2012 19th Working Conference on Reverse Engineering

An Empirical Study on Factors Impacting Bug Fixing Time

Feng Zhang[1], Foutse Khomh[2], Ying Zou[2], and Ahmed E. Hassan[1]
[1]*School of Computing, Queen's University, Canada*
{*feng, ahmed*}*@cs.queensu.ca*
[2]*Department of Electrical and Computer Engineering, Queen's University, Canada*
{*foutse.khomh, ying.zou*}*@queensu.ca*

***Abstract*—Fixing bugs is an important activity of the software development process. A typical process of bug fixing consists of the following steps: 1) a user files a bug report; 2) the bug is assigned to a developer; 3) the developer fixes the bug; 4) changed code is reviewed and verified; and 5) the bug is resolved. Many studies have investigated the process of bug fixing. However, to the best of our knowledge, none has explicitly analyzed the interval between bug assignment and the time when bug fixing starts. After a bug assignment, some developers will immediately start fixing the bug while others will start bug fixing after a long period. We are blind on developer's delays when fixing bugs. This paper explores such delays of developers through an empirical study on three open source software systems. We examine factors affecting bug fixing time along three dimensions: bug reports, source code involved in the fix, and code changes that are required to fix the bug. We further compare different factors by descriptive logistic regression models. Our results can help development teams better understand factors behind delays, and then improve bug fixing process.**

***Keywords*-bug fixing process; change request; fixing time; empirical software engineering; bug report; mylyn.**

I. Introduction

Fixing bugs[1] is a major activity in the software development process. It is estimated that 80% of the total cost of a software system is spent on fixing bugs [1]. Prior studies have proposed to predict the time needed to fix bugs [2], [3], [4], [5], [6], [7], [8], [9]. Bug fixing time is however always studied as a whole, *i.e.*, from the time a bug was introduced or reported to the time it was resolved. An ignored phenomenon is that after a bug assignment, some developers immediately start bug fixing, while other developers start fixing the bug after a long period. In addition, after a bug fixing is finished, some developers immediately update the bug status (*e.g.*, RESOLVED), while other developers update the bug status after a long period. It is still unclear whether the delays commonly exist in the process of bug fixing and what factors impact the delays.

To our best knowledge, there has been no studies explicitly investigating the delays. This is because of the difficulty in getting exact time when developers start and finish code changes for bug fixing. Thanks to Mylyn, which records developers' operations with timestamp during code edits, we can find the exact time when developers start to edit code and when developers finishes code edits. Exploring delays by developers during bug fixing process is of significant interest to help development team to

- Locate time-consuming steps
- Understand factors affecting delays by developers
- Accelerate bug fixing
- Improve the process of bug fixing

The delays incurred by developers (*i.e.*, before and after code change) can be affected by a number of factors. We categorize factors to three dimensions[2]: bug reports, source code involved in the fix, and code changes that are required to fix the bug. Developers rely on the initial bug reports to understand the bug, such as what the problem is, and how severe the bug is. Further discussions among a bug reporter, developers, and testers are often recorded as comments of bug reports. Panjer [5] finds that attributes (*e.g.*, severity, product, component, and version) of an initial bug report are the most influential factors of the bug fixing time, as well as comments. Similar findings are reported in [8], [9], [10]. Characteristics of source code to be changed may affect the bug fixing time as well. Developers may spend more time to work out a solution, before starting to fix bugs, in the files having larger size or being more complex. Lines of Code (LOC) and Function Points (FP) are mostly used code metrics [3], [11]. Code changes that are required to fix the bug are also likely to affect bug fixing time. Canfora *et al.* [2] inspect the relation of bug survival time (since a bug was introduced) with fine grained change types during bug fixing.

Our study complements prior work by inspecting factors along three dimensions. The differences to prior works are:

1) We do not study the total bug fixing time, but focus on the delays incurred by developers during bug fixing;
2) We compare the importance of factors along three dimensions (*i.e.*, bug reports, source code, and code changes) all together using descriptive regression models.

[1]We use the same term 'bug' as Eclipse Bugzilla to describe an issue, which can be an enhancement or a defect.

[2]Note that our study focuses on the comparison of different factors impacting bug fixing time other than the prediction of bug fixing time, therefore we use post-fixing data (*e.g.*, comments, and code changes) as well which are only available after code change.

1095-1350/91 $25.00 © 4891 IEEE
DOI 10.1109/WCRE.2012.32

In this paper, we explore delays by developers through an empirical study on three open source Eclipse projects: Mylyn[3], Eclipse Platform[4] and Eclipse Plug-in Development Environment (PDE)[5]. We propose three research questions and briefly summarize our findings:

RQ1: *Do delays by developers exist during bug fixing process?*

This research question help to understand the process of bug fixing. Our results show that delays before and after change are the top two intervals in a process of bug fixing. The median of delays before and after changes are 210.79 and 8.55 hours, respectively.

RQ2: *Can we characterize delays incurred by developers before and after fixing bugs?*

Understanding factors behind these delays can help improve the process of bug fixing. We observe that delays are impacted by three dimensions: *bug reports*, *source code*, and *code changes*. For example, the median of delay before fixing an enhancement is 1.96 times greater than the median of delay before fixing a defect. Also, the median of delay after fixing an enhancement is 13.8 times greater than the median of delay after fixing a defect.

RQ3: *What factors contribute to the delays most?*

We identify the most influential factors and compare their importance. We observe that the level of severity and the sum of code churns on changed files have the highest impact on delays before changes. The most influential factors of delays after changes are: the maximum length of comments and the maximum weighted methods per class (WMC).

