Noname manuscript No.
(will be inserted by the editor)

An Empirical Study of Delays in the Integration of Addressed Issues

Daniel Alencar da Costa · Shane McIntosh · Uirá Kulesza · Ahmed E. Hassan · Surafel Lemma Abebe

Received: date / Accepted: date

Abstract Predicting the time required to address an issue (*i.e.*, a new feature, bug fix, or enhancement) has long been the goal of many software engineering researchers. However, after an issue has been addressed, it must be integrated into an official release to become visible to users. In theory, issues should be quickly integrated into releases after they are addressed. However, in practice, the integration of an addressed issue might be delayed. For instance, an addressed issue might be delayed in order to assess the impact that this addressed issue may have on the system as a whole. While one can often speculate, it is not always clear why some addressed issues are integrated immediately, while others are delayed. In this paper, we empirically study the integration of 20,995 addressed issues from the ArgoUML, Eclipse, and Firefox systems. Our results indicate that: (i) despite being addressed well before the release date, the integration of 34% to 60% of addressed issues in systems with traditional release cycle (Eclipse and ArgoUML), and 98% of addressed issues in systems with rapid release cycle (Firefox) were delayed by one or more releases; (ii) using information derived from the addressed issues, our models are able to accurately predict the release in which an addressed issue will be integrated, achieving a Receiver Op-

Daniel Alencar da Costa, Uirá Kulesza
Department of Informatics and Applied Math (DIMAp)
Federal University of Rio Grande do Norte
Natal, RN, Brazil
Tel.: +55 84 3215-3814
E-mail: danielcosta@ppgsc.ufrn.br,uira@dimap.ufrn.br
Shane McIntosh
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
E-mail: shane.mcintosh@mcgill.ca
Ahmed E. Hassan, Surafel Lemma Abebe
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, Ontario, Canada
Tel.: +1 613-533-6802
E-mail: {ahmed, surafel}@cs.queensu.ca

erator Curve (ROC) area of above 0.78; and (iii) the workload of integrators is the most influential factor in our models. Furthermore, we fit models to study addressed issues that are abnormally delayed. We find that: (iv) our models can accurately predict which addressed issues will have longer integration delays than most others, achieving ROC areas above 0.87; and (v) the most influential metrics are the moment when an issue is addressed during the release cycle and, again, the workload of integrators. Our results indicate that, in addition to the backlog of issues that need to be addressed, the backlog of issues that need to be integrated introduces a software development overhead, which may lead to integration delay. Therefore, in addition to studying the triaging and addressing stages of the issue lifecycle, the integration stage should also be the target of research, tools, and techniques in order to reduce the time-to-delivery of addressed issues.

Keywords Integration Delay · Software Maintenance · Mining Software Repositories · Software Engineering

1 Introduction

Prior studies have explored several approaches to help developers to estimate the time needed to address issues (new features, enhancements, and bug fixes) [1, 2, 7, 13, 15, 22, 24]. Such studies are useful for project managers who need to allocate development resources effectively in order to deliver new releases on time without exceeding budgets.

On the other hand, users and contributors care most about when an official release of a software system will include an addressed issue. Although an issue may have been addressed, it may not be integrated into an official release for some time. Jiang *et al.* [12] find that after a change has taken 1-3 months to complete the code review process, it takes an additional 1-3 months for that change to be integrated into the Linux kernel. In this paper, we refer to the time between when an issue is addressed and when it is integrated into an official release as *integration delay*.

Although one can often speculate, it is not always clear why an addressed issue would not be integrated into an upcoming release. When the reasons for these integration delays are unclear, users and contributors may become frustrated. For example, on a recent Firefox issue, a stakeholder asked: *"So when does this stuff get added? Will it be applied to the next FF23 beta? A 22.01 release? Otherwise?"*[1]

To investigate why the integration of some addressed issues is delayed, we perform an empirical study of 20,995 issues collected from the ArgoUML, Eclipse, and Firefox systems. We investigate (1) how much delay addressed issues typically have before integration, and (2) which issues suffer from longer integration delays than most others. To that end, this paper addresses five research questions structured along these two dimensions as described below.

[1] http://goo.gl/OwzSs5

1.1 Release Integration Delay

- ***RQ1: Are issues often delayed after being addressed?*** The integration of 34% to 60% of addressed issues within traditional releasing cycles (ArgoUML and Eclipse) are delayed by at least one release. Furthermore, 98% of the addressed issues are delayed by at least one release in the rapidly released Firefox system.

- ***RQ2: Can we accurately explain how many releases an addressed issue will be delayed?*** Our models can accurately explain the integration delay of addressed issues achieving ROC areas above 0.78.

- ***RQ3: What are the most influential attributes for estimating release integration delay?*** We find that the workload of the integrators plays an influential role in estimating the release integration delay of an addressed issue. On the other hand, we find that priority and severity denoted in issue reports have little influence on release integration delay.

1.2 Abnormal integration delay

- ***RQ4: Can we accurately explain which addressed issues will take longer to be integrated than most others?*** In this regard, our models outperform random guessing, achieving ROC areas above 0.87.

- ***RQ5: What are the most influential attributes for estimating abnormally delayed issues?*** Similar to RQ3, we find that the moment when an issue is addressed during the release cycle and the workload of integrators are also the most influential attributes for identifying the issues that will suffer from abnormal integration delay.

- ***RQ6: Does the integration delay of addressed issues relates to the components that they are being modified?*** We find that there is no considerable difference between the number of days that the integration of an issue is delayed and the components of the studied systems.

Our results suggest that the integration backlog shares a strong link with both release and abnormal integration delay. Therefore, in addition to studying the triaging and addressing stages of the issue lifecycle, the integration stage should also be the target of research, tools, and techniques in order to reduce the time-to-delivery of addressed issues.

1.3 Paper Organization

This paper is an extended version of our prior work [6]. We extend our prior work to:

1. Expand the set of explanatory variables (Tables 2 and 3) to include:

   (a) The time to address an issue, *i.e.*, from OPENED to RESOLVED-FIXED.
   (b) An indicator of whether or not a stack trace has been attached to the issue report.
   (c) The moment when an issue is addressed during the current release cycle (*queue position*).

2. Study abnormally delayed fixes for addressed issues (RQ4 and RQ5 in Section 4).

3. Study the relationship between the components of the studied systems and integration delay (RQ6 in Section 4).

4. Perform an exploratory analysis of the "code freeze" stage in the integration process (Section 6).

The remainder of the paper is organized as follows. Section 2 describes the issue lifecycle. Section 3 presents our empirical study by describing the studied systems, the data collection procedures, and the dimensions that we investigate in our study, while Sections 4 and 5 present the results with respect to our release and abnormal integration delay dimensions. Section 6 discusses the role of the "code freeze" stage in integration delay. Section 7 discusses the threats to the validity of our conclusions. Section 8 positions our work with respect to previous studies. Section 9 draws conclusions and proposes avenues for future work.

2 Background & Definitions

One of the main factors that drives software evolution are the issues that are filed by users, developers, and quality assurance personnel. Below we describe what issues are and the major steps involved in addressing and integrating them.

We use the term *issue* to broadly refer to bug reports, enhancements, and feature requests. Issues can be filed by users, developers, or quality assurance personnel. To track development progress, software teams use an Issue Tracking System (ITS), such as Bugzilla or JIRA to describe the status of issues.

Each issue in an ITS has a unique identifier, a brief description of the nature of the issue, and a variety of other metadata. Large software projects receive plenty of issue reports everyday. For example, Eclipse and Mozilla received an average of 120 and 170 new issue reports expanding from January to July 2009, respectively [8]. The number of filed issues is usually greater than the size of the development team. After an issue has been filed, project managers and team leaders *triage* them, *i.e.*, assign them to developers, denoting the urgency of the issue using priority and severity fields [3].

After being triaged, issues are then *addressed*, *i.e.*, solutions to the described issues are provided by developers. Generally speaking, an issue may be in an open or closed status. An issue is marked as open when a solution has not yet been found. We consider UNCONFIRMED, CONFIRMED, and IN_PROGRESS as open statuses. An issue is considered closed when a solution has been found. Usually, a *resolution*

Table 1: An overview of the studied systems.

System	Time frame	Releases	# of releases	# addr. issues	Median time between releases (weeks)
Eclipse (JDT)	03/11/2003 - 12/02/2007	2.1.1 - 3.2.2	11	3344	16
Firefox	05/06/2012 - 04/02/2014	13 - 27	15	3121	6
ArgoUML	18/08/2003 - 15/12/2011	0.14 - 0.34	17	14530	26

is provided with a closed issue. For instance, if a developer made code changes to address an issue, the status and resolution combination should be RESOLVED-FIXED. However, if the developer was not able to reproduce the bug, then the status and resolution may be RESOLVED-WORKSFORME.[2] The issue lifecycle is thoroughly documented on the Bugzilla website.[3]

Finally, addressed issues must be integrated into an official release in order to make them available to users. The releases that contain such addressed issues could be made available every few weeks or months, depending on the project release policy. Releasing every few weeks is typically referred to as a *rapid release* cycle, while releasing monthly or yearly is typically referred to as a *traditional release* cycle [14].

3 Study Design

In this section, we describe the studied systems, explain how the data was collected, and present an overview of the two studied dimensions of integration delay, *i.e.*, *release integration delay* and *abnormal integration delay*.

3.1 Studied Systems

In order to study integration delay, we study three subject systems: Firefox, ArgoUML and Eclipse. ArgoUML[4] is a UML modeling tool that includes support for all standard UML 1.4 diagrams. Eclipse[5] is a popular open-source IDE, of which we study the JDT core subsystem. Firefox is a popular web browser.[6]

Table 1 shows the studied: (i) timeframe, (ii) quantity and range of releases, and (iii) quantity of issue reports. We focus our study on the releases for which we could recover a list of issue IDs from the release notes. We collected a total of 20,995 issue reports from the three studied systems. Each issue report corresponds to an issue that was addressed and could be mapped directly to a release.

