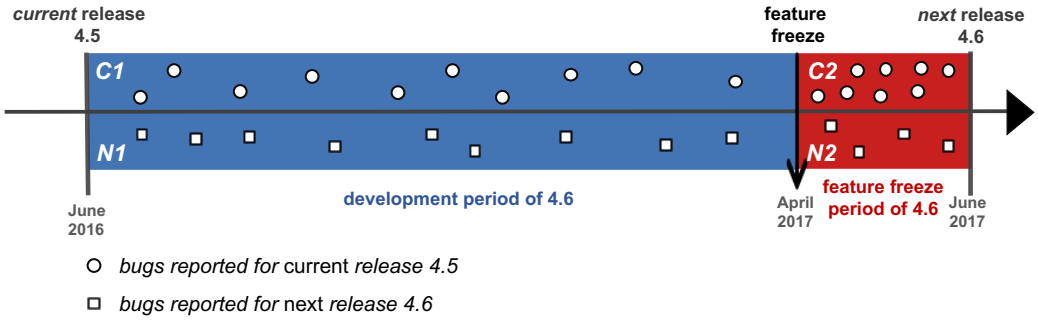


Figure 16: Categories of bugs considered for $RQ2.2$

Fig. 16 visually represents all these cases, assuming for the sake of the example that next release $r = 4.6$ and current release $prev(r) = 4.5$. The reported bugs for current release 4.5 are represented as white circles, and those for next release 4.6 as white rectangles. In the analysis for $RQ2.2$ we distinguish between four categories of bugs:

C1 All bugs of current release $prev(r)$ reported during development period $d^{<\text{freeze}}(r)$ of the next release r.

C2 All bugs of current release $prev(r)$ reported during feature freeze period $d^{>\text{freeze}}(r)$ of the next release r.

N1 All bugs of next release r reported during its development period $d^{<\text{freeze}}(r)$.

N2 All bugs of next release r reported during its feature freeze period $d^{>\text{freeze}}(r)$.

For each release r, and for each of these four categories, we compute triaging time $T_{\text{triage}}(b)$ for each bug $b \in B^{<\text{freeze}}_{\text{assigned}}(r)$ and $b \in B^{>\text{freeze}}_{\text{assigned}}(r)$. We compute, in a similar way, bug fixing time $T_{\text{fix}}(b)$ for each bug $b \in B^{<\text{freeze}}_{\text{fix}}(r)$ and $b \in B^{>\text{freeze}}_{\text{fix}}(r)$.

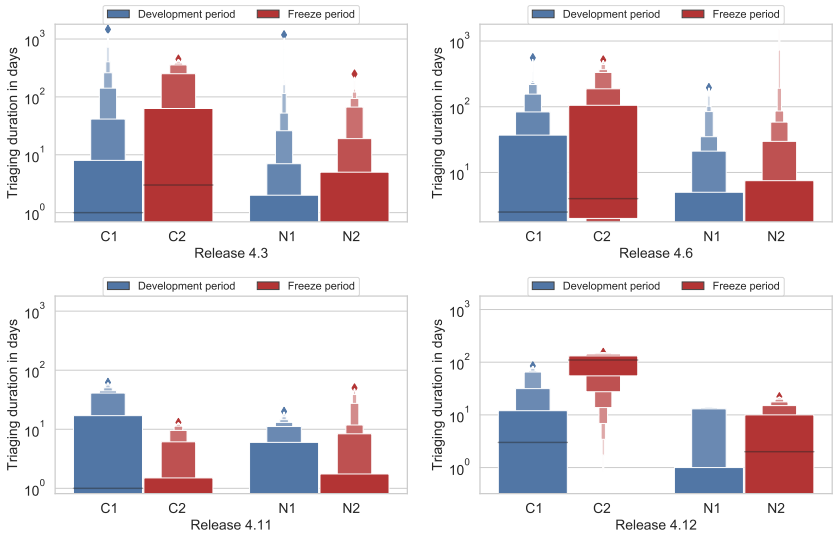


Figure 17: Boxen plots of triaging time distributions during the development and feature freeze period for each release.