In the reminder of this paper, we first summarize related work in Section II. We describe the experimental setup of our study in Section III and reports our results on investigating three open source projects in Section IV. We then present threats to validity of our work in Section V, and conclude in Section VI.

II. Related Work

Bug fixing is one of the core activities of software maintenance. Understanding factors affecting bug fixing time can help improve the process of bug fixing. Prior studies aim at predicting time needed for bug fixing (*i.e.*, from the time it was reported to the time it was fixed).

Bug reports are heavily used in prediction. Weiss *et al.* [4] mine textual information from existing bug reports. They propose to predict the fixing time of a bug by finding its most close bug reports using K-Nearest Neighbour (kNN) clustering algorithm. Panjer [5] performs a case study on Eclipse projects, and reports that most influential factors affecting bug fixing time are mined from initial bug properties (*e.g.*, severity, product, component, and version), and post-submission data (*e.g.*, comments). Besides comments, Giger *et al.* [9] find the number of involved people is also a good predictor, and similar findings are reported by Anbalagan and Vouk [8]. Hooimeijer and Weimer [10] demonstrate the correlation between reputation of a bug reporter and bug triaging time. However, Bhattacharya and Neamtiu [7] report that reputation of a bug reporter doesn't correlate with bug fixing time. Guo *et al.* [12] finds that reassignments of bugs increase the bug fixing time, indicating that the process can affect bug fixing time as well. Herbsleb and Mockus [13] find that the geographical distance (*e.g.*, longer than 30 meters) increases bug fixing time in distributed development teams. Zend and Rine [3] propose to investigate both bug properties (*e.g.*, severity) and change properties (*e.g.*, code churn, and time spent on code editing) together.

Canfora *et al.* [2] analyze the relationship between fine grained change types and bug survival time. Bug survival time is the interval between the time a bug was introduced and the time the bug was fixed. To identify bug introducing changes, Kim and Whitehead [6] propose an approach.

Source code metrics, such as Lines of Code (LOC) and Function Points (FP) are commonly used in most techniques [3], [11]. Mauczka *et al.* [14] report that Weighted Methods per Class (WMC) and Numbers Of Methods per class (NOM) are correlated to code churn.

In contrast to prior work, we analyze two specific intervals rather than the total bug fixing time. The first interval is the delay before change (DBC), which starts from bug assignment and stops when a developer starts to edit code. The second interval is the delay after change (DAC), which starts from the time a developer finishes code editing and stops when the status of a bug is marked as RESOLVED. To the best of our knowledge, no prior studies have explicitly examined the delays before and after change. Another significant difference from prior studies is that we focus on revealing factors behind the delays rather than predicting bug fixing time. Furthermore, we compare the importance of each factor affecting delays by adopting logistic regression models used by Zimmermann *et al.* [15].

III. Case Study Setup

This section presents the design of our case study, which aims to address the following three research questions:

1) *Do delays by developers exist during bug fixing process?*
2) *Can we characterize delays incurred by developers before and after fixing bugs?*
3) *What factors contribute to the delays most?*

A. Subject Projects

In this study, we use Mylyn interaction logs to extract the beginning and ending time of bug fixing activity (*i.e.*, editing code to fix a bug). As an Eclipse project, Mylyn is more frequently used in other Eclipse projects. Therefore we choose three Eclipse projects with the highest number

[3]http://www.eclipse.org/mylyn/
[4]http://www.eclipse.org/platform/
[5]http://www.eclipse.org/pde/

Table I: Top three projects with the highest number of bug reports containing Mylyn logs.

Projects	Description	# of Bugs	# of logs
Mylyn	Task and application lifecycle management framework.	2722	3883
Platform	Core frameworks, services and runtime provider of Eclipse.	606	793
PDE	Eclipse plug-in development environment.	524	638

of bug reports containing Mylyn logs. Table I shows the descriptive statistics of the subject projects. In this study, we merge data from three Eclipse projects because the number of bug reports containing Mylyn logs is small. The three projects have been used in prior studies by Lee *et al.*[16] and Ying *et al.*[17], and our early work [18]. In three papers, the data was merged.

B. Data Source

1) Bug Reports: From bug reports, we mine timestamps when a bug was reported, assigned, and resolved, and when the development team made the first response. A bug report contains severity, priority, description, and other attributes. Discussions among the reporter, developers, and testers are recorded as comments of bug reports. The attributes used in this study are shown in Table II. In addition, Eclipse Bugzilla maintains the history of modifications of bug reports as well.

The status of a bug report is typically changed in the following order: NEW, ASSIGNED, RESOLVED, VERIFIED, and CLOSED. However, the initial status can be UNCONFIRMED if a bug is filed by a user.If a bug fixing is unsatisfactory, the status will be set to REOPEN and a reassignment is needed. A detailed life cycle of a bug report is described in the work by Weiss *et al.* [4]. If a bug fixing is marked as RESOLVED, all possible resolutions used in Eclipse projects are: *Fixed*, *Worksforme*, *Invalid*, *Wontfix*, and *Duplicate*. We filter out all bugs whose final resolution is neither *Fixed* nor *Worksforme* to ensure that only fixed bugs are investigated.

2) Mylyn Logs: From Mylyn logs, we mine timestamps when developers exactly start and finish bug fixing. Mylyn logs are generated by Mylyn, an Eclipse plugin that monitors developer's programming activities, such as selection and editing of files. Each activity is recorded as an *event*, which is called *InteractionEvent* in Mylyn. An *InteractionEvent* contains the following attributes: *StartDate* (*i.e.*, timestamp when the event starts), *EndDate* (*i.e.*, timestamp when the event ends), *OriginId* (*i.e.*, identifier of the UI affordance that tracks the event), *StructureHandle* (*i.e.*, names of the files involved in the event), and *Kind* (*i.e.*, type of the event). There are eight types[6] of events. Only three of them are triggered by a developer, which are *Command*, *Edit* and *Selection*. Mylyn logs are stored in XML format, then compressed, encoded (to *Base64* format), and attached to bug reports with the name *"mylyn/context/zip"*. Thanks to this naming rule, we can easily find all bug reports that contain Mylyn logs through Eclipse Bugzilla's search engine. We use bug ids to locate corresponding code changes from CVS commit log.