3.2 Database Construction

Figure 1 provides an overview of our database construction approach — how we collect and organize the data to perform our empirical study. We create a relational

[2] http://goo.gl/jdBKWD
[3] http://goo.gl/nBMhaf
[4] http://argouml.tigris.org/
[5] https://www.eclipse.org/
[6] http://goo.gl/Qyizd2

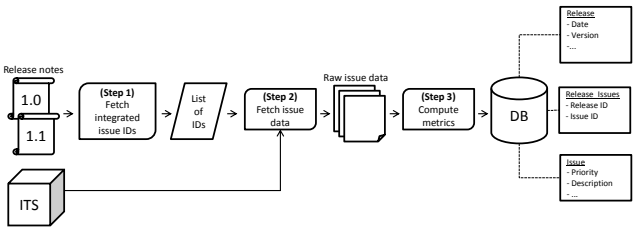


Fig. 1: An overview of our approach to construct a database for studying integration delay.

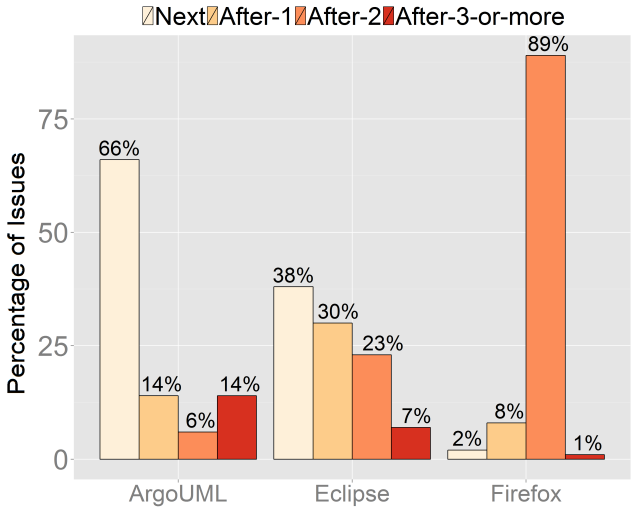


Fig. 2: Distribution of issues for each project

database describing the integration of addressed issues in the studied systems. To do so, we collect data from two sources. We briefly describe each source, and each step involved in the database construction process.

3.2.1 Step 1: Fetch integrated issue IDs

In Step 1 we consult the release notes of each studied system to identify the release that an addressed issue was integrated into. A release note is a document that describes the content of a release. For instance, a release note might provide information about the improvements included in a release (with respect to prior releases), the new features, the fixed issues, and the known problems. Eclipse, ArgoUML, and Firefox publish their release notes on their respective websites.[7]

Unfortunately, release notes may not mention all of the addressed issues that have been integrated into a release. This limitation hinders the investigation of issues that

[7] http://goo.gl/x8htzm

were addressed but have not been integrated because we cannot claim that an issue not listed in a release note was not integrated. However, the issues that are listed in a release note have certainly been integrated. Thus, we choose to use release notes despite their incompleteness to identify integrated issues in order to reduce the noise in our dataset, i.e., the release where we claim that an issue has been integrated is almost certainly correct.

The output of Step 1 is a list containing the integrated addressed issue IDs. To retrieve the list of addressed issues that have been integrated into Eclipse and Firefox, we wrote a script to extract the listed issue IDs from the release notes and insert them into our database. The retrieved issue IDs are then used to collect issue reports from the corresponding ITSs. In our database, we also store the dates and versions of the releases.

3.2.2 Step 2: Fetch issue data

Once we collect the IDs of the addressed issues that were integrated in each release note (Step 1), in Step 2 we fetch the data regarding each issue in the ITS system using the previous collected IDs. Not all release notes from ArgoUML list the issues that were addressed in that official release, and when they do, only a few issues are listed (*e.g.*, 1-4).[8] Hence, we rely on the ITS in order to map addressed issues to releases. We used the milestone field indicated in the issue reports to approximate the release that an issue was integrated into. Development milestones are counted towards the next official releases. For instance, the development milestones 0.33.7[9] is counted towards the official release 0.34. The output of Step 2 is the raw issue data collected from the ITS using the IDs fetched in Step 1.

Finally, for all of our analysis, we choose the last change to the RESOLVED-FIXED status of an issue as the moment when the issue was addressed. For instance, in case an issue changes from RESOLVED-FIXED to REOPENED, and changes to the RESOLVED-FIXED status again, we say that the issue was addressed at the last change to the RESOLVED-FIXED status. Also, we use the RESOLVED-FIXED status rather than the VERIFIED-FIXED status because we found that all of the issues that are mapped to releases went through RESOLVED-FIXED before being integrated, while only a small percentage went through VERIFIED-FIXED. For instance, only 17% of addressed issues in Firefox went through the VERIFIED-FIXED status. We focus on issues that were resolved as FIXED because they involve changes to the source and/or test code that must be integrated into a release to become visible to the public.

3.2.3 Step 3: Compute metrics

After collecting information from each addressed issue, we compute all of the information that may be related to the integration delay dimensions that we investigate

[8] http://goo.gl/nbdXp7
[9] http://goo.gl/IoK56R

in our study. We investigate two dimensions of integration delay: (i) the *release integration delay* dimension, and (ii) the *abnormal integration delay* dimension. The following subsections describe each dimension:

Release Integration Delay

We compute the release integration delay of each addressed issue, which we group into four classes: *next*, *after-1*, *after-2*, and *after-3-or-more*. The *next* class contains addressed issues that are integrated immediately. The *after-1*, *after-2*, and *after-3-or-more* classes contain addressed issues whose integration is delayed by one, two, or three or more releases, respectively. We use this data to address RQ1-RQ3.

Figure 2 shows the distributions of the addressed issues among the classes for each studied system. ArgoUML has the highest percentage of addressed issues that fall into the *next* class (71%), whereas *next* accounts for only 2% and 38% of addressed issues in Firefox and Eclipse, respectively.

We use exploratory models to study the relationship between the characteristics of addressed issues (*e.g.*, severity and priority) and the release integration delay. Our models are used to understand which characteristics are important for explaining the release integration delay of addressed issues.

Abnormal Integration Delay

In this dimension, we study issues that have been abnormally delayed by integration for a given studied system. Again, we use exploratory models to study the relationship between the characteristics of addressed issues and the abnormal integration delay. To do so, we flag each studied issue as being either *abnormally delayed* or *typically delayed*. We use the median integration delay measured in days to divide the addressed issues into these two classes. If a given addressed issue has a delay greater than the median, we classify it as *abnormally delayed*. Addressed issues with integration delay less than or equal to the median are classified as *typically delayed*. We use this data to address RQ4, RQ5, and RQ6.

4 Release Integration Delay

In this section, we present the results of our study of *release integration delay*. This study is comprised of investigations of the integration delay measured in number of releases, which address RQ1- RQ3. We present our results with respect to each research question below.

RQ1: Are issues often delayed after being addressed?

RQ1: Motivation

Users and contributors care most about when an addressed issue will be integrated into an official release rather than when it is initially addressed. In this regard, we

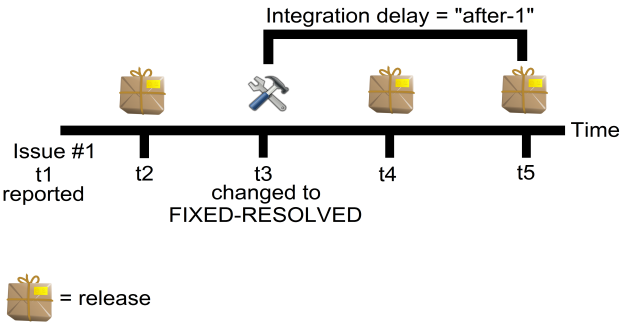


Fig. 3: Integration delay is computed by counting the releases that occur between when an issue status changes to RESOLVED-FIXED and the the date of the release note that lists that issue.

observe that some addressed issues are integrated in the next release, while others are delayed.

It is not clear why some addressed issues take more time to be integrated than others. In RQ1, we investigate the delay between when an issue is addressed and when it is integrated. In order to better understand integration delays in each studied system, we also investigate if the delays differ between rapid and traditional release cycles. The analysis of RQ1 is our first step toward understanding why integration delays differs among addressed issues.

RQ1: Approach

We compute the *integration delay* of an addressed issue as shown in Figure 3. We first collect the time when the resolution status of each issue was changed to *RESOLVED-FIXED* from the ITS. To determine the moment of integration, we analyze the release notes of each project. Finally, we count the number of releases that occurred between the time when an issue status changed to the *RESOLVED-FIXED* and the release that it was integrated into.

RQ1: Results

Addressed issues are usually delayed in the Firefox studied system. Figure 4 shows the difference between the studied systems regarding the time interval between their releases. The median time in days for Firefox (42 days) is approximately $\frac{1}{4}$ of that of ArgoUML (180 days), and $\frac{1}{3}$ of that of Eclipse (112 days). Unlike for Eclipse and Firefox, the distribution for ArgoUML is skewed. In addition, Figure 2 shows that the vast majority of addressed issues for Firefox are integrated *after-2* releases, whereas for Eclipse and ArgoUML, the majority are integrated in the *next* release.

The reason for the difference may be the release policies followed in each project. For example, Figure 4, shows that Firefox releases consistently every 42 days (six weeks), whereas the times between ArgoUML releases vary from 50 to 220 days.