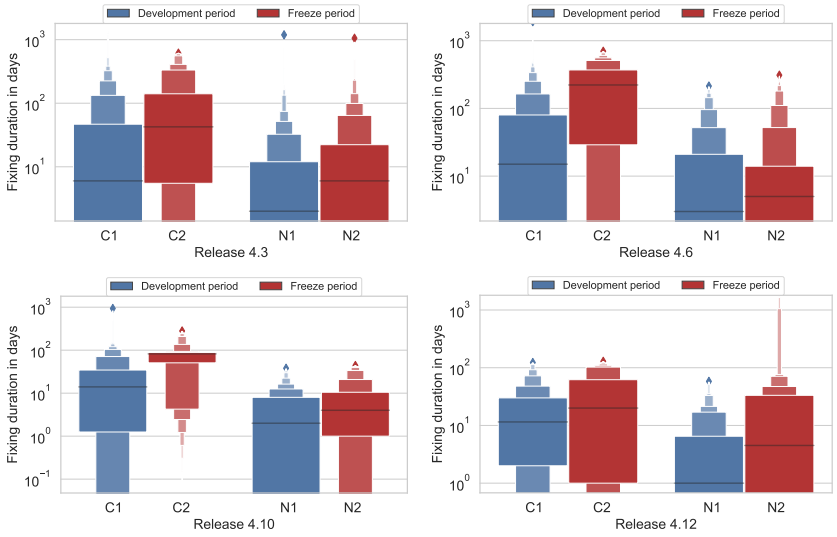


Figure 18: Boxen plots of fixing time distributions during the development and feature freeze period for each release.

(a) Comparing bugs during the feature freeze period. During the feature freeze period $d^{>\text{freeze}}(r)$ we compare the triaging and fixing time of bugs of the *current* release (C2) to those of the *next* release (N2). We visually

	triaging time			fixing time		
release	$H0^t_{4a}$	$H0^t_{4b}$	$H0^t_{4c}$	$H0^f_{4a}$	$H0^f_{4b}$	$H0^f_{4c}$
Eclipse 4.x annual releases						
4.3	$\mathbf{S}^{*}$	$\mathbf{S}^{**}$	N^{*}	$\mathbf{M}^{***}$	$\mathbf{S}^{***}$	$\mathbf{S}^{***}$
4.4	–	–	–	$\mathbf{L}^{***}$	$\mathbf{M}^{***}$	N^{***}
4.5	$\mathbf{L}^{***}$	$\mathbf{S}^{**}$	–	$\mathbf{M}^{***}$	$\mathbf{S}^{*}$	–
4.6	$\mathbf{M}^{***}$	$\mathbf{S}^{*}$	–	$\mathbf{L}^{***}$	$\mathbf{L}^{***}$	–
4.7	$\mathbf{S}^{*}$	–	–	$\mathbf{L}^{***}$	$\mathbf{L}^{***}$	–
4.8	–	–	$\mathbf{S}^{**}$	$\mathbf{L}^{***}$	$\mathbf{M}^{***}$	–
Eclipse 4.x quarterly releases						
4.9	–	–	–	$\mathbf{L}^{***}$	–	–
4.10	–	–	–	$\mathbf{L}^{***}$	$\mathbf{L}^{***}$	$\mathbf{S}^{*}$
4.11	–	–	–	$\mathbf{S}^{**}$	–	$\mathbf{S}^{**}$
4.12	–	–	$\mathbf{M}^{*}$	–	–	$\mathbf{S}^{*}$
4.13	–	–	–	$\mathbf{S}^{**}$	–	$\mathbf{S}^{**}$
4.14	–	–	$\mathbf{S}^{*}$	$\mathbf{M}^{**}$	–	–

Table 4: Mann-Whitney U test for difference between development and feature freeze period for **triaging time** and **fixing time**. – indicates that the null hypothesis could not be rejected. Whenever it could be rejected, cell values summarise the significance level α (* =0.05; ** =0.01; ***=0.001) and the effect size: N(egligible), **S**(mall), **M**(edium), **L**(arge).

observe that during this feature freeze period, bugs for the next release are triaged (Fig. 17) and fixed (Fig. 18) *faster* than for the current release. This can be explained by the fact that the feature freeze period aims to triage and fix the bugs of the next release fast and to deliver the release with as few bugs as possible.

For triaging time, Mann-Whitney U tests (see Table 4) confirm that the null hypothesis $H0^t_{4a}$, stating that the time to triage bugs of the current and next release is not different during the feature freeze period, can be rejected for all annual releases except 4.4 and 4.8. For the quarterly releases, $H0^t_{4a}$ could not be rejected. Similarly, for fixing time we verify the null hypothesis $H0^f_{4a}$ that the time to fix bugs of the current and next release is the same during the feature freeze period. $H0^f_{4a}$ can be rejected for all releases except 4.12 (see Table 4) with *small* effect size for release 4.11 and 4.13, *medium* for releases 4.3, 4.5 and 4.14, and *large* for the others. Our results indicate that bugs of the next release are fixed faster in the feature freeze period compared to the bugs of the current release.