[6]http://wiki.eclipse.org/Mylyn_Integrator_Reference#Event_Kinds

Table II: Selected attributes of a bug report used in Eclipse projects.

Field	Description
Bug Id	Unique identification number of the problem.
OS	Operating System where the problem was found.
Severity	How severe the problem is and whether it is an enhancement request.
Reporter	Person who filed the bug.
Assigned To	Person who is responsible to fix the bug.
CC	People who will be notified on any change.
Summary	Short summary of the problem.
Description	Description of the problem with timestamp.
Comment	Discussion on the problem with timestamp.

3) CVS Repository: CVS is a widely used version control system, and used in our three subject projects. From CVS repository, we mine code commits for bug fixings and then compute static and historical metrics. Each commit has a message which generally describes the purpose of the commit and often contains a bug id.

C. Data Collection and Preprocessing

1) Collection: From Eclipse Bugzilla, we downloaded all bug reports that belong to our three subject projects and contain Mylyn logs. We extracted Mylyn logs from the bug reports. We collected snapshot of CVS repositories of the three subject projects on Oct. 20, 2011.

2) Preprocessing: We generate a list of bug ids of all downloaded bug reports. A bug id is often included in the commit log. If a bug id is present, we can further link the commit to its corresponding bug report. To identify a bug id, we develop a tool to extract all digit numbers from commit logs first, and then compare these digit numbers with the list of bug ids that we obtain from Eclipse Bugzilla's search engine. If matched, we consider that the commit is dedicated to fix the corresponding bug, and record the names and revisions of all files changed in this commit. In our data, only two bugs (*i.e.*, #250640 and #269959) are not found after examining all commit logs. Based on bug ids, we link together a bug report, Mylyn logs, and a group of involved files and their corresponding revisions.

D. Intervals Extraction

A typical process of bug fixing is: 1) a user files a bug report; 2) the bug is assigned to a developer; 3) the developer fixes the bug; 4) changed code is reviewed and verified; and 5) the bug is resolved. Correspondingly, the bug fixing time can be split into five consecutive intervals:

- **Delay Before Response (DBR):**
  Interval between a bug is reported and it gets the first response from development teams.
- **Delay Before Assigned (DBA):**
  Interval between a bug gets the first response and it is assigned.
- **Delay Before Change (DBC):**
  Interval between a bug is assigned and the developer starts to fix the bug.
- **Duration of Bug Fixing (DBF):**
  Interval between the developer starts and finishes the bug fixing.
- **Delay After Change (DAC):**
  Interval between the developer finishes the bug fixing and status of the bug is changed to RESOLVED.

We mine from the history of a bug report to extract the timestamps when it was reported, when it got first response (*i.e.*, the first modification on a bug report's attributes or the first post of a comment), when it was assigned, and when its status was changed to RESOLVED with resolution as either *Fixed* or *Worksforme*. We denote above four timestamps as $T_{reported}$, $T_{responded}$, $T_{assigned}$, and $T_{resolved}$, respectively. We mine from Mylyn logs to extract the timestamps of the first and last *Edit* event (*Edit* event is triggered when a developer selects any text in opened source file). We denote above two timestamps as $T_{fixBegin}$ and T_{fixEnd}. We compute the intervals using following equations:

$$DBR = T_{responded} - T_{reported} \quad (1)$$

$$DBA = T_{assigned} - T_{reported} \quad (2)$$

$$DBC = T_{fixBegin} - T_{assigned} \quad (3)$$

$$DBF = T_{fixEnd} - T_{fixBegin} \quad (4)$$

$$DAC = T_{resolved} - T_{fixEnd} \quad (5)$$

E. Metrics Computation

1) From Bug Reports: We compute metrics based on severity, operating system, title, description and comments of bug reports.

Severity of a bug: We recover the initial state of a bug report based on its history. Based on the initial severity, we mark each bug report as either an enhancement or a defect. With the level of initial severity, we further divide all defects into two groups: defects with *Low* severity (*i.e.*, trivial, minor, or normal), and defects with *High* severity (*i.e.*, major, critical, or blocker).

Operating system where a bug was found: The operating systems in our data are: Windows, Linux, Mac OS, Solaris and All. We exclude "All" since where the bug was found is vague (In our data, 524 bugs are filtered out.). And we also exclude "Solaris" since there is only 1 bug that was reported on Solaris. Based on the operating system, we mark all remaining bugs whether it was found on Windows, Linux or Mac OS, respectively.

Description of a bug report: We compute the literal length (*i.e.*, number of characters) of the title and description of a bug report, and denote them as $lenTitle$ and $lenDesc$. In addition, we compute the average and the maximum literal length of all comments. Such post submissions can reflect the complexity to understand bug reports.

2) From Source Code: We compute metrics from the code snapshots at the revision just before the first change of each bug fixing. We made a tool to compute static metrics based on ANTLR[7].

ANTLR: ANTLR is a compiler's compiler. It supports to traverse abstract syntax tree (AST) by embedding code snippets into the description file of the grammar. We chose ANTLR based on its extensibility to process other programming languages (*e.g.*, C, C++). In this study, we embed our code snippets to a publicly available Java 1.5 grammar file[8].