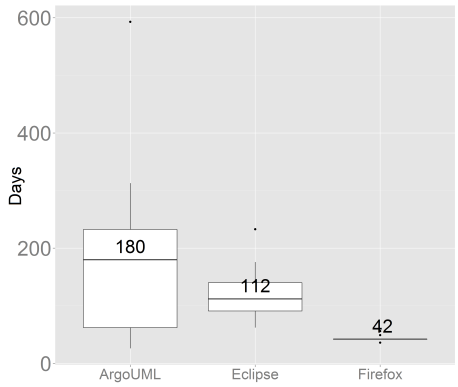


Fig. 4: Delays in days between releases of ArgoUML, Eclipse, and Firefox. The number shown over each boxplot is the median interval

Indeed, the ArgoUML releasing guidelines state that the ArgoUML team should release at least one stable release every 8 months[10] (≙ 240 days). The consistency of Firefox releases may lead to more delayed issues, since they rigidly adhere to a six-week release schedule despite accumulating issues that could not be integrated.

34% to 60% of addressed issues in the traditional release cycle systems were delayed by one or more releases. Figure 2 shows that 98% of the addressed issues in Firefox are delayed by one or more releases. We conjecture that Firefox is more likely to have delayed issues due to its rapid releasing cycle. However, 98% is still a large percentage. Furthermore, even for the systems that adopt a more traditional release cycle, 34% (ArgoUML) to 60% (Eclipse) of the addressed issues are delayed by at least one release. This result indicates that even though an issue is addressed, integration could be delayed by one or more releases.

Many delayed issues were addressed well before releases from which they were omitted. Addressed issues could be delayed from integration because they were addressed late in the release cycle, *e.g.*, one day or one week before the upcoming release date. In order to compare the rapid and traditional release cycles regarding whether delayed issues are addressed late in the release schedule, we computed the *Addressing Stage* metric (*AS*) for each issue.

The *AS* metric is calculated using the following equation: $\frac{days\ to\ next\ release}{release\ window}$, where *days to next release* is the number of days that an issue is addressed before the next release (*e.g.*, the time between *t3* to *t4* in Figure 3), and the *release window* is the time in days between the next upcoming release and the previous release (*e.g.*, *t2* to *t4*). An *AS* value close to 0 means that an issue was addressed too close to the next release, whereas a value close to 1 means that an issue was addressed at the beginning of a release cycle.

[10] http://argouml.tigris.org/wiki/Strategic_Planning

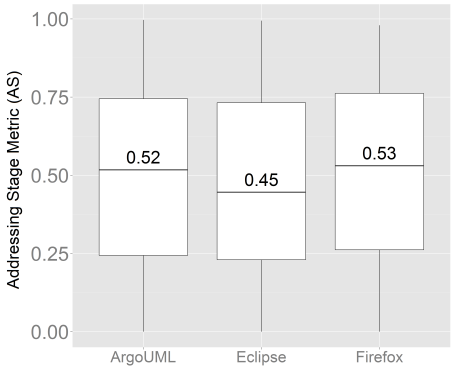


Fig. 5: Distribution of days between when an issue was addressed and the next missed release divided by the release window time.

Figure 5 shows the distribution of the *AS* metric for each project. The smallest *AS* median is observed for Eclipse, which is 0.45. For ArgoUML and Firefox, the median is 0.52 and 0.53, respectively. The *AS* medians are roughly in the middle of the release. Moreover, the boxes extend to cover between 0.25 and 0.75. The result suggests that, in the studied projects, delayed issues are usually addressed $\frac{1}{4}$ to $\frac{3}{4}$ of the way through a release. Hence, it is unlikely that most addressed issues miss the next release solely because they were addressed too close to an upcoming release date.

The integration of 34% to 60% of the addressed issues in the traditionally releasing systems and 98% in the rapidly releasing system were delayed by one or more releases. Furthermore, we find that many delayed issues were addressed well before releases from which they were omitted from.

RQ2: Can we accurately explain how many releases an addressed issue will be delayed?

RQ2: Motivation

Several studies have proposed approaches to investigate the time required to address an issue [1, 7, 13, 15, 22, 24]. These studies could help to estimate when an issue will be addressed. However, we find that an addressed issue may be delayed before being delivered to users. Even though most issues are addressed well before the next release date, many of them are not integrated until a future release. For users and contributors, however, knowing the release in which an addressed issue will be integrated is of great interest. In RQ2, we investigate if we can accurately explain how many releases an addressed issue will be delayed. Our explanatory models are important to understand which variables may impact in the integration delay of addressed issues.

Table 2: Reporter and Issue attributes used to explain the integration delay of an addressed issue

Family	Attributes	Value	Definition (d)|Rationale (r)
Reporter	Experience	Numeric	**d:** Experience in filing reports for the project. It is measured by the number of previously reported issues of a reporter.
			r: An issue reported by an experienced reporter might be integrated quickly.
	Delay of previously addressed issues	Numeric	**d:** Measured by the median of the integration delays of previous issues that were reported.
			r: If previously addressed issues were integrated quickly for a reporter, future issues reported by the same reporter may also be integrated quickly.
Issue	Component	Nominal	**d:** The component specified in the issue report.
			r: Issues related to a given component (*e.g.*, authentication) might be more important, and thus, might be integrated prior to issues in less important components.
	Platform	Nominal	**d:** The platform specified in the issue report.
			r: Issues regarding one platform (*e.g.*, MS Windows) might be integrated prior to issues in less important platforms.
	Severity	Nominal	**d:** The severity of the issue.
			r: Issues with higher severity levels (*e.g.*, blocking) might be integrated faster than other issues. Panjer observed that the severity of an issue has a large effect on its lifetime for Eclipse project [18].
	Priority	Nominal	**d:** The priority of the issue.
			r: Higher priority issues will likely be integrated before lower priority issues.
	Stack trace attached†	Boolean	**d:** We verify if the issue report has an stack trace attached in its description.
			r: A stack trace attached in the issue reported may provide useful information regard the cause of the issue, which may quicken the integration of the addressed issue [20].
	Description Size	Numeric	**d:** Description of the issue measured by the number of the words.
			r: Issues that are well-described might be more easy to integrate than issues that are difficult to understand.

† - Attributes that did not appear in our previous work [6]

Moreover, our models could estimate for users and contributors when an addressed issue will likely be integrated.

RQ2: Approach

In order to study when an addressed issue will be integrated, we collected information from both ITSs and VCSs of the studied systems. We build models using metrics from the following families: *reporter*, *issue*, *project*, and *process*.

– **Reporter** refers to the reputation of an issue reporter. Issues reported by a reporter who is known to report important issues may receive more attention from the integration team.

– **Issue** refers to reported issues. Project teams use this information to triage, address, and integrate issues. For example, integrators may not be able to properly

Table 3: Project and Process attributes used to explain the integration delay of an addressed issue

Family	Attributes	Value	Definition (d)|Rationale (r)
Project	Integration Workload	Numeric	**d:** The number of issues in the RESOLVED-FIXED state at a given time. **r:** Having a large number of addressed issues at a given time might create a high workload on integrators, and may affect the number of addressed issues that are integrated.
	Queue position†	Numeric	**d:** $\frac{\text{rank of the issue}}{\text{all addressed issues}}$, where the rank is the position in time when an issue was addressed in relation to others in the current release cycle. The rank is divided by all the issues addressed by the end of the release cyle. **r:** An issue that is near the front of the queue is more likely to be integrated quickly.
Process	Number of Impacted Files	Numeric	**d:** The number of files linked to an issue report. **r:** An integration delay might be related to a high number of impacted files because more effort would be required to properly integrate the modifications [12].
	Number of Activities	Numeric	**d:** An activity is an entry in the issue's history. **r:** A high number of activities might indicate that much work was necessary to address the issue, which can impact the integration of the issue into a release. [12].
	Number of Comments	Numeric	**d:** The number of comments of an issue report. **r:** A large number of comments might indicate the importance of an issue or the difficulty to understand it [7], which might impact the integration delay. [12].
	Number of Tosses	Numeric	**d:** The number of times the assignee has changed. **r:** The number of changes in the issue assignee might indicate a complex issue to address or a difficulty in understanding the issue, which can impact the integration delay. One of the reasons for changing the assigned developer is because additional expertise may be required to address the issue [11, 12].
	Comment Interval	Numeric	**d:** The sum of all of the time intervals between comments (measured in hours) divided by the total number of comments. **r:** A short comment time interval indicates that an active discussion took place, which suggests that the issue is important. [12].
	Churn	Numeric	**d:** The sum of the added lines and removed lines in the code repository. **r:** A higher churn suggests that a great amount of work was required to address the issue, and hence, verifying the impact of integrating the modifications may also be difficult [12, 17].
	Issue addressing time†	Numeric	**d:** Measured by number of days between when the opening date of the issue and the moment when the issue changed to RESOLVED-FIXED [7]. **r:** If issues are addressed quickly, there is also a good change that such addressed issues might be integrated quickly.

† - Attributes that did not appear in our previous work [6]

assess the importance and impact of poorly described issues, which may, in turn, lead to integration delays.

– **Project** refers to the status of the project when a specific issue is addressed. If the project team has a heavy integration workload, *i.e.*, many addressed issues waiting to be integrated, the integration of newly addressed issues may be delayed.

– **Process** refers to the process of addressing an issue. An addressed issue that involved a complex process (*e.g.*, long comment threads, large code changes) could be difficult to understand and integrate.

Tables 2 and 3 describe the information that we collect in each family. Henceforth, we refer to the collected information as attributes. For each attribute, Tables 2 and 3 presents the type and the rationale behind its use in our models.

Explanatory model. We train our models using the *random forest* technique [5], which is known to have a good overall accuracy and to be robust to outliers as well as noisy data. Model robustness is important for our study because the data in the ITSs are filed with subjective criteria and tend to be noisy [9]. In our study, we use the *random forest* implementation provided by the *bigrf* R package[11]. To build and test our explanatory models, we use a 10-fold cross-validation and 100 trees in each forest.

Evaluation metrics. We use *precision*, *recall*, *F-measure*, and *ROC area* to evaluate our models. We describe each metric below.