(b) Comparing bugs of the current release during development and feature freeze period of the next release. We investigate the differences in triaging and fixing time of bugs of the current release $prev(r)$ between the development period $d^{<\text{freeze}}(r)$ (C1 in Fig. 17) and the feature freeze period $d^{>\text{freeze}}(r)$(C2 in Fig. 17) of the next release r. We observe a longer triaging and fixing time during the feature freeze period than during the development period. These results indicate that bugs triaged and fixed during the feature freeze period have been open for a long time and that maintainers tend to focus on bugs that had lived for a long time in the current release before releasing the next one.

For triaging time, Mann-Whitney U tests (see Table 4) verify the null hypothesis $H0^t_{4b}$, stating that there is no difference in triaging time of bugs of the current release between the development period and the feature freeze period. $H0^t_{4b}$ can be rejected with small effect size for all annual releases except 4.4, 4.7 and 4.8. For the quarterly releases, the null hypothesis could not be rejected. For fixing time, the null hypothesis $H0^f_{4b}$, stating that there is no difference in fixing time of bugs of the current release between the development period and the feature freeze period, can be rejected for all annual releases. For the quarterly releases, the null hypothesis cannot be rejected except for 4.10.

Therefore, for all annual releases, bugs of the current release took longer to fix during the feature freeze period compared to the development period. This is not the case anymore for the quarterly releases.

(c) Comparing bugs of the next release during its development and feature freeze period. We investigate the differences in triaging and fixing time of bugs of the next release r between its development period $d^{<\text{freeze}}(r)$ (N1 in Fig. 17) and its feature freeze period (N2 in Fig. 17). For bug triaging time, we do not observe a difference between the feature freeze period and the development period. Maintainers appear to triage bugs of the next release as soon as possible, regardless of the considered period. Mann-Whitney U tests (Table 4) verify the null hypothesis $H0^t_{4c}$, stating that there is no difference in *triaging time* of the next release between the development period and the feature freeze period. $H0^t_{4c}$ cannot be rejected for the annual releases except for 4.3 and 4.8. Even for those releases where $H0^t_{4c}$ can be rejected, the effect size is *negligible* or *small* (for 4.8). For the quarterly release, $H0^t_{4c}$ cannot be rejected for the releases except for 4.12 and 4.14 with a medium and small effect respectively. This implies that there is

little or no difference in bug triaging time of the next release between the development period and the feature freeze period.
For bug fixing time, we visually observe a longer time during the feature freeze period than during the development period. We verified the null hypothesis $H0^f_{4c}$, stating that there is no difference in bug fixing time of the next release during the development period and the feature freeze period. $H0^f_{4c}$ can be rejected only in release 4.3 and 4.4 for annual releases. The hypothesis was rejected for the quarterly releases except 4.9 and 4.14. For all releases where $H0^f_{4c}$ can be rejected, the effect size is *small* except for 4.4 it is negligible. This implies that, with the exception of those releases, there is very little difference in the time to fix bugs of the next release between the development period and the feature freeze period.
We also tested all the previous hypotheses after grouping per bug severity level, but we found similar results to those in Table 4.[20] It takes less time to triage and fix bugs during the feature freeze period of the next release compared to the current release. Bugs of the current release took longer to fix during the feature freeze period compared to the development period for annual releases but it is not the case for the quarterly releases. The feature freeze period does not affect the triaging and fixing time of bugs for the next release. As a result, the bug severity does not seem to impact the results.
In contrast to our quantitative findings, four out five of the consulted Eclipse maintainers stated that bugs are prioritized according to their severity.

Summary: It takes less time to triage and fix bugs during the feature freeze period of the next release compared to the current release. Bugs of the current release that are triaged and fixed during the feature freeze period have stayed open longer compared to those in the development period for the annual releases while this is not the case anymore for the quarterly releases. The feature freeze period does not affect the triaging and fixing time of bugs for the next release. The severity of the bugs handled does not influence the results.

7. Discussion

Evolution of the bug handling process of Eclipse.

[20] The analysis per severity result can be found in our replication package [25].