Computation: For each file, we count its lines of code (LOC). For each class, we compute its weighted method per class (WMC) and count its number of methods (NOM). For each method, we compute its McCabe's complexity (VG). Considering that there may be more than one file involved in the fixing of one bug, we further compute the average, sum, and maximum value of these metrics.

3) From Source Code Changes: For each bug, we compute its corresponding change metrics from all commits that are linked to it. We compute code churn and number of changed files to measure the size of the change. If there are more than one revision in a same file for the same bug, we accumulate code churn of all these revisions. We further compute metrics based on fine grained change types, which are extracted by *ChangeDistiller*.

ChangeDistiller: This tool was created by Fluri *et al.* [19] to extract fine grained change types. It accepts two files as input and then extracts changes using tree differencing algorithms.In our study, *ChangeDistiller* produces more than 160 change types. We further categorize these fined grained change types into 27 groups.

F. Analysis Method

1) Analyzing the relationship between delays and factors: We divide all bugs into two groups by each individual factor, respectively. We use the Wilcoxon rank sum test [20] to compare the delays (*i.e.*, DBC and DAC) of two groups. The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions have equally large values. Non-parametric statistical methods make no assumptions about the distributions of assessed variables.

2) Analyzing the importance of different factors on affecting delays: We build a logistic regression model [15] from

[7]http://www.antlr.org

[8]http://www.antlr.org/grammar/1152141644268/Java.g

Table III: Quantiles of bug fixing time independently mined from bug reports and Mylyn logs (unit: hour).

Source	0%	25%	50%	75%	100%
Bug Reports	0.03	111	586	2746	46748
Mylyn Logs	0.02	0.17	0.68	4.16	14790

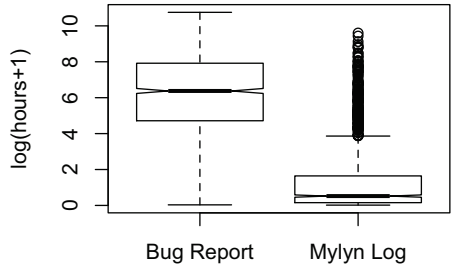

Figure 1: Boxplot of log(bugFixingTimeFromBugReport+1) and log(bugFixingTimeFromMylynLog+1).

all factors that are investigated in this study. We use AIC to select independent factors. A logistic regression model combines different factors to predict the probability of the occurrence of an event. We use the magnitude of coefficient of each factor to compare their importance to the delays. In addition, positive sign of coefficient indicates a positive correlation, and vice versa.

IV. Case Study Results

In this section, we report and discuss results of our empirical study performed on three Eclipse projects.

RQ1: Do delays by developers exist during bug fixing process?

Motivation. This question is preliminary to the other questions. We observed that bug#162007 was reported on Oct. 13rd, 2006, and assigned on Oct. 23rd, 2006, but the bug fixing was started on Mar. 25th, 2011. Another bug #246547 was reported and assigned on Sep., 8th, 2008, and bug fixing was performed on the same day and lasted for 8 minutes, but the status of this bug was marked to RESOLVED on Feb., 24th, 2011. In these two cases, there exist delays incurred by developers before and after change. In this research question, we determine if delays incurred by developers commonly exist in the process of bug fixing.

Approach. To answer this research question, we independently compute bug fixing time from bug reports and Mylyn logs and obtain two groups of bug fixing time. From bug reports, bug fixing time is the interval between bug assignment and bug resolution (status of bug is changed to RESOLVED). From Mylyn logs, bug fixing time is the interval between first and last *Edit* event.

Table IV: Quantiles of intervals in our subject systems (unit: hour).

Interval	0%	25%	50%	75%	100%	Std
DBR	0.00	0.06	1.72	27.12	23226	1423
DBA	0.00	0.00	0.11	20.38	24116	1968
DBC	0.00	22.86	210.79	1506.97	38730	4716
DBF	0.02	0.17	0.68	4.16	14790	800
DAC	0.00	3.01	8.55	274.21	29883	2269

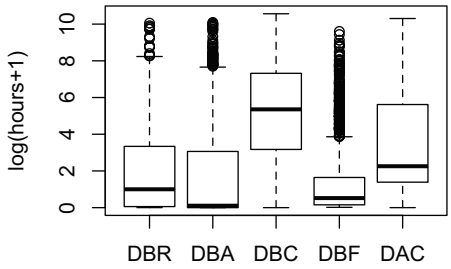

Figure 2: Boxplot of logarithm of all intervals: log(DBR+1), log(DBA+1), log(DBC+1), log(DBF+1), and log(DAC+1).

We test the following null hypothesis.

H_{01}: *there is no difference between the bug fixing time that are independently extracted from bug reports and Mylyn logs.*

We perform one sided Wilcoxon rank sum test for H_{01}, using the 5% level (*i.e.*, p-value < 0.05).

Findings. The $p\text{-}value$ of one sided Wilcoxon rank sum test is less than $2.2e$-16, we then reject H_{01}. The bug fixing time independently computed from bug reports and Mylyn logs are significantly different. The delays incurred by developers before and after bug fixing activity do exist. We show boxplot of bug fixing time in the two groups in Figure 1 and report their quantiles in Table III.

To understand the process of bug fixing, we further study the intervals defined in Section III-D. We show the boxplot of all intervals in Figure 2 and report their quantiles in Table IV. Development teams often respond to newly reported bugs in two hours (*e.g.*, the median of DBR is 1.72 hours), and finishes bug assignment quickly (*e.g.*, the median of DBA is 0.11 hours). To our surprise, the time spent on performing code change is pretty short (*e.g.*, the median of DBF is 0.68 hours). Delays before and after change are much longer than all other steps during the process of bug fixing (*e.g.*, the medians of DBC and DAC are 210.79 and 8.55 hours, respectively). It is therefore of significant interest to study factors impacting the delays before and after change.