Precision (P) measures the correctness of our models in estimating the release delay of an addressed issue. An estimation is considered correct if the estimated integration delay is the same as the actual integration delay it had. Precision is computed as the proportion of correctly estimated integration delays for each class (*e.g.*, next, after-1).

Recall (R) measures the completeness of a model. A model is considered complete if all of the addressed issues that were integrated in a given release r are estimated to appear in r. Recall is computed as the proportion of issues that actually appear in a release r that were correctly estimated as such.

F-measure (F) is the harmonic mean of precision and recall, *i.e.*, $(\frac{2\times P\times R}{P+R})$. F-measure combines the inversely related precision and recall values into a single descriptive statistic.

ROC area is used to evaluate the degree of discrimination achieved by the model. The ROC area is the area below the curve plotting the true positive rate against false positive rate. The value of ROC area ranges between 0 (worst) and 1 (best). An area greater than 0.5 indicates that the explanatory model outperforms a random predictor. We computed the ROC area for a given class c (*e.g.*, *next*) on a binary basis. In other words, the probabilities of the instances were analyzed as pertaining to a given class c or not for each class. Therefore, each class has its own ROC area value.

[11] Bigrf package http://goo.gl/fyyd33

RQ2: Results

Our explanatory models achieve a weighted average precision between 0.25 to 0.86 and a recall between 0.72 to 0.92. Figure 6 shows the precision, recall, F-measure, and ROC area of our explanatory models. The boxplots represent the distributions of the ten folds that we performed for each class.

The best precision values that we obtain for Eclipse, Firefox, and ArgoUML are related to the *next* (median of 0.81), *after-2* (median of 0.99), and *next* (median of 0.98), respectively. However, for classes with low number of instances the precisions decrease considerably. For instance, the median precision obtained in the Firefox data for the remaining classes (*next*,*after-1*, and *after-3-or-more* are very low (median of 0.05).

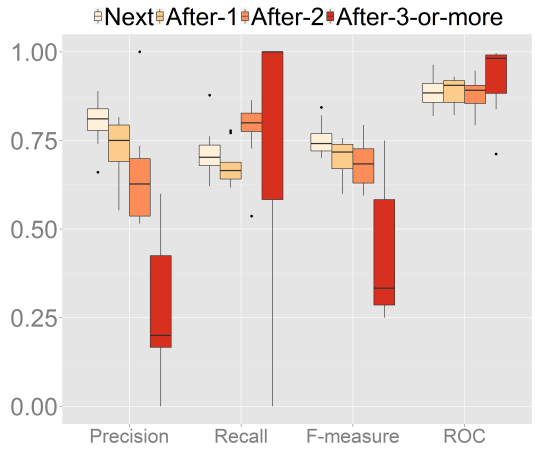
On the other hand, the obtained recall values tend to be high in classes where the number of instances are the minority. For example, the highest medians for recall in ArgoUML are for *after-2* and *after-3-or-more*, whereas in Firefox the highest medians are for *next* and *after-3-or-more*.

Moreover, we obtained ROC areas of 0.78 to 0.93 on average (median), which indicate that our model estimations are better than random guessing (ROC are of 0.5). Summarizing the results, we obtained an weighted average of the medians for precision between 0.25 to 0.86 and for recall between 0.72 to 0.92. Although there is room for improvement, our models provide a sound starting point for explaining the release that an addressed issue will be integrated into.

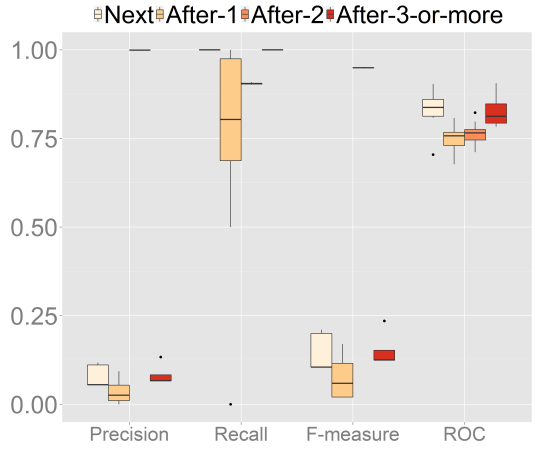
Our models achieve better F-measure values than Zero-R. We compared our models to Zero-R models as a baseline. For all test instances, Zero-R selects the class that contains the majority of the instances. Hence, the recall for the class containing the majority of instances is 1.0. We compared the F-measure of our models to the F-measure of Zero-R models. We choose to compare to the F-measure values because precision and recall are very skewed for Zero-R.

For Firefox, Zero-R has an F-measure of 0.95 for the class *after-2*, which was equal to our model. For Eclipse, Zero-R always selects *next* and achieves an F-measure of 0.58 while our model achieves 0.70. Finally, for ArgoUML, Zero-R selects always *next* with an F-measure of 0.84, whereas our model achieves 0.85. These results show that our models yield better F-measure values than naive techniques like Zero-R or random guessing (ROC = 0.5) in the majority of cases.

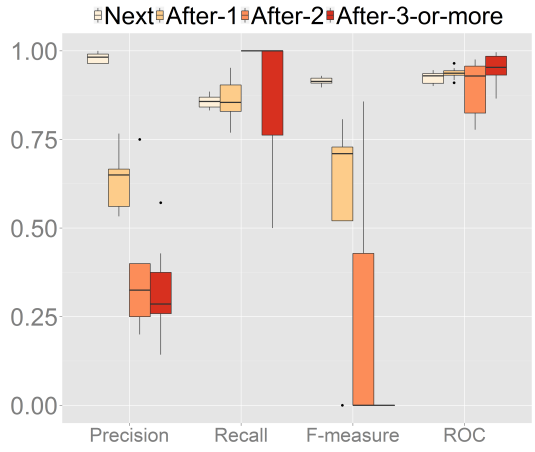
We are able to accurately explain how many releases an addressed issue is likely to be delayed. Our models outperform naive techniques such as Zero-R and random guessing, achieving an ROC area of above 0.78.



(a) Eclipse

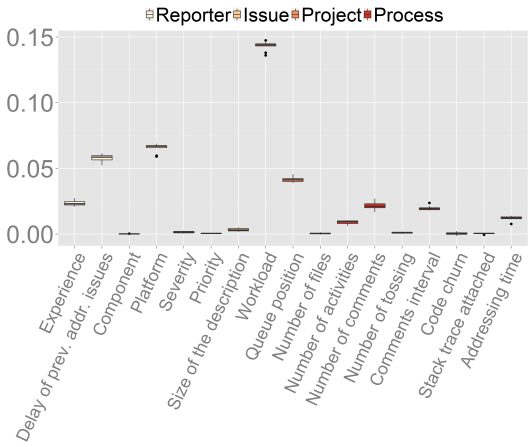


(b) Firefox

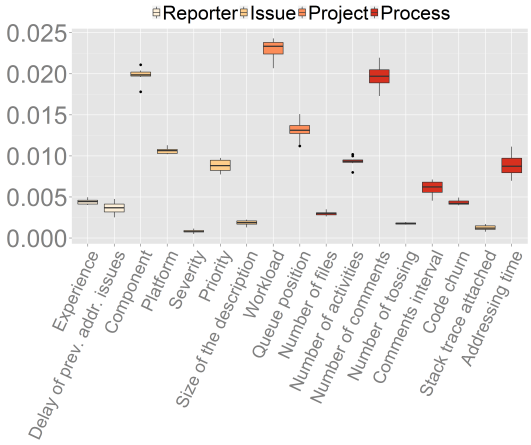


(c) ArgoUML

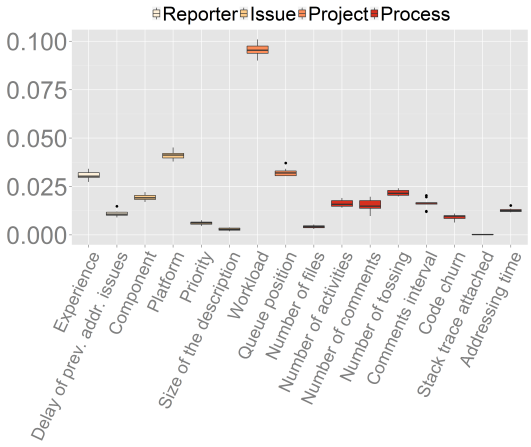
Fig. 6: The performance of our random forest models.



(a) Eclipse



(b) Firefox



(c) ArgoUML

Fig. 7: Distributions of variable importance values computed for the 10 folds performed to train the models.

RQ3: What are the most influential attributes for estimating release integration delay?

RQ3: Motivation

In RQ2, we found that our models can accurately explain the release integration delay of addressed issues. To build the models, we use attributes collected from ITSs and VCSs. As described in Tables 2 and 3, the attributes belongs to different families of relationships with the addressed issues. In RQ3, we investigate which attributes are influential in estimating the release integration delay of an addressed issue.

RQ3: Approach

To identify the most influential attributes that estimate integration delay of an addressed issue, we compute *variable importance* for each attribute in our models. The *variable importance* implementation that we use in our study is available in the *bigrf* R package. This implementation computes the importance of an attribute based on *out of the bag* (OOB) estimates. Each attribute of the dataset is randomly permuted in the OOB data. Then, the average a of the differences between the votes for the correct class in the permuted OOB and the original OOB is computed. The result of a is the importance of an attribute.

The final output of the variable importance is a rank of the attributes indicating their importance for the model. Hence, if a specific attribute has the highest rank, then it is the most influential attribute that our explanatory model is using to estimate integration delay.

Finally, to compute a statistically stable ranking of the attributes, we use the Scott-Knott technique [21], which clusters the attributes based on the importance scores obtained by our random forest models. First, two groups of statistically distinct scores are divided. Then, the Scott-Knott technique recursively divides those groups until no statistically distinct groups can be created.