Changes in the bug handling process can have a direct impact on the efficiency of bug handling activities such as triaging, fixing and resolving bugs. As an example, as illustrated in Section 4, the Eclipse Core project developers, since June 2010, stopped using resolution statuses such as `LATER` or `REMIND` that signal that the bug will be processed later on. This decision led to a natural decrease in the resolution rate, while the fixing rate increased. In fact, these resolutions appeared to be problematic for the Mozilla community as well[21] and although these resolutions remained available later on, they no longer appeared in the default list of resolutions since Bugzilla 4.0, that was released on 15 February 2011[22]. Another example of such changes is the introduction of automated tools that helped the community in handling bugs. 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 [45] commented bug reports are more than twice as likely to be fixed compared to those without user comments. Our empirical analysis has confirmed the positive effects of AERI on the bug fixing rate.

> Before changing their bug handling process, project communities should carefully assess the pros and cons of such changes upfront, plan ahead these changes to avoid negative impacts, and measure after changing the process if the targeted improvements have been reached. Researchers need to be aware of such changes in the bug handling process when carrying out empirical studies, since the effect of these changes may play an important role in the obtained results, as we have clearly observed with our Eclipse case study.

Bug handling during feature freeze period. We observed a clear difference in bug handling activity between the feature freeze period and development period of releases. The results of *RQ*2.2 revealed that, during feature freeze, maintainers focus more on triaging and fixing bugs of the next release than on those of the current release. Also, the results of *RQ*2.1 showed that

[21] A relevant discussion between Mozilla developers can be found at `https://bugzilla.mozilla.org/show_bug.cgi?id=35839`

[22] `https://www.bugzilla.org/releases/4.0/release-notes.html`

maintainers focus more on fixing bugs reported during the feature freeze period than on bugs reported earlier. In addition, during feature freeze more effort is spent on fixing bugs. While the consulted Eclipse maintainers claimed to tackle bugs of high priority during feature freeze, we observed the same proportion of severe bugs being fixed as was the case during the development period. Moreover, bugs of the current release that are triaged during the feature freeze 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.

> Feature freeze periods impose extra stress during rapid releases since they tend to be relatively short and, in addition, they take away an important part of the time that could otherwise be devoted to development of new features. Nevertheless, feature freeze periods need to be preserved, since they allow to spend more focused effort fixing bugs for the upcoming release. In addition to this, they should invest in test automation, especially in the presence of rapid releases.

Benefits and challenges in switching to a more rapid release policy. All five consulted Eclipse maintainers found the transition to a quarterly release cycle very beneficial w.r.t. the time it takes for bugs to be fixed and new features to reach users. They also indicated that it helped to reduce stress and increase community growth. One maintainer stated *"Now, if I miss a deadline, it's not the end of the world, and I don't have to rush as much..."* and another one said *"Bug fixes get out in a timely manner. No more backporting of bugfixes from master to a service release."* Overall, the consulted maintainers claimed not to have faced many difficulties because of the transition, and they agreed that faster releases lead to faster feature and bug fix integration, increasing user benefit and making development more efficient. On the downside, they mentioned that beta testing on a release decreased due to less milestone releases per cycle (3 as opposed to 7 for the annual releases, cf. Fig. 1 & Fig. 2) and lack of time to contribute larger features. They also found it harder to keep development environments up-to-date and expressed the need to reorder tasks so that more release engineering work can be automated. They highlighted the importance of good management of their platform, as well as the fact that instability and regressions might occur.

Khomh et al. [10, 15] studied the effect of the transition to a rapid release

cycle on the bug handling activity in a different project, Mozilla Firefox. Considering a period of two years of bug activity, they found this transition to lead to shorter bug fixing times, but on the downside, fewer bugs were being triaged and fixed in a timely fashion. Our results for the more recent switch of Eclipse to a quarterly release schedule partially align with these findings, as we also observed fewer bugs being triaged and fixed after the transition Section 4. Different from Firefox, however, we observed an increased fixing rate after the transition ($RQ1.1$) and bugs being fixed faster ($RQ1.2$). A possible explanation for this difference is that Firefox developers were not given enough time to prepare for the transition to smaller release cycles [15]. In contrast, the Eclipse community has been preparing the transition for over a year [47] by starting to introduce intermediate quarterly "update" releases since Eclipse 4.6 in 2016. Two of the consulted Eclipse maintainers clearly indicated that these update releases were part of the preparation. Carefully planning the transition to quarterly releases was therefore essential to its successful adoption.

As part of the preparation towards a more rapid release cycle, the consulted Eclipse maintainers highlighted the importance of a good testing plan. Adopters of rapid releases should carry out test all along the release development cycle, not only during the feature freeze period. They should also heavily invest in adequate automated tooling and support processes, especially for continuous integration and deployment, to make it "just work".