Negative Delays. To our surprise, we find that nearly 10% bugs have a negative interval DBC, which means that a developer has already started to fix a bug even before the bug is reported. This may be caused by several reasons: *e.g.*, a developer reports and fixes the bug by himself but

Table V: Results of Wilcoxon test for groups split by severity. (n.s stands for not statistically significant, the same below.)

Delay	Pair of Groups Compared	Test	p-value
DBC	*Enhancement* (19%) v.s. *Defect* (81%)	greater	< 0.01
	LowSeverity (91%) v.s. *HighSeverity* (9%)	greater	< 0.05
DAC	*Enhancement* (19%) v.s. *Defect* (81%)	greater	< 0.001
	LowSeverity (91%) v.s. *HighSeverity* (9%)	less	n.s

Table VI: Quantiles of delays in groups split by severity (unit: hour)

Delay	Severity	0%	25%	50%	75%	100%
DBC	*Enhancement*	0.03	27.99	364	3564	33737
	Defect	0.00	21.61	186	1224	38730
	LowSeverity	0.00	22.85	198	1292	38730
	HighSeverity	0.02	15.54	110	701	18159
DAC	*Enhancement*	0.00	3.02	56.40	613	20284
	Defect	0.00	3.00	4.08	181	29883

forgets to fill the field of assignee[9], or a developer just copies Mylyn contexts from previous task which contains change information of old bugs. Considering these cases are abnormal, we filter out all bugs that have negative interval DBC or DAC from our dataset.

***RQ2:* Can we characterize delays incurred by developers before and after fixing bugs?**

Motivation. This research question aims to characterize the delays DBC and DAC by factors from three dimensions: *bug reports*, *source code involved in the fix*, and *code changes that are required to fix the bug*.

Approach. To address this question, we use each individual factor f to divide all bugs into two groups, respectively. For each factor f, we test the following null hypothesis.

H_{02}^1: *there is no difference between the delay* DBC *of bugs in the two groups divided by the factor* f.

H_{02}^2: *there is no difference between the delay* DAC *of bugs in the two groups divided by the factor* f.

We perform one sided Wilcoxon rank sum test for H_{02}^1 and H_{02}^2, using the 5% level (*i.e.*, p-value < 0.05).

Findings.

1) Property of Bug Reports: We discuss the impact of initial severity, operating system where a bug was found, and description of a bug report.

Severity of a bug: We present the results of the one sided Wilcoxon rank sum test in Table V. In Table VI, we further report quantiles of delays of groups that passed the Wilcoxon test.

For the delay DBC, the $p\text{-}value$s are less than 0.05, we then reject H_{02}^1 for both factors: type of bug and severity level of defect. The median of DBC on fixing enhancements is 1.96 times that of fixing defects. The median of DBC on fixing defects with high severity is 0.56 times that of fixing defects with low severity. To optimize delays before change, development teams must be careful on deciding severity of bugs.

For the delay DAC, the $p\text{-}value$ is less than 0.05 on groups split by type of bug, but greater than 0.05 on groups split by severity level of defect. We then reject H_{02}^2 for type of bug and accept H_{02}^2 for severity level of defect. The median of DAC on fixing enhancements is 13.8 times that of fixing defects. A possible explanation is that it takes more time to verify enhancements than defects whatever the defect's severity is.

[9]We use the history of modifications on assignee field to detect bug assignment. As assignee is set to *xx-inbox* by default, changing assignee back to *xx-inbox* is not considered as bug assignment.

Table VII: Results of Wilcoxon test for groups split by operating system

Delay	Pair of Groups Compared	Test	p-value
DBC	*Windows* (67%) v.s. *Not Windows* (33%)	greater	< 0.01
	Linux (26%) v.s. *Not Linux* (74%)	less	< 0.05
	Mac OS (7%) v.s. *Not Mac OS* (93%)	less	< 0.05
DAC	*Windows* (67%) v.s. *Not Windows* (33%)	greater	n.s
	Linux (26%) v.s. *Not Linux* (74%)	less	< 0.05
	Mac OS (7%) v.s. *Not Mac OS* (93%)	greater	n.s

Table VIII: Quantiles of delays in groups split by operating system (unit: hour)

Delay	Severity	0%	25%	50%	75%	100%
DBC	*Windows*	0.00	28.61	252	1824	38730
	Not Windows	0.00	17.98	159	1080	21840
	Linux	0.00	20.24	167	1099	21840
	Not Linux	0.00	26.12	216	1699	38730
	Mac OS	0.87	8.91	117	888	13046
	Not Mac OS	0.00	26.42	212	1506	38730
DAC	*Linux*	0.00	3.01	3.10	72.14	18018
	Not Linux	0.00	2.97	16.73	327	29883

Operating system where a bug was found: We present the results of the one sided Wilcoxon rank sum test in Table VII. In Table VIII, we further report quantiles of delays of groups that pass Wilcoxon test.

For the delay DBC, the $p\text{-}value$s are less than 0.05, we then reject H_{02}^1 for three factors: whether a bug was found on Windows, Linux, or Mac OS, respectively. The median of DBC on fixing bugs for Windows is 1.58 times that of fixing bugs for other operating systems. The median of DBC on fixing bugs for Linux is 0.77 times that of fixing bugs for other operating systems. The median of DBC on fixing bugs for Mac OS is 0.55 times that of fixing bugs for other operating systems.

For the delay DAC, the $p\text{-}value$ is less than 0.05 only on groups split by the factor that whether a bug was found on Linux. We then reject H_{02}^2 for Linux and accept H_{02}^2 for the other two. The median of DAC on fixing bugs for Linux is 0.19 times that of fixing bugs for other operating

Table IX: Wilcoxon test results for groups split by description of a bug report.