RQ3: Results

The workload of integrators is the most influential attribute. By *integrators* we refer to team members that are responsible for integration tasks [12] (*e.g.*, Eclipse's release engineering team).[12] Figure 7 shows the distribution of the variable importance values computed for the ten folds of our models. The most influential attributes is the workload, which measures the amount of addressed issues still not integrated when a given issue is addressed. These results suggest that the integration backlog introduces overhead that a software team must manage. Otherwise, the integration backlog may lead to integration delays.

In addition, Table 4 shows the Scott-Knott results for the importance scores of the attributes in each of the models. Indeed, the workload of integrators is in the top group of significance in the ArgoUML and Eclipse systems. These results further

[12] http://goo.gl/q4vZzl

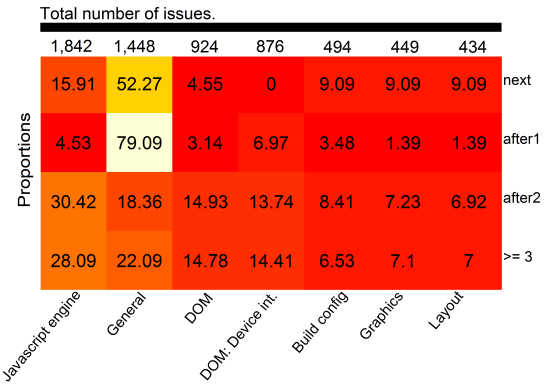
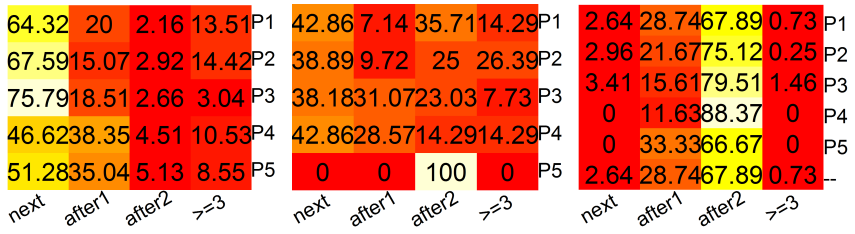
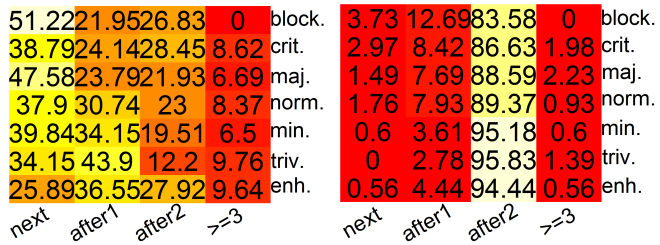


Fig. 8: The spread of issues among Firefox components. The darker the colors, the smaller the proportion of issues that impact that component.



(a) ArgoUML Priority (b) Eclipse Priority (c) Firefox Priority



(d) Eclipse Severity (e) Firefox Severity

Fig. 9: The percentage of priority and severity levels in each issue class. We expect to see light colour in the upper left corner of these graphs, indicating that high priority/severity issues are integrated rapidly. Surprinsingly, we are not seeing such a pattern in our datasets.

emphasize the importance of the integration backlog, since it may lead to delays in the integration of future issues.

Furthermore, in the Firefox system, *component* and *comments* are also influential attributes with scores close to workload. To better understand why *component* have such influence, we study the distribution of addressed issues across components. Fig-

Table 4: Scott-Knott test results for the importance of the attributes in the explanatory models. The attributes are divided into groups that have a statistically significant difference in the mean of importance scores (p-value < 0.05)

Eclipse		Firefox		ArgoUML	
Group	**Attribute**	**Group**	**Attribute**	**Group**	**Attribute**
1	Workload	1	Workload	1	Workload
2	Platform	2	Component	2	Platform
3	Delay of prev. addr. issues		Number of comments	3	Queue position
4	Queue position	3	Queue position		Experience
5	Experience	4	Platform	4	Number of tossing
6	Number of comments	5	Number of activities	5	Component
7	Comments interval		Addressing time	6	Comments interval
8	Addressing time		Priority		Number of activities
9	Number of activities	6	Comments interval		Number of comments
10	Size of the description	7	Experience	7	Addressing time
11	Severity		Code churn	8	Delay of prev. addr. issues
	Number of tossing	8	Delay of prev. addr. issues	9	Code churn
	Priority		Number of files	10	Priority
	Number of files	9	Size of the description	11	Number of files
	Stack trace attached		Number of tossing		Size of the description
	Code churn	10	Stack trace attached	12	Stack trace attached
	Component		Severity		

ure 8 shows the top seven Firefox components, each having more than 400 addressed issues. We analyze the proportion of delayed integration in the top seven components. Figure 8 shows that, for classes *next* and *after-1*, the majority of issues are related to the *General component*, whereas for *after-2* and *after-3-or-more* the majority are related to the *Javascript engine* component. Addressed issues related to the *General* component may be easy to integrate, whereas issues related to the *Javascript Engine* may require more careful analysis before integration.

Severity and priority have little influence on release integration delay. Users and contributors of software projects can denote the importance of an issue using the *priority* and *severity* fields. Previous studies have shown that priority and severity have little influence on bug fixing time [9, 16]. For example, while an issue might be severe or of high priority, it might be complex and would take a long time to fix.

However, in the integration context, we expect that priority and severity would play a bigger role, since the issue has already been addressed. One would expect that integrators would try to fast-track the integration of such high priority and severe issues. For instance, according to Eclipse guidelines for filing issue reports, priority value P1 is used for serious issues and specifies that the existence of a P1 issue should prevent a release from shipping.[13] Hence, it is surprising that priority and severity play such a small role in determining the release in which an issue will appear in. Indeed, Table 4 shows that the priority and severity metrics appear in the 5th-11th Scott-Knott importance ranks in the studied projects.

We performed an additional analysis to investigate how integration delays are related to *priority* and *severity* among the studied projects and why they had such little influence in our models. Figure 9 shows the percentage of issues with a given priority (*y-axis*) in a given delay class (*x-axis*). Note that the integration of 36% to

[13] http://goo.gl/XuR43g

97% of priority P1 addressed issues were delayed for at least one release, whereas the percentages for P2 were 32% to 96%.

In ArgoUML, while the majority of priority P1 issues (64%) were integrated in the *next* release, 36% of them were delayed by at least one release. For Firefox, 97% of the P1 issues and 96% of the *blocker* issues were delayed by at least one release. Finally, for Eclipse, 57% of P1 issues and 49% of blocker issues were delayed by at least one release. Hence, our data shows that, in the context of issue integration, *priority* and *severity* have little influence on integration delay.

The workload of the integration stage is the most influential attribute in our models. We also find that priority and severity have little influence in estimating integration delay. Indeed, 36% to 97% of top priority (P1) addressed issues were delayed by at least one release.

5 Abnormal Integration Delay

In this section, we present the *abnormal delay* dimension, which comprises our investigations regarding addressed issues that are delayed abnormally in a given studied system. Such dimension encompasses RQ4 and RQ5.

RQ4: Can we accurately explain which addressed issues will take longer to be integrated than most others?

RQ4: Motivation

In Section 4, we analyze integration delay with respect to the number of releases that an addressed issue is delayed. Additionally, we study systems with different releasing cycles. For instance, in the Firefox system, we observed that addressed issues usually get delayed — 89% of the addressed issues have an integration delay of two releases (*after-2*). These results indicate that what is an abnormal integration delay in one studied system may be normal in another. Hence, we set out to complement our previous findings by investigating if we can accurately explain if an addressed issue will be abnormally delayed.

RQ4: Approach

In order to study the average delay of the addressed issues, we count the number of days between: (1) the moment when an addressed issue is changed to RESOLVED-FIXED, and (2) the date of the release note that such addressed issue is listed. Figure 10 shows the distribution of the integration delays measured in days for each studied system. We observe that the data of Eclipse and ArgoUML systems are more skewed than the data of Firefox.

Next, we use the hexbin plots of Figure 11 to investigate the relationship between the delay measured by number of days and the delay measured by number of releases.

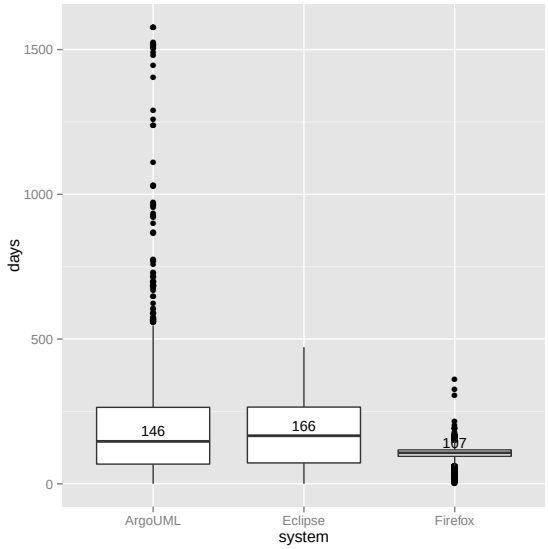


Fig. 10: Distribution of the integration delay of the addressed issues measured in days.