> In the context of ever more projects moving to faster release cycles [11, 13], the findings for Mozilla and Eclipse highlight that careful preparation and planning is key for a successful transition to faster release cycles. It enables the developer community to become more effective in bug handling activities, provided the presence of a good testing plan, a good release management policy, and adequate automated tooling and support processes. Other projects can benefit from these lessons learned.

What is the most adequate release duration? Projects that have adopted a rapid release policy have done so with cycles of different durations. For example, Eclipse opted for a 13-week cycle, while Firefox opted for a much shorter 4-week cycle. It remains an open question what constitutes the most optimal duration, and the answer will be specific to each project. We consulted the Eclipse maintainers about their opinion and four out of five

responded that 13 weeks is an adequate duration, mentioning benefits such as "*short enough for features and bugfixes to get to users at adequate speed*" and "*long enough to allow using single stream of development thus saving the team for branch merging*". However, one of these respondents was concerned that "*a lot of process time goes into building and shipping each version, and too little time is devoted to automated testing*". Another respondent would favour an even higher release frequency, but acknowledges that this "*would require way more automation, standardization etc. of all the releases train projects, and that seems impossible with the rather loosely coupled projects at eclipse.org*". In contrast, yet another maintainer signaled that even 13 weeks is already a very short period of time. In future research, we aim to investigate how the type of release cycle (fixed or variable) can impact the bug handling activity.

> The 13-week release cycle of Eclipse appears to be a good compromise; further shortening the release duration may be challenging and the added value of doing so is yet unknown.

8. Threats to Validity

Following the structure recommended by [48], we discuss the threats that may have affected the validity of our findings, and how we have tried to mitigate them.

A threat to *construct validity* in our study concerns the bug assignment identification. We minimized this threat by identifying bug assignments based on both the *Status* field (`ASSIGNED`) and the alternative practice of assigning bugs through [component_name]-triaged@eclipse.org (cf. Section 3.2).

A possible threat to *internal validity* is that our analysis only relied on Bugzilla bug reports. However, we verified that all Eclipse Core bugs are handled through Bugzilla; even the bugs submitted using AERI are stored in Bugzilla. Another threat stems from bugs not having a *Version* field, or having a *Version* field not corresponding to any of the considered releases. We excluded such bugs as it is not possible to automatically deal with such cases. A third known threat [32] concerns the presence of multiple occurrences of the same activity in a bug report, such as a bug that may be reassigned, a bug that may have had multiple resolutions or fixes, the *Version* field value that changes over time, and the *Severity* level that may be modified. We mitigated this threat by considering only the date of the *first* assignment

(reflecting the moment when the bug was triaged for the first time), the date of the *last* resolution/fixing activity (reflecting the fact that prior resolutions/fixes of the bug were not satisfactory), the last reported *Version* field value, and the latest *Severity* level of the bug.

We quantified the possible bias stemming from bugs tagged with different major versions throughout their history. We found that 4,266 bugs were reassigned to different releases throughout their history, out of which 3,973 bugs were reassigned to different major releases. From these bugs, only 21 out of 2,741 `RESOLVED` bugs are resolved in multiple major Eclipse releases, thus, the bias they introduce is insufficient to alter our findings.

Concerning possible bias due to changes in the bug severity, we found 2,016 bug reports that changed their severity over time, out of which 1,421 bugs (5% w.r.t the total number of considered bugs in the 4.x series) being reassigned to a different severity category, thus the impact on our analyses is minimal.

A fourth threat concerns our study in $RQ2.1$ and $RQ2.2$. We study the fixed bugs targeting only the upcoming release during its feature freeze, however, other bugs are fixed during this period that target other releases than the upcoming one. We measured the percentage of fixed bugs that target the upcoming release compared to the ones targeting other releases. We find that the majority of the fixed bugs during the feature freeze period of an upcoming release target it, thus, the impact of such threat is likely to be minimal.

A fifth threat concerns the presence of Eclipse Genie, an automated bot, that closes bugs that have not had any activity for a long time. It closes bugs assuming that the problem got resolved, was a duplicate of something else, or became less pressing or maybe it is still relevant but has not been triaged yet. This bot is used so that these bugs will not appear open anymore for maintainers. Starting from 2020, Genie closes bugs from the 4.x annual releases that have been open for a long time. In our analysis, bugs that are closed by Genie are not considered as resolved. Thus, our quantitative analysis is not affected by the presence of Genie.