Delay	Factor	Median	Test	p-value
DBC	lenTitle	54	less	n.s
	lenDescription	318	less	n.s
	avg_lenComment	198	less	< 0.01
	max_lenComment	635	less	< 0.001
DAC	lenTitle	54	less	n.s
	lenDescription	318	less	n.s
	avg_lenComment	198	less	< 0.001
	max_lenComment	635	less	< 0.001

systems. A possible explanation is that Linux is widely used by development teams, and it is easier to automate testing on Linux. Further studies are needed to identify the root cause of this shorter delay.

Description of a bug report: For each metric shown in Table IX, we obtain two groups using the median of metric: bugs with $metric < median(metric)$ and bugs with $metric \geq median(metric)$. We report results of the one sided Wilcoxon rank sum test in Table IX.

From Table IX, the length of title and description of bug report do not affect the delays DBC and DAC. Increasing length of the text of comments also increases the delays DBC and DAC. A possible cause of long comments is that the bug is not clearly described or the bug is too complicated.Further studies are needed to identify the reason why developers write long comments.

2) Property of Source Code: We discuss the effect of size and complexity of the source code to be changed. To measure the size, we use lines of code (LOC) and number of methods (NOM). To measure the complexity, we use weighted method per class (WMC). For each metric, we use its median to separate bugs into two groups. For every two groups, we perform a one sided Wilcoxon rank sum test to examine whether delays of the two groups are statistically different. We report our results in Table X.

Lines Of Code (LOC): We observe that only total lines of code affects the delay DBC, while the average and maximum lines of code don't affect DBC. The delay DAC is affected by all three statistic values of lines of code. The more lines of code the files (*i.e.*, files to be changed for bug fixing) have, the longer delay DBC and DAC will happen.

Number Of Methods (NOM): We observe that only the total number of methods affects the delay DBC, while the average and maximum number of methods don't affect DBC. The delay DAC is affected by total and maximum number of methods, but not average number of methods. The more number of methods the files (*i.e.*, files to be changed for bug fixing) have, the longer delay DBC and DAC will happen.

Weighted Methods per Class (WMC): We observe that the sum of WMC affects the delay DBC, while the average and maximum of WMC don't.The delay DAC is affected by total and maximum of WMC, but not by the average of WMC. The more complex the files (*i.e.*, files to be changed for bug fixing) are, the longer delay DBC and DAC will happen.

Table X: Wilcoxon test results for groups split by property of source code.

Delay	Factor	Median	Test	p-value
DBC	avg_LOC	377	greater	n.s
	sum_LOC	1106	less	< 0.001
	max_LOC	667	less	n.s
	avg_NOM	16	greater	n.s
	sum_NOM	48	less	< 0.001
	max_NOM	29	less	n.s
	avg_WMC	43	greater	n.s
	sum_WMC	128	less	< 0.01
	max_WMC	84	less	n.s
DAC	**avg_LOC**	377	less	< 0.01
	sum_LOC	1106	less	< 0.01
	max_LOC	667	less	< 0.01
	avg_NOM	16	less	n.s
	sum_NOM	48	less	< 0.01
	max_NOM	29	less	< 0.01
	avg_WMC	43	less	n.s
	sum_WMC	128	less	< 0.01
	max_WMC	84	less	< 0.05

Table XI: Wilcoxon test results for groups split by property of code change.

Delay	Factor	Median	Test	p-value
DBC	numChangedFiles	2	less	< 0.001
	avg_codeChurn	19	less	< 0.001
	sum_codeChurn	50	less	< 0.001
	max_codeChurn	35	less	< 0.001
	numMacroChangeTypes	6	less	< 0.001
	sumMacroChangeTypes	16	less	< 0.001
	numMicroChangeTypes	8	less	< 0.001
	sumMicroChangeTypes	14.5	less	< 0.001
DAC	numChangedFiles	2	less	< 0.001
	avg_codeChurn	19	less	< 0.001
	sum_codeChurn	50	less	< 0.001
	max_codeChurn	35	less	< 0.001
	numMacroChangeTypes	6	less	< 0.001
	sumMacroChangeTypes	16	less	< 0.001
	numMicroChangeTypes	8	less	< 0.001
	sumMicroChangeTypes	14.5	less	< 0.001

3) Property of Code Change: We discuss the effect of code changes, which are characterized by code churn, number of changed files, and fine grained change types. For each metric, we use its median to separate bugs into two groups. For every two groups, we perform a one sided Wilcoxon rank sum test to examine whether delays of the two groups are statistically different. We report our results in Table XI, Table XII, and Table XIII, respectively. The impact on the delays by all these metrics are statistically significant.

Code Churn: Code churn measures the size of change in terms of lines of code changed. From Table XI, the larger the code churn is, the longer delay DBC and DAC will happen.

Number of Changed Files: Number of changed files measures the propagation of change. From Table XI, we observe that the more files are changed when fixing a bug,

Table XII: Wilcoxon test results for groups split by fine grained change types.