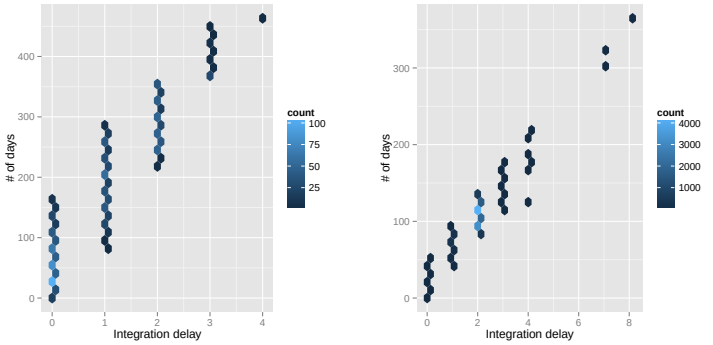
Hexbin plots are scatterplots that represent several data points with hexagon-shaped bins. The lighter the shade of the hexagon, the more data points that fall within the bin. Indeed, Figure 11 suggests that the greater the number of days, the greater is the delay in releases. This tendency is more clear in the Eclipse and Firefox systems than the ArgoUML one. However, in ArgoUML, we observe addressed issues with a longer release delay but with a smaller delay in days. For instance, we observe addressed issues with an release delay of four releases that have a shorter integration delay in days than addressed issues with an release delay of three releases. Such behaviour in the ArgoUML data may be explained by the skew in the distance between the releases of this studied system (*cf.* Figure 4).

After analyzing the integration delay measured in days, we label addressed issues as abnormally delayed by computing the median in days of the integration delay in each of the studied systems. If a given addressed issue has an integration delay greater than the median, we classify it as abnormally delayed. Thus, we produce a dichotomous response variable Y, where $Y = 1$ means the addressed issue was abnormally delayed, and $Y = 0$ means otherwise.

Finally, we build random forest models to explain if a given addressed issue will be abnormally delayed. Similar to RQ2, we evaluate our models using *precision*, *recall*, *f-measure*, and *ROC*.

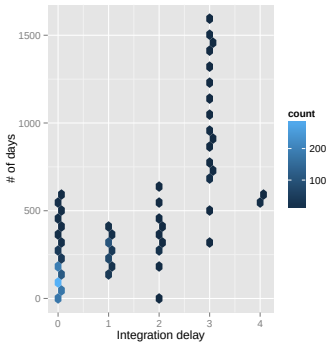
RQ4: Results

Our models obtained a median precision of 0.82 to 0.91 and a median recall of 0.78 to 0.89. Figure 12 shows the results that we obtained using our explanatory models.



(a) Eclipse

(b) Firefox

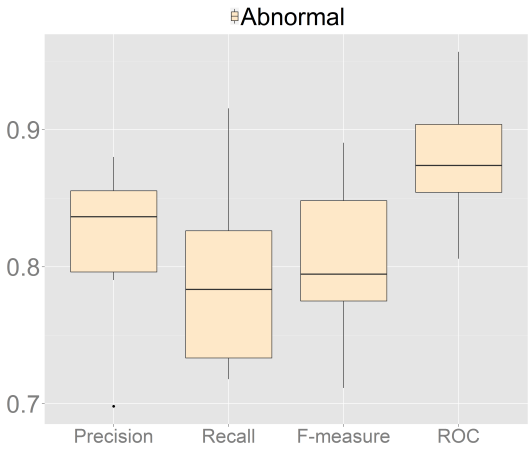


(c) ArgoUML

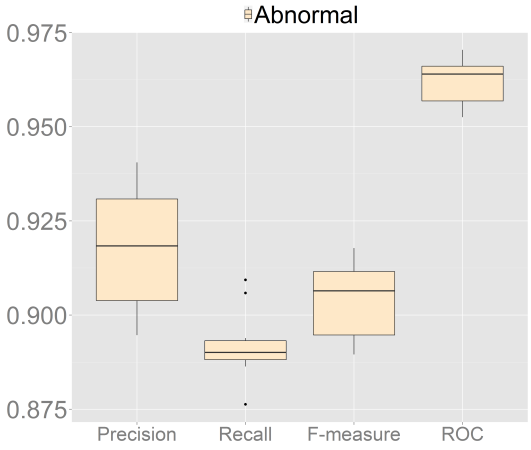
Fig. 11: Relationship between integration delay measured by number of days and number of releases.

Our Firefox models obtained the higher precision (median of 0.91), achieving a median F-measure of 0.90. On the other hand, our models built for ArgoUML obtained a median precision of 0.82, achieving a median F-measure of 0.78. Moreover, Our obtained ROC areas are of at least 0.87. These results indicate that our models outperform random guessing (ROC area of 0.50).

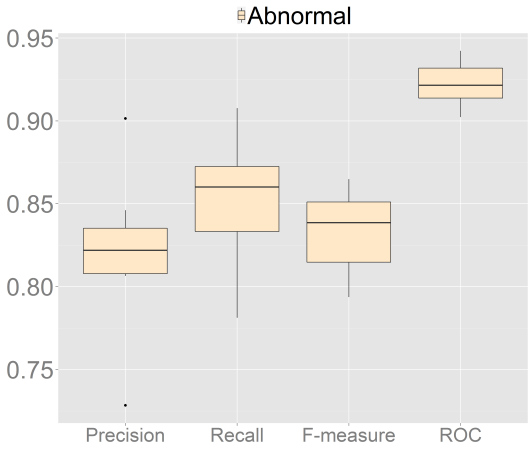
Our models achieve better F-measure values than Zero-R. We also compared the F-measures of our models to the F-measures of Zero-R. For the Eclipse, Firefox, and ArgoUML datasets, Zero-R obtained F-measures of 0.33, 0.33, and 0.45, respectively. On the other hand, our models obtained F-measures of 0.79, 0.90, and 0.78, respectively. These results indicate that our models vastly outperform naive techniques, such as Zero-R.



(a) Eclipse



(b) Firefox



(c) ArgoUML

Fig. 12: The performance of our random forest models.

Table 5: Scott-Knott test results for the importance of the attributes in the explanatory models. The attributes are divided into groups that have a statistically significant difference in the mean of importance scores (p-value < 0.05)

Eclipse		Firefox		ArgoUML	
Group	**Attribute**	**Group**	**Attribute**	**Group**	**Attribute**
1	Workload	1	Queue position	1	Queue position
2	Platform	2	Workload	2	Workload
3	Queue position	3	Number of comments	3	Platform
4	Delay of prev. addr. issues	4	Component	4	Experience
	Number of comments	4	Addressing time	5	Component
5	Experience	5	Number of activities	6	Number of activities
6	Comments interval	6	Platform	7	Number of comments
	Addressing time		Comments interval		Comments interval
	Number of activities	7	Experience	8	Number of tossing
7	Size of the description		Delay of prev. addr. issues		Priority
	Severity	8	Code churn		Addressing time
	Code churn		Priority	9	Delay of prev. addr. issues
	Number of tossing		Number of files		Code churn
	Stack trace attached		Size of the description	10	Number of files
	Priority	9	Severity		Size of the description
	Component		Number of tossing	11	Stack trace attached
	Number of files		Stack trace attached		

We are able to accurately explain when an addressed issue will be abnormally delayed. Our models outperform naive techniques, such as Zero-R and random guessing, achieving ROC areas of at least 0.87 (median).

RQ5: What are the most influential attributes for estimating abnormally delayed issues?

RQ5: Motivation

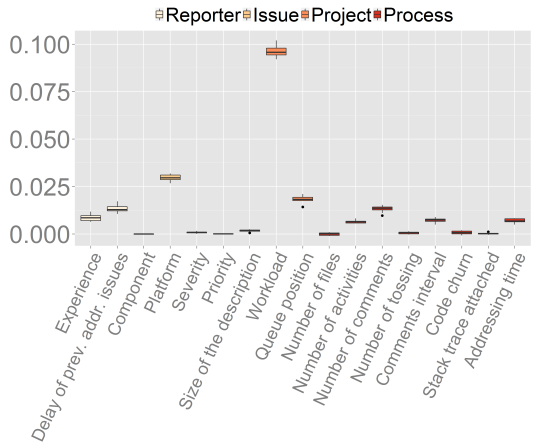
RQ4 shows that we can accurately explain if an addressed issue will be abnormally delayed or not. However, it is also important to understand what attributes are more influential to identify abnormally delayed issues — from which variables our models derive the most explanatory power.

RQ5: Approach

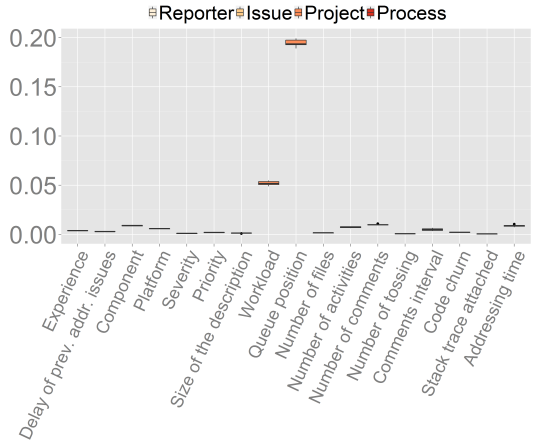
Similar to RQ3, in this research question, we analyze our explanatory models by: (1) computing the variable importance scores of our attributes, and (2) using the Scott-Knott test to identify the most influential groups of attributes.

RQ5: Results

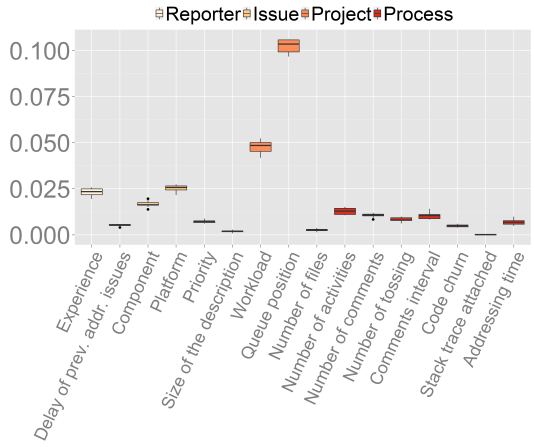
The queue position of an addressed issue and the workload of integrators are the most influential attributes. Figure 13 shows the importance scores of the 10 folds that we compute for our random forest models. We observe that the attribute "queue position" is the most influential for our Firefox and ArgoUML models. Such results



(a) Eclipse



(b) Firefox



(c) ArgoUML

Fig. 13: Distributions of variable importance values computed for the 10 folds performed to train the models.

corroborate with the intuition that, if an addressed issue is in the front of the queue of addressed issues in the release cycle, it is more likely to be integrated earlier. On the other hand, the "workload" is the most influential attribute for our Eclipse model and the second most influential in our Firefox model. These results corroborates with our previous finding that the integration backlog introduces an overhead that the software team needs to manage.