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 valid for the Core projects that have a large number of bugs and an active developer community. The analysis and findings will differ for smaller

and/or peripheral projects within Eclipse that have different versioning, release policies and evolutionary dynamics.
We mitigate threats to *reliability validity* by providing a publicly available replication package [25] and a detailed description of the followed methodology in Section 3.

9. Future Work

As future work we consider comparing the effect of changes in release policy between the Eclipse Core and other Eclipse sub-projects and plugins. We also aim to study the impact of rapid releases on long-lived bugs and on reopening of bugs. We plan to compare the findings for Eclipse with other competing projects such as Netbeans, that share similar functionalities and maturity.
We also intend to compare Eclipse with younger and less mature OSS projects that have less established policies and practices. This will allow us to investigate how the bug handling process differs, and to provide best practices for improving bug handling in those projects, especially in the presence of rapid release cycles. Such projects can be found in the recent dataset of Joshi et al. [13]. Based on this dataset, we also aim to investigate the effect of variable durations of rapid releases on bug handling activity.
More research is also needed to better understand the impact of changes in the release cycle on other aspects of collaborative software development. For example, how do rapid releases impact code reviewing, the presence of technical debt, the involvement and productivity of developers, software testing and quality, and so on? A deeper understanding of the impact of rapid releases on these aspects will allow software organisations and developer communities to make informed decisions on whether and when to safely switch to a rapid release model.

10. Conclusion

We conducted an empirical study of bug handling activity in the Eclipse Core projects. We focused the study on the 4.x release range, featuring a transition to a more rapid release cycle as of release 4.9. We compared seven annual releases before this transition to seven quarterly releases after the transition. 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. We also studied the impact of the *feature freeze* period on these metrics.

For the annual releases, the number of reported bugs per release is decreasing over time, and for the quarterly releases it is stabilising at a low value, suggesting that the Eclipse bug handling process is mature and of high quality. This difference is no longer observed for the quarterly releases, mostly because of a smaller number of reported bugs. We also could not observe a difference between bugs reported before and after a release in term of triaging and fixing time.

While the bug resolution rate is decreasing over time to rather low values ($< 50\%$, implying that less than 1 out of 2 bugs gets resolved for recent releases) 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 introduction of the automated AERI error reporting tool.

We observed more intense bug handling activity during the feature freeze periods, where bugs are triaged and fixed faster, and priority is being given to fixing bugs of the next release as opposed to bugs of the current release. During these periods, more effort is being spent on bug fixing, and this is maintained after the transition to quarterly releases. In our study, we did not find any measurable effect of the bug severity on the bug handling process.

The transition from an annual to a quarterly release cycle has allowed the Eclipse Core projects to have a more stable bug handling process since some observed differences in triaging and fixing times and rates before and after annual releases are no longer present for the quarterly releases. Moreover, we did not observe any negative effect of the switch to quarterly releases. We, therefore, believe that the transition to a more rapid release cycle has been beneficial to Eclipse in terms of bug handling activity. This was confirmed by feedback from five consulted Eclipse maintainers. It remains an open question if even faster release cycles would continue to yield further benefits or on the contrary would have negative consequences.

The success story of Eclipse has shown that feature freeze periods and faster release cycles can be beneficial for well-managed software projects, provided that Eclipse has already put in place a well-defined bug handling process. Switching to more rapid releases requiring careful planning and tracking the transition, and being aware of the possible pitfalls. Adopters of rapid releases should test and fix bugs as soon and as frequently as possible. They should raise awareness to their developer community, invest in the most appropriate

tooling and support processes, and automate as much as possible.

Acknowledgment

This research was partially supported by the Fonds de la Recherche Scientifique – FNRS under Grant numbers O.0157.18F-RG43 (Excellence of Science project "SECO-Assist") and T.0017.18. Zeinab Abou Khalil is partially supported by Région Hauts-de-France. We thank the five Eclipse Core maintainers that have been consulted for their feedback on our research findings, and Prof. Alexander Serebrenik for his relevant insights that helped us to improve our research.

References

[1] J. Bosch (Ed.), Continuous Software Engineering, Springer, 2014. doi:10.1007/978-3-319-11283-1.

[2] M. K. Ndenga, I. Ganchev, J. Mehat, F. Wabwoba, H. Akdag, Performance and cost-effectiveness of change burst metrics in predicting software faults, Knowledge and Information Systems 60 (1) (2019) 275–302. doi:10.1007/s10115-018-1241-7.