Delay	Change Type	Freq.	Test	p-value
DBC	method invocation: ins	64%	less	< 0.001
	if: ins	59%	less	< 0.001
	add functionality: method	54%	less	< 0.001
	method invocation: update	49%	less	< 0.01
	local variable: ins	46%	less	< 0.001
	add functionality: attribute	43%	less	< 0.001
	method invocation: del	40%	less	< 0.001
	if: delete	40%	less	< 0.001
	assignment: ins	39%	less	< 0.001
	local variable: update	34%	less	< 0.001
DAC	method invocation: ins	64%	less	< 0.001
	if: ins	59%	less	< 0.001
	add functionality: method	54%	less	< 0.001
	method invocation: update	49%	less	< 0.001
	local variable: ins	46%	less	< 0.001
	add functionality: attribute	43%	less	< 0.001
	method invocation: del	40%	less	< 0.001
	if: delete	40%	less	< 0.001
	assignment: ins	39%	less	< 0.001
	local variable: update	34%	less	< 0.001

Table XIII: Results of the Wilcoxon test for groups split by categories of fine grained change types.

Delay	Change Type	Freq.	Test	p-value
DBC	method invocation	81%	less	< 0.05
	if / while	66%	less	< 0.001
	add functionality	64%	less	< 0.001
	local variable	63%	less	< 0.01
	assignment	53%	less	< 0.001
	conditional expression	48%	less	< 0.05
	comment	43%	less	< 0.01
	return statement	37%	less	< 0.001
	delete functionality	28%	less	< 0.01
	java doc	26%	less	< 0.001
DAC	method invocation	81%	less	< 0.01
	if / while	66%	less	< 0.01
	add functionality	64%	less	< 0.001
	local variable	63%	less	< 0.001
	assignment	53%	less	< 0.001
	conditional expression	48%	less	< 0.01
	comment	43%	less	< 0.001
	return statement	37%	less	< 0.001
	delete functionality	28%	less	< 0.001
	java doc	26%	less	< 0.001

the longer delay DBC and DAC will happen.

Summary of Change Types: We also present statistic values computed from fine grained change types in Table XI. The more types of changes will be performed, the longer delay DBC and DAC will happen.

To investigate the effect of different change types, we further examine fine grained change types.

Fine Grained Change Types: In our data, ChangeDistiller extracted over 160 fine grained change types. To show the effect of fine grained change types, we choose the first 10 change types that occur most frequently. Based on each change type, we split all bugs into two groups: the change type doesn't occur or occurs. From Table XII, the effects of all change types are statistically significant.

We further categorize fine grained change types and examine the top 10 categories. We present our results in Table XIII. The categories still have statistically significant effect on delays.

RQ3: What factors contribute to the delays most?

Motivation. We have discussed individual factors separately, but these factors may have cross-correlations. This research question aims to find independent factors and compare their importance on affecting delays.

Approach. A logistic regression model combines different factors together to predict the probability of the occurrence of an event. To address this question, we build a logistic regression model and define two events: long DBC and long DAC. Long DBC happens if DBC is greater than the median of all DBCs. We define long DAC similarly.

In order to compare all factors together, we transform all factors to boolean or categorical values. For example, we convert severity to two boolean values: $isBug$ (1 for "defect", and 0 for "enhancement") and $isBugOfHighSeverity$ (1 for "major", "critical", and "blocker", and 0 for "trivial", "minor", and "normal"). We convert operating system to three boolean values: $isOSWindows$, $isOSLinux$, and $isOSMac$. For all other factors, we use their median to divide them into two groups (*i.e.*, 1 if equal or greater than median, and 0 otherwise).

We use our entire dataset to build two models to describe the importance of different factors on delays before and after change, respectively:

- Probability that a developer starts to fix a bug after a period longer than the median of DBC.
- Probability that the status of a bug is set to RESOLVED after a period longer than the median of DAC.

We build initial logistic regression model based on all factors, and then perform AIC to select independent factors.

Findings. Table XIV presents finally selected factors in our two models.We interpret the models to show the importance of factors on delay DBC and DAC, respectively.

1) Interpreting the Descriptive Model "Long DBC"*:* The most influential factor on the delay before change is the level of severity.Its negative sign indicates that high severity (*i.e.*, "major", "critical", or "blocker") reduces DBC. The second important factor is the sum of code churn during bug fixing. Its positive sign shows that an increase of code churn will increase DBC. A possible explanation is that a developer will spend more time preparing to fix a bug if more lines of code are to be affected.

Sum of number of methods in changed files and the maximum length of all comments in the bug reports are other two factors impacting the delay DBC. We don't build logistic regression model based on fine grained change types, since the metrics computed from fine grained change types are not selected in this model.

Table XIV: Descriptive logistic regression models for (1) Long DBC, and (2) Long DAC; n.s stands for not statistically significant and dash "-" means that the metric is not selected in the final model.

FACTOR	Long DBC		Long DAC	
	COEF.	*p-value*	COEF.	*p-value*
isBug	-	-	-0.466	< 0.01
isBugOfHighSeverity	-0.481	< 0.05	-	-
isOSLinux	n.s	n.s	-0.466	< 0.01
isOSMac	n.s	n.s	n.s	n.s
lenDescription	-	-	-0.328	< 0.05
avg_lenComment	-	-	n.s	n.s
max_lenComment	0.246	< 0.05	0.728	< 0.001
max_LOC	-	-	n.s	n.s
sum_NOM	0.307	< 0.05	-	-
avg_WMC	n.s	n.s	-	-
sum_WMC	-	-	0.521	< 0.05
max_WMC	-	-	-0.721	< 0.01
nbFiles	-	-	n.s	n.s
avg_codeChurn	-	-	0.321	< 0.05
sum_codeChurn	0.373	< 0.01	-	-
sumMicroChangeTypes	-	-	n.s	n.s

2) Interpreting the Descriptive Model "Long DAC"*:* The most influential factors on the delay after change are the maximum length of all comments in the bug report and the WMC of changed files. The sign of coefficients indicate that the maximum length of all comments and sum of WMC have a positive impact, while the maximum WMC has a negative impact on the delay DAC. The type of bug, the operating system where a bug was found, the length of bug description, and average code churn also impact the delay DAC.