By performing Scott-Knott tests, we observe that the "queue position" and "workload" are in the top groups of significance. These results suggest that abnormal integration delay is more closely related to characteristics of the release cycle than characteristics of the reporter, the issue report, or fixing process.

The queue position of the addressed issues and the workload of the integrators are the most influential attributes in identifying issues whose integration will be abnormally delayed.

RQ6: Does the integration delay of addressed issues relates to the components that they are being modified?

RQ6: Motivation

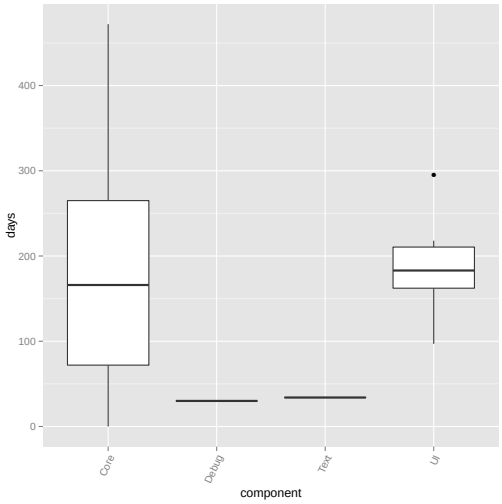
In RQ3, we find that *component* is one of the most important variables in our explanatory models of the Firefox system. We also observe that *JavaScript Engine* has the higher proportions of *after-2* and *after-3* release delays. While in the *abnormal delay* dimension, we are analysing integration delay measured in days, it is also important to investigate the relationship between this delay and the components that have been modified. Such information could be used to help users to better understand when they should expect an issue of interest to be addressed.

RQ6: Approach

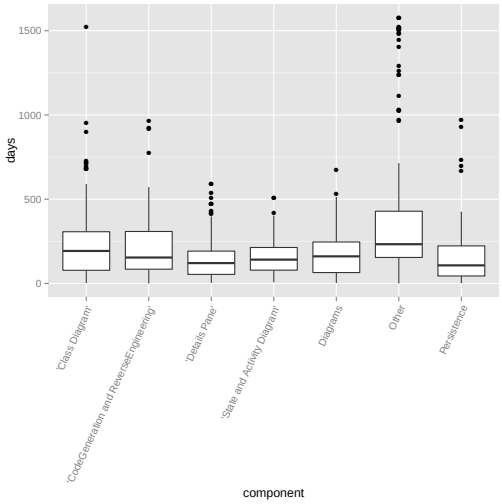
We group each addressed issue according to the components that it modifies. Since ArgoUML and Firefox have more than 50 components, we focus our analyses on a subset of components with the greatest number of addressed issues of those studied systems. We then compare the distribution of integration delay in days in these components.

RQ6: Results

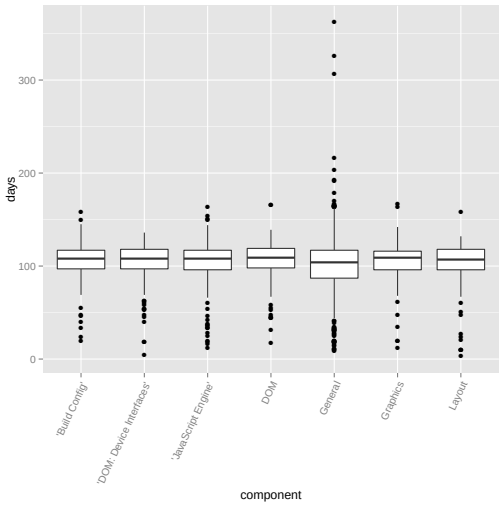
Figure 14 shows the distribution of integration delay per component for each studied system. We find that the distributions do not have a considerable difference in terms of integration delay in the ArgoUML and Firefox data. The boxplots for the components "General" and "Other" are more skewed, which is suggestive of their generic role — such components may encompass a more broad spectrum of addressed issues. On the other hand, 99% of the addressed issues in Eclipse (JDT) belongs to the "Core" component (thus its skewness). Finally, the "Debug" and "Text" Eclipse



(a) Eclipse



(b) Firefox



(c) ArgoUML

Fig. 14: Distributions of integration delay measured in days per components.

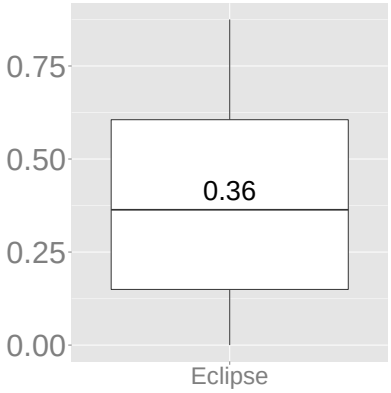


Fig. 15: The addressing stage measure for the "code freeze" period of the release cycle.

components contain only one addressed issue each.

Our component analysis reveals that there is no considerable difference between integration delays when grouped by components.

6 Exploratory Data Analysis

In this section, we discuss the "code freeze" stage of a release cycle. The "code freeze" stage is a period when the rules to make changes in the software system becomes more strict. For instance, new changes may only be integrated if it will solve special requirements such as translations or documentation fixing.[14] Such a period is important because it helps the development team to stabilize the project just before creating an official release.

In Section 5, we compute the integration delay by counting the number of days between when an issue is addressed and when its integration occurs. Furthermore, in RQ1, we find that addressed issues are unlikely to be delayed solely because they are addressed too close to an upcoming release.

To further study the integration delay, we correct our analysis for the "code freeze" period, *i.e.*, we check if the addressed issues whose integration are delayed are those issues that are addressed right before the "code freeze" stage of a release cycle.

In order to understand the "code freeze" period in our studied systems, we contacted the project leaders of each studied system. We received a reply from the project leader of Eclipse, who provided us with the code freeze window of the Eclipse Luna 4.4 release cycle. The "code freeze" occurred between June 4th, 2014 and June 25th, 2014 (21 days).

[14] http://eclipse.org/eclipse/development/plans/freeze_plan_4_4.php

Figure 15 shows the AS measure by using the "code freeze" start date rather than the official release date, *i.e.*, 21 days before the official release date. Indeed, we can observe a decrease in central tendency of the AS measure from 0.45 to 0.36 (median). However, we can observe that the majority of the data is between an AS value of 0.25 to 0.50 which is still a considerable distance in time before the "code freeze" period. In fact, the delayed addressed issues were addressed 43 days in the median before the "code freeze" stage. In addition, we find that 17% of the delayed addressed issues were addressed after the "code freeze" had started. The results indicate that even when considering the "code freeze" stage, the delayed addressed issues are unlikely to be delayed solely because they were addressed too close of a "code freeze" stage of the release cycle.

7 Threats to Validity

7.1 Construct Validity

A number of tools were developed in order to extract and analyze the integration data in the studied projects. Defects in these tools could have an influence on our results. However, we carefully tested our tools using manually-curated subsamples of the studied projects, which produced correct results.

7.2 Internal Validity

The internal threats to validity are concerned with the ability to draw conclusions from the relation between the independent and dependent variables.

The main threat in this regard is the representativeness of the data. Although Firefox and Eclipse report the list of addressed issues in their release notes, we do not know how complete this list truly is. In addition, issues may be incorrectly listed in a release note. For example, an issue that should have been listed in the release note for version 2.0 but only appears in the release note for version 3.0. Such human specification error may produce noise in our datasets.

Another threat is the method that we use to map the addressed issues to releases in ArgoUML. This mapping is based on the *target_milestone* which may be more susceptible to human error. Nonetheless, our Firefox and Eclipse results are based on addressed issues that have been denoted in the release notes — and that we are more confident about their integration time.

In addition, the way that we segment the response variable of our explanatory models is also subjected to bias. For the release delay dimension (RQ1-RQ3), we segment the response variable into *next*, *after-1*, *after-2*, and *after-3-or-more*. Although we found it to be a reasonable classification, a different classification may yield different results. Also, we use the median to split the response variable studied in the abnormal delay dimension (RQ4-RQ5) into two categories. A different threshold to split the response variable may yield different results. However, we use the median as a threshold as it is resilient to outliers.

Finally, the attributes that we considered in our explanatory models are not exhaustive. The addition of other attributes would likely improve model performance. Nonetheless, our models performed well compared to random guessing and Zero-R models with the current set of attributes and dependent variable segmentation.

7.3 External Validity

External threats are concerned with our ability to generalize our results. In our work, we investigated three open source projects. Although the projects that we considered in our study are of different sizes and domains, and prescribing to different release policies, our findings may not generalize to other systems. Replication of this work in a large set of systems is required in order to arrive at more general conclusions.

8 Related work

Estimating the effort and time required to address an issue has become an important project planning activity. To assist developers and project managers in this regard, several studies have proposed different approaches to estimate effort and time required to triage and to address issues [1, 2, 7, 10, 13, 15, 22, 24]. In each of the following subsections, we describe prior work about triaging, addressing, and integrating issues.

8.1 Triaging Issues

Triaging issues is the process of deciding which issues have to be addressed, and assigning the appropriate developer to them [3]. This decision depends of several factors, such as the impact of the issue on the software, or how much effort is required to address the issue. Projects usually receive a high number of issue reports. Issue reports come from a diverse audience that is usually larger than the developer team. Hence, effective triaging of issue reports is an important means of keeping up with user demands.