[3] D. A. da Costa, S. McIntosh, U. Kulesza, 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, IEEE, 2016, pp. 374–385. doi:10.1145/2901739.2901764.

[4] VersionOne, 7th annual state of agile survey (2013). URL https://www.stateofagile.com/#ufh-i-338592786-7th-annual-state-of-agile-report/473508

[5] B. Fitzgerald, Open source software: Lessons from and for software engineering, Computer 44 (10) (2011) 25–30. doi:10.1109/MC.2011.266.

[6] M. Michlmayr, Quality improvement in volunteer free and open source software projects: Exploring the impact of release management, Ph.D. thesis, University of Cambridge (March 2007).

[7] R. Van Solingen, V. Basili, G. Caldiera, H. D. Rombach, Goal question metric (gqm) approach, Encyclopedia of software engineering (2002).

[8] Z. Abou Khalil, E. Constantinou, T. Mens, L. Duchien, C. Quinton, A longitudinal analysis of bug handling across Eclipse releases, in: International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019. doi:10.1109/ICSME.2019.00010.

[9] A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals, Predicting the severity of a reported bug, in: Working Conference on Mining Software Repositories, IEEE, 2010, pp. 1–10. doi:10.1109/MSR.2010.5463284.

[10] F. Khomh, T. Dhaliwal, Y. Zou, B. Adams, Do faster releases improve software quality? An empirical case study of Mozilla Firefox, in: orking

Conf. Mining Software Repositories, IEEE, 2012, pp. 179–188. doi:10.1109/MSR.2012.6224279.

[11] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, K. Petersen, On rapid releases and software testing: a case study and a semi-systematic literature review, Empirical Software Engineering 20 (5) (2015) 1384–1425. doi:10.1007/s10664-014-9338-4.

[12] T. Zimmermann, Challenges in the collaborative evolution of a proof language and its ecosystem, Ph.D. thesis, Université de Paris (2019).

[13] S. D. Joshi, S. Chimalakonda, RapidRelease – a dataset of projects and issues on Github with rapid releases, in: International Conference on Mining Software Repositories (MSR), IEEE, 2019, pp. 587–591. doi:10.1109/MSR.2019.00088.

[14] M. Nayebi, G. Ruhe, T. Zimmermann, Mining treatment-outcome constructs from sequential software engineering data, IEEE Transactions on Software Engineering (2019).

[15] F. Khomh, B. Adams, T. Dhaliwal, Y. Zou, Understanding the impact of rapid releases on software quality, Empirical Software Engineering 20 (2) (2015) 336–373. doi:10.1007/s10664-014-9308-x.

[16] D. A. da Costa, S. McIntosh, C. Treude, U. Kulesza, A. E. Hassan, The impact of rapid release cycles on the integration delay of fixed issues, Empirical Software Engineering (2018) 1–70doi:10.1007/s10664-017-9548-7.

[17] R. K. Saha, S. Khurshid, D. E. Perry, Understanding the triaging and fixing processes of long lived bugs, Information and software technology 65 (2015) 114–128. doi:10.1016/j.infsof.2015.03.002.

[18] F. Zhang, F. Khomh, Y. Zou, A. E. Hassan, An empirical study on factors impacting bug fixing time, in: Working Conference on Reverse Engineering, IEEE, 2012, pp. 225–234. doi:10.1109/WCRE.2012.32.

[19] P. Hooimeijer, W. Weimer, Modeling bug report quality, in: International Conference on Automated Software Engineering (ASE), ACM, 2007, pp. 34–43. doi:10.1145/1321631.1321639.

[20] L. D. Panjer, Predicting Eclipse bug lifetimes, in: International Workshop on Mining Software Repositories, IEEE Computer Society, 2007. doi:10.1109/MSR.2007.25.

[21] E. Giger, M. Pinzger, H. Gall, Predicting the fix time of bugs, in: International Workshop on Recommendation Systems for Software Engineering, ACM, 2010, pp. 52–56. doi:10.1145/1808920.1808933.

[22] L. Marks, Y. Zou, 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. doi:10.1145/2020390.2020401.

[23] W. Zou, X. Xia, W. Zhang, Z. Chen, D. Lo, An empirical study of bug fixing rate, in: Computer Software and Applications Conference (COMPSAC), IEEE, 2015, pp. 254–263. doi:10.1109/COMPSAC.2015.57.

[24] R. Rwemalika, M. Kintis, M. Papadakis, Y. Le Traon, P. Lorrach, An industrial study on the differences between pre-release and post-release bugs, in: International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019, pp. 92–102. doi:10.1109/ICSME.2019.00019.