V. THREATS TO VALIDITY

We now discuss the threats to validity of our study following common guidelines provided in [21].

Construct validity threats concern the relation between theory and observation. Our construct validity threats are mainly due to measurement errors. We rely on Mylyn logs to identify exact starting and ending time of a bug fixing task. Because some files may be edited without using Mylyn, our file editing information might be biased.

Threats to internal validity concern our selection of subject systems and analysis methods. Although we study three software systems, some of the findings might still be specific to the bug fixing process of the three software systems which are all Eclipse projects. Future studies should consider using a different tool to collect file editing data.

Threats to external validity concern the possibility to generalize our results. We only analyzed three Eclipse projects, because of the limited adoption of Mylyn in open source projects. Further studies on different open and closed source systems are desirable to verify our findings.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. Eclipse CVS and Bugzilla are publicly available to obtain the same data. We also make our data available[10].

VI. CONCLUSION

In this paper, we analyzed delays incurred by developers in three open source software systems, Mylyn, Eclipse Platform, and Eclipse PDE. Understanding factors causing the delays can help development teams with prioritization during bug triaging. We explored factors impacting the delays from three dimensions: *bug reports*, *source code involved in the fix*, and *code changes that are required to fix the bug*. We further compared all factors and identified the most influential factors affecting the delays before and after change.

Our results show that the delays do exist in the process of bug fixing. Moreover, the delay before and after change are two major intervals of the process of bug fixing. Our findings provide development teams insights in prioritizing bugs and optimizing bug assignment, so that they can reduce delays to improve their process of bug fixing. We summarize our findings of most influential factors:

- Type of a Bug: The median of DBC (respectively DAC) on fixing enhancements is 1.96 (respectively 13.8) times that of fixing defects. Development teams should investigate in details their process after a developer finishes to implement an enhancement. If the longer DAC is caused by verification, then more resources should be allocated on testing.
- Severity of a Bug: The median of DBC on fixing defects with high severity is 0.56 times that of fixing defects with low severity. This indicates that the level of severity is an important reference for developers when deciding which bug to fix first. Development teams should make proper level of severity as early as possible.
- Operating System: The median of DBC (respectively DAC) when fixing bugs that are found on Linux is 0.77 (respectively 0.19) times that of other operating systems. Development teams should further investigate why both delays are shorter for bugs found on Linux, so that they can improve their bug fixing process for bugs found on other operating systems.
- Description of a Bug: Increasing the literal length of description can increase DAC. If a bug report has a longer description, development teams should examine whether it is necessary to split into several simpler bugs.
- Comments of a Bug: The maximum length of all comments impacts both DBC and DAC. Development teams should examine bug reports with long comments in order to understand the reasons behind long comments size, and prevent long comments in the future.

[10]http://tinyurl.com/delaystudy-zip

- Property of Source Code: Sum of NOM is more important than LOC and WMC when determining DBC. WMC is more important than LOC and NOM when determining DAC. These metrics can help development teams to prioritize bugs during bug triaging.
- Property of Code Change: An increase of the sum of code churns increases DBC and an increase of average code churns increases DAC.

In the future, we plan to study more data sources from more projects in order to make our findings more generic. We also plan to perform a survey with developers of these projects to verify the practicalness of our results.

ACKNOWLEDGMENT

The authors would like to thank Daniele Romano at Delft University of Technology and Michael Wrsch at University of Zurich for their kind help on solving ChangeDistiller problems.

REFERENCES

[1] N. I. of Standards & Technology, "The economic impacts of inadequate infrastructure for software testing," May 2002, uS Dept of Commerce.

[2] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "How long does a bug survive? an empirical study," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, oct. 2011, pp. 191 –200.

[3] H. Zeng and D. Rine, "Estimation of software defects fix effort using neural networks," in *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*, ser. COMPSAC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 20–21.

[4] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–.

[5] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 29–.

[6] S. Kim and E. J. Whitehead, Jr., "How long did it take to fix bugs?" in *Proceedings of the 2006 international workshop on Mining software repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 173–174.

[7] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 207–210.

[8] P. Anbalagan and M. Vouk, "On predicting the time taken to correct bug reports in open source projects," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, sept. 2009, pp. 523 –526.

[9] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 52–56.

[10] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 34–43.

[11] Y. Ahn, J. Suh, S. Kim, and H. Kim, "The software maintenance project effort estimation model based on function points," *Journal of Software Maintenance*, vol. 15, no. 2, pp. 71–85, Mar. 2003.

[12] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, ""not my bug!" and other reasons for software bug report reassignments," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, ser. CSCW '11. New York, NY, USA: ACM, 2011, pp. 395–404.

[13] J. Herbsleb and A. Mockus, "An empirical study of speed and communication in globally distributed software development," *Software Engineering, IEEE Transactions on*, vol. 29, no. 6, pp. 481 – 494, june 2003.

[14] A. Mauczka, T. Grechenig, and M. Bernhart, "Predicting code change by using static metrics," in *Proceedings of the 2009 Seventh ACIS International Conference on Software Engineering Research, Management and Applications*, ser. SERA '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 64–71.

[15] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Software Engineering (ICSE), 2012 34th International Conference on*, june 2012, pp. 1074 –1083.

[16] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 311–321.

[17] A. Ying and M. Robillard, "The influence of the task on programmer behaviour," in *IEEE 19th International Conference on Program Comprehension*, ser. ICPC'11, june 2011, pp. 31 –40.

[18] F. Zhang, F. Khomh, Y. Zou, and A. Hassan, "An empirical study of the effect of file editing patterns on software quality," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, oct. 2012.

[19] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, NOV 2007.

[20] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007.

[21] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.