Hooimeijer and Weimer [10] built a model to classify whether or not an issue report will be "cheap" or "expensive" to triage by measuring the quality of the report. Based on their findings, the authors state that the effort required to maintain a software system could be reduced by filtering out reports that are "expensive" to triage. Saha *et al.* [19] studied long lived issues, *i.e.*, issues that were not addressed for more than one year. They found that the time to assign a developer and address such issues is approximately two years. Our work complements these prior studies by investigating the time to integrate issues once they are addressed rather than the time to assign a developer to handle the issue.

8.2 Addressing Issues

Once an issue is properly triaged, the assigned developer starts to address it. To estimate the time required to address issues, some approaches used the similarity of an issue to existing issues [22, 24], while others built prediction models using different machine learning techniques [1, 7, 15, 18]. Kim and Whitehead [13] computed the time taken to address issues in ArgoUML and PostgreSQL. They found that the median issue-fix time is about 200 days. Guo *et al.* [8] used logistic regression model to predict the probability that an new issue will be fixed. The authors trained the model on Windows Vista issues and achieved a precision of 0.68 and recall of 0.64 when predicting Windows 7 issue reports. These approaches focus on estimating the time required to address an issue. In our study, however, we investigated in which release an addressed issue will be integrated.

Recent empirical studies assess the relationship between the attributes used to build models for estimating bug fix time. Bhattacharya and Neamtiu [4] performed univariate and multivariate regression analyses to capture the significance of four features in issue reports. Their results indicate that more independent variables are required to build better prediction models. Herraiz *et al.* [9] studied the mean time to close issues reported in Eclipse, and how the severity and priority levels of the issues affect this time. In their study, the authors used one way analysis of variance to group the different priority and severity levels used in Eclipse. Based on their result, the authors suggest to reduce the severity and priority options to three levels. Zhang *et al.* [23] investigated the delays incurred by developers in the issue addressing process. To do such analyses, they extract the beginning and ending time of an issue addressing activity from interaction logs. Using the collected information they analyzed delays in the issue addressing process. In their analysis, they investigated the impact of three dimensions related to issues: issue reports, source code involved in the issue, and code changes that are required to address the bug. They found that metrics such as severity, operating system, description of the issue, and comments are likely to impact the delays in starting to address the issue and changing the status to RESOLVED. As Zhang *et al.* [23], we use attributes related to issue reports to build explanatory models. However, our aim is to understand which attributes play an important role in the delay of addressed issues. In addition, we investigate why severity and priority levels are not relevant to distinguish issue reports that are addressed and integrated in a release prior to others.

8.3 Integrating Issues

Jiang *et al.* [12] studied attributes that could determine the acceptance and integration of a patch into the Linux kernel. A patch is a record of changes that is applied to a software system to address an issue. To identify such attributes, the authors built decision tree models and conducted top node analysis. Among the attributes studied, developer experience, patch maturity, and prior subsystem are found to play a major role in patch acceptance and integration time. Similar to Jiang *et al.* [12], we also investigate the integration of addressed issues. However, we focus on the inte-

gration delay of issues that have been addressed and not if a patch is more likely to be accepted than the others.

9 Conclusion and future works

Once an issue is addressed, what users and code contributors most care about is when the software is going to reflect the addressed issue, *i.e.*, when the integration occurs. However, we observed that the integration of several addressed issues was delayed for several releases. In this context, it is not clear why certain addressed issues take longer to be integrated than others. Hence, we performed an empirical study of 20,995 issues from the ArgoUML, Eclipse and Firefox projects. In our study, we:

- found that despite being addressed well before of an upcoming release, 34% to 60% of the addressed issues were delayed by more than one release in ArgoUML and Eclipse. Furthermore, 98% of Firefox issues were delayed by at least one release.
- built random forest models to explain the integration delay of an addressed issue. Our models achieved a median ROC area of at least 0.78. Our models outperform baseline random and Zero-R models.
- computed the variable importance and the Scott-Knott tests to understand what attributes are the most important in our random forest models of integration delay. The workload of integrators is the most influential attribute in our models of release integration delay.
- found that, surprisingly, *priority* and *severity* have little impact on our models. Indeed, 36% to 97% of priority P1 addressed issues were delayed by at least one release.
- verified that our models of abnormal integration delay can outperform random guessing achieving ROC areas of at least 0.87 (median).
- found that the moment when an issue is addressed during the release cycle (queue position) and the workload of integrators are the most influential attributes for abnormal integration delay.

Our work provides some initial insights as to why some addressed issues are integrated prior to others. Our results suggest that characteristics of the release cycle are the ones that mostly impact on abnormal and release integration delay. Therefore, we believe that our findings highlight the importance of research and tools that support integrators of software projects. It is important to improve the integration stage of a release cycle, since the availability of an addressed issue in a release is what users and contributors care most about.

References

1. Anbalagan, P., Vouk, M.: On predicting the time taken to correct bug reports in open source projects. In: Proceedings of the 2009 IEEE International Conference on Software Maintenance, ICSM '09, pp. 523–526 (2009). DOI 10.1109/ICSM.2009.5306337
2. Anvik, J., Hiew, L., Murphy, G.C.: Coping with an open bug repository. In: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, eclipse '05, pp. 35–39. ACM, New York, NY, USA (2005). DOI 10.1145/1117696.1117704. URL http://doi.acm.org/10.1145/1117696.1117704
3. Anvik, J., Hiew, L., Murphy, G.C.: Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pp. 361–370. ACM, New York, NY, USA (2006). DOI 10.1145/1134285.1134336. URL http://doi.acm.org/10.1145/1134285.1134336
4. Bhattacharya, P., Neamtiu, I.: Bug-fix time prediction models: Can we do better? In: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, pp. 207–210. ACM, New York, NY, USA (2011). DOI 10.1145/1985441.1985472. URL http://doi.acm.org/10.1145/1985441.1985472
5. Breiman, L.: Random forests. In: Machine Learning, Springer Journal no. 10994, pp. 5–32 (2001)
6. Costa, D.A.d., Abebe, S.L., McIntosh, S., Kulesza, U., Hassan, A.E.: An empirical study of delays in the integration of addressed issues. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, pp. 281–290. IEEE (2014)
7. Giger, E., Pinzger, M., Gall, H.: Predicting the fix time of bugs. In: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE '10, pp. 52–56. ACM, New York, NY, USA (2010)
8. Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B.: Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pp. 495–504. ACM, New York, NY, USA (2010). DOI 10.1145/1806799.1806871. URL http://doi.acm.org/10.1145/1806799.1806871
9. Herraiz, I., German, D.M., Gonzalez-Barahona, J.M., Robles, G.: Towards a simplification of the bug report form in eclipse. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, pp. 145–148. ACM, New York, NY, USA (2008). DOI 10.1145/1370750.1370786. URL http://doi.acm.org/10.1145/1370750.1370786
10. Hooimeijer, P., Weimer, W.: Modeling bug report quality. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pp. 34–43. ACM, New York, NY, USA (2007). DOI 10.1145/1321631.1321639. URL http://doi.acm.org/10.1145/1321631.1321639
11. Jeong, G., Kim, S., Zimmermann, T.: Improving bug triage with bug tossing graphs. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations

of software engineering, pp. 111–120. ACM (2009)
12. Jiang, Y., Adams, B., German, D.M.: Will my patch make it? and how fast?: Case study on the linux kernel. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pp. 101–110. IEEE Press, Piscataway, NJ, USA (2013). URL http://dl.acm.org/citation.cfm?id=2487085.2487111
13. Kim, S., Whitehead Jr., E.J.: How long did it take to fix bugs? In: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06, pp. 173–174. ACM, New York, NY, USA (2006). DOI 10.1145/1137983.1138027. URL http://doi.acm.org/10.1145/1137983.1138027
14. Mantyla, M.V., Khomh, F., Adams, B., Engstrom, E., Petersen, K.: On rapid releases and software testing. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on, pp. 20–29. IEEE (2013)
15. Marks, L., Zou, Y., Hassan, A.E.: Studying the fix-time for bugs in large open source projects. In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Promise '11, pp. 11:1–11:8. ACM, New York, NY, USA (2011)
16. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: Apache and mozilla. ACM Trans. Softw. Eng. Methodol. **11**(3), 309–346 (2002). DOI 10.1145/567793.567795. URL http://doi.acm.org/10.1145/567793.567795
17. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, pp. 284–292. IEEE (2005)
18. Panjer, L.D.: Predicting eclipse bug lifetimes. In: Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07, pp. 29–. IEEE Computer Society, Washington, DC, USA (2007). DOI 10.1109/MSR.2007.25. URL http://dx.doi.org/10.1109/MSR.2007.25
19. Saha, R., Khurshid, S., Perry, D.: An empirical study of long lived bugs. In: 2014 Software Evolution Week - IEEE Conference onSoftware Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), pp. 144–153 (2014). DOI 10.1109/CSMR-WCRE.2014.6747164
20. Schroter, A., Bettenburg, N., Premraj, R.: Do stack traces help developers fix bugs? In: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, pp. 118–121. IEEE (2010)
21. Scott, A., Knott, M.: A cluster analysis method for grouping means in the analysis of variance. Biometrics pp. 507–512 (1974)
22. Weib, C., Premraj, R., Zimmermann, T., Zeller, A.: How long will it take to fix this bug? In: Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07, pp. 1–. IEEE Computer Society, Washington, DC, USA (2007)
23. Zhang, F., Khomh, F., Zou, Y., Hassan, A.: An empirical study on factors impacting bug fixing time. In: Reverse Engineering (WCRE), 2012 19th Working Conference on, pp. 225–234 (2012). DOI 10.1109/WCRE.2012.32
24. Zhang, H., Gong, L., Versteeg, S.: Predicting bug-fixing time: An empirical study of commercial software projects. In: Proceedings of the 2013 International Con-

ference on Software Engineering, ICSE '13, pp. 1042–1051. IEEE Press, Piscataway, NJ, USA (2013)