[25] Z. A. Khalil, E. Constantinou, T. Mens, On the impact of release policies on bug handing activity: A case study of eclipse [replication package], https://zenodo.org/record/3762771 (April 2020). doi:10.5281/zenodo.3762771.

[26] T. Mens, J. Fernandez-Ramil, S. Degrandsart, The evolution of Eclipse, in: International Conference on Software Maintenance, 2008, pp. 386–395. doi:10.1109/ICSM.2008.4658087.

[27] J. Businge, A. Serebrenik, M. van den Brand, Survival of Eclipse third-party plug-ins, in: International Conference on Software Maintenance, IEEE, 2012, pp. 368–377. doi:10.1109/ICSM.2012.6405295.

[28] A. Kumar, A. Gupta, Evolution of developer social network and its impact on bug fixing process, in: India Software Engineering Conference, ACM, 2013, pp. 63–72. doi:10.1145/2442754.2442764.

[29] Eclipse Foundation, Eclipse Bugzilla repository (2019).
URL https://bugs.eclipse.org

[30] C. Guindon, Core – The Eclipse Foundation.
URL https://www.eclipse.org/eclipse/platform-core/

[31] R. K. Saha, J. Lawall, S. Khurshid, D. E. Perry, Are these bugs really "normal"?, in: Working Conference on Mining Software Repositories, 2015, pp. 258–268. doi:10.1109/MSR.2015.31.

[32] F. Tu, J. Zhu, Q. Zheng, 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), ACM, 2018, pp. 307–318. doi:10.1145/3236024.3236054.

[33] L. A. F. Gomes, R. da Silva Torres, M. L. Côrtes, Bug report severity level prediction in open source software: A survey and research opportunities, Information and software technology 115 (2019) 58–78.

[34] P. Louridas, D. Spinellis, V. Vlachos, Power laws in software, ACM Trans. Softw. Eng. Methodol. 18 (1) (Oct. 2008). doi:10.1145/1391984.1391986.
URL https://doi.org/10.1145/1391984.1391986

[35] A. Murgia, I. Turnu, M. Marchesi, R. Tonelli, G. Concas, On the distribution of bugs in the eclipse system, IEEE Transactions on Software Engineering 37 (06) (2011) 872–877. doi:10.1109/TSE.2011.54.

[36] M. Hollander, D. A. Wolfe, E. Chicken, Nonparametric statistical methods, 3rd Edition, Vol. 751, John Wiley & Sons, 2015. doi:10.1002/9781119196037.

[37] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions, Psychological Bulletin 114 (3) (1993) 4994–509. doi:10.1037/0033-2909.114.3.494.

[38] M. Hess, 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).

[39] O. Aalen, O. Borgan, H. Gjessing, Survival and event history analysis: a process point of view, Springer Science & Business Media, 2008. doi:10.1007/978-0-387-68560-1.

[40] I. Samoladas, L. Angelis, I. Stamelos, Survival analysis on the duration of open source projects, Information and Software Technology 52 (9) (2010) 902 – 922. doi:10.1016/j.infsof.2010.05.001.

[41] A. Decan, T. Mens, E. Constantinou, On the impact of security vulnerabilities in the npm package dependency network, in: International Conference on Mining Software Repositories, ACM, 2018, pp. 181–191. doi:10.1145/3196398.3196401.

[42] E. Kaplan, P. Meier, Nonparametric estimation from incomplete observations, J. American Statistical Association 53 (282) (2012) 457–481. doi:10.23072281868.

[43] J. M. Bland, D. G. Altman, The logrank test, BMJ 328 (7447) (2004) 1073. doi:10.1136/bmj.328.7447.1073.

[44] H. Hofmann, H. Wickham, K. Kafadar, Letter-value plots: Boxplots for large data, Journal of Computational and Graphical Statistics 26 (3) (2017) 469–477. doi:10.1080/10618600.2017.1305277.

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

[46] M. Gupta, A. Sureka, Nirikshan: Mining bug report history for discovering process maps, inefficiencies and inconsistencies, in: Proceedings of the 7th India Software Engineering Conference, 2014, pp. 1–10.

[47] M. Barbero, Simultaneous release brainstorming (27 February 2018).
URL https://www.eclipse.org/lists/eclipse.org-planning-council/msg02927.html

[48] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering - An Introduction, Kluwer, 2000. doi:10.1007/978-1-4615-4625-2.